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

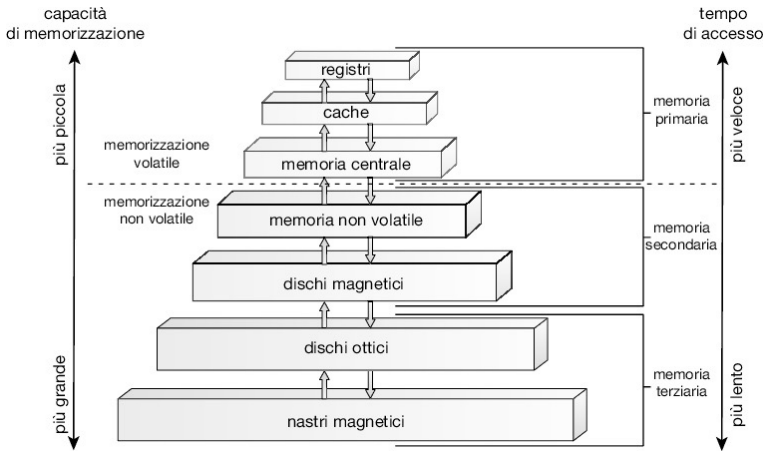
Nel contesto del **SO**, ci sono due principali tipi di memoria:

- **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:

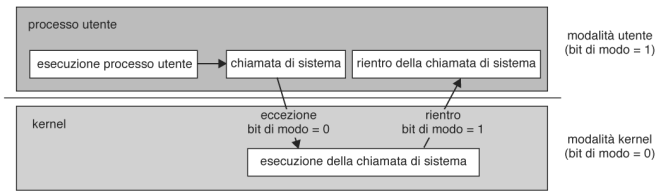
1. normale (modalità utente)
2. di sistema (modalità priviligiata, o kernel / monitor / supervisor mode)

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.

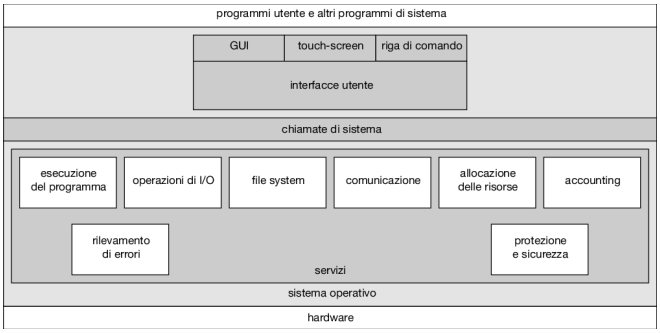
Un'altra variante: simile: in due registri appositi della CPU il SO carica rispettivamente l'indirizzo di inizio (base) e la dimensione (offset) dell'area di RAM assegnata ad un programma.

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

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:

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

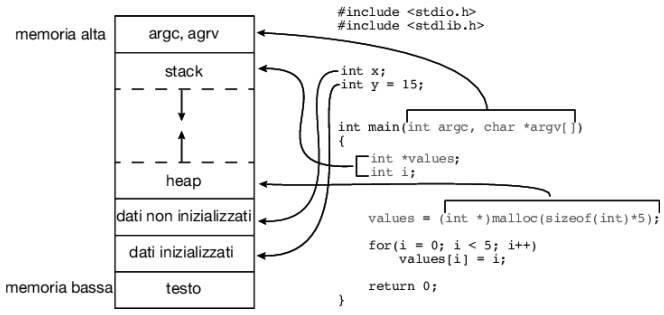


Figure 3.1: Concetto di processo

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**.

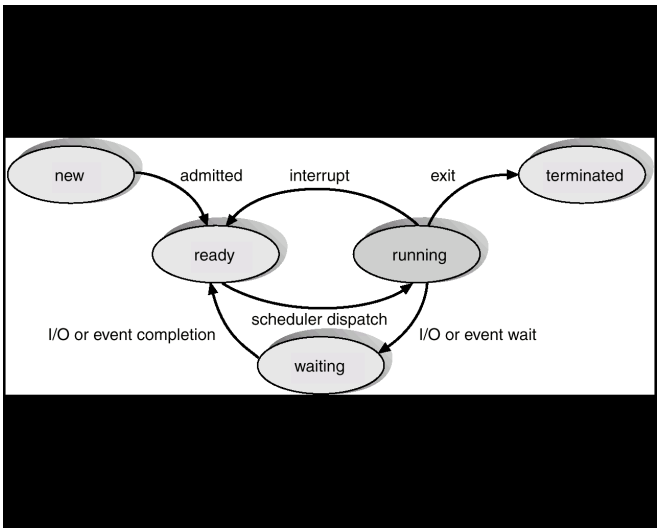
Un programma si **trasforma** in un processo quando viene lanciato, con il doppio click o da riga di comando. Un processo può anche **nascere** a partire da un altro processo, quando quest'ultimo esegue una opportuna system call (fork, spawn, etc)

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)

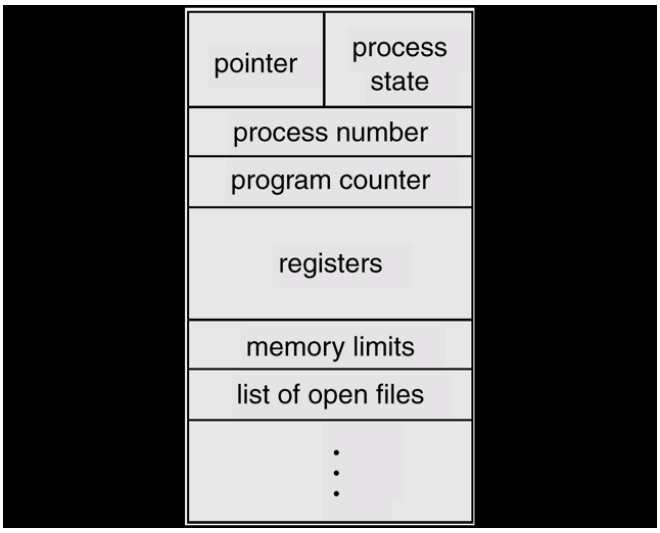
Per ogni processo, il sistema operativo (SO) mantiene una struttura dati chiamata *Process Control Block* (PCB), che contiene le informazioni necessarie per amministrare la vita di quel processo, tra cui:

- 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)

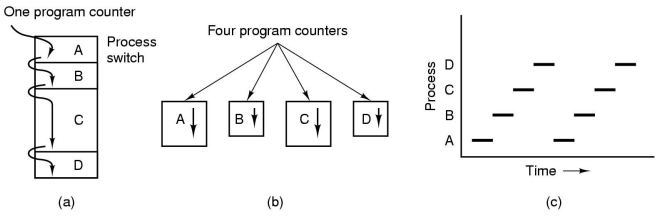


Figure 3.2: a) Ciò che succede in realtà
b) ciò che vede ogni singolo processo
c) Il risultato finale

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.

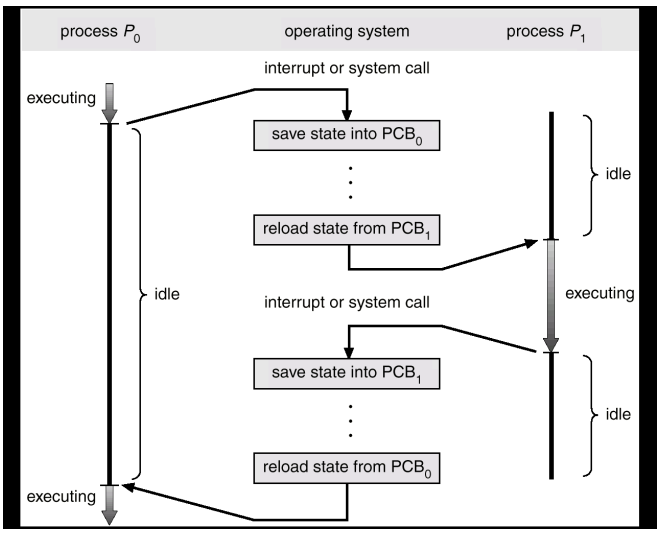
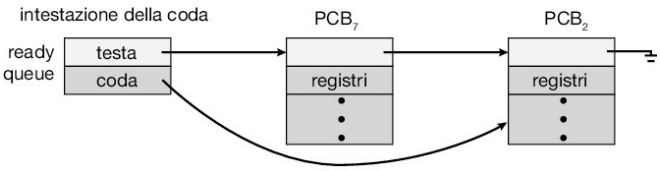


Figure 3.3: Fasi dello scheduling tra un processo e un altro

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

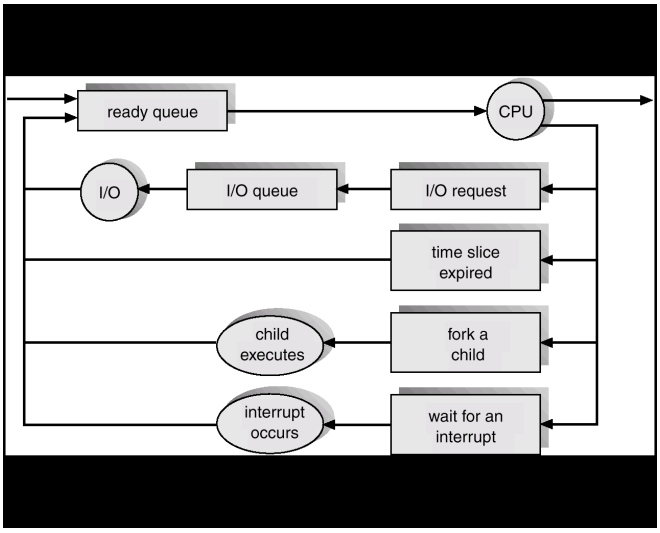


Figure 3.4: SX: new, DX: Terminated

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*).

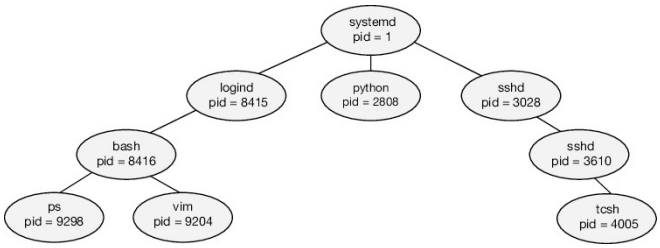
Osservazioni 3.3.1

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

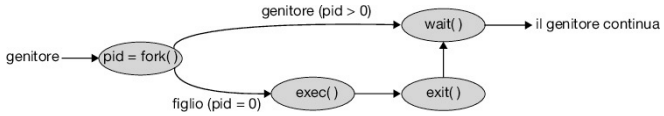
pid == 0 Lo ha solo il processo figlio.
pid = id-child lo ha solo il processo padre.
Così sono in grado di distinguere se sto operando con il figlio o con il padre.

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.

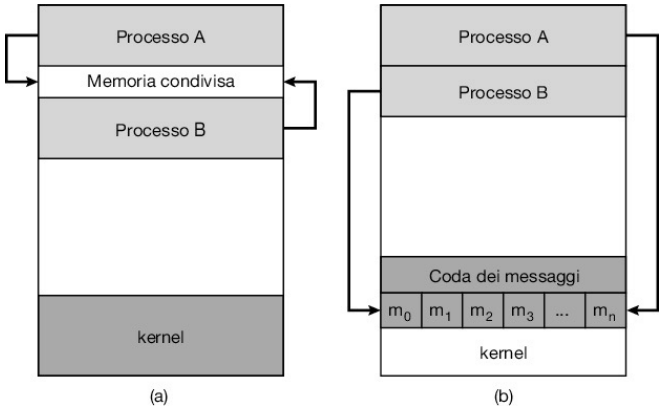


Figure 3.5: a) Memoria condivisa
b) Scambio di messaggi

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.

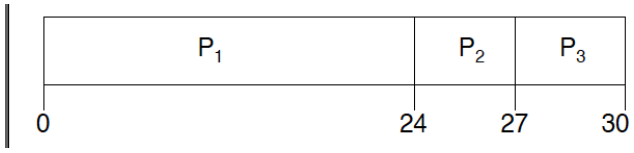
Esempio

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

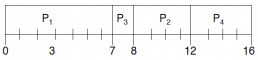


Figure 5.1: Average waiting time $(0 + 6 + 3 + 7)/4 = 4$

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**.

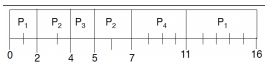


Figure 5.2: Average waiting time $(9 + 1 + 0 + 2/4 = 3$

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.

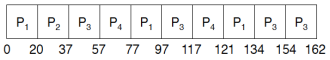
Il **Round Robin** è l'algoritmo di scheduling naturale per implementare il time sharing ed è quindi particolarmente adatto per i sistemi interattivi: nel caso peggiore, un utente non aspetta mai più di $(n-1)q$ unità di tempo prima che il suo processo venga servito.

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

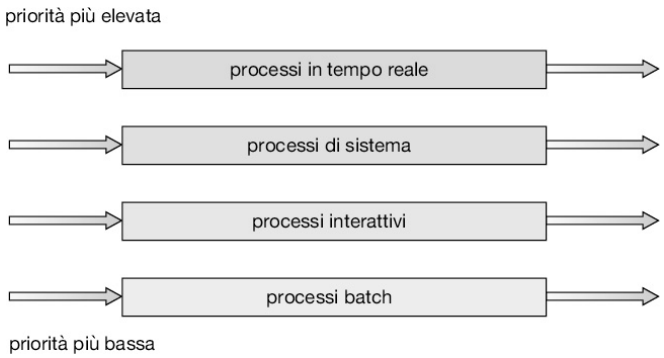


Figure 5.3: Partizionamento dei processi in più code

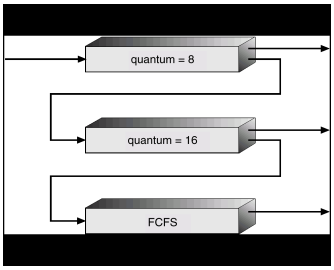


Figure 5.4: Esempio di MFQS

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

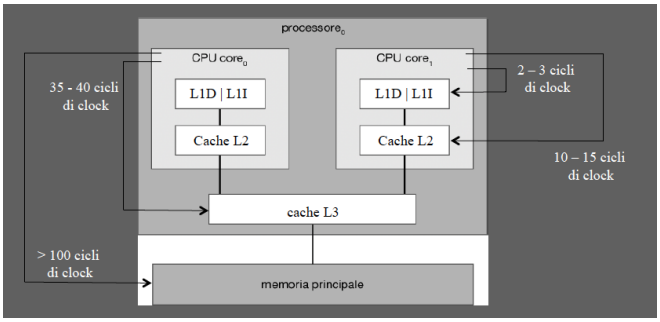


Figure 5.5: Scheduling example

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 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.

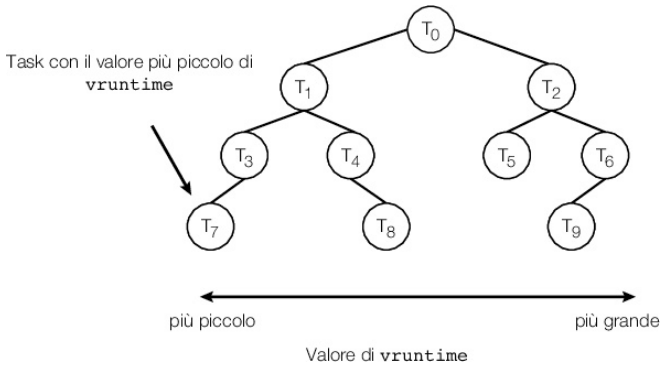


Figure 5.6: vrun time tree

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.

Struttura del Pattern

cooco