UNIVERSITÀ DEGLI STUDI DI BERGAMO

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE E METODI MATEMATICI

Corso di Laurea in Ingegneria Informatica/Meccanica

Esame di **Sistemi Operativi**

(*Modulo di Informatica II - 21013+23014*)

(Modulo del C.I. di Reti di calcolatori e Sistemi operativi)

Appello 11 Febbraio 2010

1. Spiegare il concetto di *page fault* (mancanza di pagina). **[max 4 pt]**

2. Nell'ambito dei meccanismi di gestione del deadlock: **[max 8 pt]**

I. Definire il concetto di *deadlock* ed illustrare le tecniche per "**prevenire** il deadlock".

II. E' possibile avere un deadlock che coinvolge soltanto un processo con singolo thread? Motivare la risposta.

3. Nell'ambito delle tecniche di *implementazione del file system*: **[max 8 p.t.]**

I. Descrivere la tecnica di *allocazione contigua* dei file su disco;

II. Si consideri un file system con dimensione del blocco di **512 byte**. Si assuma che il sistema adotti la tecnica di *allocazione contigua* descritta al punto I. Si assuma inoltre che nel *descrittore del file* siano riservati **32 bit** per l'indirizzamento del file. Qual è la dimensione massima del file system (ovvero la dimensione massima di un file)? Motivare la risposta.

4. *Quesito riservato agli studenti di* ***Informatica II (21013+23014)*****: [max 10 p.t.]**

Il problema del *Single Lane Bridge* Si consideri il problema di concorrenza rappresentato in Figura 1. Un ponte (*bridge*) lungo un fiume è talmente stretto da permettere una sola carreggiata di transito da usare in modo *bidirezionale* dalle autovetture provenienti da entrambe le direzioni (auto rosse da sinistra e auto blu da destra). Uno scontro tra due autovetture (violazione della proprietà di *safety*) avviene se due autovetture (vedi Figura 1) provenienti da direzioni opposte (una rossa ed una blu) entrano nel ponte nello stesso tempo.

Figura 1

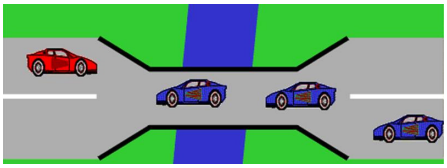

Figura 2

La Figura 2 suggerisce una soluzione al problema del transito attraverso una turnazione tra macchine blu e rosse. Una possibile soluzione in Java è riportata nel secondo riquadro sotto, vedi la classe `SafeBridge`, sfruttando il meccanismo di sincronizzazione diretta (i metodi *wait()/notify()*). La classe `SafeBridge` implementa l'interfaccia `Bridge` riportata nel primo riquadro; tale interfaccia contiene i metodi eseguiti dai thread (le auto rosse e blu) per l'ingresso (*enter*) e l'uscita (*exit*) dal ponte.

```
interface Bridge {
   abstract void redEnter() throws InterruptedException;
   abstract void redExit();
   abstract void blueEnter() throws InterruptedException;
   abstract void blueExit();
}
```

```
class SafeBridge implements Bridge{
    private int nred  = 0;
    private int nblue = 0;

    synchronized public  void redEnter() throws
      InterruptedException {
        while (nblue>0) wait();
        ++nred;
    }

    synchronized public void redExit(){
        --nred;
        if (nred==0)
            notifyAll();
    }

    synchronized public void blueEnter() throws
     InterruptedException {
        while (nred>0) wait();
        ++nblue;
    }

    synchronized public void blueExit(){
        --nblue;
        if (nblue==0)
            notifyAll();
    }
```

I. Si noti che la soluzione proposta (la classe `SafeBridge`) soffre del problema della *starvation* dei processi. Perché? Motivare la risposta.

II. Provare a modificare la soluzione data per "garantire progresso" a tutti i thread (evitando così la starvation). A tale scopo fornire una definizione per la classe `FairBridge` che nasce come modifica della classe `SafeBridge`.

[Suggerimento: Nota che è possibile modificare la classe `SafeBridge` con nuovi attributi per mantenere conteggi di varia natura sulle auto (thread) in transito sul ponte e con nuove condizioni nei metodi. Non occorre definire le classi dei thread.*]*

5. *Quesito per gli studenti del* ***C.I. di Reti di calcolatori e Sistemi operativi:*** **[max 10 pt]**

Dato il seguente codice:

```
1. int main ( ) {
2.   int a, b, stato, pid;
3.   for (a = 0; a < 1; a++) {
4.       pid = fork ( );
5.       if (pid == 0) {
6.         if (a < 1) {
7.             pid = fork ( );
8.             if (pid == 0) {
9.               b=3;
10.              exit (a + b);
```

```
11.              } /* end if * /
12.             wait (&stato);
13.           } /* end if * /
14.           exit (a);
15.       } /* end if * /
16.       if (a == 0) wait (&stato);
17.   } /* end for * /
18. pid = wait (&stato);
19.} /* end main * /
```

Per ciascun processo creato nel codice di cui sopra, fornire una tabella del tipo:

Processo	variabile a	variabile b	variabile pid	variabile stato
subito prima dell'istruzione 5*				
subito prima dell'istruzione 10				
subito dopo l'istruzione 18				

dove occorre indicare i valori delle variabili negli istanti di tempo specificati. Si assuma la prima esecuzione dell'istruzione 5. e che tutte le chiamate ai servizi di sistema abbiano sempre successo. Attenzione, indicare:

- **NE**, quando la variabile non esiste (in quanto non esiste il processo)
- **U**, quando non si può dire con certezza se la variabile esista o quale ne sia il valore
- il **pid** del processo, tenendo conto che il per il padre è 100 e poi il S.O. assegna pid consecutivi in ordine di creazione.

SOLUZIONE

1. Vedi capitolo 10, Sez.10.2.1 del libro adottato.

2.I Vedi Sez. 8.4 del libro adottato.

2.II Risp. No, segue dalla condizione necessaria "possesso e attesa" (vedi Sez. 8.2.1)

3.I Vedi Sez. 12.4 del libro adottato.

3.II Risp. Nel caso di allocazione contigua, il descrittore del file riserva un campo per memorizzare il *primo blocco* sul disco utilizzato dal file, che in questo caso ha una dimensione di 32 bit. La dimensione massima del file system è legata al numero di blocchi indirizzabili mediante l'indirizzo contenuto nel descrittore del file: in questo caso, quindi, il numero dei blocchi indirizzabili è pari a 2^{32} . La dimensione max del file system è quindi:

$$D_{max} = 2^{32} * 512 \text{ B} = 2 \text{ TB}$$

4.I La soluzione data assicura solo che non ci sono collisioni sul ponte (proprietà di *safety*), in quanto è solo l'ultima macchina di un certo colore (rossa o blu) a lasciare il ponte e a svegliare eventuali macchine in attesa di passare. Questo, però, non garantisce che le macchine di un certo colore prima o poi passeranno (possibilità di starvation); se ad es. è presente una fila lunga di macchine rosse e le "rosse" hanno avuto l'opportunità di passare, le eventuali macchine blu aspetteranno un tempo *indefinitamente lungo* (devono prima passare tutte le "rosse") per poter passare.

4.II Risp. Vengono introdotti dei contatori, i campi *waitred/waitblue*, per tener traccia del numero di macchine rosse/blu "in attesa" ("wait") di passare dal ponte, ed un campo booleano *blueturn* per stabilire il turno tra le auto rosse e le auto blu. I metodi *redEnter/blueEnter* vengono modificati in modo da incrementare (prima dell'accesso al ponte) e decrementare (dopo l'accesso al ponte) i contatori di attesa; la condizione nel ciclo "while" adesso riflette la seguente *politica di accesso*: le auto rosse [blu] possono accedere al ponte se e solo se non ci sono auto blu [rosse] (cioè `nred<=0` [`nblue<=0`]) e non ci sono auto blu [rosse] in attesa (cioè `!(waitblue>0 && blueturn)` [`!(waitred>0 && !blueturn)`]). La condizione di attesa nel while è esattamente la negazione di questa politica. Tale politica è esente da starvation.

```
class FairBridge extends Bridge {
    private int nred  = 0;
    private int nblue = 0;
    private int waitblue = 0;
    private int waitred = 0;
    private boolean blueturn = true;

    synchronized public void redEnter() throws InterruptedException {
        ++waitred;
        while (nblue>0 || (waitblue>0 && blueturn)) wait();
        --waitred;
        ++nred;
    }

    synchronized public void redExit(){
        --nred;
        blueturn = true;
        if (nred==0)
            notifyAll();
    }

   synchronized public void blueEnter()  throws InterruptedException {
        ++waitblue;
        while (nred>0 || (waitred>0 && !blueturn)) wait();
        --waitblue;
        ++nblue;
    }
```

```
    synchronized public void blueExit(){
         --nblue;
         blueturn = false;
         if (nblue==0)
             notifyAll();
     }
}
```

5.

Processo 100	variabile a	variabile b	variabile pid	variabile stato
subito prima dell'istruzione 5*	*0*	*U*	*101*	*U*
subito prima dell'istruzione 10	*0*	*U*	*101*	*U*
subito dopo l'istruzione 18	1	*U*	*102*	*0*

Processo 101	variabile a	variabile b	variabile pid	variabile stato
subito prima dell'istruzione 5	*0*	*U*	*0*	*U*
subito prima dell'istruzione 10	*0*	*U*	*102*	*U*
subito dopo l'istruzione 18	*NE*	*NE*	*NE*	*NE*

Processo 102	variabile a	variabile b	variabile pid	variabile stato
subito prima dell'istruzione 5	*NE*	*NE*	*NE*	*NE*
subito prima dell'istruzione 10	*0*	3	*0*	*U*
subito dopo l'istruzione 18	*NE*	*NE*	*NE*	*NE*