DISCALAIMER: Il testo nei riquadri rappresenta le risposte alle vecchie/possibili domande d'esame.

CAPITOLO 1

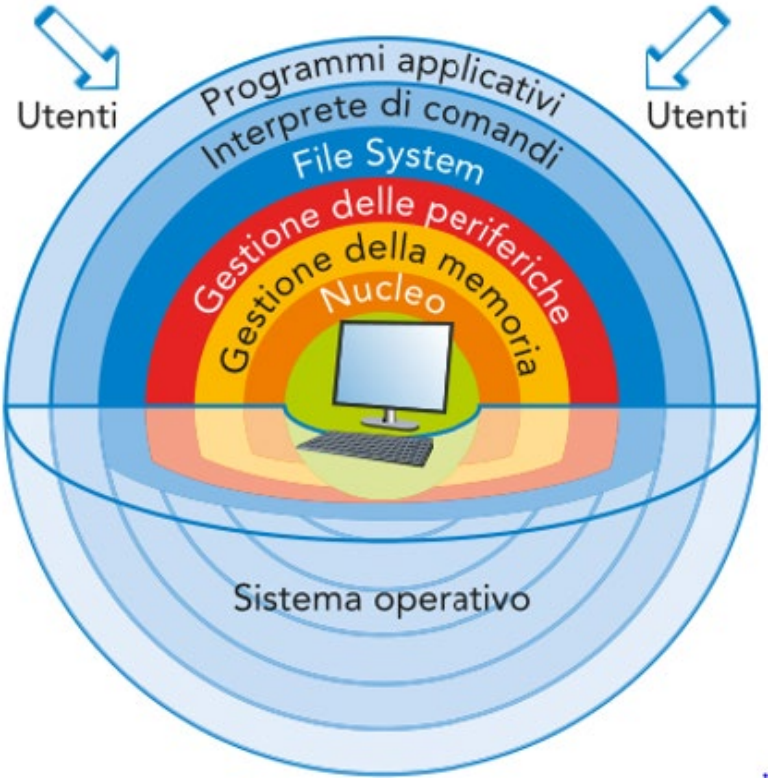
COS' É UN SISTEMA OPERATIVO?

Un sistema operativo (**S.O.**) è un insieme di programmi (**software**) che gestisce gli elementi fisici di un calcolatore (**hardware**). Il sistema operativo fornisce una piattaforma ai programmi applicativi (**applicazioni e programmi software usati dall'utente**) e agisce da intermediario fra l'utente e l'hardware dell'elaboratore.

Dal punto di vista dell'elaboratore l'S.O. è il programma più strettamente correlato al suo hardware, il cui compito è quello di gestire le risorse (**CPU, memoria, dispositivi di I/O ecc.**).

COM' È COMPOSTO UN SISTEMA OPERATIVO?

In genere, un sistema operativo ha una struttura a **cipolla**, cioè è organizzato su diversi livelli.



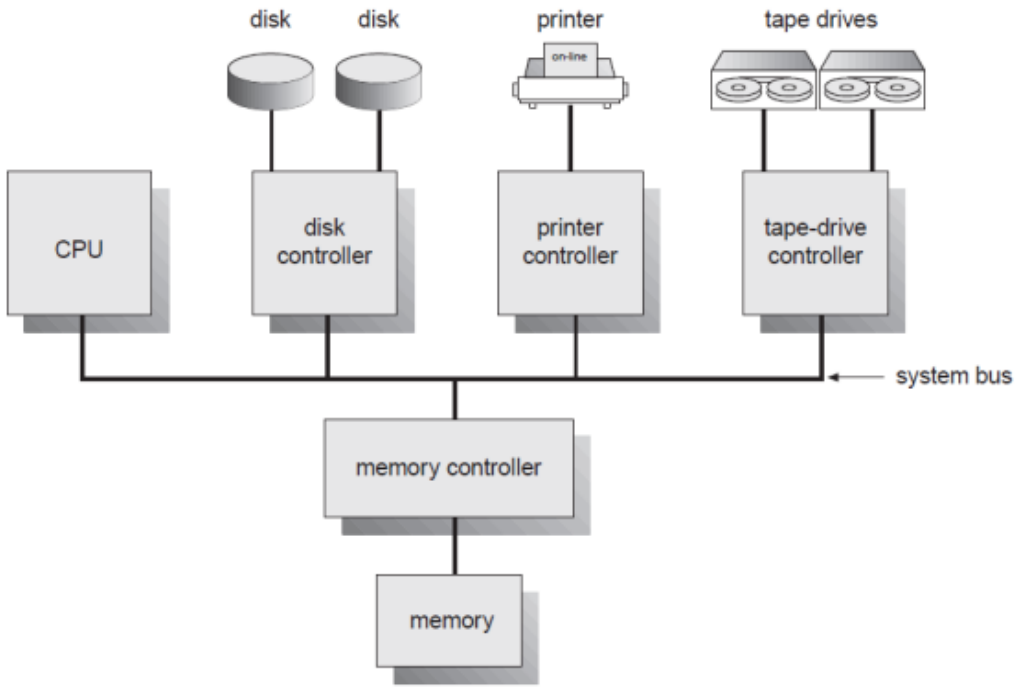
SISTEMI ELABORATIVI

Un sistema elaborativo è composto da quattro componenti:

- **Hardware**
- **Sistema operativo**
- **Programmi applicativi**
- **Utente**

Un **moderno** calcolatore **general-purpose** è composto da una o più CPU e da un certo numero di **controllori** di dispositivi connessi a un canale di comunicazione comune (**bus**).

Ciascun controller si occupa di gestire uno o più dispositivi fisici, dispone di una propria memoria interna (**buffer**) e di un insieme di registri specializzati. In genere, i sistemi operativi possiedono per ogni controllore di dispositivo un **driver del dispositivo**, cioè un pezzo di codice che descrive al sistema operativo come interagire col dispositivo.



INTERRUZIONI – I/O PROGRAMMATO

Come detto pocanzi, il compito di un sistema operativo è quello di gestire le risorse e dunque di distribuirle in maniera efficiente tra tutti i componenti del sistema elaborativo in modo da soddisfarne le richieste.

La prima generazione di elaboratori, per sopperire a questo scopo, utilizzava l'**I/O Programmato**. In breve, il sistema operativo, ad ogni intervallo di tempo prefissato, chiedeva ad ogni dispositivo collegato se avesse bisogno di qualcosa (avviare l'esecuzione di un'operazione o se avesse bisogno di determinate risorse). Questo metodo, ovviamente, si è rivelato poco efficiente

poiché il sistema, non essendo in grado di capire quando vi era effettivamente bisogno di allocare risorse per un dato dispositivo, inviava molte richieste a vuoto.

Oggi la maggior parte degli elaboratori utilizza le **interruzioni** o **interrupts**. Un'interruzione/interrupt non è altro che un segnale che viene inviato alla CPU tramite il bus di sistema. Quando la CPU riceve un segnale di interruzione, essa interrompe l'elaborazione corrente, ne salva lo stato e trasferisce il controllo alla procedura da eseguire descritta nel segnale di interrupt. Una volta terminata la procedura richiesta, la CPU riprende l'elaborazione precedente esattamente nel punto in cui era stata interrotta.

La gestione degli interrupts avviene tramite l'**array degli interrupts** che permette di gestirli anche in base alla loro priorità. L'array degli interrupts è uno dei primi servizi che vengono caricati dalla CPU all'avvio del sistema.

Possiamo considerare l'interrupt come un meccanismo hardware per il quale la CPU esce dal concetto della macchina di Von-Neumann (ovvero che non gestisce l'asincronia) e inizia a gestire le richieste in maniera asincrona.

In generale esistono due tipi di interrupt:

- **Hardware,** sono quegli interrupt generati da dispositivi esterni alla CPU, che hanno il compito di comunicare il verificarsi di eventi esterni.
- **Software,** sono delle istruzioni (INT xx) che possono essere assimilate alle chiamate di sottoprogrammi (CALL xx) ma che sfruttano il meccanismo delle interruzioni per passare il controllo dal programma chiamante a quello chiamato, e viceversa; vengono utilizzati per accedere direttamente alle risorse del Sistema Operativo (**trap ed exception**).

IMPLEMENTAZIONE DEGLI INTERRUPTS

I segnali di interrupts vengono inviati tramite un filo chiamato **linea di richiesta di interruzione** o **interrupt-request line**.

La maggior parte delle CPU ha due linee di richiesta di interruzione:

- **Nonmaskable Interrupt**, riservata ad eventi di sistema;
- **Maskable Interrupt**, utilizzata dai controllori dei dispositivi per indicare alla CPU che un dato dispositivo di I/O necessita di aiuto. Questa linea può essere disattivata durante la gestione di sequenza di istruzioni critiche (es. errori di sistema).

Per quanto riguarda l'array di interrupt, è stato implementato per evitare a un singolo gestore di interruzioni di dover cercare tutte fra tutte le possibili fonti quale ha bisogno di assistenza. Nella pratica è utilizzato il **concatenamento**

delle interruzioni in cui ogni elemento punta alla testa di un elenco di gestori; quindi, vengono chiamati a uno a uno finché non viene trovato uno in grado di gestire la richiesta.

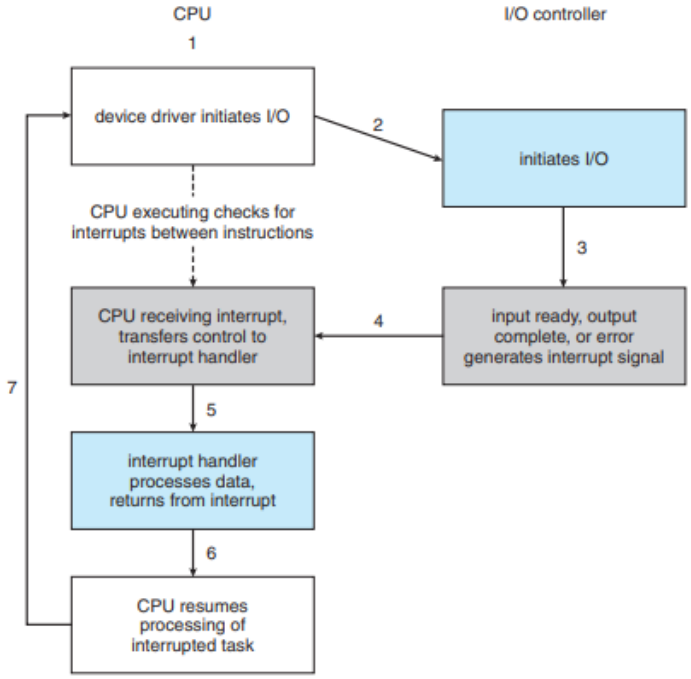


Figure 1.4 Interrupt-driven I/O cycle.

STRUTTURA DELLA MEMORIA

La CPU può prelevare ed eseguire istruzioni che si trovano **esclusivamente** nella memoria principale; quindi, i programmi per poter essere eseguiti devono essere caricati nella memoria principale detta **memoria ad accesso casuale** (**R**andom **A**ccess **M**emory, **RAM**). Solitamente, la RAM è realizzata su una tecnologia basata su semiconduttori chiamata **memoria dinamica ad accesso casuale** (**D**ynamic **R**andom Access **M**emory, **DRAM**). La RAM, tuttavia, è una memoria **volatile**, ciò significa che quando la corrente viene meno, tutti i dati ed i programmi in esecuzione vengono perduti; quindi, non è adatta per memorizzare dati che devono essere conservati nel medio-lungo periodo o permanentemente. A tale scopo, la maggior parte dei sistemi elaborativi, utilizza una o più **memorie secondarie**, che sono in grado di memorizzare permanentemente grandi quantità di informazioni.

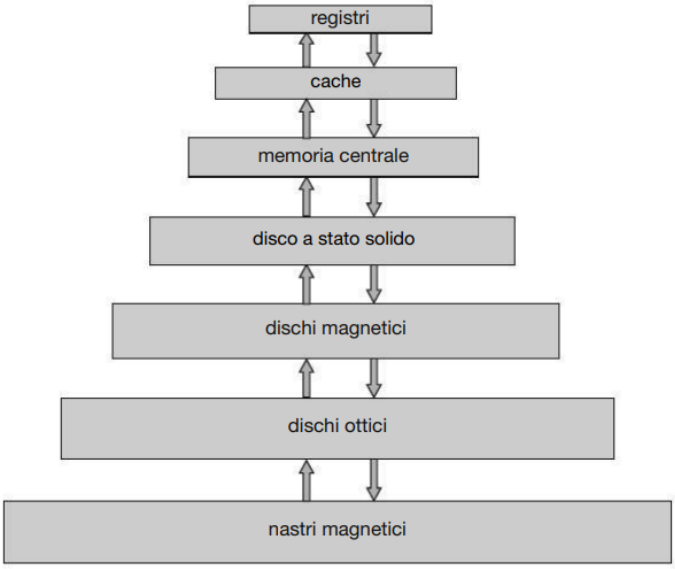
I dispositivi più comunemente utilizzati sono: i **dischi magnetici** (**H**ard-**D**isk **D**river, **HDD**) e i **dispositivi di memoria non volatile** (**NVM**).

Oltre alle memorie secondarie sopra elencate esistono altri tipi di memorie che sono usate di rado o non sono più in uso o vengono usate per conservare dati

e programmi statici come le **EEPROM** che sono memorie di sola lettura elettronicamente cancellabili e programmabili.

Le differenze sostanziali tra le varie tipologie di memoria sono: **dimensioni**, **velocità**, **costo** e **volatilità**.

SCHEMA DELLA GERARCHIA DELLA MEMORIA:



I primi quattro livelli della gerarchia della memoria sono costituiti da **memoria a semiconduttore**. I dischi NVM al quarto livello sono invece costituti da **memoria flash**. La memoria non volatile (**N**on**V**olatile **S**torage, **NVS**) è classificabile in due categorie:

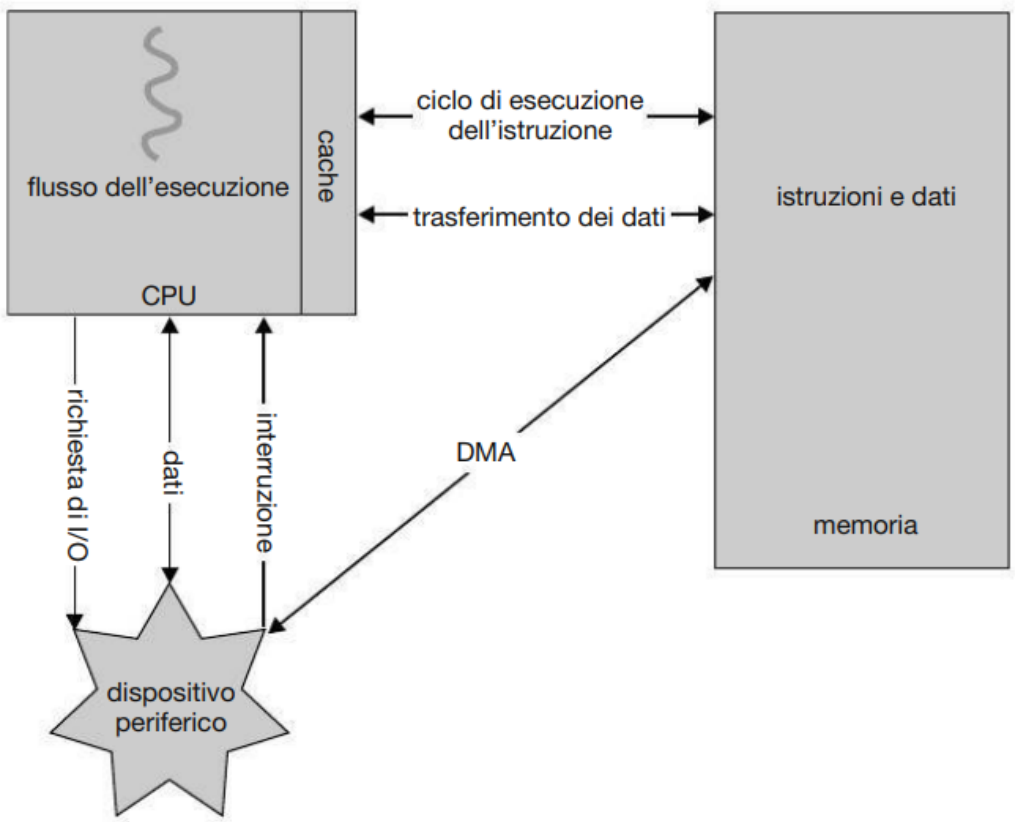
- **Meccanica**, come gli HDD.
- **Elettrica - NVM**, come gli SSD.

STRUTTURA DI I/O

Una percentuale cospicua del codice di un sistema operativo è dedicata alla gestione dell'I/O, a causa della sua importanza e per la moltitudine di tipologie di dispositivi che possono essere connessi a un calcolatore.

L'I/O guidato dalle interruzioni/interrupts è adatto al trasferimento di piccole quantità di dati ma non al trasferimento di massicce quantità di informazioni. Per risolvere questo problema si utilizza l'**accesso diretto alla memoria**

(**D**irect **M**emory **A**ccess, **DMA**). Una volta impostati i buffer, i puntatori e i contatori del dispositivo in memoria centrale, quindi dopo aver allocato una parte di memoria per il dispositivo, il controllore trasferisce interi blocchi di dati dal proprio buffer alla memoria centrale senza disturbare la CPU. Così, mentre il controllore gestisce il dispositivo associato effettuando varie operazioni, la CPU può occuparsi di altro.



GESTIONE DELL'I/O

Uno degli scopi di un sistema operativo è nascondere all'utente le peculiarità degli specifici dispositivi. In UNIX, per esempio, le caratteristiche dei dispositivi, per la maggior parte, sono nascoste dal **sottosistema di I/O**, che è composto da:

- Un componente per la gestione della memoria comprendente: i buffer di I/O, la cache e le aree di memoria per l'I/O asincrono (**spooling**).
- Un'interfaccia generale per i driver dei dispositivi.
- I driver per gli specifici dispositivi.

I dispositivi di I/O si dividono in:

- **Character Devices**, dispositivi in grado di comunicare solo un carattere alla volta;

- **Block Devices**, dispositivi in grado di comunicare interi blocchi di informazioni alla volta.

NOTA: Lo spooling I/O viene gestito tramite lo **spooler**. Lo spooler è un modulo del sistema operativo e viene descritto come un'interfaccia logica che gestisce il rapporto tra dispositivi hardware con diverse velocità di esecuzione/trasmissione.

SISTEMI MONOPROCESSORE

Diversi anni fa, la maggior parte dei sistemi erano **monoprocessore**, cioè utilizzavano un unico processore contenente una sola CPU che disponeva di un'unica **unità di elaborazione** o **core => Esegue le istruzioni e utilizza i registri per memorizzare i dati localmente**.

La singola CPU era in grado di svolgere un insieme di istruzioni di natura generale comprese quelle necessarie ai processi utenti.

I sistemi monoprocessore possiedono altri **processori specializzati** deputati a compiti particolari (es. i controllori). Questi processori sono dotati di un insieme di istruzioni riservate, non eseguono i processi utente e alleggeriscono il carico di operazioni che grava sulla CPU.

NOTA: I processori specializzati sono dispositivi di basso livello integrati nell'hardware e il SO non è in grado di comunicare, ma svolgono il loro compito in autonomia. Inoltre, l'integrazione di questi processori non trasforma il sistema monoprocessore in un sistema multiprocessore.

SISTEMI MULTIPROCESSORE

Oggi giorno, sono i sistemi **multiprocessore** a dominare il panorama computazionale. Tradizionalmente, un sistema multiprocessore dispone di due o più processori, ciascuno contenente una singola CPU **single-core**.

Il vantaggio principale dei sistemi multiprocessore è la maggiore capacità elaborativa (**throughput**).

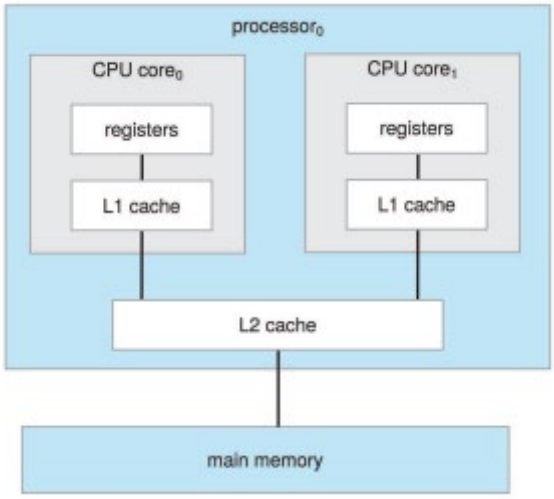
Disporre di N processori non significare incrementare di N volte il throughput, poiché avendo più processori che lavorano sullo stesso processo è necessario un'ulteriore operazione di controllo (**overhead**) per assicurarsi che il resto dei componenti funzioni correttamente.

In passato, era molto comune utilizzare la **multielaborazione asimmetrica** (**asymmetric multiprocessing, AMP**) in cui, supponendo che ci siano due processori nel sistema, un processore era abilitato esclusivamente all'esecuzione dei processi utente e l'altro all'esecuzione dei processi di sistema. Questo metodo, tuttavia, riduceva di molto le prestazioni che il

sistema poteva offrire. **1 processore dedicato alle attività del kernel ed N processori dedicati alle attività degli utenti**.

Oggi giorno, i sistemi più comuni utilizzano la **multielaborazione simmetrica** (**symmetric multiprocessing, SMP**) in cui ogni processore è abilitato all'esecuzione di tutte le operazioni di sistema, quindi processi utente e processi di sistema.

La definizione di multiprocessore si è evoluta e includono ora i **sistemi multicore**, in cui più core risiedono su un singolo chip. Ogni core dispone di una memoria cache L1 locale e di una memoria cache L2 condivisa con gli altri cores.



CLUSTER DI ELABORATORI

I **cluster di elaboratori** (**clustered systems**) o **cluster** sono un altro tipo di sistemi multiprocessore, basati sull'uso congiunto di più CPU ma composti da due o più elaboratori completi, chiamati **nodi**. I vari calcolatori che compongo il cluster sono collegati ed interagiscono tra di loro tramite una **rete LAN** o connessioni più veloci.

Ciascun nodo può tenere sotto controllo uno o più degli altri nodi; se il nodo controllato si guasta, uno dei suoi nodi controllori prende il sopravvento sulla sua memoria e riavvia le applicazioni che erano state interrotte.

I cluster di elaboratori sono sistemi **tolleranti ai guasti** (**fault-tolerant**) perché continuano a funzionare nonostante il guasto di qualsiasi singolo componente.

I cluster sono strutturati sia in maniera **simmetrica** che **asimmetrica**. Nei **cluster asimmetrici**, un calcolatore rimane nello stato di **attesa attiva** (**hot standby mode**) mentre l'altro esegue le applicazioni. Il calcolatore in hot standby monitora il server attivo e, se questo presenta un problema, il calcolatore in attesa diventa il server attivo. Nei **cluster simmetrici**, due o più

calcolatori eseguono le applicazioni e allo stesso tempo si monitorano a vicenda ottenendo così una maggiore efficienza poiché si utilizzano meglio le risorse.

I cluster, essendo formati da più elaboratori, possono essere usati per ottenere **ambienti di elaborazione ad alte prestazioni** (**high-performance computing**) che offrono una potenza di calcolo nettamente maggiore rispetto a sistemi monoprocessore o ai sistemi SMP poiché permettono l'esecuzione contemporanea di un'applicazione su tutti i computer del cluster. Tuttavia, per poter usufruire di questo vantaggio del cluster le applicazioni devono essere scritte specificatamente utilizzando la **parallelizzazione** (**parallelization**). Essa consiste nel suddividere l'applicazione in componenti separate che verranno eseguite sui singoli computer del cluster. Una volta che ogni computer ha finito di risolvere la sua parte di problema, i risultati parziali vengono combinati in un'unica soluzione finale.

Un altro vantaggio dei cluster è l'accesso simultaneo ai dati da parte di più calcolatori che viene gestita tramite la **gestione distribuita dei lock** (**distributed lock manager**), la quale evita sul nascere i conflitti tra le varie operazioni.

ATTIVITÁ DEL SISTEMA OPERATIVO

L'avvio e il riavvio del sistema richiedono la presenza di uno specifico programma detto **programma di avviamento** o **bootstrap program**. Normalmente, è memorizzato nel **firmware** (che fa parte dell'hardware) e la sua funzione consiste nell'inizializzare i vari componenti del sistema elaborativo caricando in memoria centrale il **kernel** (nucleo dell'S.O.).

Una volta caricato ed in esecuzione, il kernel inizia ad offrire al sistema e all'utente dei servizi che si distinguono in:

- **Servizi di sistema** o **demoni**, alcuni di questi servizi rimangono in esecuzione per tutta l'attività del kernel (es. systemd nel caso di UNIX e LINUX);
- **Servizi utente.**

Un evento è quasi sempre segnalato da un interrupt (**trap** ed **exception**), generato da un errore o dalla richiesta di una erogazione di un servizio. Quest'ultima avviene invocando le **chiamate di sistema** (**system call** o **syscall**).

MULTIPROGRAMMAZIONE E MULTITASKING

Fra le principali caratteristiche di un sistema operativo vi è la **multiprogrammazione**. Lo scopo della **multiprogrammazione** è quello di tenere costantemente occupata la CPU. In un sistema **multi-programmato** un programma in esecuzione viene chiamato **processo**.

L'idea su cui si fonda questa tecnica è la seguente: il sistema operativo tiene contemporaneamente in memoria centrale più processi. L'S.O. ne sceglie uno e inizia l'esecuzione; a un certo punto questo processo si ritroverà in attesa di un evento (es. completamento di un'operazione di I/O). In questi casi, in un sistema **NON** multi-programmato la CPU rimane inattiva. Nei sistemi multi-programmati invece, la CPU passa ad eseguire un altro processo e ripete continuamente questo ciclo finché non ci sono più processi da eseguire.

In parole povere, la CPU rimane attiva finché c'è almeno un processo in memoria centrale.

Il **multitasking** è un'estensione logica della multiprogrammazione; la CPU esegue più lavori contemporaneamente commutando le loro esecuzione con una frequenza tale da garantire a ciascun utente tempi di risposta rapidi. La coesistenza di più processi in memoria nello stesso lasso tempo richiede una qualche forma di gestione della memoria in modo da poter capire quale processo va eseguito per primo. Questa forma di gestione della memoria viene chiamata **scheduling della CPU.**

In un sistema multitasking l'SO deve garantire tempi di esecuzione accettabili. Un metodo comune è la **memoria virtuale**, che consente l'esecuzione di programmi anche non caricati completamente in memoria. Il vantaggio principale consiste nell'eseguire programmi di dimensioni maggiori della **memoria fisica.** Inoltre, astrae la **memoria principale** separando la **memoria logica**, vista dall'utente, dalla **memoria fisica.**

Gli svantaggi principali di un sistema multiprogrammato sono i seguenti:

- Un numero elevato di processi in esecuzione può causare un elevata sovraccarico nella macchina (**eccesso di context switching**);
- La condivisione dei dati è complicata e i processi possono comunicare solo mediante meccanismi appositi: **memoria condivisa**, **scambio di messaggi** e altre tecniche **IPC**.

DUPLICE MODALITÁ DI FUNZIONAMENTO - DUAL MODE

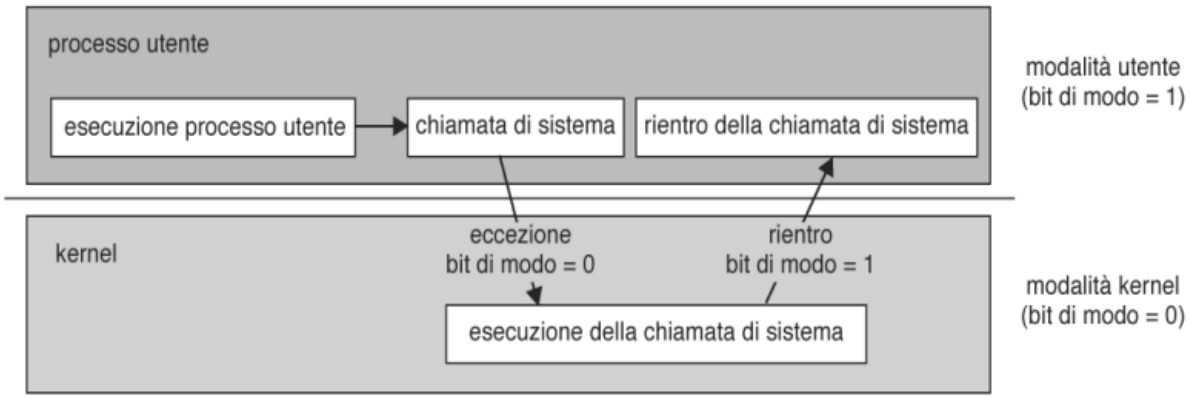
Dal momento che un sistema operativo e i suoi utenti condividono le risorse hardware e software del sistema, un buon S.O. deve evitare che un programma errato/dannoso comprometta il corretto funzionamento parziale o totale dell'ambiente. A tal fine, il sistema operativo deve essere in grado di

distinguere l'esecuzione di codice del sistema operativo e l'esecuzione di codice utente.

L'approccio più comunemente adottato è quello di fornire un supporto hardware che, tramite la modifica di un bit chiamato **bit di modalità**, permette di distinguere due modalità:

- **Modalità Utente** (**bit a 1**)
- **Modalità Kernel** (**bit a 0**)

Questa duplica modalità di funzionamento (**dual mode**) consente la protezione del sistema operativo e degli altri utenti dagli errori di un utente.



Una volta che la protezione hardware è attiva, gli errori di violazione della modalità sono rilevati dall'hardware stesso e vengono gestiti dal sistema operativo. Se un programma utente causa un errore, l'hardware genera un'eccezione (gestita tramite un interrupt), l'eccezione cede il controllo al sistema operativo che adotta una condotta consona all'interrupt ricevuto.

TIMER

Per far sì che il sistema operativo mantenga il controllo sulla CPU e dunque che un processo non restituisca più il controllo al sistema operativo viene usato un **timer**. Il timer è programmabile in modo che, a intervalli di tempo predefiniti (in genere 10ms o 20ms), invii un segnale di interruzione alla CPU che restituisce il controllo al sistema operativo, il quale può decidere se **terminare** il programma o **concedergli altro tempo**.

SICUREZZA E PROTEZIONE

Se diversi utenti usufruiscono dello stesso elaboratore che consente l'esecuzione concorrente di processi multipli, l'accesso ai dati dovrà essere disciplinato da regole, o meglio, da uno o più meccanismi di protezione (es. dual mode, timer ecc.).

Per **protezione**, si intende dunque un qualunque meccanismo di controllo dell'accesso alle risorse possedute da un elaboratore, da parte di utenti e/o processi.

Protezione e **sicurezza** presuppongo che un sistema operativo sia in grado di distinguere tra i propri utenti. Nella maggior parte degli S.O. è disponibile un elenco di nomi degli utenti e dei loro **identificatori utente** (**user ID**) o, nel caso specifico di Windows, **ID di sicurezza** (**SID**). Tramite questi ID, il sistema operativo è in grado di distinguere i suoi utenti accertandosi anche dei livelli di permessi posseduti da ognuno.

Gli identificatori sono associati a tutti i file, processi e thread legati al soggetto.

NOTA: Ogni utente ha un numero massimo di processi che può lanciare contemporaneamente (**N.MAX = 32.768 = 2^{16} in binario**).

NOTA: Se più utenti hanno gli stessi permessi, si può formare un **gruppo di utenti** al quale viene assegnato un **nome** e un **identificatore di gruppo** (**group ID**).

VIRTUALIZZAZIONE

La **virtualizzazione** è una tecnica che permette di **astrarre** l'hardware di un singolo elaboratore in diversi ambienti di esecuzione (suddivide la macchina principale in più sotto-macchine, chiamate **macchine virtuali**, **simulando** i diversi ambienti di elaborazione) a condizione che tali ambienti siano compatibili con l'hardware della macchina.

La virtualizzazione ricade nella stessa categoria di software dell'**emulazione**. L'emulazione, a differenza della virtualizzazione, permette di emulare/eseguire anche sistemi operativi ed applicazioni che non erano originariamente compatibili con l'hardware iniziale. Per esempio, quando l'Apple è passata dalle CPU IBM Power alle CPU Intel x86, ha fornito un emulatore chiamato "Rosetta" che permetteva l'uso di applicazioni compilate per le CPU IBM sulle nuove CPU Intel. L'emulazione, tuttavia, può risultare lenta nel caso in cui la CPU d'origine e quella di destinazione (quella che deve essere emulata) hanno prestazioni simili, poiché il codice emulato viene eseguito più lentamente rispetto a quello nativo.

Tornando alla virtualizzazione, la macchina che "virtualizza" viene chiamata **ospitante** (**host**) mentre la macchina "virtualizzata" viene chiamata **ospite** (**guest**)**.** La virtualizzazione viene gestita dal **gestore della macchina virtuale** (**Virtual Machine Manager, VMM**) o **hyper-visor**, il cui compito è quello di

fungere da “cuscino” tra l’host e i guests, amministrando e gestendo le risorse dei vari sistemi operativi ospitati e proteggendoli l’uno dall’altro.

SISTEMI DISTRIBUITI

Per sistema distribuito si intende un insieme di elaboratori fisicamente separati, in genere eterogenei, interconnessi da una rete per consentire agli utenti l’accesso alle varie risorse dei singoli sistemi.

Una **rete** si può considerare come un canale di comunicazione tra due o più sistemi. Le reti si distinguono per i protocolli usati, le distanze tra i nodi (i terminali che compongo la rete) e per il mezzo attraverso il quale avviene la comunicazione. Il più diffuso protocollo di comunicazione è il **TCP/IP**, il quale fornisce l’architettura fondamentale di Internet.

Nei sistemi distribuiti la stretta comunicazione che si instaura tra i diversi elaboratori dà l’impressione che vi sia un solo sistema operativo che gestisce la rete/sistema.

AMBIENTI DI ELABORAZIONE

I sistemi operativi vengono utilizzati in maniera differente in base all’ambiente di elaborazione.

ELABORAZIONE CLIENT-SERVER

Un’architettura di rete contemporanea realizza un sistema in cui alcuni server soddisfano le richieste dei **sistemi client**, cioè degli utenti. Questo sistema è una variante dei sistemi distribuiti e prende il nome di **sistema client-server** che può essere distinto in:

- **Server elaborativi**, forniscono un’interfaccia a cui i client possono inviare una richiesta (per es. di lettura dei dati); in risposta, il server esegue la richiesta e restituisce i risultati al client;
- **File server**, offrono un’interfaccia di file system che consente al client la creazione, lettura, modifica e cancellazione dei file sul server.

CLOUD COMPUTING

Il **cloud computing** è una tecnica, basata sulla virtualizzazione, che consente di fornire capacità elaborativa, storage ed applicazioni come servizi di rete.

Esistono diverse tipologie di cloud computing:

- **Cloud pubblico**, un cloud disponibile attraverso Internet a chiunque si abboni al servizio;
- **Cloud privato**, un cloud gestito da un’azienda per l’utilizzo interno;
- **Cloud ibrido**, un cloud che comprende componenti pubbliche e private;

- **SaaS (Software as a Service)**, una o più applicazioni fruibili via Internet;
- **PaaS (Platform as a Service)**, un ambiente software predisposto per usi applicativi via Internet (database server);
- **IaaS (Infrastructure as a Service)**, server o storage disponibili attraverso Internet.

NOTA: Una piattaforma cloud può offrire una combinazione di tipologie diverse di servizi, ad esempio un'azienda può offrire sia Saas sia Iaas come servizio di cloud.

SISTEMI EMBEDDED REAL-TIME

I **sistemi embedded** sono **sistemi elettronici di elaborazione** a **microprocessore** progettati appositamente per un determinato utilizzo (**special-purpose**), ovvero non riprogrammabili dall'utente per altri scopi, spesso con una **piattaforma hardware ad hoc** integrata nel sistema che controlla e gestisce tutte o parte delle funzionalità richieste.

I sistemi embedded funzionano quasi sempre come **sistemi embedded real-time**, ovvero sistemi in grado di svolgere operazioni e fornirne i risultati in tempo reale.

CAPITOLO 2

INTRODUZIONE: STRUTTURE DEI SISTEMI OPERATIVI

I sistemi operativi forniscono l'ambiente in cui si eseguono i programmi. Essendo organizzati secondo criteri la struttura interna che li caratterizza può variare molto. Un sistema operativo si può considerare da diverse angolazioni: **secondo l'interfaccia che esso fornisce agli utenti e ai programmatori, oppure secondo i suoi componenti e le relative interconnessioni**.

SERVIZI DI UN SISTEMA OPERATIVO

Un sistema operativo offre un ambiente in cui eseguire programmi e fornire servizi. I servizi specifici, variano in base al tipo di sistema operativo, ma è comunque possibile individuare una **classe di servizi comuni**. Questa classe di servizi si può dividere in due gruppi: il primo gruppo riguarda **gli utenti** del sistema operativo e comprende funzioni e servizi dedicati a rendere l'utilizzo del sistema più facile, il secondo gruppo riguarda invece il **funzionamento** e l'**ottimizzazione** del sistema stesso.

Il primo gruppo comprende i seguenti servizi relativi agli utenti:

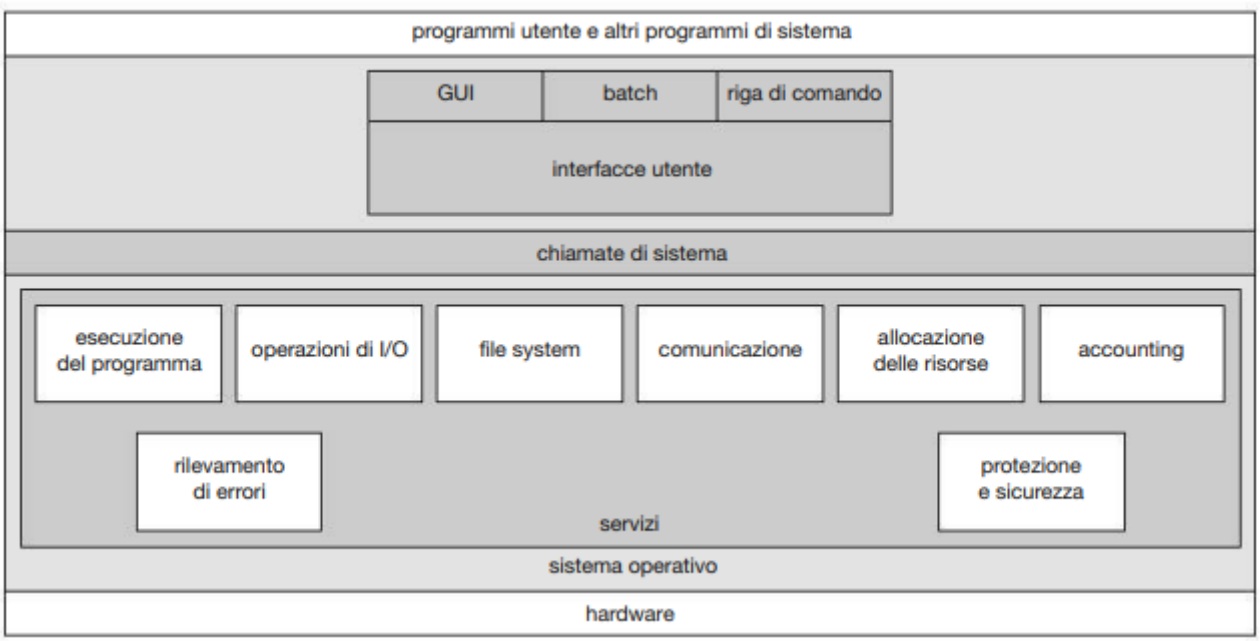
- **Interfaccia utente**, quasi tutti i sistemi operativi hanno un'interfaccia utente (**U**ser **I**nterface, **UI**). La UI può assumere diverse forme, ma la più

diffusa è un'**interfaccia utente grafica** (**G**rafic **U**ser **I**nterface, **GUI**) composta da finestre e dotata di un puntatore per effettuare le operazioni di I/O. Altre forme della UI sono il **touch-screen** e l'**interfaccia a riga di comando** (**C**ommand **L**ine **I**nterface, **CLI**);

- **Esecuzione di un programma**, un S.O. deve essere in grado di caricare in memoria un programma ed eseguirlo. Il programma deve terminare in modo normale o anomalo;
- **Operazioni di I/O**;
- **Gestione del file system**, il sistema operativo deve soddisfare la necessità degli utenti e dei vari programmi di poter leggere, creare, modificare e cancellare i file presenti sul sistema (rimanendo sempre nei limiti dei permessi posseduti dagli utenti e dei programmi).
- **Comunicazioni**, in molti casi un processo ha bisogno di comunicare con altri processi. La comunicazione può avvenire tramite **una rete**, nel caso dei cluster o dei sistemi distribuiti in cui i processi possono trovarsi in elaboratori diversi, per mezzo di una **memoria condivisa**, che permette a più processi di leggere e scrivere in una data porzione di memoria che condividono, o tramite lo **scambio di messaggi**, in questo caso il sistema operativo trasferisce pacchetti di informazioni tra i processi;
- **Rilevamento di errori**, il sistema operativo deve essere in grado di rilevare e gestire opportunamente eventuali errori che possono verificarsi sulle risorse hardware o software del sistema.

Il secondo gruppo comprende i seguenti servizi relativi al sistema:

- **Allocazione delle risorse**, se sono attivi più utenti o processi, il sistema operativo deve gestire ed allocare le risorse facendo in modo che ogni utente/processo abbia a disposizione ciò che gli serve;
- **Logging**, si tratta di un tracciato record dei programmi che sono stati eseguiti sul calcolatore e di quante e quali risorse ogni programma ha usato;
- **Protezione e sicurezza**, il sistema operativo deve essere in grado di distinguere i suoi utenti, bloccare gli accessi illegali/non identificati alle risorse del sistema elaborativo, deve garantire che l'accesso alle risorse sia controllato e sicuro, ovvero che più utenti e/o processi non interferiscano gli uni con gli altri.



CHIAMATE DI SISTEMA

Le **chiamate di sistema** o **system call** costituiscono un'interfaccia per i servizi erogati dal sistema operativo. Di norma, sono routine programmate in C/C++ o, per operazioni di basso livello, in linguaggio assembly.

Ogni chiamata di sistema è **identificata da un numero**.

INTERFACCIA PER LA PROGRAMMAZIONE DI APPLICAZIONI (API)

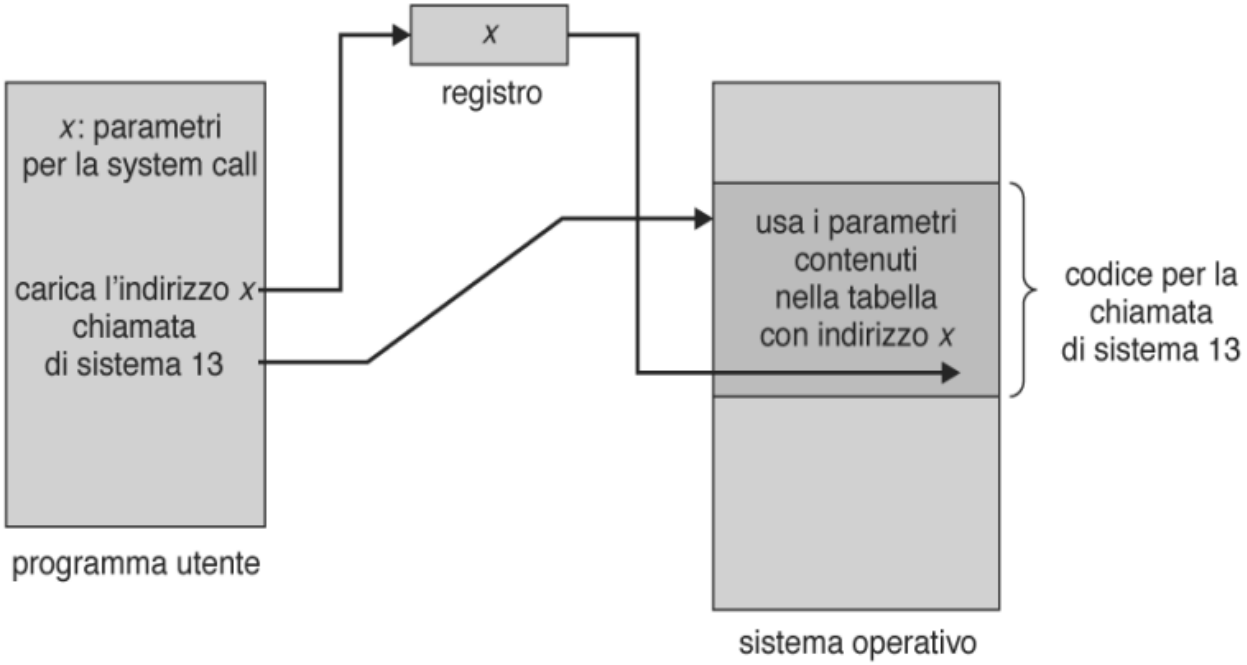
Un'**interfaccia per la programmazione di applicazioni** (**A**pplication **P**rogramming **I**nterface, **API**) è un insieme di funzioni messe a disposizione del programmatore e dettaglia i parametri necessari all'invocazione di queste funzioni, oltre ad indicare i valori restituiti.

Le API vengono privilegiate rispetto all'uso diretto delle chiamate di sistema poiché: risparmiamo al programmatore l'onere di invocare direttamente le chiamate di sistema, che spesso si rivelano molto più dettagliate e complesse delle funzioni presenti nell'API, siccome ogni chiamata a funzione di un API invoca a sua volta una o più chiamate di sistema; un'applicazione programmata utilizzando una specifica API sarà molto probabilmente compatibile su un altro ambiente che implementa la stessa API senza dover far operazioni di conversione o adattamento (**portabilità**).

Un altro fattore importante nella gestione delle chiamate di sistema è l'**ambiente di esecuzione al run-time** (**R**un-**T**ime **E**nvironment, **RTE**) composto da tutti i programmi necessari per l'esecuzione delle applicazioni (compilatori/interpreti, librerie, loaders ecc.). L'RTE fornisce un'**interfaccia alle**

chiamate di sistema (**system call interface**) che intercetta le chiamate a funzioni nell'API e invoca le relative system call. La system call interface mantiene una tabella contenente tutte le system call e invoca di volta in volta la system call richiesta, che risiede nel kernel, restituendo al chiamante lo stato della chiamata.

Per passare i parametri alle chiamate vengono utilizzati due metodi: i parametri vengono **salvati nei registri** oppure, se il numero di parametri è maggiore del numero dei registri, vengono **salvati in un blocco di memoria** e l'**indirizzo** di questo blocco **viene passato in un registro**.



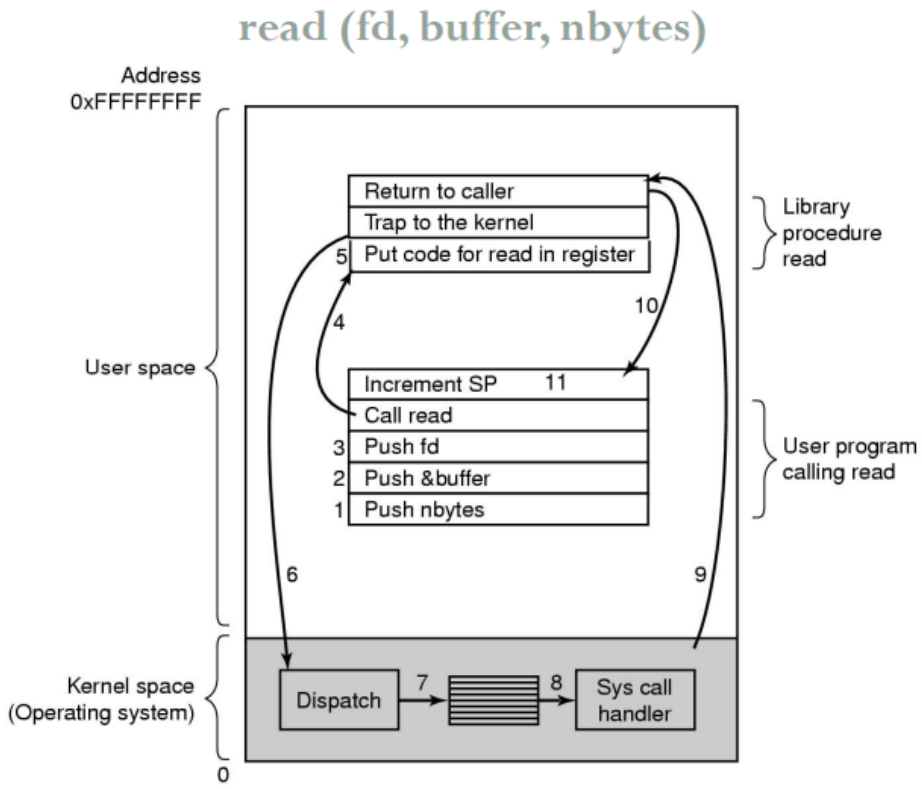
CATEGORIE DI CHIAMATE DI SISTEMA

Le chiamate di sistema sono classificabili in **sei categorie principali**:

- **Controllo dei processi;**
- **Gestione dei file;**
- **Gestione dei dispositivi;**
- **Gestione delle informazioni;**
- **Comunicazione;**
- **Protezione.**

- Controllo dei processi
  - terminazione normale e anormale
  - caricamento, esecuzione
  - creazione e arresto di un processo
  - esame e impostazione degli attributi di un processo
  - attesa per il tempo indicato
  - attesa e segnalazione di un evento
  - assegnazione e rilascio di memoria
- Gestione dei file
  - creazione e cancellazione di file
  - apertura, chiusura
  - lettura, scrittura, posizionamento
  - esame e impostazione degli attributi di un file
- Gestione dei dispositivi
  - richiesta e rilascio di un dispositivo
  - lettura, scrittura, posizionamento
  - esame e impostazione degli attributi di un dispositivo
  - inserimento logico ed esclusione logica di un dispositivo
- Gestione delle informazioni
  - esame e impostazione dell'ora e della data
  - esame e impostazione dei dati del sistema
  - esame e impostazione degli attributi dei processi, file e dispositivi
- Comunicazione
  - creazione e chiusura di una connessione
  - invio e ricezione di messaggi
  - informazioni sullo stato di un trasferimento
  - inserimento ed esclusione di dispositivi remoti

ESEMPIO DI CHIAMATA DI SISTEMA



1-3. Salvataggio dei parametri nello stack;

4. Chiamata alla syscall **read**;

5. Caricamento del codice della syscall in un registro fissato RX;

6. Esecuzione TRAP, passaggio alla kernel mode, salto del codice del dispatcher;

7-8. Selezione della syscall secondo il codice in RX ed invocazione del gestore appropriato;

9. Ritorno del controllo al chiamante e ripristino della user mode;

10-11. Incremento dello Stack Pointer (SP) per rimuovere i parametri della chiamata a read.

CONTROLLO DEI PROCESSI – SYSTEM CALLS IN GRASSETTO ()

Un programma in esecuzione deve potersi fermare in modo sia normale (**end()**) sia anomalo (**abort()**). Talvolta, se si ricorre a una chiamata di sistema per terminare in modo anomalo un programma, oppure se il programma in esecuzione incontra un problema e segnala l'errore tramite un'eccezione, un'immagine del contenuto della memoria viene salvato in un file **dump** e viene generato un messaggio di errore. Ogni sistema operativo può permettere

azioni specifiche per la gestione degli errori ma in genere, quando viene generata un'eccezione di questo tipo, il controllo viene passato all'interprete dei comandi che legge semplicemente l'istruzione successiva da eseguire.

Talvolta, un processo che esegue un programma può richiedere di caricare (**load()**) ed eseguire (**execute()**) un altro programma. In questo caso, il processo al quale restituire il controllo dipende dallo stato del programma attuale. Se al termine del nuovo processo, il controllo rientra nel processo originale, si deve salvare lo stato del processo attuale creando così un meccanismo con cui un programma può richiamare un altro programma. Se entrambi i processi continuano l'esecuzione in contemporanea, si è creato un nuovo processo da eseguire in multiprogrammazione (**create_process()**). Quando si crea un nuovo processo, o anche un insieme di processi, è necessario mantenerne il controllo; ciò richiede la capacità di determinare e reimpostare gli attributi di un processo, compresi la sua priorità, il suo tempo massimo d'esecuzione e così via (**get_process_attributes()** e **set_process_attributes()**). Inoltre, può essere necessario terminare un processo creato, se si riscontra che non è corretto o se la sua esecuzione non è più utile (**terminate_process()**). Una volta creati, può essere necessario attendere che i processi terminino la loro esecuzione. Quest'attesa si può impostare per un certo periodo di tempo (**wait_time()**), ma è più probabile che si preferisca attendere che si verifichi un dato evento (**wait_event()**). In tal caso i processi devono segnalare il verificarsi di quell'evento (**signal_event()**). Molto spesso due o più processi possono condividere dati. Per assicurare l'integrità dei dati che vengono condivisi, spesso i sistemi operativi forniscono chiamate di sistema che consentono a un processo di bloccare (**lock**) dati condivisi, in modo che nessun altro processo possa accedere ai dati fino al rilascio del blocco. In genere tali chiamate di sistema includono **acquire_lock()** e **release_lock()**.

LINKER E LOADER

Generalmente un programma risiede su disco sottoforma di file binario e per poter essere eseguito, deve essere caricato in memoria e inserito nel **contesto** di un processo.

Di seguito sono riportati i passaggi di questa procedura.

I file sorgenti (.c, .java, .py ecc.) vengono compilati in **file oggetto** noti come **file oggetto rilocabili** perché possono essere caricati in una qualsiasi posizione della memoria fisica. In seguito, il **linker** combina i file oggetti in unico file binario; in questa fase detta **collegamento** (**linking**) possono essere inclusi altri file oggetto o alcune librerie necessarie per l'esecuzione del programma.

Per caricare il file in memoria, viene usato un **loader**. Un'attività associata al collegamento e al caricamento (**loading**) è la **rilocazione** (**relocation**) che assegna gli indirizzi definitivi alle componenti del programma e le riaggiusta in base a questi indirizzi.

Per richiamare il loader, basta immettere il nome del programma su riga di comando; quando ciò accade, la shell crea un nuovo processo per eseguire il programma, tramite **fork()**, e poi richiama il loader, tramite **exec()**. Un processo simile avviene quando in un ambiente dotato di GUI si esegue doppio click sull'icona di un programma.

La maggior parte dei sistemi operativi consente a un programma di caricare e collegare dinamicamente le librerie necessarie al momento dell'esecuzione in modo da non caricare in memoria librerie che potrebbero essere inutilizzate (es. Windows con le **DLL**).

I file oggetto e gli eseguibili contengono **formati standard**, il codice macchina e metadati relativi ai simboli e alle funzioni usate nel programma. Nei sistemi UNIX e Linux il formato standard è l'**ELF** (**Executable and Linkable Format**) nel quale è contenuto l'**entry point** del programma, ovvero l'indirizzo della prima istruzione da eseguire. I sistemi Windows usano il format **PE** (**Portable Executable**) mentre macOs usa il formato **Mach-O**.

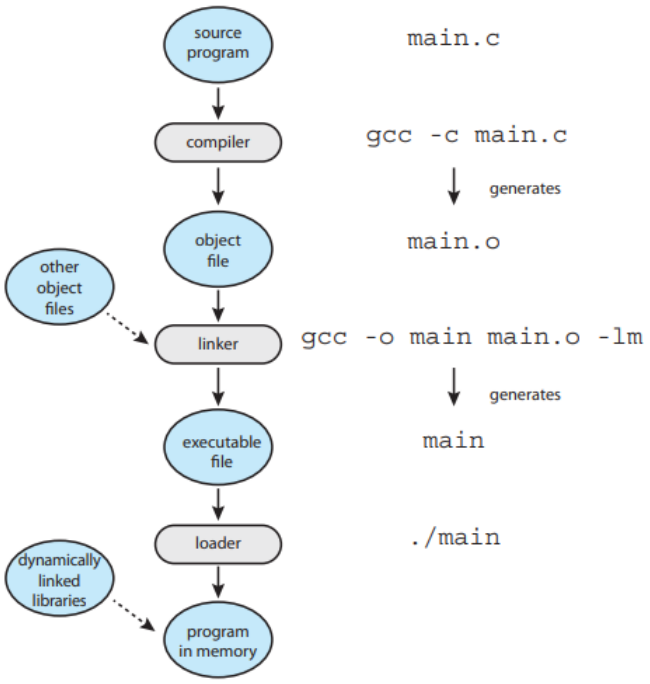


Figure 2.11 The role of the linker and loader.

STRUTTURA DEL SISTEMA OPERATIVO

Nonostante un sistema operativo sia un ambiente vasto e complesso, esso deve comunque poter essere modificato e per questo va progettato con

estrema attenzione. Uno dei metodi più diffusi è quello di suddividere il sistema operativo in **blocchi** o **moduli** (es. programmazione modulare); ciascun modulo definisce una parte specifica del sistema operativo, con interfacce e funzioni definite con precisione.

STRUTTURA MONOLITICA

La struttura più semplice per l'organizzazione di un sistema operativo è l'**assenza di struttura**: tutte le funzionalità del kernel vengono inserite in un **unico file binario statico** che viene eseguito in un **unico spazio di indirizzamento**. Questo approccio è noto come **struttura monolitica**.

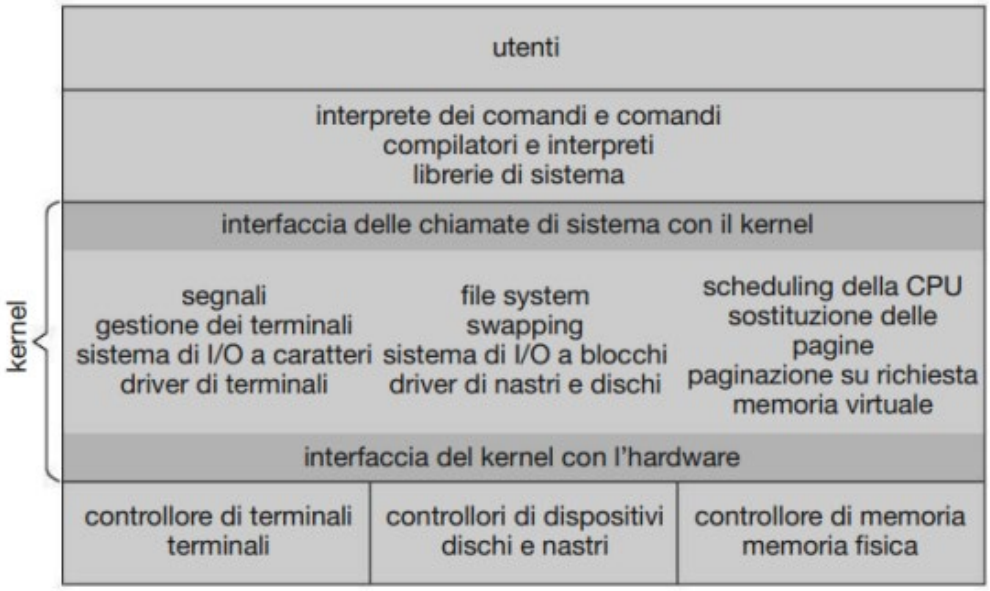
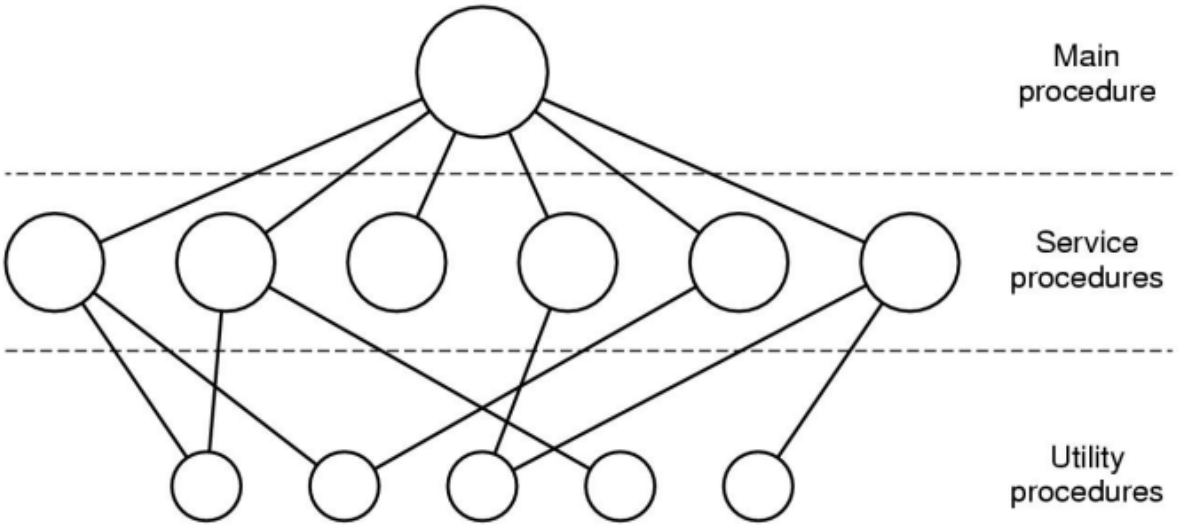


Figura 2.12 Struttura del sistema UNIX.

Le procedure di servizio vengono compilate come un unico file oggetto ed ognuna di esse può invocare le altre. Ogni processo viene eseguito parzialmente in modalità kernel.



Gli svantaggi principali di questa struttura sono:

- **Manutenzione complessa**;
- **Vulnerabilità agli errori di sistema e agli attacchi**;
- **Blocco del sistema dovuto al malfunzionamento di una procedura**;
- **La protezione fornita dal sistema non è sufficiente**.

I kernel monolitici, rispetto alle altre strutture, hanno un netto vantaggio in termini di prestazioni siccome, poiché il sistema operativo si trova in un unico file, la **comunicazione è più veloce** e **l'overhead è ridotto** avendo un miglioramento in termini di velocità ed efficienza.

STRUTTURA STRATIFICATA

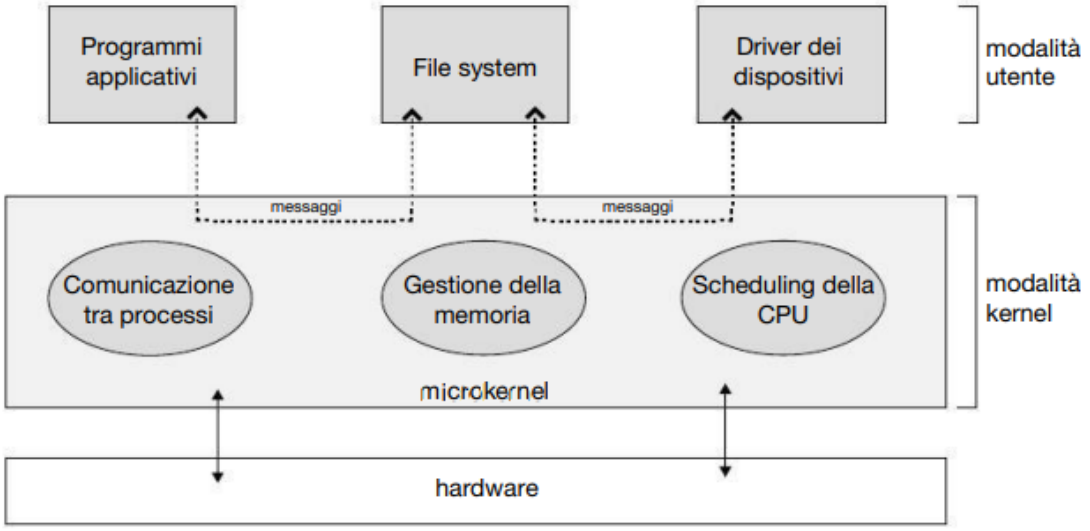
Un sistema monolitico viene anche detto **sistema strettamente accoppiato** (**tightly coupled**) perché le modifiche a una parte del sistema possono influire sulle rimanenti parti. In alternativa, è possibile progettare un **sistema debolmente accoppiato** (**loosely coupled**), in cui il sistema viene diviso in più componenti con funzionalità specifiche e limitate; in questo modo una modifica a una parte del sistema si rifletterà solo ed unicamente su quella parte.

Uno degli approcci più usati per progettare un sistema di questo tipo, detto anche **modulare**, è l'**approccio stratificato** secondo il quale il sistema è diviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (**strato 0**) e il più alto alla UI (**strato N**). Ogni strato viene progettato in modo che questo possa utilizzare funzioni e strutture dati appartenenti **solo** ai **livelli inferiori**. Lo svantaggio principale dell'approccio stratificato è l'overhead elevato dovuto al fatto che una richiesta per poter essere eseguita deve attraversare più strati.

MICROKERNEL

Il sistema UNIX originale aveva una struttura monolitica. Man mano che UNIX veniva esteso, il kernel è cresciuto notevolmente ed è diventato difficile da gestire. Verso la metà degli anni ’80 un gruppo di ricercatori della Carnegie Mellon University realizzarono un sistema operativo, **Mach**, col kernel strutturato in **moduli** secondo l’orientamento a **microkernel**. Secondo quest’orientamento il sistema operativo viene progettato rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema raggruppati in moduli.

Un microkernel offre i servizi minimi di: **gestione dei processi**, **della memoria** e **di comunicazione**.



Lo scopo principale del microkernel è quello di fornire funzioni di comunicazione tra i processi client e i vari servizi. Solitamente, la comunicazione avviene tramite uno scambio di messaggi.

Uno dei vantaggi principali del microkernel è la facilità di estensione del sistema operativo; i nuovi servizi vengono aggiunti ai programmi utente e non comportano modifiche al kernel. Offre maggiori garanzie di sicurezza e affidabilità, poiché la maggior parte dei servizi sono eseguiti come processi utenti e non come processi di sistema; dunque, se un servizio dovesse essere compromesso, il resto del sistema rimarrebbe intatto.

Lo svantaggio principale dei microkernel sono le scarse prestazioni dovute all’elevato overhead.

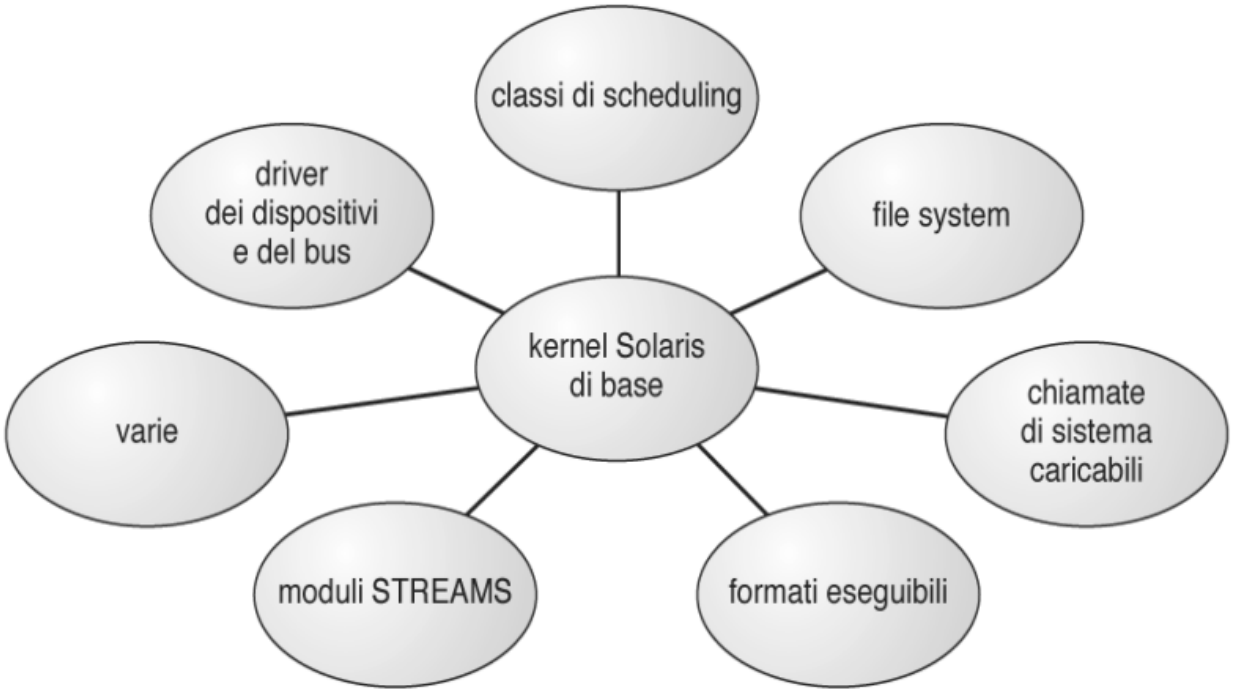
STRUTTURA MODULARE

Un altro metodo per la progettazione dei sistemi operativi si basa sull’utilizzo di **moduli del kernel caricabili dinamicamente**. Il kernel è costituito da un

insieme di componenti di base dediti all'erogazione dei servizi principali, integrati da funzionalità aggiunte dinamicamente durante l'avvio o l'esecuzione per mezzo dei moduli.

Il risultato complessivo ricorda una struttura stratificata data la suddivisione in moduli ma, ma è più flessibile ed efficiente di un sistema stratificato in quanto un modulo può chiamare e comunicare con qualsiasi altro modulo.

Linux, ad esempio, utilizza **moduli del kernel caricabili** (**L**oadable **K**ernel **M**odule, **LKM**) per supportare i driver dei dispositivi e i file system. I moduli possono essere caricati e rimossi dinamicamente.



SISTEMI IBRIDI

Nella pratica pochi sistemi operativi adottano un'unica struttura ben definita. I sistemi operativi, solitamente, tendono a combinare strutture diverse che portano a **sistemi ibridi** indirizzati alle prestazioni, all'usabilità e alla sicurezza.

Ad esempio, Linux è monolitico, perché avere tutto il kernel in unico spazio di indirizzamento fisico garantisce prestazioni migliori, ma è anche modulare siccome nuove funzioni possono essere aggiunte dinamicamente al kernel.

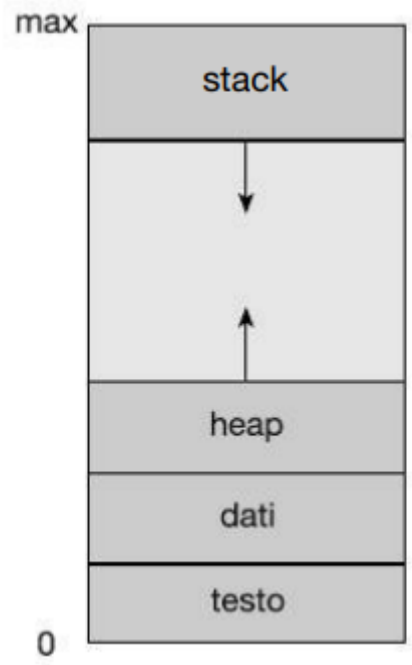
CAPITOLO 3

IL PROCESSO

Informalmente, un processo è un programma in esecuzione. Lo **stato di attività** di un processo è rappresentato dal **valore del contatore di programma** e dal **contenuto dei registri del processore**.

La struttura di un processo in memoria è suddivisa in più sezioni:

- **Sezione di testo**: contiene il codice eseguibile;
- **Sezione dati**: contiene le variabili globali;
- **Heap**: memoria allocata dinamicamente durante l'esecuzione del programma;
- **Stack**: memoria temporaneamente utilizzata durante le chiamate di funzioni.



Le dimensioni delle sezioni di testo e dati sono **fisse**, mentre l'heap e lo stack sono **dinamiche**. L'heap cresce ogni volta che viene allocata memoria dinamicamente e decresce quando questa viene restituita. Ogni qualvolta che si verifica una chiamata a funzione viene allocata memoria per un **record di attivazione** (**activation record**) che contiene i parametri, variabili locali e l'indirizzo di ritorno che viene inserito nello stack; quando il controllo viene restituito al chiamante, il record viene rimosso. Siccome l'heap e lo stack crescono l'uno verso l'altro è **compito** del sistema operativo fare in modo che **non si sovrappongano**.

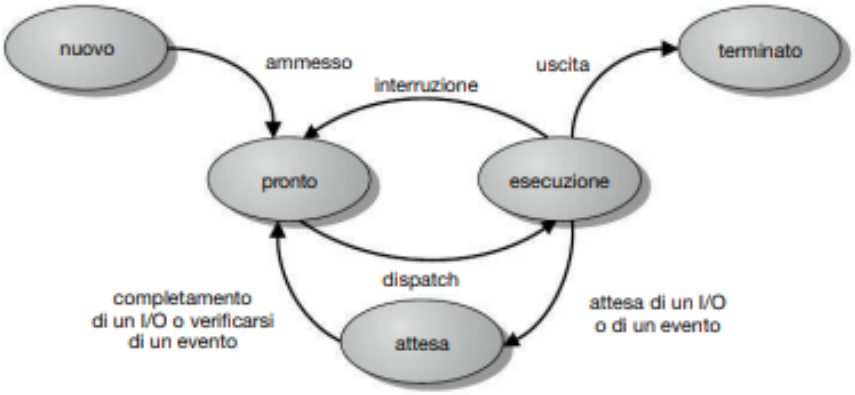
NOTA: Un programma di per sé non è un processo ma un'**entità passiva** che risiede sulla memoria secondaria in attesa di essere eseguita; mentre un processo è un'**entità attiva**, caricata in memoria principale, con un **program counter** contenente la prossima istruzione da eseguire. Un programma può generare numerosi processi, l'uno distinto dagli altri, e un processo, a sua volta, può generare più processi.

STATO DEL PROCESSO

Un processo, durante l'esecuzione, è soggetto a cambiamenti del suo **stato**.

Un processo può trovarsi in uno dei seguenti stati:

- **New – Nuovo**, creazione del processo;
- **Esecuzione – Running**, esecuzione del processo;
- **Attesa – Waiting**, il processo rimane in attesa di qualche evento (es. operazioni di I/O);
- **Pronto – Ready**, il processo è stato caricato in memoria ed è in attesa di essere assegna a un'unità di elaborazione;
- **Terminato – Terminated**, il processo è terminato.



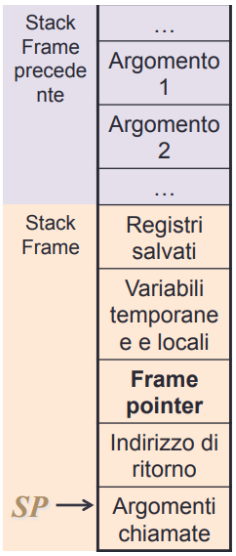
Gli stati sopra riportati sono gli **stati principali** che si usano in ogni S.O. ma, ovviamente, ogni S.O. può introdurre ulteriori distinzioni tra gli stati principali.

BLOCCO DI CONTROLLO DEL PROCESSO

Ogni processo è rappresentato da un **blocco di controllo** (**process control block, PCB,** o **task control block, TCB**) che viene usato come deposito per tutte le informazioni relative al processo a cui si riferisce.

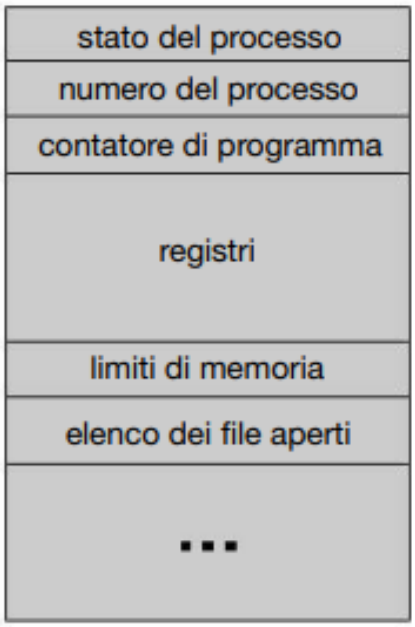
Le informazioni contenute da un **PCB/TCB** sono:

- **Stato del processo**;
- **Contatore di programma** – **Program Counter**, contiene l'indirizzo della prossima istruzione da eseguire;



- **Registri della CPU**. I registri variano in numero e tipo in base all'architettura della CPU. Essi comprendono accumulatori, registri indice, puntatori alla cima dello stack (**stack pointer**), registri d'uso generale e registri contenenti i codici di condizione (**condizioni codes**). Quando si verifica un'interruzione, questi registri devono essere salvati in modo da poter **riprendere** l'esecuzione;
- **Informazioni sullo scheduling di CPU,** comprendono la priorità del processo ed altri parametri dello scheduling;
- **Informazioni sulla gestione della memoria**, contengono varie informazioni, tra cui i registri base, i registri di limiti, le tabelle di segmentazione ed altro;

- **Informazioni di accounting**, rappresentato un report delle risorse utilizzate dal processo;
- **Informazioni sullo stato dell'I/O**, comprendono la lista dei dispositivi di I/O, dei file ecc. assegnati al processo.



THREAD

In passato, un processo non poteva svolgere più di una funzione alla volta; oggi giorno, la maggior parte dei sistemi operativi permettono a un processo di svolgere più compiti alla volta avendo più processi in esecuzione. Questa funzione viene usata in particolar modo nei sistemi **multicore**, in cui più **thread** possono essere eseguiti in parallelo. Un **thread** è un **percorso di esecuzione** che un processo segue. In un sistema che supporta i thread, il PCB viene esteso per raccogliere informazioni sui vari thread.

SCHEDULING DEI PROCESSI

L'obiettivo della **multiprogrammazione** consiste nel mantenere sempre attiva la CPU. Lo scopo dei sistemi **time-sharing** è quello di commutare l'uso della CPU tra i vari processi in modo che ogni utente sia in grado di interagire con ogni programma in esecuzione. A tali fini, lo **scheduler dei processi** seleziona un processo caricato in memoria e lo esegue. Il numero di processi eseguibili contemporaneamente in un dato momento dipende dal numero di core della CPU; **ogni core non può eseguire più di un processo per volta**. Se vi sono più processi che core, i processi in più rimarranno in attesa che uno o più core si liberino. Il numero di processi in memoria di un dato istante è detto **grado di multiprogrammazione**.

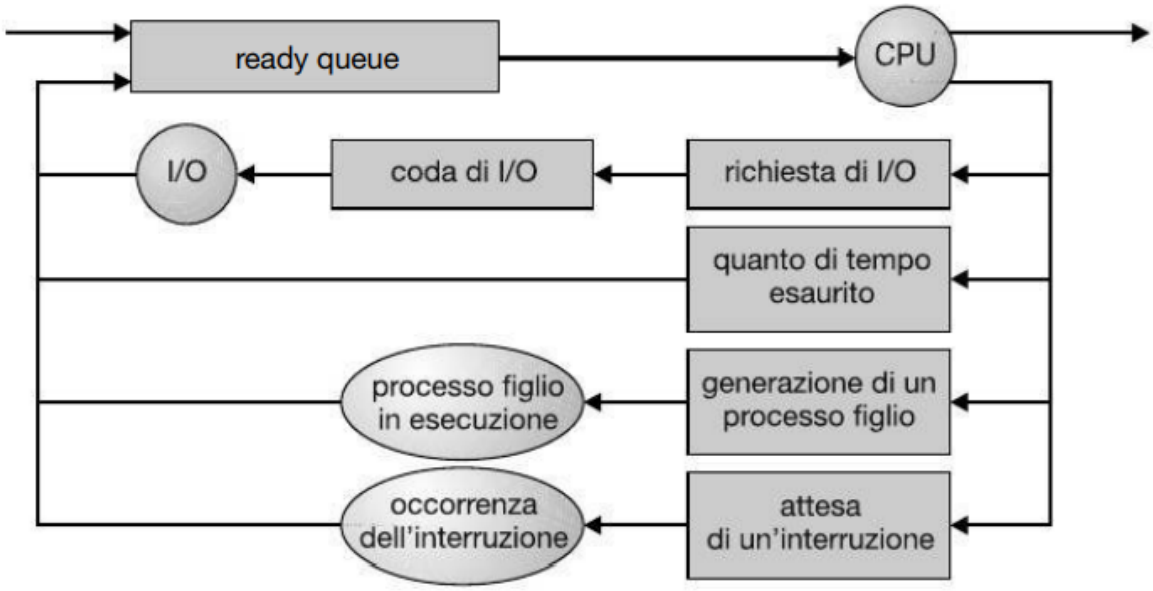
I processi possono essere:

- **I/O bound**, i processi di questo tipo passano la maggior parte del tempo a gestire operazioni di I/O;
- **CPU bound**, i processi di questo tipo impiegano la maggior parte del tempo nelle elaborazioni.

CODE DI SCHEDULING

Entrando nel sistema, ogni processo è inserito in una **coda di processi pronti per essere eseguiti** detta **coda dei processi pronti** (**ready queue**) che viene gestita come una lista concatenata.

Durante l'esecuzione, in un certo momento, un processo termina, viene interrotto o rimane **in attesa** di un certo **evento** e, in quest'ultimo caso, il processo viene inserito in una **coda di attesa** (**wait queue**).



Quando un processo in esecuzione rimane in attesa di un evento, questo viene inserito nella wait queue fino al completamento di tale evento; una volta che l'evento viene completato, il processo viene reinserito nella ready queue. **Ogni processo ripete questo ciclo finché non termina. Al termine del processo, esso viene rimosso da tutte le code, il suo PCB viene deallocato e le risorse da esso occupate vengono liberate**.

NOTA: Quando un processo viene selezionato per essere eseguito, si dice che il processo viene **spedito** (**dispatched**).

SCHEDULING DELLA CPU

Nel corso della sua vita, un processo si sposta ripetutamente tra la ready queue e diverse wait queues. Il sistema operativo, incaricato di selezionare i processi da queste queues, compie questa selezione tramite lo **scheduler**.

Lo **spooling** consiste nel caricare e tenere sul disco i programmi in più (più programmi di quanti ne possano essere eseguiti) presenti in memoria fino al momento della loro esecuzione. Tipico dei sistemi **batch**.

In un sistema operativo distinguiamo principalmente due tipi di scheduler:

- **Scheduler a lungo termine** (**job scheduler**), si occupa della gestione delle **code batch** e di selezionare i processi da caricare in memoria e dunque controlla il **grado di multiprogrammazione** in un sistema multiprogrammato;

- **Scheduler a breve termine o della CPU**, si occupa di selezionare i processi presenti in memoria nella ready queue ed assegnare loro un core per eseguirli.

La differenza sostanziale tra questi due "moduli" è la **frequenza** con la quale vengono eseguiti. Lo scheduler a breve termine viene eseguiti molto frequentemente in intervalli di tempo brevissimi (si parla di ms); mentre lo scheduler a lungo termine viene eseguito con una frequenza molto minore e spesso capita che passino minuti prima che sia eseguito nuovamente siccome viene invocato solo quando un processo abbandona il sistema. È molto importante che il job scheduler selezioni una buona combinazione di processi I/O bound e CPU bound. I sistemi **time-sharing**, come UNIX e LINUX, sono spesso privi di scheduler a lungo termine.

Lo **scheduler della CPU** deve anche interrompere forzatamente processi in esecuzione e schedularne altri.

Alcuni sistemi operativi hanno una forma **intermedia** di scheduling o **scheduler a medio termine**, nota come **swapping**. Lo **swapping** o **avvicendamento dei processi in memoria** consiste nel rimuovere un processo dalla memoria e collocarlo sul disco (**swapping out**) dove viene salvato il suo stato corrente e, successivamente, ricaricarlo in memoria (**swapping in**) ripristinando il suo stato. Lo swapping viene usato **solo** quando **la memoria è sovrautilizzata**.

CAMBIO DI CONTESTO

Come spiegato in precedenza, le interruzioni forzano il sistema a sospendere lo stato corrente della CPU per eseguire **routine** del kernel. Quando ciò avviene, il sistema deve salvare il **contesto** del processo corrente per poi poterlo riprendere in seguito. Il contesto di un processo è composto da: **lo stato del processo**, **i registri usati dalla CPU**, **informazioni sulla gestione della memoria** e **la modalità in cui la CPU viene eseguita** (**utente** o **kernel**)**.** Il contesto è salvato nel PCB del processo.

Il passaggio della CPU a un nuovo processo, che richiede il salvataggio del contesto processo corrente e il ripristino del nuovo, viene detto **cambio di contesto** (**context switch**) che consiste in un'operazione molto onerosa i cui tempi variano da sistema operativo a sistema operativo e dalle componenti hardware del calcolatore.

Quando si effettua un cambio di contesto, la CPU carica nel PCB del processo uscente il suo contesto e carica quello del processo subentrante.

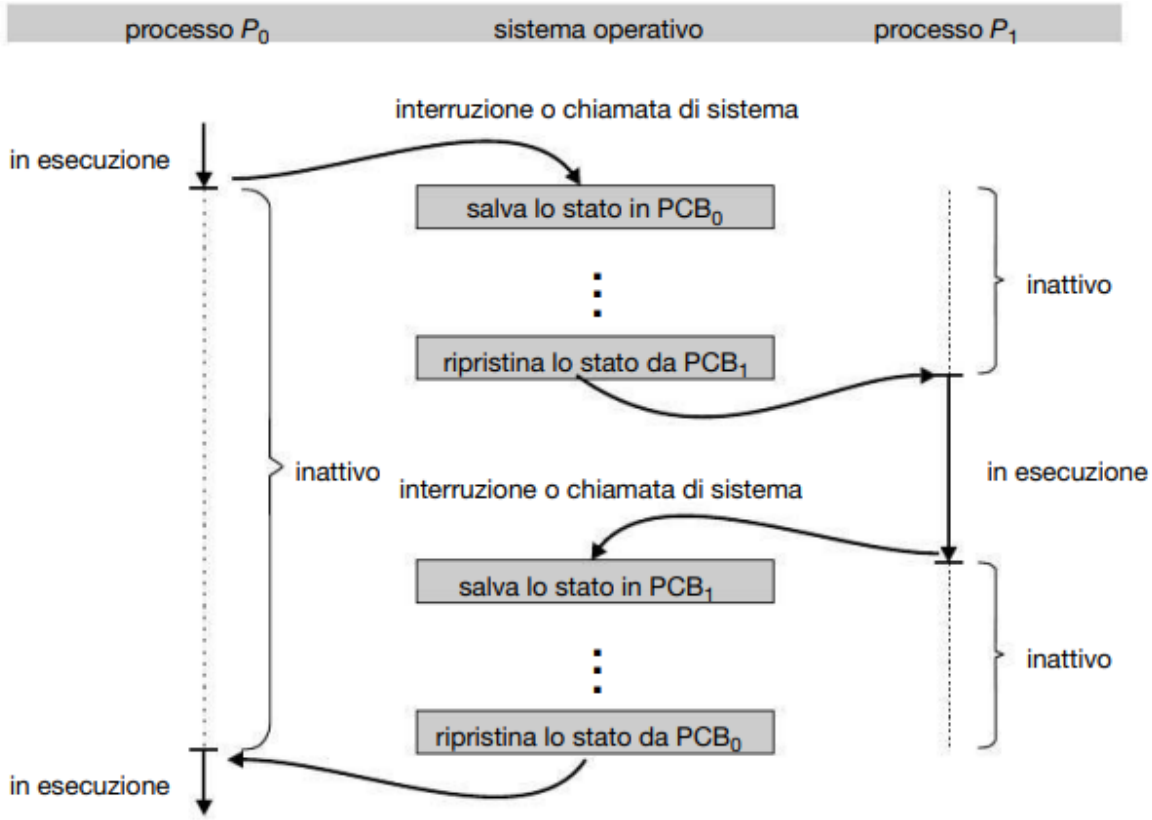
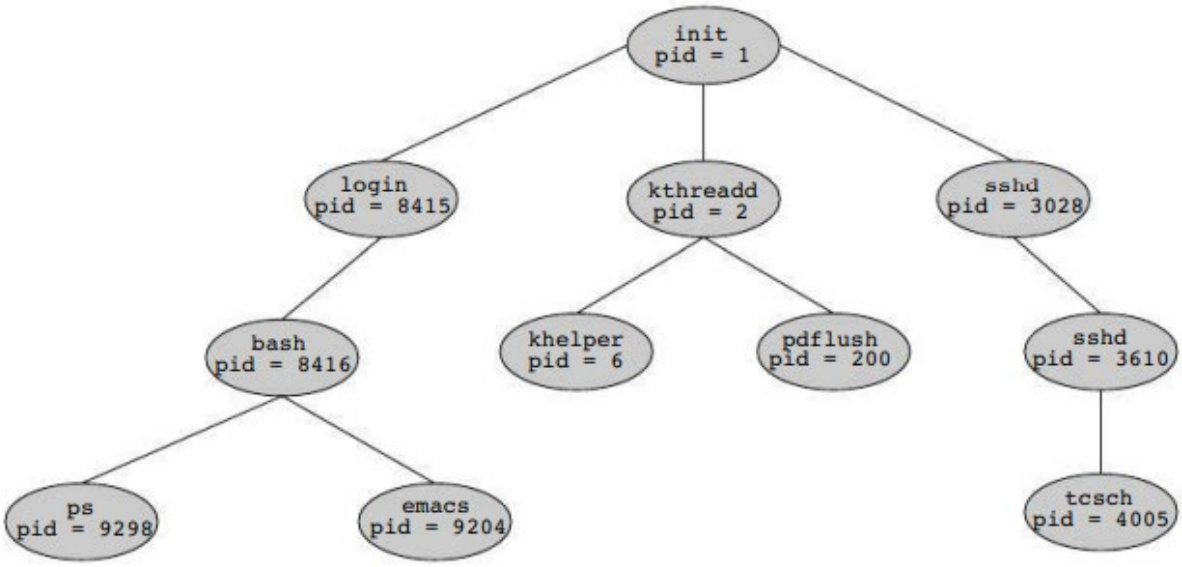


Figura 3.4 La CPU può essere commutata tra i processi.

CREAZIONE DI UN PROCESSO

Durante la propria esecuzione, un processo può generare altri processi. Il processo creante viene chiamato **genitore** o **padre** e il processo generato viene chiamato **figlio**. Ciascuno dei processi generati può generare altri processi, formando un **albero** di processi.



Ogni processo, solitamente, viene individuato da un numero univoco detto **identificatore del processo** o **pid** (**process identifier**).

Quando un processo crea un processo figlio, quest'ultimo avrà bisogno di risorse. Un processo figlio può ottenere le risorse di cui necessita direttamente dal sistema operativo oppure gli può venire assegnato un **sottoinsieme delle risorse assegnate al padre**, in questa maniera si evita che il processo figlio **sovraccarichi il sistema**. Il processo genitore può condividere alcune delle sue risorse tra più figli. Oltre alle risorse logiche e fisiche, un processo figlio "eredita" dal padre anche i dati di inizializzazione.

Quando un processo ne crea uno nuovo, vi sono due scenari possibili:

- **Il processo padre continua la sua esecuzione in contemporanea a quelle dei figli**;
- **Il processo genitore attende che alcuni o tutti i suoi figli terminino**.

Per quanto riguarda gli spazi di indirizzamento dei figli invece può verificarsi che:

- **Il processo figlio sia un DUPLICATO del padre (stessi programmi e dati)**;
- **Nel processo figlio si carica un nuovo programma**.

NOTA: Un nuovo processo si crea tramite la chiamata a sistema **fork()** che restituisce 0 nel nuovo processo e l'identificatore del processo figlio nel processo padre. Il figlio eredita attributi, privilegi e alcune risorse dal padre. In genere, quando un nuovo processo viene creato, il padre o il figlio invoca **exec()** per caricare in memoria il nuovo programma.

TERMINAZIONE DI UN PROCESSO

Un processo termina quando viene eseguita la sua ultima istruzione di codice e inoltra al sistema operativo la richiesta di essere cancellato tramite la system call **exit()**. Quando un processo viene terminato, tutte le risorse da esso occupate vengono liberate e restituite al sistema operativo.

Generalmente **un processo può essere terminato solo dal padre**, quindi un processo genitore deve poter essere in grado di distinguere arbitrariamente tra

i propri figli; perciò, quando un nuovo processo viene creato il suo pid viene passato al genitore. Un processo padre può terminare uno o più figli quando si verificano le seguenti situazioni:

- **Il processo figlio ha ecceduto nell'uso di determinate risorse**;
- **Il compito assegnato al processo figlio non è più richiesto**;
- **Il processo genitore termina e l'S.O. non consente al figlio di continuare l'esecuzione**.

Quando un processo padre viene terminato, tutti i suoi figli vengono terminati. In questo caso si parla di **terminazione a cascata**.

Un processo genitore può attendere la terminazione di uno dei suoi figli invocando **wait()** la quale restituisce lo stato di uscita e il pid del figlio. Un processo che è terminato ma il cui padre non ha chiamato la **wait()** viene detto **zombie**.

Se un genitore terminasse senza invocare la wait i suoi figli diventerebbero orfani. Ogni sistema operativo affronta questa situazione in maniera differente; nel caso di Linux e UNIX, i processi orfani vengono assegnati come figli del processo **systemd**.

COMUNICAZIONE TRA PROCESSI

I processi eseguiti concorrentemente possono essere **indipendenti** o **cooperanti**. Un processo è **indipendente** quando non influisce o viene influenzato da altri processi; mentre un processo è **cooperante** quando influisce o viene influenzato da altri processi.

Un processo che non condivide i suoi dati con gli altri processi è indipendente, viceversa, un processo che condivide i dati suoi dati con gli altri processi è cooperante.

Un ambiente che consente la cooperazione tra processi ha i seguenti vantaggi:

- **Condivisione di informazioni tra più utenti**;
- **Velocizzazione del calcolo**, in quanto le attività possono essere suddivise in sub-attività parallele;
- **Modularità**, le funzioni di sistema possono essere suddivise in moduli o thread.

Lo scambio di informazioni tra i processi cooperanti avviene tramite un meccanismo di **comunicazione tra processi** detto **IPC** (**I**nter**P**rocess **C**ommunication). I modelli dell'IPC sono due:

- **Memoria condivisa**, i processi cooperanti condividono uno spazio di memoria nel quale comunicano leggendo e scrivendo i dati. Il modello a

memoria condivisa, per quanto sia più difficile da implementare rispetto all'altro, è più veloce siccome impiega il kernel solo per allocare lo spazio di memoria condivisa;

- **Scambio di informazioni**, la comunicazione tra i processi viene gestita dal sistema operativo ed è utile per scambiare piccole quantità di dati. Sebbene sia più facile da implementare, questo modello può rivelarsi più pesante e lento siccome lo scambio avviene tramite **syscall** che impegnano il kernel.

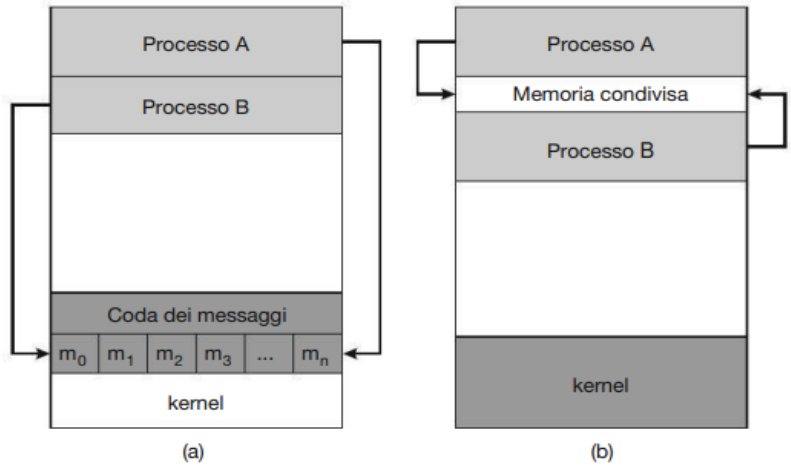


Figura 3.12 Modelli di comunicazione. (a) Scambio di messaggi. (b) Memoria condivisa.

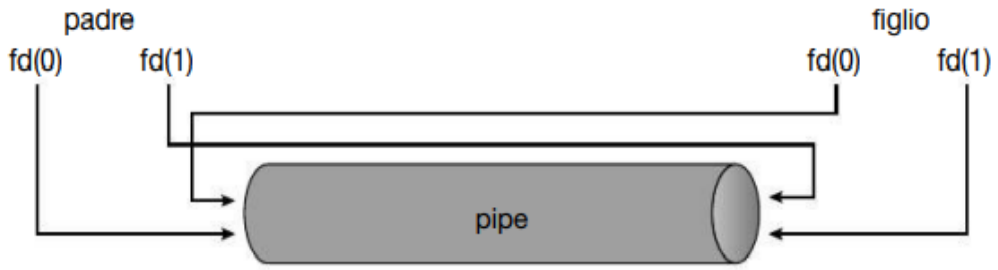
PIPE

Una **pipe** agisce come canale di comunicazione tra processi. Le pipe sono state uno dei primi meccanismi di comunicazione tra processi e forniscono ai processi uno dei metodi più semplici per comunicare l'uno con l'altro. Vi sono due tipi di pipe: **pipe convenzionali** e **named pipe**.

PIPE CONVENZIONALI

Le pipe convenzionali sono **unidirezionali** dotate di un'estremità dedicata alla lettura (**read_end**) e di un'altra estremità dedicata alla scrittura (**write_end**). Se viene richiesta la comunicazione bidirezionale bisogna usare due pipe. Le pipe convenzionali permettono a due processi di comunicare secondo una modalità standard chiamata **produttore-consumatore**. Il **produttore** scrive sulla write_end mentre il **consumatore** legge dalla read_end. Le pipe convenzionali in UNIX sono generate tramite la syscall **pipe()**. Una pipe non è accessibile al di fuori del processo che la crea. Siccome le pipe convenzionali richiedono una **relazione padre-figlio,** vengono usate dai processi genitori per

comunicare con i processi figli, i quali ereditano la pipe dal padre. La pipe è un **file speciale** e non può essere usata per la comunicazione tra processi in esecuzione su macchine diverse.



In Windows le pipe convenzionali sono chiamate **pipe anonime**.

NAMED PIPE

Le pipe convenzionali esistono solo mentre i processi stanno comunicando e cessano di esistere quando la comunicazione è terminata. Le **named pipe** costituiscono uno strumento di comunicazione molto più potente in quanto: la comunicazione può essere bidirezionale ma la trasmissione è **half-duplex** e la relazione di parentela padre-figlio non è necessaria. Una volta creata la named pipe, diversi processi possono utilizzarla per comunicare. Le named pipe continuano ad esistere anche dopo il termine della comunicazione tra i processi. Nei sistemi UNIX le named pipe sono dette **FIFO**. Una volta create (syscall **mkfifo()**), appaiono come normali file di sistema e continueranno ad esistere finché non saranno esplicitamente rimosse dal file system. Siccome le named pipe trasmettono in half-duplex se vi è la necessità di adottare il full-duplex bisogna usare due FIFO ed inoltre le named pipe possono essere usate solo da processi risiedenti sulla stessa macchina.

In Windows le named pipe possono essere usate da processi risiedenti sia sulla stessa macchina sia su macchine diverse e consentono la trasmissione full-duplex.

CAPITOLO 4

INTRODUZIONE

Fino ad adesso, abbiamo parlato di modelli basati sull'architettura **SISD** (**S**ingle **I**nstruction **S**ingle **D**ata stream), cioè un **singolo** core esegue un **singolo flusso di dati** (macchine di Von Neumann). Con l'introduzione dei thread, ci baseremo invece sull'architettura **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata stream), ovvero **più** core eseguono **simultaneamente più flussi di dati**.

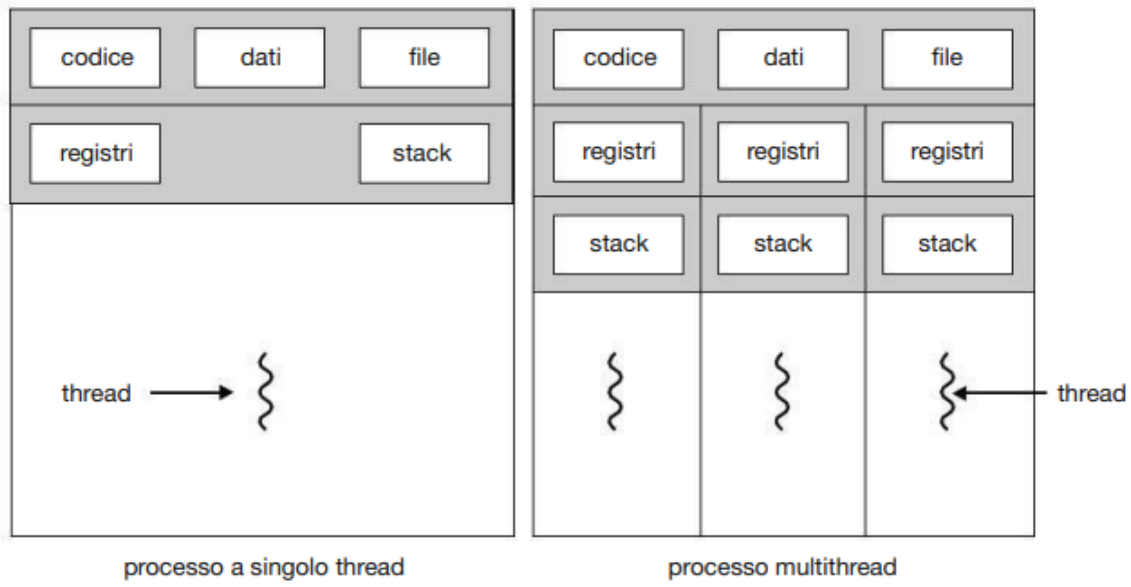
Le architetture parallele si distinguono principalmente in:

- **Architetture a Memoria Condivisa,** memoria con un unico spazio di indirizzamento, condiviso tra tutti i processori;
- **Architetture Distribuite**, ogni nodo ha la sua memoria e non la condivide con gli altri.

Nei modelli SISD un processo segue un solo percorso di controllo e può eseguire solo un compito per volta ma, in realtà la maggior parte dei sistemi che si basano su un'architettura MIMD consente ai processi di seguire **multipli** percorsi di controllo chiamati **thread** e di eseguire più compiti in contemporanea.

Un thread (**processo leggero/lightweight process**) è **un flusso indipendente d'esecuzione** e comprende: **un identificatore di thread (ID)**, **un program counter**, **un insieme di registri** e **uno stack**. Ogni thread condivide con gli altri thread che **appartengono allo stesso processo**: la sezione del codice/testo, dei dati e altre risorse come i file aperti e i segnali del processo. Un processo tradizionale, composto da un **solo thread**, è detto **processo pesante/heavyweight process**.

Ogni thread viene rappresentato da un **thread control block** (**TCB**) che punta al PCB del processo contenitore.

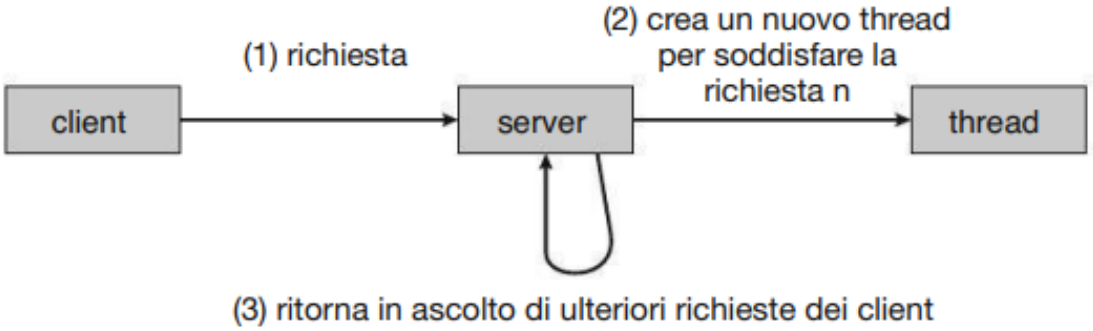


Un processo multithread è in grado di svolgere più compiti concorrentemente.

MOTIVAZIONI

La maggior parte delle applicazioni per i sistemi moderni sono **multithread**. Ad esempio, un word processor (Word, Notes ecc.) può avere un thread per la rappresentazione grafica, un altro thread per gestire l'input da tastiera e un

ulteriore thread per il controllo ortografico e grammaticale oppure, nel caso dei client-server quando il server riceve una richiesta, questo crea un thread per gestirla in modo che esso possa soddisfare più client contemporaneamente.



Le applicazioni multithread possono essere progettate per sfruttare il throughput sui sistemi multicore. Tali applicazioni possono eseguire diverse attività sfruttando la CPU in parallelo sui diversi core (algoritmi di ordinamento, per alberi e per i grafi)

La maggior parte dei kernel sono **multithread.**

VANTAGGI

I principali vantaggi della programmazione multithread sono i seguenti:

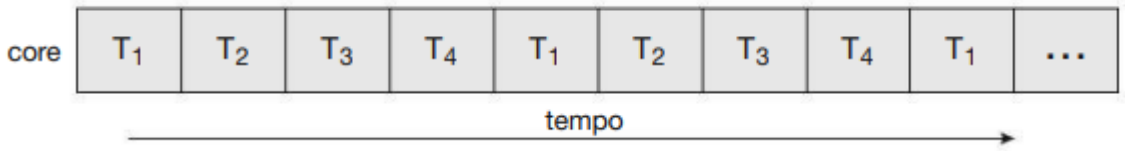
- **Tempo di risposta**. Rendere un'applicazione multithread permette a un programma di continuare la sua esecuzione anche se una parte di esso è bloccata o sta eseguendo un'operazione lunga;
- **Condivisione delle risorse**. I processi devono poter comunicare tra di loro e condividere determinante risorse, di norma questo è possibile attraverso la memoria condivisa o lo scambio di messaggio che vanno implementati; tuttavia, nel caso dei thread ciò non è necessario siccome **i thread di default condividono il codice e i dati del processo contenitore**;
- **Economia**. Quando si crea un nuovo processo bisogna allocare nuove risorse ma, nel caso dei thread, quest'operazione, per quanto rimanga onerosa, risulta più semplice siccome i **thread condividono le risorse del processo contenitore**. Di conseguenza diventa più semplice e veloce gestire gli switch context dei thread.
- **Scalabilità**. Nei sistemi multiprocessore, i thread possono essere eseguiti in **parallelo** su diversi core di elaborazione.

PROGRAMMAZIONE MULTICORE

In passato, in risposta alla necessità di una maggiore potenza di calcolo, si è passati dai sistemi monoprocessore ai sistemi multiprocessore. Ed oggi, con una tendenza simile, si stanno sviluppando e diffondendo i sistemi multicore, in cui su un singolo chip vi sono più core di elaborazione che appaiono al sistema come processori separati.

La programmazione multithread offre un meccanismo per un utilizzo più efficiente dei multicore e aiuta a sfruttare meglio la concorrenza.

NOTA: In un sistema monoprocessore, "esecuzione concorrente" significa che l'esecuzione dei thread è avvicendata nel tempo (**interleaved**) perché la CPU può eseguire solo un thread per volta.



In un sistema multiprocessore, "esecuzione concorrente" significa che i thread possono essere eseguiti in parallelo siccome il sistema può assegnare più thread a più core.

Un **sistema concorrente** supporta più task permettendo a ciascuno di progredire l'esecuzione (immagine sopra). Un **sistema parallelo**, invece, può eseguire più task simultaneamente. **È possibile avere la concorrenza senza il parallelismo**.

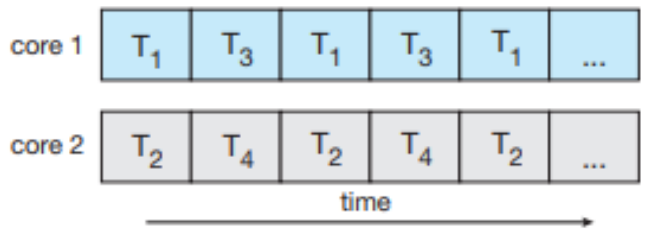


Figure 4.4 Parallel execution on a multicore system.

TIPI DI PARALLELISMO

Esistono due tipi di parallelismo:

- **Parallelismo dei dati**, riguarda la distribuzione di sottoinsiemi di dati e l'elaborazione della **stessa** operazione su più core. Ad esempio, se si considera la somma degli elementi di un vettore, in un sistema multicore si potrebbero avere due thread distinti: thread A e thread B. Il thread A potrebbe effettuare la somma degli elementi da [0] a [N/2 – 1] e il thread B la somma da [N / 2] a [N – 1];

- **Parallelismo delle attività**, prevede la distribuzione di attività (thread) su più core. I thread operano in parallelo su più core ma svolgono operazioni distinte.

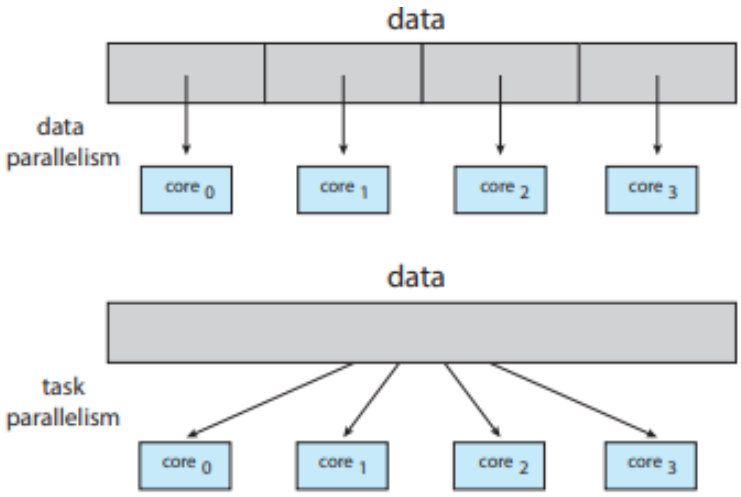


Figure 4.5 Data and task parallelism.

Le applicazioni possono sfruttare entrambi i tipi di parallelismo. **Parallelismo dei dati e delle attività NON sono mutualmente esclusivi**.

MODELLI DI SUPPORTO AL MULTITHREADING

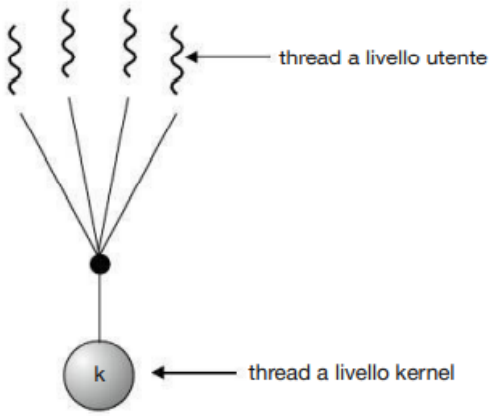
I thread possono essere distinti in:

- **Thread a livello utente**, gestiti sopra il livello del kernel e **senza** il suo supporto tramite una **libreria specifica**;
- **Thread a livello kernel**, gestiti direttamente dal sistema operativo tramite chiamate di sistema.

I thread a livello utente vengono messi in relazione con i thread a livello kernel per permettere l'accesso alle risorse. Esistono diversi tipi di relazione tra thread a livello utente e thread a livello kernel.

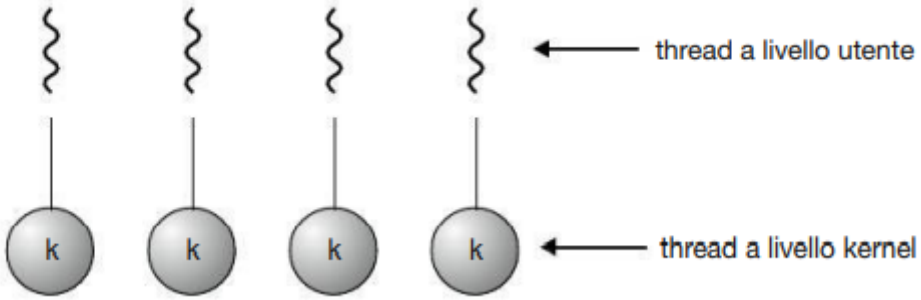
MODELLO DA MOLTI A UNO

Il modello da molti a uno fa corrispondere **molti** thread utente a un **singolo** thread kernel. La gestione dei thread risulta inefficiente per diversi motivi: i thread vengono gestiti da una libreria di thread nello spazio utente, l'intero processo viene bloccato se uno dei thread invoca una chiamata di sistema **bloccante** e, siccome solo **UN** thread per volta può accedere al kernel non è possibile gestire thread multipli in parallelo.



MODELLO DA UNO A UNO

Il modello da uno a uno mette in corrispondenza **ciascun** thread utente con **un** thread kernel. Questo modello offre una concorrenza maggiore rispetto a quello da molti a uno, siccome anche se un thread si blocca è possibile continuare ad eseguire un altro thread; il modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore. L'unico svantaggio è che la creazione di ogni thread utente comporta la creazione del corrispondente thread kernel appesantendo di un carico rilevate il sistema; infatti, la maggior parte dei sistemi che adotta questo modello stabilisce un limite del numero dei thread supportabili.

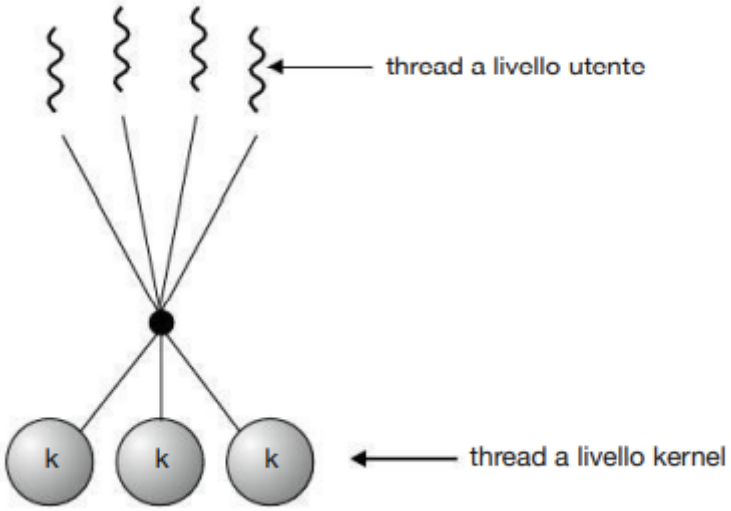


MODELLO DA MOLTI A MOLTI

Il modello da molti a molti mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread kernel. Nel modello da molti a molti i programmatori possono creare liberamente i thread che ritengono necessari e i corrispondenti thread kernel. I thread si possono eseguire in parallelo e se un thread si blocca, il kernel ne può schedulare un altro.

Una variante del modello da molti a molti, detto **modello a due livelli**, permette di associare a un thread utente un unico thread kernel.

L’implementazione di questo modello, per quanto esso risulti più flessibile rispetto agli altri, è molto più complessa ed inoltre, con un numero crescente di core, la maggior parte dei sistemi tendono ad utilizzare il modello da molti a uno.



CAPITOLO 5

INTRODUZIONE

Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati siccome, attraverso la commutazione della CPU tra i vari processi, il sistema operativo può rendere più produttivo il calcolatore.

CONCETTI FONDAMENTALI

Lo **scopo** della multiprogrammazione è quello di tenere sempre attiva la CPU. Per questo motivo, si mantengono in memoria più processi e, quando uno dei processi in esecuzione (sistemi multicore/multiprocessori) si ferma perché deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU e lo concede a un altro processo.

CICLITA’ DELLE FASI D’ELABORAZIONE E DI I/O

L’esecuzione di un processo consiste in un **ciclo di elaborazione** (svolto dalla CPU) e d’attesa del completamento delle operazioni di I/O. I processi si **alternano** tra questi due stati (**ready** – **wait**).

L’esecuzione di un processo comincia con una sequenza di istruzioni svolte dalla CPU (**CPU burst**), seguita da una sequenza di operazioni di I/O (**I/O**

burst) alla quale sussegue un'altra CPU burst e così via. L'ultima CPU burst è una richiesta al sistema di terminare l'esecuzione.

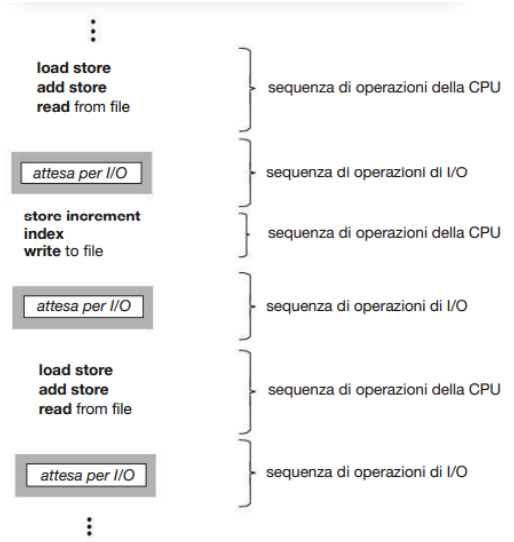


Figura 6.1 Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.

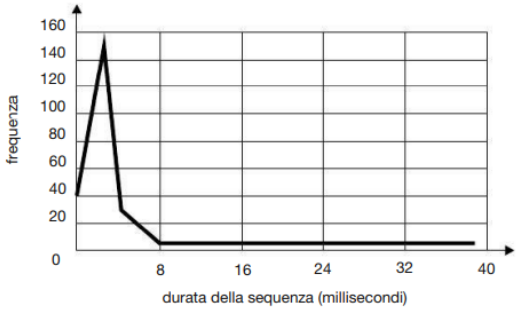


Figura 6.2 Diagramma delle durate delle sequenze di operazioni della CPU.

NOTA: Burst = Utilizzo massimo della periferica.

SCHEDULER DELLA CPU

Ogni qualvolta la CPU passa in uno stato di inattività, lo **scheduler a breve termine** o **scheduler della CPU**, sceglie, tra i processi contenuti in memoria presenti nella ready queue, un processo da assegnare alla CPU.

La ready queue non è necessariamente una coda **FIFO** (**first-in first-out**) ma può essere gestita come una coda LIFO, a priorità, ad albero o come una lista in base all'algoritmo di scheduling usato dall'S.O.

Nella coda sono contenuti i PCB dei processi.

SCHEDULING CON E SENZA PRELAZIONE

Le decisioni riguardanti lo scheduling della CPU vengono prese nelle seguenti quattro circostanze:

- Un processo passa dallo stato di esecuzione allo stato di attesa;

- Un processo passa dallo stato di esecuzione allo stato pronto;
- Un processo passa dallo stato di attesa allo stato pronto;
- Un processo termina.

Nel primo e nell'ultimo caso, lo scheduler **deve** scegliere un nuovo processo da eseguire e in questi casi si dice che lo **schema di scheduling** è **senza prelazione** (**nonpreemptive**) o **cooperativo** (**cooperative**). Negli altri casi lo schema di scheduling è **con prelazione** (**preemptive**). Nel caso dello scheduling nonpreemptive il processo scelto rimane in possesso della CPU fino al suo rilascio che può essere causato da un passaggio allo stato di attesa o dalla sua terminazione.

Lo scheduling preemptive può portare a **race conditions** quando i dati sono condivisi tra più processi o quando i dati/strutture dati di un determinato processo si trovano in uno stato incoerente. La capacità di prelazione si ripercuote sulla progettazione del kernel, il quale può essere: **senza prelazione**, quindi ha una struttura più semplice assente da race conditions che aspetta il termine o il cambio di stato di un processo/syscall prima di cambiare contesto (non si rispettano i requisiti per le elaborazioni in tempo reale), oppure **con prelazione** dove, per non causare perdite di dati, i processi, talvolta, disattivano le linee di interruzione al loro avvio e le riattivano al loro rilascio.

NOTA: **Prelazione ->** è la preferenza di un soggetto rispetto a un altro.

NOTA: Una race condition si verifica quando l'accesso ai dati condivisi non è controllato ed eventualmente può portare a valori corrotti dei dati condivisi.

DISPATCHER

Un altro componente della funzione dello scheduling della CPU è il **dispatcher**, ovvero il modulo che effettivamente si occupa di effettuare il cambio di contesto dei processi.

Il tempo richiesto dal dispatcher per fermare l'esecuzione di un processo e avviarne un altro è noto come **latenza di dispatch**.

Distinguiamo i cambi di contesto in:

- **Cambi di contesto volontari**, si verificano quando un processo cede il controllo della CPU in seguito a una richiesta che non è al momento disponibile (es. operazione di I/O);
- **Cambi di contesto involontari**, il controllo della CPU viene sottratto al processo in quanto questo ha finito il tempo a disposizione o è stata effettuata una prelazione da un processo con priorità più alta.

CRITERI DI SCHEDULING

I criteri principali usati per confrontare i diversi algoritmi di scheduling della CPU sono i seguenti:

- **Utilizzo della CPU**. La CPU deve essere più attiva possibile;
- **Throughput**, corrisponde alla **produttività** della CPU ovvero a quanti processi ha eseguito in una data unità di tempo;
- **Tempo di completamento**, il tempo necessario al completamento del processo. L'intervallo di tempo che intercorre tra la sottomissione del processo e il suo completamento è detto **tempo di completamento** o **turnaround time**;
- **Tempo d'attesa**. L'algoritmo di scheduling **influisce** sui **tempi di attesa** nella ready queue. Il tempo d'attesa è la somma di questi intervalli, ovvero quanto tempo aspetta il processo nella ready queue prima di essere eseguito;
- **Tempo di risposta**, è il tempo che intercorre tra l'effettuazione di una richiesta e la prima risposta prodotta (parziale o totale che sia).

ALGORITMI DI SCHEDULING

Lo scheduling della CPU ha il compito di scegliere quali dei processi, presenti nella ready queue, debba essere assegnato al core della CPU. Esistono diversi algoritmi di scheduling che per il momento verranno descritti su un sistema monoprocessore ed in seguito su sistemi multiprocessore.

Prima di iniziare, distinguiamo tre tipi di sistemi ad ognuno dei quali si associa un tipo di algoritmo di scheduling diverso:

- **Sistemi real-time**;
- **Sistemi batch**;
- **Sistemi time-sharing/interattivi.**

SCHEDULING IN ORDINE DI ARRIVO

L'algoritmo di scheduling più semplice è l'algoritmo di **scheduling in ordine d'arrivo** (**scheduling first-come, first-served** o **FCFS**) ed è **privo di prelazione**. Come suggerisce il nome, l'algoritmo assegna alla CPU il primo processo che la richiede e la ready queue viene gestita come una coda FIFO.

Il principale aspetto negativo di quest'algoritmo è che il tempo medio d'attesa è lungo. Ad esempio:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se i processi arrivano nell'ordine prestabilito, quindi: P_1, P_2, P_3 si verifica la seguente situazione (la durata è espressa in ms):

In breve, ciò che accade con l'algoritmo FCFS è che i processi con prevalenza di elaborazione (CPU Bound), e che quindi occupano per molto tempo la CPU, tengono ad essere eseguiti per primi rispetto ai processi I/O Bound, che utilizzano la CPU per intervalli brevi, finendo con l'aumentare i tempi di attesa e a diminuire il throughput (**processo convoglio**).

L'implementazione degli algoritmi FCFS è problematica ed è sconsigliata nei sistemi **time-sharing**.

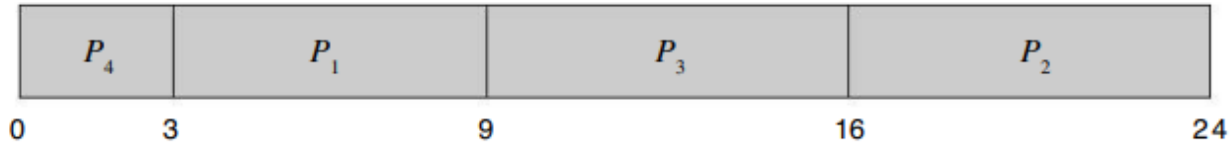
SCHEDULING SHORTEST-JOB-FIRST

Un altro algoritmo di scheduling è l'**algoritmo di scheduling per brevità** (**shortest-job-first** o **SJF**). Sarebbe più appropriato riferirsi a questo tipo di algoritmo come **shortest-next-CPU-burst** siccome questa tipologia di algoritmo associa a ogni processo la lunghezza della **successiva** CPU burst ed esegue i processi con lunghezza minore. Nel caso in cui due o più processi abbiano la stessa durata di CPU burst, l'algoritmo si comporta come un FCFS.

Diamogli un'occhiata.

Processo	Durata della sequenza
P_1	6
P_2	8
P_3	7
P_4	3

Con lo scheduling SJF l'ordine dei processi eseguiti sarebbe il seguente:



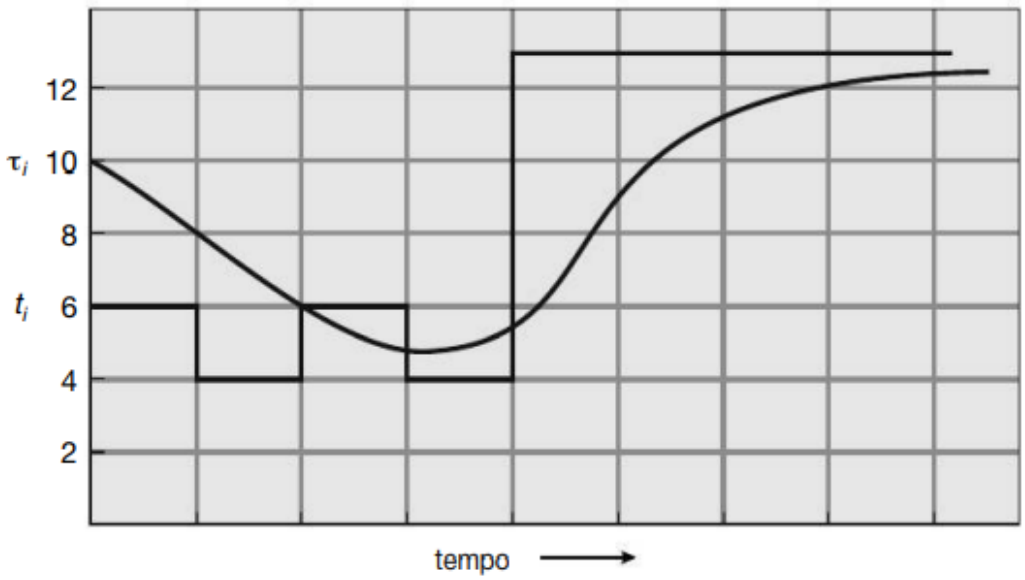
Come si può notare, siccome vengono eseguiti prima i processi più brevi, il tempo di attesa medio si abbassa notevolmente e, in un lasso di tempo più o meno simile a quello degli algoritmi FCFS, vengono eseguiti più processi aumentando il throughput.

Abbiamo detto che l'algoritmo SJF si basa sull'eseguire il processo più breve in base alla lunghezza della sua prossima CPU burst; tuttavia, non è possibile conoscere con **esattezza** il valore della prossima CPU burst, ma è possibile **predirne** il valore. Quest'operazione di "approssimazione" viene effettuata attraverso il calcolo della **media esponenziale** delle lunghezze delle precedenti CPU burst.

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

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

NOTA: Il prof non chiede la media esponenziale, quindi ho saltato la spiegazione di cosa contengono le variabili.



CPU burst (t_i)	6	4	6	4	13	13	13	...
predizione (τ_i)	10	8	6	6	5	9	11	12	...

Figura 6.3 Predizione della lunghezza della successiva sequenza di operazioni della CPU (CPU burst).

L'algoritmo SJF può essere sia **con prelazione** (**preemptive**) sia **senza prelazione** (**non preemptive**).

Quando arriva un nuovo processo nella ready queue, nell'algoritmo preemptive, chiamato anche **shortest-remaining-time-first**, se il nuovo processo ha una durata minore rispetto a quella **rimasta** al processo in esecuzione, il processo in esecuzione viene sostituito col processo appena arrivato; mentre nell'algoritmo non preemptive il processo in esecuzione rimane in possesso della CPU.

SCHEDULING CIRCOLARE

L'algoritmo di **scheduling circolare** (**round-robin**, **RR**) è simile allo scheduling FCFS ma implementa la **prelazione** in modo che il sistema possa commutare fra i vari processi.

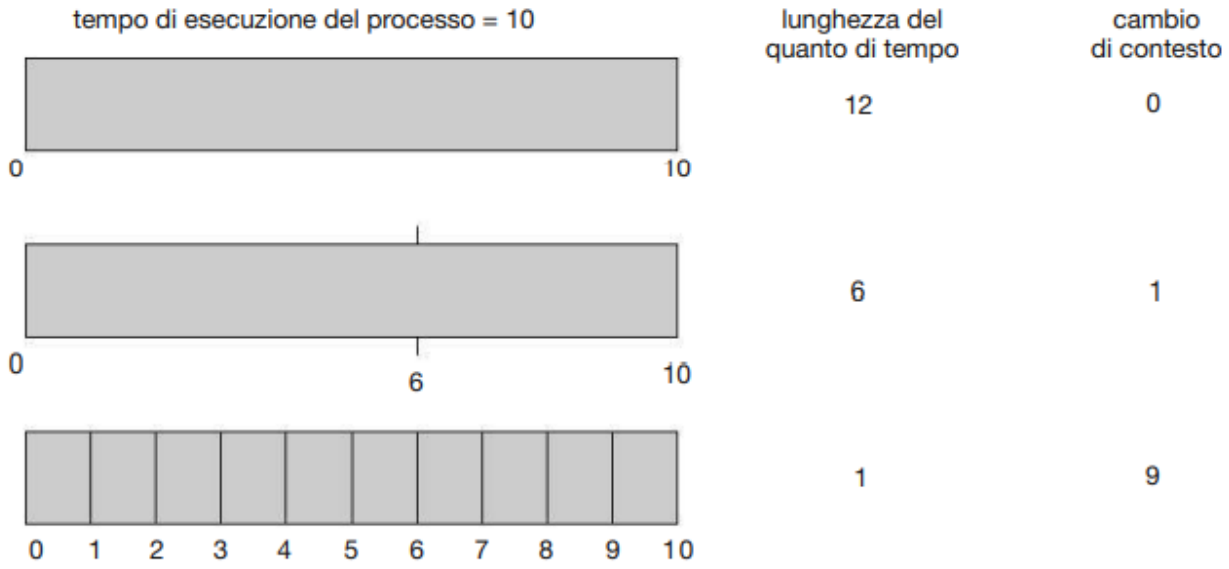
Nel round-robin, a ciascun processo viene assegnata una quantità di tempo prefissata detta **quanto di tempo**/**quantum** o **porzione di tempo**/**time slice**; la ready queue è trattata come una coda circolare e viene gestista come una coda FIFO. Lo scheduler della CPU manda in esecuzione il primo processo della ready queue, gli assegna un timer pari a un quanto di tempo e, quando il timer scatta, viene inviato un interrupt e viene attivato il dispatcher per eseguire il processo successivo.

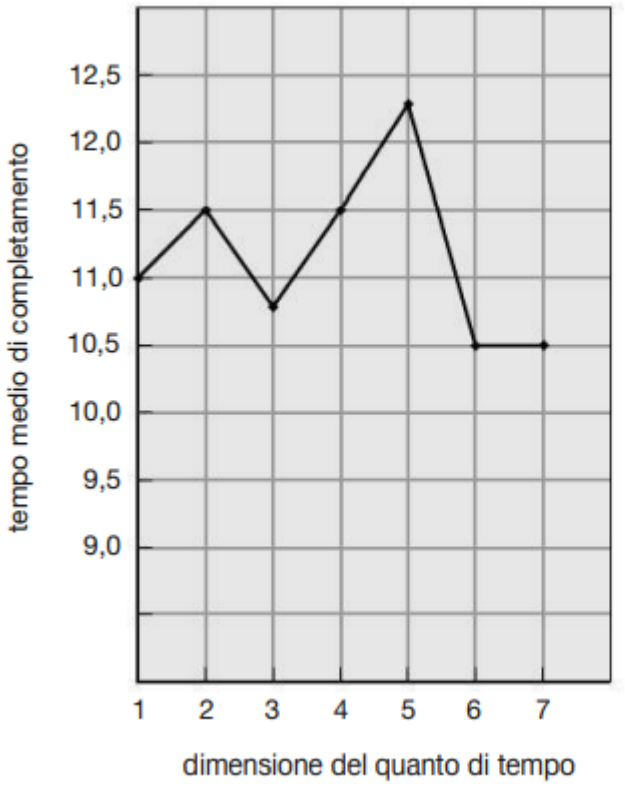
Quando un processo viene selezionato per essere eseguito possono accadere due cose: il processo termina entro il quanto di tempo e rilascia volontariamente la CPU oppure; il processo necessita di più di un quanto di tempo, quindi lo scheduler invia un interrupt, il dispatcher effettua un cambio di contesto e il processo viene reinserito alla fine della ready queue. Il processo in questione “circola” fin quando non termina.

Il tempo di attesa dell’RR è abbastanza lungo come quello dell’FCFS.

Siccome con il RR a ciascun processo viene assegnato un certo quanto di tempo, se nella ready queue si dispongono di **n** processi e il quanto di tempo è pari a **q**, ciascun processo avrà a disposizione 1/n-esimo tempo della CPU in frazioni di q unità di tempo. Ogni processo non deve attendere più di **(n – 1) * q** unità di tempo per la sua prossima esecuzione.

Le prestazioni del round-robin e il tempo di completamento dei processi dipendono molto (direttamente per il tempo di completamento) dalla dimensione del quanto di tempo. All’aumentare del quanto di tempo diminuiscono i cambi di contesto e, al diminuire del quanto di tempo aumentano i cambi di contesto e il tempo di completamento del processo cresce.





processo	tempo
P_1	6
P_2	3
P_3	1
P_4	7

Il quanto di tempo deve essere ampio rispetto alla durata del cambio di contesto (circa 10 microsecondi). Nella maggior parte dei sistemi, il quanto di tempo varia tra i 10 e i 100ms.

Empiricamente, si può stabilire che l'80% delle CPU burst debba essere più breve del quanto di tempo.

SCHEDULING CON PRIORITÀ

L'algoritmo SJF è un caso particolare di **algoritmo di scheduling con priorità**: si associa a ogni processo una priorità e si eseguono i processi con la priorità più alta; in caso di priorità uguale i processi si ordinano secondo uno schema FCFS.

A ogni priorità viene associato un numero e, in generale, più basso è il numero più è alta la priorità e, viceversa, più è alto il numero più è bassa la priorità ma, ogni sistema operativo è libero di gestire le priorità come meglio "crede".

Gli esempi di algoritmo di scheduling con priorità verranno basati sull'algoritmo SJF (più il processo è breve più è alta la priorità).

Supponiamo di avere i seguenti processi nella ready queue:

Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Secondo l'algoritmo di scheduling con priorità, l'ordine di esecuzione sarebbe la seguente:

Il tempo d'attesa medio è di 8,2 ms.

Le priorità si possono definire sia **internamente** sia **esternamente**. Le priorità **interne** usano una o più quantità misurabili per calcolare la priorità del processo, ad esempio i limiti di tempo, i file aperti, i requisiti di memoria ed altro. Le priorità **esterne** si definiscono in base a fattori, appunto, **esterni** al sistema operativo come l'importanza del processo, il tipo ecc.

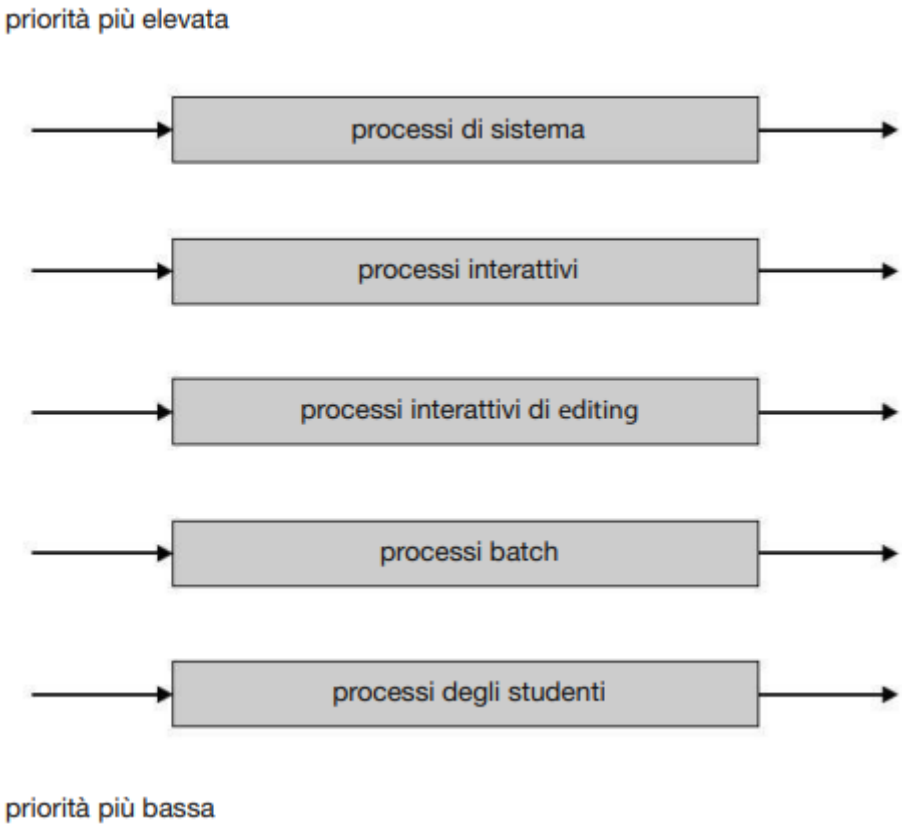
Lo scheduling con priorità può essere sia **con prelazione** sia **senza prelazione**. Nello scheduling preemptive il possesso della CPU viene sottratto al processo attualmente in esecuzione se la priorità del processo appena arrivato è superiore; mentre, nello scheduling non preemptive, l'algoritmo si limita a spostare in testa alla coda il nuovo processo.

Un problema importante degli algoritmi di scheduling con priorità è l'**attesa indefinita/starvation**. Siccome gli algoritmi con priorità preferiscono i processi ad alta priorità rispetto a quelli con bassa priorità spesso capita che quest'ultimi rimangano all'interno della ready queue senza mai essere eseguiti siccome vengono spostati sempre di più verso il basso ogni volta che arriva un processo con priorità superiore. Per ovviare a questa problematica, viene usato il processo di **invecchiamento/aging** dei processi: man mano che un processo rimane nello stato di attesa (man mano che invecchia) cresce, in maniera graduale, la sua priorità. Un'altra soluzione utilizzata consiste nel combinare lo scheduling circolare (RR) con lo scheduling con priorità.

SCHEDULING A CODE MULTILIVELLO

Nello scheduling con priorità e in quello circolare, i processi vengono inseriti in un'unica coda e questo implica che, nel caso peggiore, per ricercare il processo con la priorità più alta occorrerebbe effettuare una ricerca ***O***(***n***).

Per ridurre i tempi di ricerca, viene utilizzato lo **scheduling a code multilvello** che consiste nel suddividere processi di diverso tipo o di diverse priorità in più code, ognuna associata a un tipo o una priorità di processo diverso. A ogni coda potrebbe venire associare un algoritmo di scheduling differente. Un processo rimane nella stessa coda per tutta la sua esecuzione.



Nel caso delle code formate in base al tipo di processo è necessario stabilire quale coda ha priorità sulle altre e, di norma, viene utilizzato uno scheduling con priorità fissa preemptive. Ogni coda ha priorità **assoluta** sulle code sottostanti. Un'altra soluzione comprende definire piccole porzioni di tempo per ciascuna coda.

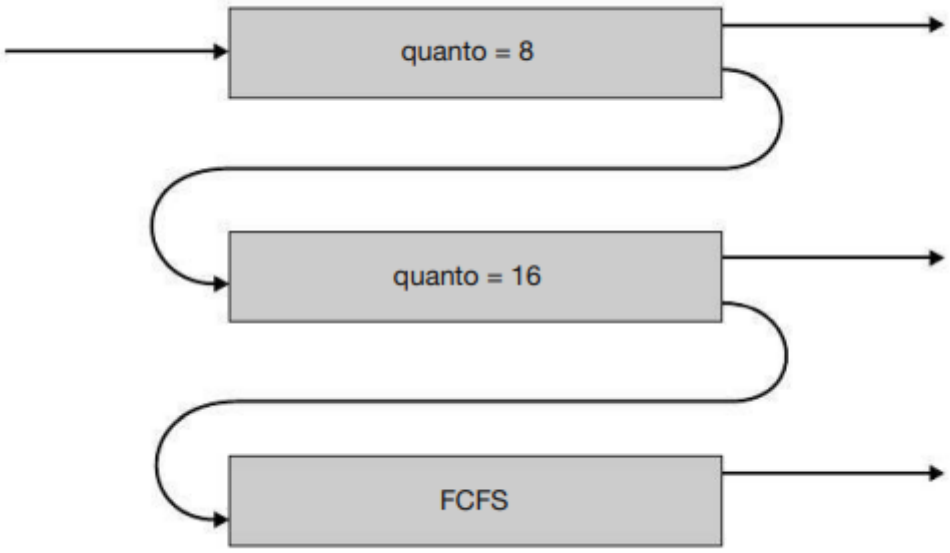
In un sistema **real-time** le code definite in base al tipo dei processi, in ordine di priorità, potrebbero essere le seguenti:

- **Processi in tempo reale**;
- **Processi di sistema**;
- **Processi interattivi**;
- **Processi batch**;

SCHEDULING A CODE MULTILIVELLO CON RETROAZIONE

Lo scheduling a code multilivello non consente ai processi di spostarsi fra le code mantenendo un basso costo di scheduling.

Lo **scheduling a code multilivello con retroazione** (**multilevel feedback queue scheduling**), invece, consente ai processi di spostarsi fra le code. L'idea di base consiste nel separare i processi che hanno caratteristiche diverse in termini di CPU burst. Se un processo impiega troppo tempo di elaborazione, viene spostato in una coda a bassa priorità. Questo schema mantiene i processi interattivi e i processi I/O bound nelle code ad alta priorità e gestisce lo **starving** spostando i processi che attendono troppo a lungo da una coda a bassa priorità a una coda ad alta priorità. Il tempo di aging varia da coda a coda.



Generalmente, uno scheduler a code multilivello con retroazione è caratterizzato dai seguenti parametri:

- **Numero di code**;
- **Algoritmo di scheduling per ciascuna coda**;
- **Metodo usato per stabilire quando spostare un processo in una coda con priorità più alta**;
- **Metodo usato per stabilire quando spostare un processo in una coda con priorità più bassa**;
- **Metodo usato per stabilire in quale coda spostare un processo quando richiede un servizio**.

La definizione di uno scheduler a code multilivello con retroazione costituisce il più generale criterio di scheduling della CPU.

SCHEDULING DEI THREAD

Nei sistemi operativi che supportano i thread e che dunque prevedono la presenza dei thread utente e dei thread kernel, il sistema non effettua lo scheduling dei processi ma bensì dei thread kernel.

AMBITO DELLA CONTESA

Una distinzione fra thread a livello utente e a livello kernel riguarda il modo in cui vengono schedulati.

Nei sistemi che implementano i modelli da molti a uno e da molti a molti, i thread utente vengono schedulati dalla libreria specifica che li associa a un LWP libero (**LightWeight Process**) e si parla in questo caso di **ambito di contesa ristretto al processo** (**process-contention scope**, **PCS**) perché la contesa per la CPU ha luogo tra thread dello stesso processo. Il PCS è solitamente basato su uno scheduling con priorità.

Anche se la libreria associa un thread utente a un LWP, non significa che vi sia un thread in esecuzione sulla CPU; ciò avviene solo quando il sistema operativo pianifica l'esecuzione di un thread kernel sulla CPU e in questo caso si parla di **ambito di contesa allargato al sistema** (**system-contention scope**, **SCS**) poiché tutti i thread del kernel sono in competizione per aggiudicarsi la CPU.

SCHEDULING PER SISTEMI MULTIPROCESSORE

Fino ad adesso, lo scheduling ha riguardato solo sistemi monoprocessore.

Nel caso dei sistemi multiprocessore, disponendo di più core, è possibile **distribuire il carico** (**load sharing**) tra più unità di elaborazione ma, ovviamente, lo scheduling diventa molto più complesso.

Oggi giorno, si distinguono diverse architetture multiprocessore:

- **CPU multicore**;
- **Core multithread**;
- **Sistemi NUMA (Non Uniform Memory Access**);
- **Sistemi multiprocessore eterogenei** (sistemi distribuiti).

Le prime tre architetture presentano processori **omogenei** in termini di funzionalità, mentre l'ultima presenta processori **eterogenei**, ovvero non identici in termini di prestazioni e architettura.

APPROCCI ALLO SCHEDULING PER MULTIPROCESSORI

Distinguiamo due tipologie di approcci per la gestione dello scheduling nei sistemi multiprocessore:

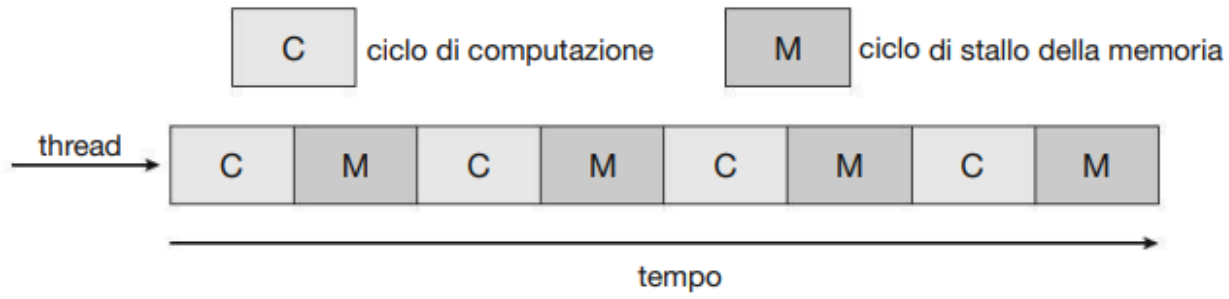
- **AMP**, ovvero la multielaborazione asimmetrica. Con questo approccio, lo scheduling della CPU affida tutte le decisioni e le altre attività di sistema a un'unica CPU detta **master server** mentre gli altri processori eseguono solo il codice utente. Ovviamente questa soluzione risulta poco efficiente in quanto non utilizza appieno tutte le risorse del sistema;
- **SMP**, ovvero la multielaborazione simmetrica. Con questo approccio, ogni processore è in grado di autogestirsi e lo scheduling viene realizzato facendo in modo che lo scheduler di ogni CPU esamini la ready queue e selezioni un thread da eseguire. Questo approccio offre due soluzioni:
    - Tutti i thread possono trovarsi in una ready queue comune;
    - Ogni processore può avere una ready queue privata.

L'approccio più comunemente utilizzato è quello che ogni processore possiede una propria ready queue privata che risolve i problemi derivanti dall'avere una ready queue pubblica e condivisa; tuttavia, quest'approccio risulta in un carico di lavoro maggiore siccome si devono gestire **n** ready queue.

PROCESSORI MULTICORE

I sistemi SMP che utilizzano processori multicore sono molto più veloci e consumano meno energia rispetto ai sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore, tuttavia, possono complicare i problemi relativi allo scheduling. Si è constatato che quando un processore accede alla memoria trascorre una significativa quantità di tempo, detta **stallo della memoria**, prima che i dati diventino disponibili.



Per rimediare a questa situazione, solitamente, si implementano dei core **multithread** in cui due o più thread hardware sono assegnati a un singolo core. In questo modo se un thread è in una situazione di stallo, il core può eseguire un altro thread. Questa tecnica è nota come **chip multithreading** (**CMT**) o, nel caso dei processori Intel, **hyper-threading** (**multithreading simultaneo** o **SMT**). Ogni thread hardware viene visto come una CPU logica dal sistema operativo.

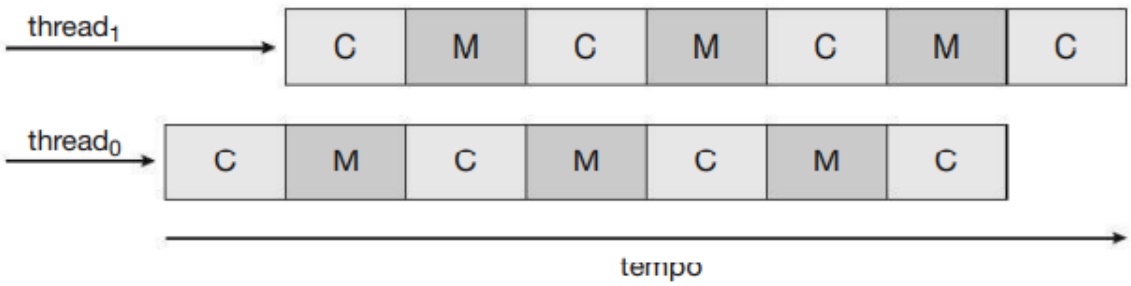


Figura 6.11 Sistema multicore e multithread.

Ci sono due modi per rendere un core multithread:

- **Multithreading a grana grossa** (**coarse-grained**), in cui un thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza (es. stallo di memoria). Quando ciò accade, il processore **deve** passare ad eseguire un altro thread; tuttavia, il cambio di thread è un'operazione onerosa siccome bisogna ripulire la pipeline dalle istruzioni del thread precedente;
- **Multithreading a grana fine** (**fine-grained**) o **interleaved multithreading**, in cui il processore passa da un thread a un altro tramite un sistema logico dedicato mantenendo il costo del cambio basso.

È essenziale che le risorse del core fisico (come cache e pipeline) siano condivise tra tutti i thread hardware e dunque un core può eseguire solo un thread per volta. Ne segue che un processore multithread e multicore necessita di due diversi livelli di scheduling. A un livello, il sistema operativo sceglie quale thread software eseguire su ogni thread hardware (CPU logica). Sul secondo livello, lo scheduler specifica in che modo ogni core decide quale thread hardware eseguire; di norma viene usato un RR. I due livelli di scheduling NON sono mutualmente esclusivi; infatti, se lo scheduler del primo livello conosce quali risorse del processore stanno venendo usate (disponibili nel secondo livello) sarà in grado di prendere decisioni più efficienti.

BILANCIAMENTO DEL CARICO

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutte le unità elaborative in modo da sfruttare appieno i vantaggi di avere più processori. Ciò avviene tramite il **bilanciamento del lavoro** che può seguire due approcci:

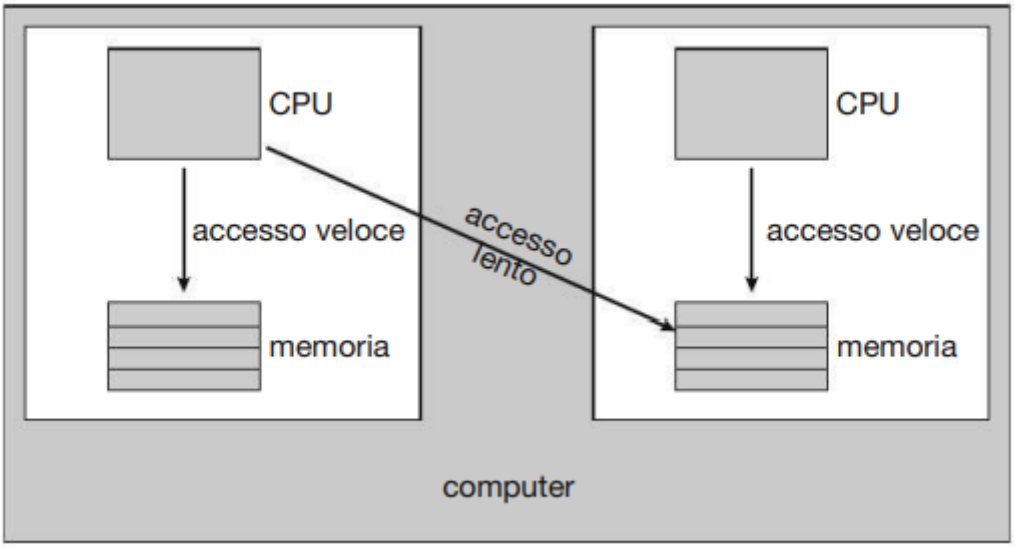
- **La migrazione push**, la quale prevede che un processo controlli periodicamente il carico di ogni processore e, se identifica uno sbilanciamento, riporta il carico in equilibrio spostando i processi dal processore saturato ad altri più liberi o inattivi;
- **La migrazione pull**, che avviene quando un processo inattivo prende un processo in attesa a un processo in sovraccarico.

Entrambe le migrazioni NON sono mutualmente esclusive.

PREDILEZIONE DEL PROCESSORE

Quando un processo viene eseguito su un processore, i dati trattati più recentemente rimangono sulla cache del processore e, di conseguenza, i successivi accessi alla memoria da parte del processore avvengono nella **cache** (**warm cache**). Se un processo, che è stato eseguito su un processore X, al suo prossimo ciclo di istruzioni viene eseguito su un processore Y bisogna svuotare la cache del processore X e riempire la cache del processore Y. Siccome questa è un'operazione costosa, i sistemi SMP tendono a far sì che un thread venga eseguito sempre sullo stesso processore. Si parla in questo caso di **predilezione per il processore** (**processor affinity**), intendendo che un processo ha una predilezione per il processore su cui è in esecuzione.

La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo sullo stesso processore ma non può garantire che sarà così, si parla di **predilezione debole** (**soft affinity**). Quando un sistema operativo invece permette a un processo di specificare un sottoinsieme di processi su cui può essere eseguito si parla di **predilezione forte** (**hard affinity**) realizzata tramite chiamate di sistema apposite. L'architettura della memoria influisce sulla predilezione (es. sistemi NUMA).



È quasi ironico che il bilanciamento del lavoro e la predilezione del processore entrino, talvolta in contrasto tra di loro siccome la predilezione del processore fa sì che un thread possa usufruire dei vantaggi della warm cache ma dall'altra parte il bilanciamento del lavoro, il cui compito è quello di spostare i processi tra un processore e l'altro, rimuove questo vantaggio.

MULTIPROCESSING ETEROGENEO

Negli esempi visti fino ad adesso, tutti i processori erano identici in termini di prestazioni e architettura. Tuttavia, vi sono sistemi, come i sistemi distribuiti o i sistemi mobili, che includono architetture multicore nonostante posseggano processori eterogenei. Questi sistemi vengono chiamati **sistemi a multielaborazione eterogena** o **sistemi HMP** (**Heterogenous Multiprocessing**). Solitamente questi sistemi offrono una combinazione di core più veloci e più lenti in modo che lo scheduler possa assegnare ogni processo al core più adatto in base alle specifiche richieste. Ad esempio, i processi che tendono ad impegnare la CPU per lassi di tempo lunghi vengono assegnati ai core più lenti che consumano meno energia e che dunque possono essere utilizzati per più tempo; mentre i processi che tendono ad usare la CPU per lassi di tempo più brevi vengono assegnati ai core più veloci che consumano più energia.

VALUTAZIONE DEGLI ALGORITMI

Il primo passo per scegliere che algoritmo di scheduling adottare per un determinato sistema è quello di stabilire i criteri di selezione e le misure che si intendono raggiungere o mantenere (es. tempo di completamento, % di utilizzo della CPU ecc.)

MODELLAZIONE DETERMINISTICA

Fra i metodi di valutazione sono di grande importanza quelli che rientrano nella classe della **valutazione analitica** che, partendo dall'algoritmo dato e dal carico di lavoro, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo.

La **modellazione deterministica** è un tipo di valutazione analitica che considera un carico di lavoro predeterminato e fornisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

SINCRONIZZAZIONE DEI PROCESSI

INTRODUZIONE

Negli ultimi anni, i calcolatori si sono elevati al di sopra della semplice esecuzione sequenziale ed adesso sono in grado di svolgere più operazioni in modo **concorrente** e/o **parallelo**. La complicazione che segue l'esecuzione concorrente e/o parallela è la condivisione delle risorse accessibili dai vari processi o thread in esecuzione. Se due o più processi/thread condividono le stesse risorse o parte di esse, ciò che spesso capita è che i dati condivisi ed utilizzati dai processi diventano **inconsistenti** (inaffidabili).

La concorrenza e la parallelizzazione hanno distaccato gli elaboratori dal calcolo **deterministico** tipico dell'architettura di Von Neumann siccome, quando si hanno due o più processi che condividono anche solo un'allocazione di memoria, la sequenza di operazioni cambia e non è più possibile stabilire a priori il risultato (**impredicibilità**) se non si gestisce la **sezione critica**.

PROBLEMA DELLA SEZIONE CRITICA

Si parla di **problema della regione/sezione critica** quando due o più processi accedono alla stessa variabile condivisa.

Il problema della regione critica consiste nel progettare un protocollo che consenta ai processi di cooperare. Ogni processo deve chiedere il permesso di entrare nella propria regione critica; la sezione di codice che realizza questa richiesta è detta **sezione di ingresso**. La regione critica può essere seguita da una **sezione d'uscita** e il resto del codice viene detto **sezione/regione non critica.** Una soluzione al problema della regione critica per poter essere considerata tale deve rispettare le seguenti proprietà:

- **Mutua esclusione**, se il processo P_i è in esecuzione nella sua regione critica, nessun altro processo potrà accedervi;
- **Progresso**, se nessun processo è in esecuzione nella sua regione critica e uno o più processi ne richiedono l'accesso, solo i processi che **NON**

sono in regione critica parteciperanno alla decisione per stabilire quale sarà il prossimo processo ad accedere alla regione critica;
- **Attesa limitata**, ogni processo dopo aver richiesto l'accesso ad una regione critica dovrà attendere un tempo finito prima di richiedere nuovamente l'accesso. Durante questo tempo di attesa, altri processi potranno accedere una o più volte alla regione critica.

Gestire la regione critica nei sistemi monoprocessore single-core è semplice siccome si potrebbe fare in modo che quando si modifica una variabile condivisa il verificarsi delle interruzioni venga sospesa. In questo modo, siccome non vengono eseguite altre istruzioni sarà impossibile apportare modifiche inaspettate a una variabile condivisa. Il discorso è diverso per i sistemi multiprocessore e specialmente per i sistemi multicore siccome disabilitare un interrupt richiederebbe molto tempo poiché il messaggio deve essere passato a tutti i processori.

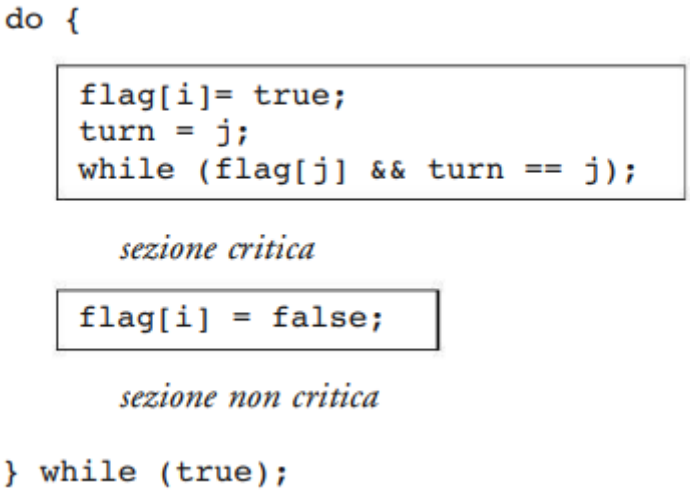
Le due strategie principali per la gestione delle regioni critiche nei sistemi operativi sono: **kernel con diritto di prelazione** e **kernel senza diritto di prelazione**. Nel kernel senza diritto di prelazione un processo in modalità kernel occupa la CPU fino alla sua terminazione; tuttavia, il kernel senza prelazione è privo di race condition siccome un solo processo per volta impiega il kernel. Nel kernel con diritto di prelazione, invece, non è consentito applicare a un processo in modalità kernel la prelazione. Questo tipo di kernel è preferibile al kernel senza diritto di prelazione siccome evita che un processo di sistema rimanga in esecuzione per troppo e fornisce una maggiore prontezza nelle risposte da parte del sistema (sistemi real-time).

SOLUZIONE DI PETERSON

Una delle soluzioni più comuni al problema della regione critica è la **soluzione di Peterson**. La soluzione di Peterson è limitata a due processi P_i e P_j.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

```
int turn;
boolean flag[2];
```



La variabile **turn** segnala di chi sia il turno per entrare nella regione critica; quindi, se turn == i, il processo P_i è autorizzato ad accedere alla propria regione critica. L'array **flag**, invece, indica quale processo è pronto ad accedere alla propria regione critica. Ad esempio, se flag[j] == true, allora P_j è pronto per accedere alla propria regione critica.

Per accedere alla propria regione critica P_i assegna flag[i] = true ed attribuisce turn = j in modo che P_j possa accedere alla sua regione critica. Quando i processi tentano un accesso contemporaneo, verrà assegnato a turn sia il valore di i sia il valore di j. In questo caso solo uno dei due permane mentre l'altro viene sovrascritto stabilendo così quale dei due processi dovrà accedere alla regione critica.

Dimostriamo la correttezza di questa soluzione:

- **Mutua esclusione preservata**. Ogni processo accederà alla sezione critica solo se flag[j] == false oppure se turn == i. Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche, allora flag[i] == flag[j] == true. Tuttavia, i due processi non eseguiranno mai la sezione critica nello stesso istante dato che turn può essere 0 o 1 ma non entrambi;
- **Progresso** e **attesa limitata soddisfatti**. L'ingresso di un processo P_i nella propria sezione critica può essere impedito solo se il processo è boccato nel ciclo **while**, con le condizioni flag[j] == true e turn == j. Qualora P_i non sia pronto ad entrare nella sezione critica, quindi flag[i] == false, allora P_j entrerà nella sezione critica. Al momento dell'uscita dalla sezione critica, P_j reimposta flag[j] = false consentendo a P_i di entrarvi. Il punto cruciale è che P_j non modificherà mai il valore di turn mentre P_i è nella sezione critica. Quindi P_i entrerà nella sezione critica

(**progresso**) solo dopo che P_j abbia effettuato non più di un ingresso (**attesa limitata**).

Il funzionamento della soluzione Peterson non è garantito sui calcolatori moderni in quanto, per migliorare le prestazioni del sistema i processi e/o compilatori possono riordinare operazioni di lettura e scritture che non hanno dipendenze. Per un'applicazione a singolo thread questo riordino è irrilevante per quanto riguarda la correttezza del programma, ma per un'applicazione multithread con dati condivisi il riordino dello istruzioni può portare a risultati incoerenti o inattesi.

SUPPORTO HARDWARE PER LA SINCRONIZZAZIONE

La soluzione di Peterson è una **soluzione basata sul software** perché l'algoritmo garantisce la mutua esclusione, il progresso e l'attesa limitata senza istruzioni hardware specifiche. Tuttavia, soluzioni di questo tipo non garantiscono il loro funzionamento su architetture moderne e a tal proposito vengono utilizzate delle istruzioni hardware specifiche che possono essere usate come strumenti di sincronizzazione o come basi per costruire meccanismi di sincronizzazione più astratti.

BARRIERE DI MEMORIA

Abbiamo visto che un sistema può riordinare le istruzioni e che ciò può portare a uno stato dei dati inaffidabile. Il **modello di memoria** è il modo in cui l'architettura di un computer determina quali garanzie relative alla memoria vengono fornite a un programma. Un modello di memoria può essere:

- **Fortemente accoppiato**, in cui una modifica alla memoria su un processore è immediatamente visibile agli altri processori;
- **Debolmente accoppiato**, in cui una modifica alla memoria su un processore potrebbe non essere immediatamente visibile agli altri processori;

I modelli di memoria variano in base al tipo di processore; quindi, per far fronte all'incognita del tipo di processore, le architetture forniscono istruzioni che forzano la propagazione di qualsiasi modifica in memoria a tutti i processori in modo tale che le modifiche siano visibili a tutti i thread su tutti i processori. Tali istruzioni sono note come **barriere di memoria** (**memory barrier**) o **recinzioni di memoria** (**memory fence**). Quando viene eseguita un'istruzione di barriera di memoria, il sistema garantisce che il completamento di tutte le istruzioni di load e store avvenga prima dell'esecuzione delle successive istruzioni di load e store. Pertanto, la barriera di memoria assicura che le operazioni di load e store siano visibili agli altri processori prima che vengano eseguite le operazioni di load e store future (a prescindere dal riordinamento).

Per la soluzione Peterson, si potrebbe inserire una barriera di memoria tra le prime due istruzioni di assegnamento della sezione per evitare il riordino delle istruzioni.

ISTRUZIONI HARDWARE

Molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria (**dato**), oppure di scambiare il contenuto di due parole di memoria in modo **atomico** – cioè come un'unità non interrompibile. Queste istruzioni sono utilizzabili per risolvere il problema della regione critica in maniera relativamente semplice. Siccome ogni architettura fornisce istruzioni specifiche, ci manterremmo su un livello astratto descrivendo le istruzioni **test_and_set()** e **compare_and_swap()/CAS**.

L'istruzione test_and_set() viene eseguita **atomicamente**; ciò vale a dire che se si eseguono contemporaneamente due istruzioni test_and_set(), su due core diversi, queste vengono eseguiti in modo sequenziali in un ordine arbitrario. La test_and_set() viene definita come segue.

```
boolean test_and_set(boolean *obiettivo){
    boolean valore = *obiettivo;
    *obiettivo = true;
    return valore;
}
```

Figura 5.3 Definizione dell'istruzione `test_and_set()`.

Per realizzare la mutua esclusione con la test_and_set() si può dichiarare una variabile booleana globale **lock** inizializzata a false ->

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

L'istruzione compare_and_swap() a differenza dell'istruzione test_and_set() utilizza tre operandi. L'operando **value** viene impostato a **new_value** solo se

l'espressione (*value == expected) è vera. A parte in questo caso, la compare_and_swap() restituisce sempre il valore originale della variabile value. Come l'istruzione di test_e_set(), la compare_and_swap() viene eseguita **atomicamente**.

```
int compare_and_swap(int *value, int expected,
    int new_value) {
       int temp = *value;
       if (*value == expected)
          *value = new_value;
      return temp;
}
```

La mutua esclusione può essere realizzata come segue. Viene dichiarata e inizializzata a 0 una variabile globale (lock). Il primo processo che richiama compare_and_swap() imposterà lock a 1. Entrerà poi nella sua sezione critica, poiché il valore originale di lock era pari al valore atteso 0. Le chiamate successive di compare_and_swap() non avranno successo, perché ora lock non è uguale al valore atteso 0. Quando un processo esce dalla sezione critica, imposta di nuovo lock al valore 0, per permettere a un altro processo di entrare nella propria sezione critica.

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* non fa niente */

          /* sezione critica */

     lock = 0;

          /* sezione non critica */
} while (true);
```

Gli algoritmi CAS sopra descritti non rispettano l'attesa di tempo limitata. L'algoritmo sottostante invece rispetta tutte e tre le proprietà richieste da una soluzione al problema della sezione critica.

```
do {
     waiting[i] = true;
     key = true;
     while (waiting[i] && key)
          key = test_and_set(&lock);
     waiting[i] = false;

          /* sezione critica */

     j = (i + 1) % n;
     while ((j != i) && !waiting[j])
          j = (j + 1) % n;

     if (j == i)
          lock = false;
     else
          waiting[j] = false;

          /* sezione non critica */
} while (true);
```

VARIABILI ATOMICHE

In genere, l'istruzione CAS **NON** viene utilizzata direttamente per fornire la mutua esclusione, bensì viene usata come base d'appoggio ad elementi di base per la costruzione di altri strumenti che risolvono il problema della sezione critica. Uno di questi elementi di base è la **variabile atomica**, che fornisce operazioni atomiche su tipi di dati di base come interi e booleani. Le variabili atomiche vengono comunemente utilizzate nei sistemi operativi e nelle applicazioni concorrenti per **garantire** la mutua esclusione in situazioni in cui potrebbe verificarsi una race condition su una singola variabile condivisa durante il suo aggiornamento (es. incremento di un contatore). L'uso delle variabili atomiche è spesso limitato ai singoli aggiornamenti di dati condivisi e generatori di sequenze.

LOCK MUTEX

Le soluzioni hardware (test_and_set & CAS) sono spesso complicate ed inaccessibili ai programmatori di applicazioni. In alternativa, i progettisti di sistemi operativi implementano **strumenti software** per risolvere lo stesso problema. Il più semplice di questi è il **lock mutex** che viene usato per proteggere le regioni critiche e prevenire le race conditions. In pratica un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica. La funzione acquire() acquisisce il lock e la funzione release() lo rilascia. Le due funzioni devono essere

eseguite **atomicamente** e per questo vengono spesso realizzate tramite uno dei meccanismi hardware descritti precedentemente (test_and_set & CAS).

```
acquire() {
   while (!available)
     ; /* attesa attiva */
   available = false;
}
```

```
release() {
  available = true;
}
```

Un lock mutex ha una variabile booleana available il cui valore indica se il lock è disponibile o meno. Se il lock è disponibile la chiamata di acquire() ha successo e il lock viene da questo momento considerato non disponibile. Un processo che tenta di acquisire un lock indisponibile viene bloccato fino al rilascio del lock.

Il principale svantaggio dell'implementazione che abbiamo fornito è che richiede **attesa attiva** (*busy waiting*): mentre un processo si trova nella sua sezione critica, ogni altro processo che cerca di entrare nella sezione critica deve ciclare continuamente effettuando la chiamata acquire(). Questo tipo di lock mutex è anche chiamato **spinlock**, perché il processo continua a "girare" (spin), in attesa che il lock diventi disponibile. Questo continuo ciclare è chiaramente un problema in un sistema multiprogrammato, dove una singola CPU è condivisa tra diversi processi. L'attesa attiva spreca cicli di CPU che qualche altro processo potrebbe utilizzare in modo produttivo. Tuttavia, gli spinlock hanno il vantaggio di non rendere necessario alcun cambio di contesto (operazione che può richiedere molto tempo) quando un processo deve attendere un lock e tornano quindi utili quando si prevede che i lock verranno trattenuti per tempi brevi. Gli spinlock sono spesso impiegati in sistemi multiprocessore in cui un thread può "girare" su un processore, mentre un altro thread esegue la sua sezione critica su un altro processore.

NOTA: mutex = **mutual exclusion**, mutua esclusione.

FRAGOLA: I lock possono essere contesi oppure no. Un lock è considerato conteso se un thread si blocca mentre tenta di acquisire il lock. Se un lock è disponibile mentre un thread cerca di acquisirlo, allora il lock è considerato non conteso. I lock contesi possono essere soggetti a una **contesa elevata** o **moderata**. I lock molto contesi riducono le prestazioni delle applicazioni concorrenti.

CILIEGINA: Gli spinlock sono spesso identificati come il meccanismo di locking da preferire sui sistemi multiprocessore quando il lock deve essere mantenuto per un **breve periodo**. Siccome l'attesa su un lock richiede **due** cambi di conteso, uno per spostare il thread allo stato di attesa e un altro per

ripristinarlo una volta che il lock diventa disponibile, per breve periodo s'intende una durata inferiore a due cambi di contesto.

SEMAFORI

I lock mutex rappresentano, generalmente, il più semplice degli strumenti di sincronizzazione tra processi. I **semafori** offrono metodi più complessi per la sincronizzazione.

Un **semaforo** S è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: wait() e signal().

```
wait(S) {
  while(S <= 0)
    ;//attesa attiva
  S--;
}
```

```
signal(S) {
  S++;
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni wait() e signal() si devono eseguire in modo **indivisibile**: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Nel caso della wait() si deve eseguire senza interruzioni anche l'operazione di controllo del semaforo (S <= 0).

USO DEI SEMAFORI

Si usa distinguere tra semafori **contatore**, il cui valore è un numero intero, e i semafori **binari**, il cui valore è limitato a 0 o 1. I semafori binari sono dunque simili ai lock mutex e vengono utilizzati al loro posto per la mutua esclusione nei sistemi dove i lock mutex non sono disponibili.

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di risorse disponibili. I processi che desiderano utilizzare una risorsa invocano la wait() sul semaforo, decrementandone così il valore (S--); i processi che restituiscono le risorse invece invocano la signal() incrementando il valore del semaforo (S++). Quando il semaforo vale 0, **tutte le risorse sono occupate**, e i processi che ne richiedono l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

IMPLEMENTAZIONE DEI SEMAFORI

Le definizioni delle operazioni sui semafori wait() e signal() appena descritte presentano lo stesso problema dei lock mutex, ovvero l'**attesa attiva**. Per superare la necessità dell'attesa attiva si possono modificare le definizioni delle operazioni wait() e signal() come segue: quando un processo invoca

l'operazione wait() e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può bloccare se stesso. L'operazione di **bloccaggio** pone il processo in una coda d'attesa associata al semaforo e pone lo stato del processo a **waiting**. Quindi, si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione.

Un processo **sospeso**/**bloccato**, che attende a un semaforo S, sarà riavviato in seguito all'esecuzione di un'operazione signal() su S da parte di qualche altro processo (rilascio della risorsa). Il processo si riavvia tramite un'operazione wakeup(), che modifica lo stato del processo da waiting a ready. Il processo entra nella coda dei processi pronti. (L'uso della CPU può essere o non essere commutato dal processo in esecuzione al processo appena divenuto pronto, a seconda del criterio di scheduling).

Il semaforo viene definito come segue:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

A ogni semaforo sono associati un valore intero (value) e una lista di processi contenente i processi in attesa a un semaforo; l'operazione signal() preleva un processo da tale lista e lo attiva.

La wait() e la signal() vengono modificate come segue:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            aggiungi questo processo a S->list;
            block();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            togli un processo P da S->list;
            wakeup(P);
            }
}
```

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, questa definizione può condurre a valori negativi. Se il valore del semaforo è negativo, il suo valore assoluto rappresenta il numero dei processi che attendono a quel semaforo. Ciò è dovuto all'inversione dell'ordine del decremento e della verifica nel codice dell'operazione wait(). La lista dei processi che attendono a un semaforo si può facilmente realizzare con un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero e un puntatore a una lista di PCB. Un modo per aggiungere e togliere processi dalla lista assicurando un'attesa limitata è usare una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo

elemento. In generale, tuttavia, si può usare qualsiasi criterio d'accodamento; il corretto uso dei semafori **NON** dipende dal particolare criterio adottato.

Siccome le operazioni della wait() e della signal() devono essere eseguite in modo atomico si deve garantire che nessuna coppia di processi le esegua in contemporaneo sullo stesso semaforo. Le soluzioni adottate variano in base all'architettura del sistema. Nei sistemi monoprocessori, durante l'esecuzione di wait() e signal(), vengono inibite le interruzioni. Nei sistemi multiprocessore, e in particolare nei sistemi multicore, questa soluzione si rivela molto complessa ed onerosa, e quindi i sistemi SMP **DEVONO** mettere a disposizione altre tecniche di realizzazione dei lock (es. CAS e spinlock). È importante rilevare che questa definizione delle operazioni wait() e signal() non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi.

NOTA: block() = sleep() nella wait() modificata.

CAPITOLO 9

INTRODUZIONE

La memoria è fondamentale nelle operazioni di un calcolatore. La memoria consiste di un grande vettore di byte, ognuno dotato di un proprio indirizzo. La CPU preleva le istruzioni in memoria basandosi sul contenuto del program counter; l'istruzione prelevata viene decodificata (ciò può comportare un ulteriore prelievo degli operandi adatti all'esecuzione dell'operazione) e poi viene eseguita tramite gli eventuali operandi. Ogni istruzione prelevata può comportare ulteriori operazioni di letture (**load**) o di scritture (**store**) in memoria.

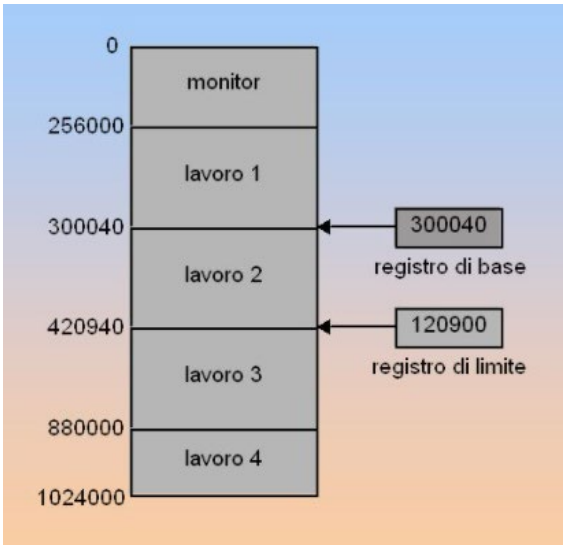
La memoria svolge un ruolo fondamentale anche per lo scheduling della CPU siccome, per migliorare le prestazioni della macchina, bisogna tenere in memoria un gran numero di processi e la memoria deve essere condivisa. La scelta degli algoritmi per la gestione della memoria varia in base all'architettura hardware del sistema siccome molti di questi algoritmi richiedono un supporto hardware per essere implementati.

HARDWARE DI BASE

La memoria centrale e i registri sono le uniche aree di memoria direttamente accessibili dalla CPU. I registri incorporati nella CPU sono accessibili in un **unico ciclo di clock** mentre, per accedere alla memoria centrale è necessario transitare sul bus di sistema. Questa transazione può richiedere molti cicli di clock e in questo caso la CPU entra in una fase di **stallo** (**stall**) poiché manca dei dati richiesti per completare l'istruzione corrente. Per risolvere questa

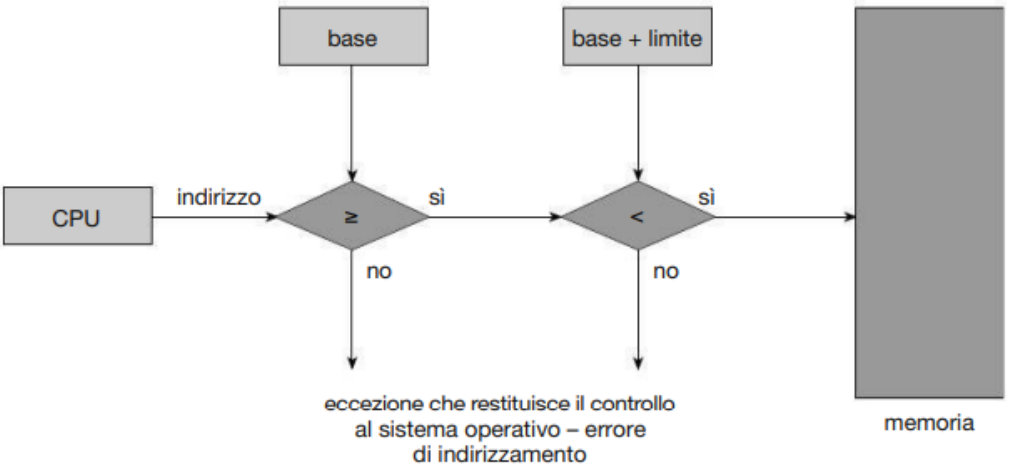
situazione intollerabile, si dispose una memoria veloce tra i registri e la memoria centrale il cui compito è quello di velocizzare gli accessi in memoria senza l'intervento del sistema operativo. Tale memoria si chiama **cache**.

Tuttavia, non basta prestare attenzione alla velocità di accesso alla memoria fisica, ma occorre anche assicura l'esecuzione corrette delle operazioni. A tal fine, bisogna proteggere il sistema operativo dall'accesso dei processi utenti e, in sistemi multithread, salvaguardare i processi utenti l'uno dall'altro. Innanzitutto, bisogna assicurarsi che ogni processo abbia uno **spazio di memoria separato**; ciò è fondamentale per avere più processi in memoria concorrentemente siccome in questa maniera ogni processo è protetto dagli altri. Uno dei metodi più comuni per separare gli spazi di indirizzamento dei processi è un meccanismo di protezione tramite due registri detti **registro di base** e **registro limite**.



Il registro di base contiene il più piccolo indirizzo legale della memoria fisica mentre, il registro limite determina la dimensione dell'intervallo ammesso. In parole povere, se il registro di base contiene l'indirizzo 256000 e il registro limite contiene 44000 allora il programma potrà accedere esclusivamente agli indirizzi compresi tra 256000 e 300039 (256000 + 44000 – 1), estremi inclusi.

Per mettere in atto il meccanismo di protezione, la CPU confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri. Se un processo tenta di accedere illegalmente ad un area riservata al sistema operativo o a un altro processo, viene generata un'eccezione (trap) che restituisce il controllo al sistema operativo che interpreta l'evento come un **errore fatale**.



Solo il sistema operativo può caricare i registri di base e i registri limite, grazie ad un'istruzione privilegiata. Tale istruzione, essendo privilegiata, può essere eseguita solo in modalità kernel. Grazie all'esecuzione in modalità kernel, il SO può accedere sia alla memoria a esso designata sia ad altre aree di memoria riservate; ciò gli consente di caricare i processi in memoria nelle aree a loro riservate, di copiare i contenuti di un area di memoria (dump) per scopi diagnostici, di modificare i parametri delle syscalls ed altro. Siccome l'esecuzione di tale istruzione è ristretta alla modalità kernel si impedisce così ai processi utente di alterare (volontariamente o involontariamente) i contenuti dei registri o di altre aree di memoria.

ASSOCIAZIONE DEGLI INDIRIZZI

In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Per essere eseguito, deve essere caricato in memoria ed inserito nel contesto di un processo. (Steps del Cap. 3).

La maggior parte dei sistemi consente ai processi utente di risiedere in qualsiasi parte della memoria fisica. Generalmente gli indirizzi del programma sorgente sono **simbolici** (es. variabile count -> count è un simbolo). Un compilatore, di norma, **associa** (**bind**) questi indirizzi simboli ad indirizzi **rilocabili** (es. "14 bytes a partire da tale modulo"). L'editor dei collegamenti (linkage editor), o il caricatore (loader), fa corrispondere questi indirizzi rilocabili a indirizzi assoluti (es. 74014). Ogni associazione rappresenta una corrispondenza da uno spazio d'indirizzi a un altro.

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in **qualsiasi** passo del seguente percorso:

- **Compilazione**. Se nella fase di compilazione si sa dove il processo risiederà in memoria si può generare **codice assoluto**; cioè se il processo inizia dalla locazione R allora anche il codice compilato

comincia da quella locazione. Se la locazione del processo cambia bisogna ricompilare il codice;

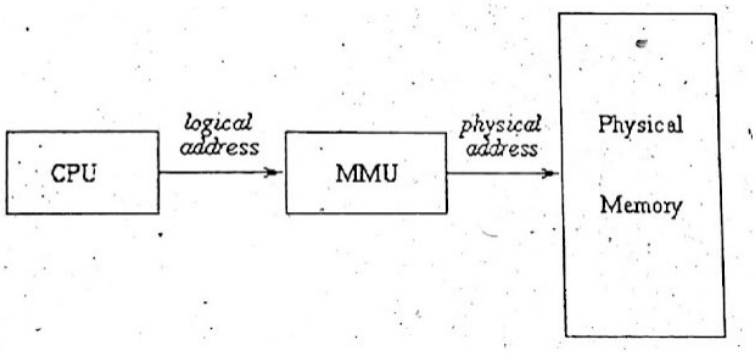
- **Caricamento**. Se nella fase di compilazione non è possibile sapere in che locazione risiederà il processo, il compilatore dovrà generare **codice rilocabile**. In questo caso l'associazione degli indirizzi finale è effettuata nella fase di caricamento. Se la locazione del processo cambia basta ricaricare il codice utente;
- **Esecuzione**. Se durante l'esecuzione il processo può essere spostato da un segmento di memoria ad un altro, bisogna ritardare l'associazione degli indirizzi assoluti alla fase di esecuzione. Per realizzare questo schema è necessario disporre di **hardware specializzato**. La maggior parte dei sistemi operativi general-purpose impiega questo metodo.

SPAZI DI INDIRIZZI LOGICI E FISICI

Un indirizzo generato dalla CPU è normalmente chiamato **indirizzo logico**, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel **registro d'indirizzo di memoria** (***memory address register***, **MAR**), è chiamato **indirizzo fisico**.

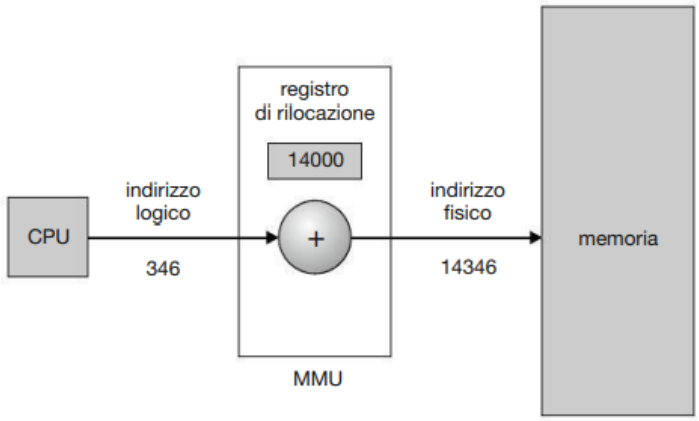
Gli indirizzi logici e fisici generati nelle fasi di compilazione e caricamento sono **identici**, ma lo stesso non vale per il metodo d'associazione in fase di esecuzione. In questo caso gli indirizzi logici vengono chiamati **indirizzi virtuali**. L'insieme degli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è detto **spazio degli indirizzi fisici**.

Con lo schema dell'associazione degli indirizzi in fase di esecuzione, lo spazio degli indirizzi logici differisce da quello degli indirizzi fisici. L'associazione in fase di esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo chiamato **unità della gestione della memoria** (***memory-management unit***, **MMU**).



Nell'MMU, il registro di base è denominato come **registro di rilocazione**: quando un processo utente genera un indirizzo, prima dell'invio all'unità di

memoria, si somma a tale indirizzo il valore contenuto nel registro di rilocazione. Per esempio, se il registro di rilocazione contiene il valore 14000, un tentativo di accesso da parte dell'utente alla locazione 346 viene dinamicamente rilocato alla locazione 14346.



Viene effettuato così un **mapping** tra lo spazio degli indirizzi logici e lo spazio degli indirizzi fisici. Il programma utente genera e tratta gli indirizzi logici e l'architettura del sistema li converte in indirizzi fisici. Esistono due tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a **max**) e gli indirizzi fisici (nell'intervallo da **r** + 0 a **r** + **max** per una certa base di r). Il programma utente pensa che il processo stia venendo eseguito negli indirizzi che vanno da 0 a max.

Gli indirizzi logici devono essere mappati in indirizzi fisici prima di essere usati. Il concetto di spazio degli indirizzi logici mappato sullo spazio degli indirizzi fisici è fondamentale per una corretta gestione della memoria.

CARICAMENTO DINAMICO

Nella discussione svolta finora era necessario che l'intero programma e i dati del processo fossero presenti in memoria affinché il processo potesse essere eseguito. Quindi, la dimensione del processo era limitata alla dimensione della memoria fisica. Per migliorare l'utilizzo della memoria si può ricorrere al **caricamento dinamico** (**dynamic loading**), mediante il quale si carica una procedura solo quando viene chiamata; tutte le procedure risiedono su disco in un formato di caricamento rilocabile. Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura ne invoca un'altra, controlla se è stata caricata. Se non è stata caricata viene chiamato il **linking loader** rilocabile per caricare in memoria la procedure appena richiesta e aggiornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. Una volta caricata, il controllo passa alla procedura richiesta.

Il vantaggio principale del caricamento dinamico è che una procedura viene caricata solo quando serve.

LINKING DINAMICO E LIBRERIE CONDIVISE

Le **librerie collegate dinamicamente** (**DLL**) sono librerie di sistema che vengono collegate ai programmi utente quando questi vengono eseguiti. Alcuni sistemi operativi consentono solo il **collegamento statico** (**static linking**), in cui le librerie di sistema sono trattate come qualsiasi altro modulo oggetto e combinato dal loader nel file binario del programma. Il concetto di linking dinamico, invece, è analogo a quello del caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento in cui è richieste, si differisce il collegamento. I vantaggi principali delle DLL sono: una ridotta dimensione dei files, i quali non sono più obbligati a tenere una copia della libreria all'interno dell'eseguibile, e possono essere **condivise** tra più processi, in modo che solo **un'istanza** della DLL sia presente in memoria. Per questo motivo le DLL sono anche conosciute come **librerie condivise**.

Le librerie collegate dinamicamente possono essere estese mediante aggiornamenti o anche sostituite da una nuova versione. In questo caso tutti i programmi che condividono la libreria utilizzeranno automaticamente la nuova versione. È possibile caricare in memoria più di una versione di una stessa libreria; i programmi identificheranno la versione da usare in base alle informazioni di cui dispongono. Senza il linking dinamico, ad ogni aggiornamento della libreria bisognerebbe ricollegare tutte i file binari.

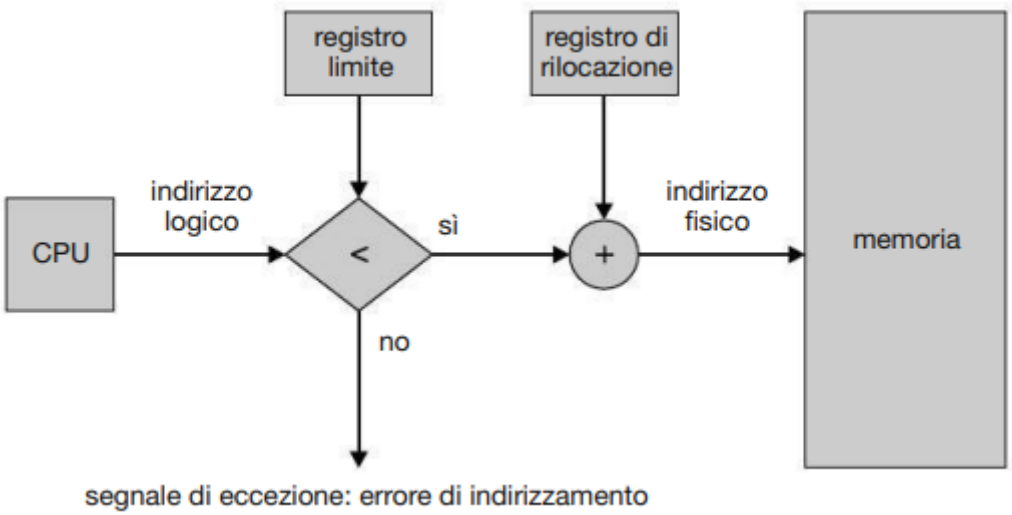
.ALLOCAZIONE CONTINGUA DELLA MEMORIA

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utente; perciò, è necessario assegnare le diverse parti della memoria nella maniera più efficiente possibile.

La memoria centrale di solito si divide in due partizioni: una per il sistema operativo che può essere collocato sia nella parte basse sia nella parte alta della memoria (la maggior parte dei sistemi lo alloca nella parte alta), e una per i processi utente. Siccome, in genere, si vuole che più processi utenti risiedano contemporaneamente in memoria è necessario stabilire come assegnare la memoria disponibile ai vari processi ma, prima di parlare di ciò è fondamentale decidere come proteggere la memoria.

PROTEZIONE DELLA MEMORIA

Per protezione della memoria si intende un qualsiasi meccanismo in grado di evitare che un processo acceda alla memoria che non gli appartiene. Tale meccanismo viene di norma implementato tramite l'MMU, quindi utilizzando il **registro di rilocazione** e il **registro di limite**.



Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni (**schema a partizione variabile**). Per esempio, il sistema operativo contiene codice e spazio di memoria per i driver dei dispositivi; se un driver non è in uso non ha senso mantenerlo in memoria. È meglio caricarlo quando serve e rimuoverlo quando non è necessario.

ALLOCAZIONE DELLA MEMORIA

I metodi di seguito descritti si basano sull'**allocazione contigua della memoria**, ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo.

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in **partizioni di dimensioni variabili**, dove ciascuna partizione può contenere esattamente un processo. Inizialmente, tutta la memoria è a disposizione dei processi utenti: si tratta di un grande blocco di memoria disponibile, un **buco** (hole). Nel lungo periodo la memoria contiene una serie di buchi di grosse dimensioni.

Quando i processi entrano nel sistema, il sistema operativo prende in considerazione i loro requisiti di memoria e la quantità di spazio di memoria disponibile e determina a quale processo allocare memoria. Quando gli viene assegnato uno spazio, il processo viene caricato in memoria e può competere per il controllo della CPU.

Quando non c'è sufficiente memoria per soddisfare i requisiti di un processo, le possibili soluzioni sono due: **fornire un messaggio di errore** o inserire il processo in una **coda di attesa**. Quando, in seguito, la memoria viene rilasciata, il sistema operativo controlla la coda di attesa per determinare se il **buco** che si è appena formato può soddisfare le richieste di un processo in attesa. Se il buco è troppo grande, viene diviso in **due** parti: una parte al processo in arrivo e l'altra si riporta nell'insieme dei buchi. Quando un processo termina, il blocco di memoria che occupava viene rilasciato e reinserito nell'insieme dei buchi e, se si trova accanto ad altri buchi, si uniscono tutti i blocchi di memoria adiacenti per formare un buco più grande.

I criteri più usati per scegliere un buco libero sono i seguenti:

- **First-fit**. Si assegna il **primo** buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme dei buchi sia dalla fine sia dal punto in cui era stata interrotta la ricerca precedente. Ci si ferma appena si incontra un buco sufficientemente grande.;
- **Best-fit**. Si assegna il **più piccolo** buco che può contenere il processo. La ricerca deve essere eseguita su tutta la lista, a meno che non sia ordinata in base alla dimensione. Le parti di buco inutilizzate sono più piccole.
- **Worst-fit.** Si assegna il buco **più grande**. Come per il best-fit, si deve esaminare tutta la lista. Si generano parti di buco inutilizzate più grandi.

Sia first-fit sia best-fit sono migliori rispetto al worst-fit in termini di risparmio di tempo e di utilizzo della memoria. Il first-fit, in genere, è più veloce del best-fit.

FRAMMENTAZIONE

Sia il first-fit sia il best-fit soffrono di **frammentazione esterna**. Caricando e rimuovendo i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna quando la memoria è "frammentata" in tante piccole parti **NON contigue**. Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera **sprecata** tra ogni coppia di processi. Una soluzione alla frammentazione esterna è la **compattazione**, disponibile solo se l'allocazione degli indirizzi è dinamica in fase di esecuzione. Quando si effettua un'operazione di compattazione è necessario calcolarne il costo. Il più

semplice algoritmo di compattazione consiste di spostare tutti i processi verso un lato della memoria in modo che dall'altro si raggruppino tutti i buchi che poi andranno a formare un unico buco. Questo metodo è molto oneroso. L'altro metodo usato consiste nel consentire la **non contiguità** dello spazio degli indirizzi logici di un processo, permettendo di assegnare la memoria fisica ovunque sia disponibile (usato nella **paginazione**).

La frammentazione oltre che esterna può essere **interna**. La frammentazione interna consiste nella memoria inutilizzata **all'interno di una partizione**. Si consideri uno schema a partizioni multiple con un buco di 18.464 bytes. Supponendo che il processo ne occupi 18.462 rimarrebbe un buco di 2 bytes. L'overhead necessario per tenere traccia di questo buco è più grande del buco stesso. Per risolvere questa problematica, di solito, si suddivide la memoria in blocchi di memoria di dimensione fissa; in questo modo la memoria assegnata può essere leggermente superiore rispetto alla memoria richiesta.

NOTA: **Regola del 50%**. Con l'algoritmo first-fit, per n blocchi assegnati si perdono 0.5 blocchi a causa della frammentazione rendendo inutilizzabile, nel caso peggiore, un terzo della memoria.

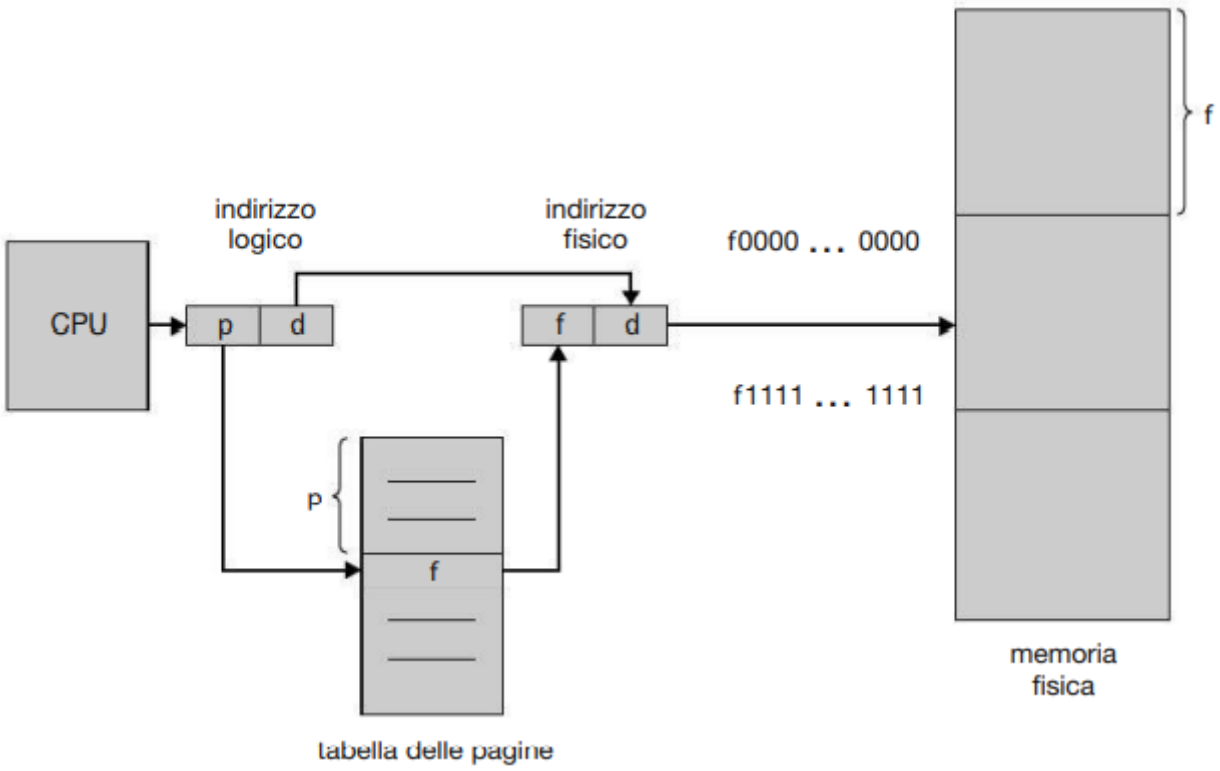
.PAGINAZIONE

La **paginazione** è uno schema per la gestione della memoria che consente allo spazio di indirizzamento fisico di un processo di non essere contiguo. La paginazione **evita** la frammentazione esterna e la necessità di compattazione e, nelle sue varie forme, è utilizzata nella maggior parte dei sistemi operativi. L'implementazione della paginazione è frutto della **collaborazione** tra il sistema operativo e l'hardware.

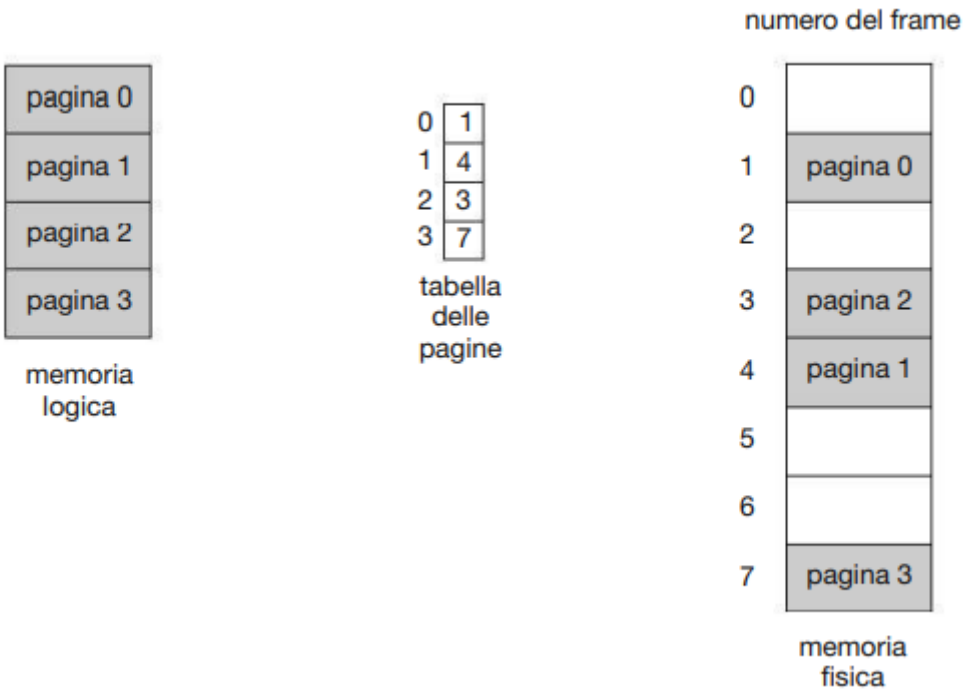
METODO DI BASE

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di **dimensione fissa**, detti **frame**, e nel suddividere la memoria logica in blocchi di pari dimensione, detti **pagine**. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria o dal file system. La memoria ausiliaria è divisa in blocchi di dimensione fissa pari a quella dei frame o da blocchi composti da più frame. In questo modo, lo spazio degli indirizzi logici è completamente separato dallo spazio degli indirizzi fisici.

Ogni indirizzo generato dalla CPU è diviso in due parti: un **numero di pagina** (p) e un **offset di pagina** (d). Il numero di pagina serve come indice per la tabelle delle pagine.



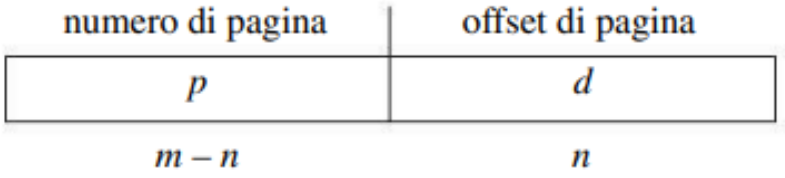
La tabella delle pagine contiene l'indirizzo di base di ciascun frame nella memoria fisica e l'offset indica la posizione nel frame a cui viene fatto riferimento. Quindi, l'indirizzo di base viene combinato con l'offset per definire l'indirizzo di memoria fisica.

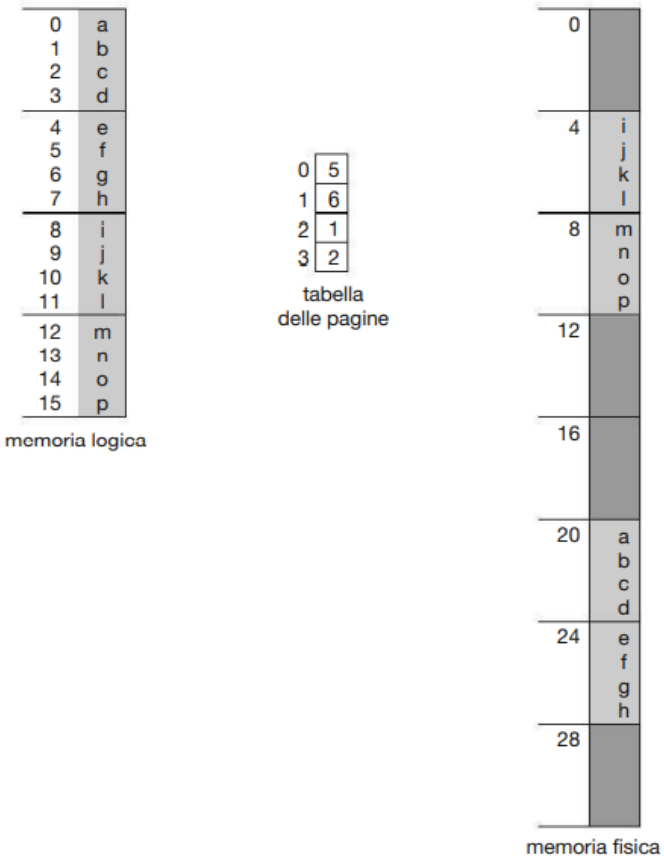


I passaggi adottati dalla MMU per tradurre un indirizzo logico in un indirizzo fisico sono i seguenti:

- Estrarre il numero di pagina ***p*** e utilizzarlo come indice nella tabella della pagine;
- Estrarre il numero del frame ***f*** corrispondente dalla tabella delle pagine;
- Sostituire il numero di pagina ***p*** nell'indirizzo logico con il numero di frame ***f***.

La dimensione di una pagina, così come quella di un frame, è definita dall'hardware ed è, in genere, una potenza di 2 compresa tra 4KB e 1GB. Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagine è 2^n bytes, allora gli **m** – **n** bit più significativi di un indirizzo indicano il numero di pagina, e gli **n** bit meno significativi indicano l'offset di pagina. L'indirizzo logico ha la seguente forma:





Come esempio concreto, anche se minimo, si consideri la memoria illustrata soprastante; qui, nell'indirizzo logico, n = 2 e m = 4. Utilizzando pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), vediamo come si fa corrispondere la memoria vista dal programmatore alla memoria fisica. L'indirizzo logico 0 è la pagina 0 con offset 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico 20 [= (5 × 4) + 0]. All'indirizzo logico 3 (pagina 0, offset 3) corrisponde l'indirizzo fisico 23 [= (5 × 4) + 3]. Per quel che riguarda l'indirizzo logico 4 (pagina 1, offset 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico 24 [= (6 × 4) + 0]. All'indirizzo logico 13 corrisponde l'indirizzo fisico 9. L'uso della tabella della pagina è simile all'uso di una tabella di registri di base/rilocazione, uno per ciascun frame.

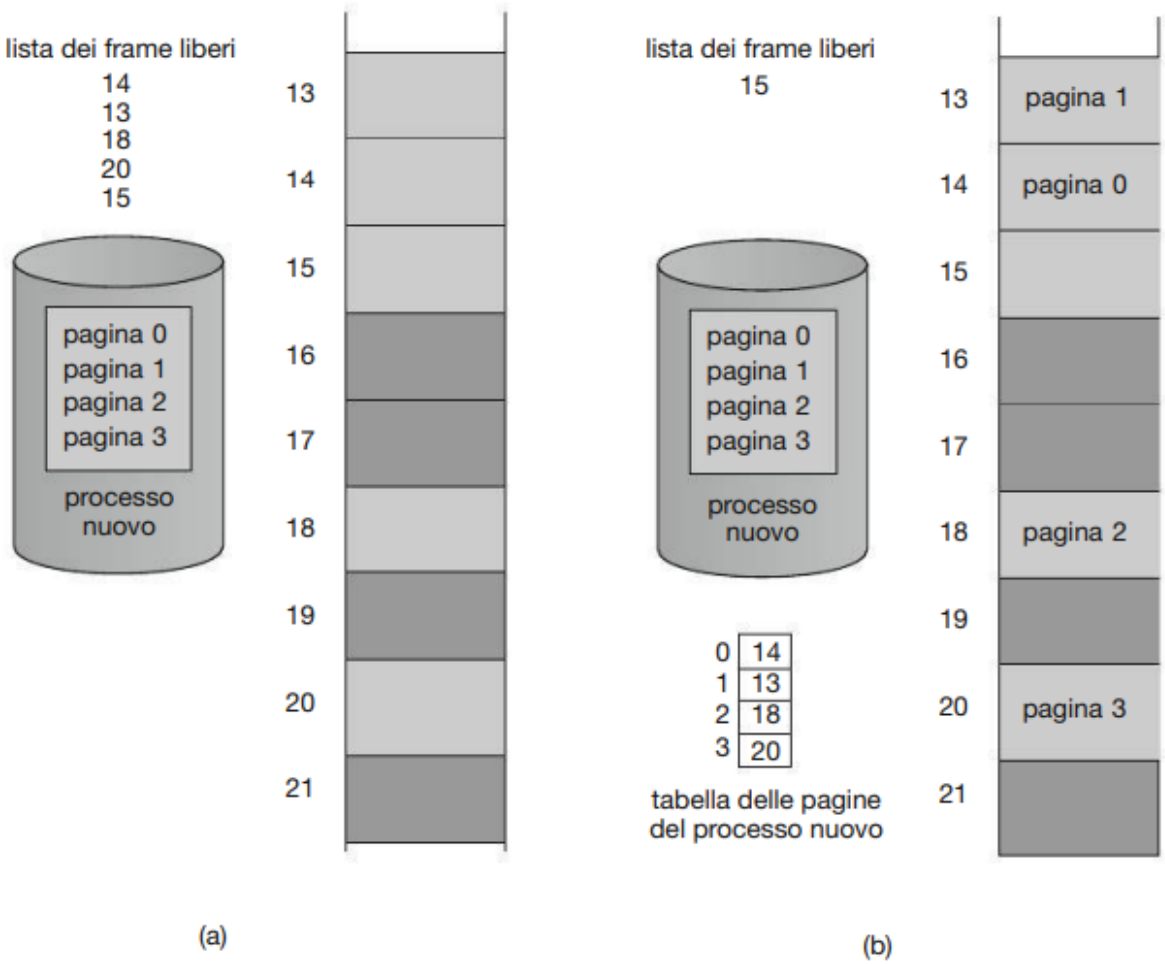
Con la paginazione si evita la frammentazione esterna: qualsiasi frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia, si può avere la frammentazione interna. I frame si assegnano come unità; poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, l'ultimo frame assegnato può non essere completamente pieno. Il caso peggiore si ha con un processo che necessita di

n pagine più un byte: si assegnano n + 1 frame; quindi, si ha una frammentazione interna di quasi un intero frame.

Se la dimensione del processo è indipendente dalla dimensione della pagina, si deve prevedere una frammentazione interna media di mezza pagina per processo. Questa considerazione suggerisce che conviene usare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un overhead che si riduce all'aumentare delle dimensioni delle pagine. Inoltre, con un maggior numero di dati da trasferire, l'I/O su disco è più efficiente. Alcune CPU e alcuni kernel gestiscono anche diverse dimensioni di pagine.

Spesso, su una Cpu a 32 bit, ogni voce della tabella delle pagine è lunga 4 byte, ma questa dimensione può variare. Una singola voce di 32 bit può puntare a uno dei 2^{32} frame di pagina fisici. Se la dimensione di un frame è di 4 kb (2^{12}), un sistema con voci della tabella delle pagine di 4 byte può indirizzare 2^{44} byte (o 16 Tb) di memoria fisica. Un sistema con voci di 32 bit è quindi in grado di indirizzare una quantità di memoria fisica **inferiore** al valore massimo teoricamente possibile. La paginazione ci consente di utilizzare una memoria fisica che è decisamente più grande rispetto a quella che potrebbe essere indirizzata dalla lunghezza del puntatore agli indirizzi della CPU.

Quando si deve eseguire un processo, il sistema esamina la sua dimensione espressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede n pagine, devono essere disponibili almeno n frame che, se ci sono, vengono assegnati al processo. Si carica la prima pagina in uno dei frame assegnati e s' inserisce il numero del frame nella tabella delle pagine relativa al processo in questione. La pagina successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella tabella delle pagine, e così via.



Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei dettagli della allocazione: quali frame sono assegnati, quali sono disponibili, il loro numero totale, e così via. In genere queste informazioni sono contenute in una struttura dati chiamata **tabella dei frame**, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

Inoltre, il sistema operativo deve sapere che i processi utenti operano nello spazio utente, e tutti gli indirizzi logici si devono far corrispondere a indirizzi fisici. Se un utente usa una chiamata di sistema (per esempio per eseguire un'operazione di I/O) e fornisce un indirizzo come parametro (per esempio l'indirizzo di un buffer), si deve tradurre questo indirizzo nell'indirizzo fisico corretto. Il sistema operativo conserva una copia della tabella delle pagine per ciascun processo, così come conserva una copia dei valori contenuti nel contatore di programma e nei registri. Questa copia si usa per tradurre gli indirizzi logici in indirizzi fisici ogni volta che il sistema operativo deve associare esplicitamente un indirizzo fisico a un indirizzo logico. La stessa copia è usata anche dal dispatcher della CPU per impostare l'hardware di paginazione

quando a un processo sta per essere assegnata la CPU. **La paginazione, quindi, aumenta la durata dei cambi di contesto**.

SUPPORTO HARDWARE ALLA PAGINAZIONE

Poiché ogni processo ha la sua tabella delle pagine, un puntatore alla tabella delle pagine viene memorizzato insieme ai valori di altri registri all'interno del **process control block** del processo. Quando lo scheduler della CPU seleziona nuovamente il processo, deve ricaricare i registri utente e i valori appropriati della tabella delle pagine dalla tabella delle pagine utente memorizzata.

L'implementazione hardware della tabella delle pagine si può realizzare in vari modi. Il metodo più semplice è quello di utilizzare registri dedicati ad alta velocità; questo metodo aumenta il tempo dei cambi di contesto siccome durante un cambio di contesto bisogna scambiare ogni registro.

L'uso dei registri per la tabella delle pagine rimane comunque un metodo efficiente nel caso in cui la tabella è di piccole dimensioni (es. 256 elementi). Tuttavia, la maggior parte degli elaboratori moderni usa tabelle molto grandi (es. 1 milione di elementi); dunque, la tabella delle pagine viene mantenuta in memoria e un **registro di base della tabella delle pagine** (**page-table base register**, **PTBR**) punta alla tabella stessa. Il cambio di contesto risulta molto più veloce rispetto a quello dei registri siccome bisogna modificare solo il valore del PTBR.

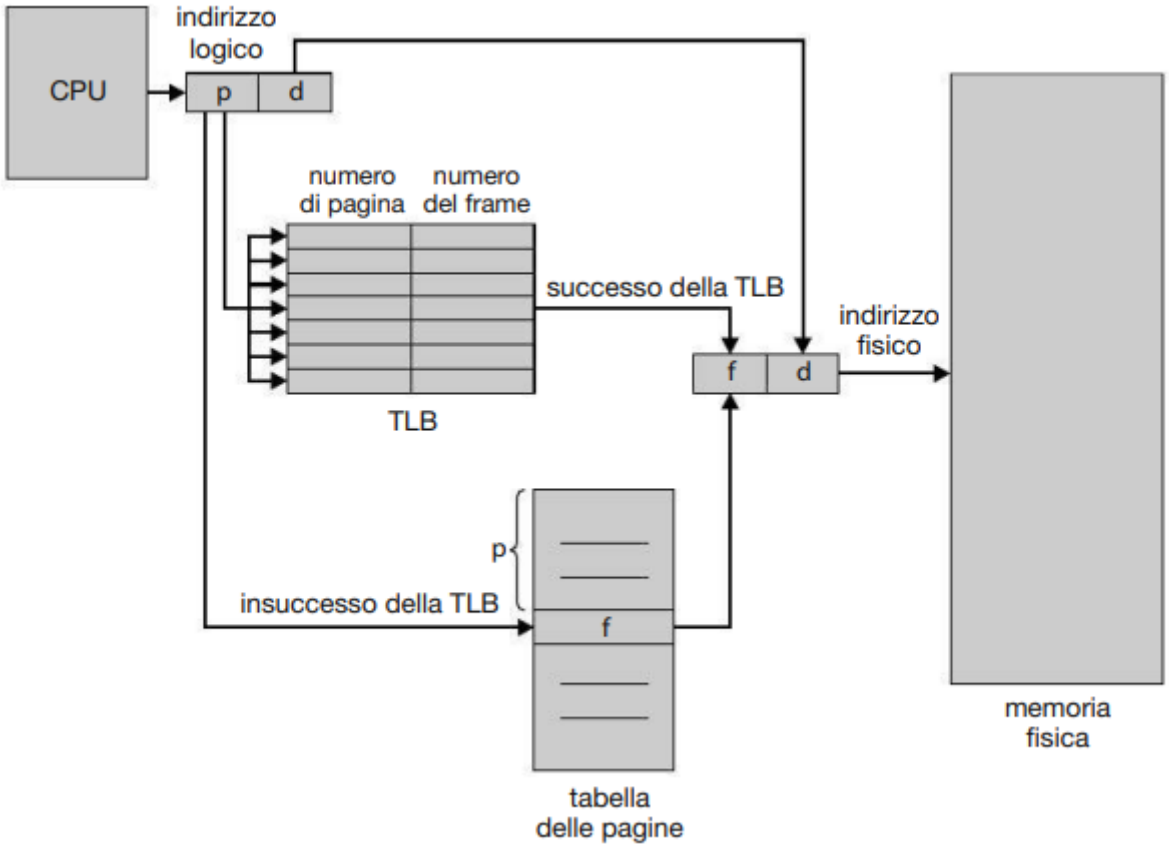
TLB

La memorizzazione della tabella delle pagine in memoria può favorire cambi di contesto più rapidi ma, al contempo, può rallentare gli accessi in memoria. Supponiamo di voler accedere alla posizione **i**. Per farlo, occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR aumentato dell'offset relativo alla pagina di **i** e questo comporta un accesso alla memoria. Così facendo si ottiene il numero del frame che, associato all'offset rispetto all'inizio della pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione in memoria. Con questo metodo sono necessari **due** accessi alla memoria per l'accesso ai dati, ritardando di un fattore due i tempi di accesso alla memoria.

La soluzione più tipica a questo problema consiste nell'impiego del **TLB** (**translation look-aside buffer**), una speciale piccola cache hardware. Il TLB

è una **memoria associativa** ad alta velocità in cui ogni elemento consiste di due parti: **una chiave** (**tag**) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi e, se trova una corrispondenza, restituisce il valore. La ricerca è molto più rapida. Per mantenere tempi di ricerca veloci è necessario che il TLB sia di ridotte dimensioni (di norma contiene tra le 32 e le 1024 voci) e, per tale motivo, varie CPU implementano due TLB, una per le istruzioni e una per i dati, in modo da raddoppiare il numero di voci disponibili. **Il TLB viene implementato come parte della pipeline, quindi non comporta nessuna penalizzazione sulle prestazioni del sistema**.

Il TLB viene usato **insieme** alla tabella delle pagine nel seguente modo: il TLB contiene una piccola parte degli elementi della tabella delle pagine; quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina al TLB; se tale numero è presente nel TLB, il corrispondente numero di frame è immediatamente disponibile e si usa per accedere alla memoria. Nel caso in cui il numero non sia presente nel TLB (**TLB miss – insuccesso del TLB**) si deve consultare la tabella delle pagine in memoria. Il numero di frame e di pagina così ottenuti vengono inseriti nel TLB e, al riferimento successivo la ricerca sarà più rapida.



Se il TLB è già pieno, si deve scegliere un elemento da sostituire. I criteri di scelta variano dalla scelta dell'elemento meno usato recentemente (**LRU**), a una politica round-robin, fino alla scelta casuale. Alcuni TLB consentono che certi elementi siano **vincolati** (**wired-down**), ovvero non rimovibili; in genere si vincolano gli elementi per il codice chiave del kernel.

Alcune TLB memorizzano gli identificatori dello spazio d'indirizzi (address-space identifier, **ASID**) in ciascun elemento della TLB. Un ASID identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, la TLB si assicura che l'ASID per il processo attualmente in esecuzione corrisponda all'ASID associato alla pagina virtuale. La mancata corrispondenza dell'ASID viene trattata come un TLB miss. Oltre a fornire la protezione dello spazio d'indirizzi, l'ASID consente che la TLB contenga nello stesso istante elementi di diversi processi. Se la TLB non permette l'uso di ASID distinti, ogni volta che si seleziona una nuova tabella delle pagine, per esempio a ogni cambio di contesto, si deve **cancellare** (**flush**) la TLB, in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione.

La percentuale di volte che il numero di pagina di interesse si trova nella TLB è detta tasso di successi (hit ratio). un tasso di successi dell'80 per cento significa che il numero di pagina desiderato si trova nella TLB nell'80 per cento dei casi. Se sono necessari 100 nanosecondi per accedere alla memoria, allora un accesso alla memoria mappata nella TLB richiede 100 nanosecondi. Se, invece, il numero non è contenuto nella TLB, occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); in totale sono necessari 200 nanosecondi. Stiamo in questo caso supponendo che una ricerca nella tabella delle pagine richieda un solo accesso alla memoria, ma, come vedremo in seguito, potrebbero talvolta essere necessari più accessi. Per calcolare il **tempo effettivo d'accesso alla memoria** occorre tener conto della probabilità dei **due casi**:

tempo effettivo d'accesso = 0,80 × 100 + 0,20 × 20 = 120 nanosecondi.

In questo esempio si verifica un rallentamento del 20% nel tempo medio d'accesso alla memoria. Per un tasso di successi del 99%, caso più realistico, si ottiene:

tempo effettivo d'accesso = 0,99 × 100 + 0,01 × 20 = 10,1 nanosecondi.

Con questo tasso di successi, il rallentamento del tempo d'accesso alla memoria scende all'1%.

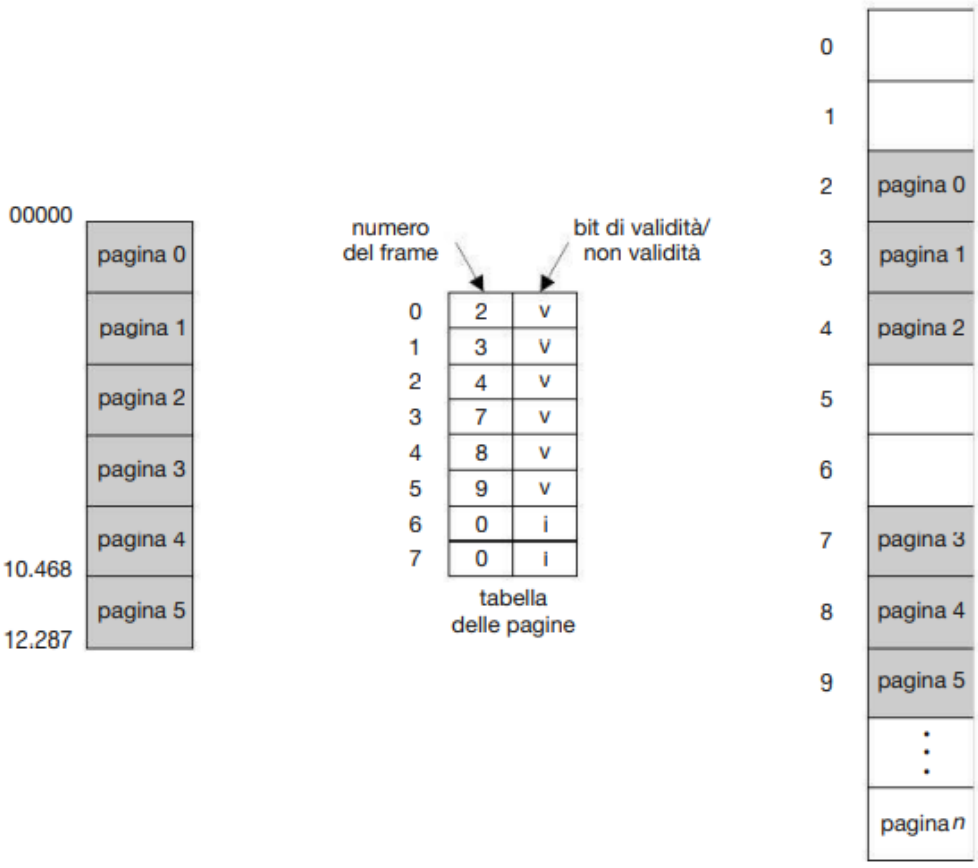
Il calcolo dei tempi di accesso sulle CPU moderne risulta più complicato di quello appena visto siccome, la maggior parte delle CPU contiene più di un TLB.

PROTEZIONE

In un ambiente paginato, la protezione della memoria è assicurata dai **bit di protezione** associati ad ogni frame; normalmente i bit si trovano nella tabella delle pagine.

Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Tutti i riferimenti alla memoria passano attraverso la tabella delle pagine per trovare il numero corretto del frame; quindi, mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura.

Questo metodo si può facilmente estendere per fornire un livello di protezione più perfezionato. Si può progettare un hardware che fornisca la protezione di sola lettura, di sola scrittura o di sola esecuzione. In alternativa, con bit di protezione distinti per ogni tipo d'accesso, si può ottenere una qualsiasi combinazione di tali tipi d'accesso; i tentativi illegali causano la generazione di un'eccezione per il sistema operativo. Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **bit di validità**. Tale bit, impostato a **valido**, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo; quindi, è una pagina valida; impostato a **non valido**, indica che la pagina non è nello spazio d'indirizzi logici del processo. Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un'eccezione. Il sistema operativo concede o impedisce l'accesso a una pagina impostando in modo appropriato tale bit.



Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi; infatti, molti processi sfruttano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella delle pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una grande parte di tale tabella resta inutilizzata. Alcune architetture dispongono di registri, detti **registri di lunghezza della tabella delle pagine** (**page-table length register**, **PTLR**) per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa la generazione di un'eccezione per il sistema operativo.

PAGINE CONDIVISE

Un vantaggio della paginazione risiede nella possibilità di **condividere codice comune** tra più processi. Si consideri ad esempio la libreria standard del C **libc**, che fornisce parte dell'interfaccia alle chiamate di sistema in molte versione di UNIX e Linux. Su un tipico sistema Linux, la maggior parte dei processi utenti richiede libc. Un'opzione sarebbe quella di fare in modo che ogni processo carichi nel proprio spazio di indirizzamento la propria copia di libc; in questa maniera si occuperebbe un grande quantitativo di spazio senza alcun motivo specifico.

Se il codice è ricorrente (non auto-modificante) può essere condiviso tra più processi. Ogni processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari per la propria esecuzione. Solo una copia di libc viene conservata in memoria fisica e la tabella delle pagine di ogni processo utente viene mappata sulla stessa copia fisica di libc.

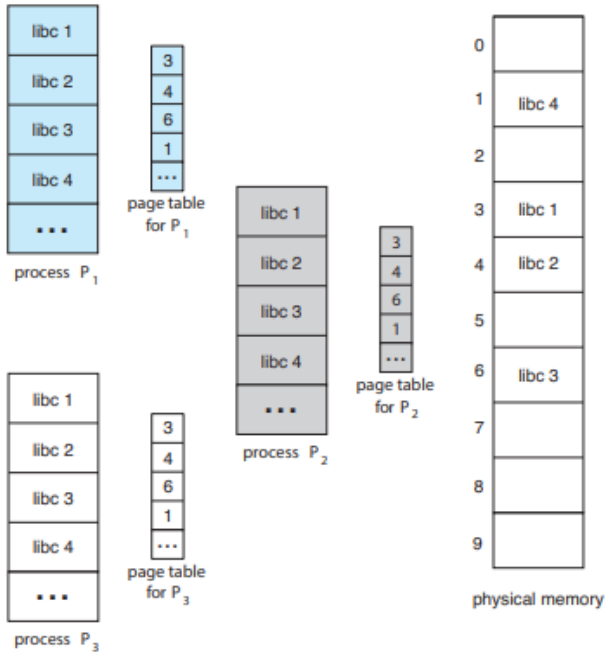


Figure 9.14 Sharing of standard C library in a paging environment.

Oltre alle librerie run-time altri programmi d'uso frequente possono essere condivisi: compilatori, interfacce a finestra, sistemi di database e così via.

La condivisione della memoria tra più processi è simile al modo in cui i thread condividono lo spazio d'indirizzi di una task, inoltre alcuni sistemi operativi realizzano l'IPC a memoria condivisa impiegando le pagine condivise.

.STRUTTURA DELLA TABELLA DELLE PAGINE

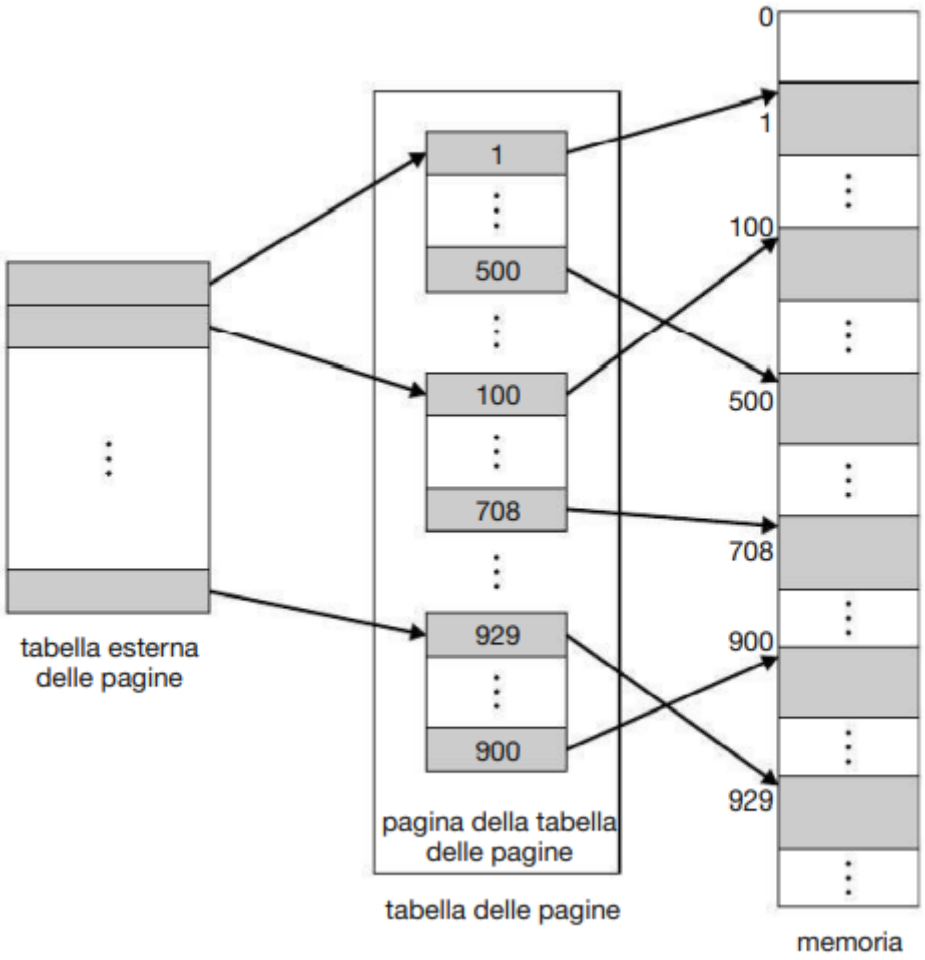
Di seguito sono riportate alcune delle tecniche più comuni per strutturare le tabella delle pagine: **paginazione gerarchica**, **la tabella delle pagine di tipo hash** e **la tabella delle pagine invertita**.

PAGINAZIONE GERARCHICA

La maggior parte dei calcolatori moderni dispone di uno spazio d'indirizzi logici molto grande (da 2^{32} a 2^{64} elementi). In un ambiente di questo tipo la stessa tabella delle pagine diventa eccessivamente grande. In un sistema a 32 bit,

assumendo che la dimensione di ciascuna pagina sia di 4KB ed ogni elemento consiste di 4byte, ogni processo potrebbe arrivare ad allocare una tabella delle pagine di 4MB. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole: questo risultato si può ottenere in vari modi.

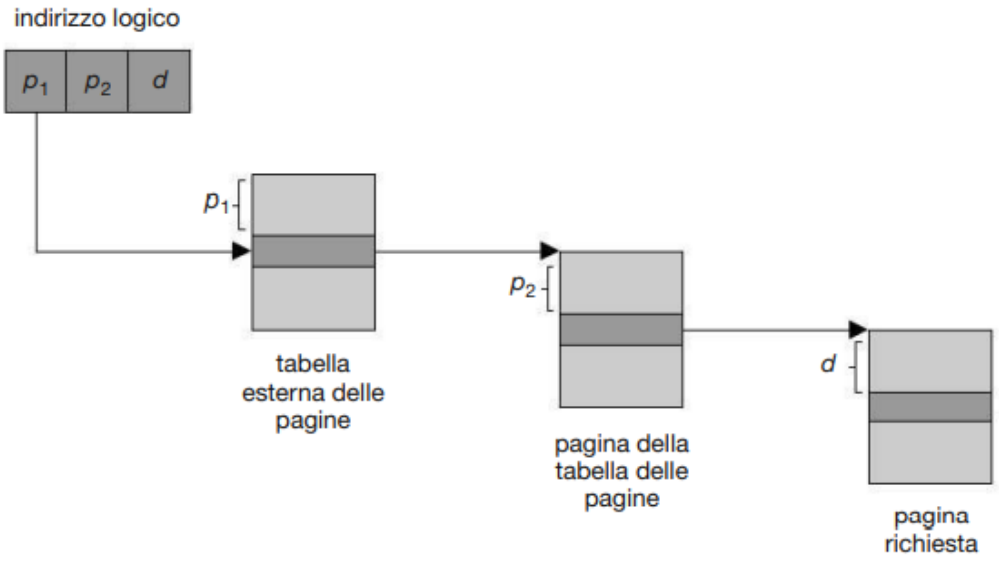
Un metodo consiste nell'adottare un algoritmo di **paginazione a due livelli**, in cui la stessa tabella è **paginata**.



Continuando sull'esempio precedente, ogni indirizzo logico è suddiviso in un numero di pagina di 20bit e un offset di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e un offset di pagina di 10 bit. L'indirizzo logico, nella paginazione gerarchica a due livelli, è strutturato come segue:

numero di pagina		offset di pagina
p_1	p_2	d
10	10	12

dove p_1 è un indice della tabella delle pagine di **primo livello** o tabella **esterna** delle pagine, e p_2 è l'offset all'interno della pagina indicata dalla tabella esterna. La traduzione degli indirizzi si svolge dalla tabella esterna alla tabella interna come di seguito riportato:



Questo metodo è anche noto come **tabella delle pagine ad associazione diretta** (**forward-mapped page table**).

Lo schema di paginazione a due livelli non è adatto nel caso di sistema a 64 bit poiché si necessiterebbe di più livelli per formare una tabella delle pagine di dimensioni accettabili. Ad esempio, l'Ultra-SPARC a 64 bit richiederebbe sette livelli di paginazione.

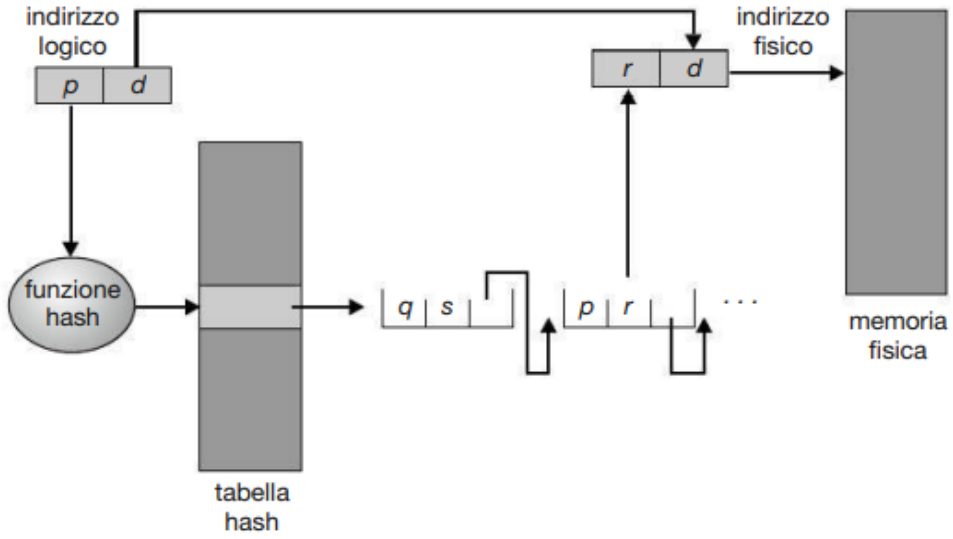
TABELLA DELLE PAGINE DI TIPO HASH

Un metodo molto comune nella gestione degli spazi d'indirizzi oltre i 32 bit consiste nell'utilizzo di una **tabella delle pagine di tipo hash**, in cui l'argomento della funzione hash è il numero della pagina virtuale. Per la gestione delle collisioni, ciascun elemento contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da **tre campi**:

1. **Numero di pagina virtuale**;
2. **Indirizzo del frame corrispondente al numero della pagina virtuale**;
3. **Un puntatore all'elemento successivo della lista**.

L'algoritmo opera nel seguente modo: si applica la funzione hash al numero della pagina virtuale identificando un elemento della tabella. Si confronta il numero di pagina virtuale col campo 1 primo elemento della lista concatenata corrispondente. Se i due corrispondono, si usa il campo 2 per generare

l'indirizzo fisico. Altrimenti, si esaminano allo stesso modo gli elementi successivi della lista concatenata.

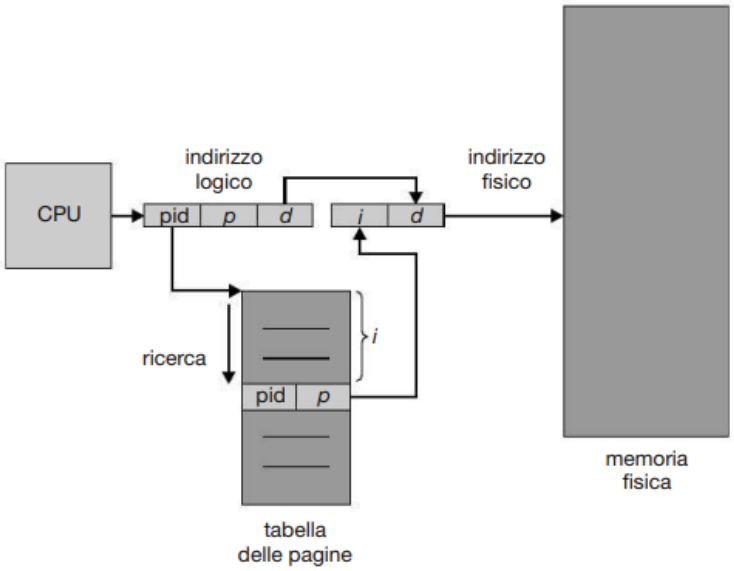


Per questo schema è stata proposta una variante adatta agli spazi di indirizzamento a 64 bit. Questa variante viene chiamata **tabella delle pagine a gruppi** (**clustered page table**) simile alla tabella delle pagine di tipo hash; la differenza è che ciascun elemento della tabella hash contiene i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue. La tabella delle pagine a gruppi è particolarmente utile per gli spazi d'indirizzi **sparsi** in cui la memoria non è allocata in modo contiguo.

TABELLA DELLE PAGINE INVERTITA

Un altro metodo per evitare di allocare per ogni processo una tabella delle pagine troppo grande, è la **tabella delle pagine invertita**.

Una normale tabella delle pagine contiene un elemento per ciascun indirizzo logico, a prescindere che questo sia valido o meno. Una tabella delle pagine invertita, come suggerisce il nome, invece, contiene un elemento per ogni frame. La tabella delle pagine invertita richiede spesso la memorizzazione di un identificatore dello spazio d'indirizzi in ciascun elemento della tabella delle pagine. Solitamente quest'identificatore corrisponde al **process-id** del processo di riferimento a quel determinato spazio d'indirizzamento. L'identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente. Esempi di sistemi che usano le tabelle delle pagine invertite sono l'Ultra-SPARC a 64 bit e il PowerPC.



Per illustrare questo metodo descriviamo una versione semplificata della tabella delle pagine invertita dell'IBM RT. IBM è stata la prima grande azienda a utilizzare le tabelle delle pagine invertite, a cominciare dal System 38, passando per il sistema RS/6000, fino alle attuali CPU IBM power. Nell'IBM RT ciascun indirizzo virtuale è una tripla del tipo seguente:

<id-processo, numero di pagina, offset>

Ogni elemento della tabella delle pagine invertita è una coppia **<id-processo, numero di pagina>** dove l'id-processo assume il ruolo di identificatore dello spazio d'indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell'indirizzo virtuale, formato da <id-processo, numero di pagina>, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull'elemento i, si genera l'indirizzo fisico. In caso contrario è stato tentato un **accesso a un indirizzo illegale**.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferimento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza potrebbe essere necessario esaminare tutta la tabella. Per ridurre l'entità del problema, si può impiegare una tabella hash che riduce la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria; quindi, un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Nei sistemi che adottano

le tabelle delle pagine invertite, l'implementazione della memoria condivisa è difficoltosa. Difatti, la condivisione si realizza solitamente tramite indirizzi virtuali multipli (uno per ogni processo che partecipa alla condivisione) associati a un unico indirizzo fisico. Questo metodo però non può essere adottato in presenza di tabelle invertite, perché, essendovi un solo elemento indicante la pagina virtuale corrispondente a ogni pagina fisica, questa non può avere più di un indirizzo virtuale associato. una semplice tecnica per superare il problema consiste nel porre nella tabella delle pagine una sola associazione fra un indirizzo virtuale e l'indirizzo fisico condiviso; ciò comporta un errore dovuto all'assenza della pagina (**page fault**) per ogni riferimento agli indirizzi virtuali non associati.

ORACLE SPARC SOLARIS

Consideriamo come ultimo esempio lo SPARC Solaris, un sistema elaborativo a 64 bit che ha lo scopo di fornire una memoria virtuale a basso overhead. Il Solaris, essendo un sistema operativo a 64 bit, deve risolvere il problema della memoria virtuale senza esaurire tutta la sua memoria fisica. L'approccio utilizzato da Solaris comporta l'uso di **due tabelle hash**: una per i processi utenti e una per il kernel. Ogni elemento della tabella hash rappresenta un'area contigua di memoria virtuale mappata, il che è più efficiente rispetto ad avere voci separate per ciascuna pagina. Ogni voce ha un **indirizzo di base** e un **intervallo** (**span**) che indica il numero di pagine rappresentate da quella voce.

Per ridurre il tempo necessario alla traduzione degli indirizzi, la CPU implementa un TLB che contiene le voci della tabella di traduzione (**TTE**) in modo da offrire ricerche hardware più rapide. Una cache di queste TTE risiede in un buffer di memoria di traduzione (**TSB**), che include una voce per ogni pagina di recente accesso. In caso di riferimento a un indirizzo virtuale, l'hardware interroga la TLB per una traduzione. Se non vengono trovate traduzioni, l'hardware scorre il buffer TSB in cerca della TTE che corrisponde all'indirizzo virtuale che ha causato la ricerca. Questa funzionalità è chiamata **TLB walk**. Se viene trovata una corrispondenza nel TSB, la CPU copia la voce TSB nella TLB, e la traduzione viene completata. Se invece non viene trovata alcuna corrispondenza, viene interrotto il kernel per effettuare una ricerca nella tabella hash. Il kernel crea quindi una TTE dalla tabella hash appropriata e la memorizza nel TSB affinché l'unità di gestione della memoria della CPU possa effettuare il caricamento automatico nella TLB. Infine, il gestore di interrupt restituisce il controllo alla MMU, che completa la conversione dell'indirizzo e recupera il byte o la parola richiesta dalla memoria principale.

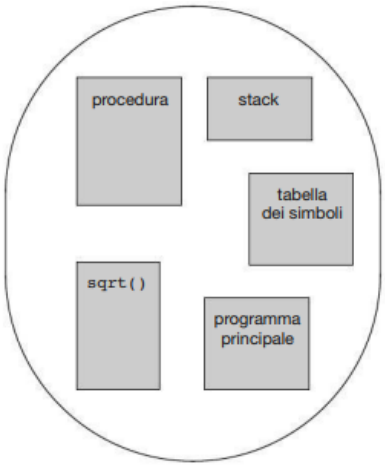
.SEGMENTAZIONE

Come abbiamo già detto, l'immagine che un utente ha della memoria non coincide con la memoria fisica. Lo stesso si può dire sulla rappresentazione

della memoria dal punto di vista del programmatore. Lavorare sulla memoria in termini di caratteristiche fisiche è scomodo sia per il sistema operativo sia per il programmatore. Che cosa si otterrebbe se l'hardware potesse fornire un meccanismo di memoria in grado di mappare il punto di vista del programmatore sulla memoria fisica? Il sistema avrebbe più libertà nella gestione della memoria e il programmatore avrebbe a disposizione un ambiente di programmazione più naturale. Un tale meccanismo ci è fornito dalla segmentazione.

METODO DI BASE

Ci si potrebbe chiedere se il programmatore pensi alla memoria come a un array lineare di byte, alcuni dei quali contengono istruzioni e altri dati. Molti programmatori risponderebbero di no. I programmatori la vedono piuttosto come un insieme di segmenti di dimensione variabile che non hanno fra loro un ordinamento particolare.



La **segmentazione** è uno schema di gestione della memoria che supporta questa rappresentazione della memoria dal punto di vista del programmatore. uno spazio d'indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza.

Gli indirizzi specificano sia il nome sia l'offset all'interno del segmento; quindi, il programmatore fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e un offset. Per semplicità di implementazione, i segmenti sono numerati, e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi, un indirizzo logico è una coppia:

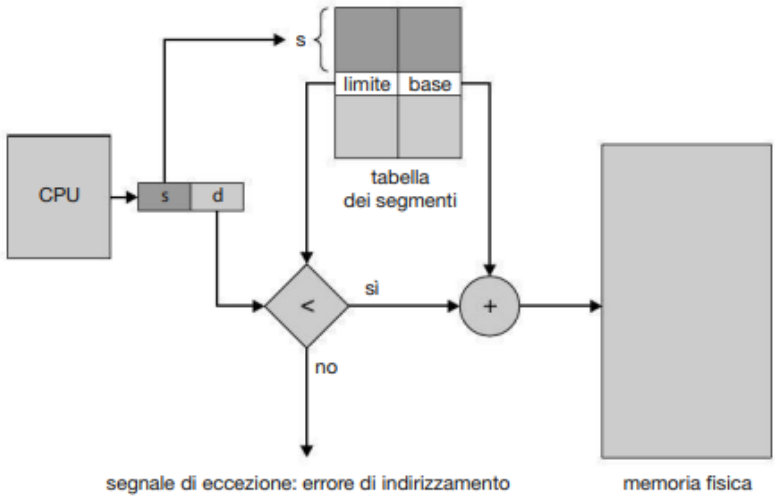
<numero segmento, offset>

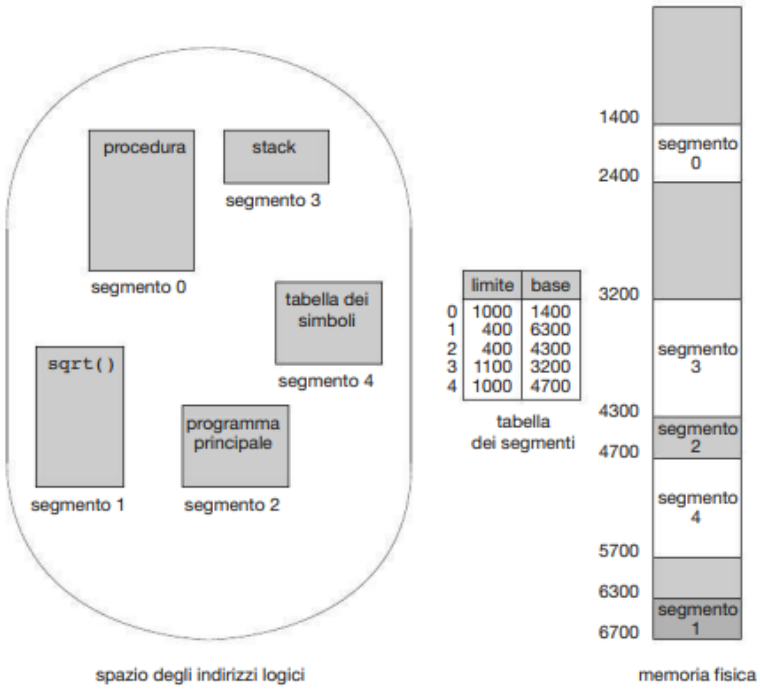
Normalmente quando un programma viene compilato il compilatore costruisce automaticamente i segmenti in rapporto al programma sorgente. Alle librerie collegate dal linker al momento della compilazione possono essere assegnati

dei segmenti separati. Il loader preleva questi segmenti e assegna loro i numeri di segmento.

HARDWARE DI SEGMENTAZIONE

Sebbene il programmatore possa far riferimento agli oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una **tabella dei segmenti**. Ogni suo elemento è una coppia ordinata: la **base del segmento** e il **limite del segmento**. La base del segmento contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento. Un indirizzo logico è formato da due parti: un numero di segmento s e un offset in tale segmento d. Il numero di segmento si usa come indice per la tabella dei segmenti; l'offset d dell'indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti si genera una eccezione per il sistema operativo (tentativo di indirizzamento logico oltre la fine del segmento). Se la condizione sull'offset è rispettata, questo viene sommato alla base del segmento per produrre l'indirizzo della memoria fisica dove si trova il byte desiderato. Quindi la tabella dei segmenti è fondamentalmente un vettore di coppie di registri di base e limite.





.AVVICENDAMENTO DEI PROCESSI (SWAPPING)

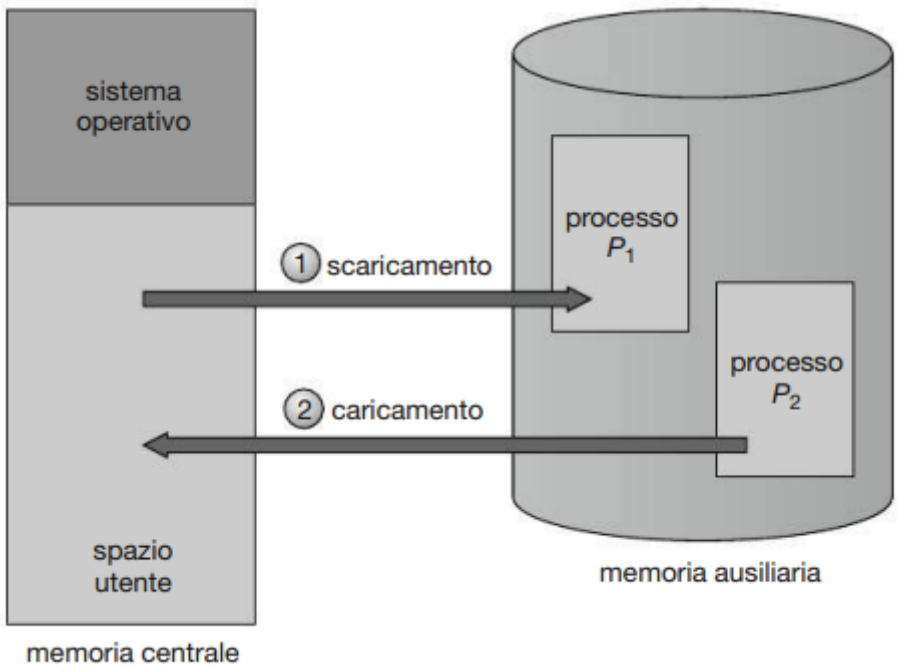
Le istruzioni di un processo e i dati su cui operano per essere eseguiti devono essere in memoria. Tuttavia, un processo o una sua parte, può essere rimosso/scaricato temporaneamente dalla memoria centrale (**swap-out**), spostato in una memoria ausiliaria (**backing store**) e riportato/caricato in memoria centrale (**swap-in**) per continuarne l'esecuzione. Grazie a questo procedimento, detto **avvicendamento dei processi** (**swapping**), lo spazio totale degli indirizzi fisici può eccedere la dimensione reale della memoria fisica, aumentando ulteriormente il grado di multiprogrammazione.

AVVICENDAMENTO STANDARD

L'avvicendamento standard riguarda lo spostamento di interi processi tra la memoria centrale e una memoria ausiliaria. La memoria ausiliaria deve essere abbastanza grande da contenere tutte le porzioni dei processi che devono essere memorizzate e recuperate, e deve fornire **accesso diretto** alle immagini dei processi memorizzati. Quando un processo, o parte di esso, viene spostato nella memoria centrale, devono essere scritte in tale memoria anche tutte le strutture dati associate al processo. Lo stesso principio si applica ai thread. Il SO inoltre, deve mantenere i metadati relativi ai processi swappati in modo da poterli ripristinare quando vengono reinseriti in memoria.

Il sistema mantiene una **coda dei processi pronti** (ready queue) formata da tutti i processi pronti per l'esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria centrale. Quando lo scheduler della CPU decide di eseguire un processo, richiama il dispatcher, che controlla se il primo

processo della coda si trova in memoria centrale. Se non si trova in memoria centrale, e in questa non c'è spazio libero, il dispatcher scarica un processo dalla memoria centrale e vi carica il processo richiesto dallo scheduler della CPU, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato. In un tale sistema d'avvicendamento, il tempo di cambio di contesto (**context-switch time**) è piuttosto elevato.

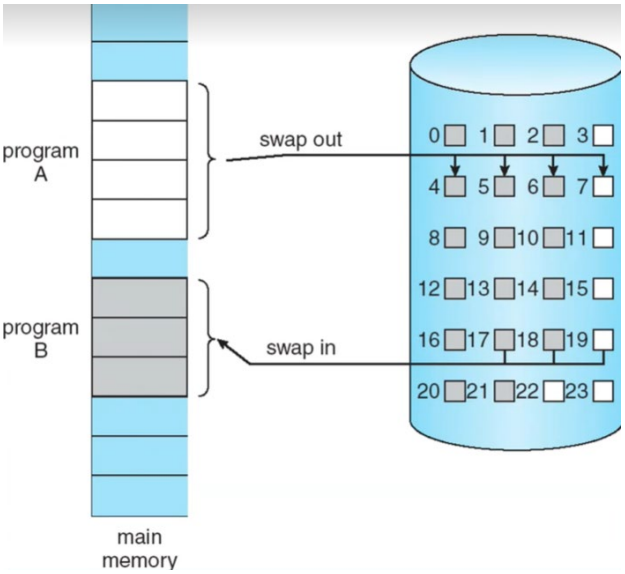


Il vantaggio dell'avvicendamento standard è che consente di sovrascrivere la memoria fisica in modo che il sistema possa ospitare più processi rispetto alla quantità di memoria fisica effettivamente disponibile. I processi inattivo o raramente attivi sono buoni candidati per l'avvicendamento in modo che la memoria da essi occupata possa essere allocata per processi attualmente attivi.

AVVICENDAMENTO CON PAGINAZIONE

L'avvicendamento standard non è più utilizzato nei sistemi operativi contemporanei poiché la quantità di tempo necessaria per spostare interi processi è proibitiva.

La maggior parte dei sistemi usa una variante dell'avvicendamento standard in cui è possibile spostare solo **alcune pagine** di un processo, piuttosto che l'intero processo. La variante prende il nome di **avvicendamento con paginazione** o **paginazione (paging)**. Un'operazione di **page out** (scaricamento della pagina) sposta una pagina dalla memoria centrale alla memoria ausiliaria; mentre, un'operazione di **page in** (caricamento della pagina) è il processo inverso. Questa tecnica funziona bene in abbinamento alla memoria virtuale.

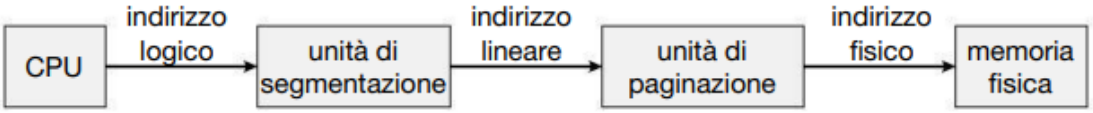


.ESEMPIO: LE ARCHITETTURE INTEL A 32 E 64 BIT

L'architettura Intel ha dominato il mondo dei personal computer per diversi anni. Il processo Intel 8086, a 16 bit, apparve alla fine degli anni '70 e fu presto seguito da un altro chip a 16 bit, l'Intel 8088, noto per essere stato il chip utilizzato nel PC IBM originale. Sia il chip 8086 sia il chip 8088 erano basati su un'architettura segmentata. Successivamente, Intel iniziò la produzione di una serie di chip a 32 bit, IA-32, che includeva la famiglia di processori Pentium. L'architettura IA-32 supportava paginazione e segmentazione. Più di recente Intel ha prodotto una serie di chip a 64 bit basati sull'architettura x86-64. Di seguito esamineremo la traduzione degli indirizzi nelle architetture IA-32 e x86-64 presentando i principali concetti della gestione della memoria di queste.

ARCHITETTURA IA-32

La gestione della memoria nei sistemi IA-32 è suddivisa in **due componenti**: **segmentazione** e **paginazione**. La CPU genera indirizzi logici che vengono passati all'unità di segmentazione che produce un indirizzo lineare per ogni indirizzi logico. L'indirizzo lineare viene passato all'unità di paginazione che genera l'indirizzo fisico. L'unità di segmentazione e l'unità di paginazione **formano l'MMU**.

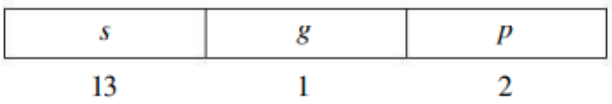


SEGMENTAZIONE IN IA-32

Nell'architettura IA-32 un segmento può raggiungere la dimensione massima di 4GB; il numero massimi di segmento per processo è pari a 16K. Lo spazio

degli indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8k segmenti riservati al processo; la seconda contiene 8k segmenti condivisi fra tutti i processi.

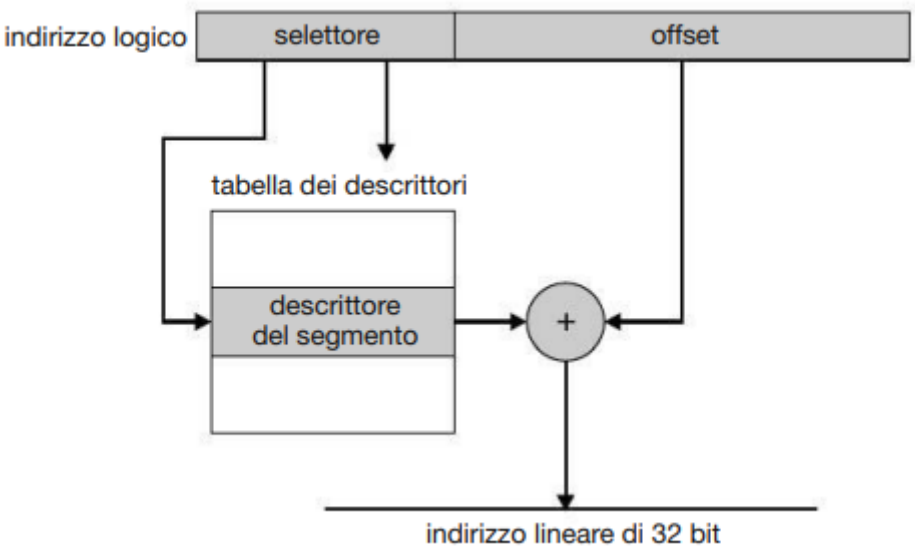
Le informazioni riguardanti la prima partizione sono contenute nella **tabella locale dei descrittori** (**local descriptor table**, **LDT**)**,** quelle relative alla seconda partizione sono memorizzate nella **tabella globale dei descrittori** (**global descriptor table**, **GDT**). Ciascun elemento nella LDT e nella GDT è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite. un indirizzo logico è una coppia (selettore, offset), dove il selettore è un numero di 16 bit:



in cui **s** indica il numero del segmento, **g** indica SE il segmento si trova nella GDT o nella LDT e **p** contiene informazioni relative alla protezione. L'offset è un numero di 32 bit che indica la posizione del byte (o della parola) all'interno del segmento in questione.

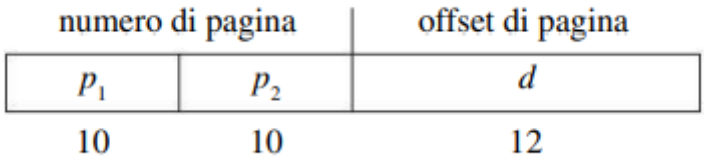
La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di microprogramma di 8 byte per i corrispondenti descrittori prelevati dalla LDT o dalla GDT. Questa cache evita alla macchina di dover prelevare dalla memoria i descrittori per ogni riferimento alla memoria.

Un indirizzo lineare di IA-32 è lungo 32 bit e si genera come segue. Il registro di segmento punta all'elemento appropriato all'interno della LDT o della GDT; le informazioni relative alla base e al limite di tale segmento si usano per generare **un indirizzo lineare**. Innanzi tutto, si usa il valore del limite per controllare la validità dell'indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa la generazione di un'eccezione e la restituzione del controllo al sistema operativo; altrimenti, si somma il valore dell'offset al valore della base, ottenendo un indirizzo lineare di 32 bit.

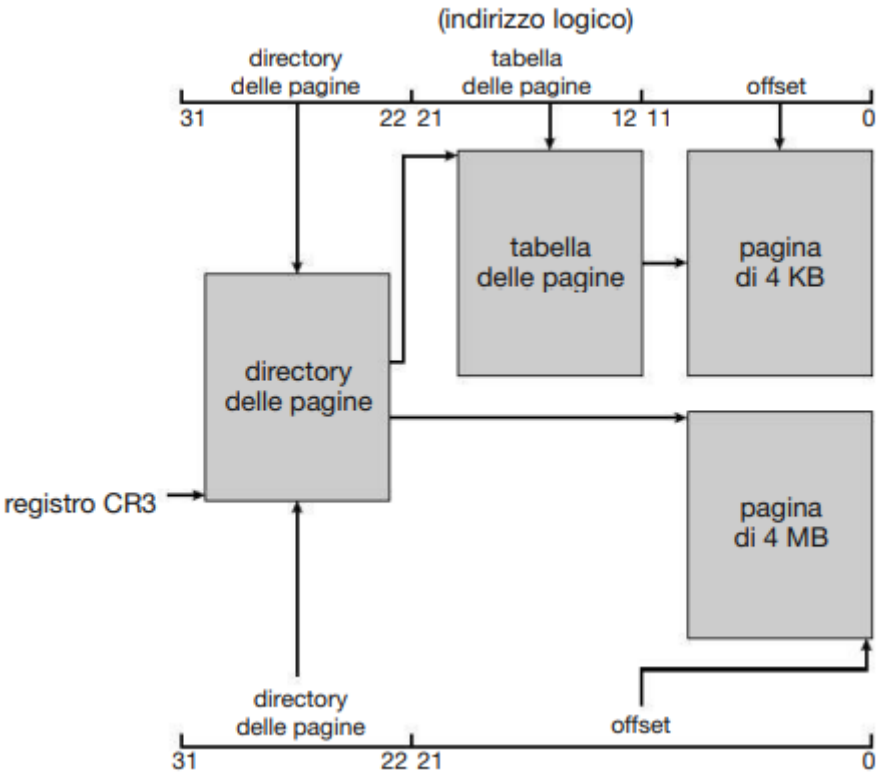


PAGINAZIONE IN IA-32

L'architettura IA-32 prevede che le pagine abbiano una misura di 4KB oppure di 4MB. Per le pagine di 4KB, in IA-32 vige uno schema di paginazione a due livelli:



La tabella esterna delle pagine, in IA-32 è detta **directory delle pagine**.



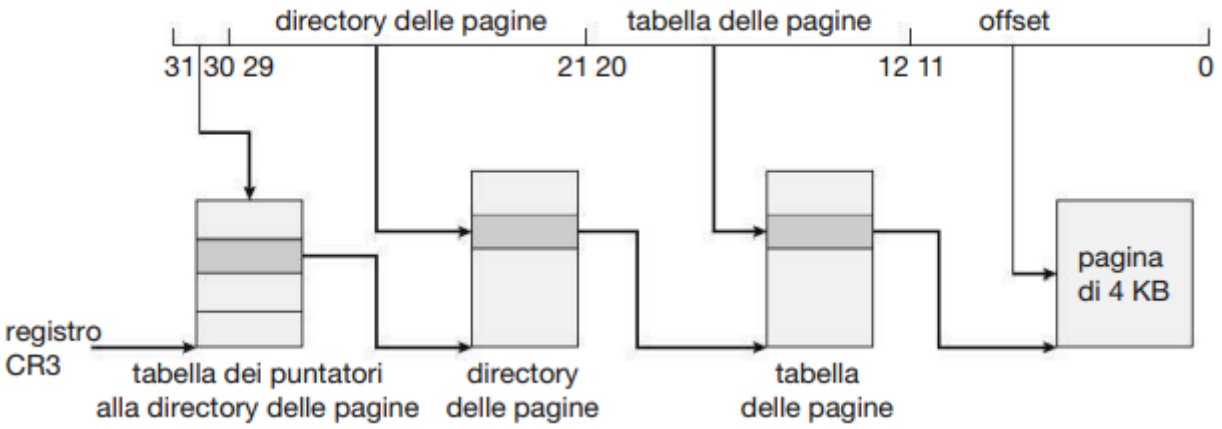
Gli elementi della directory delle pagine puntano a una tabella delle pagine interne, indicizzata da dieci bit intermedi dell'indirizzo lineare. Infine, i bit meno significativi in posizione 0-11 contengono l'offset da applicare all'interno della pagina di 4 KB cui si fa riferimento nella tabella delle pagine.

Un elemento appartenente alla directory delle pagine è il flag **Page_Size**; se impostato, indica che il frame non ha la dimensione standard di 4 MB, ma misura invece 4 KB. In questo caso, la directory di pagina punta direttamente al frame di 4 MB, scavalcando la tabella delle pagine interna; i 22 bit meno significativi nell'indirizzo lineare indicano l'offset nella pagina di 4 MB.

Per migliorare l'efficienza d'uso della memoria fisica, le tabelle delle pagine in IA-32 possono essere trasferite sul disco. In questo caso, si ricorre a un bit **invalid** in ciascun elemento della directory di pagina, per indicare se la tabella a cui l'elemento punta sia in memoria o sul disco. Se è su disco, il sistema operativo può usare i 31 bit rimanenti per specificare la collocazione della tabella sul disco; in questo modo, si può richiamare la tabella in memoria su richiesta.

Non appena gli sviluppatori di software hanno iniziato a soffrire della limitazione della memoria a 4 GB imposta dall'architettura a 32 bit, Intel ha introdotto **l'estensione di indirizzo della pagina** (**PAE**, **page address extension**), che consente ai processori a 32 bit di accedere a uno spazio di indirizzamento fisico più grande di 4 GB. La differenza fondamentale introdotta dal supporto PAE era il passaggio della paginazione da una schema a due

livelli a uno schema a tre livelli, in cui i primi due bit fanno riferimento a una tabella di puntatori alle directory di pagina.



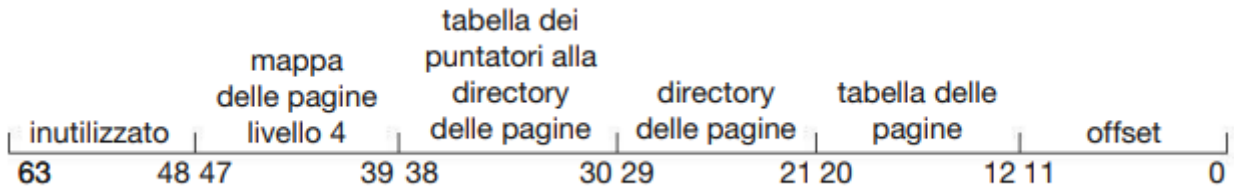
Con PAE è stata inoltre aumentata la dimensione degli elementi della directory delle pagine e della tabella delle pagine, che passa da 32 a 64 bit, permettendo di estendere l'indirizzo di base delle tabelle delle pagine e dei frame da 20 a 24 bit. In combinazione con i 12 bit di offset, l'aggiunta del supporto PAE a IA-32 ha aumentato lo spazio di indirizzamento a 36 bit, garantendo il supporto di un massimo di 64 GB di memoria fisica. È importante notare che per utilizzare PAE è necessario il supporto del sistema operativo. Linux e Intel Mac OS x supportano PAE. Tuttavia, le versioni a 32 bit dei sistemi operativi Windows per desktop supportano soltanto 4 GB di memoria fisica, anche se PAE è abilitato.

ARCHITETTURA X86-64

Lo sviluppo di architetture Intel a 64 bit ha avuto una storia curiosa. La prima di queste architetture era IA-64 (in seguito denominata **Itanium**), ma questa architettura non ha avuto un'ampia diffusione. Nel frattempo, un altro produttore di chip – AMD – ha iniziato a sviluppare un'architettura a 64 bit nota come x86-64, basata sull'estensione del set di istruzioni IA-32 esistente. L'architettura x86-64 supportava spazi di indirizzamento logico e fisico molto più grandi e introduceva diverse altre novità architetturali. Storicamente AMD aveva spesso sviluppato chip basati sull'architettura Intel, ma in questo caso i ruoli si sono invertiti e Intel ha adottato l'architettura x86-64 di AMD.

Il supporto a uno spazio di indirizzamento a 64 bit permette di indirizzare la straordinaria quantità di 2^{64} byte di memoria – un numero superiore a 16 miliardi di miliardi (o 16 exabyte). Tuttavia, anche se i sistemi a 64 bit possono potenzialmente indirizzare una tale quantità di memoria, nella pratica vengono utilizzati nei progetti attuali assai meno di 64 bit per la rappresentazione di un indirizzo. L'architettura x86-64 utilizza attualmente un indirizzo virtuale di 48 bit con supporto ai formati di pagina di 4 kb, 2 Mb o 1 GB utilizzando una

paginazione a quattro livelli. Poiché questo schema di indirizzamento può usare PAE, gli indirizzi virtuali sono di 48 bit, ma supportano indirizzi fisici a 52 bit (4096 terabyte).



CAPITOLO 10

INTRODUZIONE

Nel capitolo precedente sono state esaminate varie strategie per la gestione della memoria. Esse hanno lo scopo di aumentare il grado di multiprogrammazione; tuttavia, richiedono che l'**intero processo** sia caricato in memoria.

La **memoria virtuale** è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. La memoria virtuale libera i programmi dalle limitazioni della memoria (i programmi possono essere più grandi della memoria fisica), separa la memoria logica dalla memoria fisica, permette ai processi di condividere facilmente file e di realizzare memoria condivise, fornisce un meccanismo efficiente per la creazione dei processi. La memoria virtuale è difficile da realizzare e, se usata scorrettamente, può ridurre di molto le prestazioni del sistema.

Gli algoritmi di gestione della memoria delineati nel Capitolo 9 sono necessari a causa di un requisito fondamentale: le istruzioni da eseguire si devono trovare all'interno della memoria fisica.

Il primo metodo per far fronte a tale requisito consiste nel collocare l'intero spazio d'indirizzi logici del processo in memoria fisica. Il loading dinamico può aiutare ad attenuare gli effetti di tale limitazione, ma richiede generalmente particolari precauzioni e un ulteriore impegno dei programmatori.

La condizione che le istruzioni debbano essere nella memoria fisica è necessaria ma **riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni della memoria fisica**. Come abbiamo potuto vedere già vedere, molto spesso non c'è bisogno che l'intero programma debba risiedere interamente in memoria. Si considerino queste situazioni:

- Spesso i programmi dispongono di codice per la gestione di condizioni d'errore insolite;

- Spesso ad array, liste e tabelle si assegna più memoria di quanta sia effettivamente necessaria;
- Alcune opzioni e caratteristiche di un programma sono utilizzabili solo di rado.

Anche nei casi in cui è necessario disporre di tutto il programma, è possibile che non serva tutto in una volta. La possibilità di eseguire un programma che si trova solo parzialmente in memoria porterebbe molti benefici.

- Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno **spazio degli indirizzi virtuali** molto grande, semplificando così il compito della programmazione;
- Poiché ogni programma utente può impiegare meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, aumentando il nostro grado di programmazione e ottimizzando l'utilizzo della nostra CPU;
- Per caricare (o avvicendare) ogni programma utente in memoria sono necessarie meno operazioni di I/O; quindi, ogni programma utente è eseguito più rapidamente.

La possibilità di eseguire un programma che non si trovi completamente in memoria apporterebbe quindi vantaggi sia al sistema sia all'utente.

La **memoria virtuale** si fonda sulla **separazione** della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmatori una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola. In questo modo, il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile.

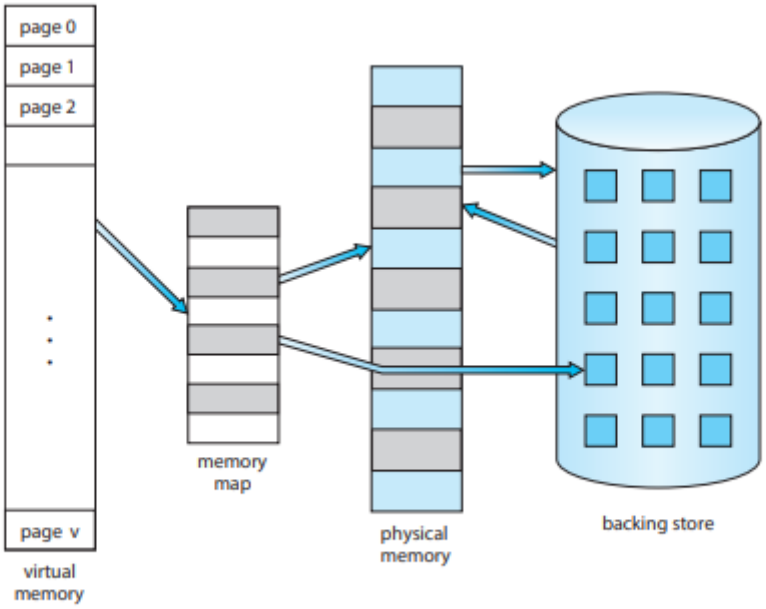


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

L'espressione **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Tipicamente, da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico – per esempio, l'indirizzo 0 – e si estende in uno spazio di memoria contigua.

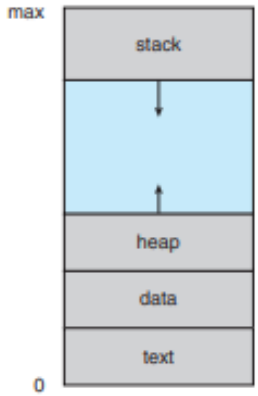


Figure 10.2 Virtual address space of a process in memory.

Tuttavia, è possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono essere non contigui. Spetta alla MMU associare in memoria le pagine logiche alle pagine fisiche.

L'ampio spazio vuoto (o buco) che separa heap dallo stack è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche reali solo nel caso che lo heap o lo stack crescono. Uno spazio degli indirizzi virtuali che contiene buchi si

definisce **sparso**. Un simile spazio degli indirizzi è utile, poiché i buchi possono essere riempiti grazie all'espansione dei segmenti heap o stack, oppure se vogliamo collegare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.

Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre il vantaggio di condividere i file e la memoria fra più processi, mediante la condivisione delle pagine. I vantaggi:

- Le librerie di sistema sono condivisibili da diversi processi associando (**mappando**) l'oggetto di memoria condiviso a uno spazio degli indirizzi virtuali. Benché ogni processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano le librerie nella memoria fisica sono in condivisione tra tutti i processi. Di norma, le librerie sono allocate in modalità di sola lettura;

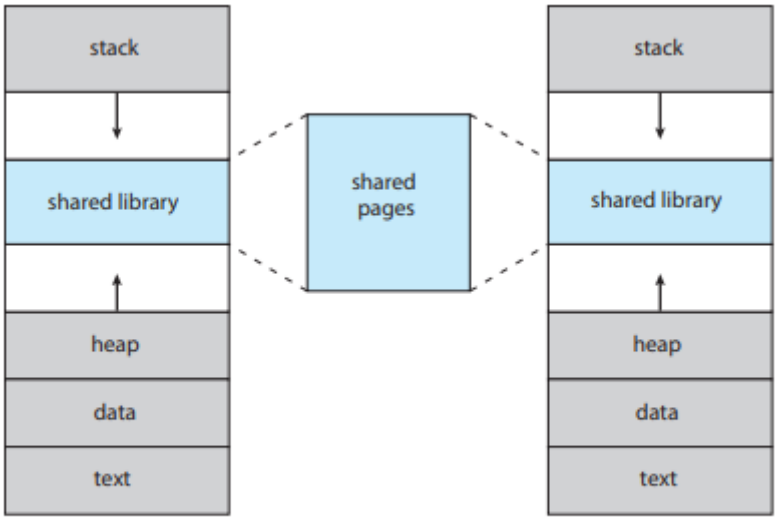


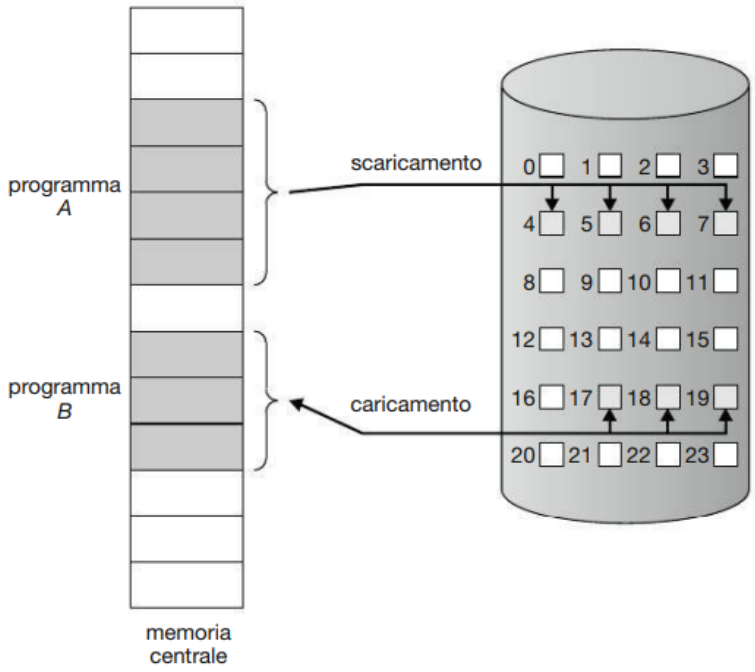
Figure 10.3 Shared library using virtual memory.

- In maniera analoga, la memoria può essere condivisa tra processi distinti. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise;
- Le pagine possono essere condivise durante la creazione di un processo mediante la chiamata di sistema fork(), così da velocizzare la generazione dei processi.

.PAGINAZIONE SU RICHIESTA

Si consideri il loading dinamico in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che

serva avere tutto il programma in memoria. Una strategia alternativa consiste nel **caricare le pagine nel momento in cui servono realmente**; si tratta di una tecnica, detta **paginazione su richiesta** o **paginazione on-demand**, comunemente adottata dai sistemi con memoria virtuale. Le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica. Un sistema di paginazione on-demand è analogo a un sistema paginato con avvicendamento dei processi in memoria. I processi risiedono in memoria secondaria (generalmente su disco). Tuttavia, anziché caricare in memoria l'intero processo, si può seguire un metodo d'avvicendamento "pigro" (**lazy swapping**): non si carica mai in memoria una pagina che non sia necessaria. Nell'ambito dei sistemi con paginazione su richiesta l'uso del termine avvicendamento o swapping non è appropriato: uno **swapper** manipola interi processi mentre un **paginatore** (**pager**) gestisce le singole pagine dei processi. La paginazione su richiesta offre uno dei principali vantaggi della memoria virtuale: caricando solo le parti necessarie dei programmi la memoria viene utilizzata in modo più efficiente.



CONCETTI FONDAMENTALI

Il concetto alla base della paginazione on-demand è di caricare una pagina in memoria solo quando è necessaria. Di conseguenza, mentre un processo è in esecuzione, alcune pagine saranno in memoria e altre si troveranno nella memoria secondaria. È dunque necessaria una forma di supporto hardware per distinguere i due casi. A tal scopo si può utilizzare lo schema basato sul **bit di validità**.

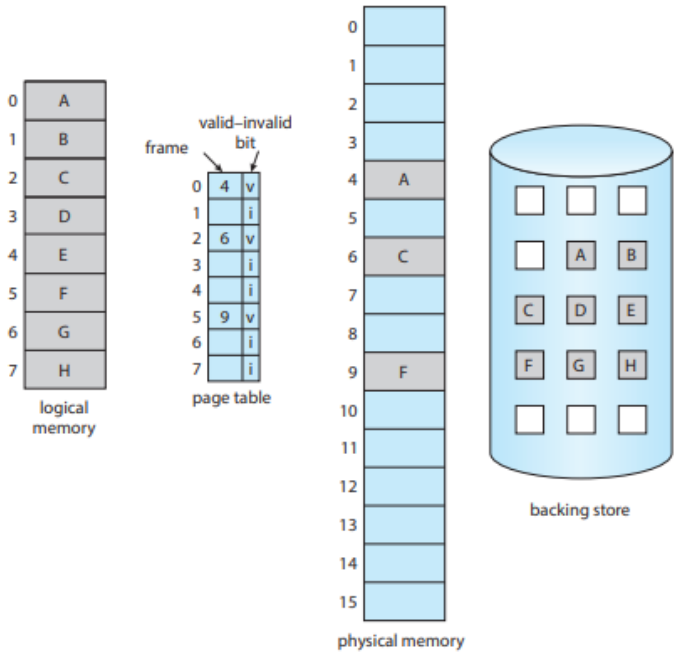


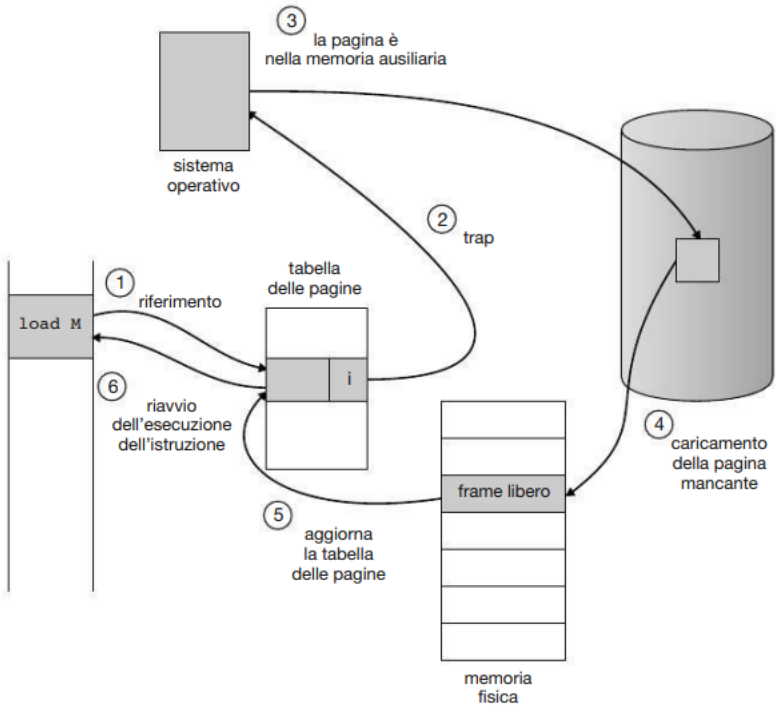
Figure 10.4 Page table when some pages are not in main memory.

In questo caso serve per stabilire se una pagina è presente in memoria (**valido**) o non appartiene allo spazio di indirizzi perché risiede nella memoria secondaria (**non valido**).

Che succede se il processo tenta l'accesso a una pagina che non era stata caricata in memoria? L'accesso a una pagina contrassegnata come non valida causa un evento o eccezione di **page fault** (**pagina mancante**). L'hardware di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit è non valido e genera una trap per il sistema operativo; tale eccezione è dovuta a un "insuccesso" del sistema operativo nella scelta delle pagine da caricare in memoria. La procedura di gestione dell'eccezione di page fault è lineare:

- Si controlla una tabella interna per questo processo (conservata insieme al PCB) allo scopo di stabilire se il riferimento fosse un accesso valido o meno;
- Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua il caricamento;
- Si individua un frame libero;
- Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame assegnato;
- Quando la lettura dal disco è completa, si modificano la tabella interna, conservata col processo, e la tabella delle pagine per indicare che la pagina si trova in memoria;

- Si riavvia l'istruzione interrotta dall'eccezione. A questo punto il processo può accedere alla pagina.



È possibile avviare l'esecuzione di un processo **senza** pagine in memoria. Quando il sistema operativo carica nel contatore di programma l'indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo genera immediatamente un page fault. Una volta portata la pagina in memoria, il processo continua l'esecuzione, generando page fault fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una **paginazione su richiesta pura**, vale a dire che una pagina non si trasferisce mai in memoria se non viene richiesta.

In teoria alcuni programmi possono accedere a diverse nuove pagine di memoria all'esecuzione di ogni istruzione (una pagina per l'istruzione e molte per i dati), eventualmente causando più page fault per ogni istruzione. In un caso simile le prestazioni del sistema sarebbero inaccettabili. Fortunatamente l'analisi dei programmi in esecuzione mostra che questo comportamento è estremamente improbabile. I programmi tendono ad avere una **località dei riferimenti**, quindi le prestazioni della paginazione su richiesta risultano ragionevoli.

L'hardware di supporto alla paginazione su richiesta è lo stesso che è richiesto per la paginazione e l'avvicendamento dei processi in memoria:

- **tabella delle pagine**. Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- **memoria secondaria**. Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo di swap; la sezione del disco usata a questo scopo si chiama **area di avvicendamento** (**swap space**).

Uno dei requisiti cruciali della paginazione su richiesta è **la possibilità di rieseguire una qualunque istruzione dopo un page fault**. Avendo salvato lo stato del processo interrotto, al momento del page fault, occorrerà riavviare il processo esattamente dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questo requisito è facile da soddisfare. Un page fault si può verificare per qualsiasi riferimento alla memoria. Se si verifica durante la fase di fetch (prelievo) di un'istruzione, l'esecuzione si può riavviare effettuando nuovamente il fetch. Se si verifica durante il fetch di un operando, bisogna effettuare nuovamente fetch e decode dell'istruzione, quindi si può prelevare l'operando. La difficoltà maggiore si presenta quando un'istruzione può modificare parecchie locazioni diverse

Il problema si può risolvere in due modi. In una delle due soluzioni il microcodice computa e tenta di accedere alle estremità delle due sequenze di byte. Un'eventuale page fault si può verificare solo in questa fase, prima che si apporti qualsiasi modifica. A questo punto si può compiere lo spostamento senza rischio di page fault perché tutte le pagine interessate si trovano in memoria. L'altra soluzione si serve di registri temporanei per conservare i valori delle locazioni sovrascritte. Nel caso di un page fault, si riscrivono tutti i vecchi valori in memoria prima che sia generata la trap. Questa operazione riporta la memoria allo stato in cui si trovava prima che l'istruzione fosse avviata; perciò, si può ripetere la sua esecuzione.

Sebbene non si tratti certo dell'unico problema da affrontare per estendere un'architettura esistente con la funzionalità della paginazione su richiesta, illustra alcune delle difficoltà da superare. Il sistema di paginazione si colloca tra la CPU e la memoria di un calcolatore e deve essere completamente trasparente al processo utente. L'opinione comune che la paginazione si possa aggiungere a qualsiasi sistema è vera per gli ambienti senza paginazione su richiesta, nei quali un'eccezione di page fault rappresenta un errore fatale, ma è falsa nei casi in cui un'eccezione di page fault implica solo la necessità di caricare in memoria un'altra pagina e quindi riavviare il processo.

LISTA DEI FRAME LIBERI

Quando si verifica un page fault, il SO deve spostare la pagina desiderata dalla memoria secondaria alla memoria principale. Per risolvere i page fault la maggior parte dei SO mantiene una **lista dei frame liberi**, ovvero un insieme di frame disponibili e utilizzabili per soddisfare le richieste.

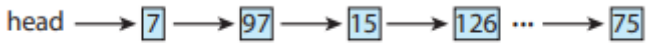


Figure 10.6 List of free frames.

I frame liberi devono essere allocati quando lo stack o l'heap di un processo si espandono.

I sistemi operativi allocano generalmente i frame liberi usando una tecnica nota come **zero-fill-on-demand**: i frame vengono "azzerati" su richiesta prima di essere allocati, cancellando così il loro precedente contenuto. All'avvio del SO tutta la memoria disponibile viene inserita nella lista dei frame liberi. Man mano che vengono richiesti frame liberi, la dimensione della lista si riduce. A un certo punto la lista diventa vuota oppure la sua dimensione scende al di sotto di una certa soglia: questo implica che deve essere ripopolata.

.COPIATURA SU SCRITTURA

Precedentemente si è visto come un processo possa cominciare rapidamente l'esecuzione richiedendo solo la pagina contenente la prima istruzione. La generazione dei processi tramite fork(), crea un processo figlio come duplicato del genitore duplicando anche le pagine. Tuttavia, la fork() può evitare la duplicazione delle pagine del genitore tramite una tecnica simile alla condivisione delle pagine che minimizza il numero di pagine allocate per il processo appena generato. La tecnica è nota come **copiatura su scrittura** (**copy-on-write**, **CoW**), che permette inizialmente ai processi genitore e ai processi figli di condividere le stesse pagine. Le pagine condivise **modificabili** vengono contrassegnate come **pagina da copiare su scrittura** (**copy-on-write**); questo significa che se il processo genitore o uno dei figli proverà a modificare una di queste pagine allora il sistema creerà in memoria una copia della pagina che il processo intende modificare e le modifiche effettuate influenzeranno solo la pagina copiata dal processo.

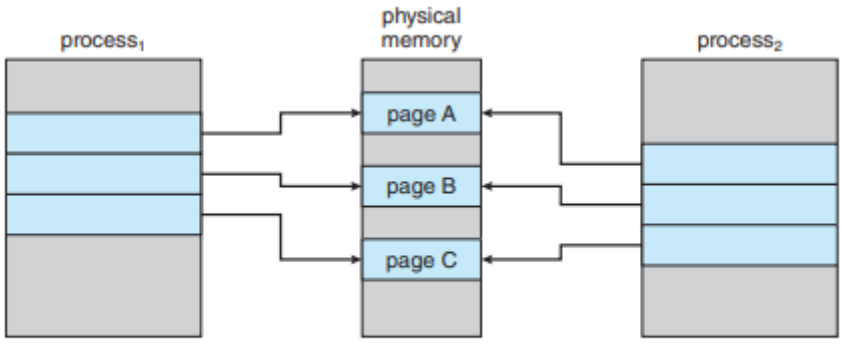


Figure 10.7 Before process 1 modifies page C.

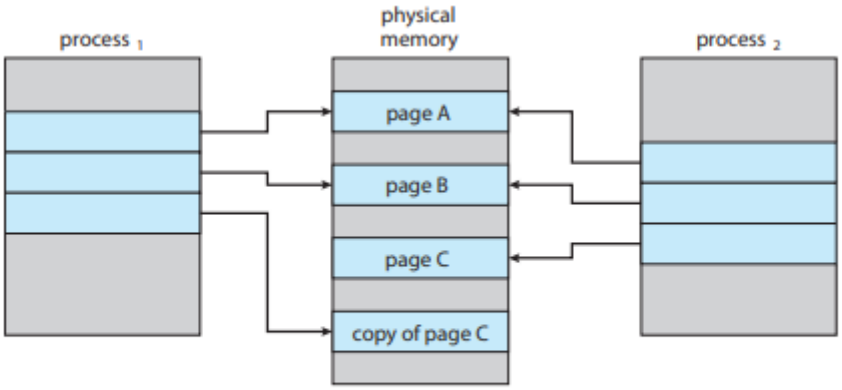


Figure 10.8 After process 1 modifies page C.

PARTIZIONE DI SWAP

Come abbiamo visto, le pagine che non vengono utilizzate vengono "parcheggiate" in memoria secondaria e viene fatto lo swap quando effettivamente ci servono. Di solito si assegna una partizione nella memoria secondaria che prende il nome di **partizione swap**. La gestione di questa partizione rispetto ad un filesystem non ha bisogno di formattazioni e all'interno di questa, vengono lette e scritte solo pagine di **dimensione uguale**. Quindi avendo i record di dimensione costante, l'accesso alla memoria sarà più semplice visto che faremo un accesso diretto in base all'indice. **Ricordiamoci che stiamo comunque accedendo ad una memoria secondaria, quindi i tempi di accesso saranno più lunghi rispetto alla memoria principale**.

.SOSTITUZIONE DELLE PAGINE

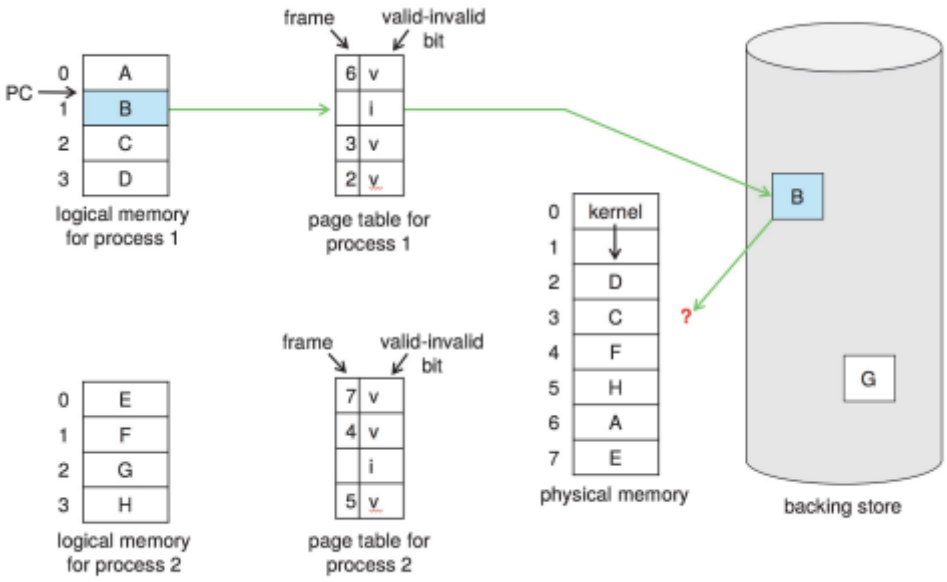
Ogni pagina può generare più di un fault!

Aumentando il grado di multiprogrammazione, si **sovrassegna** la memoria. Ad esempio, se si stanno eseguendo 6 processi, ciascuno dei quali dispone di 10 pagine, di cui solo 5 sono utilizzate, si avrebbero a disposizione ancora 30 pagine e sarebbe possibile aumentare la produttività e l'utilizzo della CPU caricando più processi. Tuttavia, potrebbe capitare che per un insieme

particolare di dati, ciascuno dei 6 processi originali abbiano bisogno di tutte e 10 le pagine a loro inizialmente allocate e dunque sarebbero necessarie 60 pagine, mentre ne sono disponibili solo 30.

Si consideri inoltre che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l'I/O impegnano una rilevante quantità di memoria. Questo rende difficile il processo di allocazione di memoria in quanto bisogna decidere quanta memoria allocare ai processi e quanta memoria assegnare all'I/O. Alcuni sistemi operativi riservano una quota fissa per l'I/O, altri permettono sia ai processi utenti sia al sottosistema di I/O di competere per tutta la memoria del sistema.

In generale, ogni processo sarà caricato in memoria in segmenti di dimensione variabile, contenenti pagine di dimensione costante. A loro volta faranno riferimento alla lista dei frame liberi per trovare il frame libero per caricare le pagine quando necessarie. Ma cosa succede se la lista è vuota? Quando la lista dei frame è vuota si verifica il fenomeno della **sovrallocazione/sovrassegnazione** (**over-allocation**) che si può illustrare come segue. Durante l'esecuzione di un processo utente si verifica un page fault. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata e scopre che la lista dei frame è vuota: tutta la memoria è in uso.

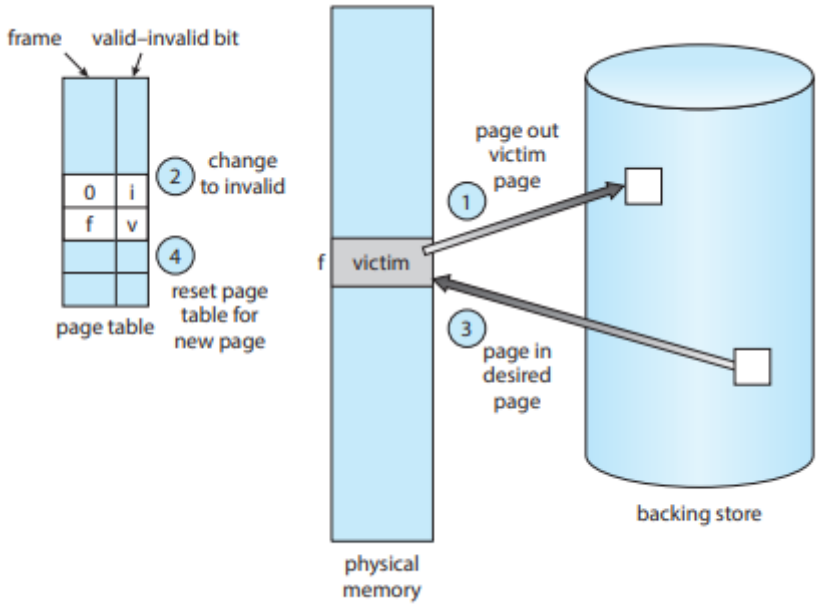


A questo punto il sistema può scegliere di **killare/terminare** il processo, ma così facendo ridurrebbe il grado di multiprogrammazione.

La soluzione ideale, e adottata dalla maggior parte dei sistemi operativi, è quella di **combinare lo swapping con la sostituzione delle pagine**.

La sostituzione delle pagine segue il seguente criterio: **se nessun frame è libero, ne viene liberato uno attualmente inutilizzato**. Il frame viene liberato scrivendo il suo contenuto nell'area di swap e modificando la tabella delle pagine per indicare che la pagina non si trova più in memoria. Il frame libero viene usato per memorizzare la pagina che ha causato il fault. Si modifica quindi la procedura di servizio dell'eccezione di page fault in modo da includere la sostituzione della pagina:

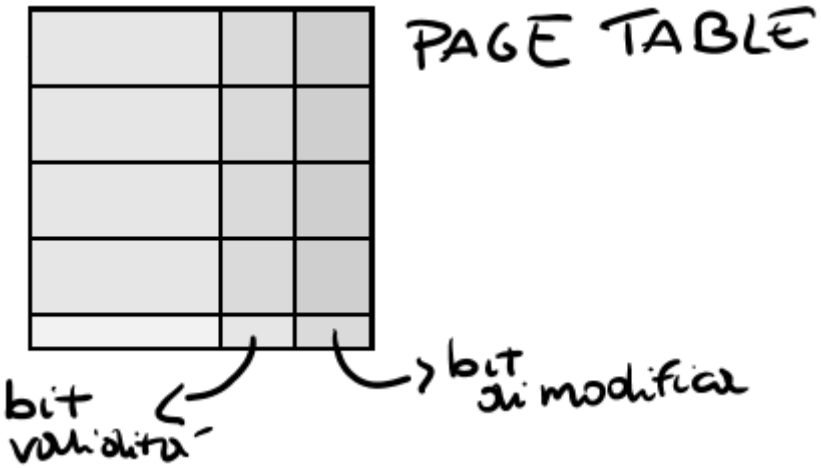
- S'individua la locazione su disco della pagina richiesta;
- Si cerca un frame libero. Se esiste lo si usa, altrimenti si impiega un **algoritmo di sostituzione delle pagine** per scegliere un **frame vittima** e si scrive la pagina **vittima** nel disco, modificando successivamente la tabella delle pagine e dei frame;
- Si scrive la pagina richiesta nel frame appena liberato, modificando la tabella delle pagine e dei frame;
- Si riprende il processo dal punto in cui si è verificato il page fault.



Occorre notare che se non esiste alcun frame libero sono necessari **due** trasferimenti di pagine, uno fuori e uno dentro la memoria. Questa situazione raddoppia il tempo di servizio del page fault e anche il tempo d'accesso effettivo.

Questo sovraccarico si può ridurre usando un **bit di modifica** (**modify bit** o **dirty bit**) che viene associato ad ogni pagina (o frame). Tale bit viene impostato a 1 ogni qualvolta che una pagina viene modificata, indicando che questa deve essere scritta sul disco. Se il bit rimane a 0, significa che la pagina non è stata modificata dall'ultima lettura e dunque non deve essere scritta sul disco. Questo schema permette di ridurre in maniera considerevole il tempo

per il servizio del page fault e di dimezzare il tempo di I/O **SE la pagina non è stata modificata**.



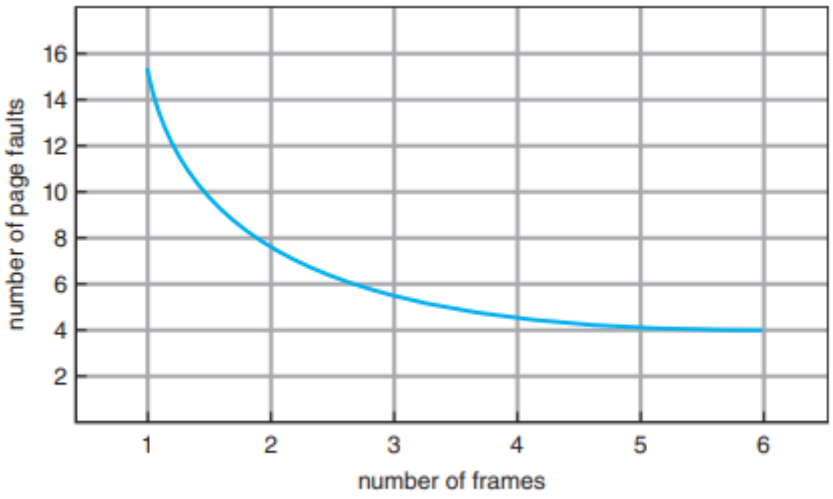
La sostituzione di una pagina è fondamentale al fine della paginazione on-demand, perché completa la separazione tra memoria logica e fisica. Con questo meccanismo si può allocare più memoria logica rispetto alla dimensione della memoria fisica, che può essere molto ridotta.

Per realizzare la paginazione on-demand è necessario risolvere due problemi necessari: ovvero la necessità di un **algoritmo di allocazione dei frame** e un **algoritmo di sostituzione delle pagine**. Ossia, se sono presenti più processi in memoria, occorre decidere quanti frame vadano assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione della pagina, occorre selezionare il frame da sostituire. La progettazione di quest'ultimi è molto importante poiché I/O sui dischi è molto costoso. Ciò riflette anche sulla scelta della pagina vittima.

Ci sono diversi algoritmi di sostituzione delle pagine, e probabilmente ogni SO dispone di un proprio schema di sostituzione. Solitamente viene scelto l'algoritmo col minor tasso di page fault.

Un algoritmo viene valutato effettuandone l'esecuzione su una particolare successione di riferimenti (**successione dei riferimenti**) e calcolando il numero di page fault.

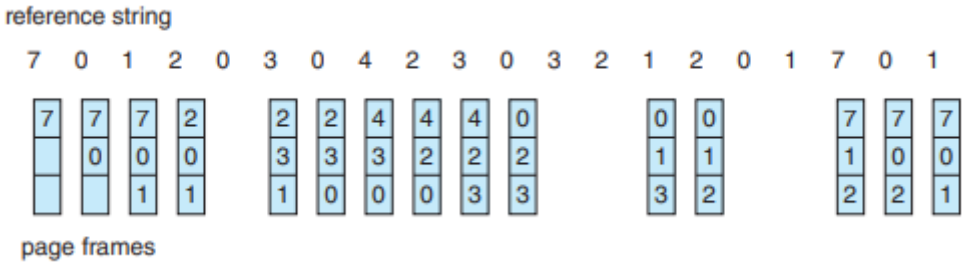
Per stabilire il numero di page fault relativo ad una successione, bisogna conoscere il numero dei frame disponibili. **Più frame disponibili, meno page fault**.



Quello che di solito si ottiene dalla valutazione degli algoritmi è un grafico di questo tipo che mostra il numero di page fault in relazione al numero di frame disponibili.

SOSTITUZIONE DELLE PAGINE SECONDO L'ORDINE DI ARRIVO (FIFO)

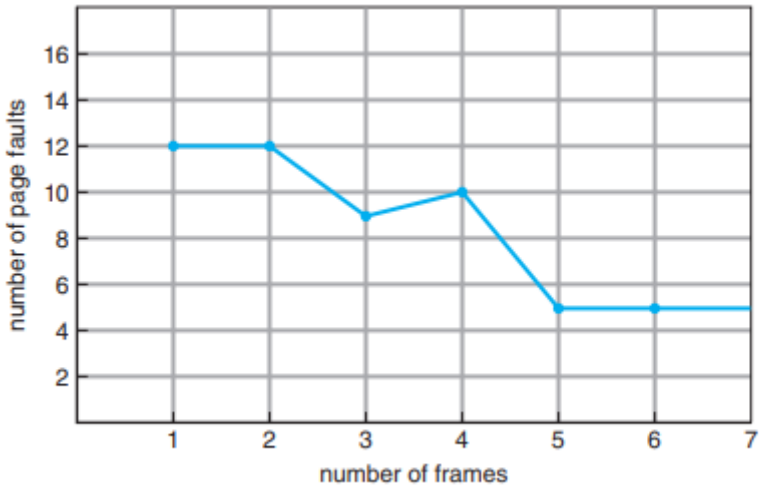
L'algoritmo di sostituzione delle pagine più semplice è un algoritmo FIFO il quale associa ad ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo (la prima ad essere stata portata in memoria).



L'algoritmo FIFO, per quanto facile da implementare, non ha delle buone prestazioni. La pagina sostituita, ad esempio, potrebbe essere un modulo usato tempo prima e che non serve più ma contiene una variabile molto usata e ancora in uso. Oppure il primo frame che caricheremo in memoria sarà l'operazione chiamante e ci servirà alla terminazione del processo.

Se si sceglie come pagina da sostituire una pagina attualmente in uso, tutto continua a funzionare correttamente perché, dopo averla rimossa, quasi immediatamente si verifica un'eccezione di page fault che riprende la pagina attiva. Per riprendere la pagina attiva, tuttavia, bisogna sostituire un'altra

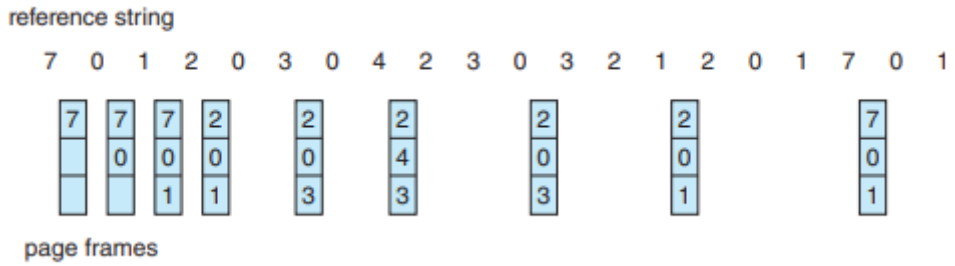
pagina; quindi, una scelta cattiva della pagina da sostituire aumenta il tasso di page fault, rallenta l'esecuzione del processo, ma non causa errori.



Notiamo dal grafico che il tasso di page fault con 4 frame disponibili è decisamente più elevato del tasso di page fault con 3 frame. Quest'anomalia è nota come **anomalia di Belady** dovuta dal fatto che con alcuni algoritmi di sostituzione delle pagine, il tasso di page fault può aumentare con l'aumentare del numero di frame assegnati.

SOSTITUZIONE OTTIMALE DELLE PAGINE

In seguito alla scoperta dell'anomalia di Belady si è ricercato un **algoritmo ottimale di sostituzione delle pagine**. Tale algoritmo presenta il tasso minimo di page fault e non presenta mai l'anomalia di Belady. Questo algoritmo è stato chiamato **OPT** o **MIN** e consiste nel **sostituire la pagine che non verrà usata per il periodo di tempo più lungo**.

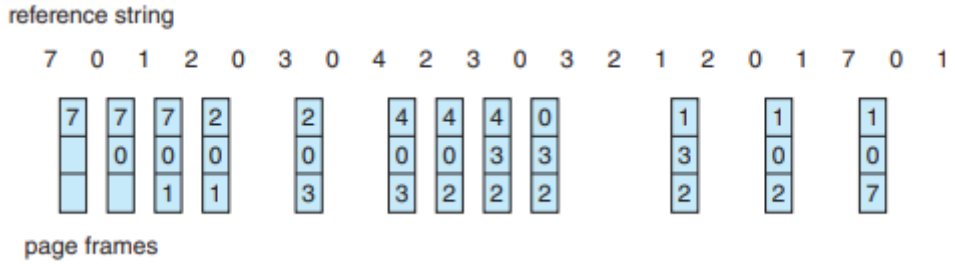


L'OPT è difficile da realizzare in quanto richiede la conoscenza futura delle successioni dei riferimenti (situazione analoga con l'SJF).

SOSTITUZIONE DELLE PAGINE USATE MENO RECENTEMENTE (LRU, LEAST RECENTLY USED)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT,

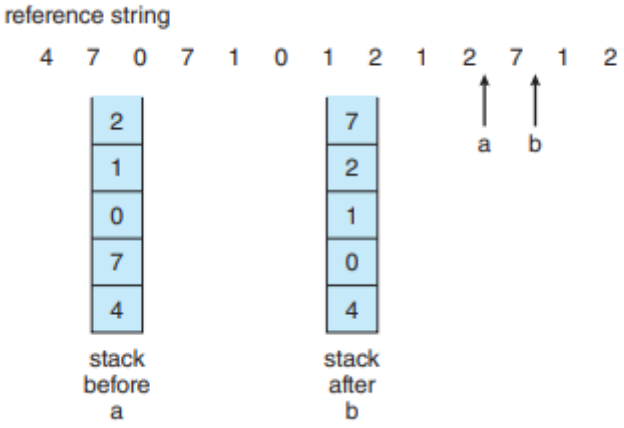
oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo FIFO impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo OPT impiega l'istante in cui una pagina sarà usata. Usando un'approssimazione, si sostituisce la pagina che **non è stata usata per il periodo più lungo**. Il metodo appena descritto è noto come **algoritmo LRU** (**Least Recently Used**). La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Lo si può definire come un algoritmo OPT con ricerca all'indietro anziché in avanti.



Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine. Il problema principale riguarda la sua implementazione. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'hardware. Il problema consiste nel determinare un ordine per i frame definito dal momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni:

- **Contatori**. Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo "momento di utilizzo", e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Si sostituisce la pagina con il valore più piccolo. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina LRU e una scrittura in memoria per ogni accesso alla memoria. I riferimenti temporali si devono mantenere anche quando si modificano le tabelle delle pagine e bisogna considerare l'overflow del contatore. Un altro problema, particolarmente rilevante, è che non resettiamo il contatore quando la pagina non viene più utilizzata; quindi, il bit dedicato non diventa più un dato indicativo. Una possibile soluzione è implementare un timer che ad intervalli regolari resetta i contatori di tutte le pagine. Ai contatori assegniamo più di un bit e nel caso in cui non venga effettuato un accesso alla pagina quest'ultima riceve una "ammonizione". A seconda del numero di bit, se la pagina riceve un numero di "ammonizioni" pari al numero di bit, essa viene espulsa.
- **Stack**. Un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede l'utilizzo di uno stack dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dallo stack e la si colloca in cima a quest'ultimo. In questo modo, in cima allo stack si trova

sempre la pagina usata per ultima, mentre in fondo si trova la pagina usata meno recentemente. Poiché gli elementi si devono estrarre dal centro dello stack, la migliore realizzazione si ottiene usando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Per estrarre una pagina dallo stack e collocarla in cima, nel caso peggiore è necessario modificare sei puntatori. Ogni aggiornamento è un po' più costoso, ma per una sostituzione non si deve compiere alcuna ricerca: il puntatore dell'elemento di coda punta alla pagina LRU. Questo metodo è adatto soprattutto alle realizzazioni programmate (o microprogrammate) della sostituzione LRU.



Né l'algoritmo OPT né l'algoritmo LRU sono soggetti all'anomalia di Belady. Essi appartengono a una categoria di algoritmi di sostituzione delle pagine chiamata **algoritmi a stack**, che non presenta l'anomalia di Belady.

NOTA: **Senza un supporto hardware nessuna implementazione LRU sarebbe possibile in quanto, senza supporto hardware, la gestione dei contatori o dello stack diventerebbe responsabilità completa del SO il quale sarebbe costretto ad adoperare un'interruzione per ogni riferimento causando un rallentamento generale e un sovraccarico della gestione della memoria**.

SOSTITUZIONE DELLE PAGINE PER APPROSSIMAZIONE A LRU

Sono pochi i sistemi di calcolo che dispongono del supporto hardware per una vera sostituzione LRU. Nei sistemi che non offrono alcun supporto hardware, si devono impiegare altri algoritmi di sostituzione delle pagine. Molti sistemi, tuttavia, possono fornire un aiuto: **un bit di riferimento**. Tale bit è impostato automaticamente dall'hardware ogni volta che si effettua un riferimento a una determinata pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il SO azzera tutti i bit e quando s'inizia l'esecuzione, l'hardware imposta a 1 il bit associati a ciascuna pagina a cui si fa riferimento; in tal modo dopo qualche tempo è possibile stabilire quali pagine sono state usate e quali no tramite il bit di riferimento. **Non è possibile stabilire l'ordine d'uso**.

ALGORITMO CON BIT SUPPLEMENTARI DI RIFERIMENTO

Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella un byte per ogni pagina. Ad esempio, ogni 100ms, il timer restituisce il controllo al sistema operativo che inserisce il bit di riferimento nel bit più significativo del byte e shifta tutti gli altri bit a destra di 1 posizione:

00000000 -> 10000000

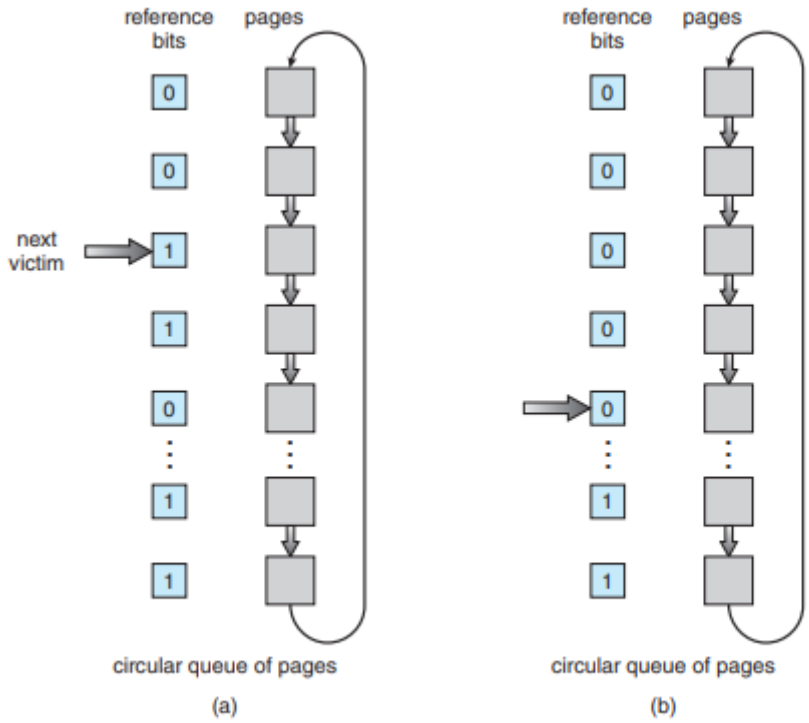
Tali registri a scorrimento a 8 bit (1 byte) contengono la storia d'utilizzo delle pagine e, se un registro a scorrimento contiene una successione di bit **00000000** allora la pagina non è stata usata negli ultimi 8 intervalli di tempo e può essere sostituita. Le successioni di 8 bit vengono interpretate come **numeri interi senza segno** e la pagina che viene sostituita è la pagina alla quale è stata associata la successione di bit col valore minore. **Non è possibile garantire l'unicità della successione**; in questo caso si possono sostituire tutte le pagine col valore minore o ricorrere a un FIFO.

Il # di bit che compongono le successioni varia in base all'hardware del sistema; e nel caso in cui tale numero sia 0, quindi si lascia solo il bit di riferimento e non si usano i registri di scorrimento a 8 bit, tale algoritmo prende il nome di **algoritmo di sostituzione delle pagine con seconda chance**.

ALGORITMO CON SECONDA CHANCE

L'algoritmo di sostituzione delle pagine con seconda chance è un algoritmo di tipo FIFO. Tale algoritmo opera nella seguente maniera: dopo aver selezionato una pagina, si controlla il bit di riferimento: se il suo valore è 0, dunque la pagina non è stata usata recentemente, si sostituisce la pagina; se il bit è impostato a 1, si dà alla pagina una seconda chance, si azzera il suo bit e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo una pagina **non viene mai sostituita finché tutte la altre pagine non siano state sostituite o finché non sia stata data loro una seconda chance**. Inoltre, se una pagina viene usata così spesso da mantenere il suo bit a 1, **non verrà mai sostituita**.

Un metodo per implementare l'algoritmo con seconda chance, detto anche ad **orologio** (**clock**), è quello di usare una coda circolare, in cui un puntatore indica qual è la prima pagina da sostituire.



Quando serve un frame, si fa avanzare il puntatore finché non si trova la **pagina vittima** col bit impostato a 0; una volta trovata la pagina vittima la si sostituisce. Nel caso peggiore, in cui tutti i bit siano impostati a 1, il puntatore percorre un ciclo completo della coda dando a **ogni pagina una seconda chance**. In questo caso la sostituzione si riduce a una sostituzione FIFO.

ALGORITMO CON SECONDA CHANCE MIGLIORATO

L'algoritmo con seconda chance si può migliorare considerando i **bit di riferimento** e **di modifica** come una **coppia ordinata**. Le coppie ordinate si dividono come segue:

- **(0,0)** né recentemente usata né modificata – **migliore pagina da sostituire**;
- **(0,1)** non usata recentemente ma modificata – la pagina prima di essere sostituita deve essere scritta in memoria secondaria (**non è una scelta così buona**);
- **(1,0)** usata recentemente ma non modificata – probabilmente la pagina sarà usata di nuovo recentemente;
- **(1,1)** usata recentemente e modificata – probabilmente la pagina sarà usata di nuovo e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Con questo schema, si esamina la coda e si sostituisce la prima pagina che si trova nella coppia minima non vuota. Questa miglioria predilige le pagine modificata al fine di **ridurre** il numero di I/O richiesti.

SOSTITUZIONE DELLE PAGINE BASATA SU CONTEGGIO

Esistono molti altri algoritmi per la sostituzione delle pagine. Per esempio, si potrebbe usare un contatore del numero di riferimenti fatti a ciascun pagina sviluppando uno dei seguenti schemi:

- **Algoritmo di sostituzione delle pagine meno frequentemente usate (Least Frequently Used, LFU)**; richiede che si sostituisca la pagina col conteggio più basso poiché si basa sul concetto che una pagina usata attivamente abbia il conteggio alto. Tuttavia, può capitare che una pagina sia stata usata nella fase iniziale di un processo, quindi ha un conteggio elevato, ma poi non viene più utilizzata; quindi, rimane in memoria anche se non serve;
- **Algoritmo di sostituzione delle pagine più frequentemente usate (Most Frequently Used, MFU)**; è basato sul fatto che, probabilmente, la pagina col contatore più basso sia stata appena inserita e, giustamente, non è stata ancora usata.

Gli algoritmi MFU e LFU non sono molti usati siccome sono onerosi da realizzare e non approssimano bene la sostituzione OPT.

.ALLOCAZIONE DEI FRAME

Occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. La domanda da porsi è: **quanti frame allochiamo al sistema operativo, quanti frame allochiamo a ogni processo?**

Vi sono molte strategie per l'allocazione dei frame. Si può richiedere che il sistema operativo assegni tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato viene sfruttato per la paginazione utente; oppure si potrebbero riservare **sempre** 3 frame liberi in modo tale che quando si verifica un page fault sia sempre disponibile un frame in cui trasferire la pagina. Le strategie sono varie ma tutte si fondano sullo stesso concetto: **al processo utente si assegna qualsiasi frame libero**.

NUMERO MINIMO DI FRAME

Le strategie per l'allocazione dei frame sono soggette a parecchi vincoli; ad esempio, non si possono assegnare più frame di quanti ne siano disponibili, a meno della condivisione delle pagine. Inoltre, è necessario assegnare almeno un numero minimo di frame.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, al decrescere del numero dei frame allocati aumenta il tasso di page fault, con un conseguente rallentamento dell'esecuzione dei

processi. Inoltre, va ricordato che, quando si verifica un page fault prima che sia stata completata l'esecuzione di un'istruzione, questa va riavviata. In linea definitiva, i frame disponibili **devono** essere in numero sufficiente per contenere **tutte le pagine** cui ogni singola istruzione può far riferimento.

Il numero minimo di frame è definito dall'architettura del calcolatore mentre il numero massimo è definito dalla quantità di memoria fisica disponibile.

ALLOCAZIONE GLOBALE E ALLOCAZIONE LOCALE

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione si distinguono in: **algoritmi di sostituzione globale** e **algoritmi di sostituzione locale**.

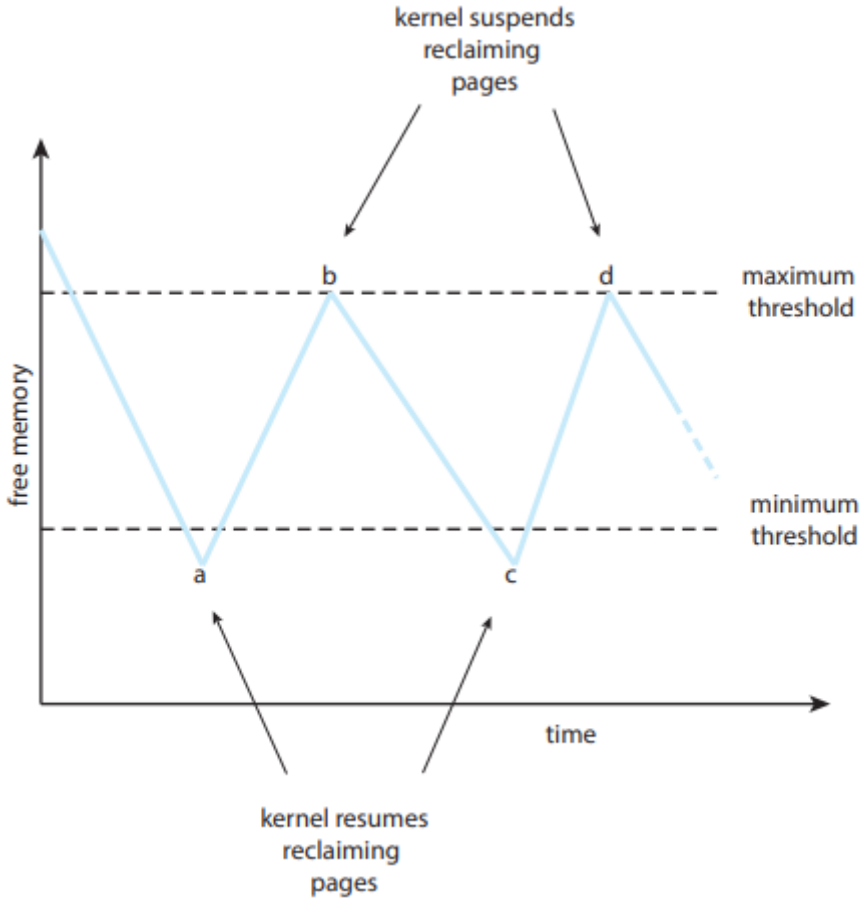
La sostituzione globale consente che per un processo si scelga un frame per la sostituzione dall'insieme di **tutti** i frame, anche se quel frame è allocato a un altro processo; **un processo può sottrarre un frame a un altro processo**. La sostituzione globale **aumenta** il numero di frame assegnati a un determinato processo, a condizione che gli altri processi non scelgano di sostituire i suoi frame.

La sostituzione locale richiede che per ogni processo si scelga un frame solo dal **proprio** insieme di frame e dunque, il numero di blocchi di memoria/frame assegnati al processo non varia.

La sostituzione globale risente di un problema: un processo **non può** controllare il proprio tasso di page fault siccome l'insieme delle pagine allocate per quel processo non dipende solo dalla paginazione del processo stesso, ma anche dalla paginazione degli altri processi. Con la sostituzione locale, invece, questo problema non vi è in quanto l'insieme delle pagine risente esclusivamente della paginazione del processo stesso; tuttavia, la sostituzione locale può limitare il processo in quanto non gli consente l'accesso ad altre aree di memoria. In linea generale, la sostituzione globale è più usata in quanto genera una maggiore produttività.

Di seguito verrà illustrata una possibile strategia per implementare la sostituzione globale, la quale attiverà la sostituzione delle pagine **NON** quando la lista dei frame liberi si svuota, ma quando la dimensione di questa scende al disotto di una certa soglia. In questa maniera, si garantisce che ci sia sempre sufficiente memoria libera per soddisfare nuove richieste.

La strategia è illustrata nell'immagine che segue:



Come detto poc'anzi, lo **scopo** è di **mantenere la quantità di memoria libera al di sopra di una soglia minima**: quando si scende al di sotto di questa soglia, viene avviata una routine del kernel che inizia a recuperare le pagine da tutti i processi. Tale routine è nota come **reaper** ("mietitrice"). Quando la quantità di memoria raggiunge la quota massima, la reaper viene sospesa, per poi essere riavviata quando la memoria libera scenderà nuovamente al di sotto della soglia minima.

La routine reaper può adottare qualsiasi algoritmo di sostituzione, ma in genere utilizza l'algoritmo LRU.

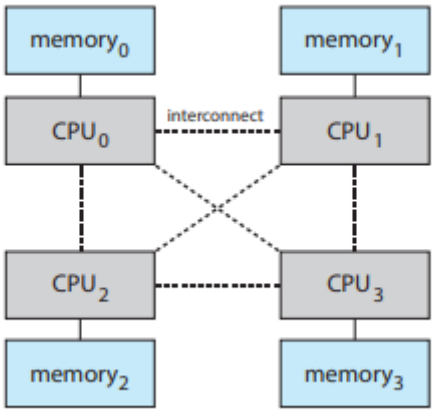
Cosa succede quando la routine reaper non è in grado di mantenere la memoria libera al di sopra della soglia minima? In questo caso, la reaper inizia a recuperare le pagine in modo più aggressivo, per esempio sospendendo l'algoritmo della seconda chance e utilizzando un FIFO puro. Un altro esempio, si può verificare in Linux: quando la quantità di memoria libera scende a livelli bassi, una routine detta **OOM** (**Out-Of-Memory killer**) seleziona un processo da killare per liberare memoria; il processo viene selezionato in base al suo punteggio OOM, calcolato in base alla % di memoria utilizzata dal processo. Più è alta la % di utilizzo della memoria, più è alto il punteggio OOM, più è probabile che il processo venga killato.

L'aggressività del reaper e i valori di soglia minima e massima variano da sistema a sistema.

BANANA: Un page fault si verifica quando una pagina non ha una mappatura valida nello spazio d'indirizzamento di un processo. Si distinguono due tipi di page fault: **page fault principali**, o **major page fault**, e **page fault secondari**, o **minor page fault**. Un major page fault si verifica quando si fa un riferimento a una pagina che non è presente in memoria. Per risolvere, bisogna caricare la pagina desiderata dalla memoria ausiliaria in un frame libero e aggiornare la tabella delle pag riine. I minor page fault si verificano quando un processo non ha una mappatura logica per una pagina, anche se la pagina si trova in memoria. I minor page fault possono verificarsi per uno dei due seguenti motivi: il processo fa riferimento a un libreria condivisa presente in memoria ma non ha una mappatura verso questa; in questo caso basta aggiornare la tabella delle pagine, oppure il minor page fault si verifica quando una pagina viene tolta dal processo e inserita nella lista dei frame liberi prima che questa potesse essere azzerata e allocata per un altro processo.

ACCESSO NON UNIFORME ALLA MEMORIA

Fino a questo momento abbiamo trattato la memoria virtuale assumendo che l'architettura alla base fosse un'architettura **UMA** (**Uniform Access Memory**), in cui è possibile accedere a ogni sezione della memoria in tempo costante. Sui sistemi **NUMA** (**NonUniform Memory Access**) non è così. I sistemi NUMA sono sistemi multiprocessore e, ogni processore può accedere con tempi diversi ad alcune sezioni della memoria centrale. Il tempo di accesso di ogni processore alla memoria è determinato dalla sua "posizione". Ad esempio, nell'immagine seguente la CPU_0 accede più velocemente alla $memoria_0$ rispetto alla CPU_1 e così via.



Di conseguenza, la gestione della memoria diventa più complicata; infatti, le decisioni su quali frame di pagina memorizzare in quale posizione possono condizionare in modo significativo le prestazioni di un sistema NUMA. A tale

scopo, gli algoritmi di allocazione sono stati modificati per adattarsi il meglio possibile ai sistemi NUMA e il loro scopo è quello di disporre i frame di memoria allocati il "**più vicino possibile**" (latenza minima) alla CPU su cui è in esecuzione il processo. Pertanto, quando un processo incorre in un page fault gli viene assegnato il frame più vicino possibile alla CPU su cui è in esecuzione. Per incrementare il numero di cache hit e ridurre i tempi di accesso alla memoria, i sistemi NUMA devono garantire la **prelazione del processo sul processore** (riferimento al Cap. 5 sullo scheduling).

La questione diventa ancora più complicata con i thread siccome alcuni sistemi, come Linux, impediscono ai thread di migrare tra domini diversi. Ogni sistema operativo gestisce la gestione dei thread su sistemi NUMA in una maniera specifica (ad esempio, Solaris usa gli **lgroup** (**gruppi di località**)).

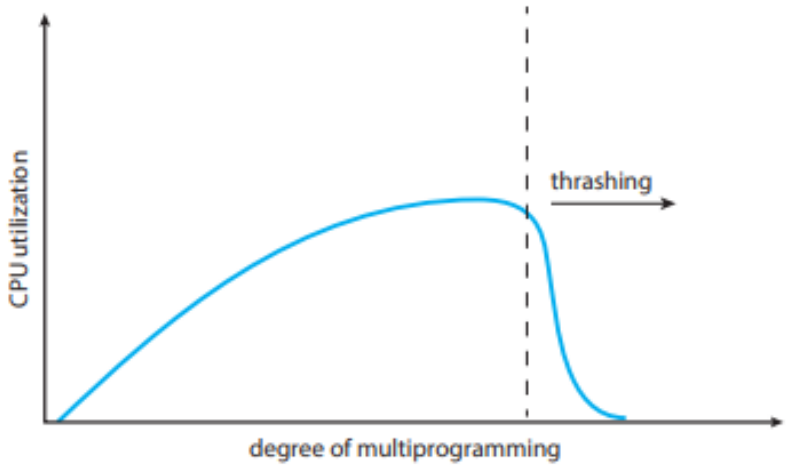
.THRASHING

Si consideri un processo che non dispone di un numero di frame sufficiente. Se non vi sono abbastanza frame per ospitare le pagine del working set, il processo incorre rapidamente in un page fault. A questo punto si deve sostituire qualche pagina, ma se tutte le pagine sono attive, si sostituirà una pagina che sarà ben presto necessaria e di conseguenza si verificheranno parecchi page fault perché si sostituiscono pagine che saranno immediatamente riportate in memoria. Quest'attività di intensa paginazione è nota come **thrashing** ed impatta negativamente sulle prestazioni del sistema in quanto un processo in thrashing spende più tempo per la sostituzione delle pagine che per l'esecuzione dei processi.

CAUSE DEL THRASHING

Il thrashing causa notevoli problemi di prestazioni. Il sistema operativo controlla l'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di page fault, cui segue la sottrazione di frame ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi dei page fault con conseguente sottrazione di frame ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della CPU diminuisce. Lo scheduler della CPU rileva questa riduzione dell'utilizzo della CPU e aumenta il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine

ai processi in esecuzione, causando ulteriori page fault e allungando la coda per il dispositivo di paginazione. Come risultato l'utilizzo della CPU scende ulteriormente, e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. Si è in una situazione di thrashing che fa precipitare la produttività del sistema. Il tasso dei page fault aumenta enormemente ed aumenta di conseguenza il tempo di accesso alla memoria. I processi non svolgono alcun lavoro poiché spendono tutto il tempo nell'attività di paginazione.



Gli effetti di questa situazione si possono limitare usando un **algoritmo di sostituzione locale**, o **algoritmo di sostituzione per priorità**. Con la sostituzione locale, se un processo entra in thrashing, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Tuttavia, il problema non è completamente risolto. I processi in thrashing rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un page fault aumenta a causa dell'allungamento della coda media d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso in memoria aumenta anche per gli altri processi. Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame "servano" a un processo si impiegano diverse tecniche. L'approccio del working-set comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo approccio definisce il **modello di località** d'esecuzione del processo. Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. **Una località è un insieme di pagine usate attivamente insieme**. Generalmente, un programma è formato da parecchie località diverse, che sono sovrapponibili. Per esempio, quando s'invoca una procedura essa definisce una nuova località. In questa località si fanno i riferimenti alla memoria per le istruzioni

della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali.

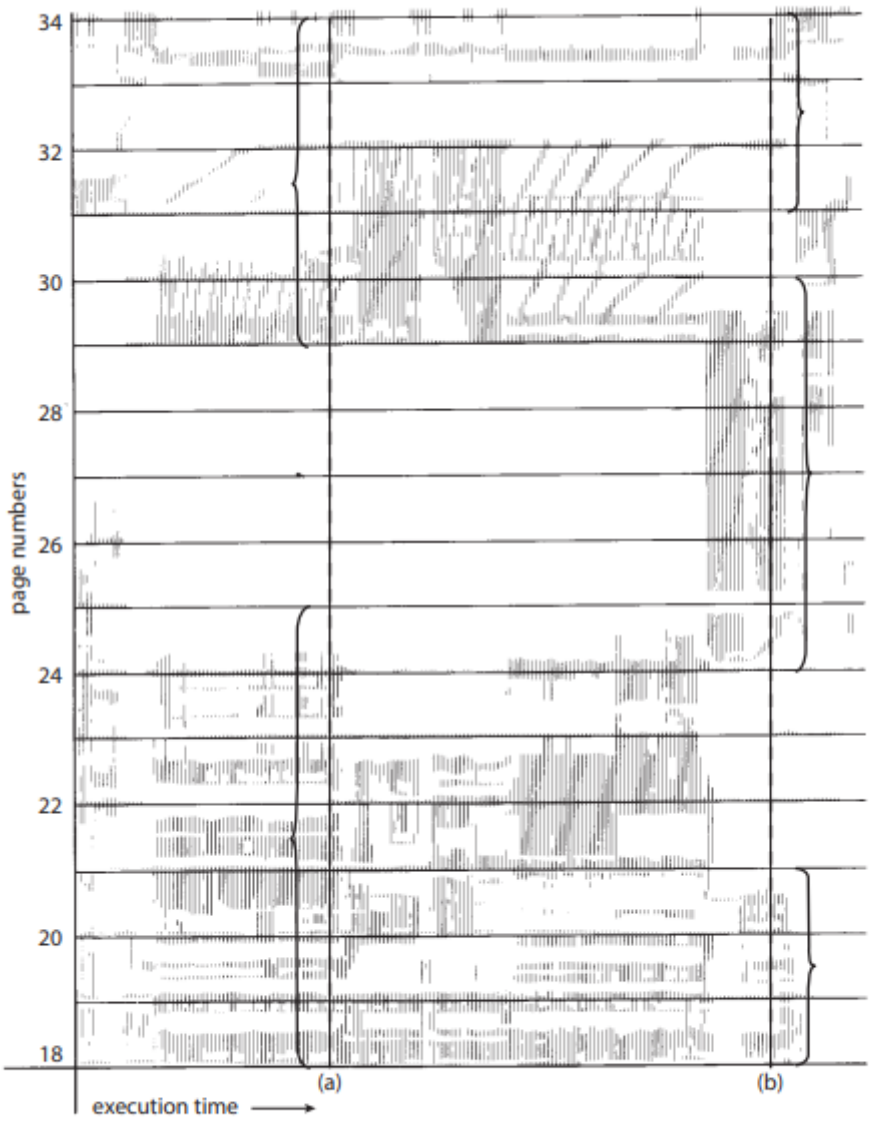
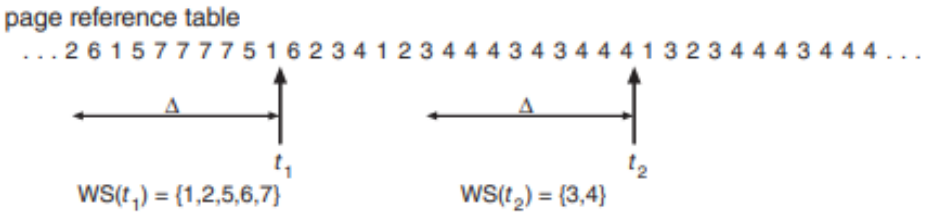


Figure 10.21 Locality in a memory-reference pattern.

Le località sono definite dalla struttura del programma e dalle relative strutture dati.

MODELLO DEL WORKING SET

Il modello del **working set** è basato sull'ipotesi di località. Questo modello usa un parametro, Δ, per definire la **finestra del working set**. L'idea consiste nell'esaminare i più recenti Δ riferimenti alle pagine. L'insieme di pagine nei più recenti Δ riferimenti è il **working set**.



Se una pagina è in uso attivo si trova nel working set; se non è più usata esce dal working set Δ unità di tempo dopo il suo ultimo riferimento. Quindi, il working set non è altro che un'approssimazione della località del programma.

La precisione del working set dipende dalla scelta del valore di Δ. Se Δ è troppo piccolo non include l'intera località, se è troppo grande può sovrapporre più località. Al limite, se Δ è infinito il working set coincide con l'insieme di pagine cui il processo fa riferimento durante la sua esecuzione. La caratteristica più importante del working set è la sua dimensione. Ogni processo usa attivamente le pagine del proprio working set. Se la richiesta totale è maggiore del numero totale di frame liberi (D > m), si avrà thrashing, poiché alcuni processi non dispongono di un numero sufficiente di frame.

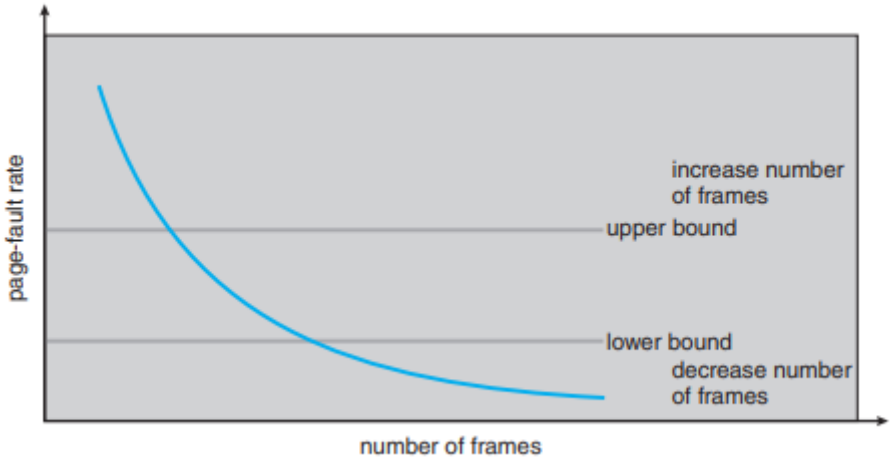
Una volta scelto Δ, l'uso del modello del working set è abbastanza semplice. Il sistema operativo controlla il working set di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo working set. Se i frame ancora liberi sono in numero sufficiente, si può iniziare un altro processo. Se la somma delle dimensioni dei working set aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Questa strategia impedisce il thrashing, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della CPU.

Poiché la finestra del working set è dinamica, la difficoltà di questo modello è tener traccia degli elementi che compongono il working set stesso.

FREQUENZA DEI PAGE FAULT

Il modello del working set, nonostante il suo successo, rappresenta un modo goffo per controllare il thrashing. La strategia basata sulla **frequenza dei page fault** (**page fault frequency, PPF**) è più diretta.

Il problema specifico è la **prevenzione** del thrashing. La frequenza dei page fault in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza è eccessiva, il processo necessita di più frame, viceversa, se la frequenza è eccessivamente bassa, il processo potrebbe disporre di troppi frame. Di norma, viene fissato un limite superiore e un limite inferiore.



Quando la frequenza eccede il limite superiore, si alloca al processo un altro frame; se la frequenza scende al di sotto del limite inferiore, si sottrae un frame al processo.

.ALTRE CONSIDERAZIONI

Le due scelte principali nella progettazione dei sistemi di paginazione riguardano l'algoritmo di sostituzione e l'algoritmo di allocazione, ma ci sono anche altri fattori da considerare.

PREPAGINAZIONE

Una caratteristica ovvia per un sistema di paginazione su richiesta puro, consiste nell'alto numero di page fault che si verificano all'avvio di un processo. Questa situazione è dovuta al tentativo di portare la località iniziale in memoria. La stessa situazione si può presentare anche in altri momenti. La **prepaginazione** rappresenta un tentativo di prevenire questo alto livello di paginazione iniziale. Alcuni sistemi operativi, e in particolare Solaris, applicano la prepaginazione ai frame dei file di dimensioni modeste.

In un sistema che usi il modello del working set, per esempio, a ogni processo si può associare una lista delle pagine contenute nel suo working set. Se occorre sospendere un processo a causa di un'attesa di I/O oppure dell'assenza di frame liberi, si memorizza il suo working set. Al momento di riprendere l'esecuzione del processo (perché l'I/O è terminato o un numero sufficiente di frame liberi è divenuto disponibile), prima di riavviare il processo, si riporta in memoria il suo intero working set.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che deve essere inferiore al costo per servire i corrispondenti page faults. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Inoltre, la prepaginazione di un programma eseguibile può essere difficile perché non è sempre chiaro quali pagina debbano essere portate in memoria.

DIMENSIONE DELLE PAGINE

È raro che chi progetta un sistema operativo per un calcolatore esistente possa scegliere le dimensioni delle pagine. Tuttavia, se si devono progettare nuovi calcolatori, occorre stabilire quali siano le dimensioni migliori per le pagine. Come s'intuisce non esiste un'unica dimensione migliore, ma più fattori sono a sostegno delle diverse dimensioni. Le dimensioni delle pagine sono invariabilmente potenze di 2, in genere comprese tra 4096 (2^{12}) e 4.194.304 (2^{22}) byte.

Un fattore da considerare nella scelta delle dimensioni di una pagina è la **dimensione della tabella delle pagine**. Per un dato spazio di memoria virtuale, diminuendo la dimensione delle pagine aumenta il numero delle stesse e quindi la dimensione della tabella pagine. Per una memoria virtuale di 4 MB (2^{22}), ci sarebbero 4096 pagine di 1024 byte, ma solo 512 pagine di 8192 byte. Poiché ogni processo attivo deve avere la propria copia della tabella delle pagine, conviene che le pagine siano grandi.

D'altra parte, la memoria è utilizzata meglio se le pagine sono piccole. Se a un processo si assegna una porzione della memoria che comincia alla locazione **00000** e continua fino alla quantità di cui necessita, è molto probabile che il processo non termini esattamente sul limite di una pagina, lasciando inutilizzata (le pagine sono unità di allocazione) una parte della pagina finale (frammentazione interna).

Un altro problema è dovuto al tempo richiesto per leggere o scrivere una pagina. Il tempo di I/O è dato dalla somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è proporzionale alla quantità trasferita, ossia alla dimensione delle pagine; tutto ciò farebbe supporre che siano preferibili pagine piccole. Il tempo di trasferimento è normalmente molto piccolo se è confrontato con il tempo di latenza e il tempo di posizionamento. A una velocità di trasferimento di 2 MB al secondo, per trasferire 512 byte s'impiegano 0,2 millisecondi. D'altra parte, il tempo di latenza è di circa 8 millisecondi e quello di posizionamento 20 millisecondi. Perciò, del tempo totale di I/O (28,2 millisecondi), solo l'1 per cento è attribuibile al trasferimento effettivo. Raddoppiando le dimensioni delle pagine, il tempo di I/O aumenta solo fino a 28,4 millisecondi. S'impiegano 28,4 millisecondi per leggere una sola pagina di 1024 byte, ma 56,4 millisecondi per leggere la stessa quantità di byte su due pagine di 512 byte l'una. Quindi, per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori.

Tuttavia, con pagine di piccole dimensioni si dovrebbe ridurre l'I/O totale, poiché si migliorano le caratteristiche di località. Pagine di piccole dimensioni permettono di adattarsi con maggior precisione alla località del programma. Con pagine di piccole dimensioni è possibile avere una migliore **risoluzione**, che permette di isolare solo la memoria effettivamente necessaria. Se le dimensioni delle pagine sono maggiori, occorre allocare e trasferire non solo quanto è necessario, ma tutto quel che è presente nella pagina, a prescindere dal fatto che sia o meno necessario. Quindi dimensioni più piccole dànno come risultato una minore attività di I/O e una minore memoria totale allocata.

Occorre considerare anche altri fattori, come la relazione tra la dimensione delle pagine e quella dei settori del mezzo di paginazione. Non esiste una risposta ottimale al problema considerato, ma la tendenza generale è quella di aumentare la dimensione delle pagine.

PORTATA DELLA TLB

Il **tasso di successi** (**hit ratio**) di una TLB si riferisce alla percentuale di traduzioni di indirizzi virtuali risolte dalla TLB anziché dalla tabella delle pagine. Il tasso di successi è evidentemente proporzionale al numero di elementi della TLB, e un modo per migliorarlo è aumentare il numero di voci nella TLB. Tuttavia, la memoria associativa che si usa per costruire le TLB è costosa e consuma molta energia.

Una metrica collegata al tasso di successi, detto **portata della TLB** (**TLB reach**), esprime la quantità di memoria accessibile dalla TLB, ed è dato semplicemente dal numero di elementi moltiplicato per la dimensione delle pagine. Un metodo per aumentare la portata della TLB consiste nell'aumentare la dimensione delle pagine oppure nell'impiegare pagine di diverse dimensioni questo però richiederebbe che la gestione della TLB venga svolta dal sistema operativo e non dall'hardware.

TABELLA DELLE PAGINE INVERTITA

La tabella delle pagine invertita contiene informazioni su quale pagina di memoria virtuale è memorizzata in ciascuna pagina fisica, la tabella delle pagine invertita riduce la quantità di memoria fisica necessaria a memorizzare tali informazioni. Tuttavia, essa non contiene le informazioni complete sullo spazio degli indirizzi logici di un processo, che sono necessarie se una pagina a cui si è fatto riferimento non è correntemente presente in memoria; la paginazione su richiesta richiede tali informazioni per elaborare le eccezioni di page fault. Per disporre di tali informazioni è necessaria una tabella delle pagine esterna per ciascun processo. Queste tabelle sono simili alle ordinarie tabelle delle pagine di ciascun processo e contengono le informazioni relative alla locazione di ciascuna pagina virtuale.

Sfortunatamente, in questo modo un primo page fault può far sì che il gestore della memoria virtuale generi un altro page fault quando carica in memoria la tabella esterna delle pagine per individuare la pagina virtuale nel backing store. Questo caso particolare richiede un'accurata gestione da parte del kernel del sistema operativo e causa un ritardo nell'elaborazione della ricerca della pagina.

STRUTTURA DEI PROGRAMMI

La paginazione su richiesta deve essere trasparente per il programma utente; spesso, l'utente non è neanche a conoscenza della natura paginata della memoria. In altri casi, però, le prestazioni del sistema si possono migliorare se l'utente (o il compilatore) è consapevole della sottostante presenza della paginazione su richiesta.

Come esempio limite, ma esplicativo, ipotizziamo pagine di 128 parole. Si consideri il seguente frammento di programma scritto in C la cui funzione è inizializzare a 0 ciascun elemento di una matrice di 128 × 128 elementi. Il tipico codice è il seguente:

```
int i, j;
int[128][128] data;
for (j = 0; j < 128; j++)
for (i = 0; i < 128; i++)
data[i][j] = 0;
```

Occorre notare che l'array è memorizzato per riga, vale a dire che è disposto in memoria secondo l'ordine data[0][0], data[0][1], ..., data[0][127], data[1][0], data[1][1], ..., data[127][127]. In pagine di 128 parole, ogni riga occupa una pagina; quindi, il frammento di codice precedente azzera una parola per pagina, poi un'altra parola per pagina, e così via. Se il sistema operativo assegna meno di 128 frame a tutto il programma, la sua esecuzione causa 128 × 128 = 16.384 page fault. D'altra parte, cambiando il codice in

```
int i, j;
int[128][128] data;
for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
                data[i][j] = 0;
```

si azzerano tutte le parole di una pagina prima che si inizi la pagina successiva, riducendo a 128 il numero di page fault. Un'attenta scelta delle

strutture dati e delle strutture di programmazione può aumentare la località e quindi ridurre il tasso di page fault e il numero di pagine del working set. Una buona località è quella di uno stack, poiché l'accesso avviene sempre alla sua parte superiore. Una tabella hash, invece, è progettata proprio per distribuire i riferimenti, causando una località non buona. Naturalmente, la località dei riferimenti rappresenta soltanto una misura dell'efficienza d'uso di una struttura dati. Altri fattori rilevanti sono rapidità di ricerca, numero totale dei riferimenti alla memoria e numero totale delle pagine coinvolte.

In uno stadio successivo, anche il compilatore e il loader possono avere un effetto notevole sulla paginazione. La separazione di codice e dati e la generazione di un codice rientrante significano che le pagine di codice si possono soltanto leggere e quindi non vengono modificate. Nel sostituire le pagine non modificate, non occorre scriverle in memoria ausiliaria. Il loader può evitare di collocare procedure lungo i limiti delle pagine, sistemando ogni procedura completamente all'interno di una pagina. Le procedure che si invocano a vicenda più volte si possono "impaccare" nella stessa pagina. Questa forma di impaccamento è una variante del problema del bin-packing della ricerca operativa: cercare di impaccare i segmenti di dimensione variabile in pagine di dimensione fissa, in modo da ridurre al minimo i riferimenti tra pagine diverse. Un metodo di questo tipo è utile soprattutto per pagine di grandi dimensioni.

APPROFONDIMENTI

SISTEMI OPERATIVI LIBERI E OPEN-SOURCE

Lo studio dei sistemi operativi è stato facilitato dalla disponibilità di un vasto numero di programmi **liberi** e **open-source**, includendo anche i sistemi operativi stessi. La differenza tra un sistema operativo libero e open-source è la seguente: il sistema operativo libero mette a disposizione il codice sorgente ed è dotato di una licenza che ne consente l'uso, la modifica e la redistribuzione senza costi; mentre un sistema operativo open-source mette a disposizione il codice sorgente ma non offre necessariamente tale licenza. **I software liberi sono tutti open-source, ma i software open-source non sono tutti liberi**.

Diverso è il discordo per i software **proprietari**, come Windows di Microsoft che ne limita l'uso e ne protegge attentamente il codice sorgente.

VIRTUALIZZAZIONE - INTRODUZIONE

La **virtualizzazione** permea tutti gli aspetti della computazione. Le macchine virtuali sono un ottimo esempio di questa tendenza. In genere per mezzo di una macchina virtuale i sistemi operativi e le applicazioni ospiti vengono

eseguiti in un ambiente che appare loro come l'**hardware nativo**, ma che inoltre li **protegge**, li **gestisce** e li **limita**.

L'idea fondamentale che sta dietro a una macchina virtuale è quella di astrarre l'hardware di un singolo computer in diversi ambienti di esecuzione differenti, creando così l'illusione che ogni ambiente distinto sia in esecuzione su un proprio computer privato.

L'implementazione della macchina virtuale coinvolge diverse componenti. Alla base vi è l'**host**, il sistema hardware che gestisce le macchine virtuali. Il **gestore della macchina virtuale** (**virtual machine manager**, **VMM**) o **hypervisor**, crea e gestisce le macchine virtuali fornendo un'interfaccia che è identica all'host (**eccetto nel caso della paravirtualizzazione**). A ogni processo **guest** viene fornita una copia virtuale dell'host.

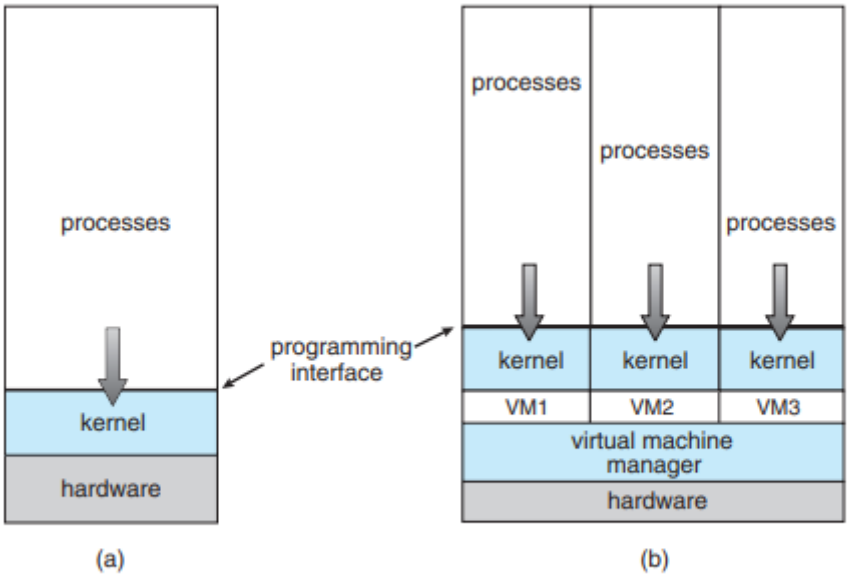


Figure 18.1 System models. (a) Nonvirtual machine. (b) Virtual machine.

L'implementazione di un VMM è molto varia. Tra le opzioni disponibili vi sono le seguenti:

- Soluzioni basate su hardware che forniscono il supporto per la creazione e la gestione della macchina virtuale tramite **firmware**. Questi VMM, noti come **hypervisor di tipo 0**, si ritrovano comunemente nei mainframe e nei server di medie/grandi dimensioni;
- Software assimilabili a sistemi operativi per fornire la virtualizzazione. Questi VMM sono noti come **hypervisor di tipo 1**;
- Applicazioni che si eseguono su sistemi operativi standard, ma forniscono funzionalità di VMM per sistemi operativi guest (VMware, Parallels e Oracle Virtual-Box). Queste applicazioni sono **hypervisor di tipo 2**;

- **Paravirtualizzazione**, una tecnica in cui il sistema operativo guest viene modificato per lavorare in collaborazione col VMM al fine di ottimizzare le prestazioni;
- **Virtualizzazione dell'ambiente di programmazione**, in cui i VMM non virtualizzano l'hardware ma creano un sistema virtuale ottimizzato;
- **Emulatori** che consentono ad applicazioni scritte per un certo ambiente hardware di funzionare su un ambiente hardware differente;
- **Contenitori di applicazioni**, che non forniscono una completa virtualizzazione ma offrono solo alcune funzionalità di virtualizzazione sperando le applicazioni dal sistema operativo.

STORIA

Le macchine virtuali hanno fatto la loro comparsa sul mercato nel 1972 con i mainframe IBM. La virtualizzazione era fornita dal sistema operativo IBM VM sui cui concetti fondamentali si basano molti altri sistemi contemporanei.

IBM VM/370 divideva un mainframe in più macchine virtuali, ciascuna con il proprio sistema operativo. La difficoltà principale consisteva nella gestione dei sistemi di dischi che si incontrava quando si volevano gestire più macchine virtuali di quanti dischi rigidi si avessero a disposizione. Per affrontare il problema, si decise di fornire dei **dischi virtuali**, chiamati **minidisk**, identici ai dischi fisici eccetto per la dimensione. Il sistema implementava ogni minidisk allocando le tracce di cui il minidisk aveva bisogno sui dischi fisici.

I requisiti di virtualizzazione richiedono:

- **Fedeltà**. Un VMM deve fornire un ambiente per i programmi identico alla macchina originale;
- **Prestazioni**. I programmi in esecuzione all'interno di un ambiente virtualizzato soffrono di un calo di prestazioni;
- **Sicurezza**. Il VM deve essere in completo controllo delle risorse di sistema.

VANTAGGI E CARATTERISTICHE

Diversi vantaggi rendono la virtualizzazione attraente. La maggior parte di questi è essenzialmente legata alla capacità di condividere lo stesso hardware, ma eseguire diversi ambienti differenti contemporaneamente.

Un importante vantaggio è che il sistema host **viene protetto** dalle macchine virtuali, e le macchine virtuali stesse sono **protette l'una dall'altra**. Ogni macchina virtuale è **completamente isolata** dalle altre.

Da questo isolamento totale, deriva un grande svantaggio, ovvero la difficoltà nel condividere le risorse tra le varie macchine. Gli approcci principali usati per

raggirare questa difficoltà consistono: nel condividere un volume del file system o di definire una rete di macchine virtuali implementata via software.

Una caratteristica comune alla maggior parte delle implementazioni della virtualizzazione è la capacità di **congelare**/**sospendere** una macchina virtuale in esecuzione. Questa capacità è resa possibile dai VMM che permettono di effettuare **copie istantanee** del/i guest/s. La copia può essere usata per creare una nuova VM o per spostare una VM da una macchina all'altra rimanendo inalterato il suo stato. Il guest sospeso potrà poi riprendere l'esecuzione dal punto in cui era stato congelato.

I sistemi virtuali vengono utilizzati per la ricerca e lo sviluppo sui sistemi operativi. La virtualizzazione, tra i suoi strumenti, presenta il **templating**, in cui un'immagine standard della macchina virtuale viene salvata e utilizzata come sorgente per più macchine virtuali in esecuzione.

La virtualizzazione, oltre a migliorare l'utilizzo delle risorse, ne migliora la gestione. Alcuni VMM includono una funzionalità di migrazione in tempo reale (**live migration**), usata principalmente nei sistemi virtuali/sistemi distribuiti e che consente di spostare un guest in esecuzione da un server fisico all'altro senza interromperne il funzionamento e le connessioni attive.

La virtualizzazione si sta diffondendo soprattutto nell'ambito della distribuzione e gestione delle applicazioni (es. la manutenzione simultanea dei data-center) e ha gettato le basi per molti altri progressi nella realizzazione, gestione e monitoraggio dei sistemi informatici. Il **cloud computing**, si basa sulla virtualizzazione, e permette l'erogazione di risorse come la CPU, la memoria ecc. come servizi attraverso la rete. Questo ha permesso a molti utenti di connettersi a macchine remote e accedere alle loro applicazioni come se fossero locali. Questa pratica può aumentare la sicurezza, perché non ci sono dati memorizzati sui dischi locali dell'utente, e può abbassare il costo delle risorse di calcolo dell'utente in quanto quest'ultimo, per poter usufruire del cloud computing, necessita di un elaboratore in grado di connettersi a una rete e di visualizzare l'immagine del guest in esecuzione in remoto (attraverso un protocollo, ad es. **RDP**).

Autori: Angelo Nazzaro, //Autore: Francesco Granozio, Roberto della Rocca.