ANNO ACCADEMICO 2024/2025

Sistemi Operativi

Teoria

Dionesalvi's Notes

UNIVERSITÀ
DI TORINO

DIPARTIMENTO DI INFORMATICA

1 Introduzione

1.1 Prima Lezione

Un Sistema Operativo (SO) agisce come intermediario tra l'utente e l'hardware, fornendo gli strumenti per un uso corretto delle risorse della macchina (CPU, memoria, periferiche). Ha due obiettivi principali:

- Dal punto di vista dell'utente: rendere il sistema facile da usare.
- Dal punto di vista della macchina: ottimizzare l'uso delle risorse in modo sicuro ed efficiente.

1.1.1 Architetture Single/Multi-Core

Negli anni 2000 si è passati da processori single-core a multi-core, con CPU dotate di più core in grado di eseguire istruzioni di programmi diversi simultaneamente.

1.1.2 Tipi di Eventi

- **Interrupt**: Eventi di natura hardware, rappresentati da segnali elettrici inviati da componenti del sistema.
- **Eccezioni**: Eventi di natura software, causati dal programma in esecuzione. Le eccezioni si dividono in:
  - *Trap*: Causate da malfunzionamenti del programma (es. accesso a memoria non autorizzato, divisione per 0).
  - *System Call*: Richiesta di servizi al SO, come l'accesso ai file.

1.1.3 Gestione degli Eventi

Quando si verifica un evento:

1. **Salvataggio dello stato della CPU**: Il Program Counter (PC) e i registri della CPU vengono salvati in appositi registri speciali per poter riprendere l'esecuzione successivamente.
2. **Esecuzione del codice del SO**: Il **PC** viene aggiornato con l'indirizzo del codice del **SO** che gestisce l'evento, memorizzato in una tabella detta *vettore delle interruzioni*. Questo vettore contiene puntatori a differenti routine di gestione eventi.
3. **Return**: Una volta gestito l'evento, il SO ripristina lo stato precedente e l'esecuzione del programma sospeso riprende.

Note:-

Nel Program Counter viene scritto l'indirizzo in RAM della porzione di codice del So che serve a gestire l'evento che si è appena verificato. All'accensione del computer, il SO stesso carica in aree della RAM che il SO riserva a se stesso le varie porzione di codice eseguibile che dovranno entrare in esecuzione quando si verifica un eccezione.

Osservazioni 1.1.1

Nei primi N indirizzi della RAM viene caricato una array di puntatori noto come **vettore delle interruzioni**. Ogni entry del vettore contiene l'indirizzo di partenza in RAM di una delle porzioni di codice del SO del punto precedente.

Quando un certo vento si verifica, il program counter viene aggiornato con il valore che è indicato nella cella di memoria collegata all'entry point dell'eccezione. L'ultima istruzione di ogni procedura di gestione di un evento sarà sempre una istruzione di "return from event" (**ra**).

1.2 Struttura della Memoria

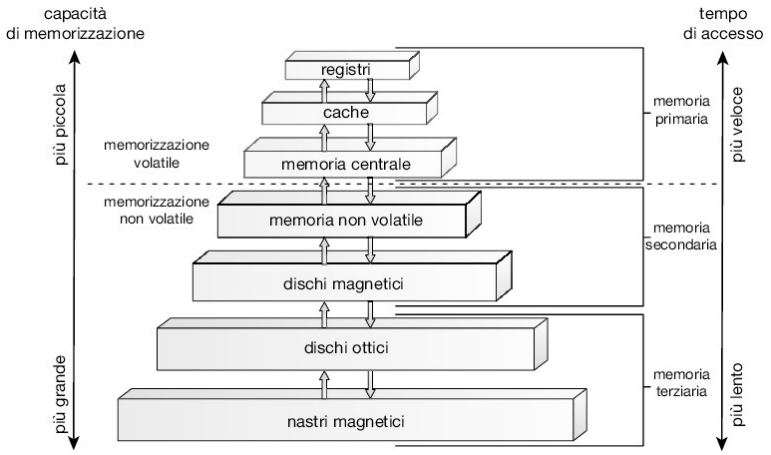
Nel contesto del **SO**, ci sono due principali tipi di memoria:

- **Memoria Principale** (*RAM*): Memoria primaria in cui risiedono programmi e dati durante l'esecuzione.
- **Memoria Secondaria**: Memoria di massa, come **hard disk** o **memorie a stato solido**, utilizzata per la conservazione permanente dei dati.

1.3 Gerarchia delle Memorie

Nella figura: *Velocità implica complessità maggiore, costo maggiore e capacità minore.*

- **Caching**: Ogni livello di memoria fa da cache per il livello successivo. Esempio: la **RAM** fa da cache per l'**hard disk**, la **CACHE** per la **RAM**, e i **registri della CPU** per la **CACHE**.



Domanda 1.1

Sarebbe bello avere 500GB di registri di CPU o Hard Disk veloci quando i registi della CPU?

Per una informazione la si deve copiare in una memoria più veloce (più costosa). La **RAM** fa da cache per l'HDD. La **CACHE** fa da cache per la RAM I **REGISTRI** fanno da cache per la CACHE.

Due tecnlogie di memoria RAM

- SRAM: per la cache e i registri della CPU
  - Creato con i FLIP-FLOP, questo porta un costo maggiore ma una maggiora efficienza rispetto alla DRAM.
- DRAM: per la memoria principale/centrale
  - Creato con i Condensatori, che tendono a perdere il loro stato. Questo obbliga a refresharli costantemente portando un dispendio maggiore di energie ma un minore costo di produzione.

1.4 Struttura di I/O

Un generico computer è composto da una CPU e da un insieme di dispositivi di I/O connessi fra loro da un bus comune. Ogni dispositivo di I/O è controllato da un apposito componente hardware detto **controller** Il controller è a sua volta un piccolo processore, con alcuni registri e una memoria interna, detto **buffer**. il SO interagisce con il controller attraverso un software apposito noto come **driver del dispositivo**.

Esempio 1.4.1 (Driver)
Il driver del dispositivo, carica nei registri dei controller opportuni valori che specificano le operazioni da compiere.
Il controller esamina i registri e intraprende l'operazione corrispondente.
Il controller trasferisce i dati dal dispositivo al proprio buffer.
Il controller invia un interrupt al SO indicando che i dati sono pronti per essere prelevati.

Questo modo di gestire l'IO con grandi quantità di dati è molto **inefficiente**. Una soluzione utile è avere un canale di comunicazione diretto tra il dispositivo e la RAm, in modo da non "disturbare" troppo il SO. Tale canale è detto **Direct Memory Access** (DMA). Il SO, tramite il driver del disco, istruisce opportunamente il controller del disco, con un comando (scritto nei registri del controller) del tipo:

Osservazioni 1.4.1
Trasferisci il blocco numero 1000 del disco in RAM a partire dalla locazione di RAM di indirizzo F2AF

Il controller trasferisce direttamente il blocco in RAM usando il DMA, e ad operazione conclusa avverte il SO mediante un interrupt opportuno.

1.5 Multitasking e Time-Sharing

Quando lanciamo un programma, il SO cerca il codice del programma sull'hard disk, lo copia in RAM, e *"fa partire il programma"* Noi utenti del SO non dobbiamo preoccuparci di sapere dov'è memorizzato il programma sull'hard disk, né dove verrà caricato in RAM per poter essere eseguito.

Dunque, il SO rende **facile** l'uso del computer. Ma il SO ha anche il compito di assicurare un uso **efficiente** delle risorse del computer, in primo luogo la CPU stessa.

Osservazioni 1.5.1
Consideriamo un programma in esecuzione: a volte deve fermarsi temporaneamente per compiere una operazione di I/O (esempio: leggere dall'hard disk dei dati da elaborare).
Fino a che l'operazione non è completata, il programma non può proseguire la computazione, e non usa la CPU.
Invece di lasciare la CPU inattiva, perché non usarla per far eseguire il codice di un altro.

Questo è il principio della multiprogrammazione (multitasking), implementato da tutti i moderni SO: il SO mantiene in memoria principale il codice e i dati di più programmi che devono essere eseguiti. (Detti anche job)

Domanda 1.2

Alcune applicazioni degli utenti però sono per loro natura interattive, come fa ad esserci una interazione continua tra il programma e l'utente che lo usa?

Oltre a questo, i sistemi di calcolo son multi-utente cioè permettono di essere connessi al sistema e di usare "contemporaneamente" il sistema stesso.

1.5.1 Time-Sharing

È meglio allora **distribuire** il tempo di CPU fra i diversi utenti (i loro programmi in "esecuzione") frequentemente (ad esempio ogni 1/10 di secondo) così da dare una impressione di **simultaneità** (che però è solo apparente).

Questo è il **time-sharing**, che estende il concetto di **multiprogrammazione**, ed è implementato in tutti i moderni sistemi operativi.

1.6 Compiti del sistema operativo

E' necessario tenere traccia di tutti i programmi **attivi** nel sistema, che stanno usando o vogliono usare la CPU, e gestire in modo appropriato il passaggio della CPU da un programma all'altro, nonché **lanciare** nuovi programmi e **gestire** la terminazione dei vecchi.

Note:-

Questo è il problema della gestione dei processi (cap. 3) e dei thread (cap. 4)

.

Quando la CPU è libera, e più programmi voglionousare, a quale programma in RAM assegnare la CPU?

Note:-

Questo è il problema di CPU Scheduling (cap. 5)

I programmi in esecuzione devono interagire fra loro senza danneggiarsi ed evitando situazioni di stallo (ad esempio, il programma A aspetta un dato da B che aspetta un dato da C che aspetta un dato da A)

Note:-

Questi sono i problemi di sincronizzazione (cap. 6/7) e di deadlock (stallo dei processi) (cap. 8)

Come gestire la RAM, in modo da poterci far stare tutti i programmi che devono essere eseguiti? Come tenere traccia di quali aree di memoria sono usate da quali programmi?

Note:-

La soluzione a questi problemi passa attraverso i concetti di gestione della memoria centrare (cap. 9) e di memoria virtuale (cap. 10).

Infine, un generico computer è spesso soprattutto un luogo dove gli utenti **memorizzano** permanentemente, organizzano e recuperano vari tipi di informazioni, all'interno di "contenitori" detti **file**, a loro volta suddivisi in cartelle (o folder, o directory) che sono organizzate in una struttura gerarchica a forma di albero (o grafo aciclico) nota come **File System**.

Note:-

Il SO deve gestire in modo efficiente e sicuro le informazioni memorizzate nella memoria di massa (o secondaria) (cap. 11) deve permettere di organizzare i propri file in modo efficiente, ossia fornire una adeguata interfaccia col file system (cap. 13), deve implementare il file system (cap. 14)

Domanda 1.3

come fa il SO a mantenere sempre il controllo della macchina?

Soprattutto, come fa anche quando non sta girando? Ad esempio, come evitare che un programma utente acceda direttamente ad un dispositivo di I/O usandolo in maniera impropria? Oppure, che succede se un programma, entra in un loop infinito? E' necessario prevedere dei modi per proteggersi dai malfunzionamenti dei programmi utente (voluti, e non)

1.6.1 Duplice modalità di funzionamento

Nei moderni processori le istruzioni macchina possono essere eseguite in due modalità diverse:

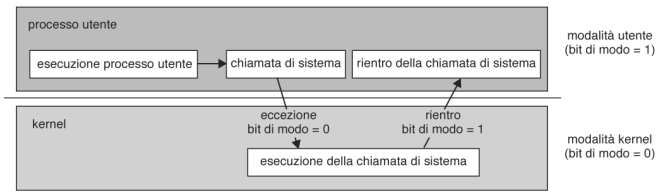
1. normale (modalità utente)
2. di sistema (modalità priviligiata, o kernel / monitor / supervisor mode)

La CPU è dotata di un "bit di modalità" di sistema (0) o utente (1), che permette di stabilire se l'istruzione corrente è in esecuzione per conto del SO o di un utente normale.

Osservazioni 1.6.1

Le istruzione macchina **sensibili**, nel senso che se usate male possono danneggiare il funzionamento del sistema nel suo complesso, possono essere eseguite solo in modalità di sistema, e quindi solo dal SO, altrimenti se nel codice di un programma normale in esecuzione è contenuta una istruzione delicata, quando questa istruzione entra nella CPU viene generata una **trap**.

I programmi utente hanno a disposizione le **system call** (chiamate di sistema) per compiere operazioni che richiedono l'esecuzione di istruzioni privilegiate.
Una system call si usa in un programma come una normale subroutine, ma in realtà provoca una **eccezione**, e il controllo passa al codice del SO di gestione di quella eccezione.
Ovviamente, quando il controllo passa al SO, il bit di modalità viene settato in modalità di **sistema** in modo automatico, via **hardware**.



Si dice di solito che il processo utente sta eseguendo in **kernel mode**

1.6.2 Timer

Domanda 1.4

Che succede se un programma utente, una volta ricevuto il controllo dalla CPU, si mette ad eseguire il seguente codice: $for(;;)i++;$?

Per evitare questo tipo di problemi, nella CPU è disponibile un **Timer**, che viene inizializzato con la quantità di tempo che si vuole concedere **consecutivamente** al programma in esecuzione.
Qualsiasi cosa faccia il programma in esecuzione, dopo 1/10 di secondo il Timer invia un **interrupt** alla CPU, e il controllo viene restituito al sistema operativo. Il SO **verifica** che tutto stia procedendo regolarmente, riinizializza il Timer e decide quale programma mandare in esecuzione, questa è l'essenza del **time-sharing**

Osservazioni 1.6.2

Ovviamente, le istruzioni macchina che gestiscono il timer, sono istruzioni privilegiate.
Altrimenti un programa utente potrebbe modifcare semplicemente i valori :D

1.6.3 Protezione della memoria

Domanda 1.5

Cosa succede se un programma in esecuzione scrive i dati di un altro programma in "esecuzione"?

E' necessario proteggere la memoria primaria da accessi ad aree riservate.

Due possibili soluzioni

Una possibile soluzione: in due registri appositi della CPU (base e limite) il SO carica gli indirizzi di inizio e fine dell'area di RAM assegnata ad un programma.
Ogni indirizzo I generato dal programma in esecuzione viene **confrontato** con i valori contenuti nei registri base e limite.

$$\text{Se } I < \text{base} \,||\, I > \text{limite} \implies \text{TRAP!}$$

I controlli vengono fatti in parallelo a livello hardware, altrimenti richiederebbero troppo tempo.

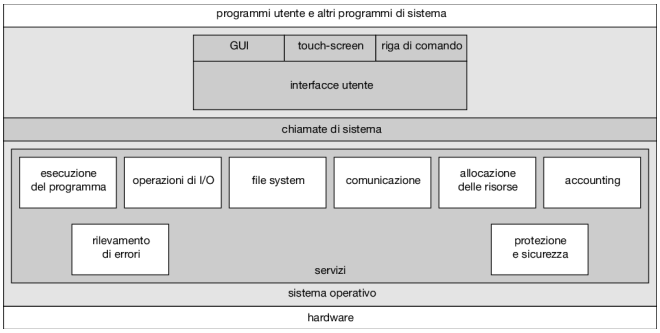
Un'altra variante: simile: in due registri appositi della CPU il SO carica rispettivamente l'indirizzo di inizio (base) e la dimensione (offset) dell'area di RAM assegnata ad un programma.

$$\text{Se } I < \text{base} \,||\, I > \text{base} + \text{offset} \implies \text{TRAP!}$$

2

Strutture dei Sistemi Operativi

Un sistema operativo mette a disposizione degli utenti (e dei loro programmi) molti servizi



Alcuni di questi servizi sono completamente invisibili agli utenti, altri sono parzialmente visibili, e altri sono direttamente usati dagli utenti. Ma il *grado di visibilità* dipende anche dal tipo di utente (Root, user, group)

Esempio 2.0.1 (Esempi di visibilità)

- Interfaccia col sistema operativo (terminale) (visibili)
- Chiamate di sistema (quasi sempre visibili)
- Gestione di processi (praticamente invisibili

2.1 Interfaccia del Sistema Operativo

L'interfaccia è lo strumento con il quale gli utenti interagiscono con il So, e ne sfruttano i servizi offerti.
Può essere un **interprete di comandi**, o un'**interfaccia grafica** con finestre e menù, ma di solito è possibile usare una combinazione di entrambi

2.1.1 Interprete dei comandi

Normalmente non fa parte del **kernel** SO; ma è un programma (o collezione di essi) fornito insieme al SO.
Un esempio d'interprete è la **shell** dell'MS-Dos oppure la **shell** Unix.
Una shell rimane semplicemente in attesa di ciò che l'utente scrive da linea di comando, ed ovviamente, esegue

il comando stesso. Spesso, i comandi che possonoe ssere usati dagli utenti del SO sono dei semplici **eseguibili**. L'interprete si occupa di trovare sull'hard disk e lanciare il codice dell'eseguibile passando eventuali argomenti specificati.

Esempio 2.1.1 (Comando shell unix)

1. L'utente scrive *rm myfile*
2. l'interprete cerca un file eseguibile di nome "rm" e lo lancia, passandogli come parameteo "myfile"

Note:-

Un comando utile può essere *ps* che ti permette di vedere i processi attaccati alla tua shell

2.2 Interfaccia grafica

I moderni SO offrono anche una interfaccia grafica (GUI) per gli utenti, spesso più facile da imparare ed usare. **Unix** offre varie interfaccie grafiche, sia proprietarie che open-source, come **KDE** e **GNOME**, e ogni utente del SO può scegliersi la sua

2.3 Programmi/servizi di Sistema

Non fanno parte del kernel del SO, ma vengono forniti insieme al SO, e rendono più facile, comodo e conveniente l'uso del Sistema.
Gli interpreti dei comandi e le interfacce grafiche sono gli esempi più evidenti di programmi di sistema.
Oltre a questi: editor, compilatori, browser, task manager etc etc.

2.4 Chiamate di sistema (Syscall)

Da ora in poi, chiameremo un programma in "esecuzione" come **processo** Le system call costituiscono la vera e propria interfaccia tra i processi degli utenti e il Sistema Operativo.
Ad esempio, in Unix assumono la forma di procedure che possono essere inserite direttamente in programmi scritti con linguaggi ad alto livello (C, C++, ...)
Sembra di usare una **subroutine**, ma l'esecuzione della system call trasferisce il controllo al SO, e in particolare alla porzione di codice del SO che implementa la particolare System Call invocata.
Ad esempio, in un programma C, per scrivere dentro ad un file:

```
fd = open("nomefile", O_WRONLY);
i = write(...)
close(fd)
```

Open, write e close sono delle syscall

2.4.1 Chiamate di sistema: le "API"

Application Programming Interface Le API non sono ltro che uno strato intermedio tra le applicazione sviluppate dai programmatori e le syscall, per rendere più **facile** l'uso e migliorare la **portabilità** tra versioni.

Esempio 2.4.1 (Chiamate)

Ad esempio, la libreria C dell'ambiente Unix è una semplice forma di API. In questa libreria esiste la funzione per aprire un file:
fopen, *fprintf* e *fclose*

2.5 Gestione dei processi

In un dato istante, all'interno di un SO sono attivi più processi (anche se uno solo è in esecuzione, in un dato istante). Si parla allora di **Processi Concorrent**, perchè si contendono l'uso delle risorse hardware della macchina.

1. La CPU
2. Lo spazio in memoria primaria e secondaria
3. I dispositivi di input e output

Il SO ha la responsabilità di fare in modo che ogni processo abbia la sua parte di risorse, senza danneggiare e venire danneggiato dal altri processi.

Il SO quindi deve gestire tutti gli aspetti riguardo la vita dei processi.

- Creazione e cancellazione dei processi
- Sospensione e riavvio dei processi
- Sincronizzazione tra i processi
- Comunicazione tra processi

Per eseguire un programma deve essere caricato in memoria principale.
In un sistema time-sharing, più processi possono essere contemporaneamnte attivi: il loro codie e i loro dati sono caricati in qualche area della RAM. Quindi il SO deve:

- Tenere traccia di quali parti della RAM sono utilizzati e da quale processo
- Distruibire la RAM tra i processi
- Gestire la RAM in base alla necessità e ai cambiamenti

2.6 Gestione dei file e del filesystem

Quasi ogni informazione presente in un sistema è contenuta in un file: una raccolta di informazioni denotata da un nome (e di solito da altre proprietà).

I file sono organizzati in una struttura **gerarchica** detta File System, mediante le cartelle (o directory, o folder) Il SO e' responsabile della: – Creazione e cancellazione
- Fornitura di strumenti per gestire i file e dir
- Memorizzazione efficiente del file system in memoria secondaria.

I file sono memorizzati permanentemente in memoria secondaria, di solito su un hard disk.
Il SO deve:
– decidere dove e come memorizzare i file su disco, ed essere in grado di ritrovarli velocemente.
– Trovare spazio libero velocemente quando un file è creato o aumenta di dimensione, e recuperare spazio alla rimozione di un file.
– Gestire efficientemente accessi concorrenti ai file dai vari processi attivi.

2.7 Macchine Virutali

Un moderno SO trasforma una macchina reale in una sorta di macchina virtuale (MV).

3
Gestione dei processi

3.1 Processi

Il processo è l'unità di lavoro del sistema operativo, perché ciò che fa un qualsiasi SO è innanzi tutto amministrare la vita dei processi che girano sul computer gestito da quel SO. Il sistema operativo è responsabile della creazione e cancellazione dei processi degli utenti, gestisce lo scheduling dei processi, fornisce dei meccanismi di sincronizzazione e comunicazione fra i processi.

3.1.1 Concetto di processo

- Un **processo** è più di un semplice programma in esecuzione, infatti, ha una struttura in memoria primaria, suddivisa in più parti assegnategli dal sistema operativo (vedi fig. 3.1).
- Le principali componenti della struttura di un processo sono:
  - **Codice** da eseguire (il "testo")
  - **Dati**
  - **Stack** (per le chiamate alle procedure/metodi e il passaggio dei parametri)
  - **Heap** (memoria dinamica)
- La somma di queste componenti forma l'immagine del processo:

$$\text{codice} + \text{dati} + \text{stack} + \text{heap} = \text{immagine del processo}$$

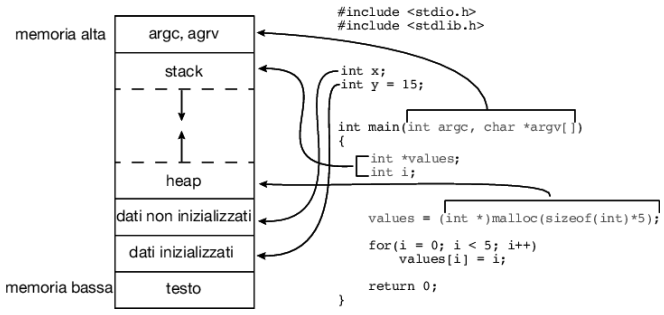


Figure 3.1: Concetto di processo

È anche corretto osservare che attraverso un programma si possono definire più processi, infatti:

- Lo stesso programma può contenere codice per generare più processi
- Più processi possono condividere lo stesso codice

Tuttavia, la distinzione fondamentale tra processo e programma è che un processo **è un'entità attiva**, mentre un programma è **un'entità statica**.

Domanda 3.1

Lo stesso programma lanciato due volte può dare origine a due processi diversi (perché?)

Attenzione: processo, task, job sono **sinonimi**.

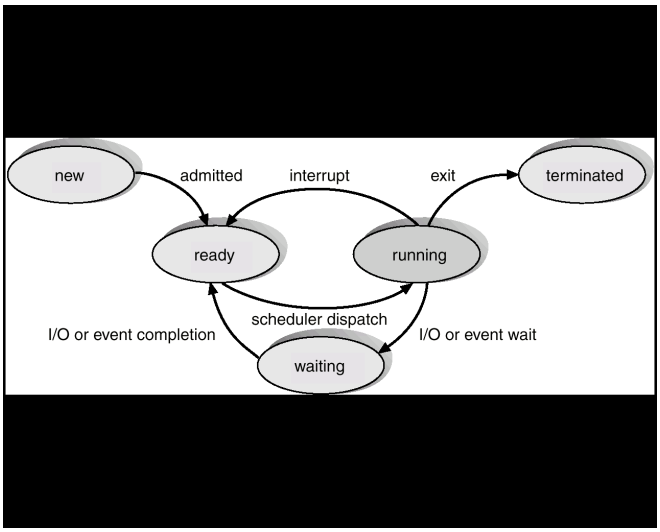
Un programma si **trasforma** in un processo quando viene lanciato, con il doppio click o da riga di comando. Un processo può anche **nascere** a partire da un altro processo, quando quest'ultimo esegue una opportuna system call (fork, spawn, etc)

Definizione 3.1.1: Processo

In realtà, non sono due meccanismi distinti: un processo nasce sempre a partire da un altro processo, e sempre sotto il controllo e con l'intervento del SO (con un'unica eccezione, all'accensione del sistema).

3.1.2 Stato del processo

Da quanto nasce a quando termina, un processo passa la sua esistenza muovendosi tra un insieme di stati, e in ogni stante ogni processo si trova in un ben determinato stato.
Lo stato di un processo evolve a causa del codice eseguito e dell'azione del SO sui processi presenti nel sistema in un dato istante, secondo quanto illustrato dal diagramma di transizione degli stati di un processo.



Gli stati

Gli stati in cui può trovarsi un processo sono:

Definizione 3.1.2: Stati del processo

- **New**: Il processo è appena stato creato
- **Ready (to Run)**: Il processo è pronto per entrare in esecuzione
- **Running**: La CPU sta eseguendo il codice del processo
- **Waiting**: Il provesso ha lasciato la CPU e attende il completamento di un evento
- **Terminated**: Il processo è terminato, il SO sta recuperando le strutture dati e le aree di memoria liberate

Il diagramma di transizione degli stati di un processo sintetizza una serie di possibili varianti del modo in cui un sistema operativo (SO) può amministrare la vita dei processi di un computer.

- Infatti, nel caso reale lo sviluppatore del SO dovrà decidere quali scelte implementative fare quando (ad esempio):
  - Mentre il processo P_x è *running*, un processo entra nello stato *Ready to Run*
  - Mentre il processo P_x è *running*, un processo più importante di P_x entra nello stato *Ready to Run*
  - Mentre il processo P_x è nello stato *Ready to Run*, un processo più importante di P_x entra nello stato *Ready to Run*

Domanda 3.2

Che significato ha eliminare l'arco "interrupt"?

Di avere un sistema non time-sharing

3.1.3 Processo Control Block (PCB)

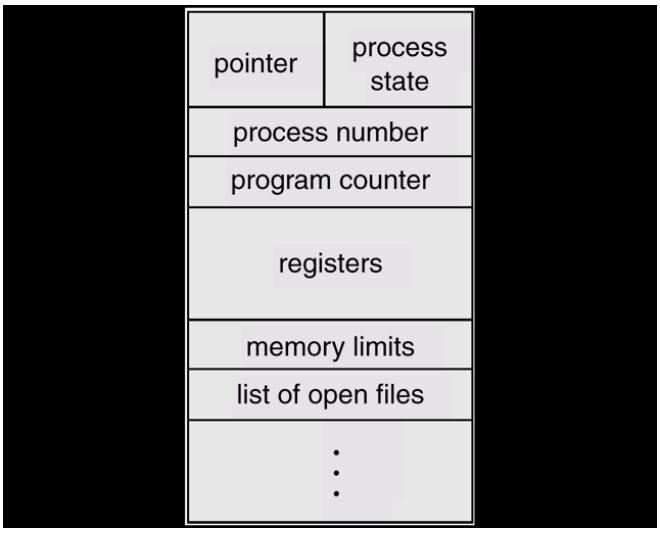
Per ogni processo, il sistema operativo (SO) mantiene una struttura dati chiamata *Process Control Block* (PCB), che contiene le informazioni necessarie per amministrare la vita di quel processo, tra cui:

- Il numero del processo (o *Process ID*)(PID)
- Lo stato del processo (*ready*, *waiting*,...)
- Il contenuto dei registri della CPU salvati nel momento in cui il processo è stato sospeso (valori significativi solo quando il processo non è *running*)
- Gli indirizzi in RAM delle aree dati e codice del processo
- I file e gli altri dispositivi di I/O correntemente in uso dal processo
- Le informazioni per lo *scheduling* della CPU (ad esempio, quanta CPU ha usato fino a quel momento il processo)

3.2 Scheduling dei processi

Conosciamo già i seguenti due concetti:

- **Multiprogrammazione**: avere sempre un processo *running* $\Rightarrow$ massima utilizzazione della CPU.
- **Time Sharing**: distribuire l'uso della CPU fra i processi a intervalli prefissati. Così più utenti possono usare "allo stesso tempo" la macchina, e i loro processi procedono in "parallelo" (notate sempre le virgolette).



Definizione 3.2.1: Scheduling

Per implementare questi due concetti, il sistema operativo deve decidere periodicamente quale sarà il prossimo processo a cui assegnare la CPU. Questa operazione è detta *Scheduling*.

In un sistema time sharing single-core, attraverso lo scheduling, ogni processo "crede" di avere a disposizione una macchina "tutta per se"... Ci pensa il SO a farglielo credere, **commutando** la CPU fra i processi (ma succede la stessa cosa in un sistema ad n-core se ci sono più di n processi attivi contemporaneamente)

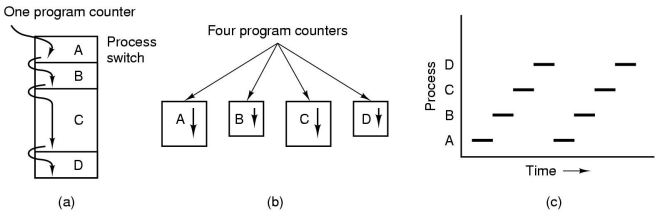


Figure 3.2: a) Ciò che succede in realtà
b) ciò che vede ogni singolo processo
c) Il risultato finale

3.2.1 Il cambio di contesto (context switch)

Per commutare la CPU tra due processi, il sistema operativo deve:

1. Riprendere il controllo della CPU (ad esempio attraverso il meccanismo del *Timer* visto nel capitolo 1).
2. Con l'aiuto dell'hardware della CPU, salvare lo stato corrente della computazione del processo che lascia la CPU, ossia copiare il valore del *Program Counter* (PC) e degli altri registri nel suo *Process Control Block* (PCB).
3. Scrivere nel PC e nei registri della CPU i valori relativi contenuti nel PCB del processo utente scelto per entrare in esecuzione.

   Questa operazione prende il nome di: **cambio di contesto**, o *context switch*.
   Notate che, tecnicamente, anche il punto 1 è già di per sé un *context switch*.

- Il *context switch* richiede tempo, perché il contesto di un processo è composto da molte informazioni (alcune le vedremo quando parleremo della gestione della memoria).

- Durante questa frazione di tempo, la CPU non è utilizzata da alcun processo utente.
- In generale, il *context switch* può costare da qualche centinaio di nanosecondi a qualche microsecondo.
- Questo tempo "sprecato" rappresenta un *overhead* (sovraccarico) per il sistema e influisce sulle sue prestazioni.

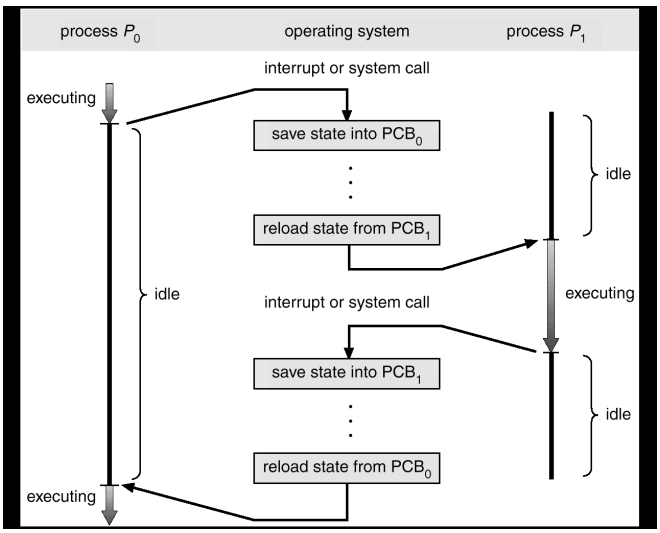
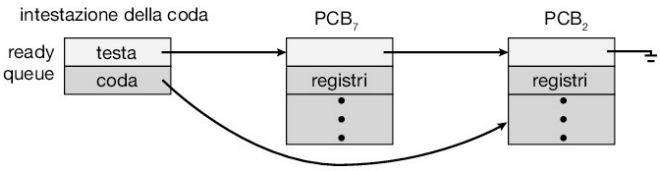


Figure 3.3: Fasi dello scheduling tra un processo e un altro

3.2.2 Code di scheduling

Per **amministrare** la vita di ciascun processo, il SO gestisce varie **code** di processi. Ogni processo "si trova" in una di queste code, a seconda di cosa sta facendo. Una coda di processi non è altro che una lista di PCB, mantenuta in una delle aree di memoria primaria che il SO riserva a se stesso.
La coda dei processi più importante è la coda **ready**, o **ready queue (RQ)**: l'insieme dei processi **ready to run**. Quando un processo rilascia la CPU, ma non termina e non torna nella *ready queue*, vuol dire che si è messo



in **attesa** di "qualcosa", e il SO lo "parcheggia" in una tra le possibili code, che possiamo dividere in due grandi categorie:

- **Device queues**: code dei processi in attesa per l'uso di un dispositivo di I/O. Una coda per ciascun dispositivo.

  Esempio 3.2.1 (Esempi)
  - Una coda d'attesa per il primo hard disk
  - Una coda per l'ssd
  - Una coda per la stampante, etc..

- **Code di waiting**: code di processi in attesa che si verifichi un certo evento. Una coda per ciascun evento (ci torneremo nella sezione 6.6).

Dunque, durante la loro vita, i processi si spostano (meglio: il SO sposta i corrispondenti PCB) tra le varie code. Quindi lo stato **waiting** nel diagramma di transizione degli stati di un processo **corrisponde a più code di attesa**

Possiamo riformulare il diagramma di transizione degli stati di un processo come un **diagramma di accodamento** in cui i processi si muovono fra le varie code

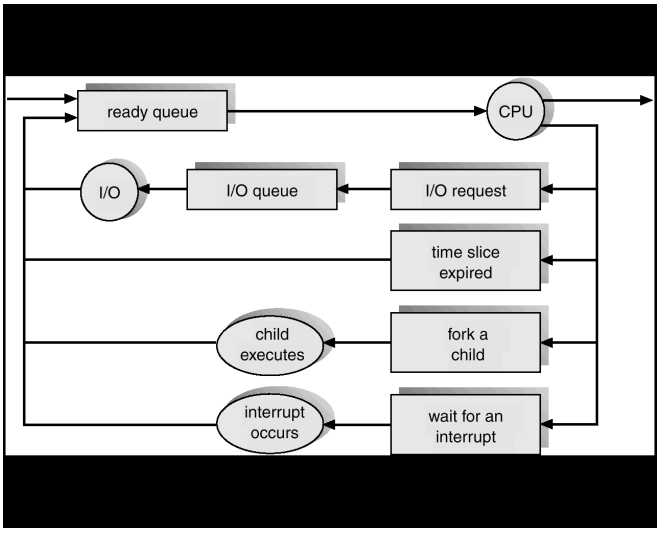


Figure 3.4: SX: new, DX: Terminated

3.2.3 CPU Scheduler

Un componente del Sistema Operativo detto *CPU Scheduler* sceglie uno dei processi nella coda *ready* e lo manda in esecuzione.

- Il *CPU scheduler* si attiva ogni 50/100 millisecondi, ed è responsabile della realizzazione del *time sharing*.
- Per limitare l'*overhead*, deve essere molto veloce.
- Il *CPU scheduler* è anche chiamato *Short Term Scheduler*.

3.3 Operazione sui processi

La creazione di un processo è di gran lunga l'operazione più importante all'interno di qualsiasi sistema operativo. Ogni SO possiede almeno una *System Call* per la creazione di processi, e ogni processo è creato a partire da un altro processo usando la system call relativa (eccetto il processo che nasce all'accensione del sistema).
Il processo "creatore" è detto *processo padre* (o *parent*).
Il processo creato è detto *processo figlio* (o *child*).

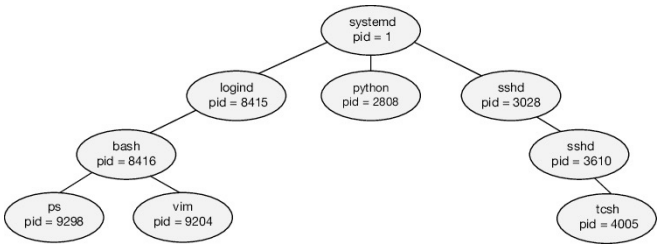
Osservazioni 3.3.1

Poiché ogni processo può a sua volta creare altri processi, nel sistema si forma un "albero di processi".

3.3.1 Creazione di un processo

Quando nasce un nuovo processo, il SO:

- gli assegna un identificatore del processo unico, un numero intero detto **pid** (process-id). È il modo con cui il SO conosce e si riferisce a quel processo.
- recupera dall'hard disk il codice da eseguire e lo carica in RAM (a meno che il codice non sia già in RAM).



- alloca un nuovo *PCB* e lo inizializza con le informazioni relative al nuovo processo.
- inserisce il *PCB* in coda *ready*.

Domanda 3.3

Che cosa fa il processo padre quando ha generato un processo figlio?

- Prosegue la sua esecuzione in modo concorrente all'esecuzione del processo figio, oppure:
- Si ferma, in attesa del completamento dell'esecuzione del processo figlio

Domanda 3.4

Quale codice esegue il processo figlio?

- al processo figlio viene data una copia del codice e dei dati in uso al processo padre, oppure:
- al processo figlio viene dato un nuovo programma, con eventualmente nuovi dati.

3.3.2 Creazione di un processo in Unix

```
int main() {
    /* fig. 3.8 modificata */
    pid_t pid, childpid;

    pid = fork(); /* genera un nuovo processo */
    printf("questa la stampano padre e figlio");

    if (pid == 0) {
        /* processo figlio */
        printf("processo figlio");
        execlp("/bin/ls", "ls", NULL);
    } else {
        /* processo padre */
        printf("sono il padre, aspetto il figlio");
        childpid = wait(NULL);
        printf("il processo figlio  terminato");
        exit(0);
    }
}
```

3.3.3 Passi dell'SO all'invocazione delle fork

1. Alloca un nuovo *PCB* per il processo figlio e gli assegna un nuovo *PID*; cerca un'area libera in RAM e vi copia le strutture dati e il codice del processo *parent* (si veda più avanti): queste copie verranno usate dal processo figlio.
2. Inizializza il *PC* del figlio con l'indirizzo della prima istruzione successiva alla *fork*.

3. **Nella cella di memoria associata alla variabile che riceve il risultato della *fork* del processo figlio scrive 0.**
4. **Nella cella di memoria associata alla variabile che riceve il risultato della *fork* del processo *parent* scrive il *PID* del figlio.**
5. Mette i processi *parent* e figlio in coda *ready*.

Osservazioni 3.3.2

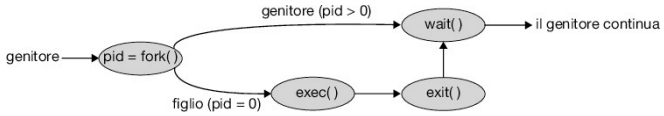
pid == 0 Lo ha solo il processo figlio.
pid = id-child lo ha solo il processo padre.
Così sono in grado di distinguere se sto operando con il figlio o con il padre.

Significato delle altre sys

Execlp: Riceve in input un puntatore ad un file contenente codice eseguibile. Il processo che la invoca prosegue eseguendo il codice specificato, senza più ritornare alla porzione di codice che viene dopo la execlp.
Wait: Invocata da un processo parent, lo sospende fino alla terminazione del processo figlio. La wait restituisce il PID del figlio appena terminato. **Exit:** provoca la terminazione istantanea del processo che la invoca.

Domanda 3.5

Come cambia lo schema se il processo parent non esegue la wait?



3.3.4 Altro esempio

```
int main() {
    /* un altro esempio */
    int a, b, c = 57;
    a = fork(); // genera un nuovo processo
    printf("questa la stampano padre e figlio");

    if (a == 0) {
        /* processo figlio */
        c = 64; // ***
        printf("c = %d", c);
    } else {
        /* processo padre */
        printf("c = %d", c);
        b = wait(NULL);
        printf("b = %d", b);
    }
}
```

3.3.5 Osservazioni

- Il codice viene condiviso tra padre e figlio, evitando duplicazione e spreco di memoria.
- Lo spazio dati viene duplicato:

  - Le modifiche di variabili non sono condivise tra padre e figlio.
  - Le nuove variabili dichiarate dopo la `fork` non sono visibili all'altro processo.

- Un padre può chiamare `fork` più volte, e usare il PID dei figli per tracciarli.
- `fork` restituisce 0 al figlio per distinguerlo dal padre.
- Se `fork` restituisse un valore maggiore di 0 al figlio, non si potrebbe distinguere facilmente tra padre e figlio, complicando la gestione delle operazioni diversificate (come illustrato in fig. 3.8).

3.3.6 Terminazione di un processo

Un processo termina dopo l'esecuzione dell'ultima istruzione del suo codice. Esiste una system call chiamata `exit()` per terminare un processo.
I dati di output, come il `pid`, possono essere inviati al processo padre in attesa della terminazione del figlio. Il sistema operativo **rimuove** le risorse allocate al processo terminato, recuperando la RAM e chiudendo eventuali file aperti.

- Un processo può uccidere esplicitamente un altro processo appartenente allo stesso utente tramite la system call `kill` (in Unix) o `TerminateProcess` (in Win32).
- In alcuni casi, il sistema operativo può decidere di terminare un processo utente, ad esempio se:
  - il processo utilizza troppe risorse.
  - il suo processo padre è morto (in questo caso può avvenire una terminazione a cascata, che non avviene però in Unix o Windows).

3.4 Comunicazione tra processi

Processi indipendenti e cooperanti
I processi attivi in un sistema possono essere classificati come:

- **Indipendenti**: quando non si influenzano esplicitamente durante l'esecuzione.
- **Cooperanti**: quando si influenzano a vicenda per:
  - Scambiarsi informazioni.
  - Collaborare su un'elaborazione suddivisa per efficienza o modularità.

I processi cooperanti necessitano di meccanismi di comunicazione e sincronizzazione.

3.5 Esempio: il problema Produttore-Consumatore

Problema del produttore-consumatore
Un classico problema di processi cooperanti è il *problema del produttore-consumatore*:

- Un **processo produttore** produce informazioni che vengono consumate da un **processo consumatore**.
- Le informazioni sono collocate in un buffer di dimensione limitata.
- Un esempio pratico è un **processo compilatore** (produttore) che genera codice assembler.
- Il **processo assemblatore** (consumatore) traduce il codice assembler in linguaggio macchina.
- L'assemblatore potrebbe poi diventare un produttore per un modulo che carica in RAM il codice.

```
#define SIZE 10

typedef struct {
    // Definizione della struttura dell'item
    ...
} item;

// Buffer condiviso
item buffer[SIZE]; (shared array)

// Variabili condivise
int in = 0, out = 0;
```

Buffer circolare di `SIZE` elementi con due puntatori `in` e `out`:

- `in`: indica la prossima posizione libera nel buffer.
- `out`: indica la prossima posizione piena da consumare.
- **Condizione di buffer vuoto**: `in == out`.
- **Condizione di buffer pieno**: `(in + 1) % SIZE == out`.

Nota: la soluzione utilizza solo `SIZE-1` elementi per evitare conflitti tra la condizione di buffer pieno e vuoto.

3.5.1 Inter-Processo Comunication (IPC)

Domanda 3.6

Come fanno due processi a scambiarsi le informazioni necessarie alla cooperazione?

- Il Sistema Operativo (SO) fornisce dei meccanismi di **Inter-Process Communication (IPC)**.
- Sono disponibili opportune **system call** che permettono a due (o più) processi di:
  - **scambiarsi messaggi** oppure
  - **usare la stessa area di memoria condivisa**, in cui possono scrivere e leggere.

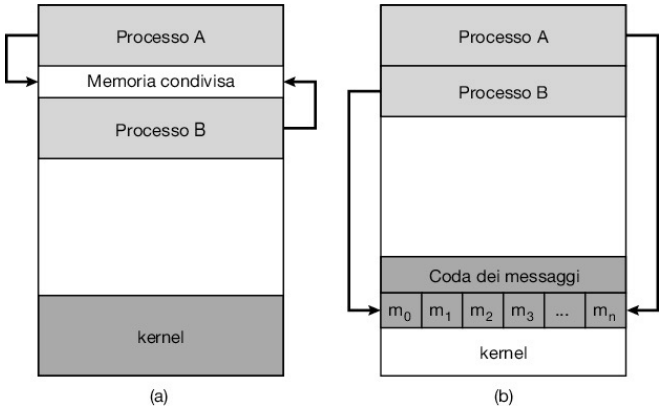


Figure 3.5: a) Memoria condivisa
b) Scambio di messaggi

In entrambi i casi, il SO mette a disposizione delle opportune **system call**. Ad esempio, per lo scambio di messaggi, saranno disponibili delle system call del tipo:

- `line = msgget();`
- `send(message, line, process-id);`
- `receive(message, line, process-id);`

(Nota:) I parametri sono solo indicativi, ogni specifica implementazione avrà il proprio insieme di argomenti.

Saranno necessarie alcune **scelte implementative**.
Nel caso dei **messaggi** (si parla spesso di *code di messaggi*):

- Una coda può essere usata da più di due processi?
- Quanti messaggi può ospitare al massimo una coda?
- Cosa deve fare un processo ricevente se non ci sono messaggi, o un processo trasmittente se la linea è piena?
- Si possono trasmettere messaggi di lunghezza variabile?

Nel caso della **memoria condivisa**:

- Può avere dimensione variabile?
- Quali processi hanno diritto di usarla?
- Cosa succede se la memoria condivisa viene rimossa?

4
Threads

5 Scheduling della CPU

5.1 Scheduling

5.1.1 Fasi di eleborazion e di I/O

Durante la vita di un processo, si alternano fasi di uso della **CPU** (CPU burst) e fasi di attesa per il completamento di operazioni di **I/O** (I/O burst).

Possiamo distinguere due categorie di processi:

- **Processi CPU-bound:** usano intensamente la CPU e interagiscono poco con i dispositivi di I/O (ad esempio un compilatore).
- **Processi I/O-bound:** usano poco la CPU ma fanno ampio uso dei dispositivi di I/O (ad esempio un editor o un browser).

5.1.2 Lo Scheduler della CPU

Consideriamo la situazione in cui un processo utente abbandona la **CPU**. Il Sistema Operativo (SO) si "sveglia" e deve decidere a quale, fra i processi in **Coda di Ready** (processi *ready to run*), assegnare la CPU. Questa operazione è detta **Scheduling della CPU**, e viene gestita dal modulo del SO detto **scheduler**.

Quando interviene lo scheduler per scegliere il successivo processo a cui assegnare la CPU? Possiamo considerare quattro situazioni, che ci porteranno a definire i concetti di **scheduling con** e **senza diritto di prelazione**:

1. Il processo che sta usando la CPU passa volontariamente dallo stato di running allo stato di waiting.
2. Il processo che sa usando la CPU termina.
   - In questi **due casi**, lo scheduler deve prendere un processo dalla coda di readt e mandarlo in eseucuzione
   - **Note:-**
     Un sistema operativo che intervenga nei casi 1 e 2 è sufficiente per implementare il multi-tasking
   - **Domanda 5.1**
     Che succede se mandiamo in eseucuzione un programma che contiene una istruzione del tipo `while(true) printf("who's carr?");`
   - Il **SO** deve poter intervenire in modo da evitare che un processo si simpossessi della CPU, quindi...

3. Il processo che sta usando la CPU viene obbligato a passare dallo stato di running allo stato di ready
   - Il passaggio non avviene mai **volontariamente**, il processo non vorrebbe lasciare la CPU a favore di qualcun'altro.
   - Nei sistemi time sharing il SO non perde **mai** completamente il controllo del sistema
   - Il SO mantiene il controllo della CPU attraverso un timer hardware, allo scadere del timer il controllo della CPU verrà restituito al SO, che sceglierà un processo dalla RQ un processo da mandare in esecuzione.
4. Un processo Px entra in coda di ready arrivando da una coda di wait oppure perchè è appena stato lanciato.

Domanda 5.2

Perchè il SO interviene in questo caso?

   - **Primo:** i processi non si spostano autonomamente da una coda all'altra. È il Sistema Operativo (SO) che gestisce i loro **PCB** (Process Control Block). Ad esempio, quando il SO si accorge del completamento di un'operazione di I/O per cui il processo P_x era in attesa, interviene per spostare (il PCB di) P_x dalla **coda di wait** alla **coda di ready**.
   - **Secondo:** se il processo P_x risulta più "importante" rispetto al processo attualmente in esecuzione, il SO può decidere di togliere quest'ultimo dalla **CPU** e mandare in esecuzione P_x.

Quando un sistema interviene solo nei casi 1 e 2 si parla di: **Scheduling senza (diritto di) prelazione**.
Quando un sistema interviene anche nei casi 3 e 4 si parla di: **Scheduling con (diritto di) prelazione**
Chiaramente, lo **scheduling preemptive** è più sicuro per gli utenti, ma la sua implementazione richiede un sistema operativo e un'architettura hardware più sofisticati (ad esempio, un **timer dedicato**).
I moderni **sistemi operativi general purpose** usano tutti una qualche variante di **scheduling preemptive**. Tuttavia, per applicazioni specifiche può essere sufficiente uno **scheduling non-preemptive**, permettendo l'uso di sistemi operativi più semplici e leggeri.
Lo **scheduling preemptive** può portare a situazioni che necessitano di essere gestite con attenzione. Ad esempio, consideriamo un processo che deve compiere un'operazione di **I/O** e chiama la relativa system call. Il controllo viene trasferito al sistema operativo, che inizia l'operazione per conto del processo utente. Nel frattempo, scade il timer e il controllo viene passato a un'altra porzione del codice del SO.
Di conseguenza, una operazione delicata (altrimenti non sarebbe stata gestita dal SO) viene interrotta a metà, e le **strutture dati** potrebbero trovarsi in uno stato inconsistente poiché la system call di **I/O** non ha finito di aggiornarle.

Domanda 5.3

Cosa succede se ora la CPU viene data a un altro processo utente che tenta di usare lo stesso dispositivo di **I/O** che il processo precedente stava utilizzando? Quale semplice soluzione può essere adottata in tali casi?

Note:-

Vogliamo che il processo riesca a completare la richiesta al controller dell'I/O, niente di più

. Mentre **Unix** è stato sviluppato fin dall'inizio come sistema di tipo **preemptive**, nei sistemi **Microsoft** la preemption è stata introdotta solo con **Windows 95**. Questo è dovuto al fatto che i sistemi operativi della famiglia **MS-Dos** sono nati come sistemi **mono-utente**, per i quali era sufficiente un sistema operativo più semplice. Inoltre, i primi sistemi per PC giravano su CPU semplici ed economiche, non dotate del supporto hardware necessario per implementare un sistema operativo **preemptive**.

5.1.3 Il Dispatcher

Quando lo scheduler ha scelto il processo a cui assegnare la CPU, interviene un altro modulo del SO, il *dispatcher*, che:

- Effettua l'operazione di *context switch*.
- Effettua il passaggio del sistema in *user mode*.
- Posiziona il *PC* della CPU alla corretta locazione del programma da far ripartire.

Si definisce **Dispatch latency** il tempo impiegato per effettuare la commutazione da un processo ad un altro.

5.2 Criteri di Scheduling

Come abbiamo visto, lo scheduler della CPU interviene per assicurare il corretto funzionamento del sistema. Tuttavia, quando lo scheduler deve mandare in esecuzione un processo, quale criterio usa per scegliere tra tutti i processi presenti nella coda di ready?

Si possono prendere in considerazione diversi obiettivi:

- Massimizzare l'**utilizzo** della CPU nell'unità di tempo, anche se questo dipende dal carico. - Massimizzare il **throughput**, ossia la produttività del sistema, che si misura come il numero di processi completati in media in una certa unità di tempo. - Minimizzare il **tempo di risposta**, cioè il tempo che intercorre da quando si avvia un processo a quando questo inizia effettivamente ad eseguire. Questo aspetto è particolarmente importante per i sistemi interattivi.

- Minimizzare il *Turnaround time*: ossia il tempo medio di completamento di un processo, che va da quando entra per la prima volta nella *ready queue* a quando termina. - Minimizzare il *Waiting time*: ossia la somma del tempo trascorso dal processo in *ready queue*, ovvero quando il processo è pronto per eseguire il suo codice ma la CPU è occupata da un altro processo.

Domanda 5.4

Che relazione c'è tra waiting time e turnaround time?

Note:-

Turnaround time - waiting time = tempo di esecuzione

5.3 Algoritmi di Scheudling

- **First Come, First Served (FCFS)**: Scheduling per ordine di arrivo.
- **Shortest Job First (SJF)**: Scheduling per brevità.
- **Priority Scheduling**: Scheduling per priorità.
- **Round Robin (RR)**: Scheduling circolare.
- **Multilevel Queue**: Scheduling a code multiple.
- **Multilevel Feedback Queue**: Scheduling a code multiple con retroazione.

Nota Bene: Nel seguito, considereremo processi con un unico *burst* di CPU, senza *burst* di I/O e con una durata espressa in generiche unità di tempo. Questo semplifica la comprensione degli algoritmi senza perdita di generalità.

5.3.1 First Come First Served (FCFS)

L'algoritmo *First Come, First Served (FCFS)* è facile da implementare: gestisce la *ready queue* (RQ) in modo FIFO (*First In, First Out*).

- Il *PCB* di un processo che entra nella *RQ* viene inserito in fondo alla coda.
- Quando la CPU si libera, viene assegnata al processo il cui *PCB* si trova in testa alla coda FIFO.

FCFS è un algoritmo non *preemptive*, per cui non è adatto per i sistemi *time-sharing*. Inoltre, con FCFS, il tempo di attesa per il completamento di un processo può risultare spesso molto lungo.

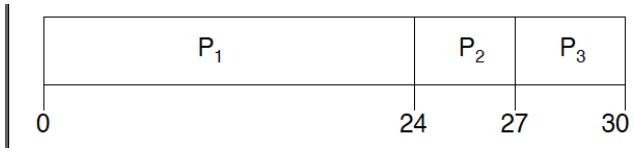
Esempio

Consideriamo tre processi che arrivano assieme al tempo t=0, e che entrano in CPU nell'ordine P1, P2, P3. Come abbiamo già detto, i tre processi eseguono per un unico burst di CPU, e poi terminano.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Table 5.1: Process and Burst Time

Usiamo un diagramma di Gnatt per rappresentare questa situazione Tempi di attesa $P_1 = 0; P_2 = 24; P_3 =$



17
Tempo medio di attesa $(0 + 24 + 27)/3 = 17$
Si dice che si è verificato **effetto convoglio**; i job più corti si sono dovuti accodare a quello lungo. Se invece

supponiamo che l'ordine di arrivo sia: P2, P3, P1 Tempi di attesa $P_1 = 6; P_2 = 0; P_3 = 3$
Tempo medio di attesa $(6 + 0 + 3)/3 = 3 \longleftrightarrow$ Molto meglio del caso precedente!

Osservazioni

Dunque, l'algoritmo *FCFS* sembra comportarsi male nei confronti dei processi brevi.
Inoltre, *FCFS* è pessimo per i sistemi *time-sharing* poiché non garantisce un tempo di risposta ragionevole.
Ancora peggio, *FCFS* non è adatto ai sistemi *real-time* perché non è *preemptive*.

Dall'esempio visto, sembra che le prestazioni migliorino facendo eseguire prima i processi più corti, indipendentemente dall'ordine di arrivo nella *ready queue*. Tuttavia, questo apre la porta a nuovi problemi, che andremo a considerare.

5.3.2 Shortest Job First (SJF)

Si esamina la durata del prossimo *burst* di CPU di ciascun processo in RQ e si assegna la CPU al processo con il *burst* di durata minima.
Il nome esatto di questo algoritmo è *Shortest Next CPU Burst*.
Può essere usato in modalità *pre-emptive* e *non pre-emptive*.
Nel caso *preemptive*, se arriva in *ready queue* un processo il cui *burst time* è inferiore al tempo rimanente del processo attualmente in esecuzione, quest'ultimo viene interrotto e la CPU passa al nuovo processo. Questo schema è noto come *Shortest-Remaining-Time-First* (SRTF).

Esempio

Non-preemptive

Esempio:

preemptive

Esempio:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Table 5.2: Process, Arrival Time, and Burst Time

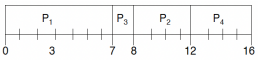


Figure 5.1: Average waiting time $(0 + 6 + 3 + 7)/4 = 4$

5.3.3 Osservazioni

Si può dimostrare che l'algoritmo *Shortest Job First* (SJF) è ottimale: spostando un processo breve prima di uno di lunga durata (anche se quest'ultimo è arrivato prima) si migliora l'attesa del processo breve più di quanto si peggiori quella del processo lungo. Di conseguenza, il tempo medio di attesa diminuisce, così come il *turnaround time*.

SJF è ottimale: nessun altro algoritmo di *scheduling* può produrre un tempo di attesa medio e un *turnaround time* medio migliori. Tuttavia, c'è un problema...
Purtroppo, la durata del prossimo burst di CPU di un processo non è nota, il che rende l'algoritmo *Shortest Job First* (SJF) non implementabile nella sua forma pura. SJF può al massimo essere approssimato utilizzando medie pesate per stimare la durata del prossimo burst di CPU di un processo, basandosi sulla durata dei burst di CPU precedenti.

Note:-
Da rivededere!!!!

Lo *scheduling* viene quindi eseguito sulla base di queste stime, fatte per tutti i processi nella *Ready Queue* in un dato momento.
In sostanza, il *First Come, First Served* (FCFS) è il peggiore degli algoritmi ragionevoli: funziona, ma spesso fornisce tempi medi di attesa e di *turnaround* pessimi.

Al contrario, lo *Shortest Job First* (SJF) è il migliore algoritmo possibile, ma non è implementabile nella pratica, e possiamo solo usarlo per fare simulazioni con processi i cui burst di CPU siano noti a priori.

FCFS e SJF rappresentano i due estremi di uno spettro di possibili algoritmi di *scheduling*. Un algoritmo di *scheduling* sarà tanto migliore quanto più le sue prestazioni si allontanano da quelle di FCFS e si avvicinano a quelle di SJF.

5.3.4 Scheduling a Priorità

SJF è un tipo scheduling a priorità, la durata del prossimo burst time è la priorità corrente di ogni processo. FCFS è uno scheduling a priorità, viene data più alta ai primi che arrivano In generale, il calcolo della priorità dei processi può essere:

- **Interna al sistema**: calcolata dal SO sulla base del comportamento di ogni processo (ad esempio, in base alle risorse usate fino a quel momento da un processo).

- **Esterna al sistema**: assegnata con criteri esterni al SO (ad esempio, una priorità che cambia in base a quale utente ha lanciato il processo).

Lo *scheduling* a priorità può essere implementato sia in modalità **preemptive** che **non preemptive**.

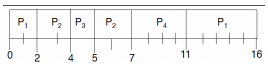


Figure 5.2: Average waiting time $(9 + 1 + 0 + 2/4 = 3$

Starvation e aging

Domanda 5.5: Problema

Che succede se un processo in RQ ha sempre una priorità peggiore di qualche altro processo in RQ?

Il processo potrebbe non essere mai scelto dallo scheduler. Questo fenomeno è noto come **starvation** (muore di fame...).

Per risolvere il problema della *starvation*, si usa un meccanismo chiamato **aging**: il SO aumenta progressivamente la priorità di un processo P_x man mano che P_x passa tempo nella Ready Queue (RQ). In questo modo, prima o poi, P_x avrà una priorità maggiore rispetto agli altri processi e verrà scelto dallo scheduler.

Domanda 5.6

Gli algoritmi *FCFS*, *SJF preemptive* e *non preemptive* possono provocare starvation?

Note:-

Per SJF, arrivano sempre processi con burst piccolissimi e quindi un processo più grande aspetterà (Sia per preemptive che non)

5.3.5 Scheduling Round Robin (RR)

Ogni processo ha a disposizione una certa quantità di tempo di CPU, chiamata **quanto di tempo** (valori ragionevoli vanno da 10 a 100 millisecondi). Per ora, assumiamo un unico quanto di tempo prefissato assegnato a tutti i processi.

Se entro questo arco di tempo il processo non lascia volontariamente la CPU, viene interrotto e rimesso nella Ready Queue (RQ). La RQ è vista come una coda circolare, e si verifica una sorta di *"girotondo"* di processi.

L'implementazione dello scheduling **round robin** è concettualmente molto semplice:

- Lo scheduler sceglie il primo processo in RQ (ad esempio secondo un criterio FCFS).
- Lancia un timer inizializzato al quanto di tempo.
- Passa la CPU al processo scelto.

Se il processo ha un CPU burst minore del quanto di tempo, il processo rilascierà la CPU volontariamente prima dello scadere del tempo assegnatogli. Se invece il CPU burst del processo è maggiore del quanto di tempo, allora:

- Il timer scade e invia un interrupt.
- Il SO riprende il controllo della CPU.
- Togliere la CPU al processo in esecuzione e metterlo in fondo alla RQ.
- Prendere il primo processo in RQ e ripetere tutto.

Osservazioni

Se ci sono n processi in coda ready e il quanto di tempo è q, allora ogni processo riceve $\frac{1}{n}$ del tempo della CPU e nessun processo aspetta per più di $(n-1)q$ unità di tempo.

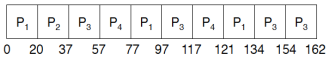
Il **Round Robin** è l'algoritmo di scheduling naturale per implementare il time sharing ed è quindi particolarmente adatto per i sistemi interattivi: nel caso peggiore, un utente non aspetta mai più di $(n-1)q$ unità di tempo prima che il suo processo venga servito.

Come vedremo negli esempi di casi reali, il SO adotta poi ulteriori misure per migliorare il tempo di risposta dei processi interattivi.

5.3.6 Esempio

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Table 5.3: Process and Burst Time



Tipicamente sia ha un *turnaround* medio maggiore di SJF, ma un migliore **tempo di risposta** Le prestazioni del **Round Robin** dipendono molto dal valore del quanto di tempo q scelto:

- q tendente a infinito rende **RR** uguale a **FCFS**.
- q tendente a zero produce un maggior effetto di "parallelismo virtuale" tra i processi.
- Tuttavia, questo aumenta il numero di context switch e, di conseguenza, l'overhead.

5.3.7 Scheduling a Code Multiple

I processi possono essere suddivisi in classi differenti:

- **foreground**: processi interattivi (es. un editor)
- **background**: processi che non interagiscono con l'utente
- **batch**: processi la cui esecuzione può essere differita

La risorsa di esecuzione (RQ) può essere partizionata in più code:

- I processi vengono inseriti in una coda basata sulle loro proprietà
- Ogni coda viene gestita con lo scheduling appropriato

Ogni coda ha quindi la sua politica di scheduling, ad esempio:

- *foreground*: **RR**
- *background e batch*: *FCFS*

Domanda 5.7

Ma come si sceglie fra le code?

- **Scheduling a priorità fissa**: servire prima tutti i processi nella coda foreground e poi quelli in background e batch. Possibilità di **starvation**.
- **Time slice**: ogni coda ha una certa quantità di tempo di CPU, ad esempio: 80% alla coda foreground e 20% alla coda background e batch

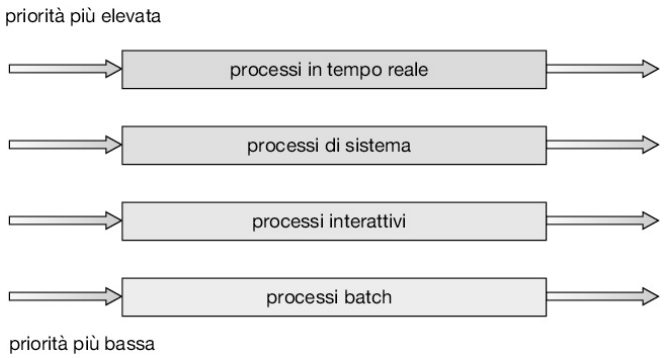


Figure 5.3: Partizionamento dei processi in più code

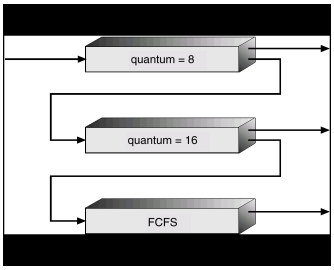


Figure 5.4: Esempio di MFQS

5.3.8 Scheduling a Code Multilivello con retroazione (MFQS)

Il tipo più generale di algoritmo di scheduling è lo **scheduling a code multilivello con retroazione** (MFQS), utilizzato dai sistemi operativi moderni.

- L'assegnamento di un processo a una coda non è fisso: i processi possono essere spostati dal SO per adattarsi alla lunghezza del loro *CPU burst*.
- Ogni coda è gestita con lo scheduling più adatto ai processi in essa contenuti.

Esempio (fig. 5.4):

- Le prime due code sono gestite con *Round Robin (RR)*, mentre la terza con *First-Come, First-Served (FCFS)*.
- Quando un processo nasce, è inserito nella prima coda ($q = 8$). Se non finisce il *CPU burst* entro il quanto, viene retrocesso alla coda successiva.
- È definita una priorità tra le code, che sono gestite con *preemption*.

La politica MFQS è caratterizzata da:

- numero di code
- algoritmo di scheduling per ogni coda
- quando declassare o promuovere un processo
- in che coda inserire un processo quando arriva (dall'esterno o da un *I/O burst*)

MFQS è il tipo di scheduling più generale e complesso da configurare.

5.4 Scheduling per sistemi multi-core

Sono ormai comuni le architetture con CPU a 2, 4, 8 core, Sono, in sostanza, dei piccoli sistemi multiprocessore in cui sullo stesso chip sono presenti due o più core che vedono la stessa memoria principale e condividono un livello di cache. La presenza di più "unità di esecuzione" dei processi, permette naturalmente di aumentare le prestazioni della macchina, posto che il SO sia in grado di sfruttare a pieno ciascun core.

I sistemi operativi moderni prevedono la **multielaborazione simmetrica** (SMP), in cui uno scheduler gira su ciascun core.

- I processi "ready to run" possono essere inseriti in una coda comune oppure in una coda separata per ogni core.
- Lo scheduler di ciascun core sceglie un processo dalla propria coda e lo manda in esecuzione.

Un aspetto chiave nei sistemi multi-core è il **bilanciamento del carico**, ossia la distribuzione omogenea dei processi tra i core.

- Con una coda comune, il bilanciamento è automatico: un core inattivo prende un processo dalla coda comune.
- Con code separate per ogni core, è necessario un meccanismo per spostare i processi dai core sovraccarichi a quelli scarichi. **Questo è il processo preferito dai moderni SO**
- Ad esempio, Linux SMP attiva il bilanciamento del carico ogni 200 ms o quando una coda si svuota.

Spostare un processo tra core può causare rallentamenti dovuti alla **cache**, poiché il processo potrebbe non trovare i dati nelle cache private del nuovo core. Non trovandoli è costretto a spendere più tempo per recuperarli, se va bene, dalla cache L3, che condivide con gli altri core. Per evitare questo problema, specifiche *system call* permettono di vincolare un processo a un certo core.

5.4.1 Esempio

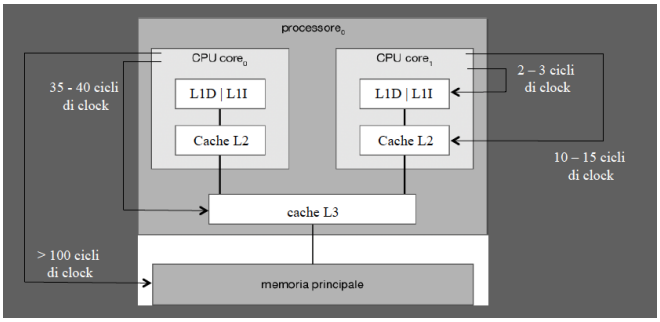


Figure 5.5: Scheduling example

Qui vediamo il costo in cicli di clock necessari per accedere ad un dato/istruzione in un certo livello di cache. I valori si riferiscono ad un processore Intel core i7, ma valgono per la maggior parte dei processori moderni 5.5.

5.5 Esempi di sistemi operativi

Solaris utilizza uno **scheduling a priorità con code multiple a retroazione**, suddividendo i processi in 4 classi:

1. real time (priorità maggiore)
2. sistema
3. interattiva

4. time sharing (priorità minore)

- Un processo usa la CPU fino a quando non termina, va in *wait*, esaurisce il quanto di tempo o è preemptato.
- I processi delle classi *time sharing* e *interattiva* hanno criteri di scheduling simili con 60 livelli di priorità.
- La priorità di un processo e il quanto di tempo assegnato sono inversamente proporzionali. Se un processo esaurisce il quanto, la sua priorità viene abbassata; al contrario, se si sospende prima, la priorità aumenta.

Il comportamento del processo stabilisce se rientra nella classe *time sharing* (priorità 0-49) o *interattiva* (priorità 50-59)

Priorità corrente	Quanto di tempo (millisecondi)	Nuova priorità (quanto esaurito)	Nuova priorità (quanto non esaurito)
0	200	0	50
20	120	10	52
30	80	25	53
59	20	49	59

Table 5.4: Tabella delle priorità e tempi

La **priorità corrente** di un processo determina il **quanto di tempo** che gli viene assegnato. Priorità e tempo assegnato sono inversamente proporzionali.

- **Quanto esaurito**: se un processo ha esaurito tutto il quanto di tempo senza sospendersi, la sua nuova priorità sarà più bassa, e in futuro riceverà un quanto di tempo più lungo.
- **Quanto non esaurito**: se un processo si sospende prima di consumare tutto il quanto, la sua nuova priorità sarà più alta, e in futuro riceverà un quanto di tempo più breve.
- I processi *real time* e di *sistema* hanno priorità fissa, superiore a quella delle classi *time sharing* e *interattiva*.
- Lo scheduler assegna la CPU al processo con la priorità globale più alta e, in caso di parità, utilizza il *Round Robin* (RR).
- L'algoritmo è **preemptive**: un processo attivo può essere interrotto da uno con priorità globale più alta.

5.5.1 Lo scheduling in Windows

Lo scheduling in Windows è basato su **priorità con retroazione e prelazione**, utilizzando 32 livelli di priorità:

- I processi *real-time* hanno priorità da 16 a 31.
- I processi non real-time hanno priorità da 1 a 15, con 0 riservato.

Per i processi non real-time:

- Quando un processo nasce, ha una priorità iniziale di 1.
- Lo scheduler assegna la CPU al processo con la priorità più alta, usando *Round Robin* (RR) in caso di parità.
- Se un processo va in *wait* prima di esaurire il quanto di tempo, la sua priorità viene aumentata (fino a 15), a seconda dell'evento in attesa (maggiore incremento per input da tastiera, minore per I/O da disco).
- Se un processo esaurisce il quanto di tempo, la sua priorità viene abbassata, ma mai sotto 1.

Questa strategia favorisce i processi interattivi (mouse e tastiera) per migliorare il **tempo di risposta**. Inoltre, quando un processo passa in *foreground*, il suo quanto di tempo viene moltiplicato per 3, consentendogli di mantenere la CPU per un periodo più lungo.

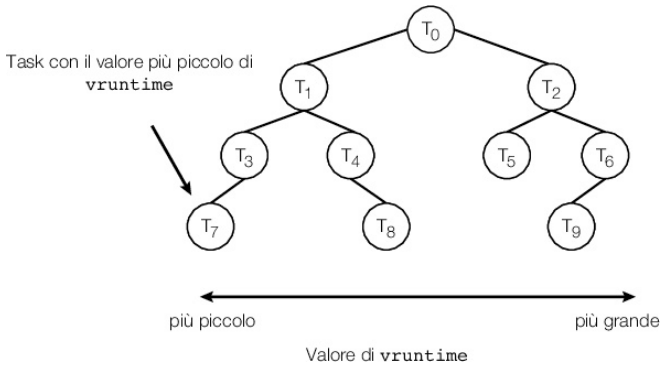


Figure 5.6: vrun time tree

5.5.2 Lo Scheduling in Linux

Dal 2007, Linux utilizza il **Completely Fair Scheduler** (CFS) come algoritmo di scheduling predefinito.

Il CFS distribuisce equamente il tempo di CPU tra i processi *ready to run*, seguendo l'assunzione che se ci sono N processi attivi, ciascun processo dovrebbe ricevere esattamente $\frac{1}{N}$ del tempo di CPU.

Ad ogni *context switch*, il CFS ricalcola per quanto tempo assegnare la CPU a un processo P, in modo che tutti abbiano la stessa quantità di tempo CPU. Siano:

- P.`expected_run_time`: il tempo di CPU spettante a P;
- P.`vruntime`: il tempo di CPU già consumato da P;
- P.`due_cputime`: il tempo di CPU che ancora spetta a P.

Dunque:

$$P.\texttt{vruntime} = P.\texttt{expected\_run\_time} - P.\texttt{due\_cputime}$$

La CPU viene assegnata al processo con il valore più basso di P.`vruntime`, ossia al processo che ha usato meno CPU fino a quel momento.

Nel CFS, i processi *ready to run* non sono organizzati in code di scheduling, ma come nodi in un **red-black tree (R-B tree)**, che consente operazioni di ricerca, inserimento e cancellazione con complessità computazionale $O(\log n)$, dove n è il numero di nodi.

Negli alberi R-B il nodo più a sinistra è sempre quello col valore chiave più basso, e nel CFS i processi sono inseriti nel R-B tree usando come chiave P.vruntime. Dunque, il processo associato al nodo più a sinistra ha il valore P.vruntime più basso, cioè è il processo che ha usato la CPU per meno tempo, e al context switch sarà scelto per entrare in esecuzione.

6

Sincronizzazione dei Processi

6.1 Introduzione

Più processi possono cooperare per compiere un determinato lavoro, e spesso **condividono dei dati**.

- È fondamentale che l'accesso ai dati condivisi da parte dei vari processi non produca dati inconsistenti.
- I processi cooperanti devono quindi **sincronizzarsi** per accedere ai dati condivisi in modo ordinato.
- **Problema**: mentre un processo P sta elaborando dati condivisi, il SO potrebbe toglierlo dalla CPU in qualsiasi momento. Altri processi non devono poter accedere ai dati condivisi finché P non ha completato l'elaborazione.

6.1.1 Esempio: Produttore-Consumatore con n eleemnti

Usiamo una variabile ***condivisa*** **counter** inizializzata a 0 che indica il numero di elementi nel buffer.

I due programmi sono corretti se considerati separatamente, ma possono non funzionare quando vengono eseguiti insieme.

- Il problema risiede nell'uso della variabile condivisa `counter`.
- Che succede se il produttore esegue `counter++` mentre *contemporaneamente* il consumatore esegue `counter-`?
- Se `counter` all'inizio vale 5, dopo `counter++` e `counter-` può valere 4, 5 o 6!
- N.B.: diciamo che possono non funzionare, e non che non funzionano, perché la condizione problematica potrebbe non verificarsi sempre.

Il problema si verifica perchè `counter++, counter-` non sono **operazioni atomiche** Le operazioni sui dati condivisi possono portare a risultati imprevisti. Consideriamo le istruzioni per il **produttore** e il **consumatore** relative alla variabile condivisa `counter`: **Produttore:**

```
load(registro1, counter);  % Carica il valore di counter in registro1
add(registro1, 1);         % Incrementa il valore nel registro di 1
store(registro1, counter);  % Salva il valore incrementato in counter
```

Consumatore:

```
load(registro1, counter);  % Carica il valore di counter in registro1
sub(registro1, 1);         % Decrementa il valore nel registro di 1
store(registro1, counter);  % Salva il valore decrementato in counter
```

Se il produttore e il consumatore accedono a `counter` in modo non sincronizzato, il valore finale di `counter` può risultare errato e instabile.

Quando i processi devono accedere e modificare dati condivisi, è fondamentale che si **sincronizzino** affinché ciascuno possa completare le proprie operazioni sui dati prima che un altro processo possa accedervi.

- Questo approccio assicura l'integrità dei dati e previene condizioni di competizione.
- Da notare che il problema non si presenta se tutti i processi coinvolti nell'accesso a un insieme di dati condivisi devono solo **leggere** quei dati.

6.2 Sezioni critiche

Siano dati n processi $P_1, \ldots, P_n$ che usano variabili condivise.

Ogni processo ha una porzione di codice, detta **sezione critica**, in cui manipola le variabili condivise (o anche solo un loro sottoinsieme).
Quando un processo P_i è dentro alla propria sezione critica, nessun altro processo P_j può eseguire il codice della propria sezione critica, poiché userebbe le stesse variabili condivise (o anche solo un loro sottoinsieme).
L'esecuzione delle sezioni critiche di $P_1, \ldots, P_n$ deve quindi essere **mutualmente esclusiva**.
Mentre un processo P_i sta eseguendo codice nella propria sezione critica, potrebbe essere tolto dalla CPU dal sistema operativo a causa del normale avvicendamento tra processi.
Fino a che P_i non ha terminato di eseguire il codice della sua sezione critica, **nessun altro processo P_j che deve manipolare le stesse variabili condivise potrà eseguire il codice della propria sezione critica**.

Osservazioni 6.2.1

È importante notare che P_j può comunque eseguire del codice, quando entra in esecuzione, ma non il codice della propria sezione critica.

Definizione 6.2.1: Sezione critica

Sezione critica: porzione di codice che deve essereeseguito senza intrecciarsi (nell'avvicendamento in CPU) col codice delle sezioni critiche di altri processi che usano le stesse variabili condivise

6.2.1 Problema della Sezione Critica

Per garantire l'accesso sicuro alle variabili condivise, è necessario stabilire un **protocollo di comportamento** per i processi.

- Un processo deve **"chiedere il permesso"** per entrare nella sezione critica, utilizzando una opportuna porzione di codice detta **entry section**.
- Un processo che esce dalla sua sezione critica deve **"segnalarlo"** agli altri processi, usando una opportuna porzione di codice detta **exit section**.

Un generico processo Pi contiene una sezine critica che avrà la seguente struttura

```
altro codice
    entry section
    sezione critica
    exit section
altro codice
```

Siano dati n processi $P_1, \ldots, P_n$ che usano delle variabili condivise. Una soluzione corretta al problema della sezione critica per $P_1, \ldots, P_n$ deve soddisfare i seguenti tre requisiti:

1. **Mutua esclusione:** Se un processo P_i è entrato nella propria sezione critica ma non ne è ancora uscito (attenzione, P_i non è necessariamente il processo in esecuzione, cioè quello che sta usando la CPU), nessun altro processo P_j può entrare nella propria sezione critica.

2. **Progresso:** Se un processo lascia la propria sezione critica, deve permettere ad un altro processo P_j di entrare nella propria (di P_j) sezione critica. Se la sezione critica è vuota e più processi vogliono entrare, uno tra questi deve essere scelto in un tempo finito (*in altre parole, esiste un processo che entrerà in sezione critica in un tempo finito)*

   - **Osservazioni 6.2.2**

     Qquesta condizione garantisce che l'insieme dei processi P1,... Pn (o anche solo un loro sottoinsieme) non finisca in una condizione di deadlock: tutti fermi in attesa di riuscire ad entrare nella loro sezione critica

3. **Attesa limitata:** se un processo P_i ha già eseguito la sua entry section (ossia ha già chiesto di entrare nella sua sezione critica), esiste un limite al numero di volte in cui altri processi possono entrare nelle loro sezioni critiche prima che tocchi a P_i *(in altre parole,* ***qualsiasi*** *processo deve riuscire ad entrare in sezione critica in un tempo finito)*

   - **Osservazioni 6.2.3**

     Quest'ultima condizione assicura che il processo Pi non subisca una forma di **starvation**: non riesce a proseguire la sua computazione perché viene sempre sopravanzato da altri processi.

Una qualsiasi soluzione corretta al problema della sezione critica deve permettere ai processi di portare avanti la loro computazione **indipendentemente** dalla velocità relativa a cui essi procedono (ossia da quanto frequentemente riescono ad usare la CPU), purchè questa sia maggiore di zero.
Notate: dire che la soluzione deve essere indipendente dalla velocità relativa a cui procedono i processi significa, più tecnicamente, che:

- la soluzione non deve dipendere dal tipo di *scheduling* della CPU adottato dal SO (ossia dall'ordine e dalla frequenza con cui i processi vengono eseguiti);
- purché, chiaramente, si usi un algoritmo di *scheduling* ragionevole, come quelli che abbiamo visto.

Il problema della sezione critica è particolarmente delicato quando sono coinvolte strutture dati del sistema operativo.

- Ad esempio, se due processi utente eseguono una `open` sullo stesso file, vi saranno due accessi concorrenti alla stessa struttura dati del SO: la tabella dei file aperti nel sistema.
- È importante che questa tabella (come tutte le strutture dati del SO) non venga lasciata in uno stato inconsistente a causa dell'accesso concorrente dei due processi.

Il progettista del SO deve decidere come vanno gestite le sezioni critiche del sistema operativo, e le due scelte possibili sono di sviluppare un *kernel* con o senza diritto di prelazione.

- In un *kernel* **con diritto di prelazione**, un processo in *kernel mode* può essere interrotto da un altro processo (ad esempio, perché è scaduto il quanto di tempo).
- In un *kernel* **senza diritto di prelazione**, un processo in *kernel mode* non può essere interrotto da un altro processo. (*Secondo voi questo potrebbe essere rischioso?*)

Un *kernel* senza diritto di prelazione è più facile da implementare: basta disattivare gli interrupt quando un processo è in *kernel mode*.

- Non c'è più bisogno di preoccuparsi dell'accesso concorrente alle sezioni critiche del *kernel*: un solo processo alla volta può accedere alle strutture dati del *kernel*, perché un solo processo alla volta può essere in *kernel mode*.

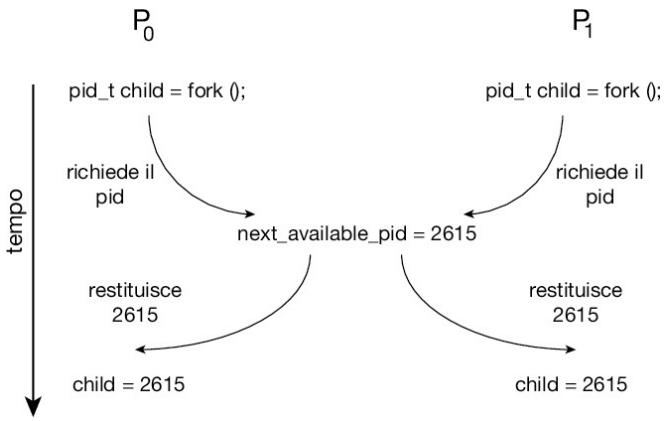


Figure 6.1: il caso di due processi che eseguono "insieme" una fork: senza opportune precauzioni, potrebbero nascere nel sistema due nuovi processi con lo stesso PID!

- Se il codice del SO è scritto correttamente, la disabilitazione degli interrupt sarà temporanea e di breve durata, e tutto continuerà a funzionare normalmente.

Del resto, i *kernel* con diritto di prelazione sono più adatti per le applicazioni *real time*, in cui la disabilitazione degli interrupt (che potrebbero essere allarmi da gestire immediatamente) non è accettabile.

- In generale, i *kernel* con diritto di prelazione hanno un tempo di risposta inferiore, per ovvie ragioni.

Note:-

- Windows 2000 e XP erano *kernel* senza diritto di prelazione, mentre i loro successori sono stati tutti progettati con diritto di prelazione.
- La maggior parte delle versioni recenti di Solaris, Unix e Linux sono *kernel* con diritto di prelazione.

Osservazioni 6.2.4

Un *kernel* senza diritto di prelazione disabilita gli interrupt per il tempo necessario al codice del SO (ad esempio di una *system call*) per accedere in modo mutuamente esclusivo a una qualche struttura dati del SO.

Domanda 6.1: Domanda d'esame

Perché una tale soluzione non è adatta per proteggere le strutture dati condivise da due processi utente, ossia per implementare le sezioni critiche dei processi utente?

Note:-

Tendenzialmente non si vuole che i processi utenti possano accedere alla modalità kernel.
Magari il codice utente non riabilita più gli interrupt? Siamo rovinati :()

6.2.2 Sincronizzazione via Hardware

Soluzioni semplici ed eleganti al problema della sezione critica (ma con un grave difetto, come vedremo) possono essere ottenute usando speciali istruzioni macchina, presenti in tutte le moderne CPU (i nomi di queste istruzioni negli *instruction set* di diversi processori possono ovviamente variare; l'importante è ciò che fanno):

- `TestAndSet(var1)`: testa e modifica il valore di una cella di memoria;
- `Swap(var1, var2)`: scambia il valore di due celle di memoria.

Importante: sono istruzioni macchina, e quindi atomiche, ovvero non possono essere interrotte a metà da un *context switch*.
Vediamo ad esempio la `TestAndSet`, che potrebbe essere implementata così:

```
boolean TestAndSet(boolean *lockvariable) {
    boolean tempvariable = *lockvariable;
    *lockvariable = true;
    return tempvariable;
}
```

Ossia:

- salva il valore di `*lockvariable` in `tempvariable`;
- setta a `true` `*lockvariable`;
- restituisce il vecchio valore di `*lockvariable`.

Ed ecco come si può **realizzare la mutua esclusione** usando la `TestAndSet`:

```
Shared data: boolean lock = false;
Processo Pi:
do {
    while (TestAndSet(&lock));
    // sezione critica (qui dentro lock = true)
    lock = false;
    // sezione non critica
} while (true);
```

Il semplice algoritmo appena visto è un esempio di soluzione al problema della sezione critica basato sull'uso di una variabile condivisa detta *lock*.

- Si dice che la sezione critica è controllata dal *lock*, e solo il processo che acquisisce il *lock* può entrare in sezione critica.

La struttura generale di queste soluzioni è quindi del tipo:

```
do {
    acquisisci il lock
    sezione critica
    restituisci il lock
    sezione non critica
} while (true);
```

Attesa limitata. NON E' GARANTITA! Infatti P1 potrebbe uscire dalla sezione critica ("lock=false") e, sempre all'interno dello stesso quanto di tempo, tornare immediatamente a tentare di acquisire il lock, riuscendoci, e la situazione può ripetersi all'infinito!

Domanda 6.2

Perché un meccanismo di aging non funzionerebbe, in questo caso?

Note:-

Non funziona perchè P2 entra in CPU, ma spreca tutto il suo quanto di tempo nel while(TestAndSet(block), quindi riperde la priorità

Ecco la soluzione corretta per n processi:

```
shared data boolean attesa[n], lock; // entrambi inizializzati a false
boolean chiave;
do {
    attesa[i] = true; // Pi annuncia di voler entrare in SC
    chiave = true;
    while (attesa[i] && chiave)
        chiave = TestAndSet(&lock);
```

```
    attesa[i] = false;

    // sezione critica

    j = (i + 1) mod n;
    while ((j != i) && !attesa[j])
        j = (j + 1) mod n;
    if (j == i)
        lock = false;
    else
        attesa[j] = false;

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

La soluzione che abbiamo visto, basata sull'uso di speciali istruzioni macchina, ha un problema di fondo:

```
while (TestAndSet(&lock));
```

Il processo che attende il proprio turno per entrare in una sezione critica occupata consuma CPU inutilmente. Tecnicamente, si dice che sta facendo *busy-waiting* (a volte si usa anche l'espressione "attesa attiva").

P1 entra nella sezione critica. Incomincia a lavorare in mutua esclusione. Scade il quanto assegnatogli. La CPU viene data a P2	P2 cerca di entrare in sezione critica, occupata da P1. P2 cicla inutilmente fino alla fine del suo quanto di tempo. La CPU viene ridata a P1	P1 termina il lavoro in sezione critica e ne esce, rilasciando il lock. Al prossimo quanto di tempo P2 potrà acquisire il lock

Figure 6.2: Busy-Waiting

Osservazioni 6.2.5

E se invece di due processi ce ne sono N che competono per acquisire lo stesso lock, un algoritmo di scheduling round robin potrebbe produrre uno spreco del tempo di CPU pari a $N-1$ quanti di temp.

6.3 Semafori

Definizione 6.3.1: Semaforo

Strumento di sincronizzazione che, con l'aiuto del sistema operativo, può essere implementato senza *busy-waiting*.

Definizione di Semaforo S:

- È una variabile intera (per ora assumiamo che sia inizializzata a 1) su cui si può operare solo tramite due operazioni atomiche (che descriviamo così, anche se non implementate in questo modo):
  - `wait(S)`:

    ```
    while (S <= 0) do no-op;
    S = S - 1;
    ```

  - `signal(S)`:

    ```
    S = S + 1;
    ```

Un semaforo può essere visto come un **oggetto** condiviso da tutti i processi che devono usarlo per sincronizzarsi fra loro.

- La variabile intera S viene di solito chiamata *variabile semaforica*, e il suo valore corrente è detto *valore del semaforo*.

- `Wait` e `Signal` sono i metodi con cui si utilizza l'oggetto semaforo (in realtà, è necessaria anche un'operazione per inizializzare il valore della variabile semaforica).

In letteratura, i termini `Wait` e `Signal` sono talvolta sostituiti da termini olandesi, usati originariamente da Dijkstra:

- `P` (*Proberen* = verificare) al posto di `Wait`;
- `V` (*Verhogen* = incrementare) al posto di `Signal`.

Altri termini usati:

- `Down` (per la `Wait`, che decrementa il semaforo);
- `Up` (per la `Signal`, che incrementa il semaforo).

6.3.1 Uso dei semafori

Adesso la soluzione del problema della sezione critica per un gruppo di processi è semplice. La variabile semaforica `mutex` (mutual exclusion) fa da variabile di *lock*:

```
shared variable semaphore mutex = 1;

Generico processo Pi:
do {
    wait(mutex);
    // sezione critica
    signal(mutex);
    // sezione non critica
} while (true);
```

In realtà, i semafori possono essere usati per **qualsiasi** problema di **sincronizzazione** (ne vedremo più avanti alcuni relativamente complessi). Ad esempio, se vogliamo eseguire una generica operazione S_1 fatta dal processo P_1 prima di S_2, fatta dal processo P_2, possiamo usare un semaforo `sync` inizializzato a 0:

- P_1 esegue:

```
S1;
signal(sync);
```

- P_2 esegue:

```
wait(sync);
S2;
```

> **Osservazioni 6.3.1**
>
> **Attenzione:** il nome scelto per un semaforo non ha nessuna relazione con l'uso che ne verrà fatto.

- Un semaforo è una **variabile** (in realtà, come vedremo tra poco, è una struttura dati) che ovviamente possiamo chiamare come preferiamo.
- Naturalmente, è meglio usare nomi che ricordino **l'uso che faremo** di un semaforo. Quindi, chiameremo `mutex` un semaforo usato per implementare una mutua esclusione, e `sync` un semaforo usato per implementare un meccanismo di sincronizzazione tra due processi.
- Ma non cambierebbe nulla se chiamassimo i due semafori rispettivamente `X` e `Y`, o anche `Pippo` e `Pluto`.
- Ciò che importa è **come li usiamo**.

6.3.2 Implementazione dei semafori

La definizione di `wait` e `signal` che abbiamo dato utilizza il *busy-waiting*, e questo è proprio ciò che vorremmo evitare.

- I semafori implementati attraverso il *busy-waiting* esistono e prendono di solito il nome di *spinlock* (nel senso che il processo "gira" mentre testa la variabile di lock, proprio come nelle tecniche di sincronizzazione via hardware).
- Per evitare il *busy-waiting* dobbiamo farci aiutare dal Sistema Operativo, che mette a disposizione opportune strutture dati e *system call* per l'implementazione delle operazioni di `wait` e `signal`.

Quando un gruppo di processi ha bisogno di un semaforo, lo richiede al Sistema Operativo tramite una *system call*.

- Il SO alloca un nuovo semaforo all'interno di una lista di semafori memorizzata nelle aree dati del kernel.
- Ogni semaforo è implementato usando due campi: `valore` e `lista di attesa`.

```
typedef struct {
    int valore;
    struct processo *waiting_list;
} semaforo;
```

Due *system call* sono disponibili per **implementare** le operazioni di `wait` e `signal`:

- `sleep()`: toglie la CPU al processo che la invoca e manda in esecuzione uno dei processi nella Ready Queue. Il processo che ha chiamato `sleep` non viene rimesso nella Ready Queue (N.B.: a volte `sleep()` è chiamata `block()`).
- `wakeup(P)`: inserisce il processo P nella Ready Queue.

Implementazione della `wait`:

```
wait(semaforo *S) {
    S->valore--;
    if (S->valore < 0) {
        aggiungi questo processo a S->waiting_list;
        sleep();
    }
}
```

La chiamata di `sleep()` provoca un *context switch*, e il processo sospeso non consuma CPU inutilmente, poiché il suo PCB non è più nella Ready Queue, ma nella lista di attesa del semaforo su cui si è sospeso. Si dice anche che il processo si è "**addormentato**" sul semaforo S. `signal(semaforo *S)`:

```
signal(semaforo *S) {
    S->valore++;
    if (S->valore <= 0) { /* c e ' qualcuno in attesa */
        togli un processo P da S->waiting_list;
        wakeup(P);
    }
}
```

Nota: `wakeup(P)` rimette P nella Ready Queue, quindi P è pronto a usare la CPU quando sarà il suo turno. Si dice che P è stato "svegliato".

NOTATE BENE: `wait` e `signal` sono di solito *system call* direttamente messe a disposizione dal Sistema Operativo, anche se a volte con nomi diversi.

- `wait` e `signal` sono esse stesse sezioni critiche. Perché? **(Perchè condividono delle variabili)**
- Sono sezioni critiche molto corte (circa 10 istruzioni macchina), quindi vanno bene implementate con *spinlock* o disabilitazione degli interrupt (che avviene sotto il controllo del SO).

Domanda 6.3

Quale soluzione è migliore per i sistemi monoprocessore e quale per i sistemi multiprocessore?

NOTATE ANCHE: All'inizio, la semantica di `wait` era:

```
wait (S):
    while (S <= 0) do no-op;
    S = S - 1;
```

Domanda 6.4

Ma nell'implementazione tramite `sleep`, vediamo che il valore del semaforo può essere negativo. Come mai?

Note:-

Per conoscere quanti processi in un certo istante sono addormentanti

- Se $S->valore < 0$, il suo valore assoluto ci dice quanti processi sono in attesa (*in wait*) su quel semaforo (si veda il codice della `wait`).

NOTATE ANCORA: Il valore del semaforo può anche essere un intero maggiore di 1. Ad esempio, se una risorsa può essere usata contemporaneamente da un massimo di tre processi:

```
semaphore counter = 3; // counter viene inizializzato a 3
Generico processo Pi:
repeat {
    wait(counter);
    // usa la risorsa
    signal(counter);
    // remainder section
} until false;
```

6.3.3 Riassunto

Riassumendo, attraverso i semafori implementati usando `sleep()` e `wakeup(P)`, i processi utente possono contenere sezioni critiche arbitrariamente lunghe senza:

- Sprecare inutilmente tempo di CPU (come accadrebbe se implementassimo le sezioni critiche con il *busy-waiting*),
- Rischiare di dare il controllo della CPU al processo (come accadrebbe se si usasse la disabilitazione degli interrupt gestita direttamente dai processi utente).

Osservazioni 6.3.2

Le operazioni di `wait` e `signal` (che sono esse stesse sezioni critiche) possono invece essere implementate con *busy-waiting* o disabilitazione degli interrupt, poiché queste operazioni durano poco tempo e avvengono sotto il controllo del Sistema Operativo.

6.4 Definizione di DeadLock

Definizione 6.4.1

Si definisce **deadlock** di un sottoinsieme di processi del sistema $\{P_1, P_2, \ldots, P_n\} \subseteq P$ la situazione in cui ciascuno degli n processi P_i è in attesa del rilascio di una risorsa detenuta da uno degli altri processi del sottoinsieme;

si forma cioè una catena circolare per cui:

$$P_1 \text{ aspetta } P_2 \dots \text{ aspetta } P_n \text{ aspetta } P_1$$

Anche se non tutti i processi del sistema sono bloccati, la situazione non è desiderabile in quanto può bloccare alcune risorse e, di conseguenza, danneggiare anche i processi non coinvolti nel deadlock.

6.5 Definizione di Starvation

Definizione 6.5.1

Un processo è in **starvation** se non riesce mai a portare avanti la propria computazione.

Questo può accadere per diverse ragioni:

- Non viene mai selezionato dallo *scheduler* per entrare in esecuzione.
- Non riesce mai ad entrare in una sezione critica.
- Non riesce mai a prelevare una risorsa necessaria per proseguire la sua computazione.

Nota: Il *deadlock* implica *starvation*, ma non vale il contrario.

6.6 Deadlock & Starvation: (stallo e attesa indefinita)

I **semafori** sono le primitive di sincronizzazione più semplici e più usate nei moderni Sistemi Operativi, e permettono di risolvere qualsiasi problema di sincronizzazione fra processi.

- Tuttavia, sono primitive di sincronizzazione *non strutturate* e quindi possono essere "rischiosi".
- Usando i semafori, non è difficile scrivere programmi che funzionano male, portando a situazioni di *deadlock* o *starvation*.

Esempio:

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

Domanda 6.5

Se S e Q sono inizializzati a 1, cosa succede se P_0 e P_1 vengono eseguiti concorrentemente?

- Se dopo la `wait(S)` di P_0, la CPU viene assegnata a P_1, che esegue `wait(Q)`, e successivamente la CPU torna a P_0, P_0 non può più proseguire, causando così un **deadlock**.

Problema dei semafori: le operazioni `wait` e `signal` sono indipendenti e possono essere usate in modo errato. Esistono primitive di sincronizzazione più strutturate (es. *Regioni Critiche Condizionali*, *Monitor*) che possono evitare questi problemi.

Approfondimento: potete leggere la sezione 6.7 del testo per una descrizione del concetto di *Monitor*.

7
Esempi di sincronizzazione

7.1 Produttori-Consumatori con memoria limitata

Utilizziamo un buffer circolare di `SIZE` posizioni in cui i produttori inseriscono i dati e i consumatori li prelevano.

Dati Condivisi e Inizializzazione dei Semafori

```
typedef struct {...} item;
item buffer[SIZE];
semaphore full, empty, mutex;
item nextp, nextc;
int in = 0, out = 0;
full = 0;
empty = SIZE;
mutex = 1;
```

- `full`: conta il numero di posizioni piene del buffer.
- `empty`: conta il numero di posizioni vuote del buffer.
- `mutex`: semaforo binario per garantire l'accesso in mutua esclusione al buffer e alle variabili `in` e `out`.
- `in` e `out`: servono per gestire l'indice del buffer circolare.

7.1.1 Codice del Produttore

Il codice per il produttore è il seguente:

```
while (true) {
    // produce un item in nextp
    wait(empty);
    wait(mutex);
    buffer[in] = nextp; // inserisce nextp nel buffer
    in = (in + 1) % SIZE; // aggiorna l'indice in
    signal(mutex);
    signal(full);
}
```

7.1.2 Codice del Consumatore

Il codice per il consumatore è il seguente:

```
while (true) {
    wait(full);
    wait(mutex);
    nextc = buffer[out]; // preleva un item dal buffer
    out = (out + 1) % SIZE; // aggiorna l'indice out
    signal(mutex);
    signal(empty);
    // consuma l'item in nextc
}
```

7.1.3 Spiegazione

- Usando il semaforo `mutex`, garantiamo che solo un processo per volta acceda in mutua esclusione al buffer e alle variabili condivise `in` e `out`.
- Il semaforo `empty` assicura che i produttori possano inserire dati solo se ci sono posizioni vuote nel buffer.
- Il semaforo `full` garantisce che i consumatori possano prelevare dati solo se nel buffer sono presenti item da consumare.

Note:-

Implementare Produttori-Consumatori esempio slide :D

Domanda 7.1

Come può essere semplificato il codice se possiamo supporre che esista un solo produttore?
Come può essere semplificato il codice se possiamo supporre che esista un solo consumatore?

7.2 Problema dei Lettori-Scrittori

Vogliamo gestire l'accesso concorrente a un file condiviso tra più processi che possono essere lettori o scrittori:

- I lettori richiedono solo l'accesso in lettura e possono accedere al file contemporaneamente ad altri lettori.
- Gli scrittori richiedono l'accesso in scrittura e devono avere accesso esclusivo al file, senza che altri lettori o scrittori possano accedervi contemporaneamente.

Strutture Dati Condivise

Le seguenti strutture dati vengono utilizzate per la sincronizzazione tra lettori e scrittori:

```
semaphore mutex = 1, scrivi = 1;
int numlettori = 0;
```

- `mutex`: semaforo per garantire la mutua esclusione quando si aggiorna la variabile `numlettori`.
- `scrivi`: semaforo che garantisce l'accesso esclusivo al file per gli scrittori.
- `numlettori`: contatore che tiene traccia del numero di lettori attivi.

7.2.1 Codice del Processo Scrittore

Il codice per uno scrittore è il seguente:

```
wait(scrivi);
// esegui la scrittura del file
signal(scrivi);
```

7.2.2 Codice del Processo Lettore

Il codice per un lettore è il seguente:

```
wait(mutex); // mutua esclusione per aggiornare numlettori
numlettori++;
if (numlettori == 1) wait(scrivi); // il primo lettore blocca eventuali scrittori
signal(mutex);

// leggi il file

wait(mutex);
numlettori--;
if (numlettori == 0) signal(scrivi); // l'ultimo lettore sblocca eventuali scrittori
signal(mutex);
```

7.2.3 Spiegazione

- Lettori quando un lettore vuole accedere al file, incrementa `numlettori` sotto mutua esclusione grazie a `mutex`. Se è il primo lettore, blocca l'accesso agli scrittori tramite il semaforo `scrivi`. Quando un lettore termina di leggere, decrementa `numlettori` e, se è l'ultimo lettore, rilascia `scrivi` per permettere agli scrittori di accedere.
- Scrittori quando uno scrittore vuole accedere al file, esegue una `wait(scrivi)` per ottenere l'accesso esclusivo. Dopo aver completato la scrittura, rilascia il semaforo `scrivi` con `signal(scrivi)`.

Domanda 7.2

La soluzione garantisce assenza di deadlock e starvation per lettori e scrittori?
Riuscite a pensare a soluzioni alternative, a partire da quella vista?

Note:-

Questa soluzione è *reader-first*, quindi se arrivano sempre lettori, gli scrittori possono andare in starvation. Esistono anche altre soluzione che possono essere writer-first

7.3 Problema di cinque filosofi

7.3.1 Dati Condivisi

```
semaphore bacchetta[5]; // tutte inizializzate a 1
```

7.3.2 Codice del Filosofo i (Soluzione Errata)

```
do {
    wait(bacchetta[i]);
    wait(bacchetta[(i+1) mod 5]);
    // mangia
    signal(bacchetta[i]);
    signal(bacchetta[(i+1) mod 5]);
    // pensa
} while (true);
```

7.3.3 Problema di Deadlock

Questa soluzione può portare a una situazione di deadlock, in cui tutti i filosofi tengono una bacchetta e aspettano l'altra, bloccandosi a vicenda.

7.3.4 Soluzioni Migliori

Alcune soluzioni possibili per evitare il deadlock includono:

- Consentire a soli 4 filosofi di sedersi a tavola contemporaneamente.
- Prendere le due bacchette solo se entrambe sono disponibili, usando una sezione critica.
- Prelievo asimmetrico delle bacchette, in cui i filosofi prendono le bacchette in un ordine diverso dai loro vicini.

Note:-

Sezione 6.7 Monitori, Capitolo 8 (Approfondire) + Esercizi

Note:-

es. e) Spreca il quanto di tempo; d) Un processo kernel mode, può essere sostituito (scadenza quanto di tempo, scelta della CPU).0

Note:-

Quarto criterio fondamentale della sezione critica: è quello di evitare il busy waiting

8 Stallo dei processi (deadlock)

Note:-
Questo capitolo è **facoltativo**, presente per dare più integrità agli appunti totali

8.1 Definizione

Definizione 8.1.1

Situazione in cui ciascun processo in un insieme di n processi ($n \geqslant 2$) si trova in uno stato di *attesa* per il verificarsi di un evento che solo uno degli aaltri processi dell'insieme può provocare

Il risultato è, chiaramente, una attesa infinita da parte di tutti gli n processi!

8.2 Situazioni simili anche nella realtà

: *When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone*
I SO di oggi non affrontano il problema, ma questo spetta a gli utenti. In futuro chi lo sa potrà diventare un compito dei SO.

8.3 Problema dei nastri

8.3.1 Dati Condivisi

```
semaphore avail = 2; // il semaforo controlla la disponibilità dei nastri
```

8.3.2 Codice dei Processi P1 e P2

```
// Processo P1
begin
    // codice preliminare
    wait(avail); // P1 prende il primo nastro
    // altre operazioni
    wait(avail); // P1 tenta di prendere il secondo nastro
    // utilizza i nastri
    signal(avail); // P1 rilascia il primo nastro
```

```
    signal(avail); // P1 rilascia il secondo nastro
    // codice finale
end

// Processo P2
begin
    // codice preliminare
    wait(avail); // P2 prende il primo nastro
    // altre operazioni
    wait(avail); // P2 tenta di prendere il secondo nastro
    // utilizza i nastri
    signal(avail); // P2 rilascia il primo nastro
    signal(avail); // P2 rilascia il secondo nastro
    // codice finale
end
```

8.3.3 Problema di Deadlock

Questo scenario può portare a un deadlock: se entrambi i processi eseguono il primo `wait(avail)` e occupano ciascuno un nastro, nessuno dei due sarà in grado di eseguire il secondo `wait(avail)` perché il semaforo `avail` è inizializzato a 2. Di conseguenza, entrambi i processi rimarranno bloccati.

8.4 Un ponte ad una sola corsia

Ciascuna posizione di marcia può essere vista come una **risorsa**, una situazione di deadlock può essere risolta se un auto **torna indietro** $\Longrightarrow$ (libera una risorsa già occupata), si verifica **starvation** se ciascuna auto sul ponte attende che l'altra liberi l'unica corsia di marcia

8.5 Modello del sistema

Un sistema (HW + SO) può essere visto come formato da:

- un insieme finito di tipi di risorse R (cicli di CPU, spazio di memoria, device di I/O),
- Ogni tipo di risorsa è formata da un certo numero di istanze indistinguibili fra loro (ad esempio la RAM può essere divisa in porzioni identiche, ciascuna delle quali può ospitare un processo),
- Un insieme di processi P che hanno bisogno di una o più istanze di alcune delle risorse per portare a termine la computazione.

Definizione di Deadlock

Si definisce **deadlock** di un sottoinsieme di processi del sistema $\{P_1, P_2, \ldots, P_n\} \subseteq P$ la situazione in cui ciascuno degli n processi P_i è in attesa del rilascio di una risorsa detenuta da uno degli altri processi del sottoinsieme; si forma cioè una catena circolare per cui:

$$P_1 \text{ aspetta } P_2 \ldots \text{ aspetta } P_n \text{ aspetta } P_1$$

Anche se non tutti i processi del sistema sono bloccati, la situazione non è desiderabile in quanto può bloccare alcune risorse e danneggiare anche i processi non coinvolti nel deadlock.

8.6 Caratterizzazione dei Deadklock

Il SO può avvalersi di una opportuna rappresentazione detta **grafo** di assegnazione delle risorse che in ogni istante registra quali risorse sono assegnate a quale processo, e quali risorse sta **aspettando** ciascun processo.

8.7 Metodi per prevenire dei Deadlock

1. Prevenire o evitare i deadlock, usando un opportuno protocollo di richiesta e assegnamento delle risorse
2. lasciare che il deadlock si verifichi, ma fornire strumenti per la scoperta e il recupero dello stesso, esplorando il grafo di assegnazione delle risorse alla ricerca di cicli.
   -
     Osservazioni 8.7.1

     Tuttavia, la soluzione 1 genera un eccessivo sottoutilizzo delle risorse, mentre la soluzione 2 non evita il problema e richiede lavoro al SO per eliminare il deadlock, dunque i SO moderni adottano la soluzione 3:
3. **Lasciare agli utenti la prevenzione/gestione dei deadlock**

9 Memoria centrale

9.1 Introduzione

Abbiamo visto che i moderni SO tentano di massimizzare l'uso delle risorse della macchina, e in primo luogo l'utilizzo della CPU.

- Questo si ottiene mediante le due tecniche fondamentali del multi-tasking e del time-sharing, che richiedono di tenere in memoria primaria contemporaneamente più processi attivi.
- Il SO deve decidere come allocare lo spazio di RAM tra i processi attivi, in modo che ciascun processo sia pronto per sfruttare la CPU quando gli viene assegnata.

Supponiamo però che, ad un certo punto, la RAM sia **completamente** occupata da 3 processi utente, P1, P2, P3 (per semplicità assumiamo che a tutti i processi venga assegnata una porzione di RAM della stessa dimensione).

Domanda 9.1

Un nuovo processo P4 viene fatto partire, è immediatamente pronto per usare la CPU, ma non c'è più spazio per caricare il suo codice in RAM, che si può fare?

- Ovviamente si potrebbe aspettare la terminazione di uno dei 3 processi già in RAM, ma supponiamo che uno dei tre processi (diciamo P2) sia temporaneamente in attesa di compiere una lunga operazione di I/O (per cui non userà la CPU a breve).

Il SO potrebbe decidere di spostare temporaneamente P2 sull'hard disk per far posto a P4, che così può concorrere all'uso della CPU.

Definizione 9.1.1: Swapping

- Che cosa viene spostato sull'hard disk? L'immagine di P2: il codice (anche se, come capiremo meglio più avanti, questo si può anche evitare), i dati e lo stack del processo.
- Dopo un po' P1 termina e libera una porzione di RAM. Il SO potrebbe riportare P2 in RAM (ma ora nello spazio che era stato inizialmente assegnato a P1).

Questa tecnica viene chiamata *swapping* (avvicendamento di processi). L'area del disco in cui il SO copia temporaneamente un processo viene detta area di *swap*.

Domanda 9.2

Lo *swapping* è raramente usato nei moderni sistemi operativi perché troppo inefficiente, ma l'esempio mette in luce un problema fondamentale nella gestione della memoria primaria: P2 contiene istruzioni che usano indirizzi di memoria primaria: funziona ancora correttamente quando viene spostato da un'area di RAM ad un'altra?

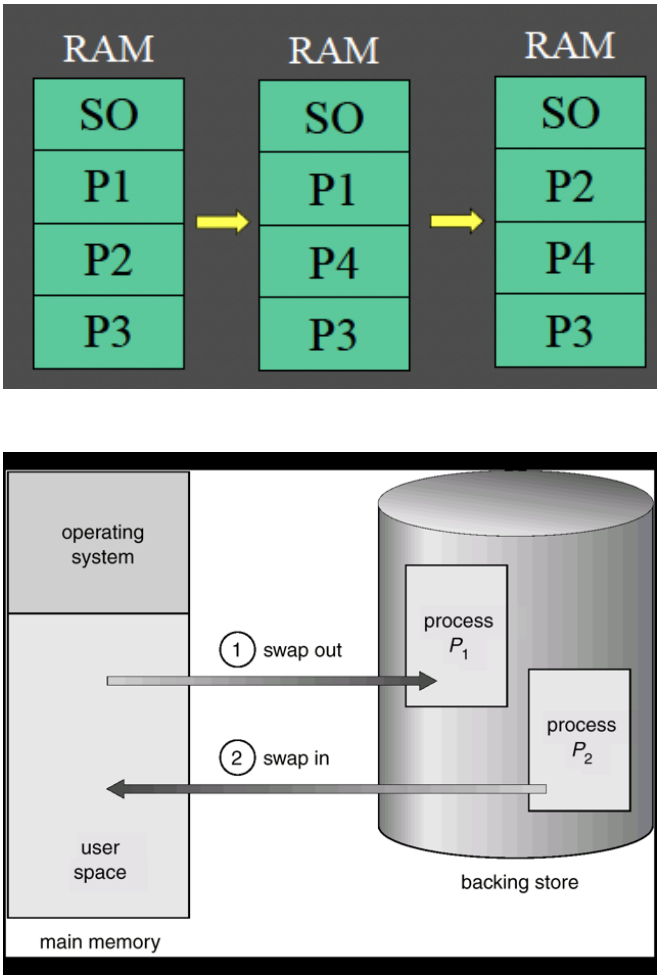


Figure 9.1: Swap mem

Perché un programma possa essere **eseguito**, il suo codice deve trovarsi in **memoria primaria** (ma rivedremo questa affermazione quando parleremo della memoria virtuale) Quindi, quando il SO riceve il **comando** di esecuzione di un programma, deve recuperare il codice del programma dalla memoria secondaria, e decidere in quale porzione della memoria primaria sistemarlo. ossia, a partire da quale indirizzo di RAM.

9.2 Binding (associazione degli indirizzi)

Un programma sorgente usa (tra l'altro) dati (variabili) e istruzioni di controllo del flusso di computazione.

- Quando il programma viene compilato e caricato in Memoria Primaria (MP) per essere eseguito, ad ogni variabile è associato l'**indirizzo** di una locazione di memoria che ne contiene il valore.
- Alle istruzioni di controllo del flusso di esecuzione del programma (ossia i salti condizionati e incondizionati) è associato l'indirizzo di destinazione del salto.
- L'operazione di associazione di variabili e istruzioni agli indirizzi di memoria è detta *binding degli indirizzi*.

In altre parole, ad ogni variabile dichiarata nel programma viene fatto corrispondere l'indirizzo di una cella di memoria di RAM in cui verrà memorizzato il valore di quella variabile.

- L'accesso alla variabile, in lettura e scrittura, corrisponde alla lettura e scrittura della cella di memoria il cui indirizzo è stato "legato" (con l'operazione di binding) alla variabile.
- Le istruzioni di salto, che permettono di implementare costrutti come *if-then-else*, *while*, ecc., sono associate agli indirizzi in RAM dove si trova l'istruzione con cui prosegue l'esecuzione del programma se il salto viene eseguito.

Ad esempio, un'istruzione C come:

```
counter = counter + 1;
```

alla fine diventerà qualcosa del tipo:

```
load(R1, 10456)
Add(R1, #1);
store(R1, 10456)
```

10456 è l'indirizzo della cella di memoria che contiene il valore della variabile *counter*. L'indirizzo 10456 è stato associato alla variabile *counter* durante la fase di binding degli indirizzi.

Analogamente, un'istruzione C come:

```
while (counter <= 100) counter++;
```

alla fine diventerà qualcosa del tipo:

```
100FC jgt(R1, #100, 10110) // jump if greater than
10100 load(R1, 10456)
10104 Add(R1, #1)
10108 store(R1, 10456)
1010C jmp(100FC)
10110 ... ...
```

Rispetto all'indirizzo di istruzione del salto stesso, il *while* della slide precedente potrebbe anche essere tradotto in assembler così:

```
100FC jgt(R1, #100, 00014) // jump if greater than
10100 load(R1, 10456)
10104 Add(R1, #1)
10108 store(R1, 10456)
1010C jmp(100FC)
10110 ... ...
```

Perché un programma sorgente possa essere eseguito deve passare attraverso varie fasi. Il binding degli indirizzi avviene in una di queste fasi:

- compilazione
- caricamento (in RAM)
- esecuzione

9.2.1 Quando?

1. In fase di Compilazione
   - viene generato codice assoluto o statico.
   - Il compilatore deve conoscere l'indirizzo della cella di RAM a partire dal quale verrà caricato il programma, in modo da effettuare il *binding* degli indirizzi (che avviene, appunto, in fase di compilazione).

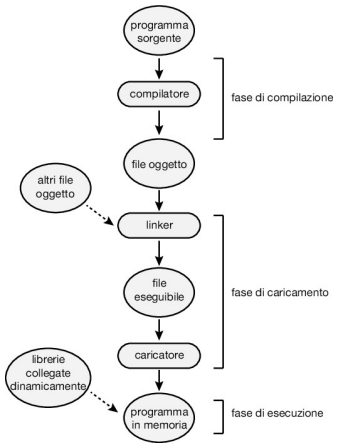


Figure 9.2: Processo di compilazione da programma sorgente

- Se il SO deve scaricare temporaneamente il processo che usa quel codice in Memoria Secondaria (MS), come nell'esempio visto a inizio capitolo, quando lo ricarica in RAM deve rimetterlo esattamente dove si trovava prima. (Oppure?)

2. In fase di caricamento in RAM

   - Viene generato codice staticamente rilocabile.
   - Il compilatore associa ad istruzioni e variabili degli indirizzi relativi rispetto all'inizio del programma, che inizia da un ipotetico indirizzo 0 virtuale.
   - Gli indirizzi assoluti finali vengono generati in fase di caricamento del codice in Memoria Primaria (MP) in base all'indirizzo di MP a partire dal quale è caricato il codice.
   - Il binding degli indirizzi, quindi, avviene in fase di caricamento del programma in RAM: se il processo che usa quel codice viene tolto dalla RAM, si può caricarlo in una posizione diversa solo rieffettuando la fase di caricamento (ma è più efficiente che ricompilare tutto).

3. In fase di esecuzione

   - Viene generato codice dinamicamente rilocabile.
   - Il codice in esecuzione usa sempre e solo indirizzi relativi.
   - La trasformazione di un indirizzo relativo in uno assoluto viene fatta nell'istante in cui viene eseguita l'istruzione che usa quell'indirizzo.
   - È necessario un opportuno supporto hardware per realizzare questo metodo senza perdita di efficienza.
   - Si parla di *binding dinamico* degli indirizzi.
   - In opportuno registro di rilocazione viene usato per trasformare un indirizzo relativo nel corrispondente indirizzo assoluto durante l'esecuzione delle istruzioni.
   - Il registro di rilocazione contiene l'indirizzo di partenza dell'area di RAM in cui è caricato il programma in esecuzione.
   - La memory management Unit (MMU) si occuperà di trasformare gli indirizzi relativi in assoluti, usando il registro di rilocazione, per accedere alle celle di RAM indirizzate dalle istruzioni
   - Lo spostamento del processo da un area all'altra della MP è realizzabile senza problema.
   - Il SO deve solo ricordarsi dell'indirizzo della locazione di MP a partire dalla quale è memorizzato il processo

Note:-

Per spostare i programmi da un'area di RAM ad un'altra ora basta cambiare l'indirizzo scritto nel registro di rilocazione (fig. 9.5 modificata)

9.3 Spazio degli indirizzi (Logici e Fisici)

Consideriamo codice dinamicamente rilocabile (d'ora in poi faremo sempre riferimento a codice dinamicamente rilocabile, se non indicato diversamente). Ogni indirizzo usato nel codice è riferito ad un ipotetico indirizzo 0 (zero): l'indirizzo della prima istruzione di cui è formato il codice.

Gli indirizzi utilizzati in un programma possono essere:

- l'indirizzo di una cella di memoria che contiene una variabile
- l'indirizzo di un'istruzione di salto

Questi indirizzi rientrano nello **spazio di indirizzamento logico o virtuale**, che va da 0 all'ultima cella di memoria occupata. Quando il codice viene caricato in RAM, gli **indirizzi logici** generati dalla CPU vengono trasformati in **indirizzi fisici** attraverso il registro di rilocazione, permettendo di indirizzare correttamente la memoria fisica (RAM).

Lo **spazio di indirizzamento fisico** è l'insieme degli indirizzi fisici che dipende dall'area di memoria dove il sistema operativo ha caricato il programma.

Per i programmi con codice rilocabile dinamicamente esistono due tipi di indirizzi:

- Indirizzi logici, che vanno da 0 a max
- Indirizzi fisici, che vanno da $r + 0$ a $r + max$, dove r è l'indirizzo iniziale della RAM in cui il programma è caricato

Gli indirizzi logici vengono sempre mappati in indirizzi fisici per accedere correttamente alla RAM.

Le espressioni **spazio di indirizzamento logico** e **spazio di indirizzamento fisico** si riferiscono principalmente all'architettura di un sistema, non a singoli programmi.

Consideriamo un computer con un massimo di 64 Kbyte di RAM, ovvero 65536 byte. In questo contesto, possiamo dire che:

1. Il computer può indirizzare 2^{16} byte di RAM.
2. Gli indirizzi dei byte della RAM vanno da 0000 a FFFF in esadecimale (da 0 a $2^{16} - 1$).
3. L'indirizzo di ciascun byte della RAM è rappresentato da 16 bit.

Pertanto, lo **spazio di indirizzamento fisico** di questo computer è scritto su 16 bit e va da 0000 a FFFF, con una dimensione di 64 Kbyte.

Se un compilatore genera codice dinamicamente rilocabile e utilizza 12 bit (**perchè 12?**) per scrivere un indirizzo logico, lo **spazio di indirizzamento logico** di un programma sarà di massimo 2^{12} byte, ovvero 4 Kbyte. Nessun programma potrà superare questo limite, anche se può usare uno spazio logico inferiore.

Quindi, possiamo dire che lo spazio di indirizzamento logico dei programmi di questo computer è scritto su 12 bit, va da 0000 a 0FFF (esadecimale) ed è di 4 Kbyte.

In seguito, quando parleremo di spazi di indirizzamento, ci riferiremo a quelli dell'intera macchina, e non a quelli dei singoli programmi. Tuttavia, è possibile considerare un programma che occupa tutto lo spazio di indirizzamento logico della macchina.

Domanda 9.3

Ha senso che la dimensione dello spazio di indirizzamento fisico sia diversa da quella dello spazio di indirizzamento logico in un sistema reale?

In effetti, è comune che lo **spazio di indirizzamento fisico** e lo **spazio di indirizzamento virtuale** siano diversi. Nei processori moderni a 64 bit, lo spazio di indirizzamento fisico può variare da 2^{40} a 2^{64} byte (da 40 a 64 bit per gli indirizzi fisici). Tuttavia, non si usano sempre 64 bit per gli indirizzi fisici perché sono eccessivi, e un computer raramente ha una quantità di RAM pari al massimo indirizzabile dal processore (ad esempio, 2^{40} byte = 1 Terabyte = 1000 Gigabyte).

Sistemi operativi e applicazioni adottano spazi di indirizzamento virtuali che variano tipicamente da 2^{48} a 2^{64} byte, ovvero da 48 a 64 bit per gli indirizzi virtuali.

In generale, per i computer moderni vale la relazione:

$$|\text{RAM}| \neq |\text{spazio di indirizzamento fisico}| \neq |\text{spazio di indirizzamento virtuale}|$$

E di solito:

$$|\text{RAM}| < |\text{spazio di indirizzamento fisico}| < |\text{spazio di indirizzamento virtuale}|$$

In molti casi, per vincoli architetturali e dimensionali, la quantità effettiva di RAM di un computer è molto inferiore allo spazio di indirizzamento fisico. Pertanto, è spesso vero che:

$$|\text{RAM}|_{\text{effettiva}} \leqslant |\text{RAM}|_{\text{massima}} \ll |\text{spazio fisico}| < |\text{spazio virtuale}|$$

Osservazioni 9.3.1

Ad esempio, nei processori Intel Core i7 lo spazio di indirizzamento fisico è scritto su 52 bit, mentre quello virtuale è su 48 bit, quindi può capitare che:

$$|\text{spazio fisico}| > |\text{spazio virtuale}|$$

Domanda 9.4

Se un sistema ha uno spazio di indirizzamento virtuale di X byte, significa che possiamo scrivere un programma che occupa al massimo X byte, cioè usa indirizzi virtuali da 0 a $X-1$. Tuttavia, un programma può girare su una macchina in cui:

$$|\text{RAM}| < |\text{spazio fisico}| < X?$$

Questo aspetto sarà approfondito nel capitolo sulla **memoria virtuale**.

9.4 Le librerie

Definizione 9.4.1

Una **libreria** è una collezione di subroutine di uso comune messe a disposizione dei programmatori per lo sviluppo software. Ad esempio, la libreria matematica del C fornisce funzioni come `sqrt(x)` per calcolare la radice quadrata.

Le librerie sono utili perché permettono di riutilizzare codice già esistente, evitando ai programmatori di doverlo riscrivere ogni volta. Sebbene "libreria" sia una traduzione impropria di "library", il termine è ormai comunemente accettato.

9.4.1 Tipi di Librerie

Esistono principalmente due tipi di librerie:

1. **Librerie statiche**: le subroutine sono collegate al programma principale durante la fase di compilazione o di caricamento, diventando parte dell'eseguibile. Tuttavia, ciò può portare a duplicazione di codice, sia su disco che in RAM, soprattutto se più programmi usano la stessa libreria. Inoltre, il codice di una libreria statica viene caricato in RAM anche se non viene utilizzato durante l'esecuzione del programma.

2. **Librerie dinamiche**: vengono caricate in RAM solo al momento in cui il programma chiama una subroutine specifica, ossia a **run-time**. Il programma specifica solo il nome della subroutine, e il sistema operativo carica la libreria nello spazio di memoria assegnato al processo. Queste librerie sono anche dette **librerie condivise**, perché possono essere utilizzate da più processi contemporaneamente, evitando la duplicazione di codice in RAM. Inoltre, le versioni aggiornate delle librerie dinamiche possono sostituire le vecchie senza dover ricompilare i programmi che le utilizzano.

9.4.2 Estensioni delle Librerie Dinamiche

In Unix, Linux e Solaris, le librerie dinamiche hanno estensione `.so` (shared object) e si trovano solitamente nella directory `/lib`. In ambiente Windows, le librerie dinamiche hanno estensione `.DLL` (Dynamic Link Library) e si trovano nella cartella `C:\WINDOWS\system32`.

9.5 Tecniche di gestione della memoria primaria

Le principali tecniche di gestione della **Memoria Principale (MP)** vanno dalle più semplici alle più complesse. Alcune di queste tecniche sono ormai obsolete, ma aiutano a comprendere concetti più avanzati. Le tecniche includono:

- **Swapping**
- **Allocazione contigua a partizioni multiple fisse**
- **Allocazione contigua a partizioni multiple variabili**
- **Paginazione**
- **Paginazione a più livelli**

Swapping

Definizione 9.5.1

Lo **swapping** consiste nel salvare in memoria secondaria (hard disk) l'immagine di un processo non in esecuzione (*swap out*) e ricaricarla in MP (*swap in*) prima di assegnarle la CPU.

Questa tecnica permette di attivare più processi di quanti la sola MP possa contenere, utilizzando un'area dell'hard disk chiamata **area di swap**, riservata al sistema operativo. Tuttavia, se il processo viene ricaricato in una diversa area di MP, è necessario utilizzare codice dinamicamente rilocabile.

Problemi dello Swapping

Il principale problema dello swapping è il tempo impiegato per copiare il codice e i dati di un processo tra l'hard disk e la RAM, che è nell'ordine dei millisecondi. Poiché in un millisecondo un singolo core di una moderna CPU può eseguire milioni di istruzioni, l'overhead di tempo risultante dallo swapping è generalmente considerato inaccettabile.

Di conseguenza, lo **swapping di interi processi** è ormai caduto in disuso nei moderni sistemi operativi, salvo rare eccezioni.

L'Idea di Fondo dello Swapping

Nonostante l'obsolescenza dello swapping, l'idea di fondo rimane valida: utilizzare parte della memoria secondaria per estendere la memoria primaria, permettendo l'esecuzione di un numero maggiore di processi rispetto a quanto potrebbe ospitare la sola RAM. Questa idea sarà ripresa nel capitolo sulla **memoria virtuale**.

9.6 Allocazione contigua della Memoria Primaria

In un computer, la **Memoria Principale (MP)** è solitamente divisa in due partizioni:

- una per il **Sistema Operativo (SO)**
- una per i **processi utente**.

Il sistema operativo si posiziona nella stessa area di memoria puntata dal **vettore delle interruzioni**, che è spesso allocato nella parte bassa della memoria.

Protezione della memoria

Nei sistemi operativi più semplici (ad esempio MS-DOS), l'area non assegnata al SO viene occupata da un solo processo. La protezione della MP consiste nella protezione delle aree di memoria del SO.

Registro Limite e Registro di Rilocazione

Il **registro limite** è inizializzato dal SO e garantisce che ogni indirizzo logico usato dal processo utente sia inferiore al valore scritto nel registro. Poiché si utilizza codice dinamicamente rilocabile, il **registro di rilocazione** viene usato per trasformare l'indirizzo logico in indirizzo fisico.

- **Registro di rilocazione**: 100.040
- **Registro limite**: 74.600

Gli indirizzi fisici validi vanno da 100.040 a 174.640 (vedi Fig. 9.3).

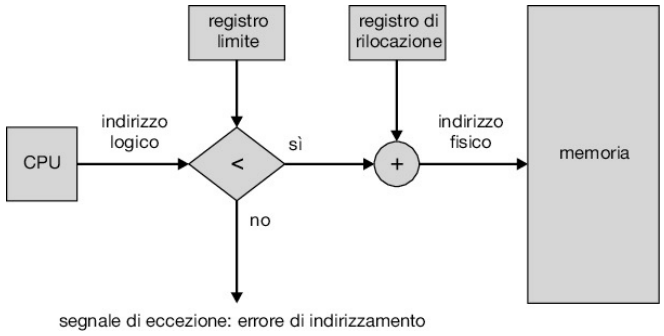


Figure 9.3: Protezione memoria

9.6.1 Allocazione a partizioni multiple fisse

Nell'allocazione a partizioni fisse:

- La memoria è divisa in **partizioni di dimensione fissa**, che non devono necessariamente essere tutte uguali.
- Ogni partizione contiene un **unico processo** dall'inizio alla fine della sua esecuzione.
- Il **grado di multiprogrammazione** è determinato dal numero di partizioni.
- Quando un processo termina, la partizione può essere occupata da un altro processo.

Il meccanismo di **registri limite e di rilocazione** protegge le partizioni da accessi non autorizzati. Durante il *context switch*, il **dispatcher** carica:

- Nel registro di rilocazione, l'indirizzo di partenza della partizione assegnata al processo.
- Nel registro limite, la dimensione della partizione.

Questa tecnica, utilizzata nel **IBM OS/360**, richiede CPU dotate di registri di rilocazione e limite, ma non è più utilizzata nei sistemi operativi moderni per i seguenti svantaggi:

- Il grado di multiprogrammazione è limitato dal numero di partizioni disponibili.
- Si verifica **frammentazione interna**, dove una parte della partizione rimane inutilizzata se il processo è più piccolo della partizione stessa.
- Si può verificare **frammentazione esterna**, quando lo spazio libero disponibile è frammentato in più aree non contigue e quindi non utilizzabile per un nuovo processo di dimensione maggiore.

- L'arrivo di un processo più grande della partizione più grande non può essere gestito.

Frammentazione interna:

- Parte dello spazio di memoria di una partizione viene sprecato se il processo è più piccolo della partizione stessa.

Frammentazione esterna:

- Se la memoria libera è frammentata in più blocchi non contigui, pur avendo spazio sufficiente in totale, non può essere utilizzata per allocare nuovi processi.

L'allocazione a partizioni fisse ha anche altri problemi:

Domanda 9.5

Che succede se arriva un processo più grande della partizione più grande?

Osservazioni 9.6.1

Notate che se si aumenta la dimensione media delle partizioni, aumenta anche la frammentazione interna, e diminuisce il grado di multiprogrammazione

9.7 Allocazione a partizioni multipli variabili

Nell'allocazione a partizioni variabili:

- Ogni processo riceve una quantità di memoria esattamente pari alla sua dimensione.
- Quando un processo termina, lascia un "buco" in RAM che può essere occupato da un altro processo.
- Tuttavia, nel tempo si creano **buchi sparsi e più piccoli**, rendendo difficile l'allocazione di nuovi processi.

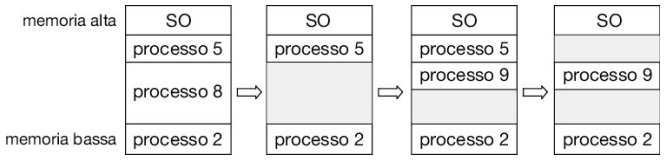


Figure 9.4: Buchi Ram

Il sistema operativo (SO) deve:

- Tenere traccia delle aree di memoria libere e occupate.
- Aggiornare continuamente le informazioni quando un processo nasce o termina.
- Assegnare una partizione sufficientemente grande quando un nuovo processo deve essere caricato.

Strategie di Allocazione

Esistono diverse strategie per scegliere quale partizione assegnare a un processo:

- **First Fit**: seleziona la prima partizione abbastanza grande da ospitare il processo.
- **Best Fit**: seleziona la partizione più piccola che può contenere il processo.
- **Worst Fit**: seleziona la partizione più grande disponibile.

Osservazioni sperimentali:

- La strategia **Worst Fit** tende a funzionare peggio in termini di utilizzo della memoria, poiché lascia spazi grandi frammentati.
- Le strategie **Best Fit** e **First Fit** hanno prestazioni simili, ma si preferisce **First Fit** poiché è più veloce, dato che interrompe la ricerca al primo spazio sufficiente.

9.7.1 La frammentazione

Nel tempo, l'allocazione a partizioni variabili può portare alla formazione di piccoli **buchi non contigui** in RAM:

- Circa **1/3 a 1/2 della memoria principale** (MP) può essere sprecato a causa della **frammentazione esterna**, ossia la presenza di buchi di memoria troppo piccoli per ospitare un processo.
- Esiste anche il problema della **frammentazione interna**, dovuto all'impossibilità di tenere traccia di buchi molto piccoli che vengono quindi aggregati a partizioni adiacenti, causando uno **spreco nascosto**.

Compattazione della Memoria

Una tecnica per recuperare la memoria inutilizzata è la **compattazione**:

- Spostare le partizioni occupate dai processi in modo che siano tutte **contigue**, liberando un unico grande buco libero.
- La compattazione richiede la **rilocazione dinamica** del codice e dei dati dei processi.
- Questo processo è **costoso in termini di tempo** e durante la compattazione il sistema non è utilizzabile.

9.8 Paginazione della memoria

Definizione 9.8.1

L'allocazione contigua della memoria principale presenta quindi diversi problemi. L'alternativa è ammettere che l'area di memoria allocata ad un processo possa essere in realtà suddivisa in tanti pezzi non contigui fra loro Se tutti i "pezzi" hanno la stessa dimensione allora il termine esatto per indicare questa tecnica è: paginazione della memoria (primaria)

9.8.1 Metodo di base

La **Memoria Primaria** (o lo **spazio di indirizzamento fisico**) è suddivisa in unità dette **frame** (o pagine fisiche), con le seguenti caratteristiche:

- I **frame** sono tutti della stessa dimensione, che è sempre una **potenza di due** (ad esempio: 512, 1024, 2048, fino a 8192 byte).
- Lo spazio di indirizzamento fisico del processo è visto come un **unico spazio contiguo** di indirizzi, ma nella realtà è suddiviso in **pagine logiche**, ciascuna di dimensione uguale ai frame fisici.

Per eseguire un processo con **x pagine**:

- Il Sistema Operativo (**SO**) cerca **x frame** liberi in cui caricare le pagine del processo. Questi frame non devono essere adiacenti e le pagine possono essere caricate in un ordine qualsiasi.
- Ogni processo ha una propria **Tabella delle Pagine** (o **Page Table**, **PT**): un array le cui *entry* contengono i numeri dei frame in cui le pagine del processo sono state caricate.
- Il SO tiene traccia dei **frame liberi** nella memoria primaria, utilizzandoli per memorizzare le pagine di nuovi processi.

Struttura della Tabella delle Pagine

- Ogni *entry* della Page Table rappresenta una pagina del processo.
- L'indice di ciascuna entry corrisponde al numero della pagina, mentre il valore dell'entry contiene il numero del frame dove è stata memorizzata la pagina.

Problema degli Indirizzi Virtuali

Gli **indirizzi relativi (o virtuali)** del programma, una volta caricati in RAM, non funzionano più come indirizzi lineari contigui. Ad esempio:

- Un'istruzione come `jmp_if_odd R1,C` deve saltare a un indirizzo specifico (ad es. 0004), ma in RAM non esiste più un punto lineare di partenza.
- La soluzione è **riconsiderare** gli indirizzi virtuali, non più come indirizzi lineari, ma come indirizzi all'interno della Page Table, utilizzando **conversioni da indirizzi logici a indirizzi fisici**.

  Indirizzi Logici e Fisici, vedere registrazione 25/10/2024 min Circa 30

Paginazione: metodo di base

Gli indirizzi logici diventano delle coppie di valori, in cui:

- il primo elemento della coppia specifica il numero della pagina all'interno della quale si trova la cella di memoria che vogliamo indirizzare;
- il secondo elemento specifica la posizione (o offset) della cella di memoria che vogliamo indirizzare rispetto ad un ipotetico indirizzo 0 (zero), ovvero l'indirizzo del primo byte della pagina specificata dal primo elemento della coppia.

Quindi un indirizzo logico assume la forma: (pagina, offset).

Note:-

Come vedremo più avanti, sotto opportune condizioni gli indirizzi logici lineari e gli indirizzi logici specificati come coppie di valori coincidono.

Un indirizzo logico viene tradotto in uno fisico secondo il seguente processo:

- Il numero di pagina p viene usato come indice nella *page table* del processo per individuare il frame f in cui è contenuta la pagina.
- Una volta noto il frame f che contiene la pagina p, l'offset d (dove "d" sta per *displacement*) può essere applicato a partire dall'inizio del frame per indirizzare il byte specificato dalla coppia (p, d).

Vediamo più in dettaglio: ogni informazione all'interno del computer, incluso un indirizzo logico, è in definitiva una sequenza di bit. Ad esempio: 001100010101.

Abbiamo deciso di dividere un indirizzo logico in due parti. Se abbiamo a disposizione 12 bit in tutto, dobbiamo scegliere quanti usare per specificare il numero della pagina e quanti per l'offset all'interno della pagina.

Ad esempio, possiamo decidere di utilizzare 4 bit per rappresentare il numero della pagina e i restanti 8 bit per l'offset. In questo caso, l'indirizzo logico 001100010101 rappresenta:

- pagina: 3
- offset: 21

La scelta del numero di bit da utilizzare per scrivere il numero di pagina e l'offset dipende dall'hardware su cui dovrà girare il sistema operativo, il quale impone:

- il numero di bit su cui va scritto un indirizzo logico (ad esempio m bit);
- la dimensione di un frame, e quindi di una pagina (ad esempio 2^n byte).

Di conseguenza, dobbiamo utilizzare n bit per rappresentare l'offset all'interno di una pagina/frame, e il numero di bit usati per rappresentare il numero di pagina sarà pari a $m - n$ bit.

Note:-

A questo punto, la dimensione dello spazio di indirizzamento logico è stabilita come $2^{(m-n)} \times 2^n$ byte.

Esempio 9.8.1 (Esempio di Spazio di Indirizzamento Logico)

Consideriamo una macchina in cui i frame hanno dimensione 2^{12} byte, ovvero 4096 byte. Di conseguenza, anche la dimensione di una pagina nello spazio di indirizzamento logico del sistema sarà di 4096 byte (quindi $n = 12$).
Se la macchina mette a disposizione $m = 22$ bit per rappresentare un indirizzo logico, allora il numero di bit necessari per il numero di pagina sarà $m - n = 10$ bit.
In questo caso, lo spazio di indirizzamento logico della macchina risulterà pari a:

$$(2^{10} \text{ pagine}) \times (2^{12} \text{ byte}) = 4 \text{ megabyte}$$

Struttura di un Indirizzo Logico

- **Numero di Pagina (p)**: $m - n$ bit
- **Offset di Pagina (d)**: n bit

Note:-

Qui possiamo osservare uno dei tanti casi di interazione tra il sistema operativo e l'hardware sottostante. È infatti l'hardware a decidere la dimensione dei frame e il numero di bit su cui rappresentare un indirizzo logico. Il sistema operativo, quindi, deve adeguarsi a queste impostazioni.
Questa configurazione consente una traduzione efficiente degli indirizzi logici in indirizzi fisici direttamente a livello hardware. Alcune parti di un sistema operativo sono infatti progettate in base allo specifico hardware su cui girerà il sistema, al fine di sfruttare al meglio le caratteristiche hardware e minimizzare l'overhead introdotto.

Osservazioni 9.8.1 Scelta della Dimensione dei Frame

Nei processori moderni, è possibile scegliere tra vari valori per la dimensione dei frame, e questa scelta, una volta effettuata, rimane fissa. Ad esempio:

- I processori ARM, utilizzati negli iPhone e negli iPad, permettono di scegliere tra 4KB, 16KB, 64KB, 1MB e 16MB come dimensione di un frame.
- La famiglia dei processori Intel, dal Pentium fino ai Core i9, consente di scegliere tra 4KB e 4MB.
- Gli Intel Itanium-2 offrono una gamma più ampia, con opzioni tra 4KB, 8KB, 64KB, 256KB, 1MB, 4MB, 16MB e 256MB.

Definizione 9.8.2: Traduzione degli Indirizzi Logici in Indirizzi Fisici

Definiamo più precisamente l'operazione di traduzione da indirizzi logici a indirizzi fisici. Un indirizzo logico è formato da due componenti (p, d):

- **Numero di Pagina (p)**: utilizzato come indice per selezionare la entry corrispondente nella *Page Table*, che contiene il numero del frame in cui è caricata la pagina.
- **Offset di Pagina (d)**: utilizzato all'interno del frame individuato al passo precedente per localizzare il byte specificato dall'indirizzo logico.

Paginazione: traduzione degli indirizzi

Possiamo ora riassumere e integrare quanto detto finora, osservando che:

1. Gli indirizzi logici nello spazio di indirizzamento logico possono essere interpretati come valori lineari oppure come coppie (pagina, offset).
2. Ciascuna entry di una tabella delle pagine può contenere il numero del frame o, alternativamente, l'indirizzo di partenza del frame. Tuttavia, per ragioni pratiche, viene scelta la prima opzione: memorizzare il numero del frame.

Note:-

Ecco quindi la doppia natura degli indirizzi logici, che sono contemporaneamente valori lineari e coppie (pagina, offset).

Consideriamo uno spazio di indirizzamento logico di dimensione 2^m byte, e assumiamo che ogni pagina abbia una dimensione pari a 2^n byte. Un indirizzo logico lineare scritto su m bit può quindi essere scomposto in:

- i bit più significativi $(m - n)$, che indicano il numero di pagina p;
- i bit meno significativi n, che rappresentano l'offset d.

Struttura di un Indirizzo Logico

- **Numero di Pagina (p)**: $(m - n)$ bit
- **Offset di Pagina (d)**: n bit

Esempio 9.8.2 (Esempio di Spazio di Indirizzamento Logico)

Supponiamo di avere uno spazio di indirizzamento logico di 16 byte, quindi gli indirizzi logici sono rappresentati con $m = 4$ bit, poiché $2^4 = 16$. Ogni pagina ha dimensione $2^2 = 4$ byte, con $n = 2$.
Di conseguenza, gli indirizzi logici vanno da 0000 a 1111, mentre il programma che occupa questo spazio è costituito dalle istruzioni/dati etichettati come "a, b, c, d, e, f, g, h, i, l, m, n" che coprono tre pagine, dove ciascuna istruzione/dato occupa un byte.

Note:-

In questo esempio, lo spazio di indirizzamento logico del programma è suddiviso in pagine. Gli indirizzi di istruzioni e dati, che costituiscono lo spazio di indirizzamento logico, sono quindi valori consecutivi che vanno da 0000 (prima istruzione) a 1011 (ultima istruzione).

Osserviamo come gli $(m - n)$ bit più significativi di ciascun indirizzo rappresentano il numero di pagina, mentre i restanti n bit meno significativi indicano l'offset all'interno della pagina.

Indirizzo Logico	Contenuto
00 00	a
00 01	b
00 10	c
00 11	d
01 00	e
01 01	f
01 10	g
01 11	h
10 00	i
10 01	l
10 10	m
10 11	n

Ogni indirizzo è quindi composto da:

- I primi due bit, che rappresentano il numero di pagina:
  - 00 per la pagina 0
  - 01 per la pagina 1
  - 10 per la pagina 2
- Gli ultimi due bit, che rappresentano l'offset all'interno della pagina.

Note:-

Supponiamo ora di caricare il programma in RAM. Utilizziamo 5 bit per rappresentare un indirizzo fisico, quindi lo spazio di indirizzamento fisico ha dimensione pari a $2^5 = 32$ byte. Tuttavia, la nostra macchina è dotata di soli 20 byte di RAM, suddivisi in 5 frame, ciascuno con un indirizzo che va da 00000 a 10011.

Esempio 9.8.3 (Esempio di Traduzione Indirizzo Logico in Indirizzo Fisico)

Consideriamo l'indirizzo logico 0010. La sua conversione in un indirizzo fisico procede così:

1. **Numero di Pagina**: il numero di pagina (0) è utilizzato per accedere alla *Page Table*, che ci indica che la pagina 0 è caricata nel frame 4.
2. **Offset**: l'offset di 2 bit (10) viene applicato a partire dall'indirizzo iniziale del frame 4.

Di conseguenza, l'indirizzo fisico risultante è 10010.

Osservazioni 9.8.2 Struttura di un Indirizzo Fisico

Notiamo un aspetto importante: nei nostri indirizzi fisici, i bit più significativi (5 - 2 = 3 bit) indicano il numero di frame corrispondente. Ad esempio, per l'indirizzo 10010, i primi 3 bit 100 identificano il frame 4.

Note:-

Le pagine e i frame sono configurati per avere una dimensione $|P|$ pari a una potenza di 2, ossia $|P| = 2^n$. Grazie a questa proprietà, l'offset all'interno di ogni pagina o frame varia da una configurazione di tutti zeri a una di tutti uno:

$$\text{Offset valido: } 00\ldots00 \text{ a } 11\ldots11.$$

Di conseguenza, l'indirizzo di partenza di ogni pagina o frame richiede che gli n bit meno significativi siano impostati a 0.

Osservazioni 9.8.3 Struttura degli Indirizzi di Partenza

Gli indirizzi logici di inizio pagina nello spazio di indirizzamento di un programma avranno la forma seguente:

$$\begin{aligned} &00\ldots00\ 00\ldots00 && \text{inizio della pagina 0}\\ &00\ldots01\ 00\ldots00 && \text{inizio della pagina 1}\\ &00\ldots10\ 00\ldots00 && \text{inizio della pagina 2}\\ &00\ldots11\ 00\ldots00 && \text{inizio della pagina 3} \end{aligned}$$

Lo stesso vale per i frame nello spazio di indirizzamento fisico.

Note:-

Se rimuoviamo gli n bit meno significativi da un indirizzo logico di m bit, i rimanenti $m - n$ bit contano semplicemente le pagine (in binario) in cui è suddiviso lo spazio di indirizzamento logico. Ad esempio:

$$\begin{aligned} 00\ldots00 &= \text{pagina 0}\\ 00\ldots01 &= \text{pagina 1}\\ 00\ldots10 &= \text{pagina 2}\\ 00\ldots11 &= \text{pagina 3} \end{aligned}$$

Questa logica vale anche per i frame nello spazio fisico, utilizzando un numero di bit adeguato per rappresentare i frame.

Esempio 9.8.4 (Ricostruzione dell'Indirizzo Fisico)

Supponiamo di avere un indirizzo logico in cui il numero di pagina e l'offset sono rappresentati come segue:

$$\text{Numero di pagina } = 001010, \quad \text{Frame } = 1110001$$

$$\text{Offset} = 101010$$

Allora, l'indirizzo logico diventa 001010101010, mentre l'indirizzo fisico corrispondente sarà 1110001101010.

Note:-

La conversione da indirizzo logico a fisico si basa sull'operazione di somma dell'indirizzo base del frame con l'offset, ad esempio:

$$\begin{aligned} &\text{Indirizzo base del frame: } 1110001000000 \\ &\text{Offset: } 101010 \end{aligned}$$

Somma:

$$1110001000000 + 101010 = 1110001101010$$

Osservazioni 9.8.4 Efficienza della Traduzione

Se le dimensioni di pagine e frame sono potenze di due:

1. Non è necessario effettuare la somma tra l'indirizzo base del frame e l'offset, risparmiando tempo di calcolo.
2. Non è necessario memorizzare l'indirizzo di base in ogni entry della page table, ma solo il numero del frame, poiché gli n bit meno significativi dell'indirizzo di partenza sono tutti a 0, con un risparmio di spazio.

Generare un indirizzo fisico diventa quindi un'operazione di concatenazione veloce tra il numero di frame e l'offset, eseguibile direttamente dall'hardware.

Note:-

Quando le dimensioni di pagine e frame sono potenze di due, possiamo interpretare l'operazione di calcolo dell'indirizzo fisico come una concatenazione, anziché una somma:

$$1110001000000 + 101010 = 1110001101010 \quad \Rightarrow \quad 1110001 \text{ “attaccato a” } 101010.$$

Usando potenze di due, la costruzione dell'indirizzo fisico è più semplice e può essere gestita velocemente dall'hardware.

Esempio 9.8.5 (Esempio: Dimensione della Page Table)

Consideriamo un sistema in cui:

- l'indirizzo fisico è su 38 bit,
- l'indirizzo logico è su 40 bit,
- una pagina è di 8 Kbyte, quindi 2^{13} byte.

La dimensione della page table più grande possibile per questo sistema si calcola come segue:

1. **Calcolo del numero di entry nella page table**:
$$\frac{2^{40} \text{ byte}}{2^{13} \text{ byte}} = 2^{27} \text{ entry}.$$
2. **Numero di bit necessari per ogni entry**: Ogni entry deve essere in grado di indirizzare un frame nello spazio fisico. Lo spazio di indirizzamento fisico è diviso in:
$$\frac{2^{38} \text{ byte}}{2^{13} \text{ byte}} = 2^{25} \text{ frame}.$$
Il numero minimo di byte per rappresentare un frame è di 4 byte.
3. **Calcolo della dimensione totale della page table**:
$$2^{27} \times 2^{2} \text{ byte} = 2^{29} \text{ byte} = 512 \text{ Mbyte}.$$

La paginazione permette di separare lo spazio di indirizzamento logico da quello fisico. Ogni programma "vede" la memoria come uno spazio contiguo che parte sempre dall'indirizzo logico 0, ma in realtà il programma è distribuito in diversi frame fisici, sparsi in memoria assieme ad altri programmi. La paginazione introduce una protezione automatica dello spazio di indirizzamento. Un processo può accedere solo ai frame elencati nella sua page table, poiché ogni page table viene gestita e costruita dal sistema operativo per ciascun processo.

Osservazioni 9.8.5 Vantaggi e Limiti della Paginazione

- **Eliminazione della frammentazione esterna**: Ogni frame libero può essere utilizzato per memorizzare una pagina, riducendo gli sprechi di memoria.
- **Frammentazione interna**: Rimane una media di mezza pagina per processo, poiché l'ultima pagina del processo può non occupare completamente il frame assegnato.
- **Rilocazione dinamica**: La paginazione implementa una forma di rilocazione dinamica, dove ogni pagina è mappata su un diverso valore del registro di rilocazione, ossia l'indirizzo di partenza del frame che contiene la pagina.

9.8.2 Rilocazione Dinamica

Rilocazione Dinamica Tramite Registri di Limite e di Rilocazione

La **rilocazione dinamica** viene gestita inizialmente utilizzando due registri principali:

- **Registro di limite**: Controlla che l'indirizzo logico rientri nello spazio di indirizzamento del processo. Se l'indirizzo eccede questo limite, il sistema lancia un'eccezione di indirizzamento.
- **Registro di rilocazione**: Viene usato per traslare l'indirizzo logico approvato in un indirizzo fisico, aggiungendo il valore del registro di rilocazione all'indirizzo logico (vedi Fig. 9.6).

Questo approccio verifica dunque prima la validità dell'indirizzo logico rispetto allo spazio di indirizzamento del processo, per poi calcolare l'indirizzo fisico finale.

Rilocazione Dinamica di Ogni Singola Pagina Tramite la Page Table

Con la **paginazione**, la gestione della rilocazione dinamica cambia in quanto:

- La suddivisione in pagine elimina la necessità di un controllo tramite il registro limite.
- Ogni indirizzo logico è diviso in **numero di pagina** e **offset**, dove l'offset rappresenta sempre una posizione interna al frame associato, eliminando il rischio di indirizzamenti fuori dai limiti (vedi Fig. 9.8).

In questo caso, il controllo e la mappatura degli indirizzi sono effettuati dalla page table, la quale memorizza l'indirizzo di ciascun frame per ogni pagina del processo attivo.

99 Esercizi

99.1 Capitolo 5

99.1.1 1

Processo	Durata	priorità
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Table 99.1: Processi con durata e priorità

FCFS:

$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$	$\mathbf{P_5}$

SJF:

$\mathbf{P_2}$	$\mathbf{P_4}$	$\mathbf{P_3}$	$\mathbf{P_5}$	$\mathbf{P_1}$