Sistemi Operativi
Compitino Advanced Primo Periodo 6 aprile 2004

1. (a) Quali sono i vantaggi e gli svantaggi dell'implementazione dei thread a livello utente e a livello kernel, rispettivamente?

ULT
- Vantaggi: efficienza (non c'è chiamata di sistema al context switch); semplicità di implementazione su sistemi preesistenti, portabilità; lo scheduling può essere studiato per l'applicazione.
- Svantaggi: non essendoci scheduling automatico, c'è il rischio che un thread possa monopolizzare la CPU; l'accesso al kernel è sequenziale, nel senso che ogni chiamata bloccante al kernel da parte di un thread, mette in attesa tutti i thread del processo; non sfrutta sistemi multiprocessore; è poco utile per processi I/O bound.

KLT
- Vantaggi: lo scheduling è per thread e non per processo e quindi un thread che si blocca non blocca l'intero processo; è utile per processi I/O bound e sistemi multiprocessore.
- Svantaggi: è meno efficiente (ogni operazione sui thread comporta una chiamata di sistema); necessita l'aggiunta e la riscrittura di system call nei kernel preesistenti; è meno portabile; la politica di scheduling è fissata e non può essere modificata a seconda dell'applicazione.

(b) Si diano esempi di situazioni in cui è più vantaggiosa l'implementazione dei thread a livello kernel e a livello utente, rispettivamente.

Nel caso di sistemi multiprocessore sono vantaggiosi i thread di livello kernel. Su sistemi uniprocessore, i thread di livello kernel sono vantaggiosi nel caso di processi I/O bound, per es. un server web, mentre i thread di livello utente sono vantaggiosi nel caso di processi CPU bound, per es. un processo che esegue il mergesort di un array.

(c) In un sistema in cui i thread sono implementati a livello utente, perché un thread dovrebbe rilasciare la CPU chiamando *thread_yield*?

Perché, per poter continuare l'esecuzione, potrebbe avere bisogno che un altro thread svolga del lavoro, per es. nel caso di un thread produttore e un thread consumatore che condividono un buffer.

2. Si consideri un sistema con scheduling SJF con prelazione (cioè SRTF), ove $\alpha = 0,5$ e $\tau_0 = 30$ msec. All'istante 0 il processore si libera e tre processi, P_1, P_2, P_3, sono in coda ready. Finora i processi P_1, P_2 sono andati in esecuzione due volte con CPU burst $30, 20$ msec per P_1 e $25, 40$ msec per P_2; mentre P_3 è andato in esecuzione una volta con CPU burst di 50 msec.

Si determini:

(a) Quale processo viene selezionato dallo scheduler all'istante 0?

P_1

(b) All'istante 10 msec entra nella coda ready un nuovo processo P_4 con CPU burst previsto di 20 msec. Il processo selezionato precedentemente è ancora in esecuzione. Che cosa succede?

Continua l'esecuzione P_1, perchè il CPU burst rimanente previsto per P_1 è inferiore al CPU burst previsto per P_4.

(c) Che cosa succede quando il processo in esecuzione termina il suo burst?

Andrà in esecuzione P_1, se il suo prossimo CPU burst previsto è inferiore a quello previsto per P_4, cioè 20 msec, altrimenti andrà in esecuzione P_4.

3. Si consideri la segreteria di uno studio dentistico che si deve occupare di gestire le prenotazioni degli appuntamenti stabilendo a priori quanto tempo allocare per ogni appuntamento in base al tipo di intervento da eseguire. Si vuole cercare di massimizzare il carico di lavoro del dentista (evitando quindi di lasciare tempi morti fra gli appuntamenti). Si deve comunque garantire la possibilità di gestire delle urgenze. Per risolvere situazioni di urgenza è ammesso riorganizzare gli appuntamenti. Si studi un algoritmo di scheduling degli appuntamenti e lo si inquadri in una delle classi di algoritmi di scheduling visti a lezione. Si provi a considerare la situazione in cui le urgenze vanno evase entro determinate deadlines e gli appuntamenti vengano spostati, in prima istanza, solo se le persone coinvolte sono daccordo allo spostamento (tramite accordi telefonici) e, come estrema risorsa, si faccia aspettare un utente che oramai si trova già nello studio dentistico. Alternativamente si semplifichi considerando il caso in cui tutti gli utenti scoprono dello slittamento dell'appuntamento una volta in loco.

4. In un museo con vari ingressi distinti si vuole evitare di far entrare troppe persone contemporaneamente. Nel museo ci sono due aree distinte che possono contenere non più di N e M persone rispettivamente. Le biglietterie consentono di entrare solo nell'area 1. Fra le 2 aree c'è una porta che lascia passare le persone una alla volta sotto controllo di un processo `transitManager` che deve evitare di superare la capienza dell'area 2.

   Si vuole associare ad ogni terminale della biglietteria un processo `ticketOffice` in grado di accettare un nuovo visitatore solo se effettivamente non si va a superare la capienza dell'area 1. Sulle uscite, presenti solo nell'area 2, abbiamo invece un conta-persone in grado di interagire con un processo `customerExits` per determinare quando si liberano posti.

   Si completi il seguente codice concorrente usando opportunamente le primitive `up(&semaph)` e `down(&semaph)`.

```
[dich. e inizializz. semafori/o]

void ticketOffice(void) {
while (TRUE) {
     wait_for_customer();
     [ sez1 ]
     let_customer_in();
     }
}

void transitManager(void) {
while (TRUE) {
     wait_passing_customer();
     [ sez2 ]
     let_customer_pass();
     }
     [ sez3 ]
}

void customerExits(void) {
while (TRUE) {
     wait_customer_exits();
     [ sez4 ]
     }
}
```

```
semaphore area1=N;
semaphore area2=M;

void ticketOffice(void) {
while (TRUE) {
     wait_for_customer();
```

```
    down(&area1);
    let_customer_in();
    }
}

void transitManager(void) {
while (TRUE) {
    wait_passing_customer();
    down(&area2);
    let_customer_pass();
    up(&area1);
    }
}

void customerExits(void) {
while (TRUE) {
    wait_customer_exits();
    up(&area2);
    }
}
```

5. Si consideri un sistema con n risorse di uno stesso tipo (p.e., n stampanti), e dove ogni processo può utilizzare al massimo m risorse di tale tipo.
   (a) Senza alcun sistema di assegnazione delle risorse, quanti processi possiamo eseguire contemporaneamente garantendo comunque l'assenza di deadlock?

   La situazione peggiore è quando tutti gli x processi detengono $m-1$ risorse (per un totale di $x(m-1)$ risorse allocate), e per continuare tutti ne chiedono ancora una. Per garantire l'assenza di deadlock, almeno una di queste richieste deve essere soddisfatta, e quindi deve essere $x(m-1) \leq n-1$, ossia $x \leq \frac{n-1}{m-1}$. Il valore massimo ammissibile è quindi $x = \left\lfloor \frac{n-1}{m-1} \right\rfloor$.

   (b) Quali metodi di deadlock detection possiamo applicare in questo caso e quali no?

   Il grafo no, abbiamo istanze multiple. Solo la matrice.

   (c) Usando l'algoritmo del banchiere, quanti processi potremmo mandare in esecuzione contemporaneamente?

   Non c'è alcun limite: l'algoritmo automaticamente blocca i processi che dovessero richiedere troppe risorse rispetto a quante disponibili, e questo indipendentemente dal numero di processi in esecuzione.