

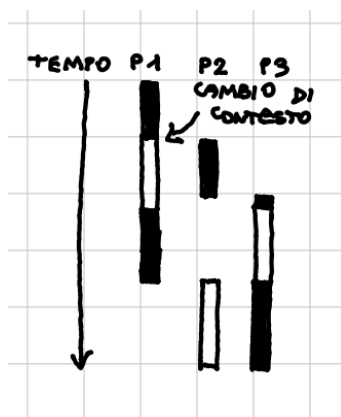
## Esame 19/02/2020

### Domanda A:

Spiegare che cosa si intende per multiprogrammazione, perché la si è introdotta e che relazione c'è tra grado di multiprogrammazione ed utilizzazione delle CPU.

### Risposta:

La multiprogrammazione è una tecnica nella quale più job vengono caricati in RAM e, quando il processo inizia un I/O Burst (quindi la CPU in questo periodo non verrebbe utilizzata) il Sistema Operativo effettua un context switch ad un altro processo, in modo tale che la CPU venga sfruttata al massimo. Quando il I/O Burst termina il sistema operativo torna ad eseguire il processo iniziale.



Si è introdotta per riuscire ad avere un pseudo parallelismo tra processi. Il livello di multiprogrammazione (ovvero il numero di processi caricati simultaneamente in RAM) è correlato alla utilizzazione della CPU.

### Domanda B:

Spiegare che cosa si intende per trap e quale relazione ha con il sistema operativo. Fare un esempio di trap relativo alla gestione della memoria.

### Risposta:

La trap deve essere vista come un interrupt software. Le trap sono un meccanismo di segnalazione con cui il sistema operativo entra in azione eseguendo il codice dell'handler di quella specifica trap. Il codice dell'handler si trova nel kernel del sistema operativo e non può essere modificato, nemmeno dal kernel stesso.

La trap può avvenire per svariati motivi, tra cui:

- un programma tenta di eseguire una divisione per zero
- OPCode non valido
- System calls
- Page Fault

Le trap posso essere gestite in modo diverso, per esempio il processo viene terminato. Oppure si gestisce la trap e si torna all'esecuzione del proceso (System Calls, o Page Fault).

In particolare il Page Fault accade quando la pagina del processo non è presente in RAM, il Page Fault Handler deve gestire questa cosa (prendendo dal disco e caricando su RAM la pagina corretta). Dopo che il Page Fault Handler ha caricato in RAM la pagina del processo viene ri eseguita l'istuzione che ha causato il page fault.

## Domanda C:

Descrivere le operazioni sui semafori ed uno pseudo-codice per la loro realizzazione.

## Risposta:

I semafori sono un meccanismo di sincronizzazione fra processi (o threads), in particolare è un meccanismo utilizzato per gestire le corse critiche, ovvero che due processi, nello stesso momento, vedono lo stesso valore in una variabile condivisa.

Godono di queste operazioni:

- `init/create` : creazione del semaforo
- `destroy` : cancellazione del semaforo
- `up` : incremento il valore del semaforo
- `down` : decremento il valore del semaforo

In particolare quando un processo effettua una `up` :

- se il valore del semaforo è  $> 0$  incremento il valore
- se il valore del semaforo è uguale a 0 non incremento il valore del semaforo ma risveglio il processo che era in attesa di entrare in corsa critica.

Invece per la `down` abbiamo che:

- se il valore del semaforo è  $> 0$  decremento il valore

- se il valore del semaforo è uguale a 0 lo stato del processo viene impostato a bloccato, e resterà tale fino a quando non verrà risvegliato da una up su quel semaforo.

Per implementare i semafori abbiamo bisogno di un'istruzione ISA che sia atomica (ovvero indivisibile), dato che sia la `up` che la `down` sono corse critiche. Questa istruzione prende il nome di `TestAndSet(&lock)`.

Infatti la realizzazione della up e della down è la seguente:

`up :`

```
while(TestAndSet(&s.lock));
if(s.queue != EMPTY){
    t = dequeue(s.queue); //Estraggo un thread dalla coda
    t.status = READY; //Imposto il thread appena estratto come pronto
}
else{
    s.val++;
}
s.lock = 0;
```

`down :`

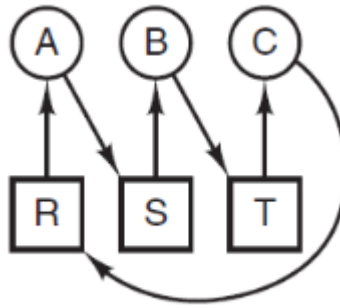
```
while(TestAndSet(&s.lock));
if(s.val == 0){
    enqueue(s.queue, t); //T è il thread corrente
    s.lock = 0;
    t.status = BLOCKED; //Imposto lo stato del thread corrente a bloccato
    scheduler(); //Lo scheduler sceglierà un nuovo thread da eseguire
    dispatcher();
}
else{
    s.val--;
    s.lock = 0;
}
```

## Domanda D:

Spiegare che cosa si intende per deadlock e quale o quali, tra le condizioni necessarie affinché si possa realizzare, sono rese false nella soluzione del problema dei filosofi vista nel corso.

## Risposta:

Il deadlock (o stallo) è una condizione in cui un processo rimane in attesa che venga sbloccato da un altro processo che a sua volta è bloccato, come nell'esempio qui sotto:



Le condizioni per la quale un processo può essere vittima di deadlock sono le seguenti (tutte e quattro devono essere vere):

1. le risorse sono allocate in mutua esclusione (e se no, perché dovrebbe attendere)
2. le risorse non sono preemptive: non ha senso portarle via al processo/thread che le sta usando (es. stampante; mentre per la CPU il costo è accettabile)
3. un p/t a cui sono allocate risorse ne può richiedere altre (hold and wait, allocazione parziale)
4. si ha attesa circolare nel senso già indicato: ogni p/t attende una risorsa detenuta da un altro dell'insieme

Per il problema dei filosofi abbiamo il caso numero 3, per risolvere questo problema utilizziamo i semafori privati, in modo da riuscire a prendere entrambe le risorse contemporaneamente.

### Domanda D:

Spiegare che cosa si intende per CPU burst. Indicare se nello scheduling a breve termine è opportuno favorire la scelta di processi/threads con CPU burst brevi oppure processi/threads con CPU burst lunghi, e perché.

### Risposta:

Con CPU burst si intende un periodo del processo nella quale viene utilizzato in modo continuativo la CPU. Nello scheduling a breve termine è opportuno dare priorità ai processi con CPU Burst brevi, perché il peso del CPU Burst in esecuzione viene *prolungato* nel tempo. Infatti sapendo la seguente formula (per 4 processi) abbiamo che:  $T_{medio} = (4T_1 + 3T_2 + 2T_3 + T_4)/4$ , dove T è la durata di un processo.

Esistono algoritmi appositi per risolvere questo problema: per esempio il Short Job First (SJF) ed il Short Remaning Tlme Next (SRTN).

## Domanda E:

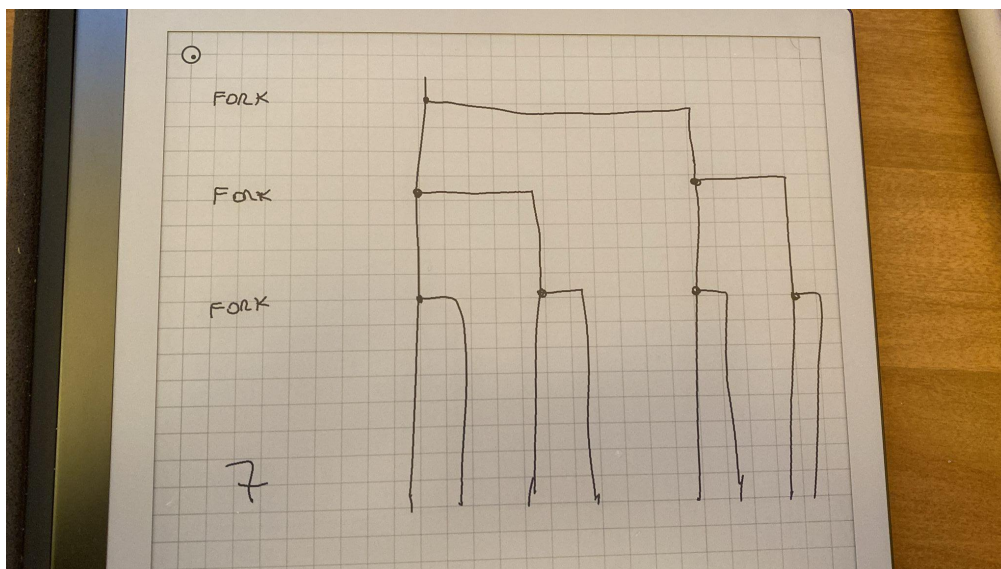
Quante chiamate di sistema vengono effettuate se si esegue il seguente programma? Motivare la risposta.

```
int main(){  
    fork();  
    fork();  
    fork();  
    printf("Duprè\n");  
}
```

Vengono eseguite 15 chiamate di sistema, per fare il calcolo conviene usare questa tecnica:

```
fork(); //1  
fork(); //2  
fork(); //4  
printf("Duprè\n"); //8  
//8 + 4 + 2 + 1 = 15
```

Volendo puoi fare così:



I pallini sono le fork , le linee finali sono le printf, sommi il tutto ed ottieni lo stesso risultato.

## Domanda G:

Illustrare l'effetto della chiamata `exec1p("prog", "prog", NULL)`.

Risposta:

Questa system call cambia il codice del processo in quello del programma che vogliamo eseguire.

### Domanda H:

Spiegare perché tipicamente è opportuno che una chiamata a `pthread_cond_wait` (della libreria dei Pthread) sia inserita all'interno di un ciclo while.

### Risposta:

É opportuno che `pthread_cond_wait` venga eseguita in un while in modo tale che siano effettivamente vera la condizione del test, perché nonostante abbia l'accesso in mutua esclusione non è detto che la condizione di test sia vera.

---

## Esame 27/06/2023

---

### Domanda A:

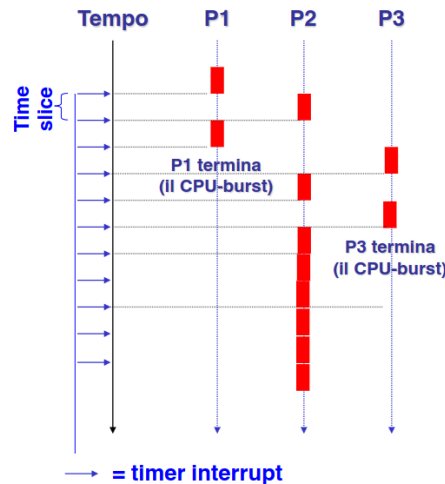
Che cosa si intende per "multiprogrammazione" e per "timesharing"? Per quale motivo sono stati introdotti?

### Risposta:

La multiprogrammazione è una tecnica nella quale più job diversi vengono caricati in memoria e, durante un I/O Burst (quindi un tempo morto per la CPU), si decide di eseguire un altro processo (effettuando un context switch), per poi (quando l'I/O Burst è terminato) riprendere il processo originale.



Il timesharing invece è una tecnica nella quale un processo può essere eseguito per un timeslice (definito a priori ed uguale per tutti, inviato tramite un interrupt da un dispositivo hardware). Se il processo riesce a compiere la sua operazione in quel quanto di tempo bene, si passa al prossimo processi. Se invece il quanto di tempo non è sufficiente viene "parcheggiato", ne viene eseguito un altro (sempre per un quanto di tempo), e poi si ritorna al processo precedente.



La tecnica della multiprogrammazione viene utilizzata per ottimizzare al massimo l'utilizzo della CPU, infatti più il livello di multiprogrammazione (processi caricati in ram) è alto più la CPU è ben utilizzata.

Invece la tecnica del timesharing è utilizzata per evitare che i processi possano monopolizzare l'utilizzo della CPU.

## Domanda B:

Illustrare in che cosa consiste la creazione di un nuovo thread all'interno di un processo, facendo anche riferimento alla chiamata di libreria descritta ed utilizzata nel corso per la creazione di un nuovo thread. Descrivere poi quali vantaggi si hanno nel realizzare un' applicazione con thread multipli, rispetto ad una con un solo processo e un solo thread, e ad una con processi multipli.

Unix

## Risposta:

La libreria vista nel corso si chiama POSIX Threads, e la creazione di un nuovo thread all'interno di un processo si effettua utilizzando la funzione `pthread_create()`.

Nella funzione `pthread_create()` oltre a passare il thread che stiamo andando a creare dobbiamo anche passare il cosiddetto "thread body", ovvero le operazioni che il thread andrà ad effettuare.

I vantaggi di realizzare un'applicazione che thread multipli invece che una con un solo processo ed un solo thread è che nel primo caso riusciamo ad introdurre il parallelismo all'interno del processo, questo significa che più thread saranno eseguiti in pseudo parallelo (o parallelo se l'hardware lo permette) incrementando la velocità di esecuzione dell'applicazione.

Invece la differenza con processi multipli (cooperanti) è che il context switch che avviene tra processi è più lento confronto al context switch presente tra threads, stiamo parlando di lievi differenze, ma che sul lungo periodo potrebbero diventare significative. In più i threads condividono tra di loro i dati, cosa che non avviene per i processi, infatti in quest'ultimo caso per far comunicare due processi tra di loro dobbiamo per forza utilizzare tecniche apposite (come per esempio le pipe).

## Domanda C:

Illustrare l'effetto ed il tipico uso della chiamata di sistema `fork()` anche in combinazione con altre chiamate di sistema.

Unix

## Risposta:

La system call `fork()` crea un nuovo processo con lo stesso codice del processo che ha chiamato la `fork()`. Un utilizzo tipico è quello in combinazione con le system call della famiglia `exec`, questo è molto utile per fare in modo che il processo figlio appena creato esegua un programma. Dopo una `exec` il processo trasformerà la sua immagine da quella di partenza a quella del programma che vogliamo eseguire, infatti in questo modo perdiamo completamente la relazione con il processo padre. L'unica cosa che resta invariata è il PID.

Esempio tipico dell'utilizzo di `fork()`:

```
if(fork() == 0){
    //Codice processo figlio
    execl("ls", "ls", NULL);
}else{
    //Codice processo padre
    int x = 2;
    ++x;
}
```

## Domanda D:

Descrivere le operazioni sui semafori ed uno pseudo-codice per la loro realizzazione.



## Risposta:

Le operazioni sui semafori sono:

- `create / init` : creazione di un semaforo
- `destroy` : eliminazione di un semaforo
- `up` : incrementa il valore del semaforo. Quando un processo effettua una `up` ad un semaforo significa che è appena uscito da una corsa critica, e se c'era qualche altro processo in attesa ora è libero di passare.
- `down` : decrementa il valore del semaforo. Quando un processo effettua una `down` su un semaforo significa che sta per entrare in una corsa critica. Per prima cosa controlla se il valore del semaforo è  $> 0$ 
  - se sì entra nella zona critica. (Questo avviene dopo aver decrementato il valore del semaforo).
  - se non lo è (quindi  $= 0$ , perchè i semafori non possono assumere valori negativi) deve aspettare finché un'altro processo esca da quella determinata zona critica. Nel frattempo il processo resta in attesa.

Di fatto sia la `up` che la `down` sono sezioni critiche, per risolvere questo problema dobbiamo fare in modo che queste due operazioni siano atomiche (ovvero indivisibili), per farlo utilizziamo il `TestAndSet(&lock)`, ovvero un'istruzione a livello ISA per garantire la mutua esclusione.

L'implementazione della `up` è la seguente:

```
while(TestAndSet(&s.lock));
if(s.queue != EMPTY){
    t = dequeue(s.queue); //Estraggo un thread dalla coda
    t.status = READY; //Imposto il thread appena estratto come pronto
}
else{
    s.val++;
}
s.lock = 0;
```

L'implementazione della `down` è la seguente:

```

while(TestAndSet(&s.lock));
if(s.val == 0){
    enqueue(s.queue, t); //T è il thread corrente
    s.lock = 0;
    t.status = BLOCKED; //Imposto lo stato del thread corrente a bloccato
    choose_new_thread(); //Lo scheduler sceglierà un nuovo thread da eseguire
}
else{
    s.val--;
    s.lock = 0;
}

```

## Domanda E:

Illustrare l'effetto della chiamata `pthread_cond_wait(&cond, &m)`, illustrando anche cosa deve accadere affinché il thread esca dalla chiamata.

## Risposta:

La chiamata `pthread_cond_wait(&cond, &m)` ha l'effetto di sospendere il thread su una variabile condizione `cond`. Questo avviene quando un thread è riuscito ad ottenere l'accesso in mutua esclusione. Se la condizione non è valida il thread va in stato wait perdendo l'accesso in mutua esclusione, attende che si verifichi la condizione, e quando si verifica la condizione cerca di riprendersi l'accesso in mutua esclusione su `m`, una volta fatto esce dalla chiamata wait.

Dopo aver eseguito la chiamata e le sue operazioni il thread deve:

1. inviare un segnale per risvegliare i thread sospesi con
  - `pthread_cond_signal(&cond)` se bisogna risvegliare un singolo thread
  - `pthread_cond_broadcast(&cond)` se bisogna risvegliare più threads
2. rilasciare la mutua esclusione con una chiamata a `pthread_mutex_unlock(&m)`.

L'ordine di queste due operazioni è fondamentale.

In più è opportuno mettere la chiamata `pthread_cond_wait(&cond, &m)` all'interno di un ciclo while perché chi segnala `cond` non necessariamente conosce quale test il thread T1 attende che diventi false, e se anche lo sapesse non è detto che T1 acquisti subito la mutua esclusione dopo la segnalazione della condizione.

## Domanda F:

Spiegare che cosa si intende per CPU Burst. Indicare se nello scheduling a breve termine è opportuno favorire la scelta di processi/threads con CPU burst brevi oppure processi/threads con CPU burst lunghi, e perché.

### Risposta:

Con CPU burst si intende un periodo del processo nella quale viene utilizzato in modo continuativo la CPU. Nello scheduling a breve termine è opportuno dare priorità ai processi con CPU Burst brevi, perché il peso del CPU Burst in esecuzione viene *prolungato* nel tempo. Infatti sapendo la seguente formula (per 4 processi) abbiamo che:  $T_{medio} = (4T_1 + 3T_2 + 2T_3 + T_4)/4$ , dove  $T$  è la durata di un processo.

Esistono algoritmi appositi per risolvere questo problema: per esempio il Short Job First (SJF) ed il Short Remaning Tlme Next (SRTN).

### Domanda G:

Descrivere che cosa si intende per località nel contesto della gestione della memoria e come si cerca di sfruttare tale fenomeno nell'algoritmo di rimpiazzamento delle pagine detto "dell'orologio" nella versione base.

### Risposta:

Per fenomeno della località si intende che per periodi significativamente lunghi un processo accede ad indirizzi in un sottoinsieme delle sue pagine, in particolare accede ripetutamente agli stessi indirizzi (località nel tempo) o accede ad indirizzi molto vicini a quelli usati poco prima (località nello spazio).

Mell'algoritmo dell'orologio si una lista circolare con all'interno tutte le pagine presenti in quel momento in RAM. Con un puntatore si scorre ogni nodo dell'orologio (come se fosse una lancetta) fino a quando non troviamo una pagina con il bit R a zero, a quel punto sfrttiamo la pagina e possiamo rimpiazzare la nostra pagina con quella puntata dalla lancetta. Se non c'è nemmeno una pagina con il bit a zero si prende come vittima la prima pagina alla quale la lancetta puntava (in pratica si comporta come l'algoritmo FIFO).

L'algoritmo sfrutta il pricipio della località temporale dato che accede ripetutamente agli stessi indirizzi per controllare il bit R, in più usa anche il principio della località spaziale dal momento che spostando la lancetta accede a indirizzi vicini a quelli usati poco prima.

**Domanda X:**

**Risposta:**

---

## Risposte del vecchio File Word

---

### Esempio prova d'esame

---

**Domanda A:**

Spiegare che cosa si intende per processo, perché è stata introdotta tale nozione nei sistemi operativi. Spiegare che cos'è una chiamata di sistema, e perché la creazione di un processo avviene mediante una chiamata di sistema. Illustrare la chiamata di sistema fornita per tale scopo nei sistemi Unix (POSIX).

Unix

**Risposta:**

Un processo è una attività di elaborazione guidata da un programma; la sua velocità di esecuzione dipende da quanti processi condividono la stessa CPU e Memoria.

Una visione più astratta: un processo è una attività di elaborazione su una CPU virtuale e una memoria virtuale grande abbastanza per contenere i dati e il codice del programma associato al processo. In questa interpretazione la CPU virtuale funziona in modo "intermittente", dato che è in grado di portare davvero avanti la computazione solo quando ad essa corrispondono CPU e memoria vere. In un dato momento ci possono essere più processi che eseguono lo stesso programma. Inoltre, il programma di un processo può anche cambiare durante la durata della sua vita. Il sistema operativo deve mantenere lo stato di elaborazione per ogni processo. Nel caso di più processi che eseguono lo stesso programma questo può essere condiviso, ma non i dati.

L'introduzione dell'astrazione dei processi è stata inizialmente motivata dalla necessità di utilizzare al meglio le risorse disponibili in particolare la CPU.

Una chiamata di sistema è una funzione che permette di agire sulle entità astratte del sistema operativo. In un certo senso può essere un tipo speciale di chiamata di procedura solo che la chiamata di sistema entra in modalità kernel. Per questi motivi è uno degli eventi che può causare la creazione di un processo.

In Unix per creare un processo viene utilizzata la `fork()` che crea un clone esatto del processo che esegue la chiamata. Dopo la `fork` i due processi, genitore e figlio, hanno la stessa immagine di memoria, le stesse stringhe di ambiente e i medesimi file. Inoltre, dopo la creazione di un processo, genitore e figlio, hanno i loro spazi degli indirizzi personali. Se uno dei due modifica qualcosa nel suo spazio degli indirizzi l'altro non lo vede. Il processo figlio poi esegue una `execve` per cambiare la propria immagine di memoria. Lo spazio degli indirizzi del figlio è una copia di quello del padre. Il figlio può condividere la memoria del genitore se la memoria condivisa è in modalità `copy-on-write` dove se uno dei due modifica una parte, questa parte viene copiata per garantire che la modifica sia apportata in un'area di memoria privata.

## Domanda B:

In quale stato si trova, o in quali stati si può trovare, un processo o thread in attesa attiva (`busy waiting`)?

## Risposta:

L'attesa attiva, o `busy waiting`, è l'azione di testare continuamente una variabile finché non viene valorizzata cioè è l'azione che compie un processo mentre attende di entrare in una regione critica (parte del programma in cui si accede alla memoria condivisa). Il `busy waiting` andrebbe generalmente evitato poiché consuma tempo di CPU ma può essere utilizzato se la probabilità di attendere è bassa e la condizione che ha causato l'attesa dura poco. Un processo/thread in `busy waiting`, solitamente, è in stato `ready` o `running` cioè pronto per usare la CPU o sta già utilizzando la CPU.

## Domanda C:

Quale output si può ottenere con il seguente programma? Motivare la risposta.

Unix

```
main(){
    printf("Hello world 1\n");
    execlp("prog","prog",NULL);
    printf("Hello world 2\n");
}
```

## Risposta:

L'output che viene stampato è "Hello world 1" poi se esiste il programma `prog` viene eseguito l'eseguibile `prog`, se non esiste si esce dalla `exec` con un errore e si stampa "Hello world 2".

## Domanda D:

Considerare la seguente ipotesi di soluzione per l'accesso a sezioni critiche, facente uso di una variabile lock inizializzata a 0:

Ingresso nella sezione critica:

```
while (lock==1);  
lock=1;
```

Uscita dalla sezione critica:

```
lock=0;
```

Indicare se è una soluzione corretta, perché, e in caso positivo quali vantaggi o svantaggi comporta rispetto alla soluzione che utilizza i semafori.

## Risposta:

Questa soluzione non è corretta poiché la procedura per entrare nella sezione critica è essa stessa una regione critica, dunque, può accadere che: due processi valutano in pseudo parallelo `lock == 1` ed entrambi la trovano falsa ed eseguono `lock = 1` entrando così tutte e due nella regione critica.

## Domanda E:

Spiegare la differenza tra semafori binari e semafori contatore. Quali di essi sono forniti nei sistemi Unix (POSIX)?

Unix

## Risposta:

I semafori binari possono assumere solo i valori 0 e 1: oppure si tratta di un semaforo generale usato solo in modo che assuma tali valori. Se si esegue una `up()` quando `s.val` è già 1 questa non ha effetto cioè l'effetto è indefinito. Possono essere anche utilizzati per garantire la mutua esclusione inizializzandoli a 1 ed in questo caso vengono chiamati mutex.

I semafori contatore possono assumere un qualsiasi valore maggiore o uguale a 0. Possono essere usati per il problema della sincronizzazione con un numero N di risorse da assegnare dove: il semaforo si inizializza a N, per prelevare si utilizza una `up()` e per rilasciare una `down()`. In

generale in ogni momento s.val è maggiore o uguale a 0 e vale:  $N - \text{numero di down() completate} + \text{numero di up() completate}$  cioè risorse totali – risorse prelevate + risorse rilasciate.

In Unix sono presenti i mutex, come semafori binari, che possono assumere solo due stati: locked in cui pende la mutua esclusione e unlocked in cui la rilascia.

## Domanda F:

Spiegare che cosa si intende per processo CPU-bound e I/O bound, come può fare il sistema operativo per identificarli come tali, e come può tenerne conto nello scheduling della CPU a breve termine.

## Risposta:

Per un processo CPU-bound si intende un processo che spende la maggior parte del suo tempo in elaborazione, dunque, che ha burst di CPU lunghi e breve attesa di I/O.

Mentre, un processo I/O bound è un processo che spende la maggior parte del suo tempo in attesa dell'I/O, dunque, ha burst di CPU brevi e lunghe attese di I/O.

Per sfruttare in modo efficiente le risorse conviene avere un mix bilanciato di processi CPU bound e I/O bound.

Il sistema operativo per identificarli come tali deve fare una stima della durata del prossimo CPU burst che viene effettuata tramite una media esponenziale:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

dove:

- $t_n$  è la durata effettiva dell'n-esimo CPU burst;
- $\alpha$  è compreso tra 0 e 1 (con annessi valori con la virgola);
- $\tau_n$  è la stima dell'n+1-iesimo CPU burst;
- $\tau_0$  ha un valore predefinito

Nello scheduling a breve termine si scelgono solitamente i processi con CPU burst piccoli quindi processi di tipo I/O bound.

## Domanda G:

Descrivere l'algoritmo di rimpiazzamento delle pagine detto "dell'orologio". Spiegare se, quanto e perché le pagine scelte possono essere considerate buone scelte come pagine da rimpiazzare.

### Risposta:

L'algoritmo dell'orologio è una lieve semplificazione implementativa dell'algoritmo della seconda possibilità: si usa una lista circolare e anziché portare in fondo alla lista la pagina non rimpiazzata, si fa avanzare il puntatore (come se fosse la lancetta di un orologio). La lancetta indica la pagina più vecchia. Quando avviene un page fault la pagina indicata dalla lancetta viene controllata:

- se  $R=0$  allora viene sfrattata e viene inserita una nuova pagina al suo posto e la lancetta avanza;
- se  $R=1$  viene azzerato e la lancetta avanza. Questo processo è ripetuto finché non si trova una pagina con  $R=0$ .

Se tutte le pagine hanno  $R = 1$  allora viene applicato l'algoritmo FIFO (si fa tutto il giro in un solo page fault scegliendo la pagina da cui si era partiti): ma se questo capita spesso, vuol dire che sono tutte molto usate e quindi non c'è da sperare di riuscire a tenere in RAM tutte le pagine "che servono".

Le pagine scelte sono considerate una buona scelta come pagine da rimpiazzare perché grazie al controllo del bit  $R$  le pagine più utilizzate non vengono rimpiazzate.

---

## Esame 30/01/2020

---

### Domanda A:

Spiegare che cosa si intende per "nucleo" (kernel) del sistema operativo, in che cosa esso si distingue da altro software che troviamo a disposizione in un sistema di elaborazione e perché è opportuna questa distinzione.

### Risposta:

Per nucleo, o kernel, si intende il vero e proprio sistema operativo, che viene eseguito in modalità kernel per avere accesso a tutto l'hardware ed eseguire qualunque istruzione macchina. Si distingue dalla modalità utente che gestisce il codice dei programmi utente e ha a disposizione solo un sottoinsieme delle istruzioni macchina. Si passa alla modalità kernel e a eseguire il codice del sistema operativo quando:



- si effettua una chiamata di sistema che proprio per il cambiamento di modalità differisce da una funzione di libreria;
- un programma cerca di eseguire un'azione non permessa incorrendo in una trap;
- c'è un'interruzione hardware e si passa ad eseguire la routine di gestione delle interruzioni.

## Domanda B:

Quali vantaggi si hanno (e perché) nel realizzare un'applicazione come un singolo processo con thread multipli? Confrontare questa soluzione con quella che prevede processi multipli cooperanti.

## Risposta:

Realizzare un processo singolo con thread multipli è più vantaggioso (nelle applicazioni COOPERANTI) poiché i thread consentono molteplici esecuzioni che hanno luogo nello stesso ambiente del processo, dunque, condividono lo spazio degli indirizzi e altre risorse.

Questo metodo garantisce: parallelismo all'interno di una singola applicazione, sovrapposizione degli I/O e computazione per i thread di una singola applicazione come i processi multipli cooperanti. Ma con i thread multipli è più semplice la condivisione delle risorse fra attività cooperanti in una applicazione e sono meno costosi la creazione di un nuovo thread e la switch fra thread rispetto ai processi multipli.

## Domanda C:

Spiegare che cosa si intende per semafori privati ed illustrare il loro uso per l'assegnazione di risorse, evidenziando la differenza con una soluzione che non prevede tale uso.

## Risposta:

I semafori privati sono tali per come vengono usati: il meccanismo messo a disposizione dalle funzioni è lo stesso, senza alcun controllo su quale processo/thread usi i semafori e come. Un semaforo privato `s_priv_P` di un processo P è inizializzato a 0. Solo il processo P può eseguire una `down(&s_priv_P)` e la esegue quando deve attendere che diventi vera una condizione (booleana) di sincronizzazione. Invece, qualsiasi processo, incluso P, può eseguire una `up(&s_priv_P)` se serve per svegliare P o serve non farlo sospendere se fa una down, perché la condizione di sincronizzazione è vera.

I semafori privati vengono utilizzati per il problema dell'assegnazione delle risorse che consiste nel considerare un insieme di processi che devono acquisire ed utilizzare un certo numero di

risorse da un pool di  $k$  risorse equivalenti. Prima di poter utilizzare una risorsa il processo  $P_i$  deve acquisirla ( `down(&ris)` ) e dopo aver usato la risorsa la deve rilasciare ( `up(&ris)` ). Se due processi  $P_j$  e  $P_i$  sono in attesa di una risorsa (bloccati su una `down(&ris)`) e un terzo processo  $P_h$  esegue una `up(&ris)` la scelta di quale dei due processi viene risvegliato dipende dall'implementazione della `up`. Questo meccanismo può andare bene ma se si vuole applicare una politica diversa bisogna esprimerla esplicitamente e lo si può fare sfruttando i semafori privati: il processo che deve attendere una risorsa viene sospeso su un proprio semaforo privato `s_priv_i` e il processo sospeso viene ricordato in una variabile chiamata `sospeso_i`. Al rilascio di una risorsa, si sceglie quale processo risvegliare tra tutti quelli sospesi e lo si risveglia con una `up(&sem_priv_i)`.

### **Domanda D:**

Spiegare che cosa si intende per starvation nello scheduling della CPU. Dopo aver evidenziato i principi generali dello scheduling a code multiple, indicare che cosa viene fatto in relazione alla starvation in tale tipo di scheduling.

### **Risposta:**

Lo scheduling consiste nella scelta di quale processo servire tra i processi in coda e un tipico problema che si può verificare è la starvation: letteralmente "morte di fame" che avviene quando un processo rimane sempre in una coda d'attesa perché gli altri processi lo superano sempre.

Lo scheduling a code multiple è caratterizzato da classi di priorità. L'appartenenza alle classi può essere dinamica e in particolare dipende dal comportamento passato del processo. Per favorire processi con CPU burst corti senza fare una stima per classificarli bisogna far sì che quando usa tutto il quanto, (intervallo di tempo assegnato a un processo), il processo passa in una classe più bassa con quanto maggiore.

Per evitare la starvation dei processi che finiscono in classi inferiori si deve, dopo un po' di tempo in cui il processo è in una classe bassa o nella più bassa, riportarlo in alto.

### **Domanda E:**

Illustrare l'algoritmo di rimpiazzamento delle pagine dell'orologio nella versione base ed in quella modificata per tenere conto del concetto di working set.

### **Risposta:**

L'algoritmo dell'orologio è una lieve semplificazione implementativa dell'algoritmo della seconda possibilità: si usa una lista circolare e anziché portare in fondo alla lista la pagina non rimpiazzata, si fa avanzare il puntatore (come se fosse la lancetta di un orologio). La lancetta

indica la pagina più vecchia. Quando avviene un page fault la pagina indicata dalla lancetta viene controllata:

- se  $R=0$  allora viene sfrattata e viene inserita una nuova pagina al suo posto e la lancetta avanza;
- se  $R=1$  viene azzerato e la lancetta avanza. Questo processo è ripetuto finché non si trova una pagina con  $R=0$ .

Se tutte le pagine hanno  $R = 1$  allora viene applicato l'algoritmo FIFO (si fa tutto il giro in un solo page fault scegliendo la pagina da cui si era partiti): ma se questo capita spesso, vuol dire che sono tutte molto usate e quindi non c'è da sperare di riuscire a tenere in RAM tutte le pagine "che servono".

L'algoritmo dell'orologio modificato cerca di non rimpiazzare le pagine che appartengono al working set, se tutte le pagine sono nel working set, allora se ne sceglie una a caso, con preferenza per quelle che hanno il dirty bit a 0. Per definire quali pagine fanno parte del working set di un processo: si fissa un'ampiezza della finestra temporale, si mantiene traccia per ogni frame di un tempo di ultimo riferimento e l'hardware imposta il bit R ad ogni riferimento, ad ogni timer interrupt e i bit R vengono azzerati:

- se il bit R di un frame era 1, viene copiato nel suo campo "ultimo riferimento" il tempo di CPU complessivamente utilizzato (chiamato tempo virtuale) dal processo a cui appartiene.
- se il bit R era 0 la pagina non è stata referenziata durante l'ultimo ciclo di clock, perciò, può essere una candidata per l'eliminazione

Per capire se rimuovere o meno una pagina bisogna tener conto se è nel WS ovvero se: tempo virtuale attuale – tempo virtuale di ultimo riferimento  $< \tau$ , se è maggiore di  $\tau$  viene eliminata.

## Domanda F:

L'autore del seguente frammento di programma:

```

int k,s;
pid_t m,n;
k=0; n=fork();
if (n==(pid_t)0){
    DOWN
    k=k+5;
    UP
}
else{
    DOWN
    k=k+10;
    UP
    m=wait(&s);
    printf("k=%d\n",k);
}

```

si aspettava di vedere, con la printf dopo la wait, il valore 15, per effetto dei due aggiornamenti; ma verifica non essere così. Dopo essersi documentato sulla questione, intende correggere il programma utilizzando un semaforo o un mutex per garantire che le istruzioni `k=k+5;` e `k=k+10;` vengano eseguite in mutua esclusione, evitando corse critiche nell'aggiornamento della variabile. Di quali modifiche si tratta esattamente? Quale valore otterrà procedendo in tal modo, e quale può aver ottenuto inizialmente? Motivare le risposte.

Unix

## Risposta:

Le modifiche che ha attuato per correggere il programma sono: l'inizializzazione di un semaforo a 1 per poi inserire una down prima di "k=k+5" e un'altra down prima di "k=k+10" e dopo entrambi questi due assegnamenti una up.

Inizialmente avrà ottenuto il valore 10 e lo stesso valore otterrà sfruttando i cambiamenti che ha attuato poiché le modifiche sono irrilevanti dato che il processo figlio ha una propria area dei dati personale e le modifiche che il padre attua su k sono presenti solo nello spazio dei dati del padre.

## Domanda G:

Spiegare a che cosa può servire la variabile s nel programma del punto precedente (specie se utilizzata anche successivamente, oltre a quanto mostrato nel frammento).

Unix

## Risposta:

La variabile `s` serve a dare informazioni al processo padre su come è terminato il processo figlio: se il processo è terminato con `exit` o se il processo è terminato da un segnale e specifica di quale segnale si tratta.