

# **Appunti Sistemi Operativi**

## **Prof. Alberto Finzi**

**DISCLAIMER: I SEGUENTI APPUNTI NON SONO COMPLETI E NON COPRONO INTERAMENTE IL PROGRAMMA. PER UNO STUDIO APPROFONDITO SI CONSIGLIA DI STUDIARE IL LIBRO “Sistemi operativi. Concetti ed esempi” (Silberschatz)**

### **Informazioni generali**

Studio: Via Claudio Palazzina 3/A, stanza 3.23

Email: [alberto.finzi@unina.it](mailto:alberto.finzi@unina.it) (specificare nell'oggetto sempre SO1)

Esame: scritto con domande aperte, quiz a risposta multipla ed esercizi; orale che parte dalla discussione della prova scritta. 2 ore.

Sistemi operativi di riferimento: unix / linux

### **Programma**

- **Introduzione ai Sistemi Operativi:** Evoluzione, strutture, architetture, componenti
- **Processi:** Il concetto di processo, stati dei processi, funzioni del kernel, algoritmi di schedulazione, sincronizzazione dei processi e deadlock
- **Memoria:** Gestione memoria principale, swapping, partizione, segmentazione e paginazione, memoria virtuale
- **Sistemi I/O:** Architetture e dispositivi di I/O, interfacce, sottosistema per l'I/O del nucleo, etc.
- **Dati permanenti:** File, metodi di allocazione, directory e metodi di accesso, file system etc.
- **Cenni su sistemi distribuiti:** file system distribuito, socket

### **Argomenti rilevanti**

Legenda:

non sottolineato → conoscenza concettuale

**grassetto e sottolineato** → conoscenza approfondita con domande ed esercizi

**grassetto** → conoscenza approfondita (senza esercizi)

#### **● 1. Introduzione ai Sistemi Operativi:**

- Evoluzione, strutture, architetture, componenti
- **Concetto di multiprogrammazione e multiutente**
- **Concetto di Nucleo: modalità operativa duale, bit di modalità, etc.**
- **Gestione interruzioni**
- **Gerarchie di memorie**
- **Servizi dei Sistemi Operativi**
- **Chiamate di Sistema: esercizi**
- **Interfaccia utente: Bash shell e comandi (esercizi, chiamata di sistema exec)**
- **Struttura dei Sistemi Operativi: Monolitici, stratificati, microkernel, modulari, ibridi**

- **2.Processi:**

- Definizione di processo
- Allocazione processo in memoria
- Stati di un processo
- Descrittore di processo
- Commutazione tra processi
- Schedulazione di processi:
  - Diagrammi di accodamento, Schedulazione di breve/lugo/medio
- Operazioni su processi e chiamate di Sistema:
  - fork, wait, exec, exit - posix (esercizi)
- Comunicazione interprocesso:
  - Message passing e shared memory
  - **Problema produttori consumatori**
  - IPC POSIX (esercizi): pipe, mmap, socket

- **3.Thread e concorrenza:**

- Definizione di thread: thread vs processo
- Multicore programming: Concorrenza vs Parallelismo
- Legge di Amdahl (esercizi)
- User thread e kernel thread
- Modelli multithreading: Many-to-One, One-to-One, Many-to-Many, Two level model
- Librerie thread: Thread Linux/Windows/Java, Pthreads (esercizi), Creazione, cancellazione, thread specific data
- Threading implicito
- Problematiche thread: exec e thread, thread e segnali, cancellazione thread

- **4. Scheduling della CPU**

- Concetti di base:
  - CPU brust, I/O brust.
  - Criteri di scheduling.
  - Prelazione.
  - dispatcher
- Tipi di scheduling: FCFS, SJF, shortest-remaining-time-first, scheduling con priorità, Round Robin, Round Robin con priorità, Code Multiple, Code Multiple con Feedback (Esercizi)
- Thread scheduling
- Scheduling multiprocessore: Concetti e problematiche
- Real-time scheduling:
  - Task periodici (rate monotonic scheduling).
  - Earliest deadline first.
  - Esercizi.
- Esempi sistemi operativi : Linux scheduling, windows scheduling
- Valutazione algoritmi di scheduling

- **5. Sincronizzazione:**
  - Concetto di corsa critica
  - Problema della sezione critica. requisiti
  - Gestione della corsa critica
  - Soluzione di Peterson
  - Hardware per sincronizzazione (Lock, barriere, istruzioni atomiche, Test and set, compare and swap)
  - Mutex, semafori. Esercizi
  - Deadlock starvation. Esercizi
  - Monitor e variabili di condizione. Esercizi. Implementazione monitor.
  - Inversione di priorità. Concetto, esempio
- **6. Deadlock:**
  - Concetti principali
  - Caratterizzazione deadlock
  - Grafo di allocazione di risorse
  - Metodi di gestione del deadlock: Prevention, avoidance, Algoritmo del grafo delle risorse, Algoritmo del banchiere, Detection, Recovery
- **7. Memoria principale:**
  - Concetti principali
  - Binding degli indirizzi
  - Indirizzi fisici e logici
  - Memory Management Unit
  - Caricamento dinamico
  - Protezione
  - Allocazione. Algoritmi ed Esercizi
  - Frammentazione interna/esterna: concetti, esempi, esercizi
  - Metodi per il Paging: Metodi ed Esercizi
  - Swapping (concetti)
  - Segmentazione (concetti)
- **8. Memoria virtuale:**
  - Concetti principali
  - Spazio degli indirizzi virtuali
  - Demand paging: Page fault e instruction restart. Frame liberi. Performance.  
Ottimizzazioni: copy-on-write. Sostituzione delle pagine (esercizi). Thrashing.
  - Locality model (esercizi). Modello working set
  - Memoria Kernel. Buddy e Slab.
  - Linux, Windows, Solaris

- **9. Memoria di massa:**

- Concetti principali (HDDs e NVMe)
- Mapping indirizzi e struttura disco
- Schedulazione del disco. Algoritmi ed esercizi
- Gestione del Disco
- Connessioni a dispositivi di memorizzazione
- Architetture RAID: Concetti. metodi.

- **10. Sistemi I/O:**

- Concetti principali
- Polling/Interrupt
- Direct Memory Access (DMA), come funziona. concetti.
- Interfacce per I/O. Caratteristiche dispositivi I/O
- Sottosistema I/O del Kernel . Strutture dati per gestione I/O (concetti)

- **11. File system:**

- Concetti principali
- File, attributi, directory
- Operazioni sui file
- Chiamate di Sistema in Linux (esercizi). Open, read, write, close, mmap
- Comandi per la gestione di file e directory (esercizi shell linux)
- Implementazione . Concetti principali: Struttura, Strutture dati. Directory
- Metodi di allocazione dei file
- Gestione dello spazio libero
- Efficienza e prestazione
- Gestione errori e recupero
- Dettagli. Partizioni, montaggio, file system virtuale
- File system distribuito

- **12. Linux**

- Overview
- Gestione di processi e thread (task)
- Scheduling
- Gestione della memoria
- Memoria virtuale: Paging, page replacement, free memory
- Caricamento ed esecuzione
- File system (ext)
- Sistema I/O

Informazioni generali

1. INTRODUZIONE AI SISTEMI OPERATIVI

2. PROCESSI

3. THREAD E CONCORRENZA

4. SCHEDULING DELLA CPU

5. SINCRONIZZAZIONE

5. DEADLOCK

## **1.INTRODUZIONE AI SISTEMI OPERATIVI**

### **COS'E' UN SISTEMA OPERATIVO?**

Un sistema operativo è un programma (software) che agisce come intermediario tra l'utente e l'hardware di un calcolatore e che ne gestisce le risorse hardware.

### **Obiettivi di un sistema operativo sono:**

- Eseguire i programmi utente e semplificare l'interazione con il calcolatore;
- Rendere efficace ed efficiente l'utilizzo del calcolatore;
- Utilizzare l'hardware in modo efficiente.

La progettazione di un nuovo sistema operativo è un compito impegnativo che richiede, in via preliminare, una chiara definizione degli obiettivi del sistema. In base a tali obiettivi si selezionano le possibili strategie e si individuano i relativi algoritmi.

### **COMPONENTI DI UN SISTEMA DI CALCOLO**

Un sistema di elaborazione si può suddividere in quattro componenti: hardware, sistema operativo, programmi applicativi e utenti.

- 1) **Hardware:** fornisce al sistema le risorse elaborative fondamentali (CPU, memoria, dispositivi I/O..)
- 2) **Sistema operativo:** controlla l'hardware e ne coordina l'utilizzo da parte dei programmi applicativi per gli utenti.
- 3) **Programmi applicativi:** definiscono il modo in cui si usano le risorse per la risoluzione dei problemi computazionali degli utenti
- 4) **Utenti:** possono essere persone, altri processi, altri calcolatori, etc.

### **DEFINIZIONE DI UN SISTEMA OPERATIVO**

Dal punto di vista del calcolatore, il sistema operativo è il programma più strettamente correlato al suo hardware. In tale contesto è possibile considerare un sistema operativo come:

- **Assegnatore di risorse:** un sistema elaborativo dispone di risorse utili per la risoluzione di un problema come tempo di CPU, spazio di memoria, spazio per la memorizzazione di file, dispositivi di I/O e così via. Il sistema operativo agisce come gestore di tali risorse. Di fronte a richieste di risorse numerose ed eventualmente conflittuali, il sistema operativo deve decidere come assegnarle agli specifici programmi e utenti affinché il sistema elaborativo operi in modo equo ed efficiente.
- **Programma di controllo:** gestisce l'esecuzione dei programmi utenti in modo da impedire che si verifichino errori o che il calcolatore sia usato in modo scorretto. Si occupa soprattutto del funzionamento e del controllo dei dispositivi di I/O.
- **Macchina estesa:** esso fornisce un'astrazione fornendo un ambiente coerente ed uniforme per l'esecuzione dei programmi, attraverso l'utilizzo di interfacce che in qualche modo "nascondono" l'hardware sottostante.

### ***In realtà non esiste una definizione universalmente accettata di S.O.***

Funzioni comuni, di controllo e assegnazione delle risorse, sono state racchiuse in un unico insieme di programmi: il sistema operativo.

- Una definizione più comune è quella secondo cui il sistema operativo è il solo programma che funziona sempre nel calcolatore e che gestisce le risorse, generalmente chiamato **kernel (nucleo)**. Il resto include i **programmi di sistema**, utilità associate al sistema operativo, ma che non fanno parte del kernel, e i **programmi applicativi**, che includono tutti i programmi non correlati al funzionamento del sistema.

## DEFINIZIONE DI KERNEL

Il **kernel** è il **nucleo del sistema operativo** il quale gestisce le risorse essenziali del calcolatore: la CPU, la memoria, le periferiche, etc. Tutto il resto, anche l'interazione con l'utente, è ottenuto tramite programmi eseguiti dal kernel, che accedono alle risorse hardware tramite delle richieste a quest'ultimo.

- Il kernel è il solo programma ad essere eseguito in **modalità privilegiata**, con il completo accesso all'hardware, gli altri programmi vengono eseguiti in modalità protetta.
- I programmi possono essere eseguiti in due modalità:
  - **kernel mode** (superuser), in cui il programma può accedere all'hardware senza restrizioni
  - **user mode**, in cui il programma viene eseguito in modalità protetta

## ORGANIZZAZIONE CALCOLATORE

Il Sistema Operativo è strettamente legato all'architettura del calcolatore. Una o più **CPU** ed i **controllori dei diversi dispositivi di I/O (driver)** sono connessi ad un **canale (bus)** che permette l'accesso alla **memoria condivisa**. La CPU e i controllori possono eseguire operazioni in parallelo competendo per l'accesso alla memoria.

- **CPU:** componente hardware che esegue le istruzioni
- **Processore:** chip che contiene una o più CPU
- **Unità di calcolo (core):** unità di elaborazione di base della CPU
- **Multicore:** che include più unità di calcolo sulla stessa CPU (processore)
- **Multiprocessore:** che include più processori

## SISTEMI CON INTERRUPT

Il programma d'avviamento del sistema (bootstrap), contenuto nella ROM (firmware) carica nella memoria centrale RAM il sistema operativo ne avvia l'esecuzione; a tale scopo individua e carica nella memoria il kernel del sistema operativo. Una volta caricato e in esecuzione, il kernel può iniziare a offrire servizi al sistema e agli utenti. Alcuni servizi vengono forniti all'esterno del kernel da programmi di sistema caricati in memoria durante l'avviamento. Questi processi di sistema, o **demoni**, restano in esecuzione per tutto il tempo in cui è in esecuzione il kernel. In UNIX il primo processo di sistema è “**init**”, che avvia diversi altri demoni. Una volta che questa fase è completata il sistema risulta completamente inizializzato e attende che si verifichi qualche **evento**.

- Un **evento** è di solito segnalato da un'**interruzione dell'attuale sequenza d'esecuzione della CPU (interrupt)**, che può essere causata dall'hardware o da un programma.

Le **interruzioni possono essere di 2 tipi:**

- **hardware:** causata da un dispositivo.
- **software (eccezioni o trap):** causate da errori software o richieste per un servizio del s.o oppure per altri errori.
- Se l'**interrupt** è causata dall'**hardware** i controllori dei dispositivi e altri elementi dell'architettura possono inviare alla CPU **segnali d'interruzione**, di solito attraverso il bus di sistema.
- Se l'**interrupt** è di tipo **software**, si genera un'interruzione eseguendo una speciale istruzione detta **chiamata di sistema (system call)**, a volte chiamata anche monitor call).

Quando riceve un segnale d'interruzione, la **CPU** interrompe l'elaborazione corrente e la gestisce. Trasferisce immediatamente l'esecuzione a una locazione fissa della memoria. Di solito, questa locazione contiene l'**indirizzo iniziale della routine di servizio per la gestione dell'interruzione**. Una volta completata l'esecuzione della procedura richiesta, la CPU riprende l'elaborazione precedentemente interrotta.

**Le interruzioni permettono di gestire le operazioni concorrenti: consentono di sovrapporre CPU e operazioni I/O ed evitare busy waiting. Un Sistema Operativo moderno è guidato dalle interruzioni.**

### INTERRUPT E POLLING

Interrupt e polling sono due modalità con cui gli eventi generati dai dispositivi I/O possono essere gestiti dalla CPU.

- Con il **polling**, la CPU tiene traccia delle comunicazioni dei dispositivi di I/O interrogandoli ad intervalli regolari per vedere se hanno bisogno di andare in esecuzione.
- Con **interrupt**, è il dispositivo di I/O ad interrompere la CPU comunicando ad essa che ha bisogno di andare in esecuzione. La CPU poi esegue una routine di servizio per la gestione dell'interruzione.

### GESTIONE DELLE INTERRUZIONI

La gestione delle interruzioni varia da sistema a sistema. Ogni interruzione deve causare il trasferimento del controllo all'appropriata procedura di servizio.

- Inizialmente il dispositivo I/O invia un segnale sulla **Interrupt Request Line (IRQ)** che serve proprio per inviare richieste d'interruzione.
- La CPU ad ogni ciclo controlla la IRQ.
- Esistono due tipi di interruzioni:
  - **nonmaskable**, che devono essere gestite urgentemente e rappresentano errori irreversibili
  - **maskable**, che possono essere gestite anche non nell'immediato

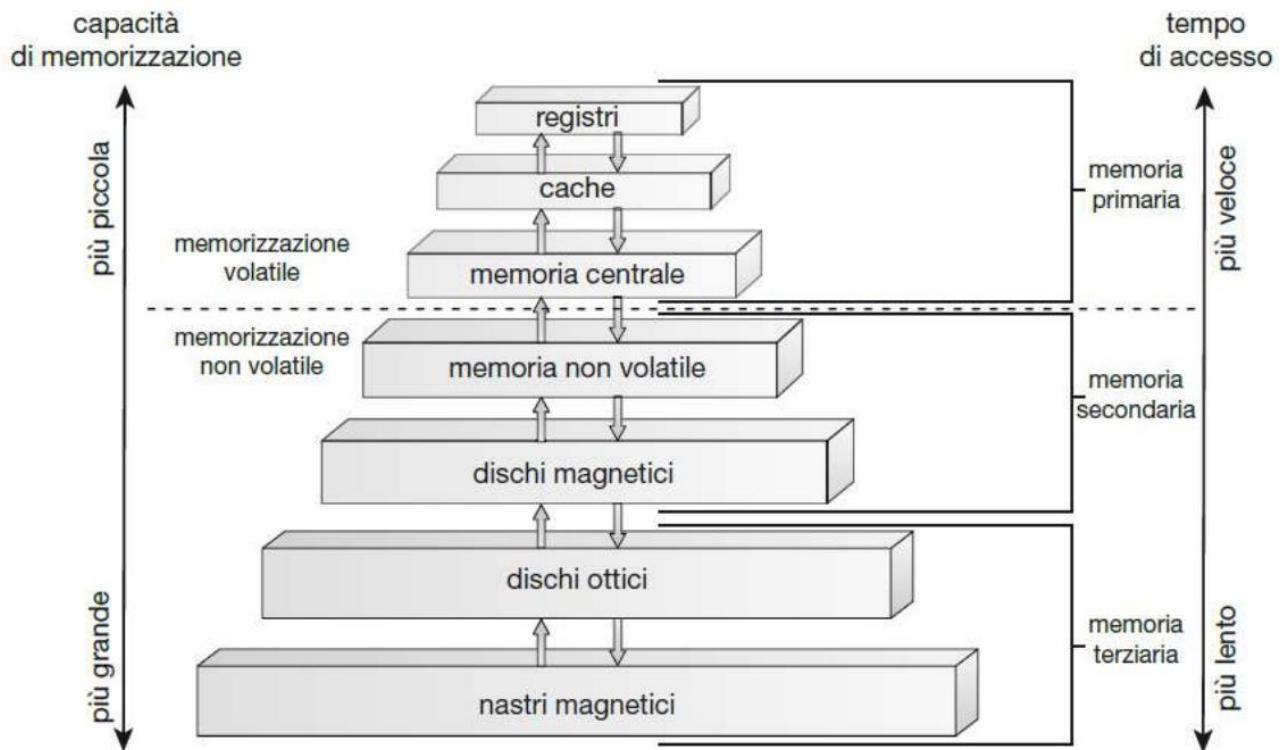
#### Per gestire l'interruzione la CPU:

- Determina il tipo di interruzione
- Passa il controllo alla procedura di servizio fornendone l'indirizzo iniziale.
  - L'indirizzo iniziale lo trova grazie al **vettore delle interruzioni** che associa all'indice del tipo di interruzione avvenuta l'indirizzo iniziale della locazione in cui è memorizzata la procedura di servizio per la gestione di quella interruzione.
- La CPU passa il controllo al **gestore dell'interruzione** il quale salva le informazioni di stato dell'operazione interrotta che stava eseguendo la CPU (registri e program counter)
- La procedura di servizio viene eseguita, determina la causa dell'interruzione e la processa
- Il controllo torna all'operazione interrotta (recuperando lo stato esecutivo)

### STRUTTURE E GERARCHIE DI MEMORIA

- **Memoria Principale:** la CPU vi accedere direttamente. Random access memory (**RAM**). Tipicamente **volatile**.
- **Memoria di sola lettura:** Es. **ROM**, PROM, EPROM, EEPROM (Elettronicamente cancellabili e programmabili)
- **Memoria Secondaria:** estensione della memoria principale che fornisce capacità di memorizzazione **non-volatile**.
  - **Hard-disk drive (hdd):** Disco logicamente diviso in tracce suddivise in settori
  - **Disco a stato solido (ssd):** più veloce, accesso casuale uniforme, tempi di accesso brevi

Le memorie sono organizzate in gerarchie in base a **velocità**, **costo**, **volatilità**. Per velocizzare il disallineamento tra diverse memorie si utilizza il meccanismo del **caching** che consiste nella copia temporanea di dati da memoria più lenta a più veloce; la memoria principale può essere vista come cache per la secondaria. Con questo meccanismo bisogna mantenere la consistenza dei dati, compito di cui si occupa il sistema operativo.



**Figura 1.6** Scala gerarchica dei sistemi di memorizzazione.

### MULTIPROGRAMMAZIONE, TIMESHARING E MULTIUTENZA

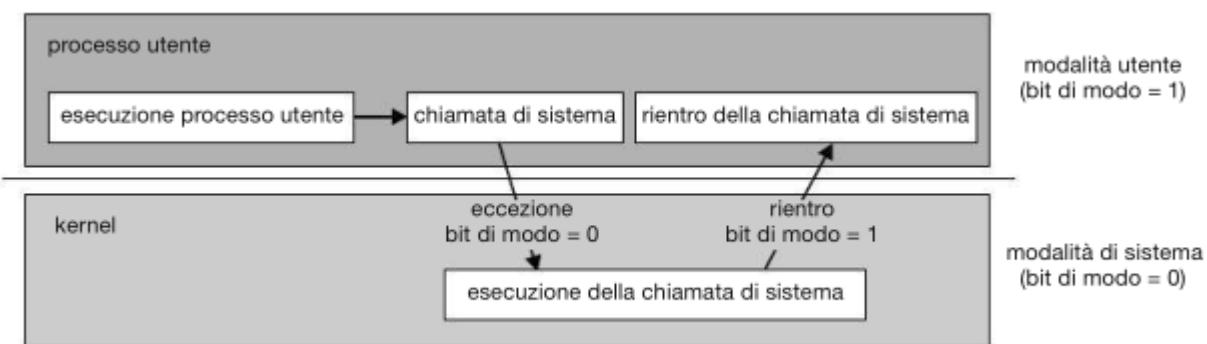
- **Multiprogrammazione:** consiste nel poter lanciare più processi contemporaneamente su una macchina che vengono eseguiti in concorrenza, ossia in competizione tra loro. La CPU è continuamente impegnata e passa da processo a processo. Si risparmia in questo modo il tempo di *busy waiting* della CPU.
  - la multiprogrammazione consente di aumentare la percentuale d'utilizzo della CPU, organizzando le attività computazionali (job) in modo tale da mantenerla in continua attività.
- **Timesharing (o multitasking):** consiste nel fatto che i processi possono condividere il tempo, ossia che un processo non debba essere per forza eseguito in modo sequenziale dall'inizio alla fine, ma si può interrompere la sua esecuzione in caso di interruzioni o eccezioni per eseguirne un altro. Ciò implica la **multiutenza** ossia il fatto che la macchina può essere usata da più utenti che eseguono ognuno il proprio programma. Bisogna inoltre implementare algoritmi di schedulazione della CPU per gestire l'esecuzione in contemporanea dei vari processi.
  - è un'estensione logica della multiprogrammazione; la CPU esegue più lavori commutando le loro esecuzioni con una frequenza tale da permettere a ciascun utente l'interazione col proprio programma durante la sua esecuzione.

## MODALITÀ OPERATIVA DUALE

Sistemi multiprogrammati e multiutente richiedono **meccanismi di protezione**. La modalità operativa duale è un meccanismo di protezione che permette di eseguire le operazioni del sistema in dual-mode; ciò permette al sistema operativo di ‘proteggersi’ e proteggere altre componenti. I due tipi di modalità sono: **user mode** e **kernel mode**.

Ad ogni meccanismo di protezione deve corrispondere un correlato hardware. Infatti per indicare quale sia la modalità attiva, l’architettura della CPU deve implementare un bit, chiamato appunto **bit di modalità**: **kernel mode (0)** o **user mode (1)**. Il sistema operativo ha accesso a questo bit il quale permette di distinguere quando il sistema esegue codice utente o kernel.

All’avviamento del sistema, il bit è posto in kernel mode. Si carica il sistema operativo che provvede all’esecuzione delle applicazioni utenti in modalità user mode. Alcune istruzioni sono privilegiate quindi solo eseguibili in modalità kernel mode. **Inoltre ogni volta che si verifica un’interruzione, un’eccezione o una chiamata di sistema, l’hardware passa dalla user mode alla kernel mode, cioè pone a 0 il bit di modo ed il sistema operativo esegue le istruzioni in modalità privilegiata. Una volta terminata la computazione il sistema ripristina la modalità user mode riportando a 1 il bit di modo.**



## PROTEZIONE CPU: TIMER

Il timer è un meccanismo di protezione che assicura che il sistema operativo mantenga il controllo della CPU. Ciò consiste nell’impedire che un programma utente entri in un ciclo infinito o non restituisca più il controllo al sistema operativo. A tale scopo si usa un timer, che si può programmare affinché invii un segnale d’interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi o variabili. Prima di dare al programma utente il controllo dell’esecuzione, il sistema operativo assegna un valore al timer. Se il programma esaurisce questo intervallo di tempo si genera un’interruzione che causa il trasferimento del controllo al sistema operativo, che può decidere se gestire l’interruzione come un errore fatale o concedere altro tempo al programma. La presenza di un timer garantisce quindi che nessun programma utente possa essere eseguito troppo a lungo.

## PROTEZIONE MEMORIA

Occorre proteggere la kernel mode. Sovrascrivendo il vettore delle interruzioni e le operazioni di servizio si potrebbe trasferire la modalità privilegiata all’utente. Occorrono meccanismi di protezione hardware della memoria. Dunque si stabiliscono indirizzi di memoria a cui un programma può accedere utilizzando due registri:

- Registro base: che contiene il più piccolo indirizzo accessibile da un programma
- Registro limite: che contiene la dimensione del range di indirizzi

Se non si rispettano i valori contenuti in questi registri si genera un’eccezione.

## GESTIONE RISORSE

Uno dei principali servizi offerti dal sistema operativo è quello di gestire le risorse del calcolatore. In particolare si occupa di:

### ❖ Gestione dei processi

Un **processo** è un programma in esecuzione. Il **programma** è una **entità passiva**, il **processo** è una **entità attiva**. Un processo ha bisogno di risorse per svolgere un task: CPU, memoria, I/O, files, dati di inizializzazione. Alla terminazione di un processo occorre il rilascio e recupero delle risorse. Ogni processo ha un suo layout di memoria. Tipicamente i sistemi hanno in esecuzione molti processi, per più utenti, in esecuzione concorrente su una o più CPU che passa da un processo all'altro continuamente.

Il Sistema Operativo fornisce diversi meccanismi per la gestione dei processi, in particolare per:

- **Creazione e cancellazione** di utenti e processi di sistema
- **Sospensione e resume** di processi
- **Sincronizzazione** di processi
- **Comunicazione** tra processi
- **Gestione del deadlock**

### ❖ Gestione della memoria

Per eseguire un programma istruzioni e dati devono essere caricati in memoria. Il sistema operativo deve occuparsi della gestione della memoria; in particolare deve:

- Tenere traccia di quali parti della memoria sono attualmente utilizzate e da chi
- Decidere quali processi (o parti di essi) e dati spostare dentro e fuori alla/dalla memoria
- Allocazione e deallocazione dello spazio di memoria secondo necessità

### ❖ Gestione dei file e archiviazione

Il sistema operativo fornisce una rappresentazione logica e uniforme dell'archiviazione delle informazioni. Effettua un'astrazione delle proprietà fisiche della memoria secondaria definendo un'**unità logica di archiviazione che è il file**.

Il sistema operativo dunque si occupa della gestione del sistema di archiviazione (**File-System**):

- Organizza i files in modo gerarchico in directory
- Controlla accesso ai file per determinare chi può accedere a cosa (permessi e privilegi)
- Creazione e cancellazione file e directory
- Primitive per modifica di file e directories
- Mapping di file nella memoria secondaria
- Backup di file su memoria non-volatile

### ❖ Gestione della cache

Occorre utilizzare il dato più recente, indipendentemente da dove è archiviato nella gerarchia di archiviazione. In ambiente multiprocessore occorre la coerenza della cache perché tutte le CPU abbiano il dato più recente nella loro cache. Il sistema operativo si occupa di mantenere questa coerenza attraverso diversi meccanismi.

### ❖ Gestione dell'I/O

Il sistema operativo si occupa di nascondere all'utente le specificità hardware dei dispositivi di input/output fornendo un'interfaccia uniforme attraverso la quale è possibile comunicare con essi. Ad esempio il sistema operativo linux vede i dispositivi di I/O come file.

## PROTEZIONE E SICUREZZA

Un'altro servizio fondamentale offerto dal sistema operativo è quello di protezione e sicurezza.

- **Protezione:** qualunque meccanismo di controllo di accesso alle risorse per processi e utenti
- **Sicurezza:** consiste nella difesa del sistema da attacchi interni ed esterni

I sistemi operativi si occupano di distinguere gli utenti, per determinare chi può fare cosa:

- Ogni utente viene associato ad un **user ID** che stabilisce la sua identità
- L'User ID viene associato a file e processi per il **controllo di accesso**
- Si possono stabilire **Identificativi di Gruppo (group ID)** i quali definiscono insiemi di utenti con determinati permessi di accesso
- Il sistema operativo inoltre può assegnare una **privilege escalation** temporanea che consente agli utenti di ottenere maggiori permessi di accesso (distinguiamo real user/effective user)

## SERVIZI DEI SISTEMI OPERATIVI

Un sistema operativo offre un ambiente in cui eseguire i programmi e fornire servizi all'utente. I servizi specifici variano secondo il sistema operativo, ma si possono identificare alcune classi di servizi comuni. Tutti i servizi si implementano mediante chiamate di sistema in modalità protetta. Un primo insieme di servizi offre funzionalità utili all'utente.

- **Interfaccia con l'utente**
- **Esecuzione di un programma**
- **Operazioni di I/O**
- **Gestione del file system**
- **Comunicazioni**
- **Rilevamento di errori**

Un secondo gruppo di funzioni del sistema operativo non riguarda direttamente gli utenti, ma assicura il funzionamento efficiente del sistema stesso.

- **Allocazione delle risorse**
- **Contabilità**
- **Protezione e sicurezza**

### ❖ Interfaccia con l'utente

Quasi tutti i sistemi operativi hanno un'interfaccia con l'utente (**UI - User Interface**). Essa può assumere diverse forme. Certi sistemi offrono alcune o anche tutte queste soluzioni.

- **Command-Line (CLI):** è un'interfaccia a riga di comando basata su stringhe che codificano i comandi, insieme a un metodo per inserirli (ad esempio da tastiera).
- **Graphics User Interface (GUI):** è la forma più diffusa ed è un'interfaccia grafica a finestre nella quale sono presenti le icone; è dotata di un dispositivo puntatore per comandare operazioni di I/O e selezionare opzioni dai menu, insieme a una tastiera per inserire del testo.
- **Batch (a lotti):** prevede che comandi e relative direttive siano codificati in file, che vengono poi eseguiti.

### ❖ Esecuzione di un programma

Il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anomalo (indicando l'errore).

### ❖ Operazioni di I/O

Un programma in esecuzione può richiedere un'operazione di I/O che implica l'uso di un file o di un dispositivo di I/O. Il SO deve nascondere la complessità del dispositivo offrendo agli utenti un interfacciamento semplice ed uniforme.

### ❖ Gestione del file system

I programmi richiedono l'esecuzione di operazioni di lettura e scrittura su file, oltre a creare e cancellare file e directory. Essi hanno anche bisogno di creare e cancellare i file, di eseguire la ricerca di un file e disporre di informazioni relative al file stesso. Alcuni sistemi operativi, infine, gestiscono i permessi di accesso ai file.

### ❖ Comunicazioni

In molti casi un processo ha bisogno di scambiare informazioni con un altro processo. Ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati in rete. La comunicazione si può realizzare tramite una **memoria condivisa**, che permette a due o più processi di leggere e scrivere in una porzione di memoria che condividono, o attraverso lo **scambio di messaggi**, in questo caso il sistema operativo trasferisce pacchetti d'informazioni in un formato predefinito tra i vari processi.

### ❖ Rilevamento di errori

Il sistema operativo deve essere sempre capace di rilevare correggere eventuali errori che possono verificarsi nella CPU e nei dispositivi di memoria, nei dispositivi di I/o e in un programma utente. Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo d'errore.

### ❖ Allocazione delle risorse

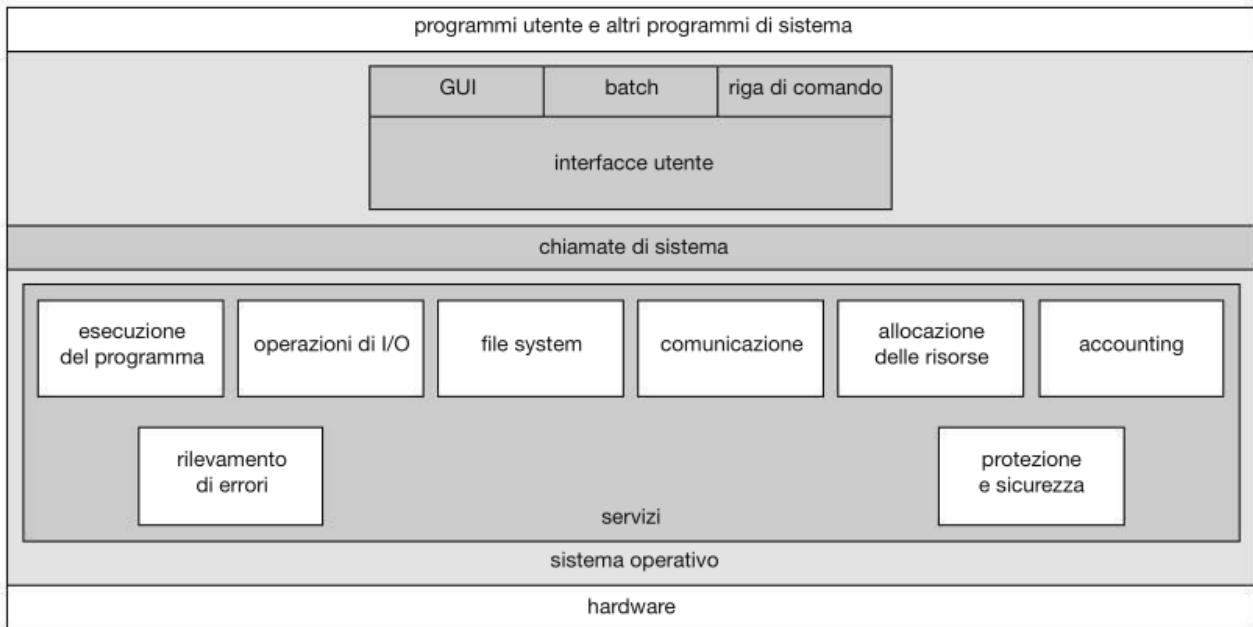
Se sono attivi più utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Molti tipi di risorse: cicli CPU, memoria principale, file, dispositivi I/O.

### ❖ Contabilità

Vogliamo mantenere traccia di quali utenti usano il calcolatore, segnalando quali e quante risorse impiegano.

### ❖ Protezione e sicurezza

La protezione deve garantire che l'accesso da parte di processi ed utenti alle risorse di sistema sia controllato. La sicurezza rispetto ad accessi esterni richiede l'autenticazione dell'utente e prevede la difesa dei dispositivi I/O esterni da accessi non consentiti.



## INTERFACCIA UTENTE

Vi sono due modi fondamentali per gli utenti di comunicare con il sistema operativo. Uno si basa su un'interfaccia a riga di comando o **interprete dei comandi (CLI)**, che permette agli utenti di inserire direttamente le istruzioni che il sistema deve eseguire. L'altro sfrutta un'**interfaccia grafica** con l'utente o **(GUI)**, che serve da tramite tra utente e sistema.

### ❖ Interprete dei comandi (CLI)

Una prima modalità di comunicazione si basa sull'interfaccia a riga di comando (CLI Command Line Interface) o interprete dei comandi. In alcuni sistemi operativi l'interprete dei comandi è una funzionalità compresa nel kernel. In altri, come Windows e UNIX, l'interprete dei comandi è considerato un programma di sistema. Quando i sistemi consentono la scelta tra molteplici interpreti dei comandi, questi vengono definiti **shell**. Le shell forniscono diversi **linguaggi di scripting**. In UNIX e Linux, per esempio, l'utente può scegliere tra svariate shell differenti. **La funzione principale dell'interprete dei comandi consiste nel raccogliere un comando con i suoi argomenti impartito dall'utente ed eseguirlo.** L'esecuzione di un comando implica l'esecuzione di diverse system calls. Si possono eseguire anche script attraverso costrutti di programmazione e linguaggi di scripting interpretati.

### ❖ Interfaccia grafica (GUI)

Un'interfaccia grafica con l'utente o GUI (Graphical User Interface) rappresenta una seconda modalità di comunicazione con il sistema operativo, più **user-friendly**. Infatti, invece di obbligare gli utenti a digitare direttamente i comandi, nella GUI l'interfaccia è costituita da una o più finestre e dai relativi menu, entro cui muoversi con il mouse. C'è un desktop in cui, con il mouse si possono selezionare icone: queste rappresentano programmi, file, directory e funzioni del sistema.

### ❖ Altre interfacce

**Interfacce touch-screen:** gesti per attivare azioni

**Interfacce a comandi vocali**

## UNIX SHELL BASH E COMANDI

- **pwd** visualizza la directory di lavoro corrente
- **ls** visualizza la lista di file nella directory (-- indica un file, **d** directory)
- permessi: **rwx** (utente, gruppo, resto del mondo)
- **cd** directory: cambia directory
- **cd ..** : torna indietro nell'albero
- **mkdir**: crea una directory
- **cat**: permette di vedere il contenuto di un file, anche concatenati uno dopo l'altro
- **more**: paginazione dei file e adatta il contenuto
- **echo**: printa a video, compreso il contenuto di variabili (es. variabili d'ambiente - echo \$HOME)
- **ps**: mostra i processi disponibili all'utente in esecuzione
- **ps -a**: mostra tutti i processi a disposizione
- **man**: spiegazione del comando
- **q**: uscire

### Compilazione:

\$ **gcc** source.c → **a.out** (eseguibile prodotto dal compilatore)

Opzioni comando gcc:

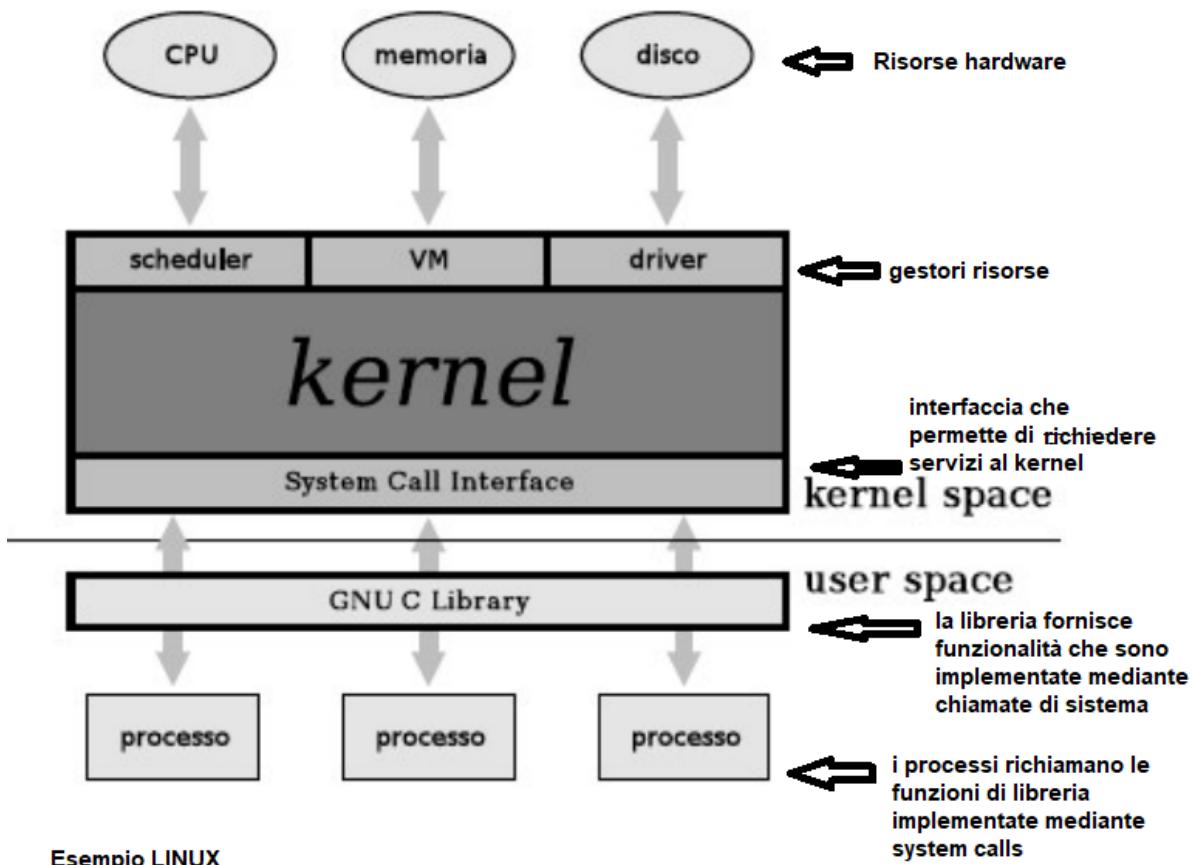
- flag “**-o**”: \$ gcc **-o eseguibile.out sorgente.c**
- flag “**-v**” : Per sapere la versione del compilatore
- flag “**-E**”: Serve per invocare solo il preprocessore
- flag “**-c**”: Per creare il codice oggetto senza chiamare il linker
- flag “**-Wall**”: per aumentare il livello dei messaggi prodotti in fase di compilazione
- \$ **man gcc** (manuale in linea)

- **./a.out** : esegue un file compilato (può prendere anche argomenti in argv es: **./a.out 15**)
- **gedit <nomefile>**: apre l'editor di modifica
- **nano <nomefile>**: apre l'editor di modifica
- **^c** interrompe l'esecuzione del processo (ctrl c)
- **gedit <nomefile> &**: apre l'editor di modifica in background e la shell rimane utilizzabile
- **kill -9 <pid>**: interrompe un processo
- **rm**: cancella un file
- **echo ciao > ciao.txt** : ridirezione su un file dell'output
- **echo mondo >> ciao.txt** : ridirezione su un file dell'output, in concatenazione a quello che già c'è
- **cat nomefile > nomefile2** : fa la copia di un file in un altro con altro nome
- **cd**: torna nella home directory (cd . corrente, cd .. precedente)
- **wc**: conta il numero di parole in un file (-l righe, -c caratteri)
- **cat ciao.txt | wc -l**: concatenazione di comandi mediante pipe
- **Gestione thread in gcc:** 

## SYSTEM CALLS (CHIAMATE DI SISTEMA)

Le system calls sono delle interruzioni di tipo software. Esse sono la modalità attraverso la quale i processi utente chiedono dei servizi al kernel. In pratica consistono in un insieme di funzioni che un programma può chiamare: quando viene richiamata una determinata funzione, viene generata un'interruzione del processo passando il controllo dal programma al kernel. Possiamo quindi definirle come delle **interfacce con cui i programmi possono accedere all'hardware** sottostante al sistema operativo.

- Le chiamate di sistema (system call) quindi costituiscono un'interfaccia per i servizi resi disponibili dal sistema operativo. Tali chiamate sono generalmente disponibili sotto forma di routine scritte in C o C++.



**Esempio di chiamata di sistema:** chiamata di Sistema per copiare il contenuto di un file in un altro file.

Non è raro che un sistema esegua migliaia di chiamate di sistema al secondo. La maggior parte dei programmatori, tuttavia, non si dovrà mai preoccupare di tutti i dettagli delle system calls: infatti, gli sviluppatori usano in genere diverse **API, application programming interface**. Esse specificano un **insieme di funzioni a disposizione del programmatore e dettagliano i parametri necessari all'invocazione di queste funzioni, insieme ai valori restituiti**. Le API differiscono per ogni sistema operativo. Le più diffuse sono:

- la **API di Windows**,
- la **API POSIX** per i sistemi basati sullo standard POSIX (il che include praticamente tutte le versioni di UNIX, Linux e Mac OS). Il compito dello standard POSIX è quello di definire alcuni concetti base che vanno seguiti durante la realizzazione del sistema operativo: definisce lo standard per le API dei sistemi operativi Unix.
- la **API Java** per applicazioni eseguite dalla macchina virtuale Java.

Un programmatore ha accesso a un'API tramite una libreria di codice fornita dal sistema operativo. Per programmi in c in ambienti Unix e Linux tale libreria si chiama **libc**.

### ❖ Esempio di API Standard POSIX: read()

Consideriamo la funzione **read()** disponibile in Unix e Linux. L'API per questa funzione si può ottenere digitando: **man read** da riga di comando. Una descrizione di questa API è la seguente:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
```

Valore restituito	Nome della funzione	Parametri
-------------------	---------------------	-----------

Un programma che utilizza la **read()** deve includere il file **unistd.h** che, tra le altre cose, definisce i tipi di dato **ssize\_t** e **size\_t**. I parametri passati alla **read()** sono i seguenti:

- **int fd** — è un intero che rappresenta il descrittore del file da leggere, ottenuto tramite **open()**
- **void \*buf** — un buffer nel quale vengono messi i dati letti
- **size\_t count** — il massimo numero di byte da leggere e inserire nel buffer

Quando una **read()** è completata con successo viene restituito il numero di byte letti (con il tipo **ssize\_t**).

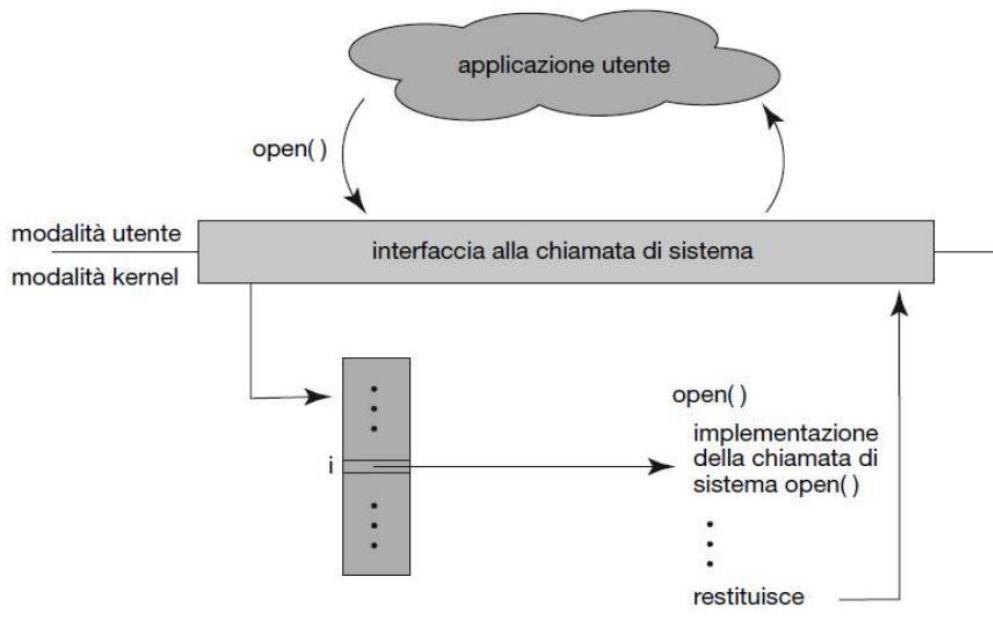
La **read()** restituisce **0** in caso di **fine del file** e **-1** quando si è verificato un **errore**.

La **open()** restituisce 0 in caso trova un file e -1 se non lo trova.

### ❖ Relazione tra API e chiamata di sistema

Ogni chiamata di sistema è codificata da un numero presente in una tabella indicizzata, gestita dal **system-call interface**. L'interfaccia invoca la chiamata nel kernel e restituisce lo status e valore di ritorno.

- La **system-call interface traduce la funzione API in un indice nella tabella delle chiamate di sistema e l'indice individuato va a puntare sull'implementazione della chiamata di sistema corrispondente a livello kernel. La chiamata viene mandata in esecuzione e una volta finita restituisce lo status e valore di ritorno.**



### ❖ Passaggio dei parametri

Alle funzioni api vengono anche passati dei parametri, oppure vengono letti. Esistono **3 metodi** generali utilizzati per **passare i parametri al sistema operativo**:

- **i parametri vengono passati direttamente nei registri:** è la modalità più veloce ma spesso i parametri sono troppi.
- **i parametri vengono inseriti in un blocco o tabella presente in memoria** e viene passato l'indirizzo del blocco che li contiene in un registro. (Linux e Solaris)
- **i parametri vengono inseriti nello stack dal programma** e estratti dallo stack dal sistema operativo.

### ❖ Tipologie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in 6 categorie principali:

- **controllo dei processi**
  - creare e terminare processi
  - caricamento ed esecuzione
  - ottenere e settare gli attributi del processo
  - gestione delle attese di tempo
  - gestione attese di eventi/segnali
  - allocazione di memoria
  - dump memory per errori
  - per il debug
  - meccanismi di regolazione per gestire l'accesso a dati condivisi
- **gestione dei file**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- **gestione dei dispositivi**
  - request device, release device
  - read, write
  - get device attributes, set device attributes
  - logically attach or detach devices
- **gestione delle informazioni**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **comunicazioni**
  - create, delete communication connection
  - send, receive messages if message passing model to hostname or process name
  - Shared-memory creazione ed accesso a regioni di memoria condivisa
  - trasferire informazione di stato
- **protezione**
  - Controllo accesso a risorse
  - Gestione dei permessi
  - Gestione accesso utenti

ESEMPIO DI CHIAMATE DI SISTEMA DI WINDOWS E UNIX		
	Windows	UNIX
<b>Controllo dei processi</b>	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
<b>Gestione dei file</b>	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
<b>Gestione dei dispositivi</b>	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
<b>Gestione delle informazioni</b>	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
<b>Comunicazione</b>	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
<b>Protezione</b>	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

### ❖ Controllo di processi (esempio chiamata di sistema in POSIX)

```
#include <sys/types.h>
# include <unistd.h>

pid_t getpid(void);  identificativo del processo
pid_t getppid(void); identificativo del genitore

pid_t fork(void);           pid_t wait(int *status);
pid_t vfork(void);

pid_t pid;

if ( (pid=fork()) < 0 )
    perror("fork"), exit(1);
else if (pid != 0) {
    // codice del padre
} else {
    // codice del figlio
}
```

- **getpid**: ottiene l'identificativo di un processo in esecuzione (intero)
- **getppid**: ottiene l'identificativo del processo padre di quello in esecuzione (intero)
- **fork**: chiede al sistema di fare una “biforcazione” di un processo ossia una clonazione del processo, creando un processo figlio.

Tutti i processi hanno un padre, a meno di quello iniziale che è il padre di tutti i processi.

- Nel codice la **fork()** restituisce al processo padre il pid del figlio (pid diverso da 0) e al processo figlio restituisce uno 0 perché sa di essere lui il figlio. Dopo la fork abbiamo quindi 2 processi e il codice è splitato in due biforazioni eseguite in parallelo.

## PROGRAMMI DI SISTEMA

I programmi di sistema forniscono un ambiente conveniente per lo sviluppo e l'esecuzione del programma. La maggior parte degli utenti di un SO usa soprattutto i programmi di sistema, meno le effettive chiamate di sistema. Alcuni di essi sono semplicemente interfacce utente per le chiamate di sistema; altri sono notevolmente più complessi. **I programmi di sistema in costante esecuzione si dicono servizi, sottosistemi, demoni. Sono eseguiti in modalità user richiamando i servizi offerti dal kernel attraverso le chiamate di sistema.**

## STRUTTURA DEI SISTEMI OPERATIVI

### ❖ Progettazione ed implementazione di un sistema operativo

Il primo problema abbiamo nella progettazione di un sistema operativo riguarda la definizione degli **obiettivi** e delle **specifiche** del sistema stesso. La progettazione del sistema è influenzata in modo decisivo dalla scelta dell'**architettura fisica** e del tipo di sistema. I requisiti possono essere molto difficili da specificare: in generale distinguiamo tra **obiettivi degli utenti** e **obiettivi del sistema**.

- **Utente:** comodo da usare, facile da imparare, affidabile, sicuro e veloce
- **Sistema:** facile da progettare, implementare e mantenere, nonché flessibile, affidabile, privo di errori ed efficiente

Un principio molto importante è quello che riguarda la distinzione tra **meccanismi** e **politiche (policy)**.

- I meccanismi determinano **come eseguire** qualcosa.
- Le politiche, invece, stabiliscono **che cosa** si debba fare.

Le politiche sono soggette a cambiamenti di luogo o di tempo. Nei casi peggiori il cambiamento di una politica può richiedere il cambiamento del meccanismo sottostante. Sarebbe preferibile disporre di **meccanismi generali**: in questo caso un cambiamento di politica implicherebbe solo la ridefinizione di alcuni parametri del sistema. Le decisioni relative alle politiche sono importanti per tutti i problemi di assegnazione delle risorse. Invece, ogni volta che un problema riguarda il come eseguire qualcosa, occorre definire un meccanismo.

### ❖ Realizzazione di un sistema operativo

I primi sistemi operativi erano scritti in linguaggio assembler. Oggi, anche se alcuni sistemi operativi sono ancora scritti in linguaggio assembler, la maggior parte è scritta in un linguaggio di alto livello come il c o in linguaggi di livello ancora più alto come il c++. In realtà un sistema operativo può essere scritto in più linguaggi, per esempio usando il linguaggio assembler per i livelli più bassi del kernel, il c per routine di livello superiore e il c, il c++ o linguaggi di scripting interpretati come Perl, Python o gli script di shell per i programmi di sistema.

- un sistema operativo scritto in un linguaggio di alto livello è più facile da adattare a un'altra architettura (**porting**).
  - **Emulazione** con CPU diverse e codice non nativo: metodo più lento, non compilazione, ma interpretazione
  - **Virtualizzazione**: sistema operativo compilato in modo nativo per la CPU, che esegue anche sistemi operativi guest compilati in modo nativo

Sebbene i sistemi operativi siano molto grandi, solo una piccola parte del codice assume un'importanza critica per le prestazioni: il gestore degli interrupt, il gestore dell'I/o, il gestore della memoria e lo scheduler della CPU sono probabilmente le procedure più critiche.

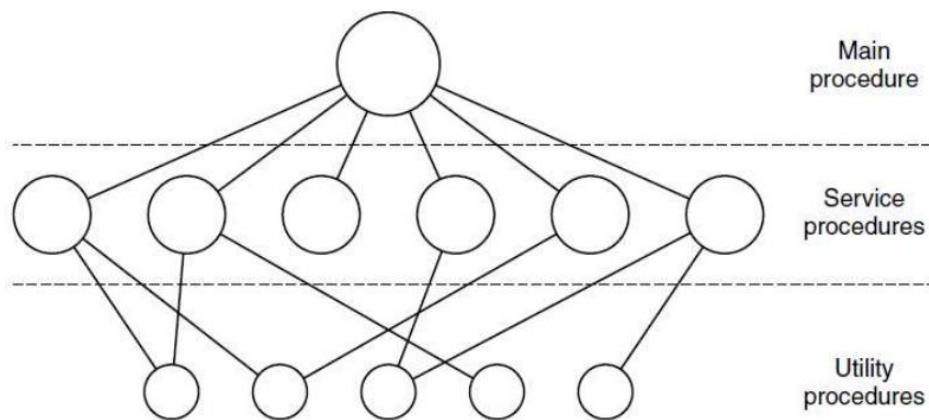
## ❖ Strutture di un sistema operativo

Un Sistema Operativo è un software molto complesso. Esistono vari modi di strutturarlo:

- **Sistemi monolitici**
  - **Struttura semplice - MS-DOS**
  - **Struttura non semplice -- UNIX**
- **Sistemi Stratificati**
- **Sistemi a micro-kernel**
- **Sistemi con moduli**
- **Sistemi ibridi**

## ❖ Sistemi monolitici

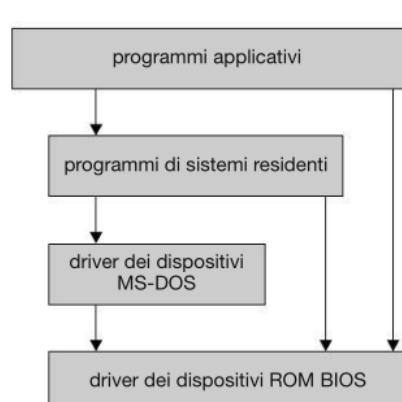
Un sistema operativo è costituito da una collezione di procedure (**moduli**) ognuna delle quali può chiamare qualsiasi altra. Un **sistema monolitico** viene anche chiamato sistema **strettamente accoppiato** (tightly coupled). In questi sistemi ogni volta che dobbiamo aggiungere un nuovo modulo esso dovrà interagire con tanti altri moduli. Dunque sono **molto difficili da sviluppare ed estendere** ma offrono **alte prestazioni** perché ogni modulo dialoga in modo diretto con un altro modulo, senza tanti passaggi tra canali di comunicazione.



### Struttura monolitica semplice → MS-DOS

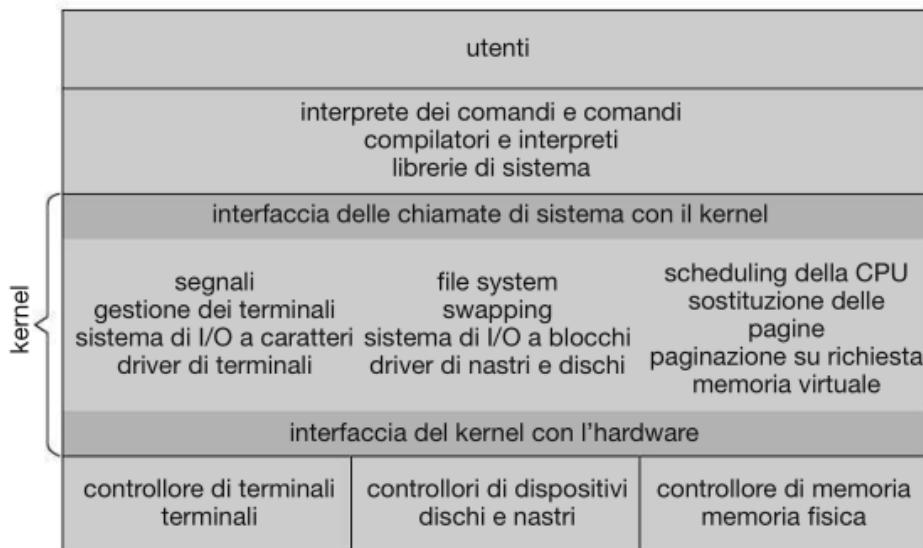
**Non suddiviso in moduli** poiché lo scopo prioritario era fornire la massima funzionalità nel minimo spazio di memoria. **Monoutente e monotask**. Non vi è una netta separazione fra le **interfacce** e i **livelli di funzionalità**, tanto che, per esempio, le applicazioni utente accedono direttamente alle routine di sistema per l'I/O, scrivendo direttamente sul video e sui dischi. Molto **vulnerabile agli errori e agli attacchi dei programmi utenti**, fino al blocco totale del sistema. Il processore Intel 8088, per il quale fu scritto, non distingueva fra modalità utente e di sistema, e non offriva protezione hardware, ciò che non lasciava altra scelta ai progettisti di MS-DOS se non permettere accesso incondizionato all'hardware.

Tutte le unità logiche comunicano tra loro



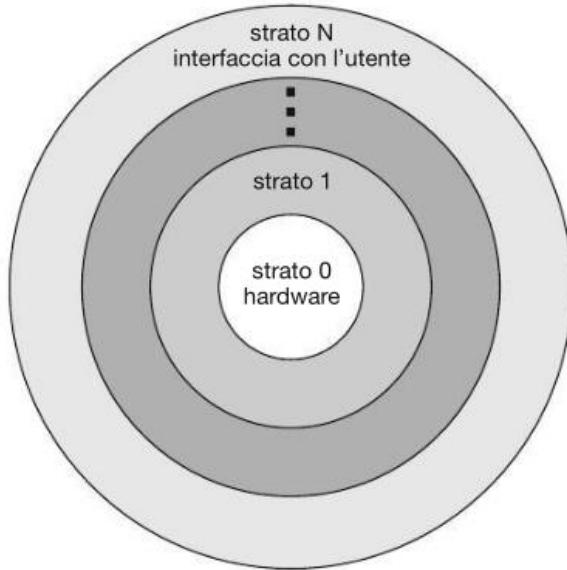
## Struttura monolitica non semplice → Unix

Il sistema consiste di due parti separate, il blocco del **kernel** e i **programmi di sistema**. A sua volta, il kernel è diviso in una serie di interfacce e driver dei dispositivi, aggiunti ed espansi nel corso dell'evoluzione di UNIX. Interfaccia standard per chiamate di sistema (POSIX).



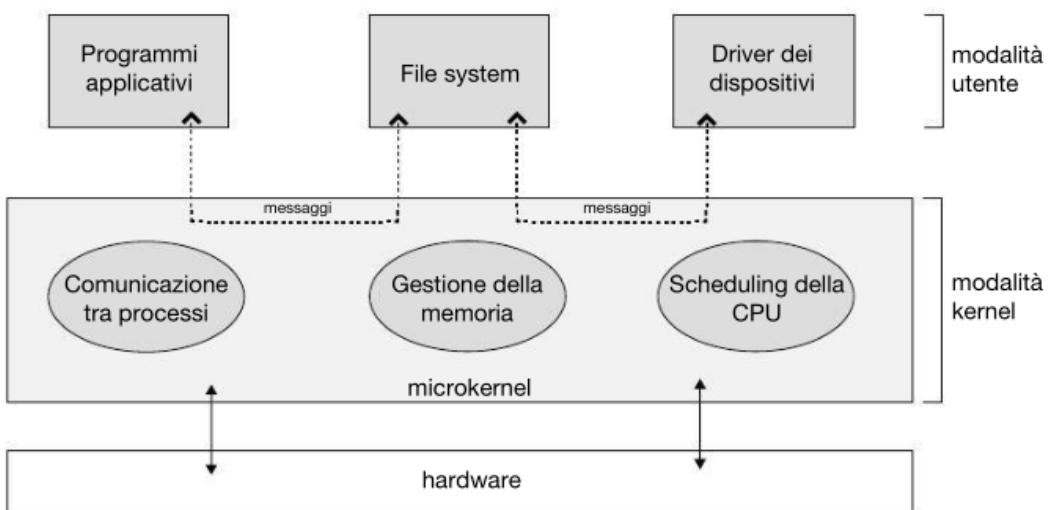
## ❖ Sistemi stratificati

Si rende il sistema operativo modulare con un approccio stratificato. **Il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (strato 0), il più alto all'interfaccia con l'utente (strato N).** Con la modularità ciascun livello utilizza funzioni (operazioni) e servizi dei livelli inferiori. Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e di debugging. La suddivisione in strati semplifica la progettazione e la realizzazione di un sistema. Ogni strato si realizza impiegando unicamente le operazioni messe a disposizione dagli strati inferiori, considerando soltanto le azioni che compiono, senza entrare nel merito di come queste sono realizzate. Di conseguenza ogni strato nasconde a quelli superiori l'esistenza di determinate strutture dati, operazioni e hardware. La principale difficoltà del metodo stratificato risiede nella definizione appropriata dei diversi strati. È necessaria una progettazione accurata, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia. Altro svantaggio con la struttura stratificata è che essa tende a essere meno efficiente delle altre: per effettuare una chiamata di sistema si dovrà comunicare con diversi strati e questo porta overhead.



### ❖ Sistemi a microkernel

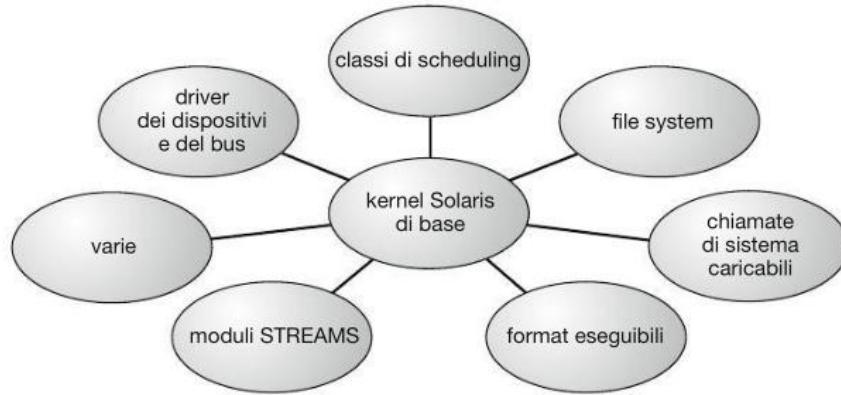
Il kernel viene strutturato in moduli secondo il cosiddetto orientamento a microkernel. Seguendo questo orientamento si progetta il sistema operativo **rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema che li gestiscono**. Ne risulta un **kernel minimale** di dimensioni assai inferiori. Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione. Uno dei **vantaggi** del microkernel è la **facilità di estensione** del sistema operativo: i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel. Poiché è ridotto all'essenziale, se il kernel deve essere modificato, i **cambiamenti da fare sono ridotti**, e il sistema operativo risultante è **più semplice da portare su diverse architetture**. Inoltre offre maggiori garanzie di **sicurezza e affidabilità**, poiché i servizi si eseguono in gran parte come processi utenti, e non come processi del kernel: se un servizio è compromesso, il resto del sistema operativo rimane intatto. Separazione massima tra meccanismi e politiche. **Svantaggio:** Purtroppo i microkernel possono incorrere in **cali di prestazioni** dovuti al sovraccarico indotto dall'esecuzione di processi di sistema in modalità utente.



### ❖ Sistemi a moduli

Fanno utilizzo di moduli del kernel caricabili dinamicamente. In questo contesto, il kernel è costituito da un insieme di componenti di base, integrati poi da funzionalità aggiunte dinamicamente durante l'avvio o l'esecuzione per mezzo di moduli. L'idea di base è che il kernel deve fornire direttamente i servizi principali, mentre gli altri servizi sono implementati in modo dinamico, quando il kernel è in esecuzione. Ogni modulo

può chiamare qualsiasi altro modulo comunicando tramite interfacce.



### ❖ Sistemi ibridi

La maggior parte dei sistemi operativi moderni non sono in realtà un modello puro. Il sistema ibrido combina più approcci per soddisfare le esigenze di prestazioni, sicurezza e usabilità.

- Kernel Linux e Solaris lavorano in un singolo spazio di indirizzi, quindi monolitici, ma modulari per il caricamento dinamico di funzionalità.
- Windows per lo più monolitico, ma con filosofia microkernel per diverse parti del sottosistema
- Apple mac OS ibrido, stratificato

### ❖ Boot di sistema

All'accensione l'esecuzione parte da una locazione di memoria fissata. Il Firmware ROM contiene il codice iniziale di boot. Questo piccolo frammento di codice, bootstrap loader, presente in ROM o EEPROM localizza il kernel, lo carica in memoria e lo avvia. A volte processa a due passi: si individua prima il boot block che è in locazione fissata e poi da esso si recupera il codice in ROM per caricare il bootstrap loader dal disco (BIOS e boot block). La partizione del disco che contiene il boot block è la partizione di avvio (boot partition). Oltre a caricare il kernel il programma di bootstrap inizialmente effettua Power-on self-test (POST) - inizializza l'hardware (CPU, memoria e disposititivi). Poi si individua il kernel, lo si esegue e il sistema operativo va in running eseguendo il primo processo (init).

## 2. PROCESSI

I primi sistemi elaborativi consentivano l'esecuzione di un solo programma alla volta, che aveva il completo controllo del sistema e accesso a tutte le sue risorse. Gli attuali sistemi elaborativi consentono, invece, che più programmi siano caricati in memoria ed eseguiti in modo concorrente. Tale evoluzione richiede un più severo controllo e una maggiore compartimentazione dei vari programmi. Da tali necessità deriva la nozione di processo che in prima istanza si può definire come un programma in esecuzione, e costituisce l'unità di lavoro dei moderni sistemi time-sharing. Un sistema è quindi costituito da un insieme di processi: quelli del sistema operativo eseguono il codice di sistema, i processi utente il codice utente. Tutti questi processi si possono eseguire potenzialmente in modo concorrente e l'uso della cpu (o di più cpu) è commutato tra i vari processi. Il sistema operativo può rendere il calcolatore più produttivo avvicendando i diversi processi nell'uso della cpu.

### ❖ Definizione di processo

- **Un processo è un'entità di computazione coerente che rappresenta un programma in esecuzione.**
- **A differenza di un programma, che è un'entità passiva archiviata su disco (file eseguibile), un processo è un'entità attiva dato che è in esecuzione.**

L'esecuzione di un processo è **sequenziale**: per poter essere eseguito esso **richiede risorse** (CPU, memoria, file, dispositivi I/O), anche per l'esecuzione di un **compito (task)**. Un programma può chiamare molti processi e ad esso possono corrispondere vari thread (sottoprocessi). I sistemi operativi sono il risultato dell'esecuzione di più processi concorrenti. I processi si dividono in processi di sistema e processi utente. Un programma diventa un processo allorquando il file eseguibile è caricato in memoria.

### ❖ Allocazione processo in memoria

Lo **stato di un processo allocato in memoria** e in esecuzione è dato da:

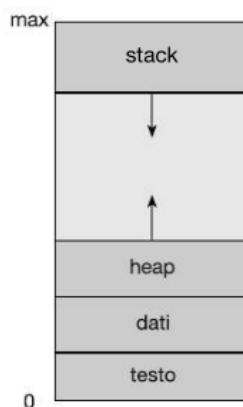
- **stato corrente del program counter**
- **contenuto dei registri del processore**
- **layout di memoria**

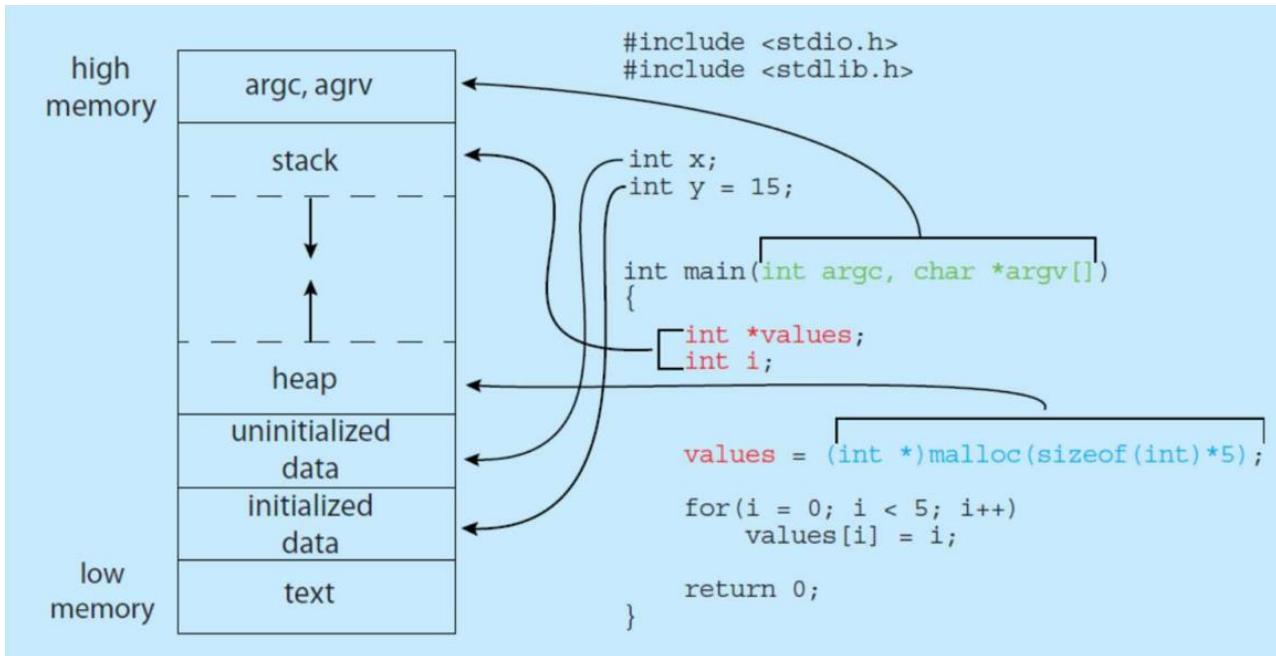
Il **layout di memoria** di un processo è costituito da:

- **sezione testo:** contiene il codice del programma in esecuzione
- **sezione dati:** contiene le variabili globali che gestisce il processo
- **uno proprio stack:** contiene le variabili temporanee come i parametri di una procedura, gli indirizzi di ritorno per le chiamate a funzione e le variabili locali
- **heap:** contenente memoria dinamicamente allocata a run time durante l'esecuzione di un task

La **sezione testo e la sezione dati** hanno una **dimensione fissata** in memoria.

Lo **stack e l'heap** del processo hanno una **dimensione variabile** a run time.

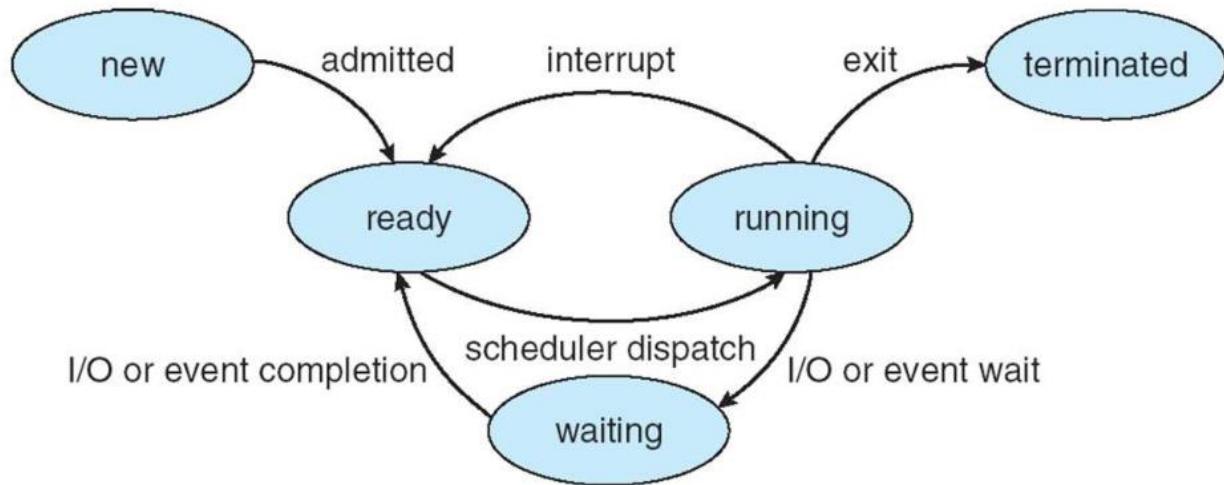




## ❖ Stati di un processo

Un processo durante l'esecuzione è soggetto a cambiamenti del suo **stato**, definito in parte dall'attività corrente del processo stesso. Un processo può trovarsi in uno tra i seguenti 5 stati:

- **new**: il processo è stato appena creato;
- **running**: esecuzione delle istruzioni del processo;
- **waiting**: il processo attende che si verifichi qualche evento per proseguire (come il completamento di un'operazione di I/o o la ricezione di un segnale);
- **ready**: il processo è pronto per essere eseguito e attende di essere assegnato alla CPU;
- **terminated**: il processo ha terminato l'esecuzione.



È importante capire che in ciascuna unità d'elaborazione può essere in esecuzione solo un processo per volta, sebbene molti processi possano essere pronti o nello stato di attesa.

## ❖ Descrittore di un processo

Per tenere traccia di ciò che succede nell'esecuzione, il sistema operativo mantiene una **tabella dei processi**. Per ogni processo c'è una entry in tabella.

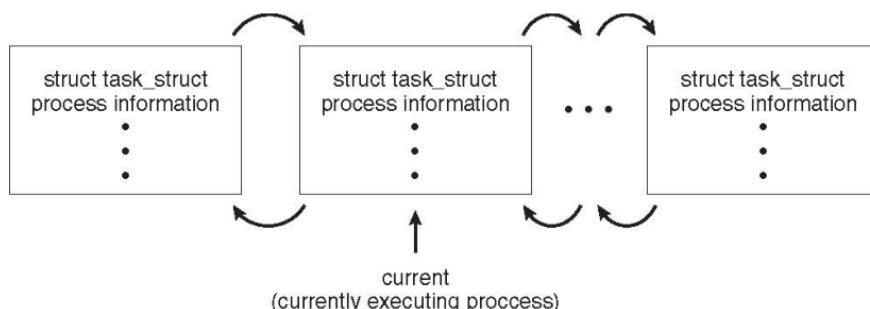
In ogni entry della tabella dei processi, ciascun processo è rappresentato da una struttura dati di tipo record detto **process control block, PCB, o task control block, TCB**. Questa struttura contiene molte informazioni connesse a un processo specifico. Alcune delle informazioni sono:

- **Stato del processo:** può essere new, ready, running, waiting, e così via.
- **IDs del processo:** identificativi del processo
- **Program counter:** contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- **Registri CPU:** contenuto dei registri della CPU quando viene interrotta l'esecuzione
- **Scheduling di CPU:** queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling
- **Memoria:** memoria allocata per il processo. Queste informazioni possono includere elementi quali il valore dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo
- **Contabilità:** informazioni su quota d'uso della CPU, tempo clock dallo start, etc.
- **Stato I/O:** queste informazioni comprendono la lista dei dispositivi di I/o assegnati a un determinato processo, l'elenco dei file aperti, e così via.

## PCB in Linux

In Linux il PCB è rappresentato in C da una `task_struct` specificata nel codice sorgente del kernel (in `/linux/sched.h`)

```
pid t_pid; /* process identifier */  
long state; /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```



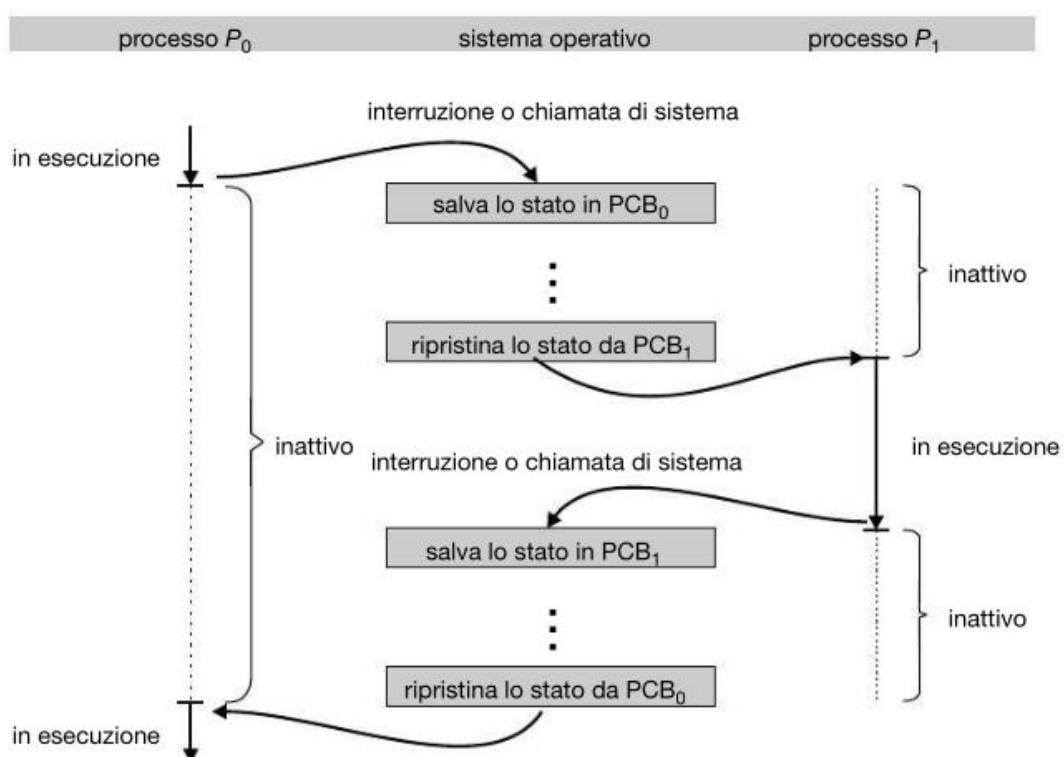
## ❖ Commutazione tra processi (context switch)

La tabella dei processi supporta la commutazione tra processi. Lo stato dei processi salvato e ripristinato durante gli switch.

In presenza di una interruzione, il sistema deve salvare il contesto del processo corrente, per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione. Il **contesto** è rappresentato all'interno del **PCB** del processo, e comprende i valori dei registri della cpu, lo stato del processo e informazioni relative alla gestione della memoria. In termini generali, si esegue un salvataggio dello stato corrente della cpu, sia che essa esega in modalità utente o in modalità di sistema; in seguito, si attuerà un corrispondente ripristino dello stato per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

- Il passaggio della cpu a un nuovo processo implica il salvataggio dello stato del processo attuale e il ripristino dello stato del nuovo processo. **Questa procedura di commutazione tra processi è nota col nome di cambio di contesto (context switch).**

Il **tempo necessario** varia da sistema a sistema, dipendendo dalla velocità della memoria, dal numero di registri da copiare, e dall'esistenza di istruzioni macchina appropriate (per esempio, una singola istruzione per caricare o trasferire in memoria tutti i registri). In genere si tratta di **qualche millisecondo**. E' un tempo di overhead il cui il sistema non fa lavoro utile. Più è complesso il sistema e la PCB più è lungo il context switch.



Al processo  $P_0$  arriva un'interruzione o una chiamata di sistema, a quel punto deve partire il processo  $P_1$  ma prima bisogna salvare il contesto esecutivo del processo  $P_0$  nella tabella dei processi. Durante il salvataggio del contesto del processo  $P_0$ , il processo  $P_1$  rimane in idle (inattivo). A questo punto viene recuperato lo stato del processo  $P_1$  e viene eseguito. Quando avrà finito l'esecuzione o arriva un interrupt, si salva lo stato del processo  $P_1$  nel PCB e si ricarica di nuovo il contesto del processo  $P_0$  per rimandarlo in esecuzione e così via.

## Processi e thread

Nella maggior parte dei sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità d'avere più percorsi d'esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta.

I processi gestiscono l'esecuzione di più flussi di controllo detti **thread** che sono delle sottounità di esecuzione detti anche "processi leggeri". Ogni unità dovrà eseguire un suo codice. **I thread condividono il layout di memoria del processo di cui fanno parte, anche se ognuno di essi possiede un diverso contesto esecutivo.** In un sistema che supporta i thread, il PCB viene esteso per includere informazioni su ogni thread. Per supportare i thread sono necessari anche altri cambiamenti nel sistema.

## ❖ Schedulazione di processi

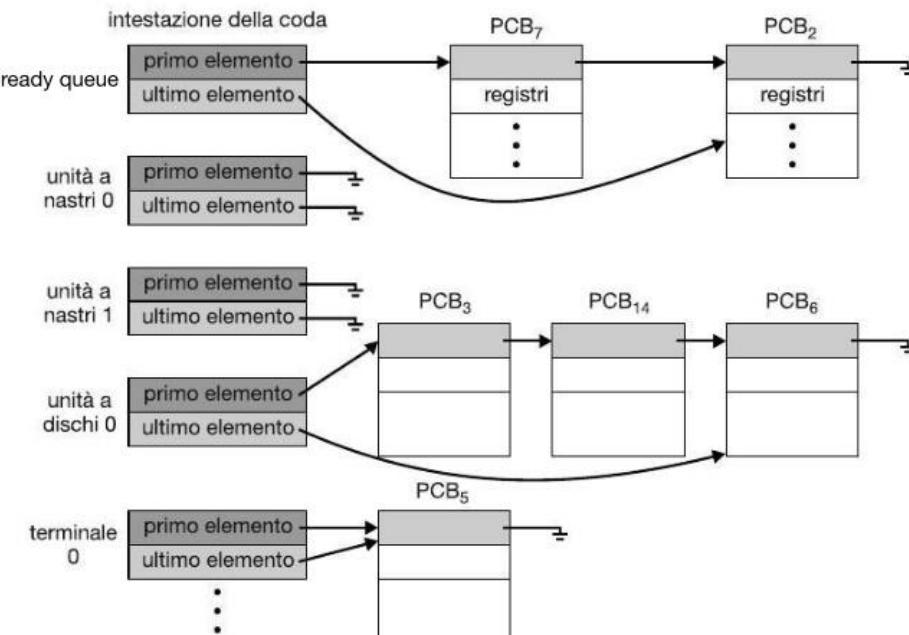
L'obiettivo della multiprogrammazione consiste nell'avere sempre un processo in esecuzione in modo da **massimizzare l'utilizzo della cpu**. L'obiettivo del time sharing è di **commutare l'uso della cpu tra i vari processi** così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, si effettua lo **scheduling dei processi**.

Lo scheduling dei processi si effettua sulla base di un **algoritmo di schedulazione** ed è a cura dello **scheduler**. L'algoritmo di schedulazione decide a quale processo dare accesso all'uso della cpu: seleziona tra i processi disponibili (in ready), quello da portare in running. Lo scheduler mantiene la **coda di scheduling dei processi**.

## Code di scheduling

- **Job queue:** è la coda in cui è presente ogni processo del sistema.
- **Ready queue:** processi presenti in memoria centrale, che sono pronti e nell'attesa d'essere eseguiti. Questa coda generalmente si memorizza come una lista concatenata: il primo nodo contiene i puntatori al primo e all'ultimo PCB dell'elenco, e ciascun PCB comprende un campo puntatore che indica il successivo processo contenuto nella coda dei processi pronti.
- **Wait queue:** insieme dei processi in attesa (waiting).
- **Device queues:** insieme di processi in attesa (waiting) di un dispositivo di I/O.

I processi migrano tra le diverse code.



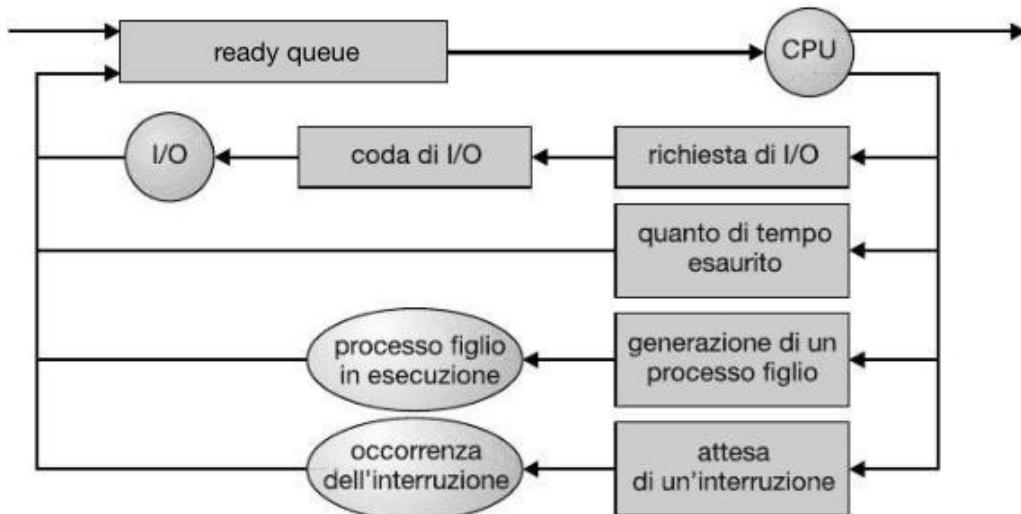
## ❖ Diagrammi di accodamento

Una comune rappresentazione dello scheduling dei processi è data da un diagramma di accodamento. Ogni riquadro rappresenta una coda. Nella figura sono presenti due tipi di coda: la ready queue e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.

Un nuovo processo si colloca inizialmente nella ready queue, dove attende finché non è selezionato per essere eseguito (**dispatched**). Una volta che il processo è assegnato alla cpu ed è nella fase d'esecuzione, si può verificare uno dei seguenti eventi:

- il processo può emettere una **richiesta di I/o** e quindi essere inserito in una coda di I/o;
- il processo può creare un nuovo **processo figlio** e attenderne la terminazione;
- il processo può essere **rimosso forzatamente dalla cpu** a causa di un'**interruzione** o per il **quanto di tempo esaurito**, ed essere reinserito nella coda dei processi pronti.

Nei primi due casi, al completamento della richiesta di I/o o al termine del processo figlio, il processo passa dallo stato d'attesa allo stato pronto ed è nuovamente inserito nella ready queue. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene rimosso da tutte le code, e vengono deallocated il suo pcb e le varie risorse.



## ❖ Scheduler: short-term scheduler, long-term scheduler, scheduler a medio termine

Un processo si trova in varie code di scheduling. Il sistema operativo, incaricato di selezionare i processi dalle suddette code, compie la selezione per mezzo di un opportuno **scheduler**. Esistono 3 tipi di scheduler:

- **Short-term scheduler (o CPU scheduler):** fa la selezione tra i processi pronti per l'esecuzione e assegna la cpu a uno di loro. A volte è l'unico scheduler del sistema. E' invocato frequentemente, una volta ogni 100 millisecondi e deve essere molto veloce.
- **Long-term scheduler (o Job scheduler):** sceglie quale processo deve essere portato nella coda ready, dalla memoria secondaria (in cui può essere caricato temporaneamente). Viene invocato meno frequentemente (minuti, secondi) ed è più lento.
  - Controlla il **grado di multiprogrammazione**, cioè il numero di processi presenti in memoria. Se è stabile, la velocità media di creazione dei processi deve essere uguale alla velocità media con cui i processi abbandonano il sistema. È importante che lo scheduler a lungo termine faccia un'accurata selezione dei processi. In generale, la maggior parte dei processi si può caratterizzare come avente una prevalenza di I/O (**I/O bound**) oppure come avente una prevalenza d'elaborazione (**CPU bound**).
  - **I/O-bound process:** spende più tempo in operazioni I/O che in computazione, facendo piccoli e brevi accessi in CPU.
  - **CPU-bound process:** spende più tempo in computazione; pochi e lunghi accessi in CPU.
  - Se tutti i processi fossero con prevalenza di I/o, la coda dei processi pronti sarebbe quasi sempre vuota e lo scheduler a breve termine resterebbe pressoché inattivo. Se tutti i processi fossero con prevalenza d'elaborazione, la coda d'attesa per l'I/o sarebbe quasi sempre vuota, i dispositivi non sarebbero utilizzati, e il sistema sarebbe sbilanciato anche in questo caso.
  - **Le prestazioni migliori sono date da una combinazione equilibrata di processi I/o bound e cpu bound, cosa che cerca di fare il long-term scheduler.**
- **Scheduler a medio termine:** in alcuni sistemi operativi come quelli in time-sharing, si può introdurre un livello di scheduling intermedio. L'idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria, **riducendo il grado di multiprogrammazione del sistema (contesa CPU)**. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta. Questo processo prende il nome di **swapping**. Il processo viene rimosso e successivamente ricaricato in memoria dallo scheduler a medio termine.

## ❖ Operazioni su processi e chiamate di sistema

Il Sistema deve fornire meccanismi per:

- **Creazione di processi,**
- **Terminazione di processi,**
- **Comunicazione tra processi**
- **Sincronizzazione tra processi**
- **Altri meccanismi di gestione di processi**

### Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi. Il processo creante si chiama processo **padre**, mentre il nuovo processo si chiama processo **figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un **albero di processi**.

La maggior parte dei sistemi operativi identifica un processo per mezzo di un numero univoco, solitamente un intero, detto **pid (process identifier)**. Il pid fornisce un valore univoco per ogni processo del sistema e può essere usato come indice per accedere a vari attributi di un processo all'interno del kernel.

Il sistema operativo nella creazione di un processo figlio deve decidere quali risorse devono condividere processo padre e figlio. **Condivisione di risorse:**

- **Padre e figlio condividono le stesse risorse in modo uguale**
- **Figlio condivide un sottoinsieme delle risorse del padre**
- **Padre e figlio non condividono alcuna risorsa (sistemi UNIX)**

In merito alle **opzioni di esecuzione:**

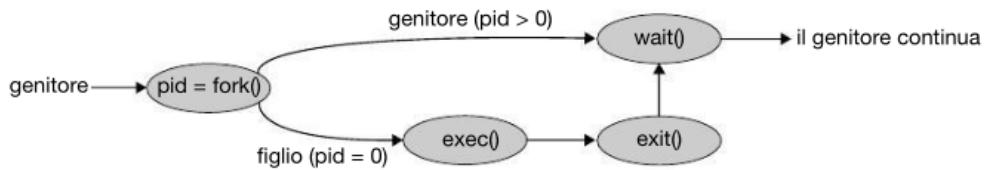
- **Padre e figlio vengono eseguiti in concorrenza**
- **Padre aspetta che il figlio finisca**

Per quel che riguarda lo **spazio d'indirizzi di memoria** del nuovo processo ci sono due opzioni:

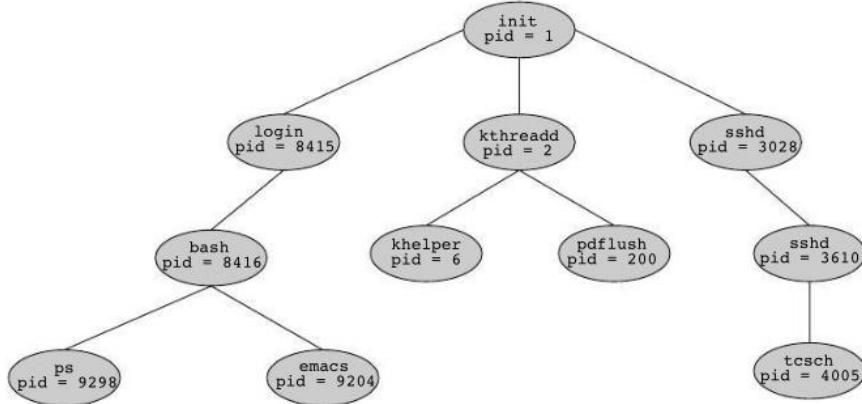
- **il processo figlio è un duplicato del processo genitore** (ha gli stessi programma e dati del genitore);
- **nel processo figlio si carica un nuovo programma.**

### Esempio della creazione di un processo in UNIX

- In UNIX ogni processo è identificato dal proprio PID, un intero univoco.
- Un nuovo processo si **crea** per mezzo della chiamata di sistema **fork()**, ed è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo permette al processo genitore di comunicare senza difficoltà con il proprio processo figlio.
- Generalmente, dopo una chiamata di sistema fork(), uno dei due processi impiega una chiamata di sistema **exec()** per sostituire lo **spazio di memoria del processo con un nuovo programma**. In questo modo i due processi possono comunicare e poi procedere in modo diverso.
- Il padre se durante l'esecuzione del processo figlio non ha nient'altro da fare, può invocare la chiamata di sistema **wait()** per rimuovere se stesso dalla coda dei processi pronti fino alla terminazione del figlio. In pratica eseguendo la chiamata di sistema wait(), **il processo padre attende che il processo figlio termini**.
- Il processo termina con la chiamata di sistema **exit()**



## Albero dei processi Linux



- Il processo **init** (che ha sempre **pid = 1**) svolge il ruolo di processo padre di tutti i processi utente.

## Esempio codice di creazione processo UNIX con fork() ed exec() in C

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* genera un nuovo processo */
pid = fork();

if (pid < 0) { /* errore */
    fprintf(stderr, "generazione del nuovo processo fallita");
    return 1;
}
else if (pid == 0) { /* processo figlio */
    execlp("/bin/ls", "ls", NULL);
}
else /* processo genitore */
    /* il genitore attende il completamento del figlio */
    wait(NULL);
    printf("il processo figlio ha terminato");
}

return 0;
}

```

- 
- Se il figlio termina prima che il padre arrivi all'istruzione **wait** (primitivo meccanismo di sincronizzazione), esso diventa **zombie**. Arrivato alla **wait()** il padre raccoglie lo stato del figlio.
  - Usando la chiamata di sistema **execlp()**, una versione della chiamata di sistema **exec()**, il processo figlio sovrappone il proprio spazio d'indirizzi con il comando **/bin/ls** di UNIX (che si usa per ottenere

l'elenco del contenuto di una directory).

### Programma C con fork

```
int main(void) {
    int i;

    for (i=0; i<2 ;i++)
        if (fork()>0) {
            printf("Padre! %d\n", i);
        } else {
            printf("Figlio! %d\n", i);
        }

    sleep(10);
    return 0;
}
```

- Qual è l'output di questo programma?
- Quanti processi vengono creati?
- Di chi è figlio ciascun processo creato?

padre 0	padre 1	padre 0	figlio! 0
padre 1	figlio! 0	padre1	figlio! 1
figlio! 1	padre 1	padre1	figlio! 1
figlio! 1	figlio! 1		

- Dopo una fork, se un processo figlio cambia una variabile globale, il processo padre non vede quel cambiamento perché esso avviene nel layout di memoria del figlio.

### Terminazione di un processo

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema **exit()**; a questo punto, il processo figlio ritorna i **dati di status** al processo genitore, che li riceve attraverso la chiamata di sistema **wait()**. Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'I/O, sono liberate dal sistema operativo.

Un processo padre può porre termine all'esecuzione di uno dei suoi processi figli con la chiamata **abort()** per diversi motivi, tra i quali i seguenti:

- Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate.
- Il compito (task) assegnato al processo figlio non è più richiesto.
- Il processo padre termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza.

In alcuni sistemi, se un processo termina si devono terminare anche i suoi figli. Si parla di **terminazione a cascata (cascading termination)** in cui tutta la gerarchia di figli termina. E' una procedura avviata di solito dal sistema operativo.

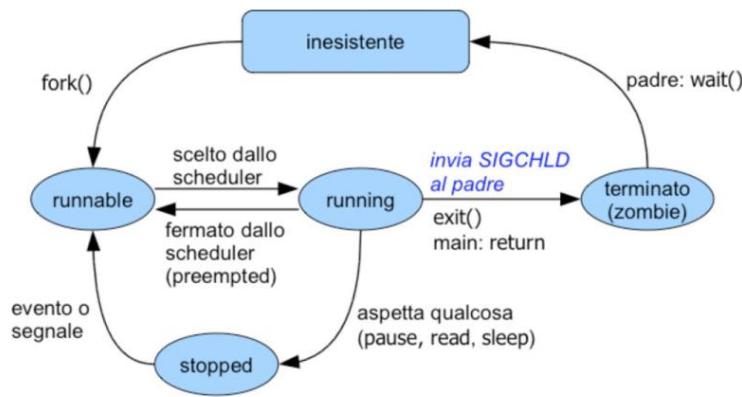
Un processo genitore può attendere la terminazione di un processo figlio utilizzando la chiamata di sistema **wait()**, a cui viene passato un parametro che permette al genitore di ottenere lo stato di uscita del figlio. Questa chiamata di sistema restituisce anche l'ID di processo del figlio terminato in modo che il genitore possa sapere di quale dei suoi figli si tratta:

```
pid_t pid;
int status;

pid = wait(&status);
```

Un processo figlio che è terminato, ma il cui genitore non ha ancora chiamato la **wait()**, è detto processo **zombie**.

In UNIX, se un processo padre termina prima dei suoi figli lasciandoli **orphan** (orfani) senza chiamare la **wait()**, viene assegnato al processo **init** (iniziale, System Daemon) il ruolo di nuovo padre dei processi orfani. Il processo init invoca periodicamente **wait()**, consentendo in tal modo di raccogliere lo stato di uscita di qualsiasi processo orfano.



## ❖ Comunicazione interprocesso

I processi eseguiti nel sistema operativo possono essere **indipendenti** o **cooperanti**.

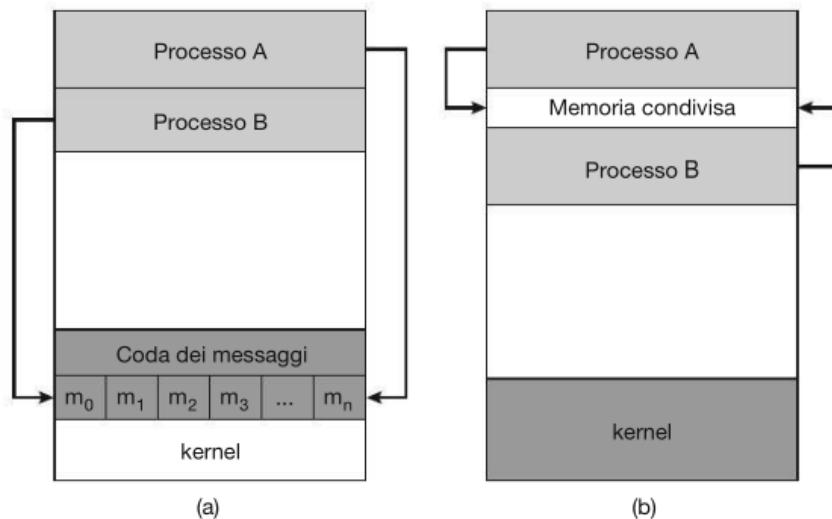
- Un processo è **indipendente** se non può influire su altri processi del sistema o subirne l'influsso. Un processo indipendente non condivide dati con altri processi.
- Un processo è **cooperante** se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Un processo cooperante condivide dati con altri processi.

Ragioni per la cooperazione dei processi:

- **Condivisione d'informazioni.**
- **Velocizzazione del calcolo.**
- **Modularità.**
- **Convenienza.**

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un **meccanismo di comunicazione tra processi (IPC, interprocess communication)**. I modelli fondamentali per la comunicazione tra processi sono:

- **Shared memory:** si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona
- **Message passing:** la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti lungo un canale di comunicazione tra i due processi messo a disposizione dal kernel



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

## Shared memory

La comunicazione tra i due processi cooperanti avviene su una **regione di shared memory** la quale è tipicamente si trova nello **spazio di indirizzamento del processo che la crea**. Gli altri processi che vogliono comunicare devono accedere a questo spazio che generalmente è proibito perché appartenente ad un processo specifico, quindi si chiede al kernel di **rilassare la restrizione**. I processi leggono e scrivono su questo spazio che è gestito da loro stessi. La **coordinazione di lettura e scrittura** è sotto la responsabilità dei processi coinvolti, quindi il **programmatore** deve implementare un meccanismo in grado di far **sincronizzare** i due processi in lettura e scrittura, il sistema operativo mette solo a disposizione la regione di memoria condivisa.

## Problema del produttore-consumatore

Tale problema è un comune paradigma per processi cooperanti.

- **Un processo produttore produce informazioni che sono consumate da un processo consumatore.** In alcuni casi il processo potrebbe avere sia il ruolo di produttore che consumatore.

Una possibile **soluzione** del problema del produttore/consumatore si basa sulla **shared memory** utilizzando aree di memoria condivisa chiamate **buffer (memoria tampone)**. Il buffer viene riempito dal produttore e svuotato dal consumatore. Non è necessario che siano sincronizzati.

Esistono 2 tipi di buffer:

- **unbounded-buffer:** non hanno limite sulla loro dimensione
- **bounded-buffer:** il buffer ha una dimensione fissa. Bisogna gestire la problematica del buffer pieno, ossia che il produttore deve aspettare che il consumatore svuoti il buffer.

## Message passing

Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi. È una tecnica particolarmente utile negli ambienti distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete.

Un meccanismo per lo scambio di messaggi deve prevedere almeno **due operazioni**:

- **send(message)**, cioè “invia messaggio”
- **receive(message)**, cioè “ricevi messaggio”.
- **I messaggi possono avere lunghezza fissa o variabile.** Il messaggio consiste in una quantità di byte che vengono letti o scritti sul canale di comunicazione.

Per scambiare i messaggi deve esistere un **canale di comunicazione (communication link)**, realizzabile in molti modi. I processi scambiano i messaggi tramite send/receive.

L'implementazione del canale deve tenere conto di alcune cose:

- come si stabilisce
- se può coinvolgere più di due processi
- quanti canali di comunicazione possono avere due processi comunicanti
- capacità del canale
- dimensione del messaggio fissa o variabile
- canale unidirezionale/bidirezionale

A livello di logica:

- Comunicazione diretta o indiretta
- Comunicazione sincrona o asincrona
- Buffering dei messaggi automatico o esplicito

## Soluzione del problema del produttore-comsumatore con Unbounded-buffer (shared memory)

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
```

Buffer implementato mediante array circolare con due indici:

- **in:** prossima locazione free dove si può scrivere
- **out:** prossima locazione full dove si può leggere
- **buffer pieno:**  $((in+1) \% BUFFER\_SIZE) == out$
- **buffer vuoto:**  $in == out$

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

### **Produttore**

```
item next_produced;

while (true) {
    /* produce an item in next produced */
    while (((in + 1) \% BUFFER_SIZE) == out)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) \% BUFFER_SIZE;
}
```

Finché il buffer è pieno non fa nulla e va in attesa.

Se il buffer non è pieno va a riempire la prossima locazione free con l'item next\_produced.

Si aggiorna poi la locazione che diventa piena.

### **Consumatore**

```
item next_consumed;

while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) \% BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Finché il buffer è vuoto deve aspettare. Altrimenti prende l'item da consumare tramite l'indice out. Dopodiché aggiorna tale indice e torna a consumare.

### **Soluzione del produttore consumatore-consumatore (Message passing bloccante)**

È banale dare soluzione al problema del produttore e del consumatore tramite le **primitive bloccanti e sincrone send() e receive()** riguardanti il message passing . Il **produttore**, infatti, si limita a invocare **send()** e attendere finché il messaggio sia giunto a destinazione; analogamente, il consumatore richiama **receive()**, bloccandosi fino all'arrivo di un messaggio.

**Blocking send: mittente è bloccato finché il messaggio non viene ricevuto**

**Blocking receive: destinatario bloccato finché il messaggio non è disponibile**

```
message next_produced;
while (true) {
    /* produce an item in next produced */
    send(next_produced);
}

message next_consumed;
while (true) {
    receive(next_consumed);

    /* consume the item in next consumed */
}
```

---

## Esempio di IPC Systems - POSIX

### ❖ Shared memory

La memoria condivisa in POSIX è organizzata utilizzando i file mappati in memoria (**memory mapped files**), che associano la regione di shared memory a un file.

Un processo deve prima creare un oggetto shared memory con la chiamata di sistema **shm\_open()**

**shm\_fd = shm\_open(name, O\_CREAT | O\_RDWR, 0666);**

- Il primo parametro specifica il nome dell'oggetto memoria condivisa.
- I parametri successivi specificano che l'oggetto memoria condivisa deve essere creato se non esiste ancora (O\_CREAT) e che l'oggetto è aperto in lettura e scrittura (O\_RDWR)

**Una chiamata a shm\_open() con esito positivo restituisce un intero, il descrittore di file per l'oggetto memoria condivisa.**

Una volta che l'oggetto è creato, viene utilizzata la funzione **ftruncate()** per specificare la **dimensione dell'oggetto in byte**.

**ftruncate(shm\_fd, 4096);**

Infine, la funzione **mmap()** crea un file mappato in memoria che contiene l'oggetto shared memory e restituisce un puntatore al file mappato in memoria che viene utilizzato per accedere all'oggetto shared memory

### ❖ Funzione mmap()

La funzione mmap mappa nel process address space in memoria principale un file residente su disco o un device. Una volta mappato accessibile in modo diretto.

```
#include <sys/mman.h>
void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)
```

- **address:** indirizzo di partenza richiesto
- **size\_t:** bytes richiesti
- **protect:** PROT\_READ | PROT\_WRITE | PROT\_EXEC | PROT\_NONE
- **flags:** MAP\_SHARED, MAP\_PRIVATE, MAP\_ANON, MAP\_FIXED
  - **MAP\_SHARED** specifica che i cambiamenti sono condivisi
- **filedes:** descrittore di file
- **offset:** offset del file

## ❖ IPC Posix - Produttore

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()

{
/* dimensione, in byte, dell'oggetto memoria condivisa */
const int SIZE 4096;
/* nome dell'oggetto memoria condivisa */
const char *name = "OS";
/* stringa scritta nella memoria condivisa */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* descrittore del file di memoria condivisa */
int shm_fd;
/* puntatore all'oggetto memoria condivisa */
void *ptr;

/* crea l'oggetto memoria condivisa */
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

/* configura la dimensione dell'oggetto memoria condivisa */
ftruncate(shm_fd, SIZE);

/* mappa in memoria l'oggetto memoria condivisa */
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

/* scrive sull'oggetto memoria condivisa */
sprintf(ptr,"%s",message_0);
ptr += strlen(message_0);
sprintf(ptr,"%s",message_1);
ptr += strlen(message_1);

return 0;
}
```

## ❖ IPC Posix - Consumatore

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* dimensione, in byte, dell'oggetto memoria condivisa */
const int SIZE 4096;
/* nome dell'oggetto memoria condivisa */
const char *name = "OS";
/* descrittore del file di memoria condivisa */
int shm_fd;
/* puntatore all'oggetto memoria condivisa */
void *ptr;

/* apre l'oggetto memoria condivisa */
shm_fd = shm_open(name, O_RDONLY, 0666);

/* mappa in memoria l'oggetto memoria condivisa */
ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

/* legge dall'oggetto memoria condivisa */
printf("%s", (char *)ptr);

/* rimuove l'oggetto memoria condivisa */
shm_unlink(name);

return 0;
}
```

## Pipe

E' un meccanismo di IPC basato sul message passing. Una pipe agisce come canale di comunicazione tra processi. In essa transita un flusso di dati che permette a due processi di comunicare. E' uno dei primi meccanismi di comunicazione in UNIX.

Problematiche nell'implementazione:

- La **comunicazione** permessa dalla pipe è **unidirezionale** o **bidirezionale**?
- Se è ammessa la comunicazione **bidirezionale**, essa è di tipo
  - **half duplex** (i dati possono viaggiare in un'unica direzione alla volta) o
  - **full duplex** (i dati possono viaggiare contemporaneamente in entrambe le direzioni)?
- Deve esistere una **relazione (del tipo padre-figlio)** tra i processi in comunicazione?
- Le pipe possono **comunicare in rete** o i processi comunicanti devono risiedere **sulla stessa macchina**?

Due tipi comuni di pipe utilizzate sia in UNIX sia in Windows: le **pipe convenzionali (o ordinarie)** e le **named pipe**.

### ❖ Pipe convenzionali (o ordinarie / anonymous in windows)

- Le pipe convenzionali permettono a due processi di comunicare secondo una modalità **produttore- consumatore**.
- Il **produttore** scrive a una estremità del canale (**l'estremità dedicata alla scrittura, o write-end**)
- Il **consumatore** legge dall'altra estremità (**l'estremità dedicata alla lettura, o read-end**).
- **Sono unidirezionali**, perché permettono la comunicazione in un'unica direzione. Se viene richiesta la comunicazione bidirezionale devono essere utilizzate due pipe, ognuna delle quali manda i dati in una direzione.
- Richiedono una **relazione parentale tra i processi** che comunicano.

All'inizio bisogna stabilire quale processo ha il ruolo di produttore e quale ha il ruolo di consumatore. Quando si crea la pipe tutte le imboccature in lettura/scrittura rimangono aperte, dovrà essere il processo che scrive a chiudere l'imboccatura di lettura e viceversa il processo che legge a chiudere l'imboccatura di scrittura.

### La funzione pipe in UNIX

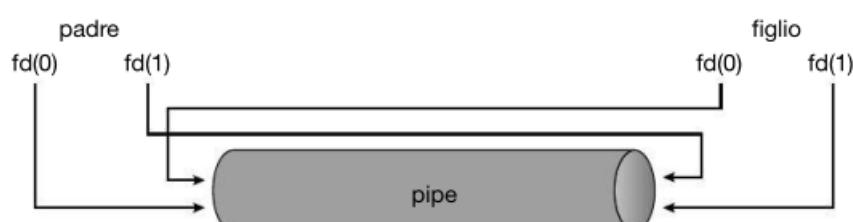
Nei sistemi UNIX le pipe convenzionali sono costruite utilizzando la funzione **pipe()**. Vengono considerate come file.

```
#include <unistd.h>
```

```
int pipe ( int filedes[2] );
```

l'argomento **filedes[]** è un array di interi di dimensione 2 costituito da due descrittori di file e fornisce alla pipe le due imboccature, una di scrittura e una di lettura.

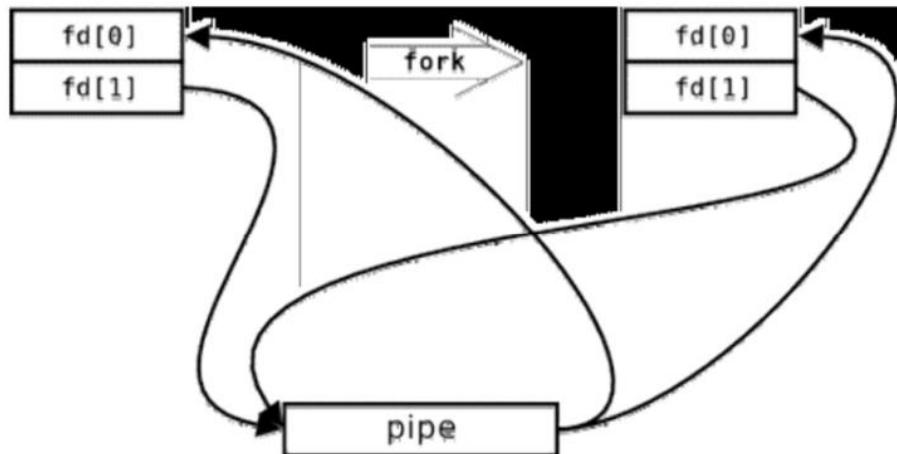
- **filedes[0]**: rappresenta il lato in lettura della pipe
- **filedes[1]**: rappresenta il lato in scrittura della pipe
- La funzione restituisce 0 in caso di successo, -1 in caso di errore.



Solitamente un processo padre crea inizialmente una pipe e la utilizza successivamente per comunicare con un processo figlio generato dopo il comando **fork()**. Il processo figlio eredita i file aperti dal processo padre. Dal momento che la pipe è un tipo speciale di file, il figlio eredita la pipe dal proprio processo padre.

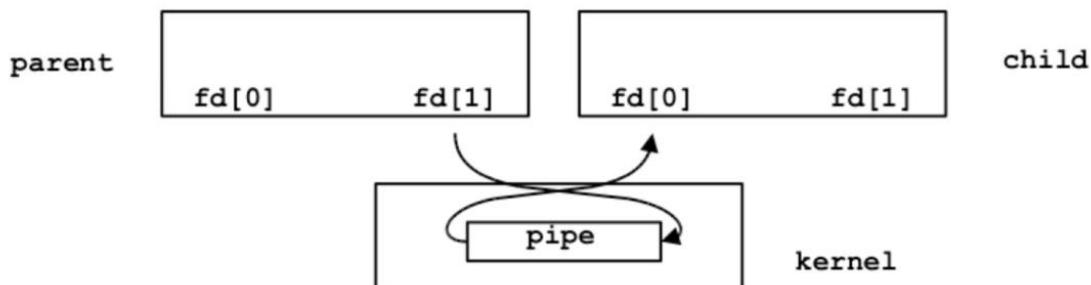
### Funzionamento:

- Il processo padre crea una pipe
- in seguito esegue una chiamata `fork()`, generando un processo figlio
- ciò che succede dopo la chiamata `fork()` dipende da come i dati devono fluire nel canale.
- I canali non utilizzati vanno chiusi: sia il processo padre sia il processo figlio chiudono inizialmente le estremità inutilizzate del canale (in lettura o scrittura)



#### Esempio: parent → child

- Il parent chiude l'estremo di read (`close(fd[0])` ;)
- Il child chiude l'estremo di write (`close(fd[1])` ;)



## Esempi di Pipe Ordinare in Unix

```
int fd[2];

if (pipe(fd) < 0)
    perror("pipe"), exit(1);
if ( (pid=fork()) < 0 )
    perror("fork"), exit(1);
else if (pid>0) { // padre
    close(fd[0]);
    write(fd[1], "ciao!", 5);
} else {           // figlio
    close(fd[1]);
    n = read(fd[0], buf, sizeof(buf));
    write(STDOUT_FILENO, buf, n);
}
```

- Dopo la fork() la pipe non si sdoppia, ma rimane unica.
- Il padre chiude l'imboccatura in lettura e scrive.
- Il figlio chiude l'imboccatura in scrittura e poi legge.
  - All'inizio una pipe è vuota
  - write aggiunge dati alla pipe
  - read legge e *rimuove* dati dalla pipe
    - non si possono leggere piu' volte gli stessi dati da una pipe
    - non si puo' chiamare lseek su una pipe
    - i dati si ottengono in ordine First In First Out
  - una pipe con una estremità chiusa si dice rottta (broken)

## ❖ Pipe con nome (named pipe)

Le named pipe costituiscono uno strumento di comunicazione molto più potente.

- La comunicazione **può essere bidirezionale**.
- La **relazione di parentela padre-figlio non è necessaria**.
- Una volta che si sia creata la named pipe, **diversi processi possono utilizzarla** per comunicare. In uno scenario tipico una named pipe ha infatti diversi scrittori.
- Le named pipe **continuano a esistere anche dopo che i processi comunicanti sono terminati**.

### Pipe con nome in Unix

Nei sistemi UNIX le named pipe sono dette **FIFO**. Una volta create, esse appaiono come normali **file** all'interno del file system.

Una FIFO viene creata mediante una chiamata di sistema **mkfifo()** e viene poi manipolata con le usuali chiamate di sistema **open()**, **read()**, **write()** e **close()**; essa continuerà a esistere finché non sarà esplicitamente eliminata dal file system.

L'unica tipologia di trasmissione consentita è quella **half duplex**. Nel caso in cui i dati debbano viaggiare in entrambe le direzioni, vengono solitamente utilizzate due FIFO. La comunicazione è di tipo **message-passing** e i due processi **conoscono il nome della pipe**.

Per utilizzare le FIFO **i processi comunicanti devono risiedere sulla stessa macchina**: se è richiesta la comunicazione tra più macchine devono essere impiegate le **socket**.

### Esempio Pipe con nome in Unix

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MAX_BUF_SIZE 1000

int main(int argc, char *argv[]){
    int fd, ret_val, count, numread;
    char buf[MAX_BUF_SIZE];
    /* Create the named - pipe */
    ret_val = mkfifo("miafifo", 0666);
    if ((ret_val == -1) && (errno != EEXIST)) {
        perror("Error creating the named pipe");
        exit (1);
    }
    /* Open the pipe for reading */
    fd = open("miafifo", O_RDONLY);
    /* Read from the pipe */
    numread = read(fd, buf, MAX_BUF_SIZE);
    buf[numread] = '\0';
    printf("Server : Read From the pipe : %s\n", buf);
    /* Convert to the string to upper case */
    count = 0;
    while (count < numread) {
        buf[count] = toupper(buf[count]);
        count++;
    }
    printf("Server : Converted String : %s\n", buf);
}
```

### Codice Server

```

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int fd;

    /* Check if an argument was specified. */

    if (argc != 2) {
        printf("Usage : %s <string to be sent to the server>\n", argv[0]);
        exit (1);
    }

    /* Open the pipe for writing */
    fd = open("miafifo", O_WRONLY);

    /* Write to the pipe */
    write(fd, argv[1], strlen(argv[1]));

```

## Codice Client

### Esempio esecuzione:

- ./servFifo &
- ./clientFifo prova
- Server : Read From the pipe : prova
- Server : Converted String : PROVA

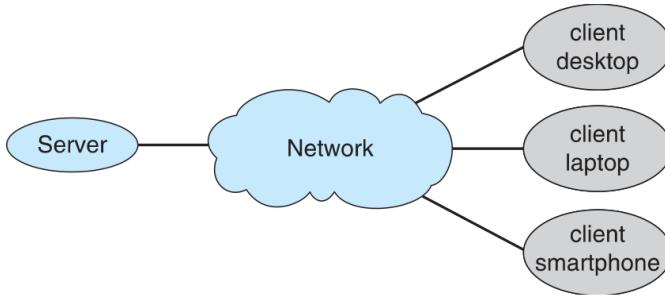
**I due processi non sono imparentati e completamente distinti**

## ❖ Comunicazione in Sistemi Client-Server: Socket e Remote procedure call

Due modalità:

- **Socket:** strumento per la comunicazione interprocesso che fornisce un interfacciamento standard per far comunicare processi residenti sulla stessa macchina o macchine diverse con la possibilità di rendere trasparente tutta la comunicazione di rete, rendendo l'idea di una comunicazione diretta.
- **Remote procedure call:** Le Remote procedure call (RPC) sono astrazioni di chiamata di procedure tra processi su sistemi in rete. Usano porte per differenziare i servizi.

**Sistemi Client-Server:** Tipica architettura di rete dove Sistemi Server rispondono alle richieste dei Sistemi client



### ❖ Socket

- Una socket è definita come l'**estremità di un canale di comunicazione (endpoint)**.

Una coppia di processi che comunicano attraverso una rete usa una coppia di socket, una per ogni processo, e ogni socket è identificata da un indirizzo IP concatenato a un numero di porta. La comunicazione avviene tra coppie di socket: su stessa macchina (IPC locali) o diverse (IPC di rete).

Una socket internet è identificata da un **indirizzo** e una **porta**: 161.25.19.8:1625 indica la porta 1625 sull'host 161.25.19.8. La **porta** è un numero che differenzia diversi servizi su un host. Tutte le porte sotto 1024 sono usate per servizi standard (porte well known).

Un IP address speciale **127.0.0.1 (loopback)** si riferisce al sistema in cui il processo gira.

### Comunicazione Client-Server:

- **Server si pone in ascolto su una porta:** avrà una socket in ascolto pronta per ricevere richieste dai client connessa su una porta non standard e libera (superiore a 1024).
- **Client richiede un servizio:** per stabilire la comunicazione deve riconoscere l'indirizzo dell'host (server) e la porta su cui viene fornito il servizio.
- Si possono definire diversi tipi di comunicazione: **Connection oriented (TCP) o connectionless (UDP)**

In **UNIX** il server si pone in ascolto tramite la chiamata bloccante **accept()** e rimane in attesa fino all'arrivo della richiesta di un client. A quel punto, la chiamata **accept()** restituisce la socket che il server può usare per comunicare con il client. Il client fa la richiesta conoscendo l'indirizzo IP della macchina server, la porta su cui sta in ascolto il server e si connette alla socket del server tramite la chiamata **connect()**.

### 3. THREAD E CONCORRENZA

Quasi tutti i sistemi operativi moderni permettono che un processo possa avere più percorsi di controllo chiamati thread.

#### ❖ Definizione di thread

- Un **thread** è un flusso di esecuzione indipendente all'interno ad un processo. I thread sono anche chiamati **processi leggeri** perché possiedono un contesto esecutivo più snello rispetto ai processi. Ogni thread possiamo considerarlo come una **sottounità di esecuzione** di un particolare processo che svolge un determinato **task** in **concorrenza** con altri thread presenti.

Un thread comprende:

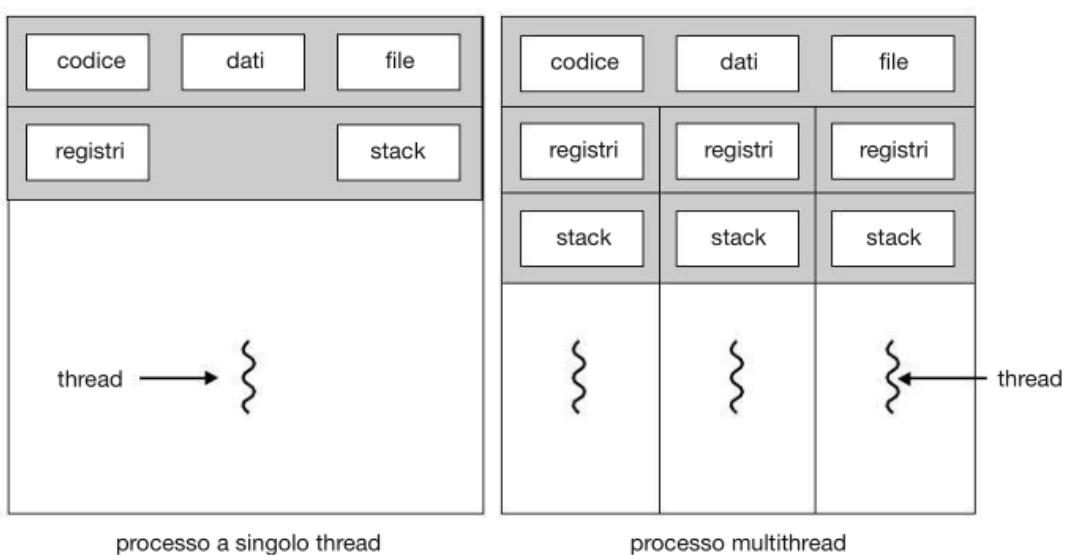
- un proprio **thread id** (ID),
- un **program counter**,
- un insieme di **registri**,
- uno **stack**.
- Un thread **condivide** con gli altri thread che appartengono allo stesso processo la **sezione del codice**, la **sezione dei dati** e altre **risorse** di sistema, come i file aperti e i segnali.

#### ❖ Motivazioni

- Le applicazioni sono generalmente multithread
- Occorrono dei flussi di esecuzione multipli ma più leggeri di un processo stesso.
- Si vuole avere una concorrenza tra flussi di controllo paralleli appartenenti ad uno stesso processo senza dover gestire layout separati di memoria derivanti dalla creazione di più processi, che comporterebbe anche continui context switch e perdite di tempo. La creazione di un processo è onerosa.

#### ❖ Processi single-threaded vs processi multithreaded

- Un **processo tradizionale**, chiamato anche processo **pesante** (heavyweight process), è composto da **un solo thread** ed è detto **single-threaded**.
- Un **processo multithreaded** è composto da più thread in grado di svolgere più compiti in modo concorrente. Ogni thread avrà la propria identità (ID, PC, Registri, Stack).



## ❖ Vantaggi dall'uso dei thread

- **Tempo di risposta:** un programma multithread continua la sua esecuzione anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta all'utente. Mentre si svolge un task il processo non è bloccato solo su quello, ma ne svolge anche altri in contemporanea. Questa caratteristica è particolarmente utile nella progettazione di interfacce utente.
- **Robustezza del processo:** se si blocca l'esecuzione di un singolo thread non si blocca l'esecuzione dell'intero processo.
- **Condivisione delle risorse:** è agevolata in quanto i **thread condividono per default la memoria e le risorse del processo** al quale appartengono. Il vantaggio della condivisione del codice e dei dati consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
- **Economia:** assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. La creazione e la gestione dei processi richiedono in generale molto più tempo. **Il thread switching comporta meno overhead del context switching.**
- **Scalabilità:** i processi possono avvantaggiarsi delle **architetture multiprocessore** dove i thread si possono eseguire in parallelo su distinti core di elaborazione. Incrementa il **parallelismo**.

## ❖ Multicore programming

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

Sistemi Multicore o multiprocessori pongono diversi **problem**i:

- Come dividere le attività sui core: computazioni parallelizzabili su più core
- Come bilanciare il calcolo sui core
- Come spartire i dati nel codice per i core
- Come gestire le dipendenze dei dati sui core
- Dipendenze possono richiedere sincronizzazioni
- Come fare test e debugging: più tracce di esecuzioni da testare

## ❖ Concorrenza vs Parallelismo

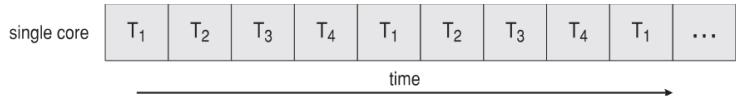
In un sistema con un **singolo core**, “esecuzione concorrente” significa solo che l'esecuzione dei thread è avvicendata nel tempo, perché la Cpu è in grado di eseguire un solo thread alla volta. Su un sistema **multicore**, invece, “esecuzione concorrente” significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core.

- **Parallelismo:** un sistema sfrutta il parallelismo se può eseguire più task simultaneamente distribuiti su più unità di calcolo
- **Concorrenza:** un sistema concorrente supporta più di un task consentendo a tutti di progredire nell'esecuzione, sfruttando la distribuzione temporale (timesharing) e lo scheduling. È dunque possibile avere concorrenza senza parallelismo.

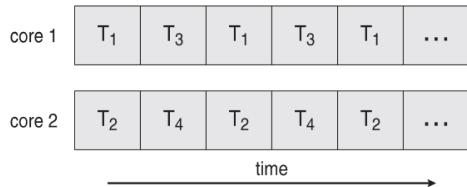
## ❖ Tipi di parallelismo

- **Parallelismo dei dati:** prevede la distribuzione di sottoinsiemi dei dati su più core di elaborazione e l'esecuzione della stessa operazione su ogni core
- **Parallelismo dei task:** distribuzione di attività (thread), e non di dati, su più core. Ogni thread realizza un'operazione distinta e thread differenti possono operare sugli stessi dati o su dati diversi.

### Esecuzione concorrente su sistema single-core:



### Parallelismo su sistema multi-core:



#### ❖ Legge di Amdahl

La legge di Amdahl è una formula che permette di determinare i potenziali **guadagni in termini di prestazioni ottenuti dall'aggiunta di ulteriori core di elaborazione**, nel caso di applicazioni che contengono sia componenti seriali sia componenti parallele.

**S** la porzione di applicazione seriale

**N** core di elaborazione per il processamento

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

#### Esempio:

- Supponiamo di avere un'applicazione che sia al 75% parallela e al 25% seriale.
- Se eseguiamo l'applicazione su un sistema con 2 core di elaborazione, possiamo ottenere un incremento di velocità pari a 1,6.
- Se aggiungiamo 4 core, l'incremento di velocità è pari a 2,28.

Un fatto interessante circa la legge di Amdahl è che **per N che tende all'infinito, l'incremento di velocità converge a 1/S**. Per esempio, se il 40 per cento di un'applicazione viene eseguita in maniera seriale, il massimo aumento di velocità è di 2,5 volte, indipendentemente dal numero di core di elaborazione che aggiungiamo.

#### ❖ User thread e kernel thread

I thread possono essere distinti in:

- **thread a livello utente (user thread):** sono gestiti sopra il livello del kernel e senza il suo supporto, mediante librerie di thread gestite a livello utente. Le principali librerie per thread sono:
  - **POSIX Pthreads**
  - Widows Threads
  - Java Thread
- **thread a livello kernel (kernel thread):** sono supportati dal kernel e sono gestiti direttamente dal sistema operativo.

#### ❖ Modelli di multithreading: Many-to-one, One-to-one, Many-to-many, Two level model

Deve esistere una relazione tra i thread a livello utente e i thread a livello kernel. Il sistema operativo deve mappare i thread a livello utente in thread a livello kernel.

#### ❖ Modello Many-to-one

- Questo modello fa corrispondere molti thread a livello utente a un singolo thread a livello kernel.

La **gestione dei thread** risulta **efficiente** perché viene effettuata da una **libreria di thread nello spazio utente**. Tuttavia l'intero processo rimane bloccato se un thread invoca una **chiamata di sistema di tipo bloccante**. Inoltre, poiché **un solo thread alla volta può accedere al kernel**, è **impossibile eseguire thread multipli in parallelo** in sistemi multicore;

#### ❖ Modello One-to-one

- Il modello da uno a uno mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel.

**E' uno dei modelli più adottati (Linux, Windows).** Questo modello offre un grado di concorrenza maggiore, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l'**esecuzione dei thread in parallelo nei sistemi multiprocessore**. L'unico **svantaggio** di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel e ciò può sovraccaricare le prestazioni di un'applicazione; la soluzione è limitare il numero di thread supportabili dal sistema. L'utente ha il compito di controllare il numero di thread.

#### ❖ Modello Many-to-many

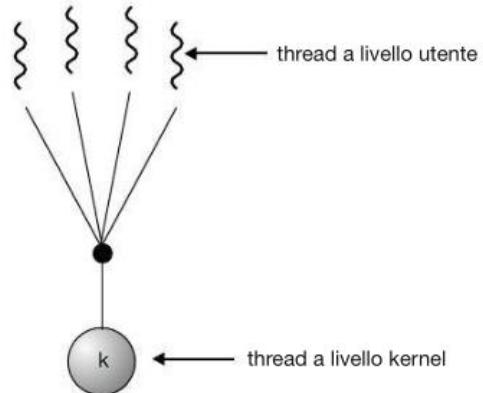
- Il modello da molti a molti mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel

In questo modello i programmati possono **creare liberamente i thread** che ritengono necessari, e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Inoltre, se un thread impiega una chiamata di sistema bloccante, il kernel può schedulare un altro thread. L'utente quindi può lanciare più thread, controlla il sistema e crea il giusto numero di thread.

#### ❖ Two level model

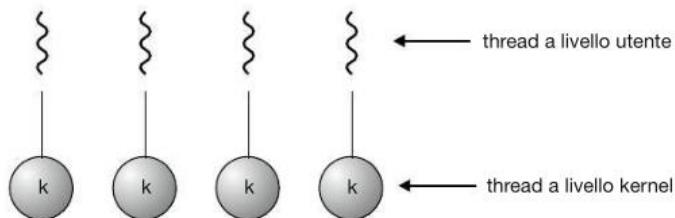
- E' una **variante del modello many to many** che mantiene la corrispondenza fra più thread utente e un numero minore o uguale di thread del kernel, ma **permette anche di vincolare un thread utente a un solo thread del kernel**.

#### ❖ Modello Many-to-one



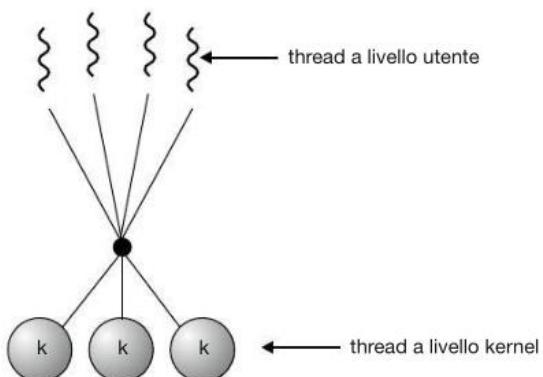
**Figura 4.5** Modello da molti a uno.

#### ❖ Modello One-to-one



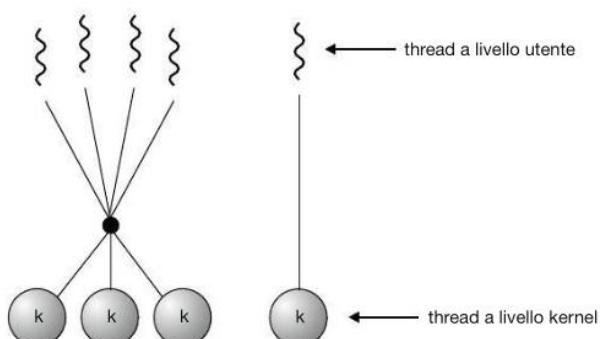
**Figura 4.6** Modello da uno a uno.

#### ❖ Modello Many-to-many



**Figura 4.7** Modello da molti a molti.

#### ❖ Two level model



## ❖ Librerie di thread

Una libreria per i thread fornisce al programmatore una **API** per la creazione e la gestione dei thread.

I metodi con cui **implementare** una libreria per i thread sono essenzialmente 2:

- **libreria collocata interamente a livello utente:** il codice e le strutture dati per la libreria risiedono tutti nello spazio utente. I thread sono implementati al livello utente (gestiti con chiamate di funzione utente).
- **implementazione di una libreria a livello kernel:** è supportata direttamente dal sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. I thread sono gestiti con chiamate di sistema al kernel.

Attualmente, sono tre le librerie di thread maggiormente in uso: Pthreads di POSIX, Windows Java thread

## ❖ Pthreads

Col termine **Pthreads** ci si riferisce allo standard **POSIX** che definisce una api per la creazione e la sincronizzazione dei thread. Non si tratta di una implementazione, ma di una specifica del comportamento dei thread, **l'implementazione è lasciata al programmatore**. Può essere realizzata sia come libreria a livello utente sia a livello kernel.

Possiamo implementare thread **sincroni** e **asincroni**. Si parte sempre da un thread principale che chiama la creazione di altri thread, attraverso una chiamata di sistema o di funzione che si avverrà di chiamate di sistema.

- Nel caso sincrono il thread genitore aspetta i thread figli
- Nel caso asincrono i thread vengono eseguiti in concorrenza

### Esempio di codice Pthreads in C

- Si include la libreria **pthread.h**
- La variabile globale **sum** è condivisa da tutti i thread.
  - Dopo una fork, se un processo figlio cambia una variabile globale, il processo padre non vede quel cambiamento perché esso avviene nel layout di memoria del figlio. **Per i thread questa cosa non avviene!**
- All'inizio dell'esecuzione il processo viene visto come single-threaded con un thread principale (il main)
- Si dichiara il thread identifier e i suoi attributi
- Il programma legge un numero tramite il primo argomento passato da linea di comando argv[1] ed effettua la somma di tutti i numeri che vanno da 0 a quel numero
- La funzione **pthread\_attr\_init** inizializza gli attributi del thread
- La funzione di creazione del thread è la **pthread\_create**, prende in ingresso: un tid, gli attributi, la funzione di avvio del thread (runner) e l'argomento passato al programma (argv[1]).
- A questo punto il thread principale crea un thread e si mette in wait aspettando il thread figlio che finisce attraverso la funzione bloccante **pthread\_join (equivalente della wait)** che riceve in ingresso il tid del thread figlio.
- La funzione di avvio del thread è la **runner** che come argomento prende un puntatore a void che viene castato in qualsiasi altra cosa: questo per essere il più generica possibile
- Il thread vede la variabile sum globale ed esegue la computazione
- Il thread termina con **pthread\_exit(0)** e restituisce il controllo al thread padre che lo stava aspettando. Se terminasse con la system call **exit()** il thread figlio chiuderebbe anche il processo principale e avremmo perso tutto!
- Se lanciamo più volte la funzione runner con diverse **pthread\_create** che insistono sulla stessa variabile globale si viene a creare una **corsa critica su una variabile condivisa tra 2 thread**. La variabile ad esempio viene incrementata da entrambi i thread → sono necessari

meccanismi di sincronizzazione.

```
#include <pthread.h>
#include <stdio.h>

int sum; /* questo dato è condiviso dai thread */
void *runner(void *param); /* i thread chiamano questa funzione */

int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore del thread */
    pthread_attr_t attr; /* insieme di attributi del thread */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* reperisce gli attributi predefiniti */
    pthread_attr_init(&attr);
    /* crea il thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* attende la terminazione del thread */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* Il thread comincia l'esecuzione da questa funzione */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

#### ❖ Pthread\_create POSIX

```
#include <pthread.h>

int pthread_create ( pthread_t *tid, const pthread_attr_t *attr,
                     void * ( *start_func) (void *), void *arg );
```

- se la chiamata ha successo, *tid* punta al thread ID;
- *attr* permette di specificare gli attributi del thread (se *attr* = NULL, gli attributi sono quelli di default);
- *start\_func* è l'indirizzo della funzione di avvio;
- *arg* è l'indirizzo dell'argomento accettato dalla funzione di avvio;
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.

#### ❖ Pthread\_exit POSIX

```
void pthread_exit(void *status);
```

- termina il thread corrente, con valore di uscita *status*
- altri thread possono raccogliere il valore di uscita usando *pthread\_join* (vedere slide successiva)

#### ❖ Pthread\_join POSIX

```
#include <pthread.h>

int pthread_join ( pthread_t *tid, void **status );
```

- *tid* è l'ID del thread del quale si vuole attendere la terminazione;
- *status* punta al valore restituito dal thread per cui si è atteso, indicante il suo stato di terminazione (se *status* = NULL, tale stato non viene restituito);
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.

#### ❖ Pthread\_self POSIX

Identifica il TID del thread corrente su cui si richiama.



#### Pthread\_cancel

POSIX

Viene cancellato il thread in modo deferred passando il tid

❖ Pthread: altro esempio di codice

```
/* thread_create: stampa i TID del main thread e di due altri
thread */

#include <pthread.h>
#include <stdio.h>
#include <errno.h>

void *start_func(void *arg) /* funzione di avvio */
{
    printf("%s", (char *)arg);
    printf(" and my TID is: %d\n", (int)pthread_self());
}

int main(void)
{
    int en;
    pthread_t tid1, tid2;
    char *msg1 = "Hello world, I am thread #1";
    char *msg2 = "Hello world, I am thread #2";

    printf("The launching process has PID:%d\n", (int)getpid());

    printf("The main thread has TID:%d\n", (int)pthread_self());

    /* crea il 1mo thread */
    if ((en = pthread_create(&tid1, NULL, start_func, msg1)) !=0)
        errno=en, perror("pthread_create"), exit(1);

    /* crea il 2ndo thread */
    if ((en = pthread_create(&tid2, NULL, start_func, msg2)) !=0)
        errno=en, perror("pthread_create"), exit(2);

    /* attende per il 1mo */
    if ((en = pthread_join(tid1, NULL)) !=0)
        errno=en, perror("pthread_join"), exit(1);

    /* attende per il 2ndo */
    if ((en = pthread_join(tid2, NULL)) !=0)
        errno=en, perror("pthread_join"), exit(2);

    return 0;
}
```

❖ **Pthread: altro esempio di codice con variabili LOCALI e variabili GLOBALI e variabili allocate DINAMICAMENTE**

```

typedef struct foo{
    int a;
    int b;
} myfoo;

myfoo test; // Variabile GLOBALE

void stampa(char *st, struct foo *test){
    printf("%s: tid=%d a=%d b=%d\n", st, pthread_self(), test->a, test->b);
}

void *fun1(void *arg){
    myfoo test2 = {1,2}; // Variabile LOCALE
    printf("%s %d\n", arg, pthread_self());
    stampa(arg, &test2);
    pthread_exit((void *)&test2);
}

void *fun2(void *arg){
    test.a = 3;
    test.b = 4; // Variabile GLOBALE
    printf("%s %d\n", arg, pthread_self());
    stampa(arg, &test);
    pthread_exit((void *)&test);
}

void *fun3(void *arg){
    myfoo *test3;
    test3=malloc(sizeof(struct foo)); // Variabile allocata dinamicamente
    test3->a = 5;
    test3->b = 6;
    printf("%s %d\n", arg, pthread_self());
    stampa(arg, test3);
    pthread_exit((void *)test3); //c
}

int main(void){
    char st[100];
    pthread_t tid1;
    pthread_t tid2;
    pthread_t tid3;

    myfoo *b; // PUNTATORE alla struttura (non allocata)

    pthread_create(&tid1, NULL, fun1, "Thread 1"); // Locale
    pthread_join(tid1, (void *)&b);
    stampa("Master ", b);

    pthread_create(&tid2, NULL, fun2, "Thread 2"); // Globale
    pthread_join(tid2, (void *)&b);
    stampa("Master ", b);

    pthread_create(&tid3, NULL, fun3, "Thread 3"); // Dinamica
    pthread_join(tid3, (void *)&b);
    stampa("Master ", b);
}

```

Thread 1: 1077283760 // Locale
Thread 1: a=1 b=2
Master : a=1075156600 b=1077281896
Thread 2: 1077283760 // Globale
Thread 2: a=3 b=4
Master : a=3 b=4
Thread 3: 1077283760 // Dinamica
Thread 3: a=5 b=6
Master : a=5 b=6

OUTPUT

## ❖ Thread in Linux (task)

Linux prevede la chiamata di sistema **clone()** per generare un nuovo thread. **Tuttavia Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine task in riferimento a un flusso del controllo nell'ambito di un programma.** Quando clone() è invocata, riceve come parametro un insieme di indicatori (flag), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio. I flag sono:

- **CLONE\_FS**: Condivisione delle informazioni sul file system
- **CLONE\_VM**: Condivisione dello stesso spazio di memoria
- **CLONE\_SIGHAND**: Condivisione dei gestori dei segnali
- **CLONE\_FILES**: Condivisione dei file aperti

## ❖ Thread in Java

- I thread in Java sono gestiti dalla JVM
- Tipicamente implementati con il modello di thread fornito dal sistema operativo utilizzato (es. Unix mediante Pthreads)

I thread java possono essere creati in 2 modi:

- **Estendendo la classe Thread con override del metodo run()**

```
public class Thread1 extends Thread{  
    @Override  
    public void run(){  
        System.out.println("Codice del thread 1");  
    }  
}  
  
public class ProvaThread1 {  
    public static void main(String[] args) {  
        Thread1 thread1 = new Thread1();  
        thread1.start();  
    }  
}
```

thread1.start(); rende il thread eleggibile per l'esecuzione.

Una volta in esecuzione parte il run()

- **Creando una classe che implementi l'interfaccia Runnable (più comunemente usato)**

```
public interface Runnable  
{  
    public abstract void run();  
}  
  
public class Thread2 implements Runnable{  
    public void run(){  
        System.out.println("Codice del thread 2");  
    }  
}
```

- per implementare runnable, una classe è tenuta a definire il metodo run(). Il codice che implementa run() sarà eseguito in un thread distinto

## ❖ Threading implicito

- E' una strategia adottata per affrontare ostacoli della programmazione multithreading e gestire al meglio la progettazione di applicazioni multithread. **Consiste nel trasferimento della creazione e della gestione del threading dai programmatori ai compilatori e alle librerie di runtime.**
- Il programmatore individua dei task non dei thread

Tre metodi considerati

- Thread Pools
- OpenMP
- Grand Central Dispatch

## ❖ Problematiche threading

Nella programmazione multithread sono presenti diverse problematiche:

- **Semantica delle chiamate fork() ed exec()**
- **Signal handling: Sincrono e asincrono**
- **Cancellazione thread: Asincrono o deferred**
- **Thread-local storage**
- **Attivazione Scheduler**

### ❖ Semantica delle chiamate fork() ed exec()

Se un thread in un programma invoca la chiamata di sistema **fork()**, il nuovo processo potrebbe contenere un duplicato di tutti i thread del processo oppure del solo thread invocante. Alcuni sistemi UNIX includono entrambe le versioni. Se un thread invoca la chiamata di sistema **exec()**, il programma specificato come parametro della exec() sostituisce l'intero processo, inclusi tutti i thread.

### ❖ Signal handling: Sincrono e asincrono

Nei sistemi UNIX si usano i **segnali** per comunicare ai processi il verificarsi di determinati eventi. Un segnale si può ricevere in modo sincrono o asincrono. Quando viene generato un segnale da un evento e mandato ad un processo questo lo processa tramite un **signal handler**. Il signal handler può essere di default o user-defined. Ogni segnale ha un default handler che il kernel esegue. Un **segnaile** possiamo definirlo come un **interrupt software (o meglio exception)** e consente una comunicazione asincrona tra processi e/o tra device e processo. I segnali vengono inviati in modo asincrono, non e' possibile sapere quando il processo ricevera' un segnale. E' possibile indicare al kernel l'azione da intraprendere quando un segnale e' generato per un processo: **Ignora** - Valida per quasi tutti i segnali tranne SIGKILL e SIGSTOP. **Catch del segnale**: Indicare una procedura da eseguire (signal handler). **Default**: Eseguire l'azione di default.

Se si manda un segnale ad un processo multithread come dobbiamo gestire la situazione?

- Invia il segnale al thread a cui si applica
- Invia a tutti i thread del processo
- Invia a specifici thread del processo
- Assegna ad un thread sepecifico il compito di ricevere tutti i segnali del processo

In UNIX il primo thread disponibile attiva l'handler del segnale.

## ❖ Cancellazione thread: Asincrono o deferred

La cancellazione dei thread è l'operazione che permette di terminare un thread prima che completi il suo compito. Un thread da cancellare è spesso chiamato **thread bersaglio** (target thread). La cancellazione di un thread bersaglio può avvenire in due modi diversi:

- **cancellazione asincrona:** un thread fa immediatamente terminare il thread target;
- **cancellazione deferred:** il thread target controlla periodicamente se deve terminare, in modo da effettuare la terminazione in maniera ordinata

La cancellazione di un thread in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema. Il **default è deferred**: la cancellazione avviene solo quando il thread raggiunge un cancellation point e allora viene invocato un cleanup handler.

**In Posix:** in ogni istante, un thread può essere cancellabile o non cancellabile. Quando partono tutti i thread sono cancellabili. Quando un altro thread chiama la funzione **pthread\_cancel**: se il thread è cancellabile, viene cancellato; se non è cancellabile, la richiesta di cancellazione viene memorizzata, in attesa che il thread diventi cancellabile.

### Impostare la cancellabilità:

```
int pthread_setcancelstate(int state, int *oldstate);
```

imposta la cancellabilità a **state** e restituisce la vecchia cancellabilità in **oldstate**

state e oldstate possono assumere i valori:

- **PTHREAD\_CANCEL\_ENABLE**
- **PTHREAD\_CANCEL\_DISABLE**

restituisce 0 se OK, un codice d'errore altrimenti

## ❖ Thread-local storage

Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati **TLS (thread-local storage)**. È facile confondere i dati specifici dei thread con le variabili locali. mentre le variabili locali sono visibili solo durante la chiamata di una singola funzione, i dati specifici sono visibili attraverso tutte le chiamate.

**In Posix** si fa uso dei **Thread specific data** in cui ogni thread possiede un'area di memoria privata, la TSD area, indicizzata da chiavi.

## ❖ Attivazione Scheduler

Sia modelli Many-to-Many che Two-level richiedono di mantenere un numero appropriato di kernel thread allocati per l'applicazione. Tipicamente usata una struttura dati intermedia tra user e kernel thread chiamata **lightweight process (LWP)** che è come un processore virtuale su cui il processo può schedulare l'esecuzione di user thread. Ogni LWP è associato ad un kernel thread.

## **4. SCHEDULING DELLA CPU**

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

### ❖ Concetti di base

- **L'obiettivo della multiprogrammazione è avere sempre un processo in esecuzione, in modo da massimizzare l'utilizzazione della cpu.**
  - **Come si ottiene?** Si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della cpu per cederlo a un altro processo.
- Lo **scheduling** è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la cpu è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi. Ci vuole un criterio per scegliere come allocare la CPU ai vari processi.
  - **Scheduling della CPU:** commuta l'uso della CPU tra i vari processi.

### ❖ CPU Brust e I/O Brust

L'esecuzione del processo consiste in un **ciclo d'elaborazione (svolto dalla cpu)** e da un'attesa del completamento delle **operazioni di I/o**. I processi si alternano tra questi due stati detti CPU brust e I/O brust.

- **CPU burst:** è un tempo (**0-8 millisecondi**) in cui si verifica una sequenza di operazioni d'elaborazione svolte esclusivamente dalla cpu. Uso continuato della CPU.
  - Un programma con prevalenza d'elaborazione (**cpu bound**) può produrre varie sequenze di operazioni della cpu molto lunghe (tanti cpu burst di lunga durata).
- **I/O burst:** è un tempo in cui si verifica una sequenza di operazioni di I/o.
  - Un programma con prevalenza di I/o (**i/o bound**) produce generalmente molte sequenze di operazioni della cpu di breve durata. (pochi cpu burst e di breve durata)

### ❖ CPU Scheduler. Scheduling preemptive e non preemptive.

Lo scheduler della CPU è detto **short-term scheduler** ed è colui che tra i processi in memoria pronti per l'esecuzione in **ready queue**, sceglie quello cui assegnare la cpu. **Le code ready possono essere ordinate in modo diverso.**

**Le decisioni di CPU scheduling occorrono quando i processi:**

1. Passano dallo stato running a waiting
2. Passano dallo stato running a ready
3. Passano dallo stato waiting a ready
4. Terminano

Quando lo scheduling interviene solo nelle condizioni **1 e 4**, si dice che lo schema di scheduling è **senza prelazione** (nonpreemptive); altrimenti, lo schema di scheduling è **con prelazione (preemptive)**.

- **Scheduling senza prelazione:** quando si assegna la cpu a un processo, questo rimane in possesso della cpu fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio nello stato di attesa.
- **Scheduling con prelazione:** la prelazione (interruzione) di processi in esecuzione prima della loro terminazione, può portare a diverse problematiche come
  - Problema accesso a dati condivisi, situazioni di inconsistenza dati
  - Problema prelazione in modalità kernel

- Problema interruzioni durante attività curciali del SO

### ❖ Dispatcher

E' il modulo che **passa effettivamente il controllo della cpu al processo scelto** dallo short term scheduler.

Il dispatcher effettua:

- il cambio di contesto;
- il passaggio alla user mode;
- salto alla corretta locazione del programma utente per ripartire con quel programma

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto **più rapido possibile**.

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

### ❖ Criteri di scheduling

Per il confronto tra gli algoritmi di scheduling della cpu sono stati suggeriti molti criteri. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore.

- **Utilizzo CPU:** percentuale di CPU utilizzata; occorre mantenere la CPU quanto più occupata possibile.
- **Throughput:** numero di processi che completano l'esecuzione per unità di tempo.
- **Turnaround time:** tempo necessario per eseguire completamente il processo stesso. E' l'intervallo che intercorre tra la sottomissione del processo e il completamento della sua esecuzione. Si ottiene come somma dei tempi passati nell'attesa dell'ingresso in memoria, nella ready queue, durante l'esecuzione nella cpu e nelle operazioni di I/o. Il **turnaround time medio** può essere influenzato da **starvation**.
- **Waiting time:** tempo di attesa per un processo nella coda ready. Si ottiene come la somma di tutti gli intervalli di tempo passati nella coda ready.
- **Response time:** tempo che intercorre tra la effettuazione di una richiesta e la prima risposta prodotta. Importante in sistemi interattivi, alternativa al Turnaround time.

Obiettivi di **ottimizzazione**:

- Massimo utilizzo CPU
- Massimo throughput
- Minimo tempo di turnaround
- Minimo tempo di waiting
- Minimo tempo di risposta
- Varianza dei tempi di risposta

### ❖ Algoritmi di scheduling

- **First-Come First-Served (FCFS) Scheduling**
- **Shortest-Job-First (SJF) Scheduling**
- **Shortest-remaining-time-first**
- **Scheduling con Priorità**
- **Round Robin (RR)**
- **Round Robin con Priorità**
- **Scheduling a Code Multiple**
- **Scheduling a Code Multiple con Feedback**

**Diagramma di Gantt:** diagramma a barre che illustra una evoluzione temporale dei processi che la cpu esegue (timeline), includendo i tempi d'inizio e fine di ogni processo partecipante.

#### ❖ First-Come First-Served (FCFS) Scheduling

E' uno scheduling in ordine d'arrivo: la cpu si assegna al processo che la richiede per primo. La realizzazione si basa su una **coda FIFO**. Quando un processo entra nella ready queue si mette in "coda" se ci sono altri processi da eseguire, in attesa di essere eseguito. **Quando la cpu è libera, viene assegnata al processo che si trova alla testa della coda, rimuovendolo da essa.**

**Svantaggio: tempo medio d'attesa abbastanza lungo.** Può variare grandemente all'aumentare della variabilità dei cpu burst dei vari processi.

L' algoritmo FCFS inoltre produce un **effetto convoglio**: può accadere che tutti i processi brevi attendono che un lungo processo liberi la cpu, il che causa una riduzione dell'utilizzo della cpu e dei dispositivi e aumento del tempo di attesa medio. **Soluzione: si eseguono per primi i processi più brevi.**

**L'algoritmo di scheduling FCFS è senza prelazione;** una volta che la cpu è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/o.

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Assumendo che i processi nell'ordine:  $P_1, P_2, P_3$   
Il Gantt Chart dello schedule è:



- Tempi di attesa per  $P_1 = 0; P_2 = 24; P_3 = 27$
- Media del tempo di attesa:  $(0 + 24 + 27)/3 = 17$

Assumendo i processi nell'ordine:

$$P_2, P_3, P_1$$

- Il Gantt diventa:



- Tempi di attesa  $P_1 = 6; P_2 = 0; P_3 = 3$
- Tempo di attesa medio:  $(6 + 0 + 3)/3 = 3$
- Molto meglio di prima
- **Effetto Convoy** - processi brevi dietro i processi lunghi
  - Considera un processo CPU-bound e tanti I/O-bound
- Sbrigare i processi brevi per migliorare i tempi di risposta

## ❖ Shortest-Job-First (SJF) Scheduling

- Questo algoritmo associa a ogni processo la lunghezza del successivo CPU Burst. Quando è disponibile, si assegna la cpu al processo che ha la più breve lunghezza del prossimo CPU Burst.
- Se due processi hanno le successive sequenze di CPU Burst della stessa lunghezza si applica lo scheduling FCFS.
- Si può dimostrare che l'algoritmo di scheduling SJF è ottimale, nel senso che rende minimo il tempo d'attesa medio per un dato insieme di processi
- Versione con prelazione: shortest-remaining-time-first

## ❖ Lunghezza del prossimo CPU Burst

La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della cpu. Un possibile approccio a questo problema consiste nel tentare di approssimare lo scheduling: se non è possibile conoscere la lunghezza della prossima sequenza di operazioni della cpu, si può cercare di predire il suo valore; è probabile, infatti, che sia simile alle precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

- uso di stime basate sul comportamento passato ed esecuzione del processo con il minor tempo di esecuzione stimato

La lunghezza della successiva sequenza di CPU Burst generalmente si stima calcolando la media esponenziale delle lunghezze misurate dei precedenti CPU Burst. Si può solo stimare, ed è simile ai precedenti. I valori più lontani pesano di meno, quelli più vicino pesano di più.

1.  $t_n$  = actual length of  $n^{th}$  CPU burst
2.  $\tau_{n+1}$  = predicted value for the next CPU burst
3.  $\alpha, 0 \leq \alpha \leq 1$
4. Define :  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .

Il parametro alfa  $\alpha$  fa da bilanciamento tra valore presente e valori passati, varia tra 0 e 1.

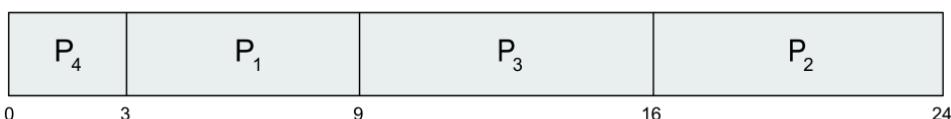
Se settato a 0, tiene conto solo dei valori passati.

Se settato a 1 tiene conto solo del valore presente del CPU Burst.

Per bilanciamento quindi solitamente viene settato a  $\frac{1}{2}$ .

Process	Burst Time
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

## □ SJF scheduling chart



□ Tempo di attesa medio =  $(3 + 16 + 9 + 0) / 4 = 7$

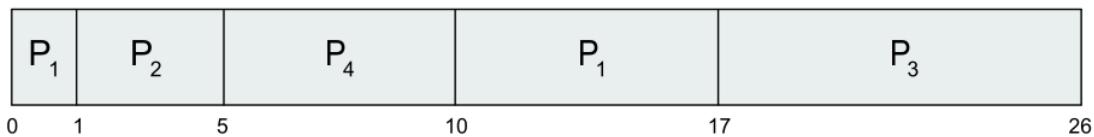
### ❖ Shortest-remaining-time-first

- E' la versione con prelazione dell'algoritmo SJF.
- La scelta si presenta quando alla ready queue arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può richiedere una CPU Burst più breve di quella che resta al processo correntemente in esecuzione, dunque tale processo viene prelazionato a favore di quello con CPU burst più breve.

- Si considerano diversi tempi di arrivo e si introduce la prelazione

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive SJF Gantt Chart



- Tempo attesa medio =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$  msec  
 □ Senza prelazione 8.75 msec

- All'istante 0 si avvia il processo p1, poiché è l'unico che si trova nella coda. All'istante 1 arriva il processo p2. Il tempo necessario per completare il processo p1 (7 millisecondi) è maggiore del tempo richiesto dal processo p2 (4 millisecondi), perciò si effettua la prelazione del processo p1 sostituendolo col processo p2. E così via.

**Il tempo d'attesa medio (tempo di inizio esecuzione - tempo di arrivo) è dato da:**

$$P1: (10 - 1) = 9$$

$$P2: (1 - 1) = 0$$

$$P3: \quad \quad \quad (17 - 2) = 15$$

$$P4: (5 - 3) = 2$$

$$\text{TOT} = 26/4 \text{ processi} = 6.5 \text{ unità di tempo}$$

### ❖ Scheduling con Priorità

- si associa una priorità a ogni processo e si assegna la cpu al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS. Numero di priorità (intero) associato ad ogni processo. I numeri bassi indicano priorità alte.

Può essere con prelazione o senza prelazione: in quello **con prelazione** si sottrae la cpu al processo attualmente in esecuzione se la priorità del processo appena arrivato è superiore

Un **problema** di questo algoritmo è l' **attesa indefinita (starvation)**: l'algoritmo può lasciare processi a bassa priorità nell'attesa indefinita della cpu. Praticamente un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla cpu.

Una **soluzione** al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento (aging)**; si tratta di una tecnica di **aumento graduale delle priorità** dei processi che attendono nel sistema da parecchio tempo.

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

### □ Priority scheduling Gantt Chart



### □ Tempo di attesa medio = 8.2 msec

$$\text{Tempo di attesa medio} = 0 + 1 + 6 + 16 + 18 = 41/5 = 8.2$$

## ❖ Round Robin

E' stato progettato appositamente per i sistemi time sharing; è simile allo scheduling FCFS, ma aggiunge la capacità di **prelazione** in modo che il sistema possa commutare fra i vari processi.

Ciascun processo riceve una piccola quantità fissata del tempo della cpu, chiamata **quanto di tempo** (time slice), che varia generalmente da **10 a 100 millisecondi**; la ready queue è trattata come una **coda circolare FIFO**.

I nuovi processi si aggiungono alla fine della ready queue. Lo scheduler della cpu prende il primo processo dalla ready queue, imposta un **timer** in modo che **invii un segnale d'interruzione alla scadenza** di un intervallo pari a **1 quanto di tempo**, e attiva il dispatcher per eseguire il processo. A questo punto si può verificare una delle due seguenti situazioni:

- **il processo ha una cpu burst di durata minore di un quanto di tempo**, quindi il processo stesso rilascia volontariamente la cpu e lo scheduler passa al **processo successivo** della ready queue;
- **la cpu burst è più lunga di un quanto di tempo**: in questo caso il timer scade e invia un segnale d'interruzione al sistema operativo, che esegue un **cambio di contesto** e mette il processo alla fine della ready queue. Lo scheduler quindi seleziona il **processo successivo** nella ready queue.

### Performance RR

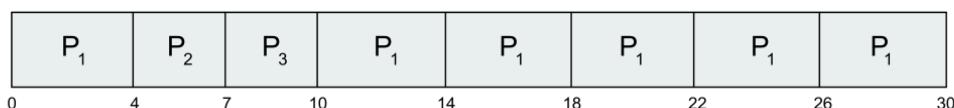
Le prestazioni dell'algoritmo rr dipendono molto dalla dimensione del quanto di tempo.

- **Se il quanto di tempo q è molto largo**, l'algoritmo si riduce ad uno scheduling **FCFS**.
- **Se il quanto di tempo q è molto breve** (per esempio, un millisecondo), rr può portare a un **numero elevato di cambi di contesto**.
- **Tempo di turnaround varia con la dimensione del quanto e non migliora necessariamente con l'aumento della dimensione del quanto di tempo**

## Esempio di RR con Quanto = 4

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Gantt chart:



- Tipicamente maggiore turnaround medio di SJF, ma migliore **risposta**
- q deve essere grande rispetto al tempo di context switch
- q di solito tra 10ms e 100ms, context switch < 10 microsec

❖ Round Robin con Priorità

## RR con Priorità

---

Si segue la priorità, processi con pari priorità con RR

Process	Burst Time	Priority	millisecondi
$P_1$	4	3	
$P_2$	5	2	
$P_3$	8	2	
$P_4$	7	1	
$P_5$	3	3	

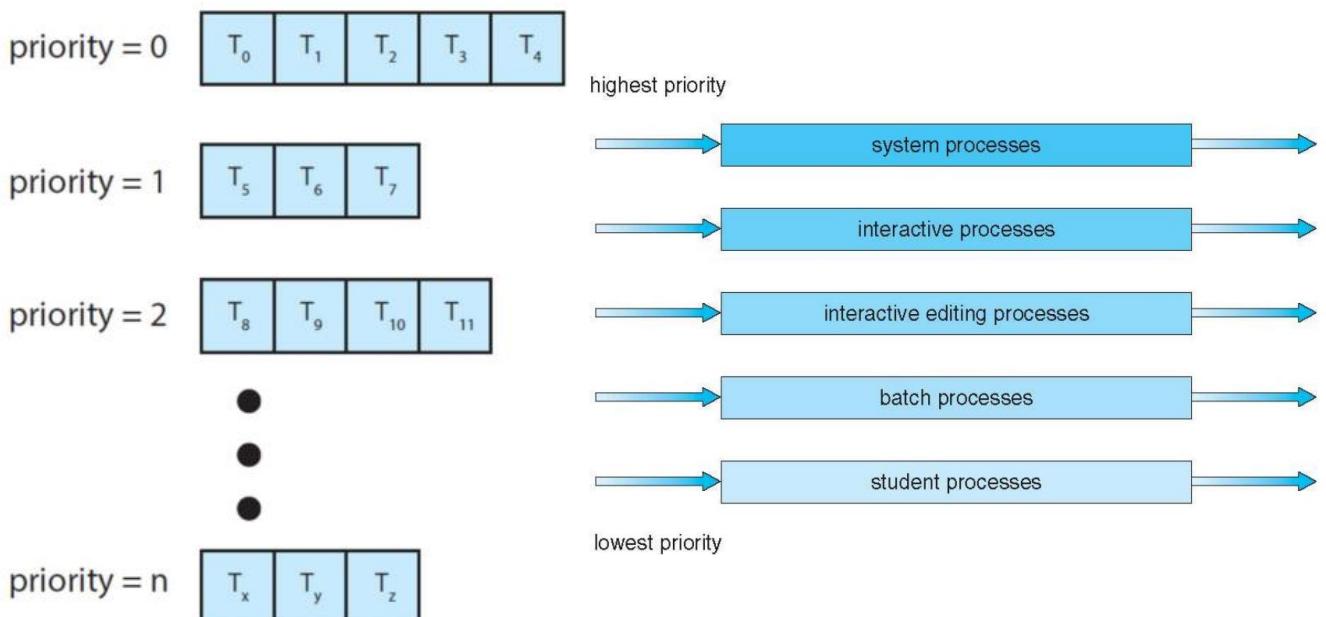
Quantum 2 millisecondi



## ❖ Scheduling a Code Multiple

Questo algoritmo classifica i processi in gruppi diversi a seconda del loro tipo. Una distinzione comune è per esempio quella che si fa tra i processi che si eseguono in **foreground** (primo piano), o interattivi, e i processi che si eseguono in **background** (sottofondo), o **batch** (a lotti). Questi due tipi di processi hanno differenti requisiti sui tempi di risposta e possono quindi avere diverse necessità di scheduling.

- L'algoritmo suddivide la **ready queue** in code distinte di una certa priorità
- I processi si assegnano in modo fisso a una coda, generalmente secondo qualche caratteristica del processo e non possono spostarsi fra le code.
- Ogni coda ha il proprio algoritmo di scheduling.
  - processi in foreground: RR, background: FCFS
- E' necessario uno **scheduling tra le code**: a priorità fissa con prelazione. Ogni coda ha la priorità assoluta sulle code di priorità più bassa. **Problema: starvation**.
- Altra soluzione: ogni coda prende un certo quanto di CPU che può schedulare tra i suoi processi



## ❖ Scheduling a Code Multiple con Feedback

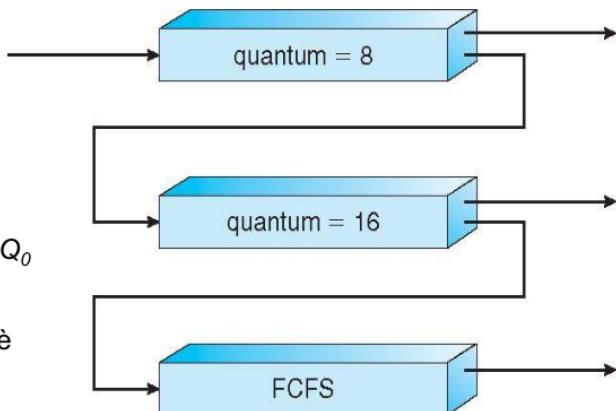
- Lo scheduling a code multiple con feedback è simile al allo scheduling a code multiple, ma permette ai processi di **spostarsi fra le code**. I processi si separano in termini di **CPU burst**: se un processo usa troppo tempo di elaborazione della cpu, viene spostato in una coda con priorità più bassa.
- Questo schema mantiene i processi con prevalenza di I/O Bound e processi interattivi nelle code con priorità più elevata.
- Inoltre, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo in una coda a priorità bassa. Questa tecnica di **aging** (aumento di priorità nel tempo) impedisce il verificarsi di **starvation**.
- Ad ogni coda è assegnato un algoritmo di scheduling diverso.
- E' il più generale criterio di scheduling della cpu, che nella fase di progettazione si può adeguare a un sistema specifico, ma è anche il più complesso.

Lo scheduler di code multiple con feedback definito da:

- Numero di code
- Algoritmi di scheduling per ogni coda
- Metodi per determinare quando alzare la priorità di processo
- Metodi per determinare quando abbassare la priorità un processo
- Metodi per determinare quale coda assegnare ad un processo quando richiede un servizio

### □ Si considerino tre code:

- $Q_0$  – RR quanto di tempo 8 msec
- $Q_1$  – RR quanto di tempo 16 msec
- $Q_2$  – FCFS



### □ Scheduling

- Prelazione con priorità 0, 1, 2
- Nuovo processo in Ready messo in coda  $Q_0$ 
  - Il processo riceve 8 msec di CPU
  - Se non finisce in 8 msec, il processo è mosso in coda  $Q_1$
- Se vuota  $Q_0$ 
  - il primo processo in  $Q_1$  riceve 16 msec
  - Se non completa viene prelazionato e si sposta sulla coda  $Q_2$
- Se vuote  $Q_0$  e  $Q_1$  vengono eseguiti i processi in coda  $Q_2$  con FCFS
- Schedulatore che dà priorità a processi rapidi e mette in coda quelli più lunghi

## ❖ Scheduling di Thread

Occorre distinguere tra thread user-level e kernel-level. I moderni SO schedulano thread kernel-level, non processi. I thread user-level sono gestiti da librerie utente ed il kernel non ne è al corrente. Per essere eseguiti e schedulati i thread user-level devono essere mappati in kernel-level.

- Nei modelli **many-to-one e many-to-many**, la libreria dei thread gestisce gli **user-level threads** per poi lanciare i lightweight processes (LWPs). In questo caso lo schema per la schedulazione dei thread è detto **Process-Contention Scope (PCS)**: la competizione per la CPU avviene tra thread dello stesso processo. Tipicamente viene applicato uno scheduling con priorità (definito dal programmatore).
- Nella schedulazione di **kernel-thread** in CPU invece il kernel usa uno schema **System-Contention Scope (SCS)**: ossia c'è una competizione tra tutti i thread nel sistema per ottenere la CPU.
  - **Sistemi che utilizzano modelli one-to-one (Linux, Windows) usano solitamente il modello SCS**

## ❖ Pthread Scheduling

Lo standard Posix fornisce API che permettono di specificare lo schema PCS o SCS durante la creazione dei thread. POSIX Pthread permette di specificare entrambe le modalità:

- **PTHREAD\_SCOPE\_PROCESS usa PCS (Process Contention Scope) scheduling**
- **PTHREAD\_SCOPE\_SYSTEM usa SCS (System Contention Scope) scheduling**
- **Linux permette solo PTHREAD\_SCOPE\_SYSTEM (SCS)**

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* ottiene gli attributi di default */
    pthread_attr_init(&attr);

    /* per prima cosa appura l'ambito della contesa */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Impossibile appurare l'ambito della contesa\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Valore d'ambito della contesa non ammesso.\n");
    }

    /* imposta l'algoritmo di scheduling a PCS o SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* genera i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* adesso aspetta la terminazione di tutti i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* ogni thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
    /* fai qualcosa ... */

    pthread_exit(0);
```

## ❖ Scheduling multiprocessore: concetti e problematiche

Lo scheduling della CPU è più complesso se ci sono più CPU. Diverse architetture disponibili:

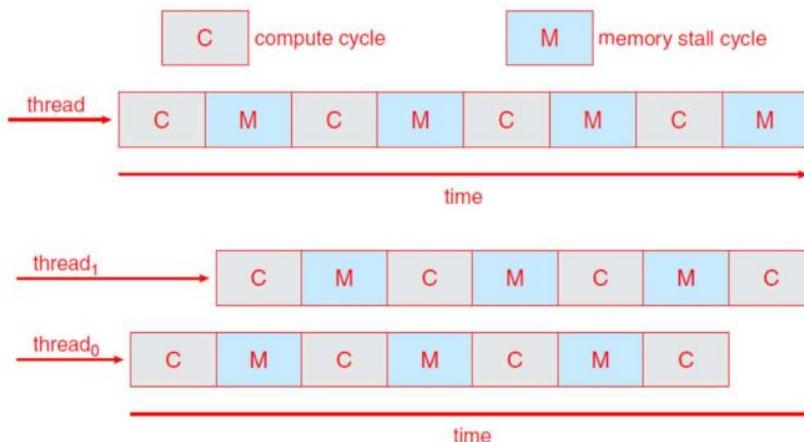
- Multicore CPU
- Multithreaded cores
- Sistemi NUMA (Non-uniform Memory Access)
- Multiprocessamento eterogeneo

Per i primi tre casi assumiamo processori omogenei (stesse capacità).

Diversi tipi di processori:

- **Homogeneous processors** – processori identici per funzionalità
- **Asymmetric multiprocessing** – un solo processore (master) prende decisioni di scheduling e accede alle strutture dati di sistema senza condivisione di dati. Semplifica la gestione, ma il processore server master fa da collo di bottiglia.
- **Symmetric multiprocessing (SMP)** – approccio standard: ogni processore fa un self-scheduling
  - tutti i processi in una coda ready
  - ogni processore ha una sua coda ready privata: soluzione più adottata.

❖ **Sistemi multicore:** ogni core appare al SO come una CPU separata. I core sono veloci, ma abbiamo problemi di **memory stall** in attesa di dati pronti in memoria. La soluzione è quella di preparare una serie di thread su cui switchare nei momenti in cui c'è un accesso alla memoria da parte di altri thread. Quindi preparamo **più thread pronti per core da gestire: chip multithreading**. Ci saranno due livelli di thread da gestire: **hardware thread (che vanno mappati)** e **software thread**.



## ❖ Problemi dei sistemi multicore

- **Bilanciamento del carico:** se i processori li consideriamo tutti simmetrici per caratteristiche e funzionalità è necessario bilanciare il loro carico in modo uguale. A volte il carico di un processore potrebbe sbilanciarsi, quindi è necessario fare un **ribilanciamento del carico** secondo 2 modi:
  - **Push migration:** un processo verifica continuamente il carico dei processori, se trova uno sbilanciamento sposta/spinge il task dalla CPU sovraccarica alle altre
  - **Pull migration:** i processori in idle prendono/tirano i task in attesa sulle code dei processori occupati.
- **Processor affinity:** un processo ha un “affinità” per il processore su cui è in running. Fare il context switch su un altro processore ha un certo costo e può essere problematico. Modalità di gestione:
  - **soft affinity:** si tenta di mantenere il processore su cui è già stato in running il thread
  - **hard affinity:** si specifica un sottoinsieme di processori per il thread su cui può runnare

## ❖ Real-time CPU scheduling

Lo scheduling della cpu nei **sistemi real-time** ha peculiarità proprie. In generale, possiamo distinguere tra **sistemi real-time soft** e **sistemi real-time hard**.

- **Soft real time systems:** non offrono garanzie sul momento in cui un processo sarà eseguito, assicurano solamente che sarà data **precedenza ai processi critici** rispetto a quelli non critici.
- **Hard real time systems:** hanno vincoli più rigidi; i **task vanno eseguiti entro una scadenza prefissata (deadline)** ed eseguirli dopo tale scadenza è del tutto inutile.

I sistemi real-time sono per loro natura guidati dagli eventi. In presenza di un evento, il sistema deve rispondere con la massima velocità possibile. L'**event latency** è il periodo di tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema ne effettua la gestione. In genere, eventi diversi hanno diversi requisti di latenza. **2 tipi di latenze** incidono sulla prestazione:

- **Interrupt latency:** tempo dall'arrivo dell'interrupt allo start della routine di servizio di gestione
- **Dispatch latency:** tempo necessario al dispatcher per bloccare un processo e avviare un altro
  - **fase di conflitto:** prelazione di ogni processo in esecuzione nel kernel; cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità.
  - **fase di dispatch:** allocazione del nuovo processo

### ❖ Priority based scheduling

La caratteristica più importante di un sistema operativo real-time è di rispondere immediatamente a un processo in tempo reale, non appena questo processo richiede la cpu. Lo scheduler di un sistema operativo real-time deve dunque utilizzare un algoritmo con prelazione basato su priorità. Ricordiamo che gli algoritmi di scheduling con priorità assegnano a ogni processo una priorità in base alla sua importanza; ai task più importanti vengono assegnate priorità più elevate rispetto a quelle assegnate ai processi ritenuti meno importanti. Se lo scheduler supporta anche la prelazione, un processo in esecuzione sulla cpu viene interrotto se diventa disponibile per l'esecuzione un processo con priorità più alta.

- **Un algoritmo di scheduling a priorità con prelazione** garantisce solo il **soft real-time systems**. Per un hard real-time system si deve anche garantire le **deadlines**: si considerano dei task periodici e si implementano diversi algoritmi come il **Rate Monotonic Scheduling** e l'**Earliest deadline first**.

### ❖ Task periodici

Sono dei task che dopo un tot. di tempo devono essere rieseguiti e ottengono la CPU a intervalli costanti. Ottenuta la CPU un task periodico ha un tempo di processo t, deadline d, e periodo p che seguono questa relazione:  $0 \leq t \leq d \leq p$ .

## ❖ Rate Monotonic Scheduling

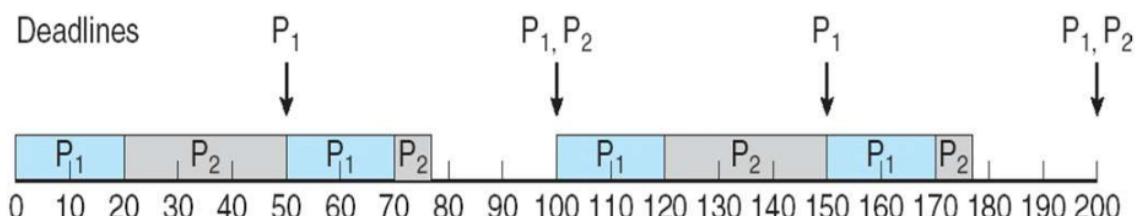
In questo algoritmo di scheduling la **priorità viene assegnata in base all'inverso del periodo**.

- **Periodi brevi = priorità alta**
- **Periodi lunghi = priorità bassa**
- **Priorità alta viene data ai task che intervengono più spesso.**
- **ottimale**, nel senso che se non è in grado di pianificare l'esecuzione di una serie di processi rispettandone i vincoli temporali, nessun altro algoritmo che assegna priorità statiche vi riuscirà
- **limitazione: l'utilizzo della cpu è limitato, per cui non sempre è possibile ricavarne un rendimento ottimale**

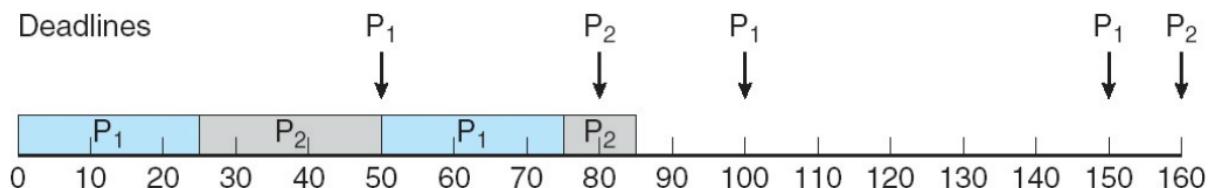
**Assegniamo a p<sub>1</sub> una priorità più elevata rispetto a p<sub>2</sub>, dato che il suo periodo è più breve (50).**

□ Esempio:

- Periodi 50 e 100, tempo di processamento 20, 35
- Deadline: terminare prima del prossimo period
- Uso CPU per P<sub>1</sub> di è 20/50 = 0.4, per P<sub>2</sub> di è 35/100 = 0.35, tot 0.75
- Assumendo RMS:
  - ▶ P<sub>1</sub> con priorità più alta di P<sub>2</sub> entrambe le deadline rispettate



- Se P<sub>1</sub> ha periodo 50 e CPU brust 25 e P<sub>2</sub> periodo 80 e CPU brust 35
- Utilizzo CPU 25/50 + 35/80 = 0.94
- P<sub>2</sub> interrotta da P<sub>1</sub> quando riprende finisce a 85, ma doveva finire ad 80



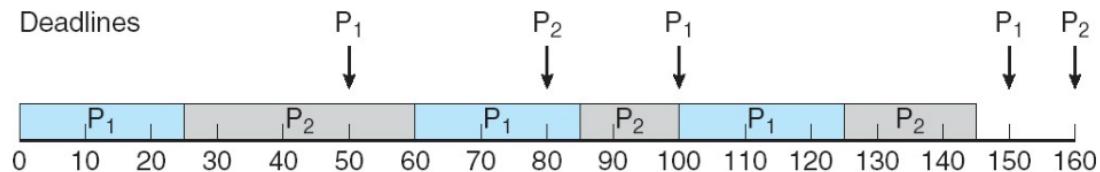
- Non sfrutta completamente la CPU

$$N(2^{1/N} - 1)$$

- Con 1 processo 100%, con 2 processi 83%, con al crescere di N 69%

### ❖ Earliest deadline first (EDF)

- In questo algoritmo le priorità sono assegnate secondo le deadline: più vicina è la scadenza del task, maggiore è la priorità; una scadenza più lontana implica una priorità più bassa.
- L'unico obbligo a carico dei processi è di notificare allo scheduler la propria prossima scadenza nel momento in cui divengano eseguibili
- Teoricamente ottimo, praticamente contano altri parametri.



P1: p1 = 50, t1 = 25; P2: p2 = 80, t2= 35

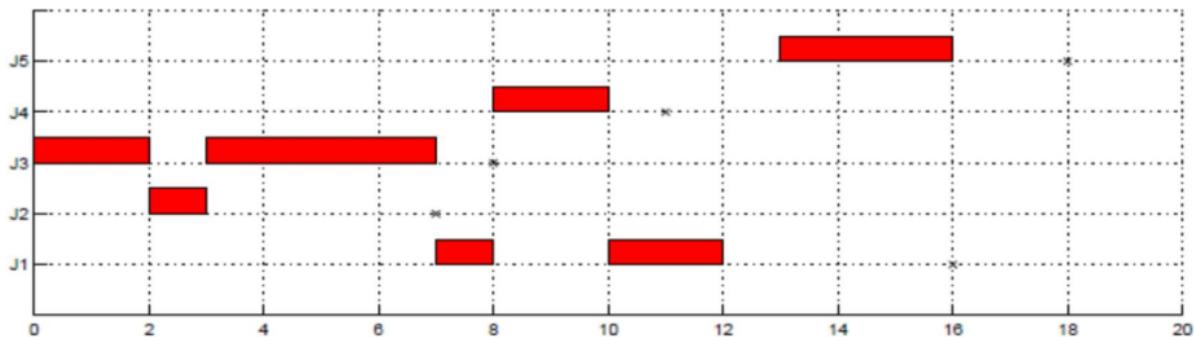
Non richiede la periodicità dei processi e neppure CPU brust costante

Ogni processo deve dichiarare la deadline quando diventa runnable

Teoricamente ottimo (se utilizzo sotto 100% rispetta la deadline) però va considerate il context switch e il costo della gestione dell'interrupt

- Tracciare il Gantt secondo EDF verificando se le scadenze sono rispettate (ammissibile)

Processo	Tempo di arrivo	Esecuzione	Scadenza
$J_1$	0	3	16
$J_2$	2	1	7
$J_3$	0	6	8
$J_4$	8	2	11
$J_5$	13	3	18



### ❖ Scheduling a Quote Proporzionali

- Quote proporzionali di CPU preallocate per le applicazioni
- T quote di CPU devono essere preallocate tra tutti i processi

## ❖ Linux Scheduling

**Prima della versione kernel 2.5** Linux utilizzava una variazione dell'algoritmo di scheduling standard di UNIX che si basava su **Code multiple con feedback gestite con RR:**

- Prelazione sulle code con priorità
- Priorità assegnata sulla base dell'utilizzo di CPU (maggiore utilizzo, minore priorità)
- Periodicamente il kernel calcola la CPU utilizzata da un processo dall'ultimo controllo, valore inversamente proporzionale alla priorità.
- Metodo non flessibile e poco adatto a Symmetric Multiprocessing

**Con la versione 2.5** si propone uno **scheduler a tempo costante O(1) indipendente dal numero di task nel sistema:**

- Preemptive e priority based
- Due intervalli di priorità: **processi time-sharing (detti nice)** e **real-time**
- I task sono in esecuzione finché hanno il time slice (active)
- Quando scade non vanno in esecuzione finché tutti gli altri tasks non usano lo slice
- Tutti i task eseguibili sono tracciati con code per ogni CPU
- Buon funzionamento, ma tempi di risposta non soddisfacenti per processi in time-sharing

Successivamente **Completely Fair Scheduler (CFS):**

- lavora su **Diverse classi di scheduling ognuna con una priorità specifica**
- Scheduler prende la più alta nella classe più alta
- 2 classi di scheduling, altre possono essere aggiunte: default e real-time
- Il kernel non fornisce garanzia sui tempi di attesa dei processi pronti. Se interruzione per real-time arriva mentre il kernel serve una chiamata di sistema, il processo real-time attende.
- Scheduler CFS supporta bilanciamento del carico, è compatibile con architetture NUMA e riduce al minimo migrazione dei thread

## ❖ Valutazione algoritmi di Scheduling

Come selezionare algoritmi di CPU-scheduling per un SO? Si determina prima un criterio, poi si valuta gli algoritmi.

- **Valutazione analitica:** definisce una formula o dati da valutare
- **Modelli deterministici:** assume un particolare carico e definisce le performance di ogni algoritmo per quel carico (quello che facciamo noi!)
  - Per ogni algoritmo calcola il **minimo tempo di attesa medio:** semplice e veloce, ma richiede numeri esatti per l'input e si applica solo a questi inputs "campione".
- **Modelli di Code:** Descrive gli arrivi dei processi, CPU e I/O bursts probabilisticamente.
  - Di solito distribuzioni esponenziali e calcolo di medie
  - Calcola medie di throughput, utilizzo, waiting time, etc
- **Simulazioni**
- **Implementazioni con test in sistemi reali**

## 5. SINCRONIZZAZIONE

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè, codice e dati) oppure condividere dati soltanto attraverso file o messaggi. Nel primo caso si fa uso dei thread. **L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati.**

Un processo può avere completato solo in parte la sua esecuzione quando viene schedulato un altro processo. In effetti, un processo può essere interrotto in qualsiasi punto del proprio flusso d'esecuzione, assegnando il core di elaborazione all'esecuzione di istruzioni di un altro processo. **La consistenza richiede meccanismi per assicurare l'esecuzione ordinata di processi cooperanti per evitare situazioni di incoerenza dei dati condivisi.**

### ❖ Produttore-consumatore con buffer illimitato

Si può introdurre un intero **counter** che conta i buffer pieni.

- Inizialmente counter = 0.
- Poi incrementato dal produttore dopo che produce un elemento nel buffer e decrementato dal consumatore dopo che consuma un elemento dal buffer

#### Produttore

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

---

#### Consumatore

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

- Questa soluzione proposta con buffer illimitato non è sufficiente e non produce una sincronizzazione tra i processi. Essi possono disturbarsi in vari modi. Dal punto di vista del calcolatore le operazioni eseguite sulla variabile “counter” non sono atomiche. **Quando si interacciano le due esecuzioni la variabile in comune “counter” non è sincronizzata e separata tra i due processi e può trovarsi in uno stato inconsistente alla fine delle operazioni.**

- Operazioni non atomiche
- Se i due processi eseguono counter++ e counter-- insieme
- **counter++** può essere implementato come

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** può essere implementato come

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Interfogliate le due esecuzioni, inizialmente “count = 5”:

S0: producer	<b>register1 = counter</b>	{register1 = 5}
S1: producer	<b>register1 = register1 + 1</b>	{register1 = 6}
S2: consumer	<b>register2 = counter</b>	{register2 = 5}
S3: consumer	<b>register2 = register2 - 1</b>	{register2 = 4}
S4: producer	<b>counter = register1</b>	{counter = 6 }
S5: consumer	<b>counter = register2</b>	{counter = 4}

#### ❖ Concetto di corsa critica

- Nei sistemi concorrenti la race condition (o corsa critica) è una situazione in cui il risultato dell'esecuzione di un insieme di processi che condividono un'area di memoria, un file, una periferica etc. dipende dall'ordine in cui essi sono eseguiti.
  - Ciò avviene quando tali processi hanno accesso in scrittura alla risorsa condivisa e l'ordine di esecuzione dipende dalla loro temporizzazione, rendendo non predicibile il valore finale assegnato a tale risorsa.
- Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette race condition) **occorre assicurare che un solo processo alla volta possa modificare la variabile contatore.** Questa garanzia richiede una forma di **sincronizzazione dei processi.**

#### ❖ Problema della sezione critica. Requisiti.

- Si considera un sistema composto di n processi  $\{P_0, P_1, \dots, P_{n-1}\}$ . Ogni processo possiede un segmento di codice, chiamato sezione critica (detto anche regione critica), in cui può modificare variabili comuni (risorse condivise).
- Quando un processo entra in sezione critica gli altri processi non dovrebbero andare nella loro sezione critica, dovrebbe esserci un suo uso esclusivo.

Il problema della sezione critica richiede un protocollo di interazione per risolverlo.

- Innanzitutto si deve ben delineare quali sono le sezioni critiche.
- Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. Il codice che implementa la richiesta per entrare nella sezione critica è la **entry section**.
- Dopo la sezione critica il processo dovrà avvertire gli altri processi e ci sarà una **exit section**.
- Il codice rimanente del processo si dice **remainder section**.

Struttura generale di un processo con sezione critica  $P_i$

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

#### ❖ Requisiti

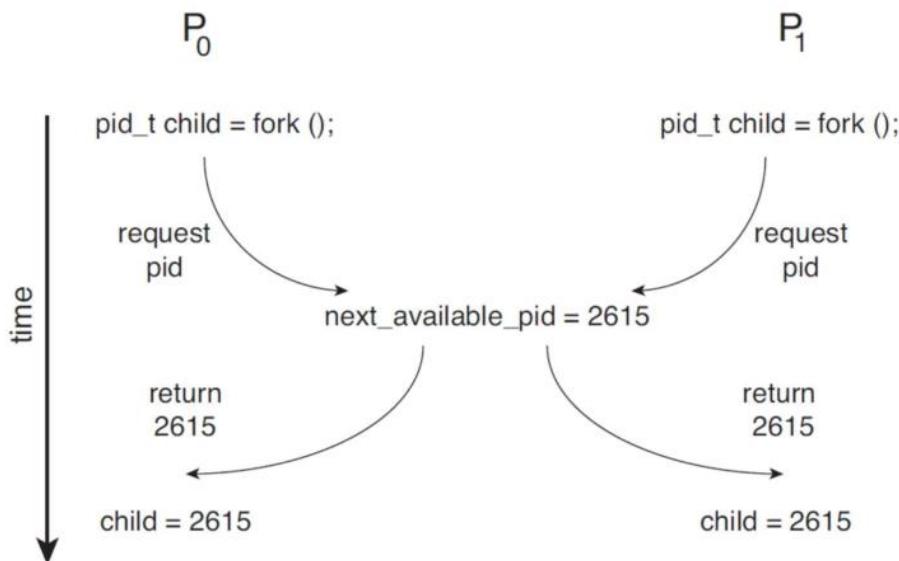
Una soluzione del problema della sezione critica deve soddisfare i 3 seguenti requisiti:

- **Mutua esclusione:** se il processo  $P_i$  è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
- **Progresso:** se nessun processo esegue la sua sezione critica e alcuni processi vogliono entrare nella loro, allora la decisione su chi sarà il prossimo a entrare nella propria sezione critica non può essere ritardata indefiniteamente. La decisione su chi può entrare in sezione critica spetta ai processi che sono **fuori dalle proprie remainder section**.
- **Bounded waiting:** esiste un limite al numero di volte in cui gli altri processi possono entrare nelle loro sezioni critiche dopo che un processo ha fatto richiesta di entrare nella propria e prima che tale richiesta sia esaudita. Altrimenti un processo non entrerebbe mai nella propria sezione critica.

## ❖ Corse critiche nel Kernel: Gestione della corsa critica.

Anche nel kernel si verificano situazioni di race condition.

- Esempio:
  - I processi P0 e P1 creano processi figlio utilizzando fork() nello stesso momento.
  - Avviene una corsa critica sulla variabile kernel che rappresenta il prossimo identificatore di processo disponibile (**pid**), **next\_available\_pid**. Questo perché questi numeri vengono "riciclati" dai processi che finiscono e il kernel mantiene in questo indice next\_available\_pid il prossimo numero libero da associare ad un processo figlio.
  - Senza mutua esclusione, lo stesso pid potrebbe essere assegnato a due processi diversi! **Situazione che viola la regola di unicità del PID!**
  - Occorre un meccanismo di sincronizzazione anche nella gestione di corse critiche nel kernel



- Esempio 2: **Corse critiche su tabella di file aperti nel sistema da parte di 2 processi.**

## ❖ Gestione della corsa critica (nel kernel)

Possiamo avere due approcci: **kernel preemptive** e **kernel non-preemptive**.

- **Preemptive:** permette la prelazione del processo quando è in kernel mode.
- **Non-preemptive:** il processo continua l'esecuzione finché è in kernel mode, si blocca, o volontariamente rilascia la CPU. In questo caso non si consente l'interruzione forzata di processi attivi in kernel mode: dunque **si disabilita la possibilità di race condition** dato che un processo viene eseguito in modo esclusivo senza possibilità di interruzioni da altri processi.

## ❖ Soluzione di Peterson

- E' una classica soluzione software al problema della sezione critica. Buona descrizione algoritmica del problema.
- Fornisce una soluzione al problema della sezione critica **per 2 processi**.

**Si assume che load e store siano istruzioni atomiche in linguaggio macchina (non interrompibili).**

Due processi, ognuno dei quali ha un propria sezione critica, condividono due variabili per sincronizzarsi:

**int turn;**  
**Boolean flag[2]**

- La variable **turn** indica il processo di turno per entrare nella sezione critica
- Il **flag array** è usato per indicare se un processo è pronto per entrare in critical section.
  - **flag[i] = true** significa che il processo **Pi è pronto**.
- **è utile convenire che se Pi denota uno dei due processi, Pj denoti l'altro; ossia, che j = 1 - i.**

## ❖ Algoritmo per il processo Pi

```
do {  
    flag[i] = true;           ENTRY  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;          EXIT  
        remainder section  
} while (true);
```

P<sub>0</sub>: flag[0]=true  
P<sub>1</sub>: flag[1]=true  
P<sub>1</sub>: turn=0  
P<sub>1</sub>: while(flag[0] && turn=0);  
P<sub>0</sub>: turn=1  
P<sub>0</sub>: while(flag[1] && turn=1);  
...

Nella prima parte siamo nella entry section.

- Il processo Pi imposta l'array flag[i] a true, indicando che è pronto per entrare in sezione critica.
- A questo punto però imposta il turno all'altro processo, di indice j.
- Poi il processo Pi si mette in attesa in un ciclo while "a vuoto" il quale cicla finché:
  - il flag[j] è impostato a true cioè che l'altro processo è pronto
  - e finché è il suo turno ossia turn == j.
- Dunque il processo Pi aspetta finchè l'altro processo Pj non finisce la sua esecuzione in sezione critica.
- Supponiamo che anche l'altro processo abbia messo flag[j] a falso:
  - allora il ciclo while per il processo Pi si sblocca, entra in sezione critica e poi mette il suo flag[i] a false nella exit section e da il permesso a questo punto all'altro processo di entrare e si sono così sincronizzati.

### Requisiti rispettati:

- **Mutua esclusione:** garantita dal valore di turn;
- **Progresso:** Pi entra nella sezione critica al massimo dopo un ingresso da parte di Pj
- **Attesa limitata:** per la turnazione

### In sistemi multithreaded ci sono problemi:

- può cambiare l'ordine delle assegnazioni perché le variabili non hanno un vincolo di dipendenza, non c'è un'effettiva sequenzialità rispettata
- si potrebbe avere accesso concorrente alla sezione, dato che le operazioni non sono atomiche
- Lo scambio delle istruzioni nel modo "**turn = j**" e "**flag[i] = true**" consentirebbe ad entrambi i processi di entrare in sezione critica.

## ❖ Hardware per sincronizzazione

Molti sistemi forniscono **supporto hardware** per implementare il codice della sezione critica. Tutte queste soluzioni si basano sul concetto di **lock (lucchetto)**, ovvero sulla protezione di regioni critiche attraverso l'uso di lock.

- In un sistema dotato di una singola CPU (**uniprocessore**) il problema della sezione critica si potrebbe risolvere semplicemente **disabilitando le interruzioni mentre si modificano le variabili condivise**. In questo modo si assicurerebbe un'esecuzione nell'ordine e senza possibilità di prelazione della corrente sequenza di istruzioni; non si potrebbe eseguire nessun'altra istruzione, quindi non si potrebbe apportare alcuna modifica inaspettata alle variabili condivise. È questo l'approccio seguito dai **kernel non-preemptive**. **Questa soluzione è troppo inefficiente su sistemi multicore**.
- Macchine moderne forniscono **soluzioni hardware, usate direttamente o utilizzate per costruire meccanismi di sincronizzazione**
  - **Barriere di memoria**
  - **Istruzioni hardware**
  - **Varibili atomiche**

## ❖ Soluzioni basate su Locks

Si basano sul seguente schema:

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Si acquisisce il lock esclusivo e si blocca l'accesso a tutti gli altri processi alla sezione critica; quando si ha finito si rilascia il lock.

## ❖ Barriere di memoria

Stabiliscono dei vincoli sulla memoria secondo due modelli:

- **memorie fortemente ordinate**: nelle architetture multiprocessore una modifica di un processore è immediatamente visibile per gli altri processori
- **memorie debolmente ordinate**: la modifica non viene vista da tutti gli altri processori
- **Si possono forzare cambiamenti in memoria che vengono propagati ad altri processori**

Esempio

```
▶ Thread 1   while (!flag)  
              memory_barrier();  
              print x;  
  
▶ Thread 2   x = 100;  
              memory_barrier();  
              flag = true;
```

## ❖ Istruzioni atomiche

Molte delle moderne architetture offrono **particolari istruzioni** che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, **in modo atomico** – cioè come **un'unità non interrompibile eseguibili sequenzialmente**.

Consideriamo 2 **tipi** di istruzioni:

- **Test memory (word) and set (value) (test and set)**
- **Swap contents of two memory words (compare and swap)**

## ❖ test\_and\_set

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

E' assicurato che a livello hardware le istruzioni siano eseguite atomicamente.

Prende il valore booleano target e lo mette nella variabile rv, setta quindi a TRUE questo target e poi restituisce il vecchio valore rv utilizzato per fare il test.

- Se si eseguono contemporaneamente due istruzioni test\_and\_set(), ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario.
- Se si dispone dell'istruzione test\_and\_set(), si può realizzare la **mutua esclusione** dichiarando una **variabile booleana globale lock, inizializzata a false**.

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
    lock = false;
    /* remainder section */
} while (true);
```

Funzionamento:

- Un processo Pi testa la variabile booleana lock con il test\_and\_set e se testandola vale false, allora la setta inizialmente a true per impedire l'accesso ad altri processi e a accede alla sezione critica. Questa operazione viene fatta in modo atomico, garantita dalla funzione test and set. Il processo Pi va in mutua esclusione.
- Un processo Pj che vuole accedere un momento dopo troverebbe la variabile lock uguale a true e quindi ciclerebbe in attesa che diventi false (**BUSY WAITING**).
- Appena il processo Pi termina la sezione critica imposta la variabile lock a false.

- A questo punto un processo Pj può usare il test\_and\_set per verificare che sia false e impostarla a true, per accedere anch'esso alla sezione critica
- E così via..

### ❖ compare\_and\_swap

L'istruzione compare\_and\_swap(), a differenza dell'istruzione test\_and\_set() utilizza tre operandi e un condizionale. Viene sempre eseguita in modo atomico.

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

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Restituisce il valore originale del parametro in “value”. Setta la variabile “value” al valore di “new\_value” ma solo se “value”==“expected”. Cioè lo swap avviene solo con questa condizione.

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

Inizialmente abbiamo un intero condiviso “**lock**” **inizializzato a 0**.

- Se un processo trova un valore di lock == 0 allora si entra nel ciclo perché è uguale all'expected e la funzione lo aggiorna col new\_value ossia 1, ritornando il valore 0 e rendendo l'accesso alla sezione critica disponibile per il processo corrente ma bloccata per altri processi. Al termine dell'esecuzione il processo reimposta il valore di lock = 0.
- Se un processo trova un valore di lock == 1 la funzione ritorna direttamente temp ossia 1 perché il condizionale fallisce (1 non è uguale all'expected value ossia 0) e l'accesso alla sezione critica è bloccato in quanto la condizione del ciclo while è rispettata ( $1 \neq 0$ ) e quindi cicla rimanendo in busy waiting fin quando il valore del lock non diventa == 0.
- 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.

- **Non soddisfa attesa limitata (bounded-waiting):** l'attesa per l'accesso alla sezione critica deve essere stabilita. Per garantire mutua esclusione e attesa limitata si introducono meccanismi di turnazione

#### Bounded-waiting Mutual Exclusion con Compare and Swap

```

while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

    /* critical section */

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

    if (j == i)
        lock = 0;
    else
        waiting[j] = false;

    /* remainder section */
}

```

#### Bounded-waiting Mutual Exclusion con test\_and\_set

```

do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);

```

## ❖ Mutex

Le soluzioni hardware al problema della sezione critica sono complicate e generalmente inaccessibili ai programmati. I progettisti di sistemi operativi implementano **strumenti software** per risolvere lo stesso problema. Il più semplice di questi strumenti è il **lock mutex** (mutex = mutual exclusion). **Usiamo il lock mutex per proteggere le regioni critiche e quindi prevenire le race condition.**

- 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. Sono atomiche implementate con istruzioni hardware atomiche.
- Implementa una **variabile booleana available** condivisa 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.
- Le chiamate alle funzioni acquire() e release() devono essere eseguite **atomicamente** tramite **istruzioni hardware bloccanti**.
- **Svantaggio:** richiede **busy waiting** perché 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. L'attesa spreca cicli di CPU che qualche altro processo potrebbe utilizzare in modo produttivo.
- **Vantaggio:** non fa context-switch quando un processo deve attendere un lock, quindi per brevi attese è ok. Inoltre su architetture multicore l'esecuzione durante uno spinlock continua.

```
□ acquire () {
    while (!available)
        ; /* busy wait */
    available = false;
}
□ release () {
    available = true;
}
□ do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```

## ❖ Semafori

Strumenti di sincronizzazione più sofisticati (dei mutex locks) per sincronizzare processi, introdotti da Dijkstra.

- Un semaforo **S** è una **variabile intera** che può essere modificata con **2 operazioni indivisibili (atomiche)**:
  - **wait()** che effettua un test ed eventualmente decrementa la variabile S
  - **signal()** che incrementa la variabile S

### □ Definizione di **wait()**

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

### □ Definizione di **signal()**

```
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.

- Il semaforo può gestire in mutua esclusione N processi grazie alla variabile intera **S**.
- Se volessimo trasformarlo in un “mutex” poniamo **S = 1**.
- Se poniamo **S = 0** il semaforo rimane bloccato: il primo processo va in blocco e in questo caso si crea una sorta di sincronizzazione in attesa che un altro processo chiami un **signal()**.

## ❖ Uso dei semafori

Si usa distinguere tra:

- **semafori contatore**, il cui valore è un numero intero che varia su un dominio non ristretto;
  - in questo caso la variabile intera S è di controllo per l'accesso a un numero di risorse pari al valore inizializzato. Utile per esempio nel problema del produttore-consumatore col buffer.
- **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.
- Un semaforo può risolvere diversi **problemi di sincronizzazione**, ad esempio quando **S = 0** e si richiedono vincoli temporali tra processi o anche per i **thread join**.

### □ Esempio – vincoli di scheduling

- Considera **P<sub>1</sub>** e **P<sub>2</sub>** che richiedono **S<sub>1</sub>** prima di **S<sub>2</sub>**

Crea un semaforo “**synch**” inizializzato a 0

**P1 :**

<b>S<sub>1</sub>;</b>	Thread join	Thread exit
<b>signal(synch);</b>	Synch = 0	

**P2 :**

<b>wait(synch);</b>	wait(S)	signal(S)
<b>S<sub>2</sub>;</b>		

Poiché **synch** è inizializzato a 0, P2 esegue S2 solo dopo che P1 ha eseguito signal(synch), che si trova dopo S1.

## ❖ Sincronizzazione problema del Produttore-Consumatore con l'uso dei Semafori (buffer limitato)

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Due semafori per vincoli di scheduling  
tra produttore e consumatore + un lock

```
while (true) {
    . . .
    /* produce an item in next_produced */
    . . .
    wait(empty);
    wait(mutex);
    . . .
    /* add next_produced to the buffer */
    . . .
    signal(mutex);
    signal(full);
}

while (true) {
    wait(full);
    wait(mutex);
    . . .
    /* remove an item from buffer to next_consumed */
    . . .
    signal(mutex);
    signal(empty);
    . . .
    /* consume the item in next_consumed */
    . . .
}
```

- $n$  è la dimensione del buffer.
- semaforo mutex: gestisce la mutua esclusione
- semaforo empty: serve per verificare quando effettivamente il buffer è vuoto
- semaforo full: serve per coordinarsi nel caso di buffer pieno
- **3 semafori: 2 semafori per vincoli di scheduling tra produttore e consumatore + 1 lock**

Si assume che i processi produttore e consumatore ciclano e producono/consumano continuamente.

- Il processo **produttore** inizialmente entra nel semaforo empty in wait() che effettua un decremento del numero di locazioni del buffer ( $n$ ) perché producendo un dato si ridurranno le locazioni libere ed in qualche modo è come se si preparasse il materiale per produrre una risorsa (o item). Accede al buffer in mutua esclusione tramite il semaforo mutex.
- A questo punto il produttore produce l'item e lo aggiunge al buffer. Rilascia il mutex tramite il signal(mutex) e tramite il signal(full) da il via al consumatore che sta aspettando di consumare qualcosa.
- Il **consumatore** inizialmente è in wait(full) che inizialmente troverà a 0 perché il produttore non ha ancora prodotto alcun item e quindi va in busy waiting. Appena il produttore produce un item e invia il signal(full) allora il consumatore si sblocca dal busy waiting del semaforo full.
- A questo punto rimuove un item dal buffer e consuma

## ❖ Implementazione semafori evitando (**busy waiting**)

Le implementazioni viste precedentemente prevedono **busy waiting** con spinning sulla variabile S:

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

Bisogna garantire che 2 processi non eseguano wait() e signal() sullo stesso semaforo nello stesso tempo. Si può implementare un **semaforo senza busy waiting** in questo modo:

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

Si associa al semaforo una struttura dati che gestisce una **coda di attesa su quel semaforo**. Ogni entry in coda di attesa ha due dati:

- valore (di tipo intero)
- puntatore al prossimo record in lista

Possiamo invocare 2 operazioni:

- **block** – mette il processo che chiede l'operazione nella appropriata coda di attesa
- **wakeup** – toglie uno dei processi in coda di attesa lo mette nella coda ready

Le implementazioni di wait() e signal() diventano:

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

## Considerazioni

In questo caso il valore “S→value” diverge dalla definizione classica perché può essere negativo.

- Invertito ordine di decremento e attesa (definizione classica è inverso)
- Se negativo indica il numero di processi in attesa da risvegliare
- **La lista dei processi in attesa si può ottenere con puntatori ai Process Control Block (PCB)**

#### Problematica sezione critica per il codice di wait e signal:

- **le operazioni sui semafori devono essere eseguite in modo atomico.** Si deve garantire che nessuna coppia di processi possa eseguire operazioni wait() e signal() contemporaneamente sullo stesso semaforo.
- Si tratta di un problema di accesso alla sezione critica, e **in un contesto monoprocessoressi lo si può risolvere semplicemente inibendo le interruzioni durante l'esecuzione di signal() e wait().**
- Nei **sistemi multiprocessore** sarebbe necessario disabilitare le interruzioni di tutti i processori, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Tuttavia, **disabilitare le interruzioni di tutti i processori può essere complesso, e causare un notevole calo delle prestazioni.** È per questo che per garantire l'esecuzione atomica di wait() e signal() devono mettere a disposizione **altre tecniche di realizzazione dei lock** (per esempio, **compare\_and\_swap()** e gli **spinlock**).
- Viene **spostato però il busy waiting** entry section alla critical section, dove il codice è breve quindi ci sarà poco busy waiting se la sezione critica è raramente occupata.

#### ❖ Deadlock e starvation. Problematiche nell'uso dei semafori a livello di programmazione.

- **Deadlock:** due o più processi sono in attesa per un evento che può essere causato solo da uno dei processi in attesa
- un evento per cui si genera deadlock può essere l'attesa indefinita dell'esecuzione di un'operazione signal(). In questo caso un processo aspetta l'altro:

$P_0$	$P_1$
wait(S) ;	wait(Q) ;
wait(Q) ;	wait(S) ;
...	...
signal(S) ;	signal(Q) ;
signal(Q) ;	signal(S) ;

- **Starvation:** un processo potrebbe non essere più rimosso dalla coda del semaforo su cui è sospeso
- **Priority Inversion:** problema di scheduling quando un processo a bassa priorità tiene un lock necessario ad un processo a più alta priorità. Risolto con un protocollo di priority-inheritance (eredità di priorità)

**Problematiche nell'uso dei semafori** possono derivare da un uso scorretto delle operazioni dei semafori da parte dei programmati

- **Inversione di signal e wait**, che porta alla violazione della mutua esclusione
- **Scambio di signal con wait**, che porta ad un blocco permanente
- **Omissione di wait (mutex) o signal (mutex) (o entrambi)**

Si introducono costrutti di sincronizzazione di alto livello per affrontare questi problemi

#### ❖ Monitor

**Un monitor è un'astrazione di alto livello che fornisce un meccanismo di sincronizzazione per i processi.** E' pensato per essere un **tipo di dato astratto (ADT)** che incapsula i dati mettendo a disposizione un insieme di funzioni per operare su di essi: in particolare fornisce strumenti per la gestione di variabili condivise con procedure predefinite.

- Il monitor è un ADT che comprende un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. **Si dice che un monitor è mutuamente esclusivo.**
- Introduce le **variabili di condizione**, associate a meccanismi di sincronizzazione e che consentono di sincronizzare i processi
- Il costrutto monitor assicura che **all'interno di un monitor possa essere attivo un solo processo alla volta**, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione. **Tutte le procedure del monitor vengono eseguite in mutua esclusione.**
- **Incapsula i dati condivisi da proteggere**

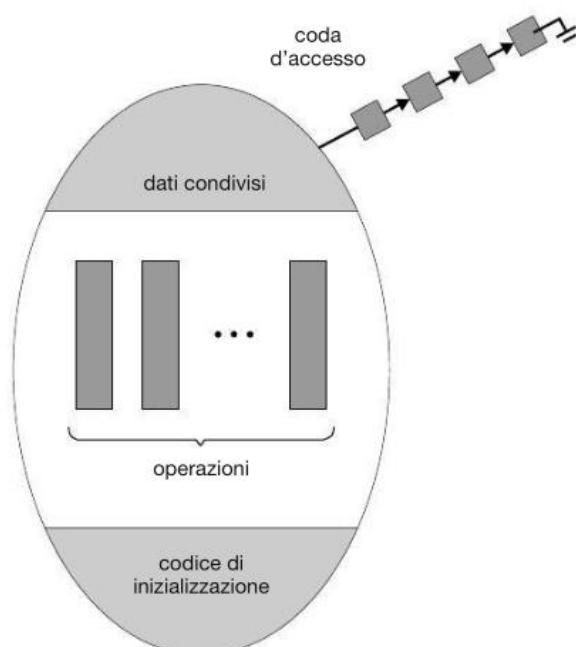
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
```

---

#### Schema di un Monitor:



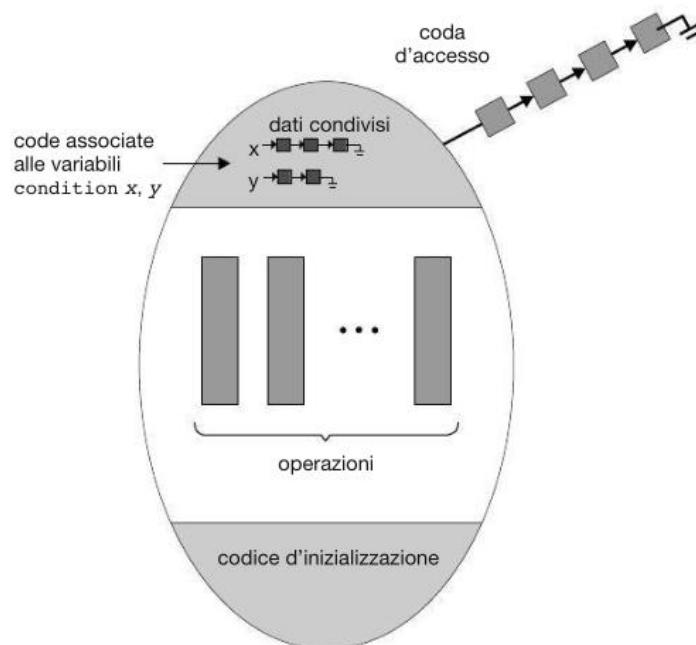
## ❖ Variabili di condizione

Un programmatore che necessita di implementare un proprio particolare schema di sincronizzazione può definire uno più **variabili di tipo condition**: sono variabili su cui si può sospendere un processo finché su quella variabile non accade un evento che può essere prodotto da un altro processo.

Le uniche operazioni eseguibili su una variabile condition sono **wait()** e **signal()**.

**condition x, y;**

- L'operazione **x.wait();** implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l'operazione **x.signal();** che risveglia esattamente un processo sospeso che ha invocato la **wait()**.
- Se non c'è **x.wait()** sulla variabile, non c'è alcun effetto della **signal()**, a differenza del semaforo contatore in cui si incrementava la variabile S se c'era un **signal()**.
- Le variabili di condizione diventano un elemento di sincronizzazione su cui vanno a **mettersi in coda** i processi che stanno **aspettando determinati eventi** generati da altri processi per mezzo di **signal** su quella determinata variabile di condizione per cui è in coda il processo in attesa.



## ❖ Implementazione monitor: Monitor implementati con Semafori

### □ Variabili

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

### □ Ogni procedura **F** sostituita da Signal and Wait

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

### □ La mutua esclusione nel monitor è assicurata

- Semaforo mutex per dare il permesso ad un processo alla volta di occupare il monitor
- Semaforo next che consente di dare il permesso di esecuzione agli altri processi
- Contatore che tiene conto di quanti processi sono in attesa per essere risvegliati su quella variabile
- Il signal su next viene effettuato solo se ci sono altri processi in attesa, altrimenti se non c'è nessuno si risveglia un processo con signal(mutex)

### □ Per ogni condition var **x**, abbiamo:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

### □ La **x.wait** implementata come: La **x.signal** implementata come:

```
x_count++;
if (next_count >
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
}
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next);
next_count--;
```

- Semaforo inizializzato a 0 perché il processo deve essere bloccante

# Scelte su Condition Variables

---

- Se il processo P invoca **x.signal()** e il processo Q è sospeso in **x.wait()** che succede?
  - Q e P non possono eseguire in parallelo nel monitor.
    - ▶ Se Q viene ripreso, P deve attendere
- Tra le opzioni abbiamo:
  - **Signal and wait** – P aspetta finché Q o lascia il monitor o aspetta per un'altra condizione
  - **Signal and continue** – Q aspetta finché P lascia il monitor o aspetta per un'altra condizione
  - Entrambe hanno pros e cons – implementatore del linguaggio decide
    - ▶ La seconda lascia al processo in esecuzione, ma la variabile potrebbe cambiare
  - Monitor implementato in Concurrent Pascal fa compromesso
    - ▶ P esegue signal e subito lascia il monitor, quindi Q è ripreso
    - ▶ Implementato in altri linguaggi (Mesa, C#, Java)

## Ripresa dei Processi nel Monitor

---

- Se molti processi accodati sulla condizione x, e avviene un **x.signal()** quali riprendere?
- FCFS spesso non adeguato
- **conditional-wait** come **x.wait(c)**
  - Dove c è il **priority number**
  - Processo con numero più basso (highest priority) è il prossimo schedulato

# Single Resource allocation

---

- Allocata una singola risorsa tra processi in competizione usando i numeri di priorità che specificano il tempo massimo di utilizzo della risorsa

```
R.acquire(t);  
...  
access the resource;  
...  
  
R.release;
```

- Con R istanza di tipo **ResourceAllocator**

```
monitor ResourceAllocator  
{  
    boolean busy;  
    condition x;  
    void acquire(int time) {  
        if (busy)  
            x.wait(time);  
        busy = TRUE;  
    }  
    void release() {  
        busy = FALSE;  
        x.signal();  
    }  
    initialization code() {  
        busy = FALSE;  
    }  
}
```

- Insieme di proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante il ciclo di vita della loro esecuzione.
- Un processo che attende indefinitamente è un esempio di “mancanza di liveness” (**liveness failure**).
- Esempi di situazioni che possono portare a liveness failure:
  - Ciclo infinito
  - **Attesa indefinita** (starvation): Un processo potrebbe non essere più rimosso dalla coda del semaforo su cui è sospeso
  - **Deadlock**: due o più processi sono in attesa per un evento che può essere causato solo da uno dei processi in attesa
  - **Inversione di priorità**: problema di scheduling quando un processo a bassa priorità tiene un lock necessario ad un processo a più alta priorità

### ❖ Inversione di priorità

- Si verifica ogniqualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa.
- Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse.

La situazione si complica ulteriormente se il processo a priorità più bassa viene **prelazionato** da un processo a priorità più alta.

3 processi Low, Medium, high priority

#### □ Tre processi L < M < H

- H chiede il semaforo S tenuto da L
- H deve aspettare L
- ... M diventa runnable e prelaziona L
- Il processo M influenza l'attesa di un processo H (a più alta priorità) su S tenuta da L

Soluzione: **Priority-inheritance protocol**

- **Quando processi accedono ad una risorsa richiesta da un processo ad alta priorità, ereditano la priorità del processo. Poi tornano ai valori originali.** L avrebbe ereditato la priorità di H bloccando M, rientrato in priorità L al rilascio, la competizione tra H ed M è vinta da H

L'inversione di priorità può avere effetti a catena causando un fallimento sistematico: **Il caso del Mars Pathfinder. Soluzione:** variabile globale per abilitare l'ereditarietà delle priorità su tutti i semafori.

## ❖ Problemi di Sincronizzazione ed Esempi di sincronizzazione

Problemi classici usati per testare schemi di sincronizzazione:

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

### ❖ Bounded-Buffer Problem (già spiegato in precedenza)

- **n** buffer, ognuno può contenere un articolo
- Semaforo **mutex** initializzato al valore 1
- Semaforo **full** initializzato al valore 0
- Semaforo **empty** initializzato al valore n
- La struttura del produttore

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

Se invertissimo `wait(mutex)` con `wait(empty)` abbiamo uno stallo perché abbiamo tolto la possibilità che accada un evento. Si aspetta prima l'evento e poi si blocca e si prende il controllo con `mutex`.

- La struttura del consumatore

```
Do {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next_consumed */  
    ...  
} while (true);
```

## ❖ Readers and Writers Problem

Il problema dei lettori scrittori è un noto problema di sincronizzazione. Un insieme di dati è condiviso tra n processi concorrenti. I processi possono essere di due tipologie:

- **Readers** – leggono solo i dati; non aggiornano
- **Writers** – leggono e scrivono

Una soluzione sarebbe mettere i processi readers e writers in mutua esclusione. Però nel momento in cui il dato è stabile perché dovremmo far accedere un lettore alla volta? **Problema** – permettere a lettori multipli di leggere allo stesso tempo. Il processo scrittore scrive invece in modo esclusivo.

Dati condivisi:

- Data set
- Semaforo binario **rw\_mutex** inizializzato a 1 per “proteggere” lo scrittore che può scrivere e leggere e il lettore
- Semaforo binario **mutex** inizializzato a 1 che protegge il contatore dei lettori “**read\_count**”
- Intero **read\_count** inizializzato a 0 che conta i lettori

### □ Struttura scrittore

```
do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);
```

### □ Struttura del lettore

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

Se uno scrittore è in sezione critica con n lettori in attesa, n-1 su mutex, 1 su rw\_mutex

Occorre notare che se uno scrittore si trova nella sezione critica e n lettori attendono di entrarvi, si accoda un lettore a rw\_mutex e n - 1 lettori a mutex. Il primo lettore è come se facesse da "pionere" e controlla se c'è lo scrittore. Non libera il mutex e blocca tutti gli altri lettori sul primo. Il primo lettore prende il lock di rw\_mutex e permette l'accesso agli altri lettori. Altri lettori evitano il wait su rw\_mutex e si alternano solo su mutex. L'ultimo lettore rilascia rw\_mutex quindi o scrittore o primo lettore potrà prenderlo.

## [\(5\) The Readers Writers Problem - YouTube](#)

**Read-write lock** è un tipo di lock che è stato introdotto per risolvere questo tipo di problema:

- **Lock in read mode o write mode**
- Più reader possono prendere in read mode
- Un solo writer in write mode

Utili:

- Quando è facile identificare quali processi leggono solo dati condivisi e quali processi scrivono solo dati condivisi.
- Quando si hanno più lettori che scrittori.

## ❖ Dining-Philosophers Problem (Problema della cena dei filosofi)

- **5 filosofi pensano e mangiano.**
- Condividono un **tavolo rotondo circondato da 5 sedie**, una per ciascun filosofo. Al centro del tavolo si trova **1 zuppiera colma di riso**, e la tavola è apparecchiata con **5 bacchette** (in inglese chopstick).
- I filosofi non interagiscono tra di loro.
- quando gli viene fame, un filosofo tenta di prendere le bacchette più vicine: quelle che si trovano alla sua destra e alla sua sinistra una alla volta.
- Due bacchette per mangiare, poi le rilasciano quando hanno finito.

Le risorse sono limitate perché abbiamo 5 bacchette e per mangiare ne servono 2, quindi per mangiare un filosofo deve sottrarre inevitabilmente una bacchetta al vicino.

### Dati condivisi

- **Ciotola di riso (data set)**
- **Semaforo chopstick [5]** inizializzato a 1



### Soluzione

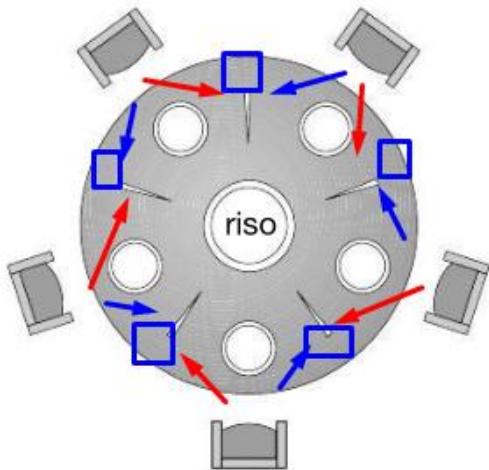
Abbiamo un semaforo “mutex” a guardia di ciascuna bacchetta: un vettore di 5 semafori, uno per bacchetta. **chopstick [5]** inizializzato a 1

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    . . .
    /* mangia */
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    /* pensa */
    . . .
} while (true);
```

### Problema

- Se tutti i filosofi prendono contemporaneamente la bacchetta sinistra deadlock

Se contemporaneamente tutti prendono la bacchetta alla loro destra o sinistra, tutti riusciranno a prendere una bacchetta ma nessuno riuscirà a prendere la seconda e ognuno aspetterà che l'altro la metta giù e visto che la procedura prevede che per mangiare servono 2 bacchette, in questa situazione si verifica uno **stallo (deadlock)**.



### Rimedi

- Permetti di sedere solo a 4 filosofi
- Permetti di prendere le bacchette solo se sono entrambe disponibili
- Soluzione asimmetrica (numero di dispari di filosofi)

## ❖ Soluzione Filosofi a cena con monitor

Un filosofo alla volta mangia (quando ha fame) prendendo entrambe le bacchette. Se ha fame controlla prima se ci sono entrambe le bacchette.

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }
    Initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura dati:

- `enum {THINKING, HUNGRY, EATING} state[5]`

Il filosofo  $i$  può impostare la variabile `state[i] = EATING` solo se i suoi due vicini non stanno mangiando. Praticamente lo stato hungry serve da sincronizzazione per effettuare il test se le bacchette sono libere

```
((state[(i + 4) % 5] != EATING) && (state[(i + 1) % 5] != EATING)).
```

Inoltre, occorre dichiarare la seguente struttura dati: `condition self[5];` che permette al filosofo  $i$  di ritardare se stesso quando ha fame, ma non riesce a ottenere le bacchette di cui ha bisogno.

Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione `pickup()`; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione `putdown()` e comincia a pensare. Il filosofo  $i$  deve quindi chiamare le operazioni `pickup()` e `putdown()` nella seguente sequenza:

**DiningPhilosophers.pickup(i); ... eat ... DiningPhilosophers.putdown(i);**

**Problema starvation:** usiamo una coda in ordine di “fame”, oppure priorità

## ❖ Pthreads meccanismi di sincronizzazione

Pthreads API sono OS-independent. Forniscono: **mutex locks, condition variable**. Estensioni includono: read-write locks, spinlocks.

### Mutex Posix

Un mutex Posix è caratterizzato dalle seguenti proprietà:

- è una variabile di tipo `pthread_mutex_t` che può essere inizializzata con diversi attributi ([tipo](#), [scopo](#), etc.)
- può assumere solo i due stati alternativi [chiuso](#) (locked) o [aperto](#) (unlocked);
- può essere chiuso solo da [un](#) processo alla volta, ed il processo che chiude il mutex ne diviene il [possessore](#) fino alla successiva chiusura;
- può essere riaperto solo dal proprio [possessore](#);
- deve essere [condiviso](#) tra tutti i processi che intendono sincronizzare l'accesso ad una regione critica ([blocco cooperativo](#)).
- Un mutex è un semaforo binario (rosso o verde)
- Un mutex è mantenuto in una struttura

`pthread_mutex_t`

- Tale struttura va allocata e inizializzata
- Per inizializzare:

- se la struttura è allocata staticamente:

```
pthread_mutex_t c = PTHREAD_MUTEX_INITIALIZER
```

- se la struttura è allocata dinamicamente (e.g. se si usa malloc): chiamare `pthread_mutex_init`

### Inizializzare e distruggere un mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
int pthread_mutex_lock          (pthread_mutex_t *mutex);
int pthread_mutex_trylock      (pthread_mutex_t *mutex);
int pthread_mutex_unlock       (pthread_mutex_t *mutex);
```

- inizializza e distrugge un mutex, rispettivamente
- Quando inizializzato è in stato aperto
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

- acquisiscono e rilasciano il semaforo
- restituiscono 0 se OK, un codice d'errore altrimenti
- se il semaforo è occupato (locked)...
  - ...lock blocca il thread finché il semaforo si libera
  - ...trylock invece non blocca, ma restituisce subito l'errore EBUSY

### Esempio 1 codice Mutex

```
myfoo test; // Variabile GLOBALE
pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;

void *inc(void *arg){ // incrementa a e b
    pthread_mutex_lock(&sem);
    test.a++;
    test.b++;
    printf("tid=%d a=%d b=%d\n", pthread_self(), test.a, test.b);
    pthread_mutex_unlock(&sem);
    pthread_exit((void *)&test);
}

int main(void){
    char st[100];
    pthread_t tid;
    int i=0;
    myfoo *b;

    while (i++<10){
        pthread_create(&tid, NULL, inc, NULL); // Thread concorrenti
    }
}
```

Quanti thread? 10

Output:

```
tid=1077283760 a=1 b=1
tid=1096194992 a=2 b=2
tid=1079385008 a=3 b=3
tid=1081486256 a=4 b=4
tid=1083587504 a=5 b=5
tid=1085688752 a=6 b=6
tid=1087790000 a=7 b=7
tid=1094093744 a=8 b=8
tid=1091992496 a=9 b=9
tid=1089891248 a=10 b=10
```

## Esempio 2 codice Mutex (dinamico)

Alla variabile di memoria condivisa myfoo test che è una struct, si aggiunge direttamente nella struct un semaforo. Successivamente inizializzazione dinamica del semaforo.

```
typedef struct foo{
    int a;
    int b;
    pthread_mutex_t sem;
} myfoo;

myfoo *test; // Variabile GLOBALE

myfoo *init_struct(){
    struct foo *fp;

    if ((fp=malloc(sizeof(myfoo)))==NULL)
        return(NULL);
    fp->a=0;
    fp->b=0;
    pthread_mutex_init(&fp->sem,NULL);
    return(fp);
}

void stampa(struct foo *test){
    printf("tid=%d a=%d b=%d\n", pthread_self(),test->a, test->b);
    fflush(stdout);
}

void *inc(void *arg){ // incrementa a e b
    pthread_mutex_lock(&test->sem);
    test->a=test->a+2;
    test->b++; // Variabile GLOBALE
    stampa(test);
    pthread_mutex_unlock(&test->sem);
    pthread_exit((void *)&test);
}

int main(void){
    char st[100];
    pthread_t tid;
    int i=0;
    myfoo *b;

    test=init_struct();
    while (i++<10){
        pthread_create(&tid, NULL, inc, NULL); // Globale
    }
    sleep(1);
    printf("Master:");
    stampa(test);
    pthread_mutex_destroy(&test->sem);
}

Quanti thread? 10
Il campo a viene incrementato di 2, il campo b di 1.
```

```
tid=1077283760 a=2 b=1
tid=1079385008 a=4 b=2
tid=1081486256 a=6 b=3
tid=1083587504 a=8 b=4
tid=1085688752 a=10 b=5
tid=1087790000 a=12 b=6
tid=1089891248 a=14 b=7
tid=1091992496 a=16 b=8
tid=1094093744 a=18 b=9
tid=1096194992 a=20 b=10
Master:tid=1075181248 a=20 b=10
```

# Tipologie di Mutex

- fast: semantica classica, pertanto un thread che esegue due `mutex_lock()` consecutivi sullo stesso mutex causa uno stallo
- recursive: conta il numero di volte che un thread blocca il mutex e lo rilascia solo se esegue un pari numero di `mutex_unlock()`
- error-checking: controlla che il thread che rilascia il mutex sia lo stesso thread che lo possiede

## Limiti del MUTEX

- Se un thread attende il verificarsi di una condizione su una risorsa condivisa con altri thread

- Con i soli mutex sarebbe necessario un ciclo del tipo:

```
while(1) {  
    lock(mutex);  
    if (<condizione sulla risorsa condivisa>)  
        break;  
    unlock(mutex);  
    ...  
}  
<sezione critica>;  
unlock(mutex);
```

Attesa, e verifica ciclica sulla var, finchè la condizione verificata non rompe il ciclo, quindi sblocco.

Il processo rimane in busy waiting, spinning tra lock e unlock fin quando non si verifica la condizione.

- I mutex sono come strumento di cooperazione risultano:
  - inefficienti
  - ineleganti
- e per risolvere elegantemente problemi di cooperazione servono altri strumenti
- Le variabili condizione sono un'implementazione delle variabili condizione teorizzate da Hoare

Quando si mette in attesa su una variabile di condizione il processo si mette in attesa e la cpu non viene impegnata (si evita spinning e busy waiting).

# Variabili di Condizione

- strumenti di sincronizzazione tra thread che consentono di:
  - attendere passivamente il verificarsi di una condizione su una risorsa condivisa
  - segnalare il verificarsi di tale condizione
- la condizione interessa sempre e comunque una risorsa condivisa
- pertanto le variabili condizioni possono sempre associarsi al mutex della stessa per evitare corse critiche sul loro utilizzo

```
int pthread_cond_wait( pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

In Posix si associano variabili di condizione direttamente ad un mutex

## Usare una condition variable

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• thread che aspetta una condizione</li></ul> <p><b>mutex_lock(m)</b><br/>while (condizione falsa)<br/>    <b>cond_wait(c, m)</b><br/><i>fa' qualcosa</i><br/><b>mutex_unlock(m)</b></p> | <ul style="list-style-type: none"><li>• thread che rende la condizione vera</li></ul> <p><b>mutex_lock(m)</b><br/><i>rendi la condizione vera</i><br/><b>cond_broadcast(c)</b><br/><b>mutex_unlock(m)</b></p> |
|--|---|

Il thread che aspetta una condizione si mette temporaneamente in attesa su una variabile di condizione e poi rilascia il mutex. Il thread che rende la condizione vera invia un segnale in broadcast agli altri thread che stavano "dormendo". Una volta risvegliati i processi vanno in competizione per il lock.

### Differenza tra Monitor e Variabili di condizione + Mutex

- Nel monitor non c'è bisogno di associare una variabile di condizione ad un mutex perché la mutua esclusione è già assicurata dal monitor stesso a livello di struttura
- Invece in posix la chiamata `cond_wait` sulla variabile di condizione si deve associare ad un mutex (che è come se fosse un monitor implicito).

```
...  
/* Chiama do_work() mentre flag è settato, altrimenti si  
   blocca in attesa che venga segnalato un cambiamento nel  
   suo valore */  
  
void* thread_function (void* thread_arg) {  
    while (1) {  
        // Attende segnale sulla variabile condizione  
        pthread_mutex_lock(&thread_flag_mutex);  
        while (!thread_flag)  
            pthread_cond_wait(&thread_flag_cv, &thread_flag_mutex);  
        pthread_mutex_unlock(&thread_flag_mutex);  
        do_work ();           /* Fa qualcosa */  
    }  
    return NULL;  
}-
```

**int pthread\_cond\_signal(pthread\_cond\_t \*cond);**

**int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);**

- signal risveglia esattamente un thread in attesa su una condition variable
- broadcast risveglia tutti i thread in attesa su una condition variable
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

## Esempio 1 variabili di condizione posix

```
pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void *dec(void *arg){

    while(1){
        pthread_mutex_lock(&sem);
        while (test.a==0)
            pthread_cond_wait(&cond, &sem);

        printf("CONSUMATORE 1 a=%d \n", test.a);
        test.a--;
        pthread_mutex_unlock(&sem);
        nanosleep(&t2,NULL);
    }
}

void *dec2(void *arg){

    while(1){
        pthread_mutex_lock(&sem);
        while (test.a==0)
            pthread_cond_wait(&cond, &sem);

        printf("CONSUMATORE 2 a=%d, b=%d \n", test.a, test.b);
        test.a--;
        test.b--;
        pthread_mutex_unlock(&sem);
        nanosleep(&t2,NULL);
    }
}

void *inc(void *arg){ // incrementa a e b

    int j=0;
    while (j++<10){
        pthread_mutex_lock(&sem);
        test.a++;
        test.b++;
        printf("PRODUTTORE tid=%d a=%d b=%d\n", pthread_self(),test.a, test.b);

        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&sem);
        nanosleep(&t2,NULL);
    }
    pthread_exit((void *)&test);
}

int main(void){
    pthread_t tid;
    pthread_create(&tid, NULL, inc, NULL);
    pthread_create(&tid, NULL, dec, NULL);
    pthread_create(&tid, NULL, dec2,NULL);
    sleep(5);

```

PRODUTTORE tid=1077283760 a=1 b=1  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=2  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=3  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=4  
CONSUMATORE 2 a=1, b=4  
PRODUTTORE tid=1077283760 a=1 b=4  
CONSUMATORE 1 a=1  
PRODUTTORE tid=1077283760 a=1 b=5  
CONSUMATORE 1 a=1

# Semafori

---

- Appartengono all'estensione Posix SEM
  - #include <semaphore.h>
  - Named e unnamed
- 
- *Named*

```
#include <semaphore.h>
sem_t *sem;
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);

/* acquire the semaphore */
sem_wait(sem);
/* critical section */
/* release the semaphore */
sem_post(sem);
```

Named = semafori condivisi da tutti i thread che ne conoscono il nome

Unnamed = thread che appartengono allo stesso processo o altre restrizioni

- *Unnamed*

- A pointer to the semaphore
- A flag indicating the level of sharing
- The semaphore's initial value

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

flag 0, shared only by threads belonging to the process that created the semaphore.

nonzero value, shared between separate processes by placing it in a region of shared memory.

```
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

## Esempio codice mutex (uscito al compito di giugno)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5
6 #ifndef NUM_THREADS
7 #define NUM_THREADS 4
8#endif
9
10 int shared = 0;
11 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12
13 void *func(void* param)
14 {
15     pthread_mutex_lock(&mutex);
16     printf("Incrementing the shared variable...\n");
17     for (int i = 0; i < 10000; ++i) {
18         shared += 1;
19     }
20     pthread_mutex_unlock(&mutex);
21     return 0;
22 }
23
24 int main()
25 {
26     pthread_t threads[NUM_THREADS];
27
28     for (int i = 0; i < NUM_THREADS; ++i) {
29         pthread_create(&threads[i], NULL, func, NULL);
30     }
31
32     for (int i = 0; i < NUM_THREADS; ++i) {
33         pthread_join(threads[i], NULL);
34     }
35
36     printf("%d\n", shared);
37     exit(EXIT_SUCCESS);
38 }
```

Quanti thread? 4

Output: 40000

## Codice condvar1

```
15     pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;
16     pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
17
18     void *dec(void *arg){
19         while(1){
20             pthread_mutex_lock(&sem);
21             while (test.a==0)
22                 pthread_cond_wait(&cond, &sem);
23             printf("CONSUMATORE 1 a=%d \n", test.a);
24             test.a--;
25             pthread_mutex_unlock(&sem);
26         }
27     }
28
29
30     void *dec2(void *arg){
31         while(1){
32             pthread_mutex_lock(&sem);
33             while (test.a==0)
34                 pthread_cond_wait(&cond, &sem);
35             printf("CONSUMATORE 2 a=%d b=%d \n", test.a,test.b);
36             test.a--;
37             test.b--;
38             pthread_mutex_unlock(&sem);
39         }
40     }
41
42     void *inc(void *arg){ // incrementa a e b
43         int j=0;
44
45         while (j++<10){
46             pthread_mutex_lock(&sem);
47             test.a++;
48             test.b++;
49             printf("PRODUTTORE tid=%ld a=%d b=%d\n", pthread_self(),test.a, test.b);
50             pthread_cond_signal(&cond);
51             pthread_mutex_unlock(&sem);
52             sleep(0.2);
53         }
54
55         pthread_exit((void *)&test);
56     }
57
58     int main(void){
59         pthread_t tid;
60         pthread_create(&tid, NULL, inc, NULL);
61         pthread_create(&tid, NULL, dec, NULL);
62         pthread_create(&tid, NULL, dec2,NULL);
63         sleep(5);
64     }
}
```

```
PRODUTTORE tid=140554443155200 a=1 b=1
CONSUMATORE 1 a=1
PRODUTTORE tid=140554443155200 a=1 b=2
CONSUMATORE 1 a=1
PRODUTTORE tid=140554443155200 a=1 b=3
CONSUMATORE 2 a=1 b=3
PRODUTTORE tid=140554443155200 a=1 b=3
CONSUMATORE 1 a=1
PRODUTTORE tid=140554443155200 a=1 b=4
CONSUMATORE 2 a=1 b=4
PRODUTTORE tid=140554443155200 a=1 b=4
CONSUMATORE 1 a=1
PRODUTTORE tid=140554443155200 a=1 b=5
CONSUMATORE 2 a=1 b=5
PRODUTTORE tid=140554443155200 a=1 b=5
CONSUMATORE 1 a=1
PRODUTTORE tid=140554443155200 a=1 b=6
CONSUMATORE 2 a=1 b=6
PRODUTTORE tid=140554443155200 a=1 b=6
CONSUMATORE 1 a=1
```

## Codice condvar2

```
1  /* Due thread si alternano per incrementare i valori di count
2   * il primo incrementa i numeri tra count < COUNT_HALT1 || count > COUNT_HALT2 il secondo gli altri fino a COUNT_DONE */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <pthread.h>
7
8  pthread_mutex_t count_mutex      = PTHREAD_MUTEX_INITIALIZER;
9  pthread_cond_t condition_var    = PTHREAD_COND_INITIALIZER;
10
11
12 void *funCount1(void * );
13 void *funCount2(void * );
14 int count = 0;
15
16 #define COUNT_DONE     8
17 #define COUNT_HALT1   3
18 #define COUNT_HALT2   5
19
20
21 int main(void)
22 {
23     pthread_t thread1, thread2;
24     pthread_create( &thread1, NULL, &funCount1, NULL);
25     pthread_create( &thread2, NULL, &funCount2, NULL);
26
27     pthread_join( thread1, NULL);
28     pthread_join( thread2, NULL);
29
30     printf("Conto finale: %d\n",count);
31     exit(EXIT_SUCCESS);
32 }
33
34
35 // Scrive i numeri 1-3 e 7-8 lasciati a funCount1() da funCount2()
36 void *funCount1(void * arg)
37 {
38     for(;;)
39     {
40         // prende il lock del mutex
41         pthread_mutex_lock( &count_mutex );
42
43         // Aspetta che funCount2() sblocca count
44         pthread_cond_wait( &condition_var, &count_mutex );
45         count++;
46         printf("valore di count - funCount1: %d\n",count);
47         pthread_mutex_unlock( &count_mutex );
48         if(count >= COUNT_DONE) return(NULL);
49     }
50 }
51
52 //Scrive i numeri 3-7
53
54 void *funCount2(void * arg)
55 {
56     for(;;)
57     {
58         pthread_mutex_lock( &count_mutex );
59
60         if( count < COUNT_HALT1 || count > COUNT_HALT2)
61             // Questi numeri sono delegati a funCount1
62             // funCount1() viene ripristinata e puo' modificare "count".
63             pthread_cond_signal( &condition_var );
64         else
65         {
66             count++;
67             printf("valore di count - funCount2: %d\n",count);
68         }
69         pthread_mutex_unlock( &count_mutex );
70         if(count >= COUNT_DONE) return(NULL);
71     }
72 }
```

valore di count - funCount1: 1  
valore di count - funCount1: 2  
valore di count - funCount1: 3  
valore di count - funCount2: 4  
valore di count - funCount2: 5  
valore di count - funCount2: 6  
valore di count - funCount1: 7  
valore di count - funCount1: 8  
Conto finale: 8

## 6. DEADLOCK

- Il deadlock indica una situazione in cui due o più processi si bloccano a vicenda, aspettando che venga eseguito un certo evento (come rilasciare il controllo su una risorsa) che serve all'altro processo e viceversa.

### ❖ Esempio in Posix

2 thread in deadlock con mutex Posix. Supponiamo esecuzione in contemporanea dei 2 thread, si produce la situazione di stallo: ma non è detto che vada sempre in stallo, alcune volte sicuramente, l'ordine in cui i thread vengono eseguiti dipende da come vengono gestiti dallo scheduler della CPU.

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);

/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

La risorsa richiesta dal secondo è detenuta dal primo, la risorsa richiesta dal primo è detenuta dal secondo e si bloccano a vicenda.

### ❖ Livelock

Altra forma di “liveness” di processi.

- Rappresenta un gruppo di thread che non è bloccato ma non riesce a procedere. C'è un continuo tentativo di eseguire un'azione che fallisce ed impedisce di avanzare.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }
    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }
    pthread_exit(0);
}
```

Il **trylock** prova a prendere il mutex ma non riesce perché è bloccato e va in loop, ma non si blocca: come il problema dell'inversione di priorità del robot rover su marte.

## ❖ Caratterizzazione del Deadlock

Un deadlock avviene se e solo se le seguenti proprietà sono verificate contemporaneamente:

1. **Mutual exclusion:** almeno una risorsa nel sistema deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta. Se un altro processo richiede tale risorsa, si deve ritardare il processo richiedente fino al rilascio della risorsa.
2. **Hold and wait:** almeno un thread deve essere in possesso di almeno una risorsa e deve attendere di acquisire risorse già in possesso di altri processi.
3. **No preemption:** le risorse non possono essere prelazionate, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. **Circular wait:** deve esistere un insieme  $\{p_0, p_1, \dots, p_n\}$  di processi, tale che  $p_0$  attende una risorsa posseduta da  $p_1$ ,  $p_1$  attende una risorsa posseduta da  $p_2$ , ...,  $p_{n-1}$  attende una risorsa posseduta da  $p_n$  e  $p_n$  attende una risorsa posseduta da  $p_0$ .

Occorre sottolineare che tutte e quattro le condizioni devono essere vere, altrimenti non si può avere alcuno stallo. La condizione dell'attesa circolare implica la condizione di possesso e attesa, quindi le quattro condizioni non sono completamente indipendenti; tuttavia è utile considerare separatamente ciascuna condizione.

## ❖ Modello del sistema

I sistemi forniscono risorse: tipi di risorse  $R_1, R_2, \dots, R_m$  (Cicli CPU, spazio di memoria, dispositivi I/O).

- Ogni tipo di risorsa  $R_i$  ha un insieme  $W_i$  di istanze equivalenti
- Ogni processo utilizza una risorsa come segue:
  - **request:** richiesta per la risorsa
  - **use:** viene impegnata
  - **release:** risorsa rilasciata
- Ogni tipo di risorsa ha un numero massimo di istanze disponibili

## ❖ Grafo di allocazione delle risorse

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta grafo di allocazione delle risorse.

- Si tratta di un **insieme di vertici V** e un **insieme di archi E**, con l'insieme di vertici V composto da due sottoinsiemi:
  - $P = \{p_1, p_2, \dots, p_n\}$ , che rappresenta tutti i **processi del sistema**, e
  - $R = \{R_1, R_2, \dots, R_m\}$ , che rappresenta tutti i tipi di **risorsa del sistema**.
- Gli archi E sono diretti e sono di 2 tipologie:
  - **request edge:** arco diretto dal processo  $P_i$  al tipo di risorsa  $R_j$  si indica  $P_i \rightarrow R_j$ , e significa che il processo  $P_i$  ha richiesto un'istanza del tipo di risorsa  $R_j$ , e attualmente attende tale risorsa.
  - **assignment edge:** arco diretto dal tipo di risorsa  $R_j$  al processo  $P_i$  si indica  $R_j \rightarrow P_i$ , e significa che un'istanza del tipo di risorsa  $R_j$  è assegnata al processo  $P_i$ .

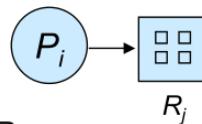
□ Processo



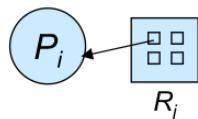
□ Tipo di risorsa con 4 istanze



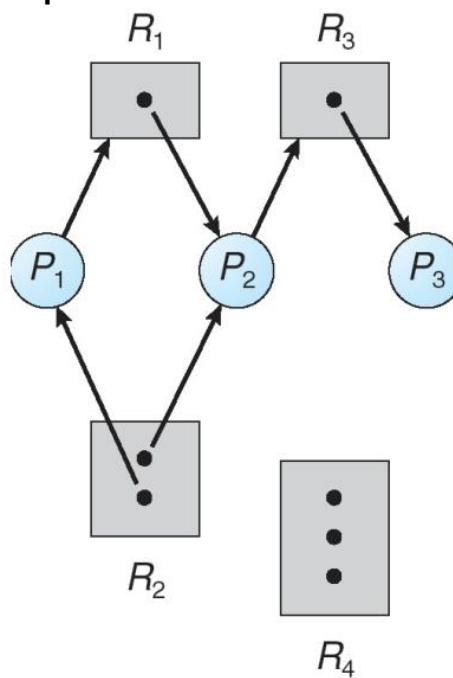
□  $P_i$  richiede istanza di  $R_j$



□  $P_i$  detiene un'istanza di  $R_j$

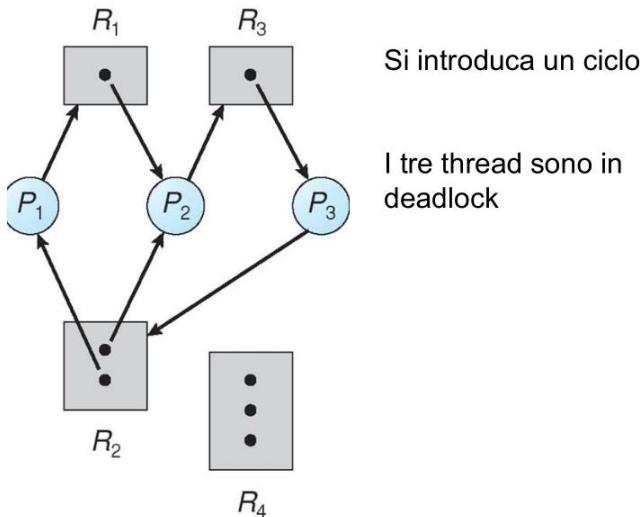


**Esempio 1**



- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$
- Se ci sono cicli vuol dire che c'è un'attesa mutua su una risorsa. In questo caso non ci sono dei cicli. **Senza cicli non c'è deadlock**, dato che sappiamo che il ciclo è una condizione necessaria (ma non sufficiente) per un deadlock. Se risorse sono uniche allora necessaria e sufficiente.

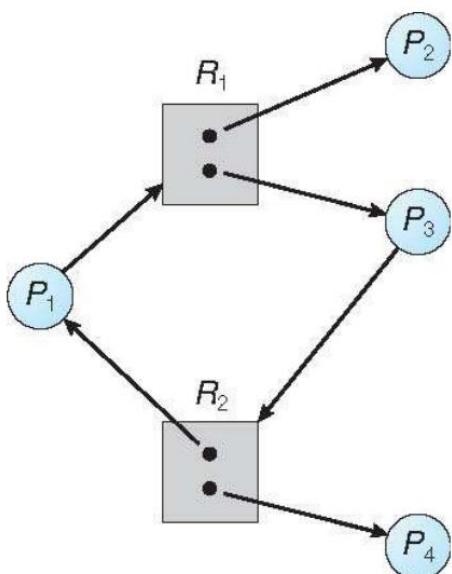
## Esempio 2



- In questo caso abbiamo 2 cicli. Questi 3 thread sono in deadlock.
- $P_2$  potrebbe liberare la risorsa  $R_2$  solo se  $P_3$  libera la sua risorsa  $R_3$  dato che la necessita,  $P_1$  potrebbe liberare la risorsa  $R_2$  ma è in attesa di  $P_2$  che a sua volta è in attesa di  $R_3$ . Quindi  $P_3$  non avrà mai la risorsa  $R_2$ .
- $P_3$  avrebbe dovuto aspettare che uno dei processi  $P_1$  o  $P_2$  liberasse la risorsa  $R_2$

## Esempio 3

Ciclo senza deadlock



- $P_2$  termina la computazione e libera  $R_1$
- Analogamente  $P_4$  può rilasciare  $R_2$  rompendo il ciclo per consentire a  $P_3$  l'esecuzione
- Dunque anche se c'è un ciclo, non è condizione sufficiente per avere un deadlock

## Riassumendo

- Se il grafo non ha cicli  $\rightarrow$  non ha deadlock
- Se il grafo contiene un ciclo
  - Se esiste una 1 istanza per tipo di risorsa, allora deadlock
  - Se Nistanze per tipo di risorsa, c'è possibilità di un deadlock, ma non si ha necessariamente il deadlock, bisogna vedere nel dettaglio. Se abbiamo esaurito tutte le risorse e non c'è nessun processo che può continuare è deadlock.

## ❖ Metodi di gestione del deadlock

Esistono diversi approcci al problema della gestione del deadlock:

- **Ignorare il problema** assumendo che i deadlocks non si presentino mai nel sistema; usato dalla maggior parte dei sistemi operativi, incluso **UNIX**. Questo perché le risorse si assume che siano talmente tante che sia difficile che si verifichi un deadlock.
- Assicurare che il sistema non entri **mai in un deadlock state**. Esistono **2 modalità**:
  - **Prevenzione di deadlock (deadlock prevention)**: limitare i modi in cui processi e thread fanno le richieste per evitare i deadlock
  - **Evitamento del deadlock (deadlock avoidance)**: il processo fa una richiesta e la si valutare per evitare situazioni pericolose; eventualmente si ritarda.
- Permettere al sistema di entrare in una stato di deadlock per poi recuperare (**deadlock recovery**)

## ❖ Prevention

Consiste nel limitare i modi in cui può essere fatta una richiesta da parte di un processo per una risorsa. Consente al sistema di non entrare mai in un deadlock state.

Si previene sulle caratteristiche del deadlock:

- **mutual exclusion**: si evita la richiesta per risorse non condivisibili, limitando le richiesta per le sole risorse condivisibili, che sono sempre disponibili. Problema: alcune risorse sono intrinsecamente non condivisibili.
- **Hold and Wait**: si deve garantire che quando un processo richiede una risorsa, non detiene altre risorse.
  - Si obbliga il processo a richiedere e allocare tutte le sue risorse prima che inizi l'esecuzione
  - Si consente di richiedere risorse ad un processo solo quando non ne è stata allocata alcuna
  - Non pratico: basso utilizzo delle risorse; possibile starvation, un processo potrebbe non ottenere mai le risorse.
- **no preemption**: se un processo che detiene risorse richiede un'altra risorsa che non può essere immediatamente data allora tutte le risorse devono essere rilasciate. Si implementa la prelazione e le risorse prelazionate sono aggiunte alla lista delle risorse per le quali il processo è in attesa. Il processo verrà riavviato solo quando può ottenere le vecchie e le nuove risorse.
- **circular wait**: si elimina imponendo un ordine totale a tutti i tipi di risorse e si obbliga ogni processo a richiedere le risorse in un ordine di enumerazione crescente. Date le risorse si assegna un numero di ordine F(R).

## ❖ Avoidance

Per evitare le situazioni di stallo consiste si richiedono a priori ulteriori informazioni sulle modalità di richiesta delle risorse.

- Il modello più semplice e più utile richiede che ciascun processo dichiari il **numero massimo delle risorse di ciascun tipo di cui necessita**. Praticamente si richiede a priori quali e quante risorse un processo impegna nella sua computazione.

Data questa informazione a priori, si può costruire un algoritmo capace di assicurare che il sistema non entri mai in stallo. Questo **algoritmo** per evitare lo stallo **esamina dinamicamente lo stato di allocazione delle risorse** per evitare che possa esistere una condizione di circular wait.

Lo **stato di allocazione delle risorse (resource-allocation state)** è dato dal numero di risorse disponibili e allocate, e dalle richieste massime per processo.

- L'avoidance cerca di mantere i processi in uno stato sicuro evitando uno stato insicuro.

## ❖ Stato insicuro

- Uno stato si dice insicuro se il sistema si trova in una situazione tale che possa verificarsi un deadlock e la situazione non si può gestire.

## ❖ Stato sicuro

- Uno stato si dice sicuro se il sistema è in grado di assegnare risorse a ciascun processo (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo. In questo stato non si può avere un deadlock.

- **Formalmente:** un sistema è in uno stato sicuro se esiste una sequenza  $\langle P_1, P_2, \dots, P_n \rangle$  di TUTTI i processi nel sistema tali che per ogni  $P_i$ , le risorse che  $P_i$  può ancora richiedere possono essere soddisfatte dalle risorse correntemente disponibili più le risorse tenute da tutti i processi  $P_j$ , con  $j < i$ 
  - **Significa** che abbiamo una sequenza di processi e il processo  $P_i$  può portare a termine la computazione tramite le risorse che sono ancora disponibili e quelle che potrebbero liberarsi dai processi  $P_j < P_i$  ossia precedenti ad esso in una sequenza ordinata di processi.

- Se le risorse richieste da  $P_i$  non sono immediatamente disponibili, allora  $P_i$  può aspettare finché tutti i  $P_j$  hanno finito
- Quando  $P_j$  ha finito,  $P_i$  può ottenere le risorse richieste, eseguirle, rilasciando le risorse allocate e terminare
- Quando  $P_i$  termina,  $P_{i+1}$  può ottenere le risorse necessarie, e così via ...

Praticamente si ordinano i processi in modo tale che tutti quelli che precedono il processo  $P_i$  ossia i processi  $P_j$  con  $j$  minore di  $i$ , possano terminare la computazione e rilasciare le risorse per consentire al processo  $P_i$  di andare avanti. Così per ogni processo della sequenza.

## Esempio Stato sicuro

- Esempio: 12 risorse e 3 thread

	Maximum Needs	Current Needs
$T_0$	10	5
$T_1$	4	2
$T_2$	9	2

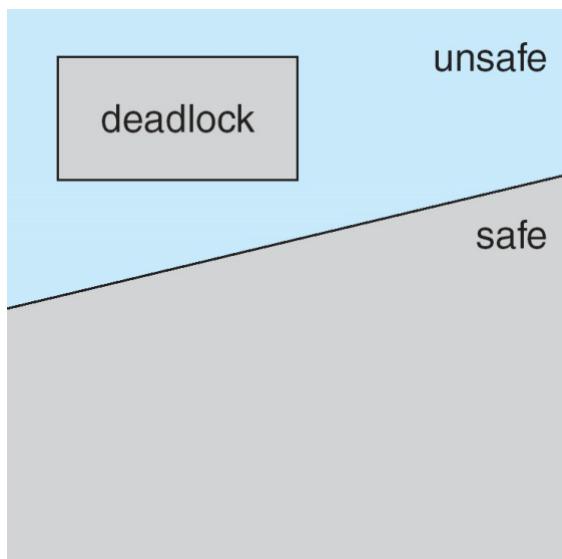
- Il Sistema è in stato sicuro
  - Allocate 9 ne rimangono 3
  - La sequenza  $\langle T_1, T_0, T_2 \rangle$  soddisfa il requisite
    - ▶  $T_1$  ne prende 2 e restituisce 4
    - ▶  $T_0$  ne prende 5 e restituisce 10
    - ▶  $T_2$  ne prende 7 e finisce
  - Se invece si alloca a  $T_2$  un'ulteriore risorsa al tempo  $t_1$  lo stato non è più safe

## Riassumendo

Se un sistema è in stato sicuro: non deadlock, c'è una controllabilità

Se un sistema è in stato non sicuro: possibilità di deadlock, non c'è una controllabilità

Avoidance: assicurare che un sistema non entri mai in uno stato non sicuro



## ❖ Algoritmi di Avoidance

Esistono 2 algoritmi per implementare l'Avoidance:

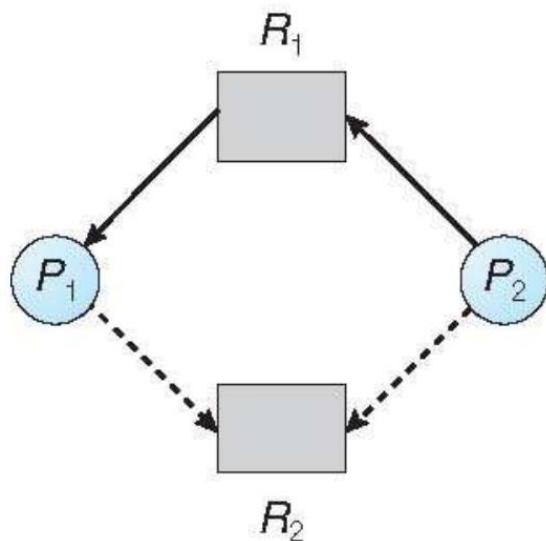
- **resource allocation graph:** considera una singola istanza per ciascun tipo di risorsa
- **algoritmo del banchiere:** considera istanze multiple per ciascun tipo di risorsa

## ❖ Algoritmo del grafo delle risorse (resource-allocation graph)

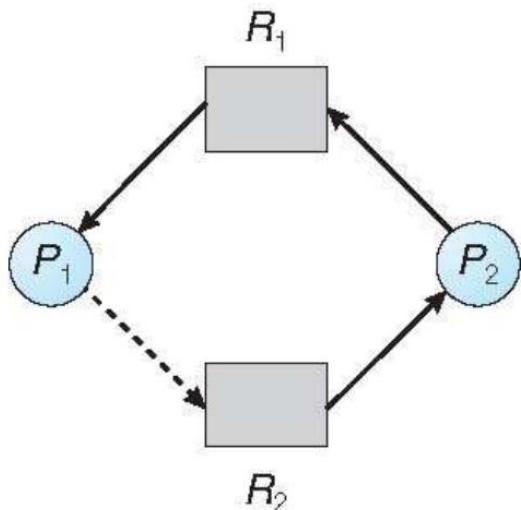
- Si introduce la **claim edge** (arco di richiesta)  $P_i \rightarrow R_j$  che indica che il processo  $P_j$  **potrebbe richiedere la risorsa**  $R_j$ ; (linea tratteggiata)
- La claim edge si converte in **request edge** quando un processo **richiede esplicitamente** la risorsa
- La request edge si converte in **assignment edge** quando la risorsa è **allocata** al processo
- Quando la **risorsa è rilasciata** dal processo, l'**assignment edge** si riconverte in **claim edge**

Questo schema si basa sul fatto che **le risorse devono essere richieste a priori nel sistema.**

### Esempio



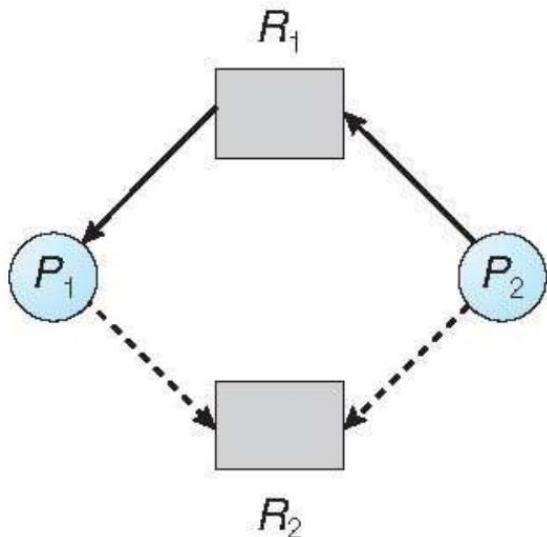
### Esempio 2



In questo caso ci troviamo in uno **stato unsafe** dovuto al fatto che potenzialmente se  $P_1$  richiede  $R_2$  si verifica un deadlock

### Algoritmo resource-allocation graph

- si suppone che il processo  $P_i$  richieda una risorsa  $R_j$  (**singola istanza**)
- La richiesta può essere garantita solo se trasformando l'arco di richiesta in un assegnamento **non porta alla formazione di un ciclo** nel grafo di allocazione delle risorse!
- Altrimenti la richiesta viene messa in attesa perché si troverebbe in uno stato unsafe.
- Complessità di rilevare un ciclo è  $n^2$  con  $n$  numero di thread del sistema



In questo caso  $P_2$  non sarebbe stata assegnata a  $R_2$  perché avrebbe portato in uno stato unsafe.

## ❖ Algoritmo del Banchiere

- E' il caso più generale, che tiene conto che per ogni tipo di risorsa possiamo avere **multiple istanze**
- Ogni processo deve dichiarare **a priori il suo massimo utilizzo di risorse** per ogni tipo di risorsa
- Quando un processo richiede una risorsa potrebbe dover aspettare perché il sistema altrimenti si troverebbe in uno stato non sicuro.
- Quando un processo ottiene tutte le sue **risorse** deve **restituirle in un tempo finito**

Ogni volta che un processo richiede un insieme di risorse (istanze multiple), il Sistema deve controllare se l'assegnazione di tali risorse lascia il Sistema in uno stato sicuro.

- Se lo stato è sicuro le risorse sono allocate
- altrimenti la richiesta viene messa in attesa finché qualche processo libera delle risorse aggiuntive

Si introducono diverse **strutture dati** per rappresentare lo **stato del sistema**:

Sia n = numero di processi ed m = numero di tipi di risorse.

- **Available:** vettore di lunghezza m (numero di tipi di risorse) in cui manteniamo memoria di quante risorse ci sono per ogni tipo. **Se available [j] = k, ci sono k istanze della risorsa di tipo R<sub>j</sub> disponibili.**
- **Max:** matrice n x m. Dato un processo e un tipo di risorsa rappresenta la richiesta massima di istanze che quel processo può fare per quel tipo di risorsa. Rappresenta il massimo utilizzo di risorse del processo Pi. **Se Max [i,j] = k, allora il processo Pi potrebbe richiedere al massimo k istanze della risorsa di tipo R<sub>j</sub>**
- **Allocation:** matrice di allocazione n x m. Rappresenta nello stato corrente l'impegno di un processo ossia il numero di risorse correntemente allocate per quel processo Pi. Il valore dovrebbe essere minore o uguale al valore massimo delle risorse per quel processo Pi. Se abbiamo allocato il massimo delle risorse per il processo Pi si auspica che al passaggio successivo il processo Pi liberi tutte le risorse. **Se Allocation[i,j] = k allora Pi ha correntemente allocate k istanze di R<sub>j</sub>**
- **Need:** matrice n x m derivata che rappresenta quante risorse ancora il processo Pi ha bisogno per ogni tipo. Supponendo che max sia il numero di risorse di cui il processo ha bisogno, si può ottenere facendo: **Need [i,j] = Max[i,j] - Allocation [i,j]. Se Need[i,j] = k, allora Pi potrebbe aver bisogno di k più istanze di R<sub>j</sub> per complare il suo task**

## ❖ Algoritmo di Safety

La prima cosa che deve fare l'algoritmo del banchiere è verificare (check) se lo stato attuale del sistema è sicuro.

- *Ricordiamo che lo stato è sicuro se abbiamo una sequenza ordinata di processi (o thread) in cui tutti i processi del sistema possono portare a termine in sequenza le loro computazioni raccogliendo le risorse impegnate dai processi precedenti più le risorse attualmente disponibili.*

1. Siano **Work** e **Finish** vettori di lunghezza  $m$  ed  $n$ , rispettivamente.  
Inizializza:

$$\mathbf{Work} = \mathbf{Available}$$

$$\mathbf{Finish}[i] = \mathbf{false} \text{ for } i = 0, 1, \dots, n-1$$

2. Trova un indice  $i$  tale che entrambi:

$$(a) \mathbf{Finish}[i] = \mathbf{false}$$

$$(b) \mathbf{Need}_i \leq \mathbf{Work}$$

Se non esiste  $i$ , vai al passo 4

3.  $\mathbf{Work} = \mathbf{Work} + \mathbf{Allocation}_i$ ,

$$\mathbf{Finish}[i] = \mathbf{true}$$

vai al passo 2

4. Se  $\mathbf{Finish}[i] == \mathbf{true}$  per tutti gli  $i$ , allora il sistema è in uno stato sicuro (safe state)

L'algoritmo verifica se lo stato è sicuro e può richiedere un ordine di  $m \times n^2$  operazioni

- Il vettore **Work** rappresenta le risorse disponibili in quel momento, grande quante sono i tipi di risorse. Viene inizializzato allo stato attuale delle risorse disponibili e non impegnate.
- Il vettore **Finish** è di lunghezza  $N$  (numero di processi) di tipo booleano inizializzato a false che serve per vedere se ogni processo ha terminato la sua computazione. Si inizializza a false perché si suppone che ogni processo debba ancora portare a termine la sua computazione.
- Si inizia a cercare un processo con indice  $i$  tale che la sua computazione non sia finita (ossia che  $\mathbf{Finish}[i] = \mathbf{false}$ ) e per il quale le risorse di cui ha bisogno sono soddisfabili da quelle attualmente disponibili ( $\mathbf{Need}(i) \leq \mathbf{Work}$ )
- se **non esiste il processo  $P_i$**  allora può essere che:
  - tutti hanno finito e quindi è verificato che **Finish[i] == true per tutti i processi** e il sistema è uno **stato sicuro**
  - oppure il sistema si trova in uno stato non sicuro!
- se **esiste il processo  $P_i$**  allora vengono **soddisfatte le sue richieste**:
  - viene aggiornata la matrice Work (Work + Allocation( $i$ ))
  - si suppone che il processo  $P_i$  abbia finito quindi **si pone Finish[i] == true**
  - a questo punto continua la ricerca per un altro indice  $i$

## ❖ Algoritmo di Richiesta di Risorsa per il Processo $P_i$

Dopo l'algoritmo di safety, si esegue questo algoritmo **per determinare se una richiesta può essere accordata in sicurezza.**

$\text{Request}_i$  = vettore richieste per il processo  $P_i$ .

Se  $\text{Request}_i[j] = k$  allora il processo  $P_i$  vuole  $k$  istanze delle risorse di tipo  $R_j$

1. Se  $\text{Request}_i \leq \text{Need}_i$  vai al passo 2. Altrimenti, segnala la condizione di errore dal momento che il processo ha superato il suo massimo dichiarato
2. Se  $\text{Request}_i \leq \text{Available}$  vai al passo 3. Altrimenti  $P_i$  deve aspettare perché le risorse non sono disponibili
3. Prova ad allocare le risorse richieste per  $P_i$  modificando lo stato come segue:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- Verifica se safe
- Se safe  $\Rightarrow$  le risorse sono allocate a  $P_i$
- Se unsafe  $\Rightarrow P_i$  deve aspettare e il vecchio stato di allocazione delle risorse viene recuperato

- Request[i] è il vettore di richieste per il processo Pi.
- Il processo Pi richiederà tramite il vettore delle richieste k istanze delle risorse di tipo Rj (Request di i [j])
- Se la richiesta di risorse è  $\leq$  di quante ne ha bisogno, allora può procedere. Altrimenti c'è un errore perché il processo fa una richiesta che va oltre a quante ne ha bisogno cioè il suo massimo dichiarato.
- Se la richiesta di risorse è  $\leq$  a quelle attualmente disponibili allora la richiesta **potrebbe** essere soddisfatta, altrimenti Pi deve attendere che si liberano perché impegnate
- Quindi se sono disponibili **si tenta di allocare le risorse** al processo Pi modificando lo stato:
  - si decrementa il vettore di risorse disponibili
  - si aggiorna la matrice allocation aggiungendo il valore di risorse impegnate per la richiesta soddisfatta
  - si aggiorna la matrice need dato che il processo ha bisogno di meno risorse
- A questo punto si applica l'**algoritmo di Safety**
  - **se lo stato è safe**  $\rightarrow$  le risorse vengono allocate e si avanza al processo successivo
  - **altrimenti se lo stato è unsafe**  $\rightarrow$  il processo deve aspettare e si ripristina il vecchio stato di allocazione delle risorse

## Esempio esercizio

- 5 processi da  $P_0$  a  $P_4$ :

3 tipi di risorse:

$A$  (10 istanze),  $B$  (5 istanze) e  $C$  (7 istanze)

- Stato al tempo  $T_0$ :

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	1	0	7	5	3
$P_1$	2	0	0	3	2	2
$P_2$	3	0	2	9	0	2
$P_3$	2	1	1	2	2	2
$P_4$	0	0	2	4	3	3

- Il contenuto della matrice **Need** è definito come **Max – Allocation**

	<u>Need</u>		
	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	7	4	3
$P_1$	1	2	2
$P_2$	6	0	0
$P_3$	0	1	1
$P_4$	4	3	1

- Il Sistema è in un stato sicuro (safe state) perché la sequenza  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  soddisfa i criteri di safety

## Esempio: $P_1$ Richiede (1,0,2)

- Controlla che Request  $\leq$  Available (cioè,  $(1,0,2) \leq (3,3,2) \Rightarrow \text{true}$

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

- Eseguendo l'algoritmo di safety si vede che la sequenza  $< P_1, P_3, P_4, P_0, P_2 >$  soddisfa i criteri
- Può la richiesta per (3,3,0) di  $P_4$  essere garantita?
- Può la richiesta per (0,2,0) di  $P_0$  essere grantita?

quando il sistema si trova in questo stato, una richiesta di (3,3,0) da parte di p4 non si può soddisfare perché non sono disponibili le risorse. Inoltre, una richiesta di (0, 2, 0) da parte di p0 non si può soddisfare, anche se le risorse sono disponibili, poiché lo stato risultante sarebbe non sicuro.

### ❖ Rilevazione del Deadlock: algoritmi di Detection Deadlock

C'è un costo per evitare il Deadlock con questi algoritmi ed è abbastanza costoso e prevedere le risorse che un processo allocherà è abbastanza difficile, per questo alcuni sistemi operativi preferiscono ignorare il problema.

**Ci sono quindi soluzioni che permettono al sistema di entrare in uno stato insicuro in cui si verifica deadlock, però bisogna riconoscerlo. Servono quindi:**

- Algoritmi di deadlock detection
- Metodi di deadlock recovery

**Costi:**

- Costo del monitoraggio (strutture dati ed algoritmi)
- Potenziale perdita di dati durante il recovery

Cosideriamo due casi:

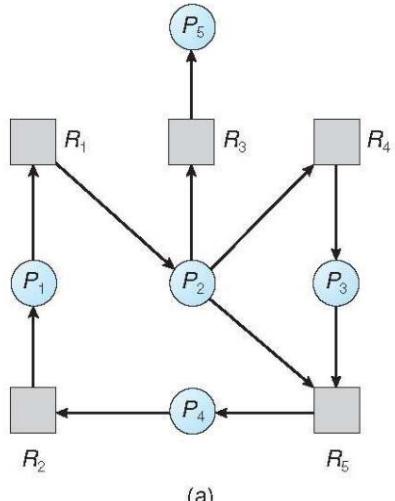
- Istanza singola per ogni tipo di risorsa
- Istanze multiple per ogni tipo di risorsa

### ❖ Algoritmo di detection per Istanza singola

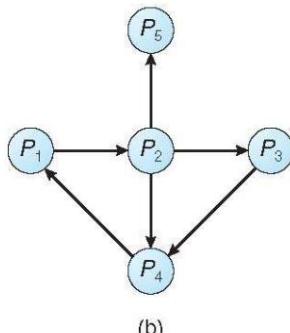
Per effettuare il detection si usa una variante del grafo di allocazione chiamato **grafo wait-for** (grafo delle attese).

- Si rimuovono le risorse e si bypassano con archi tra processi: i nodi diventano solo processi
- $P_i \rightarrow P_j$  se  $P_i$  sta aspettando  $P_j$
- Periodicamente si invoca un algoritmo che cerca i cicli nel grafo. Se c'è ciclo allora c'è un deadlock. Non esistono più cicli potenziali ma solo cicli reali dovuti allo stato attuale (non potenziale).

Un algoritmo per rilevare un ciclo in un grafo richiede un ordine di  $n^2$  operazioni, dove n è il numero di vertici nel grafo



(a)



(b)

Grafo di allocazione delle risorse    Grafo delle Attese corrispondente

- **Linux mette a disposizione un deadlock\_detector** che traccia l'uso dei `pthread_mutex_lock` e `pthread_mutex_unlock` costruendo un grafo delle attese e segnalando i deadlock

## ❖ Algoritmo di detection per Istanze multiple

Occorrono più strutture dati come l'algoritmo del banchiere

- **Available:** un vettore di lunghezza  $m$  che indica il numero di risorse disponibili per ogni tipo di risorsa
- **Allocation:** una matrice  $n \times m$  definisce il numero di risorse per tipo correntemente allocato ad ogni processo
- **Request:** una matrice  $n \times m$  indica la richiesta corrente di ogni processo. Se  $\text{Request}[i][j] = k$ , allora il processo  $P_i$  sta richiedendo  $k$  più istanze di risorsa di tipo  $R_j$ .

### Algoritmo di rilevamento

1. Siano **Work** e **Finish** vettori di lunghezza  $m$  ed  $n$

Inizializza:

- (a) **Work = Available**
- (b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_i \neq 0$ , then  
**Finish[i] = false**; otherwise, **Finish[i] = true**

2. Trova un indice  $i$  tale che entrambi:

- (a) **Finish[i] == false**
- (b) **Request<sub>i</sub> ≤ Work**

Se non esiste tale  $i$ , vai al passo 4

3. **Work = Work + Allocation<sub>i</sub>**  
**Finish[i] = true**  
vai al passo 2

Rilasciate le risorse dopo il completamento,  
si assume che non siano necessarie altre  
risorse da allocare

4. Se **Finish[i] == false**, per qualche  $i$ ,  $1 \leq i \leq n$ , allora il sistema è  
in uno stato di deadlock. Inoltre, se **Finish[i] == false**, allora  $P_i$  è  
in deadlocked

- 1) Vettore work che mantiene lo stato di disponibilità delle risorse, inizialmente quelle messe a  
disposizione del sistema tramite il vettore Available. A questo punto si inizializza i processi: se non  
hanno bisogno di alcuna allocazione per la loro computazione allora vengono direttamente  
skippati dal check e li si pone come  $\text{finish}[i] == \text{true}$ , altrimenti se hanno bisogno di allocare risorse  
( $\text{Allocation} != 0$ ) vengono inizializzati a false.
- 2) Per ogni processo bisogna trovare un indice di processo tale per cui il processo non ha ancora  
terminato la sua computazione (e quindi ha ancora bisogno delle risorse) e tale che la sua richiesta  
può essere soddisfatta.
- 3) Si assume che la richiesta viene soddisfatta e il processo termina la computazione rilasciando le  
risorse e quindi si pone  $\text{Finish}[i] == \text{true}$  e si aggiungono al vettore work di disponibilità del sistema.  
Si                  cerca                  quindi                  un                  altro                  processo                  Pi.
- 4) Nel momento in cui un processo Pi si trova in uno stato  $\text{Finish}[i] == \text{false}$  allora c'è un deadlock  
perché non c'è un modo per il processo di sbloccarsi. Non si arriva alla terminazione di tutti i  
processi con  $\text{Finish}[i] == \text{true}$ .

L'algoritmo richiede un ordine di  $O(m \times n^2)$  operazioni per rilevare se il sistema è in stato di deadlock

### Esempio

- Cinque processi da  $P_0$  a  $P_4$ ; tre tipi di risorsa A (7 istanze), B (2 istanze), and C (6 istanze)

- Stato al tempo  $T_0$ :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
$P_0$	0 1 0	0 0 0	0 0 0
$P_1$	2 0 0	2 0 2	
$P_2$	3 0 3	0 0 0	
$P_3$	2 1 1	1 0 0	
$P_4$	0 0 2	0 0 2	

- La sequenza  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$  porterà a  $Finish[i] = \text{true}$  per tutti le  $i$   
 $P_2$  richiede un'istanza addizionale di tipo C

	<u>Request</u>
	A B C
$P_0$	0 0 0
$P_1$	2 0 2
$P_2$	0 0 1
$P_3$	1 0 0
$P_4$	0 0 2

Stato del sistema?

- Può rilasciare le risorse tenute dal processo  $P_0$ , ma le risorse sono insufficienti per soddisfare le richieste degli altri processi
- Esiste un deadlock per i processi  $P_1$ ,  $P_2$ ,  $P_3$ , e  $P_4$

ora il sistema è in stallo. Anche se si possono liberare le risorse possedute dal processo p0, il numero delle risorse disponibili non è sufficiente per soddisfare le richieste degli altri processi, quindi si verifica uno stallo composto dai processi p1, p2, p3 e p4.

## Uso dell' Algoritmo di rilevamento

- **occorre stabilire quanto spesso invocare il rilevamento**
- quindi occorre considerare **quanto spesso è probabile che un deadlock avvenga:**
  - se è probabile che avvenga frequentemente allora bisogna invocarlo frequentemente l'algoritmo di rilevamento
  - bisogna considerare quanti processi può coinvolgere (magari definire un sottoinsieme su cui agire di più)
  - Nel caso estremo può essere invocato ogni volta che una richiesta non può essere soddisfatta: ma è costoso. In questo caso si può identificare anche chi ha causato il deadlock
  - Si potrebbe fare un check di rilevamento dopo una certa deadline dopo aver saputo che la richiesta non può essere soddisfatta, però se si allunga troppo il sistema potrebbe rimanere in deadlock e più processi potrebbero essere coinvolti nel deadlock
  - Meno sono i processi e più è facile fare il rilevamento e recovery del deadlock
- Quanti processi sarà necessario recuperare? uno per ogni ciclo disgiunto

**Se il detection viene ritardato sono possibili molti cicli nel grafo delle risorse:** complicato risalire a quali dei tanti processi abbia "causato" il deadlock.

## ❖ Ripristino dal Deadlock: Recovery

### Quando viene trovato un deadlock va gestito:

- avvertendo un operatore che deve **gestire a mano** la situazione
- **Gestione automatica**, secondo due modalità:
  - **terminare dei processi**, quindi obbligandoli a deallocare risorse
  - **aggiungere risorse da altri processi** che le impegnano per ripristinare il deadlock

### Gestione automatica: Terminazione

- Si fa l'**abort di tutti i processi in deadlock** e ciò è **molto dispendioso** in quanto tutta la computazione dei processi fatta è persa.
- Oppure si fa l' **Abort dei processi uno alla volta** finché il ciclo di deadlock è eliminato. In questo caso viene invocato l'algoritmo di rilevamento del deadlock frequentemente e ciò può portare ad overhead.

### Problema: In quale ordine dovremmo scegliere i processi in deadlock per fare l'abort?

Si dovrebbe scegliere i processi sacrificabili secondo alcuni criteri andando a definire una funzione di costo per il kill dei processi. Criteri a cui dare il costo:

1. **Priorità del processo:** si killa i processi a bassa priorità mantenendo quelli ad alta priorità
2. **Per quanto tempo ha lavorato e quanto manca al completamento:** ad esempio non dobbiamo killare un processo a bassa priorità che ha lavorato in background per tanto tempo e ha svolto tanta computazione
3. **Quante e quali risorse il processo ha già usato (si possono prelazionare?)** Magari si sottraggono al processo e si riassegnano (altra modalità di gestione automatica)
4. **Quante risorse il processo necessita per completare:** si sacrifica un processo che ne alloca tante per sperare che il processo che ne ha bisogno di poche si sblocchi e porti il sistema fuori dal deadlock
5. **Quantи processi necessiteranno di essere terminati:** in conseguenza di un processo che presenta dipendenze con altri processi (kill a cascata)
6. **È un processo interattivo o batch**

### **Gestione automatica: Prelazione di risorse**

Consiste nell' aggiungere risorse da altri processi che le impegnano per ripristinare il deadlock, quindi si selezionano risorse da prelazionare finché il deadlock non è risolto.

Per la selezione della risorsa occorre:

- **Selezionare una vittima:** quale risorsa e quale processo prelazionare? Occorre minimizzare il costo: valutare il numero di risorse trattenute, il tempo di computazione impiegato, etc.
- **Rollback:** se viene prelazionato un processo che farne? Dopo aver risolto il deadlock o si fa **ripartire da capo** o si fa **ripartire da prima che aveva impegnato le risorse** tornando in uno stato safe (rollback) quindi si mantiene uno stato intermedio di computazione (costoso).
- **Starvation:** una funzione di costo potrebbe selezionare sempre lo stesso processo o lo stesso insieme di processi come vittime da killare e quindi potrebbero non essere mai eseguiti perché interrotti continuamente. Per evitarlo si può includere il **numero dei rollback subiti** nei fattori di costo e **quante volte è stato prelazionato**

## **7. MEMORIA PRINCIPALE**

Si presentano diversi metodi di gestione della memoria. Gli algoritmi di gestione della memoria variano dall'approccio più semplice che accede all'hardware della macchina in maniera diretta ai metodi di paginazione e segmentazione. Ogni metodo presenta vantaggi e svantaggi. La scelta di un metodo specifico di gestione della memoria dipende da molti fattori, in particolar modo dall'architettura hardware del sistema, infatti molti algoritmi richiedono un supporto hardware.

### ❖ Concetti principali

- Un sistema di elaborazione esegue programmi. I programmi in esecuzione diventano processi. Per essere eseguito un **programma** deve essere portato (**dal disco**, unità in cui è memorizzato permanentemente) **in memoria principale** e inserito all'interno di un processo che poi viene gestito ed eseguito dalla CPU.
- La **memoria principale** ed i **registri** sono la sola memoria a cui la **CPU accede direttamente** interpretando le istruzioni dei programmi.
- Tanti processi devono essere caricati (almeno parzialmente) in memoria contemporaneamente per avere la multiprogrammazione.
- Il **tempo di accesso** in memoria impatta in modo molto elevato sulle performance del sistema.
- La CPU preleva dalla memoria le istruzioni indicate dal PC, le decodifica e poi le esegue: processo per processo. I risultati vengono scritti di nuovo in memoria. (load e store)
- La **memoria principale** può essere vista come un **grande array di bytes**, in cui ciascun byte ha un proprio **indirizzo**.
- La memoria vede soltanto un **flusso d'indirizzi di memoria**, e non sa come sono generati + **richieste di lettura o indirizzo + dati e richieste di scrittura**.

### **Hardware**

La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la Cpu può accedere direttamente.

Siccome gli accessi in memoria sono molto veloci spesso occorre un **supporto hardware** per velocizzare l'accesso in memoria. Se l'accesso ai registri richiede un clock della CPU (o anche meno), la **memoria principale** invece può richiedere **molti cicli** (si passa tra i bus, si punta alla locazione richiesta ecc.), causando uno **stallo in attesa del dato da elaborare**. Nel caso in cui facciamo accessi frequenti, senza un opportuno meccanismo di velocizzazione degli accessi ci sarebbero troppi stalli. Per questo motivo si introduce una **gerarchia di memorie cache** tra la **memoria principale e i registri della CPU** per accelerare gli accessi in memoria.

Protezioni hardware:

- Necessaria **protezione hardware** della memoria per garantirne il corretto funzionamento.
- Si distingue **modalità kernel e modalità user** attraverso il **bit di modalità**
- Oltre questo si utilizzano **registri per limitare lo spazio di memoria allocato per ciascun processo**, per evitare che un processo utente accedi al codice del kernel o degli interrupt facendo una sorta di dirottamento del sistema.
- Il S.O. non interviene tra CPU e Memoria per non rallentare: esso imposta solo le cose, tutto il resto deve avvenire via hardware.

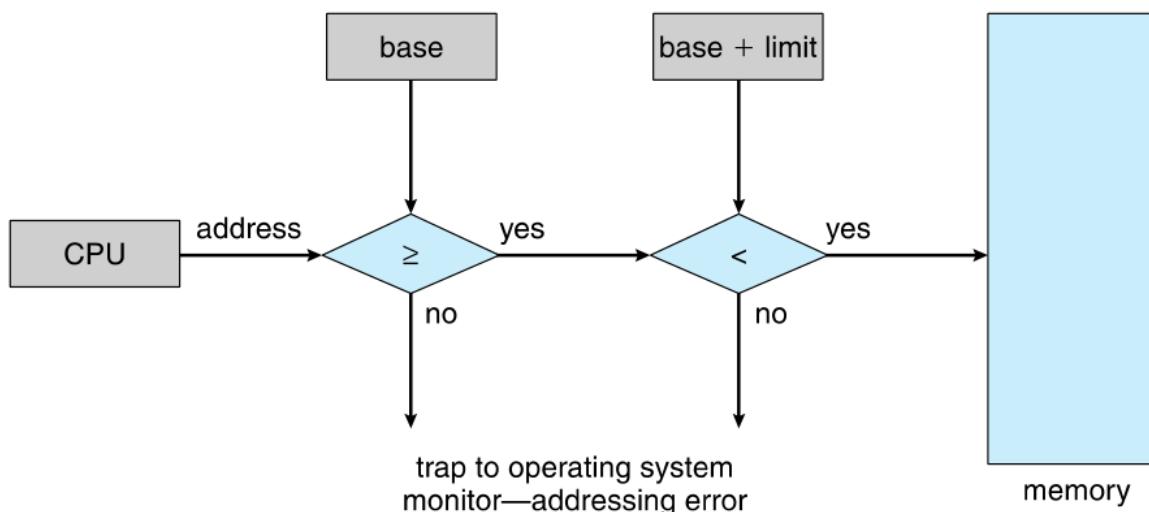
## Protezione memoria: Registri Base e Limite

In primo luogo occorre separare i processi in memoria. Si introduce una **coppia di registri: base e limite** i quali vengono precaricati dal sistema e che **definiscono lo spazio degli indirizzi logici per un processo**.

- **Registro Base:** contiene il primo indirizzo fisico da cui si può caricare il processo
- **Registro Llimite:** contiene il range che definisce la dimensione dell'intervallo d'indirizzi ammessi
- Massimo indirizzo: Primo indirizzo fisico + Limite

Per mettere in atto il meccanismo di protezione, la **Cpu confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri**. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di una **eccezione (trap)** che restituisce il controllo al **sistema operativo** che, a sua volta, interpreta l'evento come un **errore fatale**. **Il controllo di accesso è hardware**.

I registri possono essere caricati solo dal SO in modalità privilegiata, quindi solo in kernel mode. In kernel mode il SO ha accesso non ristretto sia alla memoria.



## ❖ Binding degli indirizzi

- Il **binding** rappresenta un collegamento che “mappa” (cioè fa un’associazione) uno spazio di indirizzi in un altro spazio di indirizzi

In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria. I processi in attesa di essere caricati in memoria formano una **coda di input**. Una volta caricati i processi la CPU può accedere a dati ed istruzioni. In linea di principio il processo può risiedere in ogni parte della memoria. Quando il processo termina lo spazio di memoria viene reso disponibile (rilasciato).

Quando i porgrammi vengono caricati dal disco in memoria principale per diventare processi avviene il caricamento delle istruzioni in memoria e vengono associate ad indirizzi fisici di memoria.

Gli **indirizzi sono rappresentati in modi diversi** nei momenti diversi del ciclo di vita di un programma:

- Nel codice sorgente sono simbolici (es. variabili)
- Nel codice compilato gli indirizzi sono associati (**bind**) ad indirizzi rilocabili
- Il **loader** legherà effettivamente gli **indirizzi rilocabili** agli **indirizzi assoluti** (ulteriore bind).

Il **binding** (degli indirizzi di istruzioni e dati) agli indirizzi di memoria **può avvenire in diverse fasi**:

- **Tempo di Compilazione:** quando al momento della compilazione è **già noto dove il processo risiederà in memoria** e quindi si può generare **codice assoluto**. Il codice va ricompilato se la locazione di partenza viene modificata.
- **Tempo di Caricamento:** se la **locazione di memoria non è nota a tempo di compilazione** si genera **codice rilocabile**; il collegamento finale viene ritardato finché non c'è il caricamento. Se la locazione iniziale per il processo cambia occorre ricaricare il codice.
- **Tempo di Esecuzione:** si verifica quando durante l'esecuzione il **processo può essere spostato da un segmento di memoria all'altro; il binding è ritardato fino al run-time**. Occorre supporto hardware speciale per mappare gli indirizzi.

### ❖ Indirizzi logici e fisici

La **separazione** tra indirizzi logici ed indirizzi fisici è centrale per la gestione della memoria.

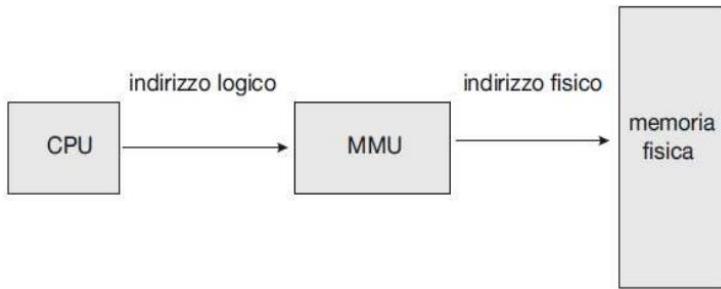
- **Indirizzo logico:** gli indirizzi logici sono quelli generati e gestiti dalla **CPU**; vengono anche chiamati **indirizzi virtuali**. Sono quelli all'interno di un processo a cui fa riferimento la CPU ma non corrispondono a quelli reali della memoria principale. Per farli corrispondere con quelli reali occorre un binding.
- **Indirizzo fisico:** indirizzo visto dall'unità di memoria; viene caricato nel registro MAR memory address register.
- Il **binding a tempo di compilazione e caricamento** produce indirizzi fisici **identici** agli indirizzi logici. Il **binding a tempo di esecuzione** produce indirizzi fisici **differenti** da quelli logici.

L'insieme di tutti gli indirizzi logici generati da un programma a cui esso e la CPU fanno riferimento è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**.

La CPU lavorerà sempre esclusivamente con **indirizzi logici**. Ci sarà un dispositivo hardware chiamato **MMU (Memory Management Unit)** che mapperà gli indirizzi logici in indirizzi fisici nella memoria fisica.

## ❖ Memory Management Unit (MMU)

- E' un dispositivo hardware che a run-time mappa gli indirizzi virtuali in indirizzi fisici



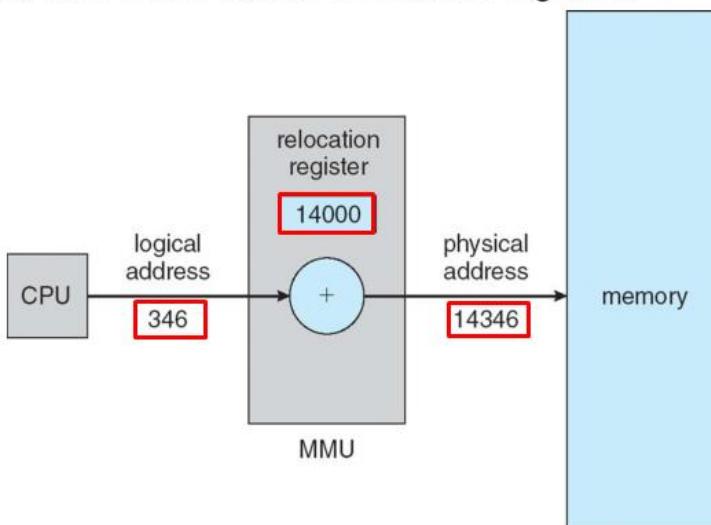
Esistono diversi metodi per fare il mapping a run-time. Consideriamo uno schema basato su **registri base** e **limite**. In questo caso il registro di base è chiamato **registro di rilocazione**. In questo registro è contenuto l'indirizzo **che indica la base del segmento di memoria assegnato ad un certo processo**.

A run-time la CPU esegue un processo che è stato schedulato e a cui è stato predisposto uno spazio di memoria. Ad un certo punto la CPU vuole fare un accesso in memoria riguardante quel processo e fa affidamento ad un indirizzo logico.

**La MMU effettua il mapping partendo dall'indirizzo logico a cui fa riferimento la CPU a cui viene sommato (e quindi si fa uno shift) l'indirizzo contenuto nel relocation register per ottenere come risultato l'indirizzo fisico.**

Quindi grazie al mapping effettuato dalla MMU (e all'hardware) la CPU riesce ad accedere alla memoria fisica partendo dall'indirizzo logico con il quale lavora.

- Es. CPU considera 346 (logico) che viene mappato su 14346
- MS-DOS su Intel 80x86 usava 4 relocation registers



Ricordiamo quindi che il processo utente vede solo gli indirizzi logici, non vede mai gli indirizzi fisici reali. Il processo può gestire un puntatore, manipolarlo etc. rimanendo sempre nello spazio degli indirizzi logici. **Il binding a tempo di esecuzione occorre solo quando si accede ad una locazione di memoria fisica.** In questo caso l'hardware per il mapping in memoria converte l'indirizzo virtuale in indirizzo fisico. **Se non c'è bisogno di accedere alla memoria fisica la CPU continua a lavorare con indirizzi logici.**

*Il processo utente lavora nello spazio di indirizzi logici (es. intervallo [0, max]) che in memoria saranno mappati in indirizzi fisici (es. [R, R + max])*

**Caricamento dinamico:** a volte non è necessario che tutto il programma sia caricato in memoria per essere eseguito. A volte vengono caricate solo parti di programma; le routine non utilizzate possono essere tenute su disco in formato rilocabile per poter essere poi dinamicamente allocate in seguito. Una routine quindi non è caricata finché non è chiamata e routine non usate non vengono mai caricate.

**Linking statico:** librerie e codice del programma vengono collegate dal loader nell'immagine binaria del programma in formato eseguibile.

**Linking dinamico:** il linking è posposto fino al tempo di esecuzione. Simile al caricamento dinamico, invece del caricamento viene posticipato il link delle librerie. Utilizzato per librerie di sistema (es. C standard library). **Dynamic Linking Libraries (DLLs)** evitano il caricamento completo per ogni programma.

#### ❖ Allocazione della memoria

La memoria centrale deve contenere sia il sistema operativo e i suoi processi di sistema sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente.

Memoria principale di solito divisa in **2 partizioni**:

- **una partizione relativa al Sistema operativo:** residente in una parte alta o bassa di memoria (Linux e Windows sono memorizzati nella parte alta)
- **una partizione relativa ai Processi utente:** assumiamo quindi sia residente in memoria bassa

Si vuole che **più processi utente risiedano contemporaneamente in memoria centrale**. Perciò è necessario considerare come assegnare la memoria disponibile ai processi. Abbiamo diverse modalità.

#### ❖ Allocazione contigua

Nell'allocazione contigua, processi che vengono caricati in modo successivo tra loro vengono caricati in locazioni contigue di memoria. **Ogni processo è contenuto in una singola sezione o partizione contigua di memoria.** Sono necessari **meccanismi di protezione della memoria** (registro base e limite) per non allocare un processo in sezioni di memoria non accessibili. Quindi la partizione in questo caso è una e tutti i processi sono caricati in modo contiguo in modo successivo.

#### ❖ Allocazione con partizioni multiple

La memoria è suddivisa in partizioni multiple e ad ogni processo è assegnata esattamente una partizione di dimensione variabile a seconda delle esigenze del processo. Il SO mantiene traccia delle partizioni occupate e di quelle libere.

Quando un processo libera una partizione si viene a creare un **hole** ossia la situazione in cui è disponibile un blocco di memoria. All'inizio tutta la memoria è disponibile e quindi abbiamo un **unico hole**. Quando diversi processi liberano le proprie partizioni possiamo trovarci in situazioni in cui abbiamo **buchi di dimensioni varie sparpagliati per la memoria**.

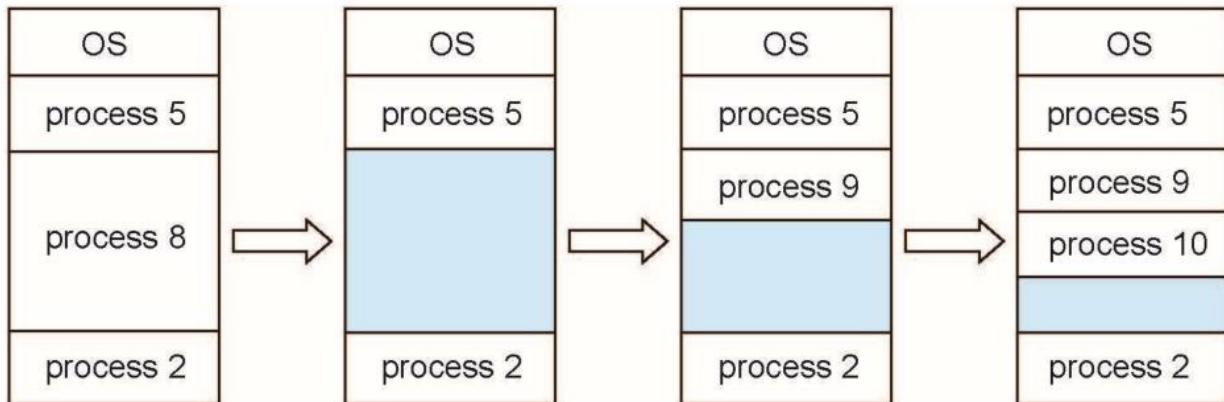
Quando un processo arriva, è allocato in un hole largo abbastanza per gestirlo. Il sistema tiene traccia di tutti gli hole e della loro dimensione. La dimensione dell'hole è variabile e dimensionata sui bisogni del processo.

**I processi uscenti liberano le partizioni e le partizioni libere e adiacenti possono essere combinate in una sola. Quindi il SO deve mantenere informazioni su:**

- **partizioni allocate**
- **partizioni libere (hole)**

**Grado di multiprogrammazione limitato** dal numero di partizioni





Se non c'è spazio sufficiente il processo può essere respinto o messo in attesa di un hole libero per contenerlo. Se il buco per un processo è grande viene diviso allocandone una parte al processo e rilasciandone una parte libera per altri processi. Esistono diversi metodi per allocare gli hole.

### Metodi per allocare gli hole

Esistono diversi metodi per allocare gli hole e soddisfare una **richiesta di partizione dimensione n da una lista di hole liberi**.

- **First-fit:** alloca il **primo hole libero che è grande abbastanza** per contenere il processo. Non deve scandire tutta la lista di hole liberi. Può scandire dall'inizio dell'insieme di buchi o dal punto in cui era terminata l'ultima ricerca di hole. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- **Best-fit:** alloca il **più piccolo buco in grado di contenere il processo**. Si deve compiere la ricerca in tutta la lista, a meno che questa non sia ordinata per dimensione. Produce il più piccolo residuo di hole liberi. **Dei residui di hole molto piccoli potrebbero essere difficili da gestire: frammentazione della memoria.**
- **Worst-fit:** Alloca il più largo hole. Anche in questo caso si deve esaminare tutta la lista, a meno che non sia ordinata per dimensione. Produce il più largo residuo di hole, che è più utile di residui più piccoli perché è più riutilizzabile.

Sono stati condotti molti studi e simulazioni sul comportamento di questi algoritmi. Le conclusioni sono che il **first fit è di norma ritenuto il più efficiente** sicuramente in termini di velocità, non chiaro in termini di storage.

## ❖ **Frammentazione**

L'allocazione della memoria in partizioni multiple porta al problema della frammentazione. Il fenomeno si verifica quando abbiamo tanti hole di dimensione variabile sparpagliati nella memoria i quali diventano inutilizzabili per allocare un nuovo processo, perché magari di dimensioni non adeguate.

**La frammentazione può essere di 2 tipi: interna ed esterna.**

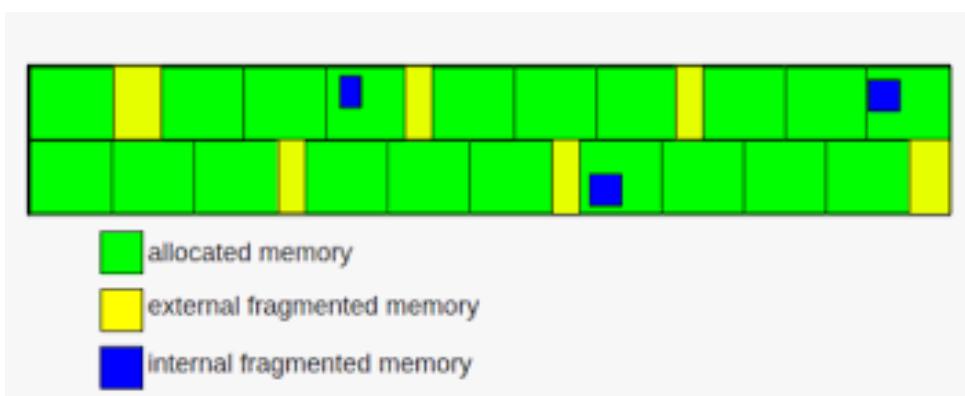
❖ **Frammentazione esterna:** è una situazione di frammentazione in cui esiste la quantità di memoria totale per gestire l'allocazione di un processo, ma non è contigua. Quindi abbiamo tanti piccoli hole non contigui i quali non permettono di soddisfare una richiesta, anche se invece sommandoli si otterrebbe memoria sufficiente per l'allocazione. **Ne soffrono gli algoritmi best-fit e first-fit.** Si dice esterna perché è esterna rispetto alla memoria allocata per i processi. Esistono metodi di compattazione della memoria che vengono eseguiti run-time ma hanno un certo costo.

- **Con first-fit dati N blocchi allocati, statisticamente  $N/2$  (la metà) dei blocchi sono persi per frammentazione.  $1/3$  dei blocchi potrebbe non essere usabile: 50% rule.**

Per evitare il problema della frammentazione esterna si può dividere la memoria in **blocchi di dimensione fissa**, ma questo porta alla **frammentazione interna**.

❖ **Frammentazione interna:** è una situazione di frammentazione in cui la memoria allocata per un processo è maggiore di quella richiesta da esso e quindi è in eccesso rispetto alle reali esigenze di allocazione del processo. Tale memoria in eccesso è inutilizzabile perché, appunto, è stata allocata. Si dice interna perché è interna ad una partizione di memoria allocata ad un processo. Si ottiene la quantità di memoria in eccesso facendo la sottrazione tra memoria effettivamente allocata e memoria realmente necessaria per allocare quel processo.

- **Esempio:** Processo richiede 18462 byte ma buco di 18464 byte, vengono persi quindi 2 byte: il buco è troppo piccolo per tenerne traccia.



## ❖ **Come si risolve allora il problema della frammentazione?**

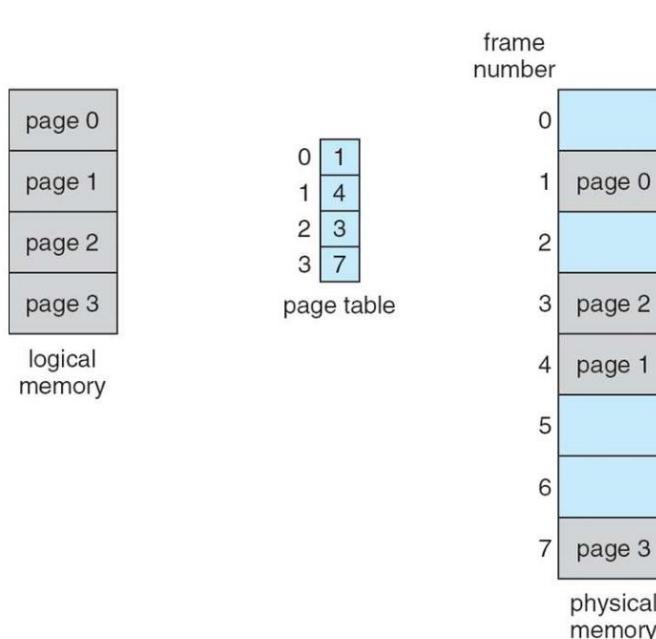
La **frammentazione esterna** si riduce con la **compattazione**. Consiste nel riassegnare i contenuti della memoria per mettere insieme tutta la memoria libera in un blocco di grandi dimensioni. La compattazione è possibile se la rilocazione è dinamica ed è fatta a tempo di esecuzione.

Un'altra strategia per evitare la frammentazione è la **paginazione** che permette l'allocazione di processi in locazioni non contigue di memoria. Il **paging** è la strategia più usata.

## ❖ Paging

Si parte dalla considerazione che **lo spazio degli indirizzi fisici di un processo può essere non contiguo**. Ad un processo è allocata memoria fisica quando è disponibile in locazioni libere sparse. In questo modo si evita completamente la frammentazione esterna e il problema di frammenti di memoria con dimensioni variabili. Lo svantaggio è quello che questo metodo richiede una traduzione più sofisticata degli indirizzi logici in fisici e la gestione di strutture dati adatte che tengono traccia di come è memorizzato il processo in memoria principale. Inoltre mantiene il problema della frammentazione interna.

- La tecnica consiste nel dividere la memoria fisica in **blocchi di dimensione fissa** detti **frame**. Di solito la dimensione del frame è espressa come potenza di 2 (solitamente da 4KB a 2GB).
- Lo spazio di indirizzamento logico dei processi su cui lavora la CPU e analogamente i processi, detto anche **memoria logica**, si divide in **blocchi della stessa dimensione detti pagine (page)**.
- **Ad ogni frame in memoria fisica corrisponderà una page in memoria logica della stessa dimensione.**
- Si tiene traccia di tutti i frame liberi. Un frame sarà o tutto libero o tutto occupato, ma tutti saranno di dimensione fissa.
- I processi ragioneranno in termini di occupazione di pagine. Per lanciare un programma che diventerà un processo di dimensione **N page**, si devono trovare **N frame liberi** e caricare il programma. I frame di cui ha bisogno un programma per essere caricato in memoria possono essere sparsi ed occorrerà non cercarli necessariamente in regioni contigue.
- Occorre una **page table** per tradurre gli indirizzi logici in fisici. **La tabella delle pagine è per processo, ogni processo avrà la sua tabella**
- Esistono ancora **problematiche di frammentazione interna** dovuti al fatto che stiamo ragionando con blocchi di memoria di dimensione fissa. Un processo potrebbe occupare anche un byte di una sola pagina.

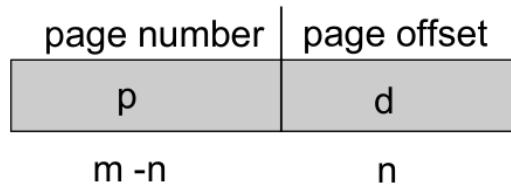


Le 4 pagine che vede il processo in modo contiguo nella sua memoria logica, non sono allocate anche in memoria fisica in modo contiguo. Le pagine sono allocate dove il sistema trova locazioni libere in memoria principale. Grazie alla page table possiamo fare il mapping tra pagine della memoria logica e frame della memoria principale. Si associa il numero di pagina all'indice del frame number corrispondente in memoria fisica.

## ❖ Schema di traduzione degli indirizzi logici

La CPU lavora su indirizzi logici. Bisogna tradurre gli indirizzi logici in fisici. Nel caso del paging l'**indirizzo logico** generato dalla CPU è diviso in **2 parti**:

- **Page number (p):** usato come indice nella page table che contiene l'indirizzo di base di ogni pagina logica mappata nella memoria fisica
- **Page offset (d):** è l'offset (spiazzamento) all'interno della pagina che combinato con l'indirizzo base definisce l'indirizzo di memoria fisica di una certa locazione all'interno di una pagina mappata in memoria

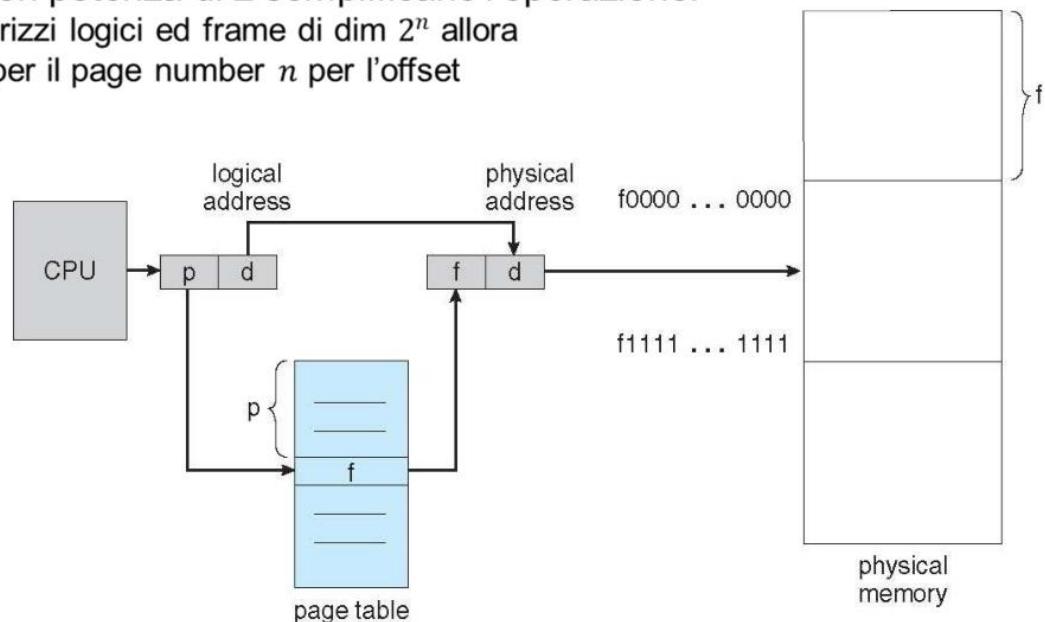


La MMU esegue i seguenti passi:

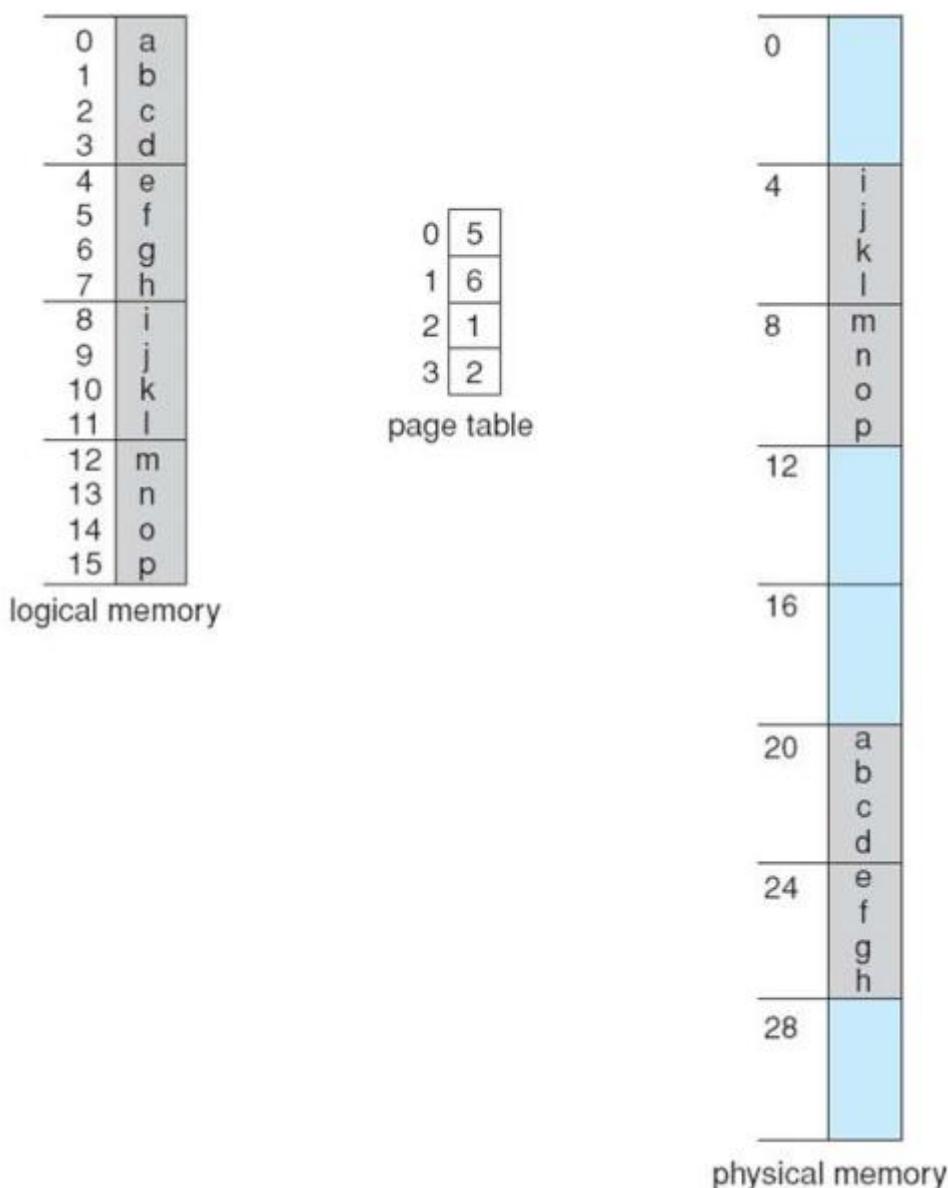
- Estraie in numero p di pagina usando come indice
- Estraie il frame number f dalla tabella
- Compone l'indirizzo sostituendo f a p

Le pagine con potenza di 2 semplificano l'operazione:

- Se  $2^m$  indirizzi logici ed frame di dim  $2^n$  allora
- $m - n$  bit per il page number  $n$  per l'offset



## ❖ Esempio paging



- Si utilizzano **pagine di 4 byte** e una **memoria fisica di 32 byte** (che può contenere  $32/4 = 8$  pagine).
- Supponiamo di voler tradurre l'**Indirizzo logico 0** nella corrispondente locazione fisica in memoria. L'indirizzo logico 0 si trova nella pagina 0 con offset 0. Vediamo la tabella delle pagine che ci dice che la pagina 0 si trova al frame 5 della memoria fisica. Andiamo al frame 5 in memoria fisica, corrispondente all'indirizzo 20 e sommiamo l'offset 0. **Quindi l'indirizzo logico 0 corrisponde all'indirizzo fisico 20 (il contenuto è la lettera 'a').**
- **Indirizzo logico 3:** pagina 0, offset 3. Pagina 0 si trova al frame 5. Frame  $5 \times 4$  byte di ogni pagina =  $20 + \text{offset } 3 = 23$  **indirizzo fisico.**
- **Indirizzo logico 4:** pagina 1, offset 0. Pagina 1 si trova al frame 6. Frame  $6 \times 4$  byte di ogni pagina =  $24 + \text{offset } 0 = 24$  **indirizzo fisico**
- **Indirizzo logico 13:** pagina 3, offset 1. Pagina 3 si trova al frame 2. Frame  $2 \times 4$  byte di ogni pagina =  $8 + \text{offset } 1 = 9$  **indirizzo fisico**

## □ Calcolo della frammentazione interna

### □ Esempio:

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Frammentazione interna di  $2,048 - 1,086 = 962$  bytes

### □ **Frammentazione worst case = 1 frame – 1 byte**

- In media frammentazione =  $1 / 2$  frame size
- Quindi piccoli frame desiderabili?
- ... ma ogni entry della page table impegna memoria per tracciarla
- Le dimensioni delle pagine crescono nel tempo

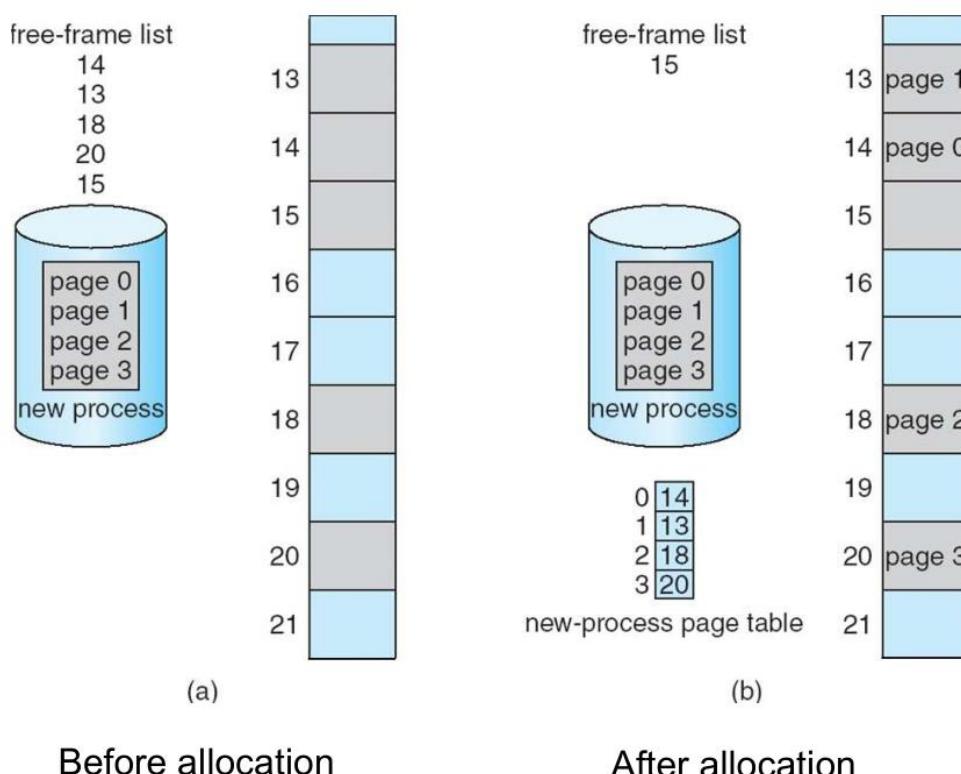
### **Calcolo frammentazione interna spiegato**

Process size 72 766 bytes / 2048 page size = 35.53 pages  
 35.53 pages sono 35 pagine da 2048 + altri  $0.53 * 2048$  page size =  $1085.44 = 1086$  bytes.  
 Quindi avremo 35 pagine + 1086 bytes.  
 Frammentazione interna =  $2048 - 1086$  bytes = 962 bytes inutilizzati.

- **Più sono piccoli i frame più pagine devono essere tenute in memoria ed essere tracciate; la dimensione della page table aumenta e la sua gestione comporta overhead.**

### ❖ Frame Liberi

Un processo da eseguire viene valutato in pagine e se richiede n pagine occorrono n frame liberi in memoria. Le pagine sono via via allocate e associate ai frame liberi



Before allocation

After allocation

## ❖ Implementazione della page table

- Esiste una page table associata ad ogni processo. Ogni processo contiene un **puntatore nel PCB che punta alla sua tabella delle pagine**. Quando un processo va in esecuzione devono essere ricaricati i registri e ricaricati i valori per la gestione della page table.

La page table può essere implementata in vari modi:

- attraverso la sua **memorizzazione in registri dedicati** e caricati dal dispatcher della CPU. Il sistema operativo vi ha accesso in modalità privilegiata. Questo metodo può essere usato soltanto se la tabella è di piccole dimensioni (massimo 256 entry) e richiede la gestione del context switch, perché bisogna ricaricare ogni tabella sui registri quando si passa da un processo all'altro.
- Per tabelle più grandi, di dimensioni  $2^{20}$  non è possibile memorizzarla in registri e la page table quindi viene memorizzata in **memoria principale**. In questo caso abbiamo:
  - **Page-table base register (PTBR)**: registro che contiene un puntatore che punta alla locazione di base in cui è memorizzata la page table in memoria principale
  - **Page-table length register (PTLR)**: che indica la dimensione della page table
- Il **context switch è veloce**, basta cambiare il PTBR. L'accesso alla memoria però è lento perché ogni istruzione di load/store di dati/istruzioni richiede **2 accessi in memoria**: un accesso per individuare e consultare la page table e recuperare l'indirizzo fisico e un accesso per individuare le corrispondenti locazioni fisiche di dati/istruzioni e fare il load/store.
- Il problema dei 2 accessi in memoria si risolve utilizzando **memorie cache speciali, ossia registri associativi detti translation look-aside buffers (TLBs)** i quali cercano di velocizzare le operazioni. Cercano di associare indice di pagina al numero di frame, memorizzando gli indici di pagine a cui si fa accesso più frequentemente. Quindi in questi registri viene salvato un **sottoinsieme** della tabella delle pagine a cui si può accedere rapidamente. Se si consulta questi registri e non si trova l'indice di pagina voluto (miss) allora bisogna consultare la tabella delle pagine memorizzata in memoria principale.
- I TLBs sono registri formati da coppie chiave-valore e devono essere di dimensioni contenute e con accesso veloce (da 64 a 1024 entry).

## ❖ Memoria associativa (TLB): implementazione e funzionamento

Per la traduzione di un indirizzo logico in fisico si consulta prima il sottoinsieme della tabella delle pagine contenuto in memoria associativa, poi se l'indirizzo non è presente si trova nella tabella memorizzata in memoria principale.

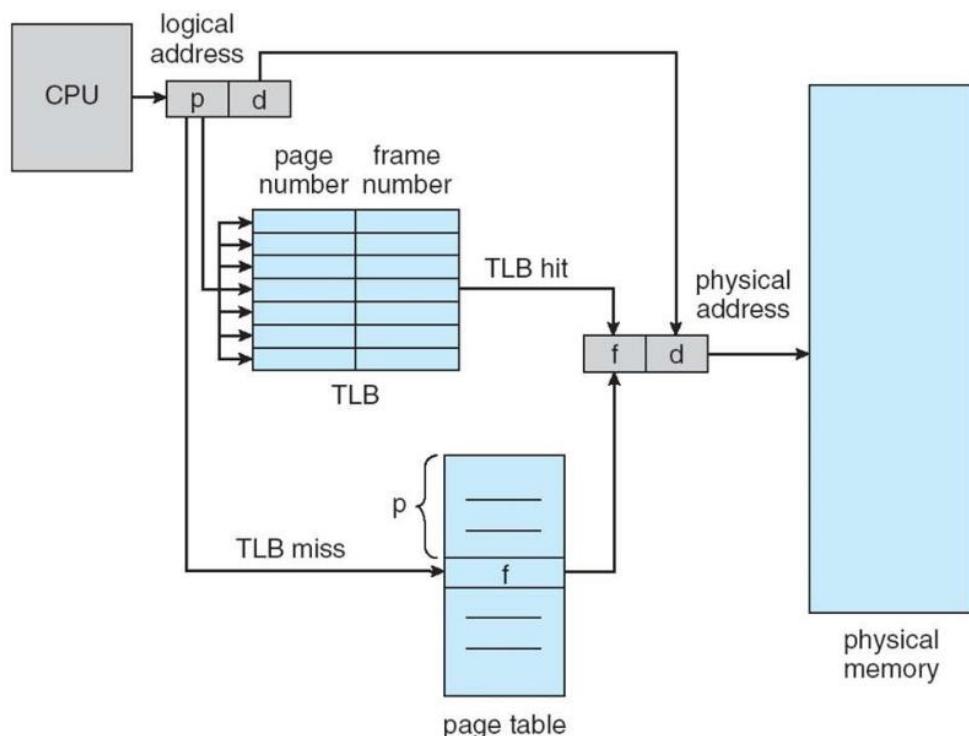
### □ Memoria associativa – ricerca parallela

Page #	Frame #

La **traduzione di un indirizzo (p, d)** avviene in questo modo:

- Se p è in un registro associativo c'è un **TLB hit** e si genera il frame # in output in modo molto veloce.
- Altrimenti c'è un **(TLB miss)** e si deve recuperare il frame # corrispondente dalla page table in memoria. In questo caso 2 accessi in memoria. La nuova coppia indirizzo logico-frame trovata può essere aggiunta nei registri associativi.

Hardware necessario per implementare il paging con **Translation Look-aside Buffers (TLBs)**



- Ogni volta che non si trova il valore in TLB e quindi si verifica un (**TLB miss**), il valore è recuperato e caricato sulla TLB per un accesso veloce la volta successiva. Si suppone che gli ultimi accessi nel passato saranno quelli più probabili nel prossimo futuro.
- Se la TLB è **piena** occorrono **politiche di rimpiazzo**:
  - **Least recently used (LRU)** ossia si sostituiscono quelle parti della tabella poco utilizzate nel passato recente
  - **Round Robin**: si eliminano le parti che stanno da più tempo
  - **Random**
- Alcune entry della TLB non possono essere rimpiazzate perché sono **cablate** per consentire l'accesso permanente
  - un esempio è il codice del kernel rilevante
- Alcune TLB hanno **address-space identifiers (ASIDs)** per ogni entry, ossia identificatori dello spazio di memoria occupato da un processo. In questo modo è possibile mantenere informazioni su diversi processi contemporaneamente evitando di fare lo svuotamento (flush) della tabella per ogni processo nel momento in cui si verifica un context switch. Quindi non abbiamo più una tabella per ogni processo ma una tabella unica con più memoria nella quale possiamo contenere più pagine appartenenti a diversi processi, individuati grazie all'ASIDs. L'ASID fornisce anche un meccanismo di protezione dello spazio degli indirizzi di un processo: quando si risolve l'indirizzo si verifica se il processo corrente corrisponde a quello indicato dall'ASID, se non corrisponde si considera un TLB miss.

### Tempo di accesso effettivo (EAT - Effective Access Time)

- **Assumiamo un unità di tempo la ricerca in TLB detta  $\epsilon$ .** Può essere < 10% del tempo di accesso in memoria, infatti è molto veloce.
- **Assumiamo una hit ratio,  $\alpha$**  ossia la percentuale di volte che si trova la pagina in TLB (dipende dal numero di registri). 80% significa che 8 volte su 10 quando si cerca una pagina in TLB, essa si trova e quindi si verifica un TLB Hit; ciò significa anche che il 20% delle volte ossia 2 volte su 10 avremo un TLB Miss e dovremo fare 2 accessi in memoria.

### Esempio di EAT

Hit Ratio  $\alpha = 80\%$

$\epsilon = 20 \text{ ns}$  per una ricerca in TLB

$100 \text{ ns}$  per una ricerca in memoria.

$$\begin{aligned} \text{EAT} &= (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha) \\ &= 2 + \epsilon - \alpha \end{aligned}$$

**Spiegazione formula:** significa che per prelevare un dato ci vuole 1 accesso in memoria, se la via è breve sommiamo il tempo  $\epsilon$  perché lo troviamo in TLB e moltiplichiamo per  $\alpha$  perché sappiamo si verifica solo l'80% delle volte. Nel momento in cui si verifichi un fallimento dobbiamo fare 2 accessi in memoria, quindi (1 in TLB +1 in memoria +  $\epsilon$  per la ricerca in TLB =  $2 + \epsilon$ ) e lo moltiplichiamo per il miss rate ossia  $1 - \alpha$ . Sviluppando otteniamo la formula finale:  $2 + \epsilon - \alpha$

- Se  $\alpha = 80\%$ ,  $\epsilon = 20\text{ns}$  per la ricerca in TLB search,  $100\text{ns}$  per accesso in memoria ...  $\text{EAT} = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns}$
- Con un più realistico hit ratio ->  $\alpha = 99\%$ ,  $\epsilon = 20\text{ns}$  per la ricerca in TLB,  $100\text{ns}$  per accesso in memoria ...  $\text{EAT} = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns}$
- **Nei sistemi moderni il calcolo è più complicato il calcolo perché sono implementati più livelli di TLB gerarchici.** Le TLB sono elementi hardware a supporto del paging.

### Protezione della memoria tramite la TLB

La protezione della memoria è implementata associando dei bit di protezione per ogni blocco per indicare i permessi (read-only, read-write). Mentre si cerca il frame si può controllare l'accesso e i permessi. Una violazione dei permessi provoca un trap hardware.

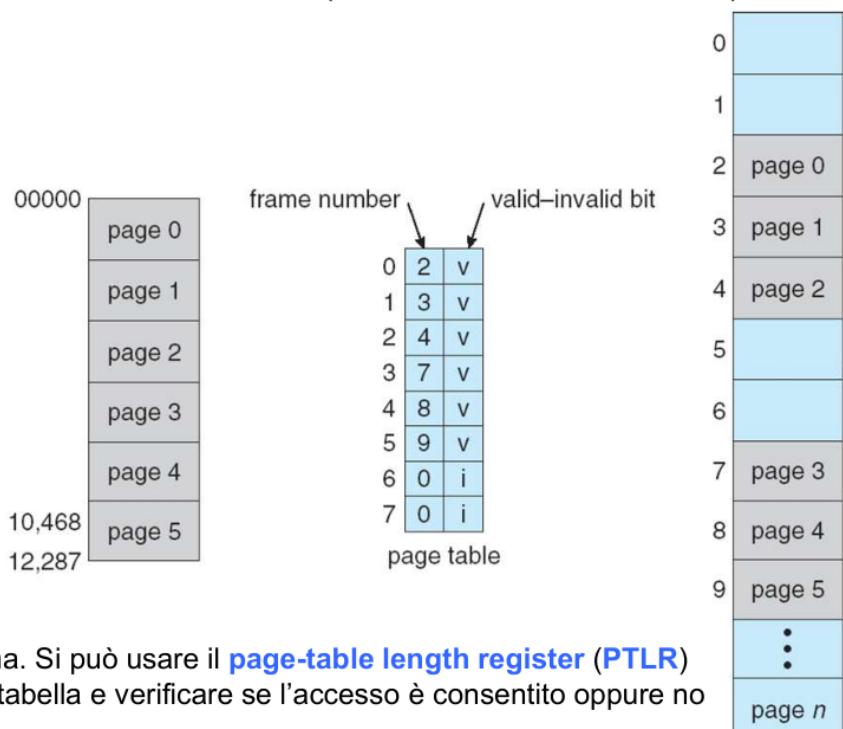
Alle entry della page table sono associate anche **bit valid-invalid**:

- “**valid**” indicata che la pagina nello spazio logico del processo è valida, quindi legale
- “**invalid**” indica che la pagina non è nello spazio logico del processo

Ogni violazione provoca un trap al kernel. Il SO usa questi bit per permettere o vietare l'accesso alle pagine. In particolare il bit valid-invalid si usa **quando manca l'ASID (address-space identifier)** del processo.

Es. 14-bit address space [0,16383], processo solo in [0, 10468]  
Pagine di 2Kb, quindi le pagine 6, 7 non sono valide

Però pagina 5 parzialmente accessibile (frammentazione interna)



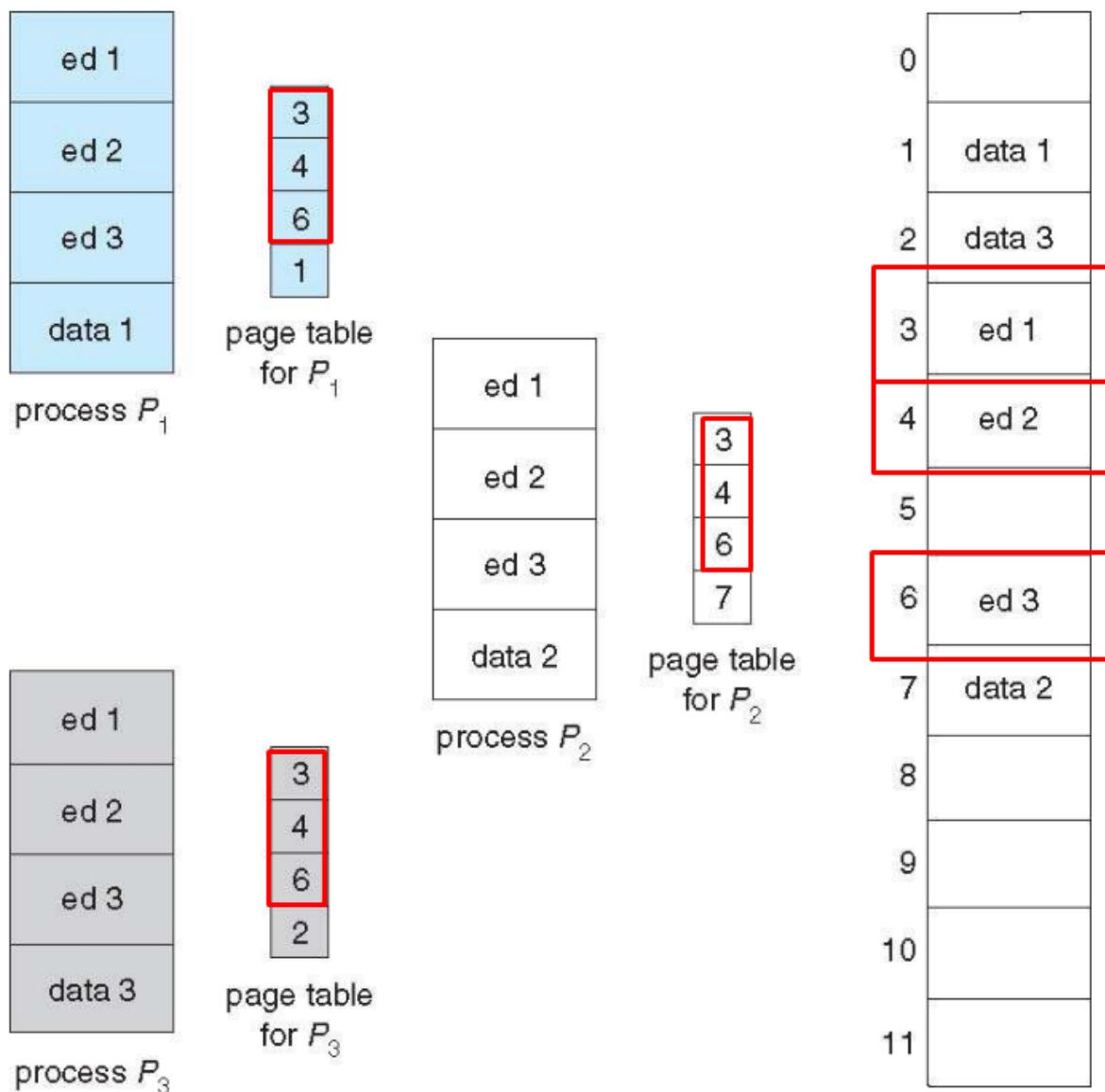
La granularità è al livello di pagina. Si può usare il **page-table length register (PTLR)** per indicare la dimensione della tabella e verificare se l'accesso è consentito oppure no  
*Operating System Concepts – 10<sup>th</sup> Edition*

PTLR serve per controllare se la pagina rispetta la sua grandezza fissa

## ❖ Pagine condivise

Un vantaggio del paging è la possibilità di condividere pagine fra processi. Siccome ogni processo ha la sua page table, più processi possono riferirsi a pagine di altri processi. Non è detto che una pagina venga assegnata in modo esclusivo ad un processo, ma può essere anche condivisa tra più processi, ad esempio per sezioni di codice ready only. Ciò è particolarmente vantaggioso in architetture multiutente.

- **Codice condiviso:** una copia di codice read-only (rientrante) condivisa tra processi (i.e., text editor, compilatori, librerie) evitando duplicazioni. Due o più processi possono condividere lo stesso codice. Utile anche per interprocess communication in cui più processi insistono su aree di memoria (pagine logiche) condivise.
- **Codice e dati privati:** ogni processo mantiene comunque una copia separata del codice e dei dati. Il layout di memoria di un processo è composto da più pagine logiche che vengono messe in corrispondenza in memoria fisica dalla tabella delle pagine.



## ❖ Struttura della tabella delle pagine

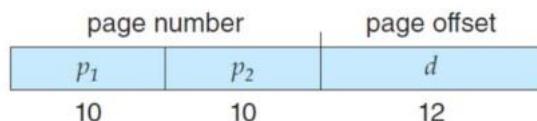
I moderni elaboratori supportano spazi di indirizzi logici molto estesi. La struttura per la paginazione può diventare enorme, ci sarebbero milioni di entry. Non possiamo usare neanché pagine di dimensioni giganti perché non avrebbe senso. Per gestire milioni di entry inoltre avremmo bisogno di strutture dati enormi e molto costose per quanto riguarda la memorizzazione.

### Metodi per supportare spazi di indirizzi logici estesi:

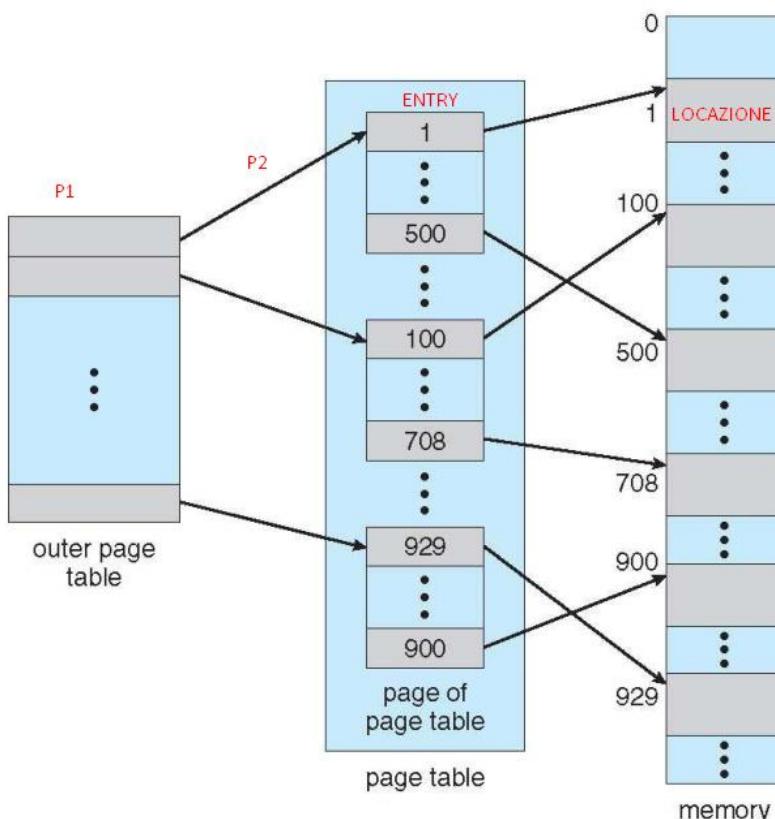
- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

## ❖ Hierarchical Paging

La tabella delle pagine viene resa gerarchica ossia si suddivide lo **spazio degli indirizzi logici su più page tables**. Una tecnica consiste nel suddividere la tabella delle pagine su più livelli, quindi si **pagina la tabella delle pagine**.



Il page number dell'indirizzo logico viene diviso in 2 ulteriori page numbers. Il primo page number serve per stabilire a quale sottotabella delle pagine ci andiamo a riferire, il secondo per trovare la corrispondente pagina logica da mappare all'interno di quella sottotabella e l'offset per mappare il frame corrispondente alla pagina logica alla locazione di memoria fisica giusta.



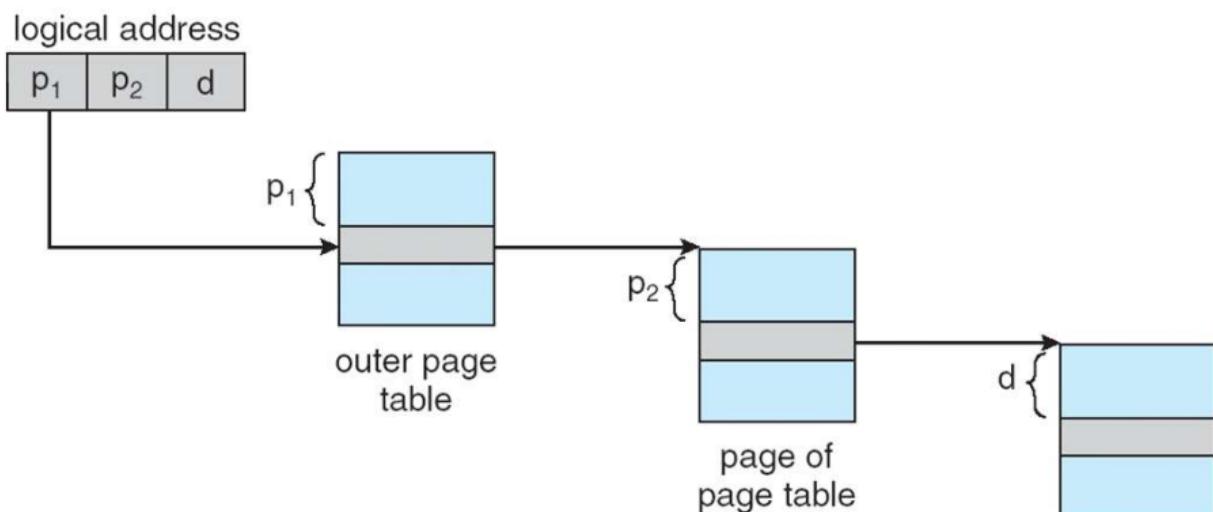
# Esempio Paginazione a due Livelli

- Un indirizzo logico (su macchina a 32-bit con pagine da 1KB) diviso in:
  - page number di 22 bit
  - page offset di 10 bit
- La tabella delle pagine è paginata e il page number è ancora suddiviso in:
  - 12-bit page number
  - 10-bit page offset
- L'indirizzo logico è quindi:

page number	page offset
$p_1$	$p_2$
12	10   10

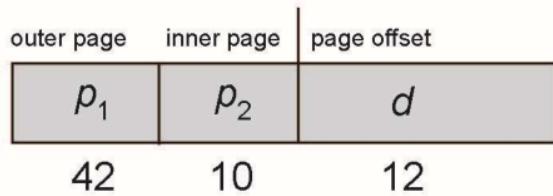
- dove  $p_1$  è indice della tabella esterna,  $p_2$  è l'offset nella pagina della tabella delle pagine esterna
- Metodo chiamato **forward-mapped page table**

Metodo chiamato **forward-mapped page table**

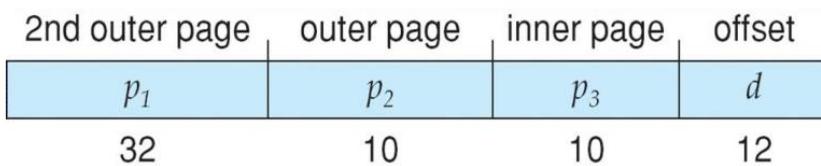
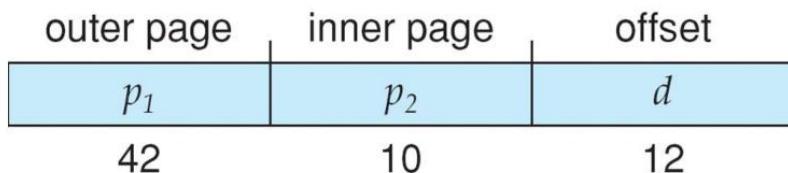


# 64-bit Logical Address Space

- Ma anche lo schema di paging a due livelli non è sufficiente
- Se la dimensione delle pagine fosse 4 KB ( $2^{12}$ )
  - La page table avrebbe  $2^{52}$  entry
  - Con lo schema a due livelli e pagine interne di  $2^{10}$  con entry di 4-byte
  - L'indirizzo sarebbe



- La tabella esterna avrebbe  $2^{42}$  entry
- Una soluzione prevede l'aggiunta di una seconda tabella esterna
- Ad esempio una seconda tabella esterna di  $2^{34}$  byte
  - ▶ Può portare fino a 4 accessi in memoria per un singolo accesso



- **Troppi accessi in memoria, 3 per recuperare il frame + 1 per quello in memoria secondaria**
- **Più le pagine sono grandi e più avremo bit per l'offset**
- **Frame piccoli porta a overhead per trovarli**

## ❖ Hashed Page Table

L'uso è fatto tipicamente su architetture con un spazio di indirizzi > 32 bits. Il **numero di pagina virtuale viene messo in una hash page table** ossia una struttura dati che è formata da campi chiave-valore e abbiamo a disposizione una funzione di hash che a partire dalla chiave consente di ricostruire il valore. È possibile creare un rapporto tra una chiave e tanti valori. **La chiave è rappresentata dal numero di pagina e l'hash function indica una locazione su cui vanno ad insistere più dati. Non è completamente associativa, perché una volta posizionati su una locazione vanno ricercati i dati.** Il dato anziché essere memorizzato su una tabella viene memorizzato su una tavola di hash che consente di raggruppare e ricercare i dati in modo più efficiente con un accesso rapido. La tabella di hash ci permette di compattare i dati.

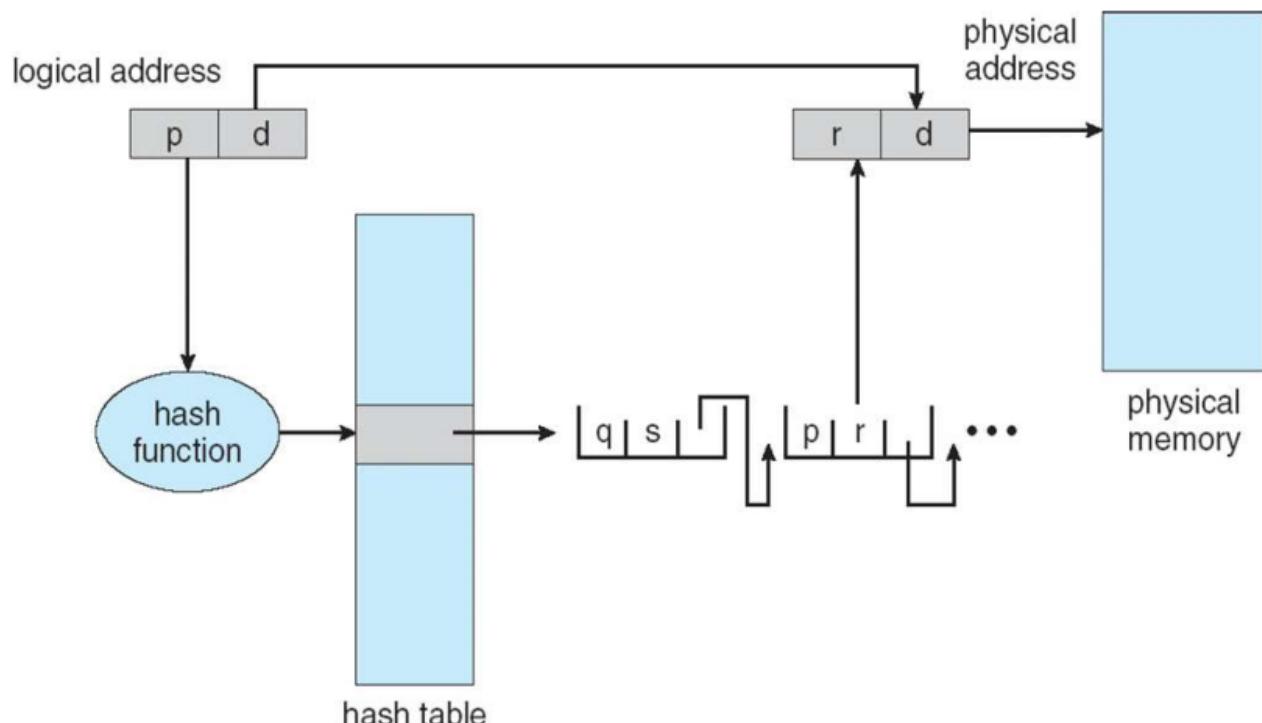
La page table contiene una catena di elementi in hash sulla stessa locazione. Ogni elemento contiene:

- (1) il numero di pagina virtuale
- (2) il valore del page frame mappato
- (3) un puntatore all'elemento successivo

I numeri di pagina virtuale sono confrontati nella catena in cerca di un match.

Se un match è trovato viene estratto il frame fisico corrispondente.

Una variazione per indirizzi a 64-bit sono le **clustered page tables** che sono simili alle hashed ma ogni entry nella hash si riferisce a molte pagine ossia un cluster di pagine (16) invece di una sola



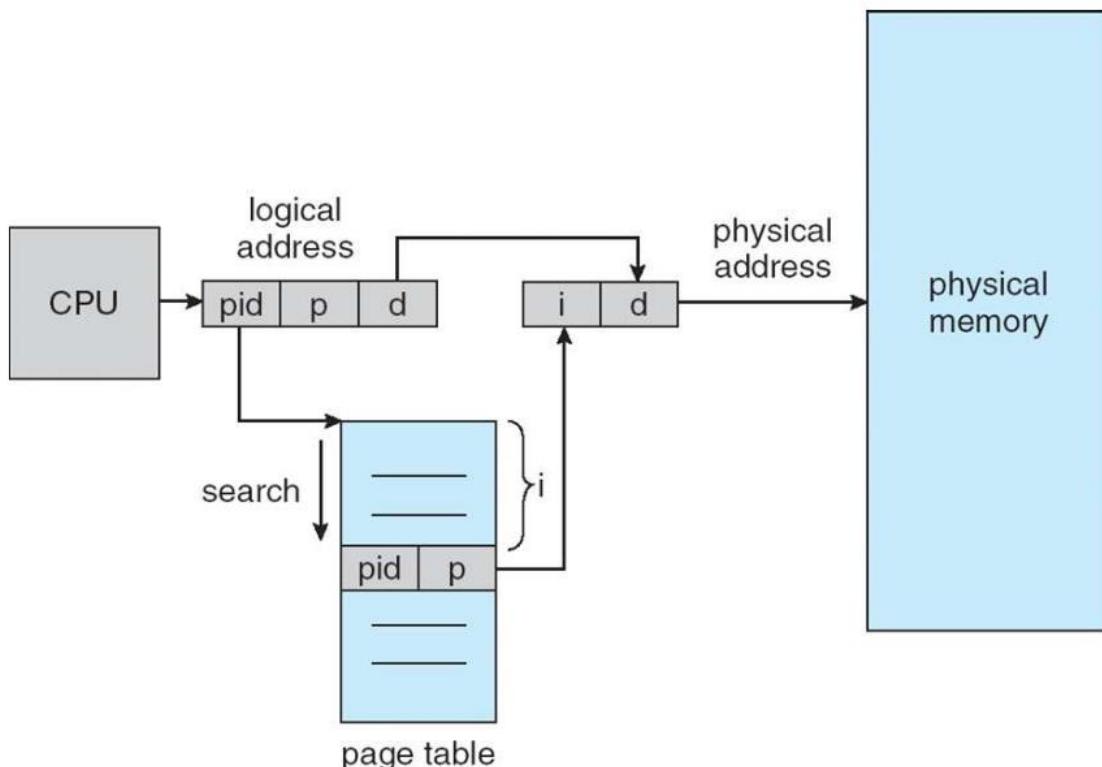
## ❖ Tabella delle pagine invertite

Invece di avere ogni processo con una page table e tener traccia di tutte le pagine logiche, con la tabella delle pagine invertite si tracciano le pagine fisiche. **Si crea una tabella frame-based che tiene traccia di quali frame della memoria principale sono assegnati ad un processo.** Dai frame si arriva alle pagine logiche del processo.

- Una entry per ogni frame di memoria
- Le entry della tabella consistono dell'indirizzo virtuale della pagina contenuta in memoria reale con informazione sul processo che possiede quella pagina

<process-id, page-number, offset>

Minore memoria per le tabelle delle pagine, ma aumenta il tempo necessario per cercare la tabella quando c'è un riferimento ad una pagina logica.



- Si aggiunge il PID del processo. Dato il PID del processo si va a ricercare il PID su questa tabella delle pagine contenente i frame. Viene recuperato l'iesimo frame del processo e tramite l'offset dell'indirizzo logico su cui lavora il processo, viene costruito l'indirizzo fisico per accedere ad una particolare locazione di memoria. L'overhead è dovuto al costo di ricerca nella tabella del pid del processo. Per velocizzare la ricerca si utilizzano meccanismi di hashing.
- **Più complesso implementare la memoria condivisa, perché i frame sono collegati ai processi tramite PID**

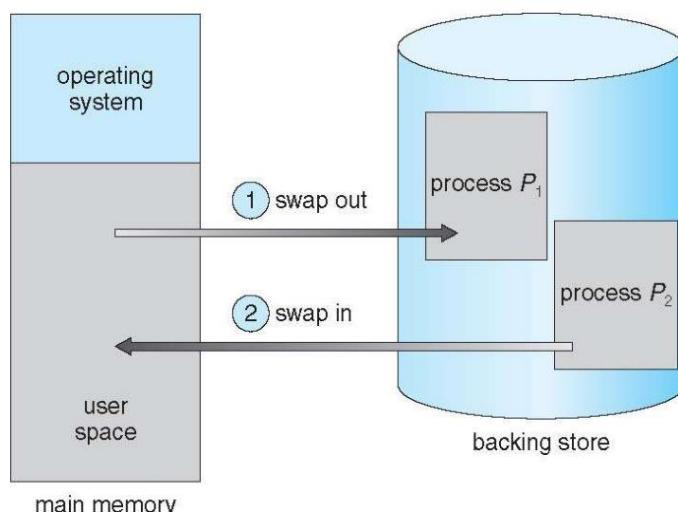
## ❖ Swapping

E' una tecnica che ci permette di estendere l'utilizzo della memoria primaria ed aumentare il grado di multiprogrammazione. **Permette lo scambio di porzioni di un processo temporaneamente dalla memoria primaria a quella secondaria (backing store) e poi riportate alla memoria principale quando servono.**

- **Un processo può essere scambiato (swapped) temporaneamente con memoria ausiliaria (backing store) e poi portato ancora nella memoria principale per continuare l'esecuzione.** La memoria fisica totale occupata dai processi può eccedere la memoria fisica disponibile, perché stocchiamo loro parti temporaneamente nello backing store.
- **Backing store:** memoria secondaria veloce grande abbastanza per accomodare i processi che devono essere stoccati e ripristinati

Quando la memoria principale è tutta occupata e quindi abbiamo un'elevata multiprogrammazione, anziché scegliere di rinunciare all'esecuzione di un processo possiamo scegliere parti di processi poco utilizzate e accantonarle dalla memoria principale al backing store per liberare spazio per l'allocazione di nuovi processi. Le parti poi verranno ripristinate quando serviranno.

- **Swap out, swap in:** swapping di processi. I processi inattivi possono essere selezionati e accantonati nel backing store (**swap out**) e poi ripresi da esso (**swap in**).
- La maggior parte del **tempo di swap** è di **tempo di trasferimento**; il tempo di trasferimento è direttamente proporzionale alla quantità di memoria trasferita (swapped)
- Il sistema mantiene una **ready queue** di processi pronti per l'esecuzione che hanno immagine di memoria su disco e che devono essere ripristinati in memoria principale.
- **Roll out, roll in:** è uno swapping con priorità, processi di bassa priorità sono portati fuori nel backing store (roll out) così i processi a più alta priorità possono essere caricati ed eseguiti (roll in).
- **Il processo portato fuori deve rientrare nello stesso indirizzo fisico?** Dipende dal metodo di binding, se è a tempo di esecuzione il processo può essere riallocato anche in un indirizzo diverso. Bisogna considerare anche gli **I/O pendenti** da/a processi nello spazio di memoria.
- Di solito nei sistemi operativi è disabilitato, si abilita solo in situazioni particolari quando si supera una certa soglia di memoria. **Viene fatto swapping con paging (page-in e page-out).**



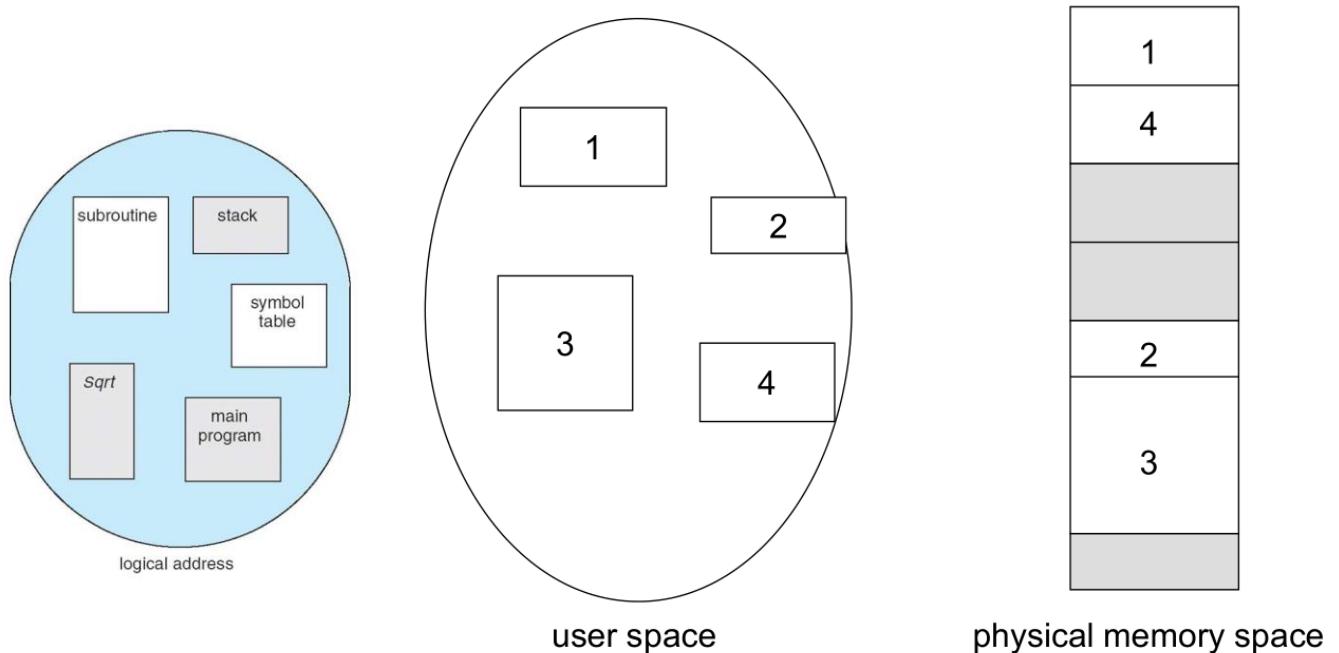
## ❖ Segmentazione

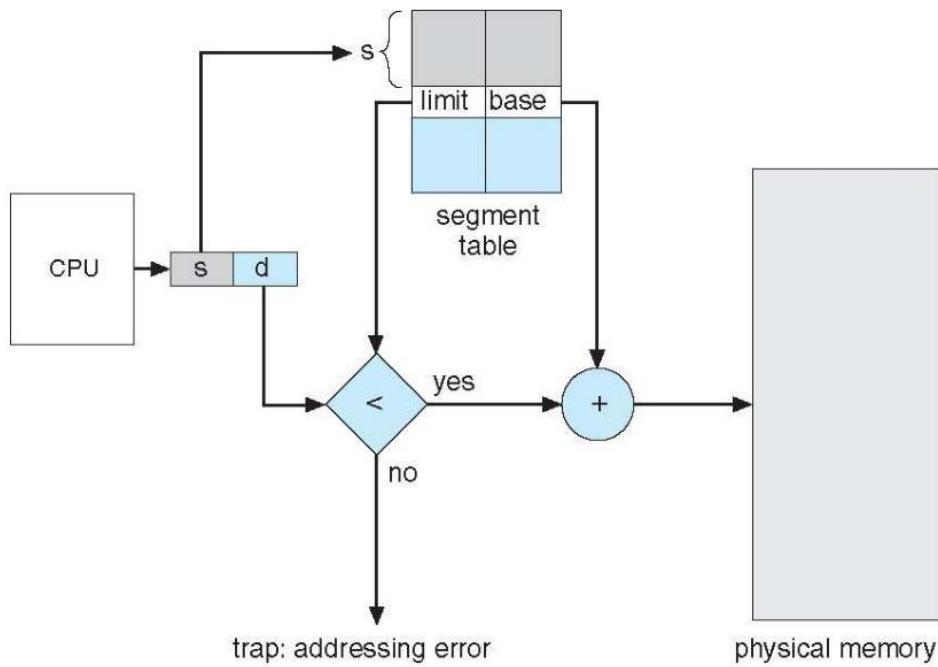
La paginazione separa memoria logica utilizzata da un processo e memoria fisica.

- **La segmentazione è una tecnica di gestione della memoria che supporta la prospettiva utente sulla memoria. La memoria viene vista come insieme di segmenti intesi come sottounità logiche sensate di un programma che si compone appunto di vari segmenti.**

Un programma è visto come una collezioni di segmenti ossia unità logiche come: main program , procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays

- **Tutti i segmenti di un processo sono numerati e mappati in memoria fisica. I blocchi sono di dimensione diverse rispetto alla paginazione in cui abbiamo pagine di dimensione fissa.**
- Gli indirizzi logici sono definiti in questo modo: <**segment-number, offset**> che indica il numero di segmento e lo scostamento al suo interno per individuare una certa locazione.
- E' necessaria una struttura che mappi gli indirizzi logici segmentati in indirizzi fisici di memoria: la **tabella dei segmenti**.
- Ogni **entry** della tabella dei segmenti ha:
  - **base** – contiene l'indirizzo fisico di partenza del segmento a cui poi viene aggiunto l'offset se non supera il limite di lunghezza del segmento
  - **limite** – specifica la lunghezza del segmento, confrontato con l'offset per verificare se è consentito l'accesso altrimenti si verifica un trap hardware.
- La memoria allocata per il processo non deve essere contigua, ma **comporta frammentazione esterna**



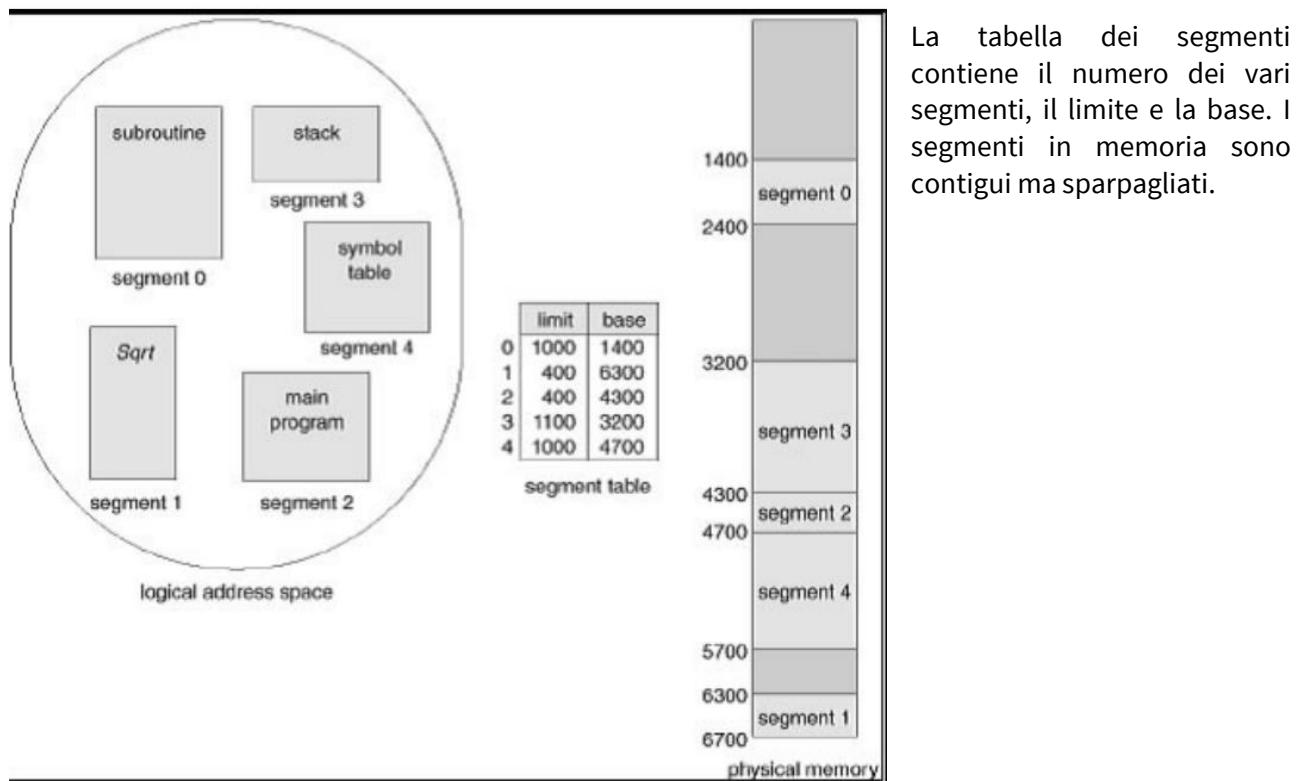


### Architettura di Segmentazione

La tabella dei segmenti non può essere tenuta in registri perché ha molte entry, quindi viene memorizzata in memoria principale.

- **Segment-table base register (STBR)** indica la locazione di memoria della tabella dei segmenti
- **Segment-table length register (STLR)** indica il numero dei segmenti usati da un program
  - Dato indirizzo logico (segment,d), il numero s è legale se  $s < \text{STLR}$  per quel processo.

Si utilizzano registri associativi per limitare i due accessi in memoria.



## Funzionalità aggiuntive che si possono aggiungere alla tabella dei segmenti

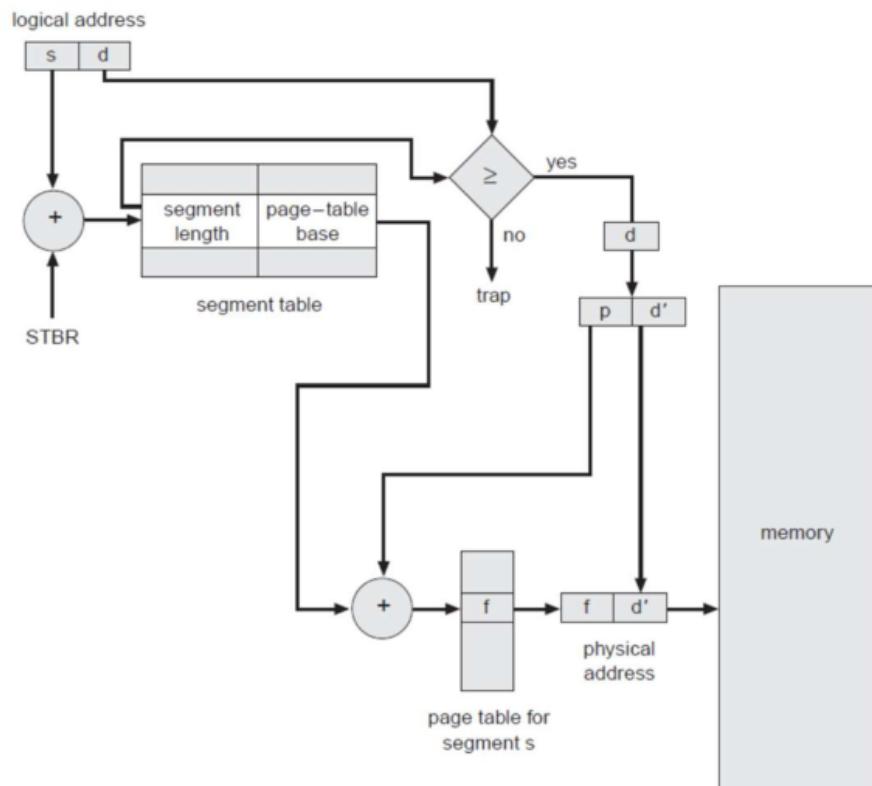
- **Protezione** (stesso segmento, stessa protezione). Ogni entry nella tabella dei segmenti è associata a un **validation bit**. Se bit = 0 il segmento è illegale. Si può associare al contenuto di ogni segmento privilegi read/write/execute.
- **Possono essere condivisi i segmenti tra processi:** la condivisione del codice avviene al livello di segmento. Come per il paging una copia di un programma condiviso (es. editor di testo). **Più programmi possono dover fare riferimento allo stesso numero di frammento e quindi vengono condivisi. Problemi:** porzioni di codice con riferimento diretto a sé stesse.
- **Problema del doppio accesso come per paginazione**
- I segmenti di un processo, come per le pagine, non sono necessariamente contigui e sono segmenti di dimensione diversa quindi problemi di frammentazione esterna
- Si possono fondere i metodi di paginazione e segmentazione: **segmentazione paginata**

### □ Tabella dei segmenti:

- ▶ lunghezza segmento e base della tabella

### □ Tabella delle pagine per il segmento s

- ▶  $\text{base tab} + p = \text{base pagina}$



## **8. MEMORIA VIRTUALE**

La memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica. Inoltre la memoria virtuale permette un'astrazione della memoria centrale che viene vista come un vettore molto grande e uniforme, come è vista logicamente dall'utente. Questa tecnica libera i programmatori dai problemi di limitazione della memoria. La memoria virtuale permette inoltre ai processi di condividere facilmente file e facilita la condivisione di dati tra processi. La memoria virtuale è però difficile da realizzare. Se è usata scorrettamente, può ridurre di molto le prestazioni del sistema.

### **❖ Concetti principali**

- **La memoria virtuale è una tecnica che consente di vedere la memoria principale più grande dal punto di vista logico di quello che non è dal punto di vista fisico.**

Un processo deve contenere il suo codice in memoria memorizzato per essere eseguito. Non è sempre necessario che tutto il suo codice e tutto il suo layout venga memorizzato contemporaneamente in memoria principale, dato che spesso ci sono parti inutilizzate. Se è possibile eseguire programmi parzialmente caricati in memoria, con le altre parti memorizzate in un backing store, allora abbiamo notevoli **vantaggi**:

- **i programmi non sono più vincolati dalla dimensione della memoria fisica;** si porta in memoria principale ciò che realmente ci serve per effettuare una computazione in un certo istante di tempo.
- ogni programma occupa **meno memoria a run-time** e quindi **più programmi** possono essere eseguiti allo stesso tempo. Da questo punto di vista c'è un miglior utilizzo della CPU e si aumenta il throughput.
- sono necessarie meno istruzioni di I/O per caricare o swappare programmi in memoria.

**La memoria Virtuale effettua una netta separazione della memoria logica dalla memoria fisica.** Dal punto di vista logico si può pensare che i programmi siano interamente caricati in memoria, ma dal punto di vista fisico sono parzialmente in memoria principale e parzialmente nella memoria secondaria. Solo una parte del programma durante l'esecuzione sarà caricata in memoria.

- **grazie alla tecnica della memoria virtuale possiamo avere l'illusione che lo spazio degli indirizzi logici sia più grande dello spazio degli indirizzi fisici perché la memoria che "manca" la recuperiamo grazie all'estensione nel backing store, in cui vengono memorizzate le pagine di un processo meno utilizzate.**
- Spazio indirizzi condiviso da molti processi
- Creazione di processi più efficiente perché allochiamo solo la memoria minima necessaria per la loro computazione.

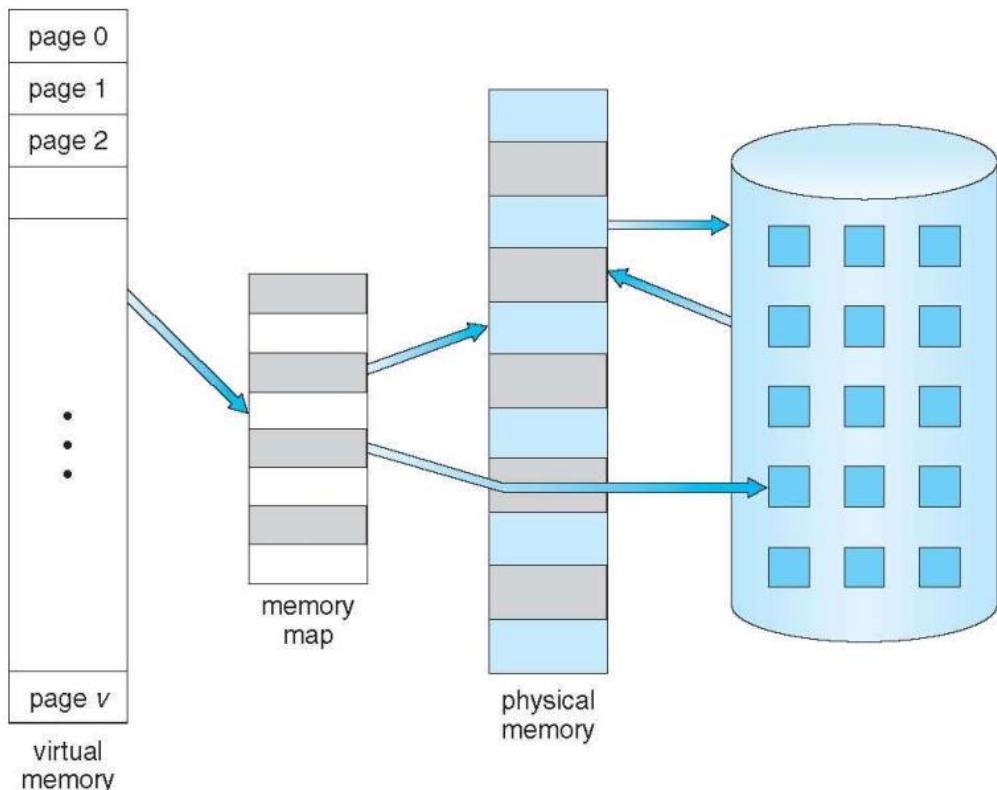
## ❖ Spazio degli indirizzi virtuali

Lo spazio degli indirizzi virtuali è una vista logica su come il processo è contenuto in memoria. Esso inizia da un indirizzo (es. indirizzo 0) che poi viene rimappato in memoria fisica. Gli indirizzi sono visti come logicamente contigui dal processo, fino alla fine dello spazio. Come si è già detto la memoria fisica è organizzata in frame si lascia alla MMU il processo di mappatura di un indirizzo logico in uno fisico.

La memoria virtuale viene implementata mediante 2 tecniche:

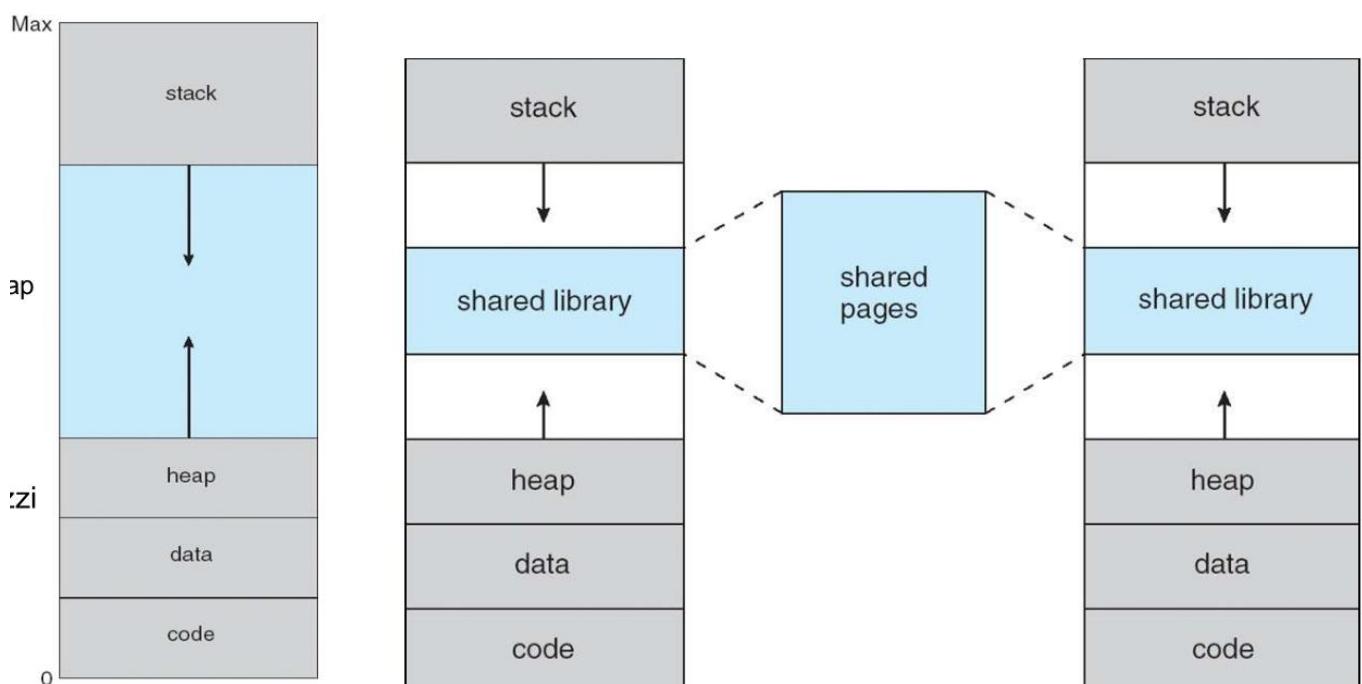
- Demand paging
- Demand segmentation

Schema che mostra memoria virtuale più grande di quella fisica



La memoria logica è più grande di quella fisica appunto perché alcune pagine vengono caricate nel backing store e poi recuperate quando se ne ha bisogno. L'operazione di caricamento della pagina in memoria fisica deve essere quanto più ottimizzata possibile: il sistema operativo deve fare in modo di prevedere quali sono le prossime pagine da caricare per un processo e se ne manca una deve essere recuperata dal backing store e caricarla. Si fa lo **sawp in** di frame dal backing store alla memoria fisica o lo **swap out** nel caso in cui la memoria fisica sia piena, cercando di fare lo swap out dei frame meno usati.

- Lo spazio degli indirizzi virtuali di un processo include solitamente uno strack, un heap, dati e codice. Lo stack inizia dall'indirizzo logico massimo e cresce “in giù”. L'Heap cresce “in su”. Abbiamo quindi uno spazio di indirizzi “non usato” che è un buco: non è necessario finché lo stack o l'heap non crescono. La memoria logica dunque si popola dinamicamente. Ciò si traduce nella memoria secondaria come uno spazio di indirizzi sparso con buchi lasciati per crescita, **dynamically linked libraries**, etc.
- Inoltre nello spazio degli indirizzi virtuali di solito capita che vengano **condivise librerie di sistema** tra i processi. Si crea una sorta di **shared memory mappando pagine do read-write nel virtual address space**.
- Le pagine possono essere condivise tra due processi nello spazio di indirizzamento virtuale attraverso chiamate di sistema come la **virtual fork()** che è una fork veloce la quale utilizza la memoria del processo padre che viene sospeso, si fa finta che è il figlio perché si deve chiamare una chiamata **exec**. Si crea un figlio logico senza che vi sia una duplicazione di memoria per il figlio.



## ❖ Demanding paging

Abbiamo detto che si può memorizzare l'intero processo in memoria a tempo di caricamento oppure nel caso più efficiente **portare soltanto una pagina in memoria quando è necessario**, in questo modo ne consegue che avremo:

- Meno operazioni di I/O
- Minor memoria richiesta
- una risposta più rapida
- maggior multiprogrammazione

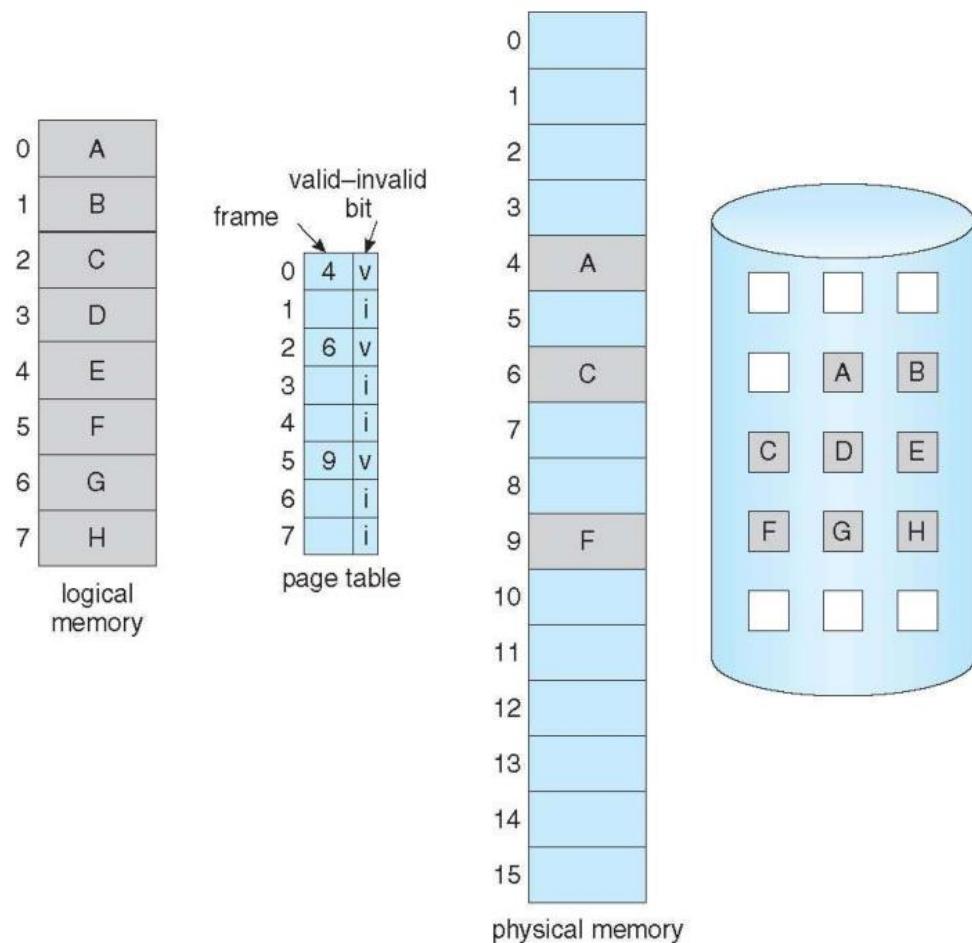
Tutto ciò grazie a tecniche come il demanding paging.

- **Il demanding paging è una tecnica che consiste nel caricare in memoria principale le pagine di un processo solo nel momento in cui servono realmente; è una tecnica comunemente adottata dai sistemi che implementano la memoria virtuale.**
- **Le pagine sono quindi caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine a cui non si accede mai non sono mai caricate nella memoria fisica e rimangono esclusivamente nel backing store (lazy swapper).**
- Simile al paging con swapping.
- Quando viene richiesta una pagina si cerca il riferimento ad essa. Se il riferimento ad essa non c'è vi è un abort, se il riferimento ad esso è presente ma non è in memoria, si fa uno **swap in** dal backing store alla memoria principale caricandola. Se la memoria principale è tutta occupata, per caricare una pagina richiesta è necessario uno **swap out** dalla memoria principale, selezionando una pagina di un altro processo come vittima sacrificale da caricare nel backing store.
- **Se una pagina di un processo in esecuzione non viene mai richiesta, essa non viene caricata in memoria. Questo comportamento viene definito Lazy swapper.**
- Il **pager** gestisce le singole pagine dei processi. Porta in memoria solo le pagine richieste facendo lo **swap in** ed inoltre **anticipa** quali pagine verranno usate da un processo prima che quelle non più utilizzate vengano portate fuori dalla memoria fisica tramite lo **swap out**.
  - Se la pagina richiesta è già residente in memoria, si fa un accesso in memoria e si cerca sulla tabella delle pagine fino a trovarla.
  - Se la pagina richiesta non è residente in memoria principale o nella tabella delle pagine, dato che si trova in quella virtuale, si genera un trap detto **page fault** che va gestito. Si deve trovare e caricare la pagina nel backing store senza cambiare il comportamento del programma e senza che il programmatore debba cambiare il codice.
- Ogni entry della page table ha associato un **bit valid-invalid**.
  - **valid** significa che la pagina è nella tabella
  - **invalid** significa che la pagina non è nella tabella o che l'accesso è vietato
  - Inizialmente il valid-invalid bit è settato a **i** (invalid) su tutte le entry.
  - viene gestito dalla MMU. Durante la traduzione della MMU se il valid-invalid bit nella page table entry è invalid si verifica un page fault
- **Il primo riferimento ad una pagina non presente nella page table genera un trap all'SO: Trap di page fault che deve essere gestito.**

Frame #	valid-invalid bit
	v
	v
	v
...	i
	i
	i

page table

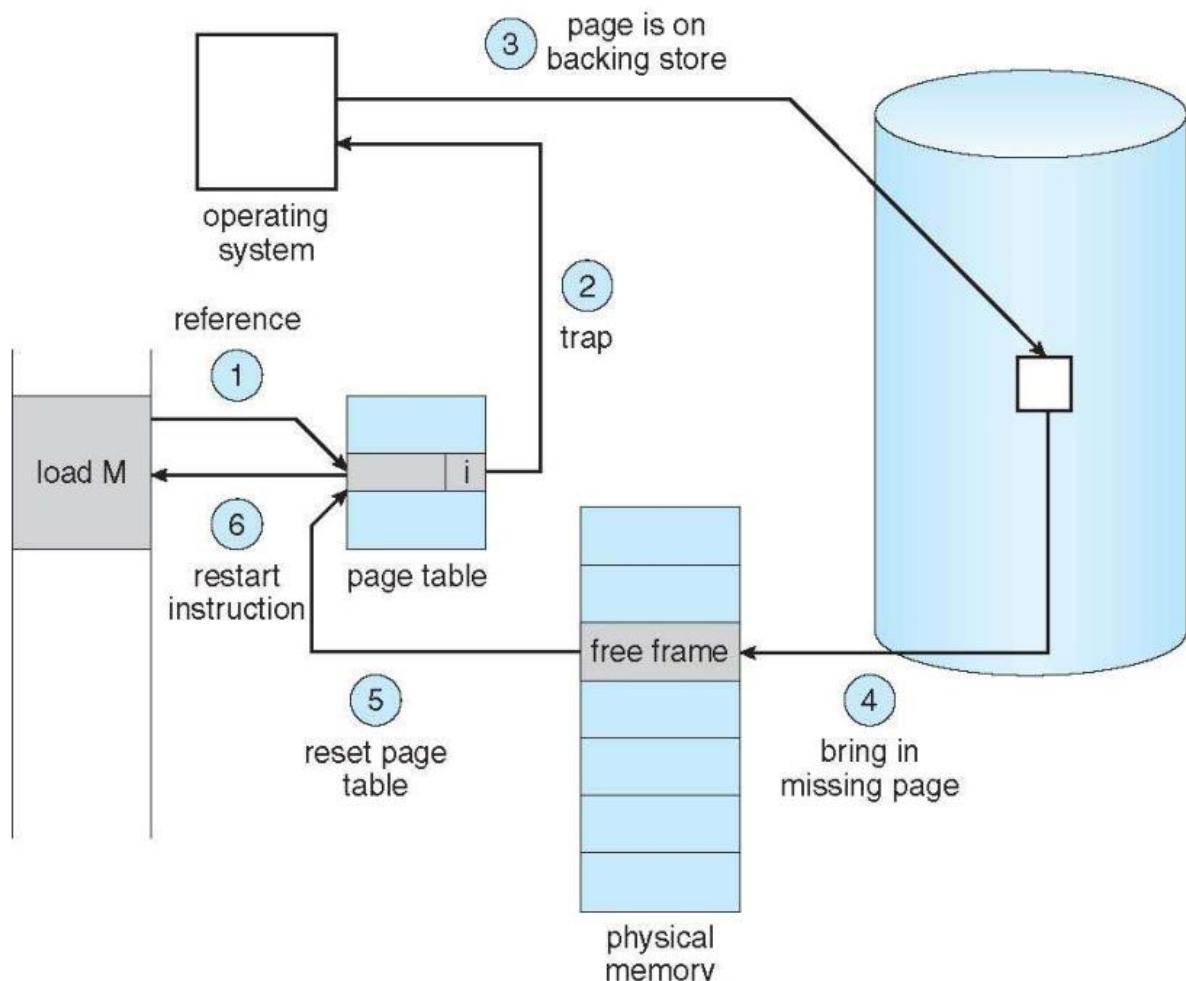
Es. 0 è valido, 1 non è valido, etc.



## ❖ Gestione del page fault

Quando si verifica un page fault inizialmente il S.O. controlla il PCB del processo per verificare le informazioni sulla memoria da allocare a quel processo:

- 1. controlla se il page fault si è verificato perché c'è stato un **riferimento non valido** e quindi c'è un **abort (kill)** del processo.
- 2. altrimenti non è presente semplicemente in memoria e va caricata la pagina. **Quando si deve inserire una pagina nella tabella delle pagine:**
  - 3. innanzitutto bisogna trovare un frame libero in memoria principale (cercando sulla lista dei frame liberi della memoria).
  - 4. A questo punto il s.o. schedula un'operazione di lettura dal disco di quella pagina richiesta e di trasferimento dal backing store della pagina alla memoria principale nella quale viene "scritta".
  - 5. A lettura completata resetta le tabella delle pagine per indicare che la pagina è in memoria e si indica a quale frame ed infine setta il validation bit = v.
  - 6. Riavvia infine l'istruzione che ha causato il page fault perché a questo punto il processo può accedere alla pagina in memoria.



## ❖ Considerazioni sul demanding paging

Nel caso estremo un processo inizia la sua esecuzione senza pagine in memoria. Appena si cerca di caricare nel program counter la prima istruzione si verifica subito un **page fault**. Il sistema operativo dovrà caricare tutte le pagine necessarie per l'esecuzione del processo dal backing store nella tabella delle pagine. Questa tecnica è detta **pure demanding paging**.

**In realtà un'istruzione di un processo può coinvolgere più pagine e se esse non sono presenti in memoria si verificano page fault multipli.** Statisticamente però è poco probabile perché ad indirizzi logici contigui corrispondono quasi sempre stesse pagine, per la località dei riferimenti.

**Per implementare il demanding paging deve essere usato diverso hardware:**

- Page table con valid / invalid bit
- Memoria secondaria (con swap space rapido)

**Restart dell'istruzione che richiede la bufferizzazione di informazioni**

## ❖ Istruction restart

Abbiamo detto che una volta gestito il page fault il sistema operativo riavvia l'istruzione che ha causato il page fault perché a questo punto il processo può accedere alla pagina in memoria di cui aveva bisogno. In realtà l'istruction restart è una delle operazioni più complesse.

- Consideriamo che il page fault può avvenire durante qualunque momento: fetch, decode, execute. Se l'esecuzione di un'istruzione del processo è interrotta durante il prelievo degli operandi, va rifatto il fetch dell'istruzione.
- Si consideri operazione ADD A, B con risultato in C
  - Prelievo e decodifica di ADD
  - Prelievo di A
  - Prelievo di B
  - Addizione A e B
  - Memorizzazione somma in C
- Se interrotta su C, va ripetuta l'operazione dal prelievo
  - Se un blocco di dati è tra due pagine e si verifica un page fault di una pagina non si può fare un rollback delle modifiche. La gestione è più complessa e non si può fare l'istruction restart. Soluzioni:
    - si controlla se prima di fare un'operazione di scrittura si possiedono tutte le pagine richieste
    - si mantiene in registri lo stato precedente di un processo e si ripristina

## ❖ Lista dei frame liberi

Quando c'è un page fault occorre caricare la pagina da memoria secondaria in un frame libero. Questo comporta il fatto che il sistema operativo deve mantenere una **free frame list**.

- **I frame che vengono allocati devono essere rimossi da questa lista:** essi vengono “azzerati” nella free frame list **poco prima della loro allocazione** (zero fill on demand).



## ❖ Performance Demanding Paging

### □ Passi del Demand Paging (caso peggiore)

1. Trap al sistema operativo
2. Salva gli user register e il process state
3. Determina se l'interrupt era un page fault
4. Verifica che il riferimento alla pagina era legale e determina la locazione della pagina su disco
5. Schedula una lettura dal disco ad un frame libero:
  1. Attendi in una coda per il dispositivo finché la richiesta di read è servita
  2. Attendi la ricerca e/o il tempo di latenza
  3. Inizia il trasferimento della pagina al frame libero
6. Mentre attendi alloca la CPU a qualche altro user
7. Ricevi un interrupt dal I/O (I/O completato)
8. Salva i registeri e lo stato del processo dell'altro user
9. Determina che l'interrupt era dal disco
10. Correggi la page table e le altre tabelle per mostrare la pagina in memoria
11. Attendi che la CPU venga di nuovo allocata a questo processo
12. Ripristina gli user register, il process state e la nuova page table, quindi riprendi l'istruzione interrotta

**La gestione è molto costosa! Occorre che ci siano page fault quanto meno possibili. La convenienza della virtual memory si valuta in base a quanto spesso viene effettuato un demanding paging.**

**Page fault rate:** è un parametro P che indica mediamente ogni quanto si verifica un page fault in memoria.  
Vale  $0 \leq P \leq 1$

- Se  $P = 0$  non si verifica mai page fault
- Se  $P = 1$  ad ogni accesso in memoria si verifica un page fault (sempre)

**EAT (Effective Access Time) per il Demanding Paging**

$(1 - p) \times \text{memory access}$

$+ p (\text{page fault time})$

Se  $p$  fosse 0 avremmo solo il memory access. Se  $p$  fosse 1 avremmo solo page fault.

Deve essere molto sporadicamente il page fault altrimenti il tempo di accesso medio in memoria si degrada di molto.

# Esempio Demand Paging

---

- Tempo accesso memoria = 200 nanosecondi
- Tempo medio di servizio del page-fault = 8 millisecondi
- $$\begin{aligned} \text{EAT} &= (1 - p) \times 200 + p (8 \text{ millisecondi}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- Se un accesso su 1000 causa un page fault allora  
EAT = 8.2 microsecondi  
È un slowdown di un fattore di 40!
- Se si vuole una performance degradation < 10 percent
  - $$220 > 200 + 7,999,800 \times p$$
$$20 > 7,999,800 \times p$$
  - $p < .0000025$
  - < una page fault per ogni 400,000 accessi in memoria

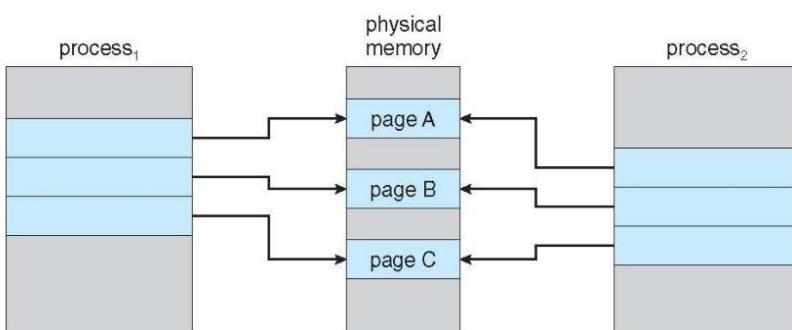
## ❖ Demanding paging: ottimizzazioni

- Le operazioni di I/O sul disco devono essere più veloci nello spazio di swap di quelle per accedere al file system. Per garantire questa velocità lo swap space viene allocato in blocchi di memoria molto grandi con un minor costo di gestione.
- Inoltre il s.o. potrebbe caricare l'immagine del processo sullo swap space a tempo di caricamento, ma è costoso.
- Linux cerca di fare la gestione del demanding paging tramite il file system inizialmente, poi le pagine vanno nello swap space.

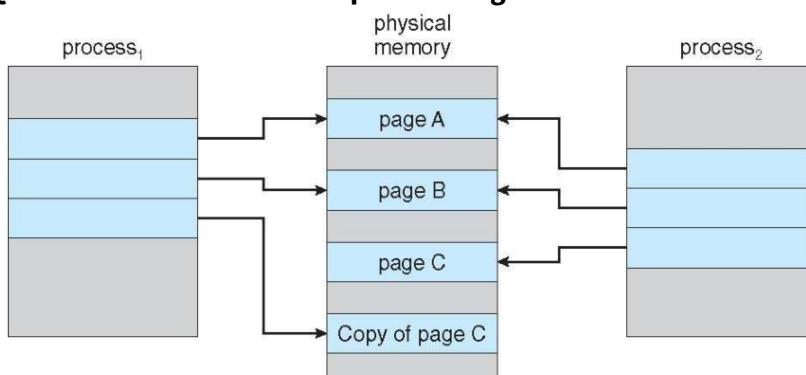
## ❖ Copy on write

E' una tecnica di ottimizzazione usata per limitare le scritture o le letture nello spazio di swap oppure per posticiparle e farle nei momenti più opportuni.

- Nel momento in cui si chiama la system call fork() un processo padre dovrebbe creare un processo figlio, duplicando la memoria del padre per il figlio. In teoria finché il processo figlio non fa delle modifiche può condividere le stesse pagine di memoria del processo padre. Inoltre, se il figlio chiama subito la chiamata exec() non è necessario il duplicato di memoria.
- Se padre e figlio condividono le stesse pagine, nel momento in cui il padre o il figlio fanno una modifica su una pagina condivisa, è necessaria la duplicazione di quella pagina e la sua copia viene caricata in memoria principale.
- **Il Copy-on-Write (COW) permette dopo una fork() ai processi padre e figlio di condividere inizialmente le stesse pagine in memoria. Le pagine condivise sono marcate come pagine copy-on-write. Permette la creazione di un processo in modo più efficiente dal momento che solo le pagine modificate sono copiate e caricate in memoria successivamente.**
- Tecnica usata da Windows, Linux



**Quando c'è una modifica da parte del figlio:**

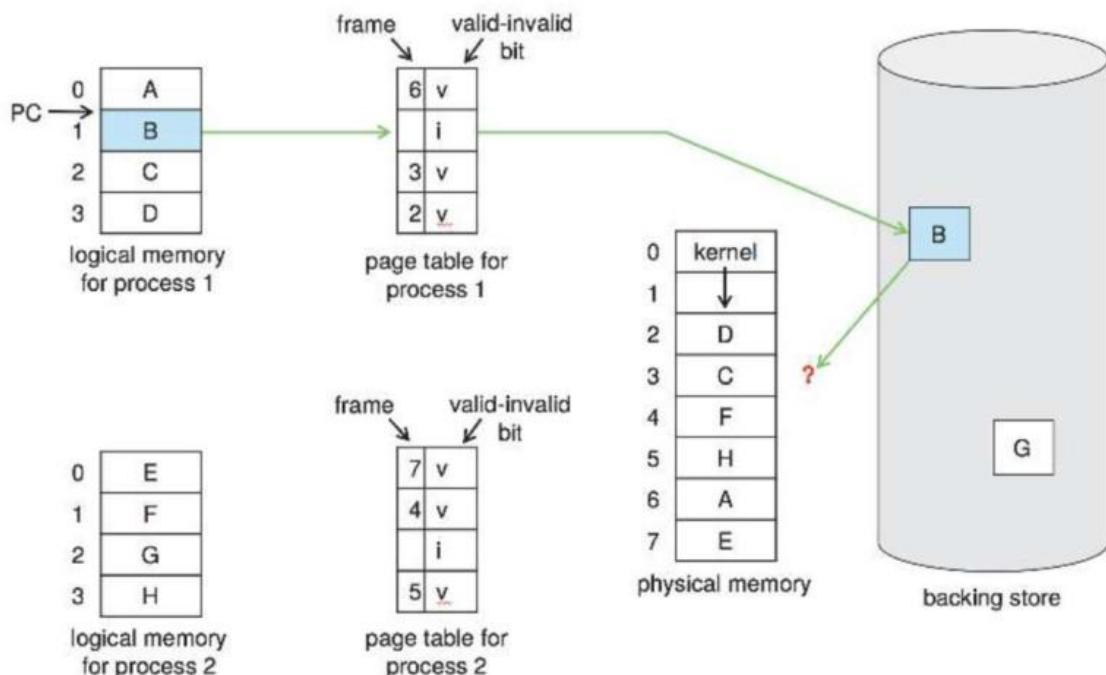


## ❖ Virtual memory fork

La system call `vfork()` può essere considerata una variante di `fork()`. Si usa quando un processo figlio deve diventare esclusivamente veicolo dell'esecuzione del codice di un altro processo, quindi quando il processo figlio invoca subito la chiamata `exec()`. Quindi si sospende il padre e si utilizza immediatamente l'address space del processo padre. In questo caso è possibile scrivere sulla memoria del padre. L'uso corretto di `vfork()` è usare subito dopo la chiamata `exec()` per caricare un'altro programma dirottando il processo, richiamando così altre pagine e non sovrascrivendo quelle del padre.

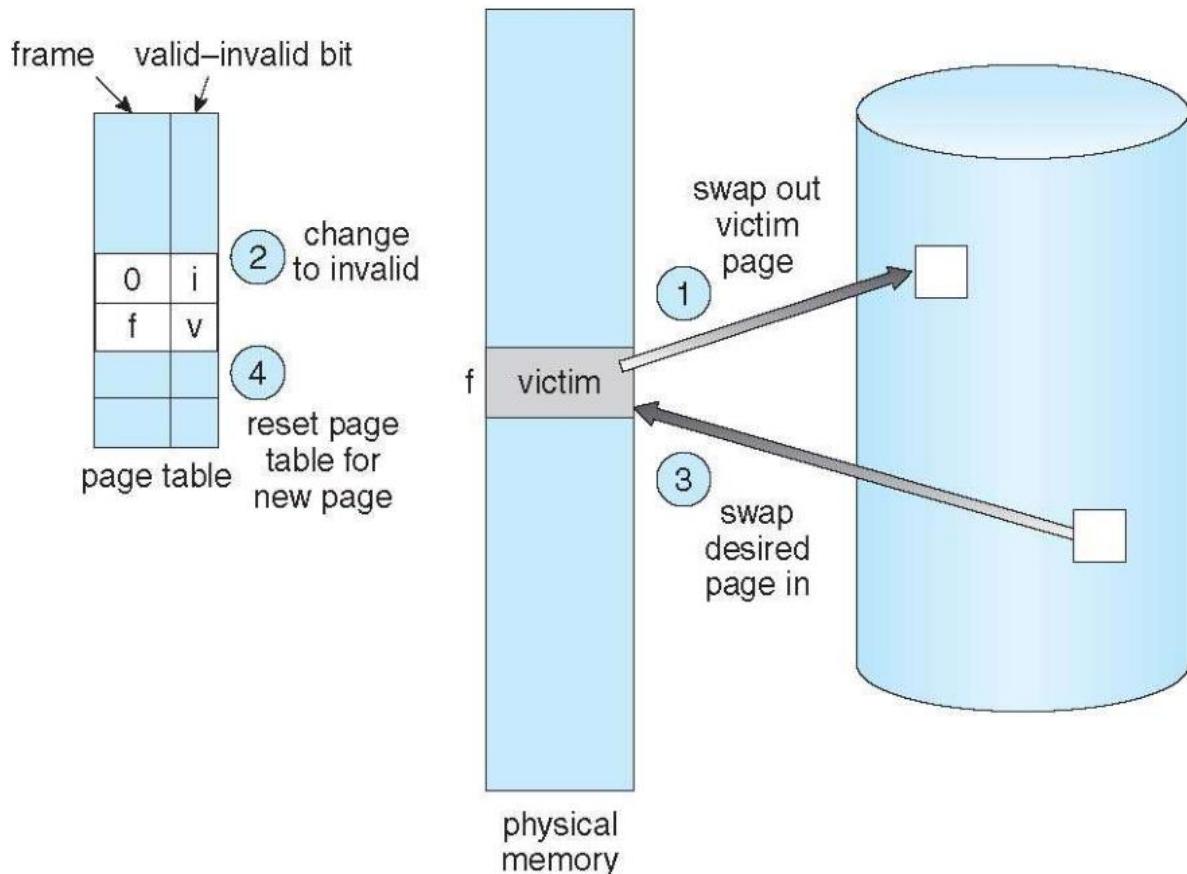
## ❖ Sostituzione delle pagine

Il problema del demanding paging non è soltanto quello di recuperare la pagina in memoria secondaria facendo una lettura con l'accesso alla memoria secondaria e poi alla memoria principale, ma occorre anche gestire il problema di dover anche liberare eventualmente memoria principale quando è tutta occupata e quindi decidere come liberarla. Ci sono varie tecniche.



## ❖ Quale pagina sostituire se non c'è un frame libero dopo un page-fault?

- Il processo che non trova un frame libero può essere terminato dal sistema operativo, ma non è la scelta migliore.
- Si può fare lo swapping di un intero processo riducendo il grado di multiprogrammazione, spostando nel backing store tutte le pagine di quel processo. E' molto costoso.
- **Il sistema operativo effettua un page replacement combinando swapping e paging.**



Si cerca un frame nella tabella delle pagine, nel momento in cui è invalido lo si cerca nel backing store e una volta trovato per poterlo caricare in memoria fisica dobbiamo trovare un frame libero. Se non esiste un frame libero in memoria fisica dobbiamo trovare un **frame vittima** da sacrificare, per cui viene fatto uno swap out nel backing store per poi fare lo swap in della pagina desiderata.

Si individua un frame vittima dalla memoria principale e si fa lo swap out verso il backing store

Si cambia il bit di validità legato a quel frame in invalido (i) nella tabella delle pagine perché non risiede più nella memoria fisica

Si fa lo swap in della pagina desiderata che viene caricata dal backing store in memoria principale al posto del frame vittima

Si resetta la entry della tabella delle pagine per quella nuova pagina di memoria appena caricata

**La sostituzione di pagine è fondamentale per il demand paging.** Se si applica in modo corretto c'è una netta separazione tra memoria virtuale e fisica. Previene la sovrallocazione di memoria e per ridurre l'overhead di trasferimento pagine si inserisce un **bit di modifica (dirty bit)** che indica le pagine modificate da un processo

- Il **dirty bit** serve per distinguere tra le pagine che durante il processamento sono state solo lette e quelle che hanno subito delle modifiche. Se una pagina è stata solo letta quando il processo termina la sua computazione e viene rimossa è inutile andarla a ricopiare nel backing store (overhead). Se una pagina è stata modificata è importante che venga modificata anche nel backing store. Si agisce solo su quelle pagine modificate, allora.

## ❖ Passi per sostituzione delle pagine

1. Trova la locazione della pagina desiderata su disco, dato che non è presente il riferimento ad essa nella tabella delle pagine (bit di validità = 1)
2. Trova un frame libero nella memoria principale controllando la lista dei frame liberi:
  - a. se c'è un frame libero usalo
  - b. se non c'è usa algoritmo di page replacement per selezionare un frame vittima
  - c. scrivi il frame vittima su disco solo se il dirty bit è impostato, perché se non è impostato possiamo risparmiare di copiarlo sul disco perché è stata caricata in memoria per solo lettura e non è stata modificata. Si sostituisce in modo diretto.
3. Porta la pagina desiderata nel nuovo frame libero; aggiorna la tabella di pagina e lista dei frame liberi
4. Continua il processo riavviando l'istruzione che ha causato il trap di page fault

**Potenzialmente 2 transferimenti di pagina per page fault quando la lista dei frame si è esaurita e il dirty bit della pagina è stato settato a 1. Utilizzando il bit di modifica si può ridurre di un accesso, avremo solo uno swap in (2 solo se modifica).**

## ❖ Algoritmi di sostituzione delle pagine

Per implementare il demand paging occorre:

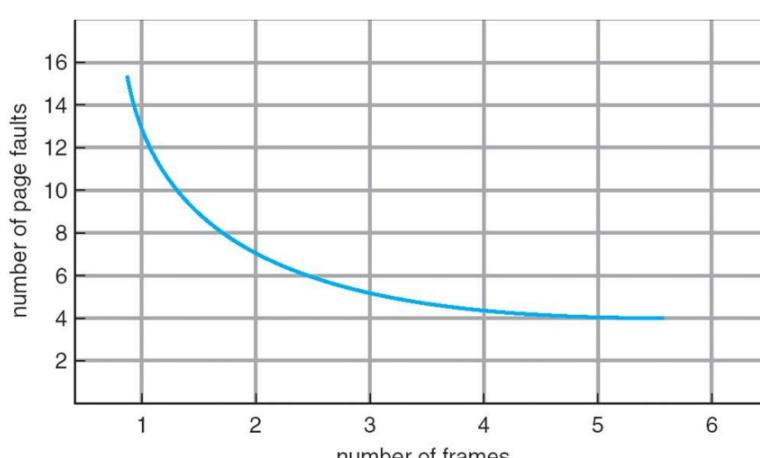
- Algoritmo di allocazione frame
- Algoritmo di sostituzione pagina

Un algoritmo di sostituzione delle pagine si valuta effettuandone l'esecuzione su una particolare **stringa di riferimenti alla memoria (ossia una sorta di log di tutti gli accessi in memoria fatti da un processo)** e calcolando il **numero di page fault generati**.

Ogni stringa indica solo numeri di pagina, non indirizzo completo. Accessi ripetuti su una stessa pagina non causano page fault.

- ▶ Esempio, per la sequenza che segue, assumendo 100 bytes per pagina  
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,  
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- ▶ Si ottiene  
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Il che vuol dire che non vado a considerare gli indirizzi generati da un processo, ma solo i frame effettivi a cui si fa riferimento con quegli indirizzi, per valutare se sono stati caricati oppure no. Un algoritmo dovrebbe anticipare i frame che servono ad un processo considerando gli indirizzi da esso generati.



**Un algoritmo è buono quanto meglio anticipa i frame che vengono richiesti da un processo.**

**All'aumentare del numero dei frame disponibili in memoria principale, diminuiscono i page-fault.**

**Se avessimo un solo frame si generano in continuazione page fault**

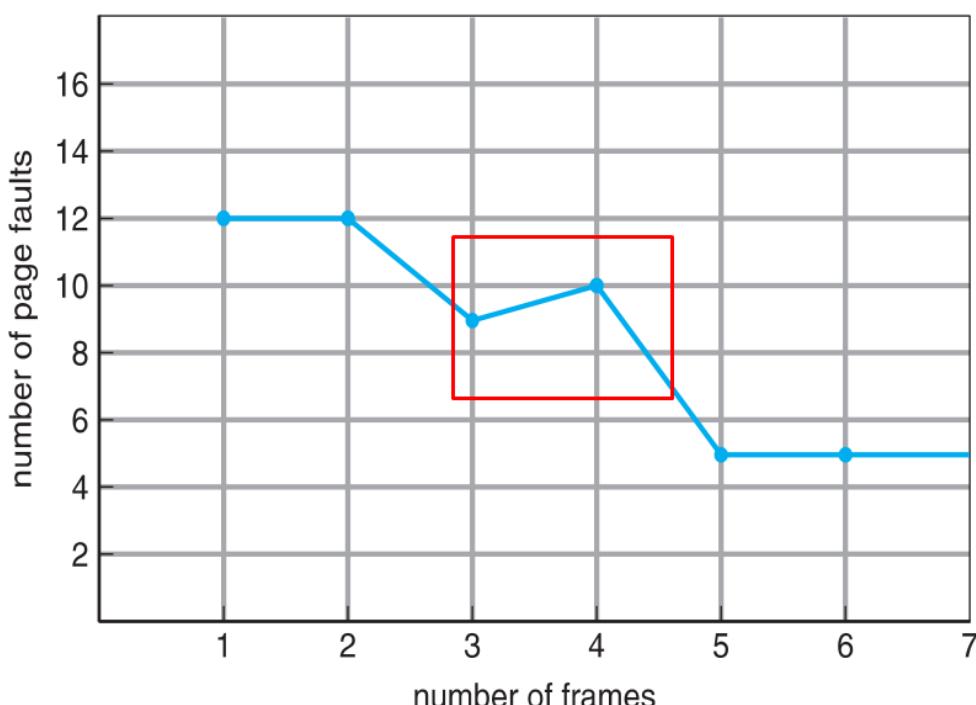
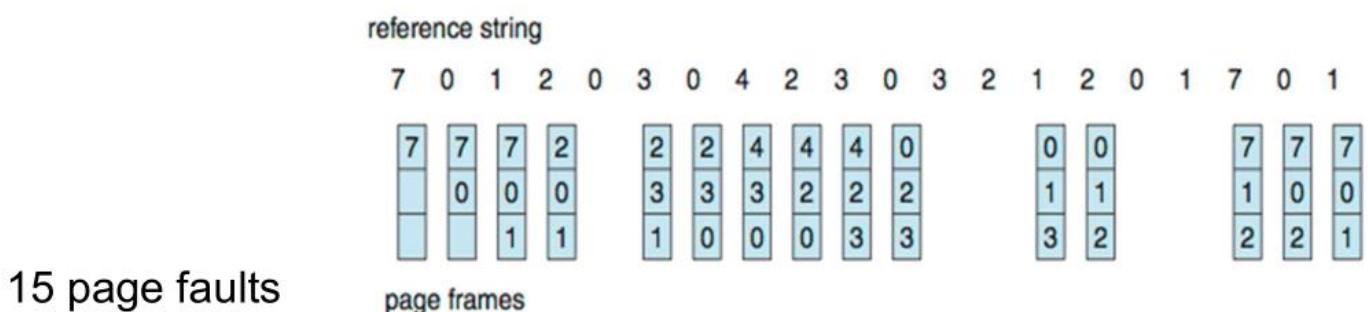
## Algoritmi di sostituzione pagina:

- **Algoritmo FIFO**
- **Algoritmo Ottimale**
- **Algoritmo LRU**
  - **Second Chance**
  - **Enhanced Second-Chance**
  - **Algoritmi contatori**
- **Algoritmi di page buffering**

**Reference string:** 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

### ❖ Algoritmo FIFO

- Si sacrifica e sostituisce la pagina più “vecchia” in memoria che si trova in essa da più tempo.
- Assumiamo di avere a disposizione 3 frame (3 pagine in memoria nello stesso momento per processo), con demanding paging puro, ossia le pagine vengono caricate quando servono e all'inizio i frame sono liberi.
- Le pagine più recenti entrano in coda le più vecchie escono (è un criterio arbitrario, le più vecchie potrebbero contenere dati importanti).
- Facile da implementare, ma la performance varia molto con la reference string.
- Aggiungendo più frames può causare più page fault: **Anomalia di Belady**

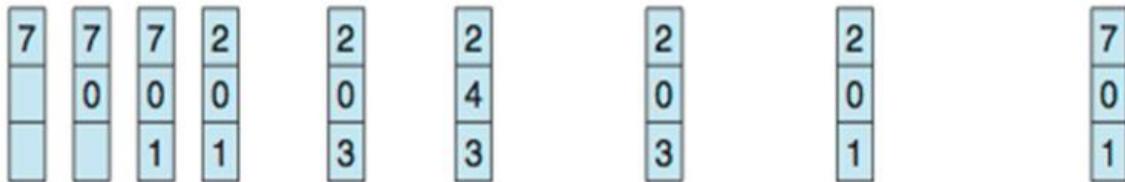


## ❖ Algoritmo Ottimale

- Dovrebbe prevedere esattamente il futuro mantenendo le pagine che vengono utilizzate per il periodo più lungo
- Si sostituisce quindi la pagina che non verrà utilizzata per il periodo più lungo
  - Come si può sapere? Non si può leggere nel futuro!
- Usato per valutare le performance degli algoritmi.
- Nell'esempio solo **9 page fault** (il minimo possibile).

reference string

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



page frames

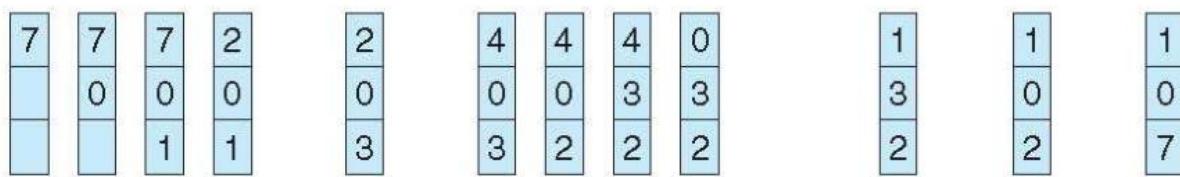
Per caricare il 2 andiamo a vedere quale delle 3 pagine correntemente caricate verrà utilizzata meno nel prossimo futuro.

## ❖ Algoritmo LRU (Least Recently Used)

- Usa la conoscenza del passato invece che quella del futuro: pagine usate nel recente passato potrebbero essere richieste nel prossimo futuro, quindi le manteniamo in memoria. Quelle non usate nel recente passato è probabile che non saranno usate nel prossimo futuro.
- **Sostituisce le pagine che per più tempo non sono state usate associando ad ogni pagina il tempo dell'ultimo uso**
- **12 fault – meglio di FIFO ma peggio di OPT.**
- Solitamente un buon algoritmo, frequentemente usato.

reference string

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



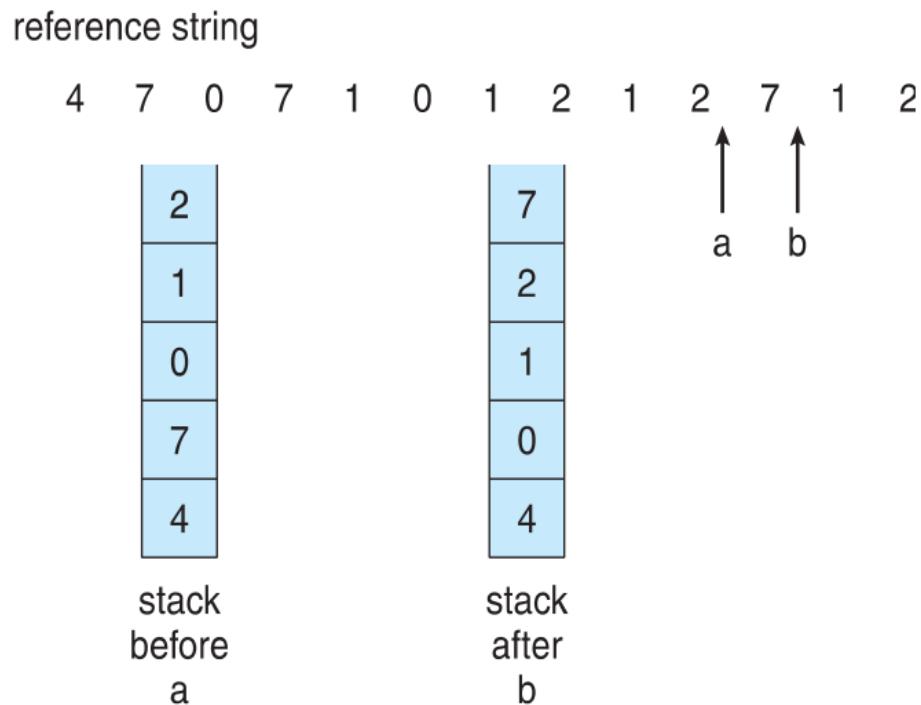
page frames

Il problema dell'algoritmo LRU è come implementarlo in modo efficiente, assistenza hardware può essere richiesta. Due possibili **implementazioni**:

- **con Counter:** sfrutta il concetto del time of use; ogni elemento della tabella delle pagine ha un registro counter. La CPU ha un clock logico/contatore incrementato per ogni riferimento in memoria. Per ogni riferimento ad una pagina di un processo specifico viene copiato il clock nel suo counter. Quando una pagina deve essere cambiata, cerca il valore minore del counter.
- **con Stack:** mantiene uno stack di numeri di pagina. Al riferimento di pagina essa si mette in cima allo stack. La più recente è in cima, la meno recente quindi sul fondo dello stack. Con una lista doppiamente linkata è facile accedere al fondo e al top dello stack.
- **LRU e OPT sono casi di stack algorithm senza l'anomalia di Belady. Non può capitare che se si aumenta il numero di frame aumenta anche il numero di page fault, al massimo rimane uguale al numero di frame usati in precedenza.**

## Implementazione con **Stack**

- Mantiene uno stack di numeri di pagina
- Al riferimento di pagina si mette in cima allo stack



## ❖ Algoritmi di approssimazione a LRU: Second Chance e Enhanced Second-Chance, Algoritmi contatori

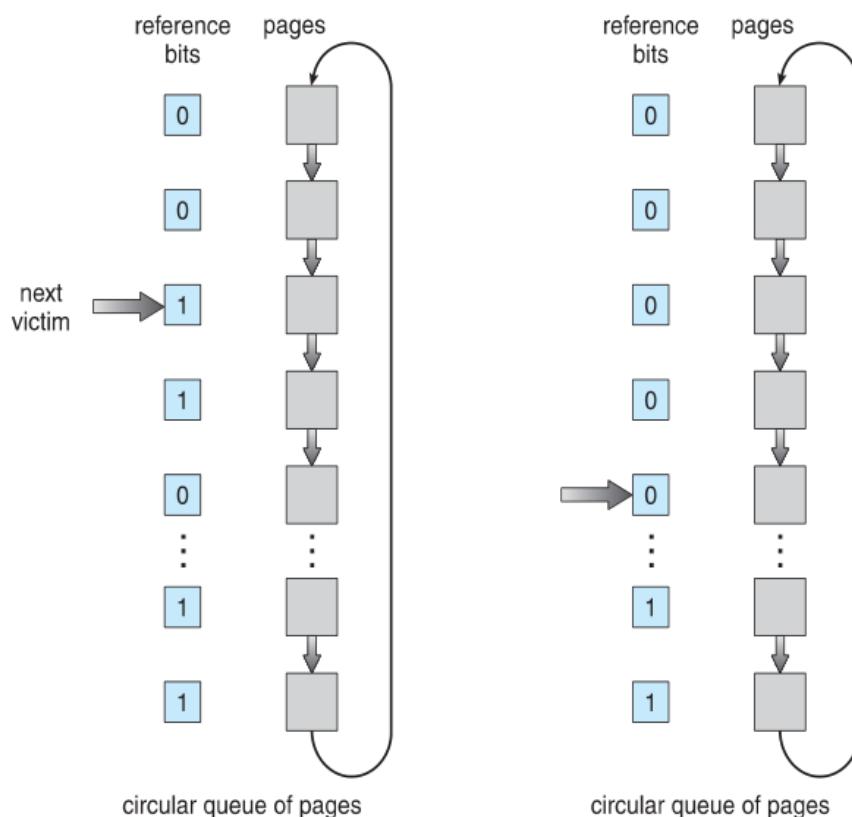
LRU ha bisogno hardware speciale che non tutti i sistemi forniscono. In alcuni casi è approssimato con il supporto di un **bit di riferimento (reference bit)**. Ogni pagina associata viene associata al bit di riferimento, che inizialmente = 0. Quando si fa riferimento alla pagina (lettura o scrittura) il bit viene settato ad 1. Si può sostituire qualunque pagina con bit = 0 (se esiste). **Non si conosce l'utilizzo effettivo.** **Attenzione a non confonderlo col bit di modifica (dirty bit).**

### Algoritmo di second chance

- da ai riferimenti una seconda chance di sopravvivere e vengono sacrificati solo dopo una seconda chance. **È un FIFO con un reference bit.** Dopo la selezione della pagina si controlla il bit di riferimento, se il bit è 0 si sacrifica la pagina, se il bit è 1 si da una “seconda chance” settando però a 0 il suo bit.
- Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine con bit = 0 non siano state sostituite, oppure non sia stata data loro una seconda chance.
- Implementato con coda circolare

### Algoritmo second-chance

- Implementato con coda circolare
- Puntatore indica la pagina da sostituire
- Quando serve un frame avanza finché non trova uno zero
- Pagina sostituita in quella posizione
- Se tutti i bit sono 1 diventa FIFO



### **Algoritmo di enhanced second chance**

- Migliora l'algoritmo di second chance usando insieme il reference bit ed il bit di modifica (se disponibile),

Se una pagina non è stata nè usata nè modificata è la migliore da sostituire.

### **Prendi coppie ordinate (reference, modify)**

1. (0, 0) né usata e né modificata – miglior pagina da sostituire
2. (0, 1) non usata ma modificata – non così buona, bisogna scrivere prima della sostituzione
3. (1, 0) usata ma non scritta – potrebbe presto essere usata di nuovo
4. (1, 1) usata e modificata – può essere riusata e deve essere salvata prima della sostituzione

**Quando occorre una sostituzione di pagina usa lo schema clock (second chance) ma cerca una pagina della classe più bassa non vuota**

### **Algoritmi contatori**

Viene mantenuto un contatore del numero di riferimenti totali per ogni pagina. Non molto usati, costosi e non approssimano bene l'ottimo.

- **Least Frequently Used (LFU) Algorithm:** sostituisce pagine con il valore del contatore più basso. Pagine intensamente usate hanno contatori alti, però bisogna tenere conto che quelle più intensamente usate potrebbero essere quelle iniziali nel caricamento di un processo.
- **Most Frequently Used (MFU) Algorithm:** sostituisce pagine con il valore del contatore più alto assumendo che le pagine con il contatore più basso siano le più recenti e debbano essere ancora usate.

### **❖ Algoritmi di page buffering**

Mantengono sempre un pool di frame liberi. Non viene liberato un frame quando si verifica un page fault, si cerca di averne sempre qualcuno a disposizione. Se si occupa qualche pagina riducendo il pool di frame liberi, si seleziona una pagina vittima da liberare successivamente per aggiungere di nuovo un frame al pool. Quando è il tempo opportuno si porta fuori la pagina vittima nei momenti meno critici. In questo modo si riduce il numero di page fault.

Variazione: si mantiene la lista di pagine modificate. Quando il sistema di paging è in idle avviene la copia delle pagine modificate in backing store e si setta il bit di modifica a non-dirty.

## ❖ Allocazione di frame

**Come allocare i frame per i processi tenendo presente che il s.o. ha processi multipli da servire?**  
Abbiamo detto che la memoria virtuale consente di memorizzare frame ridotti per un processo, però anziché partire con 0 frame con processo (demaning page puro), si può partire anche da una situazione in cui si preallocano dei frame per processo per evitare i page fault iniziali nell'avvio di un processo.

- La strategia di allocazione deve tenere conto di un minimo numero di frame richiesti da un processo. Il massimo numero dipende dai frame disponibili nel sistema.

Due principali schemi di allocazione:

- **Fixed allocation**
- **Priority allocation**

## ❖ Fixed Allocation

Si prevede un'equità nell'allocazione dei frame, **equal allocation**: se abbiamo un grado di multiprogrammazione con  $n$  processi e abbiamo disponibili  $m$  frame, vengono allocati ad ogni processo  $m/n$  frame. Mantiene un frame buffer pool.

- Per esempio, se abbiamo 100 frame disponibili (tolti quelli per SO) e 5 processi, per ogni processo vengono allocati 20 frame.

Un'allocazione di questo tipo è semplice ma non tiene conto del "peso" di questi processi. Dunque si introduce una variante:

- **proportional allocation** che alloca i frame proporzionalmente alla dimensione del processo.

$$- a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

*s(i) = size del singolo processo*

*S = size totale di tutti i processi*

*m = numero totale di frame disponibili*

## ❖ Priority allocation

Usa uno schema di proportional allocation usando le priorità invece della dimensione

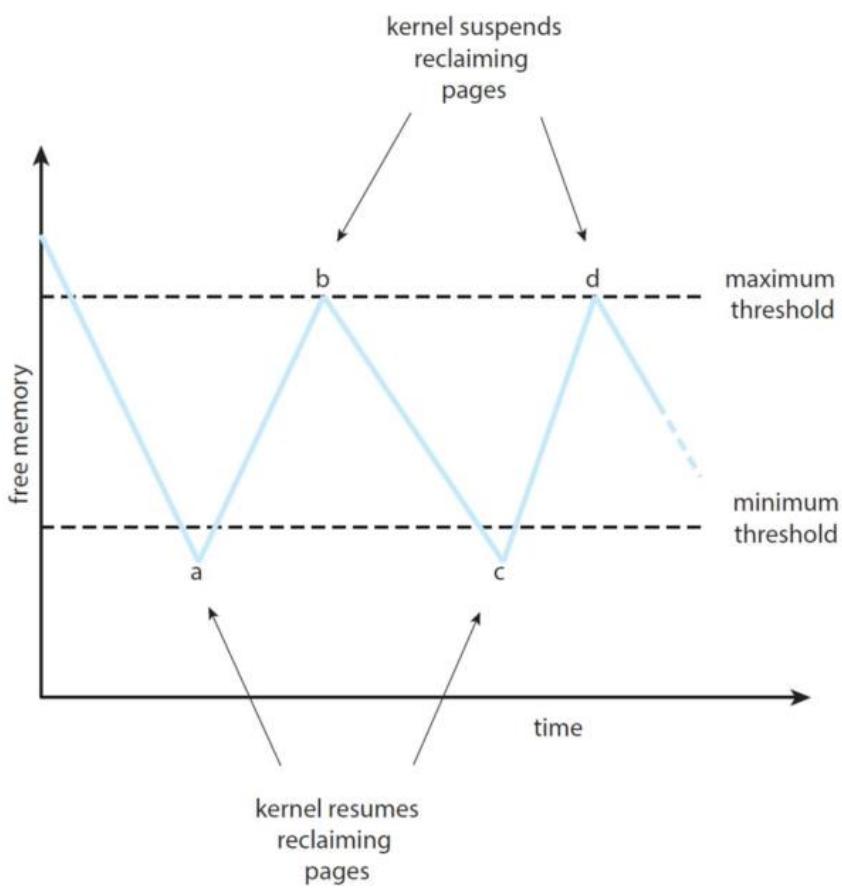
## ❖ Rimpiazzamento locale o globale dei frame

Se un processo  $P_i$  genera un page fault e bisogna rimpiazzare dei frame perché la memoria principale è satura, si può scegliere fra 2 **politiche di rimpiazzo** del frame:

- **Locale:** si seleziona uno dei frame allocati al processo  $P_i$  e lo si rimpiazza. Il processo  $P_i$  rimane nello spazio di memoria che gli è stato assegnato. Prestazioni più stabili e consistenti ma la memoria viene sottoutilizzata.
- **Globale:** si seleziona un frame da rimpiazzare dall'insieme di tutti i frame allocati. Il frame può essere quello di un processo  $P_j$  con priorità più bassa di  $P_i$  e quindi lo si rimpiazza. Il processo  $P_i$  può impattare su altri processi, perché si sottraggono frame a loro allocati e potrebbe generare anche a loro più page fault e potrebbe variare i loro tempi di esecuzione. Aumenta però il throughput. **Scelta più usata.**

## Strategia per global page-replacement

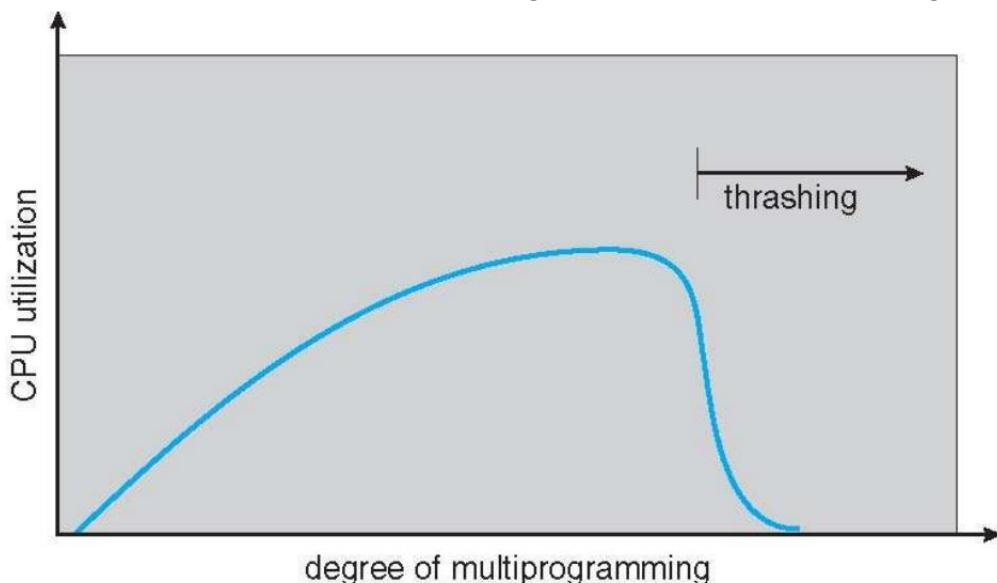
Non si attende che la free-frame-list si esaurisca, si mantiene sempre il pool di frame liberi. Si valuta una soglia critica di memoria libera, al di sotto della quale si inizia a recuperare frame tramite il rimpiazzamento globale. Si recupera frame finché non si va sopra la soglia massima di frame liberi. Strategie chiamate **reapers routine**. Le soglie possono essere settate.



## ❖ Trashing

E' uno dei problemi più gravi che si può presentare in seguito ad un'allocazione errata di frame per un processo. Se ad un processo non vengono allocati frame sufficienti avremo un frame-rate molto alto dovuto al fatto che questi frame devono essere recuperati dal backing store. In questa situazione siccome abbiamo pochi frame allocati non adeguatamente per un processo potremmo trovarci in una situazione in cui c'è un rimpiazzamento continuo di pagine e ci sono continui page-fault.

- Si crea un “effetto paradosso” perché quando ci sono i page-fault la CPU non “lavora” ed è poco impegnata poiché la loro gestione è fatta in modo hardware dal sistema di swap in/out.
- **Il processo non va avanti con la sua esecuzione per la mancanza di pagine ed è impegnato in continui swap in e swap out di pagine.** Questo porta il sistema operativo a vedere un basso utilizzo di CPU per il processo, ignaro della gestione della memoria che è hardware e quindi dal canto suo vuole incrementare il grado multiprogrammazione allocando nuovi processi.
- La situazione può portare ad una situazione paradossale in cui il sistema viene intasato soprattutto nel caso di una politica di rimpiazzo globale.
- Infatti se abbiamo un algoritmo globale di page-replacement, i processi iniziano a sottrarre frame ad altri processi, che a loro volta vanno in page-fault, intasando i paging device per swap in e swap out, si accodano i processi e si svuota la coda ready e il sistema aumenta il grado di multiprogrammazione .
- **L'utilizzo della CPU aumenta con la multiprogrammazione, poi va in thrashing**



## Come risolvere il thrashing?

- Limitare l'effetto del thrashing usando un **algoritmo di local replacement** in cui i frame sono rimpiazzati solo localmente, non possono essere “rubati”, ma non viene risolto il problema, un processo in thrashing occupa il dispositivo di paging e rallenta il page fault di tutti i processi.
- **Si definisce un modello di località (locality model) del processo.** Si cerca di capire quanti sono i frame di cui un processo ha bisogno, vedendo quanti ne sta utilizzando.

## ❖ Locality model

E' un modello che stabilisce quali sono quei frame che contemporaneamente servono ad un processo per portare avanti una determinata esecuzione.

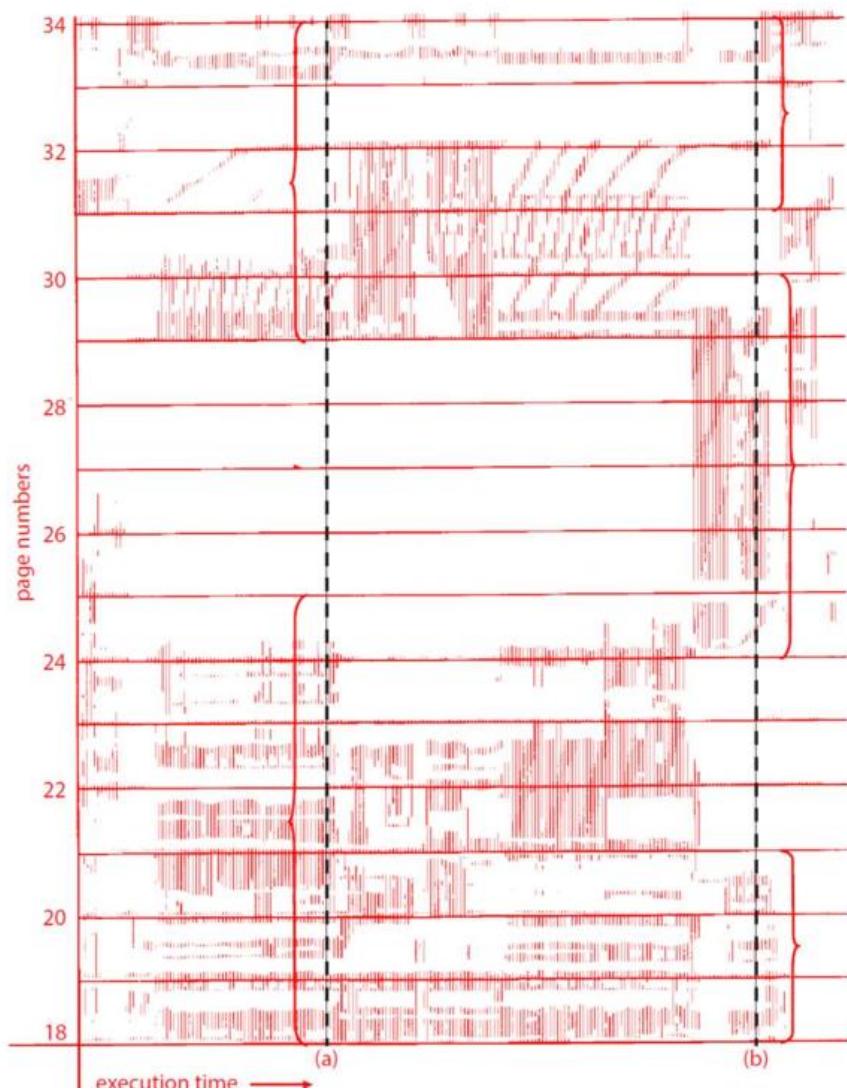
- Un processo passa da locality a locality gradualmente durante l'esecuzione, si passa da insieme di frame a insieme di frame che rappresentano lo stato esecutivo di un processo in termini di insieme di pagine che devono essere usate contemporaneamente.
- Il locality model ci dice quali pagine servono in contemporaneamente in memoria per portare avanti una determinata esecuzione di un processo
- Le località sono insiemi di pagine che sono usate insieme. Le località possono sovrapporsi e nuove funzioni corrispondono a nuove località.
- es. Località di una funzione: istruzioni, variabili locali, sottoinsieme delle globali

**Le località sono definite dalla struttura del programma e dalle sue strutture dati.** Le località indicano anche ciò che dovrebbe essere mantenuto in una cache. Se si riesce ad allocare i frame località di un processo i page fault saranno molto limitati. Avremo i page fault solo quando ci sarà la transizione da una località all'altra. Se invece non si allocano frame sufficienti per la località corrente del processo allora si andrà verso il fenomeno del thrashing.

Località al tempo (a)  
{18-24, 29-33}

Località al tempo (b)  
{18-20,24-29, 31-33}

18, 19, 20 si  
sovrappongono



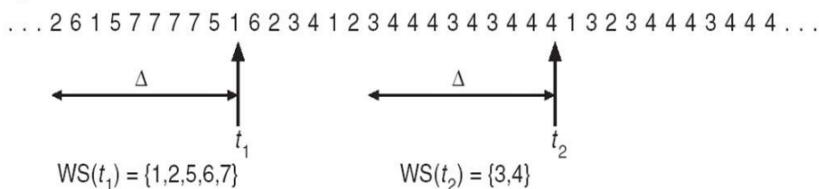
## ❖ Modello Working Set

E' un modello che consente di stabilire l'idea di quella che è la località di un processo. Si calcola per ogni processo. Utilizza un parametro  $\Delta$  per definire un **working-set windows** ossia che stabilisce i  $\Delta$  riferimenti più recenti in memoria. Se abbiamo  $\Delta$  riferimenti in memoria, per ogni riferimento corrisponderà una pagina e le pagine possono essere ripetute. Quindi abbiamo anche l'insieme delle  $\Delta$  pagine più recenti per il **Working Set** il quale sarà sicuramente minore della finestra workig-set windows perché supponiamo per località una pagina venga richiamata più volte in tot riferimenti.

- Se una pagina è in uso attivo è nel WS, se non è più utilizzata esce dal WS dopo  $\Delta$  riferimenti di memoria in termini di tempo.
- Se il working-set windows è troppo ampio rientrano tutti i frame di un processo, se è stretta non prende abbastanza frame per una determinata fase di esecuzione di un processo. Deve esserci un bilanciamento dinamico che per ogni fase di esecuzione del processo mantenga nel working set le pagine più importanti da caricare in memoria.
- Il WS approssima la località di un programma.

Esempio con  $\Delta = 10$  (dimensione di WS varia)

page reference table



- L'accuracy del WS dipende dalla sua dimensione. La dimensione del WS è detta **Working Set Size (WSS)** per un processo Pi. Se  $\Delta$  troppo piccolo il WSS non contiene l'intera località. Se  $\Delta$  dei riferimenti è troppo grande il WSS può sovrapporre più località. Se  $\Delta$  tende a  $\infty$  contiene tutto il programma.
- **WSSI è la caratteristica curciale da monitorare:** se facciamo la somma di tutti i WSS abbiamo il numero totale di frame che dovrebbero essere caricati in memoria principale in un certo  $\Delta$  temporale per evitare page fault e approssimiamo la località. Se la somma ottenuta è maggiore della memoria disponibile potremmo avere una situazione di trashing e il sistema in questo caso dovrebbe ridurre la multiprogrammazione.
- **I SO monitora il Working Set di ogni processo:**
  - ha un'indicazione di quanti e quali frame sono necessari per un processo Pi in un certo momento
  - alloca i frame secondo il WSSI per un processo Pi
  - se la somma di tutti i WSS supera la memoria disponibile allora riduce la multiprogrammazione sospendendo un processo e facendo il suo swap out
  - se la somma di tutti i WSS è minore della memoria disponibile allora ha la possibilità di allocare un nuovo processo dato che ci sono frame extra liberi

**La finestra del Working Set è mobile.** Ad ogni riferimento in memoria può entrare o uscire una pagina. Si può approssimare con un timer ad intervalli fissi e un bit di riferimento.

- Esempio: per  $\Delta = 10000$ 
  - Timer interrupt dopo ogni 5000 unità di tempo
  - Mantiene in memoria 2 bit per ogni pagina
  - Quando il timer interrompe copia e setta i valori di tutti i reference bit a 0
  - Se bit attuale o uno di 2 bit in memoria = 1  $\Rightarrow$  la pagina è nel working set
  - Non accurato, non sappiamo dove è avvenuto il riferimento in 5000 unità
  - Miglioramento: 10 bit di storia e interrupt ogni 1000 unità di tempo

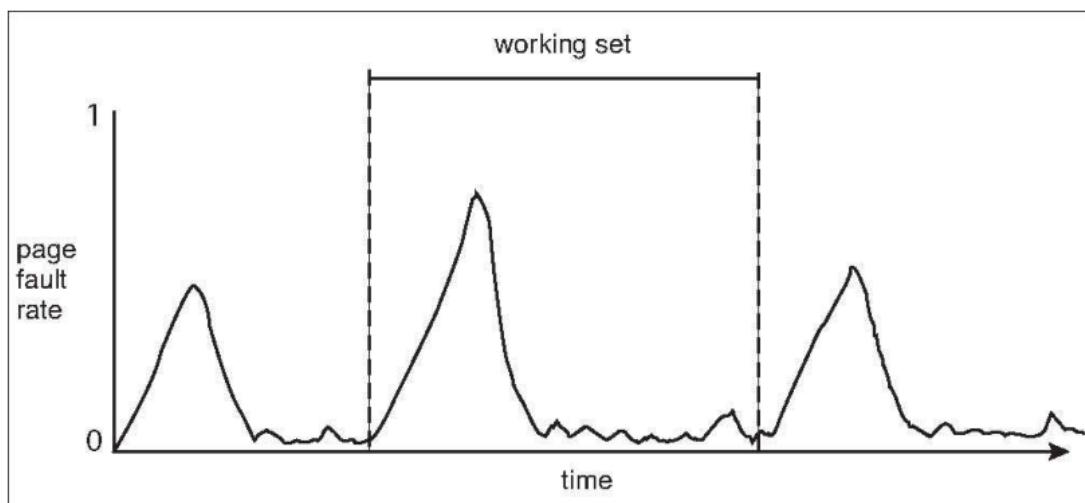
## ❖ Risoluzione trashing: Page-fault frequency

Per **risolvere il trashing** è possibile anche **stabilire un tasso “accettabile” di page-fault frequency (PFF)** che è un metodo più diretto e semplice.

- Se il tasso attuale è troppo basso il processo può anche perdere frame in una politica di rimpiazzo globale. I frame liberati possono essere distribuiti ai processi con frequenze più alte.
- Se il tasso attuale è troppo alto il processo deve guadagnare frame

**Esiste una relazione diretta tra working set di un processo e la frequenza di page-fault** dovuta al fatto che la frequenza di page fault aumenta quando c’è un cambio di località. L’aumento dei page-fault può essere un indice per capire che c’è stato un cambiamento del working set del processo Pi.

**Picchi e valli di frequenze di page fault indicano i cambiamenti di località:** possono essere monitorati per capire l’intervallo giusto di un working set. L’intervallo tra gli inizi dei picchi definiscono il Working Set.

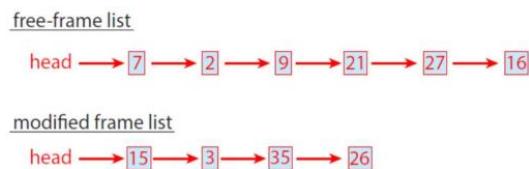


## ❖ Compressione di memoria

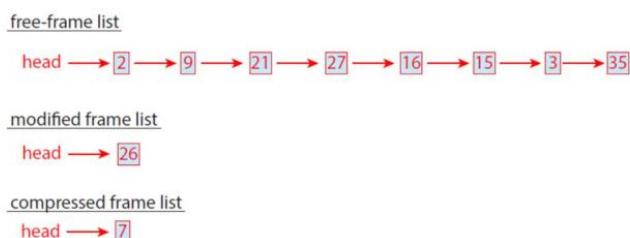
Si utilizza la compressione di memoria per ridurre l’utilizzo di memoria

### □ Esempio:

- Free frame list sotto soglia e selezionati frame 15, 3, 35, 26 per liberare memoria



- Invece di scrivere direttamente in swap space fa la compressione di alcuni frame e li mette nei compressed frame (es. 15, 3, 35 compressi in 7 e liberati)



- Quando si fa riferimento alle pagine in 7 si decomprime e rialloca

## ❖ Allocare la memoria per il kernel: Buddy e Slab

L'allocazione di memoria per i processi kernel è trattata in modo diverso dalla memoria allocata per i processi utente. Dai processi kernel dipende l'integrità e le performance dell'intero sistema. A questi processi deve essere allocata il prima possibile della free memory: hanno spesso un pool di frame liberi di memoria differente da quello dei processi utente. La memoria per il kernel viene utilizzata per usi specifici e in alcuni casi necessita di essere contigua (ad esempio per i dispositivi i/o). Quindi la memoria per il kernel deve avere una gestione ad hoc.

Due metodi per l'allocazione di memoria ai processi kernel:

- **Buddy System**
- **Allocazione slab**

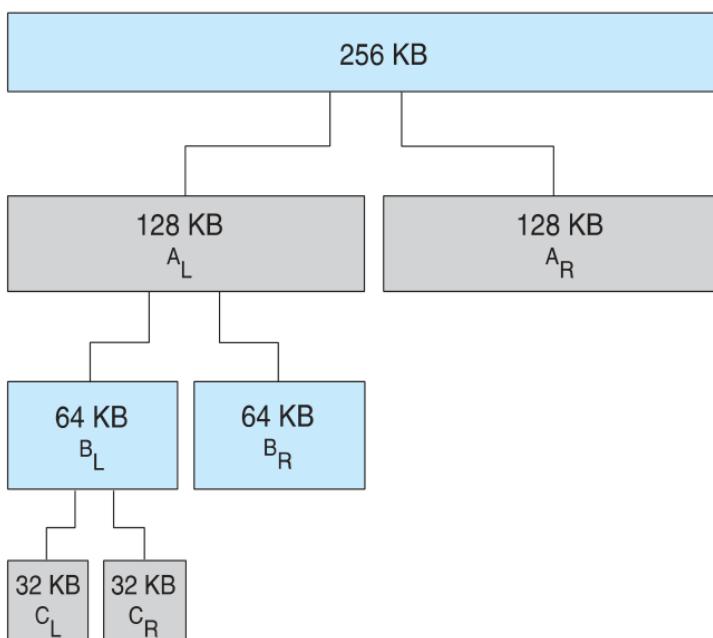
### ❖ Buddy System

- **Il sistema buddy alloca memoria attraverso segmenti di dimensioni fissate costituiti da pagine fisicamente contigue**

Un segmento largo e contiguo viene suddiviso in porzioni più piccole; più segmenti poi sono combinati rapidamente per ottenere un frammento più grande contiguo.

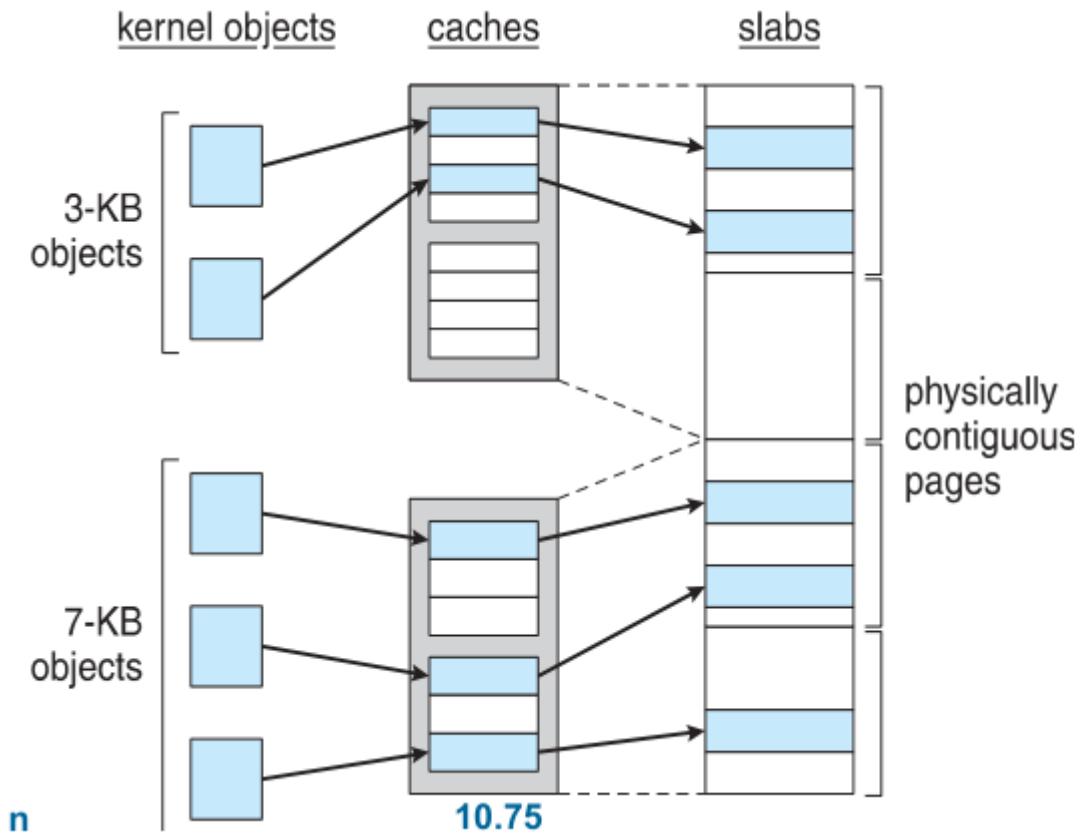
- La memoria viene allocata usando un **allocatore di potenza-di-2** il quale soddisfa richieste in unità di dimensione potenza di 2. **Le richieste di memoria contigua sono approssimate alla più alta Potenza di 2.** Quando occorre un'allocazione più piccola del frame disponibile, il chunk corrente di memoria viene diviso in due buddies della prossima potenza di 2 più bassa finché non si riesce ad allocare buddies di dimensione adeguata.
  - **Questa tecnica soffre di frammentazione interna. Vantaggio: memoria allocata velocemente e compatta.**
- Per esempio, assume chunk di 256KB disponibile, il kernel richiede 21KB
- Diviso in  $A_L$  and  $A_R$  di 128KB ognuno
    - Uno ulteriormente diviso in  $B_L$  e  $B_R$  di 64KB
      - Uno ulteriormente in  $C_L$  e  $C_R$  di 32KB ognuno – uno usato per soddisfare la richiesta

physically contiguous pages



## ❖ Slab Allocator

- Strategia alternativa che **elimina frammentazione interna**. Vengono definiti degli “slab” ossia una sorta di slot che sono dimensionati esattamente per un certo tipo di oggetti preallocati.
- **Uno slab si compone di una o più pagine fisiche contigue ed è il contenitore di un oggetto di un certo tipo.**
- **Una cache consiste di uno o più slab;** ne esiste una per ogni data structure del kernel. Ogni cache viene popolata di oggetti.



## Altre Considerazioni – Page Size

- I progettisti di SO raramente possono decidere la dimensione delle pagine
- Dimensione definite in fase di progetto
  - Sempre potenza di 2, di solito nel range  $2^{12}$  (4,096 bytes) fino a  $2^{22}$  (4,194,304 bytes)
- Selezione della dimensione delle pagine deve tenere in considerazione:
  - Frammentazione
  - Dimensione della page table
    - ▶ Per VM di 4MB ( $2^{22}$ ) si hanno 4096 pagine di 1024 byte e 512 da 8192 byte
  - Risoluzione
    - ▶ Si accede alla porzione di memoria necessaria
    - ▶ Minore I/O overhead
    - ▶ Maggior numero di page faults
- Pagine piccolo: località, meno frammentazione, meno I/O overhead
- Pagine grandi: meno page fault, page table più piccolo

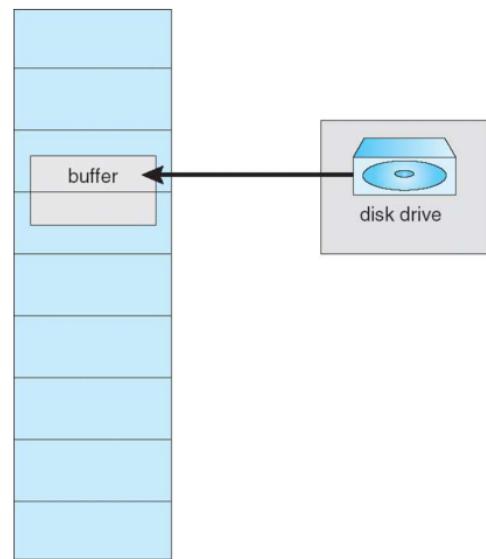
Es. Linux ha pagine di 4KB, ma anche più grandi, e.g., 2M

## Altre Considerazioni – Struttura del Programma

- A volte il programmatore può gestire
- La scelta delle strutture dati può influire sul numero di page fault
  - Stack solitamente buona località perché accesso su top
  - Hash table invece solitamente località meno buona perché accesso sparso
- Compiler e loader può avere un impatto sul paging
  - Separazione tra codice e dati, uso di codice rientrante (pagine read-only), etc.
  - Loader carica routine non a cavallo di pagine, routine che si chiamano vicendevolmente vicine (possibilmente nella stessa pagina), etc.

# Altre Considerazioni – I/O interlock

- **I/O Interlock** – le pagine qualche volta devono essere “locked” in memoria
- Nel caso di I/O - pagine usate per copiare un file da un dispositivo (es. USB) devono essere locked per evitare che siano selezionate da un algoritmo di page replacement
  - Es. Processo richiede I/O, altro processo schedulato, fa page fault e sottrae pagina in global replacement
- **Pinning** (fissare) pagine da tenere in memoria
- Lock bit per impedire il trasferimento di pagina
- Può essere usato anche durante il paging, es. per impedire che proc. ad alta priorità prenda pagine appena caricate a proc. a bassa priorità



## ❖ Memoria virtuale Linux

Per la memoria virtuale Linux usa la tecnica di demand paging con allocazione dei frame da una lista di free frame. La politica di rimpiazzo delle pagine: **Global page replacement** con l'**algoritmo LRU Second Chance**.

Due liste di pagine: **active\_list** e **inactive\_list**, le inactive sono elegibili per essere rimpiazzate.

Ogni lista ha un fronte e un retro. Le pagine attive sono quelle a cui viene fatto un riferimento e ciò viene identificato per mezzo dell'**accessed bit**. Una nuova pagina a cui si fa riferimento viene messa nel retro della active list e quindi c'è uno scorrimento delle pagine. Emergono sul fronte le pagine che non hanno ancora avuto un riferimento. Quando una pagina arriva al fronte dell'active list vuol dire che non è stata ancora referenziata e si sposta in inactive list. Appena viene referenziata si sposta in coda nell'active list venendo "promossa".

Un demone **kswapd** periodicamente verifica se occorre liberare memoria, ossia se la memoria libera va sotto una certa soglia. Per liberare la memoria il demone scorre la **inactive\_list** e seleziona un frame vittima liberando frame.

## 9. MEMORIA DI MASSA

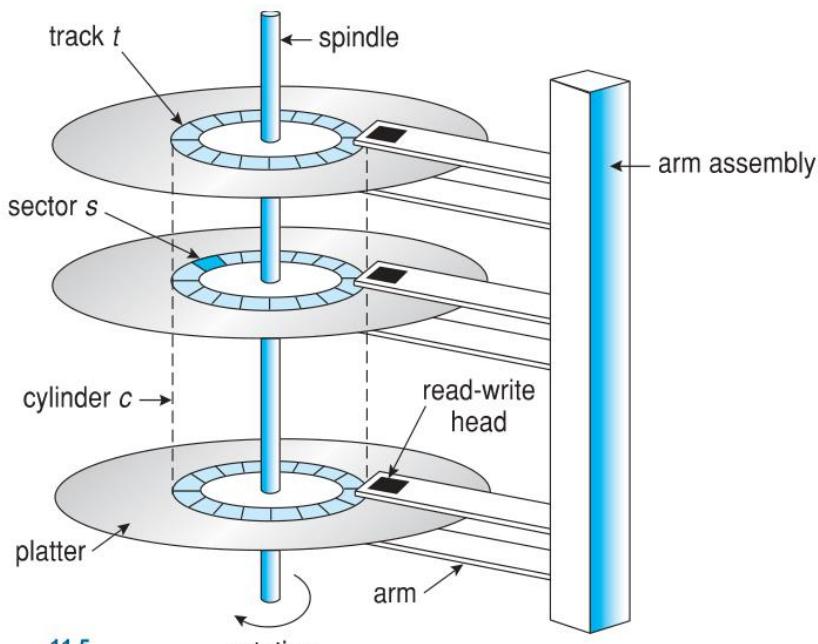
### ❖ Concetti principali

Esistono due tipologie di memoria secondaria:

- **Hard Disk Drives (HDDs)**: con parti meccaniche
- **Nonvolatile memory (NVM)**: non volatile, cancellabile e riprogrammabile elettronicamente (**memorie flash**)

### ❖ Hard Disk

I dischi magnetici sono il supporto fondamentale di memoria secondaria dei moderni sistemi elaborativi. Sono strutturati su **dischi piatti** come CD, chiamati semplicemente **piatti**. Superfici coperte da materiale magnetico. Le informazioni si memorizzano registrandole magneticamente sui piatti attraverso una **testina di read-write** che si muove sui piatti. C'è un **braccio** che muove le testine per ogni piatto. I **piatti** sono divisi in **tracce** e le tracce divise in **settori**. I **cilindri** sono un insieme di tracce, sui cui le testine si muovono. Esistono migliaia di cilindri, e per ogni traccia centinaia di settori.



- **Capacità da GB a TB**

- La **velocità** di un disco è caratterizzata da 2 valori:

- la **velocità di trasferimento**, cioè la velocità con cui i dati fluiscono dall'unità a disco al calcolatore (Transfer Rate) circa 1GB/sec di solito.
- il **tempo di posizionamento** per trovare una locazione su disco si compone da:
  - **seek time**: tempo necessario a spostare il **braccio** del disco in corrispondenza del cilindro desiderato (9 ms)
  - **rotational latency**: tempo necessario affinché il settore desiderato si porti, tramite la rotazione del disco, sotto la testina (dipende da velocità di rotazione disco). **Average latency = di solito ½ latency (mezzo giro)**

**Access Latency** = **Average access time** = average seek time + average latency

- Per dischi più veloci  $3\text{ms} + 2\text{ms} = 5\text{ms}$
- Per dischi lenti  $9\text{ms} + 5.56\text{ms} = 14.56\text{ms}$
- Average I/O time = average access time + (quantità da trasferire / velocità di trasferimento) + overhead del controllore
- Per esempio per trasferire un blocco di 4KB su un disco da 7200 RPM con un 5ms average seek time, 1Gb/sec di transfer rate con un .1ms overhead del controllore =
  - $5\text{ms} + 4.17\text{ms} + 0.1\text{ms} + \text{transfer time} =$
  - Transfer time =  $4\text{KB} / 1\text{Gb/s} = 0.031\text{ ms}$
  - Average I/O time for 4KB block =  $9.27\text{ms} + .031\text{ms} = 9.301\text{ms}$

#### ❖ NVMs

I dispositivi NVM sempre più rilevanti e diffusi. Sono elettrici e non meccanici. Tipicamente basati su memoria flash (chip NAND flash) vengono spesso inseriti in contenitori simili a unità disco, e per questa ragione sono chiamati **dischi a stato solido, o SSD**. Un dispositivo NVM può anche assumere la forma di un'unità USB. In tutte le sue forme possiamo trattarlo in modo uniforme

- **SSD:** Memoria nonvolatile usata come un hard drive. Non meccanica. Implementata con diverse tecnologie. Più costosa per MB. In generale ha meno capacità di HDD ma più veloce. Non ha parti mobili, e non ha parti meccaniche (non presenta quindi seek time e rotational latency). Connessi direttamente al bus di sistema. I **semiconduttori NAND** con cui sono implementate non possono essere sovrascritti direttamente, devono prima essere cancellati. Organizzati in pagine e presentano pagine che contengono dati non validi. La cancellazione avviene in blocco e prende tempo, inoltre ad ogni cancellazione c'è un deterioramento. Il controller contiene una tabella: **Flash Translation Layer (FTL)** indica quali pagine contengono dati validi. Per gestire i dati occorrono algoritmi di garbage collection per liberare blocchi e pagine non valide.
- Memoria volatile (**DRAM**) può essere usata per fare storage. Si parla di RAM drivers o RAM disk. Memorizzazione temporanea, ma accesso molto rapido: ad esempio per operazioni su file system.

## ❖ Mapping degli indirizzi e struttura disco

Le unità a disco, dal punto di vista logico il calcolatore li vede come grandi **array unidimensionali di blocchi logici** dove il logical block è la più piccola unità di trasferimento. Ogni blocco logico deve essere mappato su un settore del disco o su una pagina di un dispositivo NVM.

- Su Disco blocchi logici mappati sequenzialmente in settori. Blocco 0 è sul Settore 0 più esterno, si va via via verso l'interno.
- Per NVM mapping da una **tupla (chip, blocco, pagina)** ad un blocco logico. Indirizzamento basato su Logical Block Address (LBA). Per HDD (settore, cilindro, testina).

Il passaggio da indirizzi logici a fisici dovrebbe essere facile ma ...

- Settori possono essere danneggiati danneggiati
- Non-constante numero di settori per traccia. Accesso più rapido a tracce esterne.
- Gestione interna del mapping tra LBA e settori fisici
- Si tende a mantenere comunque allineati indirizzi logici e fisici (salgono insieme)

**Struttura disco:** L'array 1-dimensionale di blocchi logici è mappato in settori del disco sequenzialmente. Il numero di settori per traccia varia, esistono più settori nelle tracce esterne che interne dovuto alla geometria intrinseca del disco. Abbiamo una **velocità angolare costante** in cui la velocità di rotazione è costante però la densità con cui si memorizzano i dati è variabile.

## ❖ Scheduling del disco

Per le unità disco vogliamo un veloce accesso e alto trasferimento dati. Il sistema operativo deve occuparsi di questo cercando di **minimizzare il seek time e il latency time**. Di solito si agisce di più sul seek time che sul latency.

- **seek time:** tempo necessario a spostare il **braccio** del disco in corrispondenza del cilindro desiderato
- **disk bandwidth** è il numero totale di bytes transferiti diviso per il tempo totale tra la prima richiesta di servizio ed il completamento dell'ultimo transferimento

Le **richieste** al disco riguardano operazioni di I/O, file (lettura e scrittura) e sono fatte da s.o., processi di sistema e processi utente.

- Il SO mantiene una **coda di richieste**, per disco o dispositivo.
- Un disco in idle può subito servire la richiesta, il disco occupato gestisce una coda di richieste. Bisogna trovare in modo efficiente dalla coda il prossimo lavoro da far svolgere al disco. Gli algoritmi di ottimizzazione hanno un ruolo in questo caso. Oggi molto è gestito dal controllore del disco i quali hanno piccoli buffer e possono gestire code di richieste di I/O.

Non è accessibile la locazione esatta della testina, però si può assumere prossimità tra indirizzi fisici e logici. Molti algoritmi proposti per schedulare il modo in cui sono servite le richieste di I/O sono basati su riduzione del seek time.

## ❖ Algoritmi di Scheduling del disco

- FCFS
- SSTF (Shortest Seek Time First)
- SCAN
- C-SCAN
- C-LOOK

Negli esempi si considerano i cilindri su cui si sposta il disco:

98, 183, 37, 122, 14, 124, 65, 67

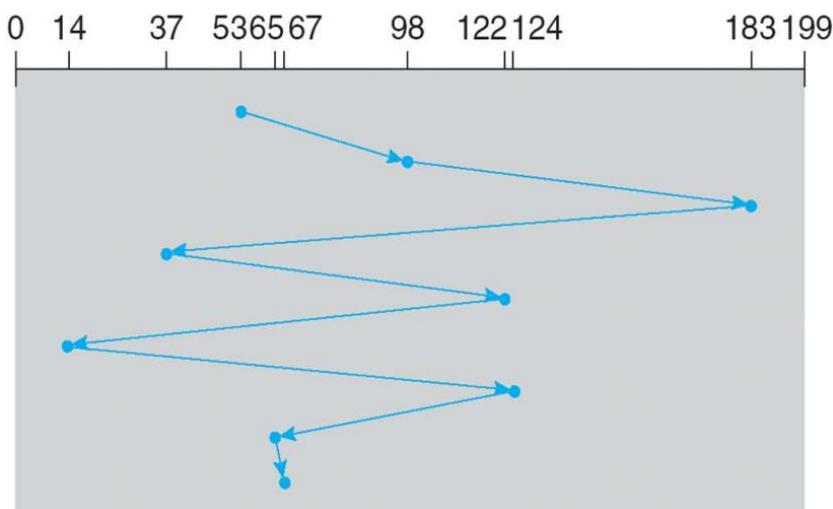
Con testina che punta a 53.

### FCFS

Algoritmo più semplice. Si accumulano nella coda le richieste e si vanno a servire. Non ha problemi di starvation.

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



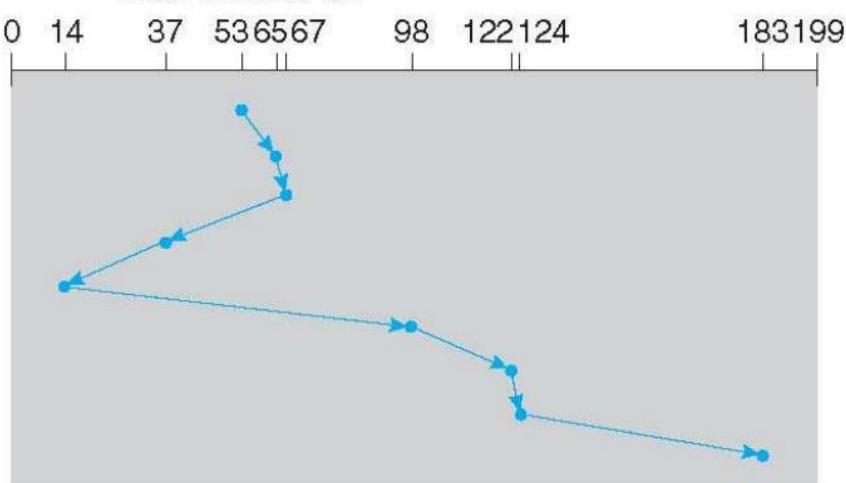
### SSTF (Shortest Seek Time First)

seleziona la richiesta con il minimo seek time dalla corrente posizione della testina; SSTF scheduling è una forma di SJF scheduling; può causare starvation di qualcuna delle richieste.

53,65,67, poi torna indietro a quelli vicini: 37,14, poi va avanti: 98,122,124,183,199

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



## SCAN

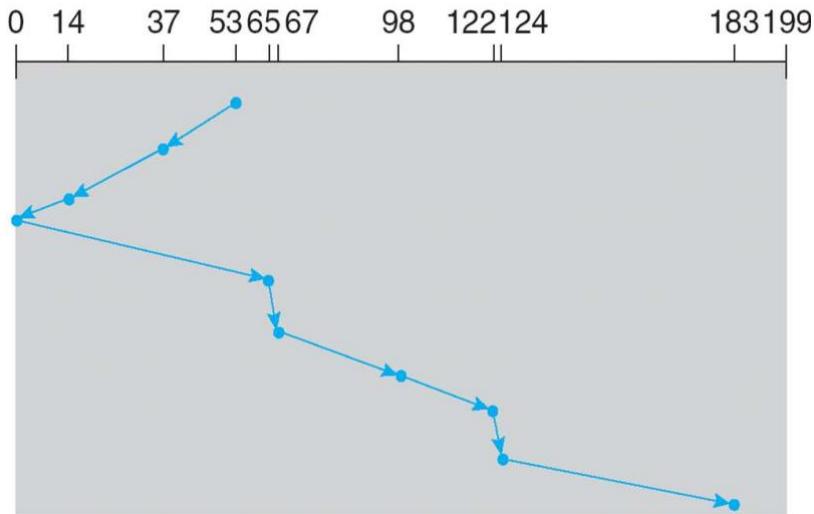
La testina inizia ad un estremo del disco e si muove verso l'altro estremo servendo le richieste, finché non arriva all'altro estremo dove poi inverte il percorso. SCAN algorithm chiamato anche **elevator algorithm**. **Se le richieste non sono uniformemente dense può portare ad attese lunghe, dipende da come è posizionata la testina.**

**Si muove prima all'indietro verso SX poi a DX.**

53, 37, 14, 0 (perché si potrebbero inserire altri processi) , 65, 67, 98, 122, 124, 183

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



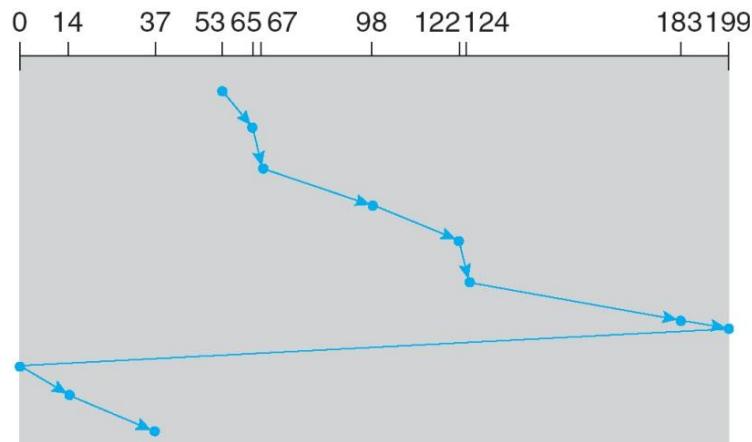
## C-SCAN

Tempi di attesa più uniformi rispetto a SCAN. La testina si muove da un estremo del disco all'altro servendo le richieste nel mentre. Quando raggiunge l'altro estremo, comunque, torna immediatamente all'inizio del disco, senza servire le richieste nel viaggio di ritorno.

**Si muove sempre verso DX.** 53,65,67,98,122,124,183,189, 0 (torna all'inizio) ,14,37

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



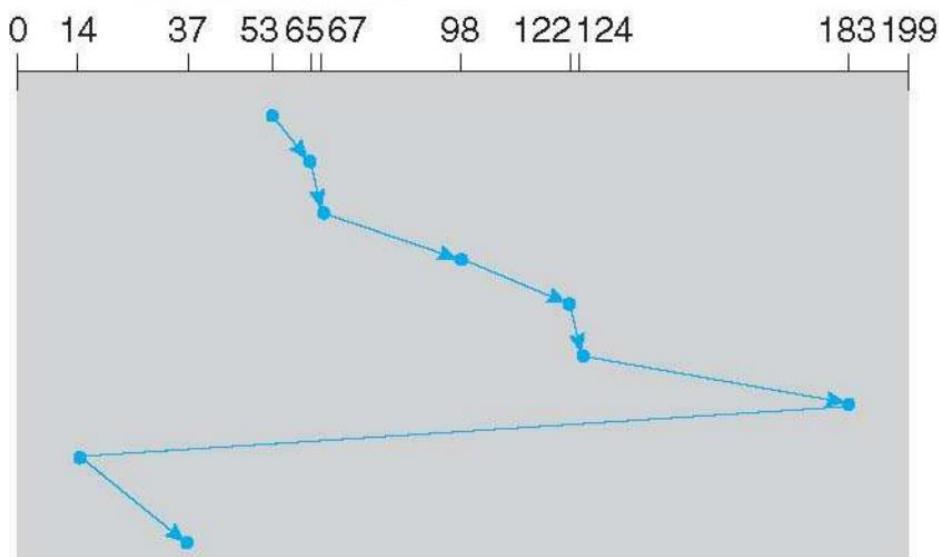
## C-LOOK

LOOK è un tipo di SCAN, C-LOOK è una versione di C-SCAN. Il braccio avanza fino all'ultima richiesta in ogni direzione, poi cambia direzione (non passa per 0)

53,65,67,98,122,124,183,189, 14,37

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



## Selezionare un Algoritmo di Disk-Scheduling

- SSTF (Shortest Seek Time First) è tipico e naturale
- SCAN e C-SCAN funzionano meglio per sistemi con grosso carico su disco. Meno starvation
- Performance dipende dal numero e dai tipi di richiesta
- Algoritmi di disk-scheduling implementati come moduli separati del SO per consentire un aggiornamento se necessario

**Linux** – deadline scheduler con 2 code di read e 2 code di write con priorità e ordine LBA (Logical Block Addressing) implementate in modo simile a C-SCAN. read maggiore priorità perché può bloccare

## NVM scheduling

Lo scheduling HDD deve minimizzare i movimenti meccanici

Nel caso di NVM non ci sono movimenti meccanici e si usa FCFS

## ❖ Gestione del disco

Il Sistema Operativo è responsabile di diverse operazioni per il disco: Inizializzazione, bootstrap, bad-block recovery

**Il dispositivo deve essere strutturato:** HDD in tracce e settori, NVM con pagine e FTL (Flash Transaltion Layer). Ogni **settore di un disco** può mantenere: **un header, dati, più un trailer**. Il trailer contiene un **error correction code (ECC)** per verificare se un settore è danneggiato (Di solito 512 bytes di dati ma si può selezionare).

Dopo la formattazione fisica, Il S.O. può **Partizionare** un disco in uno o più gruppi di cilindri, ognuno considerato come un disco logico separato

Occorrono tecniche per trovare e correggere errori nel disco:

- **Parity Bit:** ad ogni byte viene associato un bit che controlla se il numero di 1 è **pari (parity = 0)** oppure è **dispari (parity = 1)**. Se uno dei bit è danneggiato allora il parity non coincide più con quello precalcolato (incluso il danno dello stesso parity). Se non c'è corrispondenza con un bit si considera quel byte danneggiato. Il parity si calcola rapidamente con lo XOR ( $1 \text{ xor } 1 = 0$ ). Il bit di Parity è un esempio di metodo **checksum** che utilizza metodi aritmetici per controllare dati di lunghezza fissata.
- **Error Correction Code (ECC)** fa anche la correzione. Gli Hard Disk drive usano un ECC per settore, i NVM unsano un ECC per pagina. Quando si scrive un settore/pagina ECC viene scritto. Quando si legge da un settore o pagina si controlla l'ECC, se non coincide con quello scritto allora il settore/pagina è marcato come **bad**. Se esiste un soft error può essere corretto (pochi bit) altrimenti **hard error** e il settore/pagina vengono considerati danneggiati.

Il **controllore del disco** mantiene una **lista dei bad block del disco** e ha a disposizione dei settori di ricambio non visti dal SO. Quando prova ad accedere ad un settore e lo trova corrotto (ECC non corrisponde) riporta l'errore all'SO e marca il settore come bad e lo sostituisce con un settore di richiambio. Quando è richiesto il medesimo blocco logico, questo viene tradotto nel nuovo settore. I settori di ricambio si cercano nello stesso cilindro per ridurre il seek time.

**Strutturazione del disco da parte del sistema operativo:**



- **Partizione del disco** in uno o più gruppi di cilindri, ognuno trattato come un disco logico separato (info su partizioni e su locazione del disco). In Linux **fdisk** per avere info sul dispositivo di storage. Il Sistema crea una entry per quella partizione (in **Linux /dev**). **Montare una partizione** significa rendere il file system disponibile per quel disco logico.
- Il secondo step è la **creazione e gestione di un volume (drive logico)**
- Il terzo step è la **formattazione logica** o la creazione di un file system: Per incrementare l'efficienza i file system raggruppano i blocchi in **clusters**.

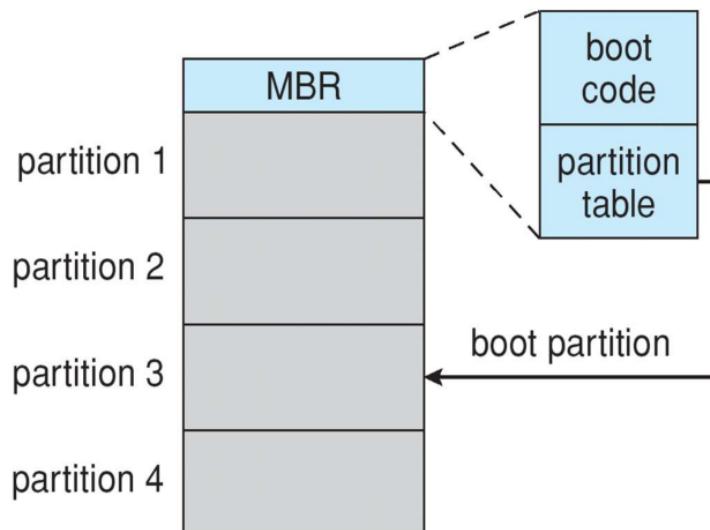
Esiste la possibilità di avere accesso al **raw disk** per applicazioni che richiedono una gestione dei blocchi di memoria dedicata, escludendo la gestione fatta dal SO .

La partizione **Boot block** inizializza il sistema e permette di fare il bootstrap. Il bootstrap è in ROM. Esiste un programma che permette di inizializzare il sistema, il **Bootstrap loader** il quale si trova nei boot block della partizione di boot. La partizione di boot fornisce la root del file system. Il file system di un computer consiste di tutti i volumi montati.

I primi passi del bootstrap sono in firmware. Il Bootstrap loader è un programma che inizializza il SO e porta in memoria il **bootstrap program** dalla memoria secondaria. Il bootstrap program è memorizzato nel “boot block” in una locazione predefinita. Il bootstrap program carica il s.o partendo da una locazione non fissata. Un dispositivo che ha il boot block è un **boot disk** o **system disk**. Il bootstrap program permette allo storage controller di caricare il boot blocks e di iniziare l'esecuzione del codice.

### Esempio Windows

- Boot code nel Master Boot Record (MBR)
- Il Bootstrap loader carica il codice di boot dal MBR (lancia il sistema)
- MBR ha boot code e tabella delle partizioni, da queste si risale alle partizioni di boot



# Gestione dello Swap-Space

---

- Swap-space — Virtual memory usa spazio su disco come estensione della memoria principale
- Lo swap space può variare da pochi megabytes a gigabytes a seconda della dimensione della memoria fisica, di quella virtuale e del modo in cui si usa la VM
  - Solitamente meglio sovrastimare che sottostimare perché se finisce i processi possono essere abortiti o arrivare al crash di Sistema
  - Es. Solaris suggerisce tanto quanto  $SS = VM - FM$ , Linux prima  $SS = FM$  ora meno
  - In Linux multipli swap space su file system o su diverse partizioni
- Swap-space può essere ottenuto dal normale file system, ma più comunemente da una partizione del disco separata (raw)
  - Le partizioni raw permettono accesso più veloce ma è meno efficiente lo stoccaggio (frammentazione interna), cmq dati in swap a breve termine
  - Linux permette swap sia su file system che su partizione raw

## Connessione Dispositivi di Memorizzazione

---

- Il computer può accedere ai dispositivi di memorizzazione in tre modi:

un dispositivo collegato alla macchina

un dispositivo connesso alla rete

un dispositivo cloud

## ❖ Architettura RAID: Mirroring e Disk Striping

I dischi sono soggetti facilmente a guasti e rotture. Devono esserci tecnologie e architetture che mantengono un'alta affidabilità di queste strutture. Multiple unità disco forniscono affidabilità e performance via **ridondanza** in cui i dati si replicano e si parallelizzano operazioni. Un'architettura che fornisce queste caratteristiche è quella RAID.

### RAID – Redundant Array of Independent Disks

La probabilità di fallimento o rottura con N dischi è più bassa, ciò si può stabilire considerando il **mean time between failure (MTBF)**.

Il Modo più semplice di replicare i dati è il **mirroring** che consiste nella duplicazione dei dischi: Ogni disco logico corrisponde a più dischi fisici. Avviene la perdita di un dato solo se anche il secondo disco fallisce prima della riparazione del primo.

**Mean time to repair:** quanto tempo ci vuole per ripristinare la situazione su un disco, tempo critico che espone alla perdita del dato.

**Mean time to data loss:** tempo medio di perdita dei dati dovuti a fallimento

La perdita di un dato con architettura raid a 2 dischi si ottiene solo quando mentre si sta cercando di riparare il primo disco che ha fallito e il secondo disco fallisce anch'esso.

**Disk striping** è una tecnica che usa un gruppo di dischi come un'unica unità di storage. I Dati sono distribuiti (strisciati) su più dischi. Non c'è più ridondanza ma si migliorano le performance.

- **Bit-level striping**, ogni i-esimo bit sul drive i-esimo
- **Block-level striping**, blocco i-esimo su diversi drive

Il disk striping serve per Due obiettivi principali

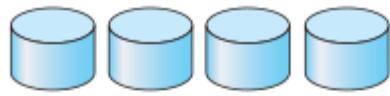
- Incrementare il throughput di piccoli accessi bilanciando il carico
- Ridurre il tempo di risposta di un accesso

**Mirroring porta ridondanza, ma è costoso.**

**Striping aumenta il trasferimento di dati, ma non l'affidabilità perché abbiamo bisogno di più dischi per accedere ad un dato**

si combinano allora queste due tecniche organizzando l'architettura in livelli

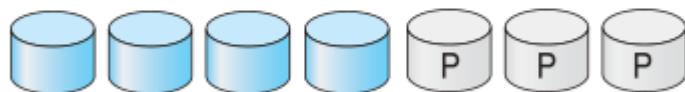
RAID è organizzato in 6 livelli differenti. Gli schemi RAID aumentano le performance e l'affidabilità dello storage memorizzando dati ridondanti.



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



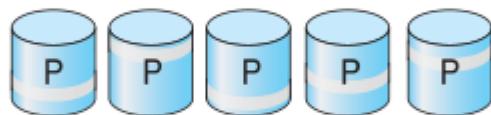
(c) RAID 2: memory-style error-correcting codes.



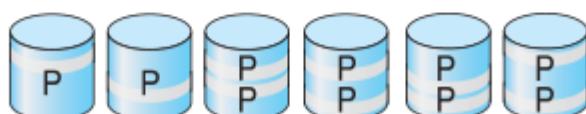
(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

- Es. 4 dischi per i dati, gli altri per ridondanza
  - Livello 0 non ridondanza, molto fragile
    - ▶ No fault tolerance: se un disco fallisce dati persi
    - ▶ Veloce, parallelismo, no parity control
    - ▶ Minimo numero di dischi 2
  - Livello 1 mirrored (C copia)
    - ▶ Dati duplicati, non overhead in velocità di scrittura
    - ▶ Velocizza la lettura (dati da più dischi)
    - ▶ Metà della capacità del disco
    - ▶ Minimo numero di dischi 2
- RAID 2 poco utilizzato
  - Utilizza striping al livello di bit
  - Utilizza hamming code per error detection/correction
  - Minimo numero di dischi 3
- RAID 3 poco utilizzato
  - Utilizza byte level striping
  - Con un parity disk
  - Minimo numero dischi 3
- RAID 4
  - Utilizza block level striping
  - Con parity disk
  - Scrittura lenta, scrittura del parity su unico disco
  - Minimo numero dischi 3
- RAID 5 distribuisce i parity su tutti i dischi, evita di usare troppo il disco parity
  - Minimo numero di dischi 3
  - Block level striping + ECC distribuito
  - ECC usato con stripping
    - ▶ Il primo blocco in drive 1, scondo in drive 2, N in drive N, error in N + 1
  - Permette anche la correzione dei dati
  - Più veloce del livello 1 nell'accesso
  - Scrittura migliore di livello 4
- RAID 6 simile
  - Minimo numero di dischi 4
  - aggiunge ridondanza per permettere recovery da fallimenti multipli
  - 2 blocchi di parity distribuiti nei dischi
  - Tollera due fallimenti di dischi
  - Penalizzazione in scrittura

## **10. SISTEMI I/O**

### ❖ Concetti principali

la gestione dell' i/o è uno dei compiti fondamentali di un os. la gestione dell'i/o rappresenta un importante aspetto delle operazioni di un computer e incide sulle prestazioni. i dispositivi di i/o variano molto, ma nello stesso tempo presentano spesso interfacce standard che nascondono la loro complessità. esistono vari metodi per controllare l'i/o. il s.o. si deve occupare della gestione delle performance dei dispositivi di i/o per evitare che facciano da collo di bottiglia dell'intero sistema.

Porte, bus, controllori di dispositivo connettono i vari dispositivi.

- I **Device driver** sono interfacce che incapsulano dettagli dei vari dispositivi di I/O fornendo un accesso uniforme ai sottosistemi I/O. Si interfaccia con il controller dei dispositivi.
- **Porta**: punto di connessione con il computer
- **Bus**: collegati in modo daisy chain. **PCI** per dispositivi veloci, **expansion bus** connette i dispositivi "lenti".
- **Controller (host adapter)**: è un componente elettrico che gestisce porte, bus, dispositivi. A volte sono integrati, a volte circuito separato (host adapter).

**Il processore comunica con i controller di un dispositivo I/O attraverso registri di dati e passando comandi ai controllori dei dispositivi.**

Il controllore di dispositivo può supportare l'**I/O memory mapped** (I/O mappato in memoria). In questa modalità di comunicazione con la cpu i registri di controllo del dispositivo sono mappati in un sottoinsieme dello spazio di indirizzi della CPU. La CPU esegue le richieste di I/O leggendo e scrivendo i registri di controllo del dispositivo alle locazioni di memoria fisica a cui sono mappati.

### ❖ Polling/Interrupt

Interrupt e polling sono due modalità con cui gli eventi generati dai dispositivi I/O possono essere gestiti dalla CPU.

- Con il **polling**, la CPU tiene traccia delle comunicazioni dei dispositivi di I/O interrogandoli ad intervalli regolari per vedere se hanno bisogno di andare in esecuzione. **La CPU si pone in un un ciclo di busy-wait in attesa di I/O dal device.**
- Con **interrupt**, è il dispositivo di I/O ad interrompere la CPU comunicando ad essa che ha bisogno di andare in esecuzione. La CPU poi esegue una routine di servizio per la gestione dell'interruzione. La CPU ha una **Interrupt-request line** che viene controllata da esso dopo ogni istruzione. Quando il controller ha finito egli manda un'interrupt alla cpu sulla irq. La gestione dell' interrupt è fatta da un **Interrupt handler routine**: Controlla la causa dell'interrupt, la serve, e ripristina lo stato. L'**interrupt vector** contiene la routine giusta per servire un particolare interrupt. Gli interrupt hanno due linee: **maskable** (possono essere posticipati) e **non maskable**. Un interrupt può avere **livelli di priorità**: un interrupt a più alto livello può prelazionarne uno di più basso.

## ❖ DMA

L'approccio DMA rappresenta la soluzione più efficiente per l'interfacciamento di una CPU con un insieme di periferiche di I/O. Consiste nell'introduzione di un circuito di controllo, denominato **DMAC** (DMA Controller), che, una volta programmato, **gestisce le periferiche senza coinvolgere la CPU**. Bypassa la CPU per trasferire dati direttamente tra dispositivi di i/o e memoria.

Il DMAC è estremamente utile in tutti i casi in cui un **dispositivo di i/o** renda disponibili **grandi quantità di dati da spostare verso la memoria**. Il DMAC provvede a questi spostamenti controllando i bus dati e indirizzi in modo autonomo, lasciando la **CPU libera** di eseguire altre operazioni.

L'operazione di spostamento dati può avvenire in:

- **modo singolo (cycle stealing);**
- **blocchi (burst).**

Mentre il DMAC è operativo la CPU non può usare il bus, quindi operare sulla memoria dati. Quando ha finito, interrupt per segnalare il completamento.

La gestione del DMAC consiste semplicemente nell'indicazione di un indirizzo di memoria a partire dal quale si scriveranno i dati e del numero di dati da spostare. Al termine dello spostamento il DMAC genera un'interruzione.

## Flusso operativo: Handshaking tra Device Controller e DMA

1. la periferica tramite il device controller richiede al DMAC un trasferimento dati
2. il DMAC richiede alla CPU il permesso di usare i bus (tramite il segnale **DMARequest**)
3. la CPU concede l'uso del Bus: gli switch isolano la CPU e connettono DMAC e periferica alla memoria. Il DMAC imposta l'indirizzo sul bus e autorizza la periferica a leggere/scrivere il dato (inviando il segnale **DMA Acknowledge**) in memoria.
4. Una volta terminato il trasferimento il DMAC rilascia il Bus che torna in possesso della CPU generando un interrupt

## ❖ Interfaccia per le applicazioni I/O

Abbiamo bisogno di un Interfaccia uniforme per gestire I/O molto differenti. Usiamo Astrazione, encapsulamento e stratificazione. I/O system call encapsulano comportamenti dei device in classi generiche. Device-driver layer nascondono le differenze tra gli I/O controller dal kernel. Ogni SO ha i suoi sottosistemi I/O e device driver

Device driver variano in diverse dimensioni

### ▫ Character-stream o block

- byte a byte o a blocchi come unità di trasferimento

### ▫ Sequential o random-access

- accesso sequenziale ai dati del dispositivo o qualunque

### ▫ Synchronous o asynchronous (o entrambi)

- Trasferimento dati sincronizzato (tempi prevedibili) altrimenti irregolare e non coordinato

### ▫ Sharable o dedicated

- Può essere usato contemporaneamente da più thread o no?

### ▫ Speed of operation

- Velocità di trasferimento da pochi byte a Gb per secondo

### ▫ read-write, read only, o write only

- alcuni dispositivi permettono solo lettura altri scritti solo una volta, etc.

# Nonblocking e Asynchronous I/O

---

- **Blocking** - processi sospesi finché I/O non è completato
  - Facile da usare e capire
  - Problematico in alcune circostanze
- **Nonblocking** - I/O call ritorna appena possibile
  - User interface (keyboard and screen)
  - data copy (buffered I/O)
  - Implementata via multi-threading
  - Ritorna rapidamente con la conta dei bytes read o written
  - `select()` per vedere se dati pronti poi `read()` o `write()` per il trasferimento
- **Asynchronous** - processo gira mentre I/O esegue
  - Difficile da usare
  - I/O subsystem segnala al processo quando I/O è completata

## Sottosistema I/O del Kernel

---

- Il Kernel fornisce molti servizi per la gestione dell'I/O che sono realizzati a partire dai dispositivi e dei driver relativi

scheduling

gestione del  
buffer

gestione delle  
cache

gestione delle  
code di spooling

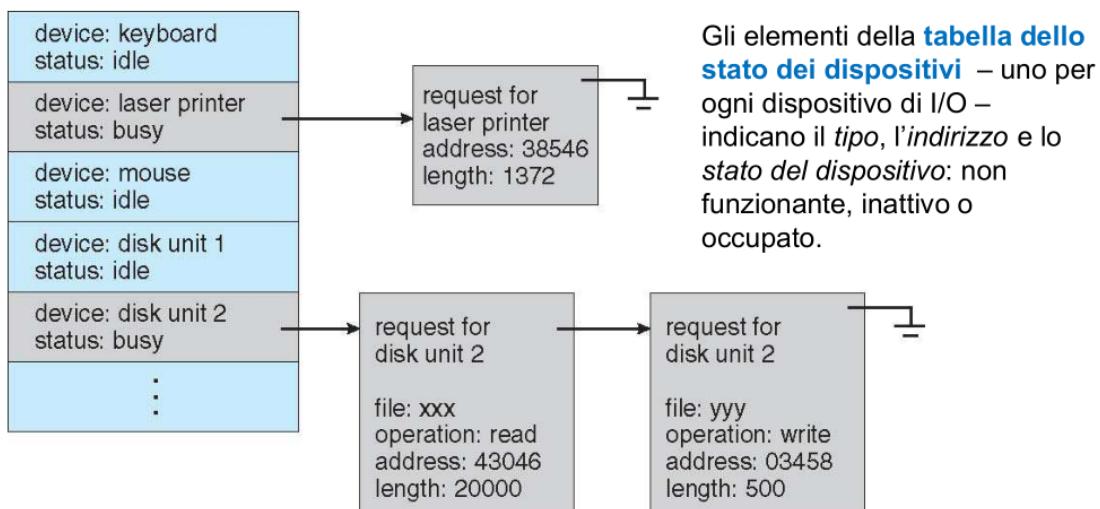
riservazione dei  
dispositivi

gestione degli  
errori →  
protezione  
dell'I/O

# Sottosistema I/O del Kernel

## □ I/O Scheduling

- Riordina le richieste di I/O sulle code per-device
- Alcuni SO mantengono un Quality Of Service (i.e., IPQOS) per le richieste oppure precedenza a richieste delay-sensitive
- SO deve mantenere traccia di diverse richieste asincrone, per questo mantiene le code su tabella dello stato dei dispositivi (device status table)



## **11. FILE SYSTEM**

- Un **File** è un'unità logica di immagazzinamento dati presenti in memoria secondaria. Esso fornisce una **Vista uniforme** sull'informazione memorizzata in memoria non volatile. I dati di un file fanno riferimento ad uno **Spazio contiguo di indirizzi logici**.

**Diversi tipi di dato:** numerico, caratteri, binari, programmi (sorgente, eseguibile, etc.). Si considerano text file, source file, executable file

Un file è associato ad un **insieme di attributi**:

- **Name** – denominazione (per utente umano)
- **Identifier** – tag unico (numero) che identifica il file nel file system
- **Type** – necessario per sistemi che supportano tipi diversi
- **Location** – pointer alla locazione del file sul device
- **Size** – dimensione corrente
- **Protection** – chi può fare reading, writing, executing
- **Time, date, and user identification** – dati per protection, security e usage monitoring (creazione, modifica, uso)

Un file è quindi un tipo di dato astratto. Su di esso si possono definire delle **operazioni** (associate a system call):

- **Create** - si compone di due passi: allocazione spazio per il file, sua creazione nella struttura a directory
- **Open(Fi)** – cerca nelle directory su disco per la entry corrispondente al file Fi, controlla i permessi, restituisce handler al file per le altre operazioni
- **Close (Fi)** – chiude il file
- **Write** – per la scrittura mantiene un write pointer
- **Read** – mantiene un read pointer
- **Reposition within file** - seek
- **Delete** – (solo dopo che tutti gli hard link cancellati)
- **Truncate** – cancellazione del file mantenendo gli attributi, size diventa zero

Con **open** si mantengono le **info sul file aperto**:

- **Open-file table**: il Sistema Operativo traccia tutti file aperti, i processi hanno info sui loro file aperti (due tabelle, per process e per sistema)
- Chiusura file rimuove file aperto
- **File-open count**: contatore del numero di volte che il file è open
- **File pointer**: per i sistemi che non usano offset, punta all'ultima locatione di read/write (mantenuto per processo che ha il file aperto)
- **Locazione del file su disco**: informazione per l'accesso diretto al file, senza dover cercare attraverso le directory
- **Permessi**: informazione su modalità di accesso per-processo

# Open File Locking

---

- Fornito da alcuni SO e file system
  - Simile a read-write locks
  - **Shared lock** simile a reader lock – molti processi possono acquisire concorrentemente
  - **Exclusive lock** simile a writer lock
- Mandatory o advisory:
  - **Mandatory** – accesso è negato a seconda dei lock mantenuti e richiesti (Win)
  - **Advisory** – processo può leggere lo status dei lock e decidere che fare (UNIX) lasciato al programmatore

## Struttura dei File

---

- Se si assumono diverse strutture di file, ogni tipo richiede algoritmi diversi
- Struttura minimale (UNIX, Windows)
- UNIX solo una – sequenza di bytes (programmi gestiscono da soli)
  - Ogni byte del file viene indirizzato con offset,
  - ogni blocco logico è 1 byte che è contenuto su un blocco del disco
- I file si definiscono in blocchi logici, ma si mappano su blocchi del disco
  - Se file di 1949 byte con blocchi di 512 byte su disco, si allocano 4 blocchi, cioè 2048 byte con spreco di 99 (frammentazione interna)

# Metodi di Accesso

---

- Diverse possono essere le modalità di accesso di un file
- **Accesso sequenziale**

```
read next  
write next  
reset
```

- **Accesso diretto (relativo) – modello disco**  
file è assunto di lunghezza fissata composto di [logical records](#)

```
read n  
write n  
position to n  
      read next  
      write next  
rewrite n
```

*n* = numero di blocco relativo

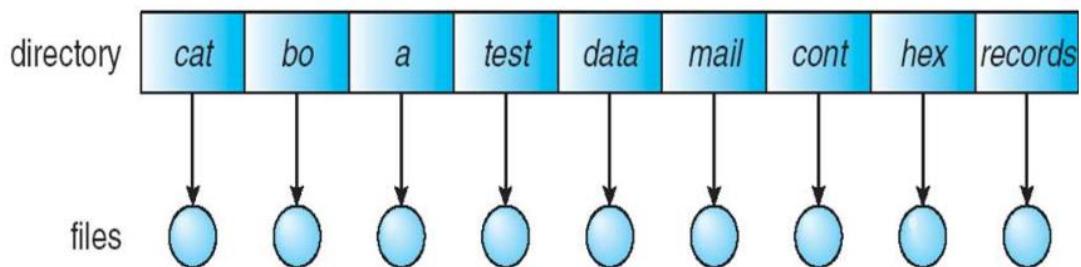
## Altri Metodi di Accesso

---

- Con accesso diretto si possono definire altri metodi di accesso
- Creazione di [indici](#) per i file
  - Mantiene indici in memoria per un veloce determinazione della locazione dei blocchi da processare (puntatori ai blocchi)
  - Prima cerca indice, poi blocco, poi record
  - Ricerca veloce per file grandi senza troppi accessi
  - Se troppo grande, indice (in memoria) dell'indice (su disco)

# Single-Level Directory

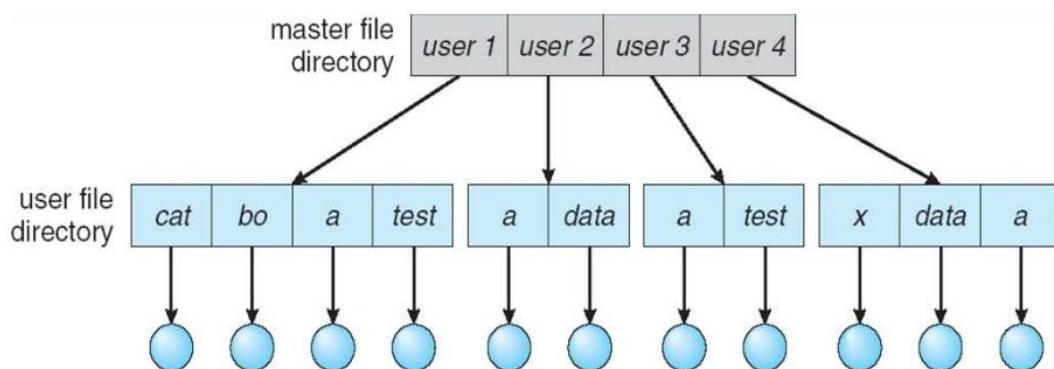
- Una singola directory per tutti gli utenti



- Problema del naming
- Problema del Grouping

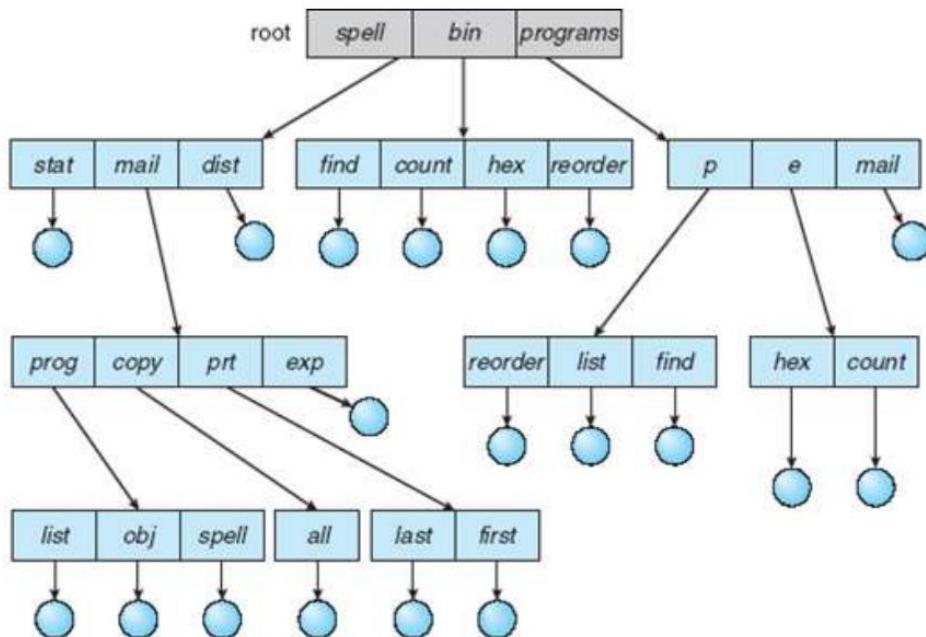
# Two-Level Directory

- Directory separate per ogni user

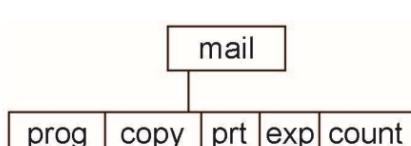


- Path name
- Si può avere lo stesso nome di file name per utenti differenti
- Ricerca efficiente
- Non si può fare grouping

# Tree-Structured Directory



- Ricerca efficiente
- Capacità di grouping
- path name **assoluti** o **relativi**
- Creazione di un nuovo file fatta nella directory corrente
- Cancellazione di file
  - `rm <file-name>`
- Creazione di una nuova subdirectory è fatto nella directory corrente
  - `mkdir <dir-name>`
- Esempio: se la directory corrente è `/mail`
  - `mkdir count`

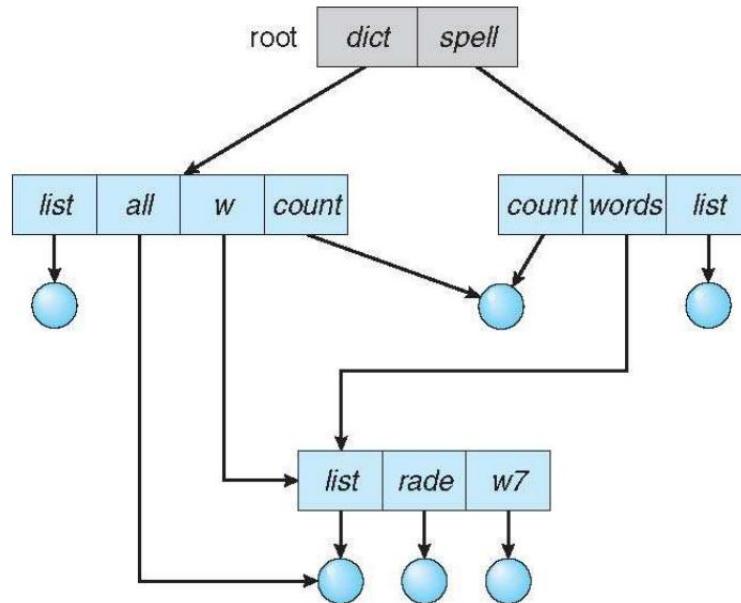


Cancellando "mail"  $\Rightarrow$  si cancella l'intero subtree con radice in "mail"

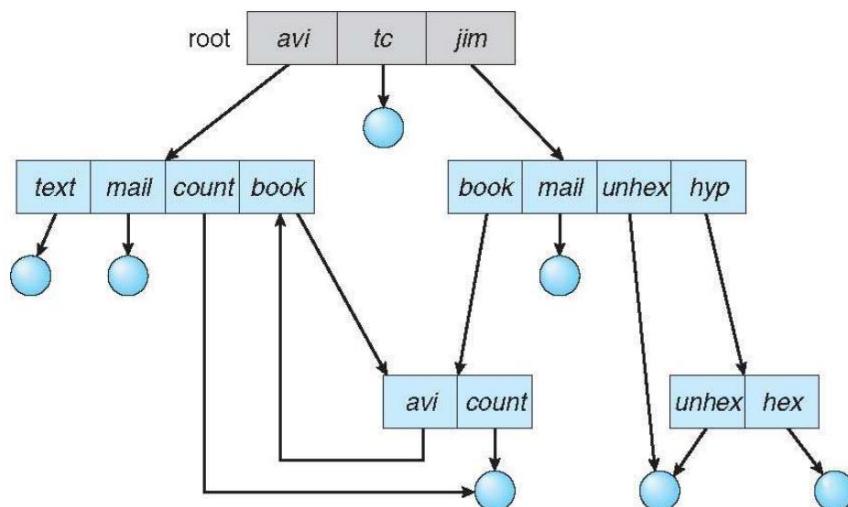
Accesso di utenti nelle directory di altri consentita

# Acyclic-Graph Directory

- Ha subdirectory e file condivisi



# General Graph Directory



Struttura gerarchica

Files senza struttura (byte streams)

Protezione da accessi non autorizzati

Semplicità di struttura

"On a UNIX system, everything is a file; if something is not a file, it is a process."

I tipi principali di File sono:

**File ordinari**

**Directory**

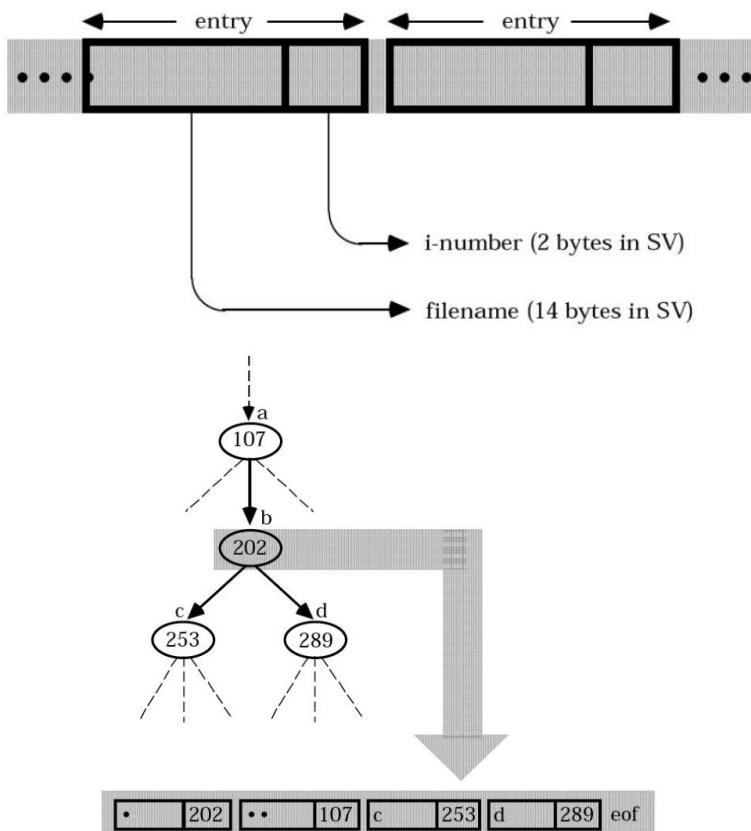
**File Speciali**

Il sistema assegna biunivocamente a ciascun file un identificatore numerico, detto i-number ("index-number"), che gli permette di rintracciarlo nel file system.

**File Ordinari:** Sono sequenze di byte (byte streams). Possono contenere informazioni qualsiasi (dati, programmi sorgente, programmi oggetto,...) Il sistema non impone alcuna struttura

Per consentire all'utente di rintracciare facilmente i propri files, Unix permette di raggrupparli in cartelle, dette **Directories**, organizzate in una (unica) struttura gerarchica. Le **directories** Sono sequenze di bytes come i file ordinari; Differiscono dai file ordinari solo perché non possono essere scritte da programmi ordinari. Il loro contenuto è una serie di directory entries: associazione fra gli i-number (usati dal sistema) e i filename mnemonicici (usati dall'utente)

Un file può avere più filename (ma sempre un solo i-number)



Almeno due entry: la directory stessa “.”, la directory padre “..”

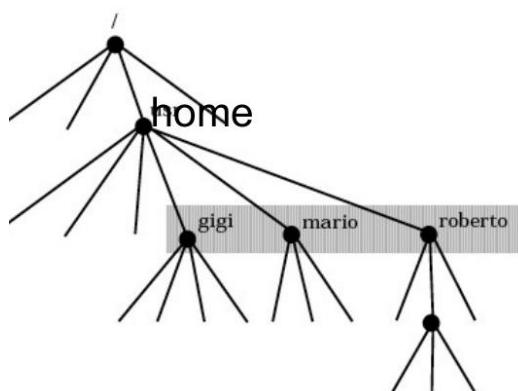
**Pathnames:** Ogni file viene identificato univocamente specificando il suo pathname, che individua il cammino dalla root-directory al file

### Tipiche Directories in Linux

/bin = comandi eseguibili  
/dev = file speciali (I/O devices)  
/etc = file per l'amministrazione del sistema, ad esempio: /etc/passwd  
/lib = librerie di programmi  
/tmp = area temporanea usata dal sistema  
/home = home directories  
/usr = Programmi, librerie, doc. etc. per i programmi user-related.

### Home Directory

Ad ogni utente viene assegnata dal system administrator una directory proprietà (home directory) che ha come nome lo username del proprietario; Ad essa, l'utente potrà appendere files (o subdir): Per denotare la propria home directory si può usare l'abbreviazione "~"



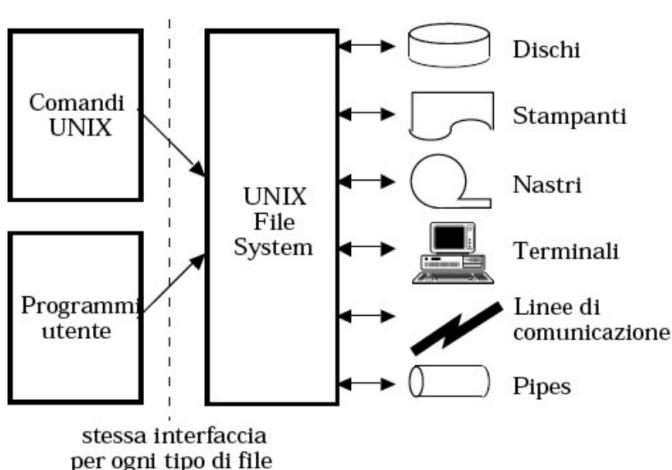
**Working Directory:** Ogni utente opera, ad ogni istante, su una directory corrente, o working directory. Subito dopo il login, la working directory è la home directory dell'utente. L'utente può cambiare la working directory con il comando (cd change directory)

**Ogni file può essere identificato univocamente specificando solamente il suo pathname relativo alla working directory**

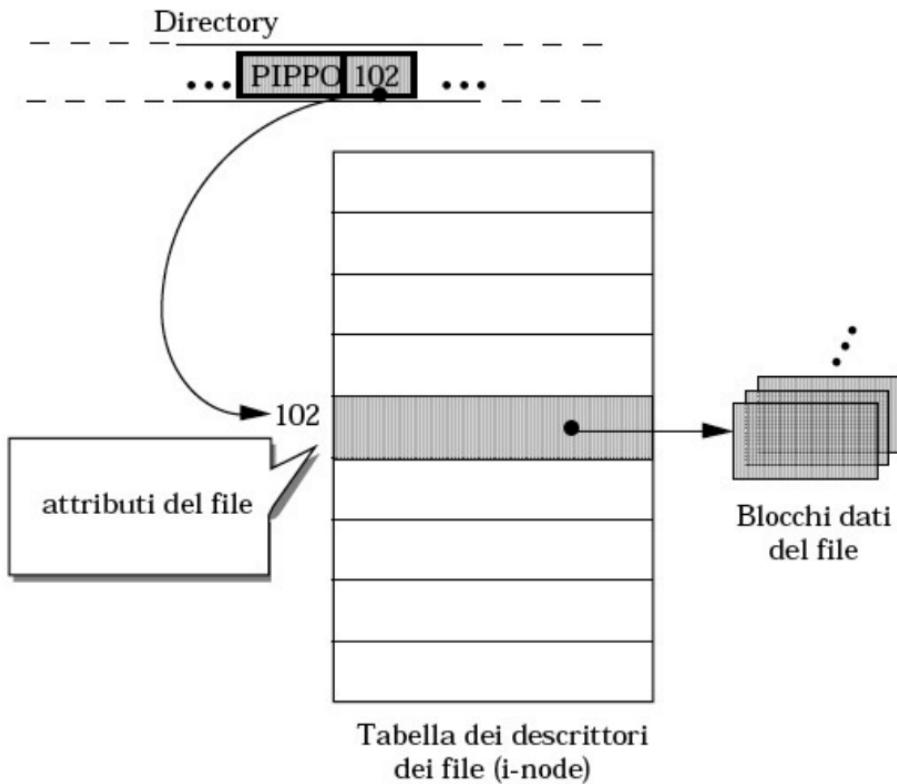
### Files Speciali

Ogni device di I/O viene visto, a tutti gli effetti, come un file (file speciale). Richieste di lettura/scrittura da/a files speciali causano operazioni di input/output dai/ai devices associati

**Vantaggi:** Trattamento uniforme di files e devices. In Unix i programmi non sanno se operano su un file o su un device



# Implementazione File



## Attributi di un File in UNIX

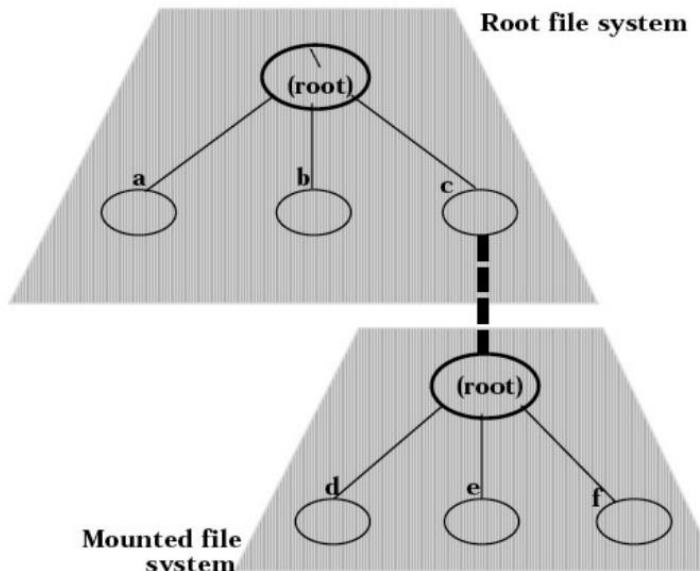
Per ogni file (ordinario, directory, speciale) Unix mantiene le seguenti informazioni nel descrittore del file:

<b>Tipo</b>	ordinario, directory, speciale?
<b>Posizione</b>	dove si trova?
<b>Dimensione</b>	quanto è grande?
<b>Numero di links</b>	quanti nomi ha?
<b>Proprietario</b>	chi lo possiede?
<b>Permessi</b>	chi può usarlo e come?
<b>Creazione</b>	quando è stato creato?
<b>Modifica</b>	quando è stato modificato più di recente?
<b>Accesso</b>	quando è stato l'accesso più recente?

# File System Montabile

Un file system Unix è sempre **unico**, ma può avere parti residenti su device rimuovibili:

- "montate" prima di potervi accedere (mount)
- "smontate" prima di rimuovere il supporto (umount)



Un file system deve essere montato prima dell'accesso. Un file system è montato su un mount point

## Gestione delle Directories

**pwd** print working directory: Stampa pathname directory corrente

**cd** change directory: La directory specificata diviene la working directory. se nessuna directory specificata, si "ritorna" alla home directory

**ls** list directory: lista (in ordine alfabetico) il contenuto della o delle directories indicate. se nessuna directory indicata, lista il contenuto della working directory. possiede numerose opzioni:

- s** fornisce la dimensione in blocchi (size)
  - t** lista nell'ordine di modifica (prima il file modificato per ultimo) (time)
  - 1** un nome per ogni riga
  - F** aggiunge / al nome delle directory e \* al nome dei files eseguibili
  - R** si chiama ls ricorsivamente su ogni sottodirectory
  - i** fornisce l'i-number del file
- I files il cui nome inizia con "." vengono listati solo specificando l'opzione -a ("all")

```

% ls
dir1 file1
% ls -s
total 4 2 dir1 2 file1
% ls -t
file1 dir1
% ls -1
dir1
file1
% ls -F
dir1/ file1
% ls -R
dir1 file1
./dir1:
file1 file2 file3 file4
% ls -i
199742 dir1 51204 file1
%
```

Totale dimensione occupata (in blocchi)

	Riferimenti al file	Dimensione (byte)	Nome
Tip	Proprietario	Gruppo primario	Data ultima modifica
lso:~>ls -l			
total 12			
-rw-rw-r--	1 lso	lso	10 Mar 4 13:29 a
-rw-rw-r--	1 lso	lso	10 Mar 4 14:12 b
drwxrwxr-x	2 lso	lso	4096 Mar 4 14:29 c

Permessi  
(r)ead, (w)rite, e(x)ecute

(d)irectory, (l)ink, (c)haracter special file, (b)lock special file, (-) ordinary file

**du** disk usage

**mkdir** make directory

**rmdir** remove directory

**rmdir** directory ... : rimuove la/le directory (deve essere vuota)

```
% rmdir dir
```

```
rmdir: dir: Directory not empty
```

```
% ls dir
```

```
a
```

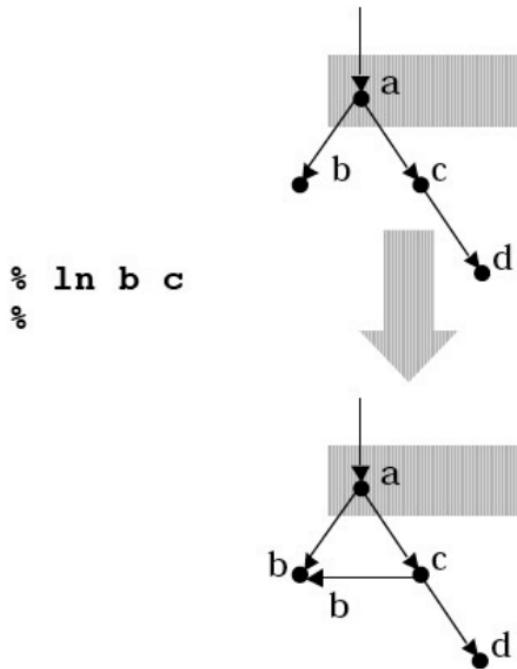
```
% rm dir/a
```

```
% rmdir dir
```

**ln link**

**ln name1 name2**

associa il nuovo nome (link) name2 al file (esistente) name1, che non può essere una directory



Tutti i link allo stesso file hanno identico status e caratteristiche. Non è possibile distinguere la entry originaria dai nuovi link- I link di questo tipo non possono essere fatti con file che stanno su file system diversi

## Link Simbolici

---

- **ln -s name1 name2**
- Permette di creare link a directory;
- Permette di creare link fra file o directory che stanno su file system diversi;
- Viene creato un file name2 che contiene il link simbolico (i.e. il path di name1)

## Esempio:

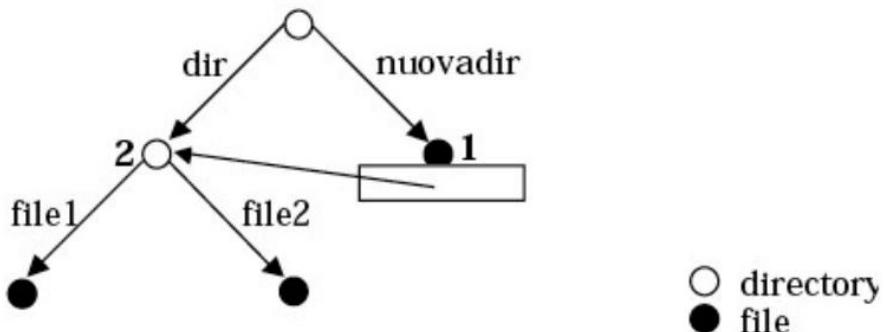
```
% ls                                C'e' una directory ...
dir

% ls dir                            ... contenuto della directory...
file1    file2

% ln -s dir nuovadir      ...link simbolico a dir da nuovadir
% ls
dir      nuovadir

% ls nuovadir
file1    file2

% ls -l          ...dir con 2 rif. nuovadir con 1 rif.
total 4
drwxr-sr-x  2 roberto  usrmail  512 Mar 11 19:24
dir
lrwxrwxrwx  1 roberto  usrmail      3 Mar 11 19:24
nuovadir -> dir
%
```



## Comando mv

---

- **mv [options] name...target**

1. muove il file o directory name sotto la directory target;
2. se name e target non sono directories, il contenuto di target viene sostituito dal contenuto di name

Comando cp

cp [options][name...] target

come mv, ma name viene copiato

## Comando rm

---

- **rm [-r] name...**
- rimuove i files indicati
- se un file indicato è una directory: messaggio di errore, a meno che non sia specificata l'opzione -r
  - ... nel qual caso, rimuove ricorsivamente il contenuto della direttrice

## Protezioni

I possessori/creatori di File devono controllare:

Cosa può essere fatti

Da chi

Tipi di accesso

Read

Write

Execute

Append

Delete

List

- Modelli di accesso: read, write, execute
- Tre classi di utenti in Unix / Linux

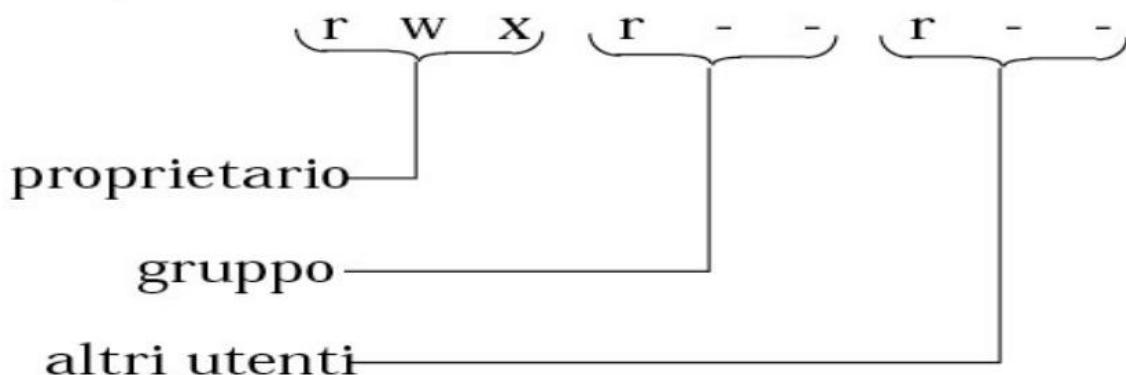
		RWX
a) owner access	7	⇒ 1 1 1
b) group access	6	⇒ 1 1 0
c) public access	1	⇒ 0 0 1

- Chiedi al manager di creare un gruppo (nome unico), diciamo G, e aggiungi qualche utente al gruppo
- Per un file particolare (es., *game*) o subdirectory si definisce un accesso appropriato (es., Solaris)

**Ad un file possono essere attribuiti i seguenti permessi:**

r : readable      w : writable      x : executable      } per      { proprietario      gruppo      altri utenti

**Esempio:**



## Permessi iniziali

---

- Alla creazione di un file, Unix assegna i seguenti permessi:

- Per i *files ordinari non eseguibili*:

rw-rw-rw

110 110 110

6 6 6

- Per i *files ordinari eseguibili* e per directories:

rwx rwx rwx

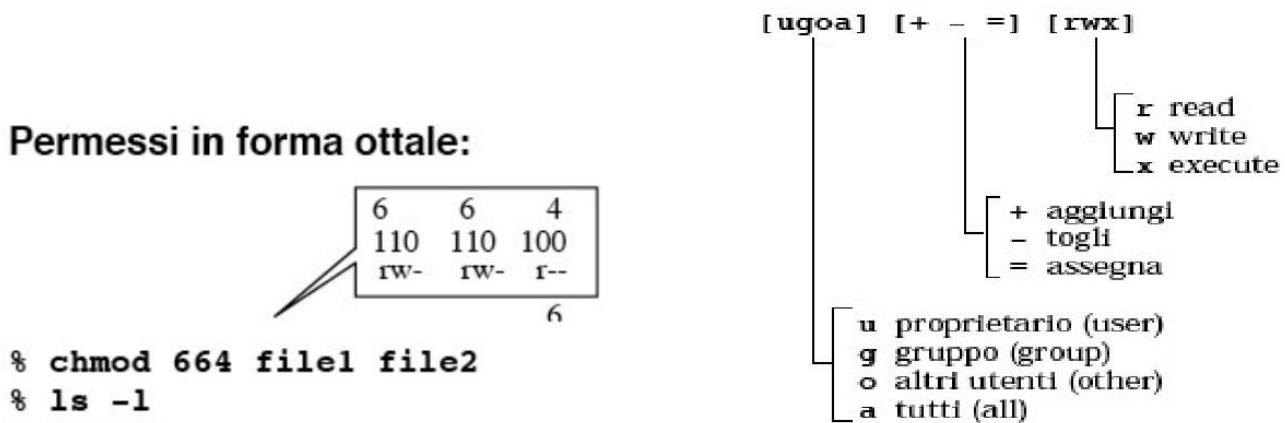
111 111 111

7 7 7

**chmod** *permissions filename...*

"change mode"

- attribuisce le *permissions* a *filename* (solo da parte del proprietario del file!)
- *permissions* può essere espresso in forma ottale o simbolica



```
% chmod 664 file1 file2
% ls -l
total 4
-rw-rw-r-- 1 roberto  usrmail  35 Mar 11 16:34 file1
-rw-rw-r-- 1 roberto  usrmail  17 Mar 11 16:17 file2
%
```

### Permessi in forma simbolica:

```
% ls -l
total 4
-rw-rw-r-- 1 roberto  usrmail  35 Mar 11 16:34 file1
-rw-rw-r-- 1 roberto  usrmail  17 Mar 11 16:17 file2
% chmod ugo+x file1
% chmod o=rwx file2
% ls -l
total 4p
-rwxrwxr-x 1 roberto  usrmail  17 Mar 11 16:34 file1
-rw-rw-rwx 1 roberto  usrmail  17 Mar 11 16:17 file2
```

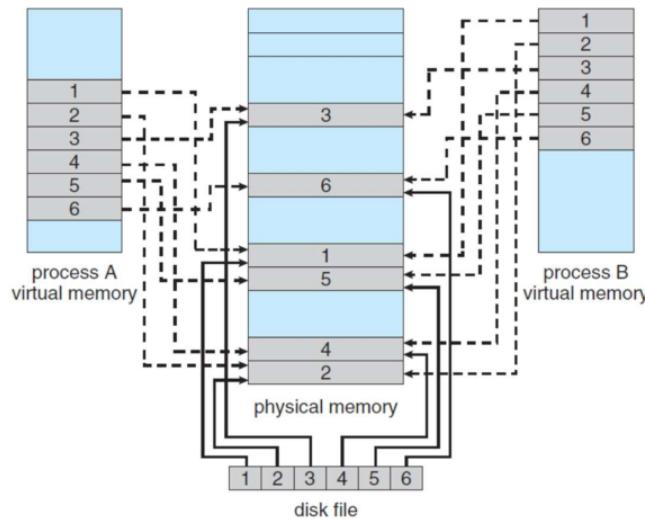
## Memory Mapped File

---

- Con il metodo memory mapped una parte dell'indirizzo virtuale viene usato per accedere ai file
- Si evita di fare accesso per ogni operazione (open, read, write, etc.)
- Blocco del disco mappato su una pagina in memoria
  - Al primo accesso page fault, poi funziona come accesso normale
  - La scrittura su disco non è immediata, in alcuni casi quando è chiuso il file
  - Alcuni sistemi solo con specifiche system call fanno memory mapping
  - Altri invece trattano scrittura come memory mapped
    - Solaris, distingue: mmap in user space se esplicito, altrimenti nel Kernel

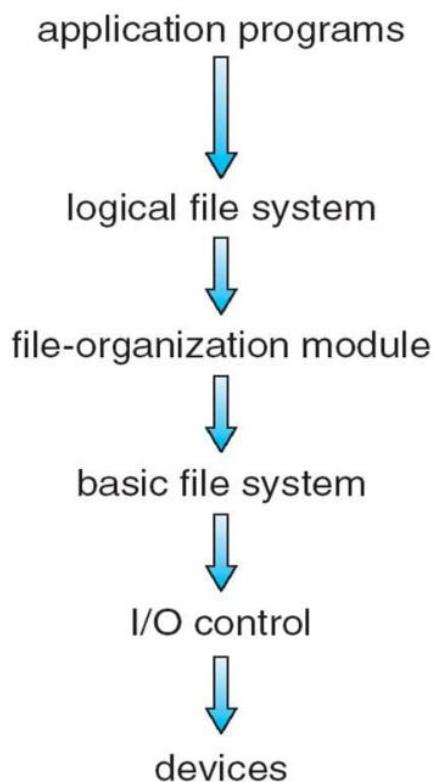
# Memory Mapped File

- Con il metodo memory mapped una parte dell'indirizzo virtuale usato per accedere ai file
- Disk block mappato su una pagina in memoria
  - Memory mapping come meccanismo che supporta anche la shared memory
  - Le pagine in memoria virtuale puntano alle stesse pagine in memoria fisica



# Struttura del File-System

- Il file system risiede permanentemente in memoria di massa
- Tipicamente questa memoria è fornita dai dischi, i quali vengono usati per due principali motivi:
  - Possono essere riscritti localmente (leggo e riscrivo lo stesso blocco)
  - Permettono l'accesso, sia sequenziale che diretto, ai blocchi fisici che memorizzano i diversi file
- La dimensione dei blocchi varia da dispositivo in dispositivo
  - Da 32 a 4096 Byte
  - Solitamente 512 Byte, 4096 per NVM
- Il file system sono tipicamente realizzati secondo una struttura a strati
- Il livello superiore si basa sui servizi offerti dal livello inferiore



## Livelli di un file system:

- **Controllo dell'I/O**: driver dei dispositivi e gestore degli interrupt
  - ▶ Si occupa del trasferimento dell'informazione dalla memoria di massa a quella centrale
- **File System di base**: invia dei comandi di base al controllo dell'I/O per scrivere e leggere blocchi fisici
  - ▶ E.g. Leggi il blocco nell'unità 1, cilindro 12, superficie 4, settore 2
- **Modulo di organizzazione dei file**: conoscendo il metodo di allocazione traduce un indirizzo di un blocco logico in un indirizzo di un blocco fisico
  - ▶ Comprende anche il gestore dello spazio libero (mantiene la lista dei blocchi liberi)
- **File System Logico**: gestisce i metadati e la struttura delle directory
  - ▶ Gli attributi dei file vengono salvati in File Control Blocks (FCB)
- **Device driver** gestiscono i dispositivi I/O al livello di controllo I/O
  - Da comandi come “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” genera comandi hardware specifici per il controllore del hardware
- **Basic file system** comandi del tipo “prendi il blocco 123” li traduce in comandi generici per i device driver
  - Comandi con indirizzi logici, schedulazione delle richieste I/O
  - Gestisce anche buffer e cache (allocazione, freeing, replacement)
  - Buffer mantengono data in transito, caches mantengono metadati frequentemente usati
- **File organization module** livello di gestione dei file, indirizzi logici e blocchi fisici
  - Ogni blocco è numerato da 0 a N
  - Traduce un blocco logico # in un blocco fisico #
  - Gestisce lo spazio libero e l'allocazione del disco

- **Logical File System** gestisce informazione sui metadati
  - Traduce i nomi di file in file number, file handle, locazioni mantenendo i File Control Blocks (**inodes** in UNIX)
  - Gestione delle directory
  - Protezione
- Stratificazione utile per ridurre la complessità e la ridondanza,
  - Il codice è più riutilizzabile
    - ▶ Diversi file system possono essere implementati nello stesso SO riusando gli strati di basso livello
  - Ma aggiunge overhead e può ridurre la performance
- La decisione di quanti strati implementare è importante nel progetto del SO

Esistono diversi file system, gli SO ne supportano più di uno

- Ognuno con il suo formato
- CD-ROM è ISO 9660;
- Unix ha **UFS**, FFS;
- Windows ha FAT, FAT32, NTFS come floppy, CD, DVD Blu-ray,
- Linux ha più di 40 tipi, con **extended file system** ext2, ext3, ext4 principali; più distributed file systems, etc.
- Nuovi ancora in arrivo – ZFS, GoogleFS, Oracle ASM, FUSE

# Implementazione del File-System

- Un file system richiede la definizione di diverse strutture dati
- Alcune risiedono in memoria di massa, altre in memoria centrale
- Strutture dati in memoria di massa:
  - **Boot Control Block**: contiene informazioni necessarie per avviare il SO contenuto nel disco da quel volume (se il volume non contiene SO è vuoto)
  - **Volume Control Block**: contiene dettagli riguardanti il volume
    - Numero di blocchi fisici e dimensione
    - Contatore dei blocchi liberi e puntatori ai blocchi
    - Contatore dei FCB presenti nel volume e puntatori ad essi
  - **Struttura della directory**: organizza i file
  - **File Control Block**: memorizza gli attributi dei file
    - inode number, permessi, dimensione, date

file permissions
file dates (create, access, write)

- Info per gestione e per aumentare la performance
  - Dati acquisiti quando il SO è montato, aggiornati e cancellati quando è smontato
- Strutture dati in memoria centrale:
  - **Tabella di montaggio**: contiene informazioni relative ai volumi attualmente montati
  - **Struttura delle directory** (in parte): contiene le informazioni relative alle directory attualmente in uso dai processi (Cache di directory ad accesso recente)
  - **Tabella dei file aperti** (livello SO)
  - **Tabelle dei file aperti** (livello processo)
  - **Blocchi del file system**: durante la loro lettura e scrittura (nei buffer)

# Creazione file

---

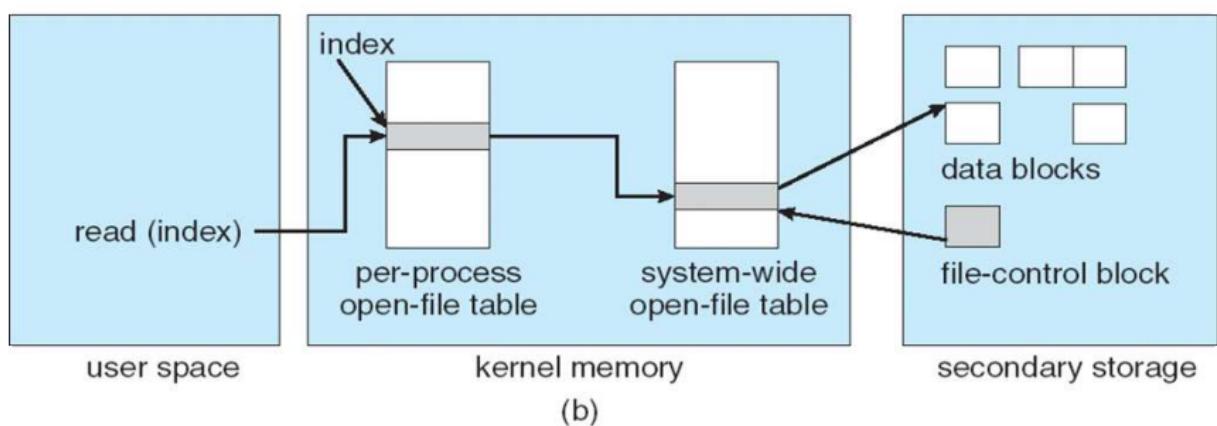
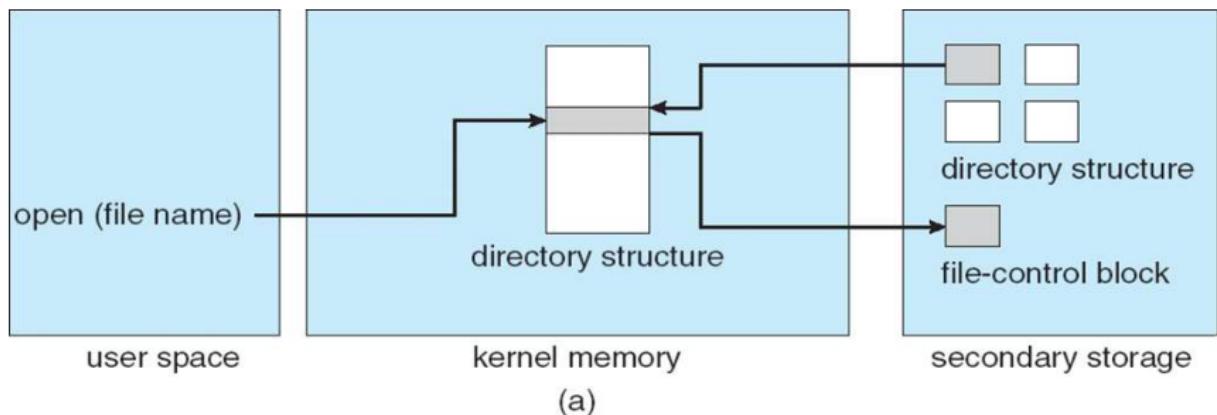
- Per creare un nuovo file un processo
  - Il SO crea e alloca un nuovo FCB
  - Carica in memoria centrale la directory che lo dovrà contenere
  - Aggiorna la directory
  - Riscrive la directory in memoria di massa
- Chiamato il Logical File System (che conosce il formato delle directory)
  - Il Logical File System alloca un nuovo FCB
  - Il Sistema legge la directory nella memoria, aggiorna con nuovo file, riscrive

## Apertura e Chiusura File

---

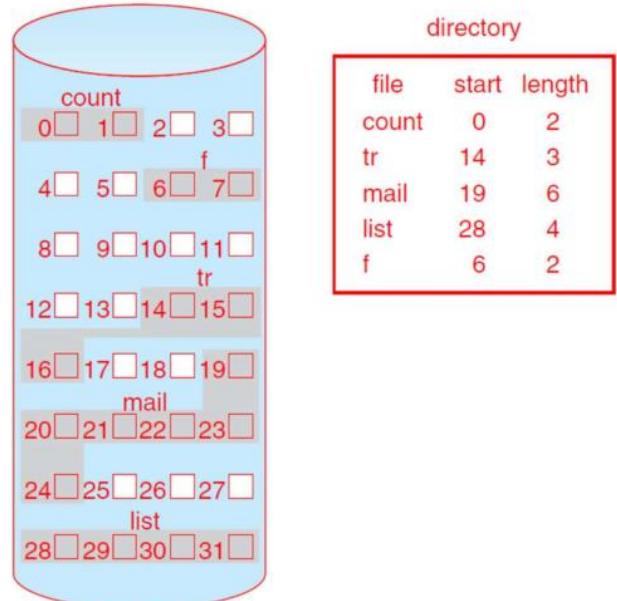
- Open() e Close()
- Il SO controlla la tabella dei file aperti per vedere se il file è già aperto
  - In caso negativo
    - Aggiunge il FCB alla tabella del SO
  - In ogni caso
    - Aggiunge alla tabella del processo chiamante un nuovo elemento che punta alla tabella del SO
    - Incrementa il contatore delle aperture del file
- Alla chiusura del file il SO decrementa il contatore e rimuove l'elemento di quel file dalla tabella del processo chiamante
  - Se il contatore di aperture vale 0 il FCB viene rimosso dalla tabella del SO

## □ apertura di un file e lettura del file



# Metodi di Allocazione - Contigua

- Un metodo di allocazione si riferisci al modo in cui i blocchi sono allocati per i file
  - Accesso veloce e spazio utilizzato in modo efficiente
- Tre metodi:
  - Contiguo
  - Linkato
  - Indicizzato
- **Allocazione Contigua** – ogni file occupa un insieme di blocchi contigui
  - Miglior performance in molti casi
  - Semplice – richiesto solo locazione di partenza (numero del blocco) e lunghezza (numero di blocchi) – dati salvati in FCB
- Pro:
  - spostamenti della testina richiesti per accedere a tutti i blocchi di un file è trascurabile
  - Allo stesso mondo è trascurabile il tempo di ricerca
- Problemi:
  - trovare spazio per il file,
  - sapere la dimensione del file,
  - frammentazione esterna,
  - Necessità di **compattare**
  - **off-line (downtime)** o **on-line**

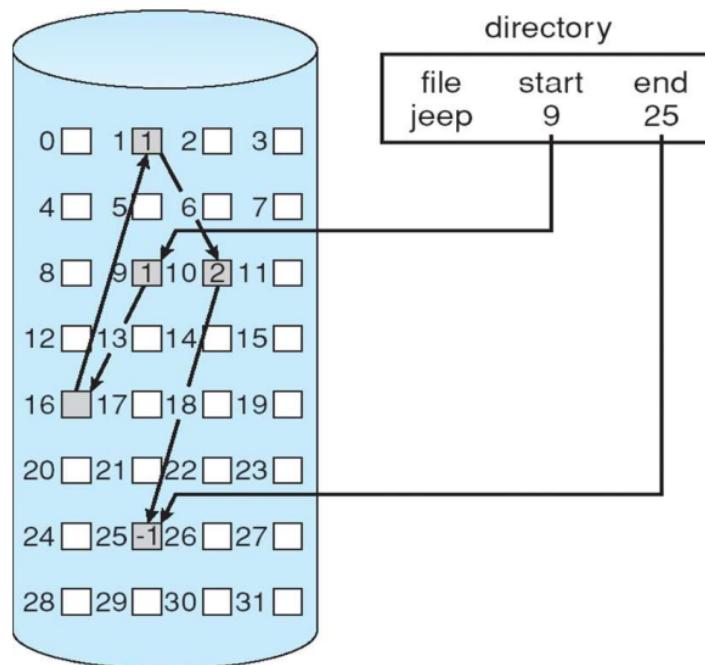


- Pro: l'accesso ai file può essere eseguito semplicemente sia in modo sequenziale che diretto (meno veloce)
- Contro 1: è necessario trovare lo spazio libero per allocare il file
  - Diversi algoritmi
    - ▶ First-fit: primo buco abbastanza grande
    - ▶ Best-fit: il buco che approssima meglio la dimensione (per eccesso)
  - Si presenta il problema della frammentazione esterna
    - ▶ Può essere risolta tramite la compattazione
    - ▶ Si spostano i dati su un altro disco
    - ▶ Si crea una così una zona contigua di spazi liberi
    - ▶ Si riportano i file su disco originale allogandoli in modo contiguo
    - ▶ La compattazione richiede molto tempo e può essere fatta
      - Off line: non è possibile usare il volume
      - On line: peggiora le prestazioni
- Contro 2: è necessario stimare quanto spazio allocare ad un file
  - Quando un file finisce lo spazio disponibile
    - ▶ Il processo può essere interrotto
    - ▶ Il file può essere spostato in una zona di memoria più grande
  - Allocare più spazio di quello inizialmente necessario può portare alla frammentazione interna
    - ▶ Specialmente nel caso di file che crescono lentamente

- **Allocazione Concatenata** – ogni file è una lista linkata di blocchi
  - Ogni blocco contiene un pointer al prossimo blocco
    - ▶ File finisce al nil pointer
  - La directory memorizza per ogni file
    - ▶ Puntatore al primo e ultimo blocco

Il puntatore non è visibile all'utente, quindi "prende" spazio:

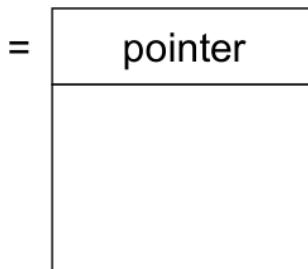
- Blocco da 512 Byte
- puntatore da 4 Byte
- Spazio utile: 508 Byte



Operating System Concepts – 10<sup>th</sup> Edition

Ogni file è una lista concatenata di blocchi del disco: i blocchi possono essere sparsi ovunque su disco

block



Il puntatore non è visibile all'utente, quindi "prende" spazio:

- Blocco da 512 Byte
- puntatore da 4 Byte
- Spazio utile: 508 Byte

- Creazione di un file:
  - si aggiunge un elemento alla directory con Puntatore al primo blocco settato a null dimensione pari a 0
- Scrittura di un file:
  - Si cerca un blocco libero e si effettua la scrittura dei dati
  - Lo si concatena all'ultimo blocco del file (se presente)
  - Si aggiornano i puntatori della directory all'ultimo e al primo blocco (nel caso il file sia ancora vuoto)
- Lettura di un file:
  - si scorrono in sequenza i blocchi seguendo la concatenazione
- L'allocazione concatenata risolve i problemi dell'allocazione contigua:
  - Non c'è frammentazione esterna, non serve la compattazione
  - Non è necessario conoscere la dimensione del file al momento della creazione
- Problemi:
  - È efficiente solo per l'accesso sequenziale (a causa dei puntatori)
  - Richiede più spostamenti della testina
  - I puntatori consumano spazio
    - Una soluzione è quella di allocare cluster di blocchi anziché singoli blocchi
    - Meno spazio sprecato per i puntatori, meno spostamenti della testina
    - Aumenta però la frammentazione interna
  - Perdita o danneggiamento di un puntatore
    - Potrebbe puntare ad un blocco vuoto o un blocco di un altro file
    - Possibile soluzione: liste doppiamente concatenate

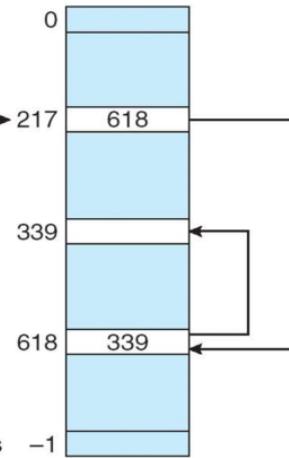
# File-Allocation Table

- Variante importante è FAT (File Allocation Table)
  - Nella prima parte del disco si istanzia un tabella
    - Contenente tanti elementi quanti sono i blocchi
    - Ordinata in base al numero del blocco
  - La directory contiene per ogni file il puntatore al suo primo elemento nella FAT
  - Seguendo i puntatori nella FAT si ricavano gli indici dei blocchi fisici di un file

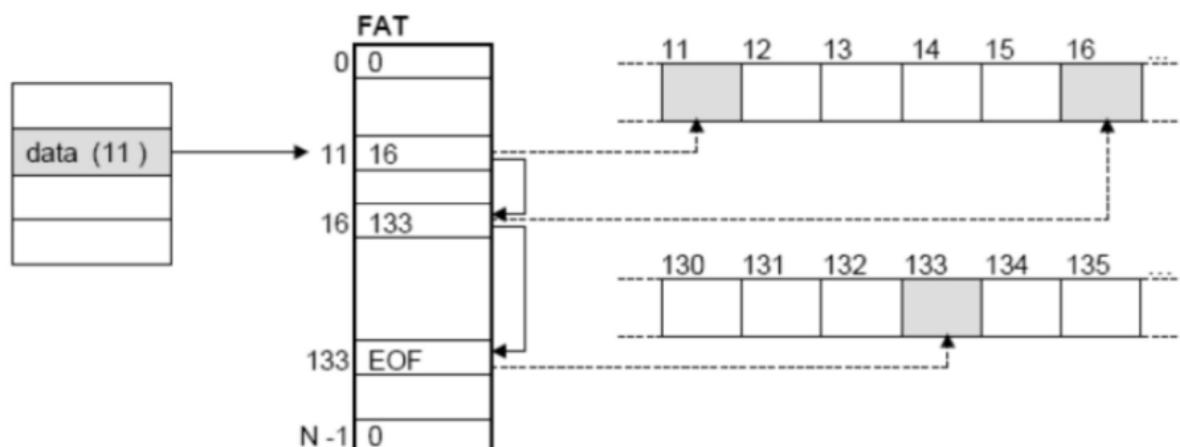
directory entry

test	...	217
name		start block

start block



Operating System Concepts – 10<sup>th</sup> Edit....

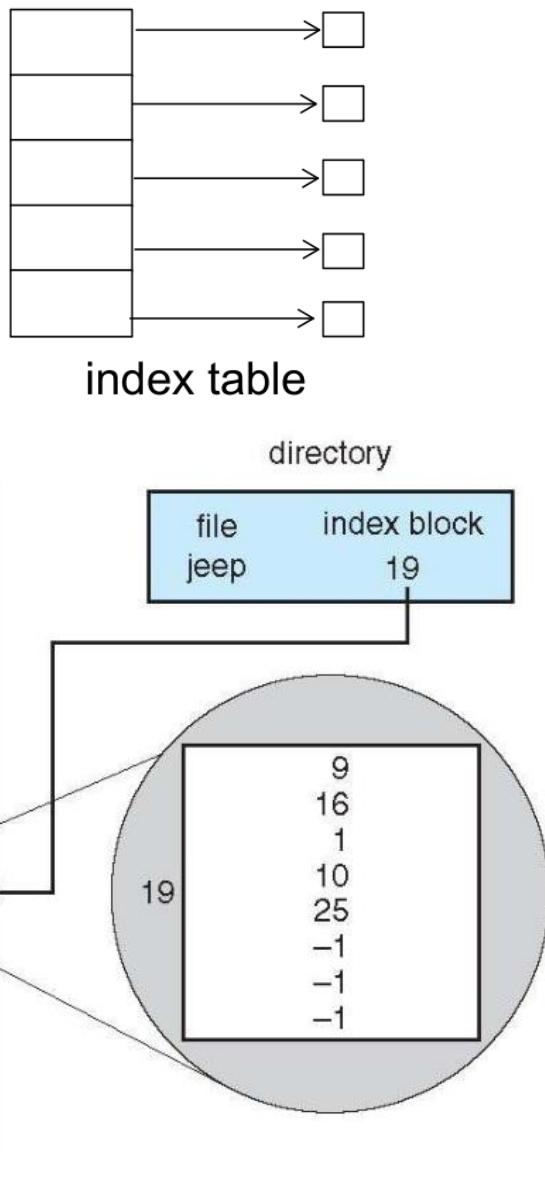


- La differenza fra FAT12, FAT16 e FAT32 consiste in quanti bit sono allocati per numerare i blocchi del disco
  - Con 12 bit, il file system può indirizzare al massimo  $2^{12} = 4096$  blocchi, mentre con 32 bit si possono gestire  $2^{32} = 4.294.967.296$  blocchi
  - L'aumento del numero di bit di indirizzo dei blocchi e la dimensione stessa del blocco sono stati resi necessari per gestire unità a disco sempre più grandi e capienti

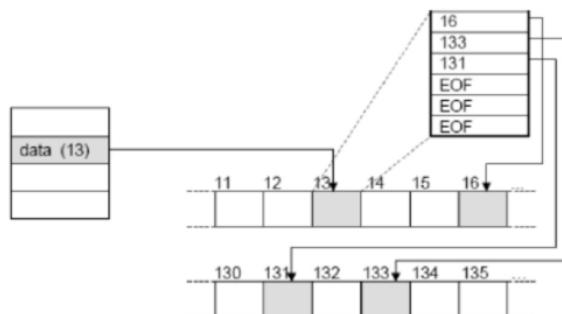
# Metodi di Allocazione - Indicizzata

## Allocazione indicizzata

- Ogni file possiede un indice dei blocchi memorizzato in un blocco indice
  - Questo blocco contiene un array di puntatori agli altri blocchi del file
  - L'i-simo elemento dell'array punta all'i-simo blocco del file
- La directory memorizza per ogni file un puntatore al suo blocco indice
- Questo risolve il problema dell'accesso diretto presente nell'allocazione concatenata ed evita la frammentazione esterna



- Creazione di un file:
  - Si crea un blocco indice con tutti gli elementi settati a null
  - Si aggiunge un elemento alla directory con puntatore ad un blocco indice
  
- Scrittura di un file:
  - Si cerca un blocco libero e si effettua la scrittura dei dati
  - Si aggiorna il blocco indice
  
- Lettura di un file:
  - si scorrono in sequenza i blocchi seguendo l'array dell'indice
  
- Problema:
  - l'indice richiede l'uso di un intero blocco, quindi nel caso di un file piccolo si ha frammentazione interna



- Si dovrebbero usare blocchi piccoli in modo da ridurre la frammentazione
- Problemi con i file grandi

- La scelta del metodo di allocazione deve considerare due fattori:
  - Efficienza di memorizzazione
  - Tempo di accesso ai dati
- La scelta è influenzata dalla modalità di accesso al file
  - L'allocazione contigua richiede un solo accesso al disco qualsiasi sia il tipo di accesso (sequenziale o diretto)
    - Anche con l'accesso diretto, noto l'indirizzo del primo blocco fisico e l'indirizzo del blocco logico desiderato, è possibile calcolare l'indirizzo fisico di accesso
  - L'allocazione concatenata invece è efficiente per l'accesso sequenziale, ma non per quello diretto
    - Accesso all'i-esimo blocco porta all'accesso di i blocchi intermedi
- Per questo motivo alcuni sistemi gestiscono
  - File ad accesso sequenziale con allocazione concatenata
  - File ad accesso random con l'allocazione contigua
    - È necessario specificare a priori la dimensione del file
  - È sempre possibile convertire il formato del file
  - Le prestazioni dell'allocazione indicizzata invece dipendono
    - Dalla profondità dell'indice (quanti blocchi indice)
    - Dalla dimensione del file (se grande occorre scorrere molti blocchi indice)
- Alcuni sistemi usano
  - Allocazione contigua per file piccoli
  - Allocazione indicizzata per file grandi

# Gestione dello Spazio Libero

---

- Per trovare in modo veloce un insieme di blocchi da allocare ad un file il Sistema Operativo tiene traccia dei blocchi non allocati
  - Lista dello spazio libero
- Creazione di un file:
  - Si cerca nella lista il numero di blocchi necessari
  - Si allocano al file
  - Si rimuovono dalla lista
- Eliminazione di un file
  - I blocchi associati al file vengono deallocati
  - Si aggiungono alla lista

Diverse possibili implementazioni:

Vettore di bit

Lista concatenata

Raggruppamento

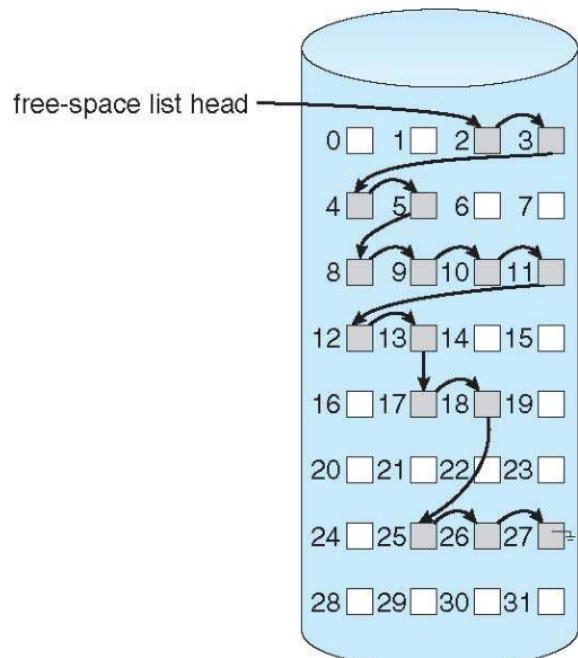
Conteggio

# Vettore di Bit o BitMap

- Si memorizza un array di lunghezza pari al numero dei blocchi
- Il valore delle celle indica la disponibilità di un blocco
  - 1 libera
  - 0 occupata
- Per creare un file
  - Ricerca del primo bit a 1 (allocazione concatenata o indicizzata)
  - Ricerca di un blocco di bit a 1 sufficientemente grande (allocazione contigua)
- Pro: semplice ed efficiente
- Contro:
  - Per essere efficiente deve però essere mantenuta in memoria centrale
  - Dischi grandi richiedono vettori di bit molto grandi

## Liste Concatenate

- Si memorizza in memoria centrale un puntatore al primo blocco libero
- Ogni blocco libero viene poi concatenato al successivo
- Contro: è poco efficiente in quanto scorrere l'intera lista richiede molti accessi alla memoria
- Fortunatamente lo scorrimento della lista è un evento raro
- Il SO si limita a cercare il primo blocco libero per allocarlo ad un file
- Allocato il blocco si aggiorna il puntatore al primo blocco libero in memoria centrale



- Raggruppamento
  - Modifica dell'approccio a free-list
  - Il primo blocco contiene gli indirizzi di n blocchi liberi
    - Al primo accesso subito info sui blocchi liberi
  - n-1 sono effettivamente liberi
  - L'ultimo contiene a sua volta n indirizzi di blocchi liberi
  
- Conteggio
  - Sfrutta il fatto che tipicamente si ha più di un blocco contiguo libero
  - Si usa un array in cui ogni elemento contiene
    - Indirizzo del primo blocco libero
    - Conteggio degli n blocchi contigui liberi

