Sistemi Operativi
16 febbraio 2011
Compitino B

Si risponda ai seguenti quesiti, giustificando le risposte.

1. Si descrivano le architetture dei kernel monolitici e di quelli a microkernel.

   Risposta: Le architetture dei kernel monolitici inglobano in questi ultimi praticamente tutto ciò che sta fra l'hardware e le chiamate di sistema. Se da un lato ciò permette di ottenere alte prestazioni, dall'altro rende il kernel poco flessibile in quanto non si distingue tra meccanismi (come fare qualcosa) e politiche (cosa deve essere fatto). Al massimo vi è una stratificazione in livelli in cui ognuno di questi ultimi fornisce delle primitive di base più astratte per l'implementazione del livello successivo. Lo strato di base (livello 0) consiste nell'hardware, mentre il più alto rappresenta l'interfaccia utente. Secondo il principio della modularità, gli strati sono pensati in modo tale che ognuno utilizza funzionalità (operazioni) e servizi solamente degli strati inferiori.

   Invece, in un sistema operativo basato su microkernel, il kernel è ridotto all'osso, fornendo soltanto i meccanismi di comunicazione tra processi, una minima gestione di questi ultimi e della memoria ed una gestione dell'hardware di basso livello (driver). Tutto il resto viene gestito da processi in spazio utente: ad esempio, tutte le politiche di gestione del file system, dello scheduling, della memoria sono implementate come processi. Ciò comporta una minor efficienza rispetto ad un kernel monolitico, ma anche una grande flessibilità ed un'immediata scalabilità in ambiente di rete.

2. Si descriva l'agoritmo di scheduling della CPU in Unix moderno.

   Risposta: Nello scheduling della CPU di Unix moderno si distinguono 3 classi di processi:

   - Real time: possono prelazionare il kernel, hanno priorità e quanto di tempo sso.
   - Kernel: prioritari su processi time shared, hanno priorità e quanto di tempo sso ed ogni coda è gestita con politica FCFS.
   - Time shared: per i processi "normali": ogni coda è gestita con politica round-robin, con quanto minore per priorità maggiore. La priorità è variabile secondo una tabella ssa: se un processo termina il suo quanto, scende di priorità. Ciò facilita i processi interattivi rispetto ai processi CPU-bound.

   Sostanzialmente si tratta di un sistema flessibile, modulare con politiche di default adatte ad un sistema time-sharing generale. C'è separazione tra meccanismi e politiche: il meccanismo è fissato, la politica di default può essere cambiata dall'amministratore di sistema, stabilendo:

   - intervallo di priorità che definisce una classe;
   - algoritmo di calcolo delle priorità;
   - quanti per i vari livelli;
   - migrazione da un livello all'altro.

3. Si consideri un sistema su cui vengono eseguiti 10 processi con prevalenza di I/O e un processo con prevalenza di elaborazione. Supponiamo che i primi processi richiedano un'operazione di I/O ogni millisecondo di elaborazione della CPU e che ciascuna di tali operazioni sia completata in 10 millisecondi. Si assuma inoltre che il tempo necessario per il cambio di contesto sia di 0.1 millisecondi e che tutti i processi siano operazioni a lungo termine. Si calcoli l'utilizzo della CPU in presenza di scheduler RR se il quanto di tempo è pari a 1 millisecondo.

   Risposta: Nei primi 11 ms la CPU serve i 10 processi $B_1,\dots B_{10}$ per 1 ms ciascuno più 1 ms totale di attività di context switch. All'istante 11 si ha 0,1 ms di context switch, poi parte il processo A (B_1 non è ancora disponibile: lo diventa all'istante 11,2) fino all'istante 12,1. A questo punto possiamo notare che ogni ms la coda ready non sarà mai vuota grazie alla presenza di A e dei vari B_i che termineranno via via le rispettive operazioni di I/O. Quindi in un ciclo abbiamo 11 ms di utilizzo della CPU e 1,1 ms di attività di context switch. Quindi la percentuale di utilizzo della CPU è data da $\frac{11}{12,1} \cong 0,9$, ovvero, il 90%.

4. Si descriva l'algoritmo del banchiere per la prevenzione dello stallo.

   Risposta: L'algoritmo del banchiere consente l'allocazione di una risorsa soltanto se lo stato conseguente rimane uno stato sicuro. Quindi, se una richiesta può portare ad uno stato non sicuro, essa non viene accettata.

L'algoritmo quindi opera dinamicamente controllando ad ogni richiesta se le risorse rimanenti sono sufficienti per soddisfare la massima richiesta di almeno un processo; in tal caso l'allocazione viene accordata, altrimenti viene negata. Funziona sia con istanze multiple che con risorse multiple.

I presupposti per il suo corretto funzionamento sono che:

- ogni processo deve dichiarare *a priori* l'uso massimo di ogni risorsa;
- quando un processo richiede una risorsa, può essere messo in attesa;
- quando un processo ottiene tutte le risorse che vuole, deve restituirle in un tempo finito.

Quindi la ragione principale che impedisce di adottare l'algoritmo del banchiere in un sistema operativo per eludere i deadlock è che necessita di conoscere a priori il tipo e la quantità di risorse che ogni processo andrà ad utilizzare nel corso della sua esecuzione. Inoltre, l'algoritmo si basa su un numero di processi fissato a priori, mentre nella realtà nuovi processi vengono continuamente creati dinamicamente.

5. Si consideri la soluzione al problema della sezione critica per due processi realizzata da Dekker. I due processi condividono le seguenti variabili:

```
int flag[2]={0, 0}; int turn=0; /* oppure 1 */
```

La struttura di P_i ($i = 0$ o 1) con P_j ($j = 1 - i$) è definita come segue:

```
do {
  /* entry section begins */
  flag[i]=1;

  while( flag[j] ) {

    if(turn == j) {
      flag[i]=0;
      while( turn == j );
      flag[i]=1;
    }

  }
  /* entry section ends */

  /* critical section */

  /* exit section begins */
  turn=j;
  flag[i]=0;
  /* exit section ends */

  /* remainder section */
} while true;
```

Si provi che l'algoritmo soddisfa i tre requisiti relativi al problema della sezione critica.

Risposta:

- Come prima cosa verifichiamo che i due processi P_i e P_j non possano accedere simultaneamente alla regione critica. Supponiamo che entrambi i processi abbiano dichiarato il loro interesse ad entrare nella sezione critica impostando il proprio flag a 1 (`flag[i]` per P_i e `flag[j]` per P_j); entrambi quindi eseguiranno le istruzioni all'interno dei cicli while esterni. A questo punto il valore della variabile `turn` consente di sbloccare uno dei due. Supponiamo che valga `0` (quindi sia a favore di P_i): P_j entrerà nel corpo dell'if, azzerando il proprio flag e ponendosi in attesa attiva nel ciclo while interno. P_i accorgendosi che il flag di P_j è stato azzerato entra nella

regione critica (uscendo dal while esterno). A questo punto soltanto P_i potrà trovarsi nella regione critica e vi rimarrà (mantenendo bloccato in attesa attiva P_j) fintanto che non eseguirà il codice dell'*exit section* impostando il valore di `turn` a `1` ed azzerando il proprio flag. La prima operazione fa uscire P_j dalla situazione di attesa attiva (while interno), riattivando il proprio flag, mentre la seconda fa effettivamente entrare P_j nella sezione critica. Se a questo punto P_i tentasse di rientrare nella sezione critica si bloccherebbe sul while interno (trovando attivato il flag di P_j e la variabile `turn` impostata a `1`).

- La seconda condizione è banalmente verificata in quanto le decisioni su quale processo possa entrare nella sezione critica vengono prese soltanto nella *entry section* e nella *exit section* (tramite la modifica dei flag e della variabile `turn`).
- Un processo al di fuori della sezione critica non deve attendere indefinitamente per entrarvi, dato che:
  - se l'altro processo non è interessato ad entrare nella sezione critica il suo flag sarà azzerato e l'accesso immediato;
  - se l'altro processo sta eseguendo nella sezione critica, al momento della sua uscita la variabile `turn` verrà aggiornata per dare la precedenza al processo attualmente all'esterno della sezione critica.

6. (a) Si descriva lo schema di traduzione degli indirizzi della memoria nella tecnica di paginazione.

   (b) Si consideri uno spazio di indirizzi logico di 32 pagine con 1024 parole per pagina e una memoria fisica di 16 frame.
   1. Quanti bit servono per specificare l'indirizzo logico?
   2. Quanti bit servono per specificare l'indirizzo fisico?
   3. Quante voci sono presenti nella tabella delle pagine?

Risposta:

(a) **Risposta:** La paginazione è una tecnica di allocazione della memoria non contigua che prevede:

- la suddivisione della memoria fisica in *frame* (blocchi di dimensione fissa, una potenza di 2, tra 512 e 8192 byte);
- la suddivisione della memoria logica in *pagine* (della stessa dimensione dei frame).

Quindi, per eseguire un programma di n pagine, servono n frame liberi in cui caricare il programma. È compito del sistema operativo tenere traccia dei frame liberi. Non esiste frammentazione esterna e la frammentazione interna è ridotta all'ultima pagina allocata al processo.

Per quanto riguarda la traduzione degli indirizzi logici in indirizzi fisici, ogni indirizzo generato dalla CPU si suddivide in un numero di pagina p e uno spiazzamento (offset) d all'interno della pagina. Il numero di pagina viene utilizzato come indice nella *page table* del processo correntemente in esecuzione: ogni entry della page table contiene l'indirizzo di base del frame fisico f che contiene la pagina in questione. Tale indirizzo di base f viene combinato (giustapposto come prefisso) con lo spiazzamento d per definire l'indirizzo fisico da inviare all'unità di memoria:

(b)
1. Per specificare l'indirizzo logico servono 15 bit in quanto $2^5 = 32$ sono le pagine logiche e ogni pagina contiene $2^{10} = 1024$ parole.
2. Per specificare l'indirizzo fisico servono 14 bit in quanto $2^4 = 16$ sono i frame fisici e ogni pagina/frame contiene $2^{10} = 1024$ parole.
3. Il numero di voci sono presenti nella tabella delle pagine coincide con il numero di pagine logiche, ovvero, 32.