

Riassunto Sistemi Operativi

Simone Acuti

2 giugno 2024

Indice

1	Introduzione	1
2	Evoluzione dei sistemi operativi	2
2.1	Sistemi batch semplici	2
2.2	Sistemi batch multi-programmati	2
2.3	Sistemi time-sharing	3
2.4	Protezione	4
2.5	System call	4
2.6	Processi	4
2.7	Interfaccia utente e programmatore	4
2.8	Struttura e organizzazione dei SO	5
2.8.1	Struttura monolitica	5
2.8.2	Struttura modulare	5
2.8.3	Micro-kernel	6
2.9	Organizzazione di UNIX	6
3	Processi e Thread	7
3.1	Concetto di processo	7
3.2	Stati di un processo	7
3.3	Rappresentazione dei processi	8
3.4	Scheduling dei processi	8
3.4.1	Scheduler a lungo termine	8
3.4.2	Scheduler a medio termine (swapper)	8
3.4.3	Scheduler a breve termine (o di CPU)	8
3.5	Cambio di contesto	9
3.6	Code di Scheduling	9
3.7	Operazioni sui processi	10
3.8	Creazione di processi (fork)	11
3.8.1	Relazioni padre-figlio	11
3.9	Terminazione	11
3.10	Processi leggeri (thread)	11
3.11	Processi interagenti	12
3.11.1	Processi interagenti	12
4	Processi UNIX	13
4.1	Stati di un processo in UNIX	13
4.1.1	Processi swapped	14
4.2	Rappresentazione dei processi in UNIX	14
4.3	Immagine di un processo UNIX	15
4.4	System call per la gestione di processi	15
4.5	Fork	15
4.6	Terminazione di processi	16
4.7	Wait	16
4.8	Exec	16
4.9	Inizializzazione processi	17

5 Interazione fra processi	18
5.1 Processi interagenti	18
5.2 Processi interagenti mediante scambio di messaggi	18
5.3 Naming	19
5.4 Buffering del canale	19
5.5 Sincronizzazione tra processi	19
5.5.1 Sincronizzazione nel modello ad ambiente locale	20
5.5.2 Sincronizzazione nel modello ad ambiente globale	20
5.6 Problema della mutua esclusione	20
5.7 Possibile soluzione: semafori di Dijkstra	20
5.7.1 Atomicità della wait e della signal	21
6 Segnali	22
6.1 Segnali UNIX	22
6.2 System call signal	23
6.3 Segnale SIGCHLD	23
6.4 Segnali e fork	23
6.5 Segnali ed exec	23
6.6 System call di segnali	23
6.7 Modello affidabile dei segnali	24
6.8 Sigaction	25
6.9 Signal vs Sigaction	25
7 Comunicazione tra processi	26
7.1 Pipe	26
7.2 Comunicazione attraverso pipe	26
7.3 Unidirezionalità / bidirezionalità	26
7.4 System call pipe	27
7.5 Processi che possono comunicare mediante pipe	27
7.6 Chiusura della pipe	27
7.7 System call dup	27
8 Java thread	28
8.1 Modello ad ambiente locale	28
8.2 Multithreading in Java	29
8.3 Problema dei metodi stop e suspend	29
8.4 Sincronizzazione di thread	30
8.4.1 Synchronized	30
9 Astrazione di File system	31
9.1 File	31
9.2 Operazioni sui file	31
9.3 Blocchi & record logici	32
9.4 Metodi di accesso	32
9.4.1 Accesso sequenziale	32
9.4.2 Accesso diretto	32
9.5 Accesso a indice	32
9.6 Direttorio	33
9.6.1 A un livello	33
9.6.2 A due livelli	33
9.6.3 Struttura ad albero	34
9.6.4 Struttura a grafo aciclico	34
9.7 Realizzazione del file system	34
9.8 Allocazione contigua	35
9.9 Allocazione dinamica	36
9.10 Allocazione a indice	36

10 File system UNIX	37
10.1 Nome, i-number, i-node	37
10.2 Organizzazione file system UNIX	37
10.2.1 Boot Block	37
10.2.2 Super Block	37
10.2.3 Data Block	38
10.2.4 i-List	38
10.3 i-Node	38
10.4 Directory	38
10.5 Gestione del file system in UNIX	38
10.6 Gestione file system: concetti generali	38
10.7 File descriptor	39
10.8 Strutture dati del kernel	39
10.9 Gestione file system UNIX: system call	39
10.10 Apertura di un file	40
11 Scheduling della CPU	41
11.1 CPU burst & I/O burst	41
11.2 Criteri di scheduling	42
11.3 Algoritmo di scheduling FCFS	42
11.3.1 Problema dell'algoritmo FCFS	42
11.4 Algoritmo di scheduling SJF	42
11.4.1 Problema dell'algoritmo SJF	43
11.5 Algoritmo di scheduling Round Robin	43
11.5.1 Problema dell'algoritmo Round Robin	43
11.6 Scheduling con priorità	43
11.6.1 Problema degli algoritmi di scheduling con priorità	44
11.7 Approcci misti	44
11.8 Scheduling in UNIX	44
11.9 Linux scheduling (da v2.5)	45
11.10 Scheduling dei thread Java	45
12 Gestione della memoria	46
12.1 Gestione della memoria centrale	46
12.2 Fasi di sviluppo di un programma	46
12.3 Accesso alla memoria	47
12.4 Binding degli indirizzi	47
12.5 Caricamento/collegamento dinamico	47
12.6 Overlay	47
12.7 Tecniche di allocazione memoria centrale	47
12.8 Allocazione contigua a partizione singola	47
12.9 Allocazione contigua a partizioni multiple	48
12.9.1 Partizioni fisse	48
12.9.2 Partizioni variabili	49
12.10 Partizioni & protezione	49
12.11 Compattazione	50
12.12 Paginazione	50
12.13 Supporto hardware per la paginazione	50
12.13.1 Tabella delle pagine	51
12.14 Paginazione & Protezione	51
12.15 Paginazione a più livelli	51
12.16 Tabella delle pagine invertita	52
12.17 Segmentazione	52
12.18 Segmentazione paginata	53
12.19 Memoria virtuale	53
12.20 Paginazione su richiesta	54
12.20.1 Trattamento page fault	54
12.20.2 Sovrallocazione	54
12.21 Località dei programmi	55
12.22 Working set	55

13 Protezione	56
13.1 Protezione e Sicurezza	56
13.2 Sicurezza	56
13.3 Protezione (e least privilege)	57
13.4 Protezione e controllo degli accessi	57
13.5 Dominio di protezione	57
13.5.1 Matrice degli accessi (modello di protezione)	57
13.6 Access Conrol List (ACL)	57
13.7 Capability List	58

Sommario

Riassunto del corso di sistemi operativi svolto presso l'università di Ferrara.

Capitolo 1

Introduzione

Un **sistema operativo** è un insieme di programmi che agisce come intermediario tra l'utente e l'hardware del computer.

Grazie ad esso viene offerto a livello applicativo una visione astratta dell'hardware. Così facendo si semplifica la realizzazione dei programmi applicativi.

Il sistema operativo gestisce in modo efficiente le risorse dell'hardware. Impedisce che le applicazioni accedano in modo concorrente alla stessa risorsa.

Possiamo vedere il sistema operativo come un'interfaccia tra ciò che offre l'hardware e il livello applicativo.

Capitolo 2

Evoluzione dei sistemi operativi

- **Prima generazione:** programmati in linguaggio macchina, dati e programmi salvati su schede perforate.
- **Seconda generazione:** emergono i primi *sistemi batch semplici*. Dotati di linguaggi di alto livello, come il fortnar. L'input è dato sempre mediante schede perforate. I programmi vengono caricati in lotti (batch) con esigenze simili.

2.1 Sistemi batch semplici

Batch: insieme di programmi (job) da eseguire in modo sequenziale.

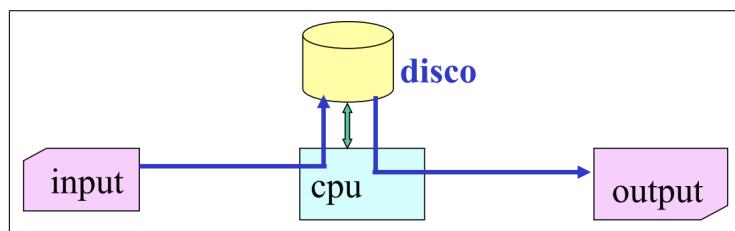
In questo caso il sistema operativo (SO) viene detto **monitor**; offre funzionalità di trasferimento e di controllo da un job appena terminato, al prossimo da eseguire. Ovvero il SO monitora il job e quando termina il SO manda in esecuzione il prossimo job (senza l'intervento umano).

I sistemi batch semplici non sono efficienti nelle operazioni di I/O perché durante queste operazioni la CPU rimane inattiva.

Questo perché nei sistemi batch semplici viene caricato un solo job alla volta in memoria centrale.

Per migliorare l'utilizzo della CPU è stato introdotto il meccanismo di **spooling** (Simultaneous Peripheral Operation On Line).

Tale meccanismo permette di caricare su disco (più veloce rispetto alla scheda perforata) i dati e i programmi dei job successivi, mentre la CPU è utilizzata da altri job.



Problemi sistemi batch semplici:

- Finché il job corrente è in esecuzione non si può iniziare ad eseguire il job successivo.
- Se un job si sospende in attesa di un evento, la CPU rimane inattiva.
- Non c'è iterazione con l'utente.

2.2 Sistemi batch multi-programmati

Per evitare il problema dell'inattività della CPU vengono creati i sistemi batch con multi-programmazione.

Un pool di job viene caricato contemporaneamente su disco:

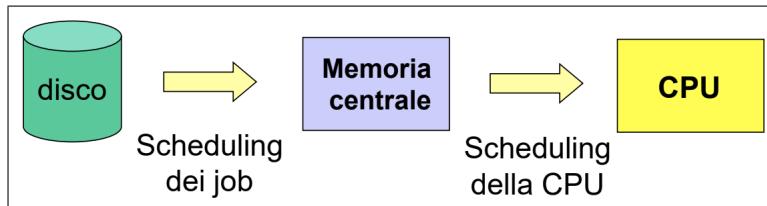
- SO seleziona un sottoinsieme dei job appartenenti al pool da caricare in memoria centrale.

- Mentre un job è in attesa di un evento, il sistema operativo assegna CPU a un altro job.

Quando il job che sta utilizzando la CPU si sospende in attesa di un evento il SO decide a quale job assegnare la CPU ed effettua lo scambio (**scheduling**).

SO effettua delle scelte tra tutti i job:

- Quali job caricare in memoria centrale: scheduling dei job.
- A quale job assegnare la CPU: scheduling della CPU.



Nei sistemi batch multi-programmati in memoria centrale abbiamo: SO e i job selezionati dall'operazione di scheduling.

Dato che in memoria centrale ci sono diversi job c'è necessità di **protezione**. Ovvero il SO deve assicurare che ogni job operi solo nello spazio di memoria ad esso associato.

2.3 Sistemi time-sharing

Nascono dalle necessità di:

- **interattività** con l'utente.
- **multi-utenza** più utenti interagiscono contemporaneamente con il SO.

Multi-utenza: il SO presenta ad ogni utente una macchina virtuale dedicata in termini di: utilizzo della CPU, utilizzo delle risorse (file system).

interattività: è necessario fornire agli utenti la possibilità di interagire in modo più veloce con i job del SO. Per fare questo il SO non aspetta che un job si sospenda o che il job termini, ma interrompe l'esecuzione di ciascun job in modo periodico ad intervalli di tempi prefissati. Questi intervalli sono detti quanti di tempo o time-slice.

Dunque i sistemi time-sharing sono sistemi che permettono a più job di essere eseguiti dalla CPU. Il SO ha il compito di alternare ciclicamente l'assegnamento della CPU ai diversi job che sono in esecuzione.

Questi intervalli di tempo sono sufficientemente piccoli da fornire all'utente finale l'illusione che il proprio job sia in esecuzione continua.

Ogni volta che il SO assegna la CPU a un nuovo job si ha un **context switch**. Questa operazione dei costi aggiuntivi e quindi un **overhead**.

Requisiti per i sistemi time-sharing:

- Il SO deve fornire meccanismi di gestione e protezione della memoria. In quanto essendoci più job in memoria centrale è necessario che ciascun job debba interagire con lo spazio di memoria che gli è stato assegnato.
- Migliorare lo scheduling della CPU.
- **Sincronizzazione/comunicazione** tra job: diversi job possono avere la necessità di scambiarsi informazioni e di accedere a risorse condivise. Inoltre bisogna verificare che l'accesso a risorse condivise non sia bloccante e che non generi **deadlock**.
- Per supportare l'interattività è necessario che il SO fornisca strumenti di accesso online al file-system.

2.4 Protezione

Per garantire protezione, molte architetture di CPU prevedono un duplice modo di funzionamento (**dual mode**).

1. **User mode.**

2. **Kernel mode.**

Per poter realizzare questa modalità di protezione le architetture HW mettono a disposizione un bit di modo. Il bit vale 0 in kernel mode e 1 in user mode.

Le istruzioni privilegiate (quelle più pericolose), devono essere accedute in **kernel mode**.

Il SO esegue in kernel mode.

I programmi utente eseguono in **user mode**. Quando un programma ha bisogno di eseguire istruzioni privilegiate viene fatta una **system call**.

2.5 System call

Una system call è il meccanismo usato da un programma a livello utente per richiedere un servizio a livello kernel del SO.

- Invio di un'interruzione SW al SO.
- Salvataggio dello stato (PC, registri, bit di modo, ...) del programma chiamante e trasferimento del controllo al SO.
- Il SO esegue in modo kernel l'operazione richiesta.
- Al termine dell'operazione, il controllo ritorna al programma chiamante (ritorno al modo user).

2.6 Processi

Un programma è un'entità **passiva**, cioè un insieme di byte che contengono le istruzioni che dovranno essere eseguite.

Un processo invece è un'entità **attiva**. Ovvero l'unità di lavoro/esecuzione che può essere attivata all'interno del sistema. Ogni attività del SO è rappresentata da un processo. Quindi un processo è un'istanza di un programma in esecuzione.

2.7 Interfaccia utente e programmatore

Il SO presenta un'interfaccia che consente la l'interazione con l'utente.

- **Interprete dei comandi (shell)**: l'interazione avviene mediante una linea di comando.
- **interfaccia grafica (GUI)**: l'interazione avviene mediante interazione mouse/touch con elementi grafici su desktop.

Il SO offre un'interfaccia per i programmatori, ovvero vengono offerte delle system call con cui il programmatore può richiedere di interagire con le risorse del SO.

Un programma è detto **programma di sistema** quando esso effettua delle system call.

2.8 Struttura e organizzazione dei SO

Visto che un SO è l'insieme delle componenti:

- Gestione dei processi.
- Gestione della memoria centrale.
- Gestione dei file.
- Gestione dell'I/O.
- Gestione della memoria secondaria.
- Protezione e sicurezza.
- Interfaccia utente/programmatore.

È possibile realizzare un SO in modi differenti:

- Struttura monolitica.
- Struttura modulare.
- Micro-kernel.

2.8.1 Struttura monolitica

Si organizza il SO in un unico modulo che contiene tutte le procedure per realizzare le varie componenti.

L'interazione tra le varie componenti avviene tramite una chiamata a procedura.

Vantaggi:

- Basso costo di interazione tra le componenti → efficienza.

Svantaggi:

- Difficoltà nel soddisfare requisiti di: estensibilità, manutenibilità, riutilizzo, ecc...

2.8.2 Struttura modulare

Le varie componenti del SO vengono organizzate in moduli caratterizzati da interfacce ben definite. Esso è costituito da livelli sovrapposti, ognuno dei quali realizza un insieme di funzionalità:

- Ogni livello realizza un insieme di funzionalità che vengono offerte al livello superiore mediante un'interfaccia.
- Ogni livello utilizza le funzionalità offerte dal livello sottostante, per realizzare altre funzionalità.

Esempio dell'organizzazione dei livelli del SO modulare THE.

livello 5: programmi di utente
livello 4: buffering dei dispositivi di I/O
livello 3: driver della console
livello 2: gestione della memoria
livello 1: scheduling CPU
livello 0: hardware

Vantaggi:

- Astrazione: ogni livello è un oggetto astratto, che fornisce ai livelli superiori una visione astratta del sistema.
- Modularità: relazioni tra livelli sono chiaramente esplicitate dalle interfacce.

Svantaggi:

- Organizzazione gerarchica tra i componenti: non sempre è possibile → difficoltà di realizzazione.
- Scarsa efficienza (costo di attraversamento dei livelli)

2.8.3 Micro-kernel

Il **kernel** è la parte di SO che esegue le istruzioni in modo privilegiato (modo kernel).

Esso si interfaccia direttamente con l'HW della macchina. Le funzioni realizzate all'interno del nucleo variano a seconda del particolare SO

Per un sistema multi-programmato a divisione di tempo, il nucleo deve, almeno:

- Gestire il salvataggio/ripristino dei contesti (context-switching).
- Realizzare lo scheduling della CPU.
- Gestire le interruzioni.
- Realizzare il meccanismo di system call.

Nei SO a micro-kernel la struttura del kernel è ridotta a poche funzionalità di base:

- Gestione della CPU.
- Gestione della memoria.
- Gestione meccanismi di comunicazione I/O.

Il resto del SO è mappato su processi di utente.

2.9 Organizzazione di UNIX

Programmi utente e di sistema
Utenti shell per i comandi compilatori interpreti librerie di sistema

Kernel	
System call per interagire col kernel	Kernel per interagire con l'HW
Gestione dei segnali. Gestione del file system. Gestione dei driver per accedere ai dischi. Controllo della memoria Gestione dello scheduling della CPU e della memoria virtuale.	Terminali di controllo Controllo dei dispositivi

Capitolo 3

Processi e Thread

3.1 Concetto di processo

Il processo è un programma in esecuzione.

Soltanamente, esecuzione sequenziale (istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma).

Un SO multi-programmato consente l'esecuzione **concorrente** di più processi.

Ricordare che: il programma è un'entità passiva; il processo è un'entità attiva. Quindi un programma diventa processo quando viene attivato e messo in esecuzione.

Il processo è rappresentato da:

- **Codice** (text) del programma eseguito.
- **Dati**: variabili globali.
- Alcuni **registri** della CPU.
- **Stack**: parametri, variabili locali a funzioni/procedure.

3.2 Stati di un processo

Un processo può trovarsi in diversi stati:

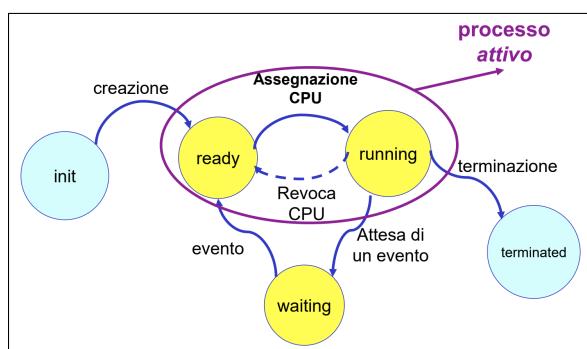
- **init**: stato transitorio durante il quale il processo viene caricato in memoria e il SO inizializza i dati che lo rappresentano.
- **Ready**: il processo è pronto per utilizzare la CPU.
- **Running**: il processo sta utilizzando la CPU.
- **Waiting**: il processo è sospeso in attesa di un evento.
- **Terminated**: stato transitorio relativo alla fase di terminazione e deallocazione del processo dalla memoria.

Un processo viene detto attivo quando si trova nello stato di running o nello stato di ready.

In un sistema mono-processore e multi-programmato: un solo processo (al massimo) si trova nello stato running; più processi possono trovarsi negli stati ready e waiting.

Si ha la necessità di usare strutture dati, a cui il SO ha accesso, per mantenere in memoria le informazioni dei processi: in attesa di acquisire la CPU (ready); in attesa di eventi. (waiting).

Queste strutture dati vengono dette **descrittori di processo** che descrivono lo stato di ciascun processo.



3.3 Rappresentazione dei processi

All'interno del SO il descrittore che descrive lo stato dei processi si chiama **process control block (PCB)**.

Il PCB contiene le informazioni relative ad un processo:

- Stato del processo.
- Program counter.
- Contenuto dei registri di CPU.
- ...

3.4 Scheduling dei processi

È l'attività mediante la quale il SO effettua delle scelte tra i processi, riguardo a:

- Caricamento in memoria centrale.
- Assegnazione della CPU.

In generale, il SO compie tre diverse attività di scheduling:

- Scheduling **a breve termine** (o di CPU).
- Scheduling **a medio termine** (o di swapping).
- Scheduling **a lungo termine** (per SO a batch).

3.4.1 Scheduler a lungo termine

Lo scheduler a lungo termine è quella componente del SO che seleziona i programmi da eseguire dalla memoria secondaria per caricarli in memoria centrale (creando i corrispondenti processi).

Lo scheduling a lungo termine serve per gestire il grado di multi-programmazione e permette al SO di variare il numero di processi attivi all'interno di esso.

Lo scheduling a lungo termine è un componente importante dei sistemi batch multi-programmati. Nei sistemi time-sharing esso non è presente.

3.4.2 Scheduler a medio termine (swapper)

Swapping: trasferimento temporaneo in memoria secondaria di processi (o di parti di processi), in modo da consentire l'esecuzione di altri processi.

Nei sistemi operativi multi-programmati: la memoria fisica totale, può essere minore alla somma di tutti gli spazi logici di indirizzamento che sono stati allocati per ciascun processo; il grado di multi-programmazione non è vincolato dalle esigenze di spazio dei processi.

3.4.3 Scheduler a breve termine (o di CPU)

È quella parte del SO che si occupa della selezione dei processi a cui assegnare la CPU.

Nei sistemi time sharing, allo scadere di ogni quanto di tempo, il SO:

- Decide a quale processo assegnare la CPU (scheduling di CPU).
- Effettua il **cambio di contesto** (content-switch).

Come già detto all'interno di un SO mono-processore, multi-programmato abbiamo un solo processo in esecuzione ma possiamo avere più processi nello stato di ready.

La coda dei processi pronti: contiene i PCB dei processi che si trovano in stato Ready.

Lo stesso avviene per i processi che sono nello stato di waiting.

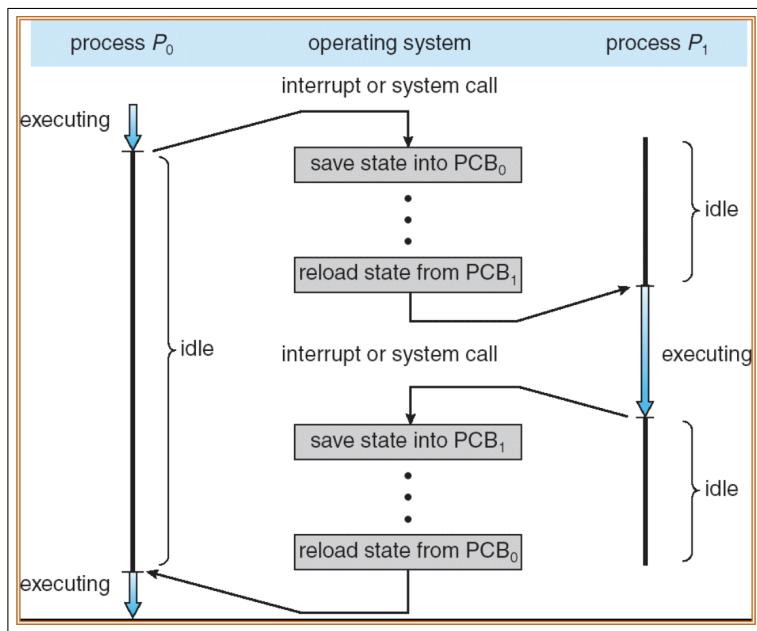
Code di waiting (una per ogni tipo di attesa: dispositivi I/O, timer, ...): ognuna di esse contiene i PCB dei processi waiting in attesa di un evento del tipo associato alla coda.

3.5 Cambio di contesto

Per cambio di contesto si intende quella fase in cui l'uso della CPU viene commutato da un processo ad un altro.

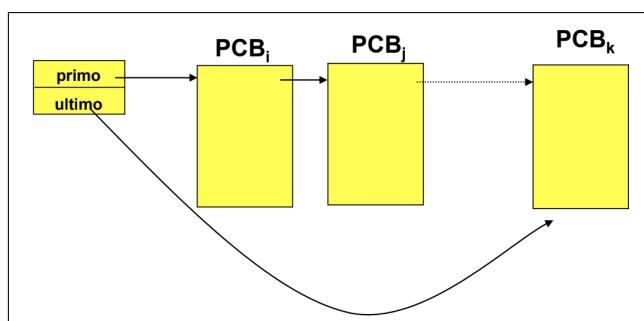
Quando avviene un cambio di contesto tra un processo P_i ad un processo P_{i+1} :

- Salvataggio dello stato di P_i : il SO copia PC, registri, e le parti di memoria del processo de-schedulato P_i nel suo PCB.
- Ripristino dello stato di P_{i+1} : il SO trasferisce i dati del processo P_{i+1} dal suo PCB nei registri di CPU, che può così riprendere l'esecuzione.



3.6 Code di Scheduling

Coda dei processi pronti (ready queue):

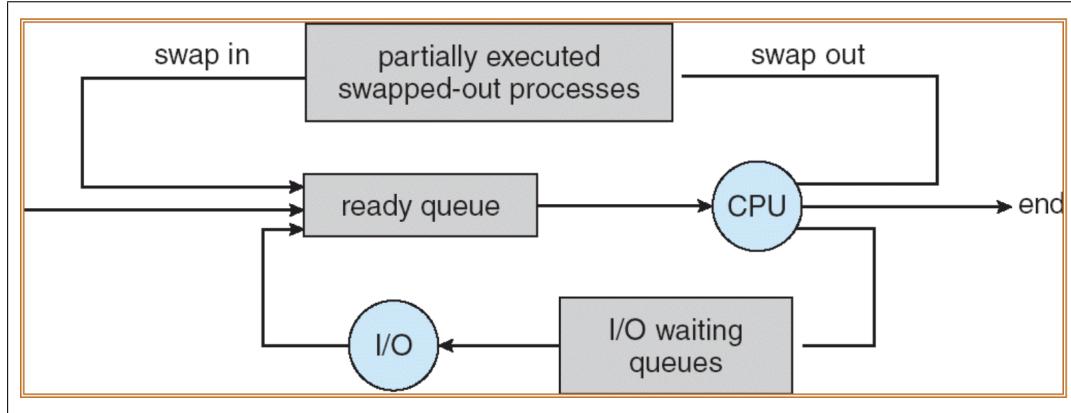


Abbiamo 2 puntatori:

- Un puntatore al primo PCB (PCB_i).
- Un puntatore all'ultimo PCB (PCB_k).

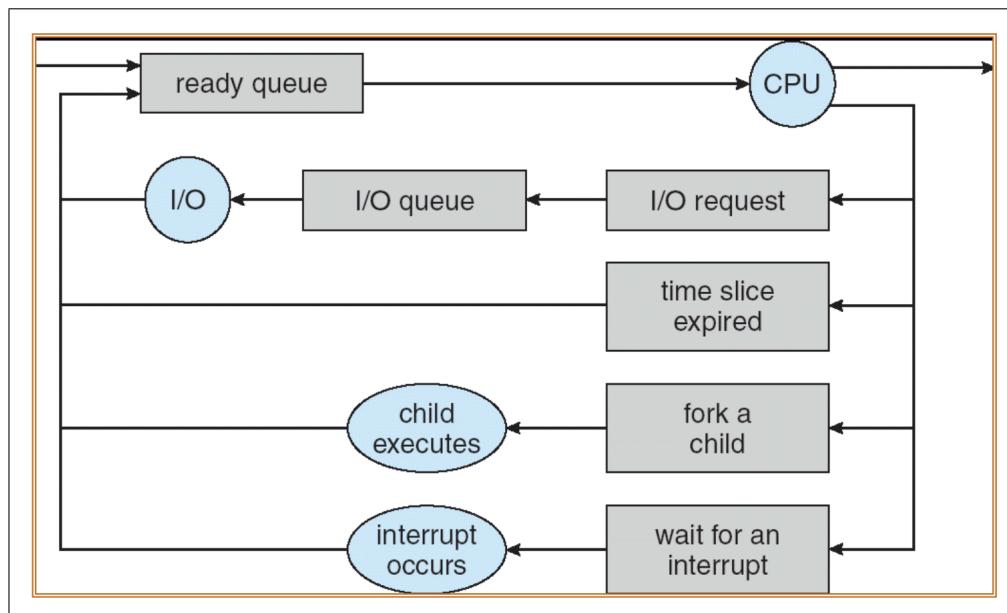
La strategia di gestione della ready queue dipende dalle politiche (algoritmi) di scheduling adottate dal SO.

Short term & medium term scheduling:



Nel SO abbiamo un uso congiunto dello scheduling di medio e breve termine. Lo scheduling di breve termine seleziona un processo da essere messo in esecuzione. Questo però può finire nello stato di attesa di un evento, e quando questo avviene il processo ha l'accesso alla risorsa, per poi tornare nella coda dei processi pronti. Lo scheduling di medio termine seleziona un processo per essere messo in memoria secondaria (swap-out). Esso non verrà messo in esecuzione finché lo scheduling di medio termine non lo seleziona (swap-in).

Short term scheduling:



Lo scheduler di breve termine seleziona un processo dalla coda dei processi pronti da mettere nello stato di running. Esso può: terminare; essere messo in memoria secondaria; collocato nello stato di attesa di qualche evento.

Gli eventi possono essere:

- Richiesta I/O.
- Attesa di un interrupt.
- Generazione di una **fork**.

3.7 Operazioni sui processi

Nei SO multi-programmati esistono diverse operazioni che è necessario eseguire sui processi. In particolare è necessario che il SO supporti 3 meccanismi principali:

- **Creazione.**
- **Terminazione.**

- **Interazione** tra processi.

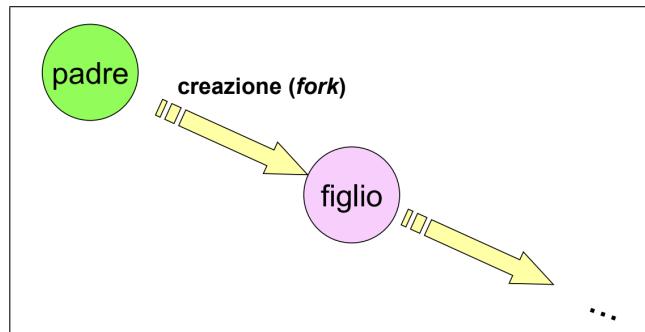
Sono operazioni privilegiate eseguite in modo kernel. Queste operazioni vengono definite grazie alle system call.

3.8 Creazione di processi (fork)

Un processo detto padre può generare un nuovo processo detto figlio. Un figlio può creare uno o più nipoti e un padre può creare uno o più figli.

3.8.1 Relazioni padre-figlio

- **Concorrenza**: padre e figlio procedono in modo parallelo (UNIX), oppure il padre si sospende in attesa della terminazione del figlio.
- **Condivisione delle risorse**: le risorse del padre sono condivise con i figli (UNIX). oppure il figlio utilizza risorse soltanto se esplicitamente richieste da se stesso.
- **Spazio degli indirizzi**: lo spazio può essere **duplicato**: il figlio ha una copia degli indirizzi del padre (fork in UNIX). Oppure **differenziato**: gli spazi degli indirizzi di padre e figlio con codice e dati diversi (VMS).



3.9 Terminazione

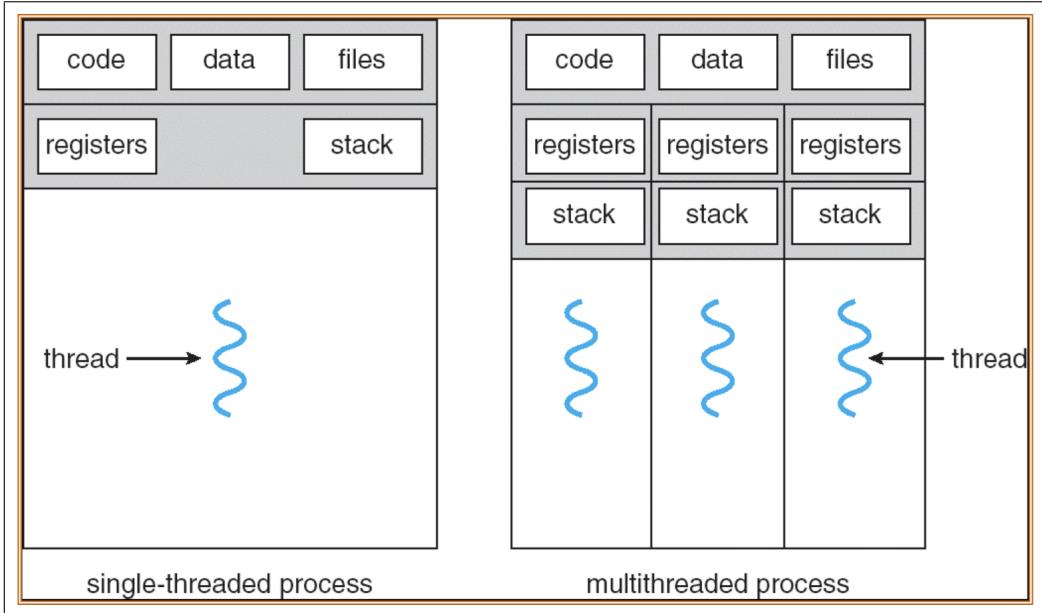
In fase di terminazione dobbiamo considerare che ogni processo è figlio di un altro processo e a sua volta può essere padre di altri processi. Il SO mantiene questa gerarchia in quanto nel PCB di ogni processo vi è un riferimento al padre.

Quando un processo termina **tutti** i suoi figli terminano e il padre di questo terminato può rilevarne la terminazione.

3.10 Processi leggeri (thread)

Un **thread** è un'entità di esecuzione che condivide codice e dati con altri thread ad esso associati.
Task: insieme di thread che riferiscono lo stesso codice e gli stessi dati.

Un **processo pesante** equivale a un task con un solo thread.



3.11 Processi interagenti

Il SO mette a disposizione dei meccanismi che possono permettere a processi di comunicare.
So classificano i processi come:

- Processi **indipendenti**: due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa.
- Processi **interagenti**: P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata da P2, e/o viceversa.

3.11.1 Processi interagenti

Tipi di interazione:

- **Cooperazione**: l'interazione consiste nello scambio di informazioni al fine di eseguire un'attività comune.
- **Competizione**: i processi interagiscono per sincronizzarsi nell'accesso a risorse comuni.
- **Interferenza**: interazione non desiderata e potenzialmente deleteria tra processi.

L'interazione tra processi può avvenire mediante:

- **Memoria condivisa**: il SO consente ai processi (thread) di condividere variabili; l'interazione avviene tramite l'accesso a variabili condivise.
- **Competizione**: i processi non condividono variabili e interagiscono mediante meccanismi di trasmissione/ricezione di messaggi; il SO prevede dei meccanismi a supporto dello scambio di messaggi.

Capitolo 4

Processi UNIX

UNIX è un SO *multi-programmato* a divisione di tempo: ovvero ad ogni processo viene assegnato un quanto di tempo terminato il quale il SO assegna la CPU ad un altro processo.

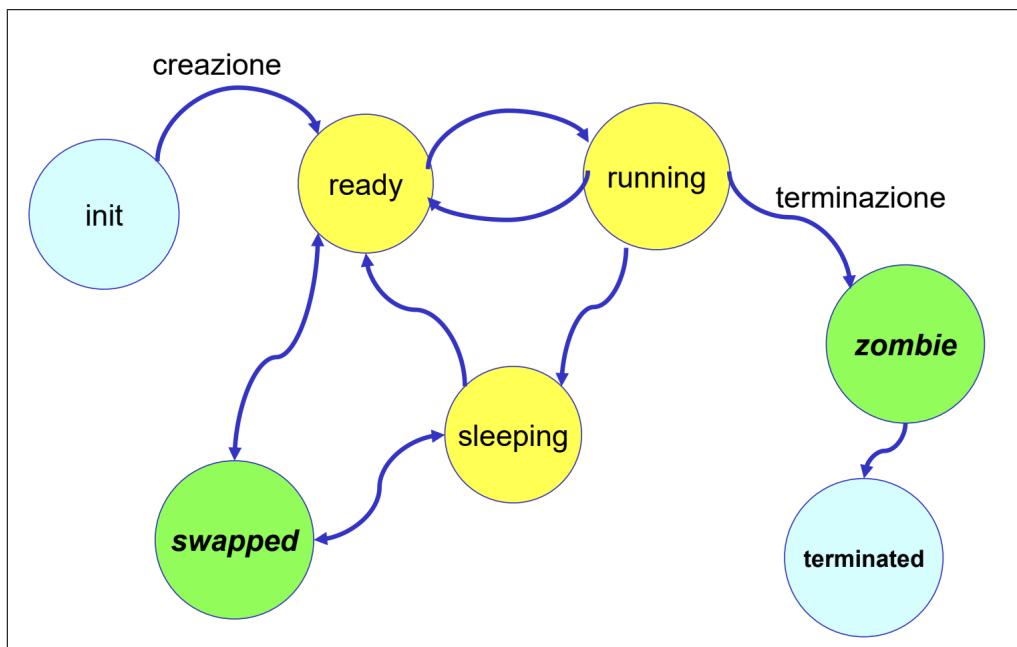
In UNIX il padre di tutti i processi si chiama **init**. Init genera dei processi figli detti: shell, che permettono all'utente di interagire con il SO.

Ogni processo ha un proprio spazio di indirizzamento locale e non condiviso.

Eccezioni:

- Il codice può essere condiviso.
- Il file system rappresenta un ambiente condiviso.

4.1 Stati di un processo in UNIX



Come nel caso generale:

- **Init**: caricamento in memoria del processo e inizializzazione delle strutture dati del SO.
- **Ready**: processo pronto.
- **Running**: processo a cui è stata assegnata la CPU.
- **Sleeping**: processo in attesa di un evento.
- **Terminated**: deallocazione del processo dalla memoria.

In aggiunta:

- **Zombie**: processo è terminato, ma è in attesa che il padre ne rilevi lo stato di terminazione.
- **Swapped**: processo (o parte di esso) è temporaneamente trasferito in memoria secondaria.

4.1.1 Processi swapped

Sono processi gestiti dallo scheduler a medio termine. Esso decide quali processi devono passare da sleeping a swapped e viceversa oppure passare a ready.

Lo scheduler a medio termine può effettuare due azioni:

- **Swap-out**: passaggio delle strutture dati associate ai processi dalla memoria centrale alla memoria secondaria.
- **Swap-in**: azione opposta dello swap-out.

4.2 Rappresentazione dei processi in UNIX

In UNIX il codice dei processi è rientrante: più processi possono condividere lo stesso codice (text). Dunque codice e dati devono essere separati. In memoria codice e dati sono allocati in posizioni distinte; questo modello viene detto: a codice puro.

Nel SO esiste la **text table**, una struttura dati globale che contiene i puntatori dei codici utilizzati.

L'elemento della text table si chiama **text structure**, e contiene: il puntatore al codice (se il processo è swapped, riferimento a memoria secondaria); il numero di processi che lo condividono.

Process Control Block (PCB): il descrittore del processo in UNIX è rappresentato da 2 strutture dati:

- **Process structure**: informazioni necessarie al sistema per la gestione del processo.
- **User structure**: informazioni necessarie al SO solo se il processo è residente in memoria centrale.

Process structure:

- Process identifier (PID): intero positivo che individua univocamente il processo.
- Stato del processo.
- Puntatori alle varie aree dati e stack associati al processo.
- Riferimento indiretto al codice: la process structure contiene il riferimento all'elemento della text table associato al codice del processo.
- Informazioni di scheduling.
- Riferimento al processo padre (PID del padre)
- Info relative alla gestione di segnali.
- Puntatori al processo successivo in code di scheduling.
- Puntatore alla user structure.

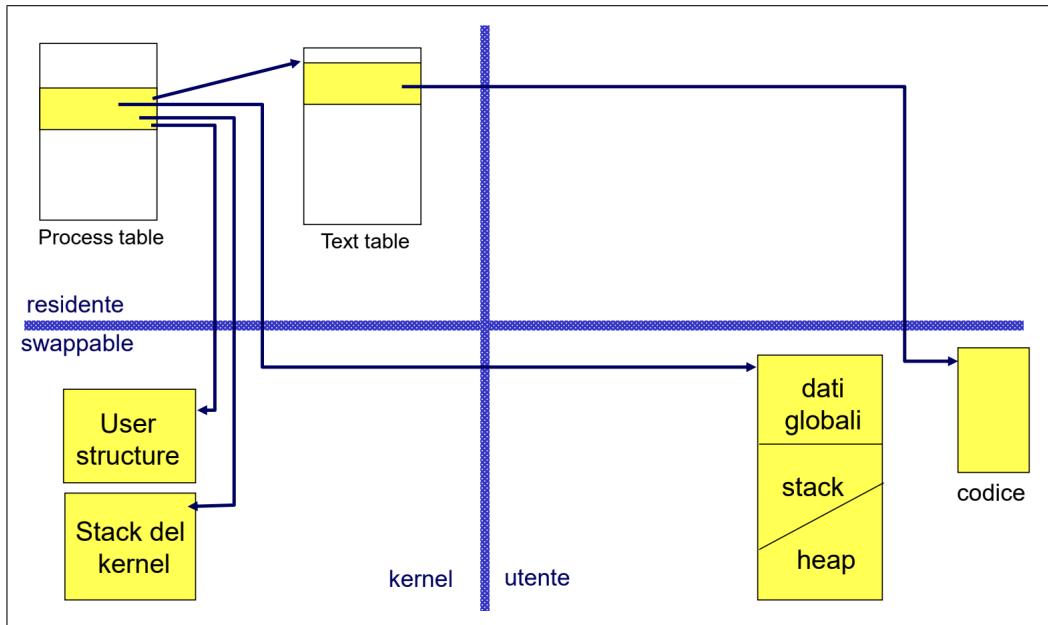
User structure:

- Copia dei registri di CPU
- Informazioni sulle risorse allocate
- Informazioni sulla gestione di segnali
- Ambiente del processo: direttorio corrente, utente, gruppo, argc/argv, path, ...

4.3 Immagine di un processo UNIX

L'immagine di un processo è insieme aree di memoria e strutture dati associate al processo.

L'immagine è accessibile nei modi user e kernel, e suddivisa in parte swappable e non swappable.



4.4 System call per la gestione di processi

- Creazione di processi: `fork()`
- Sostituzione di codice e dati: `exec...()`
- Terminazione: `exit()`
- Sospensione in attesa della terminazione di figli: `wait()`

4.5 Fork

La funzione `fork()` permette a un processo di generare un processo figlio.

Padre e figlio condividono lo stesso codice. Il figlio eredita una copia dei dati (di utente e di kernel) del padre.

Effetti della fork:

- Allocazione di una nuova process structure nella process table associata al processo figlio e sua inizializzazione.
- Allocazione di una nuova user structure nella quale viene copiata la user structure del padre.
- Allocazione dei segmenti di dati e stack del figlio nei quali vengono copiati dati e stack del padre.
- Aggiornamento del riferimento text al codice eseguito.

Dopo la generazione del figlio il padre può decidere se: operare contemporaneamente ad esso, oppure attendere la sua terminazione (system call `wait()`).

Quindi dopo una `fork` abbiamo:

- Concorrenza: padre e figlio procedono in parallelo (è possibile che il padre si metta in attesa della terminazione del figlio se lo sviluppatore utilizza una `wait()`).
- Lo spazio degli indirizzi è duplicato: ogni variabile del figlio è inizializzata con il valore assegnatole dal padre prima della `fork()`.
- La user structure è duplicata.

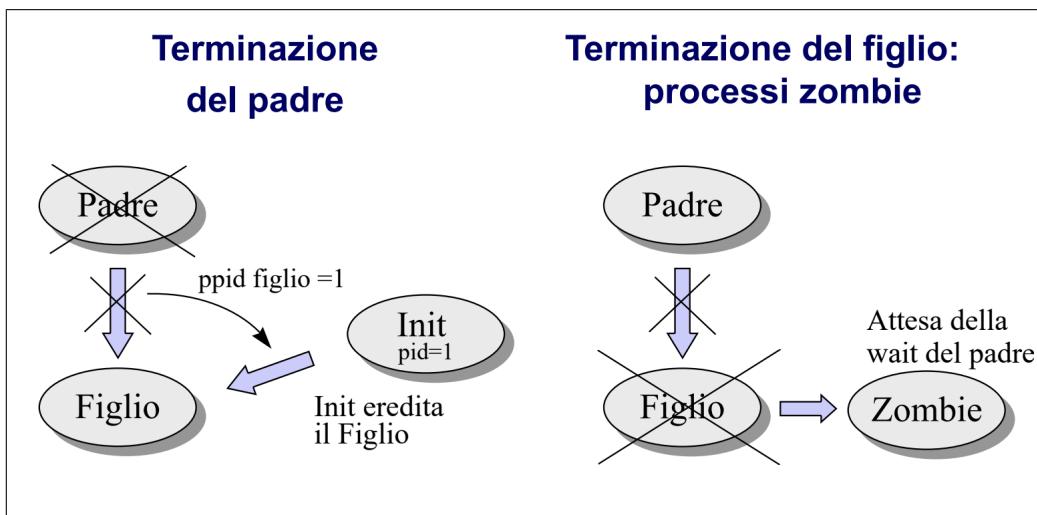
4.6 Terminazione di processi

La terminazione di processi può avvenire in due modi:

- Involontariamente: tentativi di azioni illegali; interruzione mediante segnale.
- Volontariamente: chiamata alla funzione exit(); esecuzione dell'ultima istruzione.

Effetti della exit:

- Chiusura dei file aperti non condivisi da altri processi.
- Terminazione del processo con due possibili casi:
 1. Se il processo che termina ha figli in esecuzione, il processo init adotta i figli dopo la terminazione del padre.
 2. Se il processo termina prima che il padre ne rilevi lo stato di terminazione con la system call wait(), il processo passa nello stato zombie.



4.7 Wait

Lo stato di terminazione può essere rilevato dal padre tramite le system call wait.

Effetti della wait:

- Se tutti i figli non sono ancora terminati, il processo si sospende in attesa della terminazione del primo di essi.
- Se almeno un figlio è già terminato ed il suo stato non è stato ancora rilevato (cioè è in stato zombie), wait() ritorna immediatamente con il suo stato di terminazione (nella variabile status).
- Se non esiste neanche un figlio, wait() NON è sospensiva e ritorna un codice di errore.

4.8 Exec

Mediante fork() i processi padre e figlio condividono il codice e lavorano su aree dati duplicate. In UNIX è possibile differenziare il codice dei due processi mediante una system call della famiglia exec: `exec1()` `execle()` `execlp()` `execv()` `execve()`, `execvp()`...

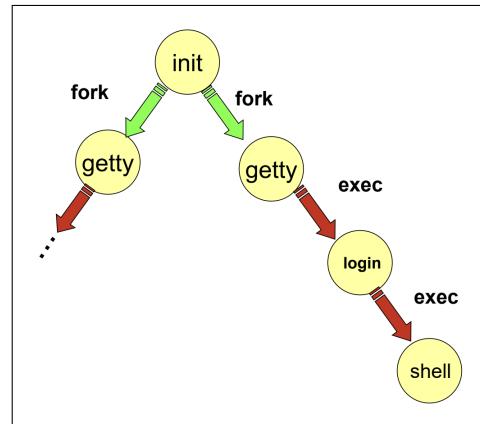
Effetto principale di system call famiglia exec:

Vengono sostituiti codice ed eventuali argomenti di invocazione del processo che chiama la system call, con codice e argomenti di un programma specificato come parametro della system call.

4.9 Inizializzazione processi

I processi vengono inizializzati in questo modo:

- Init genera un processo figlio per ogni terminale.
- Tramite il programma getty si controlla l'accesso al sistema effettuando exec login.
- In caso di accesso corretto, login esegue la shell.



Capitolo 5

Interazione fra processi

Classificazione dei tipi di interazione tra processi:

- Processi indipendenti: due processi sono indipendenti se l'esecuzione di ognuno non è in alcun modo influenzata dall'altro.
- Processi interagenti:
 - Cooperanti: i processi interagiscono volontariamente per raggiungere obiettivi comuni.
 - Competizione: i processi, in generale, non fanno parte della stessa applicazione, ma interagiscono indirettamente per l'acquisizione di risorse comuni.

5.1 Processi interagenti

Nei processi interagenti l'interazione può avvenire in due modi:

- **Comunicazione**: scambio di informazioni tra i processi interagenti.
- **Sincronizzazione**: l'interazione avviene tramite l'imposizione di vincoli temporali assoluti relativi.

Nei processi interagenti è possibile realizzare l'interazione ad ambiente locale, oppure ad ambiente globale.

Ambiente locale: non c'è condivisione di variabili e si parla di processo pesante. L'interazione avviene tramite comunicazione, quindi scambio di messaggi; oppure tramite sincronizzazione, invio di segnali tra processi.

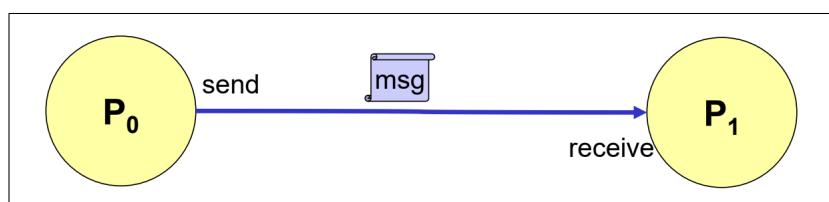
Ambiente globale: processi diversi possono condividere parte dello spazio di indirizzamento. L'interazione può avvenire tramite la condivisione di variabili.

5.2 Processi interagenti mediante scambio di messaggi

Facendo riferimento al modello ad ambiente locale: non abbiamo memoria condivisa; i processi, per comunicare, devono utilizzare il mezzo di comunicazione.

Il SO offre dei meccanismi di supporto alla comunicazione tra processi.

È necessario che il sistema operativo metta a disposizione due operazioni per lo scambio di messaggi: send e receive. Questo avviene mediante un canale di comunicazione tra i due processi.



5.3 Naming

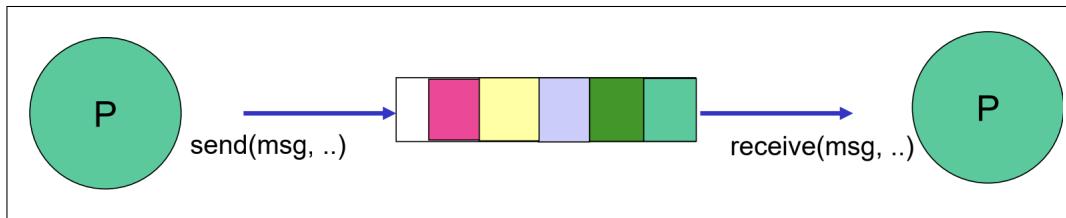
La destinazione di un messaggio viene specificata in due modi:

1. **Comunicazione diretta:** al messaggio viene associato l'identificatore del processo destinatario (naming esplicito).
2. **Comunicazione indiretta:** il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio.

5.4 Buffering del canale

Ogni canale di comunicazione è caratterizzato da una capacità: numero dei messaggi che è in grado di gestire contemporaneamente.

La gestione avviene tramite una coda di tipo FIFO. I messaggi vengono posti in una coda in attesa di essere ricevuti. La lunghezza massima della coda rappresenta la capacità del canale.



Canale con capacità nulla: non vi è accodamento perché il canale non è in grado di gestire i messaggi in attesa. Il processi comunicanti devono sincronizzarsi all'atto di spedire/ ricevere il messaggio: comunicazione sincrona o rendev vous. La send e la receive possono sospendersi.

Capacità limitata: esiste un limite N alla dimensione della coda.

- Se la coda non è piena il nuovo messaggio viene posto in fondo della coda.
- Se la coda è piena la send si sospende in attesa che la coda si svuoti di un messaggio.
- Se la coda è vuota: la receive può sospendersi.

Capacità illimitata: lunghezza della coda teoricamente infinita. L'invio sul canale non è sospensivo. Mentre la ricezione può essere sospensiva nel caso il canale sia vuoto.

5.5 Sincronizzazione tra processi

Si è visto che due processi possono interagire per:

- **Cooperare:** i processi interagiscono allo scopo di perseguire un obiettivo comune.
- **Competere:** i processi possono essere logicamente indipendenti. Ma, necessitano della stessa risorsa (file, variabile, ...) per la quale sono stati imposti dei vincoli di accesso. I processi sono sincronizzarsi devono sincronizzarsi per escludersi mutualmente nel tempo per accedere alla stessa risorsa.

In entrambi i casi è necessario dotarsi di strumenti di sincronizzazione.

La sincronizzazione permette di imporre vincoli temporali sulle operazioni dei processi interagenti.

Si possono fare degli esempi per cooperazione e competizione.

Nella competizione: si impone un ordine cronologico delle azioni fra processi distinti; le operazioni di comunicazione devono avvenire secondo un ordine prefissato.

Nella competizione: la sincronizzazione dei processi avviene tramite mutua esclusione. Cioè solo un processo alla volta può accedere alla risorsa condivisa.

5.5.1 Sincronizzazione nel modello ad ambiente locale

Gli accessi alle risorse condivise vengono controllati e coordinati da SO. La sincronizzazione avviene mediante meccanismi offerti da SO che consentono la notifica di eventi asincroni (di solito privi di contenuto informativo o con contenuto minimale) tra un processo ed altri.

5.5.2 Sincronizzazione nel modello ad ambiente globale

- Cooperazione: lo scambio di messaggi avviene attraverso strutture dati condivise (ad es. mailbox).
- Competizione: le risorse sono rappresentate da variabili condivise (ad esempio, puntatori a file).

In entrambi i casi è necessario sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa. In ogni caso quindi abbiamo il problema della **mutua esclusione**.

5.6 Problema della mutua esclusione

Il problema nasce dal fatto che abbiamo una condivisione di variabili, e potrebbe essere necessario impostare accessi concorrenti alla stessa risorsa.

Sezione critica: sequenza di istruzioni mediante la quale un processo accede e può aggiornare variabili condivise.

Mutua esclusione: Ogni processo esegue le proprie sezioni critiche in modo mutuamente esclusivo rispetto agli altri processi.

In generale, per garantire la mutua esclusione nell'accesso a variabili condivise, ogni sezione critica è:

- Preceduta da un prologo (entry section), mediante il quale il processo ottiene l'autorizzazione all'accesso in modo esclusivo.
- Seguita da un epilogo (exit section), mediante il quale il processo rilascia la risorsa.

5.7 Possibile soluzione: semafori di Dijkstra

Un semaforo è un tipo di dato astratto condiviso fra due o più processi al quale sono applicabili due operazioni (system call a esecuzione non interrompibile):

- **Wait(*s*).**
- **Signal(*s*).**

s è una variabile di tipo semaforo alla quale sono associate: una variabile intera non negativa (*s.value*) con valore iniziale ≥ 0 ; una coda di processi (*s.queue*).

```
void wait(s){  
    if (s.value == 0)  
        <processo viene sospeso e descrittore  
        inserito in s.queue>  
    else s.value = s.value-1;  
}
```

```
void signal(s){  
    if (<esiste un processo in s.queue>)  
        <descrittore viene estratto da s.queue  
        e stato modificato in pronto>  
    else s.value = s.value+1;  
}
```

5.7.1 Atomicità della wait e della signal

Affinché sia rispettato il vincolo di mutua esclusione dei processi nell'accesso al semaforo (mediante wait/signal), wait() e signal() devono essere operazioni indivisibili (azioni atomiche).

Durante un'operazione sul semaforo (wait() o signal()) nessun altro processo può accedere al semaforo fino a che l'operazione non è completa o bloccata (sospensione nella coda).

Capitolo 6

Segnali

I processi interagenti possono aver bisogno di meccanismi di sincronizzazione. In UNIX non c'è condivisione di spazio di indirizzamento tra processi. Serve quindi un meccanismo per un modello ad ambiente locale. Cioè i **segnali**.

I segnali sono delle interruzione SW a un processo, che notifica un evento asincrono. (Esempio di segnale: CTRL+C).

6.1 Segnali UNIX

Un segnale può essere inviato da:

- Dal kernel del SO a un processo.
- Da un processo utente ad altri processi utente (es: comando `kill()`).

Quando un processo riceve un segnale ha tre comportamenti:

- Gestire il segnale con una funzione detta **handler**, che il programmatore ha definito per quello specifico segnale.
- Eseguire un'azione predefinita dal SO (azioni di default).
- Ignorare il segnale.

Nei primi due casi, il processo reagisce in modo asincrono al segnale.

1. Interruzione dell'esecuzione delle istruzioni che stava svolgendo.
2. Esecuzione del codice relativo alla funzione handler. Se non definita esegue l'azione di default.
3. Ritorno alla prossima istruzione del codice del processo interrotto.

Non tutti segnali che non possono essere gestiti in modalità scelta esplicitamente dai processi: **SIGKILL** e **SIGSTOP** non sono né intercettabili, né ignorabili. Qualunque processo, alla ricezione di **SIGKILL** o **SIGSTOP** esegue sempre l'azione di default.

```

#define SIGHUP 1 /* Hangup (POSIX). Action: exit */
#define SIGINT 2 /* Interrupt (ANSI). CTRL+C. Action: exit */
#define SIGQUIT 3 /* Quit (POSIX). Action: exit, core dump */
#define SIGILL 4 /* Illegal instr (ANSI). Action: exit,
                           core dump */
...
#define SIGKILL 9 /* Kill, unblockable (POSIX). Action: exit */
#define SIGUSR1 10 /* User-def sig1 (POSIX). Action: exit */

#define SIGSEGV 11 /* Segm. violation (ANSI). Action: exit, core
                           dump */
#define SIGUSR2 12 /* User-def sig2 (POSIX). Action: exit */
#define SIGPIPE 13 /* Broken pipe (POSIX). Action: exit */
#define SIGALRM 14 /* Alarm clock (POSIX). Action: exit */
#define SIGTERM 15 /* Termination (ANSI). Action: exit */
...
#define SIGCHLD 17 /* Chld stat changed (POSIX). Action: ignore */
#define SIGCONT 18 /* Continue (POSIX). Action ignore */
#define SIGSTOP 19 /* Stop, unblockable (POSIX). Action: stop */
...

```

6.2 System call signal

La system call signal ha l'obiettivo di permettere la gestione esplicita di un segnale. Quindi siamo in grado di dire quale funzione il SO deve invocare nel caso il processo riceva un certo segnale.

6.3 Segnale SIGCHLD

SIGCHLD è il segnale che il kernel dell'SO invia al processo padre quando uno dei suoi figli termina. Tramite l'uso di esso è possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante una funzione apposita handler per la gestione di SIGCHLD:

- La funzione handler verrà attivata in modo asincrono alla ricezione del segnale.
- Handler chiamerà `wait()` con cui il padre potrà raccogliere ed eventualmente gestire lo stato di terminazione del figlio.

6.4 Segnali e fork

Le associazioni segnali-azioni vengono registrate nella user structure del processo.

Siccome la fork copia la user structure del padre in quella del figlio, padre e figlio condividono lo stesso codice. Quindi, il figlio eredita dal padre le informazioni relative alla gestione dei segnali.

6.5 Segnali ed exec

Il comando exec sostituisce codice e dati del processo invocante. La user structure viene mantenuta, tranne le informazioni legate al codice del processo (ad esempio, le funzioni di gestione dei segnali, che dopo `exec()` non sono più visibili).

Quindi dopo exec: un processo ignora gli stessi segnali ignorati prima dell'exec; i segnali a default, ma, i segnali che prima erano gestiti vengono riportati a default.

6.6 System call di segnali

- `kill()` permette di inviare un segnale ad un processo.
- `sleep()` provoca la sospensione del processo per N secondi. Se il processo riceve un segnale durante il periodo di sleep, esso viene risvegliato.
- `alarm()` imposta un timer che dopo N secondi invierà allo stesso processo il segnale SIGALRM.

- `pause()` sospende il processo fino alla ricezione di un qualunque segnale.

6.7 Modello affidabile dei segnali

Alcune domande di affidabilità sul modello di gestione dei segnali.

Il gestore rimane installato?

In caso negativo è possibile reinstallare il gestore all'interno dell'handler. Rimane ancora da chiedersi cosa succede se arriva un segnale tra l'inizio della funzione handler, ma prima della signal.

```
void handler(int s){  
    signal(SIGUSR1, handler);  
    printf("Processo %d: segnale %d\n", getpid(), s);  
    ... }  
  
Ma che cosa  
succede se qui arriva  
un nuovo segnale?
```

Così perderemo il segnale.

Che cosa succede se arriva un segnale durante l'esecuzione dell'handler?

Alcune possibilità sono:

- Innestamento della routine di gestione.
- Perdita del segnale.
- Accodamento dei segnali.

6.8 Sigaction

La primitiva signal() non è portabile perché ha una semantica diversa in diverse versioni di UNIX. Per ovviare a questo problema, POSIX.1 introduce la sigaction().

La sigaction() permette di esaminare e/o modificare l'azione associata con un particolare segnale. Si noti che POSIX.1 richiede che un segnale rimanga installato (fino a una modifica esplicita del comportamento).

Con la sigaction() è anche possibile specificare il riavvio automatico delle system call interrotte da un segnale.

6.9 Signal vs Sigaction

Signal ha una semantica variabile reliable/unreliable:

- Unreliable in alcune versioni di Unix/Linux.
- Segnali da reinstallare ogni volta, corsa critica tra inizio handler e reinstallazione handler come prima istruzione dell'handler.
- Possibile esecuzione innestata dell'handler se ricezione dello stesso segnale quando siamo ancora nell'handler.

Sigaction invece è sempre reliable:

- Semantica ben definita, identica in ogni versione di UNIX/Linux.
- Non c'è bisogno di reinstallare l'handler.
- Non perdiamo segnali: il segnale che ha causato l'attivazione dell'handler è automaticamente bloccato fino alla fine dell'esecuzione dell'handler stesso.

Capitolo 7

Comunicazione tra processi

Si ricordi che UNIX è un modello ad ambiente locale, e l'interazione tra processi può avvenire con la condivisione di file. Questo però è solitamente uno strumento di difficile gestione.

Esistono degli strumenti specifici nel SO che permettono la comunicazione tra processi:

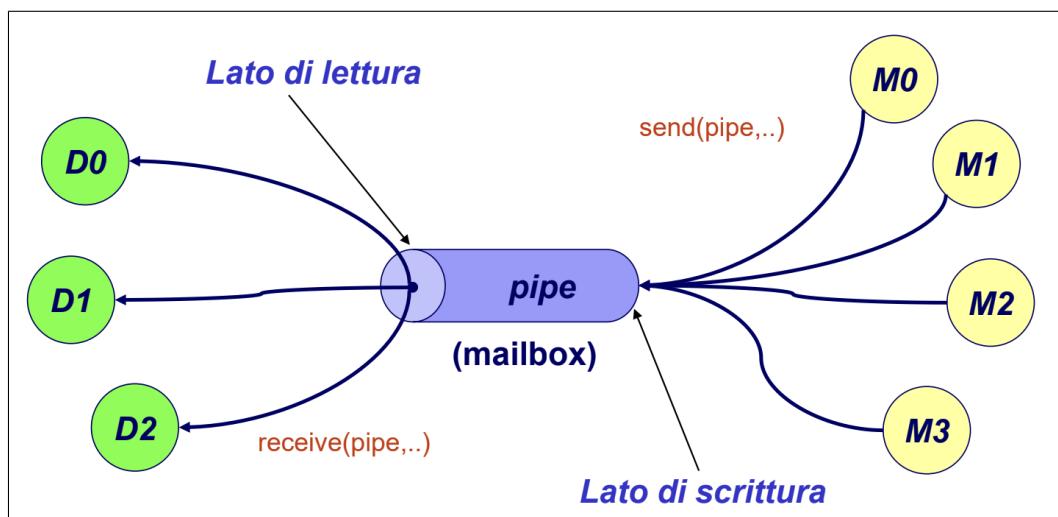
- Pipe.
- Fifo.
- Socket.

7.1 Pipe

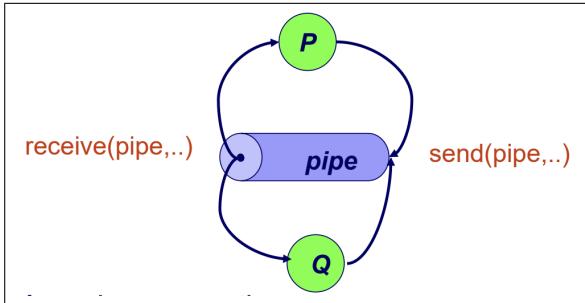
La pipe è un canale di comunicazione tra processi. Ha un canale di comunicazione unidirezionale, accessibile mediante due estremi distinti, uno di lettura e uno di scrittura. Teoricamente è uno strumento di comunicazione multi-a-molti, in quanto la stessa pipe può essere utilizzata da più processi per spedire/ricevere messaggi. Ha una capacità limitata, cioè in una pipe possono essere depositati un numero limitato di messaggi.

7.2 Comunicazione attraverso pipe

Mediante la pipe, la comunicazione tra processi è indiretta (senza naming esplicito): modello mailbox.



7.3 Unidirezionalità / bidirezionalità



Uno stesso processo potrebbe sia depositare (send) che prelevare (receive) messaggi dalla pipe.

la pipe può anche consentire una comunicazione "bidirezionale" tra P e Q (ma il programmatore deve rigidamente disciplinarne l'uso per l'utilizzo corretto).

7.4 System call pipe

Per invocare una pipe bisogna invocare la system call `pipe()`. Passando come argomento un array di 2 interi. Dove il primo elemento rappresenta il lato di lettura e il secondo il lato di scrittura.

L'array rappresenterebbe una sequenza di due **file descriptor**.

Se la creazione della pipe ha successo vengono allocati due nuovi elementi nella tabella dei file aperti del processo.

si può accedere alla pipe mediante le system call di lettura/scrittura su file `read()`, `write()`.

7.5 Processi che possono comunicare mediante pipe

È possibile far comunicare solo i processi che appartengono alla stessa gerarchia. Ovvero devono avere un antenato in comune.

Per esempio:

- Due processi fratelli.
- Un padre e un figlio.
- Un nonno e un nipote.

7.6 Chiusura della pipe

Ogni processo può chiudere un estremo della pipe con la system call `close()`.

È importante notare che la comunicazione non è più possibile su di un estremo della pipe, quando tutti i processi che avevano visibilità di quell'estremo hanno compiuto una close.

7.7 System call dup

La system call `dup()` serve per duplicare un elemento della tabella dei file aperti di processo.

`dup` copia l'elemento della tabella dei file aperti nella prima posizione libera.

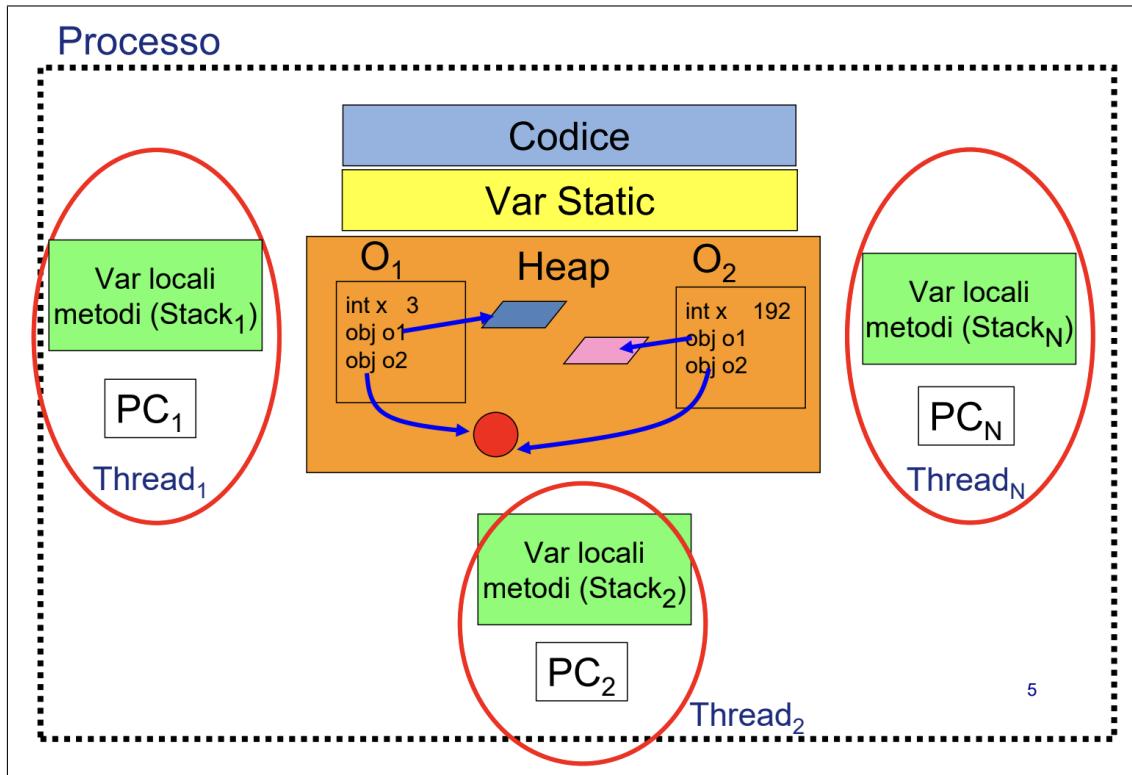
Capitolo 8

Java thread

Si ricorda che per thread si intende un singolo flusso sequenziale di controllo all'interno di un processo. Esso condivide codice e dati con altri thread ad esso associati.

Multithreading: esecuzione concorrente (in parallelo o interleaved) di diversi thread nel contesto di un unico processo.

Un thread non ha spazio di memoria riservato per dati e heap: tutti i thread appartenenti allo stesso processo condividono lo stesso spazio di indirizzamento. Ha stack e program counter privati.



8.1 Modello ad ambiente locale

Caratteristiche del modello computazionale multithreaded (modello ad ambiente globale):

- I thread non hanno uno spazio di indirizzamento riservato. Si ha la possibilità di definire dati thread local, cioè accessibili da un unico thread.
- I thread hanno execution stack e un program counter privati. Ciò permette a thread diversi dello stesso processo di trovarsi in punti diversi del codice.
- La comunicazione fra thread può avvenire direttamente, tramite la condivisione di aree di memoria. Bisogna dotarsi di meccanismi di sincronizzazione.

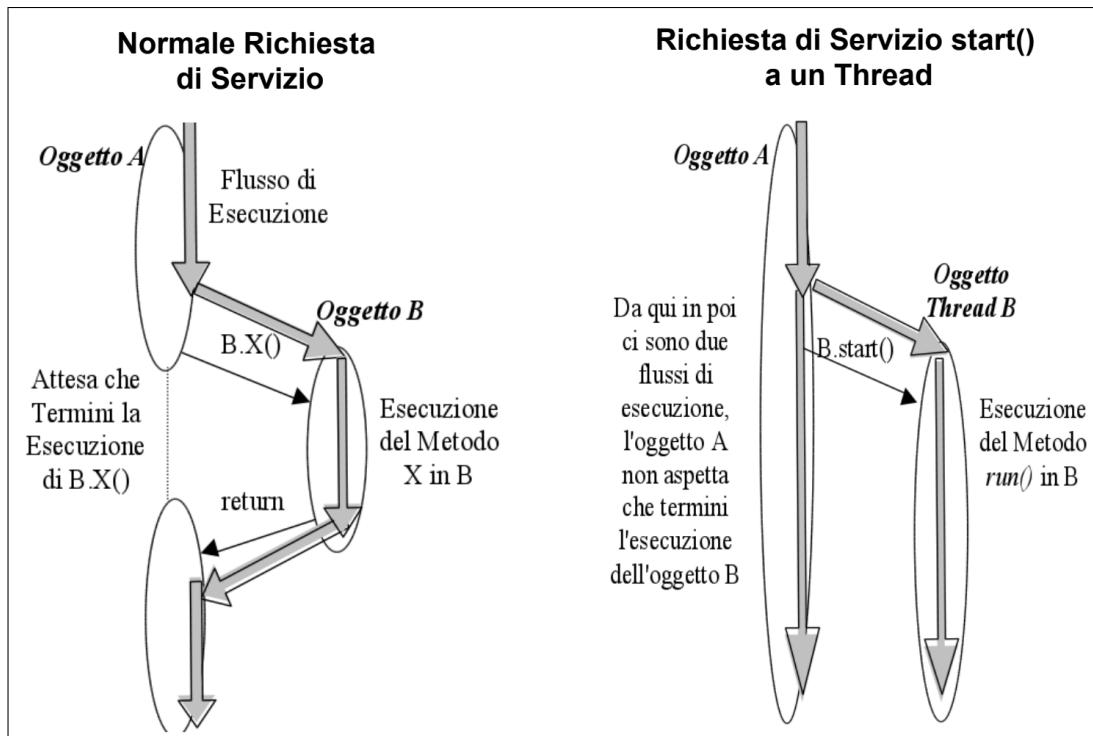
Nel modello ad ambiente globale il termine "processo" non identifica più una singolo flusso di esecuzione di un programma, ma invece il contesto di esecuzione di più thread, con tutti i dati che possono essere condivisi da questi ultimi.

8.2 Multithreading in Java

Il linguaggio Java supporta nativamente il multithreading.

L'esecuzione di una JVM corrisponde all' esecuzione di un unico processo. Tutto quello che viene eseguito dalla JVM corrisponde ad un thread.

Un thread in Java è un oggetto di tipo `Thread` che ha la particolarità di non terminare. Esso procede in parallelo con il thread che invoca il metodo `start()` del thread appena creato.



A livello di linguaggio ci sono due metodi per creare un thread:

- Istanziare `Thread` passando come parametro un oggetto ottenuto implementando l'interfaccia `Runnable`.
- Estendere direttamente la classe `Thread`.

8.3 Problema dei metodi stop e suspend

Il metodo `stop()` forza la terminazione dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente liberate (lock inclusi), come effetto della propagazione dell'eccezione `ThreadDeath`.

Il metodo `suspend()` blocca l'esecuzione di un thread in attesa di una successiva operazione di resume. Non libera le risorse (neanche i lock) impegnate dal thread (possibilità di deadlock).

`stop` e `suspend` rappresentano azioni "brutali" sul ciclo di vita di un thread → rischio di determinare situazioni di deadlock o di inconsistenze:

- Se il thread sospeso aveva acquisito una risorsa in maniera esclusiva, tale risorsa rimane bloccata e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il lock su di essa.

- Se il thread interrotto stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera atomica, l'interruzione può condurre a uno stato inconsistente del Sistema.

I metodi `stop` e `resume` sono **deprecati**. Si consiglia di realizzare tutte le azioni di sincronizzazione fra thread tramite i meccanismi di sincronizzazione forniti dal linguaggio.

Il modo più corretto per stoppare un thread è utilizzare una variabile di appoggio.

8.4 Sincronizzazione di thread

Differenti thread condividono lo stesso spazio di memoria (heap). È possibile che più thread accedano contemporaneamente a uno stesso oggetto, invocando un metodo che modifica lo stato dell'oggetto. Lo stato finale dell'oggetto sarà funzione dell'ordine con cui i thread accedono ai dati.

In Java, la sincronizzazione è implementata a livello di linguaggio (modificatore `synchronized`) e attraverso primitive di sincronizzazione di basso (`wait()`, `notify()`, `notifyAll()`) e di alto livello (concurrency utilities).

8.4.1 Synchronized

In pratica:

- A ogni oggetto Java è automaticamente associato un unico lock.
- Quando un thread vuole accedere ad un metodo/blocco `synchronized`, si deve acquisire il lock dell'oggetto (impedendo così l'accesso ad ogni altro thread).
- Il lock viene automaticamente rilasciato quando il thread esce dal metodo/blocco `synchronized`.
- Il thread che non riesce ad acquisire un lock rimane sospeso sulla richiesta della risorsa fino a che il lock non è disponibile.

Ad ogni oggetto viene assegnato un solo lock a livello di oggetto (non di classe né di metodo in Java). Questo implica che due thread non possono accedere contemporaneamente a due metodi/blocchi `synchronized` diversi di uno stesso oggetto.

Tuttavia altri thread sono liberi di accedere a metodi/blocchi non `synchronized` associati allo stesso oggetto.

Capitolo 9

Astrazione di File system

Il SO fornisce servizi di gestione (file system) di come i dati vengono acceduti e memorizzati all'interno della memoria di massa. Il file system offre un accesso astratto alle informazioni. Realizzando concetti astratti come:

- **File:** unità logica di memorizzazione.
- **Direttorio:** insieme di file (e direttori).
- **Partizione:** insieme di file associato ad un particolare dispositivo fisico.

9.1 File

Un file è un insieme di informazioni di qualunque tipo (programmi, dati in binario, dati in testuale, ...). Sono rappresentati come insiemi di **record logici**.

Ogni file è individuato da un nome simbolico mediante il quale può essere riferito. Ogni file è caratterizzato da un insieme di **attributi**. Esempi di attributi:

- Tipo.
- Indirizzo.
- Dimensione.
- Data e ora.

Descrittore del file: struttura dati che contiene gli attributi di un file. Ogni descrittore di file deve essere memorizzato in modo persistente: SO mantiene l'insieme dei descrittori di tutti i file presenti nel file system in apposite strutture in memoria secondaria.

9.2 Operazioni sui file

Il compito del SO è consentire l'accesso on-line ai file: ogni volta che un processo modifica un file, tale cambiamento è immediatamente visibile per tutti gli altri processi.

Tipiche operazioni:

- **Creazione:** allocazione di un file in memoria secondaria e inizializzazione dei suoi attributi.
- **Lettura** di record logici dal file.
- **Scrittura:** inserimento di nuovi record logici all'interno del file.
- **Cancellazione:** eliminazione del file dal file system.

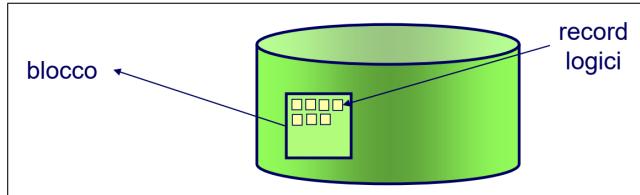
Se per ciascuna di queste operazioni fosse necessario per il SO localizzare il file nella memoria di massa e accedere alla memoria per recuperare/modificare il file, questo comporterebbe un costo troppo elevato.

Il SO per rendere tutto più efficiente mantiene in memoria principale una struttura dati che ha il compito di tenere traccia di tutti i file in uso. Questa struttura è detta: **tabella dei file aperti**.

Per ogni apertura di file va ad aggiungere un nuovo elemento nella tabella dei file aperti. In chiusura di un file avviene il salvataggio in memoria secondaria ed eliminazione dell'elemento associato nella tabella dei file aperti.

9.3 Blocchi & record logici

Ogni dispositivo di memorizzazione secondaria viene partizionato in blocchi (o record fisici).



Blocco: unità di trasferimento fisico nelle operazioni di I/O da/verso il dispositivo. Sempre di dimensione fissa.

L'utente vede il file come un insieme di record logici.

Uno dei compiti del SO è stabilire una corrispondenza tra record logici e blocchi.
Usualmente la dimensione di un blocco è molto più grande di quella di un record logico. Quindi all'interno di un blocco che contiene porzione di un certo file, troviamo numerosi record logici.

9.4 Metodi di accesso

L'accesso ai file può avvenire secondo varie modalità:

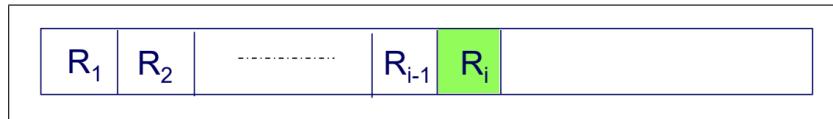
- Accesso sequenziale
- Accesso diretto
- Accesso a indice

Il metodo di accesso è indipendente dal tipo di dispositivo utilizzato e dalla tecnica di allocazione dei blocchi in memoria secondaria.

9.4.1 Accesso sequenziale

Il file è rappresentato da una sequenza di record logici.

Per accedere ad un particolare record logico R_i , è necessario accedere prima agli $i - 1$ record che lo precedono nella sequenza:



Ogni operazione di accesso (lettura/scrittura) posiziona il puntatore al file sull'elemento successivo a quello appena letto/scritto.

UNIX supporta questo tipo di accesso

9.4.2 Accesso diretto

Il file è un insieme di record logici numerati con associata la nozione di posizione.

Si può accedere direttamente a un particolare record logico specificandone il numero.

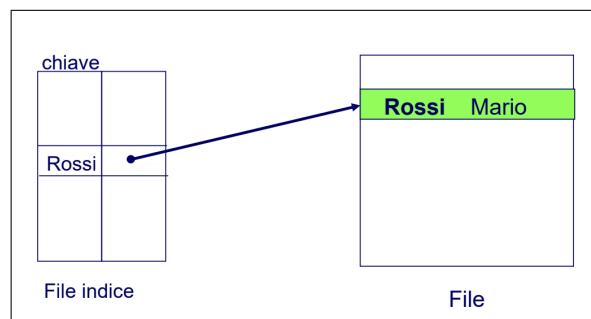
9.5 Accesso a indice

Ad ogni file viene associata una struttura dati contenente l'indice delle informazioni contenute.

Per accedere a un record logico, si esegue una ricerca nell'indice (utilizzando una chiave).

9.6 Direttorio

È uno strumento per organizzare i file all'interno del file system. I direttori sono realizzati mediante una struttura dati che associa al nome di ogni file come e/o dove esso è allocato in memoria di massa.



Operazioni sui direttori:

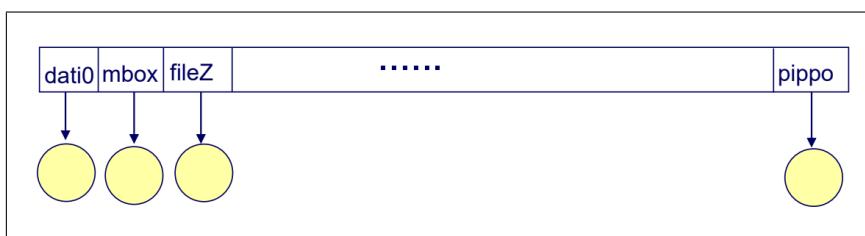
- **Creazione/cancellazione** di direttori.
- **Aggiunta/cancellazione** di file
- **Listing**: elenco di tutti i file contenuti nel directory.
- **Attraversamento** della directory.
- **Ricerca** di file nel directory.

La struttura logica delle directory può variare a seconda del SO:

- **A un livello**.
- **A due livelli**.
- **Ad albero**.
- **A grafo a-ciclico**.

9.6.1 A un livello

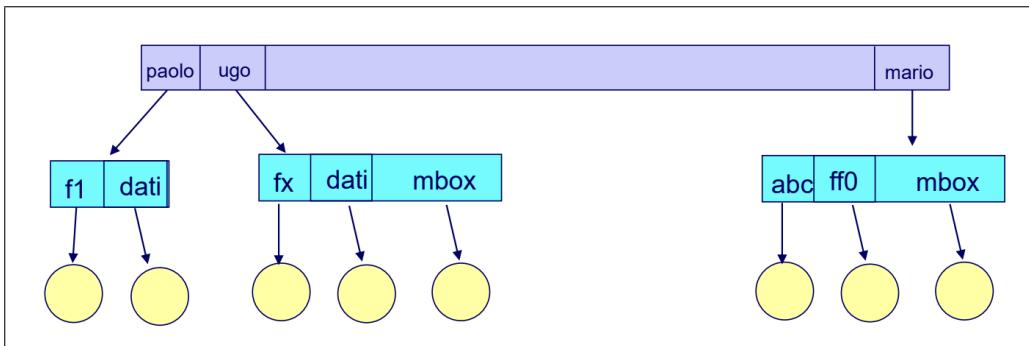
Una sola directory per ogni file system.



Con questa struttura abbiamo una serie di problemi legati all'unicità dei nomi e alla separazione di file per la multi-utenza.

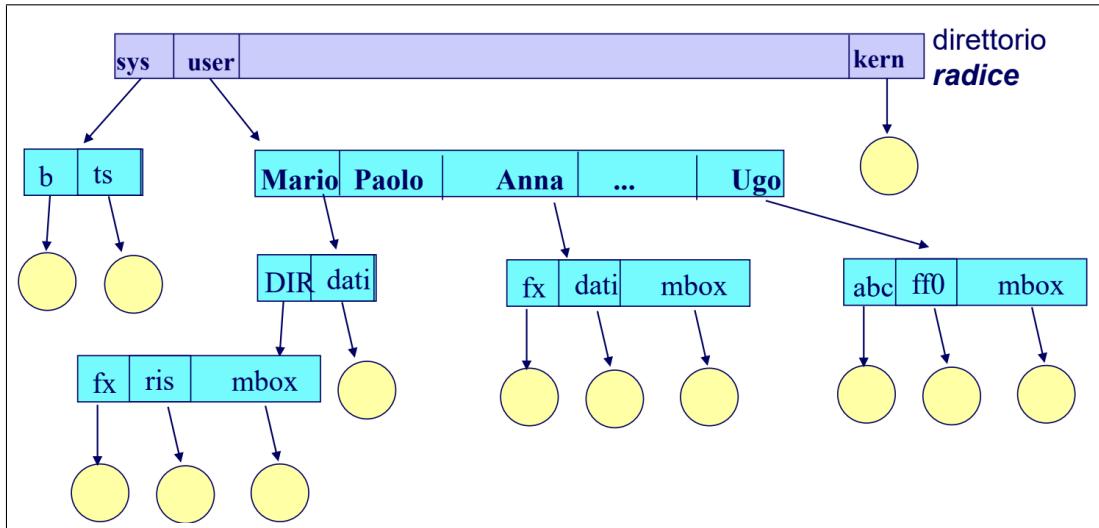
9.6.2 A due livelli

Il primo livello contiene una directory per ogni utente del sistema. Il secondo livello è una directory utenti a un livello.



9.6.3 Struttura ad albero

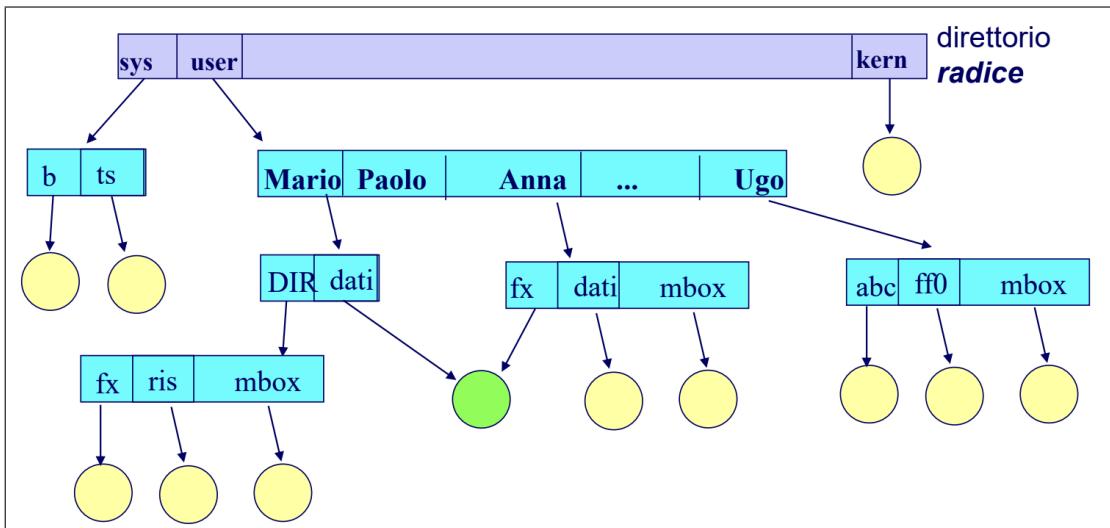
Organizzazione gerarchica ad N livelli. Ogni direttorio può contenere file e altri direttori.



9.6.4 Struttura a grafo aciclico

Questa struttura è quella utilizzata su UNIX e Linux. Estende la struttura ad albero con la possibilità di inserire dei link differenti allo stesso file.

Uno stesso file può essere riferito con nomi logici diversi.



9.7 Realizzazione del file system

Il SO si occupa di realizzare il file system sui dispositivi di memorizzazione di massa.

- Realizzazione dei descrittori e loro organizzazione.
- Allocazione dei blocchi fisici.
- Gestione dello spazio libero.

Come può essere realizzato il file system sulle unità disco?

Ogni blocco contiene un insieme di record logici contigui. Abbiamo tre metodi di allocazione:

- **Allocazione contigua.**
- **Allocazione a lista.**
- **Allocazione a indice.**

Un metodo di allocazione è il modo in cui il SO va a disporre i blocchi fisici all'interno della memoria di massa.

9.8 Allocazione contigua

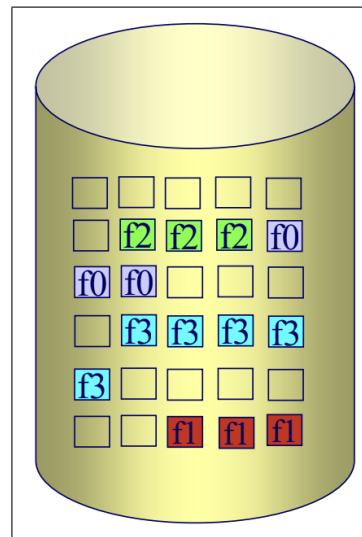
Ogni file è mappato su un insieme di blocchi fisicamente contingui.

Vantaggi:

- Costo della ricerca di un blocco.
- Possibilità di accesso sequenziale e diretto del file.

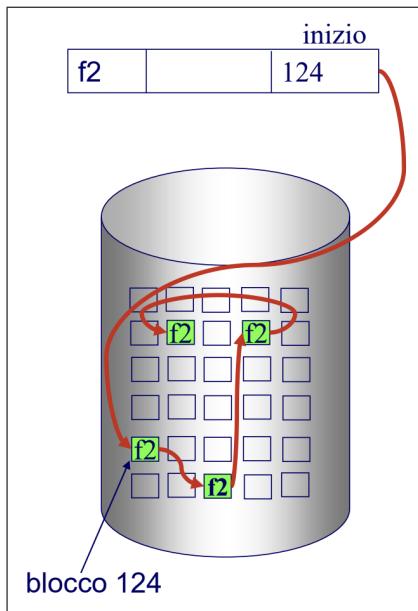
Svantaggi:

- Individuazione dello spazio libero per l'allocazione di un nuovo file.
- Frammentazione esterna: man mano che si riempie il disco, rimangono zone contigue sempre più piccole, a volte inutilizzabili. Questo necessita di operazioni di compattazione.
- Aumento dinamico delle dimensioni di file.



9.9 Allocazione dinamica

I blocchi sui quali viene mappato ogni file sono organizzati in una lista concatenata.



Per ciascuna directory abbiamo, oltre ai parametri, l'indirizzo del primo blocco che contiene parte del file.

Vantaggi:

- Non c'è frammentazione esterna.
- Minor costo di allocazione.

Svantaggi:

- Possibilità di errore se link danneggiato.
- Maggior occupazione (spazio occupato dai puntatori).
- Difficoltà di realizzazione dell'accesso diretto.
- Costo della ricerca di un blocco.

Un modo per rendere più efficiente l'accesso a lista concatenata è utilizzare delle tabelle di allocazione dei file (FAT, file allocation table). Per ogni partizione, viene mantenuta una tabella in cui ogni elemento rappresenta un blocco fisico. Ogni elemento contiene l'indice dei blocchi successivi.

9.10 Allocazione a indice

Nell'allocazione a lista permette di specificare all'interno dei blocchi dei puntatori dei blocchi successivi. Questo comporta però un tempo medio di accesso elevato e complessità della realizzazione del metodo di accesso diretto.

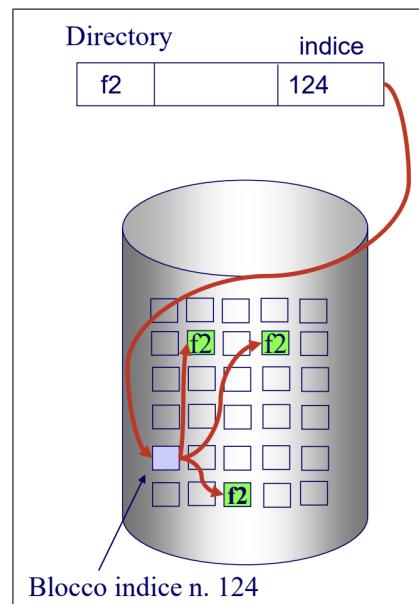
Nell'allocazione a indice tutti i puntatori ai blocchi utilizzati per l'allocazione di un determinato file sono concentrati in un unico blocco per quel file (blocco indice).

Vantaggi:

- Stessi vantaggi della lista.
- Supporto dell'accesso diretto più efficiente.
- Accesso più rapido ai singoli blocchi rispetto alle liste.

Svantaggi:

- Possibile scarso utilizzo dei blocchi indice per file piccoli.



Capitolo 10

File system UNIX

All'interno del file system di UNIX abbiamo un approccio omogeneo, ovvero tutto è un file. Vi sono tre categorie di file:

- File ordinari.
- Direttori.
- Dispositivi fisici: file speciali.

10.1 Nome, i-number, i-node

Ad ogni file può essere associato uno o più nomi simbolici. Ma, ad ogni file è associato uno e un solo descrittore (**i-node**), univocamente identificato da un intero, detto **i-number**.

10.2 Organizzazione file system UNIX

Il metodo di allocazione all'intero del file system di UNIX è ad indice, con più livelli di indirizzamento.

La formattazione del disco è in blocchi fisici.

La superficie del disco è suddivisa in quattro regioni:

- Boot block.
- Super block.
- i-list.
- Data blocks.

10.2.1 Boot Block

Contiene le procedure di inizializzazione del SO. Quindi tutte le funzionalità, moduli e componenti che consentono di eseguire il bootstrap del SO.

10.2.2 Super Block

Esso fornisce:

- Limite delle quattro regioni.
- Il puntatore a una lista di blocchi liberi.
- Il puntatore a una lista degli i-node liberi.
- Data blocks.

10.2.3 Data Block

Area del disco utilizzata per memorizzare i file. Contiene:

- Blocchi allocati.
- Blocchi liberi organizzati in una lista collegata.

10.2.4 i-List

i-List contiene la lista di tutti i descrittori (i-node) dei file normali, direttori e dispositivi presenti nel file system.

10.3 i-Node

i-Node è il descrittore del file. Contiene gli attributi che permettono di capire com'è strutturato il file:

- Tipo file.
- Proprietario del file, gruppo di utenti in cui il file è associato.
- Dimensione del file.
- Data di creazione e ultima modifica.
- 12 bit di protezione.
- Numero di link.
- 13-15 indirizzi di blocchi.

10.4 Directory

Anche le directory sono rappresentate nel file system da file. Ogni file-directory contiene un insieme di record logiccon struttura:

- Nome relativo.
- i-number.

Ogni record rappresenta un file appartenente alla directory.

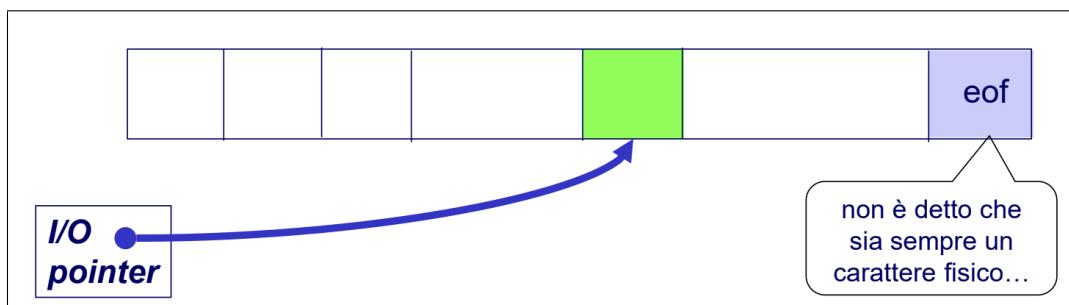
per ogni file (o directory) appartenente alla directory considerata, viene memorizzato il suo nome relativo, a cui viene associato il relativo i-number (che lo identifica univocamente).

10.5 Gestione del file system in UNIX

Per la gestione del file system vanno considerati: le strutture dati di sistema per il supporto all'accesso e alla gestione di file; principali system call per l'accesso e la gestione di file.

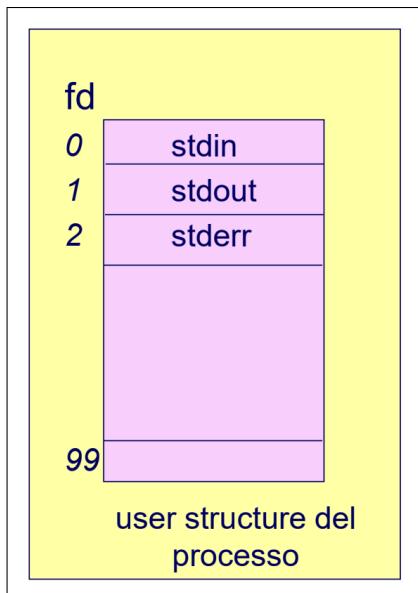
10.6 Gestione file system: concetti generali

L'accesso al singolo file è un accesso di tipo sequenziale. Vi è un'assenza della strutturazione, il file è una semplice sequenza di byte. Il file termina con un carattere speciale con un carattere speciale detto EOF. Abbiamo anche un I/O pointer che punta al byte corrente.



Vi sono varie modalità di accesso (lettura, scrittura, lettura/scrittura, ...). L'accesso al file è subordinato all'apertura.

10.7 File descriptor



All'interno del file system per ogni processo è associato una tabella dei file aperti. Essa elenca tutti i file aperti da quello specifico processo. Ogni file aperto è identificato da un indice intero detto file descriptor.

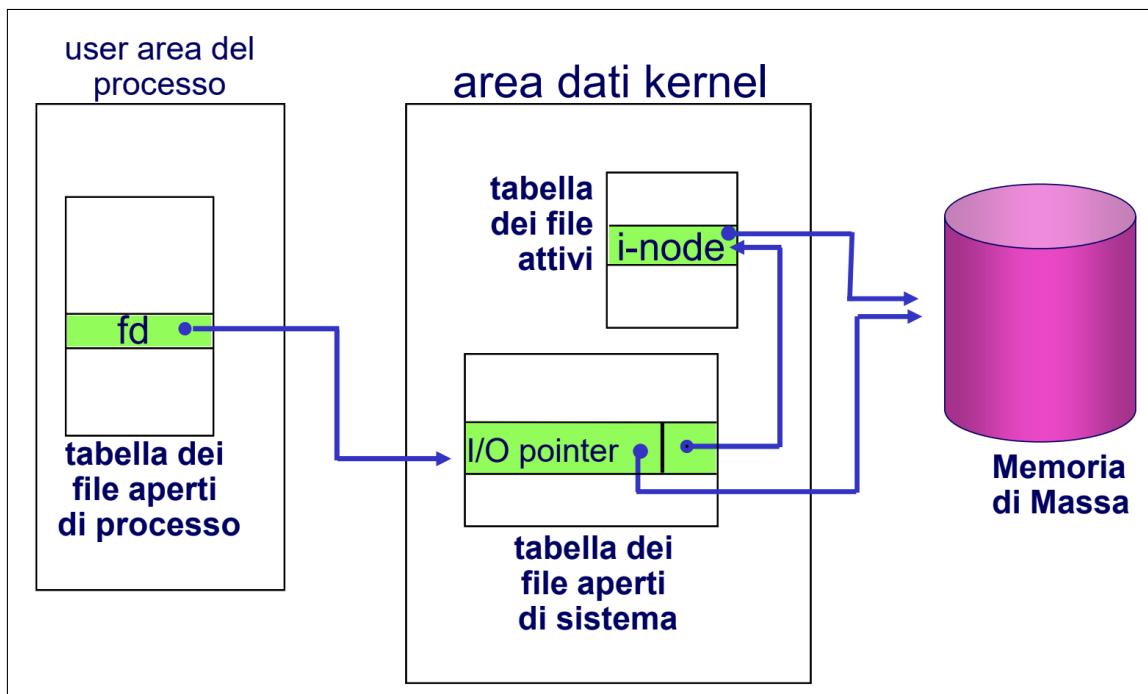
Ogni processo creato nasce di default con i primi tre file descriptor: 0, 1, 2. Essi individuano: lo standard input per lo 0; lo standard output per l'1; lo standard error per il 2. La tabella dei file aperti è localizzata nella user structure.

10.8 Strutture dati del kernel

Per realizzare l'accesso ai file, SO utilizza due strutture dati globali, allocate nell'area dati del kernel.

Tabella dei file attivi: per ogni file aperto contiene una copia del suo i-node. Serve per rendere più efficienti le operazioni su file evitando accessi al disco per ottenere attributi dei file acceduti.

tabella dei file aperti di sistema: ha un elemento per ogni operazione di apertura relativa ai file aperti e non ancora chiusi. Ogni elemento contiene: l'I/O pointer, posizione corrente all'interno del file; un puntatore all'i-node del file nella tabella dei file attivi.



Più processi possono accedere contemporaneamente allo stesso file, ma hanno I/O pointer distinti.

10.9 Gestione file system UNIX: system call

UNIX permette ai processi di accedere a file, mediante un insieme di system call, tra le quali:

- `open()`
- `close()`
- `read()`
- `write()`
- `link()`
- `unlink()`
- `lseek()`

10.10 Apertura di un file

L'apertura di un file provoca:

1. L'inserimento di un elemento (individuato da un file descriptor) nella prima posizione libera della tabella dei file aperti del processo
2. L'inserimento di un nuovo record nella tabella dei file aperti di sistema
3. La copia dell'i-node nella tabella dei file attivi (solo se il file non è già in uso).

Capitolo 11

Scheduling della CPU

Ci troviamo in un contesto di multi-programmazione, in cui più processi, anche di più utenti corrono per l'utilizzo dell'CPU. L'obiettivo della multi-programmazione è massimizzare l'utilizzo della CPU.

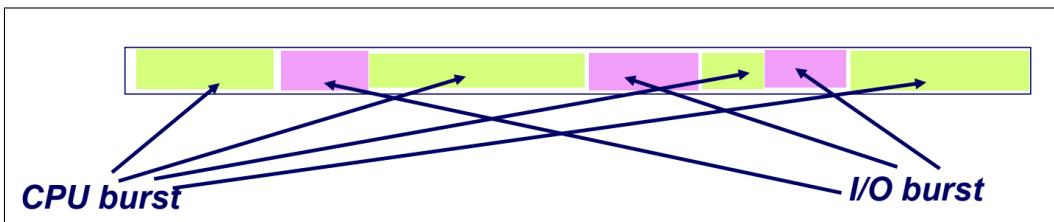
Lo **Scheduling della CPU** è quel meccanismo che permette di commutare l'uso della CPU tra i diversi processi.

Lo **Scheduler della CPU a breve termine** è quella parte di SO che fra tutti i processi pronti seleziona quello che dovrà essere eseguito, attraverso algoritmi di scheduling.

Ogni volta che avviene un cambio di processo abbiamo un **context switching**.

11.1 CPU burst & I/O burst

- **CPU Burst:** fase in cui viene utilizzata soltanto la CPU, senza interruzioni di tipo I/O.
- **I/O Burst:** fase in cui il processo effettua I/O da/verso una risorsa del sistema



Quando un processo è in I/O burst, la CPU non viene utilizzata: in un sistema multiprogrammato, short-term scheduler assegna la CPU a un nuovo processo.

A seconda delle caratteristiche dei programmi eseguiti dai processi, è possibile classificare i processi in:

- **I/O Bound:** prevalenza di attività di I/O. Vi sono molti CPU burst, ma sono di breve durata, mentre gli I/O burst sono di lunga durata.
- **CPU Bound:** prevalenza di attività di computazione. I CPU burst sono di durata molto lunga, mentre gli I/O sono pochi e di breve durata.

Gli algoritmi di scheduling si possono classificare in due categorie:

- **Senza prelazione (non pre-emptive):** la CPU rimane allocata al processo running finché esso non si sospende volontariamente o non termina, perché lo scheduler non è in grado di togliere la CPU al processo.
- **Con prelazione (pre-emptive):** lo scheduler decide il processo successivo a cui assegnare la CPU e forza il processo in esecuzione a rilasciare la CPU, anche se il processo è in fase di running.

I sistemi a divisione di tempo hanno sempre uno scheduling pre-emptive.

Lo **scheduler** decide a quale processo assegnare la CPU. Il **dispatcher** è la parte di SO che realizza il cambio di contesto.

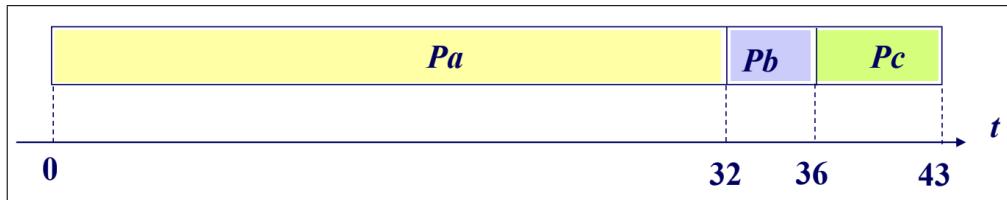
11.2 Criteri di scheduling

Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni indicatori di performance:

- **Utilizzo della CPU**: percentuale media di utilizzo CPU nell'unità di tempo.
- **Throughput**(del sistema): numero di processi completati nell'unità di tempo.
- **Tempo di Attesa**(di un processo): tempo totale trascorso nella ready queue.
- **Turnaround**(di un processo): tempo tra la sottomissione del job e il suo completamento.
- **Tempo di Risposta**(di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta (a differenza del turnaround, non dipende dalla velocità dei dispositivi di I/O).

11.3 Algoritmo di scheduling FCFS

First-Come-First-Served: la coda dei processi pronti viene gestita in modo FIFO (coda). Viene assegnato come processo a cui assegnare la CPU il processo che per primo entra nella coda dei processi pronti. L'algoritmo è non pre-emptive.



11.3.1 Problema dell'algoritmo FCFS

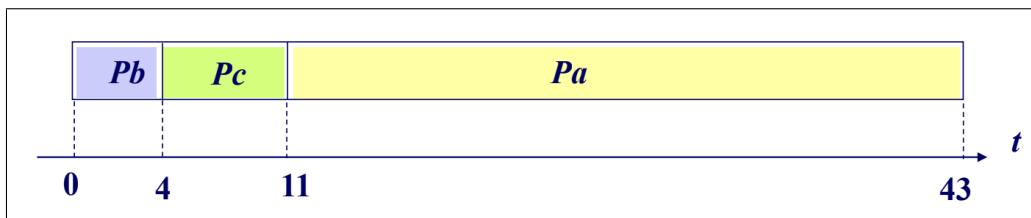
Il problema di questo algoritmo è che non è possibile influire sull'ordine dei processi.

Nel caso di processi in attesa dietro a processi con lunghi CPU burst (processi CPU bound), il tempo di attesa è alto.

Possibilità dell'effetto convoglio: se molti processi I/O bound seguono un processo CPU bound abbiamo uno scarso grado di utilizzo della CPU.

11.4 Algoritmo di scheduling SJF

Shortest Job First: si introduce questo algoritmo per risolvere i problemi del FCFS. Ha l'obiettivo di selezionare il processo con il CPU burst più corto nella coda dei processi pronti.



SJF può essere:

- Non pre-emptive: si aspetta che un processo termini o si blocchi per mettere in esecuzione il processo successivo.
- pre-emptive: in questo caso si parla di **Shortest Remaining Time First** (SRTF). Se nella coda dei processi pronti arriva un nuovo processo (Q) con CPU burst minore del CPU burst del processo in esecuzione si ha pre-emption.

11.4.1 Problema dell'algoritmo SJF

Il problema di questo algoritmo è che è difficile stimare la lunghezza del prossimo CPU burst di un processo.

Per risolvere questo problema si stima la lunghezza del CPU burst futuro in relazione a quello che è stata la sua lunghezza di CPU burst nel passato.

Una tecnica che si utilizza per prevedere il CPU burst futuro è l'**exponential averaging**.

11.5 Algoritmo di scheduling Round Robin

È tipicamente usato in sistemi time sharing. La coda dei processi pronti è gestita come una coda FIFO circolare (FCFS). Ad ogni processo viene allocata la CPU per un intervallo di tempo costante Δt (time slice o quanto di tempo). Il processo utilizza la CPU per Δt (oppure si blocca prima). Allo scadere del quanto di tempo avviene la prelazione della CPU e re-inserimento in coda

<u>Process</u>	<u>Burst Time</u>										
P_1	53										
P_2	17	P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	
P_3	68	0	20	37	57	77	97	117	121	134	154
P_4	24										162

Il Round Robin può essere visto come un'estensione di FCFS con pre-emption periodica.

L'obiettivo dell'algoritmo Round Robin è quello di minimizzare il tempo di risposta (adeguato per sistemi interattivi). Tutti i processi sono trattati allo stesso modo (assenza di starvation)

11.5.1 Problema dell'algoritmo Round Robin

Problema del dimensionamento del quanto di tempo:

- Δt piccolo (ma non troppo piccolo: $\Delta t \gg t_{\text{context switch}}$): tempi di risposta ridotti e alta frequenza di context switch. Passeremo più tempo ad effettuare context switch rispetto al tempo impiegato per eseguire un processo.
- Δt grande: overhead di context switch ridotto, ma tempi di risposta più alti.

Un altro aspetto da sottolineare è che non sempre i processi devono essere trattati in modo equo. In particolare se tutti i processi avessero la stessa importanza, allora i processi del SO avrebbero la stessa importanza dei processi utente.

11.6 Scheduling con priorità

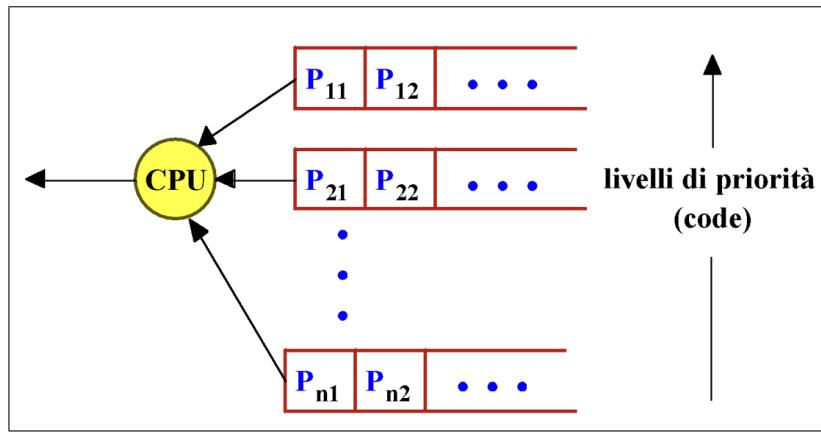
Ad ogni processo viene assegnata una priorità. Lo scheduler seleziona il processo pronto con priorità massima. I processi con uguale priorità vengono trattati in modo FCFS.

Le priorità possono essere definite:

- **Internamente:** il SO attribuisce ad ogni processo una qualche priorità in base a delle politiche interne.
- **Esteriormente:** priorità identificata esternamente al SO (es: comandi).

Le priorità possono essere costanti o variare dinamicamente.

In ogni istante è in esecuzione il processo pronto a priorità massima (algoritmi pre-emptive e non pre-emptive).



11.6.1 Problema degli algoritmi di scheduling con priorità

Il problema è che vi è la **starvation** dei processi. La starvation si verifica quando uno o più processi di priorità bassa vengono lasciati indefinitamente nella coda dei processi pronti, perché vi è sempre almeno un processo pronto di priorità più alta.

La soluzione è implementare l'invecchiamento (aging) dei processi. Ad esempio: la priorità cresce dinamicamente con il tempo di attesa del processo; mentre decresce al crescere del tempo di CPU già utilizzato.

11.7 Approcci misti

Nei SO reali, spesso si combinano diversi algoritmi di scheduling. Ad esempio possiamo avere l'algoritmo di scheduling: **Multiple Level Feedback Queues**. In questo algoritmo abbiamo:

- Più code, ognuna associata a un tipo di job diverso (batch, interactive, CPU-bound, ...).
- Ogni coda ha una diversa priorità: scheduling delle code con priorità.
- Ogni coda viene gestita con scheduling FCFS o Round Robin.
- I processi possono muoversi da una coda all'altra, in base alla loro storia:
 - Passaggio da priorità bassa ad alta: processi in attesa da molto tempo (feedback positivo).
 - Passaggio da priorità alta a bassa: processi che hanno già utilizzato molto tempo di CPU (feedback negativo).

11.8 Scheduling in UNIX

Obiettivo: privilegiare i processi interattivi. Si utilizza l'algoritmo **Multiple Level Feedback Queues**.

Abbiamo più livelli di priorità (160): più grande è il valore e più bassa è la priorità. Viene definito un valore di riferimento p_0 dove:

- Priorità $\geq p_0$: processi utente ordinari.
- Priorità $< p_0$: processi di sistema (ad es. esecuzione di system call), non possono essere interrotti da segnali (kill).

Ad ogni livello è associata una coda, gestita Round Robin.

L'aggiornamento delle priorità avviene in modo dinamico ogni secondo. La priorità di un processo decresce al crescere del tempo di CPU già utilizzato.

11.9 Linux scheduling (da v2.5)

Abbiamo due algoritmi: time-sharing e real-time:

- **Time-sharing:** le priorità sono dinamiche, basate su *crediti* - i processi con più crediti vengono schedulati prima. Quando un processo è in esecuzione scatta un *timer*, e man mano che passa il tempo i crediti vengono decrementati fino ad arrivare allo 0, a questo punto il processo viene deschedulato (pre-emptive). Si rialza il credito di tutti quando tutti i processi arrivano a credito 0. La conseguenza immediata è che si può avere starvation.
- **Real-time:** le priorità sono statiche. Le priorità corrispondono a code diverse, all'interno dello stesso livello di priorità i processi possono essere gestiti tramite: FCFS o RR. Il processo con priorità maggiore viene sempre eseguito per primo.

11.10 Scheduling dei thread Java

La Java Virtual Machine (JVM) usa scheduling con prelazione e basato su priorità; se i processi hanno la stessa priorità allora: FCFS.

La JVM mette in stato di running un thread quando:

1. Il thread che sta usando la CPU esce dallo stato runnable.
2. Se un thread ha una priorità più alta del thread in stato runnable.

Capitolo 12

Gestione della memoria

Per far sì che vi sia multi-programmazione bisogna mantenere più processi in memoria centrale: il SO deve gestire la memoria in modo da consentire la presenza contemporanea di più processi.

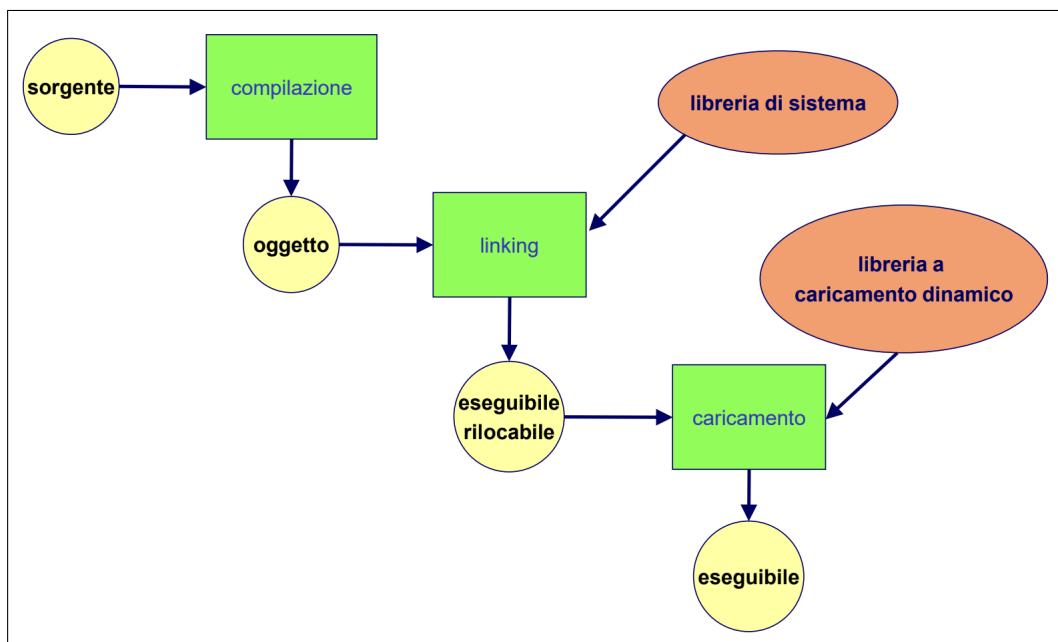
12.1 Gestione della memoria centrale

A livello hardware, ogni sistema è equipaggiato con un unico spazio di memoria accessibile direttamente da CPU e dispositivi.

I compiti dell'SO sono:

- Allocare memoria ai processi.
- Deallocare memoria.
- Separare gli spazi di indirizzi associati ai processi (protezione).
- Realizzare i collegamenti (binding) tra gli indirizzi logici specificati dai processi e le corrispondenti locazioni nella memoria fisica.
- Gestire spazi di indirizzi logici di dimensioni superiori allo spazio fisico → **memoria virtuale**.

12.2 Fasi di sviluppo di un programma



12.3 Accesso alla memoria

La *memoria centrale* è vista come un vettore di celle, identificata in modo univoco da un indirizzo. Le operazioni che si possono effettuare sulla memoria sono semplici: *load/store* di dati o istruzioni. Dal punto di vista degli indirizzi abbiamo 3 tipologie di essi:

1. **Simbolici**: riferimenti a celle di memoria nei programmi in forma sorgente mediante nomi simbolici (variabili).
2. **Logici**: riferimenti a celle nello spazio logico di indirizzamento del processo.
3. **Fisici**: riferimenti assoluti delle celle in memoria a livello hardware.

12.4 Binding degli indirizzi

Ad ogni indirizzo logico/simbolico viene fatto corrispondere un indirizzo fisico: l'associazione tra indirizzi relativi e indirizzi assoluti viene detta **binding**. Il binding può essere effettuato:

- **Staticamente**:
 - A tempo di compilazione: il compilatore genera degli indirizzi assoluti.
 - A tempo di caricamento: il compilatore genera degli indirizzi relativi che vengono convertiti in indirizzi assoluti dal loader.
- **Dinamicamente**: a tempo di esecuzione un processo può essere spostato da un'area all'altra.

12.5 Caricamento/collegamento dinamico

L'obiettivo è l'ottimizzazione della gestione della memoria.

In alcuni casi è possibile caricare in memoria una funzione/procedura a runtime solo quando avviene la chiamata.

Si carica e si collega dinamicamente la funzione al programma che la usa → **loader di collegamento rilocabile**.

Una funzione può essere usata da più processi simultaneamente.

12.6 Overlay

In generale, la memoria disponibile può non essere sufficiente ad accogliere codice e dati di un processo. Per risolvere questo problema si può utilizzare il meccanismo di **overlay**, che immagazina le istruzioni e i dati più frequentemente utilizzati, e quelli che sono necessari nella fase corrente.

12.7 Tecniche di allocazione memoria centrale

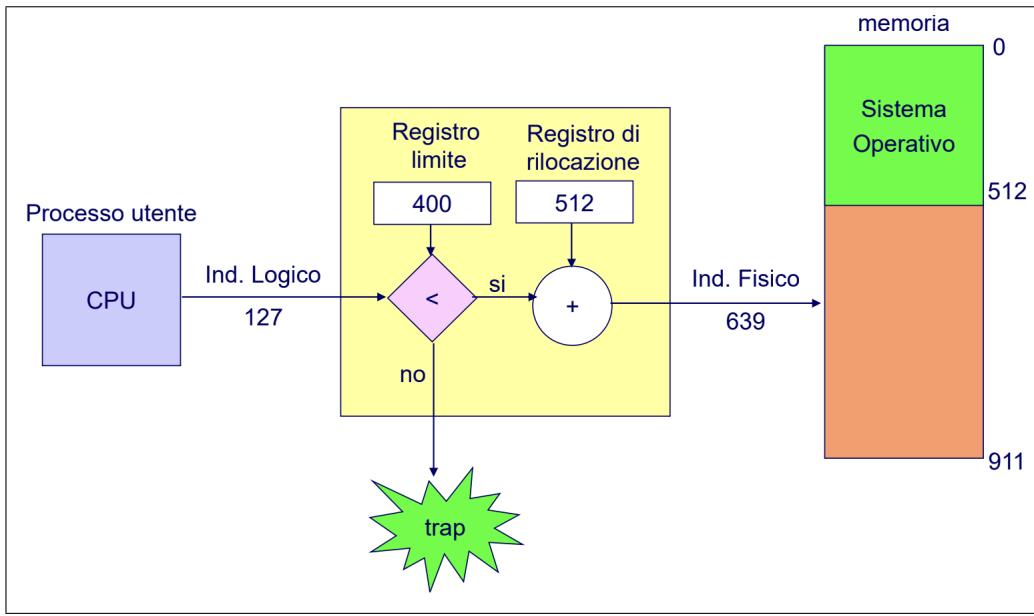
Vi sono varie tecniche per allocare codice e dati dei processi in memoria centrale:

- **Allocazione contigua**: a partizione singola o a partizioni multiple.
- **Allocazione non contigua**: paginazione o segmentazione.

12.8 Allocazione contigua a partizione singola

la parte di memoria disponibile per l'allocazione dei processi di utente non è partizionata: un solo processo alla volta può essere allocato in memoria → non c'è multiprogrammazione.

In questa tecnica il SO risiede nella memoria bassa [0, max]. Il SO deve proteggere codice e dati di SO da accessi di processi utente.



12.9 Allocazione contigua a partizioni multiple

In questo caso abbiamo multi-programmazione perché oltre all'area di memoria associata al SO il resto della memoria centrale può essere partizionato, e ognuna di queste partizioni può essere associata a un processo distinto.

In questo caso dobbiamo proteggere il codice e i dati di un processo dagli altri processi.
Possiamo avere due tipi di partizioni:

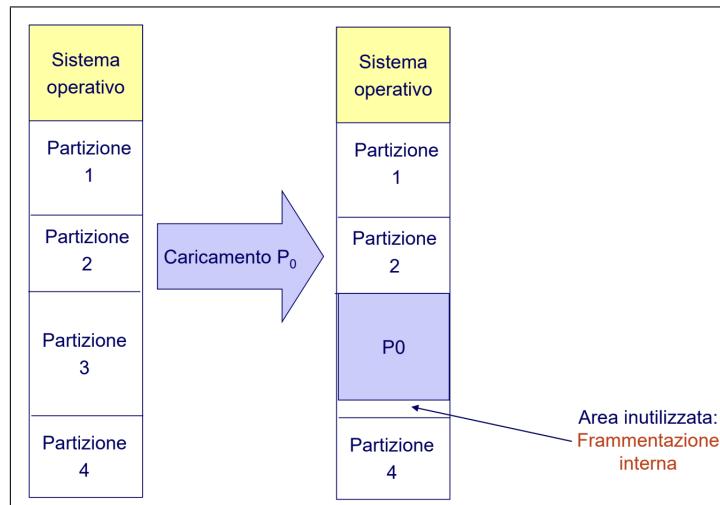
- Partizioni singole.
- Partizioni variabili.

12.9.1 Partizioni fisse

Ogni partizione ha dimensione prefissata, eventualmente diversa da partizione a partizione. Quando un processo viene schedulato, il SO cerca una partizione libera di dimensione sufficiente.

I problemi delle partizioni fisse sono:

- **Frammentazione interna:** abbiamo un sotto-utilizzo della partizione, cioè potrebbe essere allocato più spazio al processo schedulato di quanto c'è ne sia effettivamente bisogno.
- **Grado di multiprogrammazione limitato** al numero di partizioni.
- **Dimensione massima** dello spazio di indirizzamento di un processo, limitata dalla dimensione della partizione più estesa.



12.9.2 Partizioni variabili

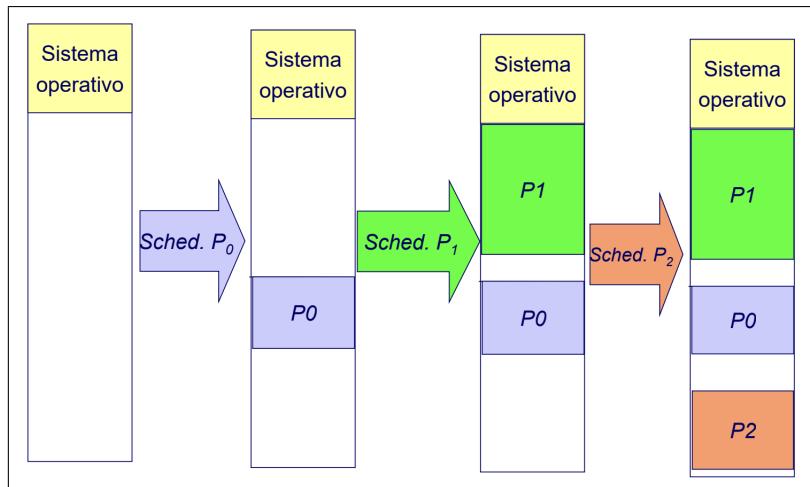
Ogni partizione viene allocata dinamicamente e dimensionata in base a dim processo da allocare. Quando un processo viene schedulato, il SO cerca un'area sufficientemente grande per allocarvi dinamicamente la partizione associata.

Vantaggi rispetto alle partizioni fisse:

- Eliminazione della frammentazione interna (ogni partizione è della esatta dimensione del processo).
- Grado di multiprogrammazione più elevato rispetto alle partizioni fisse.
- La dimensione massima dello spazio di indirizzamento di ogni processo è limitata dalla dimensione dello spazio fisico.

Problemi delle partizioni variabili:

- Bisogna scegliere bene l'area da allocare al singolo processo. Ci possono essere differenti meccanismi (best fit, worst fit, first fit...).
- **Frammentazione esterna:** man mano che si allocano nuove partizioni, la memoria libera è sempre più frammentata. Il SO deve, ogni tanto, compattare la memoria.

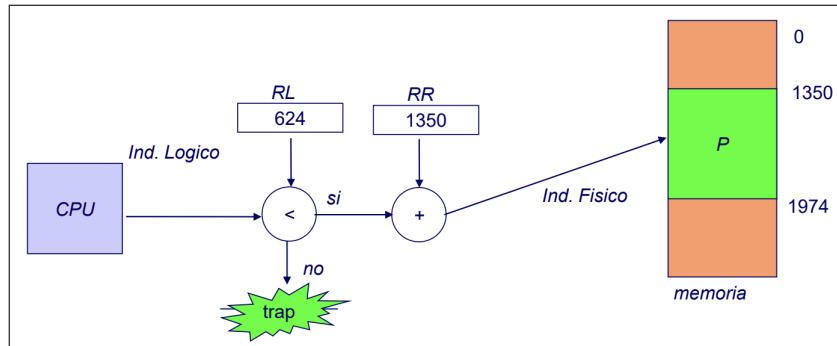


12.10 Partizioni & protezione

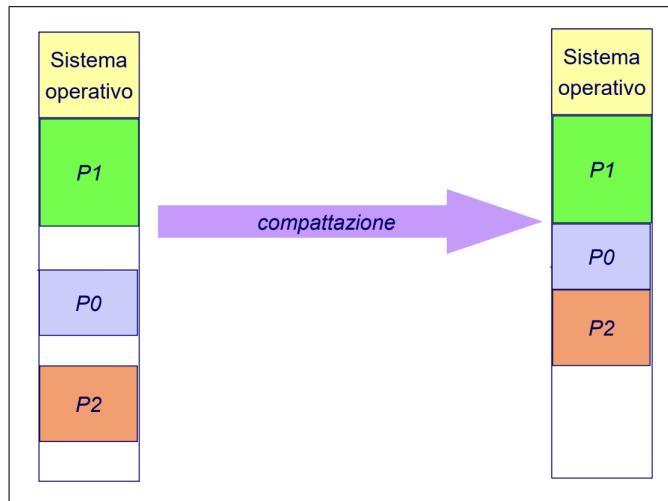
La protezione è realizzata a livello hardware mediante due registri:

- Registro di riallocazione - RR.
- Registri limite - RL.

Ad ogni processo è associata una coppia di valori $\langle V_{RR}, V_{RL} \rangle$. Quando un processo P viene schedulato, il dispatcher carica RR e RL con i valori associati al processo $\langle V_{RR}, V_{RL} \rangle$



12.11 Compattazione



Bisogna però considerare la crescita dinamica di un processo, quindi lasciare dello spazio di memoria fra un processo e l'altro può essere utile per questo aspetto.

12.12 Paginazione

Il problema principale dell'allocazione contigua a partizioni multiple è la frammentazione esterna. La soluzione a questo problema è utilizzare un'allocazione non contigua, detta **paginazione**.

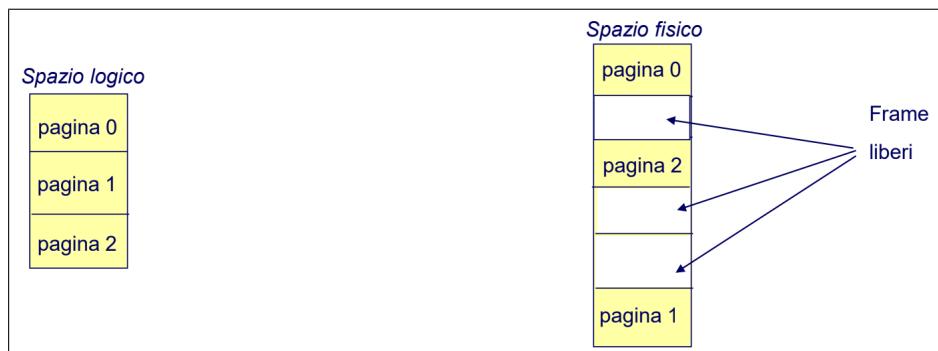
In questo modo abbiamo:

- Eliminazione della frammentazione esterna.
- Riduzione della frammentazione interna.

L'idea di base è la memoria viene partizionata in pagine (frame) di dimensione costante e limitata (ad es. 4KB), sulle quali mappare porzioni dei processi da allocare.

- **Spazio fisico**: insieme di frame di dimensione D_f costante prefissata.
- **Spazio logico**: insieme di pagine di dim uguale a D_f .

Ogni pagina logica di un processo caricato in memoria viene mappata su una pagina fisica in memoria centrale.



12.13 Supporto hardware per la paginazione

Struttura dell'indirizzo logico composto da m bit:

- p : numero di pagina logica (dimensione: $m - n$ bit).
- d : offset della cella rispetto all'inizio della pagina (dimensione: n bit).

Per ipotesiabbiamo indirizzi logici di lunghezza m bit (n bit per offset, e $m - n$ per la pagina):

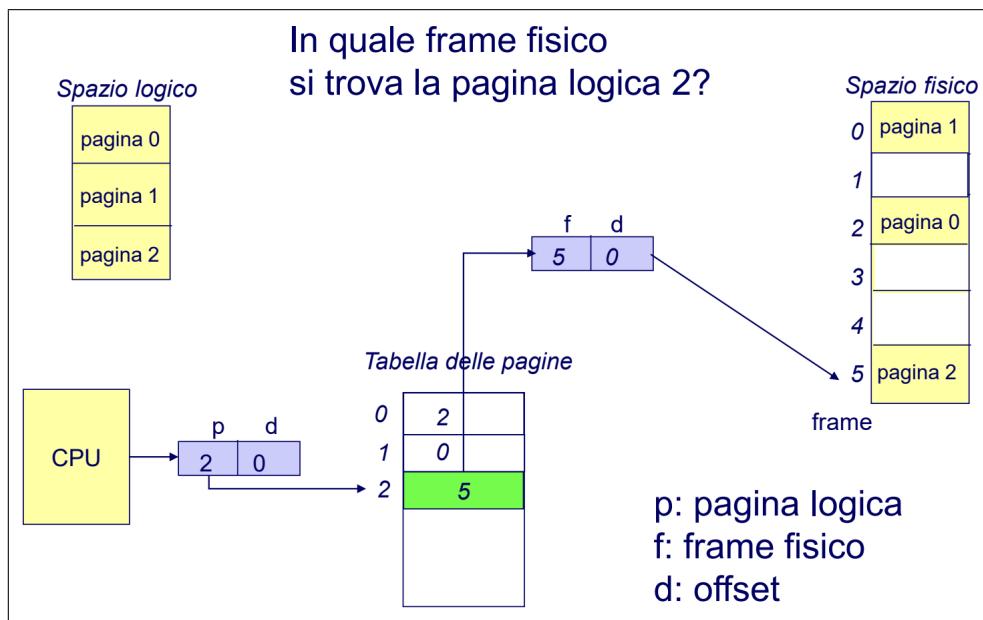
- Dimensione massima dello spazio logico di indirizzamento $\rightarrow 2^m$.
- Dimensione della pagina $\rightarrow 2^n$.
- Numero di pagine 2^{m-n} .

Struttura dell'indirizzo fisico:

- f : numero di frame (pagina fisica).
- d : offset della cella rispetto all'inizio del frame.

Il *binding* tra indirizzi logici e fisici può essere realizzato mediante **tabella delle pagine** (associata al processo).

12.13.1 Tabella delle pagine



12.14 Paginazione & Protezione

La tabella delle pagine ha dimensione fissa e non sempre viene utilizzata completamente.

Per distinguere le porzioni della tabella che non sono state utilizzate si utilizza:

- **Bit di validità**: ogni elemento della tabella delle pagine è associato un bit. Se il bit è a 1 la pagina appartiene allo spazio logico del processo, 0 altrimenti.
- **Page Table Length Register**: registro che contiene il numero di elementi validi all'interno della tabella delle pagine

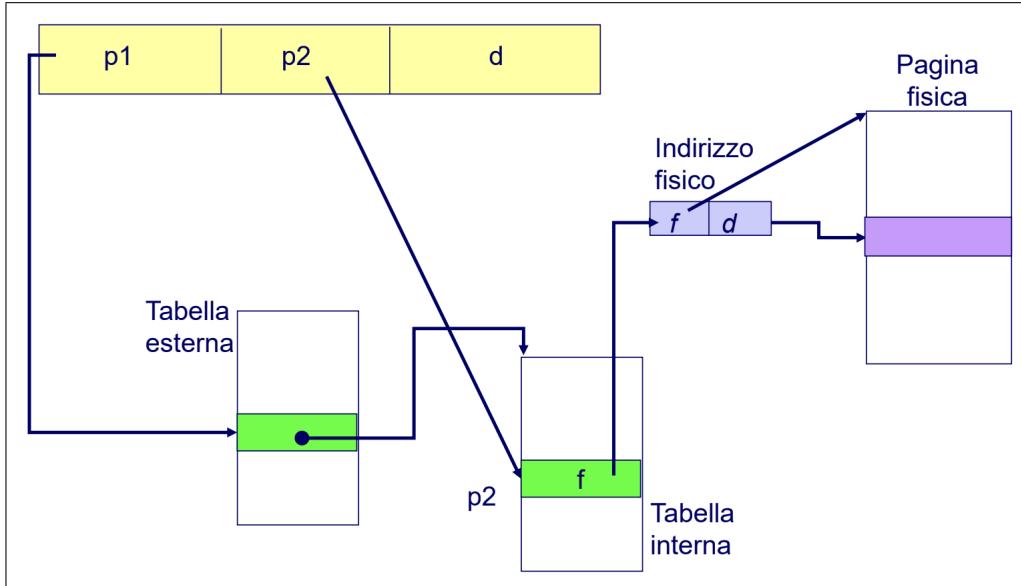
In aggiunta, per ogni entry della tabella delle pagine, possono esserci uno o più bit di protezione che esprimono le modalità di accesso alla pagina (es. read-only).

12.15 Paginazione a più livelli

Lo spazio logico di indirizzamento di un processo può essere molto esteso, quindi un numero di pagine molto grande e quindi le tabelle delle pagine diventano di grandi dimensioni.

Ipotesi: indirizzi a 32 bit \rightarrow spazio logico di 4GB e dimensione pagina 4KB (2^{12}). la tabella delle pagine dovrebbe contenere $\frac{2^{32}}{2^{12}}$ elementi $\rightarrow 2^{20}$ elementi (circa 1M).

Per rendere il tutto più efficiente si utilizza una paginazione a più livelli: allocazione non contigua anche della tabella delle pagine. Si applica ancora la tecnica di paginazione alla tabella delle pagine.



Vantaggi:

- Possibilità di indirizzare spazi logici di dimensioni elevate riducendo i problemi di allocazione delle tabelle.
- Possibilità di mantenere in memoria soltanto le pagine della tabella che servono.

Svantaggio:

- Tempo di accesso più elevato: per tradurre un indirizzo logico sono necessari più accessi in memoria.

12.16 Tabella delle pagine invertita

Per limitare l'occupazione di memoria, in alcuni SO si usa un'unica struttura dati globale che ha un elemento per ogni frame: **tabella delle pagine invertita**.

Ogni elemento della tabella delle pagine invertita rappresenta un frame (indirizzo pari alla posizione nella tabella) e, in caso di frame allocato, contiene:

- **PID**: identificatore del processo a cui è assegnato il frame.
- p : numero della pagina logica.

Quindi l'indirizzo logico contiene: il pid; il numero della pagina logica; e l'offset all'interno della pagina.

Per tradurre un indirizzo logico $<pid, p, d>$ dobbiamo ricercare nella tabella delle pagine invertite quell'elemento che contiene la coppia (pid, p) . L'indice dell'elemento trovato rappresenta il numero del frame allocato alla pagina logica p .

Problemi:

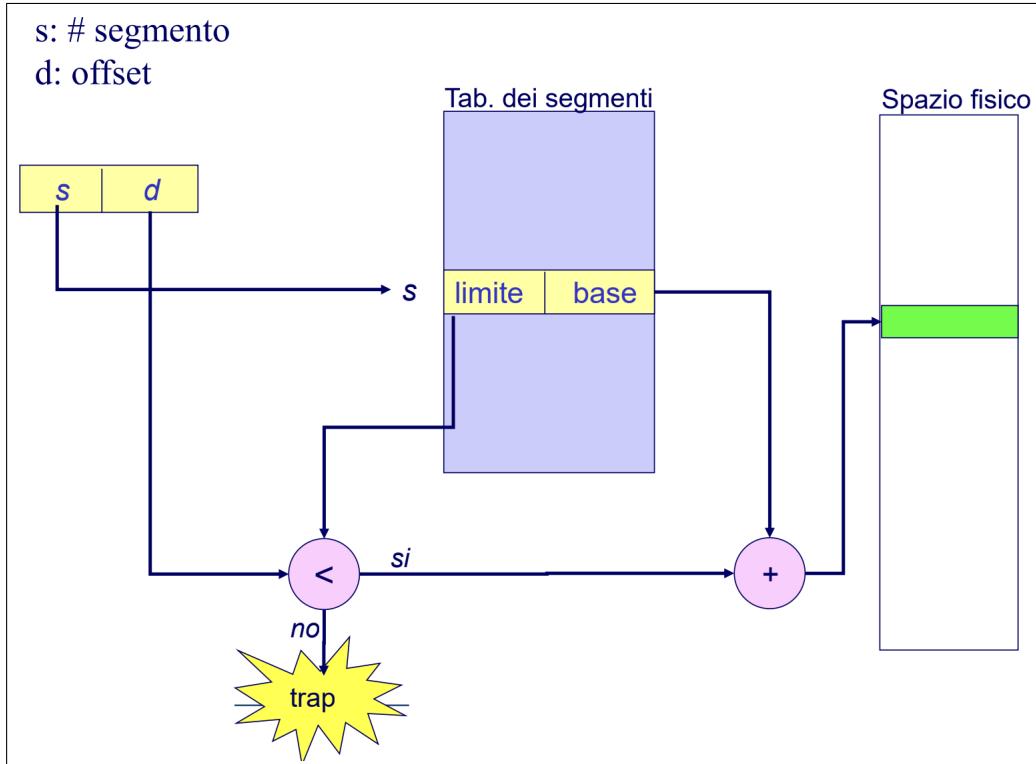
- Tempo di ricerca nella tabella invertita.
- Difficoltà di realizzazione della condivisione di codice tra processi (rientranza).

12.17 Segmentazione

La struttura degli indirizzi logici è costituita dalla coppia: $<segmento, offset>$. Un segmento è numero che individua il segmento nel sistema. L'offset è la posizione della cella all'interno del segmento.

Il supporto hardware per la segmentazione avviene tramite la **tabella dei segmenti**: ha una entry per ogni segmento che ne descrive l'allocazione in memoria fisica mediante la coppia $< \text{base}, \text{limate} >$.

- Base: indirizzo prima cella del segmento nello spazio fisico.
- Limite: indica la dimensione del segmento.



12.18 Segmentazione paginata

Per segmentazione paginata si intende che lo spazio logico viene suddiviso in segmenti e ogni segmento viene suddiviso in pagine.

Vantaggi:

- Eliminazione della frammentazione esterna (ma introduzione di frammentazione interna...).
- Non è necessario mantenere in memoria centrale l'intero segmento, ma è possibile caricare soltanto le pagine necessarie. Per esempio un processo che dovrà eseguire un programma caricherà solo il segmento del codice in memoria centrale.

Per supportare la segmentazione paginata c'è bisogno di 2 strutture dati: tabella dei segmenti, e la tabella delle pagine per ogni segmento.

12.19 Memoria virtuale

La dimensione della memoria rappresenta un limite importante per quanto riguarda il grado di multi-programmazione. In quanto la dimensione di questa va a limitare il numero di processi che potrebbero essere eseguiti in modo concorrente.

Può essere desiderabile un sistema di gestione della memoria che:

- Consenta la presenza di *più processi* in memoria (ad es. partizioni multiple, paginazione e segmentazione), indipendentemente dalla dimensione dello spazio disponibile.
- Svincoli il grado di multi-programmazione dalla dimensione effettiva della memoria

Soluzione a ciò → **memoria virtuale**.

Con le tecniche viste fin ora l'intero spazio logico di ogni processo è allocato in memoria. Oppure con l'overlay si possono allocare/deallocare parti dello spazio degli indirizzi; tutto è gestito dal programmatore.

Con l'utilizzo della memoria virtuale possiamo eseguire processi non completamente allocati in memoria principale.

Vantaggi:

- La dimensione logica totale degli indirizzi non è vincolata dalla dimensione della memoria fisica.
- Il grado di multi-programmazione non è vincolato dalla dimensione della memoria fisica.
- Il caricamento di un processo e swapping hanno un costo più limitato (meno I/O).
- Il programmatore non deve preoccuparsi dei vincoli relativi alla dimensione della memoria.

12.20 Paginazione su richiesta

Di solito la memoria virtuale è realizzata mediante tecniche di paginazione su richiesta. Tutte le pagine di ogni processo risiedono in memoria di massa; durante l'esecuzione alcune di esse vengono trasferite, all'occorrenza, in memoria centrale.

Pager: modulo del SO che realizza i trasferimenti delle pagine da/verso memoria secondaria/centrale ("swapper" di pagine).

L'esecuzione di un processo può richiedere swap-in del processo:

- **Swapper:** gestisce i trasferimenti di interi processi dalla memoria centrale alla memoria secondaria e viceversa.
- **Pager:** gestisce i trasferimenti di singole pagine.

Il pager può prevedere le pagine di cui (probabilmente) il processo avrà bisogno inizialmente.

Quindi, in generale, una pagina dello spazio logico di un processo può essere allocata in memoria centrale, oppure essere in memoria secondaria.

Per distinguere i 2 casi la tabella delle pagine ha un meccanismo chiamato **bit di validità**. Quest'ultimo ci dice se la pagina è presente in memoria centrale o in memoria secondaria, oppure se è invalida (l'indirizzo logico è al di fuori dello spazio logico del processo → **page fault**).

12.20.1 Trattamento page fault

Quando il kernel del SO riceve un'interruzione dovuta al page fault:

1. Salvataggio del contesto di esecuzione del processo.
2. Verifica del motivo del page fault (mediante una tabella interna al kernel).
3. Copia della pagina in un frame libero.
4. Aggiornamento della tabella delle pagine.
5. Ripristino del processo: esecuzione dell'istruzione interrotta dal page fault

12.20.2 Sovrallocazione

Può succedere che in memoria centrale non ci siano frame liberi ma che venga caricata una pagina all'interno di essa → sovrallocazione.

Per risolvere questo problema dobbiamo effettuare un'azione di **sostituzione**. Ovvero dobbiamo identificare un frame all'interno della memoria centrale (pagina vittima) e portare la pagina vittima dalla memoria centrale alla memoria secondaria per poter caricare la pagina di cui abbiamo bisogno.

12.21 Località dei programmi

Si è osservato che durante la sua fase di esecuzione un processo utilizza solamente un piccolo sottosinsieme delle sue pagine logiche. Inoltre questo sottosinsieme varia lentamente nel tempo.

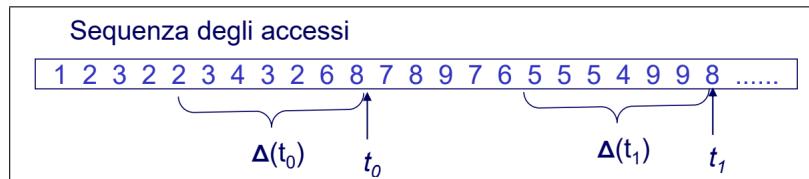
- **Località spaziale:** esiste un'alta probabilità di accedere a locazioni vicine, nello spazio logico, a locazione appena accedute (ad esempio, elementi di un vettore, codice sequenziale, ...).
- **Località temporale:** i processi tendono ad accedere alle pagine accedute precedentemente (ad esempio cicli).

12.22 Working set

In virtù di quello che è stato visto con il working set abbiamo un'alternativa alla paginazione su domanda utilizzando tecniche di gestione della memoria che si basano su pre-paginazione. Si prevede il set di pagine (→ **working set**) di cui il processo da caricare ha bisogno per la prossima fase di esecuzione.

Il working set può essere individuato in base a criteri di località temporale.

Dato un intero Δ , il working set di un processo P (nell'istante t) è l'insieme di pagine Δt indirizzate da P nei più recenti Δ riferimenti. Δ definisce la "finestra" del working set.



$$\Delta t_0 = \{2; 3; 4; 6; 8\}$$

$$\Delta t_1 = \{5; 4; 9; 8\}$$

Pre-paginazione con working set:

- Il caricamento di un processo consiste nel caricamento di un working set iniziale.
- Il SO mantiene in memoria centrale tutte le pagine che fanno parte del working set aggiornandolo dinamicamente, in base al principio di località temporale:
 - All'istante t vengono mantenute le pagine usate dal processo nell'ultima finestra Δt .
 - Le altre pagine (esterne a Δt) possono essere sostituite.

Il grande vantaggio del working set è la diminuzione dei **page fault**.

Il parametro Δ caratterizza il working set, esprimendo l'estensione della finestra dei riferimenti:

- Δ piccolo: working set insufficiente a garantire località (alto numero di page fault).
- Δ grande: allocazione di pagine non necessarie.

Ad ogni istante, data la dimensione corrente del working set WSS_i di ogni processo P_i , si può individuare la **richiesta totale di frame** D .

$$D = \sum_i WSS_i$$

Se m è il numero totale di frame:

- $D < m$: può esserci spazio per allocazione nuovi processi.
- $D \geq m$: swapping di uno (o più) processi.

Capitolo 13

Protezione

- **Protezione:** Garantire che le risorse di un sistema di elaborazione siano accedute solo dai soggetti autorizzati.
- **Risorse** fisiche e logiche (fisiche: CPU, memoria, stampanti; logiche: file, semafori, ...).
- **Soggetti:** utenti, processi o procedure.

All'interno di un SO bisogna identificare metodologie, modelli e strumenti che permettano di specificare dei controlli e la loro realizzazione.

L'obbiettivo della protezione è assicurare che ogni componente di programma/processo/utente attivo in un sistema usi le risorse del sistema solo in modi consistenti con le politiche stabilite per il loro uso.

Un sistema di protezione deve essere in grado di realizzare una varietà di politiche.

13.1 Protezione e Sicurezza

- La **protezione** serve per prevenire errori o usi scorretti da parte di processi/utenti che operano nel sistema.
- La **sicurezza** serve per difendere un sistema dagli attacchi esterni.

13.2 Sicurezza

La sicurezza ha molti aspetti. La sicurezza si applica mediante meccanismi di autenticazione e autorizzazione.

L'autenticazione è la verifica dell'identità dell'utente attraverso:

- Possesso di un oggetto (es: nfc).
- Conoscenza di un segreto (es: password).
- Caratteristica personale fisiologica (es: impronta digitale).

Problema della mutua autenticazione: quando si fa il login su un server non solo il server deve essere certo che noi siamo autenticati, ma noi dobbiamo avere la certezza che il server a cui vogliamo cedere sia il server che dichiara di essere (fishing).

- **Riservatezza:** previene la lettura non autorizzata delle informazioni.
- **Integrità:** previene la modifica non autorizzata delle informazioni.
- **Disponibilità:** garantire in qualunque momento la possibilità di usare le risorse..
- **Paternità:** chi esegue un'azione non può negarne la paternità (per esempio un assegno firmato).

13.3 Protezione (e least privilege)

In un qualunque momento un processo (o un utente) può accedere solo agli oggetti per cui è autorizzato.

Nell'informatica moderna è importante rispettare il principio del “least privilege”, cioè in ogni istante un processo deve poter accedere solo a quelle risorse strettamente necessarie per compiere la sua funzione (in questo modo si limita il danno che un processo con errori può creare nel sistema).

13.4 Protezione e controllo degli accessi

Nei sistemi operativi ci sono dei componenti incaricati di verificare che i processi possano accedere alle sole risorse per cui sono autorizzati.

Si parla di Reference Monitor, come il componente del sistema operativo che media tra le richieste di accesso dei processi e le risorse. TUTTE le richieste di accesso passano dal Reference Monitor.

È importante che tutte le decisioni di accesso alle risorse siano concentrate in un unico componente.

13.5 Dominio di protezione

Definisce un insieme di risorse (oggetti) e i relativi tipi di operazione (diritti di accesso) sugli oggetti stessi che sono permesse a processi (soggetti) appartenenti a tale dominio.

Il dominio è un concetto astratto che può essere realizzato in una varietà di modi:

- Un dominio per ogni utente. L'insieme degli oggetti che l'utente può accedere dipende dall'identità dell'utente. Il cambio di dominio è legato all'identità dell'utente.
- Un dominio per ogni processo. Ogni riga descrive gli oggetti e i diritti di accesso per un processo. Il cambio di dominio corrisponde all'invio di un messaggio a un altro processo.
- Un dominio per ogni procedura. Il cambio del dominio corrisponde alla chiamata di procedura.

13.5.1 Matrice degli accessi (modello di protezione)

oggetto dominio	F ₁	F ₂	F ₃	disco	stamp.
D ₁	read		read		
D ₂				read	print
D ₃		read	execute		
D ₄	read write		read write		

diritto di
accesso

Il problema della matrice degli accessi è che le dimensioni matrice sono troppo grandi (e sparse).

Si utilizzano soluzioni meno generali ma più efficienti: access control list, capability.

13.6 Access Control List (ACL)

Per ogni oggetto viene indicata la coppia ordinata < dominio, insieme dei diritti > limitatamente ai domini con un insieme di diritti non vuoto.

Quando deve essere eseguita un'operazione M su un oggetto O_j nel dominio D_i , si cerca nella lista degli accessi $\langle D_i, R_k \rangle$ con $M \in R_k$.

Se non esiste, si cerca in una lista di “default”; se non esiste, si ha condizione di errore.

13.7 Capability List

Per ogni dominio viene indicato l'insieme degli oggetti e dei relativi diritti di accesso.

Spesso un oggetto è identificato dal suo nome fisico o dal suo indirizzo (capability). Il possesso della capability corrisponde all'autorizzazione a eseguire una certa operazione.

Quando un processo opera in un dominio, chiede di esercitare un diritto di accesso su un oggetto. Se ciò è consentito, il processo entra in possesso di una capability per l'oggetto e può eseguire l'operazione.