

## Processi – parte II

---

## Processi – parte II

---

- Scheduling dei processi
- Operazioni sui processi
- Thread

## Assegnazione della CPU

---

- Scheduler: parte del S.O. che decide a quale dei processi pronti assegnare la CPU.
- I processi possono essere in memoria principale o in memoria secondaria.
- Algoritmo di scheduling: realizza un particolare criterio di scelta.

## Criteri di scelta

---

- Utilizzo della CPU
- Produttività (throughput).
- Tempo di risposta (sistemi interattivi).
- Tempo di turnaround (sistemi batch).
- Non privilegio (fairness).
- ...

## Classificazione dei processi

- I processi possono essere classificati in:
  - *I/O-bound* – passano più tempo effettuando I/O che calcoli, richiedono molti servizi corti da parte della CPU.
  - *CPU-bound* - passano più tempo effettuando calcoli che I/O, richiedono pochi servizi molto lunghi da parte della CPU.

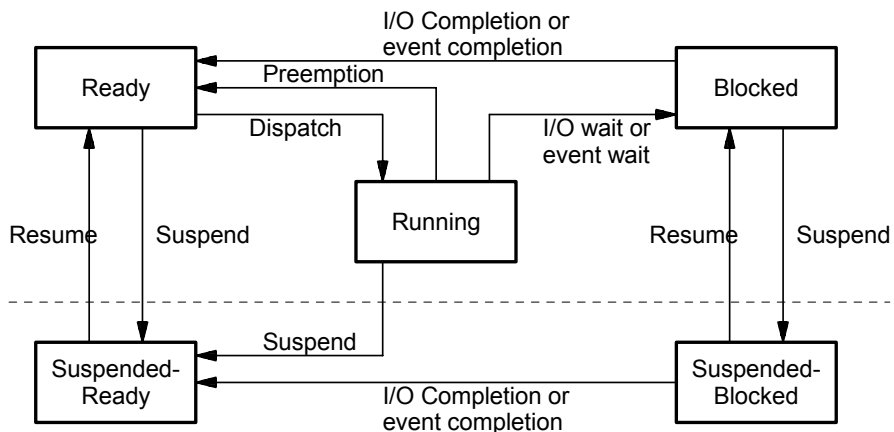
## Proprietà degli algoritmi di scheduling

- Proprietà degli algoritmi di scheduling:
  - preemptive scheduling: un processo in esecuzione perde il controllo della CPU anche se logicamente può proseguire.
  - non preemptive scheduling: un processo continua fino a rilascio spontaneo della CPU.

## Livelli di scheduling

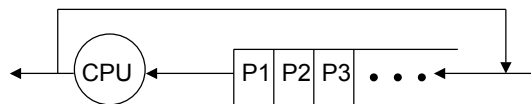
- Long term scheduling: determina quali programmi devono essere caricati dalla memoria di massa alla memoria principale.
- Medium term scheduling: rimuove temporaneamente dalla memoria principale un processo in esecuzione.
- Short term scheduling: seleziona tra i processi pronti in memoria principale quello a cui assegnare la CPU.
  - Dispatcher: realizza le operazioni necessarie per assegnare la CPU ad un processo scelto dal short term scheduler.

## Diagramma degli stati



## Algoritmi di short term scheduling

- Round Robin: La coda dei processi è gestita FIFO, ma ad ogni processo è assegnata la CPU per un quanto di tempo  $Q_t$  prefissato.
- $Q_t$  deve essere sufficientemente grande rispetto al tempo di cambio del contesto.



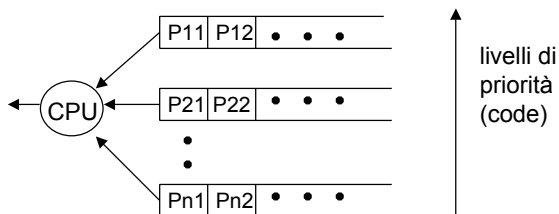
## Algoritmi di short term scheduling

- Priorità statica: fissata alla creazione dei processi in base alle loro caratteristiche:
  - processi foreground (sistemi interattivi) ---> alta priorità.
  - processi background (batch) ---> bassa priorità.
- Priorità dinamica: può essere modificata durante l'esecuzione dei processi.
- Starvation: ad un processo a bassa priorità non viene mai assegnata la CPU

## Algoritmi di short term scheduling

- Unix:
  - Più livelli di priorità
  - Round Robin con quanti di tempo di 100msec.
  - Aggiornamento dinamico della priorità: nuova  $p. = \text{base} + \text{uso CPU}$  base normalmente =0, assume un valore  $> 0$  tramite la system call **nice**, uso CPU contatore incrementato di uno ad ogni colpo di clock e diviso per 2 per ridurre la penalizzazione.

## Algoritmi di short term scheduling



## Algoritmi di short term scheduling

- First Come First Served:
  - CPU assegnata ai processi in ordine di arrivo temporale nel sistema.
  - Il processo mantiene la CPU sino a rilascio spontaneo.
  - Tempi medi di attesa alti (dipendono dall'ordine di arrivo).
  - Utilizzazione nei sistemi batch.

## FCFS: esempio

- Processi P1, P2, e P3 con tempi di esecuzione di 30, 55 e 5 unità di tempo rispettivamente e ordine di arrivo P1, P2, P3.
- Tempo medio di turnaround:  
 $(30+85+90)/3=68.33$

## Algoritmi di short term scheduling

- Shortest Job First (S.J.F.)
  - Tra tutti i processi in coda si esegue prima quello con durata minore.
  - occorre conoscere le caratteristiche dei processi.
  - fornisce soluzione ottima (tempo di attesa medio minimo).

## SJF: esempio

- P1 con tempo a, P2 tempo b, P3 tempo c e P4 tempo d:
  - $t. \text{ medio} = (4a+3b+2c+d)/4$
  - a incide più degli altri e quindi deve essere il minimo

## Unix: creazione e terminazione dei processi

---

- **fork**: crea un nuovo processo duplicando il processo chiamante
- **exec**: famiglia di funzioni che attiva l'esecuzione di un programma memorizzato in un file eseguibile.
- **wait**: sospende l'esecuzione di un processo in attesa del completamento di un altro processo ad esso legato.
- **exit** termina un processo.

## System call fork

---

- A seguito di una fork viene generato un processo figlio con le seguenti caratteristiche:
  - stesso codice del padre (codice condiviso).
  - area dati e stack distinti ma duplicati
  - stesso ambiente di esecuzione: file aperti, direttorio corrente, etc.
  - stesso contesto hardware di esecuzione (PC, registri, etc.)
- **int pid;**  
**pid = fork();** { pid -> process identifier}

## System call fork

- I processi , padre e figlio, operano in modo concorrente, a partire dall'istruzione di fork
- Padre e figlio possono eseguire istruzioni diverse
- Significato del pid:
  - pid = 0 nel figlio
  - pid > 0 nel padre
  - pid = -1 condizione di errore

## System call fork: esempio

```
main()
{
    int pid;
    printf("Un solo processo finora\n");
    pid = fork();
    if (pid==0)
        printf("Sono il figlio\n");
    else if (pid > 0)
        printf("Sono il padre, pid figlio=%d",pid);
    else
        printf("Situazione di errore \n");
}
```

## System call wait

- Usata dal processo padre per attendere la terminazione del figlio.
- Wait è sospensiva se ci sono processi figli in esecuzione, non sospensiva se tutti i processi figli sono terminati.

int pid, status

```
pid = wait(&status);
```

- pid: identificatore del processo figlio terminato, status: informazioni sulla terminazione del figlio

## System call wait

- Attesa per la terminazione di un particolare processo figlio:

```
while((ris=wait(&status))!=pid);
```

- Attesa per la terminazione di tutti i figli:

```
for(i=0;i<Nfigli;i++)
```

```
    wait(&status)
```

## System call wait: esempio

```
main()
{
    int pid, status;
    if((pid=fork())==0)
    {
        /* codice figlio*/
    }
    else
    {
        /*codice eseguito dal padre*/
        pid =wait(&status);
        etc.
    }
}
```

## Condizioni di terminazione

- Modo involontario:
  - azioni non consentite (indirizzi scorretti, istruzioni illegali, etc.)
  - interruzione da tastiera da parte dell'utente
  - segnali da altri processi tramite la system call **KILL**
- Modo volontario:
  - alla conclusione del codice
  - tramite la system call **EXIT**

## System call exit

- Usata da un processo per terminare in modo volontario.
- Exit chiude tutti i file aperti.  
int status;  
void exit(status);
- Status è restituito al processo padre, se in attesa. Il valore 0 costituisce una terminazione normale.

## System call exit

- Se il figlio esegue la exit prima che il padre esegua la wait allora passa nello stato ZOMBIE e viene rimosso dal sistema quando il padre esegue la wait.
- Se il padre termina prima dei figli, questi vengono adottati dal processo INIT (inclusi quelli ZOMBIE).

## System call exec

- La fork consente ai processi padre e figlio di utilizzare solo lo stesso codice.
- Con le funzioni della famiglia exec è possibile per un processo attivare l'esecuzione di un programma memorizzato in un file eseguibile.
- Le funzioni exec rilasciano l'area di codice e di dati occupata, sostituendola con quella del nuovo programma.
- Le exec non creano un nuovo processo concorrente al chiamante, ma lo sostituiscono con uno nuovo (mantiene lo stesso pid).

## System call exec

- Esistono diverse versioni di procedure che chiamano la exec e che differiscono solo per il formato degli argomenti:  
`execl(name, arg0, arg1, arg2, ..., (char*)0);`
- name= path del file da eseguire, arg0 = nome del file eseguibile.
- Non si ha ritorno al codice che ha invocato la exec, tranne in caso di errore.
- Si ereditano: direttorio corrente, file aperti, maschera dei segnali, etc.

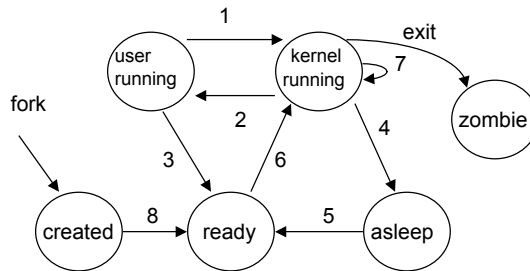
## System call exec: esempio

```
main()
{
    printf("esecuzione di ls \n");
    execl("/bin/ls","ls","-l",(char*)0);
    /*se si ritorna al processo chiamante, la
    chiamata è fallita*/
    exit(1);
}
```

## Exec e fork

```
main()
{
    int pid;
    pid=fork();
    /*il padre attende il completamento del figlio*/
    if (pid>0)
    {
        wait(&status);
    }
    if(pid==0)
    {
        execl("/bin/ls","ls","-l",(char*)0);
        exit(1);
    }
}
```

## Unix: grafo di stato dei processi



## Unix: grafo di stato dei processi

- 1 (user running -> kernel): system call o inter.
- 2 (kernel -> user): return (iret)
- 3 (running -> ready): revoca per tempo
- 4 (kernel running -> asleep): sleep
- 5 (asleep -> ready): wakeup
- 6 (ready -> kernel running): schedule
- 7 (kernel running -> kernel running): int.
- 8 (created -> ready): allocazione in memoria.

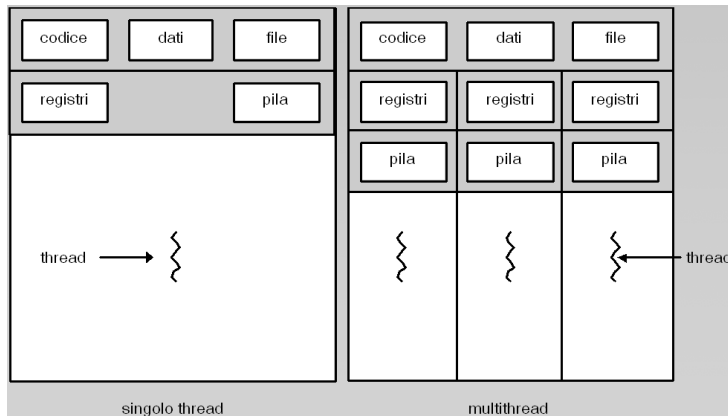
## System call getpid e getppid

- Getpid: ritorna il process ID del processo corrente
- Getppid(): ritorna il process ID del processo padre

## Thread

- Un *thread* (o *lightweight process*) è l'unità elementare di utilizzo della CPU e consiste di:
  - program counter
  - insieme di registri
  - stack
- Un thread condivide con gli altri thread concorrenti:
  - sezione di codice
  - sezione di dati
  - risorse del SOcioè il dominio del *task*.
- Un processo o processo *heavyweight* corrisponde ad un task con un solo thread.

# Thread



# Thread

- In un task composto da thread multipli, mentre un thread è in attesa, può essere eseguito un altro thread dello stesso task.
- La cooperazione di thread multipli nello stesso task consente di ottenere i seguenti vantaggi:
  - Si possono eseguire in concorrenza parti di una stessa applicazione.
  - Si possono usare più copie di una sezione di codice per servire più clienti.
  - Chiamare funzioni bloccanti e lasciare all'applicazione di far altro nel frattempo.
  - Sfruttare le schede multiprocessore.

## Thread

---

- Applicazioni che richiedono l'uso di un buffer comune (per es. produttore-consumatore) traggono beneficio da un'implementazione mediante thread.
- Esistono due tipi di thread:
  - Kernel thread (derivati da Mach, OS/2).
  - User-level thread; supportati mediante un insieme di chiamate, a livello utente, a librerie di sistema.

## User thread

---

- Vantaggi
  - Context switch veloce tra thread di uno stesso task.
  - Si possono implementare sopra un kernel qualsiasi.
- Svantaggi
  - Se un thread effettua una system call bloccante, sono bloccati tutti i thread dello stesso task.
  - Solo un thread running per task anche in un sistema multiprocessore



# Kernel thread

---

- Vantaggi

- I thread ready possono essere schedulati anche se appartengono allo stesso task di un thread che ha chiamato una system call bloccante.
- In un sistema multiprocessore si possono eseguire thread multipli per task.

- Svantaggi

- Context switch più costoso perché richiede il passaggio al modo kernel.
- Limitazione nel numero massimo di thread per task