PARTE PRIMA: GENERALITÀ

1. INTRODUZIONE

Un SO è un programma che agisce come intermediario tra l'utente e l'hardware di un computer. Lo scopo di un SO è di fornire un ambiente nel quale un utente sia un grado di eseguire dei programmi. Il suo obiettivo e quindi di rendere conveniente l'utilizzo di un computer. Uno scopo secondario è quello di utilizzare l'hardware del computer in maniera efficiente

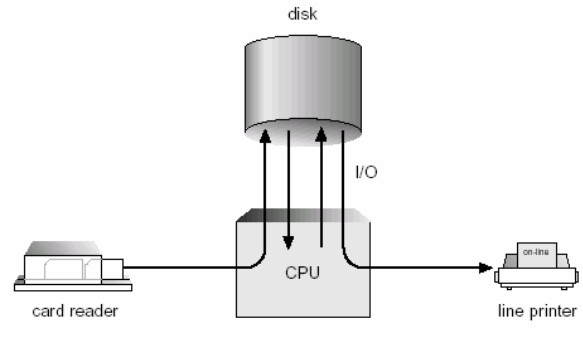
1.1 COS'È UN SO?

Un sistema di calcolo può venir suddiviso in quattro componenti:
- l'hardware (CPU, memorie e dispositivi di I/O)
- il SO (controlla e coordina l'utilizzo dell'hardware da parte dei programmi d'applicazione e degli utenti)
- i programmi d'applicazione
- gli utenti (persone, macchine, altri computer)

È possibile considerare un SO come un allocatore di risorse: il SO gestisce le risorse del sistema di calcolo allocandole in base alla necessità.
Un SO è anche un programma di controllo che controlla l'esecuzione dei programmi utenti in modo da impedire essenzialmente che vengano commessi errori o che il computer venga utilizzato in modo non corretta.
Lo scopo primario di un SO è di essere conveniente per l'utente, uno scopo secondario è rendere efficiente l'uso di un sistema di calcolo.

1.2 SISTEMI BATCH SEMPLICI

I primi computer erano delle enormi macchine il cui funzionamento era controllato da una console. Gli usuali dispositivi di input erano i lettori di schede e le unità nastro; gli usuali dispositivi di output erano le stampanti, le unità nastro e i perforatori di schede. L'utente non interagiva direttamente con il sistema di calcolo ma preparava un Job — composto dal programma, i dati e le informazioni sul tipo di job (schede di controllo) — e lo affidava, di solito sotto forma di schede perforate, all'operatore del computer. Dopo un certo tempo si otteneva l'output che consisteva dei risultati del programma e, in presenza di errori del programma, dell'immagine del contenuto della memoria e dei registri. Per accelerare l'elaborazione, i job con requisiti simili erano raggruppati insieme e fatti eseguire dal computer in un unico blocco. Così i programmatori lasciavano le schede all'operatore, il quale ordinava i programmi in lotti (batch) con requisiti analoghi e, non appena il computer era disponibile, li mandava in esecuzione. La caratteristica che contraddistingue un sistema batch è l'assenza di interazione tra l'utente e il job durante l'esecuzione di questo. In tale ambiente esecutivo la cpu restava spesso inattiva; ciò si verificava perché i dispositivi meccanici di I/O sono intrinsecamente più lenti dei dispositivi elettronici. L'introduzione dei dischi è stata d'aiuto: anziché trasferire direttamente dal lettore di schede alla memoria il contenuto delle schede, quindi elaborare il job, le schede sono lette direttamente dal lettore su disco. Analogamente, quando il job richiede alla stampante di stampare una riga, quella riga è copiata in un buffer del sistema e scritta sul disco. L'output viene effettivamente stampato quando il lavoro è completato.



Questo tipo di gestione è chiamato spooling. Il nome è acronimo di simultaneous peripheral operation on-line. Lo spooling usa fondamentalmente il disco come un buffer molto ampio, per leggere in anticipo il maggior numero possibile di dati dai dispositivi di input e per memorizzare i dati di output fino a che i dispositivi di output siano in grado di accettarli. Lo spooling sovrappone l'I/O di un job con la computazione di altri job; anche in un sistema semplice è infatti possibile che lo spooler legga l'input di un job mentre viene stampato l'output di un altro job, contemporaneamente un terzo job viene eseguito. Lo spooling ha un effetto positivo sulle prestazione del sistema: impiegando poco spazio sul disco e poche tabelle, la computazione di un job può essere sovrapposta all'I/O di altri job; quindi lo spooling permette alla CPU e ai dispositivi di I/O di lavorare a velocità maggiori.

1.3 SISTEMI BATCH MULTIPROGRAMMATI

Lo spooling presenta una struttura di dati molto importante: il pool di job. Con lo spooling in genere si hanno più job già letti in attesa sul disco, pronti ad essere eseguiti. Un pool di job permette al SO di selezionare il job successivo da eseguire, in modo da aumentare l'utilizzo della CPU. Se i job arrivano direttamente su schede o persino su nastro magnetico, non è possibile eseguire i job in ordine diverso da quello di arrivo. I job devono essere eseguiti in sequenza: il primo ad arrivare è il primo ad essere eseguito. Invece, se su un dispositivo ad accesso diretto, come un disco, si trovano più job, diventa possibile effettuare uno scheduling dei job.
L'aspetto più importante dello scheduling dei job è la possibilità di effettuare la multiprogrammazione. L'idea alla base è la seguente: il SO tiene contemporaneamente in memoria centrale diversi job; tale insieme di job è generalmente un sottoinsieme dei job presenti nel pool di job, a causa delle limitate dimensioni della memoria centrale che di solito consentono di contenere meno job di quanti ne siano presenti nel pool. Prima o poi tale job si trova in attesa di qualche evento, come il montaggio di un nastro o il completamento di una operazione do I/O. In questi casi, in un sistema non multiprogrammato, la CPU rimane inattiva. In un sistema multiprogrammato, invece, il SO passa semplicemente ad un altro job e lo esegue, e così di seguito. Quando il job ha terminato l'attesa, ritorna alla cpu. Finché c'è un job da eseguire, la CPU non è mai inattiva.
Se più job sono pronti per essere caricati ma non c'è abbastanza spazio per tutti, il sistema deve scegliere tra essi: in tale decisione consiste lo scheduling dei job. Quando il SO seleziona un job dal pool di job, lo carica in memoria per l'esecuzione. La presenza contemporanea di più programmi in memoria implica la gestione della memoria stessa. Inoltre, se in un dato momento più job sono pronti per essere eseguiti, il sistema deve scegliere il job da eseguire. Questa decisione è lo scheduling dei job.

1.4 SISTEMI TIME–SHARING

Il time–sharing (o multitasking) è un'estensione logica della multiprogrammazione; più job sono eseguiti dalla CPU che commuta le loro esecuzioni con una frequenza tale da permettere agli utenti di interagire con ciascun programma durante la sua esecuzione. Un sistema di calcolo interattivo, invece, permette la comunicazione on–line tra utente e sistema. L'utente impartisce le istruzioni direttamente al sistema operativo oppure a un programma e riceve risposta immediata. Per consentire agli utenti di accedere comodamente a codice e dati è necessario disporre di un file system on–line.
I sistemi batch sono idonei all'esecuzione di job di grandi dimensioni che necessitano di scarsa interazione. L'utente può sottoporre i job da esaminare i risultati in un secondo tempo; non è infatti necessario attendere l'esecuzione del job. I job interattivi tendono ad essere composti da più azioni brevi, dove i risultati del comando successivo possono non essere previsti.
I primi computer erano sistemi interattivi, vale a dire che tutto il sistema era a disposizione immediata del programmatore/operatore. Questa condizione offriva al programmatore una grande flessibilità e libertà nella prova e nello sviluppo dei programmi. Come si è visto, però, il funzionamento risentiva della presenza di tempi morti elevati. I sistemi batch furono sviluppati proprio per risolvere questo problema; migliorarono infatti l'utilizzo del sistema.
I sistemi con time–sharing furono sviluppati per garantire l'uso interattivo di un sistema di calcolo a un costo ragionevole. Un SO con time–sharing utilizza lo scheduling della cpu e la multiprogrammazione per assicurare a ciascun utente una piccola parte di computer. Ciascun utente dispone almeno di un proprio programma in memoria. Un programma caricato in memoria e in esecuzione è noto come processo. Normalmente un processo, durante la propria esecuzione, impegna la CPU per un breve periodo di tempo prima di richiedere delle operazioni di I/O o di terminare; tali operazioni possono essere interattive. Piuttosto che lasciare la cpu inattiva, durante l'input interattivo il SO commuta velocemente la cpu al programma di un altro utente.
I SO con time–sharing sono più complessi dei sistemi multiprogrammati. Come accadeva con la multiprogrammazione, il time–sharing prevede la coesistenza di più job in memoria; tale condizione richiede una forma di gestione e protezione della memoria. Per ottenere un tempo di risposta ragionevole, i job possono essere scaricati dalla memoria centrale su disco e viceversa; in questi casi il disco serve come memoria ausiliaria della memoria centrale. Un modo comune per ottenere tale risultato è l'uso della memoria virtuale, una tecnica che consente l'esecuzione di job anche non interamente caricati in memoria. Il più evidente vantaggio di tale schema è che i programmatori possono avere dimensioni maggiori della memoria fisica; inoltre esso astrae la memoria centrale, in un grande e uniforme array di memoria, separando la memoria logica, come è vista dall'utente dalla memoria fisica; ciò solleva i programmatori dai problemi legati ai limiti della memoria. I sistemi time–sharing devono inoltre fornire un file system on–line residente su un insieme di dischi, i quali a loro volta necessitano di una gestione. Inoltre richiedono lo scheduling della cpu, e dei meccanismi per la comunicazione e la sincronizzazione dei job. Inoltre questi sistemi si devono assicurare che i job non inceppino in un deadlock.

1.5 PERSONAL COMPUTER

Quando i costi dell'hardware sono diminuiti, è stato di nuovo possibile dedicare un solo sistema di calcolo a un singolo utente. Questi sistemi di calcolo sono chiamata generalmente personal computer (PC). Invece di puntare al massimo utilizzo della cpu e delle periferiche adesso i sistemi devono rispondere a requisiti di convenienza e prontezza d'uso per l'utente.

1.6 SISTEMI PARALLELI

Oggi la maggior parte dei sistemi è costituita da un unico processore, tuttavia su fa sempre più marcata la tendenza all'adozione di sistemi multiprocessore. Questo tipo di sistema dispone di più processori in comunicazione stretta, che condividono il bus del computer, il clock e talvolta i dispositivi si memoria e periferici. Questi sistemi sono noti anche come sistemi strettamente accoppiati (tightly coupled).
I motivi che hanno spinto alla costruzione di questi sistemi sono molteplici. Un primo vantaggio è il maggiore throughput (quantità di job elaborati per unità di tempo). Lavorando con n processori la velocità non aumenta comunque di n volte, ma in misura minore. Infatti, quando più processori collaborano all'esecuzione di un compito, il SO assorbe una certa quantità di tempo per gestire le operazioni che garantiscano che tutte le componenti funzionino correttamente. Questo tempo, chiamato overhead, aggiunto alla conflittualità dovuta alle risorse condivise, riduce il guadagno di più processori, così come un gruppo di n programmatori non produce n volte quanto verrebbe prodotto da un solo programmatore.
I sistemi multiprocessore possono inoltre consentire dei risparmi rispetto a più sistemi singoli poiché i processori possono condividere periferiche, chassis e alimentatori. Un secondo vantaggio è l'incremento dell'affidabilità. Se le funzioni possono essere distribuite adeguatamente tra più processori, un guasto ad un processore non blocca il sistema, ma semplicemente lo rallenta. La possibilità di continuare il servizio in misura proporzionale al livello di hardware correttamente funzionante è chiamata degradazione controllata (graceful degradation). I sistemi strutturati in modo da garantire una degradazione controllata sono detti anche tolleranti ai guasti (fault–tolerant).
Per continuare le operazioni anche in presenza di guasti occorre un meccanismo che permetta l'identificazione, la diagnosi e, se possibile, la correzione dei guasti stessi. Il sistema Tandem impiega un duplicato di hardware e software.
I sistemi multiprocessore più comuni oggi utilizzano il modello multiprocessing simmetrico, nel quale ciascun processore esegue una copia identica del SO; queste copie comunicano fra di loro quando necessario. Alcuni sistemi utilizzano il multiprocessing asimmetrico, nel quale ad ogni processore viene assegnato un compito specifico. Un processore master controlla il sistema; gli altri processori aspettano istruzioni dal master, oppure hanno già compiti predefiniti. Questo schema definisce una relazione master–slave. Il processore master effettua lo scheduling e alloca il lavoro ai processori slave.
Il multiprocessing asimmetrico è più diffuso in sistemi molto grandi, nei quali la gestione degli I/O comporta un notevole dispendio di tempo. Nei vecchi sistemi batch erano usati piccoli processori, posizionati ad una certa distanza dalla cpu centrale, per gestire i lettori di schede e le stampanti e trasferire i relativi job nel e dal computer principale. Queste locazioni sono chiamate remote job entry (RJE).

Con la riduzione dei costi e l'aumento della potenza dei microprocessori, ai processori slave sono affidate sempre più funzioni di un SO. È abbastanza semplice, ad esempio, impiegare un microprocessore e la sua memoria per gestire un sistema a disco. Il microprocessore può ricevere una sequenza di richieste dalla cpu principale e realizzare la propria coda per il disco ed eseguire il proprio algoritmo di scheduling. Questo sistema evita alla cpu principale il sovraccarico dello scheduling. La tastiera dei PC, ad esempio, contiene un microprocessore che converte le sequenze di pressioni dei tasti in codici da inviare alla cpu. Questo tipo di impiego dei microprocessori si è diffusa a tal punto che non lo si considera più multiprocessing.

1.7 Sistemi distribuiti

Nel campo dei sistemi di calcolo si è recentemente diffusa la tendenza a distribuire il calcolo tra diversi processori, e al contrario dei sistemi tightly coupled, i processori non condividono né memoria né clock. Ciascun processore dispone di una memoria locale. I processori comunicano tra loro attraverso linee si comunicazione, come bus ad alta velocità o linee telefoniche. Generalmente questi sistemi sono chiamati debolmente accoppiati (loosely coupled) o sistemi distribuiti.
I processori di un sistema distribuito possono avere dimensioni e funzioni diverse, possono contenere piccoli microprocessori, stazioni di lavoro, minicomputer e grossi sistemi. A questi processori sono attribuiti nomi diversi, come siti, nodi, computer…
I sistemi distribuiti vengono costruiti per diversi motivi, i più importanti dei quali sono:

- Condivisione di risorse. Se più siti, con diverse capacità, sono collegati tra loro, l'utente di un sito ha la possibilità di usare risorse disponibili sull'altro sito
- Accelerazione del calcolo. Quando un calcolo può essere suddiviso in più sottoclassi e possibile ottenere un'esecuzione concorrente di questi ultimi distribuendoli fra i diversi siti di un sistema distribuito. Inoltre, se in sito particolare è temporaneamente sovraccarico di job, alcuni di questi possono essere trasferiti ad altri siti con minor carico. Questo spostamento di job viene chiamato anche condivisione di carico.
- Affidabilità. Se un sito apparentemente a un sistema distribuito si guasta, i restanti siti possono potenzialmente continuare a lavorare.
- Comunicazione. In molti casi i programmi richiedono lo scambio di dati su un unico sistema, come ad esempio nei sistemi con finestre dove si ha condivisione o trasferimento di dati tra le finestre stesse. Se più siti sono collegati tra loro tramite una rete di comunicazione, i processi nei diversi siti hanno la possibilità di scambiarsi informazioni. Ad esempio, gli utenti possono effettuare trasferimenti di file o comunicare tra loro per mezzo della posta elettronica; la comunicazione può avvenire nello stesso sito o tra siti diversi.

1.8 Sistemi real–time

Un'altra forma di SO è il sistema real–time. Un sistema real–time è usato quando sono richiesti rigidi vincoli di tempo sulle operazioni del processore o sul flusso di dati, quindi è spesso impiegato come dispositivo di controllo per un'applicazione dedicata. I sensori portano i dati al computer che deve analizzarli e regolare i controlli in modo da

modificare gli input dei sensori. I sistemi che controllano esperimenti scientifici, come i sistemi per la rappresentazione di immagini in medicina, i sistemi di controllo industriale e alcuni sistemi di visualizzazione sono tutti sistemi in real–time. Un SO real–time presenta vincoli di tempo fissati e ben definiti entro i quali deve essere effettuata l'elaborazione. Ad esempio non sarebbe accettabile che il braccio di un robot ricevesse l'istruzione di fermarsi dopo aver già distrutto la macchina che stava costruendo.
Esistono due tipi di sistemi real–time. Un sistema hard real–time garantisce che i compiti critici siano completati in un dato intervallo di tempo. L'uso di memoria secondaria è limitato, poiché i dati sono memorizzati in una memoria a breve termine o in una di sola lettura (ROM). I requisiti dei sistemi real–time sono quindi in conflitto con le modalità di funzionamento dei sistemi time–sharing. Nessuno degli attuali SO general purpose supporta modalità hard real–time.
Sistemi real–time con caratteristiche meno restrittive sono i sistemi soft real–time dove i task critici hanno priorità sugli altri task e la mantengono fino al completamento dell'esecuzione. Come nei sistemi hard real–time le attese del kernel devono essere limitate: un task real–time non può restare indefinitamente in attesa che il kernel lo metta in esecuzione.

2. STRUTTURE DEI SISTEMI DI CALCOLO

2.1 FUNZIONAMENTO DI UN PROGRAMMA DI CALCOLO

Un moderno computer general–purpose è composto da una CPU e da un certo numero di controller di dispositivi connessi attraverso un bus comune che permette l'accesso alla memoria condivisa del sistema.
L'avvio di un computer, conseguente all'accensione fisica del sistema, così come il riavvio di un computer già acceso, richiede la presenza di un particolare programma iniziale, di solito non troppo complesso, detto programma di bootstrap. La sua funzione è inizializzare le diverse componenti di sistema, e deve essere in grado di caricare in memoria il SO e avviarne l'esecuzione. Per far ciò deve localizzare il kernel del SO e caricarlo in memoria. Quindi il SO avvia l'esecuzione del primo processo, per esempio init, e si mette in attesa che si verifichi qualche evento, la cui occorrenza è di solito segnalata dalla ricezione di un interrupt generato dall'hardware o dal software.
L'hardware è in grado di causare un interrupt in qualsiasi istante inviando un segnale alla cpu, di solito attraverso il bus del sistema. Il software può causare un interrupt attraverso l'esecuzione di una speciale operazione detta system call, o monitor call.
Ogniqualvolta riceve un interrupt, la cpu interrompe l'elaborazione corrente e trasferisce immediatamente l'esecuzione in una fissata locazione di memoria.
L'interrupt deve causare il trasferimento del controllo all'appropriata routine di servizio dell'evento a esso associato. Il modo più semplice per gestire questa operazione sarebbe quello di impiegare una generica routine per esaminare le informazioni presenti nell'interrupt; essa, a sua volta, richiamerebbe il gestore dello specifico interrupt. D'altra parte la gestione di un interrupt deve essere estremamente rapida, perciò considerando che il numero dei possibili interrupt è predefinito, si potrebbe usare una tabella di puntatori alle specifiche routine. In genere la tabella dei puntatori contenente gli indirizzi

delle routine di servizio è mantenuta nella memoria bassa (ad esempio le prime 100 locazioni).
L'accesso a questo vettore di indirizzo, chiamato vettore di interrupt, avviene per mezzo di un indice, codificato nello stesso segnale di interrupt, allo scopo di fornire l'indirizzo della routine di servizio associata all'evento segnalato dall'interrupt. Sistemi operativi radicalmente differenti, quali Ms-Dos e Unix, usano lo stesso meccanismo di gestione degli interrupt.
Di solito gli altri interrupt sono disabilitati durante la gestione di un interrupt, ritardando il servizio di qualsiasi altro interrupt pendente fino a quando il sistema ha terminato con quello corrente, quindi gli interrupt sono nuovamente abilitati. Se non si seguisse questo modo di operare, l'elaborazione del secondo interrupt mentre è servito il primo causerebbe una sovrascrittura dei dati del primo interrupt, che diventerebbe così un interrupt perso. Architetture di interrupt più sofisticate consentono l'elaborazione di un interrupt durante il servizio di un altro. Tali architetture spesso fanno uso di uno schema a priorità nel quale a ogni tipo di richiesta è assegnato un livello di priorità in relazione all'importanza del servizio richiesto, e le informazioni sull'elaborazione dell'interrupt sono memorizzate separatamente per ogni livello di priorità. Un interrupt ad alta priorità è accettato anche se è attivo un interrupt con priorità minore: gli interrupt dello stesso livello o di livello inferiore sono invece mascherati o disabilitati selettivamente, al fine di impedire la perdita di un interrupt ed evitare la gestione di interrupt che possono essere ritardati.
I SO moderni sono interrupt driven: se non ci sono processi da eseguire, dispositivi di I/O da servire o utenti con cui interagire, un computer resterà inattivo, nell'attesa che succeda qualcosa. Il verificarsi di un evento è quasi sempre segnalato da un interrupt o da una trap. Un trap (o eccezione) è un interrupt generato dal software a causa di un errore (una divisione per zero o un accesso in memoria non valido) o a seguito di una richiesta specifica effettuata da un programma utente per ottenere l'esecuzione di un servizio del SO.

2.2 Struttura di I/O

Un computer general–purpose è composto da una cpu e da un insieme di controller di dispositivi connessi mediante un bus comune. Ciascun controller deve occuparsi di un particolare tipo di dispositivo e, in base alla sua natura, può gestire uno a più dispositivi a esso connessi. Un controller mantiene al suo interno un buffer locale e un insieme di registri speciali. Il controller è responsabile del trasferimento dei dati tra i dispositivi periferici a esso connessi e il proprio buffer locale. La dimensione del buffer può variare.

2.2.1 Interrupt di I/O

per iniziare un'operazione di I/O la cpu carica i registri appropriati del controller associato al dispositivo con il quale intende comunicare. Il controller, a sua volta, esamina il contenuto in questi registri allo scopo di determinare l'azione da compiere. Ad esempio, se il processore richiede l'esecuzione di un'operazione di lettura, il controller provvede al trasferimento dei dati dal dispositivo al proprio buffer locale. Una volta completato il trasferimento, il controller informa la CPU dell'avvenuto completamento

dell'operazione generando un interrupt. Una volta iniziata l'operazione di I/O, il corso dell'esecuzione può seguire due percorsi distinti. Nel caso più semplice, il controllo è restituito al processo utente solamente una volta completata l'operazione I/O: questo modo di operare prende il nome di I/O sincrono. L'altra possibilità, detta I/O asincrono, prevede la restituzione immediata del controllo al processo utente, senza attendere il completamento dell'operazione; in questo modo, l'operazione di I/O può proseguire mentre il sistema sta eseguendo altre operazioni.
L'attesa del completamento di un'operazione di I/O può essere realizzata in due modi diversi. Alcuni computer mettono a disposizione una particolare istruzione di attesa, wait, che provoca la sospensione della cpu fino al successivo interrupt. Nelle macchine che non prevedono un'istruzione del genere si può far uso di un ciclo di attesa:

Loop: jmp Loop

Il precedente ciclo dovrà inoltre verificare le eventuali richieste di comunicazione che tutti i controller dei dispositivi di I/O che non sono in grado di generare interrupt, ma che si limitano ad impostare un flag in uno dei propri registri aspettando che il sistema lo moti.
Se la cpu attende sempre il completamento dell'operazione di I/O si ha al più una richiesta pendente alla volta. Quindi, ogniqualvolta si verifica un interrupt di I/O il sistema operativo riconosce immediatamente il dispositivo che ha generato l'interrupt. D'altra parte questo approccio esclude la possibilità di I/O concorrenti su diversi dispositivi, così come esclude la possibilità di sovrapporre utili elaborazioni con l'I/O. Un'alternativa migliore consiste nell'avviare l'operazione di I/O quindi proseguire con l'elaborazione di altro codice del SO o di un programma utente. In questo caso è necessaria la presenza di una system call che consenta al programma utente di attendere, se richiesto, il completamento dell'operazione. Nel caso in cui nessun programma utente sia pronto per l'esecuzione, e il SO non abbia altro lavoro da fare, si ha ancora bisogno dell'istruzione o del ciclo di attesa di cui si è parlato in precedenza: il sistema deve inoltre poter tener traccia delle molteplici richieste di I/O eventualmente attive nello stesso istante. A questo scopo, il SO mantiene una tabella di stato dei dispositivi, composta da tanti elementi quanti sono i dispositivi di I/O connessi al computer. Ciascun elemento della tabella specifica il tipo di dispositivo cui fa riferimento, il suo indirizzo e il suo stato (disattivato, inattivo o occupato).
Se un dispositivo è impegnato a soddisfare una precedente richiesta, il tipo di richiesta è memorizzato assieme con altri parametri nell'elemento della tabella associato a quel dispositivo. Poiché altri processi possono sottoporre una richiesta allo stesso dispositivo, il SO mantiene una coda d'attesa per ciascun dispositivo di I/O.
Quando un controller di un dispositivo di I/O invia un interrupt al sistema per richiedere un servizio, il SO deve per prima cosa individuare il controller del dispositivo che ha generato l'interrupt. Quindi, accede alla tabella dei dispositivi, risale allo stato in cui il dispositivo si trova e modifica l'elemento della tabella riflettendo l'occorrenza dell'interrupt. La maggior parte dei dispositivi sfrutta l'interrupt per segnalare il completamento di una richiesta di I/O. Qualora ci fossero ulteriori richieste in attesa nella coda di quel dispositivo, il SO avvierebbe l'elaborazione della prima richiesta. Infine il controllo acquisito tramite l'interrupt è restituito.
Esistono dispositivi di input che seguono uno schema differente. Molti sistemi interattivi consento all'utente di inerire i dati dal proprio terminale prima ancora che il programma

cui sono diretti ne faccia richiesta. In questo caso, il controller associato al dispositivo lancia un interrupt, col quale segnala la presenza delle informazioni in arrivo dal terminale, anche se le informazioni registrate nella tabella del suo stato indicano che nessun programma ha richiesto input da quel dispositivo. Per consentire questa eventualità è necessario fornire un buffer nel quale memorizzare i dati inseriti dall'utente nell'attesa che un programma ne faccia richiesta. Solitamente esiste un buffer associato a ciascun terminale connesso al sistema. Il principale vantaggio dell'I/O asincrono è l'incremento dell'efficienza del sistema.

2.2.2 DMA

Si prenda in considerazione il driver di un terminale. Supponendo di scrivere dei caratteri su di un terminale dotato di velocità di trasmissione di 9600 bps, la frequenza con la quale il dispositivo è in grado di accettare e inviare caratteri è approssimativamente di 1 carattere ogni millisecondo (1000 microsecondi). Una routine ben codificata è in grado di trasferire nel buffer un carattere ogni 2 microsecondi, lasciando alla CPU 998 microsecondi su 1000 per altre elaborazioni o la gestione di altri interrupt. Questi parametri evidenziano i motivi per i quali alle operazioni di I/O gestite in modo asincrono sono solitamente associati interrupt a bassa priorità, consentendo a quelle più urgenti di essere gestite prima o, addirittura, di interrompere la gestione di interrupt a bassa priorità. D'altra parte un dispositivo ad alta velocità, come un nastro, o un disco o una rete di trasmissione dati può trasmettere informazioni a una velocità paragonabile a quella della memoria, perciò la cpu richiederebbe 2 microsecondi per servire interrupt che si verificano, ad esempio, ogni 4 microsecondi. Risulta evidente come una simile eventualità non lascerebbe molto spazio ad altri processi.
Il problema può essere risolto con l'ausilio di una tecnica chiamata DMA (Direct Memory Access). Una volta impostati i buffer, i puntatori e i contatori da associare al dispositivo di I/O, il controller trasferisce un intero blocco di dato dal proprio buffer interno direttamente in memoria, o viceversa, senza intervento da parte del processore. In questo modo il trasferimento richiede la generazione di un solo interrupt per ogni blocco di dati trasferito, piuttosto che per ogni byte come avviene per la gestione dei dispositivi più lenti.
Per quanto riguarda le operazioni di base della cpu il principio è lo stesso. Un programma utente, o lo stesso SO, richiede il trasferimento di un insieme di informazioni. Il sistema ricerca un buffer (vuoto in caso in operazioni di input, pieno in caso di output) in un gruppo di buffer; le dimensioni di questi buffer generalmente variano da 128 a 4096 byte, in base al tipo di dispositivo. Quindi una componente del SO, chiamata driver di dispositivo, imposta gli appositi registri del controller DMA affinché questo impieghi gli appropriati indirizzi di sorgente e destinazione e la lunghezza del blocco da trasferire. Il controller DMA quindi riceve l'istruzione d'avvio dell'operazione di I/O; mentre il controller DMA esegue il trasferimento dati, la cpu è libera di eseguire altri compiti. Siccome la memoria generalmente può trasferire una sola parola per volta, il controller DMA sottrae cicli di memoria alla cpu; ciò può rallentare l'esecuzione da parte della cpu mentre è in atto il trasferimento DMA. Il controller DMA notificherà il completamento dell'operazione inviando un interrupt alla cpu.

2.3 STRUTTURA DELLA MEMORIA

Per poter essere eseguito da un computer , un programma deve risiedere nella memoria centrale di quest'ultimo. Infatti, la memoria centrale è la sola area di memoria di grandi dimensioni direttamente accessibile dalla cpu; essa è strutturata come un vettore di parole o byte, di dimensioni variabili tra le centinaia di migliaia e le centinaia di milioni di elementi. Ciascuna parola possiede un proprio indirizzo. L'interazione avviene per mezzo di una sequenza di istruzioni load e store opportunamente indirizzate. Oltre alle operazioni load e store esplicitamente richieste, la cpu preleva automaticamente dalla memoria centrale le istruzioni da eseguire.
Una sequenza tipica di un'istruzione in un sistema con architettura von Neumann comincia con il prelievo (fetch) di un istruzione dalla memoria centrale e il trasferimento nel registro di istruzione. L'istruzione è quindi decodificate e può eventualmente richiedere il trasferimento di alcuni operando dalla memoria in alcuni registri interni.
In linea di principio, è auspicabile che tanto i programmi quanto i dati da essi trattati risiedano permanentemente nella memoria centrale. Questo non è possibile per i seguenti due motivi:

1. La capacità della memoria centrale non è di solito sufficientemente grande per contenere permanentemente tutti i dati richiesti
2. la memoria centrale è un dispositivo di memorizzazione volatile

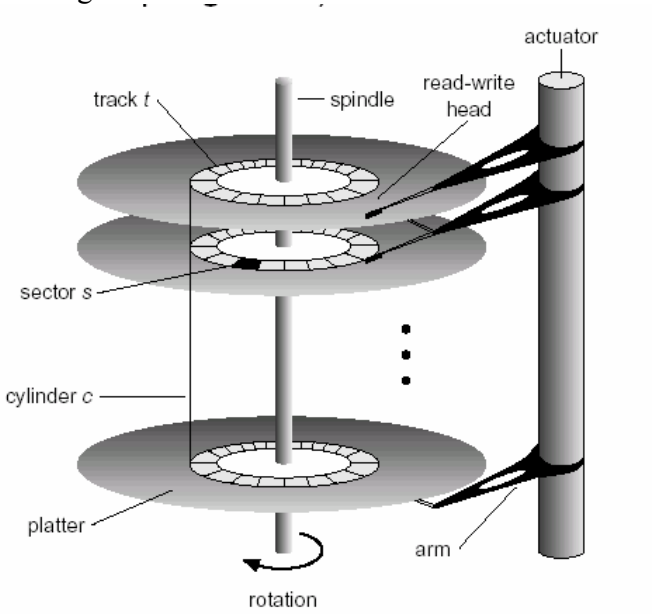
Per queste ragioni la maggior parte dei sistemi di calcolo comprende una memoria secondaria come estensione della memoria centrale. Il requisito fondamentale di questi dispositivi è la capacità di mantenere in modo permanente una grossa quantità di informazioni.

2.3.1 MEMORIA CENTRALE

Nel caso delle operazioni di I/O ciascun controller è dotato di registri nei quali mantenere i comandi e i dato da trasferire. Solitamente speciali istruzioni di I/O permettono il trasferimento dei dati tra questi registri e la memoria centrale.
Per rendere più conveniente l'accesso ai dispositivi di I/O molte architetture di computer forniscono una particolare tecnica di I/O che prende il nome di I/O mappato in memoria. Questa tecnica consiste nel riservare un certo insieme di indirizzi di memoria da associare ai registri dei dispositivi. Questo metodo è particolarmente adatto alla gestione di dispositivi dotati di brevi tempi di risposta, come i controller video.
L'uso dell'I/O mappato in memoria è conveniente anche per altri dispositivi come le porte seriali e parallele, usate ad esempio per collegare ai computer modem e stampanti. La cpu trasferisce i dati attraverso questo tipo di dispositivi, detti porte di I/O, leggendo e scrivendo in alcuni registri in esso contenuti. Per inviare una sequenza di byte tramite una porta seriale mappata in memoria, la cpu scrive un byte nel registro dei dati, quindi imposta un bit nel registro di controllo per segnalare che il byte è disponibile. Il dispositivo riceve tale byte quindi cancella il bit nel registro di controllo per segnalare che è pronto a ricevere un nuovo byte. La Cpu può così trasferire il byte successivo. Il metodo nel quale la cpu impiega l'interrogazione ciclica del bit di controllo, per verificare se il dispositivo è pronto, è detto I/O programmato. Il metodo che non prevede l'interrogazione ciclica ma l'attesa di un interrupt che segnala che il dispositivo è pronto per il byte successivo è detto interrupt driven.

2.3.2 DISCHI MAGNETICI.

I piatti dei dischi hanno una forma piana e rotonda come quella dei cd, con un diametro che comunemente varia tra 1,8 e 5,25 pollici, e le due superfici ricoperte di materiale magnetico simile a quello de nastri magnetici; le informazioni sono memorizzate registrandole magneticamente sui piatti. Le testine di lettura–scrittura sono sospese su ciascuna superficie d'ogni piatto e sono attaccate al braccio del disco che le muove in blocco. La superficie di un piatto è divisa logicamente in tracce circolari a loro volta suddivise in settori. L'insieme delle tracce corrispondenti a una posizione del braccio costituisce un cilindro.



Quando un disco è in uso, un motore lo fa ruotare ad alta velocità; la maggior parte dei dischi ruota ad una velocità compresa tra i 3600 e i 9000 giri al minuto. La velocità di un disco è caratterizzata da due valori: la velocità di trasferimento, cioè la velocità con cui i dati fluiscono dall'unita disco al computer, e il tempo di posizionamento, talvolta detto tempo d'accesso casuale, che consiste nel tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato, detto tempo di ricerca, e del tempo necessario affinché il settore desiderato si porti, ruotando, sotto la testina, setta latenza rotazionale. Un unità disco è connessa a un computer attraverso un insieme di fili detto bus di I/O; di tale bus esistono diversi tipi (es. EIDE e SCSI). Il trasferimento dei dati su un bus è eseguito da speciali processori elettronici detti controller: l'host controller è il controller posto all'estremità relativa al computer del bus; il disk controller è incorporato in ciascuna unità a disco. Per eseguire un'operazione di I/O il computer inserisce un comando nell'hot controller — tipicamente mediante delle porte di I/O mappate in memoria —, esso invia il comando tramite dei messaggi al disk controller che agisce sull'hardware dell'unità disco per portare a termine il comando. I controller dei dischi di solito hanno una cache incorporata: il trasferimento dei dati nell'unità disco avviene tra la cache e la superficie del disco; il trasferimento dei dati con l'host avviene, alla velocità propria dei dispositivi elettronici, tra la cache e l'host controller.

2.4 GERARCHIA DELLE MEMORIE

Le numerosi componenti di memoria di un sistema di calcolo possono essere organizzate in una struttura gerarchica sulla base della loro velocità e del loro costo. I dispositivi che

occupano posizioni più alte sono caratterizzate da un alto costo e da un'elevata velocità di servizio.
Un'altra caratteristica di rilevante importanza oltre il costo e la velocità e la volatilità delle informazioni.

2.4.1 CACHE

Il concetto di caching è un principio importante in un sistema di calcolo. Quando si deve accedere a una particolare informazione, innanzitutto di controlla se questa è già presente all'interno della cache; in tal caso si adopera direttamente la copia contenuta nella cache, altrimenti la si preleva dalla memoria centrale e la si copia nella cache poiché esiste un alta probabilità che questa informazione possa servire nuovamente.
La gestione della cache, data la capacità limitata di questi dispositivi, è un importante problema di progettazione.
La memoria centrale può essere considerata una cache veloce per la memoria secondaria, giacché i dati in essa contenuti devono essere riportati in memoria centrale per poter essere usati e qui devono risiedere prima di essere trasferiti nella memoria secondaria. Il movimento delle informazioni tra i vari livelli della gerarchia può essere quindi sia implicito che esplicito, a seconda dell'organizzazione tra l'hardware e il software del SO che lo controlla. Ad esempio, il trasferimento dei dati tra la cache e i registri della cpu è di solito una funzione hardware che non richiede alcun intervento da parte del SO. Diversamente, il trasferimento dei dati da disco alla memoria è di solito una funzione gestita dal SO.

2.4.2 COERENZA E CONSISTENZA.

Il problema della coerenza della cache è di solito risolto a livello hardware, e quindi gestito a un livello più basso di quello nel quale risiede il SO.
In un sistema distribuito la situazione diventa ancora più complessa, poiché può accadere che più copie (chiamate in questo caso repliche) dello stesso file siano mantenute su differenti computer, dislocati a loro volta in luoghi fisici diversi. Essendoci la possibilità di accedere e modificare concorrentemente queste repliche, è necessario fare in modo che qualsiasi modifica a una qualsiasi replica si rifletta quanto prima sulle altre.

2.5 PROTEZIONI HARDWARE

Al fine di migliorare l'utilizzazione del sistema, i SO resero possibile la condivisione delle risorse del sistema tra più programmi simultaneamente. L'introduzione dello spooling consentì l'esecuzione di un programma, mentre altri processi erano impegnati in operazioni di I/O; l'introduzione dei dischi consentì di mantenere simultaneamente i dati di più processi, mentre con la multiprogrammazione vi fu la possibilità di far risiedere contemporaneamente più programmi in memoria.
Un SO ben progettato deve garantire che un programma non possa compromettere il funzionamento degli altri programmi presenti nel sistema.
Molti errori di programmazione sono riconosciuti direttamente dall'hardware e sono di norma gestiti dal SO. Se un programma utente commette un'operazione illecita, ad

esempio tentando di eseguire un'istruzione non consentita o di accedere a una locazione di memoria al di fuori del proprio spazio di indirizzi, l'hardware invia una trap al SO: il trap trasferisce il controllo, come accade con gli interrupt, al SO attraverso il vettore di interrupt. Ogniqualvolta si verifica un errore in un programma, il SO deve porre fine all'esecuzione di tale programma. Questa situazione è gestita dallo stesso codice eseguito quando un utente richiede la terminazione anormale del proprio programma. L'immagine della memoria (dump) è di solito scritta su un file, ciò permette all'utente di esaminarla allo scopo di correggere e riavviare il programma.

2.5.1 DUPLICE MODO DI FUNZIONAMENTO

Per garantire il corretto funzionamento del sistema è necessario proteggere tanto il SO quanto gli altri programmi, unitamente ai loro dati, da qualsiasi programma malfunzionante. L'approccio seguito consiste nel fornire un supporto hardware che consenta la gestione di differenti modi di esecuzione. Sono necessari almeno due diversi modi di funzionamento: modo utente e modo monitor (detto anche modo supervisore, modo di sistema, o modo privilegiato). Per indicare quale modo sia attivo, all'hardware del computer è aggiunto un bit, detto bit di modo: monitor (0) o utente (1). Questo bit consente di stabilire se l'istruzione corrente è eseguita per conto del SO o per conto dell'utente.
All'avvio del sistema l'hardware è posto in modo monitor. Viene caricato il SO che provvede all'esecuzione dei processi utente in modo utente. Ogni volta che si verifica un interrupt o una trap l'hardware passa dal modo utente al modo monitor. Perciò quando il SO riprende il controllo del computer si trova in modo monitor. Prima di passare il controllo al programma utente, il sistema ripristina il modo utente.
Il duplice funzionamento (dual mode) consente sia di proteggere il SO dal comportamento degli utenti che di proteggere gli utenti dagli altri utenti. Questo livello di protezione è ottenuto definendo le istruzioni macchina in grado di causare danni allo stato del sistema come istruzioni privilegiate. Poiché l'hardware consente l'esecuzione di queste operazioni soltanto in modo monitor, se si tenta di eseguire in modo utente un'istruzione privilegiata l'hardware non lo esegue ma la tratta come un'istruzione illegale generando un trap al sistema operativo.
Se l'hardware non supporta questi due modi di funzionamento il SO può andare incontro a serie limitazioni. Ad esempio, il sistema MS-DOS è stato sviluppato per l'architettura 8088 Intel priva di bit di modo e, quindi, dei due modi operativi. Un programma utente ha la possibilità di cancellare il SO scrivendo nuove informazioni nelle locazioni in cui questo è memorizzato, e più programmi possono contemporaneamente scrivere sullo stesso dispositivo, con la possibilità di ottenere risultati disastrosi.

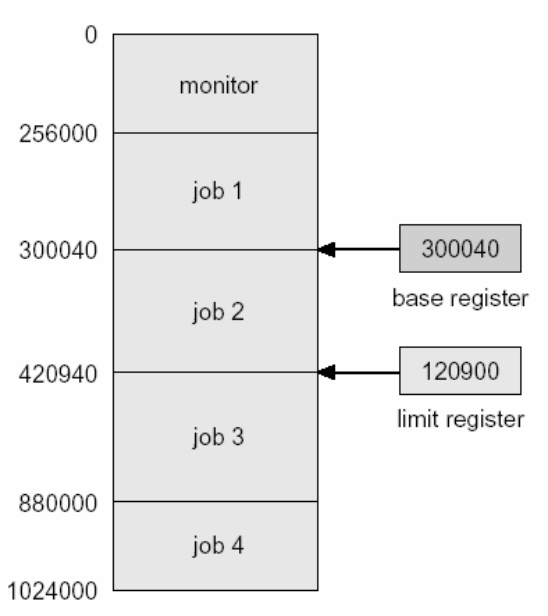
2.5.2 PROTEZIONE DELL'I/O

Un programma utente è in grado di alterare il normale funzionamento del SO effettuando operazioni illegali di I/O, accedendo a locazioni di memoria all'interno dello stesso SO o rifiutandosi di rilasciare il controllo della CPU. Esistono diversi meccanismi per garantire che ciò non possa avvenire.

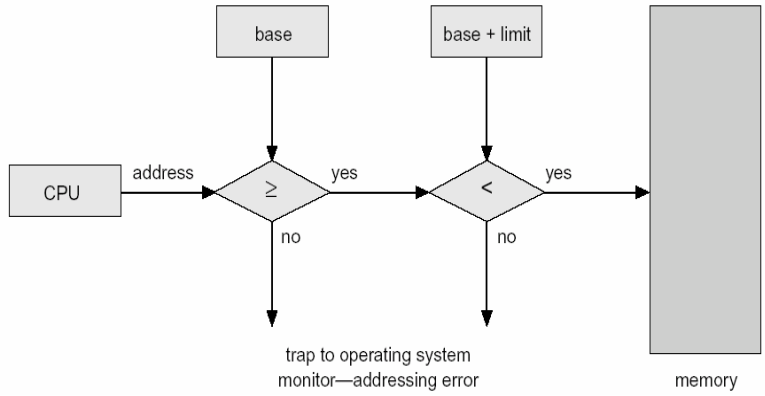
Allo scopo di impedire l'esecuzione di operazioni illegali di I/O da parte dell'utente, tutte le istruzioni di I/O sono definite come istruzioni privilegiate. In questo modo l'utente non può usare direttamente le istruzioni di I/O, ma deve farlo attraverso il SO. Affinché la protezione dell'I/O sia totale, è necessario evitare che l'utente possa in qualsiasi modo ottenere il controllo del computer quando è in modo monitor; se così non fosse, l'integrità delle operazioni di I/O potrebbe venir meno.
Si supponga che un computer si trovi momentaneamente nel modo utente. Il verificarsi di un interrupt o di una trap riporterà immediatamente il computer in modo monitor e trasferirà il controllo dell'esecuzione all'indirizzo contenuto nel vettore di interrupt. Se un programma utente riuscisse a modificare il contenuto di questo vettore, potrebbe scrivervi un nuovo indirizzo con un riferimento al proprio codice. Quindi, al verificarsi di un interrupt o di un trap corrispondente l'hardware passerebbe allo stato monitor e trasferirebbe il controllo, attraverso il vettore di interrupt modificato, al programma utente. Il processo utente riuscirebbe così ad ottenere il controllo del computer proprio mentre si trova nello stato monitor.

2.5.3 PROTEZIONE DELLA MEMORIA

Al fine di garantire il corretto funzionamento del sistema è necessario proteggere il vettore di interrupt da ogni possibile alterazione da parte dei programmi utenti. Inoltre è necessario proteggere anche le routine di servizio degli interrupt contenute nel codice del SO; altrimenti, un programma utente potrebbe sovrascrivere le istruzioni che costituiscono le routine di gestione degli interrupt, sostituendole con operazioni di salto direttamente indirizzate all'interno del proprio codice. In questo modo il programma utente potrebbe ancora ottenere il controllo del computer direttamente dalle routine di servizio degli interrupt eseguite in modo monitor.
In generale, comunque, la protezione va estesa all'intero SO. Questa protezione deve essere garantita dall'hardware e può essere realizzata in diversi modi.
Ciò che è necessario al fine di separare lo spazio di memoria associato a ogni singolo programma è la capacità di determinare l'intervallo di indirizzi legali cui il programma può accedere, proteggendo solo la memoria al di fuori di esso. Questo tipo di protezione è realizzata usando due registri, solitamente chiamati base e limite. Il registro base contiene il più basso indirizzo di memoria fisica al quale il programma può accedere legalmente, mentre il registro limite contiene la dimensione dell'intervallo. Ad esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, il programma può accedere legalmente alle locazioni di memoria compreso tra 300040 e 420940 inclusi.



Per ottenere questa protezione l'hardware della cpu confronta ogni indirizzo generato nel modo utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modo utente di accedere alle aree di memoria riservate al monitor o a una qualsiasi area di memoria associata ad altri utenti comporterà l'invio di una trap al monitor, il quale gestirà l'evento come un errore fatale.



Solamente il SO è in grado di modificare il contenuto dei registri base e limite usando una speciale istruzione privilegiata. Poiché istruzioni di questo tipo possono essere eseguite solo in modalità monitor, e poiché solo il SO può passare a questa modalità, solamente il SO ha la possibilità di modificare il valore di questi registri. Questo schema consente al monitor di modificare il valore di questi registri e impedisce che sia un programma utente a fare lo stesso.

2.6.4 Protezione della CPU

La terza tessera del mosaico delle protezioni consiste nell'assicurare che il SO mantenga il controllo dell'elaborazione. Ciò significa impedire che un programma utente entri in un ciclo infinito senza più restituire il controllo al SO. A tale scopo si può usare un timer, programmato affinché invii un interrupt al computer una volta trascorso un lasso di tempo specificato, che può essere fisso o variabile. Un timer variabile è solitamente realizzato mediante un clock a frequenza fissa e un contatore.
Prima di restituire all'utente il controllo dell'esecuzione, il SO assegna un valore al timer. Se il timer esaurisce questo intervallo genera un interrupt che provoca il trasferimento automatico del controllo al SO, il quale può decidere se gestire l'interrupt come un errore fatale o può concedere altro tempo al programma. Naturalmente, anche le istruzioni usate dal sistema per modificare il funzionamento del timer sono privilegiate.
La presenza di un timer garantisce che nessun programma utente possa essere eseguito troppo a lungo.
Più comunemente i timer sono usati per realizzare il time–sharing. Nel caso più immediato, il timer è istruito affinché interrompa l'esecuzione ogni N millisecondi, dove N è il quanto di tempo o time slice concesso a ciascun utente prima che il controllo ritorni al SO, il quale si preoccupa di eseguire le necessarie operazioni di amministrazione del sistema, come ad esempio sommare N all'informazione che (per scopi di contabilizzazione delle risorse) specifica la quantità di tempo che il programma utente ha speso in esecuzione fino a quel momento. Il SO si preoccupa inoltre si ripristinare i valori corretti dei registri, allo scopo di preparare l'esecuzione del programma successivo. Questa procedura prende il nome di cambio di contesto (context switch). Dopo ogni cambio di contesto il programma che riceve il controllo della CPU riprende l'esecuzione

dal punto esatto in cui era stata interrotta allo scadere del precedente quanto di tempo assegnatogli.

2.7 ARCHITETTURA DI UN GENERICO SISTEMA

Le istruzioni per il controllo dell'I/O e le istruzioni per la modifica dei registri di controllo della memoria o del timer sono istruzioni privilegiate.
Naturalmente esistono altre istruzioni che appartengono a questa categoria. Un esempio è costituito dall'istruzione halt, visto che un programma utente non deve avere la possibilità di arrestare il computer. Anche le istruzioni impiegate per l'abilitazione o la disabilitazione degli interrupt sono privilegiate, il corretto funzionamento del timer e dell'I/O dipendono dalla capacità di rispondere adeguatamente agli interrupt.
Essendo privilegiate, le istruzioni di I/O possono essere eseguite solamente dal SO. In che modo, allora, un programma utente riesce ad effettuare dell'I/O, visto che rendendo le operazioni privilegiate tale possibilità è stata negata? Poiché il monitor è l'unica componente software in grado di eseguire le operazioni di I/O, l'utente dovrà richiedere al monitor di eseguire l'I/O al suo posto.
Questa richiesta prende il nome di system call (o monitor call o operating system function call). In qualsiasi forma sia specificata, questo è il modo in cui un processo richiede al SO di eseguire una qualsiasi funzione. Solitamente le system call prendono la forma di un trap a una specifica locazione del vettore di interrupt. Tale trap può essere eseguito attraverso la generica istruzione trap, anche se alcuni sistemi hanno una specifica funzione syscall.
Attraverso il vettore di interrupt, il controllo passa all'apposita routine di servizio presente all'interno del SO e il bit di modo è posto al modo monitor. La routine di servizio della system call è parte integrante del SO. Il monitor esamina l'istruzione che ha causato l'interrupt al fine di stabilire la natura della sistema call, mentre un parametro di tale istruzione definisce il tipo di servizio richiesto dal programma utente. Ulteriori informazioni indispensabili per il completamento della richiesta possono essere ricopiate nei registri, sullo stack o direttamente nella memoria centrale. Il monitor verifica la correttezza e la legalità dei parametri, soddisfa la richiesta e restituisce il controllo dell'esecuzione all'istruzione immediatamente seguente la system call.

3 STRUTTURE DEI SISTEMI OPERATIVI

3.1 COMPONENTI DEL SISTEMA

3.1.1 GESTIONE DEI PROCESSI

Un processo può essere pensato come un programma in esecuzione. Per svolgere i propri compiti un processo necessita di alcune risorse, tra cui tempo della cpu, memoria, file e dispositivi di I/O. Queste risorse possono essere attribuite dal processo al momento della sua creazione, oppure essergli allocate durante l'esecuzione. Si sottolinea che un

programma non è di per sé un processo; un programma è un'entità passiva, come il contenuto di un file memorizzato su disco, mentre un processo è un'entità attiva, con un program counter che indica la successiva istruzione da eseguire. L'esecuzione di un processo deve avvenire in maniera sequenziale. La cpu esegue le istruzioni del processo una dopo l'altra fino a che il processo termina, inoltre a ogni istante viene eseguita al più una istruzione del processo. Quindi, anche se due processi possono essere associati allo stesso programma, questi sono comunque considerati due sequenze esecutive separate. Succede frequentemente che un programma generi molti processi durante il proprio svolgimento. Tutti questi processi possono potenzialmente essere eseguiti in concorrenza, scambiandosi vicendevolmente l'uso della cpu. Il SO è responsabile delle seguenti attività connesse alla gestione dei processi:

- Creazione e cancellazione dei processi utente e di sistema.
- Sospensione e ripristino dei processi.
- Fornitura di meccanismi per la sincronizzazione dei processi
- Fornitura di meccanismi per la comunicazione tra processi.
- Fornitura di meccanismi per la gestione dei deadlock

3.1.2 GESTIONE DELLA MEMORIA CENTRALE

Per eseguire un programma è necessario che questo sia associato a degli indirizzi assoluti e caricato in memoria. Durante l'esecuzione del programma, la cpu accede alle sue istruzioni e ai dati in arrivo dalla memoria.

Quando il programma termina, il suo spazio di memoria è dichiarato disponibile; a questo punto è possibile caricare ed eseguire il programma successivo.

Per migliorare l'utilizzo della cpu e la velocità con la quale il computer risponde ai sui utenti, occorre tenere molti programmi in memoria. Esistono diversi schemi di gestione della memoria derivanti da vari approcci; l'efficacia di ogni algoritmo dipende dalla situazione specifica.

Il SO è responsabile delle seguenti attività connesse alla gestione della memoria centrale:

- Tenere traccia di quali parti della memoria sono attualmente usate e da chi
- Decidere quali processi debbano essere caricati in memoria quando vi sia spazio disponibile
- Allocare e deallocare lo spazio di memoria in base alla necessità

3.1.3 GESTIONE DEI FILE

Per rendere conveniente l'utilizzo del computer, il SO fornisce una visione logica uniforme del processo di memorizzazione delle informazioni. Il SO astrae dalle caratteristiche fisiche dei propri dispositivi di memoria per definire un'unità di memoria logica, il file. Il SO associa i file ai supporti fisici e vi accede attraverso i dispositivi di memorizzazione. Il file è una raccolta di informazioni correlate definite dal loro creatore. I file rappresentano programmi, sia sorgente che oggetto, e dati. I file di dati possono essere numerici, alfabetici o alfanumerici. Il SO realizza il concetto astratto di file gestendo i supporti di memoria di massa, come nastri e dischi, e i dispositivi che li controllano. Inoltre i file sono generalmente organizzati in directory, che ne facilitano

l'utilizzo. Inoltre, se più utenti hanno accesso ai file, è necessario controllare chi ha la possibilità di accedervi e in che modo.
Il SO è responsabile delle seguenti attività connesse alla gestione dei file:

- Creazione e cancellazione dei file
- Creazione e cancellazione di directory
- Fornitura delle funzioni fondamentali per la manipolazione di file e directory
- Associazione dei file ai dispositivi di memoria secondaria
- Backup dei file su dispositivi di memorizzazione stabili, non volatili.

3.1.4 GESTIONE DEL SISTEMA DI I/O

Uno degli scopi di un SO è di nascondere all'utente le caratteristiche di specifici dispositivi hardware. In UNIX, ad esempio, le caratteristiche dei dispositivi di I/O sono nascoste alla maggior parte dello stesso SO dal sottosistema di I/O, che è composto delle seguenti parti:

- Un componente di gestione della memoria comprendente il buffering, il caching e lo spooling
- Un'interfaccia generale per i driver dei dispositivi
- I driver per gli specifici dispositivi hardware

Soltanto i driver del dispositivo conosce le caratteristiche dello specifico dispositivo cui è assegnato.

3.1.5 GESTIONE DELLA MEMORIA SECONDARIA

Lo scopo principale di un sistema di calcolo è quello di eseguire programmi. Durante l'esecuzione, questi programmi, insieme con i dati cui accedono, devono trovarsi in memoria centrale. Giacché la memoria centrale è troppo piccola per contenere tutti i dati e tutti i programmi e il suo contenuto va perduto quando il sistema viene spento, il computer deve disporre di una memoria secondaria a sostegno della memoria centrale.
Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei dischi:

- Gestione dello spazio libero
- Allocazione dello spazio
- Scheduling del disco

3.1.6 RETI

Un sistema distribuito è un insieme di processori che non condividono la memoria, i dispositivi periferici o un clock. Nel sistema i processori sono collegati da una rete di comunicazione che può essere configurata in vari modi, e può essere completamente o parzialmente connessa
Normalmente i SO generalizzano l'accesso alla rete come una forma di accesso ai file, dove i particolari riguardanti l'interconnessione sono contenuti nei driver del dispositivo d'interfaccia della rete stessa.

3.1.7 SISTEMA DI PROTEZIONE

Se un sistema di calcolo ha più utenti e consente che più processi siano eseguiti in concorrenza, i diversi processi devono essere protetti dalle attività di altri processi. Per tale ragione possono essere controllate solo dai processi che hanno ricevuto l'idonea autorizzazione dallo stesso SO. La protezione è quindi un meccanismo che controlla l'accesso da parte di programmi, processi o utenti alle risorse di un sistema di calcolo. Questo meccanismo deve fornire un mezzo che specifichi quali controlli debbano essere eseguiti e i mezzi per effettuarli

3.1.8 INTERPRETE DEI COMANDI

Uno dei programmi di un sistema è l'interprete dei comandi, che è l'interfaccia tra l'utente e il SO. Molti comandi vengono dati al SO attraverso istruzioni di controllo. Quando in un sistema batch si inizia un job nuovo, oppure quando un utente apre una sessione in un sistema con time–sharing, viene eseguito automaticamente un programma che legge e interpreta le istruzioni di controllo. Questo programma ha diversi nomi: interprete di schede di controllo, interprete di riga di comando, shell.

3.2 SERVIZI DI UN SISTEMA OPERATIVO

Un SO offre un ambiente in cui eseguire i programmi e fornire dei servizi ai programmi e ai loro utenti. Naturalmente i servizi specifici variano a seconda del SO, ma si possono identificare alcune classi di servizi comuni:

- Esecuzione di un programma: il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria terminazione normale o anormale
- Operazioni di I/O: un programma in esecuzione può richiedere un'operazione di I/O che implichi l'uso di un file o di un dispositivo di I/O. Per particolari dispositivi possono essere necessarie funzioni speciali, come il riavvolgimento di un unità nastro, oppure la pulizia dello schermo di un tubo a raggi catodici. Per motivi di efficienza e protezione in utente non può controllare direttamente i dispositivi di I/O, quindi il SO deve offrire i mezzi per farlo.
- Manipolazione del file system: il file system riveste un interesse particolare. I programmi effettuano operazioni di lettura e scrittura su file, oltre a creare e cancellare file.
- Comunicazioni: in molti casi un processo ha bisogno di scambiare informazioni con in altro processo. Esistono due modi principali per effettuare tail comunicazioni. Il primo ha luogo tra processi in esecuzione sullo stesso computer, mentre il secondo ha luogo tra processi in esecuzione su computer diversi collegati per mezzo di una rete. La comunicazione può essere realizzata tramite una memoria condivisa o attraverso la tecnica dello scambio di messaggi, nella quale il SO trasferisce pacchetti di informazioni tra i vari processi.
- Rilevamento di errori: il SO deve essere in grado di rilevare possibili errori che possono verificarsi nella cpu o nell'hardware della memoria, come un errore di

memoria o un guasto nell'alimentazione elettrica. Per ciascun tipo di errore il SO deve intraprendere l'azione giusta per assicurare un calcolo corretto e coerente.

Esiste anche un altro gruppo di funzioni riguardanti il SO che non è di ausilio per l'utente, ma assicura il funzionamento efficiente del sistema stesso.

- Allocazione delle risorse: il SO provvede all'allocazione delle risorse necessarie i job in esecuzione temporanea.
- Contabilizzazione dell'uso delle risorse. È possibile registrare quali utenti usino il computer, segnalando quali e quante risorse impieghino. Questo tipo di registrazione può essere utile per contabilizzare l'uso delle risorse, per addebitare il costo agli utenti, oppure per redigere statistiche.
- Protezione I proprietari di informazioni memorizzate in un sistema di calcolo multiutente possono desiderare di controllarne l'utilizzo. La sicurezza di un sistema inizia con l'obbligo da parte di ciascun utente di farsi identificare dal sistema, di solito attraverso delle parole d'ordine che permettano l'accesso alle risorse, e si estende alla difesa dei dispositivi di I/O esterni (compresi i modem e gli adattatori di rete) dai tentativi di accesso illegali e provvede al loro rilevamento.

3.3 SYSTEM CALL

Le system call costituiscono l'interfaccia tra un processo e il SO. Esse sono generalmente disponibili in forma di istruzioni in linguaggio assembly. Alcuni sistemi possono permettere che le system call siano invocate direttamente da un programma scritto in un linguaggio di alto livello. In tal caso si richiamano delle funzioni o delle procedure predefinite. Questi sistemi possono generare direttamente le system call, oppure generare una chiamata ad una speciale routine per la fase d'esecuzione che effettua la system call Per passare parametri al SO sono utilizzati tre metodi generali. Il più semplice consiste nel passare i parametri in registri. Si possono però presentare dei casi in cui vi sono più parametri che registri. Quando ciò si verifica, i parametri sono generalmente memorizzati in un blocco o tabella di memoria e l'indirizzo del blocco è passato in un registro in forma di parametro. Il programma può anche collocare (push) i parametri in uno stack da dove vengono prelevati (pop) dal SO. Le system call possono essere raggruppate approssimativamente in cinque categorie principali: controllo di processo, manipolazione di file, manipolazione di dispositivi, manutenzione delle informazioni e comunicazione

3.3.1 CONTROLLO DEI PROCESSI E DEI JOB

Un programma in esecuzione deve potersi fermare sia normalmente (nel qual caso si parla di end) che anormalmente (abort). Se per terminare in modo anormale un programma in esecuzione viene emessa una system call, oppure se il programma incontra delle difficoltà e causa una trap d'errore, talvolta si ha la registrazione su disco di un'immagine del contenuto della memoria (dump) e viene generato un messaggio d'errore. Sia in condizioni normali che anormali il SO deve trasferire il controllo all'interprete dei comandi che legge il comando successivo. In un sistema interattivo l'interprete dei comandi continua semplicemente a interpretare il comando successivo, supponendo che l'utente invii un comando idoneo per rispondere a qualsiasi errore. In un

sistema batch l'interprete dei comandi semplicemente termina il job e continua con il job successivo.
Un processo o un altro job che esegue un programma può richiedere di caricare (load) ed eseguire (execute) un altro programma. In questo caso l'interprete dei comandi esegue un programma come se tale richiesta fosse stata impartita, ad esempio, da un comando utente, oppure dal clic di un mouse. È interessante chiedersi dove debba ritornare il controllo una volta terminato il programma caricato. La questione è legata all'eventualità che il programma attuale sia andato perso, salvato, oppure che abbia continuato l'esecuzione in concorrenza con il nuovo programma.
Se al termine del nuovo programma il controllo rientra nel programma esistente, l'immagine della memoria del programma attuale deve essere salvata, creando così effettivamente un meccanismo con cui un programma può chiamare un altro programma. Se entrambi i programmi continuano in concorrenza, si è creato un nuovo job o processo da sottoporre a multiprogrammazione. Spesso a questo scopo è fornita una system call specifica, e precisamente create process o submit job.
Quando si crea un job o un processo nuovo, o un gruppo di job o di processi, è necessario mantenere il controllo. Questo controllo richiede l'abilità di determinare e reimpostare gli attributi di un job o di un processo, compresi la priorità, il suo tempo massimo di esecuzione e così via (get process attributes e set process attributes). Inoltre, può essere necessario terminare un job o un processo creato se si riscontra che non è corretto o non serve più (terminate process).
Una volta creati nuovi job o nuovi processi, può essere necessario attendere che essi terminino la loro esecuzione. Questa attesa può essere impostata per un certo periodo di tempo (wait time), ma è più probabile che si preferisca attendere che si verifichi un determinato evento (signal event).
Un altro gruppo di system call è utile per la messa a punto di un programma. Molti sistemi forniscono una system call per il dump della memoria. Un trace di programma elenca ogni istruzione nel momento in cui viene eseguita.
Il controllo di processi di job può essere effettuato in molti modi diversi. Il sistema operativo Ms-Dos è un esempio di sistema single–tasking, che dispone di un interprete dei comandi richiamato all'avvio del computer. L'MS–DOS, essendo un sistema single–tasking, esegue un programma impiegando un metodo semplice e senza creare alcun processo nuovo. Esso carica il programma in memoria, riscrivendo il più possibile anche sulla memoria che esso occupa, in modo da lasciare al programma quanta più memoria è possibile; quindi imposta il program counter sulla prima istruzione. A questo punto il programma va in esecuzione e possono verificarsi due condizioni: un errore causa una trap, oppure il programma esegue una system call per terminare la propria esecuzione. In entrambi i casi il codice d'errore è salvato nella memoria di sistema per un'eventuale uso successivo, quindi quella piccola parte dell'interprete che non era stata soprascritta riprende l'esecuzione e il suo primo compito consiste nel ricaricare da disco la parte rimanente dell'interprete stesso. Eseguito questo compito, quest'ultimo mette a disposizione dell'utente, o del programma successivo, il codice di errore registrato.
Benché il SO MS-DOS non abbia capacità generali di multitasking, fornisce un metodo per una sia pur limitata esecuzione concorrente. Un programma TSR è un programma che intercetta un interrupt, quindi termina la propria esecuzione con la system call terminate and stay resident. Esso può ad esempio intercettare l'interrupt del clock mettendo

l'indirizzo di una delle proprie sottoprocedure nella lista delle procedure da chiamare quando è attivato il timer di sistema; in questo modo la procedura TSR è eseguita diverse volte al secondo, a ogni impulso di clock. La system call terminate and stay resident fa sì che l'MS-DOS riservi lo spazio occupato dal TSR affinché non venga soprascritto quando è nuovamente caricato l'interprete dei comandi.
Berkley Unix è un esempio di sistema multitasking. Quando un utente effettua un login, è eseguito un interprete dei comandi, chiamato shell. Poiché unix è un sistema multitasking l'interprete dei comandi può continuare l'esecuzione mentre viene eseguito un altro programma.
Per avviare un processo nuovo, la shell esegue una system call fork; il programma selezionato è quindi caricato in memoria tramite una system call exec e infine eseguito. A seconda di come il comando era stato impartito, la shell attende che il comando finisca, oppure esegue il processo in background. In quest'ultimo caso la shell richiede immediatamente un altro comando. Se un processo è eseguito in background, non può ricevere input da tastiera, visto che la shell sta usando tale risorsa. Nel frattempo l'utente è libero di chiedere alla shell di eseguire altri programmi, di controllare lo svolgimento del processo in esecuzione, di modificare la priorità di quel programma e così via.
Terminato il proprio compito, il processo esegue una system call exit per terminare, riportando al processo chiamante un codice di stato 0 oppure un codice di errore diverso da 0. Questo codice di stato o di errore rimane disponibile per la shell o per altri programmi.

3.3.2 MANIPOLAZIONE DEI FILE

Innanzitutto è necessario poter creare (create) e cancellare (delete) files. Ogni system call richiede il nome del file e probabilmente anche altri attributi. Una volta creato il file è necessario aprirlo (open) e usarlo. Si può anche leggere (read), scrivere (write) o riposizionare (reposition), ad esempio riavvolgendo e saltando all'inizio del file. Si deve infine poter chiudere (close) un file per indicare che non lo si sta più utilizzando.
Queste stesse operazioni possono essere necessarie anche per le directory. È inoltre necessario poter determinare i valori degli attributi dei file o delle directory ed eventualmente modificarli. Per questa funzione sono richieste due system call: get file attributes e set file attributes. Alcuni SO forniscono molte più system call.

3.3.3 GESTIONE DEI DISPOSITIVI.

Poiché i file possono essere immaginati come dispositivi astratti o virtuali, molte system call necessarie per i file sono usate anche per i dispositivi. Tuttavia, se un sistema ha più utenti, occorre prima richiedere (request) l'uso del dispositivo, per assicurarsene l'esclusiva, quindi occorre rilasciarlo (release). Una volta richiesto e allocato il dispositivo, è possibile effettuare una lettura (read), una scrittura (write) ed eventualmente un riposizionamento.

3.3.4 GESTIONE DELLE INFORMAZIONI

Molte system call hanno semplicemente lo scopo di trasferire informazioni tra il programma utente e il SO. La maggior parte dei sistemi, ad esempio, ha una system call per richiamare l'ora o la data del sistema. Altre system call possono richiamare informazioni sul sistema, come il numero degli utenti collegati, il numero della versione del sistema operativo, la quantità di memoria disponibile su disco…
Il SO contiene informazioni sui propri job e processi; a queste informazioni si può accedere tramite alcune system call. In genere esistono anche system call per modificare le informazioni sui processi (get process attributes e set process attributes).

3.3.5 COMUNICAZIONE

Esistono due modelli molto diffusi per la comunicazione:

- Nel modello a scambio di messaggi le informazioni sono scambiate per mezzo di una funzione di comunicazione tra processi messa a disposizione dal SO. Prima di effettuare la comunicazione occorre aprire un collegamento. Il nome dell'altro comunicante deve essere conosciuto, sia questo un altro processo sulla stessa cpu, oppure il processo di un altro computer collegato attraverso una rete di comunicazione. Tutti i computer di una rete hanno un nome host con il quale sono conosciuti. Analogamente, ogni processo ha un nome processo, che viene convertito in un identificatore equivalente grazie al quale il SO può farvi riferimento. La conversione in identificatore è effettuata dalle system call get hostId e get processId. Questi identificatori sono quindi passati alle system call di uso generale open e close messe a disposizione dal file system, oppure, secondo il modello di comunicazione del sistema, dalle system call specifiche open (close) connection. Generalmente il processo ricevente deve acconsentire alla comunicazione con un call accept connection. Nella maggior parte dei casi i processi che gestiscono la comunicazione sono dei daemon speciali, cioè programmi di sistema realizzati esplicitamente per questo scopo. Questi programmi eseguono una system call wait for connection e vengono chiamati in causa quando si stabilisce un collegamento. L'origine della comunicazione, conosciuta come client, e il daemon ricevente, conosciuto come server, possono quindi scambiarsi messaggi per mezzo delle system call read (write) message. La system call close connection pone fine alla comunicazione.
- Nel modello con memoria condivisa, invece, per accedere alle aree di memoria possedute da altri processi, i processi usano system call map memory. Occorre ricordare che, normalmente, il SO tenta di impedire a un processo l'accesso alla memoria di un altro processo. La memoria condivisa richiede che due o più processi concordino col rimuovere tale restrizione; a questo punto tali processi possono scambiarsi le informazioni leggendo e scrivendo dati nelle memorie condivise.

Lo scambio di messaggi è utile soprattutto quando è necessario trasferire un basso numero di dati, dal momento che, in questo caso, non sussiste la necessità di evitare conflitti e inoltre è più facile da realizzare rispetto alla condivisione di memoria per la comunicazione tra computer diversi.

La memoria condivisa permette la massima velocità e convenienza nelle comunicazioni, dal momento che queste ultime, quando avvengono all'interno del computer, possono essere effettuate alla massima velocità della memoria.

3.4 PROGRAMMI DI SISTEMA

Un'altra caratteristica importante di un sistema moderno è quello che riguarda la raccolta di programmi di sistema.
Alcuni di essi sono semplici interfacce per le system call, altri sono considerevolmente più complessi. Essi possono essere suddivisi nelle seguenti categorie:

- Manipolazione di file: questi programmi creano, cancellano, copiano… file e directory
- Informazioni di stato: Alcuni programmi richiedono semplicemente al sistema di indicare la data, l'ora, la quantità di memoria disponibile o lo spazio su disco, il numero degli utenti o le informazioni di stato L'informazione fornita è poi formattata e scritta sul terminale o su un altro dispositivo di output o su un file.
- Modifica dei file: possono essere disponibili diversi editor per creare o modificare il contenuto dei file disponibili su disco o su nastro.
- Supporto ai linguaggi di compilazione: compilatori, assembler, interpreti…
- Caricamento ed esecuzione dei programmi: loader, linker, debugger…
- Comunicazioni: questi programmi mettono a disposizione il meccanismo grazie al quale è possibile creare collegamenti virtuali tra i processi, utenti e computer diversi (messaggi, e-mail, trasferimento file, login remoto…)

Probabilmente il programma di sistema più importante è l'interprete dei comandi, la cui funzione principale consiste nel ricevere il comando successivo specificato da un utente ed eseguirlo.
Molti comandi impartiti a questo livello sono usati per la manipolazione di file, vale a dire per creare, cancellare, elencare… Questi comandi possono essere realizzati in due modi: in uno di questi è lo stesso interprete dei comandi a contenere il codice per eseguire il comando. Il comando per cancellare un file, ad esempio, può causare un salto dell'interprete dei comandi su una sezione del suo codice che imposta i parametri e invoca la system call idonea; in questo caso il numero dei comandi che possono essere dati determina la dimensione dell'interprete dei comandi, ciò poiché ogni comando richiede il proprio codice.
L'altro approccio, usato, ad esempio, in Unix, realizza la maggior parte dei comandi per mezzo di programmi di sistema speciali; in questo caso l'interprete dei comandi non comprende il comando, ma ne impiega semplicemente il nome per identificare un file da caricare in memoria per essere eseguito. Quindi un comando delete G ricerca un file chiamato delete, lo carica in memoria e lo esegue con il parametro G. La funzione associata al comando delete è interamente definita dal codice del file delete. In questo modo i programmatori possono aggiungere nuovi comandi al sistema, semplicemente creando nuovi file con il nome appropriato.
Per la maggior parte degli utenti la visione del SO risulta dunque definita dai programmi di sistema piuttosto che dalle system call.

3.5 STRUTTURA DEL SISTEMA

Un approccio diffuso consiste nella suddivisione del compito in piccole componenti piuttosto che progettare un sistema monolitico. Ciascuna di queste componenti deve essere un modulo ben definito del sistema, con input, output e funzioni definiti con precisione.

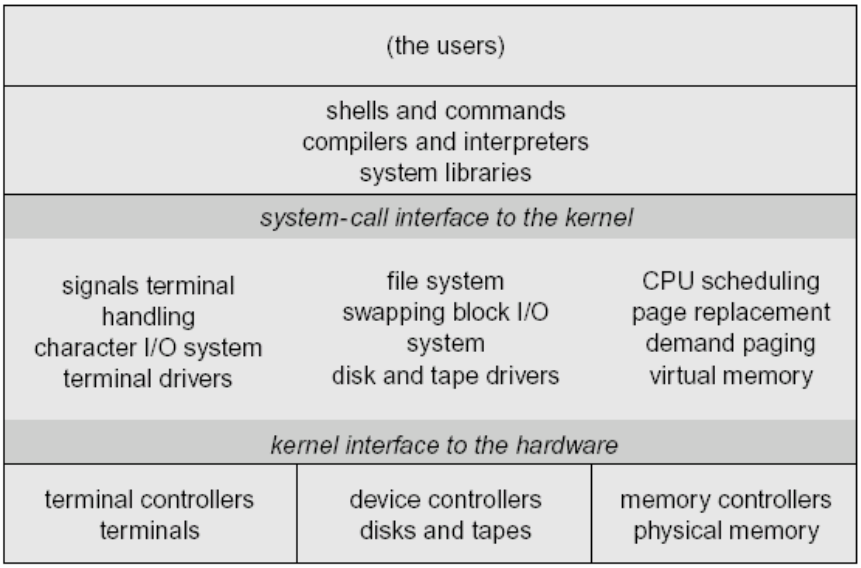
3.5.1 STRUTTURA SEMPLICE

Esistono numerosi sistemi commerciali che non hanno una struttura definita. Spesso sistemi operativi di questo tipo sono cresciuti superando il loro scopo originale. Un sistema di questo tipo è MS-DOS. Esso fu originariamente realizzato da persone che non avrebbero mai immaginato una simile popolarità. MS-DOS non fu diviso attentamente in moduli perché, a causa dei limiti dell'hardware su cui era eseguito, fu progettato con lo scopo prioritario di fornire la massima funzionalità nel minimo spazio.
Nell'MS-DOS le interfacce e i livelli di funzionalità non sono ben separati. Ad esempio, i programmi di applicazione possono accedere alle routine di base per le operazioni di I/O per scrivere direttamente sul video o sul disco. Queste libertà rende MS-DOS vulnerabile nei confronti di programmi errati o malvagi, che possono causare crolli di tutto il sistema o cancellazioni del disco in caso di programmi utente non funzionanti.
Un altro esempio di strutturazione limitata è quella del SO originale Unix; si tratta di un altro sistema che inizialmente era limitato dalla funzionalità dell'hardware. Unix è formato da due parti: il kernel e i programmi di sistema. Il kernel è a sua volta diviso in una serie di interfacce e driver dei dispositivi, che sono stati aggiunti ed espansi con l'evolversi di Unix stesso. Il SO Unix può essere considerato stratificato. Tutto quanto si trova sotto l'interfaccia delle system call e sopra l'hardware è il kernel. Per mezzo delle system call il kernel offre il file system, lo scheduling della cpu, la gestione della memoria e altre funzionalità riguardanti il SO. In un solo livello occorre combinare un'enorme quantità di funzionalità. Per fornire funzioni come la compilazione e la manipolazione di file, i programmi di sistema usano le system call supportate dal kernel. Le system call definiscono l'interfaccia per il programmatore di Unix. I programmi di sistema disponibili definiscono invece l'interfaccia utente. Le interfacce definiscono il contesto che il kernel deve realizzare. Sono state sviluppate parecchie versioni di unix in cui il kernel è ulteriormente partizionato lungo confini funzionali. Spostando tutto ciò che non è essenziali in programmi di sistema e persino programmi utenti si ottiene il microkernel (es. Mach).

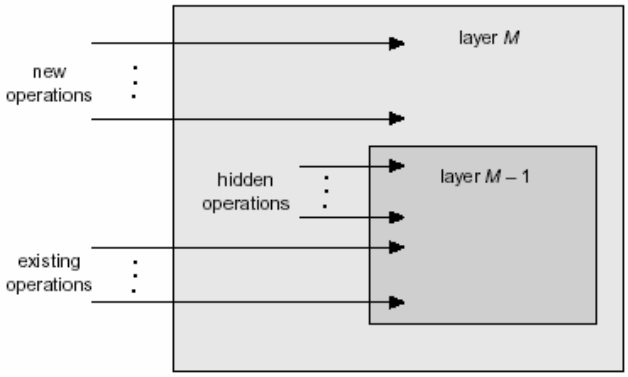
3.5.2 APPROCCIO STRATIFICATO

Le nuove versioni di Unix sono state progettate per usare un hardware più avanzato. Una volta assicurato un supporto hardware adeguato, i SO possono essere divisi in moduli più piccoli e più adatti rispetto a quelli disponibili nei sistemi Ms–Dos e Unix originali. Questo fatto consente al SO di mantenere un controllo maggiore sul computer e sulle applicazioni che fanno uso di tale computer, oltre a offrire maggiore libertà a chi desideri effettuare delle modifiche al funzionamento interno del sistema. La creazione di SO modulari è possibili grazie all'aiuto di tecniche conosciute. Seguendo un approccio top–down è possibile determinare e separare in componenti le funzionalità generali e le

caratteristiche. Anche l'information hiding è importante, poiché offre ai programmatori libertà di codificare come desiderano le routine di basso livello, ammesso che l'interfaccia esterna delle routine rimanga immutata e che la routine stessa esegua il compito a cui è preposta.



La modularizzazione di un sistema si ottiene in vari modi; il più affascinante è quello della stratificazione, che consiste nella divisione del SO in un certo numero di strati (livelli), ciascuno costruito sopra gli strati inferiori. Lo strato inferiore (strato 0) è l'hardware e lo strato superiore (strato N) è l'interfaccia utente.

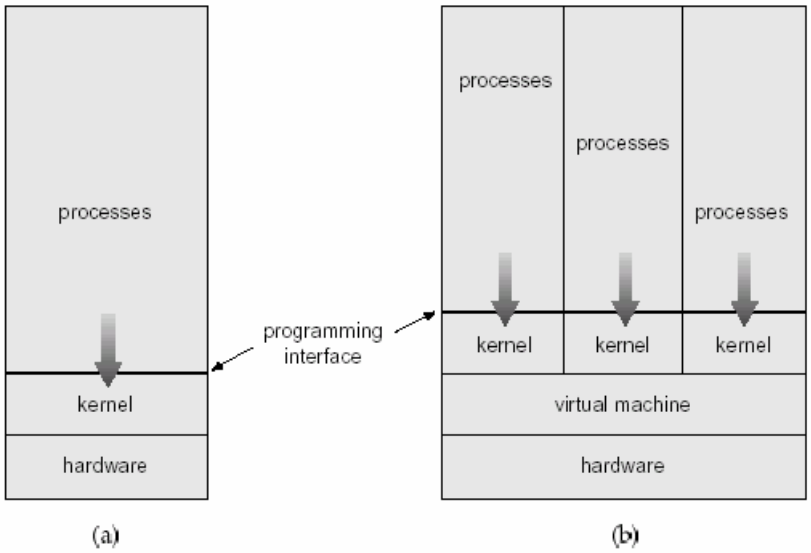


Un tipico strato di SO è formato da alcune strutture dati e da un gruppo di routine richiamabili dagli strati di livello superiore, mentre può richiamare operazioni sugli strati inferiori.
Il vantaggio principale offerto da questo tipo di approccio è la modularità. Questo approccio semplifica la messa a punto e la verifica del sistema. Il primo strato può essere messo a punto senza intaccare il resto del sistema, poiché, per definizione, tale livello usa solo l'hardware di base per realizzare le sue funzioni, e si presuppone che l'hardware sia corretto. Passando alla lavorazione del secondo strato si presuppone, dopo la messa a punto, la correttezza del primo. Il procedimento si ripete quindi per ogni strato.
Ogni strato è realizzato impiegando unicamente le operazioni messe a disposizione dagli strati di livello inferiore. Uno strato si limita a usare queste operazioni in base alle relative funzioni, senza entrare nel merito di come queste siano realizzate.
La maggiore difficoltà connessa all'approccio stratificato riguarda l'appropriata definizione degli strati. Giacché uno strato può usare solo strati che si trovano a un livello inferiore, la pianificazione degli strati deve essere effettuata con la massima attenzione.
Un ulteriore problema che si pone con la struttura stratificata è che essa tende ad essere meno efficiente delle altre; ad esempio, un programma utente per eseguire una operazione di I/O, esegue una system call che viene intercettata dal sottostrato di I/O che, a sua volta, esegue una chiamata alla gestione della memoria che attraverso lo scheduling

della cpu raggiunge l'hardware. In ciascun strato i parametri possono essere modificati, può essere necessario il passaggio di dati, e così via; ciascun strato aggiunge overhead alle system call.

3.6 MACCHINE VIRTUALI

Concettualmente il sistema di calcolo è costituito da strati. In tutti i sistemi di questo tipo l'hardware è il livello più basso. Il kernel, operante sul livello successivo, sfrutta l'hardware per creare un gruppo di system call che sono usate da strati esterni. Quindi, i programmi di sistema che si trovano sopra il kernel possono usare le system call o le istruzioni dell'hardware senza fare differenza tra le due. Così, sebbene il kernel e l'hardware siano raggiunti per diverse vie, contribuiscono entrambi a fornire funzionalità che il programma può usare per creare funzioni più avanzate. I programmi di sistema, a loro volta, trattano l'hardware e le system call come se fossero entrambe allo stesso livello.



(a) Macchina non virtuale; (b) Macchine virtuali

Alcuni sistemi estendono questo schema, permettendo ai programmi d'applicazione di chiamare i programmi di sistema. Anche in questo caso, sebbene i programmi di sistema si trovino ad un livello inferiore a quello delle altre routine, i programmi d'applicazione possono osservare quanto si trova a un livello gerarchico inferiore, come fosse parte della macchina stessa. L'approccio stratificato conduce in modo naturale al concetto di macchina virtuale. Usando lo scheduling della cpu e le tecniche di memoria virtuale un SO è in grado di creare l'illusione che siano in corso più processi, ciascuno operante sul proprio processore e con la propria memoria. Normalmente il processo dispone di ulteriori caratteristiche, come system call e un file system, che non sono offerte dall'hardware in quanto tale. D'altra parte l'approccio basato sulle macchine virtuali non offre alcuna funzione aggiuntiva, ma è piuttosto un interfaccia identica all'hardware sottostante. Ogni processo dispone di una copia (virtuale) del computer sottostante.
Le risorse del computer fisico sono condivise in modo da creare macchine virtuali. Lo scheduling della cpu può essere usato sua per condividere la cpu che per dare l'illusione che ciascun utente abbia una propria cpu. Lo spooling, unito al file system, consente di creare lettori di schede e stampanti virtuali. Un normale terminale di un sistema con time–sharing funge da console d'operatore della macchina virtuale.

Una grossa difficoltà legata all'approccio della macchina virtuale si incontra nella gestione dei dischi; si supponga, ad esempio, che la macchina fisica disponga di tre unità a disco, e che voglia realizzare sette macchine virtuali. Chiaramente, la macchina non è in grado di allocare un'unità disco per ogni macchina virtuale. Occorre ricordare che il software necessario alle macchine virtuali richiede un notevole spazio su disco per offrire le funzioni di memoria virtuale e spooling. La soluzione è offerta dai dischi virtuali, che sono identici sotto tutti i punti di vista, escluse le dimensioni, a quelli fisici. Il sistema realizza ogni minidisco allocando sui dischi fisici tante tracce quante ne richiede il minidisco.
Ogni utente dispone quindi della propria macchina virtuale. Gli utenti possono eseguire qualsiasi SO o programma che sia disponibile sulla macchina sottostante.

3.7 PROGETTAZIONE E REALIZZAZIONE DI UN SISTEMA

3.7.1 SCOPI DELLA PROGETTAZIONE

Il primo problema che si incontra nella progettazione di un sistema riguarda la definizione degli scopi e delle caratteristiche del sistema stesso. A più alto livello la progettazione del sistema è influenzata in modo determinante dalla scelta dell'hardware e del tipo di sistema: batch o time–sharing, mono o multiutente, distribuito, real–time o d'uso generale.
D'altra parte, oltre questo livello di progettazione, i requisiti risultano ancora più difficili da specificare. Questi possono essere distinti in due gruppi fondamentali: scopi dell'utente e scopi del sistema.

- Per gli utenti deve essere utile, facile da imparare e da usare, affidabile, sicuro, rapido.
- Per chi deve progettare: il sistema deve essere di facile progettazione, realizzazione e manutenzione, deve essere flessibile, affidabile, senza errori ed efficiente.

Stabilire le caratteristiche e progettare un SO è, dunque, un compito estremamente creativo. Il settore in cui sono sviluppati i principi generali è quello dell'ingegneria del software; alcune idee provenienti da questo settore sono applicabili soprattutto ai SO.

3.7.2 MECCANISMI E POLITICHE

Un principio molto importante è quello che riguarda la distinzione delle politiche dai meccanismi. I meccanismi determinano come eseguire qualcosa, le politiche cosa debba essere fatto.
La distinzione tra politica è meccanismo è molto importante ai fini della flessibilità. Le politiche sono soggette a cambiamenti da luogo a luogo o da momento a momento. Nel caso peggiore un cambiamento alla politica richiede un cambiamento anche al meccanismo sottostante. Sarebbe preferibile disporre di un meccanismo generale: in questo caso un cambiamento nella politica implicherebbe solo la ridefinizione di alcuni parametri del sistema. Ad esempio, se in un computer si decide che i programmi con uso intensivo di I/O debbano avere priorità su quelli con uso intensivo della cpu, ma si

dispone di un meccanismo adeguatamente separato e indipendente dalla politica, si potrebbe facilmente istituire, in un secondo momento, una politica opposta.
I SO basati su microkernel portano all'estremo la separazione dei meccanismi dalle politiche fornendo un insieme di funzioni fondamentali da impiegare come elementi di base; tali funzioni, quasi indipendenti dalle politiche, consentono l'aggiunta di meccanismi e politiche più avanzati attraverso moduli del kernel creati dall'utente o dagli stessi programmi utente. All'estremo opposto si trovano i sistema come Apple Macintosh, in cui i meccanismi e politiche sono codificati in modo da ottenere un ambiente omogeneo per l'intero sistema.
Le decisioni relative alla politica sono importanti per tutti i problemi di allocazione delle risorse e di scheduling. Ogni volta che è richiesta una decisione dell'allocazione di una risorsa è necessario prendere una decisione politica. Ogni volta che il problema riguarda il come piuttosto che il cosa occorre determinare un meccanismo.

3.7.3 REALIZZAZIONE

Una volta progettato, un SO deve essere realizzato. Tradizionalmente i SO erano scritti in linguaggio assembly, mentre oggi vengono scritti in linguaggio di alto livello.
I vantaggi derivanti dall'uso di un linguaggio ad alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, sono quelli che si ottengono quando il linguaggio è usato per i programmi d'applicazione; il codice viene scritto più velocemente, è più compatto, ed è facile da comprendere e mettere a punto. Inoltre il perfezionamento della tecnologia di compilazione consente di migliorare il codice generato per l'intero SO con una semplice ricompilazione. Infine, un SO scritto in un linguaggio di alto livello è facile da trasportare su un altro hardware.
I principali svantaggi solitamente evidenziati sono la minore velocità di esecuzione e la maggiore occupazione di spazio di memoria.
Come negli altri sistemi, i miglioramenti principali nel rendimento sono dovuti più a strutture dati e algoritmi migliori che a un ottimo codice in linguaggio assembly. Inoltre, sebbene i SO siano sistemi molto grandi, solo una piccola parte del codice assume un'importanza critica riguardo al rendimento: il gestore della memoria e lo scheduler della cpu sono probabilmente le routine più critiche. Una volta scritto il sistema e verificato il suo corretto funzionamento, è possibile individuare le routine più problematiche e sostituirle con routine equivalenti in linguaggio assembly.
Per identificare le routine problematiche occorre controllare il rendimento del sistema, aggiungendo strumento diagnostici.

3.8 GENERAZIONE DI SISTEMI

Il sistema deve quindi essere configurato o generato per ciascuna situazione specifica. Questo processo è conosciuto con il nome di generazione di sistemi (SYSGEN). Il SO è generalmente distribuito su nastro o su disco. Per generare un sistema è necessario usare un programma speciale che può leggere da un file o chiedere all'operatore le informazioni riguardanti la configurazione specifica dell'hardware; o anche esplorare l'hardware per determinarne i comportamenti. Sono necessarie le seguenti informazioni:

- La cpu che deve essere usata e le opzioni installate. Nel caso di sistemi con più cpu occorre anche descrivere ogni cpu.
- La quantità di memoria disponibile. Alcuni sistemi determinano autonomamente questi valori, accedendo a tutte le posizioni di memoria, fino a che viene generato un errore di indirizzo illegale.
- I dispositivi disponibili. Il sistema deve conoscere gli indirizzi di ogni dispositivo e sapere come farvi riferimento (numero di dispositivo), deve conoscere il numero di interrupt del dispositivo, il tipo e il modello del dispositivo e tutte le sue caratteristiche particolari.
- Le opzioni del SO richieste o i valori dei parametri che è necessario usare. Queste informazioni possono contenere il numero di buffer da usare e la loro dimensione, l'algoritmo di scheduling della cpu richiesto, il numero massimo di processi da sostenere…

Una volta ottenute, queste informazioni possono essere impiegate in molti modi diversi. In un caso limite, esse possono essere usate per modificare una copia del codice sorgente del SO, che andrebbe poi completamente ricompilato. Dichiarazioni di dati, inizializzazioni e costanti, associate a una compilazione condizionale, produrrebbero una versione in codice macchina del SO specifica per il sistema desiderato.
Per ottenere una versione meno specifica m comunque adatta al sistema desiderato, al descrizione del sistema può essere associata alla creazione di tabelle e alla selezione di moduli da una libreria precompilata. Questi moduli sono collegati per formare il SO richiesto. La selezione permette alla libreria di contenere i driver di tutti i dispositivi di I/O previsti, sebbene solo quelli effettivamente necessari siano inclusi nel SO. Poiché il sistema non viene ricompilato, la sua generazione risulta più rapida, ma il SO ottenuto può prevedere una maggiore generalità di caratteristiche di quante ne siano effettivamente necessarie.
Un altro caso limite è rappresentato dalla costruzione di un sistema completamente controllato mediante tabelle. In questo caso tutto il codice è sempre parte del sistema e la selezione viene effettuata al momento dell'esecuzione del sistema stesso anziché al momento della compilazione o del collegamento. La generazione del sistema implica semplicemente la creazione di tabelle idonee a descrivere il sistema stesso.
Le differenze più rilevanti tra questi approcci riguardano la dimensione e la generalità del sistema generato oltre alla facilità di apporvi delle modifiche in seguito a cambiamenti della configurazione hardware. Dopo che un SO è stato generato, deve essere eseguito dall'hardware. Ma come fa l'hardware a sapere dove si trova il kernel e caricarlo in memoria? La procedura di avvio di un computer attraverso il caricamento del kernel è nota come il booting del sistema: sulla maggior parte dei sistemi c'è un piccolo segmento di codice, memorizzato in rom e noto come bootstrap program o bootstrap loader, che è in grado di localizzare il kernel, caricarlo in memoria e avviarne l'esecuzione. Alcuni sistemi, come i PC, effettuano tale compito in due passi: un bootstrap loader molto semplice preleva un più complesso programma di boot dal disco, che a sua volta carica il kernel.

PARTE SECONDA: GESTIONE DEI PROCESSI

Un processo può essere pensato come un programma in esecuzione. Per svolgere il proprio compito un processo richiede determinate risorse, come tempo di cpu, memoria, file e dispositivi di I/O. Queste risorse vengono allocate al processo al momento della creazione o al momento dell'esecuzione.
Il processo è l'unità di lavoro nella maggior parte dei sistemi. Un sistema di questo tipo è formato da un insieme di processi: processi di SO che eseguono il codice di sistema e Processi utente che eseguono il codice utente. Tutti questi processi possono essere eseguiti potenzialmente in concorrenza.
Il SO è responsabile delle seguenti attività connesse alla gestione dei processi: creazione e cancellazione dei processi di sistema e dei processi utenti; scheduling dei processi, offerta di meccanismi di sincronizzazione, comunicazione, e gestione dei deadlock per i processi.

4 PROCESSI

Maggiore è la complessità di un SO, maggiori sono i servizi che si suppone che esso fornisca ai propri utenti. Benché il suo compito principale sia l'esecuzione dei programmi utenti, esso deve anche occuparsi dei vari task di sistema che è meglio lasciare al di fuori del kernel. Un sistema è quindi costituito da un insieme di processi: processi del SO che eseguono codice di sistema, e processi utente che eseguono codice utente. Tutti questi processi possono essere eseguiti potenzialmente in concorrenza e l'uso della cpu è commutato tra i vari processi. Il SO può rendere il computer più produttivo commutando la cpu tra diversi processi.

4.1 CONCETTO DI PROCESSO

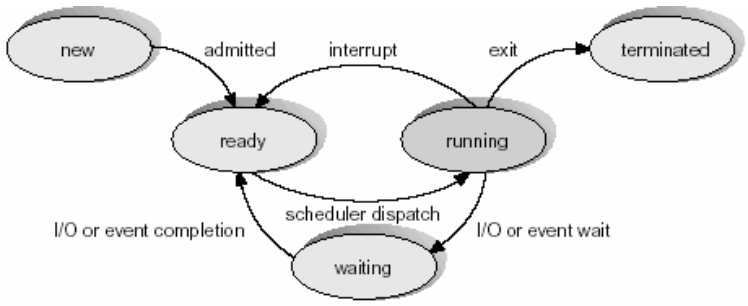
La questione riguardante i nomi da attribuire a tutte le attività della cpu costituisce un ostacolo alla discussione dei SO: un sistema batch esegue un job, mentre sistemi con time–sharing esegue programmi tenti, o task. Persino su un sistema monoutente, come MS-DOS o Macintosh OS, un utente può eseguire parecchi programmi contemporaneamente: un programma interattivo e parecchi programmi batch. Anche se l'utente esegue un solo programma alla volta, il SO deve supportare tutte le sue attività interne, come ad esempio lo spooling. Queste attività sono simili per molti aspetti, per ciò verranno denominate processi (processi = job d'ora in poi).

4.1.1 PROCESSO

Un processo è qualcosa di più di del codice di programma (noto anche come sezione testo). Esso comprende l'attività attuale, rappresentata dal calore del program counter e dal contenuto dei registri del processore. Un processo comprende normalmente anche il proprio stack, che contiene a sua volta i dati temporanei, come parametri di subroutine, indirizzi di rientro e variabili temporanee, e una sezione dati contenete variabili globali. Si sottolinea che un programma di per sé non è un processo; un programma è un'entità passiva, come il contenuto di un file memorizzato su disco, mentre un processo è un entità attiva.

4.1.2 STATO DEL PROCESSO

Mentre un processo è in esecuzione subisce un cambiamento di stato. Ogni processo può trovarsi in uno dei seguenti stati:

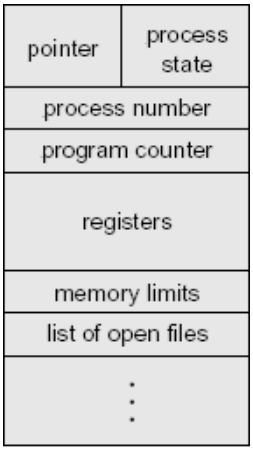


- New il processo viene creato
- Running Le istruzioni vengono eseguite
- Waiting Il processo attende che si verifichi qualche evento

- Ready Il processo attende di essere assegnato a un processore
- Terminated Il processo ha terminato l'esecuzione

Su ciascun processore può essere running un solo processo per volta. Tuttavia, molti processi possono essere ready o waiting.

4.1.3 PROCESS CONTROL BLOCK



Ogni processo è rappresentato nel SO da un process control block (PCB, chiamato anche task control block). Un pcb contiene gran parte delle informazioni connesse a un processo specifico. Tra cui queste:

- Stato del processo (new, ready, running, waiting, halted…)
- Program counter: indica l'indirizzo della prossima istruzione da eseguire.
- Registri di cpu: tutte le informazioni devono essere salvate in caso di interrupt
- Informazioni sullo scheduling della cpu: priorità del processo, puntatori alle code di scheduling e tutti gli altri parametri di scheduling
- Informazioni sulla gestione della memoria: valori dei registri base e limite, le tabelle delle pagine…
- Informazioni di contabilizzazione delle risorse: tempo di cpu, limiti di tempo, i numeri di account…
- Informazioni sello stato di I/O: la lista dei dispositivi allocato a un dato processo, l'elenco dei file aperti…

Il PCB viene utilizzato semplicemente come deposito per tutte le informazioni relative ai vari processi.

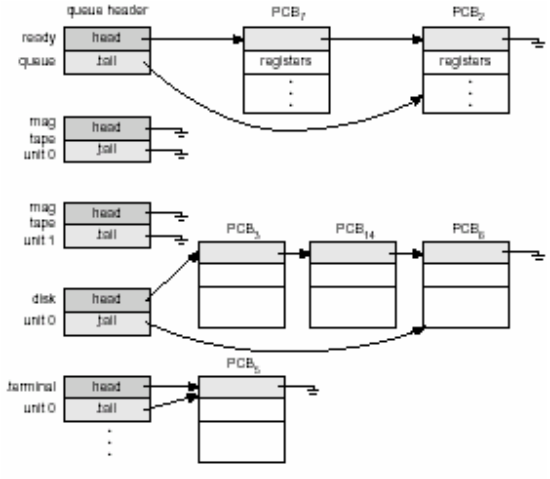
4.2 SCHEDULING DEI PROCESSI

L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'utilizzo della cpu. L'obiettivo del time–sharing è di commutare l'uso della cpu tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre esso è in esecuzione.

4.2.1 CODE DI SCHEDULING

Quando i processi entrano nel sistema vengono inseriti in una coda di processi, formata da tutti i processi del sistema. I processi che risiedono in memoria centrale e che sono pronti e in attesa di essere eseguiti si trovano in una lista detta ready queue. Questa coda è generalmente memorizzata come una lista concatenata. Un'intestazione della ready queue contiene i puntatori al primo e all'ultimo PCB dell'elenco. Ogni PCB ha un campo puntatore che indica il successivo processo contenuto nella ready queue.

L'elenco dei processi in attesa di un particolare dispositivo di I/O è chiamato coda del dispositivo. Una comune rappresentazione è data da un diagramma di accoramento. Ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la ready queue e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, mentre le frecce indicano il flusso di processi nel sistema.



Un nuovo processo viene posto inizialmente nella ready queue. Qui attende finché non viene selezionato per essere eseguito (dispatched); a questo punto gli viene assegnata la cpu. Una volta che gli è stata allocata la cpu ed entra in esecuzione, può verificarsi uno dei seguenti eventi:

- Il processo può emettere una richiesta di I/O e quindi essere posto in una coda di I/O
- Il processo può creare un nuovo processo e attenderne la terminazione
- Il processo può essere rimosso d'autorità dalla cpu a causa di un interrupt, ed essere riportato nella ready queue.

Nei primi due casi, al completamento della richiesta di I/O o al termine del processo figlio il processo passa dallo stato di waiting allo stato ready e viene nuovamente inserito nella ready queue. Un processo continua questo ciclo finché la sua esecuzione termina, che corrisponde al momento in cui viene rimosso da tutte le code e il suo PCB e le varie risorse vengono deallocate.

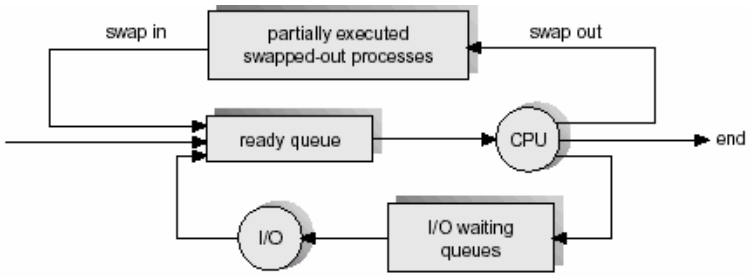
4.2.2 SCHEDULER

Nel corso della sua esistenza, un processo viene a trovarsi in varie code di scheduling. Il SO, che è incaricato di selezionare i processi dalle suddette code, effettua la selezione per mezzo di un opportuno scheduler.

In un sistema batch accade spesso che vengano sottoposti più processi di quanti ne possano essere eseguiti immediatamente. Questi processi vengono sottoposti a spooling su un dispositivo di memoria di massa, generalmente un disco, dove vengono tenuti fino al momento della loro esecuzione. Lo scheduler a lungo termine, o scheduler di job, seleziona i processi da questo insieme e li carica in memoria perché siano eseguiti. Lo scheduler a breve termine, o scheduler di cpu, effettua la selezione tra i processi pronti per essere eseguiti e alloca la cpu ad uno di essi.

Questi due scheduler si differenziano per la frequenza con cui vengono eseguiti: quello a breve termine spesso almeno una volta ogni 100 millisecondi, impiegando quindi il 9% di tempo della cpu. Per lo scheduler a lungo termine può venir eseguito con una frequenza molto minore: tra la creazione di nuovi processi possono trascorrere diversi minuti. Lo scheduler a lungo termine controlla il grado di multiprogrammazione, cioè il numero di processi in memoria. Se il grado di multiprogrammazione è stabile, la velocità media di creazione dei processi deve essere uguale alla velocità con cui i processi abbandonano il

sistema; quindi lo scheduler a lungo termine può essere richiamato solo quando il processo abbandona il sistema. A causa del maggiore intervallo di tempo che intercorre tra le esecuzioni, lo scheduler a lungo termine dispone di più tempo per decidere quale processo debba essere selezionato per l'esecuzione.
In generale, la maggior parte dei processi può essere descritta come I/O bound, o come Cpu bound. Un processo I/O bound impiega più tempo nell'esecuzione di I/O che nell'esecuzione di calcoli. Un processo cpu bound, invece, non genera frequenti richieste di I/O e utilizza più tempo di calcolo di quanto ne utilizzi un processo I/O bound. È importante che lo scheduler a lungo termine selezioni una buona combinazione di processi I/O e cpu bound. Se tutti i processi sono I/O bound, la ready queue è quasi sempre vuota e lo scheduler a breve termine ha poco da fare. Se tutti i processi sono cpu bound, la I/O waiting queue è quasi sempre vuota, i dispositivi non vengono utilizzati, e il sistema risulta nuovamente sbilanciato. Le prestazioni migliori sono date da una combinazione equilibrata di processi I/O bound e processi cpu bound.
Alcuni SO, come i sistemi time–sharing, possono introdurre un livello di scheduling intermedio. L'idea chiave che sta alla base dello scheduler a medio termine consiste nel vantaggio che in alcuni casi può derivare dalla rimozione di alcuni processi dalla memoria (e dalla contesa attiva per la cpu) e quindi dalla riduzione del rado di multiprogrammazione. Il processo può essere reintrodotto in memoria in un momento successivo e la sua esecuzione può ricominciare dal punto in cui era stata interrotta. Questo schema viene chiamato spesso swapping. Le operazioni di swap–in e swap–out vengono eseguite su un processo dallo scheduler a medio termine. Lo swapping può essere necessario per migliorare la combinazione di processi, oppure perché una modifica ai requisiti di memoria ha sovraimpegnato la memoria disponibile e occorre liberarne una parte.



4.2.3 CONTEXT SWITCH

Il passaggio della cpu a un nuovo processo implica il salvataggio dello stato del processo vecchio e il caricamento dello stato salvato del nuovo processo. Questa procedura è conosciuta con il nome di context switch. Il tempo di context switch è puro overhead, infatti il sistema non compie nessun lavoro utile durante la commutazione, ed il tempo varia a seconda del supporto hardware.

4.3 OPERAZIONI SUI PROCESSI

I processi in un sistema possono essere eseguiti concorrentemente, e devono essere creati e cancellati dinamicamente; a tal fine il SO deve offrire un meccanismo che permetta di creare e terminare un processo.

4.3.1 CREAZIONE DI UN PROCESSO

Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita system call. Il processo creante è chiamate anche processo padre, mentre i nuovi processi sono chiamati figli. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un albero di processi.
In generale, per eseguire il proprio compito un processo necessita di alcune risorse. Quando un processo crea un sottoprocesso, quest'ultimo può essere in grado di ottenere le proprie risorse direttamente dal SO, oppure può essere vincolato a un sottoinsieme delle risorse del processo padre. Il padre può avere la necessità di spartire le proprie risorse tra i suoi figli, oppure può essere in grado di condividerne alcune, come memoria o file, tra più figli. Limitando le risorse del padre, è possibile che un processo sovraccarichi il sistema creando troppi sottoprocessi.
Oltre alle risorse fisiche e logiche che un processo può ottenere al momento della sua creazione, il processo padre può passare al processo figlio i dati di inizializzazione (input).
Quando un processo crea un nuovo processo, per quanto riguarda l'esecuzione dei figli ci sono due possibilità:

- Il padre continua l'esecuzione concorrentemente con i propri figli
- Il padre aspetta che alcuni o tutti i suoi figli abbiano terminato

Ci sono due possibilità anche per quanto riguarda lo spazio di indirizzi del nuovo processo:

- Il processo figlio è un duplicato del processo padre
- Nel processo figlio è stato caricato un programma.

Per esempio in Unix ogni processo viene identificato con il proprio identificatore di processo, che è un intero unico. Un nuovo processo viene creato attraverso la system call fork. Il nuovo processo è formato da una copia dello spazio degli indirizzi del processo originale. Questo meccanismo permette al processo padre di comunicare senza difficoltà con il proprio figlio. Il codice di ritorno per la fork è zero per il nuovo processo (figlio) mentre l'identificatore del processo figlio, che è diverso da zero, viene riportato al padre. Generalmente, dopo una fork, uno dei due processi utilizza una system call execve per sostituire lo spazio di memoria del processo con un nuovo programma. La system call execve carica in memoria un file binario, distruggendo l'immagine di memoria del programma contenente la stessa system call execve, quindi avvia la propria esecuzione. In questo modo i due processi possono comunicare e quindi procedere in modo diverso. Il processo padre può anche generare più figli, oppure, se durante l'esecuzione del figlio non ha niente da fare, può invocare una system call wait per rimuovere sé stesso dalla ready queue fino alla terminazione del figlio.

4.3.2 TERMINAZIONE DI UN PROCESSO

Un processo termina assieme all'esecuzione dell'ultima istruzione, e chiede al SO di essere cancellato usando la specifica system call exit, che può restituire dei dati al processo padre. Tutte le risorse vengono deallocate.

Si può avere una terminazione del processo anche in altri casi: un processo può causare la terminazione di un altro processo per mezzo di un'opportuna system call, ad esempio abort. Generalmente un padre può terminare il figlio perché

- Il figlio ha ecceduto nell'uso di alcune risorse che gli sono state allocate
- Il compito assegnato al figlio non è più richiesto
- Il padre termina e il SO non consente a un figlio di continuare la sua esecuzione (terminazione a cascata).

In Unix un processo può terminare per mezzo della system call exit, e il suo processo padre può attendere l'evento utilizzando wait. Quest'ultima fa rientrare l'identificatore di processo del figlio al padre, per cui il padre può stabilire quale dei suoi figli ha terminato.

4.4 PROCESSI COOPERANTI

I processi concorrenti in esecuzione nel SO possono essere indipendenti o cooperanti. Un processo è indipendente se non può influire su altri processi del sistema o anche subirne l'influsso. Chiaramente, un processo che non condivida dati con altri processi è indipendente. D'altra parte un processo è cooperante se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Ovviamente, qualsiasi processo che condivida dati con altri processi è un processo cooperante.
Ci sono diversi motivi per fornire un ambiente che consenta la cooperazione tra processi:

- Condivisione di informazioni
- Accelerazione del calcolo (più subtask in parallelo)
- Modularità (dividere le funzioni in processi distinti)
- Convenienza (lavorare su più task contemporaneamente).

Per illustrare il concetto di cooperazione tra processi, si consideri il problema del produttore-consumatore. Un processo produttore produce informazioni che vengono consumate da un processo consumatore.
Per permettere un'esecuzione concorrente dei processi produttore e consumatore, occorre avere la disponibilità di un buffer di elementi che possano essere riempiti dal produttore e svuotati dal consumatore. Un produttore può produrre un elemento mentre il consumatore ne sta consumando un altro. Il produttore e il consumatore devono essere sincronizzati in modo che il consumatore non tenti di consumare un elemento che non è stato ancora prodotto; in questo caso il consumatore deve attendere la produzione di un elemento.
Il problema dei processi produttore-consumatore con buffer illimitato non pone limiti alla dimensione del buffer. Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrne. Il problema del produttore-consumatore con buffer limitato presuppone l'esistenza di una dimensione fissata del buffer. In questo caso, il consumatore deve attendere se il buffer è vuoto, viceversa, il produttore deve attendere se il buffer è pieno. Il buffer può essere fornito dal SO attraverso l'uso di IPC, oppure codificato esplicitamente, facendo uso di memoria condivisa, dal programmatore dell'applicazione. Di seguito viene illustrata una soluzione, facente uso di memoria condivisa, al problema del buffer limitato. Il produttore ed il consumatore devono avere le seguenti variabili:

```
var n;
type item = ...;
var buffer: array [0..n-1] of item;
```

in, out: 0..n-1;
Dove le variabili in ed out sono inizializzate a 0. Il buffer condiviso è realizzato come un array circolare con due puntatori logici: in ed out. La variabile in indica la successiva posizione libera nel buffer, out la prima posizione piena. Il buffer è vuoto quando in=out; il buffer è vuoto quando in+1 mod n = out.
No-op è un'istruzione di non operatività.
Il processo produttore ha una variabile nextp in cui è memorizzato l'elemento da produrre:

```
repeat
        ...
        produce un elemento in nextp
        ...
        while in+1 mod n = out do no-op;
        buffer[in]:=nextp;
        in:=in+1 mod n;
until false
```

Il processo consumatore ha una variabile nextc in cui viene memorizzato l'elemento da consumare:

```
repeat
        while in=out do no-op;
        nextc:=buffer[out];
        out=out+1 mod n;
        ...
        consuma l'elemento in nextc
        ...
until false
```

Questo schema permette di avere al massimo n-1 elementi contemporaneamente nel buffer.

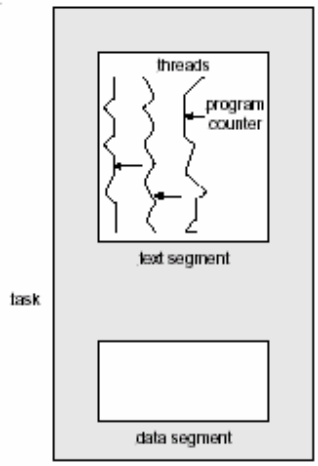
4.5 THREAD

Qualsiasi processo è definito dalle risorse che utilizza e dalla locazione alla quale è in esecuzione. Comunque esistono situazioni nelle quali sarebbe utile ammettere la condivisione di risorse per consentirne poi l'accesso in concorrenza. Questa situazione è simile a quella che si verifica quando si invoca la system call fork specificando un diverso percorso di esecuzione, individuato un nuovo valore nel program counter, da eseguire all'interno del medesimo spazio di indirizzi. L'utilità di questo concetto è dimostrata dal fatto che diversi SO di nuova concezione hanno provveduto a fornire il supporto mediante il meccanismo dei thread.

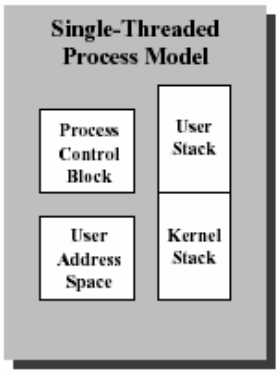
4.5.1 STRUTTURA DEI THREAD

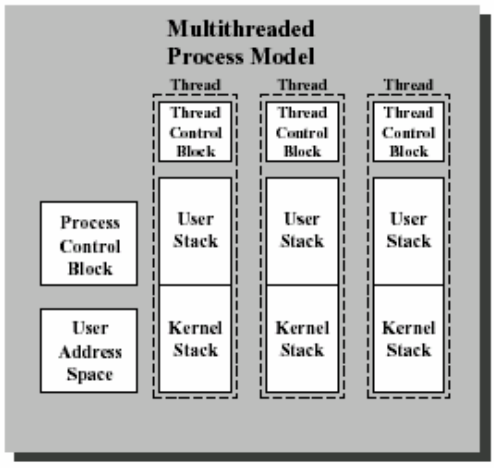
Un thread, chiamato talvolta lightweight process, o LWP, è l'unità base si utilizzo della cpu e consiste di un program counter, un insieme di registri e uno stack. Esso condivide con i thread ad esso associati le sezioni codice e dati, oltre che le risorse fornite dal SO, quali i file aperti e i segnali. L'insieme dei thread e dell'ambiente da essi condiviso prende il nome di task. Un processo tradizionale, o heavyweight, equivale a un task

dotato di un unico thread. Un task privo di thread non esegue nessun compito e ciascun thread deve essere contenuto in esattamente un task. L'alto grado di condivisione fa sì che la creazione dei thread e il passaggio del controllo della cpu da un thread all'altro di uno stesso task siano operazioni meno costose del context switch tra processi tradizionali. Nonostante richieda anch'esso una reimpostazione dei valori contenuti nei registri dei processori, un context switch tra i thread non implica nessuna operazione legata alla gestione della memoria. Come accade in ogni ambiente di elaborazione parallela, l'uso di più thread per un processo può introdurre dei problemi di controllo della concorrenza che richiedono l'uso di sezioni critiche o dei lock. Alcuni sistemi implementano anche un meccanismo di thread a livello utente attraverso librerie a livello utente da utilizzare in sostituzione delle system call; in questo modo lo scambio dei thread non richiede alcuna chiamata al SO che causerebbe un interrupt al kernel.



Si fa un confronto tra la gestione del controllo con thread multipli e il modo di operare con più processi: nel caso di processi multipli, ciascuno di essi opera indipendentemente dagli altri e possiede i propri program counter, stack pointer e spazio degli indirizzi. Questa organizzazione è conveniente quando i processi svolgono job non correlati tra loro. È comunque possibile che dei processi multipli eseguano lo stesso task; ad esempio, fornire i dati alle macchine remote in un file system di rete. È però più efficiente avere un processo contenete più thread per lo stesso scopo. Nella realizzazione con processi multipli, ciascun processo esegue lo stesso codice ma ha le proprie risorse di memoria e i propri file aperti. Un processo con thread multipli impiega minori risorse — memoria, file aperti, scheduling della cpu di più processi ridondanti. Sotto molti punti di vista, i thread rispecchiano il comportamento dei processi. Un thread può trovarsi in uno stato ready, blocked, running o terminated. Comunque a differenza di quanto accade per i processi, i thread non solo tra loro indipendenti; infatti la condivisione dello spazio degli indirizzi all'interno del task consente a un thread di leggere o scrivere nello stack di uno qualsiasi degli altri thread del task. Questa struttura non garantisce alcuna protezione tra i singoli thread di uno stesso task; tale protezione non dovrebbe comunque essere necessaria. Infatti, laddove i singoli processi possono essere generati da utenti diversi, con differenti interessi e talvolta in conflitto tra loro, un task con più thread appartiene a un unico utente. Quindi, i singoli thread saranno





probabilmente progettati per cooperare e non richiederanno perciò la presenza di alcun meccanismo di protezione.
I thread forniscono un meccanismo attraverso il quale i processi sequenziali ottengono del parallelismo pur eseguendo delle system call bloccanti. Per illustrare i vantaggi introdotti da questo meccanismo, si consideri la codifica di un file server privo di thread. In un server a thread singolo il processo server è costretto a portare al termine il servizio richiesto prima di poter prendere in considerazione nuove richieste. Se la richiesta attualmente in esame comporta un attesa per l'accesso su disco, durante l'attesa la cpu resta inattiva. Quindi il numero di richieste al secondo che possono essere elaborate è decisamente minore di quanto si potrebbe ottenere con un'esecuzione parallela. Ciò comporta che un progettista di sistemi, in assenza di thread multipli, per cercare si minimizzare il rallentamento delle prestazioni dovrebbe simulare la struttura parallela dei thread con dei processi heavyweight. La conseguenza più immediata è la necessità di dover ricorrere a una struttura complessa non sequenziale. L'astrazione proposta da un gruppo di processi lightweight è quella di thread multipli di controllo associati a più risorse condivise. Ci sono molte alternative riguardanti i thread, e di seguito ne vengono menzionate alcune. Il supporto dei thread può essere fornito dal kernel (Mach e OS/2). In questo caso il sistema mette a disposizione dell'utente un insieme di system call analoghe a quelle che si usano per la gestione dei processi tradizionali. In alternativa, il supporto può risiedere al di sopra del kernel ed essere fornito da un insieme di funzioni di libreria implementate a livello utente.
Quali sono le motivazioni che potrebbero indurre a seguire una via piuttosto che l'altra? La gestione dei thread a livello utente, non coinvolgendo il kernel, consente di passare da un thread all'altro più velocemente rispetto ai thread supportati dal kernel. D'altro canto, ogni chiamata al SO mette in attesa l'intero processo, poiché il kernel gestisce solo processi non contemplando la gestione dei thread) e un processo in attesa non ottiene tempo di cpu. Inoltre una scelta in questo senso potrebbe avere delle conseguenze anche sull'imparzialità dello scheduling dei thread. Si considerino due processi, uno composto da un unico thread (processo a) e l'altro con 100 thread (processo b). in generale, ciascuno dei due processi ottiene lo stesso numero di quanti di tempo, per cui l'unico thread a avrà prestazioni 100 volte superiori rispetto i singoli thread di b. Nei sistemi in cui il supporto ai thread viene fornito dal kernel, il passaggio da un thread all'altro è sicuramente più costoso in termini di tempo, visto che sarà lo stesso kernel a gestirlo (via interrupt). Lo scheduling di ciascun thread potrà, comunque, essere eseguito indipendentemente dagli altri, per cui il processo b potrebbe ricevere un tempo di utilizzo della cpu 100 volte superiore a quello ricevuto dal processo a. Inoltre il processo b potrebbe eseguire 100 system call concorrentemente, raggiungendo prestazioni molto più elevate di quelle conseguibili dallo stesso processo in esecuzione su un sistema che supporta solo thread a livello utente.
Nel tentativo di eliminare i problemi connessi alle implementazioni presentate, alcuni sistemi preferiscono seguire un approccio ibrido che implementa entrambe le versioni dei thread: a livello utente e al livello kernel (Solaris 2).
I thread devono la loro crescente popolarità al fatto di possedere alcune delle caratteristiche proprie dei processi heavyweight, pur essendo eseguiti più efficientemente. Questa combinazione risulta efficace in molte applicazioni. Ad esempio in alcune versioni di Unix il kernel è composto da un singolo task: ciò significa che a un solo

processo alla volta è concesso di eseguire codice contenuto nel kernel. Questo meccanismo consente di evitare diversi problemi: ad esempio, non è necessario sincronizzare l'accesso ai dati, poiché un solo processo alla volta potrà operare su di essi. D'altra parte, grazie alla sua gestione multithread, Mach consente al kernel di servire più richieste contemporaneamente. In questo caso sono gli stessi thread a essere sincroni: un altro thread appartenente allo stesso gruppo può essere eseguito solo se il thread attualmente in esecuzione cede il controllo del processore. Naturalmente, questo non accadrà durante la fase di modifica di dati condivisi. Nei sistemi che implementano thread asincroni è necessario ricorrere a qualche meccanismo di lock esplicito come nei sistemi in cui i dati sono condivisi da più processi.

4.6 COMUNICAZIONE TRA PROCESSI

Attraverso le funzionalità dell'IPC i processi sono in grado di comunicare tra loro e di sincronizzare le proprie azioni. La migliore implementazione dell'IPC è quella basata sullo scambio di messaggi. Tali sistemi possono essere definiti in diversi modi: sistemi di comunicazione a memoria condivisa e quelli a scambio di messaggi non sono mutualmente esclusivi, per cui possono essere adoperati simultaneamente non solo nello stesso SO, ma anche da un singolo processo.

4.6.1 STRUTTURA DI BASE

Un sistema di messaggi permette ai processi di comunicare tra loro senza ricorrere a variabili condivise. Un IPC fornisce almeno le due operazioni send(message) e receive(message).
I messaggi inviati da un processo possono avere dimensione fissa o variable.
Se i processi P e Q vogliono comunicare, devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un canale di comunicazione, che può essere implementato in molti modi.
Per effettuare l'implementazione è necessario fondamentalmente sapere:

- Come vengono stabiliti i canali
- Se un canale può essere associato a più processi
- Quanti canali possono esistere tra ogni coppia di processi.
- Che cosa si intende per capacità di un canale, cioè se il canale dispone di uno spazio buffer, e in caso ne disponga occorre conoscere la dimensione di questo spazio
- Che cosa si intende per dimensione dei messaggi. Occorre sapere se il canale può supportare messaggi con dimensione variabile o messaggi con dimensione fissa.
- Se un canale è unidirezionale o bidirezionele.

Il termine unidirezionale significa che un processo può solo inviare o solo ricevere.
Inoltre esistono diversi modi per effettuare l'implementazione logica di un canale e delle operazioni send/receive:

- Comunicazione diretta o indiretta.
- Comunicazione simmetrica o asimmetrica.
- Buffering automatico o esplicito.

- Invio per copia o per riferimento
- Messaggi con dimensione fissa o variabile.

4.6.2 NOMINAZIONE

I processi che vogliono comunicare devono disporre di un modo con cui riferirsi agli altri processi; a tale scopo è possibile utilizzare una comunicazione diretta oppure una comunicazione indiretta

4.6.2.1 COMUNICAZIONE DIRETTA

Nello schema di comunicazione diretta ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione. In questo schema le primitive di send e receive sono:
send (P message) --> Invia message al processo P
receive (Q, message) --> Riceve il messaggio dal processo Q
All'interno di questo schema, un canale di comunicazione ha le seguenti caratteristiche:

- Tra ogni coppia di processi che intendono comunicare viene stabilito automaticamente un canale. Per comunicare i processi devono conoscere la reciproca identità.
- Un canale è associato esattamente a due processi
- Tra ogni coppia di processi comunicanti esiste esattamente un canale.
- Il canale può essere unidirezionale, ma usualmente è bidirezionale.

4.6.2.2 COMUNICAZIONE INDIRETTA

Con la comunicazione indiretta i messaggi vengono inviati a dei mailbox (chiamati anche porte) e da essi ricevuti. Un mailbox può essere visualizzato astrattamente come un oggetto nel quale i processi possono introdurre messaggi o dal quale possono prelevarne. Ogni mailbox è identificato inequivocabilmente. In questo schema un processo può comunicare con un altro processo solo se hanno una mailbox in comune. Le primitive send e receive sono definite come segue:
send (A, message) --> Inia message al mailbox A
receive (A, message) --> Riceve, in message, un messaggio dal mailbox A.
In questo schema un canale di comunicazione ha le seguenti proprietà:

- Tra una coppia di processi viene stabilito un canale solo se i processi hanno una mailbox in comune
- Un canale può essere associato a più di due processi
- Tra ogni coppia di processi comunicanti può esserci un certo numero di canali diversi, ciascuno corrispondente ad una mailbox.
- Un canale può essere unidirezionale o bidirezionale.

A questo punto si supponga che i processi P1, P2 e P3 condividano il mailbox A. Il processo P1 invia un messaggio ad A, mentre sia P2 che P3 eseguono una receive da A. Sorge il problema di sapere quale processo riceverà il messaggio. Tale problema può essere affrontato adottando una delle seguenti soluzioni:

- E' possibile fare in modo che un canale sia associato solo a due processi.
- E' possibile consentire ad un solo processo alla volta di eseguire un'operazione receive.
- E' possibile consentire al sistema di decidere arbitrariamente quale processo riceverà il messaggio (ma uno solo dei due).

Un mailbox può appartenere al processo o al sistema. Nel caso appartenga a un processo, vale a dire che il mailbox è associato al processo o definito come parte di esso, occorre distinguere ulteriormente un proprietario, che può soltanto ricevere messaggi tramite questo mailbox, e utente, che può solo inviare messaggi a questo mailbox. Dal momento che ogni mailbox ha un solo proprietario, non può sorgere confusione su chi debba ricevere un messaggio inviato a un dato mailbox.
Esistono parecchi modi per designare il proprietario e gli utenti di uno specifico mailbox. Una possibilità consiste nel permettere a un processo di dichiarare variabili di tipo mailbox. Il mailbox appartiene al processo che lo dichiara, e può essere utilizzato da qualsiasi altro processo che ne conosca il nome.
D'altra parte, un mailbox posseduto dal SO ha una sua esistenza propria: è indipendente e non è legato a nessun processo in particolare. Il SO offre un meccanismo che permette a un processo le seguenti operazioni:

- Creare un nuovo mailbox
- Inviare e ricevere messaggi tramite il mailbox
- Distruggere un mailbox

Il processo che crea il mailbox è il proprietario. Il diritto di proprietà e il privilegio di ricezione possono essere passati ad altri processi tramite idonee system call. I processi possono inoltre condividere mailbox.

4.6.3 BUFFERING

Un canale ha una capacità che determina il numero di messaggi che possono risiedere temporaneamente al suo interno. Questa caratteistica può essere immaginata come una coda di messaggi legata al canale. Fondamentalmente esistono tre modi per implementare questa coda:

- Capacità zero. La coda ha lunghezza massima 0, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve attendere che il ricevente abbia ricevuto il messaggio. affinché il trasferimento dei messaggi abbia luogo, i due processi devono essere sincronizzati. Questo tipo di sincronizzazione è chiamata rendezvous.
- Capacità limitata. La coda ha lunghezza massima finita n, quindi al suo interno possono risiedere al massimo n messaggi. Se la coda non è piena quando viene inviato un messaggio, quest'ultimo viene posto in fondo alla coda, il messaggio viene copiato oppure viene tenuto un puntatore a quel messaggio. Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Ma il canale ha comunque una capacità limitata. Se il canale è pieno, il trasmittente deve attendere che nella coda ci sia spazio disponibile.
- Capacità non limitata. La coda ha potenzialmente lunghezza infinita, quindi al suo interno può essere in attesa un numero indeterminato di messaggi. Il trasmittente non resta mai in attesa.

Il caso della capacità zero viene anche definito come sistema di messaggi senza buffering. I processi non sono in grado di sapere se dopo una send il messaggio sia arrivato a destinazione. Se questa informazione è indispensabile ai fini del calcolo, il trasmittente deve comunicare esplicitamente con il ricevente per sapere se ques't'ultimo abbia ricevuto il messaggio. Si supponga, ad esempio, che il processo P invii un messaggio al processo Q e possa continuare la propria esecuzione solo dopo che il messaggio sia stato ricevuto. Il processo P esegue questa sequenza:
send (Q, message);
receive (Q, message);
Il processo Q esegue:
receive (P, message);
send (P, "acknowledge");
Processi di questo tipo comunicano asincronicamente.
Esistono casi particolari:

- Il processo trasmittente non resta mai in attesa, ma se invia un altro messaggio prima che il ricevente abbia ricevuto il primo, quest'ultimo va perduto. Serve quindi la sincronizzazione dei processi
- Il processo che invia un messaggio viene ritardato fino a che non riceve una risposta. Questo metodo di comunicazione sincrono può essere facilmente ampliato in un ben delineato sistema di chiamata di procedura remota (RPC). Un sistema RPC è basato sul presupposto che una chiamata di subroutine o procedura in un sistema con processore singolo agisca esattamente come un sistema di messaggi nel quale il trasmittente si blocca finché non ha ricevuto risposta. Il messaggio allora è simile a una chiamata ad una subroutine, e il messaggio di risposta contiene il valore calcolato dalla subroutine. Perciò il successivo passo logico consiste nella possibilità, per i processi concorrenti, di chiamarsi a vicenda delle subroutine utilizzando le RPC.

4.6.4 CONDIZIONI DI ECCEZIONE

Un sistema di messaggi risulta utile soprattutto in un ambiente distribuito, dove i processi possono risiedere su siti diversi. La probabilità di un errore è quindi alta. In presenza di un guasto è necessario effettuare un ripristino dall'errore (gestione della condizione di errore).

4.6.4.1 TERMINAZIONE DI UN PROCESSO

Un processo trasmittente o ricevente può terminare prima che un messaggio sia stato elaborato. Questa situazione comporta la presenza di messaggi che non vengono mai ricevuti o di processi che restano in attesa di messaggi che non verranno mai inviati. Possono verificarsi i due seguenti casi:

1) Un processo ricevente P attende un messaggio dal processo Q che ha terminato. Se non viene intrapresa alcuna azione, P rimane bloccato per sempre. In questo caso, però, il sistema può terminare l'esecuzione di P, oppure informare P che Q ha terminato.

2) Il processo P invia un messaggio al processo Q che ha terminato. Nello schema con il buffering automatico non viene prodotto alcun danno: P continua semplicemente la propria esecuzione. Per sapere se il messaggio sia stato elaborato da Q, P deve essere esplicitamente programmato per la ricezione di acknowledge. Nel caso di buffering, invece, P rimane bloccato per sempre. Allora si ritorna al caso 1.

4.6.4.2 MESSAGGI PERDUTI

Un messaggio dal processo P al processo Q può andare perduti in qualche punto della rete di comunicazione a causa di un guasto all'hardware o alla linea di comunicazione. Per trattare questo tipo di eventi è possibile ricorrere a uno dei seguenti tre metodi fondamentali.

1) Il SO è responsabile del rilevamento dell'evento in questione e del nuovo invio del messaggio.
2) Il processo trasmittente è responsabile del rilevamento dell'evento in questione e dell'eventuale nuova trasmissione del messaggio.
3) Il SO è responsabile del rilevamento dell'evento in questione, quindi informa il processo trasmittente può procedere come preferisce.

Non sempre è necessario rilevare i messaggi perduti. Infatti, alcuni protocolli di rete specificano che i messaggi sono inaffidabili, mentre altri garantiscono l'affidabilità. L'utente deve specificare che il rilevamento deve essere effettuato, cioè informare il sistema, oppure programmando tale requisito di affidabilità. Per rilevare che un messaggio è andato perduto, il metodo di rilevamento più comune è basato sull'utilizzo di timeout. Quando viene inviato un messaggio, viene sempre spedito indietro un messaggio di acknowledgement che notifica il ricevimento del primo. Se il periodo di tempo trascorre prima che arrivi il messaggio di acknowledgement , il SO o il processo presumono che il messaggio sia andato perduto e provvedono a inviarlo di nuovo. E' possibile che un messaggio non sia andato perduto, ma che impieghi semplicemente più tempo del previsto ad attraversare la rete. In questo caso sono in circolazione più copie dello stesso messaggio, quindi ci deve essere un meccanismo che distingue tra i vari tipi di messaggio.

4.6.4.3 MESSAGGI ALTERATI

Il messaggio può essere consegnato alla propria destinazione, ma durante il percorso può subire delle interferenze, ad esempio a causa di disturbi sul canale di comunicazione. Questo caso è simile a quello del messaggio perduto; di solito il SO ritrasmette il messaggio originale. Generalmente per individuare errori di questo tipo vengono impiegati codici di controllo (Somme di controllo, bit di parità o CRC).

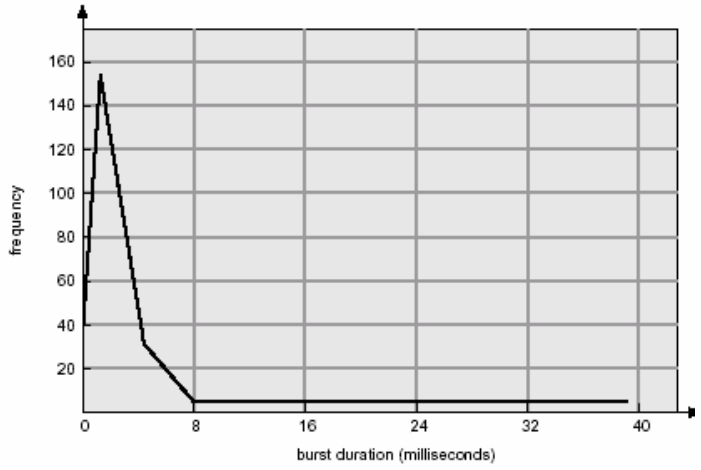
5. SCHEDULING DELLA CPU

5.1 CONCETTI DI BASE

L'obiettivo della multiprogrammazione è di avere sempre dei processi in esecuzione al fine di massimizzare l'utilizzo della cpu. In un sistema con un solo processore potrà essere eseguito al più un processo alla volta; gli altri processi dovranno attendere che la cpu sia libera e possa essere nuovamente sottopost a scheduling.
L'idea della multiprogrammazione è relativamente semplice. Un processo è in esecuzione finché non debba attendere un evento, generalmente il completamento di qualche richiesta di I/O; in un sistema di calcolo semplice la cpu resterebbe inattiva e tutto il tempo di attesa sarebbe sprecato, nel senso che non verrebbe utilizzato per nessun lavoro utile. Con la multiprogrammazione si cerca di utilizzare i tempi di attesa in un modo protettivo: più processi vengono tenuti contemporaneamente in memoria, e quando un processo deve attendere, il SO gli sottrae il controllo della CPU per cederlo a un altro processo.

5.1.1 CICLO DI CPU–I/O BURST

Il successo dello scheduling della cpu dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo è costituita da un ciclo di esecuzione di cpu e di attesa di I/O. I processi si alternano tra questi due stati. L'esecuzione di un processo inizia con un cpu burst, seguito da un I/O burst, che a sua volta è seguito da un altro cpu burst… Infine l'ultimo cpu burst si conclude con una richiesta a sistema di terminare l'esecuzione.
Le durate di questi cpu burst sono state misurate. Sebbene variano molto da processo a processo e da computer a computer, la loro curva di frequenza è generalmente di tipo esponenziale o iperesponenziale. Il numero dei cpu burst brevi è molto alto, mentre è basso il numero di cpu burst molto lunghi.



5.1.2 Scheduler della cpu
Ogniqualvolta la cpu passa in stato di inattività, il So seleziona per l'esecuzione dei processi presenti nella ready queue. La selezione dei processi viene effettuata dallo scheduler a breve termine, o scheduler di cpu. Lo scheduler opera una selezione tra i processi in memoria pronti per l'esecuzione e alloca la cpu a uno di essi.

5.1.3 SCHEDULING CON PRELAZIONE

Le decisioni riguardanti lo scheduling di cpu possono essere prese nelle seguenti circostanze:

- Quando un processo passa dallo stato running a stato waiting (ad esempio, richiesta di I/O o richiesta di attesa per la terminazione di uno dei processi figli)
- Quando un processo passa da stato running a stato ready (ad esempio, quando si presenta un interrupt)
- Quando un processo passa da stato waiting a stato ready (ad esempio, al completamento di I/O)
- Quando un processo termina

Nei casi 1 e 4 non ha luogo alcuna scelta in termini di scheduling. Un processo nuovo, ammesso che ne esista uno nella ready queue, deve essere selezionato per l'esecuzione. Una scelta va invece effettuata nei casi 2 e 3.
Quando si ha scheduling solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è non preemptive (senza prelazione); altrimenti, lo schema di scheduling è preemptive (con prelazione). In caso di scheduling non preemptive, quando la cpu viene allocata a un processo, questo rimane in possesso della cpu fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio allo stato waiting. Questo metodo di scheduling, utilizzando nell'ambiente Windows della Microsoft, è l'unico che può essere usato su certe piattaforme hardware perché non richiede speciali caratteristiche, come la presenza di timer, necessarie allo scheduling preemptive.
Lo scheduling preemptive presenta un inconveniente. Si consideri il caso in cui due processi condividono dei dati e uno di questi, mentre aggiorna i dati, viene prelazionato per l'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato inconsistente. Sono quindi necessari nuovi meccanismi per coordinare l'accesso ai dati condivisi.
La prelazione ha delle ripercussioni anche sul progetto del kernel del SO. Durante l'elaborazione di una system call, il kernel può essere impegnato in attività in favore di un processo, tali attività possono comportare delle modifiche a importanti dati del kernel, come le code di I/O. Se il processo viene prelazionato nel mezzo di tali modifiche e il kernel deve leggere o modificare gli stessi dati, si avrà il caos. Alcuni attendono o il completamento della system call o che abbia luogo il blocco dell'I/O prima di eseguire un context switch. Quindi, il kernel non può prelazionare un processo mentre le strutture dati del kernel si trovano in uno stato inconsistente. Tale schema assicura una semplice struttura del kernel. Sfortunatamente, questo modello di esecuzione del kernel non è adeguato al supporto delle computazioni real–time e del multiprocessing.
Per quanto riguarda Unix, Sono ancora presenti delle sezioni di codice a rischio. Poiché gli interrupt possono, per definizione, verificarsi in ogni istante e non possono essere sempre ignorati dal kernel, le sezioni di codice eseguite per effetto degli interrupt devono essere protette da un uso simultaneo. Il SO deve accettare gli interrupt quasi sempre, altrimenti potrebbero essere persi degli input, o degli output potrebbero essere soprascritti. Perché tali sezioni di codice non siano accessibili concorrentemente da più processi, esse disabilitano gli interrupt al loro inizio e li riabilitano alla fine.

5.1.4 DISPATCHER

Un altro elemento implicato nella funzione di scheduling della cpu è il dispatcher. Questo è il modulo che passa effettivamente il controllo della cpu ai processi selezionati dallo scheduler a breve termine. Questa funzione riguarda quanto segue:

- Il context switch
- Il passaggio al modo utente
- Il salto alla posizione giusta del programma utente per riavviarne l'esecuzione.

Dato che viene richiamato a ogni context switch, il dispatcher dovrebbe essere dotato della massima velocità. Il tempo richiesto dal dispatcher per fermare un processo e avviarne l'esecuzione di un altro è nota come latenza di dispatch.

5.2 CRITERI DI SCHEDULING

Diversi algoritmi di scheduling della cpu hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una situazione particolare, occorre considerare le caratteristiche dei diversi algoritmi. I maggiori criteri sono:

- Utilizzo di cpu: la cpu deve essere più attiva possibile (40% – poco carico; 90% – sistema con utilizzo intensivo)
- Throughput. È una misura del lavoro della cpu: numero di processi completati per unità di tempo.
- Tempo di turnaround. Considerando un processo particolare, un criterio importante può essere relativo al tempo necessario per eseguire il processo stesso. L'intervallo di tempo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato tempo di turnaround, ed è la somma dei tempi passati in attesa di entrare in memoria, in attesa nella ready queue, in esecuzione sulla cpu ed effettuando operazioni di I/O.
- Tempo di attesa. L'algoritmo di scheduling della cpu non influisce sulla quantità di tempo impiegata per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sulla durata del tempo in attesa nella ready queue. Il tempo di attesa è la somma degli intervalli di tempo passati in attesa nella ready queue.
- Tempo di risposta. In un sistema interattivo il tempo di turnaround può non costituire il miglior criterio di valutazione. Accade spesso che un processo produca un output abbastanza presto e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase di emissione. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta.

Utilizzo e throughput di cpu devono essere massimizzati, mentre il tempo di turnaround, di attesa e tempo di risposta devono essere minimizzati. Nella maggior parte dei casi vengono ottimizzati i valori medi; ci sono, comunque, delle circostanze nelle quali è conveniente ottimizzare i valori minimi o massimi, anziché i valori medi. Ad esempio, per garantire che tutti gli utenti ottengano un buon servizio, può essere desiderabile minimizzare il massimo tempo di risposta.
Per i sistemi interattivi, come i sistemi time–sharing, è più importante minimizzare la varianza del tempo di risposta piuttosto che minimizzare il tempo medio di risposta. Un sistema in cui il tempo di risposta sia ragionevole e prevedibile può essere considerato migliore di un sistema mediamente più veloce, ma molto variabile.

5.3 ALGORITMI DI SCHEDULING

5.3.1 SCHEDULING FIRST COME, FIRST SERVED (FCFS)

Il più semplice algoritmo di scheduling di cpu è l'algoritmo di scheduling first come, first served (FCFS). Con questo schema la cpu viene allocata al processo che la richiede per primo. L'implementazione della politica FCFS è gestita semplicemente con una coda FIFO. Quando un processo entra nella ready queue, il suo PCB viene collegato all'ultimo elemento della coda. Quando la cpu non è libera, viene allocata al processo che si trova in testa alla ready queue. Il tempo medio di attesa per l'algoritmo FCFS è spesso abbastanza lungo. Si consideri il seguente insieme di processi:

Processo	Tempo di burst
P_1	24
P_2	3
P_3	3



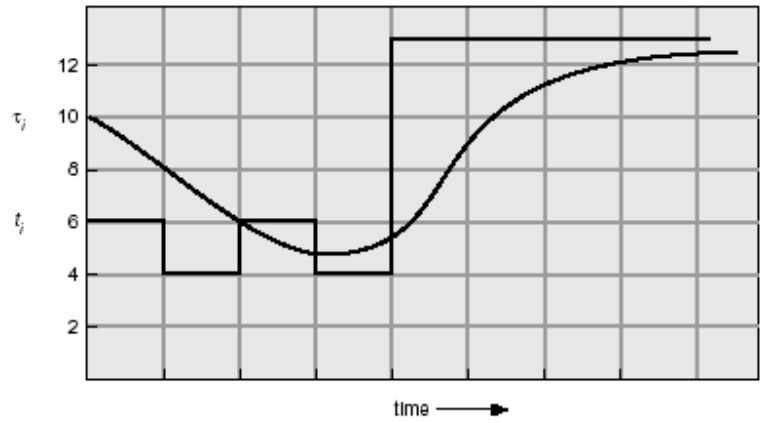
Se i processi arrivano nell'ordine 1 2 3 e sono serviti in ordine FCFS si ottiene che il tempo medio di attesa è (0+24+27)/3=17 millisecondi. Se i processi arrivano nell'ordine 2 3 1 il tempo medio è ora di (0+3+6)/3 = 3 millisecondi. Si tratta di una notevole riduzione. Quindi il tempo medio di attesa in condizioni di FCFS non è in genere minimo, e può varare sostanzialmente al variare dei tempi di burst dei processi. Si ha inoltre un effetto convoglio, poiché tutti i processi attendono che un grande processo liberi la cpu. L'algoritmo FCFS è senza prelazione. Una volta che la cpu è stata allocata a un processo, quel processo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di I/O. L'algoritmo FCFS risulta particolarmente problematico nei sistemi con time–sharing, dove è importante che ogni utente disponga di una parte della cpu a intervalli regolari.

5.3.2 SCHEDULING SHORTEST JOB FIRST (SJF)

Un approccio diverso allo scheduling della cpu può essere ottenuto con l'algoritmo shortest job first (SJF). Questo algoritmo associa a ogni processo la lunghezza del suo cpu burst successivo. Quando la cpu è disponibile, viene assegnata al processo che ha il cpu burst successivo più breve. Se due processi hanno i cpu burst della stessa lunghezza, viene usato l'algoritmo FCFS. Sarebbe più appropriato il termine Shortest next cpu burst, infatti lo scheduling viene eseguito esaminando la lunghezza del successivo burst di cpu del processo piuttosto che la sua totale lunghezza.
Esempio

Processo	Tempo di burst
P_2	8
P_3	7
P_4	3
P_1	6

Quindi il tempo di attesa medio è (0+3+9+16)/4 millisecondi. La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la lunghezza della successiva richiesta di cpu. Sebbene l'algoritmo SJF sia ottimale, non può essere implementato al livello di scheduling della cpu a breve termine, poiché non esiste alcun modo per conoscere la lunghezza del cpu burst successivo. Un possibile approccio consiste nel provare ad approssimare lo scheduling SJF: se non è possibile conoscere la lunghezza del prossimo cpu burst, è comunque possibile predire il suo valore. È probabile, infatti, che il cpu burst successivo abbia lunghezza simile a quelli precedenti; quindi, calcolando il valore approssimato della lunghezza del cpu burst successivo, è possibile scegliere il processo con il cpu burst previsto più breve.
Il cpu burst successivo è generalmente ottenuto effettuando la media esponenziale delle lunghezze misurate dei cpu burst precedenti. Denominando con t_n la lunghezza dell'n–esimo cpu burst e con τ_{n+1} il valore previsto per il cpu burst successivo, per α tale che $0 \leq \alpha \leq 1$, si definisce la media esponenziale



$$\tau_{n+1} = \alpha t_n + (1- \alpha) \tau_n$$

Il valore di t_n contienile informazioni più recenti; τ_n memorizza la sua storia passata. Il parametro α controlla il peso relativo della storia recente e di quella passata sulla predizione. Se $\alpha=0$, allora $\tau_{n+1} = \tau_n$, e la storia recente non ha effetto; viene supposto, cioè, che le condizioni attuali siano transitorie; se $\alpha=1$, allora $\tau_{n+1} = t_n$, e ha significato solo il cpu burst più recente: viene supposto, cioè, che la storia sia vecchia e irrilevante. Più comune è la condizione per cui $\alpha = ½$. Il τ_0 iniziale può essere definito come una costante o come media globale del sistema.

$$\tau_{n+1} = \alpha t_n + (1- \alpha)\tau_{n-1} + \ldots + (1- \alpha)^j \tau_{n-j} + \ldots + (1- \alpha)^{n+1} \tau_0$$

Dal momento che sia α che $1-\alpha$ sono minori o uguali a 1, ogni termine ha peso inferiore a quello del suo predecessore.
L'algoritmo SJF può essere sia preemptive che non preemptive. La scelta si presenta quando alla ready queue arriva un nuovo processo mentre è in esecuzione un processo precedente. Il nuovo processo può avere un cpu burst successivo più corto rispetto a quanto è rimasto sul processo attualmente in esecuzione. Un algoritmo SJF preemptive rimpiazza il processo attualmente in esecuzione, mentre un algoritmo non preemptive no. Lo scheduling SJF preemptive viene talvolta chiamato scheduling shortest-remaining-time-first.
Esempio

Processo	Tempo di arrivo	Tempo di burst
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Il tempo medio di attesa è quindi ((10-1)+(1-1)+(17-2)+(5-3))/4=6,5 millisecondi

5.3.3 SCHEDULING CON PRIORITÀ

L'algoritmo SJF è un caso particolare del più generale algoritmo di scheduling con priorità. A ogni processo viene associata una priorità e la cpu viene allocata al processo con priorità più alta. Processi con uguale priorità vengono ordinati seguendo uno schema FCFS.
Un algoritmo SJF è semplicemente un algoritmo con priorità dove la priorità è l'inverso del cpu burst successivo.
Esempio

Processo	Priorità	Tempo di burst
P_1	3	10
P_2	1	1
P_3	3	2
P_4	4	1
P_5	2	5

Il tempo di attesa medio è di 8,2 millisecondi. Le priorità possono essere definite sia internamente che esternamente. Le priorità definite internamente utilizzano una o più quantità misurabili per calcolare la priorità del processo. Ad esempio, nel calcolo delle priorità possono essere utilizzati i limiti di tempo, i requisiti di memori, il numero di file aperti… Le priorità esterne sono definite in base a criteri esterni del SO, come l'importanza del processo, e altri fattori, spesso di ordine politico…
Lo scheduling con priorità può essere sia preemptive che non preemptive. Quando un processo arriva alla ready queue, la sua priorità viene confrontata con la priorità del processo attualmente in esecuzione. L'algoritmo preemptive prelaziona la cpu se la priorità dell'ultimo processo arrivato è maggiore del processo attualmente in esecuzione. L'algoritmo non preemptive si limiterà a porre l'ultimo processo arrivato in testa alla ready queue.
Un problema importante relativo agli algoritmi di scheduling con priorità e il blocco indefinito o starvation. Un processo pronto per l'esecuzione, ma che non dispone della cpu, può essere considerato bloccato, in attesa della cpu. Un algoritmo di scheduling con priorità può lasciare processi con bassa priorità in attesa indefinita della cpu. Un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla cpu. Una soluzione al problema della starvation di processi con bassa priorità è costituita dall'invecchiamento (aging). Per invecchiamento si intende una

tecnica di aumento graduale della priorità dei processi che si trovano in attesa nel sistema da parecchio tempo.

5.3.4 SCHEDULING ROUND–ROBIN

L'algoritmo di scheduling round–robin (RR) è stato progettato appositamente per I sistemi time–sharing; è simile allo scheduling FCFS ma ha in più la prelazione per effettuare la commutazione dei processi. Viene fissata una piccola unità di tempo, chiamata quanto di tempo o time slice, che varia generalmente da 10 a 100 millisecondi. La ready queue viene trattata come una coda circolare. Los scheduler della cpu scorre la ready queue, allocando la cpu a ciascun processo per un intervallo di tempo della durata massima di 1 quanto di tempo.
Per implementare lo scheduling RR la ready queue viene trattata come una coda FIFO. Possono verificarsi una delle seguenti cose: il processo ha un CPU burst inferiore a 1 quanto di tempo, nel qual caso il processo stesso rilascia volontariamente la cpu, e lo scheduler passa al processo successivo della ready queue. Una situazione diversa si presenta quando il cpu burst del processo attualmente in esecuzione è più lungo di un quanto di tempo. In questo caso il timer si azzera e genera un interrupt al SO, viene eseguito un context switch e il processo viene accodato alla ready queue. Il tempo di attesa medio per politiche RR è abbastanza lungo.
Esempio

Processo	Tempo di burst
P_1	24
P_2	3
P_3	3

Il tempo di attesa medio è 17/3 = 5.66 millisecondi.
Se il cpu burst di un processo eccede il quanto di tempo, quel processo viene prelazionato e riportato nella ready queue. L'algoritmo RR è quindi con prelazione.
Se nella ready queue esistono n processi e il quanto di tempo è q, ciascun processo ottiene 1\n del tempo di cpu in frazioni di, al più, q unità di tempo, allora ogni processo non deve attendere per più di (n-1)*q unità di tempo.
Le prestazioni dell'algoritmo RR dipendono molto dalla dimensione del quanto di tempo. Nel caso limite, quando il quanto di tempo è molto ampio (infinito), la politica RR è analoga alla politica FCFS. Se il quanto di tempo è molto piccolo, l'approccio RR viene chiamato processor sharing e teoricamente gli utenti hanno l'impressione che ciascuno degli n processi abbia a sua disposizione un processore di 1\n-esimo della velocità del processore reale.
Tuttavia nel software occorre anche considerare anche l'effetto del context switch sulle prestazioni dello scheduling RR.
Anche il tempo di turnaround dipende dalla dimensione del quanto di tempo. In generale, il tempo di turnaround medio può essere migliorato se la maggior parte dei processi termina il cpu burst successivo in un unico quanto di tempo. Però aggiungendo anche il tempo di context switch, con un piccolo quanto di tempo il tempo medio di turnaround

aumenta poiché sono richiesti più context switch. Empiricamente si può stabilire come l'80% dei cpu burst debbano essere più brevi del quanto di tempo.

5.3.5 SCHEDULING CON CODE MULTIPLE

È stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi possano essere facilmente classificati in gruppi diversi. Ad esempio, una suddivisione diffusa è quella che viene effettuata tra processi in foreground e processi in background. Questi due tipi di processi hanno tempi di risposta diversi e possono quindi avere anche necessità di scheduling diverse. Inoltre, i processi in foreground possono avere priorità, definita esternamente, sui processi in background.
L'algoritmo di scheduling a code multiple suddivide la ready queue in code separate: i processi sono assegnati permanentemente a una coda, generalmente in base a qualche caratteristica del processo, come dimensione della memoria, priorità o tipo di processo. Ogni coda ha il proprio algoritmo di scheduling. In questa situazione è inoltre necessario avere uno scheduling tra code. Si tratta comunque in genere di scheduling a priorità fissa con prelazione. Esiste anche la possibilità di impostare quanti di tempo tra le code.

5.3.6 SCHEDULING DI CODE MULTIPLE CON FEEDBACK

Lo scheduling di code multiple con feedback permette a un processo di spostarsi tra le code. L'idea consiste nel separare processi con diverse caratteristiche di cpu burst. Se un processo utilizza troppo tempo di cpu, viene spostato in una coda con priorità più bassa. Questo schema mantiene i processi I/O bound e i processi interattivi nelle code con priorità più alta. Analogamente, un processo che attende troppo a lungo in una coda con priorità più bassa può essere spostato in una coda con priorità più alta. Questo impedisce il verificarsi di starvation.

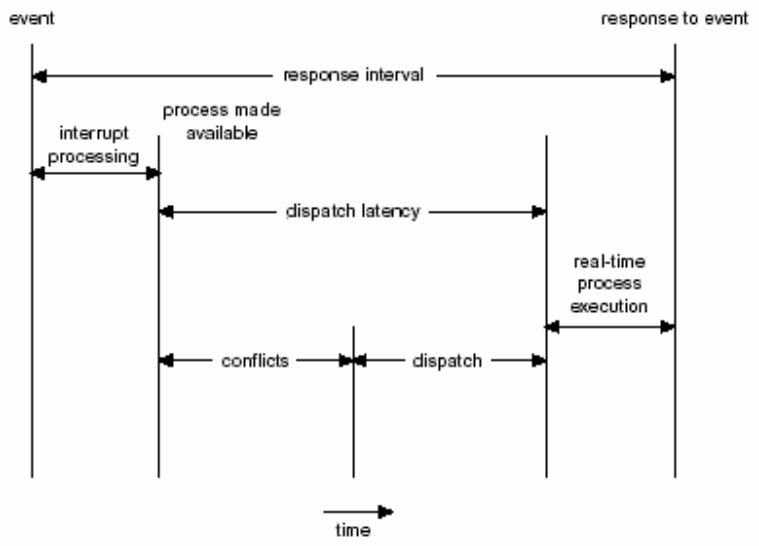
5.4 SCHEDULING IN SISTEMI CON PIÙ PROCESSORI.

Sono considerati i sistemi nei quali i processori sono, in relazione alla loro funzionalità, identici (omogenei): ciascun processore può essere usato per eseguire qualsiasi processo nella coda.
Qualora siano disponibili più processori identici, invece, può verificarsi una condivisione di carico. Sarebbe possibile disporre di una coda distinta per ogni processore, ma in questo caso un processore può essere inattivo, mentre l'altro sovraccarico. Per impedire il verificarsi di tale situazione si utilizza una ready queue comune. Tutti i processi fanno parte di un'unica coda e sono assegnati a un processore disponibile qualsiasi.
In uno schema di questo tipo è possibile seguire due approcci di scheduling. In un primo approccio ogni processore esamina la ready queue comune e seleziona un processo da eseguire. Ma quando più processori tentano di accedere e aggiornare una struttura dati comune, ciascun processore deve essere programmato con la massima attenzione: occorre essere certi che due processori non scelgano lo stesso processo e che i processi non vadano perduti nella coda. L'altro approccio evita questo problema fissando un processore come scheduler per gli altri processori, creando così una struttura master–slave. In alcuni sistemi viene fatto un altro passo nella stessa direzione: tute le decisioni

di scheduling, l'elaborazione dell'I/O e le altre attività di sistema vengono gestiti da un unico processore, il master server, e gli altri processori eseguono soltanto il codice utente. Tale schema, detto multiprocessing asimmetrico, è assai più semplice del multiprocessing simmetrico perché, essendoci un solo processore che può accedere alle strutture di dati di sistema, diminuisce la condivisione di dati.

5.5 SCHEDULING REAL–TIME

Esistono due categorie di elaborazioni real–time. Con i sistemi hard real–time lo scheduler accetta un processo, garantendone il completamento entro i termini di tempo dichiarati, oppure rifiuta la richiesta, in quanto impossibile. Questo comportamento prende il nome di prenotazione delle risorse. Per poter garantire il rispetto delle scadenze, lo scheduler deve conoscere esattamente quanto tempo richiede l'esecuzione di qualsiasi tipo di funzione di sistema, per cui è necessario che l'esecuzione di ogni operazione non richieda più di un tempo massimo specificato. Questa garanzia non può essere rispettata dai sistemi con memoria virtuale o con memorie secondarie.
I vincoli imposti dal soft real–time sono meno restrittivi. È quindi possibile estendere le funzionalità dei sistemi time–sharing, anche se ciò può significare una politica non imparziale di allocazione delle risorse e può comportare l'introduzione di ritardi nell'esecuzione di alcuni processi, con addirittura la possibilità di starvation. Innanzitutto il sistema deve avere uno scheduling con priorità, associando ai processi real–time o valori di priorità più alti. Inoltre, la priorità dei processi real–time non deve diminuire con il trascorrere del tempo, anche se questo può accadere per i processi tradizionali. Infine, la latenza di dispatch deve essere bassa. Quanto più breve è questo intervallo, tanto più celermente un processo real–time pronto per essere eseguito può iniziare le proprie elaborazioni. Il problema risiede nel fatto che molti SO, come UNIX, prima di poter eseguire un context switch devono attendere il completamento della system call attualmente in esecuzione o il verificarsi di un blocco di I/O. Quindi, la latenza di dispatch in questi sistemi può diventare piuttosto lunga, vista la complessità di alcune system call e la lentezza di certi dispositivi di I/O.



Per mantenere bassa la latenza di dispatch è necessario consentire che anche le system call siano prelazionabili. Questo risultato può essere ottenuto in diversi modi. Uno consiste nell'inserire punti di prelazionabilità all'interno di system call di durata particolarmente lunga, in corrispondenza dei quali verificare la presenza di processi ad alta priorità che devono essere eseguiti. In caso affermativo il sistema esegue il context

switch e, una volta terminata l'esecuzione del processo ad alta priorità, il processo interrotto continua con la system call. Questi punti possono essere inseriti solamente in locazioni "sicure" del codice kernel, cioè dove non vengono modificate le strutture di dati dello stesso kernel. Anche questo metodo, comunque, non garantisce la soluzione del problema, poiché il numero di punti di prelazionabilità applicabili al kernel è, nella pratica, assai ridotto.
Un'alternativa consiste nel rendere prelazionabile l'intero kernel. Al fine di assicurare il corretto svolgimento delle operazioni di sistema, tutte le strutture di dati del kernel devono essere protette attraverso l'uso di metodologie di sincronizzazione. Questo approccio consente di prelazionare il kernel in qualsiasi istante, in quanto garantisce l'integrità delle strutture di dati di sistema contro eventuali modifiche da parte di processi ad alta priorità.
Cosa accadrebbe qualora il processo ad alta priorità richiedesse di leggere o modificare le strutture di dati alle quali ha correntemente accesso un altro processo, con minore priorità? Il primo dovrebbe comunque attendere il completamento del secondo, determinando una situazione nota come inversione della priorità. In realtà, ci si potrebbe trovare in una situazione in cui esiste una catena di processi che accedono alle risorse richieste dal processo ad alta priorità. Il problema può essere risolto usando un protocollo di ereditarietà delle priorità, grazie al quale i processi che accedono alle risorse richieste dal processo a priorità più elevata ereditano temporaneamente l'alto grado di priorità, fino al rilascio della risorsa contesa. Dopodiché i processi riprenderanno la loro priorità normale.
La fase conflittuale della latenza di dispatch consiste di tre componenti:

1. La prelazione di tutti i processi attualmente in esecuzione all'interno del kernel.
2. Il rilascio da parte dei processi a bassa priorità delle risorse richieste dal processo ad alta priorità
3. Il context switch tra il processo attualmente in esecuzione e il processo ad alta priorità.

6 SINCRONIZZAZIONE DEI PROCESSI

Un processo cooperante è un processo che può influenzare un altro processo in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè codice e dati) oppure possono condividere dati soltanto attraverso file. Nel primo caso si fa uso di processi lightweight o thread. Dall'accesso concorrente a dati condivisi possono derivare delle situazioni di incoerenza degli stessi dati. In questo capitolo sono trattati vari meccanismi atti ad assicurare una ordinaria esecuzione di processi cooperanti, che condividono uno spazio logico di indirizzi, in modo da preservare la coerenza dei dati.

6.1 INTRODUZIONE

La soluzione del problema del buffer limitato facendo uso di memoria condivisa, consente la presenza nel buffer di non più di n-1 elementi. Si supponga di voler modificare l'algoritmo per rimediare a questa carenza. Una possibilità consiste nell'aggiungere una variabile intera counter inizializzata a 0. Counter viene incrementata

ogni qual volta un nuovo elemento viene aggiunto nel buffer e viene decrementata quando un elemento viene tolto dal buffer.
Il codice per il processo produttore può essere modificato così:

```
repeat
...
produce un elemento in nextp
...
while counter = n do no-op;
buffer[in]:= nextp;
in:=in+1 mod n;
counter:= counter+1;
until false
```

Il codice per il processo consumatore può essere modificato come segue:

```
repeat
while counter = 0 do no-op;
nextc:= buffer[out];
out:=out+1 mod n;
counter:=counter-1;
...
consuma l'elemento in nextc
...
until false
```

Sebbene le routine produttore e consumatore siano corrette quando considerate separatamente, possono non funzionare correttamente quando vengono eseguite in concorrenza. Ad esempio, si supponga che il valore della variabile counter sia attualmente 5, e che i processi produttore e consumatore eseguano le istruzioni counter:=counter+1 e counter:=counter-1 in concorrenza. Terminata l'esecuzione delle due istruzioni, il valore della variabile counter potrebbe essere 4, 5 o 6. Il risultato corretto 5 viene generato solo se le istruzioni vengono eseguite separatamente.
E' possibile dimostrare che il valore di counter può non essere corretto: l'istruzione counter:=counter+1 può essere codificata nel linguaggio macchina come:

```
register1:=counter;
register1:=register1+1;
counter:=register1;
```

dove register1 è un registro locale della CPU. Analogamente, l'istruzione counter:=counter-1 può essere modificata ome

```
register2:=counter;
register2=register2-1;
counter=register2;
```

L'esecuzione concorrente delle istruzioni counter:=counter+1 e counter:=counter-1 equivale a una esecuzione sequenziale delle istruzioni del linguaggio macchina introdotto precedentemente, interfogliate (interleaved) in un qualunque ordine che però conseri l'ordine interno di ogni singola istruzione di alto livello. Uno di questi interleaving è:

T0:	produttore	execute	register1:=counter	(register1=5)
T1:	produttore	execute	register1:=register1+1	(register1=6)
T2:	consumatore	execute	register2:=counter	(register2=5)
T3:	consumatore	execute	register2:=register2-1	(register2=4)
T4:	produttore	execute	counter:=register1	(counter=6)
T5:	consumatore	execute	counter:=register2	(counter=4)

Il procedimento conduce al risultato errato in cui counter=4, ma invertendo l'ordine di t4 e t5 il risultato può diventare 6.
Le situazioni di questo tipo, dove più processi accedono e manipolano gli stessi dati concorrentemente e l'esito dell'esecuzione dipende dall'ordine col quale sono avvenuti gli accessi, sono dette race condition. Per evitare tali situazioni occorre consentire l'accesso alla variabile counter a un solo processo per volta. Questa condizione richiede che i processi vengano in qualche modo sincronizzati. Tali situazioni capitano di frequente nei sistemi operativi: diverse componenti del sistema manipolano delle risorse ma le modifiche effettuate non devono provocare delle interferenze indesiderate per tali componenti.

6.2 Problema della sezione critica.

SI consideri un sistema formato da n processi {P0, P1...Pn-1}. Ciascuno di questi processi ha un segmento di codice, chiamato sezione critica, nel quale il processo può modificare variabili comuni, aggiornare una tabella... Quando un processo è in esecuzione nella sua sezione critica, non deve essere consentito a nessun altro processo di essere in esecuzione nella propria sezione critica. quindi, l'esecuzione di sezioni critiche da parte di processi è mutuamente esclusiva nel tempo. Il problema della sezione critica consiste nel progettare un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che implementa questa richiesta è la entry section. La sezione critica può essere seguita da una exit section, e la restante parte del codice è detta sezione non critica.
Una soluzione del problema della sezione critica deve soddisfare i tre seguenti tre requisiti:
1) Mutua esclusione. Se il processo Pi è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella sua sezione critica.
2) Progresso. Se nessun processo è in esecuzione nella sua sezione critica ed esiste qualche processo che desidera entrare nella sua sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non può essere rimandata indefinitivamante.
3) Attesa limitata. Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite di numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che la richiesta del primo processo sia stata accordata.
Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non è possibile fare alcuna ipotesi sulla velocità relativa degli altri processi.

6.2.1 SOLUZIONE PER DUE PROCESSI

I processi sono indicati con P0 e P1. per motivi di convenienza, una volta introdotto Pi, l'altro processo viene denotato con Pj, con j=i-1

6.2.1.1 ALGORITMO 1

Il primo approccio consiste nel far condividere ai processi una variabile intera, turn, inizializzata a 0. Se turn =i, al processo Pi è permesso di entrare in esecuzione sulla propria sezione critica.
Questa soluzione assicura che, in un dato momento, solo un processo può trovarsi nella propria sezione critica. Tuttavia la soluzione non soddisfa il requisito di progresso, perché richiede una stretta alternanza di processi in esecuzione della sezione sezione critica.. Se, ad esempio, turn=0, P1 non può entrare nella sua sezione critica, anche se P0 si trova nella propria sezione non critica.

6.2.1.2 ALGORITMO 2

L'algoritmo 2 non possiede informazioni sufficienti sullo stato di ogni processo; esso ricorda solo il processo a cui è permesso di entrare nella propria sezione critica. per rimediare a questo problema è possibile sostituire la variabile turn con ilo seguente array:
var flag: array[0..1] of boolean;
Gli elementi dell'array sono inizializzati a false. Se flag[i] è a true, significa che pi è pronto per entrare nella sezione critica.
In questo algoritmo kl processo Pi come prima cosa pone flag[i] a true, segnalando che è pronto per entrare nella propria sezione critica, quindi, verifica che il processo Pj non sia pronto per entrare nella propria sezione critica. Se Pj fosse pronto, allora Pi attenderebbe fino a che Pj avesse indicato che non intende più restare nella sua sezione critica, vale a dire che falg[j]=false. A questo punto Pi entrerebbe nella sua sezione critica. Uscendo dalla sezione critica, il processo Pi imposterebbe il proprio flag a false, permettendo a un altro processo, ammesso che ce ne sia qualcuno in attesa, di entrare nella propria sezione critica.
Questa soluzione soddisfa il requisito di mutua esclusione. Sfortunatamente non è soddisfatto il prerequisito del progresso.
Per illustrare questo problema si consideri la seguente sequenza esecutiva:
T0: P0 pone flag[0]=true;
T1: P1 pone flag[1]=true;
Ora P0 e P1 entrano in un ciclo infinito nelle rispettive istruzioni while.

6.2.1.3 ALGORITMO 3

Combinando i concetti chiave dell'algoritmo 1 e 2 è possibile ottenere una soluzione corretta per il problema della sezione critica, in cui tutti e tre i requisiti sono soddisfatti. I processi condividono due variabili:
var flag: array[0..1] of boolean;
turn:0..1;
Inizialmente flag[0]=falg[1]=false, è il valore di turn è irrilevante (o 0 o 1).

Per entrare nella sezione critica il processo Pi deve prima porre flag[i]=true, e poi asserire che il turno di entrata è dell'altro processo. Se entrambi i processi tentano di entrare contemporaneamente, turn viene impostato sia su i che su j più o meno nello stesso momento. Solo una di queste due assegnazioni può conservarsi, mentre l'altra si verificherà, ma viene immediatamente sovrascritta. Il valore finale di turn consente di stabilire a quale dei due processi debba essere permesso di entrare per primo nella propria sezione critica.

6.2.2 SOLUZIONE PER PIÙ PROCESSI

L'algoritmo 3 risolve il problema della sezione critica per due processi. in questo paragrafo è sviluppato un algoritmo allo scopo di risolvere il problema della sezione critica per n processi. Tale algoritmo, noto anche come algoritmo del fornaio, è basato su uno schema di servizio comunemente usato nelle panetterie: al suo ingresso in negozio, ogni cliente riceve un numero. Viene servito di volta in volta il cliente con il numero più basso. Sfortunatamente l'algoritmo del fornaio non può garantire che due processi non ricevano lo stesso numero. In questo caso viene servito il processo con il nome più basso. Dal momento che i nomi dei processi sono unici e completamente ordinati, l'algoritmo è del tutto deterministico.
Le strutture di dati comuni sono
var choosing: array[0..n-1] of boolean;
number: array[0..n-1] of integer;
Da principio queste strutture di dati sono inizializzate rispettivamente a false e a 0. Per motivi di convenienza si definisce la seguente notazione:
(a,b)<(c,d) se a<c oppure se a=c e b<d:
max(a0,...,an-1) è un numero k, tale che k=>ai per i=0,...,n-1
Per provare che l'algoritmo del fornaio è corretto, occorre innanzitutto dimostrare che se Pi si trova nella propria sezione critica e Pk (K!=i) ha già scelto il proprio number[k]!=0, allora (number[i], i)<(number[k], k).
Da to questo risultato, è facile dimostrare che la mutua esclusione è soddisfatta. Si consideri ora di avere Pi nella propria sezione critica, e Pk che tenta di entrarci. Quando il processo Pk esegue la seconda istruzione while per j=i, trova che
- number[i] != 0
- (number[i], i)<(number[k], k).
Quindi, continua il ciclo dell'istruzione while fino a che Pi non lascia la propria sezione critica. Per dimostrare i requisiti di progresso e di attesa limitata, e che l'algoritmo assicura imparzialità, è sufficiente osservare che i processi entrano nelle rispettive sezioni critiche seguendo il criterio first-come, first-served.

6.3 HARDWARE DI SINCRONIZZAZIONE

Tale problema potrebbe essere risolto semplicemente se si potessero disabilitare gli interrupt mentre una variabile condivisa viene modificata. In questo modo si assicurerebbe una esecuzione ordinata e non prelazionabile della corrente sequenza di istruzioni. Nessun altra istruzione potrebbe essere eseguita, quindi nessuna modifica inaspettata potrebbe essere apportata alla variabile condivisa. Sfortunatamente questa

soluzione non è sempre praticabile. La disabilitazione degli interrupt sui sistemi multiprocessore può comportare degli sprechi di tempo dovuti alla necessità di trasmettere la richiesta di disabilitazione a tutti i processori. Tale trasmissione ritarda l'accesso a ogni sezione critica determinando una diminuzione dell'efficienza. Si considerino, inoltre, gli effetti sul clock di sistema se questo viene aggiornato ad interrupt. Per questo motivo molte macchine offrono particolari istruzioni hardware che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare atomicamente il contenuto di due parole di memoria. Queste speciali istruzioni possono essere utilizzate per risolvere il problema della sezione critica in modo relativamente semplice. L'istruzione test-and-set viene eseguita atomicamente, e cioè come unità non soggetta ad interrupt. Quindi, se due istruzioni test-and-set vengono eseguite contemporaneamente, ciascuna su una cpu diversa, queste vengono eseguite sequenzialmente in ordine arbitrario.
Se la macchina supporta l'istruzione test-and-set, è possibile implementare la mutua esclusione dichiarando la variabile booleana lock, inizializzata a false.
L'istruzione swap agisce sul contenuto di due parole, e come l'istruzione test-and-set è eseguita atomicamente.
Se la macchina supporta l'istruzione swap, allora la mutua esclusione viene garantita dichiarando e inizializzando a false una variabile booleana globale lock. Inoltre, ogni processo possiede anche una variabile locale key.
Questi algoritmi non soddisfano il requisito di attesa limitata. Tutti e tre i requisiti sono soddisfatti usando anche l'istruzione test-and-set.

6.4 SEMAFORI

Le soluzioni al problema della sezione critica non possono essere eseguite facilmente su problemi più complessi. Per superare questa difficoltà è possibile utilizzare uno strumento di sincronizzazione, chiamato semaforo. Un semaforo s è una variabile intera alla quale è possibile accedere, a prescindere dall'inizializzazione, solo tramite due operazioni atomiche standard: wait e signal. Le definizioni classiche di wait e signal sono le seguenti:

```
wait(s):    while s=<= do no-op;
            s:=s-1;

signal(s):  s=s+1;
```

Le modifiche al valore del semaforo contenute nelle operazioni wait e signal devono essere eseguite in modo indivisibile. Ciò significa che, se un processo modifica il valore di un semaforo, nessun altro processo può modificare contemporaneamente quel valore. Inoltre, nel caso di wait(s) devono essere eseguiti senza interruzione anche il test del valore intero s e la sua possibile modifica.

6.4.1 USO DEI SEMAFORI

I semafori possono essere utilizzati anche per risolvere il problema della sezione critica con n processi. Gli n processi condividono un semaforo comune, mutex (da mutual exlusion), inizializzato a 1.

I semafori possono essere utilizzati anche per risolvere diversi problemi di sincronizzazione. Si considerino, ad esempio, due processi in esecuzione concorrente: P1 con un'istruzione S1 e P2 con un'istruzione S2. Si supponga di voler eseguire S2 solo dopo che S1 è terminata. Questo schema può essere prontamente implementato facendo condividere a P1 e a P2 un semaforo comune sync, inizializzato a 0, e inserendo nel processo P1 le istruzioni
S1;
signal(sync);
e nel processo P2 le istruzioni
wait(sync);
S2;
Poiché sync è inizializzato a 0, P2 esegue S2 solo dopo che S1 ha richiamato signal(sync), che si trova dopo S1.

6.4.2 REALIZZAZIONE

Lo svantaggio maggiore delle soluzioni al problema della mutua esclusione e della definizione di semaforo, è che esse richiedono una condizione busy waiting. Mentre un processo si trova nella sezione critica, qualsiasi altro processo tenti di entrare in una sezione critica viene sempre a trovarsi nel loop del codice della entry section. Chiaramente questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione di busy waiting spreca cicli di cpu che potrebbero essere sfruttati produttivamente da un altro processo. Questo tipo di semaforo è anche detto spinlock, perché gira (spins) finché rimane bloccato (lock) in attesa. Gli spinlock sono utili nei sistemi con multiprocessore. Il vantaggio apportato da uno spinlock consiste nel fatto che nessun context switch deve essere effettuato quando un processo attende su un blocco; occorre ricordare che un context switch può richiedere parecchio tempo. Quindi, gli spinlock risultano utili quando è probabile che i blocchi permangono per tempi molto brevi. Per superare la necessità di restare in busy waiting, è possibile modificare le definizioni di wait e signal. Quando un processo esegue l'operazione wait e trova che il valore del semaforo non è positivo, deve attendere. Tuttavia, anziché rimanere in busy waiting, il processo può bloccare se stesso. L'operazione di blocco pone un processo in una coda d'attesa associata al semaforo e lo stato del processo viene commutato in stato waiting. Quindi, il controllo viene trasferito allo scheduler della cpu che seleziona un altro processo pronto per l'esecuzione.
Un processo bloccato, in attesa su un semaforo S, deve essere riavviato da un'operazione signal eseguita su S da un'altro processo. Il processo viene riavviato da un'operazione wakeup che modifica lo stato del processo da waiting a ready. Quindi, il processo viene posto nella ready queue. La cpu può essere o meno commutata dal processo in esecuzione al processo appena pronto, a seconda dell'algoritmo di scheduling della cpu.
Per implementare i semafori secondo quanto detto è possibile definire il semaforo come un record:

```
type semaphore = record
                    value:nteger;
                    L: list of process;
                    end
```

Ogni semaforo è associato a un valore intero e a una lista di processi. L'operazione signal preleva un processo dalla lista dei processi in attesa e lo attiva.
A questo punto le operazioni del semaforo possono essere definite come segue:

```
wait(s):        S.value:=S.value-1;
                if S.value<0
                        then begin
                                aggiungi questo processo a S.L;
                                block;
                        end;
signal(s);      S.value:=S.value+1
                if S.value=<0
                        then begin
                                togli un processo P da S.L;
                                wakeup(P);
                        end;
```

7 STALLO (DEADLOCK)

Tutti i sistemi operativi sono dotati della possibilità di assegnare temporaneamente l'accesso esclusivo di un processo ad una certa risorsa. Lo *stallo* è una situazione nella quale i processi rimangono sospesi per un tempo infinito.

7.1 LE RISORSE

Dividiamo le risorse in:

- *prerilasciabile* può essere rimossa dal processo che la sta usando senza avere effetti dannosi (es. memoria centrale)
- *non prerilasciabile* viene rilasciata provocando il fallimento dell'operazione in atto (es. stampanti).

7.2 GLI STALLI

Definizione: un insieme di processi si trova in una situazione di stallo se ogni processo dell'insieme aspetta un evento che solo un altro processo dell'insieme può provocare.
Devono valere quattro condizioni perché possa verificarsi lo stallo:

1. condizione di mutua esclusione: ogni risorsa è assegnata ad un solo processo oppure è disponibile
2. condizione di prendi e aspetta: i processi che già hanno avuto e ottenuto delle risorse ne possono chiedere delle altre
3. condizione di mancanza di prerilascio: le risorse che sono già state assegnate ad un processo non gli possono essere tolte in modo forzato, ma devono essere rilasciate volontariamente dal processo che le detiene
4. condizione di attesa circolare: deve esistere una catena circolare di processi ognuno dei quali aspetta il rilascio di una risorsa da parte del processo che lo segue.

Si può rappresentare graficamente lo stallo utilizzando i grafi orientati: processi rappresentati da cerchietti e risorse da quadratini. Da □ a O indica che la risorsa è assegnata al processo, viceversa il processo attende la risorsa.
In generale per trattare adeguatamente le situazioni di stallo esistono quattro strategie:
1. ignorare semplicemente il problema
2. cercare di scoprire le situazioni di stallo e risolverle
3. cercare dinamicamente di evitare le situazioni di stallo con un'accorta politica di allocazione delle risorse
4. prevenire le situazioni di stallo negando una delle quattro condizioni necessarie.

7.3 ALGORITMO DELLO STRUZZO

Facciamo finta che il problema non esista, la situazione di stallo si presenta talmente raramente che risolvere il problema genera un costo non conveniente.

7.4 IDENTIFICAZIONE E RISOLUZIONE DELLO STALLO

Con questa tecnica non si previene lo stallo ma si cerca di risolverlo. La situazione più semplice è quando si ha una sola risorsa per tipo.
Algoritmo per ispezionare il grafo:
1. per ogni nodo N del grafo eseguire i 5 passi successivi con N come nodo di partenza
2. inizializzare L come lista vuota e definire tutti gli archi come non marcati
3. aggiungere il nodo corrente alla fine di L e controllare che non appaia in L due volte. Se così fosse contiene un ciclo e termina
4. da un dato nodo guardare se vi sono archi uscenti non marcati, in questo caso andare al passo 5 altrimenti al passo 6
5. considerare un arco non marcato a caso e marcarlo. Poi seguirlo fino al nuovo nodo corrente e andare al passo 3
6. termine. Ritornare al nodo precedente, renderlo corrente e andare al passo 3. Se il nodo è quello di partenza il grafo non contiene cicli.

7.4.1 RISOLUZIONE DELLO STALLO MEDIANTE PRERILASCIO

Tolgo la risorsa a un processo e l'assegno ad un altro. Alcune volte è un'operazione manuale come nel caso di stampanti. Dipende dal tipo di risorsa in oggetto.

7.4.2 RISOLUZIONE MEDIANTE RITORNO A UNO STATO PRECEDENTE

Si può costruire un punto di controllo check point cioè lo stato di un processo è scritto in un file, e contiene non solo l'immagine della memoria ma anche lo stato delle risorse. Per risolvere lo stallo un processo che possiede una risorsa richiesta è riportato ad uno stato precedente al tempo che ha acquistato tale risorsa facendolo ripartire da uno dei suoi punti di controllo.

7.4.3 RISOLUZIONE DELLO STALLO MEDIANTE ELIMINAZIONE DEL PROCESSO

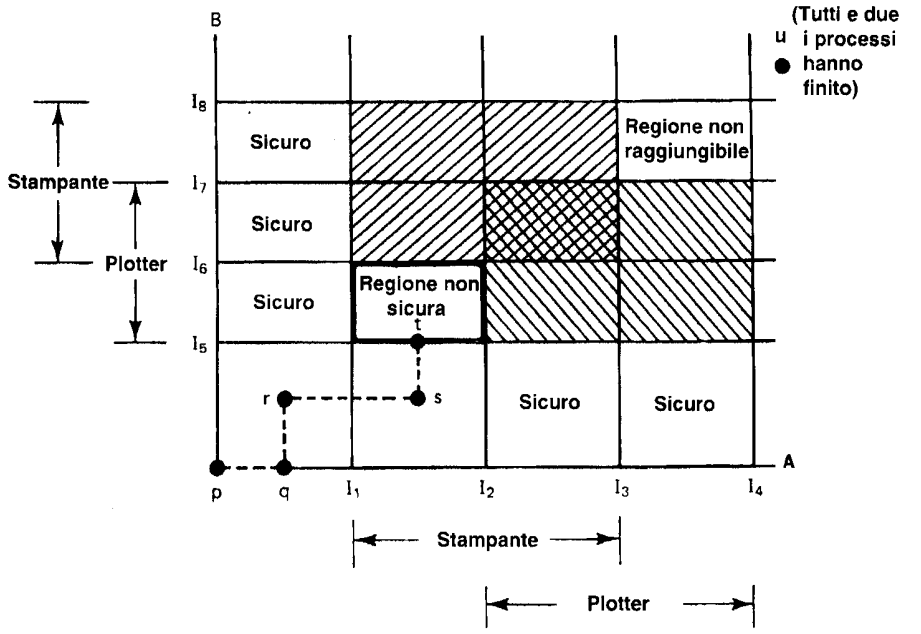
Si può eliminare uno o più processi del ciclo.

7.5 EVITARE LO STALLO

Si possono evitare gli stalli ma solo se alcune informazioni sono disponibili in anticipo.

7.5.1 TRAIETTORIE DI RISORSA

Gli algoritmi principali per evitare lo stallo si basano sul concetto degli *stati sicuri*. Prima di descrivere gli algoritmi si parlerà brevemente del problema della sicurezza utilizzando un disegno che ne renderà più facile la comprensione. Sebbene 1'approccio grafico non si possa tradurre



direttamente in un algoritmo, esso permette di capire abbastanza bene, in modo intuitivo, il problema.
In figura si vede un modello in grado di trattare due processi e due risorse: una stampante ed un plotter, per esempio. L'asse orizzontale rappresenta il numero di istruzioni eseguite dal processo A. L'asse verticale rappresenta il numero di istruzioni eseguite dal processo B. In I1, A richiede una stampante; in I2, ha bisogno di un plotter. La stampante ed il plotter sono rilasciati in I3 e in I4, rispettivamente. Il processo B ha bisogno del plotter da I5 a I7 e della stampante da I6 a I8.
Ogni punto nel diagramma rappresenta uno stato congiunto dei due processi. Inizialmente, lo stato è in p, e nessuno dei due processi ha eseguito alcuna istruzione. Se lo scheduler sceglie A per farlo girare per primo, si arriva al punto q, nel quale A ha eseguito un certo numero di istruzioni e B non ne ha eseguita nessuna. Nel punto q, la traiettoria diventa verticale, indicando che lo scheduler ha scelto di eseguire B. *Se si ha un singolo processore, tutti i cammini devono essere orizzontali o verticali, cioè non devono essere mai diagonali*.
In più, lo spostamento avviene sempre verso nord o verso est e mai verso sud o verso ovest (i processi non possono ritornare indietro).
Quando A attraversa la linea I1 sul cammino da r a s, significa che esso ha richiesto ed ottenuto la stampante. Quando B raggiunge il punto t, esso richiede il plotter.
Le regioni di piano tratteggiate sono particolarmente interessanti. La regione tratteggiata con le linee da sud-ovest a nord-est (///), rappresenta lo stato in cui entrambi i processi detengono la stampante. La regola di *mutua esclusione* fa sì che sia impossibile entrare in

questa zona. In maniera analoga, la regione tratteggiata in maniera opposta rappresenta lo stato in cui tutti e due i processi hanno il plotter, il che è ancora vietato dalla mutua esclusione.
Se il sistema entrasse nella scatola delimitata da I1 e I2 sui lati verticali e da I5 e I6 sopra e sotto, esso si bloccherebbe sull'incrocio di I6 con I2. A questo punto, A sta richiedendo il plotter e B la stampante, ma entrambe queste risorse sono già state assegnate. L'intera scatola non è sicura e quindi non vi si deve mai entrare. Al punto t, la sola cosa sicura che si può fare è di far girare il processo A finché arriva a I4. Dopo questo punto, una qualunque traiettoria che arrivi ad u può essere scelta con successo.
La cosa importante da notare è che, nel punto t, B sta richiedendo una risorsa. Il sistema deve decidere se assegnarla o meno, ma se la assegna il sistema entrerà in una regione non sicura e probabilmente andrà in stallo. Per evitare lo stallo B deve essere sospeso finché A non abbia richiesto e rilasciato il plotter.
Uno stato è detto **sicuro** se non è in stallo e se c'è un modo per soddisfare tutte le richieste correntemente pendenti eseguendo processi in un qualche ordine. Uno stato è sicuro quando il sistema mediante un'opportuna schedulazione può evitare lo stallo. La differenza tra uno stato sicuro e uno non sicuro è che in uno stato sicuro il sistema può garantire che tutti i processi avranno termine, mentre in uno stato non sicuro non c'è nessuna garanzia.

7.5.2 ALGORITMO DEL BANCHIERE PER UNA SINGOLA RISORSA

L'algoritmo del banchiere considera ogni richiesta nel momento in cui essa si presenta e analizza se essa porta ad uno stato sicuro. Se lo fa, soddisfa la richiesta, altrimenti la richiesta viene postposta. Per vedere se uno stato è da ritenersi sicuro, il banchiere controlla di avere abbastanza risorse a disposizione per poter soddisfare le richieste del cliente che si trova più vicino degli altri al massimo della linea di credito. Se è così questi prestiti possono essere considerati restituiti e si può controllare il cliente che si trova ora più vicino al massimo della propria linea di credito. Se tutti i prestiti possono essere restituiti, lo stato è sicuro e la richiesta iniziale può essere soddisfatta.

7.5.3 ALGORITMO DEL BANCHIERE PER RISORSE MULTIPLE

Sono usate due matrici che indicano il numero di risorse utilizzate e il numero di risorse ancora necessarie per completare l'operazione. Prima di iniziare l'esecuzione i processi devono comunicare il loro fabbisogno globale di risorse. Tre vettori indicano le risorse esistenti (E), quelle assegnate (P) e quelle disponibili (A).

1. Si cerchi una riga R per la quale le risorse ancora non richieste sono meno *o uguali* di quelle in A. Se non esiste una riga di questo tipo il sistema andrà in stallo perché nessun processo può essere eseguito e terminare.
2. Si assuma che il processo della riga scelta richieda tutte le risorse di cui ancora abbisogna e finisca il proprio lavoro. Si marchi il processo come terminato e si aggiungano al vettore A tutte le risorse che deteneva.
3. Si ripetano i punti 1 e 2 fino a quando o tutti i processi vengono marcati e quindi lo stato di partenza era sicuro, oppure *non* ci si trova in una situazione di stallo nel qual caso lo stato di partenza non era sicuro.

Problemi: raramente i processi sanno in anticipo il numero di risorse richieste, il numero di processi non è fisso, alcune risorse possono non essere più disponibili (si danneggiano). In pratica pochi o nessuno dei S.O. usa questo algoritmo.

7.6 PREVENZIONE DELLA SITUAZIONE DI STALLO

1. Negazione mutua esclusione: ad esempio con la stampante si può usare lo spool e l'unico processo che richiede fisicamente la stampante è il demone di stampa. Si può inoltre evitare di assegnare una risorsa quando non è strettamente necessario e assicurare che pochi processi, il minor numero possibile, esigano la risorsa.
2. Negazione della condizione hold-and-wait: bisogna evitare che un processo che già detiene delle risorse ne chieda delle altre, ciò è possibile se tutti i processi richiedono tutte le risorse prima di iniziare l'esecuzione. Problema: molti processi non sanno le risorse che chiederanno e inoltre non viene ottimizzato l'uso delle risorse. Un modo differente è far sì che un processo che richiede una risorsa rilasci temporaneamente tutte le risorse che detiene in quel momento, se la richiesta ha successo può riavere indietro le risorse.
3. Negazione della condizione di mancanza di prerilascio: poco conveniente.
4. Negazione della condizione di attesa circolare: ogni processo ha una sola risorsa, se ne chiede una seconda deve rilasciare la prima, poco conveniente. Oppure si numerano le risorse e un processo per richiederle deve seguire l'ordine. Il problema è che il numero delle risorse potenziali e dei differenti utenti può essere tanto grande che nessun ordinamento potrebbe funzionare.

7.7 ALTRI ARGOMENTI SULLO STALLO

7.7.1 IL BLOCCO A DUE FASI

Ad esempio nelle basi di dati il processo prima prova a bloccare tutti i record, uno alla volta, esegue gli aggiornamenti e rilascia i lock. Se durante la prima fase si ha bisogno di qualche record bloccato, il processo li rilascia e ricomincia da capo. L'algoritmo funziona solo in quelle situazioni in cui il programma può essere interrotto in qualsiasi punto durante la prima fase e fatto ripartire.
Si possono avere gli stalli anche senza le risorse, ad esempio con i semafori

7.7.2 LA MANCANZA DI RISORSE

Un processo non viene mai servito cioè sarà postposto indefinitamente anche se non è bloccato. La mancanza di risorse si può evitare usando una politica first-in first-out.

PARTE TERZA: GESTIONE DELLA MEMORIA

8 GESTIONE DELLA MEMORIA

Gli algoritmi di gestione della memoria variano da un approccio primitivo sulla nuda macchina fino a strategie di paginazione e segmentazione. La scelta di uno schema specifico di gestione della memoria dipende da molti fattori, in particolar modo dalla struttura dell'hardware del sistema, poiché molti algoritmi richiedono il supporto dell'hardware.

8.1 INTRODUZIONE

Con il termine memoria viene indicato un ampio array di parole o byte, ciascuno con il proprio indirizzo. La cpu preleva le istruzioni dalla memoria in base al valore del program counter; tali istruzioni possono determinare ulteriori letture (load) e scrittura (store) in specifici indirizzi di memoria.

8.1.1 BINDING DEGLI INDIRIZZI

In genere un programma risiede su un disco in forma di file binario eseguibile. Il programma per essere eseguito deve essere portato in memoria e inserito all'interno di un processo. Secondo il tipo di gestione della memoria utilizzato, durante la sua esecuzione il processo può essere trasferito dalla memoria la disco e viceversa. L'insieme dei processi su disco in attesa di essere trasferiti in memoria per essere eseguiti forma la coda di input. Generalmente gli indirizzi del programma sorgente sono simbolici (come, ad esempio, count). Un compilatore di solito associato (bind) questi indirizzi simbolici a indirizzi rilocabili. Il linkage editor, o il loader, collega a sua volta questi indirizzi rilocabili a indirizzi assoluti (ad es. 74014). Ogni collegamento rappresenta un mapping da uno spazio di indirizzi a un altro.
Generalmente l'associazione (binding) di istruzioni e dati a indirizzi di memoria può essere effettuata in qualsiasi fase del seguente percorso:

- Compilazione: se la momento della compilazione è possibile sapere dove il processo risiederà in memoria, può essere generato un codice assoluto.
- Caricamento: se la momento della compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare un codice rilocabile. In questo caso il collegamento finale viene ritardato fino al momento del caricamento.
- Esecuzione: se durante l'esecuzione il processo può essere spostato da un segmento di memoria ad un altro, il collegamento deve essere ritardato fino al momento dell'esecuzione.

8.1.2 CARICAMENTO DINAMICO

Per migliorare l'utilizzo della memoria è possibile ricorrere al caricamento dinamico, mediante il quale una routine viene caricata solo quando viene richiamata. Tutte le routine vengono conservate su disco in un formato di caricamento rilocabile. Il programma principale viene caricato in memoria e quindi eseguito. Quando una routine deve chiamarne un'altra, la prima controlla innanzitutto se la seconda sia stata caricata.

Se non lo è, viene chiamato il loader di collegamento rilocabile per caricare in memoria la routine richiesta e aggiornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla routine appena caricata.
Il vantaggio dato dal caricamento dinamico consiste nel fatto che una routine non utilizzata non viene mai caricata.

8.1.3 COLLEGAMENTO DINAMICO

La maggior parte dei SO supporta solo un collegamento statico, nel quale le librerie di sistema del linguaggio sono trattate come qualsiasi altro modulo oggetto e vengono combinate dal loader nell'immagine binaria del programma. Il concetto di collegamento dinamico è analogo a quello del caricamento dinamico. Invece di posticipare il caricamento di una routine fino al momento dell'esecuzione, viene posticipato il collegamento. Questa caratteristica viene utilizzata soprattutto con librerie di sistema, come, ad esempio, le librerie di subroutine del linguaggio. Senza questo strumento tutti i programmi di sistema dovrebbero disporre all'interno dell'immagine eseguibile di una copia delle librerie del linguaggio. Tutto ciò richiede spazio su disco e memoria principale. Con il collegamento dinamico, invece, per ogni riferimento a una routine di libreria viene inserito uno stub: si tratta di una piccola porzione di codice che indica come localizzare la giusta routine di libreria residente in memoria o come caricare la libreria se la routine non è già presente.
Quando lo stub viene eseguito, esso controlla se la routine richiesta è già in memoria, se non lo è provvede a caricarla. In ogni caso lo stub rimpiazza se sé tesso con l'indirizzo della routine, che viene poi eseguita. Quindi, quando viene nuovamente raggiunto quel segmento di codice, la routine di libreria viene eseguita direttamente, senza costi aggiuntivi per il collegamento dinamico.
In memoria è possibile caricare più di una versione della stessa libreria, e ciascun programma utilizza l'informazione sulla versione per decidere quale copia debba usare. Se le modifiche sono di piccola entità, il numero di versione rimane invariato, mentre se l'entità delle modifiche diviene significativa, aumenta anche il numero di versione. Perciò solo i programmi compilati con la nuova versione della libreria subiscono gli effetti delle modifiche incompatibili incorporati nella libreria stessa. Questo sistema è conosciuto con il nome di librerie condivise.

8.1.4 OVERLAY

Fin qui è stato detto che affinché un processo possa essere eseguito il programma e i dati di tale processo devono interamente trovarsi nella memoria fisica. La dimensione del processo è limitata dalla dimensione della memoria fisica. Talvolta, per permettere a un processo di essere più grande della memoria che gli viene allocata, viene utilizzata una tecnica chiamata overlay. Il concetto di overlay è basato sulla possibilità di mantenere in memoria solo le istruzioni e i dati che vengono utilizzati con maggiore frequenza. Quando sono necessarie altre istruzioni, queste vengono caricate nello spazio che erano precedentemente occupato dalle istruzioni che non vengono più utilizzate.

Gli overlay non richiedono alcun supporto speciale da parte del SO, ma possono essere implementati direttamente dall'utente. L'implementazione può essere effettuata per mezzo di semplici strutture di file, leggendo da questi in memoria e quindi trasferendo il controllo a quest'ultima per eseguire le istruzioni appena lette. Il SO annovera solo la presenza di I/O supplementari.
Il programmatore, quindi, deve progettare e programmare adeguatamente la struttura degli overlay. Questo compito può essere un'impresa considerevole, poiché richiede la conoscenza completa della struttura del programma, del suo codice e delle sue strutture dati. Poiché, per definizione, il programma è di grandi dimensioni (programmi piccoli non richiedono overlay) può essere difficile comprenderne sufficientemente la struttura. Per questi motivi, l'utilizzo degli overlay è attualmente limitato ai microcomputer ed altri sistemi dotati di quantità di memoria limitate e che non dispongono del supporto hardware necessario per implementare tecniche più avanzate. Alcuni compilatori per microcomputer supportano gli overlay per semplificare il compito ai programmatori. Sono, comunque, sicuramente preferibili tecniche automatiche per l'esecuzione di grossi programmi in quantità limitate di memoria fisica.

8.2 SPAZI DI INDIRIZZO LOGICI E FISICI

Un indirizzo generato dalla cpu viene di solito indicato come indirizzo logico, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel memory address register della memoria, è di solito indicato come indirizzo fisico.
Dagli schemi di binding di indirizzi a tempo di compilazione e a tempo di caricamento deriva un ambiente in cui gli indirizzi logici sono gli stessi di quelli fisici. Con gli schemi di binding a tempo di esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce, di solito, agli indirizzi logici col termine di indirizzi virtuali; in questo caso tali termini vengono usati in modo intercambiabile.
L'insieme di tutti gli indirizzi logici generati da un programma è definito spazio di indirizzi logici; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è chiamato spazio di indirizzi fisici. Quindi, con lo schema di binding degli indirizzi a tempo di esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.
Il mapping run–time degli indirizzi virtuali sugli indirizzi fisici viene effettuato da un dispositivo hardware detto memory–management unit (MMU). Il registro base viene ora denominato registro di rilocazione; il valore di quest'ultimo viene sommato ad ogni indirizzo generato da un processo utente nel momento in cui questo viene inviato in memoria. Ad esempio, se la base si trova a 14.000, allora un tentativo da parte dell'utente di indirizzare la locazione 0 viene rilocato dinamicamente alla locazione 14.000; un accesso alla locazione 356 viene mappato sulla locazione 14.356.
Il programma utente non considera mai gli indirizzi fisici reali. Il programma crea un puntatore alla locazione 356, lo memorizza, lo manipola, lo confronta con altri indirizzi, il tutto in il numero 356. Solo quando è utilizzato come indirizzo di memoria viene rilocato relativamente al registro base. Il programma utente tratta gli indirizzi logici. L'hardware che effettua il mapping della memoria converte gi indirizzi logici in indirizzi fisici.

In questo caso esistono due tipi di indirizzi: indirizzi logici (nell'intervallo da 0 a max) e indirizzi fisici (da 0+R a R+max per un valore base di R).

8.3 SWAPPING

Un processo deve trovarsi in memoria per essere eseguito. Tuttavia, esso può essere temporaneamente riversato (swapped) nella backing store (memoria secondaria di supporto) dalla quale viene riportato in memoria al momento di riprendere l'esecuzione. Si consideri, ad esempio, un ambiente di multiprogrammazione con un algoritmo RR per lo scheduling della cpu. Trascorso un quanto di tempo, il gestore di memoria esegue lo swap out del processo appena terminato ed effettua lo swap in di un altro processo nello spazio di memoria appena liberato. Nel frattempo lo scheduler della cpu alloca un quanto di tempo a un altro processo presente in memoria.
Una variante di questa politica di swapping viene utilizzata per gli algoritmi basati sulle priorità. Se un processo con priorità maggiore arriva e richiede il servizio, il gestore di memoria può effettuare lo swap out del processo con priorità inferiore in modo da poter caricare ed eseguire il processo con priorità maggiore. Quando il processo con priorità maggiore termina, quello con priorità minore può essere richiamato in memoria e continuare la propria esecuzione. Questo tipo di swapping viene talvolta chiamato roll out, roll in.
Normalmente, un processo sottoposto a swap out, al momento dello swap in viene riversato nello spazio di memoria occupato in precedenza. Questa limitazione è dovuta al metodo di binding degli indirizzi. Se il binding viene effettuato al momento dell'assemblaggio o del caricamento, il processo non può essere riversato in posizioni diverse. Se il binding viene effettuato al tempo dell'esecuzione, un processo può essere riversato in uno spazio di memoria diverso, poiché gli indirizzi fisici vengono calcolati nel momento dell'esecuzione.
Lo swapping richiede una backing store. Tale memoria di supporto deve essere sufficientemente ampia da contenere copie di tutte le immagini di memoria di tutti i processi utente, e deve permettere un accesso diretto a dette immagini di memoria. Il sistema conserva una ready queue formata da tutti i processi le cui immagini di memoria si trovano nella backing store o in memoria e sono pronti per l'esecuzione. Quando lo scheduler della cpu decide di eseguire un processo, richiama il dispatcher, il quale a sua volta controlla se il primo processo della coda si trova in memoria. Se il processo non si trova in memoria, e in questa non c'è spazio libero, il dispatcher sottopone a swap out un processo attualmente presente in memoria e a swap in il processo richiesto dallo scheduler della cpu, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato.
Dovrebbe essere evidente che, in un siffatto sistema di swapping, il tempo di context switch è abbastanza elevato. Per avere un'idea del tempo di context switch si pensi a un processo utente di 100K e a una backing store costituita da un hard disk standard con velocità di trasferimento di 1 MB al secondo. Il trasferimento effettivo del processo di 100K da e in memoria richiede 11K/1.000K al secondo = 1/10 secondo = 100 millisecondi. Assumendo che non siano necessari movimenti della testina (seek) e una latenza media di 8 millisecondi, l'operazione di swap richiede 108 millisecondi. Dal

momento che è necessario effettuare sia swap in che swap out, il tempo totale di swap è 216 millisecondi.
Per utilizzare efficacemente la cpu è necessario che il tempo di esecuzione di ogni processo sia lungo rispetto al tempo di swap. Ad esempio, in un algoritmo di scheduling RR il quanto di tempo deve essere > 0.216 secondi.
Occorre notare che la maggior parte del tempo di swap è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria sottoposta a swap. In un computer con 1MB di memoria principale e un SO residente di 100K, la dimensione massima del processo utente è 900K. Tuttavia possono essere presenti molti processi utenti con dimensione minore, ad esempio 100K. Lo swap out di un processo da 100K può concludersi in 108 millisecondi, mentre per lo swap di 900K ne occorrono 908. Perciò sarebbe utile sapere quanta memoria viene effettivamente utilizzata da un processo utente e non solo quanta questo potrebbe teoricamente utilizzarne, poiché in questo caso è necessario effettuare lo swap out solo su quanto viene effettivamente usato, riducendo così il tempo di swap.
Attualmente lo swapping standard è usato in pochi sistemi; esso richiede infatti un elevato tempo di swapping, e consente un tempo di esecuzione troppo breve per poter essere considerato una soluzione ragionevole al problema di gestione della memoria. Versioni modificate di swapping si trovano comunque su molti sistemi.
Una forma modificata di swapping era usata in molte versioni UNIX, lo swapping normalmente era disabilitato e veniva avviato solo nel caso in cui molti processi si fossero trovati in esecuzione utilizzando una quantità limite di memoria. Lo swapping sarebbe stato di nuovo fermato qualora il carico sul sistema fosse diminuito.

8.4 ALLOCAZIONE CONTIGUA.

La memoria centrale deve contenere sia il SO che i vari processi utenti. La memoria è di solito divisa in due partizioni, una per il SO residente e una per i processi utenti. Il SO può essere collocato sia nella memoria bassa che nella memoria alta. Il fattore che incide in modo determinante su tale scelta è generalmente la posizione del vettore di interrupt. Poiché questo si trova spesso nella memoria bassa, generalmente anche il SO viene posto nella memoria bassa.

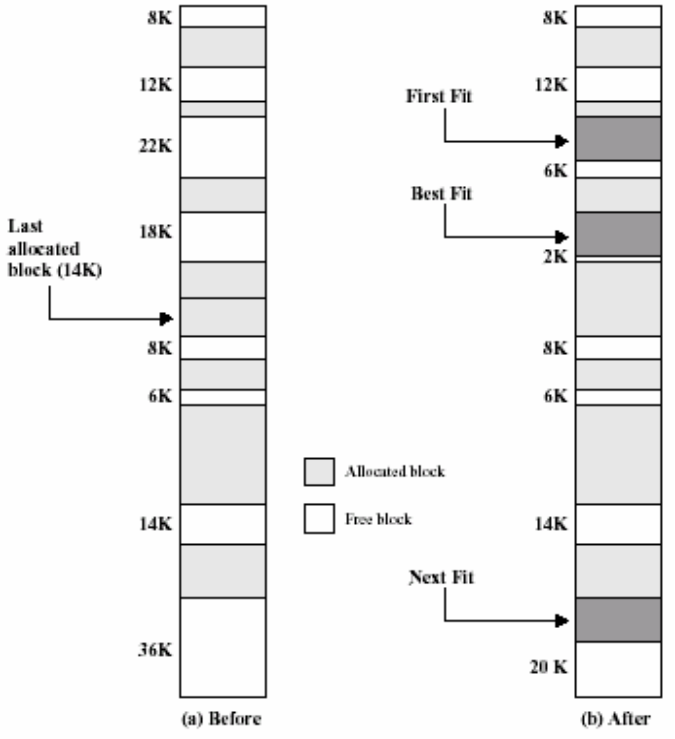
8.4.1 ALLOCAZIONE CON PARTIZIONE SINGOLA

Se il SO risiede nella memoria bassa, e un processo utente è in esecuzione nella memoria alta, occorre proteggere il codice e i dati del SO a modifiche. Tale protezione può essere realizzata utilizzando un registro di rilocazione, con un registro limite. Il registro di rilocazione contiene il valore dell'indirizzo fisico più piccolo; il registro limite contiene l'intervallo di indirizzi logici, ad esempio, rilocazione = 100.040 e limite 74.600. Con i registri di rilocazione e limite, ogni indirizzo logico deve essere minore del contenuto del registro limite; l'MMU mappa dinamicamente l'indirizzo logico sommandogli il valore contenuto nel registro di rilocazione. L'indirizzo risultante viene inviato in memoria.
Quando lo scheduler della cpu seleziona un processo per l'esecuzione, il dispatcher, durante l'esecuzione del context switch, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché ogni indirizzo generato dalla cpu viene confrontato con

questo registri, il SO e i programmi e i dati di altri utenti possono essere protetti da tentativi di modifiche da parte del processo in esecuzione.

8.4.2 ALLOCAZIONE CON PARTIZIONI MULTIPLE

Poiché in generale è utile che più processi utenti siano contemporaneamente residenti in memoria, è necessario considerare il modo in cui la memoria disponibile deve essere allocata ai diversi processi che si trovano nella coda di input in attesa di essere portati in memoria. Uno degli schemi più semplici per l'allocazione della memoria consiste nel dividere la stessa in un certo numero di partizioni di dimensione fissa. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato dal numero di partizioni. Quando una partizione è libera può essere occupata da un processo selezionato nella coda di input; terminato il processo la partizione torna a essere disponibile per un altro processo.
Il SO conserva una tabella nella quale sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un buco (hole). Quando viene caricato un processo che necessita di memoria, occorre cercare un buco sufficientemente grande da contenerlo. Se ne esiste uno viene allocata solo la parte di memoria necessaria al processo; la parte rimanente viene utilizzata per soddisfare eventuali richieste future.
In generale in ogni momento è presente un insieme di buchi di diverse dimensioni sparsi per la memoria. Quando arriva un processo che necessita di memoria, viene ricercato nel gruppo un buco di dimensioni sufficienti a contenerlo. Se il buco è troppo grande viene diviso in due: una parte viene allocata al processo in arrivo e l'altra viene riportata nell'insieme di buchi. Se il nuovo buco si trova accanto ad altri buchi, tutti i buchi vicini vengono uniti per formarne uno più grande. Questa procedura è una particolare istanza del più generale problema di allocazione dinamica della memoria, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le soluzioni sono molteplici. Viene esaminato il gruppo di buchi per stabilire quale sia il buco migliore per effettuare l'allocazione. Le strategie più utilizzate sono:



- First–fit
- Best–fit
- Worst–fit

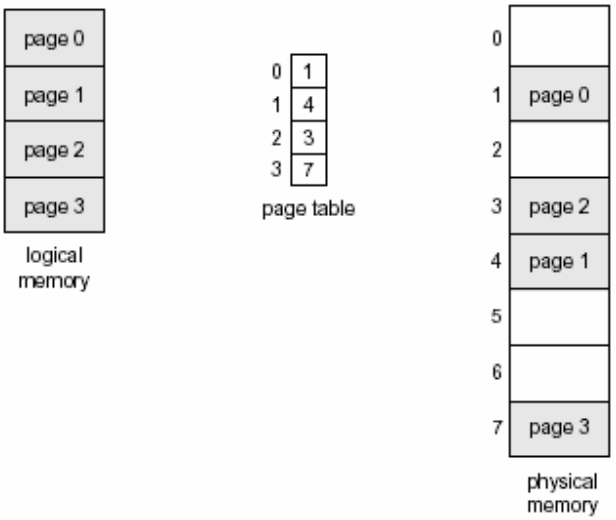
8.4.3 FRAMMENTAZIONE INTERNA ED ESTERNA

Gli algoritmi descritti soffrono di frammentazione esterna. Si ha la frammentazione esterna quando lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria viene frammentata in tanti piccoli buchi.
Questo problema di frammentazione può essere molto grave. Nel caso peggiore può esistere un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutta questa memoria si trovasse su un unico blocco libero di grosse dimensioni, sarebbe possibile eseguire molti più processi. La scelta di first–fit rispetto a best–fit può influire sulla quantità di frammentazione. First–fit risulta migliore per alcuni sistemi, mentre per latri è preferibile best–fit. L'analisi statistica di first–fit, ad esempio, rivela che, pur con una certa ottimizzazione, dati N blocchi allocati, altri 0,5N blocchi allocati vanno perduti a causa della frammentazione, e ciò significa che un terzo della memoria potrebbe risultare inutilizzabile. Questa proprietà è nota come la regole del 50 percento.
Si consideri un buco di 18.464 byte. Supponendo che un processo successivo richieda 18.462, allocando esattamente il blocco richiesto rimane un buco di 2 byte. L'overhead necessario per tenere traccia di questo buco è sostanzialmente più grande del buco stesso. L'approccio generale prevede di allocare buchi molto piccoli come parte di una richiesta più ampia, quindi la memoria allocata può essere leggermente maggiore della memoria richiesta. La frammentazione interna consiste nella differenza tra questi due numeri; la memoria è interna ad una partizione, ma non viene utilizzata.
Una soluzione al problema della frammentazione esterna è data dalla compattazione. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco.
La compattazione non è sempre possibile. Occorre notare che i processi devono venir spostato. Affinché questi processi possano essere eseguiti nelle loro nuove posizioni occorre rilocare tutti gli indirizzi interni. La compattazione non può essere effettuata se la rilocazione è statica e viene effettuata in fase di assemblaggio o di caricamento. La compattazione è possibile solo se la rilocazione è di tipo dinamico e viene eseguita in fase di esecuzione.
Se è possibile effettuare la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria, mentre tutti i buchi vengono spostati nell'altra direzione formando un grosso buco di memoria. Questo schema può essere abbastanza costoso.
Alla compattazione può essere associato anche lo swapping. Un processo può essere estratto dalla memoria centrale e portato sulla backing store, e, in un secondo tempo, può essere riportato nella memoria centrale. Quando il processo viene estratto dalla memoria centrale, la sua memoria viene rilasciata e può essere riutilizzata per un altro processo. Al momento di riportare il processo in memoria centrale possono sorgere parecchi problemi; se viene utilizzata la rilocazione statica, il processo deve essere riportato esattamente sulle stesse posizioni di memoria che occupava in precedenza. Questa restrizione può implicare l'estrazione di altri processi allo scopo di liberare memoria.

8.5 PAGINAZIONE

Un'altra possibile soluzione al problema della frammentazione esterna è data dal consentire la non contiguità dello spazio degli indirizzi logici di un processo,

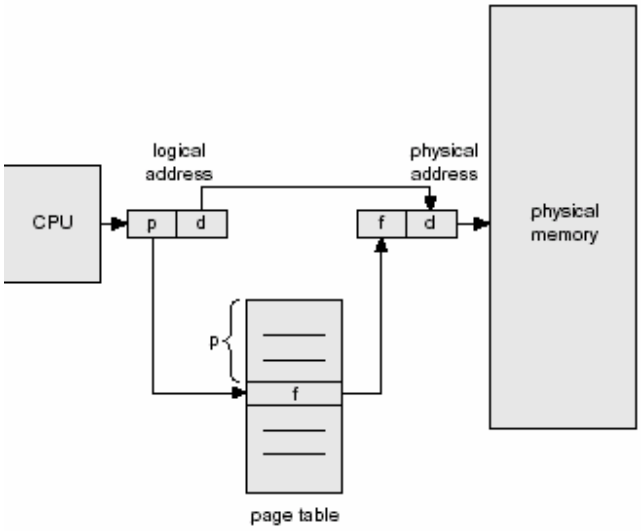
permettendo così di allocare la memoria fisica ai processi ovunque essa sia disponibile. Tale soluzione può essere implementata attraverso lo schema di paginazione. La paginazione elimina il grosso problema della sistemazione di blocchi di memoria di diverse dimensioni nella backing store, un problema che riguarda la maggior parte degli schemi di gestione della memoria precedentemente analizzati. Quando alcuni frammenti di codice o dati residenti in memoria centrale devono essere sottoposti a swap out, deve essere trovato dello spazio nella backing store. I problemi di frammentazione relativi alla memoria centrale valgono anche per la backing store, con la differenza che in questo caso l'accesso è molto più lento, quindi è impossibile effettuare la compattazione. Grazie ai vantaggi offerti rispetto ai metodi precedenti, la paginazione nelle sue varie forme è comunemente usata in molti SO.



8.5.1 METODO DI BASE

La memoria fisica viene suddivisa in blocchi si dimensioni fisse, detti frame, e la memoria logica viene suddivisa in blocchi di uguali dimensione chiamati pagine. Quando un processo deve essere eseguito, le sue pagine vengono caricate nei frame della memoria disponibili dalla backing store, che è divisa in blocchi di dimensione fissa uguale a quella dei frame della memoria.

Ogni indirizzo generato dalla cpu è diviso in due parti: un numero di pagina e un offset di pagina. Il numero di pagina serve come indice per la tabella delle pagine, che contiene l'indirizzo base di ogni pagina nella memoria fisica. Questo indirizzo base viene associato all'offset di pagina per definire l'indirizzo di memoria fisica, che viene inviato all'unità di memoria.



la dimensione di una pagina, come quella di un frame, è definita dall'hardware ed è una potenza di 2, così è più facile la traduzione di un indirizzo logico nel corrispondente numero di pagina e offset di pagina. Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è 2^n unità di indirizzamento (byte o parole), allora gli m-n bit più

significativi indicano l'offset di pagina.
Utilizzando uno schema di paginazione è possibile evitare la frammentazione esterna: qualsiasi frame libero può essere allocato a un processo che ne ha bisogno. Tuttavia, può aver luogo la frammentazione interna. I frame sono allocati come unità. Se i requisiti di memoria di un processo non combaciano con i limiti di pagina, l'ultimo frame allocato può non essere completamente pieno. Se, ad esempio, le pagine sono di 2048 byte, un processo di 72.766 byte necessita di 35 pagine più 1086 byte. Vengono allocati 36 frame, per cui ha luogo la frammentazione interna di 2048-1086 = 962 byte. Il caso peggiore è rappresentato da un processo che necessita di n pagine più un byte. Vengono allocati n+1 frame, per cui ha luogo la frammentazione interna di quasi tutto un frame. Se la dimensione del processo è indipendente dalla dimensione della pagina, ci si deve aspettare una frammentazione di mezza pagina per processo. Questa considerazione suggerisce che conviene utilizzare pagine di piccole dimensioni. Tuttavia, a ogni elemento della tabella delle pagine è associato non poco overhead che può essere ridotto aumentando le dimensioni delle pagine.
Quando un processo arriva per essere eseguito, viene esaminata la sua dimensione stressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede n pagine, in memoria devono essere disponibili almeno n frame, che, se ci sono, vengono allocati al processo stesso. La prima pagina del processo viene caricata su uno dei frame allocati e il numero del frame viene inserito nella tabelle delle pagine relativa al processo in questione. La pagina successiva viene caricata in un altro frame, e, anche in questo caso, il numero del frame viene inserito nella tabella delle pagine, e così via.
Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dall'utente e la memoria fisica effettiva. Il programma utente vede la memoria come un unico spazio contiguo, contenente solo il programma stesso. In realtà, il programma utente è sparso in una memoria fisica che contiene anche altri programmi. La differenza tra la memorialista dall'utente e la memoria fisica effettiva viene ricomposta dall'hardware di traduzione degli indirizzi.
Poiché il SO gestisce la memoria fisica, deve essere informato dei relativi particolari di allocazione: quali frame sono allocati, quali sono disponibili, il numero totale dei frame… In genere queste informazioni sono contenute in una struttura di dati chiamata tabella dei frame, che contiene un elemento per ogni frame di pagina fisica, indicando se il frame è libero oppure è allocato e, se allocato, a quale pagina di quale processo.

8.5.2 Struttura della tabella delle pagine

Ogni SO segue metodi propri per memorizzare le tabelle delle pagine. La maggior parte dei sistemi alloca una tabella delle pagine per ogni processo. Un puntatore alla tabella delle pagine viene memorizzato nel PCB. Quando al dispatcher viene comunicato di avviare un processo, esso deve caricare i registri utente e definire i corretti valori della tabella delle pagine hardware utilizzando la tabella delle pagine, presente in memoria, relativa al processo.

8.5.2.1 SUPPORTO HARDWARE

La realizzazione hardware delle pagine può essere effettuata in modi diversi. Nel caso più semplice, la tabella delle pagine viene implementata come insieme di registri dedicati. L'utilizzo di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, dell'ordine, per esempio, di 256 elemento. La maggior parte dei computer contemporanei utilizza comunque tabelle molto grandi, ad esempio di un milione di elementi. In queste macchine non è possibile adottare registri veloci per implementare la tabella delle pagine; quest'ultima viene conservata nella memoria principale e un suo registro base, chiamato page–table base register (PTBR), che punta alla tabella stessa.
Questo approccio presenta un problema connesso al tempo necessario per accedere a una locazione di memoria utente.
La soluzione standard a questo problema consiste nell'utilizzare una speciale, piccola cache hardware di ricerca veloce, detta anche registri associativi o translation look–asode buffers (TLB). Un insieme di registri associativi è formato da una memoria ad alta velocità. Ogni registro è a sua volta formato da una chiave e un valore.
I registri associativi vengono utilizzati insieme alle tabelle delle pagine nel seguente modo: i registri associativi contengono una piccola parte degli elemento della tabella delle pagine. Quando la cpu genera un indirizzo logico, il suo numero di pagina viene presentato a un insieme di registri associativi contenenti numeri di pagina e i corrispondenti numeri di frame. Quando nei registri viene trovato il numero di pagina, il corrispondente numero di frame risulta immediatamente disponibile e viene utilizzato per accedere alla memoria. Tutta l'operazione può richiedere un tempo inferiore al 10% in più del tempo che sarebbe richiesto per un riferimento di memoria senza mapping.
Se il numero di pagina non è presente nei registri associativi, occorre fare un riferimento di memoria alla tabella delle pagine. Il numero di frame così ottenuto può essere eventualmente utilizzato per accedere alla memoria. Inoltre, inserendo il numero di pagina e di frame nei registri associativi, al riferimento successivo la ricerca sarà molto più veloce. Se il TLB è già pieno di elementi, il SO deve selezionarne uno per rimpiazzarlo. Sfortunatamente, ogni volta che una tabella delle pagine viene selezionata, ad esempio a ogni context switch, il TLB deve essere ripulito (cancellato) in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione. Potrebbero altrimenti esserci dei vecchi elementi del TLB che contengono indirizzi virtuali validi ma corrispondenti a indirizzi fisici sbagliati o non validi.
La percentuale di volte che un numero di pagina viene trovato nei registri associativi viene chiamato hit ratio. Un hit ratio dell'80% significa che il numero di pagina cercato viene trovato nell'80% dei casi. Se la ricerca nei registri associativi richiede 20 nanosecondi e sono necessari 100 nanosecondi per accedere alla memoria, allora un accesso mappato in memoria richiede 120 nanosecondi, supponendo che il numero di magia si trovi nei registri associativi. Se, invece, il numero non è contenuto nei registri associativi (20 nanosecondi), occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero di frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); in totale sono necessari 220 nanosecondi. Per calcolare l'effettivo tempo di accesso alla memoria occorre tenere conto della probabilità in ogni caso:

tempo di accesso effettivo = 0,80*120 + 0,20*220 = 140 nanosecondi

In questo esempio si verifica un rallentamento del 40% nel tempo di accesso in memoria (da 100 a 140 nanosecondi).
Per un hit ratio del 98% si ottiene il seguente risultato:

tempo di accesso effettivo = 0,98*120 + 0,02*220 = 122 nanosecondi

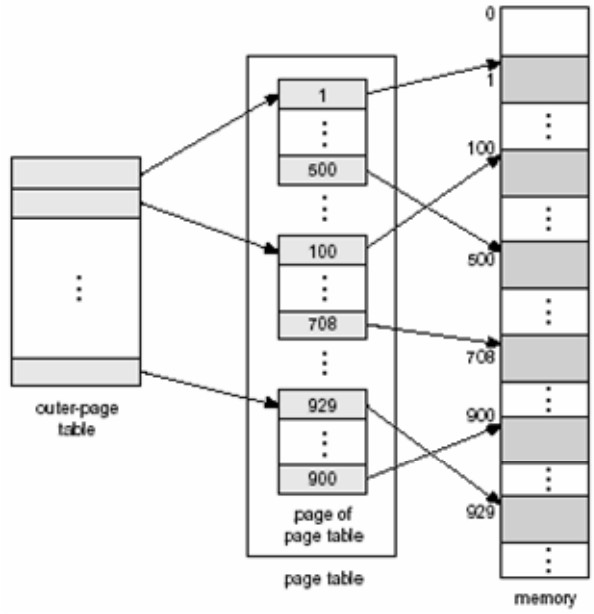
Aumentando l'hit ratio, il rallentamento del tempo di accesso in memoria scende al 22%.

8.5.2.2 PROTEZIONE

In un ambiente paginato, la protezione della memoria viene assicurato dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine. Un bit può determinare se una pagina è di lettura e scrittura oppure di sola lettura. Un tentativo di scrittura su un frame di sola lettura provoca una trap al SO.
Questo approccio alla protezione può essere facilmente esteso per fornire un livello di protezione più perfezionato. L'hardware può essere creato in modo da fornire protezione di sola lettura, di sola scrittura o di sola esecuzione. Inoltre, fornendo bit di protezione separati per ogni tipo di accesso, può essere permessa qualsiasi combinazione di tali accessi.
Un ulteriore bit, detto bit di validità, viene di solito associato a ciascun elemento della tabella delle pagine. Quando tale bit è impostato a vaglio, indica che la pagina associata è nello spazio di indirizzi logici del processo, e quindi è una pagina legale.
Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio di indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina del range di indirizzi, poiché gran parte di questa tabella rimane inutilizzata e occupa del prezioso spazio di memoria. Alcuni sistemi dispongono di registri hardware contenenti la lunghezza della tabella delle pagine (page–table length register PLTR) per indicare la dimensione della tabella. Questo valore dimensionale viene confrontato a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore di questo test causa una trap al SO.

8.5.3 PAGINAZIONE A PIÙ LIVELLI

La maggior parte dei moderni computer supporta uno spazio di indirizzi logici estremamente grande (da 2^{32} a 2^{64}). In un ambiente di questo tipo la stessa tabella delle pagine finirebbe per diventare eccessivamente grande. Si consideri, ad esempio, un sistema con uno spazio di indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4K byte (2^{12}), la tabella delle pagine potrebbe arrivare a contenere fino a un milione di elementi ($2^{32} / 2^{12}$). Poiché ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4MB di spazio di indirizzo fisico solo per la tabella delle pagine. Chiaramente, sarebbe meglio evitare di allocare la tabella delle pagine in modo contiguo in memoria centrale. Una soluzione semplice consiste nel dividere la tabella delle pagine in parti più piccole; questo risultato può essere ottenuto in molti modi differenti.
Un metodo consiste nell'adottare uno schema di paginazione a due livelli, nel quale la stessa tabella delle pagine viene paginata. Per illustrare questo schema, si ritorni al precedente esempio di macchina a 32 bit con dimensione di pagina di 4K byte. Un indirizzo logico viene suddiviso in un numero di pagina composto da 20 bit e un offset di

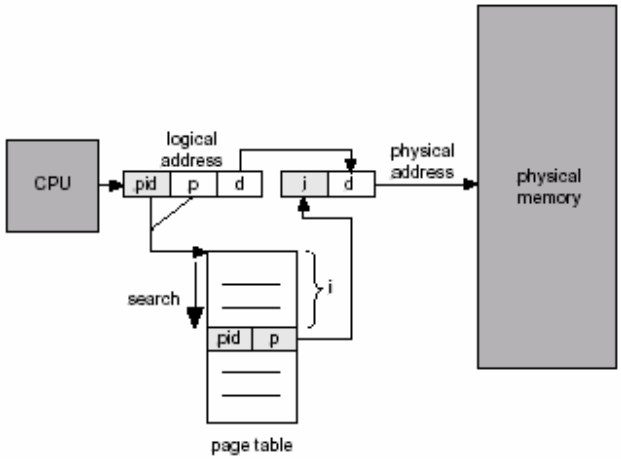


pagina composto da 12 bit. Paginando la tabella delle pagine, anche il numero di pagina viene a sua volta diviso in un numero di pagina di 10 bit e un offset di pagina di 10 bit. .
Lo schema di paginazione a due livelli non è più adatto in caso di sistemi con uno spazio di indirizzi logici a 64 bit. Per illustrare questo aspetto, si supponga che la dimensione delle pagine di questo sistema si di 4K byte (2^{12}). In questo caso, la tabella delle pagine conterrà fino a 2^{52} elementi. Adottando uno schema di paginazione a due livelli, le tabelle interne delle pagine possono convenientemente occupare una pagina, o contenere 2^{10} elementi di 4 byte. La tabella esterna delle pagine consiste di 2^{42} elementi, o 2^{44} byte. La soluzione più ovvia per evitare una tabella delle pagine tanto grande consiste nel dividere la tabella in componenti più piccole. Questo approccio è adottato anche da alcuni processori a 32 bit allo scopo di fornire maggiore flessibilità ed efficienza.
Ciò può essere fatto in vari modi. È possibile paginare la tabella esterna delle pagine, ottenendo uno schema di paginazione a tre livelli. Il passo successivo sarebbe uno schema di paginazione a 4 livelli (Motorola 98030).
In che modo la paginazione a più livelli influenza le prestazioni del sistema? Dato che ciascun livello è memorizzato in una tabella contenuta in memoria, la conversione di un indirizzo logico nel corrispondente indirizzo fisico può richiedere quattro accessi in memoria, quintuplicando il tempo di accesso! Anche in questo caso, però, l'uso delle cache offre dei vantaggi, consentendo di mantenere le prestazioni a un livello ragionevole. Dato un hit rate del 98% si ottiene il seguente risultato:
tempo di accesso effettivo = 0,98*120+0,02*520 = 128 nanosecondi

8.5.4 TABELLA DELLE PAGINE INVERTITA

Uno degli inconvenienti insiti in questo schema è costituito dalla dimensione di ciascuna tabella delle pagine, che può avere milioni di elementi e consumare grandi quantità di memoria fisica, necessaria proprio per sapere come viene utilizzata la rimanente memoria fisica.
Per risolvere questo problema si può far uso della tabella delle pagine invertita. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (frame) di memoria. Ciascun elemento è quindi costituito dall'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste solo una tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però la quantità di tempo necessaria per cercare la tabella quando viene fatto riferimento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzo fisico, mentre le ricerche vengono effettuate per indirizzi virtuali, per cercare una corrispondenza occorre cercare in tutta la tabella, e questa ricerca richiede molto tempo. Per alleviare il problema può essere utilizzata una tabella hash per limitare la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge alla procedura un riferimento in memoria, per cui un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Per migliorare le prestazioni vengono utilizzati registri associativi si memoria che conservano gli elementi cercati recentemente.



8.5.5 PAGINE CONDIVISE

Un altro vantaggio della paginazione consiste nella possibilità di condividere un codice comune. Questa considerazione risulta importante soprattutto in un ambiente con time–sharing. Si consideri un sistema che supporta 40 utenti, ciascuno dei quali esegue un text editor. Se il text editor è formato da 150K di codice e 50K di spazio dati, per supportare 40 utenti sono necessari 8000K. Se però il codice è rientrante può essere condiviso.
Il codice rientrante, detto anche codice puro, è un codice non automodificante, poiché non cambia mai durante l'esecuzione. Quindi due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia di registri e di una memoria dove conserva i dati necessari alla propria esecuzione. I dati per due differenti processi varieranno, ovviamente, per ciascun processo.
Nei sistemi che fanno uso delle tabelle delle pagine invertite, la condivisione della memoria è di più difficile implementazione. Infatti, mentre la memoria condivisa viene di solito implementata mappando due o più indirizzi virtuali sullo stesso indirizzo fisico, tale metodo standard non può essere utilizzato in questo caso, poiché c'è un solo elemento di pagina virtuale per ogni pagina fisica, quindi una pagina fisica non può avere due o più indirizzi virtuali condivisi.

8.6 SEGMENTAZIONE

8.6.1 METODO DI BASE

L'utente potrebbe pensare alla memoria come a un array lineare di byte, alcuni dei quali contengono istruzioni e altri dati, oppure averne un'altra visione.

Il programmatore di un sistema vede la memoria come un insieme di segmenti di dimensione variabile non necessariamente ordinati. Ciascuno di questi moduli o elemento di dati viene identificato da un nome come “tabella dei simboli”, “funzione sqrt”, “programma principale”, indipendentemente dagli indirizzi che questi elementi occupano in memoria. Ciascuno di questi segmenti ha lunghezza variabile, definita intrinsecamente dallo scopo che il segmento stesso ha all’interno del programma. Gli elemento che si trovano all’interno di un segmento sono indicati dal loro offset, misurato a partire dall’inizio del segmento: la prima istruzione del programma, il diciassettesimo elemento della tabella dei simboli…
La segmentazione è uno schema di gestione della memoria che consente di supportare questa visione di memoria dell’utente. Uno spazio di indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificaano sia il nome che l’offset del segmento, quindi l’utente specifica ogni indirizzo indicando due quantità: un nome di segmento e un offset. Questo schema contrasta con la paginazione, nella quale l’utente specificava un singolo indirizzo, che veniva suddiviso dall’hardware in un numero di pagina e un offset non visibili al programmatore.

8.6.2 HARDWARE

Sebbene l’utente possa far riferimento a oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica effettiva rimane ancora una sequenza di byte unidimensionale. Per questo motivo occorre definire un’implementazione per mappare gli indirizzi bidimensionali definiti dall’utente in indirizzi fisici unidimensionali. Questa operazione di mapping viene effettuata da una tabella dei segmenti. Ogni elemento di questa tabella ha una base del segmento e un limite del segmento. La base del segmento contiene l’indirizzo fisico iniziale della memoria nella quale il segmento risiede, mentre il limite del segmento specifica la lunghezza del segmento.
Un indirizzo è formato da due parti: un numero di segmento s e un offset di tale segmento d. Il numero di segmento viene utilizzato come indice per la tabella di segmenti. L’offset d dell’indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti viene effettuato un trap al SO (tentativo di indirizzamento logico oltre la fine del segmento). Se questo offset è legale, viene sommato alla base del segmento per produrre l’indirizzo in memoria fisica del byte desiderato. Quindi la tabella dei segmenti è fondamentalmente una coppia di registri base–limite.

8.6.3 REALIZZAZIONE DELLE TABELLE DEI SEGMENTI

La segmentazione è strettamente connessa ai modelli di gestione della memoria per partizioni: la differenza principale consiste nel fatto che un programma può essere formato da più segmenti.
Come la tabella delle pagine, anche la tabella dei segmenti può essere messa nei registri veloci oppure in memoria. Il riferimento a una tabella dei segmenti contenuta in registri è molto rapido, poiché la somma alla base e il confronto con il limite possono essere effettuati contemporaneamente, consentendo di risparmiare tempo.
Nel caso in cui un programmato sia formato da un elevato numero di segmenti, la tabella dei segmenti non può essere tenuta in registri, per cui viene tenuto in memoria. Un

registro base della tabella dei segmenti (segment–table base register — SBTR) punta alla tabella dei segmenti. Quindi, poiché il numero dei segmenti utilizzati da un programma può variare ampiamente, viene utilizzato un registro di lunghezza della tabella dei segmenti (segmenttabel length register — SLTR). Dato un indirizzo logico (s,d), occorre innanzitutto verificare se il numero di segmento s sia legale, vale a dire s<SBTR. Quindi, sommando il numero di segmento all'SBTR è possibile ottenere l'indirizzo (SBTR+s) di memoria dell'elemento della tabella dei segmenti. Questo elemento viene letto dalla memoria, viene controllato l'offset rispetto alla lunghezza del segmento e, infine, viene calcolato l'indirizzo fisico del byte desiderato sommando la base del segmento con l'offset.
Come nel caso della paginazione, questo tipo di mapping richiede due riferimenti alla memoria per ogni indirizzo logico; questo fatto rallenta il computer di un fattore 2, a meno che il rallentamento non venga impedito. La soluzione normale consiste nell'utilizzare un insieme di registri associativi che conservino gli elementi della tabella dei segmenti utilizzati più di recente. Inoltre, un insieme relativamente piccolo di registri associativi può fare in modo che il tempo necessario per gli accessi in memoria sia più del 10 o 15 percento più lento rispetto agli accessi in memoria non mappati.

8.6.4 PROTEZIONE E CONDIVISIONE

un vantaggio particolare dato dalla segmentazione è l'associazione tra protezione e segmenti. Poiché i segmenti rappresentano una porzione del programma semanticamente definita, è probabile che tutti gli elementi del segmento vengano utilizzati nello stesso modo. Di conseguenza, alcuni segmenti sono istruzioni, mentre altri sono dati. In un architettura moderna le istruzioni sono non automodificanti, quindi i segmenti di istruzione possono essere definiti come di sola lettura o di sola esecuzione. L'hardware di mapping della memoria controlla i bit di protezione associati a ogni elemento della tabella dei segmenti in modo da impedire accessi illegali alla memoria (tentare scrittura su segmento si sola lettura…)
Un altro vantaggio dato dalla segmentazione riguarda la condivisione di codice o di dati. Ogni processo ha una tabella dei segmenti, associata al proprio PCB, che viene utilizzata dal dispatcher per definire la tabella dei segmenti hardware quando al processo viene assegnata la cpu. I segmenti vengono condivisi quando gli elementi delle tabelle dei segmenti di due processi diversi puntano sulle stesse locazioni fisiche. È possibile anche condividere solo parti di programma. Ad esempio, pacchetti di subroutine comuni possono essere condivisi tra molti utenti, ammesso che sia stato definito come segmento condivisibile.

8.6.5 FRAMMENTAZIONE

Lo scheduler a lungo termine deve trovare e allocare memoria per tutti i segmenti di un programma utente. Questa situazione è analoga alla paginazione, con la differenza che i segmenti hanno lunghezza variabile, mentre le pagine hanno tutte la stessa dimensione. Quindi, come accade anche con lo schema di partizioni a dimensione variabile, l'allocazione della memoria diviene un problema di allocazione dinamica, che viene generalmente risolto con l'algoritmo best–fit o first–fit.

La segmentazione può causare anche frammentazione esterna quando tutti i blocchi di memoria libera sono troppo piccoli per contenere un segmento. In questo caso è possibile ritardare il processo fino a che non si renda disponibile più memoria oppure può essere utilizzata la compattazione per creare un buco più grande.
La segmentazione è, per sua natura, un algoritmo di rilocazione dinamico, grazie al quale la memoria può essere compattata in qualsiasi momento. Se lo scheduler della cpu deve attendere un processo a causa di un problema di allocazione di memoria, può esaminare la coda di cpu alla ricerca di un processo da eseguire che sia piccolo e con priorità più bassa.

9 MEMORIA VIRTUALE

La memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente in memoria. Il maggior vantaggio visibile dato da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica. Questa tecnica libera i programmatori da quanto riguarda i limiti di memoria. Però la memoria virtuale è di difficile implementazione e, se usata scorrettamente, può ridurre di molto le prestazioni del sistema.

9.1 INTRODUZIONE

Gli algoritmi di gestione della memoria sono necessari perché, prima di eseguire un processo, le istruzioni che vengono eseguite devono trovarsi all'interno della memoria fisica. Un primo approccio per far fronte a tale requisito consiste nel mettere l'intero spazio di indirizzi logici del processo nella memoria fisica. Gli overlay e il caricamento dinamico possono aiutare ad attenuare tale restrizione, ma generalmente richiedono particolari precauzioni e uno sforzo aggiuntivo da parte del programmatore.
In molti casi non è necessario tutto il programma, ad esempio

- Spesso i programmi dispongono di codice per gestire condizioni di errore insolito. Dal momento che questi errori sono rari, se non inesistenti, anche il codice relativo non verrà mai eseguito.
- Spesso ad array, liste e tabelle viene allocata più memoria di quanto ne sia effettivamente necessaria.
- Alcune opzioni e caratteristiche di un programma possono essere utilizzate solo di rado.

Anche nei casi di cui è necessario disporre di tutto il programma è possibile che non serva tutto in una volta. La possibilità di eseguire un programma che si trova solo parzialmente in memoria può essere vantaggiosa per i seguenti motivi:

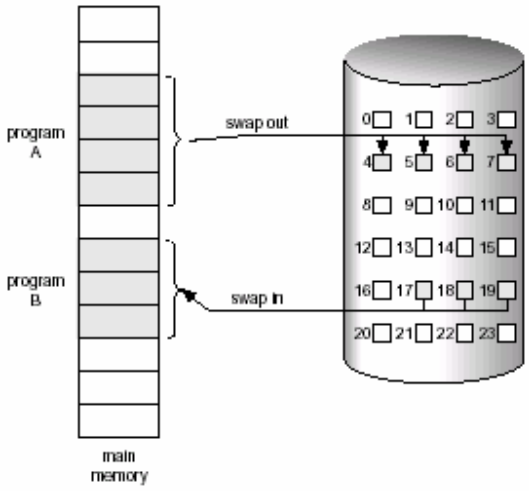
- Un programma non è più vincolato dalla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno spazio di indirizzi virtuali molto grande.
- Poiché ogni utente utilizza meno memoria fisica, possono essere eseguiti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e del throughput della cpu senza aumentare il tempo di risposta o di turnaround

- Per caricare o sottoporre a swapping ogni programma utente in memoria sono necessari meno I/O, per cui ogni programma utente viene eseguito più rapidamente.

La memoria virtuale rappresenta la separazione tra la memoria logica dell'utente e la memoria fisica. Questa separazione permette di offrire ai programmatori una memoria virtuale estremamente ampia anche se la memoria fisica disponibile è piccola.
La memoria virtuale viene generalmente implementata tramite la paginazione su richiesta, ma può essere implementata anche in un sistema di segmentazione. Parecchi sistemi offrono uno schema di segmentazione paginata, dove i segmenti sono spezzati in pagine. Quindi l'utente vede la segmentazione, ma il SO può implementare questa visione con paginazione su richiesta. Per ottenere una memoria virtuale può essere utilizzata anche una segmentazione su richiesta.

9.2 PAGINAZIONE SU RICHIESTA

Un sistema di paginazione su richiesta è analogo a un sistema paginato con swapping. I processi risiedono nella memoria secondaria, generalmente costituita da un disco. Per eseguire un processo occorre effettuare uno swap in memoria del processo, Tuttavia, anziché effettuare lo swapping in memoria di tutto un processo, è possibile utilizzare uno swapper pigro, il quale non riversa mai una pagina in memoria a meno che quella pagina non sia necessaria. Uno swapper manipola processi interi, mentre un pager ha a che fare con le singole pagine di un processo. Quindi, in questo testo invece del termine swapper, parlando di paginazione su richiesta, viene utilizzato il termine pager.
Quando un processo sta per essere sottoposto a swap in, il pager fa una predizione delle pagine che saranno utilizzate, prima che il processo venga nuovamente sottoposto a swap out. Anziché sottoporre tutto il processo allo swap in, il pager trasferisce in memoria solo le pagine che ritiene necessarie. In questo modo è possibile evitare la lettura in memoria di pagine che non vengono comunque utilizzate, riducendo il tempo di swap e la quantità di memoria fisica richiesta.



Con tale schema è necessaria qualche forma di supporto hardware che consenta di distinguere le pagine in memoria da quelle su disco. A tal fine può essere utilizzato lo schema con bit di validità. In questo caso, però, se il bit è impostato a valido significa che la pagina associata è legale ed è presente in memoria. Se il bit è impostato come non valido indica che la pagina non è valida oppure è valida ma è attualmente su disco. L'elemento della tabella delle pagine di una pagina portata in memoria viene impostata come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria viene semplicemente contrassegnato come non valido o contenente l'indirizzo della pagina su disco.

Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il programma non tenta mai di accedere a quella pagina. Quindi. Se la predizione del pager è esatta e vengono caricate tutte e sole le pagine che servono effettivamente, il processo viene eseguito come se fossero caricate tutte le pagine. Durante l'esecuzione il processo accede alle pagine residenti in memoria, e l'esecuzione procede come al solito.
Se il processo tenta di utilizzare una pagina che non era stata portata in memoria, l'accesso a una pagine contrassegnata come non valida causa una trap di page–fault. L'hardware di paginazione, traducendo l'indirizzo attraverso al tabella della pagine, nota che il bit è non valido e causa un trap al SO; tale trap è dovuto a un insuccesso del SO nel portare la pagina desiderata in memoria, piuttosto che a un indirizzo non valido risultante dal tentativo di accedere a un indirizzo di memoria illegale. A tale insuccesso si deve quindi rimediare.
La procedura di gestione del page fault è semplice:

1. Viene controllata una tabella interna per questo processo; in genere tale tabella viene conservata insieme al PCB, allo scopo di stabilire se il riferimento fosse un accesso in memoria valido o non valido
2. Se il riferimento non era valido, il processo viene terminato. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, ne viene effettuato l'inserimento
3. Viene individuato un frame libero, ad esempio prelevandone uno dall'elenco dei frame liberi
4. Viene programmata un'operazione su disco per leggere la pagina desiderata nel frame appena allocato
5. Quando la lettura su disco è completata, vengono modificate la tabella interna conservata con il processo e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
6. Viene riavviata l'istruzione che era stata interrotta dal trap di indirizzo illegale. A questo punto il processo può accedere alla pagina come se questa fosse sempre stata in memoria.

È addirittura possibile avviare l'esecuzione di un processo senza pagine in memoria. Quando il SO carica nel program counter l'indirizzo della prima istruzione del processo, che è su una pagina non residente in memoria, il processo continua l'esecuzione, causando page fault fino a che tutte le pagine necessarie non si trovano in memoria.
In teoria alcuni programmi possono accedere a più pagine di memoria all'esecuzione di ogni istruzione, causando più possibili page fault per ogni istruzione. Un'analisi dei processi in esecuzione mostra che questo comportamento è altamente improbabile: i programmi tendono ad avere località di riferimento.
L'hardware di supporto per la paginazione su richiesta è quello richiesto per paginazione e swapping:

- Tabella delle pagine: ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione
- Memoria secondaria: conserva le pagine che non sono presenti nella memoria centrale. La sezione del disco utilizzata q questo scopo è nota come spazio di swap o backing store.

È necessario imporre ulteriori vincoli architettonici. Un fattore cruciale riguarda la necessità di riavviare qualsiasi istruzione dopo un page fault; nella maggior parte dei casi

questo requisito viene soddisfatto facilmente. Un page fault può verificarsi su qualsiasi riferimento di memoria. Se il page fault si presenta durante la fase di prelievo di un'istruzione, l'esecuzione può essere riavviata prelevando nuovamente l'istruzione. Se il page fault si verifica durante il prelievo di un istruzione, l'esecuzione può essere riavviata prelevando nuovamente l'istruzione. Se il page fault si verifica durante il prelievo di un operando, l'istruzione deve essere prelevato di nuovo, decodificata e quindi può essere prelevato l'operando.

9.3 PRESTAZIONI DELLA PAGINAZIONE SU RICHIESTA

La pagina su richiesta può avere un effetto significativo sulle prestazioni di un computer. Il motivo può essere compreso calcolando il tempo di accesso effettivo per una memoria con paginazione su richiesta. Attualmente, nella maggior parte dei computer il tempo di accesso in memoria, 'ma', varia da 10 a 200 nanosecondi. Fino a che no si verificano page fault, il tempo di accesso effettivo è uguale al tempo di accesso in memoria. Se però ha luogo un page fault, occorre prima leggere dal disco la pagina interessata e quindi accedere alla parola desiderata.
Supponendo che p sia la probabilità che si verifichi un page fault ($0 \leq p \geq 1$), è probabile che p sia molto vicina a zero, cioè che ci saranno solo pochi page fault. Il tempo di accesso effettivo è dato dalla seguente espressione:

Tempo di accesso effettivo = (1 – p) * ma + p * tempo di page fault

In presenza di un page fault viene eseguita la seguente sequenza.

1. Trap al SO
2. Salvataggio dei registri utente e dello stato del processo
3. Determinazione della natura di page fault dell'interrupt
4. Controllo della legalità del riferimento alla pagina e determinazione della locazione della pagine su disco.
5. Effettuazione di una lettura dal disco a un frame libero
    a. Attesa nella coda associata a questo dispositivo fino a che la richiesta di lettura non viene servita
    b. Attesa del tempo di riposizionamento e/o latenza del dispositivo
    c. Inizio del trasferimento dalla pagina a un frame libero
6. Durante l'attesa, allocazione della cpu a un altro utente (scheduling della cpu, facoltativo)
7. Interrupt dal disco (I/O completato)
8. Salvataggio dei registri e dello stato del processo dell'altro utente (se è stato eseguito il passo 6).
9. Determinazione della provenienza dell'interrupt dal disco
10. Correzione della tabella delle pagine e di altre tabelle per segnalare che la pagina desiderata è attualmente presente in memoria
11. Attesa che la cpu venga nuovamente allocata a questo processo
12. Recupero dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell'istruzione interrotta.

Considerando un tempo medio di servizio di page fault di 25 millisecondi e un tempo di accesso in memoria di 100 nanosecondi, il tempo effettivo di accesso in nanosecondi è:

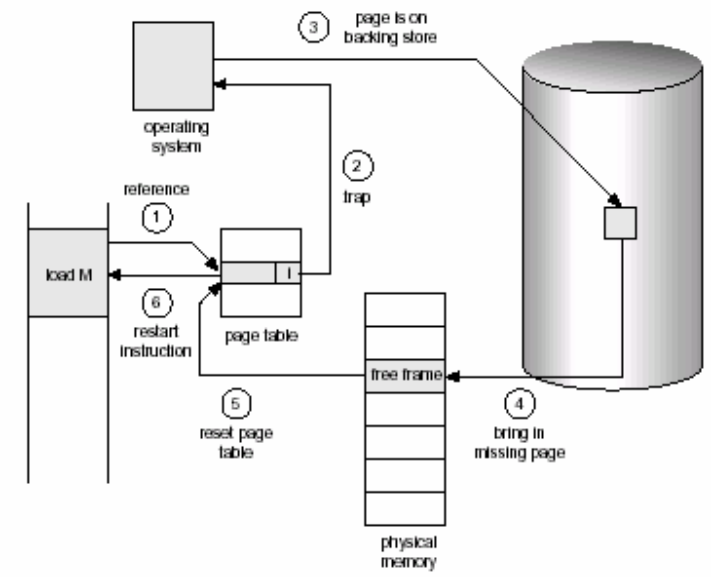
tempo di accesso effettivo = (1 – p) * 100 + p * (25 millisecondi) =

$$= (1-p) * 100 + p * 25.000.000 = 100 + 24.999.900*p$$

Il tempo di accesso effettivo è direttamente proporzionale alla frequenza di page fault. Se un accesso su 1000 causa un page fault, il tempo di accesso effettivo è di 25 microsecondi. Utilizzando la paginazione su richiesta il computer viene rallentato di un fattore pari a 250. Se si desidera un rallentamento inferiore al 10 %:
$110 > 100+25.000.000*p \rightarrow p < 0,0000004$
cioè per mantenere a un livello ragionevole il rallentamento dovuto alla paginazione, può essere permesso meno di un page fault ogni 2.500.000 accessi in memoria

9.4 SOSTITUZIONE DELLE PAGINE

Nelle descrizioni effettuate finora, la frequenza di page fault non costituiva un problema grave, in quanto ogni pagina era sottoposta a fault al massimo una volta, e precisamente la prima volta in cui veniva effettuato un riferimento ad essa. Tale rappresentazione non è molto precisa. Va precisato il fatto che se un processo di 10 pagine ne utilizza effettivamente solo la metà, la paginazione su richiesta fa risparmiare l'I/O necessario per caricare le cinque pagine che non vengono mai utilizzate. Il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, potrebbero essere eseguiti otto processi anziché i quattro che verrebbero usati se ciascuno di essi richiedesse 10 frame, cinque dei quali non verrebbero mai usati.
Aumentando il grado di multiprogrammazione, la memoria viene sovrallocata. Eseguendo sei processi, utilizzo e throughput della cpu verrebbero comunque elevati e si risparmierebbero anche 10 frame. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisa necessità di usare tutte le 10 pagine, per cui sarebbero necessari 60 frame mentre ne sono disponibili solo 40. La probabilità di una situazione di questo tipo aumenta con il livello di multiprogrammazione, di modo che la quantità media della memoria utilizzata si avvicina alla memoria fisica disponibile.
La sovrallocazione può essere illustrata come segue. Durante l'esecuzione di un processo utente si verifica un page fault. L'hardware esegue una trap al SO, il quale controlla le sue tabelle interne per controllare che si tratti di un page fault e non di accesso illegale in memoria. Il SO determina il punto in cui la pagina desiderata risiede sul disco, ma poi scopre che l'elenco dei frame liberi è vuoto: tutta la memoria è utilizzata.



Si può sottoporre il processo a swap out, liberando tutti i suoi frame e riducendo il livello di multiprogrammazione. Ma si può anche sostituire le pagine: la sostituzione delle pagine è basata sul fatto che se nessun frame è libero è possibile liberarne uno attualmente inutilizzato. Un frame può essere liberato scrivendo il suo contenuto nello spazio di swap e modificando la tabella delle

pagine per indicare che la pagina non si trova più in memoria. Il frame liberato può essere utilizzato per conservare in memoria la pagina che ha causato il fault. La routine di servizio di page fault viene modificata in modo da includere la sostituzione della pagina:

1. Individuazione della locazione della pagina sul disco
2. Individuazione di un frame libero
    a. Se esiste un frame libero, esso viene utilizzato
    b. Altrimenti viene utilizzato un algoritmo di sostituzione pagina per selezionare un frame vittima
    c. La pagina vittima viene scritta sul disco; le tabelle delle pagine e dei frame vengono modificate conformemente
3. Lettura della pagina richiesta nel frame appena liberato; modifica delle tabelle delle pagine e dei frame
4. Riavvio del processo utente

Occorre notare che se non esiste alcun frame libero sono necessari due trasferimenti di pagina, uno all'esterno e uno all'interno. Questa situazione raddoppia il tempo di servizio di page fault e aumenta di conseguenza anche il tempo effettivo di accesso. Questo overhead può essere ridotto utilizzando un bit di modifica (dirty bit). Ogni pagina (o frame) può avere associato un bit di modifica nell'hardware. Il bit di modifica per una pagina è impostato dall'hardware ogni volta che nella pagina viene scritto un byte, cosicché, se si decide di sostituire la pagina, non occorre salvarla perché è esattamente identica alla copia già presente su disco.
Per implementare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un algoritmo di allocazione dei frame e un algoritmo di sostituzione delle pagine. Se in memoria sono presenti più processi, occorre decidere quanti frame debbano essere allocati a ciascun processo. Inoltre, quando è richiesta una sostituzione di una pagina occorre selezionare i frame da sostituire. La costruzione di algoritmi idonei a risolvere questi problemi è un compito importante, in quanto l'I/O su disco è piuttosto costoso.

9.5 ALGORITMI DI SOSTITUZIONE DELLE PAGINE

Una algoritmo viene valutato effettuandone l'esecuzione su una particolare stringa di riferimenti alla memoria e calcolando il numero di page fault. La stringa dei riferimenti alla memoria è chiamata stringa dei riferimenti. Queste stringhe possono essere generate artificialmente oppure utilizzando un dato sistema e registrandone l'indirizzo di ciascun riferimento alla memoria. Quest'ultima opzione permette di ottenere un numero elevato di dati, dell'ordine del milione di indirizzi al secondo. Per effettuare la riduzione di questa quantità di dati, occorre notare due cose.
Innanzitutto, di una pagina di date dimensioni, generalmente fissate dall'hardware del sistema, viene considerato solo il numero della pagina e non l'indirizzo intero. In secondo luogo, quando viene effettuato un riferimento a una pagina p, tutti i riferimenti immediatamente successivi non subiscono fault. Ad esempio, esaminando un processo particolare può essere registrata la seguente sequenza di indirizzi:
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611… che, a 100 byte per pagina, viene ridotta alla seguente stringa di riferimenti: 1,4,1,6,1,6… Per stabilire il numero di page fault relativo a una particolare stringa di riferimenti a un particolare algoritmo di

sostituzione delle pagine, occorre conoscere anche il numero di frame disponibili. Per illustrare gli algoritmi di sostituzione delle pagine viene utilizzata la seguente stringa dei riferimenti:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

per una memoria con tre frame.

9.5.1 Algoritmo FIFO

Questo algoritmo associa a ogni pagina l'ora in cui una pagina è stata portata in memoria. Se una pagina deve essere sostituita, viene selezionata la pagina più vecchia. Nella stringa dei riferimenti adottata, i tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) causano un page fault e vengono portati nei frame vuoti. Il riferimento successivo (2) sostituisce la pagina 7, perché essa era stata portata per prima in memoria…

Per illustrare i problemi che possono insorgere con l'uso dell'algoritmo di sostituzione delle pagine FIFO, si prenda in considerazione la seguente stringa dei riferimenti:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Occorre notare che il numero di fault (10) per quattro frame è maggiore del numero di fault (9) per tre frame. Questo risultato è alquanto indesiderato ed è conosciuto con il nome di anomalia di Belady. Questa anomalia riflette il fatto che in alcuni algoritmi di sostituzione delle pagine la frequenza di page fault può aumentare con l'aumentare del numero dei frame allocati.

9.5.2 Algoritmo ottimale

A seguito della scoperta dell'anomalia di Belady, la ricerca è stata diretta verso un algoritmo ottimale di sostituzione delle pagine. Tale algoritmo è quello che tra tutti gli algoritmi presenta la minima frequenza di page fault e non presenta mai l'anomalia di Belady. In effetti esiste un algoritmo ottimale chiamato OPT o MIN: Sostituisce la pagina che non verrà utilizzata per il periodo di tempo più lungo.

L'algoritmo ottimale di sostituzione delle pagine è difficile da implementare, in quanto richiede la conoscenza futura della stringa di riferimenti. Quindi l'algoritmo ottimale viene utilizzato soprattutto per studi comparativi. Ad esempio, può risultare abbastanza utile sapere che sebbene un algoritmo nuovo non sia ottimale, nel peggiore dei casi è inferiore del 12,3 % rispetto a quella dell'algoritmo ottimale, e mediamente questa percentuale è del 4,7%.

9.5.3 Algoritmo LRU

Utilizzando come approssimazione di un futuro vicino un passato recente, allora viene sostituita la pagina che non è stata usata per il periodo di tempo più lungo. L'approccio appena descritto è l'algoritmo LRU, last recently used. La sostituzione LRU associa a ogni pagina l'ora in cui è stata utilizzata l'ultima volta. L'algoritmo LRU produce 12 fault (i primi 5 sono gli stessi della sostituzione ottimale).

La politica LRU viene spesso utilizzata come algoritmo di sostituzione delle pagine ed è considerata abbastanza valida. Il problema principale riguarda l'implementazione della

sostituzione stessa. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza hardware. Il problema consiste nel determinare un ordine per i frame definito in base la momento dell'ultimo utilizzo. Sono realizzabili le due seguenti soluzioni:

1. Contatori. Nel caso più semplice, a ogni elemento della tabella delle pagine viene associato un campo dell'ora di utilizzo, e alla cpu viene aggiunto un clock logico, o contatore. Il clock viene incrementato per ogni riferimento di memoria. Ogni volta che viene effettuato un riferimento a una pagina, il contenuto del registro clock viene copiato sul campo dell'ora di utilizzo nella tabella relativa a quella pagina specificata. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina meno recentemente usata, e una scrittura in memoria per ogni accesso in memoria, nel campo dell'ora di utilizzo della tabella delle pagine.
2. Stack. Un altro approccio all'implementazione della sostituzione LRU prevede la presenza di uno stack di numeri di pagine. Ogni volta viene effettuato un riferimento a una pagina, essa viene presa e portata in cima allo stack. In questo modo in cima allo stack si trova sempre la pagina utilizzata per ultima, mentre in fondo si trova la pagina LRU. Poiché alcuni elementi devono essere rimossi dal centro dello stack, la migliore implementazione viene effettuata utilizzando una lista doppiamente concatenata, con un puntatore sull'elemento iniziale e uno su quello finale. Togliendo una pagina dallo stack e mettendola in cima allo stack, nel caso peggiore è necessario modificare sei puntatori.

Né la sostituzione ottimale né quella LRU sono soggette all'anomalia di Belady. Esiste una classe di algoritmi di sostituzione delle pagine, chiamati algoritmi di stack, che non può mai presentare l'anomalia di Belady. Un algoritmo di stack è un algoritmo per cui è possibile mostrare che l'insieme delle pagine in memoria per n frame è sempre un sottoinsieme dell'insieme delle pagine che dovrebbero essere in memoria per n+1 frame. Per la sostituzione LRU, l'insieme di pagine in memoria è formato dalle n pagine per cui è stato fatto riferimento più recentemente. Se il numero di frame è aumentato, queste n pagine continuano a essere quelle a cui è stato fatto riferimento più recentemente e quindi restano in memoria.
Senza il supporto hardware, oltre ai registri TLB standard sarebbe inconcepibile anche l'implementazione della soluzione LRU. La modifica dei campi di clock o dello stack deve essere effettuata per ogni riferimento in memoria.
Se per ogni riferimento fosse necessario effettuare un interrupt per permettere al software di modificare tali strutture dati, tutti i riferimenti di memoria verrebbero rallentati di un fattore pari a 10. Sono pochi i sistemi che possono permettersi tale livello di overhead per la gestione della memoria.

9.5.4 Algoritmo di approssimazione a LRU

Nei sistemi che non offrono il supporto hardware devono essere utilizzati altri algoritmo di sostituzione delle pagine, come ad esempio l'algoritmo FIFO. Molti sistemi possono fornire un aiuto: un bit di riferimento. Il bit di riferimento per una pagina viene impostato dall'hardware ogni volta che viene fatto riferimento a quella pagina, che sia una lettura o

una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ogni elemento della tabella delle pagine.
Inizialmente, tutti i bit sono azzerati dal SO. Quando un processo utente entra in esecuzione, il bit associato a ciascuna pagina a cui viene fatto riferimento viene impostato a 1 dall'hardware. Dopo qualche tempo è possibile stabilire quali pagine siano state utilizzate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'ordine di utilizzo. Queste informazioni parziali sull'ordinamento portato ad alcuni algoritmi di sostituzione che approssimano una sostituzione LRU.

9.5.4.1 ALGORITMO CON BIT SUPPLEMENTARE DI RIFERIMENTO

Ulteriori informazioni sull'ordinamento possono essere ottenute registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria un byte per ogni pagina. A intervalli regolari, per esempio 100 millisecondi, un interrupt del timer trasferisce il controllo al SO. Questo sposta il bit di riferimento per ciascuna pagina nel bit più significativo del suo byte da 8 bit, spostando il bit di riferimento per ciascuna pagina nel bit più significativo del suo byte, spostando gli altri bit a destra di 1 bit e scartando il bit meno significativo. Questi registri a scorrimento di 8 bit contengono la storia dell'utilizzo delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene 00000000, significa che la pagina non è stata utilizzata da otto periodi di tempo; una pagina che viene utilizzata almeno una volta per ogni periodo di tempo ha un valore nel registro a scorrimento 11111111.
Una pagina con valore di 11000100 nel registro è stata usata più recentemente non lo sia stata una con valore 01110111. Interpretando questi byte di 8 bit come interi senza segno, segue che la pagina con il numero più basso è la pagina LRU e può essere sostituita. Il numero dei bit di storia può essere naturalmente variato: viene selezionato a seconda dell'hardware disponibile per velocizzare al massimo la modifica. Nel caso limite, tale numero può essere ridotto a zero, lasciando soltanto il bit di riferimento. Questo algoritmo è chiamato algoritmo di sostituzione delle pagine seconda chance.

9.5.4.2 ALGORITMO SECONDA CHANCE

L'algoritmo di base per la sostituzione seconda chance è un algoritmo di sostituzione FIFO. Dopo aver selezionato una pagina viene comunque controllato il bit di riferimento. Se il suo valore è 0, la pagina viene sostituita; se il bit di riferimento è 1, viene data una seconda chance alla pagina e la selezione passa alla successiva pagina FIFO. Quando una pagina riceve la seconda chance, il suo bit di riferimento viene azzerato e il suo tempo di arrivo viene aggiornato al tempo attuale. Così, una pagina a cui viene offerta una seconda chance non viene mai sostituita fino a che tutte le altre pagine non vengono sostituite, oppure non sia stata attribuita loro una seconda chance. Inoltre, se una pagina viene utilizzata abbastanza spesso, in modo che il suo bit sia sempre impostato, quella pagina non viene mai sostituita.
Un metodo per implementare l'algoritmo seconda chance, detto anche clock, è dato da una coda circolare. Un puntatore indica quale sia la prima pagina da sostituire. Quando serve un frame, il puntatore avanza fino a che non trova una pagina con bit di riferimento a 0. Avanzando, azzera tutti i bit di riferimento. Una volta trovata una pagina vittima,

essa viene sostituita e la nuova pagine viene inserita nella coda circolare nella posizione corrispondente. Si noti che nel caso peggiore, quando tutti i bit sono impostati a uno, il puntatore esegue un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit di sono a uno, la sostituzione seconda chance degenera in sostituzione FIFO.

9.5.4.3 ALGORITMO SECONDA CHANCE MIGLIORATO

L'algoritmo seconda chance descritto precedentemente può essere migliorato considerando il bit di riferimento e il bit di modifica come una coppia ordinata:

1. (0,0) né recentemente utilizzato né modificato — migliore pagina da sostituire
2. (0,1) non utilizzato recentemente, ma modificato
3. (1,0) utilizzato recentemente ma non modificato
4. (1,1) recentemente utilizzato e modificato.

Questo algoritmo è usato nello schema di gestione della memoria virtuale dell'Apple Macintosh. La differenza principale tra questo algoritmo e il più semplice algoritmo clock è che nel primo viene data la preferenza alle pagina che sono state modificate, al fine di ridurre il numero di I/O richiesti.

9.5.5 ALGORITMI CON CONTEGGIO

- LFU, least frequently used richiede che venga sostituita la pagina con il conteggio più basso. Il punto debole è che se una pagina viene utilizzata molto inizialmente, ma poi non viene riutilizzata, non viene sostituita.
- MFU, most frequently used. È basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è ancora stata utilizzata.

9.5.6 ALGORITMI CON BUFFERING DELLE PAGINE

Oltre a uno specifico algoritmo, per la sostituzione delle pagine vengono spesso utilizzate anche altre procedure. Ad esempio, i sistemi hanno generalmente un pool di frame liberi. Quando si verifica un page fault, per prima cosa viene scelto un frame vittima, ma prima che la vittima venga registrata in uscita, la pagina richiesta viene letta su un frame libero del pool. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima viene registrata in uscita. Quando la vittima viene registrata in uscita, il suo frame viene aggiunto al pool dei frame liberi.
Questa idea può essere ampliata conservando un elenco delle pagine modificate. Ogniqualvolta il dispositivo di paginazione è inattivo, una pagina modificata viene selezionata, scritta sul disco e il suo bit di modifica viene reimpostato. Questo schema aumenta la probabilità che, al momento della selezione per la sostituzione, la pagina non abbia subito modifiche e non debba essere registrata in uscita.
Anche un'altra modifica prevede l'utilizzo di un pool di frame liberi, ma, in questo caso, occorre ricordare la pagina contenuta in ogni frame. Dal momento che il contenuto dei frame non viene modificato quando un frame viene scritto su disco, in caso di necessità la pagina vecchia può essere riutilizzata direttamente dal pool di frame liberi prima che quel frame venga riutilizzato. In questo caso non è necessario alcun I/O. Se si verifica un page

fault occorre controllare se la pagina richiesta si trovi nel pool di frame liberi; se non c'è occorre selezionare un frame libero e leggervi la pagina.
Questa tecnica, insieme all'algoritmo di sostituzione FIFO, è quella utilizzata dal sistema VAX/VMS. Quando l'algoritmo FIFO sostituisce per errore una pagina che è ancora in uso, la pagina viene richiamata velocemente dal buffer dei frame liberi.

9.6 ALLOCAZIONE DEI FRAME

A questo punto occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, è possibile considerare il caso nel quale 93 frame liberi debbano essere assegnati a due processi. Il caso più semplice di memoria virtuale è il sistema con utente singolo con 128K di memoria, formato da pagina di dimensione di 1K. In tutto sono presenti 128 frame. Il SO può occupare 35K, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 frame vengono inizialmente posti sull'elenco dei frame liberi. Quando un processo utente inizia l'esecuzione, genera una sequenza di page fault. I primi 93 page fault ricevono i frame liberi dall'elenco. Una volta esaurito l'elenco, per stabilire quale delle 93 pagine presenti in memoria debba essere sostituita con la novantaquattresima, può essere utilizzato un algoritmo di sostituzione delle pagine. Terminato il processo, i 93 frame vengono posti di nuovo nell'elenco dei frame liberi.
Un altro problema nasce quando la paginazione su richiesta è associata alla multiprogrammazione. Infatti, questa mette in memoria due o più processi contemporaneamente.

9.6.1 NUMERO MINIMO DI FRAME

Naturalmente le strategie di allocazione dei frame hanno parecchi vincoli. Non possono essere allocati più frame di quanti ne siano disponibili, a meno che non vi sia condivisione di pagine. Inoltre, esiste anche un numero minimo di frame che possono essere allocati. Naturalmente, col diminuire del numero dei frame allocati a ciascun processo aumento la frequenza di page fault, con conseguente rallentamento dell'esecuzione del processo.
Il numero minimo di frame allocabili è definito dall'architettura del set di istruzioni. Occorre ricordare che quando ha luogo un page fault prima che sia completata l'esecuzione dell'istruzione, quest'ultima deve essere riavviata. Di conseguenza, i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine a cui ogni singola istruzione può fare riferimento. Il numero minimo di frame è definito dall'architettura, mentre il numero massimo dalla quantità di memoria fisica disponibile.

9.6.2 ALGORITMI DI ALLOCAZIONE

Il modo più semplice per dividere m frame tra n processi è quello per cui a ciascuno viene data una fetta uguale, m/n frame. Ad esempio, se ci sono 93 frame e cinque processi, ogni processo riceve 18 frame. I tre frame lasciati liberi potrebbero essere utilizzati come buffer di frame liberi. Questo schema è chiamato allocazione uguale.

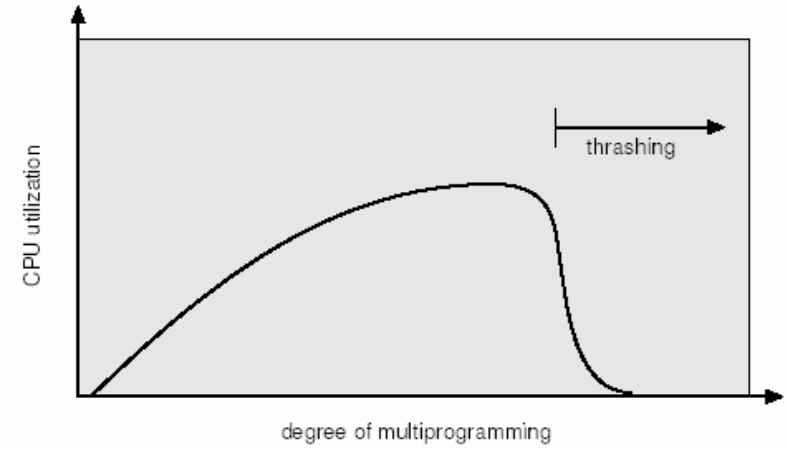
Un'alternativa consiste nel riconoscere che vari processi hanno bisogno di quantità di memoria diverse. Se un piccolo programma utente da 10K e un database interattivo da 127K sono gli unici processi in esecuzione su un sistema con 62 frame liberi, non ha senso assegnare a ciascun processo 31 frame. Al processo utente non ne servono più di 10, per cui gli altri 21 sarebbero semplicemente sprecati.
Per risolvere questo problema è possibile ricorre all'allocazione proporzionale, per cui la memoria disponibile viene allocata a ciascun processo in base alla sua dimensione.
Sia nell'allocazione uguale che in quella proporzionale, l'allocazione a ogni processo può variare in base al livello di multiprogrammazione. Se il livello di multiprogrammazione viene aumentato, ciascun processo perde alcuni frame per fornire la memoria necessaria per il processo nuovo. D'altra parte, se il livello di multiprogrammazione diminuisce, i frame che erano stati allocato al processo allontanato possono essere allocati tra i processi che restano.
Occorre notare che con l'allocazione uguale o con l'allocazione proporzionale, un processo viene trattato come un processo con priorità bassa anche se, per definizione, si vorrebbe che al processo con elevata priorità venisse attribuita più memoria per accelerarne l'esecuzione, a discapito dei processi a bassa priorità.,
Un approccio a tale problema prevede l'utilizzo di uno schema di allocazione proporzionale in cui iol rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità (o da tutte e due).

9.6.3 Allocazione globale e allocazione locale.

Un altro importante fattore riguardante il modo in cui i frame sono allocati ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine possono essere classificati in due grandi categorie: sostituzione globale e sostituzione locale. La sostituzione globale permette a un processo di selezionare un frame per la sostituzione per la sostituzione dall'insieme di tutti i frame, anche se quel frame è attualmente allocato a un altro processo; un processo può, dunque, prelevare un frame da un altro processo. La sostituzione locale richiede invece che ogni processo selezioni un frame solo dal proprio insieme di frame.
Si consideri ad esempio uno schema di allocazione che permetta ai processi ad alta priorità di selezionare frame dai processi a bassa priorità per effettuare una sostituzione. Un processo può selezionare una sostituzione tra i propri frame oppure tra i frame di qualsiasi processo con priorità più bassa. Questo approccio permette a un processo ad alta priorità di aumentare la propria allocazione di frame a discapito del processo a bassa priorità.
Con la strategia di sostituzione locale, il numero di frame allocati a un processo non cambia. Con la sostituzione globale, invece, può succedere che un processo selezioni solo frame allocati ad altri processi, aumentando così il numero di frame allocati a quel processo, supponendo che gli altri processi non scelgano i suoi frame per la sostituzione.
L'algoritmi di sostituzione globale risente di un problema: un processo non può controllare la propria frequenza di page fault. L'insieme delle pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi. Quindi, lo stesso

processo può comportarsi in modi piuttosto diversi, ad esempio impiegando 0,5 secondi per un'esecuzione e 10,3 secondi per l'esecuzione successiva, a causa di circostanze esterne. Nell'algoritmo di sostituzione locale questo problema non si presenta, la quello globale genera un maggiore throughput del sistema, e perciò è il metodo più utilizzato.

9.7 THRASHING



Se il numero dei frame allocati a un processo con priorità bassa scende al di sotto del numero minimi richiesto dall'architettura del computer, occorre sospendere l'esecuzione del processo. Occorre anche togliere le pagine restanti, liberando tutti i frame allocati. Questa operazione introduce un livello di swap in, swap out di scheduling della cpu intermedio.
Infatti, si consideri un qualsiasi processo che non abbia frame sufficienti. Anche se tecnicamente è possibile ridurre il numero minimo di frame allocati, esiste un certo (grande) numero di pagine in uso attivo. Se il processo non ha questo numero di frame causa subito un page fault. A questo punto deve sostituire qualche pagina; tuttavia poiché tutte le sue pagine sono in uso attivo, deve sostituire una pagina che sarà immediatamente necessaria, e di conseguenza emette parecchi page fault. Il processo continua ad emettere fault, sostituendo pagine per le quali emetterà fault e che riprenderà immediatamente. Questa intensa attività di paginazione è chiamata thrashing. Un processo è in thrashing quando spende più tempo per la paginazione che per l'esecuzione.

9.7.1 CAUSE DEL THRASHING

Il thrashing causa parecchi problemi di prestazioni. Si consideri il seguente scenario, basato sull'effettivo comportamento dei primi sistemi di paginazione.
Il SO monitorizza l'utilizzo della cpu. Se questo è troppo basso, viene aumentato il grado di multiprogrammazione introducendo nel sistema un processo nuovo. Viene utilizzato un algoritmo di sostituzione delle pagine globale, il quale sostituisce le pagine senza tenere conto del processo al quale appartengono. Per ora viene ipotizzato che un processo entri in una nuova fase esecutiva e richieda più frame; se ciò si verifica inizia il faulting, e nuove pagine vengono prelevate da altri processi. Questi processi hanno però bisogno di quelle pagine e quindi emettono anch'essi fault, prendendo pagine da altri processi. Questi processi di faulting devono utilizzare il dispositivo di paginazione per effettuare

swap in e out delle pagine. Mentre i processi si mettono in coda per il dispositivo di paginazione, la ready queue si svuota, quindi l'utilizzo della cpu diminuisce.
Lo scheduler della cpu si accorge di questa riduzione dell'utilizzo della cpu e aumenta il gradi di multiprogrammazione. Il processo nuovo tenta di partire prendendo pagine dai processi in esecuzione, allungando la coda per il dispositivo di paginazione. L'utilizzo della cpu scende ulteriormente, e lo scheduler della cpu tenta di aumentare ancora il grado di multiprogrammazione. Si è verificato thrashing e il throughput del sistema precipita. La frequenza di page fault aumenta in maniera impressionante, e di conseguenza aumenta il tempo effettivo di accesso in memoria. Non viene svolto alcun lavoro, in quanto i processi stanno spendendo tutto il loro tempo in paginazione. A questo punto, per aumentare l'utilizzo della cpu e fermare il thrashing, occorre ridurre il grado di multiprogrammazione.
Gli effetti del thrashing possono essere limitati utilizzando un algoritmo di sostituzione locale, o di priorità. Con la sostituzione locale, se un processo va in thrashing, non può rubare frame da un altro processo, causando il thrashing. Le pagine vengono sostituire tenendo conto del processo di cui fanno parte. Tuttavia, se i processi sono in thrashing, rimangono nella coda del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio per un page fault aumenta a causa dell'allungamento medio della coda di attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo di accesso aumenta anche per un processo che non è in thrashing.
Per evitare il verificarsi del thrashing, occorre fornire a un processo tanti frame quanti ne necessita. Esistono diverse tecniche che permettono di sapere quanto frame servono a un processo. La strategia working–set inizia osservando quanti frame stia utilizzando effettivamente un processo. Questo approccio definisce il modello di località di esecuzione del processo.
Il modello di località dichiara che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine che vengono utilizzate attivamente insieme. Generalmente u programma è formato da parecchie località diverse, che possono sovrapporsi.
Ad esempio, quando viene chiamata una subroutine, essa definisce una località nuova. In questa località vengono fatti riferimenti in memoria alle istruzioni della subroutine, alle sue variabili locali e a un sottoinsieme delle variabili globali.
Quando la subroutine termina, il processo lascia questa località, poiché le variabili locali e le istruzioni della subroutine non sono più utilizzate attivamente. Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati.
Si supponga di allocare a un processo un numero di frame sufficienti per sistemare le sue località attuali. Fino a che queste pagine non si trovano in memoria, il processo esegue page fault per le pagine della propria località: quindi non hanno luogo page fault fino a che le località non vengono modificate. Se vengono allocati meno frame rispetto alla dimensione della località attuale il processo entra in thrashing, poiché non può tenere in memoria tutte le pagine che sta utilizzando attivamente.

9.7.2 Modello working–set

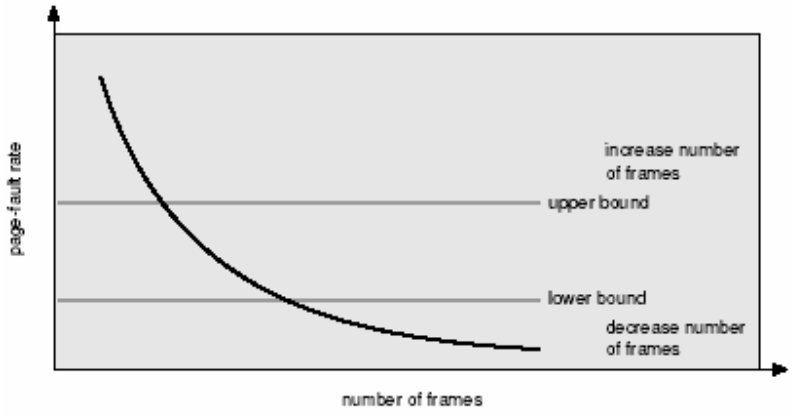
Il modello working–set è basato sull'ipotesi di località. Questo modello utilizza un parametro, Δ, per definire la finestra del working set. L'idea consiste nell'esaminare i più

recenti Δ riferimenti alle pagine. L'insieme di pagine nei più recenti Δ riferimenti è il working–set. Se una pagina è in uso attivo si trova nel working–set; se non è più utilizzata esce dal working–set Δ unità di tempo dopo il suo riferimento. Quindi il working–set non è altro che un'approssimazione della località di programma.
La precisione del working–set dipende dalla selezione di Δ. Se Δ è troppo piccolo non include l'intera località, se è troppo grande può sovrapporre più località. Al limite, se Δ è infinito, il working–set è l'insieme di pagine per cui il processo fa riferimento durante la sua esecuzione.
La caratteristica più importante del working–set è la sua dimensione. Calcolando la dimensione del working–set, WSS_i, per ciascun processo del sistema, è possibile determinare la richiesta totale di frame:

$$D= \sum WSS_i$$

Ogni processo sta utilizzando attivamente le pagine del proprio working–set. Quindi il processo i necessaria di WSS_i frame. Se la richiesta totale è maggiore del numero totale di frame liberi (D>m), si verifica thrashing, in quanto alcuni processi non dispongono di frame sufficienti. L'utilizzo del modello working–set è abbastanza semplice. Il SO monitorizza il working–set di ogni processo e gli alloca un numero di frame sufficienti per fornirgli la sua dimensione del working–set. Se i frame extra sono in numero sufficiente, può essere iniziato un nuovo processo. Se la somma delle dimensioni dei working set aumenta, superando il numero totale dei frame disponibili, il SO seleziona un altro processo da sospendere.
Questa strategia impedisce il verificarsi di thrashing, mantenendo il grado di multiprogrammazione più alto possibile, quindi risulta ottimizzato l'utilizzo della cpu.
La difficoltà insita nel modello working–set consiste nel tener traccia del working set stesso, poiché la finestra del working–set è una finestra in movimento, A ogni riferimento in memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nel working–set se esiste un riferimento a essa in qualsiasi punto della finestra del working–set. Il modello working–set può essere approssimato con un interrupt del timer a intervalli fissi e un bit di riferimento.

9.7.3 FREQUENZA DI PAGE FAULT



La strategia basata sulla frequenza di page fault (PFF) rappresenta un approccio più diretto. Il problema specifico è la prevenzione del thrashing. La frequenza di page fault nel thrashing è alta, ed è proprio questa che deve essere controllata. Quando la frequenza di page fault è eccessiva, significa che il processo necessita di più frame.

Analogamente, se la frequenza di page fault è troppo bassa, allora il processo può avere troppi frame. È possibile fissare un limite inferiore un limite superiore per la frequenza di page fault desiderata. Se la frequenza effettiva di page fault supera il limite superiore, occorre allocare un altro frame a quel processo; se la frequenza scende sotto il limite inferiore, viene sottratto un frame a quel processo. Quindi, per prevenire il thrashing, è possibile misurare e controllare direttamente la frequenza di page fault.

10 INTERFACCIA DEL FILE SYSTEM

Il file system consiste di due parti distinte: una collezione di file, ciascuno dei quali contenente dati correlati, e una struttura di directory, che organizza tutti i file nel sistema e fornisce informazioni su di essi. Alcuni file system hanno una terza componente: le partizioni, usate per separare fisicamente o logicamente grossi gruppi di directory.

10.1 CONCETTO DI FILE

il SO astrae dalle caratteristiche fisiche dei propri dispositivi di memoria per definire un'unità di memoria logica, il file. I file vengono mappati dal SO su dispositivi fisici di memorizzazione.
Un file è un insieme di informazioni, correlate e registrate nella memoria secondaria, a cui è stato assegnato un nome. Dal punto di vista dell'utente, un file è la più piccola porzione di memoria secondaria logica; i dati possono cioè essere scritti nella memoria secondaria soltanto all'interno di un file.
Le informazioni contenute in un file sono definite dal suo creatore. In un file possono essere memorizzate molti tipi di informazioni. Un file ha una struttura definita in base al tipo: un file di testo è formato da una sequenza di caratteri organizzati in righe, e probabilmente pagine; un file sorgente è formato da una serie di subroutine e funzioni…

10.1.1 ATTRIBUTI DEI FILE

Per comodità degli utenti umani, ogni file ha un nome che viene usato come riferimento. Una volta ricevuto il nome, il file diventa indipendente dal processo, dall'utente e anche dal sistema in cui è stato creato.
Un file ha altri attributi che possono variare a seconda del SO, ma che tipicamente comprendono i seguenti:

- Nome: il nome simbolico del file è l'unica informazione in forma umana leggibile
- Tipo: questa informazione è necessaria ai sistemi che supportano tipi di file diversi.
- Locazione: si tratta di un puntatore al dispositivo e alla locazione del file su quel dispositivo.
- Dimensione: si tratta della dimensione attuale del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.
- Protezione: le informazioni di controllo sull'accesso controllano chi può leggere, scrivere o eseguire il file.

- Ora, data e identificazione dell'utente: queste informazioni possono essere relative a (1) creazione, (2) ultima modifica e (3) ultimo utilizzo. Questi dati possono essere utili ai fini della protezione e per monitorare l'utilizzo.

Le informazioni sui file sono conservate nella struttura di directory, che risiede a sua volta nella memoria secondaria. Per registrare tali informazioni per ogni file possono essere necessari da 16 fino a più di 1000 byte; in un sistema con molti file, la dimensione della stessa directory può essere di megabyte.

10.1.2 Operazioni sui file

Un file è un tipo di dato astratto. Per definire adeguatamente un file è necessario prendere in considerazioni che possono essere eseguite su di esso. Il SO mette a disposizione delle system call per creare, scrivere, leggere, spostare, cancellare e troncare un file.

- Creazione di un file. Per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system. Secondariamente, per il file deve essere creato nella directory un nuovo elemento nel quale registrare il nome del file e la sua posizione nel file system.
- Scrittura di un file. Per scrivere su di un file è indispensabile una system call che specifichi il nome del file e le informazioni che si vogliono scrivere su di esso. Dato il nome del file, il sistema cerca nella directory la sua posizione. Il file system deve mantenere un puntatore di scrittura alla locazione nel file in cui deve aver luogo la successiva operazione di scrittura. Il puntatore deve essere aggiornato ogniqualvolta si esegue una scrittura.
- Lettura di un file. Per leggere un file è necessaria una system call che specifichi il nome del file e la posizione in memoria dove porre il successivo blocco del file. Anche in questo caso viene ricercato l'elemento corrispondente nella directory e il sistema deve mantenere un puntatore di lettura alla locazione del file in cui deve aver luogo la successiva operazione di lettura. Una volta completata la lettura, il puntatore viene aggiornato. Poiché generalmente un file viene o letto o scritto, la maggior parte dei sistemi mantiene soltanto un puntatore alla posizione corrente del file. Sia le operazioni di lettura che di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- Riposizionamento di un file. Si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Il riposizionamento non richiede alcuna operazione di I/O. Questa operazione è nota come posizionamento (seek) di u file.
- Cancellazione di un file. Per cancellare un file si cerca l'elemento della directory associato al file designato, quindi si rilascia tutto lo spazio associato al file (in modo che possa essere adoperato da altri file) e si elimina l'elemento della directory.
- Troncamento di un file. Si presentano situazioni in cui gli utenti desiderano che un file mantenga gli stessi attributi, pur volendone cancellare il contenuto. Invece di forzare gli utenti a cancellare il file e quindi ricrearlo, questa funzione consente di mantenere immutati gli attributi (ad esclusione della lunghezza del file) pur azzerando la lunghezza del file.

Queste sei operazioni di base comprendono sicuramente l'insieme minimo delle operazioni richieste sui file. Altre operazioni comuni comprendono l'accodamento di nuov informazioni alla fine di un file esistente e la rinomina di un file esistente. Queste operazioni primitive possono essere combinate per effettuare altre operazioni.
La maggior parte delle operazioni richiede una ricerca nella directory dell'elemento associato al file specificato. Per evitare questa ricerca costante, molti sistemi aprono un file la prima volta che viene adoperato in maniera attiva. Il So mantiene una piccola tabella contenente delle informazioni su tutti i file aperti (la tabella dei file aperti). Quando viene richiesta un'operazione su un file viene utilizzato un indice in questa tabella per evitare qualsiasi ricerca. Quando un file non viene più attivamente usato viene chiuso dal processo, e il SO rimuove l'elemento ad esso associato nella tabella dei file aperti.
L'operazione open prende il nome del file, lo cerca nella directory e copia l'elemento a esso associato nella tabella dei file aperti, sempre che le protezioni ne consentano l'accesso. La system call open restituisce di solito un puntatore all'elemento nella tabella dei file aperti: questo puntatore viene adoperato al posto dell'effettivo nome del file in tutte le operazioni di I/O, evitando così successive operazioni di ricerca e semplificando l'interfaccia delle system call.
L'implementazione delle operazioni open e close in un ambiente multiutente è più complessa. In questo sistema molti utenti possono aprire un file contemporaneamente; solitamente il SO introduce due livelli di tabelle interne: una tabella per processo contiene i riferimenti ai file aperti da quel processo; ad esempio, il puntatore alla posizione di ciascun file si trova in questa tabella e specifica la posizione nel file in cui opereranno le successive read o write.
Ciascun elemento della tabella associata a ciascun processo punta a sua volta a una tabella di sistema dei file aperti, la quale contiene informazioni indipendenti dai processi come la posizione dei file su disco, le date degli accessi e la dimensione del file. Quando un file è già stato aperto da un processo, una open eseguita da un altro processo comporta solamente l'aggiunta di un nuovo elemento nella tabella dei file aperti associata a quel processo, nella quale verranno registrati una nuova posizione corrente e il puntatore al corrispondente elemento della tabella del sistema. Tipicamente, la tabella dei file aperti ha anche un contatore delle open associato a ciascun file, che indica il numero di processi che hanno aperto quel file. Ogni close decrementa questo contatore, e quando esso raggiunge il valore zero il file non è più in uso e viene eliminato l'elemento corrispondente dalla tabella dei file aperti. Riassumendo, a ciascun file aperto sono associate diverse informazioni:

- Puntatore al file: il sistema deve tenere traccia dell'ultima posizione di lettura/scrittura sotto forma di puntatore alla posizione corrente del file. Questo puntatore è unico per ogni processo che opera sul file e quindi deve essere tenuto separato dagli attributi del file residenti su disco.
- Contatore dei file aperti
- Posizione su disco del file. La maggior parte delle operazioni richiede al sistema di modificare i dati contenuti nel file. L'informazione necessaria per localizzare il file su disco viene mantenuta in memoria per evitare di doverla prelevare dal disco a ogni operazione.

Alcuni SO mettono a disposizioni funzionalità per il locking di sezioni di un file aperto per l'accesso a più processi, per condividere delle sezioni di file tra più processi e, su sistemi con memoria virtuale, addirittura per mappare un file in memoria. Quest'ultima funzionalità prende il nome di memory mapping di un file e consente di associare logicamente parte dello spazio virtuale degli indirizzi a una sezione di un file. Le operazioni di lettura e scrittura in quella regione di memoria vengono considerate come letture e o scritture nel file, semplificando enormemente l'uso del file stesso. La chiusura del file comporta la scrittura su disco di tutti i dati mappati in memoria e la loro rimozione dalla memoria virtuale del processo.

10.1.3 Tipi di file

Una delle principali considerazioni relative alla progettazione di un file system e dell'intero SO riguarda la possibilità o meno di quest'ultimo di riconoscere e supportare i tipi di file. Un SO che riconosce il tipo di un file ha la possibilità di manipolare il file in modo ragionevole. Ad esempio, un errore abbastanza comune consiste nel tentativo da parte di utenti, di stampare un programma oggetto in forma binaria; solitamente questo tentativo porta solo ad uno spreco di carta, ma potrebbe essere comunque impedito se il SO fosse stato informato del fatto che il file è un programma oggetto un forma binaria. Una tecnica comune per implementare i tipi di file consiste nell'includere il tipo del file nel nome del programma. Il nome viene suddiviso in due parti, un nome e un'estensione, di solito separate da un punto.
Nel SO Apple Macintosh ogni file ha un proprio tipo, come "text" o "pict". Ciascun file possiede anche un attributo di creazione contenente il nome del programma che lo ha creato. Quando un utente apre il file il programma viene richiamato automaticamente e il file viene caricato, pronto per essere sottoposto a editing.
Il SO Unix non è in grado di gestire funzionalità di questo tipo, poiché si limita a memorizzare un magic number all'inizio di alcuni file allo scopo di indicare a grandi linee il tipo del file: eseguibili, batch, postscript… Non tutti i file possiedono un magic number, per cui il sistema non può affidarsi unicamente a questo tipo di informazione. Inoltre Unix non memorizza il nome del programma che ha creato il file. Unix consente di sfruttare le estensioni come suggerimento del tipo di file, non vengono però imposte né dipendono dal SO; il loro compito consiste principalmente nell'aiutare gli utenti a riconoscere il tipo di contenuto del file.

10.1.4 Struttura di file

I tipi di file possono essere anche adoperati per indicare la struttura interna del file. Inoltre alcuni file devono rispettare una determinata struttura comprensibile al SO. Alcuni SO impongono (e supportano) un numero minimo di strutture di file. Questo approccio è stato seguito da Unix, Ms–dos e altri. Unix considera ciascun file come una sequenza di byte di 8 bit, senza interpretare questi bit in alcun modo. Questo schema garantisce la massima flessibilità, ma il minimo supporto. Qualsiasi programma d'applicazione deve mantenere il proprio codice per interpretare un file di input nell'appropriata struttura. A ogni modo, tutti i SO devono supportare almeno un tipo si

struttura (quella dei file eseguibili) per essere in grado di caricare ed eseguire i programmi.
Un altro esempio di SO che supporta un numero ridotto di strutture file è Macintosh, il quale si aspetta che i file eseguibili consistano in due parti: una resource fork e una data fork. La prima contiene le informazioni che interessano all'utente, come ad esempio le etichette dei bottoni visualizzati in un programma. Un utente straniero che vorrebbe modificarli potrebbe adoperare gli strumenti messi a disposizione dal SO per la modifica delle informazioni contenute nella resource fork. La data fork contiene codice di un programma o dati: l'usuale contenuto del file. È quindi utile che un SO supporti le strutture di file che sono adoperate di frequente, risparmiando al programmatore un notevole sforzo implementativo. Un numero troppo limitato di strutture rende scomoda la programmazione, mentre troppe strutture ingrossano il SO e confondono i programmatori.

10.1.5 STRUTTURA INTERNA DEI FILE

Per il SO la localizzazione di un offset all'interno di un file può essere complicata. Una soluzione diffusa a questo tipo di problema consiste nell'impaccamento di un certo numero di record logici in blocchi fisici.
Il SO Unix definisce tutti i file semplicemente come un flusso di byte. Ogni byte è indirizzabile individualmente dal proprio offset a partire dall'inizio. In questo caso il record logico è un byte. Il file system impacca e disimpacca automaticamente i byte in blocchi fisici (ad esempio 512 byte per blocco) come necessario.
La dimensione dei record logici, quella dei blocchi fisici e la tecnica di impaccamento determina il numero dei record logici all'interno di ogni blocco fisico. L'impaccamento può essere effettuato dal programma d'applicazione dell'utente oppure dal SO.
In entrambi i casi il file può essere considerato come una sequenza di blocchi. Tutte le funzioni di I/O di base operano in termini di blocchi. La conversione da record logici a blocchi fisici è un problema relativamente semplice.
Poiché lo spazio del disco è sempre allocato in blocchi, una parte dell'ultimo blocco del file può andare sprecata. Se ogni blocco è formato da 512 byte, allora per un file di 1949 byte vengono allocati quattro blocchi, sprecando 99 byte (frammentazione interna).

10.2 METODI DI ACCESSO

I file memorizzano informazioni; al momento dell'utilizzo è necessario accedere a queste informazioni e leggerle in memoria. Esistono molti metodi per accedere alle informazioni del file. Alcuni sistemi forniscono solo un metodo di accesso ai file, mentre su altri sistemi, come gli IBM, sono supportati diversi metodi di accesso; la scelta del metodo giusto per una particolare applicazione è un importante problema di progettazione.

10.2.1 ACCESSO SEQUENZIALE

il più semplice metodo di accesso è l'accesso sequenziale: le informazioni del file vengono elaborate in ordine, un record dopo l'altro; questo metodo di accesso è di gran lunga il più comune, ed è utilizzato, ad esempio, dagli editor e dai compilatori.

10.2.2 ACCESSO DIRETTO

Un altro metodo è l'accesso diretto (o accesso relativo). Un file è costituito da record logici di lunghezza fissa, ciò consente ai programmi di leggere e scrivere rapidamente record senza un ordine particolare. Il metodo ad accesso diretto è basato sul modello di disco di un filo: i dischi permettono, infatti, l'accesso casuale a ogni blocco di file. Nel metodo ad accesso diretto permette di leggere o scrivere blocchi arbitrari, perciò è possibile leggere il blocco 14, quindi il blocco 53 e poi scrivere il blocco 7. non esistono limiti sull'ordine di lettura o scrittura di un file con accesso diretto.
I file ad accesso diretto sono molto utili quando sia necessario accedere immediatamente a grandi quantità di informazioni. Spesso i database sono di questo tipo.
Le operazioni sui file devono essere modificate per inserire il numero del blocco in forma di parametro. Quindi, si hanno read n, dove n è il numero del blocco, al posto di read next, e write n, invece che write next. Un approccio alternativo prevede di mantenere read e write next, come nell'accesso sequenziale, e di aggiungere un'operazione position file to n, dove n è il numero del blocco. Quindi per effettuare una read n occorre una position to n e quindi una read next.
Il numero del blocco fornito dall'utente al SO è normalmente un numero di blocco relativo. Si tratta di un indice relativo all'inizio del file, quindi il primo blocco relativo del file è 0, il successivo 1… L'utilizzo dei numero di blocco relativi permette al SO dove posizionare il file e aiuta a impedire che l'utente acceda a porzioni che l'utente acceda a porzioni del file system che possono non fare parte del suo file.

10.2.3 ALTRI METODI DI ACCESSO

Sulla base di un metodo di accesso diretto ne possono essere costruiti altri, che implicano generalmente la costruzione di un indice per il file. L'indice contiene puntatori ai vari blocchi; per trovare un elemento del file occorre prima cercare nell'indice, e quindi utilizzare il puntatore per accedere direttamente al file e trovare l'elemento desiderato.
Si consideri, ad esempio, un file contenente prezzi al dettaglio; questo può contenere un elenco di codici universali dei prodotti (UPC, universal product codes), a ciascuno dei quali è associato un prezzo. Dato un elemento di 15 byte, questo è formato da un codice UPC a 10 cifre e un prezzo di 6 cifre. Se il disco utilizzato ha 1024 byte per blocco, in ogni blocco possono essere memorizzati 64 elementi. Un file di 120.000 elementi occupa circa 2000 blocchi. Ordinando il file secondo il codice UPC è possibile definire un indice formato dal primo codice UPC di ogni blocco. Tale indice è costituito da 2000 elementi di 10 cifre ciascuno, oppure 20.000 byte, e quindi può essere tenuto in memoria. Per trovare il prezzo di un oggetto specifico è possibile effettuare una ricerca nell'indice. Da questa ricerca è possibile sapere esattamente quale blocco contiene l'elemento desiderato e quindi accedere a quel blocco. Questa struttura permette di effettuare ricerche in un grosso file limitando il numero degli I/O.
Nel caso di grossi file, lo stesso file indice può diventare troppo grande per essere tenuto in memoria. Una soluzione a questo problema è la creazione di un indice per il file indice. Il metodo di accesso sequenziale indicizzato IMB (index sequential access method), ad esempio, utilizza un piccolo indice master che punta ai blocchi su disco di un indice secondario, e i blocchi dell'indice secondario puntano ai blocchi del file effettivo. Il file è ordinato rispetto a una chiave definita. Per trovare una particolare voce, viene effettuata inizialmente una ricerca binaria nell'indice master, che fornisce il numero del blocco

dell'indice secondario. Questo blocco viene letto, quindi viene effettuata una seconda ricerca binaria per trovare il blocco contenete il record richiesto. Infine, sul blocco viene effettuata una ricerca sequenziale. In questo modo ogni record può essere localizzato grazie al proprio codice effettuando al massimo due letture in accesso diretto.

10.3 STRUTTURA DI DIRECTORY

Il file system di un computer può essere molto ampio. Alcuni sistemi memorizzano migliaia di file su centinaia di megabyte di spazio su disco. Per gestire tutti questi dati è necessario che essi siano organizzati, e questa organizzazione viene effettuata solitamente in due parti. Innanzitutto il file system viene diviso in partizioni, note anche come minidischi in ambiente IBM e volumi tra utenti PC e Mac. Tipicamente, ciascun disco di un sistema contiene almeno una partizione, che è una struttura a basso livello nella quale risiedono file e directory.
In secondo luogo, ciascuna partizione contiene le informazioni sui file in essa contenuti. Queste informazioni sono mantenute negli elementi della directory del dispositivo (device directory) o tabella dei contenuti del volume. La directory del dispositivo (più nota semplicemente come directory) registra le informazioni (nome, posizione, dimensione, tipo…) di tutti i file della partizione.
La directory può essere considerata come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. Da questo punto di vista, risulta evidente come anche la directory possa essere organizzata in molti modi diversi; deve essere possibile inserire nuovi elementi, cancellarne di esistenti, cercare un elemento ed elencare tutti gli elementi di una directory.
In particolare si deve poter fare:

- Ricerca di un file. Deve esserci la possibilità di scandire una directory per trovare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.
- Creazione di un file. Deve essere possibile creare nuovi file e aggiungerli alla directory.
- Cancellazione di un file. Quando un file non è più necessario lo si deve poter rimuovere dalla directory.
- Listare una directory.
- Rinomina di un file.
- Attraversamento del file system. È utile acere la possibilità di accedere a ogni directory e a ciascun file contenuto in una directory. Per motivi di affidabilità, è una buona idea salvare il contenuto e la struttura dell'intero file system a intervalli regolati (backup).

10.3.1 DIRECTORY A LIVELLO SINGOLO

La struttura più semplice per una directory è quella a livello singolo. Tutti i file sono contenuti nella stessa directory, la quale è facilmente supportabile e comprensibile.
Una directory a livello singolo presenta però limiti notevoli, che si manifesta quando aumenta il numero di file in essa contenuti, oppure quando viene utilizzata da più utenti.

Dal momento che tutti i file si trovano nella stessa directory, essi devono avere nomi unici. Quando due utenti attribuiscono lo stesso nome al loro file dati, ad esempio test, viene violata la regola del nome unico. Anche se i nomi dei file vengono generalmente scelti in modo da riflettere il contenuto del file stesso, spesso hanno una lunghezza limitata (Ms-Dos 8, Unix 255).
Anche se l'utente è unico, aumentando il numero dei file diventa difficile ricordarne i nomi per avere solo file con nomi unici. È comunque abbastanza diffuso il caso di utenti con centinaia di file su un computer e altrettanti file su un altro sistema. In un tale ambiente, sarebbe terribile dover ricordare tanti file.

10.3.2 Directory a due livelli

Lo svantaggio principale dato da una directory a livello singolo è la confusione che viene creata tra diversi utenti a causa dei nomi dei file. La soluzione standard prevede la creazione di una directory separata per ogni utente.
Nella struttura a due livelli, ogni utente dispone della propria file directory (user file directory — UFD). Tutte le UFD hanno una struttura simile, ma in ciascuna vengono elencati solo i file del proprietario. Quando inizia un job utente, oppure un utente effettua il login, viene effettuata una ricerca nel master file directory (MFD) del sistema. La master file directory viene indirizzata con il nome dell'utente o il numero di account, e ogni suo elemento punta alla UFD di tale utente.
Quando un utente effettua un riferimento a un file particolare, il SO effettua la ricerca solo nella UFD di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno della UFD siano unici.
Le stesse directory utente devono essere create e cancellate come necessario; a tale scopo viene eseguito uno speciale programma di sistema con nome utente e dati contabili adeguati. Il programma crea una nuova file directory utente e aggiunge l'elemento a essa corrispondente nella master file directory.
La struttura di directory a due livelli risole il problema delle collisioni dei nomi, ma presenta ancora dei problemi: una struttura simile isola un utente dagli altri utenti.
Se l'accesso è autorizzato, un utente deve avere la possibilità di riferirsi a file di altri utenti. Una struttura simile può essere pensata come un albero di altezza due, per cui basta indicare sia il nome utente che il nome del file (path name o nome di percorso) per accedervi.
Un caso particolare di questa situazione riguarda i file di sistema, che invece di avere una copia per ogni utente, sia ha solo una directory condivisa.

10.3.3 Directory con struttura ad albero.

Il file system Ms-Dos è strutturato come un albero, che è infatti il più comune tipo di struttura delle directory. L'albero ha come root directory, e ogni file del sistema ha un unico path name. Un path name descrive il percorso che parte dalla radice, passa attraverso tutte le sottodirectory e arriva a una file specifico.
Una directory, o una sottodirectory, contiene un insieme di file o sottodirectory. Le directory sono semplicemente dei file, che vengono però trattati in maniera speciale.
Tutte le directory hanno lo stesso formato intero. La distinzione tra file e sottodirectory è

data dal bit di directory. Per creare e cancellare le directory vengono utilizzate system call specifiche.
Normalmente, ogni utente dispone di una directory corrente. La directory corrente deve contenere la maggior parte dei file di interesse corrente. Quando viene fatto riferimento a un file, viene effettuata una ricerca nella directory corrente. Se si necessita di un file non presente nella directory corrente, allora l'utente deve specificare il path name. Per cambiare directory si usa una system call che prende come parametro una directory e la imposta come directory corrente.
I path name possono essere o assoluti i relativi.

10.3.4 DIRECTORY CON STRUTTURA A GRAFO ACICLICO

Se nel sistema esiste una directory condivisa (cioè non copie di un file o una directory, ma uno solo, in modo che le modifiche si riflettano anche agli altri utenti), la struttura ad albero non basta più. Un grafo aciclico permette alle directory di avere sottodirectory e file condivisi. Un grafo aciclico, cioè senza cicli, rappresenta la generalizzazione naturale dello schema delle directory con struttura ad albero.
I file e le sottodirectory condivisi possono essere implementati in molti modi. In metodo diffuso (Unix), prevede la creazione di un nuovo elemento di directory, chiamato link. Un link è un puntatore ad un altro file o un'altra sottodirectory. Ad esempio, un link può essere implementato come path name assoluto o relativo (link simbolico). Quando viene effettuato un riferimento a un file viene effettuata una ricerca nella directory, l'elemento cercato è contrassegnato come link e fornisce il nome del file, o della directory, reale. Il link viene determinato utilizzando il path name per localizzare il file reale. I collegamenti vengono facilmente identificati grazie al loro formato nell'elemento della directory, e vengono infatti chiamati puntatori indiretti.
L'altro approccio per l'implementazione dei file condivisi prevede semplicemente la duplicazione di tutte le informazioni relative ai file in entrambe le directory di condivisione, quindi i due elementi sono identici. Un link è chiaramente diverso dagli altri elementi della directory. Duplicando gli elementi della directory, invece, la copia e l'originale vengono resi indistinguibili: sorge allora il problema di mantenere la coerenza se il file viene modificato.
Una struttura di directory con grafo aciclico è più flessibile di una semplice struttura ad albero, ma anche più complessa. Devono essere presi in considerazione parecchi problemi. Occorre notare che un file può avere più path name assoluti, per cui nomi diversi possono riferirsi allo stesso file.
Un altro problema riguarda la cancellazione, poiché è necessario stabilire in quali casi è possibile riallocare e riutilizzare lo spazio allocato a un file condiviso. Una possibilità è rimuovere il file ogni volta che qualcuno lo cancella, però…
In un sistema dove la condivisione è implementata da link simbolici la gestione di questa situazione è relativamente semplice. La cancellazione di un link non influisce sul file originale, poiché viene rimosso solo il link. Se viene cancellato il file, lo spazio corrispondente deve essere deallocato lasciando in sospeso il link; a questo punto è possibile cercare tutti questi link e rimuoverli, ma se in ogni file non esiste una lista dei link associati al file stesso questa ricerca può essere abbastanza costosa. In alternativa, i link possono essere lasciati fino a che non viene fatto un tentativo di utilizzarli. In quel

momento viene "scoperto" che il file con il nome dato dal link non esiste e non si riesce a determinare il link in base a quel nome; l'accesso viene trattato come qualsiasi altro nome di file illegale.
Un altro approccio alla cancellazione prevede la conservazione del file fino a che non siano stati cancellati tutti i riferimenti allo stesso. Per implementare questo approccio è necessario disporre di un meccanismo per determinare che l'ultimo riferimento a quel file è stato cancellato; è possibile tenere la lista di tutti i riferimenti a un file, o meglio, un contatore del numero di riferimenti. Quando il contatore è a 0 allora si può cancellare il file. Unix utilizza questo approccio per i link non simbolici, o hard link.

10.3.5 DIRECTORY CON STRUTTURA A GRAFO GENERALE

Un serio problema connesso all'utilizzo di una struttura con grafo aciclico consiste nell'assicurare che non vi siano cicli.
Il vantaggio principale di un grafo aciclico è dato dalla semplicità degli algoritmi necessari per attraversarlo e per determinare quando non ci sono più riferimento a un file. È preferibile evitare un duplice attraversamento di sezioni condivise di un grafo aciclico, soprattutto per motivi di prestazioni. Se un file particolare è appena stato creato in una sottodirectory condivisa, ma non viene trovato, è preferibile evitare una seconda ricerca nella stessa sottodirectory, che costituirebbe solo una perdita di tempo.
Se è permesso che nelle directory esistano dei cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni. Un algoritmo mal progettato potrebbe causare un loop infinito che cerca continuamente nel ciclo senza mai terminare la ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory a cui si farà accesso durante la ricerca.
Un problema analogo si presenta al momento di stabilire quando sia possibile cancellare un file. Come con strutture a grafo aciclico, la presenza di uno 0 nel contatore dei riferimenti significa che non esistono più riferimenti a quel file o a quella directory, e quindi può essere cancellato. Tuttavia, se esistono cicli, è possibile che il contatore possa essere non nullo, anche se non è possibile far riferimento a quel file o a quella directory. Questa anomalia è dovuta alla possibilità di autoriferimento a una directory o a un file. In questo caso è generalmente necessario usare una garbage collection per stabilire quando sia stato cancellato l'ultimo riferimento e quando sia possibile riallocare lo spazio su disco. La garbage collection implica l'attraversamento del file system, durante il quale tutto quanto è accessibile viene contrassegnato. Quindi, in un secondo passaggio, tutto quanto non è contrassegnato viene raccolto su una lista di blocchi liberi. La garbage collection per un file system basato su dischi richiede però molto tempo, e quindi viene tentata solo di rado.
Generalmente le directory con struttura ad albero sono più diffuse di quelle con grafo aciclico.

10.4 PROTEZIONE

Quando in un computer esistono informazioni, la loro protezione contro danni fisici e accesso improprio assume una notevole importanza; la protezione contro i danni fisici

determina l'affidabilità del sistema, mentre quella contro accessi impropri determina la protezione vera e propria.

10.4.1 Tipi d'accesso

La necessità di proteggere i file deriva direttamente dalla possibilità di accedervi. Su sistemi che non permettono l'accesso ai file di altri utenti la protezione non è necessaria, quindi, una soluzione estrema potrebbe essere quella di proibirne l'accesso. L'altro estremo prevede la libertà d'accesso, quindi assenza di protezione. Questi approcci sono entrambi eccessivi, quindi non possono essere applicati in generale; ciò che serve in realtà è un accesso controllato.
Il controllo offerto dai meccanismi di protezione viene ottenuto limitando i possibili tipi di accesso. L'accesso viene permesso o negato in base a diversi fattori, uno dei quali è il tipo di accesso richiesto.
È possibile controllare diversi tipi di operazioni:

- Lettura
- Scrittura
- Esecuzione
- Aggiunta
- Cancellazione
- Lista
- …

10.4.2 Liste di accesso e gruppi

L'approccio più comune al problema della protezione prevede di rendere l'accesso dipendente dall'identità dell'utente. Più utenti possono richiedere diversi tipi di accesso a un file o a una directory. Lo schema più generale per implementare gli accessi dipendenti dall'identità consiste nell'associare un lista d'accesso a ogni file e directory; in tale lista sono specificati il nome dell'utente e i tipi di accesso permessi all'utente.
Il problema maggiore delle liste di accesso è la loro lunghezza: per permettere a tutti di leggere un file, nella lista devono essere contenuti tutti gli utenti con accesso alla lettura. Questa tecnica comporta due inconvenienti:

1. La costruzione di una lista di questo tipo può essere un compito noioso, soprattutto se non è già nota la lista degli utenti del sistema.
2. L'elemento della directory, che precedentemente aveva dimensione fissa, richiede in questo caso una dimensione variabile, per cui anche la gestione dello spazio risulta più complicata

Questi problemi possono essere risolti usando una versione condensata della lista d'accesso. Per condensare la lunghezza della lista, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte:

- Proprietario
- Gruppo
- Universo

Per definire la protezione, data questa più limitata classificazione, occorrono solo tre campi (rwx rwx rwx).

10.4.3 ALTRI METODI DI PROTEZIONE

Un altro approccio al problema della protezione consiste nell'associare una password a ciascun file. Proprio come l'accesso al computer stesso è spesso controllato da password, anche l'accesso a ogni file può avere lo stesso tipo di protezione. Tuttavia anche questa schema presenta numerosi svantaggi. In primo luogo, associando una password diversa per ogni file, il numero di password da ricordare è alto, per cui questo sistema risulta poco pratico. Se viene utilizzata la stessa password per tutti i file, una volta scoperta la parola tutti i file divengono accessibili.

10.5 SEMANTICA DELLA CONSISTENZA

La semantica della consistenza è un importante criterio per la valutazione di qualsiasi file system che supporti la condivisione dei file. Si tratta di una caratterizzazione del sistema che specifica la semantica delle operazioni in cui più utenti accedono contemporaneamente a un file condiviso. In particolare, questa semantica deve specificare quando le modifiche ai dati apportate da un utente possano essere da altri utenti.
Nella discussione seguente viene supposto che una serie di accessi, cioè letture e scritture, tentati da un utente sullo stesso file sia sempre compresa tra le operazioni open e close. Tale serie di accessi è chiamata sessione di file. Per illustrare questo concetto vengono schematizzati qui di seguito alcuni esempi di semantica della consistenza.

10.5.1 SEMANTICA UNIX

Il file system Unix utilizza la seguente semantica della consistenza:

- Le scritture su un file aperto da un utente sono immediatamente visibili ad altri utenti che hanno contemporaneamente lo stesso file
- Esiste un metodo di condivisione in cui gli utenti condividono il puntatore alla locazione corrente nel file, quindi l'avanzamento del puntatore da parte di un utente influisce su tutti gli utenti che condividono il file. In questa caso il file ha una singola immagine e tutti gli accessi vengono eseguiti in interleaving (interfogliati) a prescindere dalla loro origine.

Questa semantica si presta a un'implementazione in cui il file viene associato a una singola immagine fisica alla quale è possibile accedere come a una risorsa esclusiva, La contesa di questa immagine singola causa ritardi ai processi utenti.

10.5.2 SEMANTICA DELLE SESSIONI

Il file system Andrei utilizza la seguente semantica della consistenza

- Le scritture su un file aperto da parte di un utente non sono immediatamente visibili ad altri utenti che hanno aperto contemporaneamente lo stesso file.

- Una volta chiuso il file, le modifiche apportate sono visibili solo nelle sessioni che inizieranno successivamente. Le istanze già aperte non riportano queste modifiche.

10.5.3 SEMANTICA DEI FILE CONDIVISI IMMUTABILI

Un diverso e unico approccio è quello dei file condivisi immutabili; una volta che un file è stato dichiarato dal suo creatore come un file condiviso, non può essere modificato. Un file immutabile presenta due caratteristiche importanti: il suo nome non può essere riutilizzato e il suo contenuto non può essere modificato. Quindi, il nome di un file immutabile designa il contenuto fissato del file, piuttosto che un file come contenitore di informazioni variabili.

11 REALIZZAZIONE DEL FILE SYSTEM

Il file system fornisce il meccanismo per la memorizzazione e l'accesso on–line a dati e programmi. Il file system risiede permanentemente sulla memoria secondaria, il primo requisito della quale deve essere la capacità di contenere permanentemente grandi quantità di dati. Questo capitolo riguarda principalmente i problemi connessi alla memorizzazione e all'accesso relativi ai file sul più comune supporto di memoria secondaria, il disco. Vengono esaminati i vari metodi di allocazione dello spazio su disco, di recupero dello spazio liberato, di registrazione delle locazioni dei dati, e di interfaccia di altre componenti del SO alla memoria secondaria.

11.1 STRUTTURA DEL FILE SYSTEM

I dischi costituiscono la maggior parte della memoria secondaria su cui viene conservato il file system. Per migliorare l'efficienza di I/O, i trasferimenti di I/O tra memoria e sisco vengono eseguiti in unità di blocchi. Ogni blocco è costituito da uno o più settori, la dimensione dei quali può variare d 32 a 4096 byte a seconda del tipo di unità di disco (solitamente 512). I dischi hanno due caratteristiche importanti che ne fanno un mezzo conveniente per la memorizzazione di più file:

1. Possono essere riscritti localmente; è possibile leggere un blocco dati dal disco, modificarlo e quindi riscriverlo nella stessa posizione
2. È possibile accedere direttamente a qualsiasi blocco di informazioni del disco, quindi risulta semplice accedere a qualsiasi file, sia in modo sequenziale che casuale, e passare da un file all'altro solamente spostando le testine di lettura–scrittura e attendendo la rotazione del disco.

11.1.1 ORGANIZZAZIONE DEL FILE SYSTEM

Per fornire un efficiente e conveniente accesso al disco, il SO fa uso di un file system che consente di memorizzare, individuare e recuperare facilmente i dati. Un file system presenta due problemi di progettazione piuttosto diversi. Il primo problema riguarda la definizione dell'aspetto del file system agli occhi dell'utente. Questo compito implica le definizione di un file e dei suoi attributi.

Lo stesso file system è generalmente composto da molti livelli distinti. Ogni livello utilizza le funzioni dei livelli inferiori per creare nuove funzioni che vengano utilizzate dai livelli superiori.
Il livello più basso, il *controllo dell'I/O*, è costituito dai driver dei dispositivi e dai gestori di interrupt per trasferire le informazioni tra la memoria di sistema e i dischi. Un driver di dispositivo può essere pensato come un traduttore. Il suo input consiste di comandi ad alto livello, come "recupera il blocco 123"; il suo output consiste di istruzioni di basso livello, specifiche dell'hardware.
Il file system di base deve soltanto inviare dei generici comandi all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici sul disco. Ogni blocco fisico è identificato dal suo indirizzo numerico del disco, ad esempio drive 1, cilindro 73, superficie 2, settore 10.
Il modulo di organizzazione dei file è a conoscenza dei file e dei loro blocchi logici così come dei blocchi fisici del disco. Conoscendo il tipo di allocazione dei file utilizzato e la locazione dei file, il modulo di organizzazione dei file può tradurre gli indirizzi dei blocchi logici, che il file system deve trasferire, negli indirizzi dei blocchi fisici. I blocchi logici di ciascun file sono numerati da 0 a N; i blocchi fisici contenenti tali dati non corrispondono ai numeri di blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione. Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non allocati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.
Il file system logico utilizza la struttura di directory per fornire al modulo di organizzazione dei file le informazioni di cui quest'ultimo ha bisogno, dato un nome simbolico di file.
Per creare un nuovo file un programma d'applicazione richiama il file system logico, che conosce il formato della struttura di directory. Per creare un nuovo file esso legge sulla directory appropriata in memoria, la aggiorna col nuovo elemento e la riscrive sul disco. Una volta creato, il file può essere usato per l'I/O. La struttura di directory potrebbe essere esaminata per ciascuna operazione di I/O allo scopo di individuare il file, controllandone i parametri, determinare i blocchi dei dati, quindi eseguire operazioni su di essi. Ciascuna operazione può, in questo modo, comportare un elevato sovraccarico. Prima che possa essere impiegato per operazioni di I/O, è meglio che il file debba essere aperto. Quando viene aperto un file, nella struttura di directory viene cercato l'elemento del file richiesto. Parti della struttura di directory sono di solito sottoposte a caching in memoria per accelerare le operazioni sulle directory. Una volta che il file è stato individuato, le informazioni ad esso associate sono copiate in una tabella mantenuta in memoria, detta tabella dei file aperti, contenente le informazioni su tutti i file correntemente aperti.
Il primo riferimento a un file (open) attiva la ricerca nella directory e l'elemento della directory corrispondente a tale file viene copiato nella tabella dei file aperti. L'indice di questa tabella viene riportato al programma utente e tutti i successivi riferimenti vengono effettuati attraverso tale indice anziché col nome simbolico del file. L'indice prende diversi nomi: descrittore di file (Unix), file handle (Windows NT), file control block…
Quindi, finché il file non viene chiuso, tutte le operazioni sui file vengono effettuate nella tabella dei file aperti. Quando il file viene chiuso da tutti gli utenti che lo hanno aperto, le informazioni aggiornate vengono ricopiate nella struttura di directory su disco.

Alcuni sistemi rendono ancora più complicato questo schema facendo uso di più livelli di tabelle in memoria. Ad esempio, nel file system di BSD Unix ciascun processo ha una tabella dei file aperti che contiene una lista di puntatori indicizzati dal descrittore; i puntatori conducono a una tabella dei file aperti per l'intero sistema. Tale tabella contiene informazioni sulle soggiacenti entità aperte: per quanto riguarda i file, fa riferimento a una tabella di inode attivi; per le connessioni di rete e di dispositivi fa riferimento a informazioni d'accesso simili. La tabella degli inode attivi è una cache in memoria degli inode attualmente in uso, e include i campi indice che puntano ai blocchi di dati sul disco. In questo modo, una volta che un file è aperto, tutto ciò che serve per un rapido accesso al file da parte di un processo, tranne gli effettivi blocchi di dati, è presente in memoria. In realtà, la open esamina innanzitutto la tabella dei file aperti per stabilire se il file è già in uso da parte di un altro processo. In tal caso è creato un elemento di tabella dei file aperti per quel processo, cui fa riferimento la tabella dei file aperti per l'intero sistema. Altrimenti, l'inode è copiato nella tabella degli inode attivi e sono creati un nuovo elemento per l'intero sistema e un nuovo elemento per processo.

11.1.2 MONTAGGIO DEI FILE SYSTEM

Così come un file deve essere parto prima di poter essere utilizzato, un file system deve essere montato per essere messo a disposizione dei processi del sistema. La procedura di montaggio è elementare. Il SO riceve il nome del dispositivo e il punta della struttura dei file (detto punto di montaggio) in cui attaccare il file system. Ad esempio in Unix un file system contente le home directory degli utenti potrebbe essere montato come /home; a questo punto, per accedere alla struttura di directory interna a quel file system sarà sufficiente far precedere il nome delle directory da /home (/home/Michele). Montando questo file system sotto /users il percorso necessario per raggiungere la stessa directory sarà /users/Michele.
Inoltre il SO verifica che il dispositivo contenga un file system valido chiedendo al driver del dispositivo di leggere la directory del dispositivo e di verificare che essa abbia il formato atteso. Infine, il SO annota nella directory che è stato montato un file system nel punto specificato. Questo schema consente al SO di attraversare la sua struttura di directory, passando, quando risulta opportuno, da un file system all'altro.

11.2 METODI DI ALLOCAZIONE

Il problema principale consiste nell'allocare spazio per questi file in modo che lo spazio sul disco venga utilizzato efficientemente e l'accesso ai file sia rapido. Esistono tre metodi principali per l'allocazione dello spazio su disco: contigua, concatenata o indicizzata

11.2.1 ALLOCAZIONE CONTIGUA

Per utilizzare il metodo di allocazione contigua, ogni file deve occupare un insieme di blocchi contigui sul disco. Gli indirizzi del disco definiscono un ordinamento lineare del disco stesso. Occorre notare che con questo ordinamento l'accesso al blocco b+1 dopo il

blocco b non richiede normalmente alcun spostamento della testina. Il SO IBM VM/CMS utilizza l'allocazione contigua perché consente buone prestazioni.
L'allocazione contigua di un file è definita dall'indirizzo del disco e dalla lunghezza, in unità di blocco, del primo blocco. Se il file è lungo n blocchi e inizia dalla locazione b, allora occupa i blocchi b, b+1, b+2, …, b+n-1. l'elemento di directory per ciascun file indica l'indirizzo del blocco di inizio e la lunghezza dell'area allocata per questo file.
Una difficoltà insita in questo tipo di allocazione riguarda l'individuazione dello spazio libero per un nuovo file. L'implementazione del sistema di gestione dello spazio libero (11.) determina il modo in cui tale compito viene eseguito.
Il problema dell'allocazione contigua dello spazio su disco può essere considerato un'applicazione particolare del problema generale dell'allocazione dinamica di memoria: il problema è quello di soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le strategie sono best fit e first fit.
Questi algoritmi soffrono di frammentazione esterna. Non appena i file vengono allocati e cancellati, lo spazio libero sul disco viene frammentato in tanti piccoli pezzi. La frammentazione esterna ha luogo ogniqualvolta lo spazio libero è suddiviso in pezzi.
A seconda della quantità totale di memoria su disco e della dimensione media dei file, la frammentazione può costituire un problema più i meno grave.
Alcuni vecchi sistemi di microcomputer utilizzavano l'allocazione contigua su floppy disk. Per impedire lo spreco di quantità notevoli di spazio a causa della frammentazione esterna, l'utente doveva eseguire una routine di compattazione che copiava l'intero file system, su un altro floppy o su nastro. Il floppy originale veniva quindi completamente liberato creando un vasto buco libero contiguo. La routine provvedeva quindi a copiare nuovamente i file su floppy, allocando lo spazio contiguo di questo grande buco. Questo schema compatta efficacemente tutto lo spazio libero in uno spazio contiguo, risolvendo il problema della frammentazione. Il costo di questa compattazione è rappresentato dal tempo; ed è particolarmente pesante per gli hard disk di grande capacità. Durante questo tempo morto il normale funzionamento del sistema non può essere consentito, quindi tale compattazione va evitata a tutti i costi sulle macchine in attività.
L'allocazione contigua implica, però, anche altri problemi, come la quantità di spazio necessaria per un file. Quando un file viene creato, occorre trovare e allocare lo spazio di cui necessita. Esiste il problema di conoscere la dimensione del file da creare.
Se a un file viene allocato poco spazio, può essere impossibile estendere il file. Soprattutto utilizzando la strategia best fit, può essere in uso lo spazio su ambedue le estremità del file, quindi non è possibile ampliare il file sul posto. Allora bisogna trovare un buco più grande, copiare il contenuto del file nel nuovo spazio e rilasciare lo spazio precedente. Queste operazioni possono essere ripetute finché esiste spazio, anche se ciò può far perdere tempo.
Anche conoscendo in anticipo la quantità di spazio necessaria per un file, la preallocazione può risultare inefficiente. A un file che cresce lentamente in un periodo di tempo lungo (mesi o anni) deve essere allocato spazio per la sua dimensione finale, anche se molto di quello spazio può rimanere inutilizzato per parecchio tempo. Il file ha perciò un'estesa frammentazione interna.
Per evitare alcuni di questi inconvenienti, alcuni SO fanno uso di uno schema di allocazione contigua modificato: viene inizialmente allocata una porzione di spazio contiguo, e se questa non è abbastanza grande un'altra porzione di spazio contiguo,

un'extent, viene aggiunta alla prima. La locazione dei blocchi dei file viene registrata come una locazione e un contatore di blocchi, insieme all'indirizzo del primo blocco dell'extent seguente.

11.2.2 ALLOCAZIONE CONCATENATA

L'allocazione concatenata risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione, infatti, ogni file è costituito da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Ad esempio, un file di 5 blocchi può cominciare al blocco 9, continuare al blocco 16, quindi al blocco 1, al blocco 10 e infine terminare al blocco 25. Ogni blocco contiene un puntatore al blocco successivo.
Per creare un nuovo file viene semplicemente creato un nuovo elemento nella directory. Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file. Questo puntatore è inizializzato a nil che sta ad indicare che il file è vuoto. Anche il campo della dimensione è impostato a 0. Un'operazione di scrittura sul file determina la ricerca di un blocco libero attraverso il sistema di gestione dello spazio libero, la scrittura su tale blocco, e la concatenazione di tale blocco alla fine del file. Per leggere un file occorre semplicemente leggere i blocchi seguendo i puntatori.
Con l'allocazione concatenata non esiste la frammentazione esterna.
Il problema principale dell'allocazione concatenata riguarda il fatto che può essere utilizzata efficacemente solo per file con accesso sequenziale. Per trovare l'i-esimo blocco di un file bisogna partire dall'inizio e seguire i puntatori. Ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina sul disco.
Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Se un puntatore richiede 4 byte di un blocco di 512 byte, allora lo 0,78% del disco viene sprecato per i puntatori.
La soluzione più comune a questo problema consiste nel raccogliere i blocchi in gruppi, detti cluster, e nell'allocare i cluster anziché i blocchi. Ad esempio, il file system può definire un cluster di 4 blocchi e operare soltanto in unità di cluster. I puntatori useranno quindi una percentuale molto più piccola di spazio. Questo metodo permette che il mapping dei blocchi logici sui blocchi fisici rimanga semplice, ma migliora il throughput del disco: si hanno meno posizionamenti della testina sul disco e diminuisce lo spazio necessario per l'allocazione dei blocchi e la gestione dei blocchi liberi. Il costo di questo approccio è dato da un incremento della frammentazione interna, poiché se un cluster è parzialmente pieno viene sprecato più spazio di quanto se ne sprechi quando a essere parzialmente pieno è un blocco. I cluster possono essere utilizzati per ottimizzare l'accesso al disco in molti algoritmi, quindi sono impiegati nella maggior parte dei SO.
Un altro problema riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi su tutto il disco, si provi a immaginare cosa accadrebbe se un puntatore andasse perduto o danneggiato. Una soluzione parziale è di utilizzare liste doppiamente concatenate, oppure di memorizzare il nome del file e il relativo numero di blocco in ogni blocco.

Una variante importante del metodo di allocazione concatenata consiste nell'utilizzo della tabella di allocazione file (FAT) (Usato da Ms-Dos e OS/2). Per contenere tale tabella viene riservata una porzione del disco all'inizio di ciascuna partizione; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; essa viene utilizzata essenzialmente come una lista concatenata. L'elemento di directory contiene il numero di blocco del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero di blocco del blocco successivo del file. Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine–file. I blocchi inutilizzati sono indicati da uno 0 nella tabella. L'allocazione di un nuovo blocco a un file implica semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine–file precedente con l'indirizzo del nuovo blocco.
Lo schema di allocazione basato su FAT, a meno che la FAT non venga sottoposta a caching, può causare un significativo numero di posizionamenti della testina. La testina del disco deve spostarsi all'inizio della partizione per leggere la FAT e trovare la locazione del blocco in questione, quindi raggiungere la locazione del blocco stesso; nel caso peggiore sono necessari ambedue i movimenti per ciascun blocco. Un vantaggio è dato dall'ottimizzazione del tempo d'accesso casuale, poiché la testina del disco può trovare la locazione di ogni blocco leggendo le informazioni contenute nella cache.

11.2.3 Allocazione indicizzata

L'allocazione concatenata risolve il problema della frammentazione esterna e della dichiarazione della dimensione del file, entrambi presenti nell'allocazione contigua. Tuttavia, in mancanza di una FAT, l'allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, in quanto i puntatori ai blocchi sono sparsi, con i blocchi stessi, su tutto il disco e devono essere recuperati con ordine. L'allocazione indicizzata risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il blocco indice.
Ogni file ha il proprio blocco indice: si tratta di un vettore di indirizzi di blocchi del disco. L'i-esimo elemento del blocco indice punta sull'i-esimo blocco del file. La directory contiene l'indirizzo del blocco indice.
Una volta creato il file, tutti i puntatori del blocco indice sono impostati a nil. Quando l'i-esimo blocco viene scritto per la prima volta, viene fornito un blocco dal gestore dei blocchi liberi; l'indirizzo di questo blocco viene inserito nell'i-esimo elemento del blocco indice.
L'allocazione indicizzata supporta l'accesso diretto senza soffrire di frammentazione esterna, poiché ogni blocco libero del disco può soddisfare una richiesta maggiore di spazio.
Con l'allocazione indicizzata si ha un certo spreco di spazio. L'overhead dei puntatori del blocco indice è generalmente maggiore di quello necessario per l'allocazione concatenata. Si consideri il comune caso di un file con uno o due blocchi; con l'allocazione concatenata viene perso solo lo spazio di un puntatore per blocco, in tutto uno o due puntatori; con l'allocazione indicizzata occorre allocare un intero blocco indice, anche se solo uno o due puntatori risultano non nil.

Questo punto solleva il problema della dimensione del blocco indice. Ogni file deve avere un blocco indice, per cui è auspicabile che questo sia il più piccolo possibile; ma se il blocco indice è troppo piccolo non è in grado di contenere puntatori sufficienti per un file di grandi dimensioni, quindi è necessario disporre dio un meccanismo per gestire queste situazioni:

- Schema concatenato. Un blocco indice è formato normalmente da un solo blocco del disco; perciò ciascun blocco indice può essere letto e scritto esattamente con una operazione. Per permettere la presenza di grossi file è possibile collegare tra loro parecchi blocchi indice.
- Indice multilivello. Una variante consiste nell'impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file stessi. Per accedere a un blocco il SO usa l'indice di primo livello, con il quale individua il blocco indice di secondo livello, e con esso trova il blocco dati richiesto. Questo approccio può continuare fino a un terzo o quarto livello…
- Schema combinato. Un'altra alternativa (usata da BSD Unix), consiste nel tenere i primi 15 puntatori del blocco indice (o inode) del file, l'elemento di directory punta all'inode. I primi 12 di questi puntatori puntato a blocchi diretti. Quindi, i dati per i piccoli file non richiedono un blocco indice distinto. Se la dimensione dei blocchi è di 4K, si può accedere fino a 48K di dati. Gli altri tre puntano a blocchi indiretti (indiretto singolo, doppio e triplo).

Si noti che gli schemi di allocazione indicizzata soffrono di alcuni degli stessi problemi di prestazioni che riguardano l'allocazione concatenata.

12.2.4 PRESTAZIONI

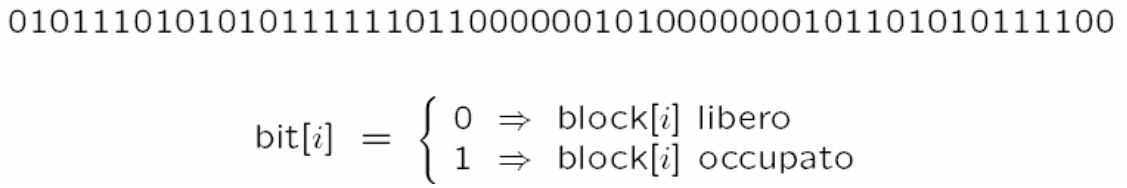
Per confrontare prestazioni di sistemi diversi è necessario sapere come essi vengono utilizzati: un sistema con prevalenza di accesso sequenziale farà uso di un metodo differente da quello di un sistema con prevalente accesso casuale. Per qualsiasi tipo di accesso, l'allocazione contigua richiede solo un accesso per ottenere in blocco. Poiché è facile tenere l'indirizzo iniziale del file in memoria, è possibile calcolare immediatamente l'indirizzo del disco dell'i-esimo blocco, e leggerlo immediatamente.
Utilizzando l'allocazione concatenata è possibile tenere in memoria anche l'indirizzo del blocco successivo e leggerlo direttamente. Questo metodo è valido per l'accesso sequenziale, mentre, per quanto riguarda l'accesso diretto, un accesso all'i-esimo blocco può richiedere i letture del disco. Questo spiega perché l'allocazione concatenata non deve essere usata per un'applicazione che richiede accesso diretto
Alcuni sistemi supportano file ad accesso diretto utilizzando l'allocazione contigua, mentre altri supportano file ad accesso sequenziale attraverso l'allocazione concatenata. Un file creato per l'accesso sequenziale è un file concatenato e non può essere utilizzato per l'accesso diretto. U file creato per accesso diretto è contiguo e può supportare entrambi i tipi di accessi, purché ne venga dichiarata la lunghezza massima al momento della creazione.
L'allocazione indicizzata è più complessa. Se il blocco indice è già in memoria, allora l'accesso può essere diretto. Tuttavia, per tenere il blocco indice in memoria occorre una quantità di spazio considerevole. Se questo spazio in memoria non è disponibile, occorre

leggere prima il blocco indice e quindi il blocco dati desiderato. Per un indice a due livelli possono essere necessarie due letture del blocco indice e quindi la lettura del blocco dati desiderato. Per un indice a due livelli possono essere necessarie due letture del blocco indice. Se un file è estremamente grande, per effettuare l'accesso di un blocco che si trovi vicino alla fine del file, prima di leggere il blocco dati occorre effettuare la lettura di tutti blocchi indice per seguire la catena dei puntatori. Quindi le prestazioni dell'allocazione indicizzata dipendono dalla struttura dell'indice, dalla dimensione del file e dalla posizione del blocco desiderato.
Alcuni sistemi combinano l'allocazione contigua con quella indicizzata, utilizzando quella contigua per i piccoli file (fino a tre o quattro blocchi) e passando automaticamente a quella indicizzata per grandi file. Poiché generalmente i file sono piccoli, e in questo caso l'allocazione contigua è efficiente, le prestazioni medie risultano abbastanza buone.

11.3 GESTIONE DELLO SPAZIO LIBERO

Per tener traccia dello spazio su disco il sistema conserva una lista dei blocchi liberi. Per creare un file occorre cercare nella lista dei blocchi liberi la quantità di spazio necessaria e allocarla al file. Quindi lo spazio deve essere rimosso dalla lista dei blocchi liberi. Quando un file viene cancellato, i suoi blocchi devono venir aggiunti alla lista.

11.3.1 VETTORE DI BIT



Il vantaggio principale che deriva da questo approccio è la relativa semplicità ed efficienza nel trovare il primo blocco libero o n blocchi liberi consecutivi del disco. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettori è conservato nella memoria centrale, ma ciò è possibile solo per i dischi picoli.

11.3.2 LISTA CONCATENATA

Un altro approccio consiste nel collegare tutti blocchi liberi, tenendo un puntatore al primo di questi in una speciale locazione del disco e sottoponendola a caching in memoria. Questo primo blocco contiene l'indirizzo del secondo e così via. Questa schema non è comunque efficiente.

11.3.3 RAGGRUPPAMENTO

Una possibile modifica all'approccio della lista dei blocchi liberi prevede la memorizzazione degli indirizzi di n blocchi liberi nel primo di questi. I primi n-1 blocchi sono effettivamente liberi; l'ultimo blocco contiene gli indirizzi di altri n blocchi liberi e così via. L'importanza di questa implementazione, diversamente dall'approccio con lista

concatenata standard è data dalla possibilità di trovare rapidamente gli indirizzi di un gran numero di blocchi liberi.

11.3.4 CONTEGGIO

Anziché tenere una lista di n indirizzi liberi, è sufficiente tenere l'indirizzo del primo blocco libero e il numero n di blocchi liberi contigui che seguono il primo blocco. Ogni elemento della lista dei blocchi liberi è formato da un indirizzo del disco e un contatore. Anche se ogni elemento richiede più spazio di quanto en richieda un semplice indirizzo del disco, se il contatore è generalmente maggiore di 1 la lista globale risulta più corta.

11.4 REALIZZAZIONE DELLE DIRECTORY

11.4.1 LISTA LINEARE

La lista contiene i nomi dei file con dei puntatori ai blocchi di dati. L'individuazione di un elemento particolare richiede una ricerca lineare. Questo metodo è di facile programmazione ma la sua esecuzione è svantaggiosa in termini di tempo.
Può essere migliorata sottoponendo a caching le directory.

11.4.2 TABELLA HASH

Un'altra struttura dati utilizzata per implementare una directory è la tabella hash. Questa struttura di dati può diminuire notevolmente il tempo di ricerca nella directory. Le maggiori difficoltà legate a una tabella hash sono le dimensioni della tabella stessa, che in genere è fissa, e la dipendenza della funzione hash dalla dimensione della tabella hash. Bisogna inoltre gestire le collisioni (ad esempio con una lista concatenata).

11.5 EFFICIENZA E PRESTAZIONI

11.5.1 EFFICIENZA

L'uso efficiente di un disco dipende fortemente dagli algoritmi utilizzati per l'allocazione del disco e la gestione delle directory. Ad esempio, gli inode di Unix sono preallocati in una partizione. Anche un disco vuoto impiega una certa percentuale di spazio per gli inode. D'altra parte preallocandoli e distribuendoli lungo la partizione si migliorano le prestazioni del file system. Queste migliori prestazioni sono il risultato degli algoritmi di allocazione e di gestione dei blocchi liberi adottati da Unix, i quali cercano di mantenere i blocchi di dati di un file vicini al blocco che ne contiene l'inode allo scopo di ridurre il tempo di posizionamento.
Utilizzando i cluster inoltre aumenta la frammentazione interna. Per ridurre questa frammentazione BSD Unix varia la dimensione del cluster al crescere della dimensione del file. Cluster più grandi sono utilizzati laddove possono essere riempiti, cluster piccoli per file piccoli.
Anche il tipo di dati normalmente contenuti in un elemento di una directory devono essere tenuti in considerazione. Solitamente viene memorizzata la "data di ultima

scrittura", spesso la "data di ultimo accesso"… Il risultato di mantenere queste informazioni è che, ogniqualvolta un file viene letto, si dovrà aggiornare un campo della directory. Questa modifica richiede la lettura in memoria del blocco, la modifica della sezione e la riscrittura del blocco su disco. Quindi, ogni volta che un file viene aperto in lettura, anche l'elemento della directory ad esso associato deve essere letto e scritto. Questo requisito può risultare inefficiente per file a cui si accede frequentemente, quindi al momento della progettazione del file system è necessario considerare l'influenza sull'efficienza e sulle prestazioni di ogni informazione che si vuole associare a un file.

11.5.2 PRESTAZIONI

Alcuni sistemi riservano una sezione separata della memoria centrale come cache del disco, dove memorizzare i blocchi in previsione di un loro utilizzo entro breve tempo. LRU è un algoritmo general–purpose ragionevole per la sostituzione dei blocchi. Altri sistemi considerano tutta la memoria fisica non utilizzata come un insieme di buffer condiviso dal sistema di paginazione e dal sistema di caching dei blocchi di disco.
Alcuni sistemi ottimizzano la cache del disco adottando differenti algoritmi di sostituzione, a seconda del tipo di accesso del file. I blocchi di un file letto o scritto in modo sequenziale non dovrebbero essere rimpiazzati in ordine LRU, poiché il blocco utilizzato più di recente verrà utilizzato nuovamente per ultimo, o forse mai.
Diversamente gli accessi sequenziali potrebbero essere ottimizzate da tecniche note come rilascio indietro e lettura anticipata.
La prima rimuove un blocco dal buffer non appena di verifica una richiesta del blocco successivo; i blocchi precedenti con tutta probabilità non saranno più utilizzati e quindi sprecano spazio nel buffer. Con la tecnica di lettura anticipata vengono letti e posti nella cache il blocco richiesto e parecchi blocchi successivi; è probabile che questi blocchi verranno richiesti una volta terminata l'elaborazione del blocco corrente. Il recupero di questi blocchi con un unico trasferimento e la memorizzazione nella cache consentono di risparmiare una quantità di tempo considerevole.
Nei PC viene comunemente utilizzata anche una sezione di memoria come disco virtuale o disco ram; in questo caso l driver di un disco ram accetta tutte le operazioni standard dei dischi, eseguendole però in questa sezione della memoria anziché su un disco. Tutte le operazioni su disco possono essere eseguite su questo disco ram senza che gli utenti se ne accorgano, se non per la velocità eccezionalmente elevata. Sfortunatamente i dischi ram sono utili solamente come supporto temporaneo, in quanto volatili.
La differenza tra un disco ram e la cache di un disco è che il contenuto del primo è totalmente controllato dall'utente, mentre quello della cache è sotto il controllo del SO.

Recovery da failures

- Programmi di controllo della consistenza (*scandisk*, *fsck*): confrontano i metadati con i blocchi realmente allocati, cercando di risolvere le inconsistenze. Non sempre funzionano.
- Backup (automatico o manuale) dei dati dal disco ad altro supporto (altro disco, nastri, . . .)
- Recupero dei file perduti (o interi file system) dal backup

12.2.3 ACCESSO DIRETTO ALLA MEMORIA (DMA)

Quando un dispositivo effettua trasferimenti di grandi quantità di dati, come ad esempio nel caso di unità disco, l'uso di una costosa cpu per il controllo dei bit di stato e per la scrittura di dati sul registro del controller un byte alla volta, detta I/O programmato (PIO), sembra essere uno spreco. Molti computer evitano di sovraccaricare la cpu assegnando parte di questi compiti a un processore specializzato detto controller dell'accesso diretto in memoria (DMA). Per dare avvio a una trasferimento DMA, l'host scrive un comando strutturato DMA in memoria. Esso contiene un puntatore alla destinazione dei dati, e il numero dei byte da trasferire. La cpu scrive l'indirizzo di questo comando strutturato sul controller DMA, e prosegue con l'esecuzione di altro codice. Il controller DMA agisce quindi sul bus di memoria direttamente, presentando al bus gli indirizzi di memoria necessari per eseguire il trasferimento senza l'aiuto della cpu. Un semplice controller DMA è un componente standard dei pc, e le schede di I/O dette bus-mastering di un pc includono solitamente hardware DMA ad alta velocità.
La procedura di handshaking tra il controller del DMA e il controller del dispositivo si effettua grazie a una coppia di fili detti DMA-request e DMA-acknowledge.

12.3 INTERFACCIA DI I/O PER LE APPLICAZIONI

E' possibile astrarre dai dettagli delle differenze tra i dispositivi per l'I/O identificandone alcuni tipi generali. A ognuno di questi tipi sia ha accesso per mezzo di un unico insieme di funzioni (un'interfaccia). Le effettive differenze sono incapsulate in moduli del kernel detti driver dei dispositivi che sono specializzati negli specifici dispositivi, ma che comunicano con l'esterno per mezzo delle interfacce standard.
Lo scopo dello strato dei driver dei dispositivi è di nascondere al sottosistema di I/O del kernel le differenze fra i controller dei dispositivi, in modo simile a quello in cui le system call di I/O incapsulano il comportamento dei dispositivi di alcune classi generiche che nascondono alle applicazioni le differenze dell'hardware, Il fatto che così il

sottosistema di I/O sia reso indipendente dall'hardware semplifica il lavoro di hi sviluppa il SO, e va inoltre a vantaggio dei costruttori di hardware. Questi ultimi, infatti, o progettano i nuovi dispositivi in modo che siano compatibili con un interfaccia host-controller già esistente, oppure scrivono dei driver che permettano al nuovo hardware di essere gestito dai SO più diffusi. IN questo modo, i nuovi dispositivi sono utilizzabili da un computer senza che occorra attendere lo sviluppo del codice relativo da parte del produttore del SO.
I dispositivi possono differire per molteplici aspetti:
- trasferimento a flusso di caratteri o a blocchi: il primo trasferisce dati un byte alla volta, il secondo un blocco di byte in un'unica soluzione.
- Accesso sequenziale o casuale
- Dispositivi sincroni o asincroni: il primo trasferisce dati con un tempo di risposta prevedibile, il secondo no.
- Condivisibili o dedicati: il primo può essere usato in modo concorrente da più processi, il secondo no.
- Velocità di funzionamento
- Lettura e scrittura, sola lettura o sola scrittura.

La maggior parte dei sistemi ha anche una chiamata 'scappatoia' (escape o back-door) che permette ilo passaggio trasparente di comandi arbitrari da un'applicazione a un driver di dispositivo. In unix questa system call è ioctl (che sta per I/O control). Questa funzione permette a un'applicazione di utilizzare qualsiasi servizio fornito da ogni driver di dispositivo, senza che per questo sia necessario creare delle nuove system call. Gli argomenti di ioctl sono tre:
1) descrittore di file che individua il driver desiderato facendo riferimento a un dispositivo gestito da quel driver.
2) numero intero che seleziona uno dei comandi forniti dal driver.
3) puntatore a un'arbitraria struttura di dati in memoria, tramite le quale l'applicazione e il driver si scambiano ogni necessaria informazione.

12.3.1 DISPOSITIVI CON TRASFERIMENTO A BLOCCHI O A CARATTERI.

L'interfaccia per i dispositivi a blocchi sintetizza tutti gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati: comprende read, write, seek.
L'accesso mappato in memoria può essere posto ad un livello gerarchico immediatamente superiore a quello dei dispositivi a blocchi. Piuttosto che offrire funzioni di lettura e scrittura, un'interfaccia per l'accesso mappato in memoria fornisce la possibilità di utilizzare un'unità a disco tramite il vettore di byte della memoria centrale. La system call che associa un file a una regione di memoria restituisce l'indirizzo di memoria virtuale di un vettore di caratteri che contiene una copia del file. Gli effettivi trasferimenti di dati sono eseguiti solo quando sono necessari per soddisfare una richiesta d'accesso all'immagine in memoria.
La tastiera è una tipica esempio di dispositivo a flusso di caratteri, che ha le system call put e get. Basandosi su questa interfaccia è possibile costruire servizi aggiuntivi quali l'accesso riga per riga, la gestione di un buffer, la correzione (ad esempio quando l'utente preme il tasto backspace, il carattere precedente viene rimosso dalla sequenza di input).

12.3.2 DISPOSITIVI DI RETE

Poiché i modi di indirizzamento e le prestazioni tipiche dell'I/O di rete sono notevolmente differenti da quelle dell'I/O delle unità di disco, la maggior parte dei SO fornisce un'interfaccia per l'I/O diversa da quella del read-write-seek. Un'interfaccia disponibile su molti SO è l'interfaccia di rete socket (cioè presa di corrente).
Si pensi a una presa a muro di corrente elettrica: vi si può collegare qualunque apparecchiatura elettrica. Per analogia, le system call di un'interfaccia socket permettono a un'applicazione di creare una socket, colegare una socket locale all'indirizzo di un altro punto della rete, controllare se un'applicazione si sia inserita nella socket locale, e inviare e ricevere pacchetti di dati lungo la connessione. Per mettere lo sviluppo di server, l'interfaccia socket fornisce anche una funzione chiamata select che gestisce un insieme di socket. Essa restituisce informazioni sulle socket per le quali sono presenti pacchetti in attesa di essere ricevuti e su quelle che hanno spazio per accettare un pacchetto da inviare. L'uso della funzione select elimina l'interrogazione ciclica altrimenti necessaria per l'I/O di rete. Queste funzioni incapsulano il comportamento essenziale delle reti, facilitando notevolmente la creazione di applicazioni distribuite che possano sfruttare l'hardware e i protocolli di rete.

12.3.3 OROLOGI E TIMER

La maggior parte dei computer ha timer ed orologi hardware che forniscono tre funzioni essenziali:
1) Segnalare l'ora corrente
2) Segnalare il lasso di tempo trascorso
3) Regolare un timer in modo da avviare l'operazione X al tempo Y
L'hardware in grado di misurare la durata di un lasso di tempo a di avviare un operazione è detto timer programmabile. Questo meccanismo è usato dallo scheduler per generare un interrupt che sospende un processo quando il suo tempo è scaduto. E' usato dal sottosistema di I/O delle unità a disco per riversare periodicamente su disco il contento della buffer cache, e dal sottosistema di rete per annullare operazioni che procedono troppo lentamente a causa di congestionamenti o di fallimenti.
In molti computer, la frequenza degli interrupt generati dall'orologio hardware è fra i 18 e i 60 al secondo.

12.3.4 I/O BLOCCANTE E NON BLOCCANTE

Quando un'applicazione effettua una system call bloccante, l'esecuzione dell'applicazione è sospesa. Essa è spostata dalla coda run alla coda wait del sistema. Nel momento in cui la system call termina, l'applicazione è posta nuovamente nella coda run in modo che possa riprendere l'esecuzione; solo allora riceverà i valori di ritorno della system call. Le operazioni fisiche compiute dai dispositivi di I/O sono in genere asincrone - richiedono un tempo variabile e non prevedibile. Ciò nondimeno, la maggior parte dei SO impiega system call bloccanti come interfaccia per le applicazioni, perché in questo modo il

codice delle applicazioni è più facilmente comprensibile del corrispondente codice non bloccante.
Alcuni processi a livello utente necessitano di una forma di I/O non bloccante. Un esempio è quello di un interfaccia utente che riceva input dal mouse e dalla tastiera mentre elabora dati e mostra output sullo schermo. Un altro esempio è un'applicazione che legge fotogrammi da un file su un unità a disco e simultaneamente li decomprime e li mostra sul video.
Uno dei modi in cui chi progetta un'applicazione che può sovrapporre elaborazione e I/O è di scrivere un'applicazione a più thread. Alcuni di essi eseguono system call bloccanti, mentre altri continuano l'elaborazione.
Alcuni SO forniscono system call non bloccanti per l'I/O. Una chiamata d questo tipo non arresta l'esecuzione dell'applicazione per un lasso di tempo significativo. Al contrario, essa restituisce rapidamente il controllo all'applicazione, fornendo un parametro che indica quanti byte di dati sono stati trasferiti.
Una possibile alternativa alle system call non bloccanti è rappresentata dalle system call asincrone. Esse restituiscono immediatamente il controllo al chiamante, senza attendere che l'I/O sia stato completato. L'applicazione continua ad essere eseguita, e il completamento dell'I/O è successivamente comunicato all'applicazione per mezzo dell'impostazione del valore di una variabile nello spazio di indirizzamento dell'applicazione o tramite la generazione di un interrupt software, o ancora tramite una routine di ritorno eseguita al di fuori del normale flusso lineare di elaborazione dell'applicazione.

12.4 Sottosistema per l'I/O del kernel

12.4.1 Scheduling

Effettuare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine di esecuzione efficace. L'ordine in cui si verificano le system call delle applicazioni è raramente la scelta migliore. Lo scheduling può migliorare le prestazioni globali del sistema, distribuire equamente gli accessi dei processi ai dispositivi e ridurre il tempo di attesa medio per il completamento di un operazioni di I/O. I progettisti di SO realizzano lo scheduling mantenendo una coda di richieste per ogni dispositivo. Quando una'applicazione impartisce una system call di I/O bloccante, la richiesta è aggiunta alla coda relativa al dispositivo appropriato. Lo scheduler dell'I/O riorganizza l'ordine della coda per migliorare l'efficienza globale del sistema e il tempo medio di attesa cui sono sottoposte le applicazioni. Il SO può anche tentare di essere equo, in modo che nessuna applicazione riceva un servizio carente, o può dare priorità a quelle richieste la cui corretta esecuzione potrebbe essere inficiata da un ritardo nel servizio.

12.4.2 Buffering

Un buffer è una regione di memoria che contiene dei dati mentre essi non trasferiti tra due dispositivi o tra un'applicazione e un dispositivo. La tecnica del buffering è utilizzata per tre ragioni:

1) Necessità di gestire la differenza di velocità tra il produttore e il consumatore di un flusso di dati. Si può usare anche un doppio buffer: uno viene riempito, quando è saturo il consumatore lo vuota mentre il produttore riempie l'altro.
2) Gestione dei dispositivi che gestiscono blocchi di dati di dimensioni diverse.
3) Uso per la semantica di copia per l'I/O. Si supponga che un'applicazione disponga di un buffer di dati da trasferire sul disco: essa impartisce la system call write, fornendole un puntatore al buffer, e un numero intero che specifichi il numero di byte da trasferire. Ci si può chiedere che cosa succeda se, dopo che la system call restituisce il controllo all'applicazione, quest'ultima modifica il contenuto del buffer. Ebbene, la semantica di copia garantisce che la versione dei dati che è scritta su disco è conforme a quella contenuta nel buffer al momento della system call, indipendentemente da ogni successiva modifica. Una semplice maniera di realizzare questa semantica consiste nel far sì che la system call write copi i dati forniti dall'applicazione in un buffer del kernel prima di restituire il controllo all'applicazione stessa. La scrittura su disco è effettuata nel buffer del kernel, cosicché ogni successivo cambiamento del buffer dell'applicazione non impedirà che la versione dei dati trasferita su disco sia quella corretta. In molti SO si usa il metodo sopra descritto: nonostante esso implichi una diminuzione dell'efficienza di certe operazioni di I/O, la sua semantica è chiara. Lo stesso effetto, tuttavia, può essere ottenuto più efficientemente tramite un uso intelligente della memoria virtuale e della protezione data dalla copia su scrittura delle pagine.

12.4.4 Spooling e uso esclusivo dei dispositivi

Uno spool è un buffer contenente output per un dispositivo che non può accettare flussi interfogliati di dati, come ad esempio una stampante. Sebbene una stampante possa servire una sola richiesta alla volta, diverse applicazioni devono poter richiedere contemporaneamente la stampa del loro output, senza che i loro dati siano mischiati. Il SO risolve questo problema filtrando tutto l'output per la stampante: l'output di ogni singola applicazione è memorizzato in un determinato file su disco detto file di spool. Quando un'applicazione termina di generare output, il sistema di spooling aggiunge il relativo file di spool alla coda di stampa; quest'ultima è copiata sulla stampante, un file per volta. In certi SO, lo spooling è gestito da un processo di sistema specializzato (daemon), in altri da un thread del kernel.
Alcuni dispositivi, come le unità a nastro e le stampanti, non sono di solito in grado di gestire autonomamente più richieste di I/O da parte di diverse applicazioni. Lo spooling è uno dei modi in cui il SO può coordinare l'output simultaneo; un altro è quello di fornire esplicite funzioni di coordinamento.

12.4.5 Gestione degli errori

Il SO unix usa una variabile interna detta errno per codificare piuttosto genericamente il tipo di errore avvento; i possibili valori sono un centinaio.
Per contro, alcuni tipi di hardware sono in grado di fornire informazioni molto dettagliate sugli errori (vedi le unità scsi).

12.4.6 STRUTTURE DI DATI DEL KERNEL

Il kernel ha bisogno di mantenere informazioni sullo stato di componenti coinvolti nelle operazioni di I/O, e usa a questo fine diverse strutture di dati interne.
Unix, per mezzo del file system permette l'accesso a diversi oggetti: i file degli utenti, i dispositivi, lo spazio di indirizzamento dei processi... Sebbene ognuno di questi oggetti sia in grado di effettuare una chiamata read, le semantiche sono diverse a seconda dei casi. Quando il kernel, ad esempio, deve leggere un file d'utente, ha bisogno di controllare la buffer cache prima di decidere l'effettiva esecuzione di un operazioni di I/O su disco. Per leggere un disco tramnite un'interfaccia raw, il kernel deve accertarsi del fatto che la dimensione dell'insieme dei dati di cui è stato richiesto il trasferimento sia un multiplo della dimensione dei settori del disco e sia allineato con il settore interessato. Per leggere l'immagine di un processo, tutto ciò che occorre è copiare i dati dalla memoria. Unix incapsula queste differenze in una struttura uniforme usando una tecnica orientata agli oggetti. il record dei file aperti contiene una tabella dei puntatori alle routine più appropriate a seconda del tipo di file in questione.
Alcuni SO applicano metodi orientati agli oggetti in misura più rilevante: windows NT usa un sistema per l'I/O basato sullo scambio di messaggi. Una richiesta di I/O è convertita in un messaggio che è inviato tramite il kernel al sottosistema per la gestione dell'I/O, quindi al driver del dispositivo-, i contenuti del messaggio possono essere modificati a ogni passaggio intermedio. Questo approccio può implicare minore efficienza rispetto alle tecniche procedurali basate sulla condivisione di strutture di dati, ma semplifica il progetto e la struttura del sistema di I/O e permette una maggiore flessibilità.
Riassumendo, il sistema per l'I/O coordina un'ampia raccolta di servizi disponibili per le applicazioni e per altre parti del kernel. Esso sovrintende alla seguenti funzioni:
- gestione dello spazio dei nomi per file e dispositivi
- controllo dell'accesso ai file e ai dispositivi
- controllo delle operazioni (ad esempio un modem non può eseguire una chiamata seek)
- Allocazione dello spazio per il file system
- Allocazione dei dispositivi
- Buffering, cahing e spooling
- Scheduling dell'I/O
- Controllo dello stato dei dispositivi, gestione degli errori e procedura di ripristino
- Configurazione e inizializzazione dei driver dei dispositivi.

12.5 TRASFORMAZIONE DELLE RICHIESTE DI I/O IN OPERAZIONI DELL'HARDWARE

Come il sistema associa alla richiesta di un applicazione un insieme di fili di rete o uno specifico settore del disco?
Si consideri ad esempio la lettura di un file dall'unità disco. L'applicazione fa riferimento ai dati per mezzo del nome del file: sarà compito del file system fornire il modo di giungere, attraverso la struttura delle directory, alla regione del disco appropriata, cioè quella dove i dati del file sono fisicamente residenti. In ms-dos il nome del file è associato a un numero che individua un elemento della tabella d'accesso ai file, tale elemento identifica i blocchi del disco associati al file. In Unix, il nome è associato a un

elemento di una lista di oggetti detti numeri di Inode, l'inode corrispondente contiene le informazioni necessarie per individuare lo spazio allocato.
per illustrare come si possa giungere dal nome del file al controller del disco (all'indirizzo della sua porta hardware o ai suoi registri mappati in memoria), conviene innanzitutto esaminare un sistema relativamente semplice come ms-dos. La prima parte di un nome di file, precisamente la parte che precede i due punti, è una stringa che identifica uno specifico dispositivo hardware (ad esempio C:). Questa convenzione è codificata all'interno del SO: c: è associato a uno specifico indirizzo di porta per mezzo di una tabella dei dispositivi. Grazie all'uso dei due punti come separatore, lo spazio per il nome dei dispositivi è distinto dallo spazio per il nome del file, ciò semplifica il SO l'associazione di funzionalità aggiuntive ai dispositivi. Ad esempio, è facile richiedere lo spooling dei file mandati in stampa.
Se, invece, i nomi dei dispositivi sono inclusi nell'ordinario spazio dei nomi dei file, come in unix, i servizi dei file system riguardanti il nome del file sono automaticamente disponibili. Ad esempio, se il file system associa dei possessori ai nomi dei file e fornisce il controllo dell'accesso a ogni nome del file, allora anche l'accesso ai dispositivi potrà essere controllato ed essi avranno un possessore. visto che i file risiedono sui dispositivi, una tale interfaccia fornisce due livelli di accesso al sistema di I/O: i nomi possono essere usati per accedere ai dispositivi stessi o ai file che essi contengono.
Unix rappresenta i nomi dei dispositivi all'interno dell'ordinario spazio dei nomi del file system. Nessuna parte del path name di un file è il nome di un dispositivo: Unix impiega una particolare tabella, detta tabella di montaggio, per associare prefissi di path name a specifici nomi di dispositivo. Quando deve risolvere un path name, il sistema esamina la tabella per trovare il più lungo prefisso corrispondente: questo elemento della tabella indicherà allora il nome del dispositivo voluto. Anche questo nome, però, è rappresentato come un oggetto del file system: tuttavia, quando Unix cerca questo nome nelle strutture di directory del file system, non trova il numero di un inode, ma una coppia di numeri <principale, secondario> (<major, minor>) che identifica un dispositivo. Il numero principale individua il driver di dispositivo che deve essere usato per gestore l'I/O sul dispositivo in questione: mentre il numero secondario deve essere passato a questo driver affinché esso possa determinare, per mezzo di un apposita tabella, l'indirizzo della porta o l'indirizzo mappato in memoria del controller del dispositivo interessato.
Unix System V include un interessante meccanismo, detto flusso (stream), che permette a un'applicazione di costruire dinamicamnete liste di codice per la gestione dei dispositivi. Un flusso è una connessione full-duplex fra un driver di dispositivo w un processo utente, e consiste di un intestazione del flusso (stream head) che funge da interfaccia per il processo utente, una terminazione del driver (driver end) che controlla il dispositivo, ed eventualmente un certo numero di moduli del flusso (stream modules) fra questi due estremi. A un flusso possono essere aggiunti moduli per realizzare nuove funzionalità secondo uno schema stratificato: un processo, ad esempio, può accedere tramite un flusso a un dispositivo collegato alla porta seriale, e può aggiungere un modulo per controllare la manipolazione dell'input. I flussi possono anche essere usati per la comunicazione tra i processi e la rete: in effetti, il meccanismo delle socket è realizzato in system V per mezzo dei flussi.

La seguente descrizione del tipico svolgimento di una richiesta di lettura bloccante indica che l'esecuzione di un'operazione di I/O richiede una grande quantità di passi, ciò implica l'uso di un enorme numero di cicli di cpu.
1) Un processo impartisce una system call read bloccante relativa a un descrittore di file di un file precedentemente aperto
2) il codice della system call all'interno del kernel controlla la correttezza dei parametri. Se già presenti nella buffer cache, i dati richiesti sono passati al processo chiamante e l'operazione è conclusa.
3) Altrimenti, è necessario effettuare un'operazione fisica di I/O, così il processo è rimosso della run queue e posto sulla wait queue relativa al dispositivo interessato ed è effettuato lo scheduling della richiesta di I/O. Infine il sottosistema di I/O invia la richiesta al driver del dispositivo: a seconda del SO, ciò avviene tramite la chiamata di una subroutine, o per mezzo di emissione di messaggi all'interno del kernel.
4) Il driver del dispositivo alloca un buffer nello spazio di indirizzamento del kernel che serve per ricevere i dati in input, ed effettua lo scheduling dell'I/O. Infine il driver impartisce i comandi al controller del dispositivo scrivendo sui suoi registri.
5) Il controller del dispositivo aziona l'hardware per eseguire il trasferimento dei dati.
6) il driver può eseguire un'interrogazione ciclica, o può aver predisposto un trasferimento DMA nella memoria del kernel. Si assume che il trasferimento sia gestito dal controller DMA, il quale genera un interrupt ad operazione terminata.
7) L'appropriato gestore degli interrupt è attivato per mezzo del vettore di interrupt: dopo aver memorizzato i dati necessari, avverte con un segnale il driver del dispositivo e restituisce il controllo
9) Il kernel trasferisce i dati (in caso di successo) e/o codici di stato (per comunicare, ad esempio, la mancata riuscita) nello spazio di indirizzamento del processo chiamante, e lo sposta dalla coda wait alla coda ready.
10) Nel momento in cui è posto sulla coda ready, il processo non è più bloccato: quando lo scheduler gli assegnerà la cpu, esso riprenderà l'elaborazione: l'esecuzione della system call è terminata.

12.6 PRESTAZIONI

Per migliorare l'efficienza dell'I/O si possono applicare diversi principi:
- Ridurre il numero di context switch (necessari in caso di interrupt)
- RIdurre il numero di copiature dei dati in memoria durante i trasferimenti fra dispositivi e applicazioni
- Ridurre la frequenza degli interrupt tramite il trasferimento di grandi quantità di dati in un'unica soluzione, l'uso di controller intelligenti, e l'interrogazione ciclica (nel caso in cui i tempi di attesa passivi possano essere minimizzati).
Aumentare il tasso di concorrenza usando controller DMA intelligenti i canali di I/O per sollevare la cpu dal compito di effettuare semplici copiature di dati
- Realizzare le primitive direttamente tramite hardware, così da permettere che la loro esecuzione sia simultanea alle operazioni di bus e di cpu.
- Equilibrare le prestazioni della cpu, del sottosistema per la gestione della memoria, del bus e dell'I/O, giacché il sovraccarico di uno qualunque di questi settori provoca l'inutilizzo degli altri.

13 MEMORIA SECONDARIA

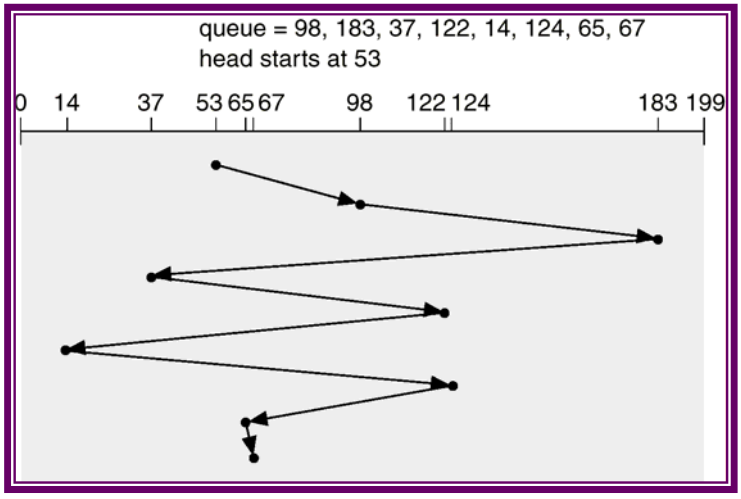
13.1 STRUTTURA DEL DISCO

Dal punto di vista dell'indirizzamento, i moderni dischi sono considerati come un vettore monodimensionale di blocchi logici, dove un blocco logico è la minima unità di trasferimento. La dimensione di un blocco logico è solitamente di 512 byte, sebbene alcuni dischi possono essere formattati a basso livello allo scopo di ottenere una diversa dimensione dei blocchi logici, ad esempio 1024 byte.

13.2 SCHEDULING DEL DISCO

Il tempo di accesso si può scindere in due componenti: seek time e rotational latency. Il bandwith è il numero di byte trasferiti per tempo.
Per mezzo dello scheduling delle richieste di I/O relative al disco si possono migliorare sia il tempo di accesso che l'ampiezza di banda.
Ogni volta che un processo deve effettuare dell'I/O con un unità a disco, esso impartisce una system call al SO. La richiesta contiene diverse informazioni:
- se l'operazione sia di input o di output.
- l'indirizzo su disco in base al quale effettuare il trasferimento
- l'indirizzo di memoria in base al quale effettuare il trasferimento
- il numero di byte da trasferire

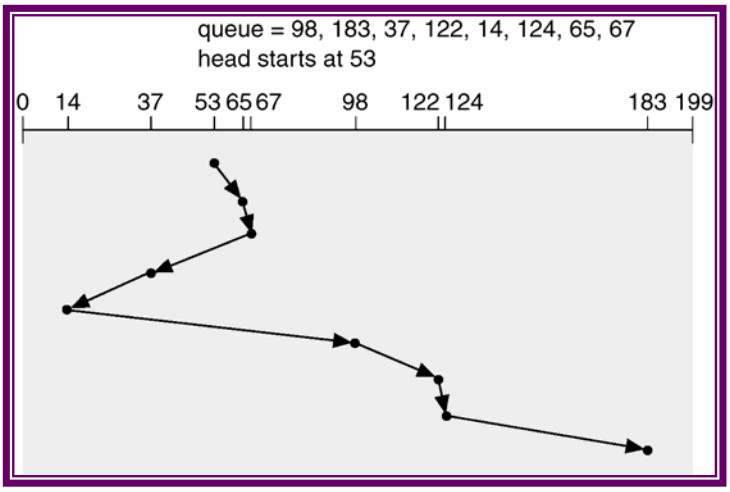
Se l'unità a disco desiderata e il controller sono disponibili, la richiesta può essere immediatamente soddisfatta; altrimenti le nuove richieste devono essere aggiunte alla coda delle richieste inevase relative a quell'unità. La coda relativa a unità disco in un sistema multiprogrammato può essere piuttosto lunga, cosicché il SO ha l'opportunità di scegliere quale tra le richieste inevase conviene servire prima.



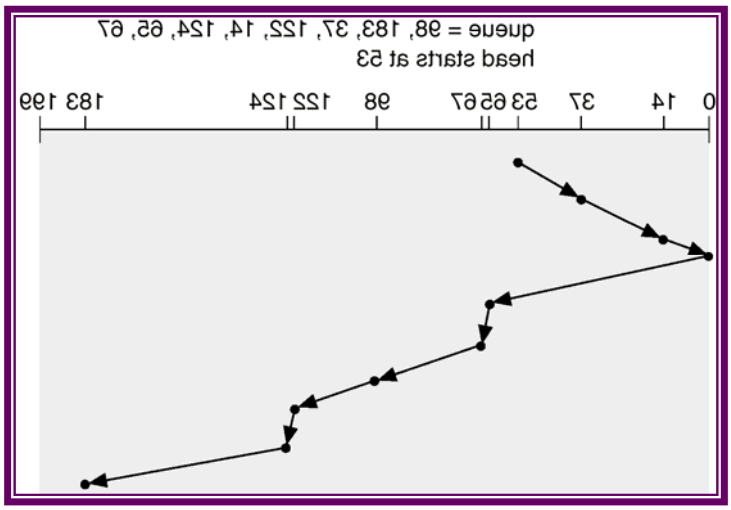
13.2.1 SCHEDULING FCFS

Si tratta di un algoritmo intrinsecamente equo ma lento:
Si suppone che la coda delle richieste sia:
98, 183, 37, 122, 14, 124, 65, 67

13.2.2 SCHEDULING SSTF

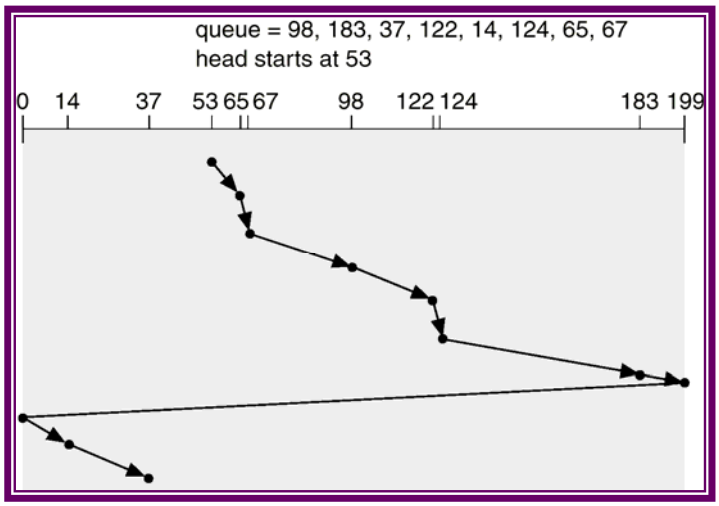


Sembra ragionevole servire tutte le richieste vicine alla posizione corrente della testina prima di spostarla in aree lontane per soddisfarne altre: questa considerazione è alla base dell'algoritmo SSTF (shortest seek time first). Esso sceglie la richiesta che dà il minimo tempo di ricerca rispetto all'attuale posizione della testina: poiché questo tempo cresce al crescere della distanza dei cilindri della testine, l'algoritmo sceglie le richieste relative ai cilindri più vicini alla posizione della testina. Lo scheduling SSTF è essenzialmente una forma di scheduling SJF e, al pari di esso, può indurre un blocco indefinito (starvation) di alcune richieste.



13.2.3 SCHEDULING SCAN

Secondo l'algoritmo SCAN (cioè attraversamento o scansione) il braccio dell'unità a disco parte da un estremo del disco e si sposta nella sola direzione possibile, servendo le richieste mentre attraversa i cilindri, fino a che non giunge all'altro estremo del disco: a questo punto il braccio inverte la marcia e la procedura continua. Le testine attraversano il disco continuamente in entrambe le direzioni.



13.2.4 SCHEDULING C-SCAN

L'algoritmo SCAN circolare (C-SCAN) è una variante dello scheduling SCAN concepita per garantire un tempo d'attesa meno variabile. Anche C-SCAN sposta la testina da un estremo all'altro

del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all'altro estremo del disco, ritorna immediatamente all'inizio del disco stesso, senza servire richieste durante il viaggio di ritorno. L'algoritmo C-SCAN essenzialmente tratta il disco come una lista circolare, cioè come se il primo cilindro e l'ultimo fossero adiacenti.

13.2.5 SCHEDULING LOOK E C-LOOK

Essi anziché andare dall'inizio alla fine del disco, vanno dalla richiesta del cilindro più basso a quella più alto, senza arrivare fino alla fine (o all'inizio) del disco.

13.2.6 SCELTA DI UN ALGORITMO DI SCHEDULING

SCAN e C-SCAN danno migliori prestazioni in sistemi che sfruttano molto le unità a disco, perché conducono con minor probabilità ad un blocco indefinito. Per una data ma arbitraria lista di richieste è sempre possibile definire un ordine ottimale di servizio, ma la computazione richiesta può non essere giustificata dal miglioramento in prestazioni rispetto a SSTF o a SCAN.
Si noti che le richieste di I/O per le unità a disco possono essere notevolmente influenzate dal metodo adottato per l'allocazione dei file. Un programma che legga un file allocato contiguamente genererà molte richieste raggruppate, con un conseguente limitato spostamento della testina. Un file con allocazione concatenata o indicizzata, d'altro canto, potrebbe includere blocchi sparsi su tutto il disco, e richiedere quindi maggiore movimento della testina.
Anche la posizione delle directory e dei blocchi indice è importante: poiché ogni file deve essere aperto per essere usato, e visto che l'apertura di un file richiede una ricerca attraverso la struttura delle directory, vi saranno frequenti accessi alle directory. Si supponga che un elemento di directory risieda sul primo cilindro e che io dati relativi al file in questione si trovino sull'ultimo cilindro; la testina dovrà allora percorrere l'intera ampiezza del disco in caso di apertura del file in questione. Se l'elemento di directory che rappresenta logicamente il file fosse sul cilindro di mezzo, la testina dovrebbe spostarsi al più di metà dell'ampiezza. Anche il caching in memoria centrale delle directory e dei blocchi indice può aiutare a ridurre i movimenti del braccio delle unità a disco, in particolare quando si tratta di operazioni di lettura.
E' difficile tuttavia per il SO adottare una strategia di scheduling che porti a miglioramenti di tempi di latenza rotazionale, perché le moderne unità a disco non rendono trasparente la posizione fisica dei blocchi logici.

15 STRUTTURE DELLE RETI DI COMUNICAZIONE

Recentemente si è diffusa la tendenza a distribuire il calcolo tra diversi processori fisici. Sistemi di questo tipo possono essere tightly coupled (multiprocessore) o loosely coupled (distribuito).

15.1 INTRODUZIONE

Un sistema distribuito è un insieme di processori loosely coupled interconnessi tramite una rete di comunicazione. IL processore specifico di un sistema distribuito considera remoti gli altri processori del sistema e le rispettive risorse, mentre considera locali le proprie risorse.
I processori di un sistema distribuito possono variare per dimensioni e per funzioni; possono comprendere microprocessori, stazioni di lavoro, minicomputer e grandi computer general purpose. Questi processori vengono chiamati in molti modi: siti, nodi, macchine, host...
Un SO distribuito offre agli utenti l'accesso a varie risorse di cui il sistema dispone. Con il termine risorse vengono indicate sia le risorse hardware, come stampanti e unità a nastri, che le risorse software, come file e programmi. L'accesso a queste risorse è controllato dal SO. Fondamentalmente esistono due schemi complementari che garantiscono questo servizio:
- SO di rete. Gli utenti sono a conoscenza delle numerose macchine presenti e devono accedere a queste risorse effettuando un login nella macchina remota appropriata, oppure trasferendo dati dalla macchina remota alle loro macchine
- SO disrtibuiti. Gli utenti non hanno bisogno di essere a conoscenza delle numerose macchine presenti; essi accedono alle risorse remote nello stesso modo in cui accedono a quelle locali.

15.2 MOTIVAZIONE

15.2.1 CONDIVISIONE DELLE RISORSE

Se siti diversi, con risorse diverse, sono collegati tra loro, allora l'utente di un sito può avere la possibilità di utilizzare le risorse disponibili su un altro sito.

15.2.2 ACCELERAZIONE DEI CALCOLI

Se un calcolo particolare può essere suddiviso in più sottocalcoli eseguibili concorrentemente, disponendo di un SO distribuito è possibile distribuire la computazione tra i diversi siti per un'esecuzione concorrente. Inoltre, se un particolare sito è attualmente sovraccarico di job, alcuni di essi possono essere spostati su siti con carico minore. Lo spostamento dei job è chiamato condivisione di carico. Una distribuzione del carico automatizzata, in cui il SO sposta i job automaticamente, non è ancora comune nei sistemi commerciali ma è comunque un'attività di ricerca.

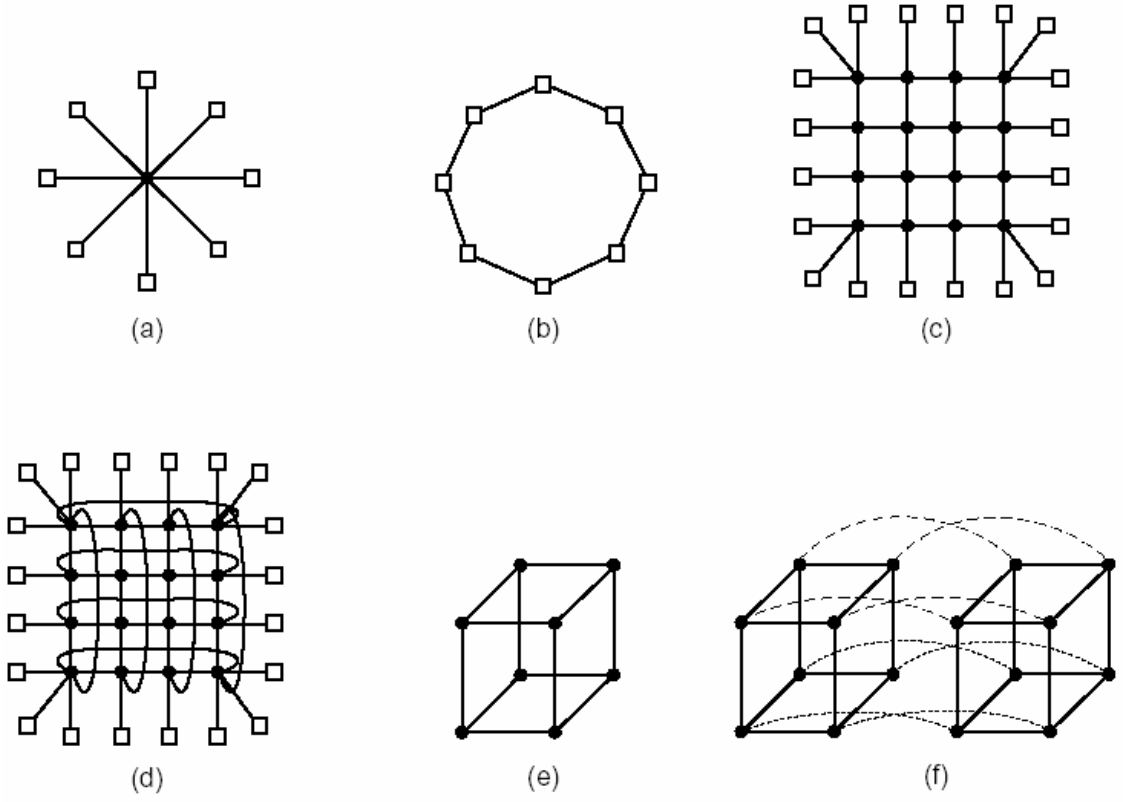
15.2.3 AFFIDABILITÀ

Se un sito di un sistema distribuito si guasta, i restanti possono potenzialmente continuare a lavorare. In generale, se il sistema è abbastanza ridondante (sia a livello hardware, sia a livello di dati), può continuare a funzionare anche se qualcuno dei siti si guasta.

15.2.4 COMUNICAZIONE

Quando più siti sono collegati per mezzo di una rete di comunicazione, gli utenti dei diversi siti hanno la possibilità di scambiarsi informazioni. A un livello basso, tra i sistemi vengono scambiati messaggi in modo simile a quello dei messaggi di un singolo computer. Disponendo dello scambio di messaggi, tutta la funzionalità di livello superiore disponibile nei sistemi centralizzati può essere estesa in modo da adattarsi al sistema distribuito. Tali funzioni comprendono trasferimento di file, login, posta elettronica e chiamate di procedura remota (RPC).
Considerati nel loro complesso, i vantaggi dei sistemi distribuiti hanno determinato una tendenza, tra le industrie, verso il ridimensionamento: molte aziende sostituiscono i loro mainframe con reti di stazioni di lavoro o pc. I vantaggi per le aziende sono un miglior rapporto prezzo-prestazioni, una maggior flessibilità nella dislocazione delle risorse e un aumento delle funzionalità, migliori interfacce utente e manutenzione più semplice.

15.3 TOPOLOGIA



15.4 TIPI DI RETE

Esistono due tipi fondamentali: le reti locali (local-area network) e le reti geografiche (wide-area network. La principale differenza tra le due consiste nella loro distribuzione geografica. Le reti locali sono formate da processori distribuiti su piccole aree geografiche, come un unico edificio o alcuni edifici adiacenti. Le reti geografiche, invece,

sono formate da un certo numero di processori autonomi distribuiti su una vasta area geografica, come l'Italia. Queste differenze implicano notevoli variazioni relativamente alla velocità e all'affidabilità delle reti di comunicazione, e si riflettono nel progetto del SO distribuito.

15.4.1 Reti locali

Le reti locali sono nate nei primi anni '70 per sostituire i mainframe. Generalmente le LAN sono progettate per coprire piccole aree geografiche e di solito vengono utilizzate in ambienti di uffici. In questi sistemi tutti i siti sono vicini fra loro, i collegamenti tendono ad avere una maggiore velocità e una minore frequenza di errori rispetto alle reti geografiche. Per ottenere tale velocità e affidabilità sono necessari cavi di elevata qualità. Inoltre è possibile utilizzare i cavi esclusivamente per il traffico dei dati sulla rete. Su lunghe distanze il costo dovuto all'utilizzo di cavi di elevata qualità è enorme, e l'uso esclusivo di cavi tende a diventare proibitivo.
Nelle reti locali i collegamenti sono in genere realizzati con cavi a doppino intrecciato e cablaggi con fibre ottiche. Le configurazioni più diffuse sono le reti con bus multiaccesso, ad anello e a stella. La velocità di comunicazione varia dall'ordine del megabit al secondo fino a un gigabit al secondo. Per costruire le lan viene generalmente utilizzato uno schema ethernet. In una rete ethernet non c’è un controller centrale poiché è una rete con bus multiaccesso, quindi nuovi host possono essere facilmente aggiunti alla rete.

15.4.2 Reti geografiche

Le reti geografiche sono nate alla fine degli anni '60 (Arpanet) come progetto di ricerca accademiche per fornire un sistema di comunicazione efficiente tra siti, permettendo a un'ampia comunità di utenti di condividere in modo conveniente ed economico hardware e software.
Poiché i siti di una wan sono fisicamente distribuiti su una vasta area geografica, i collegamenti per le comunicazioni sono per loro natura lenti ed inaffidabili. I collegamenti tipici sono le linee telefoniche, i collegamenti a microonde e i canali via satellite. Questi collegamenti sono controllati da speciali processori di comunicazione responsabili della definizione dell'interfaccia attraverso la quale i siti comunicano sulla rete,nonché del trasferimento delle informazioni tra i diversi siti.
Si consideri, ad esempio, la wan internet. Il sistema offre agli host in siti geograficamente separati la possibilità di comunicare tra loro. Generalmente i computer host differiscono per tipo, velocità, lunghezza delle parole, SO... Gli host vengono inseriti in LAN, che a loro volta sono collegate a internet per mezzo di reti regionali-. Le reti regionali sono interconnesse tramite router per formare la rete mondiale. I collegamenti tra le reti utilizzano spesso un servizio telefonico, T1, che fornisce una velocità di trasferimento di 1,544 megabit al secondo su linea noleggiata. Per siti che richiedono un accesso a internet più veloce, i T1 sono raccolti in unità di più T1 che lavorano in parallelo per fornire una maggiore produttività. Ad esempio, T3 è composto da 28 connessioni T1 ed ha una velocità di trasferimento di 45 megabit al secondo. I router controllano i percorsi seguiti da ogni messaggio attraverso la rete; questo instradamento può essere dinamico, per

aumentare l'efficienza delle comunicazioni, oppure statico, per ridurre i rischi per la sicurezza o per permettere di calcolare le spese di comunicazione.
Altre wan utilizzano linee telefoniche standard come mezzo di comunicazione principale. I modem sono dei dispositivi che ricevono dati digitali da un computer e li convertono nei segnali analogici delle linee telefoniche. Le wan sono generalmente più lente delle LAN; le loro velocità di trasmissione variano da 1200 bit al secondo a oltre 1 megabit al secondo.

15.5 COMUNICAZIONE

Dopo aver trattato gli aspetti fisici della comunicazione su reti, è possibile considerarne il funzionamento intero. Il progettista di una rete di comunicazione deve considerare quattro problemi fondamentali:
- Nominazione e risoluzione dei nomi. Trattano di come due processi si individuano l'un l'altro per comunicare.
- Strategie di instradamento. Trattano di come vengono inviati messaggi attraverso la rete.
- Strategie riguardanti i pacchetti. Trattano l'invio individuale o in sequenza di pacchetti.
- Strategie di connessione. Trattano dello scambio di sequenze di messaggi fra due processi.
- Contesa. Riguarda i problemi di conflittualità legati all'utilizzo della rete come risorsa condivisa.

15.5.1 NOMINAZIONE E RISOLUZIONE DEI NOMI

I processo di un sistema remoto sono generalmente identificati dalla coppia <nome dell'host, identificatore>, dove nome dell'host è solitamente un nome unico all'interno della rete, e identificatore può essere un process-id o un altro numero unico all'interno dell'host.
Un nome dell'host è solitamente un identificatore alfanumerico, anziché un numero per rendere più semplice l'utilizzo da parte degli utenti.
Sebbene i nomi siano comodi per gli esseri umani, i calcolatori preferiscono fare uso dei numeri, per motivi sia di velocità che di semplicità. Per questa ragione ci deve essere un meccanismo per la risoluzione del nome dell'host nel corrispondente host-id che descrive il sistema di destinazione all'hardware della rete.
Si presentano due possibilità:
1) ciascun host può avere dei file di dati contenente tutti i nomi e gli indirizzi di tutti gli host raggiungibili dalla rete. Il problema di questo modello è che l'aggiunta o la rimozione di un host dalla rete richiede l'aggiornamento su tutti gli host dei suddetti file di dati.
2) Distribuire le informazioni tra i sistemi connessi alla rete, la quale quindi deve adoperare un protocollo per distribuire e recuperare queste informazioni.
Il primo metodo è quello originariamente adottato da internet, che ora, a causa della sua inarrestabile crescita, adotta il secondo metodo chiamato domain name service (DNS). DNS specifica la struttura di nominazione degli host, nonché la risoluzione dei nomi in indirizzi. Gli host di internet vengono indirizzati logicamente con un nome composto, le cui componenti progrediscono dalla parte più specifica a quella più generale dell'indirizzo,

con i diversi campi separati da un punto. Ad esempio, "bob.cs.brown.edu" si riferisce all'host bob del Department of computer Science della Brown University. In generale, il sistema risolve l'indirizzo esaminando le componenti del nome dell'host in ordine inverso. Ciascuna componente ha un server dei nomi (semplicemente un processo sul sistema) che accetta un nome e restituisce l'indirizzo dl server dei nomi responsabile per la risoluzione di quel nome. Come passo finale viene contattato il server dei nomi associato all'host in questione, il quale provvederà a restituire l'host-id.
Per quanto riguarda l'esempio precedente, una richiesta inoltrata da un processo A di comunicare con l'host bob.cs.brown.edu comporterebbe l'esecuzione dei seguenti passi:
1) Il kernel del sistema A inoltra una richiesta al server dei nomi del dominio .edu richiedendo l'indirizzo del server dei nomi brown.edu. Per poter essere interrogato, il server dei nomi .edu deve essere noto.
2-3-4) Il server dei nomi di edu restituisce l'indirizzo dell'host sul quale risiede il server dei nomi brown.edu. Quindi ugualmente per cs.brown.edu e per bob.cs.brown.edu, che restituirà però l'indirizzo internet di quel sistema.

15.5.2 STRATEGIE DI INSTRADAMENTO

In questo paragrafo è descritto il modo in cui viene trasmesso un messaggio inviato da un processo del sito A che vuole comunicare con un processo del sito B. Se tra A e B esiste un solo percorso fisico, come in una rete a stella o gerarchica, il messaggio deve passare per quel percorso, ma se i percorsi fisici da A a B sono più di uno esistono diverse opzioni di instradamento. Ogni sito ha una tabella di instradamento che indica i percorsi che possono essere eseguiti per inviare un messaggio ad altri siti.
La tabella può contenere informazioni sulla velocità e sul costo dei diversi percorsi di comunicazione e, in caso di necessità, può anche essere aggiornata manualmente oppure tramite programmi che scambiano informazioni di instradamento. I tre schemi di instradamento più diffusi sono:
1) Instradamento fisso: un percorso da A a B viene specificato in anticipo e non cambia a meno che si verifichi un guasto hardware che disabiliti il percorso stesso. Generalmente viene scelto il percorso più breve, in modo da minimizzare i costi di connessione
2) Circuito virtuale: viene fissato un percorso da A a B per la durata di una sessione. Sessioni diverse con messaggi che vanno da A a B possono avere percorsi diversi. Una sessione può essere breve, come un trasferimento file, o lunga, come la durata di un login remoto.
3) Instradamento dinamico: il percorso da utilizzare per inviare un messaggio dal sito A al sito B viene scelta solo al momento dell'invio del messaggio. Poiché la decisione viene presa dinamicamente, a messaggi distinti possono essere assegnati percorsi diversi. Il sito A prende la decisione di inviare un messaggio al sito C; a sua volta C decide di inviare il messaggio a D e così via. Alla fine, un sito invia il messaggio a B. Generalmente un sito invia un messaggio al sito che risulta meno utilizzato sul collegamento.

15.5.3 STRATEGIE RIGUARDANTI I PACCHETTI

I messaggi sono generalmente di lunghezza variabile. Per semplificare il progetto del sistema, la comunicazione è di solito realizzata usando messaggi di lunghezza costante

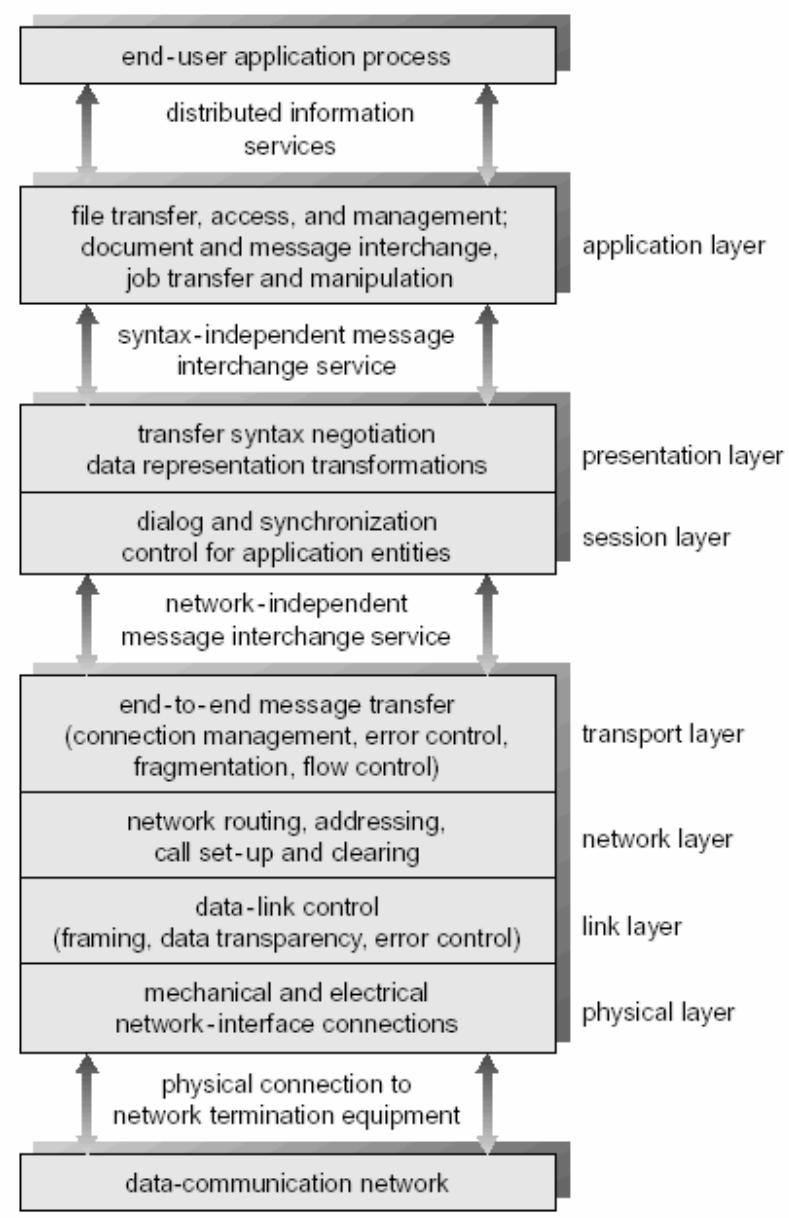
detti pacchetti, frame o datagram. L'informazione codificata in un singolo pacchetto può essere inviata al destinatario tramite una comunicazione priva di connessione. Essa può essere inaffidabile, nel senso che il mittente non ha alcuna garanzia del fatto che il pacchetto giunga a destinazione e non ha disposto alcun modo per verificare se un pacchetto spedito sia andato perduto. In alternativa, il pacchetto può essere affidabile, nel senso che un altro pacchetto è inviato dal destinatario come conferma dell'arrivo del primo pacchetto. Se un messaggio è troppo lungo per essere codificato in un solo pacchetto, o se i pacchetti devono essere ripetutamente scambiati tra i due interlocutori, è necessario stabilire una connessione al fine di permettere una comunicazione affidabile.

15.5.4 STRATEGIE DI CONNESSIONE

Una volta che i messaggi sono in grado di raggiungere le loro destinazioni, i processi possono instaurare sessioni di comunicazione per scambiarsi informazioni. Esistono diversi modi per collegare coppie di processi che intendono comunicare attraverso la rete. I tre schemi più diffusi sono:
1) Commutazione di circuito. Se due processi vogliono comunicare, tra essi viene fissato un collegamento fisico permanente. Questo collegamento rimane allocato per tutta la durata della comunicazione e nessun altro processo può utilizzarlo per tutto questo periodo di tempo, anche se esistono intervalli di tempo nei quali i due processi non comunicano attivamente. Questo schema è simile a quello utilizzato nel sistema telefonico. Una volta aperta una linea di comunicazione tra due parti, vale a dire che la parte A chiama la parte B, nessun altro può usare questo circuito finché la comunicazione non viene esplicitamente terminata (ad esempio quando una delle due parti riappende).
2) Commutazione di messaggio. Se due processi vogliono comunicare, viene fissato un collegamento temporaneo per la durata del trasferimento del messaggio. I collegamenti fisici tra i corrispondenti vengono allocati dinamicamente in base alle necessità. Tale allocazione ha breve durata. Ogni messaggio è costituito da un blocco di dati e da informazioni di sistema, come l'origine, la destinazione e i codici di correzione degli errori.
3) Commutazione di pacchetto. Un messaggio logico può essere diviso in un dato numero di pacchetti; ciascuno dei quali può essere inviato a destinazione separatamente, perciò deve contenere, oltre ai dati, un indirizzo di sorgente e uno di destinazione; ogni pacchetto può seguire un percorso diverso attraverso la rete. Quando i pacchetti arrivano a destinazione devono essere ricomposti in messaggi.

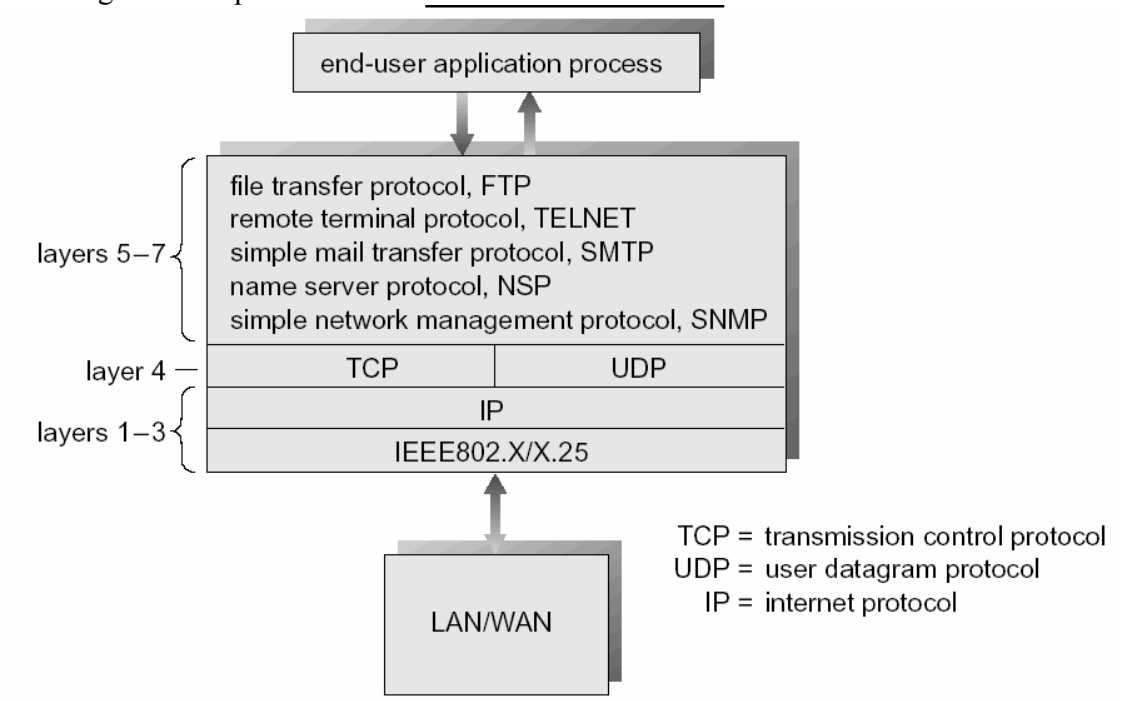
15.6 STRATEGIE DI PROGETTO

Il problema della progettazione e della relativa implementazione può essere semplificato mediante una suddivisione in più strati. Ciascun strato su un sistema comunica con lo strato corrispondente sugli altri sistemi. Ogni strato può avere i suoi protocolli o può essere il risultato di una suddivisione logica. I protocolli possono essere implementati in hardware o in software. Secondo la ISO gli strati sono definiti come segue:
1) Strato fisico. E' responsabile della gestione dei particolari meccanici ed elettrici della trasmissione fisica di un flusso di bit. Nello strato fisico il sistema deve accordarsi sulla rappresentazione elettrica delle cifre binarie 0 e 1, in modo che quando i dati vengono



inviati come flusso di segnali elettrici il ricevitore sia in grado interpretare correttamente i dati. Questo strato è implementato nell'hardware del dispositivo di rete.
2) Strato di collegamento dati. E' responsabile della gestione dei frame, o di parti di pacchetti di lunghezza fissa comprese l'individuazione e la correzione degli errori che si sono verificati nello strato fisico.
3) Strato di rete. E' responsabile della fornitura di collegamenti e dell'instradamento dei pacchetti nella rete di comunicazione, comprese la gestione dell'indirizzo dei pacchetti in uscita, la decodifica dell'indirizzo dei pacchetti in arrivo e la gestione delle informazioni di instradamento per rispondere adeguatamente ai cambiamenti dei livelli di carico. I router operano in questo strato.
4) Strato di trasporto. E' responsabile dell'accesso a basso livello alla rete e del trasferimento dei messaggi tra i client, compresi la suddivisione dei messaggi in pacchetti, la gestione dell'ordine dei pacchetti, il controllo del flusso e la generazione degli indirizzi fisici.
5) Strato di sessione. E' responsabile dell'implementazione di sessioni, o di protocolli di comunicazione da processo a processo. Generalmente questi protocolli sono le effettive comunicazioni per i login remoti, i trasferimenti di file e la posta elettronica.

6) Strato di presentazione. E' responsabile della risoluzione delle differenze di formato che possono presentarsi tra i diversi siti della rete, fra cui la conversione di caratteri e i modi half-duplex e full-duplex (invio del carattere di eco).
7) Strato di applicazione. E' responsabile dell'interazione diretta con gli utenti. Questo strato tratta il trasferimento di file, i protocolli di login remoto e la posta elettronica, nonché gli schemi per i database distribuiti.



Molti siti internet comunicano attraverso l'Internet Protocol (IP). I servizi sono realizzati sopra l'IP attraverso il protocollo senza connessione UDP (User Datagram Protocol) e attraverso il protocollo orientato alla connessione TCP (Transmission Control Protocol). La pila di protocolli TCP/IP ha meno strati di quelli stabiliti dal modello ISO , e poichè combina diverse funzioni in ciascun strato è, teoricamente, più difficile da realizzare ma più efficiente delle reti ISO. Il protocollo IP è responsabile della trasmissione del datagram IP, l'unità di base dell'informazione, attraverso un interconnessione di reti basata sul TCP/IP. TCP utilizza IP per trasportare in modo affidabile un flusso di informazioni tra due processi. L'altro usuale protocollo di trasmissione tra reti interconnesse è UDP/IP. L'UDP è un protocollo di trasporto non affidabile e senza connessione, usa IP per trasferire pacchetti, ma in più offre la correzione degli errori e un protocollo di indirizzo di porta per specificare il processo sul sistema remoto cui è destinato il pacchetto.

16 STRUTTURE DEI SISTEMI DISTRIBUITI

16.1 SO DI RETE

Un SO di rete offre un ambiente nel quale gli utenti, che sono a conoscenza della presenza di più macchine, possono accedere alle risorse remote effettuando un login sulle

appropriate macchine remote oppure attraverso il trasferimento di dati dalla macchina remota alla loro macchina.

16.1.1 LOGIN REMOTO

Una funzione importante dei SO di rete è di consentire agli utenti di iniziare una sessione di lavoro a distanza su un altro computer. A tale scopo INternet fornisce la funzione telnet. Per effettuare un login remoto l'utente utilizza il seguente comando: telnet cs.utexas.edu. Questo comando fa in modo che venga realizzata una connessione tra la macchina locale nella Brown University e il computer cs.utexas.edu. Dopo che la connessione è stabilita, il software di rete crea un collegamento bidirezionale trasparente tale che tutti i caratteri inseriti dall'utente vengono inviati a un processo su cs.utexas.edu, e tutti gli output di questo processo vengono rispediti all'utente. Il processo sulla macchina remota chiede all'utente il nome del login e una password; ricevute le corrette informazioni, il processo agisce per conto dell'utente, che può effettuare le sue elaborazioni sulla macchina remota come ogni altro utente locale.

16.1.2 TRASFERIMENTO DI FILE REMOTI

Il meccanismo FTP è implementato in modo simile a telnet. Un daemon sul sito remoto verifica le richieste di connessione sulla porta di sistema FTP; viene effettuata la convalida del login e all'utente viene consentito di richiedere a distanza l'esecuzione dei comandi. Diversamente dal daemon di telnet, che esegue per l'utente ogni comando, il daemon FTP risponde soltanto a un insieme predefinito di comandi relativi ai file, tra i quali get, put, ls o dir, cd...

16.2 SO DISTRIBUITI

In un sistema distribuito gli utenti accedono alle risorse remote nello stesso modo in cui accedono alle risorse locali. Il trasferimento di dati e processi da un sito all'altro avviene sotto il controllo del SO distribuito.

16.2.1 MIGRAZIONE DEI DATI

Si supponga che un utente del sito A voglia accedere a dati (ad esempio un file) che risiedono nel sito B. Il sistema dispone di due metodi di base per trasferire i dati. Un approccio consiste nel trasferire tutto il file sul sito A. Da questo punto in poi l'intero accesso al file è di tipo locale.
L'altro approccio consiste nel trasferire sul sito A solo le parti del file immediatamente necessarie. Se in seguito viene richiesta anche l'altra parte, viene effettuato un altro trasferimento. Quando l'utente non avrà più la necessità di accedere al file, qualsiasi parte di esso sia stata modificata viene ritrasmessa al sito B. Occorre sottolineare l'analogia con la paginazione su richiesta. Questo metodo è utilizzato dal protocollo Network File System (NFS) della Sun Microsystems. Anche il protocollo SMB della Microsoft (eseguito sopra il protocollo TCP/IP o sopra NETBUI, della stessa Microsoft) consente la condivisione dei file in rete.

Per accedere a una piccola parte di un file di grosse dimensioni, ovviamente conviene utilizzare il secondo approccio. Quando è necessario accedere a parti considerevoli del file, conviene copiare tutto il file.

16.2.2 MIGRAZIONE DELLE COMPUTAZIONI

In alcuni casi può essere più efficiente trasferire le computazioni anziché i dati. Si consideri, ad esempio, un job che ha bisogno di accedere a diversi file di grosse dimensioni, che risiedono in diversi siti, per ottenere un sommario di tutti questi file. E' più conveniente accedere ai file sui siti nei quali risiedono e riportare i risultati al sito che ha iniziato il calcolo. In generale viene utilizzato il comadno remoto se il tempo di trasferimento dei dati è maggiore del tempo di esecuzione del comando remoto.
Tale calcolo può essere eseguito in diversi modi. Si supponga che il processo P voglia accedere a un file del sito A. L'accesso al file viene effettuato sul sito A e può essere avviato da una chiamata di procedura remota (remote procedure call - RPC). Una RPC utilizza un protocollo con datagram (UDP su internet) per eseguire una routine su un sistema remoto. Il processo P richiama una procedura predefinita sul sito A, la procedura esegue adeguatamente il proprio compito e restituisce i risultati a P.
In alternativa il processo P può inviare un messaggio al sito A. Il SO di A crea un nuovo processo Q la cui funzione è quella di eseguire il compito designato; quando il processo Q termina, invia il risultato richiesto a P per mezzo del sistema di messaggi. Occorre notare che in questo schema il processo P può essere eseguito concorrentemente al processo Q e, in effetti, più processi possono essere eseguiti concorrentemente su diversi siti.
Entrambi i metodi possono essere utilizzati per accedere a più file residenti su vari siti. Una RPC può richiamare un'altra RPC, o addirittura trasferire messaggi a un altro sito. Analogamente il processo Q, durante la propria esecuzione, può inviare un messaggio ad un altro sito che a sua volta crea un processo. Questo processo può inviare un messaggio a Q, oppure ripetere il ciclo.

16.2.3 MIGRAZIONE DEI PROCESSI

una logica estensione della migrazione delle computazioni è la migrazione dei processi. Quando un processo viene eseguito, non sempre ha luogo sul sito sul quale viene avviata. Può risultare vantaggioso eseguire tutto il processo, o parti di esso, su siti diversi. Questo schema è preferibile per diversi motivi.
- Bilanciamento del carico
- Accelerazione del calcolo (anziché un grande processo, molti sottoprocessi).
- Preferenza di hardware (un processo può venir eseguito su dell'hardware specializzato).
- Preferenza di software
- Accesso ai dati

17 FILE SYSTEM DISTRIBUITI

Un file system distribuito (DFS) è un'implementazione distribuita del modello classico di un file system in un sistema time sharing, dove più utenti condividono file e risorse di

memoria. Un DFS consente di estendere lo stesso tipo di condivisione al caso in cui siano fisicamente dispersi tra siti diversi in un sistema distribuito.

17.1 INTRODUZIONE

Il servizio è un'entità software in esecuzione su una o più macchine e fornisce un tipo particolare di funzione a client sconosciuti a priori. Il server è il software di servizio in esecuzione su una singola macchina. Il client è un processo che può richiedere un servizio utilizzando una serie di operazioni che costituiscono la sua interfaccia, detta interfaccia del client. Talvolta viene definita un'interfaccia di livello inferiore per l'effettiva interazione tra le macchine, alla quale viene fatto riferimento con il termine interfaccia macchina.
Data questa terminologia, è possibile affermare che un file system fornisce servizi ai suoi client. Un'interfaccia del client per un servizio relativo a file è formata da un insieme di operazioni su file primitive.
Un DFS è un file system in cui i client, server e dispositivi di memoria sono sparsi tra le varie macchine di un sistema distribuito. Di conseguenza l'attività di servizio deve essere eseguita attraverso la rete e, anziché da un unico magazzino di dati centralizzato, esistono più dispositivi di memoria indipendenti. La configurazione e l'implementazione concrete di un DFS possono essere di vario tipo. Esistono configurazioni dove i server vengono eseguiti su macchine dedicate, nonché configurazioni in cui una macchina può essere sia server che client.
In teoria un DFS deve apparire ai client come un file system centralizzato convenzionale. La molteplicità e la dispersione dei suoi server e dispositivi di memoria devono essere rese trasparenti. Ciò significa che l'interfaccia del client di un DFS non deve distinguere tra file locali e file remoti. Spetta al DFS localizzare i file e predisporre per il trasporto dei dati. Un DFS trasparente facilita la mobilità dell'utente portando il suo ambiente, cioè la sua home directory, ovunque egli apra una sessione.
La misura delle prestazioni più importante per un DFS è rappresentata dalla quantità di tempo necessaria per soddisfare varie richieste di servizio. Nei sistemi convenzionali questo tempo è dato dal tempo di accesso la disco e da una piccola quantità di tempo di cpu. In un DFS invece un accesso remoto risente anche di un ulteriore overhead attribuito alla struttura distribuita. In questo overhead rientrano il tempo necessario per inviare la richiesta ad un server e il tempo necessario per ricevere la risposta. Pertanto per ogni direzione occorre considerare, oltre al trasferimento effettivo dell'informazione, anche l'overhead di cpu dovuto all'esecuzione del protocollo di comunicazione. Le prestazioni di un DFS possono essere viste come un'altra misura della trasparenza del dfs. Ciò significa che le prestazioni di un DFS ideale devono essere paragonabili a quelle di un file system tradizionale.

17.2 NOMINAZIONE E TRASPARENZA

La nominazione (naming) è una funzione di mapping tra oggetti logici e oggetti fisici. Ad esempio, gli utenti trattano oggetti di dati logici, rappresentati dai nomi dei file, mentre il sistema manipola blocchi fisici di dati, memorizzati sulle tracce di un disco.
Generalmente l'utente fa riferimento a un file attraverso un nome testuale. Quest'ultimo

viene mappato su un identificatore numerico di livello inferiore che a sua volta viene mappato sui blocchi dei dischi. Questo mapping multilivello fornisce agli utenti un'astrazione del file che nasconde i particolari relativi all'effettiva memorizzazione del file.
In un DFS trasparente una nuova dimensione viene aggiunta all'astrazione: nascondere la posizione attraverso la rete. In un file system convenzionale un valore della funzione di nominazione è un indirizzo su un disco. In un DFS l'insieme dei valori viene esteso in modo da comprendere anche la macchina specifica sul cui disco è memorizzato il file.
Astraendo ulteriormente il concetto di file, è possibile arrivare alla replicazione del file. Dato il nome di un file, il mapping restituisce una serie di locazioni delle repliche di questo file. In questa astrazione sono nascoste all'utente sia l'esistenza di copie multiple che la loro locazione.

17.2.1 STRUTTURE DI NOMINAZIONE

Esistono due nozioni correlate per il mapping dei nomi in un DFS e occorre differenziarle:

- Trasparenza di locazione. Il nome di un file non rileva alcun indizio sulla locazione fisica del file
- Indipendenza di locazione. Il nome di un file non deve essere modificato se cambia la locazione di memoria fisica del file stesso.

Entrambe le definizioni si riferiscono al livello di nominazione discusso in precedenza, poiché i file hanno nomi diversi a livelli diversi: a livello utente vengono utilizzati nomi, mentre a livello di sistema vengono usati identificatori numerici. Il mapping dinamico è uno schema di nominazione indipendente dalla locazione, in quanto può mappare lo stesso nome di file su locazioni diverse in momenti diversi. Perciò l'indipendenza di locazione è una caratteristica più forte di quanto non lo sia la trasparenza di locazione.
In pratica la maggior parte degli attuali DFS offre un mapping statico con trasparenza di locazione per i nomi a livello utente. Tuttavia questi schemi non supportano la migrazione dei file. Ciò significa che è impossibile cambiare autonomamente la locazione di un file, e quindi, per questi schemi, la nozione di indipendenza di locazione è irrilevante. I file vengono associati definitivamente a uno specifico gruppo di blocchi sui dischi.
L'indipendenza della locazione la trasparenza di locazione statica si differenziano ancora per altri aspetti:

- La separazione dei dati dalla locazione, come nel caso dell'indipendenza della locazione, offre una migliore astrazione per i file. Il nome di un file deve indicare gli attributi più significativi del file stesso, come il suo contenuto, piuttosto che la sua locazione. I file indipendenti dalla locazione possono essere considerati come contenitori di dati logici che non vengono assegnati ad una specifica locazione di memoria. Se è supportata soltanto la trasparenza di locazione statica, il nome del file continua ad indicare un gruppo specifico, anche se nascosto, di blocchi fisici di dischi.
- La trasparenza di locazione statica fornisce agli utenti un modo per condividere convenientemente i dati. Gli utenti possono condividere file remoti utilizzando la trasparenza di locazione per effettuare la nominazione dei file, esattamente come se questi fossero dei file locali. Ciononostante la condivisione dello spazio di

memoria secondaria risulta scomoda in quanto i nomi logici sono staticamente vincolati ai dispositivi fisici di memoria. L'indipendenza dalla locazione consente la condivisione dello spazio di memoria stesso e degli oggetti di dati. Quando i file possono essere resi mobili, lo spazio di memoria dell'intero sistema assume l'aspetto di una singola risorsa virtuale. Un vantaggio dato da questa visione consiste nella possibilità di bilanciare l'utilizzo dei dischi all'intero sistema.

- L'indipendenza dalla locazione separa la gerarchia di nominazione dalla gerarchia dei dispositivi di memoria e dalla struttura intercomputer. Per contrasto, utilizzando la trasparenza di locazione statica, è possibile evidenziare la corrispondenza tra unità componenti e macchine anche se i nomi sono trasparenti. Le macchine sono configurate utilizzando un modello simile alla struttura di nominazione, questo fatto può portare l'architettura del sistema a rispettare vincoli non necessari, ed è anche in conflitto con altre considerazioni. Un server che gestisca una directory root è un esempio di struttura dettata dalla gerarchia di nominazione e contraddice le direttive del decentramento.

17.2.2 SCHEMI DI NOMINAZIONE

Sono disponibili tre approcci principali per gli schemi di nominazione in un DFS:

1) Il più semplice prevede di nominare i file per mezzo di una combinazione dei loro nomi host e locale; ciò garantisce un unico nome su tutto il sistema. Questo schema di nominazione non gode di trasparenza di locazione e tanto meno di indipendenza di locazione. Ciononostante possono essere utilizzate le stesse operazioni su file che vengono usate per file locali e remoti. La struttura del DFS è rappresentata da un insieme di unità componenti isolate costituite da interi file system convenzionali. In questo primo approccio le unità componenti rimangono isolate, anche se sono disponibili mezzi per fare riferimento a un file remoto.
2) Il secondo approccio si è diffuso con il Network File System della Sun. NFS offre i mezzi per unire le directory remote alle proprie directory locali, dando all'utente l'impressione di un albero di directory coerente. Nelle prime versioni era possibile accedere trasparentemente solo alle directory remote montate in precedenza. Con l'avvento della funzione di automontaggio i montaggi avvengono su richiesta in base a una tabella di punti di montaggio e nomi di strutture di file. E' inoltre necessaria una forma di integrazione di componenti per supportare una condivisione, ma l'integrazione è limitata e non uniforme poiché ogni macchina può aggiungere diverse directory remote al proprio albero. La struttura risultante è incostante. Normalmente viene ottenuta una foresta di alberi Unix con sottoalberi condivisi.
3) L'integrazione totale dei componenti del file system è ottenuta per mezzo del terzo approccio. Una struttura globale di nomi si estende a tutti i file del sistema. In teoria, la struttura composta del file system dev’essere isomorfa alla struttura di un file system convenzionale. In pratica, comunque, esistono molti file speciali, ad esempio i dispositivi in Unix e le directory di file eseguibili, che rendono difficile il conseguimento di questo scopo.

Un criterio importante per la valutazione delle strutture di nominazione è dato dalla loro complessità amministrativa. La struttura più complessa e più difficile da mantenere è la

struttura NFS. Poiché ogni directory remota può essere montata in qualsiasi punto dell'albero di directory locale, la gerarchia risultante può essere quindi quasi priva di struttura.
Quando un server diventa non disponibile, diventa non disponibile anche un insieme arbitrario di directory su macchine diverse. Inoltre, per controllare quale directory di quale macchina sia possibile aggiungere all'albero, viene utilizzato un meccanismo separato di accreditamento. Questo meccanismo può condurre alla situazione nella quale su un client un utente può accedere a un albero di directory remoto, mentre su un altro client l'accesso viene negato allo stesso utente.

17.2.3 Tecniche di implementazione

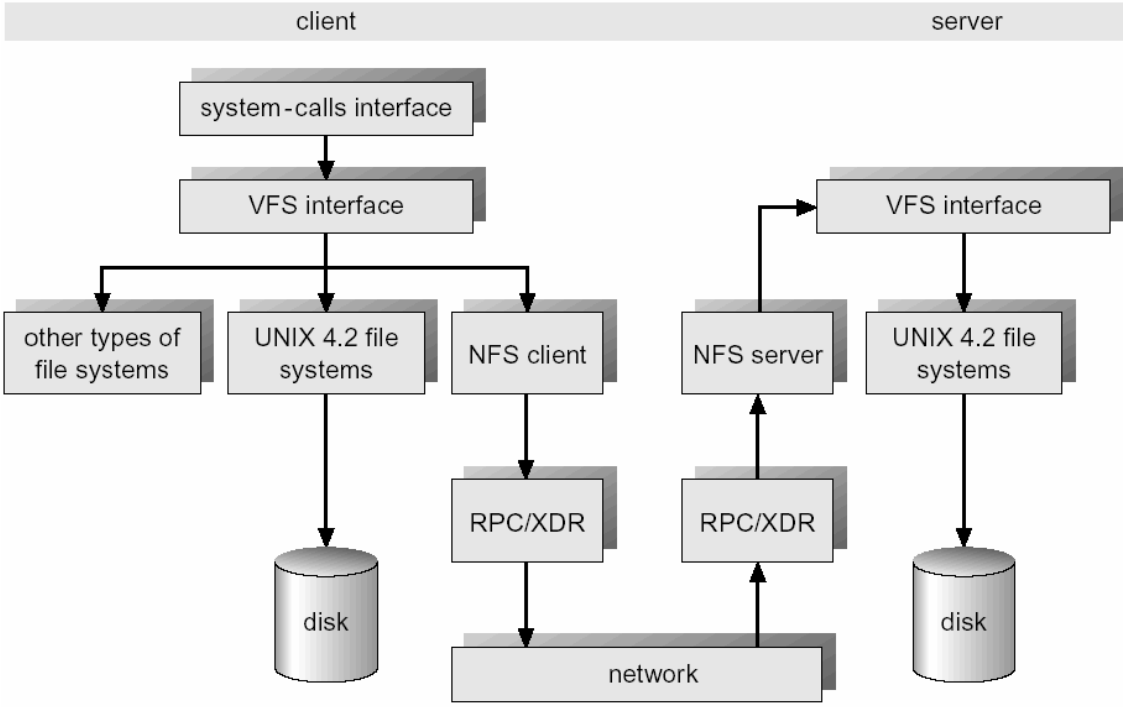
L'implementazione della nominazione trasparente necessita di un mapping per associare il nome di un file a una locazione di memoria. Per mantenere gestibile tale mapping è necessario aggregare insieme di file in unità componenti e fornire un mapping sulla base dell'unità componente anziché sulla base di un singolo file. Detta aggregazione è utile anche per scopi amministrativi. Sistemi del tipo Unix utilizzano l'albero gerarchico delle directory per fornire il mapping nome-locazione e per aggregare i file nelle directory in modo ricorsivo.
Per aumentare la disponibilità delle informazioni fondamentali sul mapping possono essere usati metodi come la replicazione, il caching locale, o entrambi.
L'indipendenza della locazione implica che il mapping cambi con il tempo; quindi, replicando il mapping, diventa impossibile ottenere un aggiornamento semplice e coerente di queste informazioni. per superare questo ostacolo è possibile utilizzare una tecnica che prevede di introdurre identificatori di file indipendenti dalla locazione di basso livello. IK nomi dei file vengono mappati sugli identificatori di file di basso livello, che indicano a quale unità componente appartiene il file; tali identificatori sono comunque indipendenti dalla locazione. Essi possono essere liberamente replicati e sottoposti a caching, senza che siano invalidati dalla migrazione delle unità componenti. L'inevitabile prezzo di tale tecnica è un secondo livello di mapping per mappare le unità componenti sulle locazioni; esso richiede un meccanismo di aggiornamento semplice e coerente.
L'implementazione degli alberi di directory di tipo Unix, che utilizzano questi identificatori di basso livello indipendenti dalla locazione, rende invariante l'intera gerarchia nel caso di migrazione di unità componenti. l'unica variazione riguarda il mapping delle locazioni delle unità componenti.
Un metodo diffuso per implementare questi identificatori di basso livello consiste nell'utilizzo di nomi strutturati. Questi nomi sono costituiti da stringhe di bit formate normalmente da due parti. La prima parte identica l'unità componente alla quale appartiene il file, la seconda parte identifica il file all'interno dell'unità. Sono possibili varianti con più parti. L'invariante dei nomi strutturati, comunque, è che le parti individuabili dal nome sono uniche in ogni momento soltanto all'interno del contesto delle parti restanti. L'unicità in ogni momento può essere ottenuto avendo cura di non riutilizzare un nome già utilizzato, oppure aggiungendo ulteriori bit oppure utilizzando un timestamp come se fosse parte del nome. Un altro metodo per effettuare questo processo consiste nel prendere un sistema con trasparenza di locazione e aggiungervi un altro

livello di astrazione per produrre uno schema di nominazione con indipendenza dalla locazione.

17.6 ESEMPI DI SISTEMI

17.6.2 IL NETWORK FILE SYSTEM DI SUN

Il Network File System (NFS) è sia un'implementazione che una specifica di un sistema software per l'accesso a file remoti attraverso LAN o WAN. Esso utilizza un protocollo datagram affidabile (UDP/IP) ed ethernet (o altro sistema di rete).



17.6.2.1 GENERALITÀ

NFS considera un insieme di stazioni di lavoro interconnesse come un insieme di macchine indipendenti con file system indipendenti. Lo scopo è quello di permettere un certo grado di condivisione tra questi file system, su richiesta esplicita, in modo trasparente. La condivisione è basata su una relazione server-client. Una macchina può essere sia un client che un server. La condivisione è ammessa tra ogni coppia di macchine, anziché essere limitata alle macchine server dedicate. Per assicurare l'indipendenza delle macchine, la condivisione di un file system remoto ha effetto esclusivamente sulla macchina client.
Affinché una directory remota sia accessibile in modo trasparente a una macchina particolare, ad esempio M1, un client di quella macchina deve prima eseguire un'operazione di montaggio. La semantica dell'operazione consiste nel fatto che una directory remota viene montata sopra una directory di un file system locale. Una volta completata l'operazione di montaggio, la directory assume l'aspetto di un sottoalbero

integrante il file system locale, sostituendo il sottoalbero che discende dalla directory locale; questa, a sua volta, rappresenta la radice della directory appena montata. La directory remota viene specificata come argomento dell'operazione di montaggio in modo non trasparente; deve essere fornita la locazione, cioè il nome dell'host, della directory remota. Tuttavia, da questo momento in poi gli utenti della macchina M1 possono accedere ai file della directory remota in modo completamente trasparente.
Potenzialmente ogni file system o directory interna al file system, nel rispetto dei diritti di accesso, possono essere montati in remoto in cima a qualsiasi directory locale. Le stazioni di lavoro senza disco possono anche montare i propri file prelevandoli dal server.
Sono permessi anche montaggi a cascata; ciò significa che un file system può essere montato sopra un altro file system che non sia locale, ma remoto. Tuttavia una macchina viene sottoposta alle mount da essa richieste.
Montando un file system remoto, il client non acquisisce l'accesso ai file system che sono montati sopra il primo file system; così il meccanismo di montaggio non ha la proprietà transitiva.
Se un file condiviso viene montato sulla home directory utente di tutte le macchine di una rete, un utente può aprire una sessione su qualsiasi stazione di lavoro e prelevare il proprio ambiente home. Questa proprietà viene detta mobilità dell'utente.
Uno degli scopi del progetto di NFS era quello di operare in un ambiente eterogeneo di macchine, sistemi operativi e architetture di rete. La specifica di NFS è indipendente da questi mezzi e incoraggia quindi altre implementazioni. Questa indipendenza viene realizzata utilizzando primitive RPC costruite su un protocollo External Date Rapresentation (XDR) utilizzando tra due macchine interfacce indipendenti dall'implementazione. Quindi, se il sistema è formato da macchine eterogenei adeguatamente interfacciati a NFS, possono essere montati file system di diversi tipi, sia localmente che in remoto.
La specifica di NFS distingue tra i servizi offerti da un meccanismo di montaggio e gli effettivi servizi di accesso ai file remoti. Di conseguenza, per questi servizi sono specificati due protocolli distinti: un protocollo di montaggio e un protocollo per gli accessi a file remoti, chiamato protocollo NFS. I protocolli sono specificati come insiemi di RPC. Queste RPC sono gli elementi di base utilizzati per implementare, in modo trasparente, l'accesso remoto ai file.

17.6.2.2 Protocollo di montaggio

Un'operazione di montaggio comprende il nome della directory remota da montare e il nome della macchina server in cui tale directory è memorizzata. La richiesta di montaggio viene mappata sulla RPC corrispondente e viene inviata al server di montaggio che è in esecuzione sulla specifica macchina server. Il server conserva una lista di esportazione (/etc/exports) che specifica i file system locali esportati per il montaggio, nonché i nomi di macchine alle quali è permessa tale operazione. La lunghezza di tale lista è limitata e questo ne riduce la scalabilità. Quando il server riceve una richiesta di montaggio conforme alla propria lista di esportazione, restituisce al client un file handle da utilizzare come chiave per ulteriori accessi ai file che si trovano all'interno del file system montato. La file handle contiene tutte le informazioni di cui ha bisogno il server per gestire un proprio file. In termini Unix, il file handle è costituito da

un identificatore di file system e da un numero di inode per identificare la directory montata all'interno del file system esportato.
Il server contiene anche un elenco delle macchine client e delle corrispondenti directory attualmente montate. Questo elenco viene utilizzato soprattutto per scopi amministrativi, ad esempio per informare i client che un server sta andando fuori servizio. L'aggiunta o la cancellazione di un elemento da questa lista è l'unico modo in cui il protocollo dell'operazione mount può modificare lo stato del server.

17.6.2.3 PROTOCOLLO NFS

Il protocollo NFS offre un insieme di RPC per operazioni su file remoti. Le procedure supportano le seguenti operazioni:

- Ricerca di un file in una directory
- Lettura di un insieme di elementi in una directory
- Manipolazione di link e directory
- Accesso ad attributi di file
- Lettura e scrittura di file

Queste procedure possono essere richiamate solo dopo aver stabilito un file handle per la directory montata in modo remoto.
L'omissione delle operazioni open e close è intenzionale. una caratteristica dei server NFS è l'assenza di informazioni di stato (stateless). I server non conservano informazioni sui loro client da un accesso all'altro. Non esistono parallelismi con la tabella dei file aperti o le strutture di file di unix da parte del server. Di conseguenza, ogni richiesta deve fornire un insieme completo di argomenti, tra cui un unico identificatore di file e un offset assoluto all'interno del file per svolgere le operazioni appropriate. La struttura che ne deriva è robusta; non devono essere prese misure speciali per ripristinare un server dopo un guasto. Per tale ragione, le operazioni sui file devono essere idempotenti.
La presenza della suddetta lista di client sembra violare la proprietà di assenza di informazioni di stato sul server, ma non essendo essenziale ai fini del funzionamento del server stesso, non è necessario recuperarla dopo un crash.
Un'ulteriore implicazione della filosofia dei server senza informazioni di stato e un risultato della sincronia di una RPC consiste nel fatto che i dati modificati, tra cui i blocchi di indirezione e di stato, devono essere riscritti sul disco del server prima che i risultati vengano restituiti al client. Un client può sottoporre a caching i blocchi di scrittura, ma quando li invia al server assume che abbiano raggiunto i dischi del server, così il crash del server e il successivo ripristino saranno invisibili a tutti i client; tutti i blocchi che il server gestisce per il client resteranno intatti. La conseguente perdita di prestazioni può essere rilevante poiché vengono perduti i vantaggi del caching.
Di una singola chiamata di procedura di scrittura di NFS sono garantite l'atomicità e la non interferenza con altre chiamate di scrittura sullo stesso file. Tuttavia, il protocollo NFS non fornisce meccanismi di controllo della concorrenza, e poiché ogni chiamata di scrittura o di lettura di NFS può contenere non più di 8K di dati e i pacchetti UDP sono limitati a 1500 byte, può essere necessario spezzare una system call write in diverse RPC di scrittura; quindi, due utenti che scrivono sullo stesso file remoto possono riscontrare delle interferenze nei loro dati. Poiché la gestione di meccanismi di lock richiede

l'informazione di stato, viene richiesto che un servizio esterno al NFS debba fornire tali meccanismi.

17.6.2.4 ARCHITETTURA DI NFS

L'architettura NFS è costituita da tre strati principali. Il primo strato è l'interfaccia del file system unix, basata sulle chiamate open, read, write, close e sui descrittori di file.
Il secondo strato è chiamato strato del file system virtuale (VFS) e svolge due funzioni importanti:

1) Separa le operazioni generiche del file system dalla loro implementazione definendo un interfaccia VFS pulita. Sulla stessa macchina possono coesistere più implementazioni per interfaccia VFS, che permettono un accesso trasparente a diversi tipi di file system montati localmente.
2) Il VFS è basato su una struttura di rappresentazione di file chiamata vnode che contiene un indicatore numerico per un file unico su tutta la rete. Il kernel contiene una struttura vnode per ogni nodo attivo, sia questo un file o una directory.

Quindi il VFS distingue i file locali e i file remoti, e i file locali vengono a loro volta distinti in base ai relativi tipi di file system.
Nel caso Unix, il kernel contiene una tabella (/etc/mtab) che registra i particolari sulle operazioni di mount a cui hanno preso parte in form di client. Inoltre, i vnode per ogni directory montata vengono sempre tenuti in memoria; questi vnode vengono contrassegnati in modo che le richieste relative a tali directory possono essere dirette tramite la tabella di montaggio ai corrispondenti file system che sono stati montati. Fondamentalmente, la struttura di vnode, assieme alla tabella di montaggio, forniscono per ogni file un puntatore al suo file system padre, nonché al file system sopra il quale è montato.
Il VFS attiva le operazioni specifiche del file system per gestire le richieste locali in base ai tipi di file system, e richiama le procedure NFS per le richieste remote. Le file hadle vengono costruite a partire dai vnode relativi e vengono inviate a queste procedure come argomenti. Lo strato che implementa il protocollo NFS è il più basso dell'architettura ed è chiamato strato di servizio di NFS.
Per illustrare quest'architettura è possibile descrivere la gestione di un'operazione su un file remoto già aperto. Il client inizia l'operazione per mezzo di una normale system call. Lo strato del SO mappa questa call su un'operazione VFS sul vnode appropriato. Lo strato VFS identifica il file come remoto e richiama la procedura NFS appropriata. Una chiamata RPC viene inviata allo strato di servizio NFS sul server remoto. Quetsa chiamata viene inviata di nuovo allo strato VFS sul sistema remoto, il quale la riconosce come locale e richiama l'opportuna operazione file system. Questo stesso tragitto viene ripercorso per riportare indietro il risultato. Il vantaggio di questa architettura consiste nel fatto che il client e il server sono identici; quindi una macchina può essere un client, un server o entrambi.

17.6.2.5 TRADUZIONE DEL PATH NAME.

La traduzione del path name viene effettuata suddividendo il percorso stesso in nomi componenti ed eseguendo una call di ricerca NFS separata per ogni coppia composta da un nome componente e un vnode di directory. Quando si incontra un punto di montaggio, la ricerca di un componente causa una rpc separata al server. Questo schema di attraversamento del path name è costoso, ma necessario, in quanto ogni client ha un unico configurazione del proprio spazio di nomi logico, dettata dai montaggi che ha eseguito. Per accelerare la ricerca, una cache di ricerca dei nomi delle directory, nel sito del client, conserva i vnode per i nomi delle directory remote. Questa cache accelera i riferimenti a file con lo stesso path name iniziale.

17.6.2.6 OPERAZIONI REMOTE

Facendo un'eccezione per l'apertura e la chiusura dei file, tra le normali system call Unix per operazioni sui file e le RPC del protocollo NFS esiste una corrispondenza quasi uno a uno. Quindi, un'operazione remota su file può essere tradotta direttamente sulla ROC corrispondente. Dal punto di vista concettuale NFS aderisce al paradigma del servizio remoto, ma in pratica vengono utilizzate tecniche di buffering e di caching per migliorare le prestazioni.

21 UNIX

UNIX Chronology

"The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the UNIX system has led a generation of software designers to new ways of thinking about programming. The genius of the UNIX system is its framework, which enables programmers to stand on the work of others."

Citation from the 1983 ACM Turing Award presented to Kenneth Thompson and Dennis Ritchie

"The Swiss Army Knife of Software."
"Some consider UNIX to be the second most important invention to come out of AT&T Bell Labs after the transistor."
"UNIX is simple and coherent, but it takes a genius (or at any rate a programmer) to understand and appreciate the simplicity."

Dennis Ritchie

1949

The Antitrust Department of the Department of Justice of the US Government sues Western Electric and AT&T for restraint of trade. Western Electric was a wholly owned subsidiary of AT&T and that Bell Telephone Laboratories (BTL or Bell Labs) was jointly

owned by Western Electric and AT&T (50% each). The consent decree required that AT&T license use of all its patents at nominal fees, which laid the groundwork for subsequent licensing of UNIX to universities.

1956

AT&T enters into "consent decree" with US government and agrees to restrict its business to furnishing "common carrier communications services", which keeps it out of the computer business.

1957

Bill Norris starts Control Data Corporation (CDC).

1965

AT&T, GE, IBM and Project MAC at MIT join together to develop the time-sharing system MULTICS (Multiplexed Information and Computing Service).

1966

Ken Thompson finishes studies at University of California at Berkeley (UCB) and joins technical staff at AT&T Bell Telephone Laboratories to work on MULTICS.

1968

Dennis Ritchie completes work on his doctorate at Harvard and joins Bell Labs to work on MULTICS project.

1969

AT&T Bell Labs drops out of MULTICS project. A system which was supposed to support 1000 on line users can barely handle three. Out of the ashes grows the most influential operating system in history.

Thompson gets an idea for a new type of file system and hashes out his ideas with Ritchie and Rudd Canaday.

Thompson writes first version of UNICS for PDP-7 in one month while wife is on vacation. He allocates one week each to the operating system functions: the kernel, the shell, the editor, and the assembler. He does this on a machine with 4K of 18 bit words. UNICS is pun on MULTICS and stands for Uniplexed Information and Computing Services. Name is changed to UNIX which is not an acronym. This version is in assembly language.

Thompson develops the interpretive language B based upon BCPL. Ritchie improved on "B" and called it "C"

1970

DEC begins shipping PDP-11 and revolutionizes the computer industry by selling 250,000 systems.

Bell Labs gets a PDP-11 to do text processing for the legal department. System is developed and implemented in UNIX. The standard DEC OS is never installed.

1971

The First Edition of UNIX manual is written.

1972

UNIX OS is rewritten in C which opened the door for porting.

1973

First UNIX development support group is formed in Bell Labs.

Pipes are invented with the Third Edition of UNIX and the UNIX philosophy begins to emerge:

- Write programs that do one thing and do it well.
- Write programs that work together
- Write programs that handle text streams, because that is the universal interface.

Thompson delivers first UNIX paper at the ACM Symposium on Operating Systems at the Thomas J. Watson Research Center in Yorktown Heights, NY. Within six months, the number of UNIX sites triples from 16 to 48.

1974

The UNIX Time-Sharing System is published in CACM by Ken Thompson and Dennis Ritchie. It is a revision of the 1973 paper.

University of California at Berkeley (UCB) gets Version 4 of UNIX.

Keith Standiford converts UNIX to PDP 11/45.

Berkeley begins making major enhancements to UNIX and sets the stage for becoming a major distribution center for their version of UNIX.

The Elements of Programming Style by Kernighan and Plauger is published.

1975

Thompson begins one year sabbatical at Berkeley.

AT&T officially begins licensing UNIX to universities.

1976

Boggs and Metcalfe invent Ethernet at Xerox in Palo Alto.

1978

Bill Joy produces first Berkeley Software Distribution (BSD) of UNIX.

Ritchie and Steve Johnson complete first port of UNIX to an Interdata 8/32, the first non-DEC computer to run UNIX. Note that this is nearly ten years after running only on DEC equipment.

UNIX is ported to a DEC VAX, but not by Thompson and Ritchie, since they had become disenchanted by DEC and its unwillingness to support UNIX. DEC's refusal to support UNIX must be one of the all time great blunders of the computer industry.

The C Programming Language by Kernighan and Ritchie is published.

Doug and Larry Michels start Santa Cruz Operations, Inc. (SCO) to sell UNIX on a PC. By 1992, they grow to $175 million in revenues.

1979

Seventh Edition UNIX PROGRAMMERS MANUAL (UNIX Version 7) is published. It is the first edition without Thompson's or RitchiÈs names. It is titled "UNIX (with a TM sign) Time-Sharing System." Bell Labs starts to protect its assets.

Microsoft licenses UNIX from AT&T and announces XENIX, which is soon overshadowed by MS-DOS.

1980

BSD UNIX finds its way back into Bell Labs as a new improved version.

Berkeley lands large DARPA contract and forms Computer Systems Research Group (CSRG).

SCO becomes a distributor for Microsoft XENIX and licenses the name XENIX because they sold their trade name DYNIX to Sequent.

1981

The IBM PC is released running Microsoft DOS; XENIX is pushed into the background.

1982

AT&T announces official support for UNIX and makes its first commercial release: UNIX System III.

Bill Joy, the inspiration behind BSD, leaves CSRG at Berkeley to co-found Sun Microsystems.

Sun gets its name from the Stanford University Network (SUN) board. The workstation is based on the Motorola 68000 chip running SunOS based on 4.2BSD. It includes an

optional local area network based on Ethernet. The commercial UNIX industry is in full gear.

HP announces support for UNIX on its 9000 workstations.

DEC releases ULTRIX.

IBM releases CPIX.

1983

Thompson and Ritchie receive ACM Turing award for their work on UNIX.

Richard Stallman, creator of EMACS and the Lisp Machine, starts GNU at MIT. First GPL.

1984

AT&T agrees to divest itself of the Bell Operating Companies and obtains the right to enter the computer business.

Fortune runs an article saying that 750 universities around the world, about 80% of those offering computer science degrees, have UNIX licenses.

X/Open is formed, by five European computer manufacturers: Bull, ICL, Siemens, Olivetti, and Nixdorf. The press refers to them as BISON.

Andrew Tanenbaum writes the first version of Minix, a Unix intended for educational purposes.

Apple releases the Macintosh.

1985

AT&T publishes the System V Interface Definition (SVID) in an attempt to standardize the UNIX interfaces, which was strongly influenced by the 1984 /usr/group standard.

The ISO introduced the POSIX standard.

1986

IBM releases AIX.

1988

AT&T BUYS 20% OF SUN MICROSYSTEMS, and the battle lines are formed.

IBM, DEC, HP, and others form Open Software Foundation (OSF) to compete with the AT&T/Sun alliance. They decide to use the AIX Kernel.

UNIX International (UI) is formed in response to OSF as an international consortium of System V UNIX users to work closely with AT&T to promote open systems and influence future development.

NeXT computer selects Mach Kernel for its NeXTStep OS.

David Cutler leaves DEC and joins Microsoft (October 31) to develop Windows NT.

Microsoft begins evaluating the Mach Kernel.

HP releases HP/UX.

1989

The C programming language is standardized by ANSI as X3.159.1989 which became an international standard ISO/IEC 9899:1990.

HP becomes the second largest supplier of UNIX based workstations by acquiring Apollo.

SCO is short of cash, and Microsoft, along with other investors, puts $25 million into SCO in exchange for 16% of the stock.

1990

OSF designates the Mach 3.0 Kernel from Carnegie-Mellon University as their choice for their version of UNIX, OSF/1.

Motif is released by OSF.

UNIX International releases SVR4 which is a unification of System V, BSD, and XENIX.

X/Open releases XPG3.

1991

AT&T incorporates UNIX System Laboratories (USL) with Novell, Amdahl, Fujitsu, Sun, Motorola, ICL, Olivetti, NEC, OKI Electric, III of Taiwan, and Toshiba.

Sun creates the SunSoft subsidiary and announces Solaris.

A finnish student at the University of Helsinki, Linus Torvalds, learns of Minix, and writes a kernel based on it. Linux 0.01 goes on the net under GPL. Instantly, he started receiving patches and enhancements.

GNU begins work on HURD, based on Mach 3.

1992

BSDI (Berkeley Software Design, Inc.) is formed and releases BSD/386, a PC version of UNIX, including source, for the low price of $995.

BSDI is sued by USL.

AT&T sells its ownership interest in Sun.

DEC introduces Alpha AXP, its 64 bit RISC processor.

Peter McDonald compiles the first Linux distribution, the SLS (Soft Landing Software).

1993

UNIX International (UI) terminates business.

4.4BSD is released.

In April, Patrick Volkerding compiles the Slackware distribution, based on SLS.

Novell buys USL from AT&T in June.

Novell gives the UNIX brand and trademark to X/Open in October.

The Common Open Software Environment (COSE) is created at UniForum.

Microsoft releases Windows NT.

The Common Desktop Environment (CDE) is demonstrated.

Linux is ported to non-Intel platforms (MIPS, Alpha,...).

FreeBSD and NetBSD are released, under the BSD licence.

1994

A settlement is reached between USL and BSDI. 4.4BSD-Lite is released at Berkeley. It differs from the previous version for just three files among 18,000.

SunSoft, AT&T GIS, Novell, and Fujitsu pay $1 million to join OSF. This is primarily in recognition of the threat from Microsoft and the need to further standardize.

RedHat is founded.

Linux 1.0 is released.

First Beowulf cluster at NASA.

1995

4.4BSD-Lite Release 2 is released and CSRG is disbanded at Berkeley (June 1995). Now both Berkeley and AT&T are out of the UNIX development business.

X/Open releases the UNIX95 brand. A UNIX95 products must conform in four areas:

- X/Open portability guide (XPG)
- Specification 1170 (System APIs)
- International terminal interfaces
- Network APIs including Berkeley sockets and XTI

Explosion of the World Wide Web.

1996

X/OPEN and OSF merge, founding the Open Group.

MkLinux, Linux on Mach microkernel, is released.

Linux 2.0 is released.

1997

The Single Unix Specification, V2, is released.

Linux becomes the operating system of choice of ISP.

Windows NT share in industry is arising, at the expense of UNIX.

1998

The UNIX98 brand is released. It comprises three categories of UNIX: base, workstation, server.

Compaq buys DEC.

The Open Source movement get momentum. The press discovers Linux and the Open Source movement. Torvalds appears on Forbes.

Oracle, Informix, IBM, Compaq and others announce support for Linux.

Netscape goes open source with the name Mozilla.

1999

Industry is getting interested in Linux. Renewed interested in UNIX products.

Major commercial software developers begin to release versions for Linux. SAP announces SAP/R3 for Linux.

Apple releases Mac OS X, based on Mach, and begins the Darwin project.

SCO, IBM, Sequent and Intel join in the Monterey Project, whose goal "is to deliver a single UNIX system product line consisting of UnixWare 7 (on the Intel IA-32 architecture), AIX (on the IBM Power architecture), and Monterey/64 (on the Intel IA-64 architecture)."

Sun incorporates StarDivision, and start to distribute part of Solaris sources under the Sun Community Licence.

Microsoft is ruled of monopoly in the market for personal computers.

2000

Major commercial hardware vendors (Compaq, IBM, Dell, SGI, Fujitsu) begin to sell desktop and laptop computers with Linux pre-installed.

Sun releases Solaris 8 sources under the Foundation Source code license.

Microsoft is condemned to be split in two companies.

Linux is ported to the IBM S/390.

Lockheed Martin Corp. is using Linux NetworX's cluster technology to analyze U.S. Navy aircraft.

Sun announces that it will release the source code for its Star Office suite

Sun announces that it will join the GNOME project, and to adopt the GNOME environment on its workstations in place of CDE.

IBM invests more than $200 million in a series of Linux initiatives in Europe over the next four years.

IBM and RedHat will develop versions of Linux native for all IBM platforms (AS/400, RS/6000, S390 and Netfinity). The IBM Application Developer Kit will be ported to Linux.

IBM announces AIX 5L, one of the few results of the Monterey project.

Caldera acquires UnixWare, the Server Software Division and the Professional Services Division from SCO. SCO will retain its Tarantella Division. Linux will be available on UnixWare as the Linux Kernel Personality.

Many firms of the *new economy* are in financial troubles.

Sun starts to distribute the UltraSPARC III.

Sun acquires Cobalt Networks, which builds Linux-based servers.

Apple distributes MacOS X client, based on FreeBSD and Mach microkernel.

HP releases HP-UX 11i, a full 64 bit Unix which will be compatible with Linux.

SAP releases the famous ERP SAP/R3 as Open Source.

2001

Linux 2.4 is released.

Sony announces a porting of Linux to the PlayStation 2.

Sun launches the JXTA project, an open source protocol allowing peer-to-peer communication and distributed computing between any peers that are recognized on the network.

IBM presents AIX 5L, as a result of the Project Monterey. It is compatible with Linux.

Microsoft attacks the GNU Public Licence, claiming that it cannot protect the intellectual property of private and public software houses. But even the Software & Information Industry Association (SIIA) criticizes Microsoft on this, saying that "Microsoft is once again publicly making the case that innovation in the software industry should happen only at the discretion and direction of Microsoft, [...] Microsoft is employing public relations tactics to incite fear among businesses that are considering migrating to the Open Source model."

IDC forecasts in the U.S. Multiuser System Market by 2004:

OS	1999	2004	difference
Unix	$11.4 billion	$14.9 billion	+30%
Windows NT	$4.7 billion	$12.8 billion	+172%
Linux	$367 million	$4.1 billion	+1017%

Nokia adopts Linux to develop applications for its Media Terminal home entertainment system.

HP adopts Debian as the selected development platform for Linux work at HP.

IBM announces a new supercluster running Linux, installed at the National Center for Supercomputing Applications (NCSA). Capable of 1 trillion calculations per second, the cluster of 160 new IBM Itanium-based systems will be the most powerful Linux supercluster in academia.

Fujitsu, Hitachi, IBM, and NEC announce a partnership "to refine features needed to drive Linux further into the enterprise", working together with the Linux community to develop various open source projects.

Microsoft releases Office X, the porting of the Office suite to MacOS X - which is, ultimately, a Unix variant.

2002

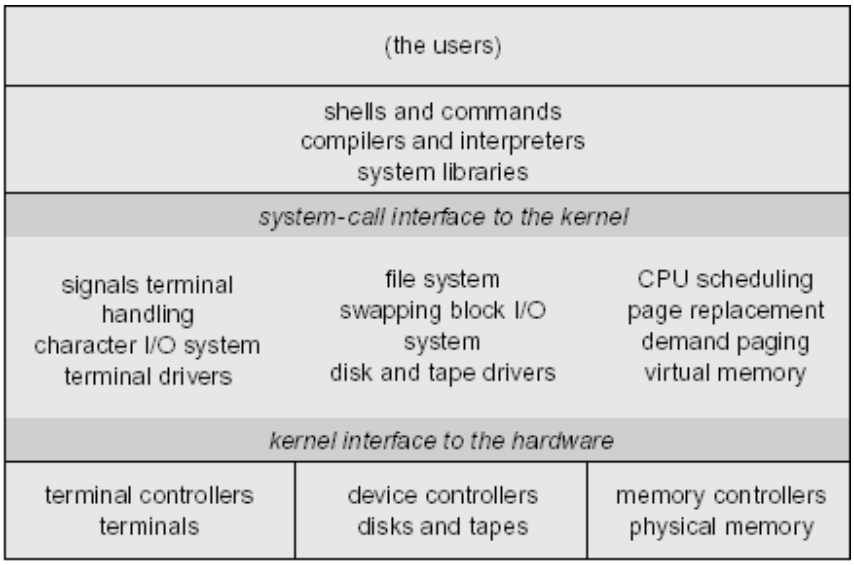
The Free Standards Group released LSB 1.1 (including a full set of common APIs and a development package), and Li18nux (an internationalization guide), two tools intended to ensure that all Linux applications can run on any Linux Standard Base-compliant version.

After the acquisition of Cobalt, Sun embraces aggressively the Linux operating system with a multipart program that will significantly broaden the offerings of Linux on low-end Sun servers and commit new resources to the ongoing development of the Open Source operating system.

21.2 PRINCIPI DI PROGETTO

Unix è stato progettato come sistema time–sharing. L'interfaccia dell'utente standard (shell) è semplice e può essere sostituita, volendo, da un'altra interfaccia. Il file system è un albero multilivello che permette agli utenti di creare le proprie sottodirectory. Ogni file di dati dell'utente è composto semplicemente da una sequenza di byte.
I file su disco e i dispositivi di I/O sono trattati nel mood più simile possibile. Quindi le dipendenze e le caratteristiche dei dispositivi sono conservate il più possibile nel kernel, dove la maggior parte di essi è racchiusa nei driver dei dispositivi.
Unix supporta processi multipli. Un processo può creare altri processi con facilità. Lo scheduling della CPU è un semplice algoritmo di priorità. Il sistema Unix 4.3BSD utilizza la paginazione su richiesta come meccanismo di supporto per la gestione della memoria e come meccanismo decisionale per lo scheduling della cpu. Lo swapping viene utilizzato se il sistema è in difficoltà a causa di un'eccessiva paginazione.
La maggior parte degli algoritmi è stata scelta per motivi di semplicità e non di velocità o raffinatezza. Lo scopo iniziale era quello di avere un kernel e alcune librerie che fornissero un piccolo gruppo di funzioni tali da permettere la costruzione di un sistema più complesso.
Unix è stato progettato da programmatori per programmatori, quindi è sempre stato interattivo ed è sempre stata attribuita priorità alle funzioni di sviluppo di programmi. Tra queste funzioni sono presenti il programma make e il Source Code Control System (SCCS).
Il SO è scritto prevalentemente in C: in questo modo sono stati enormemente semplificati i problemi di spostamento di Unix da un sistema hardware ad un altro.
I vincoli di dimensione del PDP-11, e dei primi computer utilizzati per Unix, hanno reso necessaria una certa eleganza. Dove altri sistemi utilizzano algoritmi elaborati per la gestione di condizioni patologiche, Unix esegue semplicemente un crash controllato, chiamato panic. Anziché tentare di porre rimedio a tali condizioni, unix tenta di prevenirle. Dove altri sistemi utilizzano la forza bruta o un'espansione di macro, per Unix sono stati sviluppati degli approcci più sottili o, perlomeno, più semplici.

21.3 INTERFACCIA DEL PROGRAMMATORE



Così come la maggior parte dei sistemi di calcolo, unix è formato da due parti separabili: il kernel e i programmi di sistema. Unix può essere suddiviso in strati: tutto quanto si trova sotto l'interfaccia delle system call e sopra l'hardware fisico è il kernel. Per mezzo delle system call il kernel fornisce lo scheduling della cpu, la gestione della memoria e altre funzioni del SO. I programmi di sistema utilizzano le system call supportate dal kernel per fornire funzioni utili, come la compilazione e la manipolazione di file.
Le system call definiscono l'interfaccia del programmatore con Unix; l'insieme dei programmi di sistema comunemente disponibili definisce l'interfaccia dell'utente. Le interfacce del programmatore e dell'utente definiscono il contesto che il kernel deve supportare.
La maggior parte dei programmi di sistema è scritta in C. I dettagli delle system call sono noto solo al compilatore. Questa caratteristica è una delle più importanti ragioni della portabilità dei programmi in Unix.
Le system call per Unix possono essere raggruppate approssimativamente in tre categorie: manipolazione di file, controllo dei processi e manipolazione di informazioni. La manipolazione dei dispositivi ricade sotto quella dei file in quanto sono supportati entrambi dalle stesse system call.

21.3.1 MANIPOLAZIONE DI FILE

In Unix un file è una sequenza di byte. Programmi diversi possono necessitare di diversi livelli di struttura, ma il kernel non impone una struttura ai file. Ad esempio, la convenzione per i file di testo prevede che essi siano costituiti da righe di caratteri ascii separate da un unico carattere di nuova riga, ma il kernel non è al corrente di tale convenzione.
I file sono organizzati in directory strutturate ad albero. Le stesse directory sono file che contengono informazioni su come trovare altri file. Il path–name per un file è una stringa di testo che identifica un file specificando un percorso che passa attraverso la struttura della directory e arriva fino al file. Unix ha sia path–name assoluti che path–name relativi.

Un file può essere noto con più di un nome in più di una directory. I nomi multipli sono noti con il termine di link e sono trattati allo stesso modo dal SO. La versione 4.3BSD supporta anche i link simbolici, che sono file contenenti il path–name di un altro file. I due tipi di link sono conosciuti anche come hard link e soft link. I soft link (simbolici) diversamente dagli hard link, possono puntare su directory e possono superare i confini del file system.
Nel file system i dispositivi hardware hanno dei nomi. Questi file speciali di dispositivo sono riconosciuti dal kernel come interfacce dei dispositivi, tuttavia l'utente può accedervi tramite la maggior parte delle stesse system call con cui accede agli altri file.
Tutte le convenzioni riguardanti la locazione di file e directory specifici sono state definite dai programmatori e dai programmi; il kernel del SO necessita solo di etc/init, che viene utilizzata per inizializzare i processi di attivazione dei terminali, per poter essere utilizzato.
Le system call di base per la manipolazione dei file sono create, open, read, write, close, unlink (rimozione dei link), trunc (riduce a 0 la lunghezza del file), lseek (reimpostare la posizione del puntatore del file), dup e dup2 (producono la copia del descrittore di file già esistente), fcntl (imposta i parametri di file), ioctl (manipolazione dei parametri dei dispositivi), stat (ottenere le informazioni su dispositivi), rename, chown, symlink (crea link simbolico).

21.3.2 CONTROLLO DEI PROCESSI

Un processo è un programma in esecuzione. I processi sono identificati dal loro identificatore di processo (numero intero, PID). Un nuovo processo viene creato attraverso la system call fork, ed è formato da una copia dello spazio di indirizzi del processo originale: è costituito cioè dallo stesso programma e dalle stesse variabili con gli stessi valori. Entrambi i processi, padre e figlio, continuano l'esecuzione dall'istruzione che segue la fork, con una differenza: per il nuovo processo (figlio) il codice di ritorno per la fork è 0, mentre l'identificatore del processo figlio (diverso da zero) viene riportato al padre.
Generalmente dopo una fork viene utilizzata la system call execve per sostituire il proprio spazio di memoria virtuale con un programma nuovo. Questa system call carica in memoria un file binario (distruggendo l'immagine in memoria che contiene le system call execve) e ne avvia l'esecuzione.
Un processo può terminare utilizzando exit, mentre il processo padre può attendere che questo evento si verifichi utilizzando wait. Se il processo figlio si interrompe, il SO simula la call exit. La system call wait fornisce al processo l'ID del figlio che ha terminato, per cui il padre sa quale dei suoi figli ha terminato . una seconda system call, wait3, permette al padre di raccogliere anche informazioni statistiche sul figlio. Nel lasso di tempo che intercorre tra l'uscita del figlio e il completamento di una delle system call wait da parte del padre, il figlio è defunto. Un processo defunto on può far nulla, ma esiste esclusivamente per permettere al padre di raccogliere informazioni sul suo stato. Se il processo padre di un processo defunto termina prima del figlio, il processo defunto viene ereditato dal processo init e diviene un processo zombie.
Il metodo più semplice per la comunicazione tra i processi viene realizzato tramite pipe che possono essere create prima della fork, i cui punti terminali vengono impostati tra

fork ed execve. Una pipe è fondamentalmente una coda di byte tra due processi. È possibile accederci dal descrittore di file, come avviene per un file ordinario. Un processo scrive nella pipe e l'altro legge dalla pipe. L dimensione del pipe system originale è fissata dal sistema.
Tutti i processi utenti discendono da un unico processo originale, chiamato init. Ciascuna porta di terminale disponibile per uso interattivo da un processo getty generato, per essa, da init. Il processo getty inizializza i parametri della linea del terminale e di mette in attesa del nome di login di qualche utente che passa attraverso una execve come argomento a un processo login. Il processo login raccoglie la password dell'utente, la cifra e la confronta con la stringa cifrata prelevata dal file etc/passwd. Il processo login esegue una shell, o interprete dei comandi, dopo aver impostato l'identificatore utente numerico (UID) del processo a quello dell'utente che ha effettuato il login.
Il kernel può utilizzare due identificatori d'utente; l'identificatore d'utente effettivo viene utilizzato per determinare i permessi di accesso ai file. Se un file contenente un programma che viene caricato da una call execve ha il bit setuis impostato nel suo inode, l'identificatore d'utente effettivo del processo viene impostato al valore dell'identificatore utente del proprietario del file, mentre l'identificatore d'utente reale viene lasciato com'era. Esiste anche setgid (set group ID).

21.3.3 SIGNAL

Le signal sono uno strumento che permette di gestire condizioni eccezionali simili agli interrupt software. Esistono venti signal diverse, ciascuna delle quali corrisponde a una condizione distinta. Una signal può essere generata da un interrupt della tastiera, da un errore in un processo oppure da un certo numero di eventi asincroni (come timer o segnali di controllo di processo in arrivo alla shell). Quasi tutte le signal possono essere generate anche dalla system call kill.
La signal interrupt, SIGINT, viene utilizzata per fermare un comando prima che venga completato. La signal quit, SIGQUIT ferma il programma attualmente in esecuzione e riversa la sua corrente immagine di memoria su un file chiamato core, che si trova nella directory corrente. Il file core può essere utilizzato dai debugger. SIGILL viene prodotta da un istruzione illegale e SIGSEGV da un tentativo di indirizzare la memoria fuori dallo spazio legale di memoria virtuale di un processo.
Possono essere effettuate delle modifiche sia per far ignorare la maggior parte delle signal che per richiamare una routine del processo utente. Un gestore di signal può sicuramente eseguire una delle due seguenti cose prima di restituire il controllo dalla cattura di una signal: richiamo della system call exit, oppure modifica di una variabile globale. Esiste una signal kill, SIGKILL che viene utilizzata per sopprimere un processo fuggiasco che sta ignorando altre signal, come SIGINT e SIGQUIT. Non esiste priorità relativa tra signal unix; se due signal diverse vengono inviate allo stesso processo nello stesso momento, non si può sapere quale delle due arriverà per prima.
L'intenzione originale per cui erano state create le signal era per gestire condizioni eccezionali, ma ora il loro uso è stato ampliato.

21.3.4 GRUPPI DI PROCESSI

Spesso gruppi di processi correlati collaborano nello svolgimento di un compito comune. Ad esempio, alcuni processi possono creare pipe e utilizzarle per comunicare. Un siffatto insieme di processi viene chiamato gruppo di processi. A tutti i processi di un gruppo possono essere inviate delle signal. Normalmente un processo eredita il proprio gruppo dal padre, ma la system call setgrp permette ad un processo di cambiare il proprio gruppo di appartenenza.

21.3.5 MANIPOLAZIONE DI INFORMAZIONI

Esistono system call per impostare e restituire un intervallo del timer (gettimer, settimer) e l'ora corrente (gettimeofday, settimeofday) in microsecondi. Inoltre, i processi possono richiedere il loro identificatore di processo (getpid) o di gruppo (getgid), il nome della macchina su cui sono in esecuzione (gethostname) e molti altri valori.

21.3.6 ROUTINE DI LIBRERIA

L'interfaccia delle system call con Unix è supportata ed ampliata da una vasta raccolta di routine di libreria e di file header. Questi ultimi forniscono la definizione di strutture dati complesse utilizzate nelle system call. Inoltre, una vasta libreria di funzioni fornisce un ulteriore supporto di programmazione.
Ad esempio, le system call di I/O di Unix procedono alla lettura e alla scrittura di blocchi di byte. Alcune applicazioni possono richiedere di leggere o scrivere un solo byte alla volta. Anche se fosse possibile, tale operazione richiederebbe una system call per byte, con un overhead molto elevato. Esiste invece un insieme di routine standard (<stdio.h>) che fornisce un'altra interfaccia per leggere e scrivere parecchie migliaia di byte alla volta usando buffer locali e, quando è richiesto l'I/O, esegue i trasferimenti tra questi buffer, che si trovano in memoria utente. Il pacchetto di I/O standard supporta anche l'I/O formattato.

21.5 GESTIONE DEI PROCESSI

Un importante problema da affrontare nella progettazione dei SO è al rappresentazione dei processi. La differenza fondamentale tra unix e molti altri sistemi consiste nella facilità con cui più processi possono essere creati o manipolati. In Unix questi processi vengono rappresentati da vai blocchi di controllo. Non esistono blocchi di controllo di sistema accessibili nello spazio degli indirizzi virtuali di un processo utente; i blocchi di controllo associati a un processo sono memorizzati nel kernel. Le informazioni in questi blocchi sono utilizzate dal kernel per il controllo dei processi e lo scheduling della cpu.

21.5.1 PROCESS CONTROL BLOCK

La più essenziale struttura di dati associata ai processi è la process structure. Tale struttura contiene tutto quanto è necessario conoscere su un processo quando questo viene sottoposto a swap out, come il suo unico identificatore di processo, informazioni sullo

scheduling (come la priorità di processo) oltre alle strutture di dati relative a signal, timer e quote e puntatori ad latri blocchi di controllo. Esiste un array di process structure la cui lunghezza è definita dalla fase di link del sistema. Le process structure che contengono i processi pronti vengono collegate tra loro dallo scheduler che le organizza in una lista doppiamente concatenata (la ready queue); un ogni process structure esistono diversi puntatori: uno al padre del processo, uno al figlio più giovane vivente, diversi puntatori ad altri parenti interessati; esiste anche un elenco dei processi che condividono lo stesso codice del programma.
Lo spazio degli indirizzi virtuali di un processo utente è suddiviso nei segmenti di testo, dati e stack.
Ogni processo con testo condivisibile ha nella propria process structure un puntatore a una text structure. La text structure registra il numero dei processi che stanno usando quel segmento di testo, contiene un puntatore a una lista delle loro process structure e la posizione necessaria sul disco della tabella delle pagine del segmento di testo quando questo viene sottoposto a swapping. La text structure è sempre residente nella memoria centrale; l'array di tali strutture viene allocato nella fase di link del sistema.
Le tabelle delle pagine registrano informazioni relative al mapping della memoria virtuale dei processi sulla memoria fisica. La process structure contiene puntatori alla tabella delle pagine, da utilizzare quando il processo risiede nella memoria centrale, oppure l'indirizzo del processo sul dispositivo di swap quando il processo viene sottoposto a swapping. Non esiste una tabella delle pagine distinta per un segmento di testo condiviso; gli elementi relativi alle pagine di ogni processo che condivide il segmento di testo si trovano nella tabella delle pagine del processo.
Le informazioni riguardanti il processo che servono solo quando il processo è residente (cioè non sottoposto a swap out) sono tenute nella user structure (o u structure) anziché nella process structure. La u structure è mappata in sola lettura nello spazio di indirizzi virtuale dell'utente in modo che i processi possano leggerne i contenuti; essa può essere scritta sul kernel.
La system call fork alloca una nuova process structure (con un nuovo identificatore di processo) per il processo figlio e copia la user structure. Normalmente non è necessaria una nuova text structure, in quanto i processi condividono il testo; i contatori e le liste appropriate vengono solamente aggiornati. Viene costruita una nuova tabella delle pagine e viene allocata nuova memoria centrale per i segmenti di dati e di stack del processo figlio. Copiando la user structure vengono conservati i descrittori dei file aperti, gli identificatori d'utente e di gruppo, la gestione delle signal e la maggior parte delle caratteristiche di questi tipo relative a un processo.
La system call vfork non copia i dati e lo stack del nuovo processo, piuttosto quest'ultimo condivide semplicemente la tabella delle pagine del vecchio processo. Vengono comunque create una nuova user structure e una nuova process structure. Questa system call viene comunemente utilizzata da una shell per eseguire un comando ed attendere che questo comando venga completato. Il processo padre utilizza vfork per produrre il processo figlio. Dal momento che il processo figlio vuole utilizzare immediatamente una execve per cambiare completamente il proprio spazio di indirizzi virtuali, non è necessario effettuare una copia completa del processo padre. Le strutture di dati necessarie per manipolare le pipe possono essere conservate in registri tra la vfork e la execve. In un processo è possibile chiudere file senza influire su un altro processo, in

quanto le strutture di dati del kernel implicate dipendono dalla user structure, che non è condivisa. Il padre, quando chiama una vfork, viene sospeso finché il figlio non chiama una execve o termina, per cui il padre no cambia la memoria di cui il figlio necessita. Se il processo padre è grande, la vfork può produrre un notevole risparmio di tempo di cpu. Tuttavia la vfork è una system call abbastanza pericolosa, in quanto ogni cambiamento nella memoria si verifica in entrambi i processi fiche non si verifica una execve.

21.5.2 SCHEDULING DELLA CPU

Lo scheduling della cpu di Unix è progettato in modo da portare alcuni benefici ai processi interattivi. Ai processi vengono assegnati piccoli quanti di tempo per mezzo di un algoritmo di priorità che si riduce allo scheduling round–robin per job CPU–bound.
A ogni processo è associata una priorità di scheduling; più grandi sono i numeri, minore è la priorità.
Più tempo di cpu viene accumulato da un processo, minore diventa la sua priorità. Per impedire la starvation si ricorre all'invecchiamento del processo.
I sistemi Unix precedenti utilizzavano un quanto di secondo per lo scheduling round–robin. La versione 4.3BSD effettua lo scheduling dei processi ogni 0.1 secondi e ricalcala le priorità ogni secondo. Nel kernel un processo non può essere prelazionato da un altro processo.

21.6 GESTIONE DELLA MEMORIA

21.6.1 SWAPPING

I sistemi unix che precedevano il 3BSD utilizzavano lo swapping esclusivamente per gestire le contese di memoria tra processi. Se la contesa era eccessiva, i processi venivano sottoposti a swap out finché non si rendeva disponibile una quantità di memoria sufficiente. Quindi, alcuni processi grandi potevano imporre a molti processi piccoli di uscire dalla memoria e un processo più grande della memoria centrale non–kernl non poteva assolutamente essere eseguito. Il segmento di dati del sistema (la u structure e lo stack del kernel) e il segmento dei dati utente erano conservati in memoria centrale contigua per motivi di efficienza nel trasferimento per swapping, quindi la frammentazione esterna della memoria poteva costituire un grave problema.
L'allocazione della memoria centrale e dello swapping veniva effettuata con il criterio first–fit. Quando la dimensione dell'immagine aumentava, o per espansione dello stack, o per espansione dei dati, veniva allocato un nuovo pezzo di memoria sufficientemente grande per contenere tutta l'immagine. L'immagine della memoria veniva copiata, la memoria vecchi liberata e le tabelle appropriate venivano aggiornate. Se nessun singolo pezzo di memoria centrale era sufficientemente grande, il processo veniva sottoposto a swap out in modo da poter essere poi sottoposto a swap in una volta ottenuta la nuova dimensione.
Lo scheduler process (conosciuto anche come swapper) si occupava di decidere quali processi dovevano essere sottoposti a swap in o swap out. Lo scheduler veniva attivato, almeno ogni 4 secondi. Era più probabile che un processo venisse sottoposto a swap out se era inattivo, se era rimasto a lungo in memoria centrale, oppure se era grande; se non

venivano trovati candidati ovvi, venivano selezionati altri processi in base all'età. Era più probabile che un processo venisse sottoposto a swap in se era stato sottoposto al lungo a swap out, oppure se era piccolo.
Se non esisteva la necessità di sottoporre dei job a swap out, nella tabella dei processi veniva cercato un processo che meritasse di essere caricato in memoria; tale processo veniva determinato in base alla sua piccola dimensione e al periodo di tempo per cui era rimasto nell'area di swap. Se la memoria disponibile non era sufficiente, dei processi venivano sottoposto a swap out fino a crearla.

21.6.2 PAGINAZIONE

La frammentazione esterna della memoria è eliminata dalla paginazione. Naturalmente esiste sempre la frammentazione interna, ma è trascurabile se le pagine sono di dimensioni ragionevolmente piccole. Anche lo swapping può essere ridotto al minimo perché più job possono essere conservati in memoria centrale, poiché la paginazione consente l'esecuzione con solo parti di ciascun processo in memoria.
La paginazione su richiesta viene effettuata in maniera molto semplice. Quando un processo richiede una pagina che non è in memoria si verifica un page fault al kernel, viene allocato un frame di memoria centrale e la pagina richiesta, che si trova sul disco, viene letta nel frame.
Le ottimizzazioni sono poche. Se la pagina richiesta si trova ancora nella tabella delle pagine per il processo, ma è stata contrassegnata come invalida dal processo di sostituzione delle pagine, può essere contrassegnata come valida e utilizzata senza effettuare nessun trasferimento d I/O. Analogamente, le pagine possono essere recuperate dalla lista dei frame liberi. Se la maggior parte dei processi è avviata, molte delle relative pagine vengono prepaginate e inserite nella free list per il recupero con questo meccanismo.
Se la pagina deve essere prelevata dal disco, deve rimanere bloccata in memoria per tutta la durata del trasferimento. Questo blocco assicura che la pagina non può essere selezionata per la sostituzione. Una volta prelevata e mappata adeguatamente, la pagina deve rimanere bloccata se su di essa è in corso un I/O fisico.
L'algoritmo di sostituzione delle pagine è una versione modificata dell'algoritmo seconda chance (clock). La mappa di tutta la memoria centrale non kernel (la core map o cmap) viene percorsa linearmente e ripetutamente da una clock hand software. Quando la clock hand raggiunge un certo frame, se questo è contrassegnato come frame utilizzato da qualche condizione software oppure se il frame è già libero, quest'ultimo non viene toccato e la clock hand si sposta rapidamente verso il frame successivo, altrimenti, viene localizzato il testo corrispondente oppure l'elemento della tabella delle pagine del processo per questo frame. Se l'elemento è già invalido, il frame viene aggiunto alla free list, altrimenti l'elemento della tabella delle pagine viene reso invalido, ma recuperabile; questo significa che se non viene paginato in uscita la successiva volta che viene richiesto, può essere reso di nuovo valido.
La clock hand LRU è implementata nel pagedaemon, che è il processo 2 (lo scheduler 0 e init 1). Il processo 2 passa quasi tutto il suo tempo dormendo, ma secondo una programmazione di timeout nel giro di qualche secondo vengono effettuati diversi controlli per vedere se sia necessaria qualche operazione, nel qual caso il processo 2

viene svegliato. Ogni volta che il numero dei frame liberi scende al di sotto di una determinata soglia lostfree, il pagedaemon viene svegliato; tuttavia, se esiste sempre una grande quantità di memoria libera il pagedaemon non impone carico sul sistema perché non viene mai eseguito.
Lo scorrimento della clock hand a ogni risveglio del processo pagedaemon (cioè il numero dei frame sottoposto a scansione, che in genere è più alto del numero dei frame paginati in uscita), viene determinato sia dal numero dei frame che mancano per raggiungere il valore lostfree, che dal numero dei frame che lo scheduler ritiene necessario per vari motivi. Se il numero dei frame liberi arriva alla soglia lostfree prima che lo scorrimento previsto sia completato, la clock hand si ferma e il processo pagedaemon riprende a dormire. I parametri che determinano l'intervallo di scorrimento della clock hand sono determinati all'avvio del sistema in base alla quantità di memoria centrale in modo che il pagedaemon non utilizzi più del 10% di tutto il tempo di cpu.
Se lo scheduler decide che il sistema di paginazione è sovraccarico, interi processi vengono sottoposto a swap out finchè il sovraccarico non si esaurisce. Generalmente tale swapping avviene solo se sono soddisfatte diverse condizioni: il carico medio è alto, la memoria libera è scesa al di sotto di un basso limite, minfree, e la memoria media disponibile in un periodo recente è inferiore a una quantità desiderabile, desfree, dove lostfree>desfree>minfree. In altre parole, lo swapping è causato solo da una carenza cronica di memoria, con diversi processi che tentano di entrare in esecuzione; anche in questo caso la memoria libera deve essere estremamente bassa in quell'istante.
Il parametro lostfree è generalmente un quarto della memoria nella mappa attraversata dalla clock hand, mentre desfree e minfree sono generalmente costanti, anche su sistemi diversi, ma sono limitati a frazioni della memoria disponibile.
Lo scheduling della cpu, lo swapping e la paginazione interagiscono tra loro: minore è la priorità di un processo, più è probabile che le sue pagine vengano paginate in uscita e che venga sottoposto interamente a swapping.
Nella scelta dei processi da sottoporre a swapping, l'invecchiamento protegge contro il thrashing, ma la paginazione permette di ottenere lo stesso risultato con maggiore efficacia. In teoria, i processi non vengono sottoposti a swap out a meno che non siano inattivi, poiché ogni processo necessita in ogni momento soltanto di un piccolo working set di pagine in memoria centrale, e il pagedaemon recupera le pagine non utilizzate per assegnarle ad altri processi.

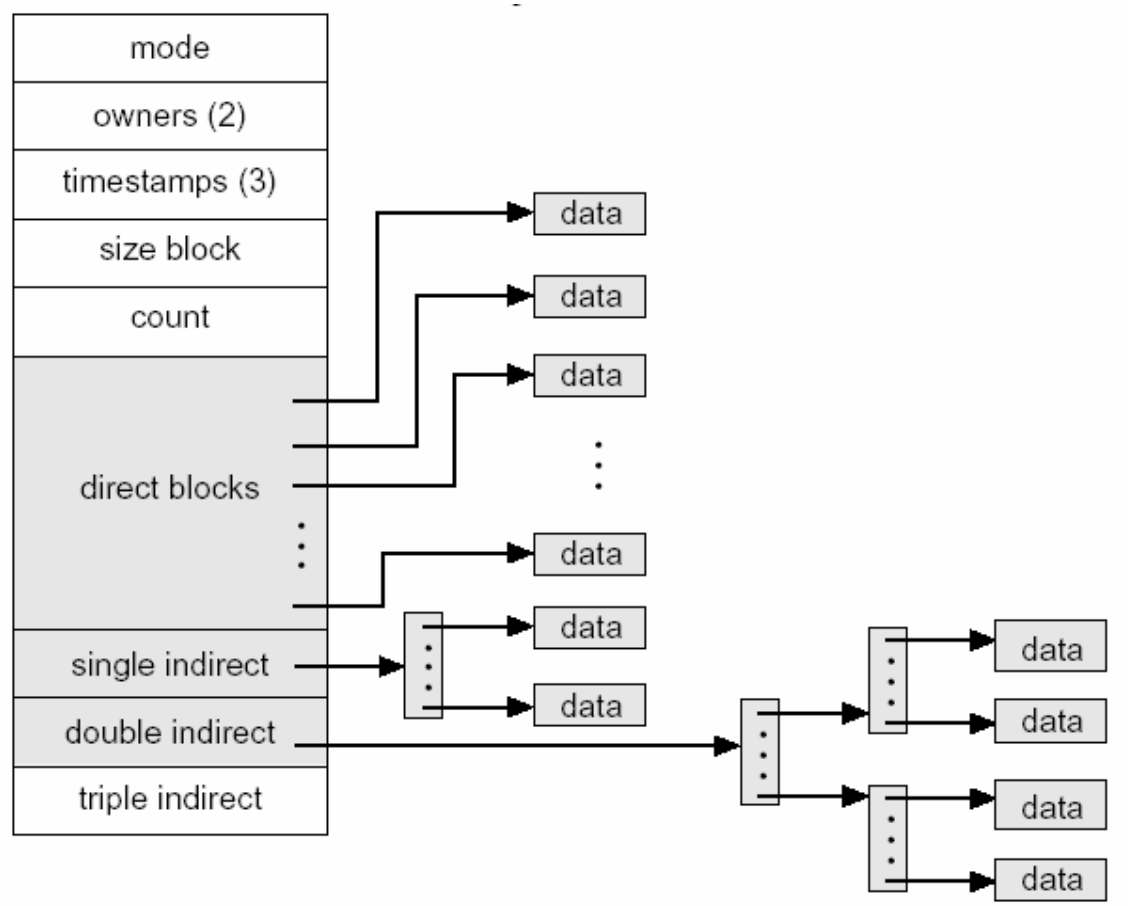
21.7 FILE SYSTEM

21.7.1 BLOCCHI E FRAMMENTI

La maggior parte del file system è costituita da blocchi dati contenenti qualsiasi cosa gli utenti abbiano inserito nei loro file. Viene ora descritto come questi blocchi di dati vengono memorizzati sul disco.
Generalmente, il settore hardware del disco ha una dimensione di 512 byte. Per motivi di velocità viene richiesta una dimensione dei blocchi maggiore di 512 byte. Tuttavia, poiché i file system unix contengono generalmente un numero molto elevato di piccoli file, blocchi molto più grandi causerebbero una frammentazione interna eccessiva. Nei primi 4.1BSD il file system era limitato a un blocco da 1024 byte.

Nel 4.2BSD si utilizzano due dimensioni dei blocchi per i file che non hanno blocchi indiretti: tutti i blocchi di un file sono di grande dimensione, tranne l'ultimo, il frammento.
Le dimensioni del blocco e del frammento sono impostate al momento della creazione del file system in base all'utilizzo a cui è destinato il file system stesso: se sono previsti molti piccoli file la dimensione dei frammenti deve essere piccola; se sono previsti trasferimenti frequenti di grandi file, la dimensione dei blocchi deve essere grande. I dettagli di implementazione impongono un rapporto massimo blocco/frammento 8:1, e una dimensione minima dei blocchi di 4K.

21.7.2 INODE



Un file è rappresentato da un inode. L'inode è u record che memorizza la maggior parte delle informazioni relative al file specifico del disco. Il nome inode deriva da index node. L'inode contiene gli identificatori dell'utente e del gruppo del file, l'ora dell'ultima modifica e dell'ultimo accesso al file, un contatore del numero di hard link (elementi di directory) al file e il tipo di file (file di base, directory, link simbolico, dispositivo a caratteri, dispositivo a blocchi o soocket). Inoltre, l'I–node contiene 15 puntatori ai blocchi del disco con il contenuto di dati del file. I primi 12 puntatori puntano a blocchi diretti, il successivo è indiretto singolo, poi un indiretto doppio e quindi un indiretto triplo (inutile perché i file non possono essere più grandi di 2^{32} byte).

21.7.3 DIRECTORY

A questo livello di implementazione non esiste distinzione tra file di base e directory; il contenuto delle directory viene conservato in blocchi di dati e le directory sono rappresentate da un inode, così come avviene per i file di base. Soltanto il campo tipo dell'inode permette di distinguere i file di base dalle directory.
Nel sistema 4.2BSD i nomi dei file hanno lunghezza variabile fino a 255 byte, per cui anche gli elementi delle directory hanno dimensione variabile. Ogni elemento contiene, nell'ordine, la lunghezza dell'elemento, il nome del file e l'inode number. Questo elemento di lunghezza variabile complica la gestione delle directory e delle routine di ricerca, ma migliora in modo considerevole la possibilità degli utenti di scegliere nomi significativi per i loro file e le loro directory.
I primi due nomi di ogni directory sono "." e "..". I nuovi elementi della directory vengono aggiunti nel primo spazio disponibile nella directory, generalmente dopo i file esistenti. Per tale scopo viene utilizzata una ricerca lineare.
L'utente fa riferimento a un file per mezzo di un pathname, mentre il file system utilizza l'inode come propria definizione del file. Quindi ilo kernel deve mappare il path–name fornito dall'utente su un inode, e a questo scopo si utilizzano le directory.
Innanzitutto viene stabilita una directory di partenza. Se il primo carattere è / la directory di partenza è root, altrimenti è la directory corrente. Nella directory di partenza vengono controllati il tipo di file e i permessi di accesso; se necessario viene indicato l'errore trovato. L'inode della directory di partenza è sempre disponibile.
Gli hard link sono elementi di directory come tutti gli altri. I link simbolici sono gestiti per la maggior parte iniziando la ricerca e prelevando il path name dal contenuto del link simbolico. E' possibile prevenire loop infiniti contando il numero di link simbolici incontrati durante una ricerca di path–name e restituendo un errore quando viene superato un dato limite (otto).
I file speciali (come i dispositivi) non hanno blocchi dati allocati sul disco. Il kernel rileva questi tipi di file, come indicato nell'inode, e richiama gli opportuni driver per gestire i loro I/O.
Una volta che l'inode è stato trovato, ad esempio dalla system call open, viene allocata una struttura di file che punta all'inode. Il descrittore di file dato all'utente fa riferimento a questa struttura. 4.3BSD utilizza una cache di nomi di directory per mantenere le traduzioni directory/inode più recenti. Questo perfezionamento aumenta considerevolmente le prestazioni del file system.

21.7.4 ASSOCIAZIONE DI UN DESCRITTORE DI FILE A UN INODE

Le system call che fanno riferimenti a file aperti indicano il file passando come argomento un descrittore di file. Il descrittore di file viene utilizzato dal kernel come indice della tabella dei file aperti per il processo in coro. Ogni elemento della tabella contiene un puntatore a una struttura file; a sua volta questa struttura di file punta sull'inode. La tabella dei file ha una lunghezza fissa che può essere impostata soltanto nella fase di boot, quindi il numero dei file aperti concorrentemente nel sistema ha un limite fissato.

Le system call read e write non hanno argomenti che fanno riferimento a una posizione nel file; piuttosto, il kernel conserva un offset del file che viene aggiornato della quantità opportuna dopo ogni read e write. L'offset può essere impostato dalla system call lseek. Poiché uno stesso file può essere aperto da più di un processo, e ciascuno di questi processi richiede un proprio offset per il file, non è opportuno conservare l'offset nell'inode. Quindi, la struttura del file è utilizzata per contenere l'offset.
Il processo figlio, dopo una fork, eredita le strutture del file, per cui diversi processi possono condividere la stessa locazione dell'offset per un file.
La struttura dell'inode a cui punta una struttura di file è una copia, in memoria centrale, dell'inode su disco. L'inode caricato in memoria ha alcuni campi in più, come un contatore del numero delle strutture di file che puntano all'inode; anche la struttura di file ha un contatore di riferimento simile a questo, che riguarda il numero dei descrittori di file che vi fanno riferimento. Quando il contatore và a 0, l'elemento non è più utilizzato e può essere recuperato e riutilizzato.

21.7.5 Strutture dei dischi

Il file system visto dall'utente è supportato da dati depositati su un dispositivo di memoria di massa, generalmente un disco. Normalmente l'utente è a conoscenza di un unico file system, ma in effetti questo file system logico può essere formato da più file system fisici, situati su dispositivi diversi. Poiché le caratteristiche dei dispostivi sono diverse, ogni dispositivo hardware definisce il proprio file system fisico. Infatti, in genere è preferibile suddividere grossi dispositivi, come i dischi, in più dispositivi logici. Ogni dispositivo logico definisce un file system fisico.
Suddividendo un dispositivo fisico in più file system è possibile ottenere parecchi vantaggi. File system diversi possono supportare usi diversi. Sebbene la maggior parte delle partizioni venga utilizzata dal file system, almeno una di esse rimane a disposizione di un'area di swap per il software della memoria virtuale. L'affidabilità è maggiore, in quanto il danno al software è generalmente limitato a l solo file system. L'efficienza può essere migliorata variando i parametri del file system (come le dimensioni dei blocchi e dei frammenti) per ciascuna partizione. Inoltre, file system separati impediscono che un programma utilizzi tutto lo spazio disponibile per un grosso file, poiché i file possono essere divisi in diversi file system. INfine, i backup dei dischi vengono eseguiti per partizioni; la ricerca di un file in un nastro di backup è più veloce se la partizione è di dimensione minore, ed è più veloce anche il restore dal nastro dell'intera partizione.
Il numero effettivo dei file system su un drive varia in base alla dimensione del disco e allo scopo del sistema di calcolo nel suo insieme. UN file system, il file system root, è sempre disponibile. Altri file system possono essere monatti, vale a dire integrati nella gerarchia delle directory del file system root.
Un bit nella struttura degli inode indica che l'inode ha un file system montato su di esso. Un riferimento a questo file causa una ricerca nella tabella di montaggio allo scopo di trovare il numero del dispositivo montato. IL numero del dispositivo è indispensabile per trovare l'inode della directory root del file system montato e utilizzarlo. Viceversa, se un elemento del path name è ".." e la directory su cui viene effettuata la ricerca è la directory root di un file system montato, viene effettuata una ricerca nella tabella di montaggio per trovare l'inode su cui è montata e utilizzarlo.

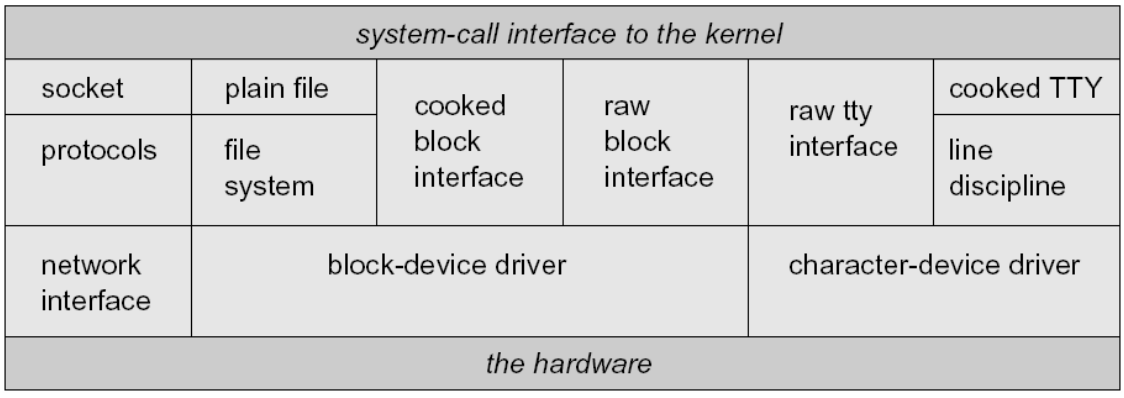
Ogni file system è una risorsa di sistema distinta e rappresenta un insieme di file. Il primo settore sul dispositivo logico è il blocco di boot, che può contenere un programma di bootstrap primario, il quale può essere utilizzato per chiamare un programma di bootstrap secondario che risiede nei successivi 7,5K. Per un sistema è sufficiente che una sola partizione contenga i dati del blocco di boot, dei duplicati possono comunque essere installati dal gestore del sistema, attraverso programma privilegiati, per consentire il booting quando la copia principale è danneggiata. Il superblock contiene parametri statici del file system, come la dimensione totale del file system, la dimensione del blocco e frammento dei blocchi di dati e parametri che influiscono sulle politiche di allocazione.

21.7.7 POLITICHE DI CONFIGURAZIONE E ALLOCAZIONE

Per identificare un file, il kernel utilizza una coppia <numero di dispositivo logico, numero di inode. Il numero di dispositivo logico definisce il file system implicato. Gli inode del file system sono numerati in sequenza. Nel file system della versione 7, tutti gli inode si trovano in un array che segue immediatamente un singolo superblock all'inizio del dispositivo logico, mentre i blocchi di dati seguono gli inode. L'inode number è effettivamente solo un indice in questo array.
Con il file system della versione 7, il blocco di un file può trovarsi in un qualsiasi punto del disco, tra la fine dell'array degli inode e la fine del file system. I blocchi liberi sono conservati in una lista concatenata all'interno del superblock. I blocchi vengono inseriti in testa alla lista dei blocchi liberi e da qui vengono rimossi quando sono richiesti per servire nuovi file o per estendere file esistenti. Quindi, i blocchi di un file possono trovarsi a distanza arbitraria dall'inode e l'uno dall'altro. Inoltre, più viene utilizzato un file system di questo tipo, più i blocchi di un file diventano disorganizzati. Questo processo può essere invertito solo reinizializzando ed effettuando il restore di tutto il file system, ma la osa non è molto conveniente.
Un'altra difficoltà consiste nel fatto che l'affidabilità del file system non è garantita. Per motivi di velocità, il superblock di ogni file system montato viene conservato in memoria. In questo modo, al kernel viene permesso di accedere rapidamente a un superblock, soprattutto per utilizzare la lista dei blocchi liberi. Ogni 30 secondi il superblock viene scritto su disco, grazie a un programma di aggiornamento che usa la system call sync. Tuttavia, non è raro che dei bug nel sistema o guasti all'hardware causino un crash del sistema che, tra due aggiornamenti del disco, distrugge il superblock caricato in memoria centrale. Quindi la lista dei blocchi liberi non riflette accuratamente lo stato del disco; per ricostruirlo è necessaria l'esecuzione di un lungo esame di tutti i blocchi del file system.
L'implementazione del file system 4.2BSD è radicalmente diversa. La differenza fondamentale (oltre all'aggiunta dei link simbolici e nomi lunghi fino a 255 caratteri) è legata all'allocazione dello spazio. La novità più importante del 4.3BSD è il gruppo di cilindri che è stato introdotto per permettere di localizzare i blocchi di un file. Ogni gruppo di cilindri occupa uno o più cilindri consecutivi del disco, per cui gli accessi al disco all'interno del gruppo di cilindri richiede un movimento minimo della testina. Ogni gruppo di cilindri ha un superblock, un blocco di cilindri, un array di inode e alcuni blocchi di dati.
Il superblock è identico in qualsiasi gruppo di cilindri, per cui, in caso di guasto del disco, può essere recuperato da un gruppo qualsiasi. Il blocco dei cilindri contiene i parametri

dinamici del particolare gruppo di cilindri, coma la mappa dei bit dei blocchi e frammenti di dati liberi e la mappa di bit degli inode liberi. Qui è conservata anche al statistica sul recente progresso della strategia di allocazione.
Le informazioni di intestazione di un gruppo di cilindri, il superblock, il blocco dei cilindri e gli inode, non si trovano sempre all'inizio di un blocco di cilindri. Se così fosse, le informazioni di intestazione di ogni gruppo potrebbero trovarsi tutte sullo stesso piatto del disco: la rottura della testina potrebbe cancellarle tutte. Quindi, ogni gruppo di cilindri ha le proprie informazioni di intestazione a un offset diverso dall'inizio del gruppo.
E' probabile che il comando ls legga tutti gli inode dei file contenuti in una directory, per cui conviene che tutti questi inode siano molto vicini fra loro. Per questo motivo, l'inode di un file è generalmente allocato nello stesso gruppo di cilindri in cui è allocato l'inode della directory padre del file. Tuttavia non è possibile circoscrivere tutto, per cui l'inode di una directory nuova viene inserito in un gruppo di cilindri diverso da quello della sua directory padre. Il gruppo di cilindri scelto per l'inode di questa nuova directory è quello con il maggior numero di inode non utilizzati.
Per ridurre i tempi di posizionamento della testina relativi all'accesso ai blocchi di dati di un file, i blocchi vengono allocati nello stesso gruppo di cilindri il più spesso possibile. Poiché a un file singolo non può essere permesso di prelevare tutti i blocchi di un gruppo di cilindri, un file che supera una certa dimensione (come 2 MB) presenta un'ulteriore allocazione di blocchi rediretta su un gruppo di cilindri diverso, dove il nuovo gruppo viene scelto tra quelli che hanno mediamente più spazio libero. Se il file continua a crescere, l'allocazione viene nuovamente rediretta (ad ogni Megabyte) su un altro gruppo di cilindri. Quindi, è probabile che tutti i blocchi di un piccolo file si trovino nello stesso gruppo di cilindri, così come è tenuto basso il numero dei lunghi riposizionamenti della testina che si verificano all'accesso di un grosso file.

21.8 SISTEMA DI I/O



Il sistema di I/O è formato da un sistema di buffer cache, codice generale per driver di dispositivi e driver per specifici dispositivi hardware. Soltanto il driver di un dispositivo conosce le caratteristiche di quel dispositivo specifico.
Nel 4.3BSD esistono tre principali tipi di I/O: dispositivi a blocchi, dispositivi a carattere e interfaccia socket.

I dispositivi a blocchi comprendono dischi e nastri. La loro caratteristica discriminante è che sono direttamente indirizzabili con una dimensione di blocco fissa, in genere 512 byte.
Per isolare i dettagli delle tracce, cilindri e simili dal resto del kernel occorre un driver per dispositivi a blocchi. Ai dispositivi a blocchi è possibile accedere direttamente tramite speciali file di dispositivi, come /dev/rp0, ma generalmente l'accesso è effettuato indirettamente tramite il file system. In entrambi i casi, i trasferimenti sono sottoposti a buffering tramite la block buffer cache, la quale incide profondamente sull'efficienza.
Tra i dispositivi a carattere sono compresi terminali, stampanti, ma anche quasi tutti i dispositivi che non utilizzano la block buffer cache. Ad esempio, /dev/mem è un'interfaccia alla memoria centrale fisica e /dev/null è un pozzo senza fondo per dati. Terminali e dispositivi simili utilizzano la C-list, che sono buffer più piccoli rispetto alla block buffer cache.
I dispositivi a blocchi e quelli a carattere costituiscono le classi di dispositivi più importanti. Ai driver dei dispositivi è possibile accedere attraverso un array di punti di ingresso. Esistono due array di punti di ingresso: uno per i dispositivi a blocchi, l'altro per i dispositivi a carattere. I dispositivi sono contraddistinti da una classe (blocco o carattere) e da un numero di dispositivo (device number). Il numero di dispositivo a sua volta è formato da due elementi: il major device number viene utilizzando come indice per l'array dei dispositivi che gli permette di accedere al driver appropriato. Il minor device number viene interpretato dal driver del dispositivo, ad esempio come partizione logica del disco o linea di terminale.
Il driver di un dispositivo è collegato al resto del kernel solo tramite punti di ingresso registrati nell'array per la sua classe e per l'utilizzo di sistemi di buffering comuni. Questa separazione è importante ai fini della portabilità e della configurazione del sistema.

21.8.1 BLOCK BUFFER CACHE

I dispositivi a blocchi utilizzano una block buffer cache. La BBC è formata da un certo numero di buffer header, ciascuno dei quali può puntare a una parte di memoria fisica, nonché a un numero di dispositivo e un numero di blocco sul dispositivo. I buffer header per i blocchi non correntemente in uso sono contenuti in alcune liste concatenate, per i seguenti buffer:

- Buffer recentemente utilizzati, concatenati in ordine LRU (lista LRU).
- Buffer non recentemente utilizzati o senza contenuti validi (lista AGE).
- Buffer vuoti senza memoria fisica ad essi associata (lista EMPTY).

Per migliorare l'efficienza della ricerca, i buffer di queste liste sono organizzati in modo da potervi effettuare ricerche hash in base al numero di dispositivo e di blocco. Quando un blocco viene richiesto da un dispositivo (una lettura), viene effettuata una ricerca nella cache. Se il blocco viene trovato, viene utilizzato e non è necessario il trasferimento di I/O. Se, invece, il blocco non viene trovato, viene scelto un buffer dalla lista AGE o dalla lista LRU se la lista AGE è vuota; quindi il numero di dispositivo e il numero di blocco a esso associati vengono aggiornati, se necessario gli viene trovata memoria e i dati nuovi vengono trasferiti dal dispostivo al buffer. Se non vi sono buffer vuoti, il contenuto del

buffer LRU viene scritto sul suo dispostivo, se è stato modificato, e il buffer viene riutilizzato.
In un caso di scrittura, se il blocco in questione si trova già nella block buffer cache, i nuovi dati vengono inseriti nel buffer, sovrascrivendo quelli precedenti, il buffer header viene contrassegnato ad indicare che il buffer è stato modificato e non è immediatamente necessario alcun I/O. I dati vengono scritti quando il buffer viene richiesto per altri dati. Se il blocco non si trova nella buffer cache viene scelto un buffer vuoto, come nel caso della lettura, e viene eseguito un trasferimento in questo buffer.
Le scritture di blocchi di buffer modificati vengono periodicamente imposte per minimizzare le potenziali incoerenza del file system dovute a guasti.
Il numero dei dati di un buffer del 4.3BSD può variare fino a un massimo di 8K su tutti i file system. La dimensione minima è quella del cluster di paginazione, generalmente 1024 byte. I buffer sono cluster di pagine allineati e ciascuno di essi può essere mappato soltanto su un buffer alla volta. La lista empty contiene i buffer header che vengono utilizzati quando un blocco di memoria fisica di 8K viene suddiviso in modo da contenere più blocchi di minore dimensione. Gli header necessari per questi blocchi vengono recuperati da empty.
Il numero dei dati in un buffer può aumentare vie via che un processo utente scrive altri dati oltre a quelli che sono già nel buffer. Quando si verifica questa circostanza viene allocato un nuovo buffer di grandezza sufficiente a contenere tutti i dati, e i dati originali vengono copiati nel buffer, seguiti da quelli nuovi. Se un buffer si restringe, dalla coda dei buffer vuoti ne viene estratto uno, nel quale vengono inserite le pagine in eccesso, e il buffer originario viene rilasciato per la scrittura su disco.
La dimensione della BBC può influire considerevolmente sulle prestazioni di un sistema, poiché se la buffer cache è sufficientemente grande la percentuale di cache hit può essere elevata, e il numero di trasferimenti di I/O basso.

21.8.2 Interfacce per l'accesso diretto ai dispositivi

Quasi tutti i dispositivi a blocchi hanno un'interfaccia a caratteri; queste interfacce per l'accesso diretto ai dispositivi sono chiamate raw device interfaces. Esse differiscono dalle interfacce a blocchi per il fatto che la BBC viene bypassata.
Ogni driver di disco contiene una coda di trasferimenti pendenti. Ogni record della coda specifica se si tratti di una lettura o di una scrittura; inoltre specifica un indirizzo di memoria centrale (generalmente ad incrementi di 512 byte), un indirizzo di dispositivo per il trasferimento (generalmente l'indirizzo del settore del disco) e una dimensione di trasferimento (in settori). Mappare le informazioni da un blocco di buffer su quanto richiesto per questa coda è semplice quasi quanto mappare un pezzo di memoria centrale corrispondente ad una parte di uno spazio di indirizzi virtuali di un processo utente. Questo mapping è ciò che viene eseguito, ad esempio, da una raw disk interface. In questo modo sono possibili trasferimenti diretti non sottoposti a buffering e da uno spazio di indirizzi virtuali di un utente. La dimensione del trasferimento è limitata dai dispositivi fisici, alcuni dei quali richiedono un numero pari di byte.
Il kernel esegue trasferimenti per swapping e paginazione semplicemente inserendo l'opportuna richiesta nella coda di attesa del dispositivo appropriato. Non è necessario alcun driver speciale per il dispositivo di swapping o paginazione.

21.8.3 C-LIST

I driver dei terminali utilizzano un sistema di buffering dei caratteri. Questo sistema implica il mantenimento di blocchi di caratteri, generalmente 28 byte, in liste concatenate. Esistono routine che permettono di accodare e disaccodare caratteri da queste liste. Sebbene tutti i buffer liberi di caratteri siano conservati in una singola lista, la maggior parte dei driver di dispostivi che li utilizzano limita il numero dei caratteri che possono essere accodati di volta in volta per una linea di terminale.
Una system call write su un terminale accoda i caratteri su una lista per il dispositivo. Viene avviato un trasferimento iniziale e una serie di interrupt causa il disaccodamento di caratteri e gli ulteriori trasferimenti.
Anche l'input è controllato da interrupt, però i driver dei terminali supportano generalmente due code di input. La conversione dalla prima (raw queue) alla seconda (canonical queue) viene attivata dalla routine di interrupt che inserisce un carattere di fine file sulla raw queue. Il processo che sta effettuando una lettura sul dispositivo viene svegliato e la sua fase di sistema esegue la conversione; i caratteri così inseriti nella canonical queue sono disponibili per essere riportati al processo utente per la lettura.
E' anche possibile fare in modo che il driver del dispositivo bypassi la canonical queue e restituisca i caratteri direttamente dalla raw queue. Questo modo di funzionamento, noto anche come raw mode, viene utilizzato da editor a pieno schermo e altri programmi che necessitano di reagire ad ogni pressione del tasto.

21.9 COMUNICAZIONE TRA PROCESSI

21.9.1 SOCKET

La pipe è il meccanismo IPC più caratteristico di Unix. Una pipe permette di avere un flusso unidirezionale affidabile di byte tra due processi. Tradizionalmente viene implementata come file ordinario, con poche eccezioni. Non ha nome nel file system essendo creata dalla system call pipe. La sua dimensione è fissa e quando un processo tenta di scrivere su una pipe piena viene sospeso. Una volta che tutti i dati precedentemente scritti sulla pipe sono stati letti, la scrittura continua all'inizio del file; le pipe non sono veri buffer circolari. Il vantaggio dato dalla piccola dimensione (normalmente 4096 byte) delle pipe consiste nel fatto che i dati delle pipe vengono raramente scritti effettivamente su disco; generalmente vengono conservati in memoria dalla normale block buffer cache.
Nel 4.3BSD le pipe sono implementate come un caso speciale del meccanismo socket, il quale non fornisce un'interfaccia generale solo a funzioni di tipo pipe, che sono locali su una macchina, ma anche funzioni di rete. Una pipe, persino sulla stessa macchina, può essere effettivamente utilizzata solo da due processi correlati dall'utilizzo della system call fork. Il meccanismo socket può essere utilizzato da processi non correlati.
La socket è un punto finale di comunicazione. Generalmente a una socket è connesso un indirizzo, la cui natura dipende dal dominio di comunicazione della socket. Una caratteristica di questo dominio è quella per cui processi che comunicano all'interno dello

stesso dominio utilizzano lo stesso formato di indirizzo. Una singola socket può comunicare solo in un dominio.
I domini implementati sono descritti in <sys/socket.h>. I principali sono:

- il dominio UNIX (AF UNIX)
- il dominio Internet (AF INET, AF INET6)
- il dominio Novell (AF IPX)
- il dominio AppleTalk (AF APPLETALK)

Tipi di Socket

- Stream socket forniscono stream di dati a_dabili, duplex, ordinati. Nel dominio Internet sono supportati dal protocollo TCP.
- Socket per pacchetti in sequenza forniscono stream di dati, ma i confini dei pacchetti sono preservati. Supportato nel dominio AF NS.
- Socket a datagrammi trasferiscono messaggi di dimensione variabile, preservando i confini ma senza garantire ordine o arrivo dei pacchetti. Supportate nel dominio Internet dal protocollo UDP.
- Socket per datagrammi affidabili come quelle a datagrammi, ma l'arrivo è garantito. Attualmente non supportate.
- Socket raw permettono di accedere direttamente ai protocolli che supportano gli altri tipi di socket; p.e., accedere TCP, IP o direttamente Ethernet. Utili per sviluppare nuovi protocolli.

La funzione socket ha un insieme di system call specifiche. La system call socket crea una socket. Preleva come argomento le specificazioni del dominio di comunicazione, il tipo di socket, e il protocollo da utilizzare per supportare quel tipo. Il valore restituito è un piccolo intero chiamato descrittore di socket, il quale si trova nello stesso spazio di nomi dei descrittori. Il descrittore di socket indicizza l'array dei file aperti nella u structure del kernel e ha una struttura di file allocata per esso. La struttura di file del 4.3BSD può puntare su una struttura di socket invece che su un inode. In questo caso certe informazioni sulla socket, come ad esempio il tipo, il numero di messaggi e i dati nelle sue code di input e di output sono conservate direttamente nella struttura di socket.
Perché un altro processo possa fare riferimento a una socket, questa deve avere un nome. Il nome viene associato alla socket dalla system call blind, la quale preleva il descrittore della socket, un puntatore al nome e la lunghezza del nome come stringa di byte. Il contenuto e la lunghezza della stringa di byte dipendono dal formato dell'indirizzo. Per iniziare una connessione viene utilizzata la system call connect. Dal punto di vista sintattico gli argomenti sono gli stessi della system call blind; il descrittore di socket rappresenta la socket locale e l'indirizzo è quello della socket straniera sulla quale viene effettuato il tentativo di connessione.
Molti processi che comunicano utilizzando il socket IPC seguono il modello client–server. In questo modello il processo server fornisce un servizio al processo client. Quando il servizio è disponibile, il processo server è in ascolto su un indirizzo ben noto e il processo client utilizza la connect per raggiungere il server.
Il processo server utilizza la system call listen per dire al kernel che è pronto ad accettare connessioni dai client e per specificare il numero massimo di connessioni pendenti che il kernel può accodare per l'accettazione. Infine, il server utilizza la system call accept per accettare le singole connessioni. Sia la listen che l'accept prelevano come argomento il descrittore della socket originale. L'accept restituisce un nuovo descrittore di socket

corrispondente alla nuova connessione; il descrittore della socket originale rimane ancora aperto per ulteriori connessioni.
Esistono anche system call per l'impostazione dei parametri di una connessione e per restituire l'indirizzo della socket straniera dopo un accept.
Una volta stabilita una connessione per un tipo di socket, come la strema socket, gli indirizzi dei due punti finali sono noti e per trasferire i dati non sono necessarie ulteriori informazioni. Per trasferire i dati si usano le solite read e write.
Il modo più semplice per terminare una connessione e distruggere la socket associata prevede l'utilizzo della system call close sul suo descrittore di socket. E' possibile anche terminare solo una direzione di comunicazione di una connessione duplex; in questo caso è possibile usare la system call shutdown.
Con la select si possono controllare trasferimenti di dati da più file descriptors/socket descriptor.
Le socket a messaggi si usano con le system call sendto e recvfrom.

21.9.2 SUPPORTO DI RETE

<table>
<tr><th>ISO
reference
model</th><th>ARPANET
reference
model</th><th>4.2BSD
layers</th><th>example
layering</th></tr>
<tr><td>application</td><td rowspan="3">process
applications</td><td rowspan="2">user programs
and libraries</td><td rowspan="2">telnet</td></tr>
<tr><td>presentation</td></tr>
<tr><td>session</td><td>sockets</td><td>sock_stream</td></tr>
<tr><td>transport</td><td rowspan="2">host–host</td><td rowspan="2">protocol</td><td>TCP</td></tr>
<tr><td rowspan="2">network
data link</td><td>IP</td></tr>
<tr><td>hardware</td><td rowspan="2">network
interface</td><td rowspan="2">network
interfaces</td><td rowspan="2">Ethernet
driver</td></tr>
<tr><td></td></tr>
<tr><td></td><td>network
hardware</td><td>network
hardware</td><td>interlan
controller</td></tr>
</table>