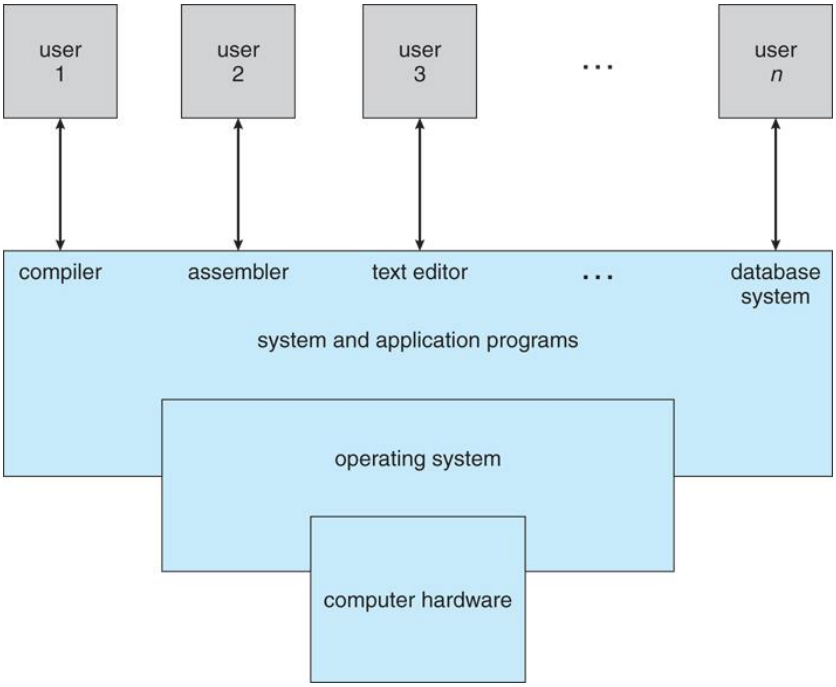
APPUNTI DI SISTEMI OPERATIVI

INTRODUZIONE



L'hardware (CPU, memoria, I/O) forniscono le risorse di calcolo di base per il sistema. Le applicazioni forniscono il modo attraverso il quale queste risorse hardware si rendono utili per risolvere i problemi degli utenti. Il sistema operativo gestisce e coordina le interazioni tra hardware, utenti e applicazioni.

2 PUNTI DI VISTA:

- Utente:
  la maggior parte degli utenti stanno davanti ad un computer composto da monitor, tastiera, mouse e periferiche varie. Questa tipologia di sistema è progettata per far sì che un utente ne monopolizzi l'utilizzo delle risorse. Il progettista deve tenere a mente la facilità di utilizzo con attenzione all'utilizzo delle risorse (sia hardware che software).
- Sistema:
  il sistema operativo è il programma più coinvolto "intimamente" con l'hardware. In questo contesto il sistema operativo può essere visto come allocatore di risorse. Ci sono tanti tipi di risorse (tempo CPU, spazio di memoria, spazio di salvataggio dei file, dispositivi I/O, ecc.) e il sistema operativo deve gestire la loro allocazione tenendo conto delle necessità di tutti i programmi e di tutti gli utenti che ne fanno uso.

ORGANIZZAZIONE DI UN SISTEMA

Un computer general-purpose moderno consiste di una o più CPU ed un certo numero di controller di dispositivi connessi lungo un bus comune che fornisce loro accesso alla memoria condivisa.

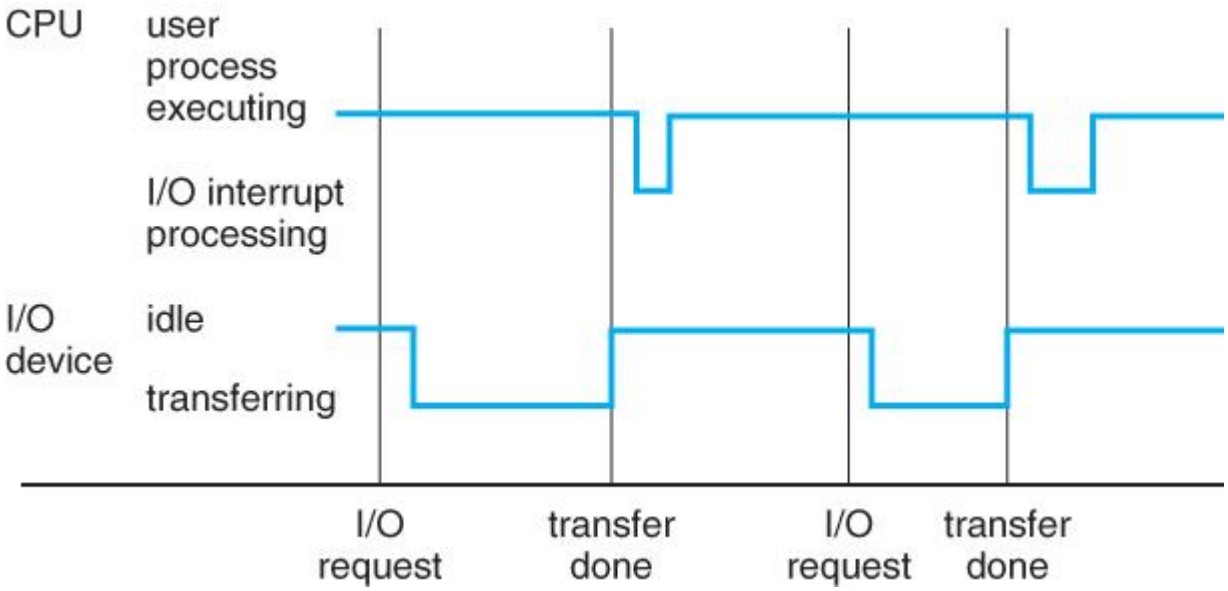
La CPU e i controller di dispositivo posso essere in esecuzione in concorrenza, competendo così per i cicli di memoria. Per assicurare un accesso ordinato alla memoria condivisa, esiste un controller della memoria che si occupa di sincronizzare gli accessi alla memoria.

Quando un computer viene avviato ha bisogno di un programma iniziale da eseguire. Questo programma si chiama **programma di bootstrap**. Il boostrap è salvato in una EEPROM (erasable programmable read-only memory), conosciuta col nome di firmware, che fa parte dell'hardware.

Il bootstrap inizializza tutti gli aspetti del sistema, dai registri della CPU, passando per i controller e finendo con il contenuto della memoria. Il bootstrap deve sapere come caricare il sistema operativo per poi lanciarlo in esecuzione. Per fare ciò deve localizzare, caricare il kernel del sistema operativo, eseguire il primo processo (es. init su Linux) ed aspettare che accada un qualche evento.

L'occorrenza di un evento è solitamente segnalata con un **interrupt** che può essere via hardware o via software. L'hardware può triggerare un interrupt in qualunque momento mandando un segnale alla CPU, solitamente attraverso il bus di sistema. Il software può triggerare un interrupt eseguendo un'operazione speciale chiamata **system call**.

Quando la CPU riceve l'interrupt, stoppa quanto sta facendo e trasferisce l'esecuzione ad una locazione fissata che solitamente contiene l'indirizzo di partenza che punta alla **routine di interrupt**.



I programmi del computer devono risiedere in **memoria centrale** (RAM) per essere eseguiti. La memoria centrale è l'unica area di memoria alla quale il processore può accedere direttamente. L'interazione tra CPU e memoria avviene attraverso le istruzioni di load e store verso specifici indirizzi.

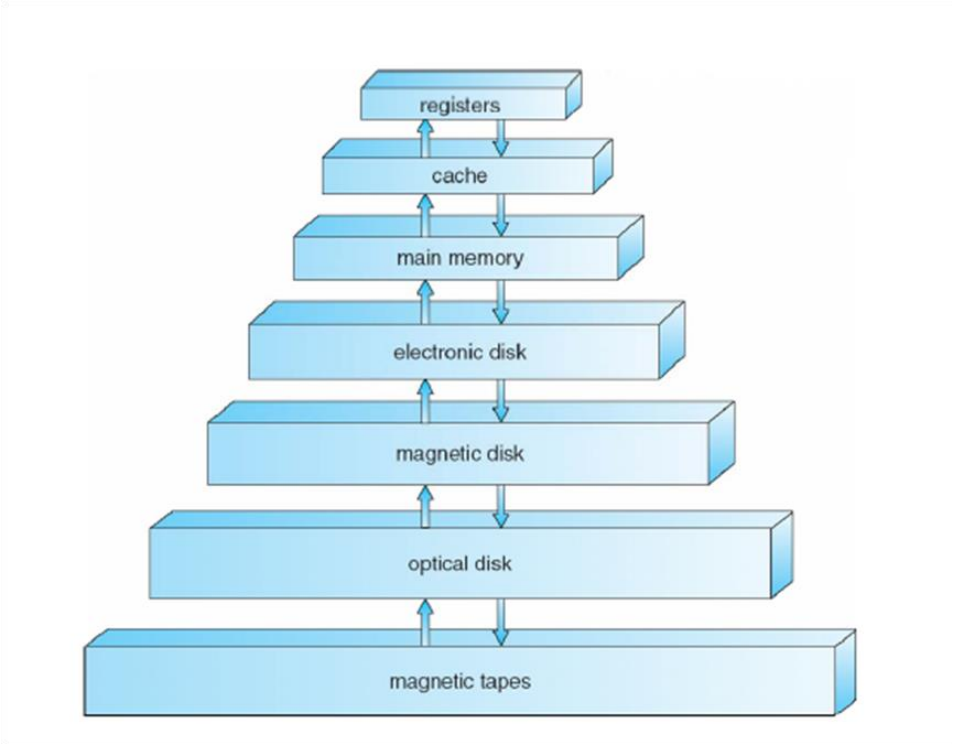
Un tipico ciclo istruzione-esecuzione eseguito in un sistema con architettura di von Neumann, può consistere dei seguenti passi:

1. FETCH di un'istruzione dalla memoria che viene salvata;
2. DECODE dell'istruzione che consente poi di eseguire fetch degli operandi dalla memoria con salvataggio degli stessi in uno o più registri;
3. EXECUTE dell'istruzione con successivo salvataggio del risultato in memoria.

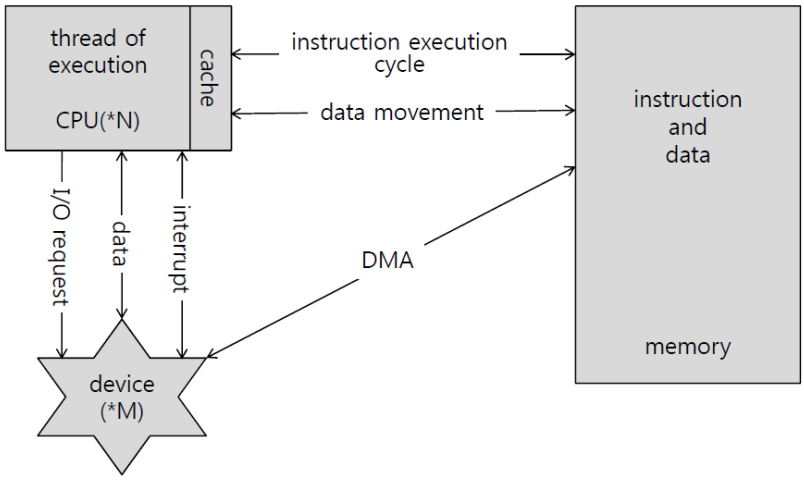
Idealmente, vorremmo che il programma e i dati risiedano permanentemente in memoria centrale ma ciò non è possibile per le seguenti due ragioni:

1. Troppo piccola;
2. È volatile.

Di conseguenza, la maggior parte dei sistemi hanno a disposizione una memoria secondaria che fa da estensione della memoria centrale. L'unico requisito per la memoria secondaria è che sia in grado di contenere grandi quantità di dati permanentemente.



La memoria è solo una delle tante tipologie di dispositivi I/O presenti in un computer. Una grande parte del sistema operativo è dedicata alla gestione dell'I/O sia per la sua importanza nell'affidabilità e nelle performance del sistema, sia per le tante tipologie di dispositivi esistenti.



STRUTTURA DI UN SISTEMA OPERATIVO

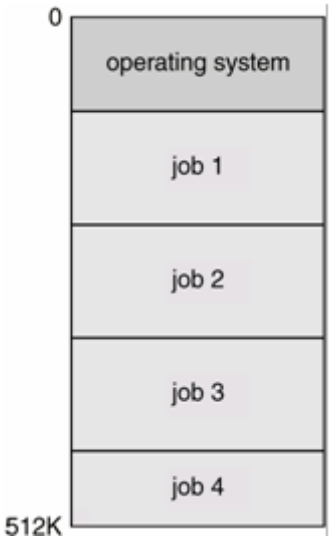
Un sistema operativo fornisce l'ambiente all'interno del quale i programmi vengono eseguiti. Internamente, i sistemi operativi possono variare molto ma tra tutti ci sono dei punti in comune che vale la pena analizzare.

MULTIPROGRAMMAZIONE

Un singolo utente non può, in generale, mantenere sia la CPU che i dispositivi di I/O occupati tutto il tempo. La multiprogrammazione incrementa l'utilizzazione della CPU organizzando job (codice + dati) in maniera tale che la CPU ne abbia sempre uno da eseguire.

L'idea è la seguente: il sistema operativo mantiene vari job in memoria nello stesso momento. Questo insieme di job può essere un sottoinsieme dei job presenti all'interno del job pool (che contiene tutti i job presenti nel sistema),

considerato che il numero di job che possono essere mantenuti simultaneamente è solitamente più piccolo del numero di job che possono essere tenuti nel job pool.
Il sistema operativo sceglie un job dalla memoria e ne inizia l'esecuzione. Eventualmente il job potrebbe dover aspettare un qualche compito (es. operazione di I/O) per essere completato. In tal caso, invece di lasciare la CPU in idle, il sistema operativo manda in esecuzione un altro job.
Eventualmente, il primo job finisce di aspettare e gli viene ridata la CPU. Finché resterà almeno un job da essere eseguito, la CPU non andrà mai in idle.



TIME SHARING (MULTITASKING)

I sistemi multiprogrammati forniscono un ambiente nel quale le varie risorse di sistema sono utilizzate efficientemente ma non forniscono l'interazione con l'utente.

I sistemi time sharing sono un'estensione logica della multiprogrammazione. Nei sistemi a time sharing la CPU esegue più job switchando tra questi, ma gli switch avvengono così frequentemente che l'utente può interagire con ciascun programma mentre questo è in esecuzione.

Il time sharing necessita di un computer interattivo che fornisca una comunicazione diretta tra l'utente e il sistema e a sua volta deve fornire un tempo di risposta sufficientemente corto (< 1 secondo).

PROCESSO

Un programma caricato in memoria e in esecuzione si chiama **processo**.

I sistemi a time sharing e multiprogrammati necessitano di mantenere più job in memoria alla volta. Essendo solitamente la memoria centrale troppo piccola per contenere tutti i job, questi vengono tenuti inizialmente in memoria secondaria all'interno del cosiddetto **job pool**.

Il job pool consiste di tutti i processi che risiedono in memoria secondaria e che aspettano di essere allocati in memoria centrale. Se tanti job sono pronti per essere portati in memoria centrale, e lo spazio non basta per tutti, allora il sistema deve effettuare delle scelte.
L'effettuazione di queste scelte è chiamata **job scheduling**.

Quano il sistema operativo sceglie un job dal job pool, lo carica in memoria per l'esecuzione. Avere molti programmi in memoria nello stesso momento necessita di una gestione della memoria.

Se molti job son pronti ad essere eseguiti nello stesso momento, allora il sistema deve effettuare delle scelte.
L'effettuazione di queste scelte è chiamata **CPU scheduling**.

OPERAZIONI DEL SISTEMA OPERATIVO

I sistemi operativi moderni sono **interrupt driven**. Se non ci sono processi da eseguire, nessun dispositivo da servire e nessun utente al quale rispondere, un sistema operativo si mette ad attendere che accada qualcosa.

Gli eventi sono segnalati con l'occorrenza di un interrupt o di una trap (interrupt generato via software).

Considerato che il sistema operativo e gli utenti condividono sia le risorse hardware che software del sistema, dobbiamo assicurarci che un errore in un programma utente causi problemi solo al programma in esecuzione.

OPERAZIONE DUAL-MODE

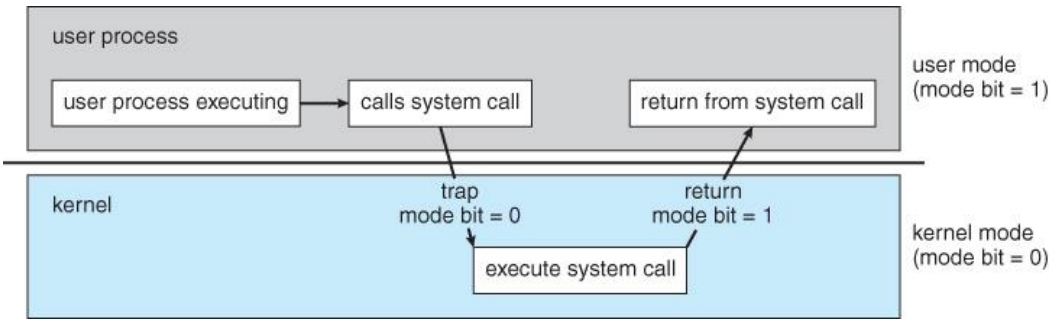
Per assicurare l'esecuzione corretta del sistema operativo, bisogna essere in grado di distinguere l'esecuzione di codice di sistema da quella di codice utente. Solitamente questa possibilità è fornita via hardware.

Abbiamo bisogno di due modalità:

- **Modalità utente**;
- **Modalità kernel**.

A fare la differenza è tra le due modalità è un bit (bit di modalità) che viene aggiunto all'hardware del computer per indicare la modalità corrente, 0 per il kernel e 1 per l'utente.

Grazie al bit di modalità sappiamo a che livello viene eseguito il task corrente.



INFORMAZIONI AGGIUNTIVE

- Sistema real time: specializzato per il supporto di applicazioni software real-time, ovvero quelle in cui è necessario ottenere una risposta dal sistema entro un tempo prefissato

DOMANDE

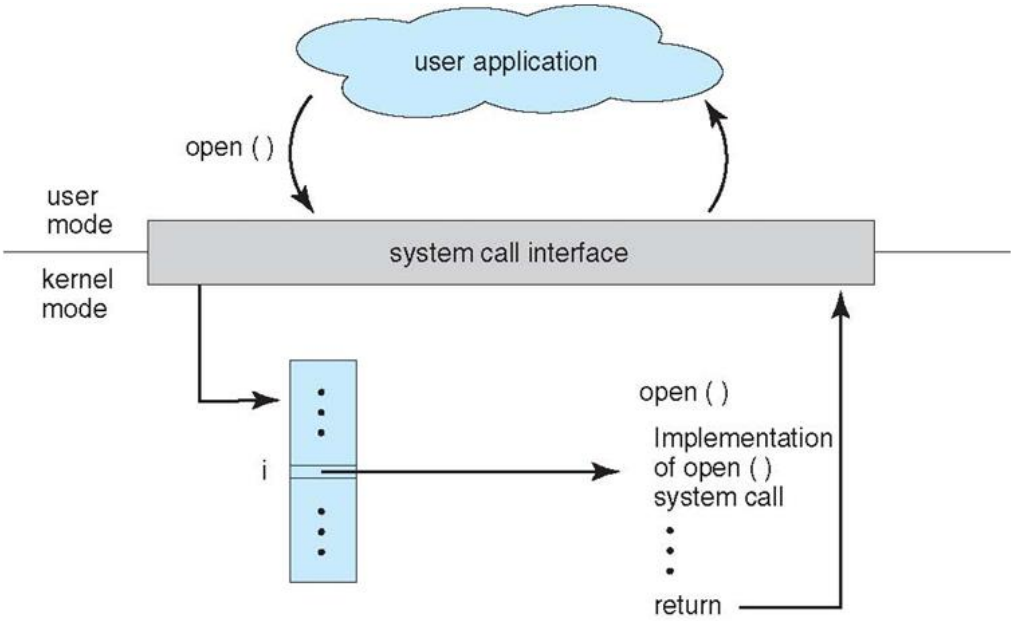
1. Descrivere le differenze fra funzionamento in modalità utente e modalità kernel del sistema.

STRUTTURE DEI SISTEMI OPERATIVI

CHIAMATE DI SISTEMA

Le **chiamate di sistema** forniscono un'interfaccia ai servizi resi disponibili dal sistema operativo.

Il sistema di supporto a runtime dei linguaggi di programmazione fornisce una system call interface che fa da collegamento alle chiamate di sistema rese disponibili dal sistema operativo.



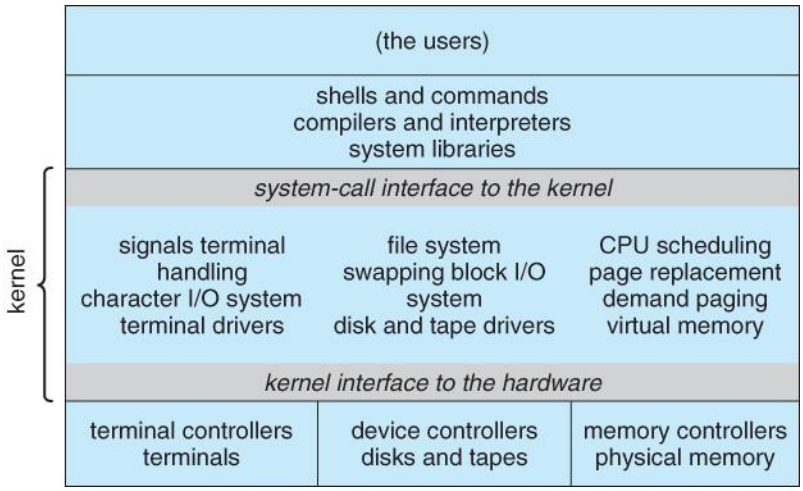
TIPOLOGIE DI CHIAMATE DI SISTEMA

Le chiamate di sistema possono essere raggruppate in 5 categorie:

- Controllo di processi:
  - End, abort
  - Load, execute
  - Create/terminate process
  - Get/set process attributes
  - Wait for time
  - Wait event, signal event
  - Allocate and free memory
- Manipolazione di file:
  - Create file, delete file
  - Open/close
  - Read/write
  - Get/set file attributes
- Manipolazione di dispositivi:
  - Request/release device
  - Read/write from/to device
  - Get/set device attributes
  - Attach/detach devices
- Manutenzione di informazioni:
  - Get/set time or date
  - Get/set system data

- Get/set process/file/device attributes

- Comunicazione:
  - Create/delete communication connection
  - Send/receive messages
  - Transfer status information
  - Attach/detach remote devices



KERNEL

In quali casi viene eseguito codice kernel (LINUX):

1. Avvio e setup del sistema;
2. Esecuzione di servizi per i processi in seguito a system call invocate dai processi stessi;
3. Eccezioni di esecuzione dei processi;
4. Attività di gestione del sistema triggerate da interrupt da periferica;
5. Processi del kernel.

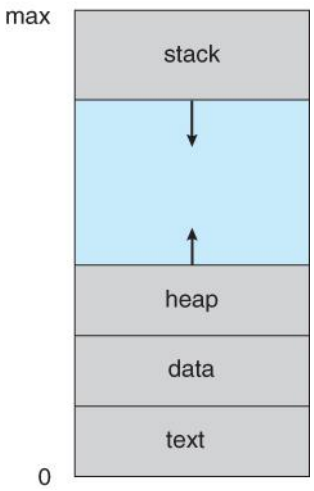
Il passaggio da modalità utente a modalità kernel funziona come segue:

1. Il programma in user-mode piazza dei valori nei registri per indicare quale servizio si richiede al sistema operativo;
2. Il programma in user-mode effettua l'istruzione **trap**;
3. La CPU passa immediatamente in modalità kernel e salta verso istruzioni in una locazione di memoria fissata iniziando l'esecuzione del trap/syscall handler;
4. Il system call handler legge i dettagli del servizio richiesto e gli argomenti, dopodiché porta a termine queste richieste in modalità kernel;
5. Una volta terminata la chiamata di sistema, il sistema operativo torna in modalità utente e restituisce un valore dalla chiamata di sistema (o si fanno tutte e due le cose assieme).

PROCESSI

CONCETTO DI PROCESSO

Un processo è un programma in esecuzione. In quanto tale non è composto solo dal codice (text section) bensì anche dai dati che rappresentano l'attività corrente (program counter, contenuto registri, process stack, variabili globali, heap, ecc).

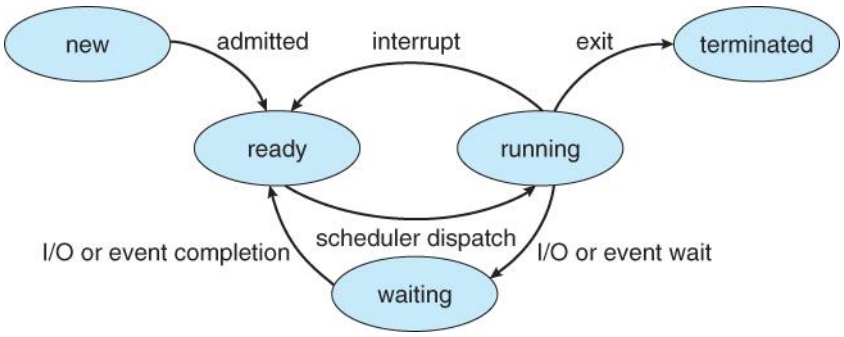


Un programma è un'entità passiva, mentre un processo è un'entità attiva.

STATO DEL PROCESSO

Ogni processo può essere in uno dei seguenti stati:

- **NEW** il processo è stato creato
- **RUNNING** le istruzioni stanno venendo eseguite
- **WAITING** si attende che accada qualche evento
- **READY** il processo attende di essere mandato in esecuzione
- **TERMINATED** il processo ha finito l'esecuzione



PROCESS CONTROL BLOCK

Il process control block è la rappresentazione in memoria dello stato del processo. Il PCB contiene:

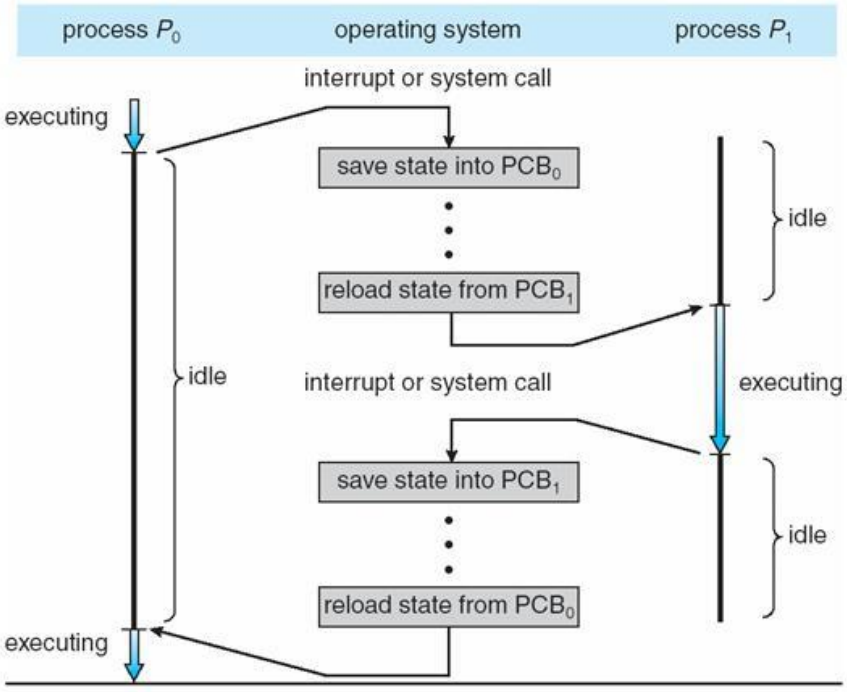
- Stato del processo;
- Program counter;
- Registri CPU;
- Informazioni per CPU-scheduling;
- Informazioni per memory-management;

- Informazioni di contabilità;
- Informazioni di stato I/O.

Nella prima versione del kernel Linux, il codice che rappresentava il PCB è il seguente:

```
struct task_struct {
/* these are hardcoded - don't touch */
        long state;   /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        fn_ptr sig_restorer;
        fn_ptr sig_fn[32];
/* various fields */
        int exit_code;
        unsigned long end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
        unsigned short used_math;
/* file system info */
        int tty;            /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
};
```

CONTEXT SWITCH



PROCESS SCHEDULING

L'obiettivo della multiprogrammazione è avere, in ogni momento, almeno un processo in esecuzione, in maniera tale da massimizzare l'utilizzo della CPU.

L'obiettivo del time sharing è switchare la CPU tra i vari processi così frequentemente da far sì che l'utente possa interagire con ogni programma mentre questo è in esecuzione.

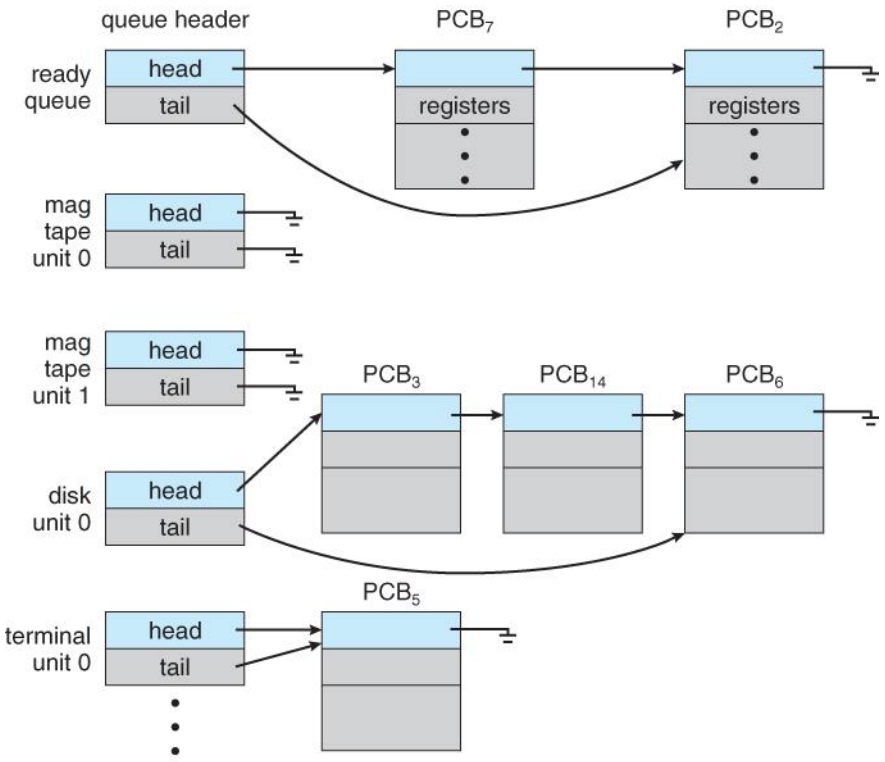
Per raggiungere questi obiettivi, il **process scheduler** seleziona un processo disponibile per essere eseguito nella CPU. Se la CPU è monoprocessore, allora potrà esserci in esecuzione un solo processo per volta, altrimenti di più.

CODE DI SCHEDULING

Appena i processi entrano nel sistema, vengono messi in una **coda di job**, che consiste in tutti i processi del sistema.

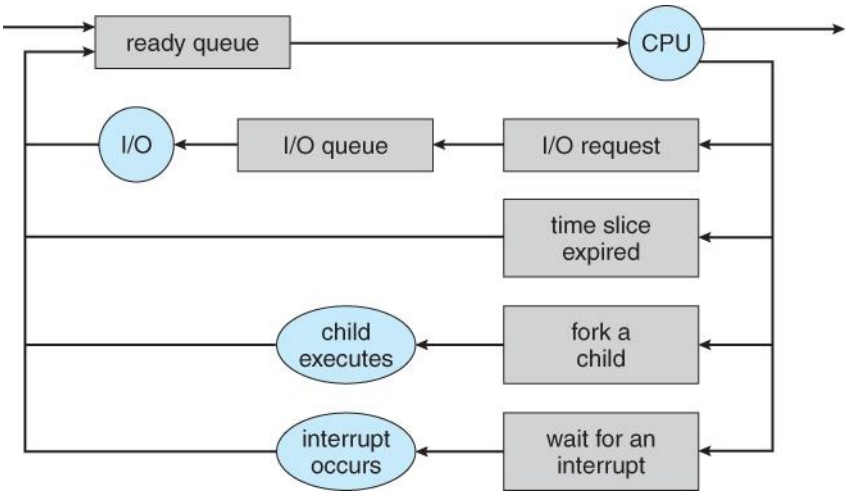
I processi che risiedono in memoria centrale e che son pronti per l'esecuzione vengono tenuti nella **coda di ready**. L'header della coda di ready contiene puntatori al primo e all'ultimo PCB della lista. Ciascun PCB in coda contiene un puntatore al successivo.

Quando un processo viene eseguito per un po' e poi passa in attesa del completamento di un'operazione di I/O, allora risiede nella **coda del dispositivo** dal quale si attende il completamento dell'operazione di I/O. Ogni dispositivo ha una sua coda.



Un nuovo processo viene inserito in coda di ready e attende finché non avviene il suo **dispatch** (viene scelto per l'esecuzione). Quando il processo è in esecuzione può succedere una delle seguenti cose:

- Il processo effettua richiesta di I/O e va in una coda di I/O;
- Il processo crea un nuovo sottoprocesso e si mette in attesa della sua terminazione;
- Il processo viene rimosso forzatamente dalla CPU a causa di un interrupt e viene rimesso in coda di ready (fine time slice).



SCHEDULER

Un processo, lungo il suo ciclo di vita, attraversa varie code di scheduling. In ognuna di queste è necessario un criterio attraverso il quale si seleziona un determinato processo. Questa selezione viene effettuata dallo **scheduler**.

In un sistema batch, si inviano più processi per l'esecuzione in un pool di processi e verranno eseguiti solo uno alla volta.

Lo **scheduler a lungo termine** seleziona i processo da questo pool e li carica in memoria per l'esecuzione.

Lo **scheduler a breve termine** seleziona tra uno tra i processi pronti ad essere eseguiti e gli alloca la CPU.

Lo scheduler a lungo termine effettua una scelta basata sulla tipologia di processo, che può essere:

- CPU-bound: passa la quasi totalità del tempo a effettuare calcoli sulla CPU;
- I/O-bound: spende molto tempo attendendo il completamento di operazioni di I/O.

È importante che lo scheduler a lungo termine mescoli bene processi CPU-bound e I/O-bound.

CONTEXT SWITCH

Quando avviene un interrupt, il sistema deve salvare il contesto del processo attualmente in esecuzione (il suo PCB) in maniera tale da poterlo ripristinare successivamente.

L'operazione di salvataggio del contesto si chiama **state save**, mentre l'operazione di ripristino si chiama **state restore**.

Il passaggio della CPU ad un altro processo richiede il **state save** del processo corrente e il **state restore** di un altro processo. Quest'operazione è chiamata **context switch** ed è puro overhead.

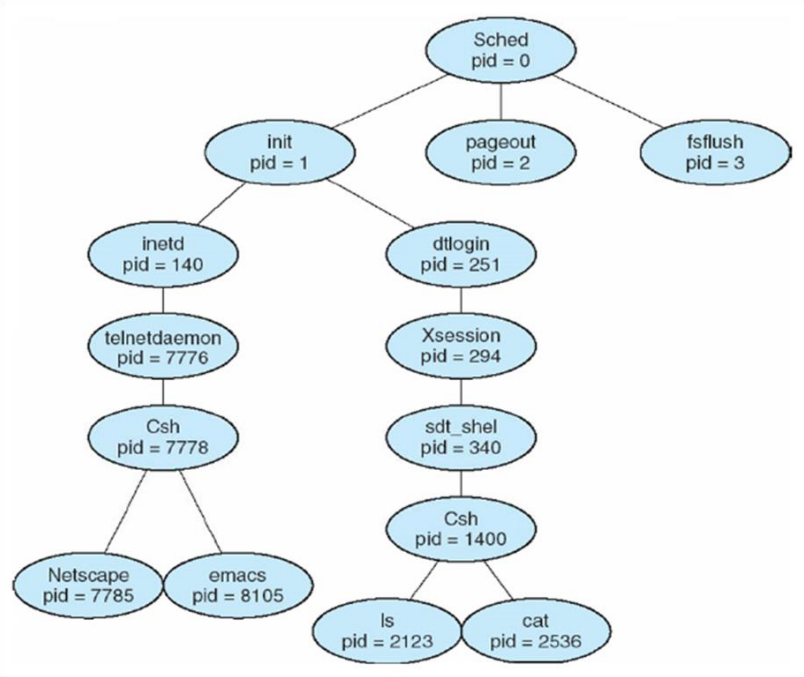
OPERAZIONI SUI PROCESSI

CREAZIONE

Un processo può creare più processi nuovi attraverso la chiamata di sistema apposita. Il processo creante è il processo **padre**, mentre i nuovi processi creati sono i **figli** di quel processo.

Ognuno di questi processi può a sua volta creare altri processi, formando così un **albero di processi**.

Un processo viene identificato attraverso un **process id** (pid), che solitamente viene rappresentato con un numero intero.



Quando un processo crea un altro processo, ci sono due possibilità in termini di esecuzione:

1. Il padre continua l'esecuzione in concorrenza col figlio;
2. Il padre aspetta che uno o più figli terminino la loro esecuzione.

Esistono due possibilità anche in termini di spazi degli indirizzi del nuovo processo:

1. Il processo figlio è un duplicato del padre (stesso programma e stessi dati);
2. Il processo figlio ha un nuovo programma caricato al suo interno.

Esempio di creazione di un processo in C:

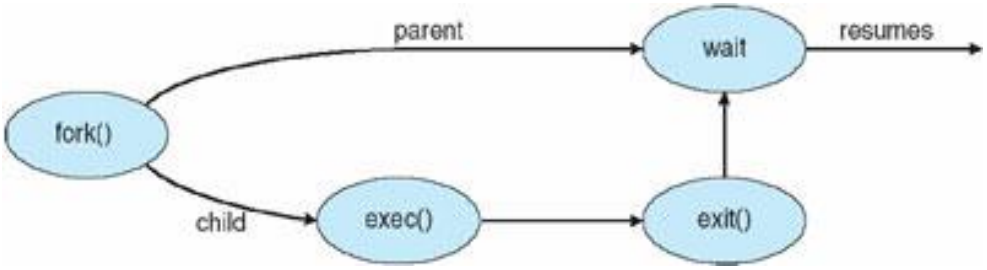
```
#include <stdio.h>
#include <unistd.h>

int main() {
    int id;
    printf("Hello, World!\n");

    id = fork();
    if(id > 0) {
        /* processo padre */
        printf("This is parent section [Process id: %d].\n", getpid());
    } else if(id == 0) {
        /* processo figlio */
        printf("fork created [Process id: %d].\n", getpid());
        printf("fork parent process id: %d.\n", getppid());
    } else {
        /* fork fallita */
        printf("fork creation failed!!!\n");
    }

    return 0;
}
```

Tipicamente, dopo l'esecuzione della fork() si utilizza exec() per rimpiazzare il programma del processo figlio con un altro programma, diverso da quello del padre.



TERMINAZIONE

Un processo termina quando finisce di eseguire la sua ultima istruzione e chiede al sistema operativo di essere eliminato utilizzando la system call exit(). A quel punto, il processo può restituire un valore al processo padre. Tutte le risorse del processo vengono deallocate dal sistema operativo.

Un padre può terminare l'esecuzione di un figlio per varie ragioni, tra cui:

- Il figlio ha usato più risorse del consentito;
- Il task assegnato al figlio non serve più;
- Il padre sta terminando e il sistema operativo non consente al figlio di proseguire se il padre termina.

INTERPROCESS COMMUNICATION

I processi in esecuzione concorrente su un sistema operativo possono essere **indipendenti** oppure **cooperanti**.

Un processo è **indipendente** se non viene condizionato da e non condiziona altri processi. Se ciò avviene, allora un processo è **cooperante**.

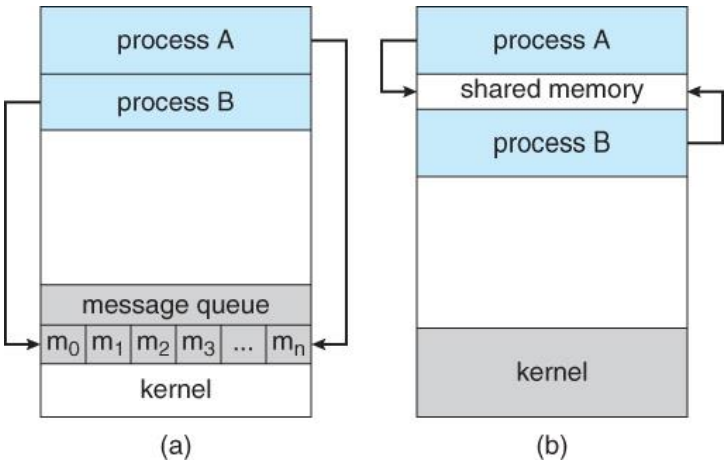
Ci sono vari motivi per fornire un ambiente che permette ai processi di comunicare:

- Condivisione di informazioni;
- Accelerazione dei calcoli;
- Modularità;
- Convenienza.

I processi cooperanti necessitano di un meccanismo di comunicazione interprocesso che gli permette di scambiare dati e informazioni.

Esistono due modelli fondamentali per la comunicazione interprocesso:

1. **Memoria condivisa**: i processi si scambiano i dati scrivendo e leggendo in una regione di memoria;
2. **Scambio di messaggi**.



SISTEMI A MEMORIA CONDIVISA

Solitamente una regione di memoria condivisa risiede nello spazio degli indirizzi del processo che la crea. Gli altri processi, per comunicare, devono "attaccare" questa regione di memoria al loro spazio degli indirizzi.

Considerato il problema produttore-consumatore, avremo un buffer di elementi che può essere riempito dal produttore e svuotato dal consumatore. Questo buffer starà nella regione di memoria condivisa.

Un produttore può produrre un elemento mentre il consumatore ne sta consumando un altro, ciò significa che produttore e consumatore devono essere **sincronizzati**.

Possono essere usati due tipi di buffer:

1. **Buffer illimitato**, ovvero senza limiti di dimensioni;
   Il consumatore potrebbe dover aspettare nuovi elementi mentre il produttore può produrne quanti ne vuole.
2. **Buffer limitato**, con dimensione fissata.
   Il consumatore aspetta se il buffer è vuoto mentre il produttore aspetta se il buffer è pieno.

SISTEMI A SCAMBIO DI MESSAGGI

Lo scambio di messaggi fornisce un meccanismo che permette ai processi di comunicare e di sincronizzare le loro azioni senza condividere lo stesso spazio degli indirizzi ed è utile nel calcolo distribuito, dove i processi comunicanti risiedono in diverse parti del mondo.

Un sistema a scambio di messaggi fornisce almeno due operazioni:

- send(message);
- receive(message).

Se due processi vogliono comunicare, devono inviare e ricevere messaggi tra loro, di conseguenza è necessario instaurare un **collegamento** tra i due.

Questo collegamento può essere implementato in vari modi:

- Comunicazione diretta o indiretta;
- Comunicazione sincrona o asincrona;
- Buffering automatico o esplicito.

NAMING

I processi che vogliono comunicare devono avere un modo per farsi riferimento l'un l'altro. Posso usare la comunicazione diretta o indiretta.

Con la comunicazione diretta, ogni processo che vuole comunicare deve esplicitamente indicare il mittente della comunicazione. In questo schema, send e receive sono definite come:

- send(P, message) invia messaggio al processo P
- receive(Q, message) ricevi un messaggio dal processo Q

Un collegamento in questo schema ha le seguenti proprietà:

- i processi devono conoscere l'identità degli altri per comunicarci;
- un collegamento è associato ad esattamente due processi;
- tra ogni coppia di processi dev'esserci esattamente un collegamento.

Questo schema esibisce *simmetria* nell'indirizzamento: ognuno deve conoscere il nome dell'altro per comunicare.

Una variante di questo schema impiega l'*asimmetria* nell'indirizzamento. Qui solo il mittente nomina il destinatario, mentre il destinatario non ha bisogno di indicare un mittente. Ciò significa che un processo può ricevere messaggi da qualunque altro processo.

Lo svantaggio di questi schemi è la limitata modularità delle definizioni di processo risultanti: cambiare l'id di un processo potrebbe portare a controllarli tutti (problema di hard-coding).

Con la comunicazione indiretta, i messaggi sono inviati e ricevuti da caselle di posta. In tal caso, send e receive son definite come:

- send(A, message) – invia un messaggio alla casella di posta A;
- receive(A, message) – ricevi un messaggio dalla casella di posta A.

Qui abbiamo la possibilità di associare un collegamento a più di due processi perché in una casella di posta possono esserci più processi in ascolto.

SINCRONIZZAZIONE

Partiamo sempre da send e receive. Ci sono diverse opzioni per implementare ciascuna primitiva. L'invio di un messaggio può essere **sincrono** o **asincrono**.

- **invio sincrono**: il processo mittente è bloccato finché il destinatario (casella di posta o processo) non riceve;
- **invio asincrono**: il processo mittente invia il messaggio e prosegue;
- **ricezione sincrona**: il ricevente si blocca finché un messaggio non è disponibile;
- **ricezione asincrona**: il ricevente ritrova un messaggio valido o null.

Sono possibili diverse combinazioni tra send e receive.

LE PIPE IN LINUX

Una **pipe** è un dispositivo di comunicazione che consente la comunicazione unidirezionale. I dati scritti da una parte vengono letti dall'altra parte del "tubo".

Le pipe sono dispositivi seriali, ovvero: i dati sono letti nello stesso ordine con cui vengono scritti.

La capacità di una pipe è limitata, e se il processo scrittore scrive più velocemente di quanto il lettore consuma i dati e la pipe non può più memorizzare dati, allora il processo scrittore si blocca finché non aumenta la capacità. Quindi la pipe sincronizza automaticamente i due processi comunicanti.

Una FIFO è una pipe che ha un nome nel file system. Per questo è chiamata anche *pipe con nome*.

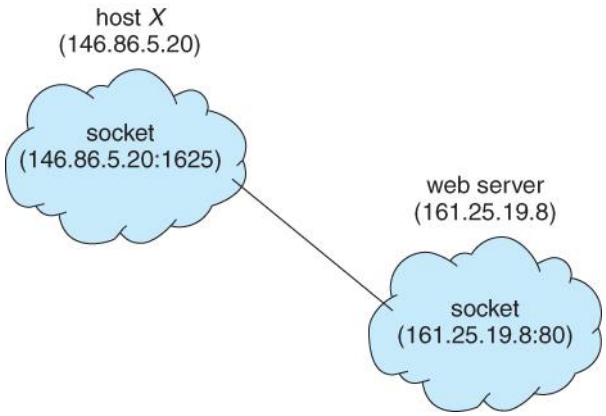
COMUNICAZIONE IN SISTEMI CLIENT-SERVER

SOCKET

Un socket è definito come estremo in una comunicazione. Due processi che comunicano in rete utilizzano due socket, uno per ciascuno identificati da un indirizzo IP concatenato ad un numero di porta.

Quando si crea un socket vanno definiti tre parametri:

- stile di comunicazione: determina come vengono gestiti e indirizzati i pacchetti dal mittente al destinatario;
- namespace: specifica come vengono scritti gli indirizzi dei socket (es. indirizzi IP);
- protocollo: specifica come avviene la trasmissione dei dati (es. TCP/IP, AppleTalk).



CHIAMATE DI PROCEDURA REMOTE

Si tratta di un paradigma che consente di astrarre il meccanismo di chiamata di procedura tra sistemi connessi in rete.

DOMANDE

1. Utilizzando il programma descritto nel seguito, identificare i valori assunti dalla variabile PID in corrispondenza delle linee A, B, C e D assumendo che i valori correnti dei pid del padre e del figlio siano rispettivamente 2600 e 2603:

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

int main() {
    pid_t pid, pid1;
    pid = fork();

    if(pid < 0) {
        fprintf(stderr, “Fork fallita.\n”);
        exit(-1);
    } else if(pid == 0) {
        pid1 = getpid();
        printf(“Figlio: pid = %d\n”, pid);   /* A */
        printf(“Figlio: pid1 = %d\n”, pid1); /* B */
    } else {
        pid1 = getpid();
        printf(“Padre: pid = %d\n”, pid);    /* C */
        printf(“Padre: pid1 = %d\n”, pid1);  /* D */
    }

    return(0);
}
```

2. Fare un esempio di una situazione in cui le pipe ordinarie sono migliori delle pipe con nome e un esempio in cui le pipe con nome sono migliori di quelle ordinarie.
3. Descrivi le azioni messe in atto dal kernel durante il context-switch tra processi.
4. Disegnare un possibile diagramma di accodamento per lo scheduling dei processi.
5. Elencare le informazioni codificate nel PCB di un processo e descriverne la funzione.
6. Quali sono i meccanismi tramite i quali un SO riceve o sottrae l’utilizzo della CPU a un processo? Per ciascun meccanismo, descriverne il significato.
7. Elencare le primitive di sistema e descriverne le caratteristiche relative alla implementazione della comunicazione per messaggi nei seguenti casi:
    a. comunicazione diretta;
    b. comunicazione indiretta.

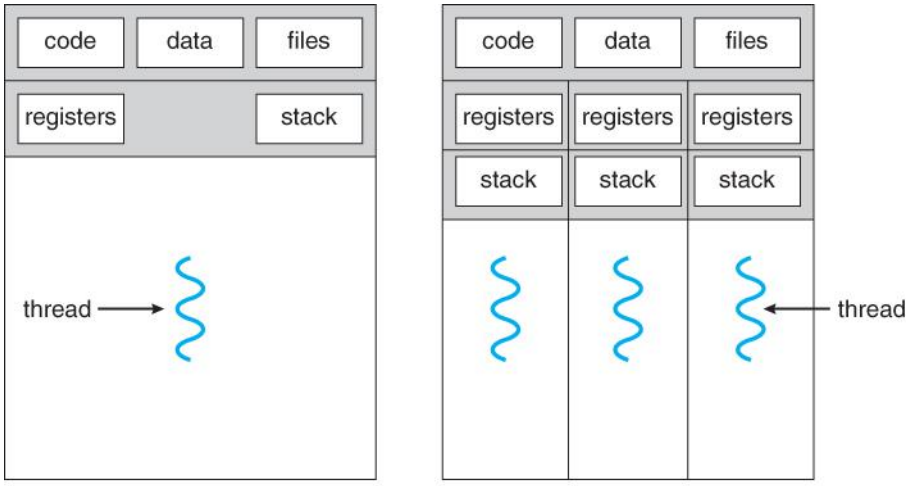
8. In quali casi vengono eseguite le kernel routines di Linux? Elencare e spiegare.
9. In quali casi due processi condividono la memoria in Linux? Elencare e spiegare.

THREADS

PANORAMICA

Un thread è un'unità base di utilizzazione della CPU. Comprende un ID, un program counter, un set di registri e uno stack. Condivide, assieme agli altri thread appartenenti allo stesso processo, la sezione codice, la sezione dati e altre risorse del sistema operativo.

Un processo può essere formato da uno o più thread (processo **single-threaded** o **multi-threaded**).



single-threaded process multithreaded process

BENEFICI

1. **Responsiveness**: il multithreading in un'applicazione interattiva fa sì che l'utente possa continuare a interagirci anche se questa sta facendo altro;
2. **Condivisione delle risorse**: un'applicazione può avere diversi thread all'interno dello spazio degli indirizzi;
3. **Economia**: allocare memoria e risorse per creare un processo è un'attività costosa, idem il context-switch. Con i thread questi costi si riducono di molto.
4. **Utilizzazione di architetture multiprocessore**: più thread possono essere eseguiti in parallelo.

MODELLI DI MULTITHREADING

Il supporto dei thread avviene a livello utente (user thread) e a livello kernel (kernel thread).

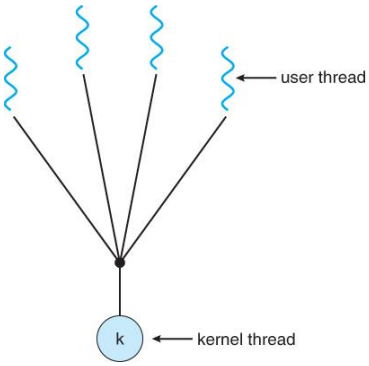
Gli user thread vengono gestiti senza il supporto del kernel, mentre i kernel thread sì.

Si può instaurare una relazione tra le due tipologie di thread in base all'utilizzo che se ne fa.

MODELLO MANY-TO-ONE

Più thread a livello utente vengono mappati a un solo kernel thread. La gestione dei thread viene effettuata da una libreria a livello utente, quindi è efficiente. Il problema è che un blocco in un thread (es. stato di wait o esecuzione system call) implica un blocco dell'intero processo.

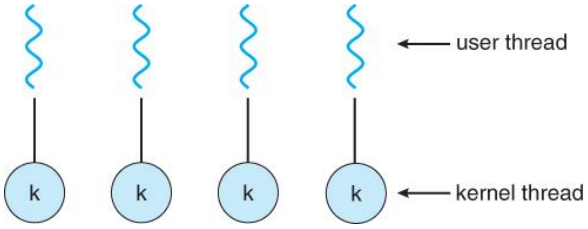
Altro problema è il non poter sfruttare l'esecuzione in parallelo perché solo l'accesso alla CPU sarà fatto da un solo thread per volta.



Modello Many-to-One

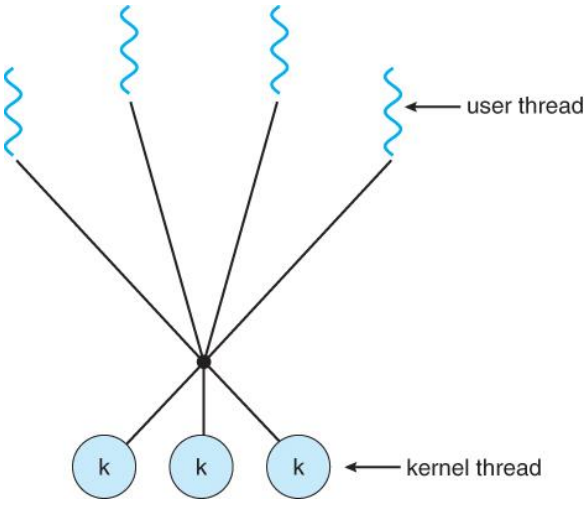
MODELLO ONE-TO-ONE

Ogni user thread è mappato a un kernel thread, ciò significa che il parallelismo multiprocessore può essere sfruttato così come si può evitare il blocco dell'intero processo in caso di system call da parte di un thread (verranno eseguiti gli altri thread). Si ha overhead che può influire pesantemente sulle prestazioni.



Modello One-to-One

MODELLO MANY-TO-MANY



Modello Many-to-Many

DIFFICOLTÀ LEGATE AI THREAD

CANCELLAZIONE DI UN THREAD

La cancellazione di un thread è il task di terminazione di un thread prima del suo completamento.

Esempio: se si effettua una ricerca su più thread all'interno di un database, allora ci si fermerà appena si trova il risultato cercato, cancellando tutti i thread. La cancellazione di un thread target può avvenire in due modi:

1. **cancellazione asincrona**: un thread termina immediatamente il thread target;
2. **cancellazione rinviata**: il thread target controlla periodicamente se deve terminare o meno, facendolo se è il caso.

Le difficoltà relative alla cancellazione si hanno con la gestione delle risorse assegnate a un thread (potrebbero non venire completamente rilasciate) e con la cancellazione di un thread nel mentre che questo aggiorna dei dati condivisi (possibile non coerenza dei dati).

Diventa quindi necessario definire dei **punti di cancellazione**.

POOL DI THREAD

In una situazione come la gestione delle richieste di un server web potrebbe diventare troppo dispendioso aprire e chiudere thread ogni volta che si gestisce una richiesta:

- overhead dovuto alla creazione dei thread;
- non esiste un limite sul numero di thread, quindi è facile che di fronte a richieste più alte del previsto si possano esaurire le risorse di sistema.

La soluzione è l'utilizzo di un **pool di thread**.

L'idea è quella di avere un certo numero di thread sempre in esecuzione che attendono l'arrivo delle richieste e che non si chiudono una volta che la richiesta viene esaudita. Questa soluzione offre i seguenti benefici:

- non c'è overhead dovuto alla creazione di un thread;
- esiste un limite al numero di thread.

Il numero di thread facenti parte del pool possono essere settati in base a tecniche euristiche.

DOMANDE

1. Sotto quali circostanze una soluzione multithread che utilizza più kernel thread fornisce prestazioni migliori di una soluzione single-thread su un computer con processore singolo?
2. Quali dei seguenti componenti dello stato di un processo sono condivisi fra i thread in un processo multithread?
    a. Valori dei registri;
    b. Memoria heap;
    c. Variabili globali;
    d. Memoria stack.
3. Può, una soluzione multithread che utilizza più thread a livello utente, ottenere prestazioni migliori in un sistema multiprocessore rispetto a quelle che otterrebbe in un sistema a processore singolo?
4. Si consideri un sistema multiprocessore e un programma multithread scritto utilizzando il modello di threading many-to-many, e sia questo l'unico processo in esecuzione. Sia il numero di thread a livello utente maggiore del numero di processori nel sistema. Si discutano le implicazioni nelle prestazioni nei seguenti scenari:
    a. il numero di kernel thread è minore rispetto al numero di processori;
    b. il numero di kernel thread è uguale al numero di processori;
    c. il numero di kernel thread è maggiore del numero di processori ma minore rispetto al numero di thread a livello utente.
5. Elencare vantaggi e svantaggi reciproci delle implementazioni dei thread a livello utente e a livello kernel.
6. Argomentare relativamente ai pool di thread.

SCHEDULING DELLA CPU

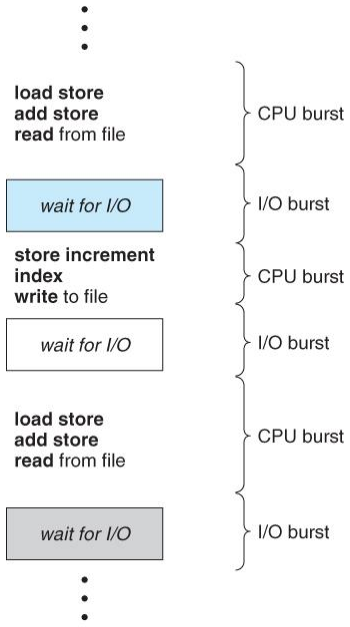
CONCETTI DI BASE

In un sistema a processore singolo può essere in esecuzione un solo processo per volta. Gli altri devono aspettare che la CPU si liberi e che questa venga schedulata nuovamente.

L'obiettivo della multiprogrammazione è avere un processo in esecuzione in qualunque momento in modo da massimizzare l'utilizzo della CPU, e lo scheduling è un tema centrale nella progettazione di un sistema operativo.

CICLO DI BURST (CPU-I/O)

I processi consistono di un ciclo di esecuzione su CPU e un ciclo di attesa di I/O. I processi si alternano in questi due stati.
L'esecuzione inizia con un CPU burst, è seguita da un I/O burst che a sua volta è seguita da un CPU burst e così via finché non si conclude l'esecuzione.



In un programma I/O-bound solitamente si hanno molti brevi burst di CPU, mentre un programma CPU-bound avrà pochi ma lunghi burst di CPU.

SCHEDULER DELLA CPU

Ogni volta che la CPU va in idle, il sistema deve scegliere un processo in coda di ready da eseguire. Questo lavoro viene effettuato dallo **scheduler a breve termine**. Si noti che la coda di ready non è necessariamente FIFO.

SCHEDULING PREEMPTIVE

Le decisioni di scheduling della CPU possono avere effetto sotto le seguenti circostanze:

1. Quando un processo passa da running a wait (es. richiesta di I/O);
2. Quando un processo passa da running a ready (es. avviene interrupt);
3. Quando un processo passa da waiting a ready (es. completamento I/O);
4. Quando un processo termina.

Per le situazioni 1 e 4 non si fanno scelte in termini di scheduling, un nuovo processo va selezionato per l'esecuzione. Si effettua una scelta nella situazioni 2 e 3.

Quando lo scheduling avviene solo sotto le circostanze 1 e 4, allora lo schema di scheduling è **nonpreemptive**, altrimenti è **preemptive**.

Con lo scheduling nonpreemptive, una volta che la CPU viene allocata ad un processo, questo la tiene finché non la rilascia di sua volontà terminando o passando in waiting.

Con lo scheduling preemptive questo non succede ma ci sono dei costi in termini di hardware particolare richiesto e in termini di costi di accesso ai dati condivisi (diventa necessario sincronizzare gli accessi).

CRITERI DI SCHEDULING

- **Utilizzazione della CPU**: vogliamo tenere la CPU occupata il più possibile;
- **Throughput**: numero di processi completati per unità di tempo;
- **Tempo di turnaround**: somma del tempo che un processo impiega tra attesa di entrare in memoria, attesa in ready, esecuzione nella CPU e I/O;
- **Tempo di attesa**: somma dei periodi trascorsi in attesa sulla coda di ready;
- **Tempo di risposta**: tempo impiegato per rispondere all'input dell'utente (sistema interattivo).

ALGORITMI DI SCHEDULING

SCHEDULING FIRST-COME, FIRST-SERVED

Il processo che richiede la CPU per primo viene allocato per primo, tant'è che viene implementato con una semplice coda FIFO.

Il **tempo medio di attesa** è molto lungo.

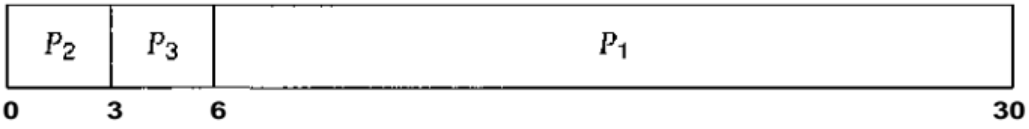
Si considerino i seguenti processi che arrivano in tempo 0.

PROCESSO	TEMPO DI BURST
P_1	24
P_2	3
P_3	3

Se i processi arrivano in ordine P_1, P_2 e P_3, si ottiene il seguente **diagramma di Gantt**:

P_1 avrà un tempo di attesa di 0 ms, P_2 di 24 ms e P_3 di 27. Il tempo medio di attesa è $\frac{0+24+27}{3} = 17ms$

Se i processi fossero invece arrivati in ordine P_2, P_3 e P_1, allora il diagramma sarebbe stato il seguente:



Il tempo medio di attesa sarebbe invece stato $\frac{6+0+3}{3} = 3ms$.

Si può affermare che il FCFS non ottimizza i tempi di attesa e non è preemptive (il processo tiene la CPU finché non la rilascia di sua volontà).

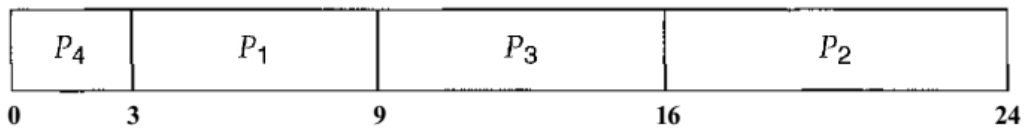
SCHEDULING SHORTEST-JOB-FIRST

Questo algoritmo associa ad ogni processo la lunghezza del suo prossimo burst di CPU. Quando la CPU è disponibile, viene associata al processo che ha il prossimo burst di CPU più piccolo. Se questo valore è uguale per due processi, allora entra in gioco il FCFS.

Esempio:

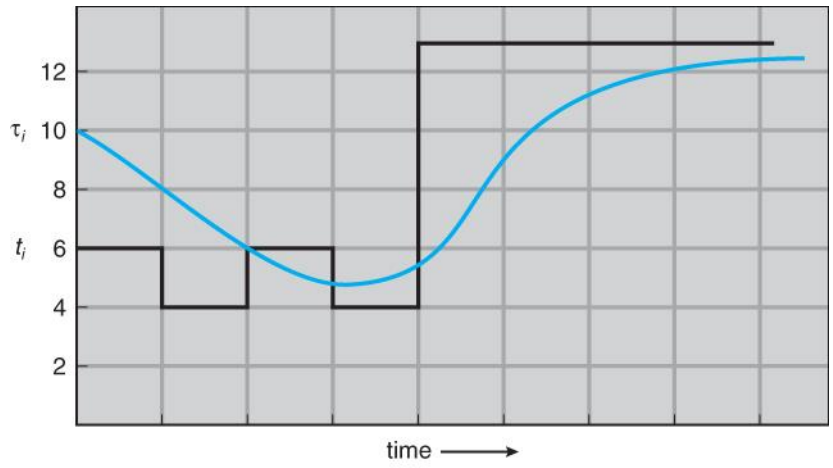
PROCESSO	TEMPO DI BURST
P_1	6
P_2	8
P_3	7
P_4	3

Il seguente è il diagramma di Gantt risultante:



Il tempo medio di attesa è $\frac{3+16+9+0}{4} = 7ms$.

L'algoritmo SJF è *ottimale* (tempo di attesa medio minimo dato un insieme di processi), il problema sta nel fatto che non esiste modo per conoscere la durata del prossimo burst di CPU, sebbene esista un modo per predirne il valore basandoci su quelli dei burst precedenti.



CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

L'algoritmo SJF può essere preemptive oppure no. La preemption può arrivare nel momento in cui un nuovo processo entra nella coda di ready e qui si rivaluta lo scheduling in base ai nuovi valori.

La versione preemptive dell'algoritmo prende il nome di **shortest-remaining-time-first-scheduling**.

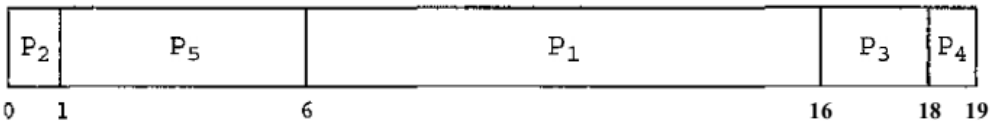
SCHEDULING PER PRIORITÀ

Ad ogni processo viene associata una priorità e la CPU viene allocata al processo con la priorità più alta. Solitamente numeri bassi indicano priorità più alta.

Esempio:

PROCESSO	TEMPO DI BURST	PRIORITÀ
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando lo scheduling per priorità si ottiene il seguente diagramma di Gantt:



Il tempo di attesa medio è 8.2 ms.

Come il SJF, anche qui si può avere preemption con tutto ciò che ne consegue.

Un problema che nasce con lo scheduling per priorità è la **starvation**, che viene risolta attraverso l'*invecchiamento*, ovvero l'aumento della priorità man mano che aumenta il tempo trascorso in coda di ready.

SCHEDULING ROUND-ROBIN

Lo scheduling RR è progettato appositamente per i sistemi time-sharing. È simile al FCFS ma è presente la preemption. Viene definita una piccola unità di tempo chiamata **quanto di tempo**.

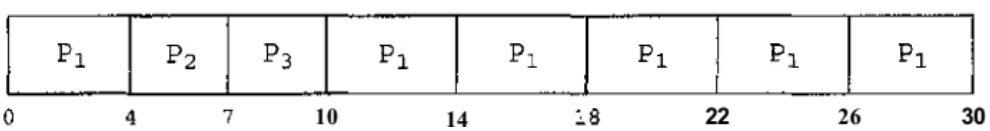
Lo scheduler alloca la CPU a un processo per un tempo che al massimo corrisponde al quanto di tempo, superato il quale viene rimosso dall'esecuzione per tornare alla fine della coda di ready.

Se la durata del burst è minore di quella del quanto di tempo, allora il processo stesso rilascia la CPU tornando alla fine della coda di ready. Il tempo di attesa medio è solitamente abbastanza alto.

Esempio:

PROCESSO	TEMPO DI BURST
P_1	24
P_2	3
P_3	3

Se usiamo un quanto di tempo di 4 ms, allora il processo P_1 si prende i primi 4 ms, dopo i quali viene rimosso a causa della preemption tornando alla fine della coda di ready. Il resto è abbastanza banale osservando il disegno:

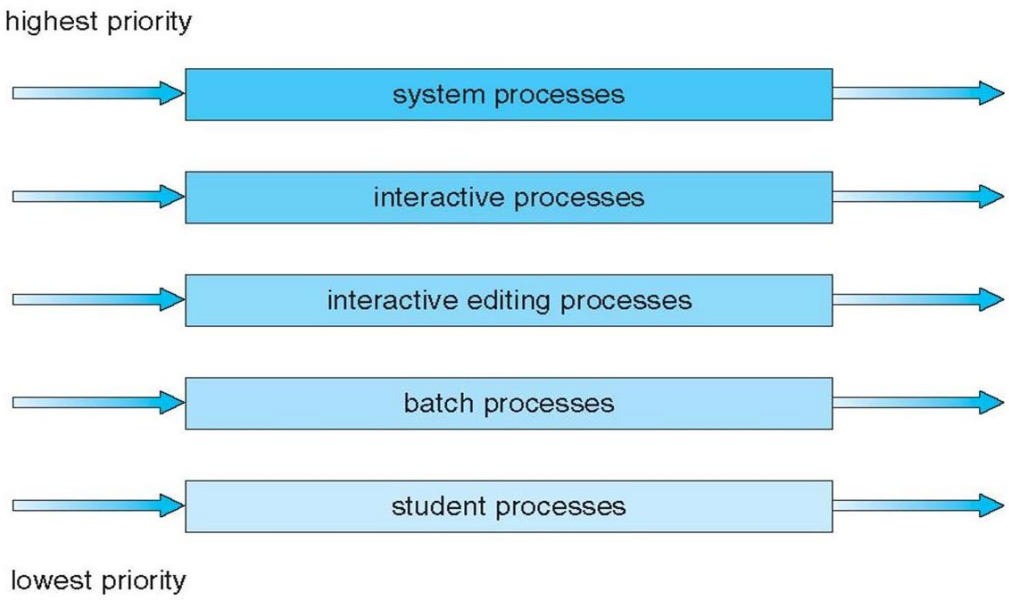


Il tempo di attesa medio è di 5.66 ms.

Le prestazioni del RR dipendono molto dalla dimensione del quanto di tempo. Se il quanto di tempo è troppo corto, allora si ha overhead dovuto ai numerosi context switch. Se il quanto di tempo è troppo alto si ha un tempo di turnaround troppo basso.

SCHEDULING SU CODE MULTILIVELLO

Lo scheduling si divide su più code suddivise per tipologia di processi che avranno al loro interno. Ogni coda può avere diverse necessità e di conseguenza algoritmi e priorità diversi.



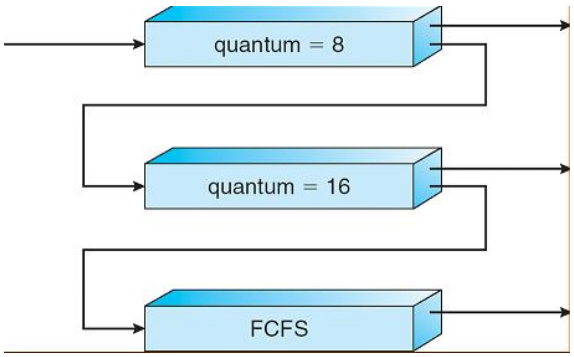
Ogni coda ha la sua priorità, ciò significa che avverrà preemption su processi di priorità più bassa nel caso uno di priorità più alta sia pronto ad essere eseguito.

Un'altra possibilità, complementare alla suddivisione delle code per priorità, è dare un **time-slice** a ciascuna coda in maniera tale da concedere a ogni coda una percentuale di tempo di CPU.

SCHEDULING SU CODE MULTILIVELLO CON FEEDBACK

A differenza dello scheduling su code multilivello semplice, questo permette a un processo di essere spostato da una coda all'altra. L'idea è quella di suddividere i processi in base alle caratteristiche dei loro burst di CPU, ciò implica quindi la dinamicità delle code.

Se un processo utilizza troppo tempo di CPU, allora verrà spostato su una coda di priorità più bassa. Questo schema lascerà i processi I/O-bound sulle code di priorità più alta.



In generale, lo scheduling su code multilivello con feedback è definito dai seguenti parametri:

- numero di code;
- algoritmo di scheduling utilizzato su ciascuna coda;
- criterio utilizzato per innalzare la priorità di un processo;
- criterio utilizzato per abbassare la priorità di un processo;
- criterio utilizzato per determinare in quale coda entrerà un processo è pronto per essere eseguito.

DOMANDE

1. Si considerino i seguenti processi:

PROCESSO	TEMPO DI BURST	PRIORITÀ
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

   Si assuma che arrivino in ordine P_1, P_2, P_3, P_4, P_5 al tempo 0.
   a. disegnare quattro diagrammi di Gantt illustrando l'esecuzione di questi processi con l'utilizzo degli algoritmi FCFS, SJF, scheduling per priorità non preemptive e round-robin (quanto di tempo = 1ms).
   b. Qual è il tempo di turnaround per ciascun processo per ognuno degli algoritmi di scheduling?
   c. Qual è il tempo di attesa per ciascun processo per ognuno degli algoritmi di scheduling?
   d. Quale algoritmo di scheduling assicura il minor tempo di attesa medio?
2. Discutere come confliggono i seguenti criteri di scheduling:
   a. utilizzazione della CPU e tempo di risposta;
   b. tempo medio di turnaround e tempo massimo di attesa;
   c. utilizzazione dei dispositivi di I/O e della CPU.
3. Elencare e commentare i possibili criteri di scheduling e le funzioni di costo utilizzate.
4. Descrivere l'algoritmo di scheduling round-robin ed argomentare relativamente ai suoi parametri di funzionamento.
5. Descrivere ed argomentare relativamente all'inversione delle priorità, anche utilizzando opportuni diagrammi di Gantt.

SINCRONIZZAZIONE DEI PROCESSI

Un processo cooperante è un processo in grado di influenzare e di essere influenzato da altri processi. Dei processi cooperanti collaborano attraverso aree di memoria condivise o attraverso lo scambio di messaggi.

L'accesso concorrente ai dati condivisi può portare all'inconsistenza dei dati stessi, per questo occorrono dei meccanismi che consentano di assicurare un'esecuzione corretta dei processi cooperanti.

PRODUTTORE-CONSUMATORE

Abbiamo un buffer condiviso da due processi secondo il paradigma produttore-consumatore, in cui abbiamo un processo produttore e un processo consumatore.

Quello che segue è il codice del produttore:

```
while(true) {
    while(counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Quello che segue è invece il codice del consumatore:

```
while(true) {
    while(counter == 0);
    nextConsumed = buffer[out];
    out += (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Sebbene le routine, prese singolarmente, siano corrette, non è detto che il funzionamento sia quello previsto. Se lo scheduler decidesse di interrompere uno dei due processi prima della fine del codice, è facile che i dati sul buffer vengano scritti/letti dall'altro processo, portando a uno stato di **inconsistenza dei dati**.

Occorre quindi, quando si accede ai dati, una **sincronizzazione** tra i processi.

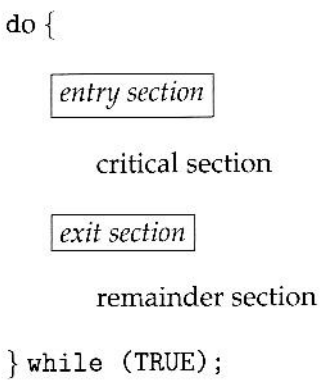
PROBLEMA DELLA SEZIONE CRITICA

Si consideri un sistema consistente in un insieme di n processi $\{P_0, P_1, \ldots, P_{N-1}\}$. Ciascun processo ha un segmento di codice chiamata **sezione critica** in cui il processo effettua operazioni su dati condivisi.

L'importante caratteristica di questo sistema è che, quando un processo sta eseguendo la propria sezione critica, allora nessun altro processo sta eseguendo la sua. Quindi due processi non possono eseguire la loro sezione critica nello stesso momento.

Il problema della sezione critica consiste nel trovare un protocollo utilizzabile dai processi per cooperare.

Ogni processo deve chiedere il permesso per entrare nella sua sezione critica.
La componente di codice che si occupa di chiedere il permesso si chiama **entry section** e verrà seguita, alla fine dei lavori, dalla **exit section**. Dopo la exit section viene eseguita la **remainder section** (sezione rimanente).



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

1. **Esclusione mutua**: solo un processo per volta può eseguire la sua sezione critica;
2. **Progresso**: se nessun processo sta eseguendo la sua sezione critica e alcuni processi vogliono entrarci, solo quelli che ancora non si son trovati nella remainder section potranno partecipare alla decisione su chi sarà il prossimo a entrare in sezione critica, e questa decisione non può essere rimandata indefinitamente;
3. **Attesa limitata**: deve esistere un limite sul numero di volte che altri processi possano entrare nella loro sezione critica dopo che un processo ha fatto richiesta per entrarci e prima che la richiesta sia soddisfatta (un processo non può entrare e uscire di continuo, bensì deve lasciar entrare anche altri processi).

L'obiettivo è far sì che due processi non si trovino in **race conditions**, ovvero il risultato della cooperazione non deve variare in base a "chi arriva prima sui dati".

Ci sono due approcci per la gestione del problema:

1. **kernel preemptive**: può avvenire preemption su un processo anche se questo sta venendo eseguito in modalità kernel;
2. **kernel non-preemptive**: non può esserci preemption su un processo in modalità kernel.

SOLUZIONE DI PETERSON

Due processi si alternano nell'esecuzione delle loro sezioni critiche e delle loro sezioni restanti. È necessario che tra i processi vengano condivisi due elementi, ovvero:

```
int turn; // indica a chi tocca entrare in sezione critica
bool flag[2]; // indica se un processo è pronto a entrare in sezione critica
```

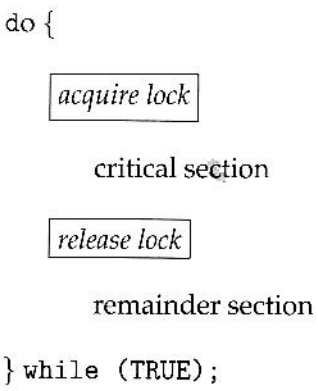
Per entrare in una sezione critica, il processo P_i prima setta flag[i] a true e poi setta turn al valore j, asserendo così che anche l'altro processo voglia entrare in sezione critica e che può farlo "se vuole".

```
do {
    flag[i] = true;
    turn = j; // assumo che tocchi all'altro processo
    while(flag[j] && turn == j); // attendo che l'altro processo termini la sua sezione critica
    <sezione critica>
    flag[i] = false; // faccio sapere che sono uscito dalla sezione critica
    <sezione restante>
} while(true);
```

La soluzione è corretta perché vengono soddisfatti i requisiti di cui sopra (esclusione mutua, progresso e attesa limitata).

SINCRONIZZAZIONE HARDWARE

Le race conditions vengono prevenute facendo sì che le sezioni critiche vengano protette da un lock. Il lock viene acquisito prima di entrare in sezione critica e rilasciato all'uscita. La sua utilità è quella di far sì che gli altri processi non entrino in sezione critica.



L'implementazione di un lock può essere abbastanza sofisticata e spesso è necessario effettuarla via hardware.

In un ambiente a singolo processore si potrebbe gestire il problema disabilitando gli interrupt in maniera tale che un processo non venga interrotto (nonpreemption necessaria) e che altri processi non tocchino le variabili condivise. Questa soluzione non è fattibile su ambienti multiprocessore perché si perderebbero troppi cicli macchina.

Molti sistemi moderni forniscono via hardware delle **istruzioni atomiche**, ovvero non interrompibili. Una di queste è la **test&set**.

TEST-AND-SET

Definizione:

```
bool testAndSet(bool * target) {
    bool registerValue = *target;
    *target = true;
    return registerValue;
}
```

Utilizzo:

```
do {
    while(testAndSet(&lock));      // attesa attiva finché non viene restituito false
    <sezione critica>
    lock = false;                  // avviene unlock, quindi un altro processo può entrare
    <sezione restante>
} while(true);
```

Non viene soddisfatta l'attesa limitata perché se un processo è "fortunato" può eseguire la sua sezione critica tante volte di fila, mentre agli altri tocca aspettare.

SEMAFORI

Un semaforo S è una variabile intera alla quale dopo l'inizializzazione si accede solo attraverso due operazioni **atomiche**:

- wait(): decrementa S se è maggiore di zero, altrimenti aspetta che lo diventi;
- signal(): incrementa S.

I semafori possono essere di due tipi:

- **semaforo binario**: può avere solo valori 0 e 1, è detto anche mutex (**mut**ual **ex**clusion);
- **semaforo contatore**: può avere interi qualunque (nella versione con attesa attiva solo interi non negativi).

L'entrata nella sezione critica è avviene attraverso la wait() mentre l'uscita avviene con la signal().

Si può intendere che il valore del semaforo indica quanti processi possono entrare in sezione critica tant'è che decrementa quando uno entra e si incrementa quando uno esce. Vien facile anche capire in che modo il semaforo binario fornisce l'esclusione mutua.

IMPLEMENTAZIONE

Esistono due possibili implementazioni della funzione wait().

Quella più brutta prevede che il processo attenda che S si incrementi semplicemente con un while, causando così attesa attiva detta, in questo caso, **spinlock**.

Un'implementazione intelligente è quella che prevede che il processo vada in waiting con la funzione **block()**, e che venga poi risvegliato con l'operazione **wakeup()** che fa sì che il processo torni in ready appena viene eseguita la signal(). Così facendo si evita lo spreco di cicli di CPU.

Implementazione wait:

```
wait(Semaphore *s) {
    s->value--;
    if(s->value < 0) {
        aggiungi questo processo in coda al semaforo (s->list)
        block(); // viene bloccato il processo e mandato in waiting
    }
}
```

Va notato che il valore del semaforo può essere negativo. In tal caso, il suo valore assoluto indica il numero di processi in attesa che si liberi il semaforo.

Implementazione signal:

```
signal(Semaphore *s) {
    s->value++;
    if(s->value <= 0) {
        rimuovi un processo P dalla coda del semaforo
        wakeup(P); // P si risveglia e torna in ready
    }
}
```

DEADLOCK E STARVATION

L'implementazione di un semaforo con una coda di attesa può portare a una situazione in cui due o più processi attendono indefinitamente un evento che può essere causato solo da uno dei processi in attesa (in questo caso la signal()). In tal caso si dice che si verifica un **deadlock**.

Con un'implementazione LIFO della lista dei processi in attesa al semaforo potremmo provocare **starvation** di uno o più processi.

PROBLEMI CLASSICI DI SINCRONIZZAZIONE

PRODUTTORE-CONSUMATORE CON BUFFER LIMITATO

Abbiamo due semafori:

- empty: conta il numero di elementi vuoti del buffer ed è inizializzato a n (numero massimo di elementi);
- full: conta il numero di elementi del buffer riempiti ed è inizializzato a 0.

Implementazione del produttore:

```
do {
    ...
    // produce un elemento in nextp
    ...
    wait(empty);            // si aspetta che ci siano elementi da riempire
    wait(mutex);            // c'è un posto da riempire, aspettiamo che termini il consumatore
    ...
    // aggiunge nextp al buffer
    ...
    signal(mutex);          // c'è sicuramente un elemento che può essere consumato
    signal(full);           // facciamo sapere che c'è un elemento pieno in più
} while(true);
```

Implementazione del consumatore:

```
do {
    wait(full);             // si aspetta che ci sia almeno un elemento
    wait(mutex);            // dobbiamo consumare, aspettiamo che termini il produttore
    ...
    // si rimuove dal buffer un elemento nextc
    ...
    signal(mutex);          // abbiamo consumato un elemento
    signal(empty);          // facciamo sapere che c'è un elemento vuoto in più
    ...
    // viene consumato l'elemento nextc
    ...
} while(true);
```

LETTORI E SCRITTORI

L'accesso a un database viene condiviso tra numerosi processi concorrenti. Alcuni vogliono solo leggere il database (= nessuna modifica), altri ci vogliono scrivere (= modifiche).

Se ci fossero solo lettori non ci sarebbero problemi nell'effettuare tutti gli accessi che si vogliono, mentre per gli scrittori è chiaro che debba essercene uno solo per volta. Il problema è che un lettore non deve poter accedere quando sta avvenendo una scrittura perché i dati potrebbero essere inconsistenti.

È necessario quindi che gli scrittori abbiano accesso esclusivo al database e questo si può fare con la sincronizzazione.

Dati:

```
Semaforo mutex; // inizializzato a 1, condiviso tra i lettori, serve a incrementare
                  contatoreLetture senza interferenze di altri processi
Semaforo scritture; // inizializzato a 1, ci hanno accesso sia lettori che scrittori, serve ad
                      assicurare esclusione mutua tra gli scrittori
int contatoreLetture; // inizializzato a 0
```

Struttura processo scrittore:

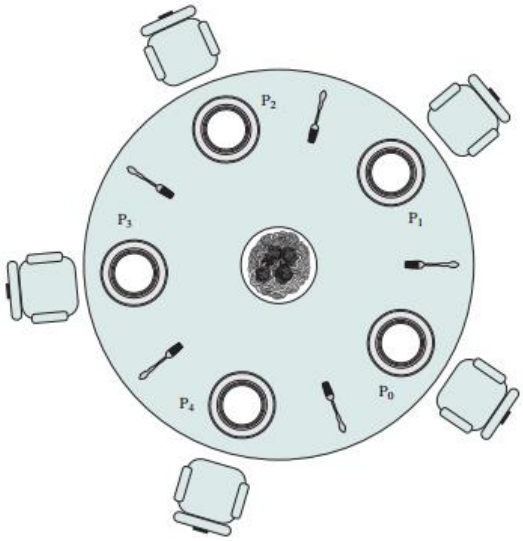
```
do {
    wait(scritture);
    ...
    // si effettuano scritture
    ...
```

```
    signal(scritture);
} while(true);
```

Struttura processo lettore:

```
do {
    wait(mutex);
    contatoreLetture++;
    if(contatoreLetture == 1) {
        wait(scritture);
    }
    signal(mutex);
    ...
    // si effettuano letture
    ...
    wait(mutex);
    contatoreLetture--;
    if(contatoreLetture == 0);
        signal(scritture);
    signal(mutex);
} while(true);
```

PROBLEMA DEI 5 FILOSOFI



Si verifica deadlock e conseguente starvation se ciascun filosofo inizia prendendo la forchetta che sta alla sua destra. Non potrà prendere quella che sta a sinistra perché sarà stata presa dal filosofo che sta alla sua sinistra. Una cosa simile può succedere con le risorse del sistema.

DOMANDE

1. Spiegare perché gli spinlocks non sono appropriati per sistemi multiprocessore mentre sono tollerabili su sistemi multiprocessore.
2. Descrivere due strutture dati del kernel nelle quali è possibile avere race conditions. Descrivere anche come può verificarsi una corsa critica.
3. Mostrare che, se le operazioni wait() e signal() del semaforo non vengono eseguite correttamente, allora l'esclusione mutua può essere violata.

DEADLOCK

Un processo potrebbe essere in attesa di una risorsa assegnata a un altro processo in attesa. Questa situazione è chiamata **deadlock**.

Un insieme di processi è in stato di deadlock quando ciascun processo dell'insieme è in attesa di un evento che può essere causato solo da un altro processo nell'insieme. Gli eventi di cui si parla sono acquisizione e rilascio di risorse.

Le risorse possono essere:

- fisiche (stampanti, spazio di memoria, cicli di CPU, ecc.);
- logiche (file, semafori e monitor).

CARATTERIZZAZIONE DI UN DEADLOCK

CONDIZIONI NECESSARIE

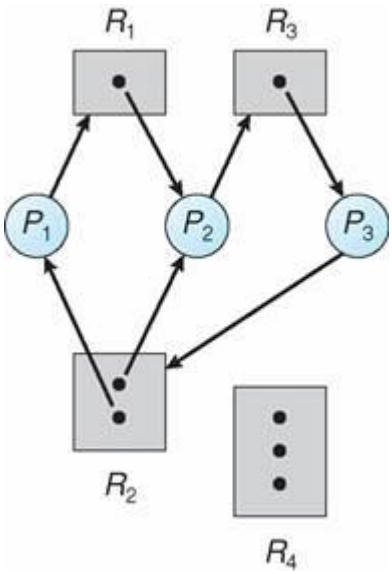
Un deadlock può sopravvenire se le seguenti quattro condizioni si verificano contemporaneamente:

1. **Esclusione mutua**: un solo processo alla volta può usare la risorsa, chi ne fa richiesta attende che questa si liberi;
2. **Mantieni e aspetta**: un processo deve possedere già almeno una risorsa ed essere in attesa di acquisizione di un'altra mantenuta attualmente da altri processi;
3. **No preemption**: le risorse possono essere rilasciate solo dal processo che le mantiene;
4. **Attesa circolare**: esiste un gruppo di processi in attesa $\{P_0, \ldots, P_n\}$ tale che P_0 attende P_1, P_1 attende P_2 e così via.

GRAFO DI ALLOCAZIONE DELLE RISORSE

G = {V, E} è un grafo orientato in cui l'insieme dei vertici V è partizionato in processi attivi (P) e tipi di risorse del sistema (R) e in cui gli archi E hanno i seguenti significati:

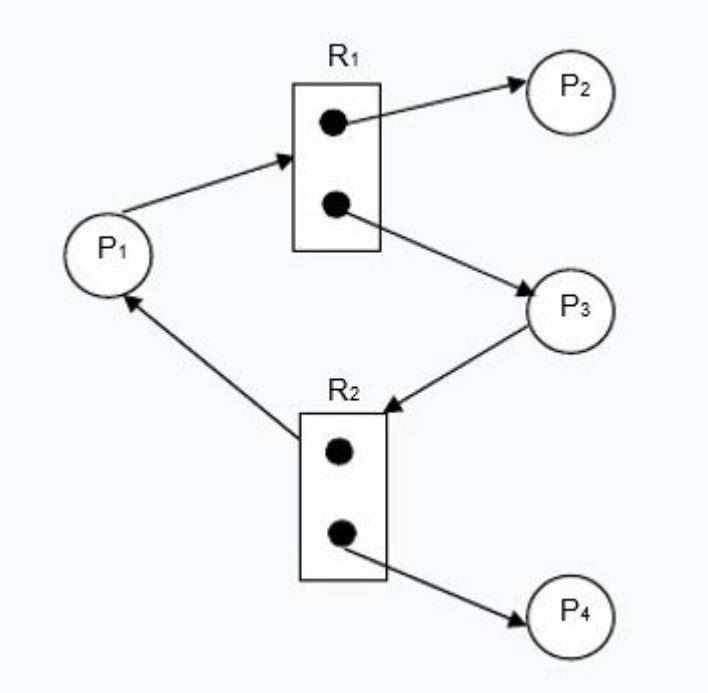
- $P_i \rightarrow R_j$ indica che il processo P_i sta richiedendo la risorsa R_j;
- $R_j \rightarrow P_i$ indica che il la risorsa R_j è assegnata al processo P_i.



I processi P_1, P_2 e P_3 sono in deadlock. P_2 sta aspettando la risorsa R_3 che è mantenuta dal processo P_3. Il processo P_3 sta aspettando che o P_1 o P_2 rilasci la risorsa R_2. In più, P_1 aspetta che P_2 rilasci la risorsa R_1.

Se un grafo non contiene nessun ciclo, allora nessun processo è in deadlock. Se un grafo contiene uno o più cicli, allora potrebbe verificarsi un deadlock.

Se ogni tipo di risorsa ha esattamente una istanza, allora un ciclo implica un deadlock.



Qui abbiamo un ciclo ma non abbiamo deadlock.

METODI PER GESTIRE I DEADLOCK

Possiamo risolvere il problema in tre modi:

- usare un protocollo che li prevenga o li eviti, assicurandosi che non si raggiunga mai lo stato di deadlock;
- possiamo rilevare uno stato di deadlock e recuperare la situazione;
- possiamo ignorare il problema e far finta che i deadlock non esistano.

La terza è la soluzione più usata perché si lascia a chi sviluppa le applicazioni il compito di gestire la non disponibilità delle risorse.

Evitare i deadlock corrisponde a necessitare di informazioni aggiuntive riguardo alle risorse richieste da un processo, in base alle quali per ogni richiesta si decide se un processo deve aspettare o meno una risorsa.

Tutto ciò porta overhead capace di abbattere le prestazioni di un sistema, quindi non viene implementato.