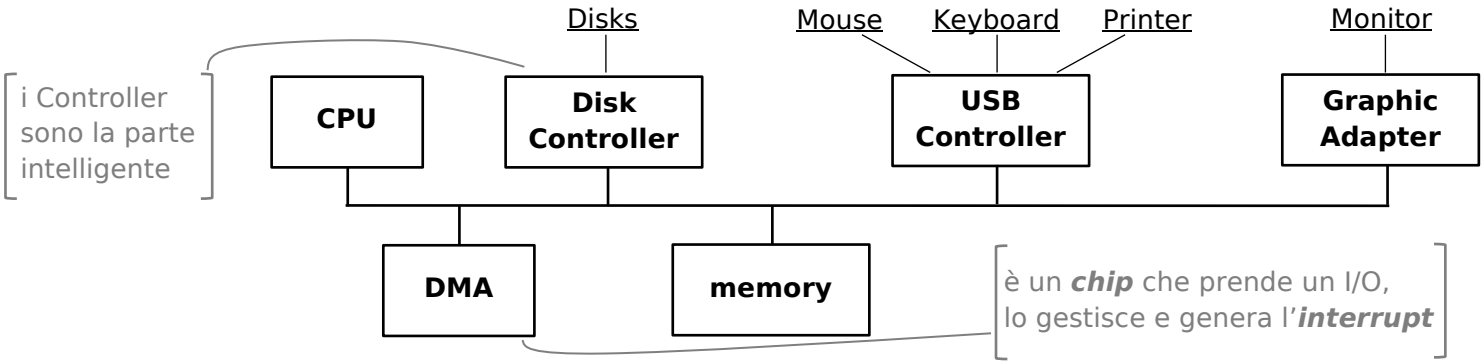
teoria di ***SISTEMI OPERATIVI (OS)***

prof. Vittorio Cortellessa

INDICE :

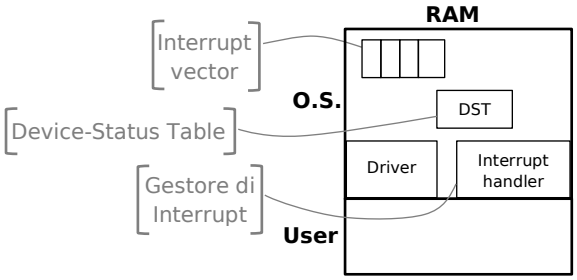
capitolo.1

Struttura di un Elaboratore

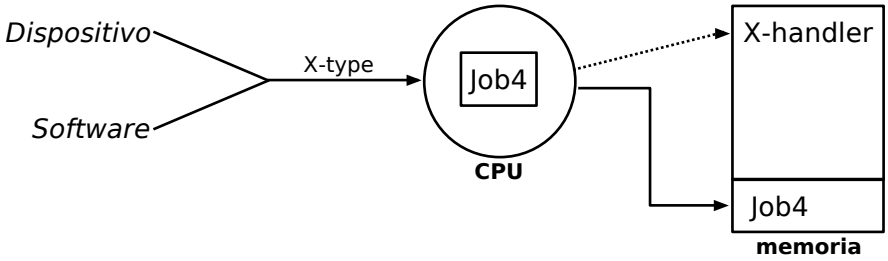


Il ***sistema operativo*** sposta i dati da/a ***memoria principale*** a/da ***buffer*** del dispositivo.

Il *controller* informa la *CPU*, mediante un interrupt , che ha finito l'operazione.
Un *interrupt* è un evento della CPU *asincrono* (può arrivare in qualunque momento). Serve ad interrompere una operazione.



L' *interrupt vector* è un vettore lungo quanto il numero di interrupt presenti. Le celle contengono l'indirizzo dell' *iterrupt handler* del tipo richiesto dall'operatore.



capitolo.2

Ruolo del Sistema Operativo

SYSTEM CALL

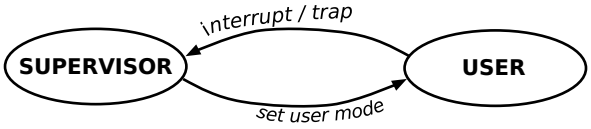
Le ***system call*** sono tutte quelle funzioni (*pezzi di codice*) che richiamano il sistema operativo. Forniscono l'interfaccia tra un programma in esecuzione e un OS.

SHELL

La ***shell*** è l'insieme di comandi racchiusi in un programma che permette agli utenti di interagire col sistema operativo.

modalità USER e SUPERVISOR

In un sistema a condivisione di risorse OS deve assicurare che un programma non possa danneggiare altri programmi. Quindi ogni supporto hardware deve avere due modi di operare: ***user*** e ***supervisor***.
Le istruzioni privilegiate (pericolose) posso essere eseguite solo in *supervisor*.



PROCESSO

Un ***processo*** è un *programma in esecuzione*.

Un *processo utente* deriva da un programma eseguito da utente.

Un *processo di sistema* corrisponde a un *return* di OS.

Un processo ha bisogno di risorse:
- tempo di CPU
- memoria
- files
- dispositivi I/O

Compiti di OS rispetto ai processi:
- Creazione e Cancellazione
- Sospensione e Ripristino (***scheduling***)
- Sincronizzazione (***deadlock***)
- Comunicazione

due o più processi (o azioni) si bloccano a vicenda aspettando che uno esegua una certa azione che serve all'altro e viceversa

GESTIONE MEMORIA PRINCIPALE

- Tenere traccia di quali parti della memori sono correttamente usate e da chi.
- Decidere quali processi caricare da disco quando diventa disponibile lo spazio in memoria.
- Allocare e Deallocare spazio di memoria quando necessario.

GESTIINE DELL' I/O

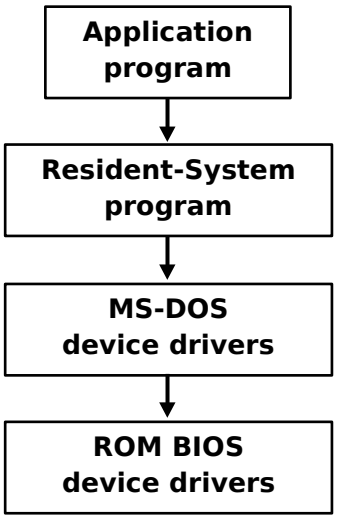
Sistema di :
- ***Buffering*** : preparare i dati di I/O, predisposti registri e aree di memoria.
- ***Caching*** : conservare in memorie veloci i dati usati più frequentemente per I/O.
- ***Spooling*** : virtualizzazione dei dispositivi di I/O mediante l'uso di aree di memoria per caricare e scaricare informazioni.

Driver per ogni tipo di dispositivo: routine di I/O che colloquia con il controller del dispositivo.

Interfaccia generale per i driver: programma che si invoca per iniziare qualsiasi operazione di I/O.

STRUTTURA INTERNA DI UN OS

esempio MS-DOS :



- OS è suddiviso in un numero di livelli, ognuno costruito su sottosistemi
- Il livello più basso è l'hardware (livello 0) e il più alto è l'interfaccia utente (livello N)

MACCHINA VIRTUALE

Si definisce ***macchina virtuale*** l'insieme costituito dall'hardware e dall'OS.

La macchina virtuale è un software che, attraverso un processo di virtualizzazione, crea uno ambiente che emula il comportamento di una macchina fisica.

ISTALLAZIONE E PARTENZA

- OS deve essere configurato per ogni specifica piattaforma di istallazione.
- Input della configurazione da adottare.
- Due alternative: ricompilazione del kernel o tabelle di selezione dei moduli.
- ***Booting*** : far partire un computer caricando in memoria il kernel dell'OS.
- ***Bootstrap*** : codice residente nella ROM capace di localizzare il kernel, caricarlo in memoria e avviarlo in esecuzione.

la procedura di ***boot*** può essere cambiata nel BIOS.

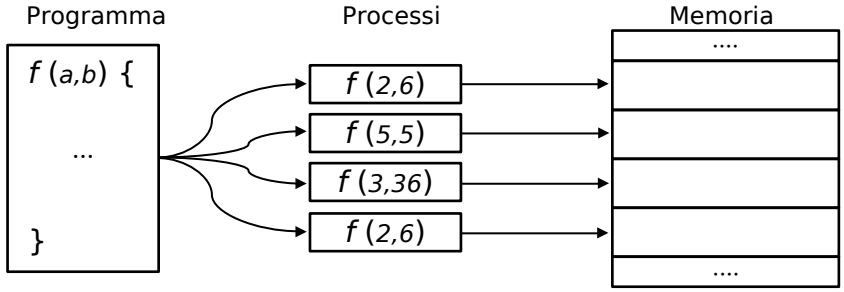
capitolo.3

I Processi : *processi e programmi*

Un ***processo*** è un ***programma in esecuzione***.

- **PROGRAMMA** : entità statica.
- **PROCESSO** : entità dinamica.

A un programma sono associati più processi ⟷ A un processo è associato un solo programma.



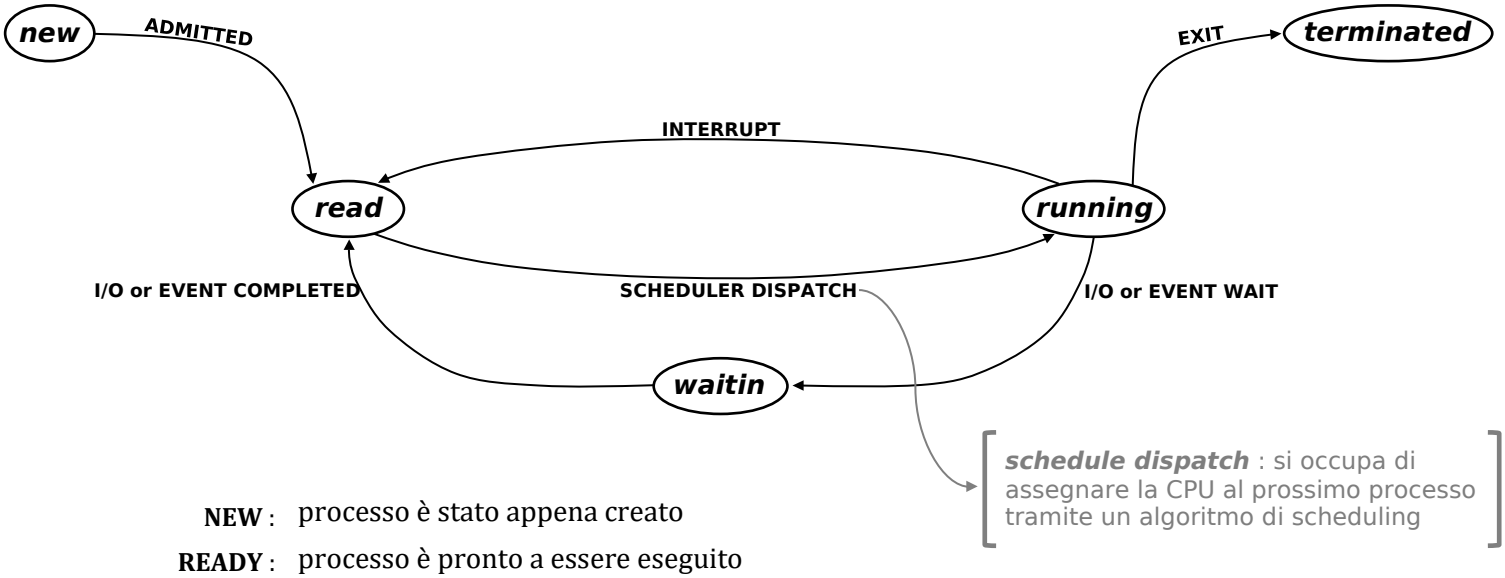
Un processo è caratterizzato da :
- sezione testo (*codice*)
- program counter
- (valori dei) registri della CPU
- stack
- sezione dati
- (stato)

Un processo esegue un'istruzione alla volta, in maniera sequenziale.

VITA DI UN PROCESSO

Durante la sua vita un *processo cambia stato* .

CAMBIAMENTI DI STATO



NEW : processo è stato appena creato

READY : processo è pronto a essere eseguito

RUNNING : processo è in esecuzione

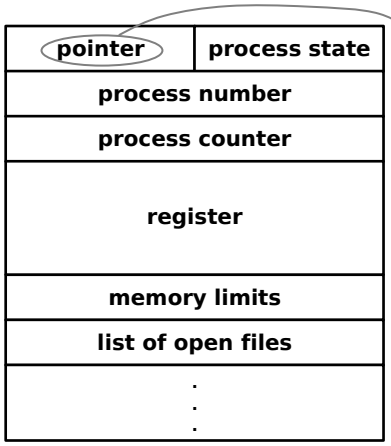
WAITING : processo sta attendendo che accada qualcosa

TERMINATED : processo ha finito la sua esecuzione

PROCESS CONTROL CLOCK (rappresentazione di un processo**)**

Il ***PBC*** è la struttura dati che il OS utilizza per conservare tutte le informazioni di un processo in vita.

- IDENTIFICATORE
- STATO DEL PROCESSO
- PROGRAM COUNTER
- REGISTRI DELLA CPU
- INFORMAZIONI DI SCHEDULING
- IDENTIFICATORI DI MEMORIA
- IDENTIFICATORI DI I/O
    - CONTABILITA' USO RISORSE



Ad uso delle code. Viene usato per essere limitato ad altri PCB nella sua stessa situazione

CODE DI PROCESSI DEL SISTEMA

OS deve gestire più processi alla volta.

Più processi possono voler usare la stessa risorsa contemporaneamente

os deve dotarsi di

- strutture dati (solitamente code)
- strategie di scheduling da applicare sulle code

Tipi di code:

- ***Ready Queue*** : insieme dei processi che risiedono in memoria principale, pronti in attesa di esecuzione.
- ***Coda di Dispositivo*** : insieme di processi che ottengono l'utilizzo di un certo dispositivo.

non esiste una I/O queue, bensì esiste una coda per ogni singolo device

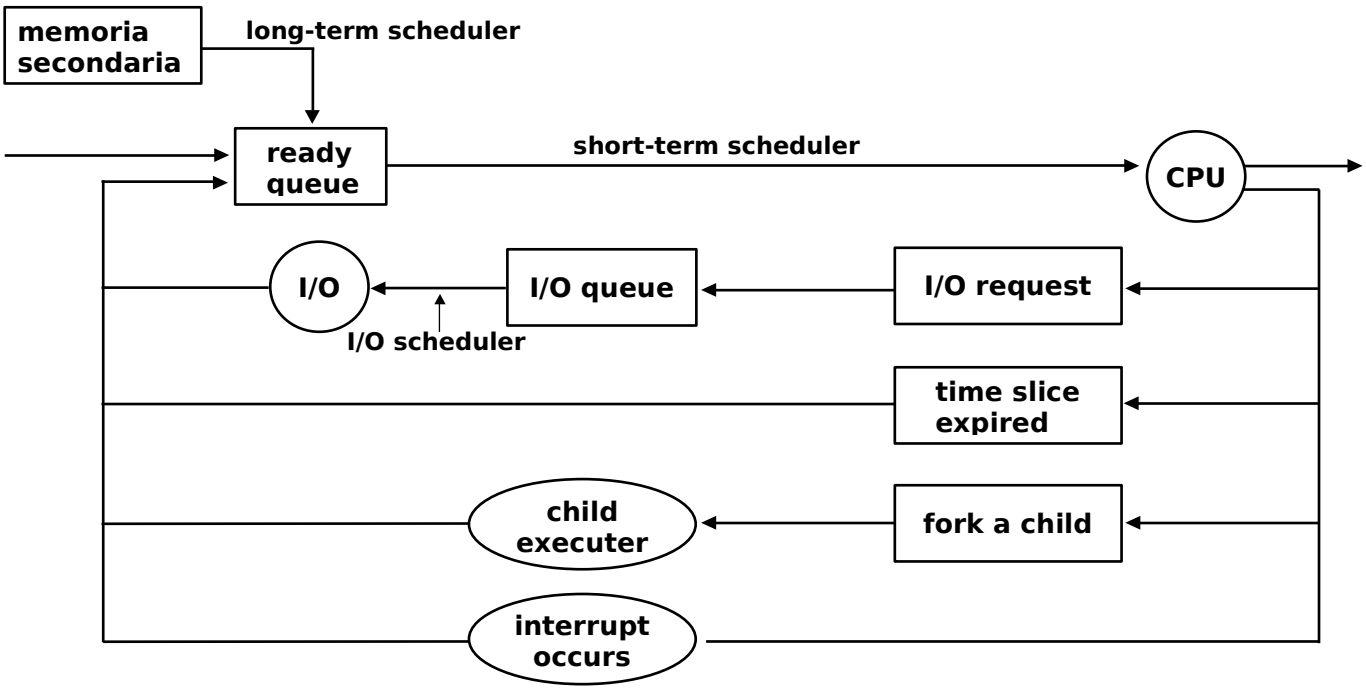
PRINCIPALI OS SCHEDULERS

Short-Term Schedulers **(** o *CPU Schedulers* **)**

- Seleziona il prossimo processo da mandare in esecuzione
- Viene invocato molto frequentemente ⇒ deve essere veloce

Long-Term Schedulers

- Seleziona quali processi devono risiedere nella *ready queue* rispetto a tutti quelli pronti nel sistema
- Controlla il grado di multiprogrammazione del sistema
- Viene invocato poco frequentemente ⇒ può essere lento



CONTEXT SWITCHING

Il ***context switching*** è l'operatore di avvicendamento di un processo in esecuzione con un altro.

- OS salva il PCB del vecchio processo e carica quello del nuovo.
- Il tempo di CS dipende dall'hardware.
- In questo tempo OS non fa lavoro utile per l'utente, quindi si parla di *overhead*.

CREAZIONE DI UN PROCESSO

Un processo (*padre*) crea altri processi (*figli*) che, a loro volta, possono avere altri processi, il tutto a formare un *albero di processi*.

SYSTEM CALL FORK

Fork crea un altro processo che ha una copia dello spazio di indirizzo del padre:

- ripartono dalla stessa istruzione dopo la fork, cioè si origina un nuovo *programma couter* all'interno dello stesso codice.
- per poterli distinguere il codice di ritorno del fork è differente tra padre e figlio.

TERMINAZIONE " ANORMALE " di un processo

Il padre può terminare l'esecuzione di uno dei figli (*istruzione* ***abort***) perché:

- Il figlio ha ecceduto nell'uso delle risorse
- Il compito del figlio non è più richiesto
- Il padre sta terminando

COPERAZIONE TRA PROCESSI

I processi possono influenzare l'esecuzione di un altro. Vantaggi:

- Condivisione informazione
- Velocizzazione del calcolo
- Modularità di compiti (suddividere le funzioni di sistema in processi o thread distinti)
- Convenienza dell'utente

Per regolare questa "influenza" OS ha bisogno di meccanismi di comunicazione (**IPC** , ***interprocess communication***) o di sincronizzazione.

PROBLEMA PRODUTTORE-CONSUMATORE

Il *processo produttore* produce informazioni che sono consumate da un *processo consumatore*.

Con un *buffer* (il *buffer* è una memoria di transito e intermediaria) di taglia finita bisogna regolamentare le operazioni di consumo in maniera "opportuna".

Il buffer dovrà risiedere in una zona di memoria condivisa (ICP a memoria condivisa).

Due tipi di Buffer : - limitato (dimensione fissa)
- illimitato (non pone limiti alla dimensione)

Il consumatore e il produttore dovranno sincronizzarsi in modo tale che il consumatore non provi mai a consumare un'unità non ancora prodotta.

INTERPROCESS COMMUNICATION (ICP)

L'***interprocess communication*** è la parte di OS che gestisce la comunicazione tra processi. Può essere a *memoria condivisa* o a *scambio di messaggi*.

Mette a disposizione degli *user process* due system call: - send
- receive

Se due processi vogliono comunicare devono:

- Stabilire un canale di comunicazione (a un canale logico ne corrisponde uno fisico, come una cella di memoria)
- Scambiare messaggi usando *send* e *receive*

Ci sono diversi modi di creare un canale di comunicazione e le operzioni **send( )** e **receive( )**:

- comunicazione diretta o indiretta
- comunicazione sincrona o asincrona
- gestione automatica o esplicicita dei buffer

COMUNICAZIONE DIRETTA

Naming : i processi hanno bisogno di far riferimento ad altri processi, quindi si può usare una comunicazione diretta o indiretta.

Ogni processo che intendi comunicare deve memorizzare esplicitamente il processo ricevente o trasmittente della comunicazione.

es . *send*(Q,message) ⟵ manda *message* al processo *Q*

receive(P,buffer) ⟵ riceve *in buffer* dal processo *P*

Il canale di comunicazione si genera automaticamente ed è aossociato sempre esattamente a due processi.

Esiste un canale per ogni coppia di processi.

COMUNICAZIONE INDIRETTA

I processi comunicano attraverso delle *mailbox* o delle porte. Ogni mailbox ha un identificatore unico. Due processi possono comunicare solo se condividono una mailbox.

es . *send*(A,message) ⟵ manda un messaggio alla mailbox A

receive(A,message) ⟵ riceve un messaggio dalla mailbox A

Inoltre la mailbox funge da canale e in questo caso: **-** un canale può essere associato a più di due processi
- tra due processi comunicanti possono esserci più canali

Una mailbox ha vita autonoma e l'OS fornisce a un processo le seguenti operazioni:

- Creare una nuova mailbox (in questo caso il processo è il proprietario predefinito della mailbox)
- Inviare (send) e Ricevere (receive) attraverso la mailbox
- Rimuovere una mailbox

CODA DI MESSAGGI

Sia nella comunicazione diretta che indiretta i messaggi risiedono all'interno di *code temporanee*.

Esistono 3 modi per realizzare queste code :

1. ***Capacità Zero*** : la coda ha lunghezza massima 0 , quindi il canale non può avere messaggi in attesa al suo interno.
2. ***Capacità Limitata*** : la coda ha una lunghezza finita *n* , quindi al suo interno possono esserci al massimo *n* messaggi.
3. ***Capacità Illimitata*** : la coda ha lunghezza potenzialmente infinita , quindi non c'è limite alla capienza di messaggi.

Il caso *0* è detto "*sistema a scambio di messaggi senza buffering*" , gli altri due "*sistema con buffering automatico*".

SINCRONIZZAZIONE

Lo scambio di messaggi tramite *receive* e *send* può essere ***sincrono*** (o *boccante*) o ***asincrono*** (o *non bloccante*).

- ***Invio Sincrono*** : il processo che invia il messaggio si blocca in attesa che il processo ricevente, o la mailbox, riceva il messaggio.
- ***Invio Asincrono*** : il processo invia il messaggio e riprende la propria esecuzione.
- ***Ricezione Sincrona*** : il ricevente si si blocca nell'attesa dell'arrivo di un messaggio.
- ***Ricezione Asincrona*** : il ricevente riceve un valore nullo o un messaggio valido.

capitolo.4

Threads

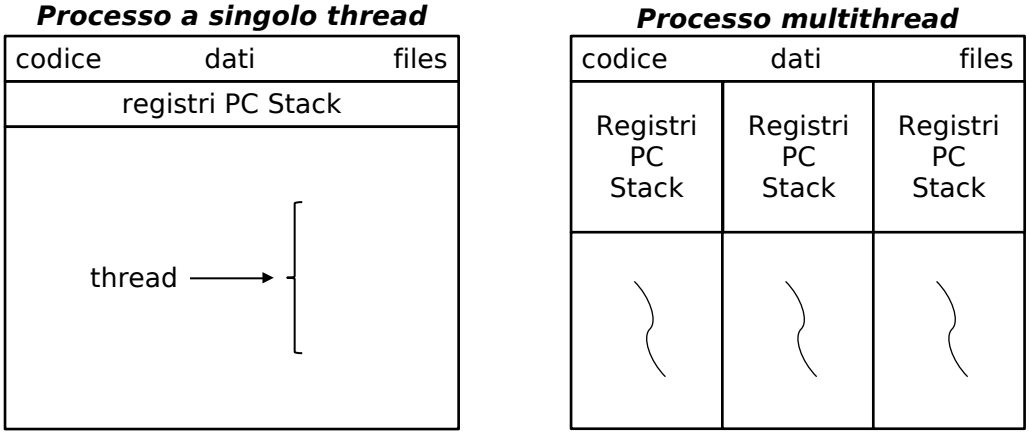
Un ***threads*** è l'*unità base della CPU*. Comprende:

- un identificatore (**ID**)
- un program counter
- un insieme di regitri
- una pila (***stack***)

Tutti i *threads* appartenenti allo stesso processo possono condividere :

- sezioni di codice
- sezioni di dati
- risorse OS

Un processo pesante (*heavyweigh process*) è composto da un solo thread. Un processo *multithread* è in grado di svolgere più compiti in modo concorrente.



PROPRIETA' DEI THREADS

- Un solo threads alla volta in esecuzione
- Un threads può leggere o scrivere nello spazio di altri thread dello stesso task (quindi appartiene allo stesso processo)
- In caso di bloccaggio di un thread tutti gli altri possono continuare la loro esecuzione e, quindi, il processo non si blocca
- Il CS a livello di thread è più rapido

THREADS : GESTIONE A LIVELLO UTNTE E A LIVELLO KERNEL

I thread possono essere distinti in *thread a livello utente* e *thread a livello kernel* : i primi sono gestiti sopra il livello del kernel e senza il suo supporto, i secondi sono gestiti da OS.

A livello utente, per gestire i thread, abbiamo bisogno di una *libreria dei thread* : essa fornisce al programmatore un'API per la creazione e gestione dei thread.

capitolo.5

CPU Scheduling

In un sistema dotato di singoli core si esegue al massimo un processo per volta, costringendo tutti gli altri ad aspettare che la CPU venga liberata.

L'obbiettivo del *multiprogrammatore* (***multitasking***) è quello di avere sempre un processo in esecuzione.

CPU BURST e I/O BURST

Il comportamento di un processore è quello di alternare una sequenza di uso continuo della CPU (***CPU burst***) con una sequenza dio I/O (***I/O burst***).

Quindi, per utilizzare al meglio la CPU, bisogna mandare in esecuzione qualche altro processo durante *I/O burst* del processo corrente.

La scelta del processo da mandare in esecuzione è effettuata dallo scheduler.

CPU SCHEDULER

Lo *scheduler* della CPU interviene ogni volta che la CPU passa nello stato di inattività, scegliendo quale processo pronto in memoria (quindi nello stato di ready) mandare in esecuzione.

La CPU diventa inattiva quando :

1. un processo va da running a waiting
2. un processo in esecuzione termina
3. un processo va da runnig a ready
4. un processo va da waiting a ready

DISPATCHER

Il ***dispatcher*** è un modulo che passa il controllo della CPU al processo scelto dallo scheduler.
Questa funzione comprende :

- *Context Switching* da un processo all'altro
- Passaggio alla modalità utente
- Il salto alla giusta posizione del programma utente per riavviarne l'esecuzione

Il tempo richiesto dal dispatcher per fermare un programma e avviarne un altro è noto come *dispatch latency*.

SCHEDULING PREEMPTIVE e NONPREEMPTIVE

Qualsiasi azione scheduling si dice ***preemptive*** se ha possibilità di interromper il processo in esecuzione per sostituirlo con un altro.

Un'azione scheduling è, quindi, ***nonpreemptive*** quando attende che il processo in esecuzione abbandoni autonomamente la CPU.

POLITICA DI SCHEDULING

Una politica di scheduling è un algoritmo che regola ogni azione di scheduling e quindi decide quando e come selezionare un processo da eseguire.

Per confrontare le varie politiche ci si basa su diversi criteri :

- *Utilizzazione dlla CPU* : percentuale di tempo in cui la CPU è occupata a fare qualcosa
- *Throughput* : numero di processi che completano la loro esecuzione per unità di tempo
- *Turnaround Time* : tempo necessario ad eseguire un processo
- *Tempo di Attesa* : tempo totale di attesa di un processo nella ready queue
- *Tempo di Risposta* : tempo che intercorre tra un richiesta e l'arrivo della prima risposta.

POLITICHE CHE CONSIDERIAMO

1. ***First Come First Served***
   I processi vengono schedulati nello stesso ordine con cui arrivano nella ready queue.
2. ***Shortest Job First***
   Schedula il processo che presenta il prossimo CPU burst più breve.
   - *STF preemptive* (shortest remaning time first, SRTFF) : se un nuovo processo arriva nella ready queue con un burst più corto del tempo rimanente del processo corrente, lo si rimpiazza

- *STF nonpreemptive* : anche se arrivano nuovi processi, quello corrente continuerà fino a quando ne ha bisogno

3. ***Con Priorità***

   Un numero (intero) di priorità è assegnato ad ogni processo.
   La CPU è allocata al processo con priorità più alta (quindi intero più piccolo).
   Il problema di *starvation* (o fame, processi con bassa priorità che non vengono mai eseguiti) si può risolvere con *aging* (col passare del tempo la priorità cresce).

4. ***Round-Rolin***
   A ogni processo viene assegnato una piccola unità di tempo di CPU (*quanto*).
   Quando questo tempo scade, il processo è preempted (viene sostituito) e viene inserito alla fine della ready queue

5. ***Code Multilivello***
   Le ready queue viene partizionata in code separate, ciascuna con la sua priorità.
   Si servivano prima tutti i processi in coda ad alta priorità e poi via via gli altri.
   Sia il caso preemptive che quello non preemptive soffrono di starvation.
   Ogni coda, al suo interno , ha la propria politica di scheduling.
   Per risolvere il problema dello starvation abbiamo 2 possibilità :
   a. ***CPU slicing*** : ad ogni coda viene assegnata una certa percentuale di tempo di CPU, che dividerà tra i suoi processi
   b. ***Code con Feedback*** : in questo caso al processo è permesso spostarsi fra le code. Se un processo attende da troppo tempo si può spostare in una coda con priorità più alta.
      Uno *scheduler a code multilivello con feedback* è caratterizzato da:
      - numero di code
      - algoritmi di scheduling di ogni coda
      - metodo usato per la migrazione dei processi tra code (dal basso vero l'alto / dall'alto verso il basso)
      - metodo usato per determinare in quale coda deve finire un processo quando richiede un servizio

capitolo.6

Sincronizzazione tra Processi

Processi cooperanti condividono uno spazio di indirizzi : bisogna in qualche modo garantire l'integrità dei dati e questo richiede meccanismi che assicurano l'esecuzione in ordine appropriato dei processi cooperanti (IPC).

SEZIONE CRTICA

Ogni zona di codice in cui un processo può modificare variabili, tabelle, file, etc. comuni è detto *sezione critica*.

Quando un processo è in esecuzione nella propria sezione critica tutti gli altri processi non possono accedere alla propria. Bisogna quindi progettare un protocollo per garantire la cooperazione tra i processi.

Per accedere alla sezione critica ogni processo deve chiedere il permesso in una sezione di codice detta ***entry section*** .

La sezione critica può essere seguita da un ***exit section*** e la restante parte è detta *sezione non critica*.

Una soluzione al problema della sezione critica deve soddisfare 3 requisiti :

1. *Mutua Esclusione* : Se un processo sta eseguendo la sua sezione critica allora nessun altro processo può stare ad eseguire la sua sezione critica.
2. *Progresso* : se nessun processo sta eseguendo la sua sezione critica è c'è qualche processo che vuole entrare nella sua, allora la decisione su quale processo far entrare nella sezione critica deve essere presa in tempo finito e solo dai processi che non stanno nella loro sezione normale.
3. *Attesa Limitata* : dopo che un processo ha fatto richiesta di ingresso nella sua sezione critica e prima che la sua richiesta venga soddisfatta deve essere limitato il numero di volte che ad altri processi è consentito entrare nelle loro sezioni critiche.

ALGORITMO DEL FORNAIO

- Prima di entrare nella sezione critica un processo riceve un numero
- Il processo che detiene il numero più piccolo entra
- Se i processi P_i e P_j ricevono lo stesso numero allora P_i entra per primo, altrimenti P_j
- Lo schema di numerazione genera sempre numeri in ordine non decrescente

Variabili condivise :

var **choosing : array[** 0, … , n-1 **]** ***of boolean***

number : array[0, … , n-1 **]** ***of integer***

Tutte inizializzate a *0* e a *false*

Idee di base : **-** quando si sceglie viene assegnato il numero più alto.
- si attende se qualcuno sta scegliendo il numero
- si attende se qualcuno ha scelto il numero ed è più piccolo del proprio
- All'uscita della sezione critica si azzera il proprio numero

TEST-and-SET

Per risolvere il problema della sezione critica in maniera relativamente semplice possiamo utilizzare delle *istruzioni atomiche*.

L'istruzione ***test-and-set*** testa e modifica il contenuto di una cella di memoria in maniera atomica, *non interrompibile*.

Se si dispone di un *test-and-set* si può garantire la mutua esclusione aggiungendo una variabile booleana ***lock***.

Il processo P_i avrà la seguente struttura:

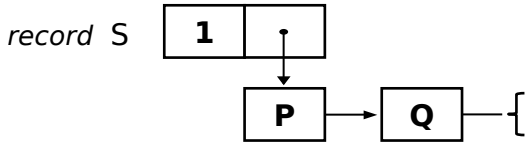
```
var lock : boolean    ( inizialmente falsa )
    repeat
      while Test-and-Set (lock) do no-op ;   ( sezione critica )
            lock := false ;                  ( sezione normale )
    unhl false
```

BFW (Busy Form of Waiting)

Quando un processo non può entrare nella sua sezione critica rimane ad attendere il suo turno "facendo qualcosa". Questo porta ad uno spreco di tempo di CPU. Bisogna, quindi, trovare una soluzione che permetta al processo in attesa di andare nello stato di waiting.

SEMAFORI

Un ***semaforo*** serve a risolvere il BFW. E' un record con due campi : ***value*** (un intero) e ***L*** (una coda).



Il *value* viene inizializzato a 1 per garantire la mutua esclusione.

La *coda L*, all'inizio vuota, è di tipo FIFO per non far alternare tra i processi all'infinito.

Le operazioni che si possono effettuare sui semafori sono :
- ***Inizializzazione***
- ***wait***
- ***signal***

} sono system call

WAIT e SIGNAL

Wait e ***Signal*** sono *operazioni atomiche*. Esse utilizzano due semplici operazioni :
- *block* : sospende il processo P che la invoca
- *wakeup*(*S*) : riattiva l'esecuzione di un processo P

```
Wait( S ) :   S.value := S.value – 1 ;
              if ( S.value < 0 )
                 then begin
                      aggiungi questo processo a S.L. ;
                      block ;
              end ;

Signal( S ) : S.value := S.value + 1 ;
              if ( S.value ≤ 0 )
                 then begin
                      rimuovi un processo P da S.L. ;
                      wakeup( P ) ;
              end ;
```

n.b.
Due processi non possono eseguire *wait* contemporaneamente in quanto non verrebbe garantita la mutua esclusione.

MUTUA ESCLUSIONE TRA *n*-PROCESSI

```
var mutex : semaphore   ( variabili condivise )
Processo Pi :  repeat
                    wait ( mutex ) ;        ( sezione critica )
                    signal ( mutex ) ;      ( sezione "normale" )
               unhl false ;
```

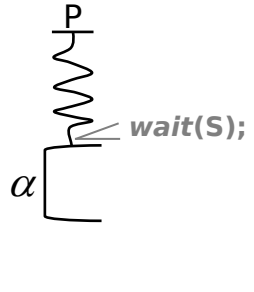
DIVERSI TIPI DI SEMAFORI

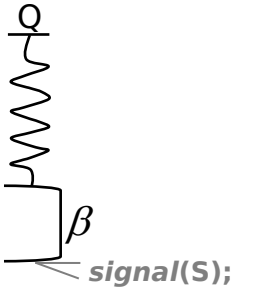
Abbiamo visto che, inizializzando *value* a 1, si può risolvere il problema della sezione critica. Tuttavia i semplici semafori (inizializzati ad altri valori) possono avere diversi scopi :

a. Semaforo per la precedenza

In questo caso ho bisogno di inizializzare *value* a 0.

$\beta < \alpha$ (sezione di codice), *P e Q* processi.





Vogliamo far si che β venga eseguito prima di α.
P quindi aspetta (*wait*) al "cancello"subito prima di α che β venga eseguito nel processo *Q*. Al termine dell'esecuzione di β, il "cancello" verrà aperto (*signal*).

b. Semaforo per ottenere un rendez-vous

In questo caso, utilizziamo i semafori per sincronizzare due processi (e per fargli eseguire una sezione di codice contemporaneamente).

Quindi, in questo caso, ho bisogno di una coppia di semafori ***S*** e ***T*** . Il primo processo che raggiunge la sezione di codice dovrà comunicare il suo arrivo e aspettare l'arrivo dell'altro processo.