Esercizi sul problema del deadlock

1. Si consideri la situazione in cui vi siano in esecuzione quattro processi P_A, P_B, P_C e P_D con la matrice delle risorse allocate e la matrice del numero massimo di risorse di cui possono disporre come segue:

$$C = \begin{bmatrix} 1 & 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$Max = \begin{bmatrix} 1 & 1 & 2 & 1 & 3 \\ 2 & 2 & 2 & 1 & 1 \\ 2 & 1 & 3 & 1 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{bmatrix}$$

   Se il vettore delle risorse disponibili è $A = (0, 0, x, 1, 1)$, qual è il minimo valore di x che rende lo stato attuale *safe*?

   Soluzione: come prima cosa calcoliamo la matrice R delle richieste come risultato della sottrazione $Max - C$:

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 & 2 \\ 0 & 2 & 1 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

   Ovviamente, se $x = 0$, lo stato non è sicuro in quanto non esiste alcuna riga R_i tale che $R_i \leq A$.

   Quindi proviamo a supporre che x valga 1: l'unica riga minore o uguale al vettore A è la quarta (corrispondente alle richieste del processo P_D):

$$R_4 = (0, 0, 1, 1, 1) = A$$

   Quindi viene eseguito il processo P_D a cui vengono assegnate le risorse richieste. Quando termina restituisce le risorse allocate (riga C_4) al sistema; il vettore A diventa quindi $(1, 1, 2, 2, 1)$. A questo punto il nuovo stato (e quindi anche lo stato di partenza) non è sicuro in quanto non esiste alcuna riga $R_i \leq A$ $(i = 1, 2, 3)$. Quindi anche il caso $x = 1$ non va bene.

   Se $x = 2$ invece riusciamo a trovare una sequenza di esecuzione di P_A, P_B, P_C e P_D tale che tutti i processi terminano vedendo soddisfatte le proprie richieste:

   - al primo passo esegue P_D e A diventa $(1, 1, 3, 2, 1)$;
   - al secondo passo esegue P_C e A diventa $(2, 2, 3, 3, 1)$;
   - al terzo passo esegue P_B e A diventa $(4, 2, 4, 4, 2)$;
   - al quarto passo esegue P_A e A diventa $(5, 2, 6, 5, 3)$.

   Pertanto il minimo valore di x tale da rendere lo stato iniziale sicuro è 2.

2. Si consideri un sistema con due processi e tre risorse identiche (ovvero, tre istanze di un'unica tipologia di risorsa). Se ogni processo necessita al massimo di due risorse, può avvenire un deadlock? Perché?

   Risposta: nelle condizioni esposte non è possibile arrivare al deadlock in quanto, anche nella condizione peggiore in cui ognuno dei due processi disponga di una risorsa e sia in attesa della seconda, il sistema dispone comunque di un'ulteriore risorsa libera da allocare ad uno dei due processi in attesa.

3. Consideriamo una generalizzazione del caso precedente con p processi che necessitino al massimo di m istanze di una tipologia di risorsa con r istanze disponibili. Qual è la condizione che rende il sistema "deadlock free"?

   Soluzione: generalizzando il caso precedente, nella situazione peggiore ognuno dei p processi avrebbe a disposizione $m - 1$ istanze e sarebbe in attesa dell'ultima istanza necessaria per completare il proprio lavoro. Quindi il vincolo si esprime con la seguente espressione:

$$r \geq p(m-1) + 1$$

4. Senza un sistema di assegnazione delle risorse con una politica di deadlock avoidance, con sei unità a nastro quanti processi che necessitino di al massimo due unità può mandare in esecuzione senza rischiare il deadlock?

   Usando invece l'algoritmo del banchiere, quanti processi potremmo mandare in esecuzione contemporaneamente senza rischiare l'insorgere di deadlock?

   Soluzione: isolando la variabile p nell'espressione del vincolo dell'esercizio precedente si ottiene quanto segue:

$$p \leq \left\lfloor \frac{r-1}{m-1} \right\rfloor$$

   quindi nel caso in cui $r = 6$ e $m = 2$, si ha $p = 5$.

   Con l'algoritmo del banchiere invece non ci sono restrizioni sul numero dei processi che è possibile mandare in esecuzione in quanto l'algoritmo consentirà l'assegnamento di una risorsa soltanto nel caso in cui lo stato rimanga *sicuro* (altrimenti l'assegnamento sarà rifiutato ed il processo sospeso).

5. Cenerentola e il principe vogliono divorziare e, per dividere il loro patrimonio, concordano sull'utilizzo del seguente algoritmo:

   - ogni mattina Cenerentola ed il principe inviano al legale dell'altro una lettera in cui richiedono un oggetto del loro patrimonio;
   - visto che le lettere ci mettono una giornata per giungere a destinazione, si accordano sul fatto che, in caso vengano a richiedere nello stesso giorno lo stesso oggetto, il giorno successivo invieranno una lettera con cui annulleranno la richiesta;
   - tra le loro proprietà vi sono il cane Woofer, la sua cuccia, il canarino Tweeter e la sua gabbia: siccome questi animali tengono molto alle

loro "case", gli ex sposi concordano sul fatto che ogni divisione che separi un animale dalla sua casa, debba essere annullata e debba ripartire da zero.

Sia Cenerentola che il principe vogliono a tutti i costi ottenere Woofer e prima di partire in vacanze (separate), affidano la gestione del negoziato ai loro computer.

Quando ritornano dalle loro vacanze trovano i computer che sono ancora in fase di negoziazione. Come mai? È possibile un deadlock? E una starvation?

Risposta: il deadlock è ovviamente possibile nel caso in cui i due contendenti richiedano entrambi un oggetto già precedentemente assegnato all'altro (in questo caso infatti le richieste anche se avvenissero lo stesso giorno riguarderebbero oggetti distinti e non vi sarebbero annullamenti delle richieste).

Anche la starvation è possibile; infatti, dato che entrambi vogliono ottenere Woofer, è molto probabile che lo richiedano come prima cosa. Quindi vi saranno due richieste identiche il primo giorno, due annullamenti il giorno successivo, due nuove richieste identiche per Woofer il terzo giorno, due nuovi annullamenti il quarto giorno ecc. In pratica i due processi non riusciranno a progredire (starvation).

Un'altra possibile causa di starvation è una suddivisione degli oggetti che separi almeno uno dei due animali dalla propria "casa"; infatti in questo caso l'intero procedimento di suddivisione dovrebbe ricominciare da zero.

6. Usando solo semafori binari, come si può garantire una corretta sincronizzazione tra due produttori e tre consumatori che accedano ad un buffer realizzato come variabile condivisa?

   A quali condizioni viene associato ciascuno dei semafori?

   Cosa dovrà fare ogni produttore e ogni consumatore?

   Soluzione: tenendo presente che una variabile condivisa è sostanzialmente equivalente ad un buffer di lunghezza 1, introduciamo due semafori binari:

   (a) SNV associato alla condizione *buffer non vuoto*;

   (b) SNP associato alla condizione *buffer non pieno*.

   Il codice eseguito da ogni produttore sarà conforme a quanto segue:

```
down(&SNP);
produce_item();
up(&SNV);
```

   Il codice eseguito da ogni consumatore sarà conforme a quanto segue:

```
down(&SNV);
consume_item();
up(&SNP);
```

La soluzione proposta è indipendente dal numero di produttori e consumatori.

7. 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.

Soluzione:

- 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.

8. Si consideri un sistema con m risorse della stessa tipologia contese da n processi. Assumendo che le risorse possano essere richieste e rilasciate dai processi una alla volta, sotto quali condizioni il sistema può essere considerato *deadlock free*?

   Soluzione: indicando con max_i $(1 \leq i \leq n)$ il fabbisogno massimo del processo P_i, la situazione peggiore è quella in cui ogni processo P_i ha ottenuto $max_i - 1$ risorse e necessita di un'ulteriore istanza per terminare il proprio lavoro. Quindi abbiamo il seguente vincolo per evitare deadlock:

   $$m \geq \sum_{i=1}^{n} (max_i - 1) + 1$$

   da cui:

   $$\begin{gathered} m \geq \textstyle\sum_{i=1}^{n} max_i - n + 1 \\ m > \textstyle\sum_{i=1}^{n} max_i - n \\ \textstyle\sum_{i=1}^{n} max_i < m + n \end{gathered}$$

   Quindi la somma dei fabbisogni massimi dei processi deve essere inferiore alla somma del numero di istanze m e del numero dei processi in gioco n.

9. Due processi P_A e P_B necessitano entrambi di tre record (1, 2 e 3) di un database per completare la propria esecuzione. Si discutano le possibili situazioni di deadlock a seconda dell'ordine della richiesta dei tre record da parte di P_A e P_B.

Risposta: ogni processo può richiedere i tre record in 3!=6 modi diversi:
1 2 3
2 1 3
2 3 1
1 3 2
3 1 2
3 2 1

Quindi ci sono $6^2 = 36$ combinazioni possibili. Le combinazioni che sicuramente non portano a deadlock sono quelle che iniziano con lo stesso record sia da parte di P_A che di P_B. Infatti in questo caso il record verrà assegnato ad uno dei due, mentre l'altro verrà sospeso; a questo punto il processo che ha ottenuto la risorsa potrà bloccare anche gli altri due record, completare il proprio lavoro e rilasciare le risorse in modo che anche l'altro processo possa completare la propria esecuzione. Quindi le combinazioni "sicure" sono soltanto 12 su 36, ovvero, un terzo del totale.

10. Si consideri un sistema informatico che esegua 5.000 job al mese; non essendo dotato di nessuno schema di prevenzione/elusione dei deadlock, capitano in media due stalli al mese. In questi casi deve terminare manualmente dieci job e rimandarli in esecuzione.

    Si supponga che il costo del tempo di CPU di ogni job sia di 2 $ e che i job abbiano completato metà del loro lavoro nel momento in cui l'operatore è costretto a terminarli per risolvere lo stallo.

    Un programmatore incaricato di studiare il problema stima che, introducendo un algoritmo di deadlock avoidance (e.g., l'algoritmo del banchiere), il tempo di esecuzione di ogni job incrementerà del 10 %.

    Sapendo che al momento attuale la CPU rimane inattiva per il 30 % del tempo,

    - il sistema sarà ancora in grado di far "girare" i 5.000 job/mese?
    - discutere vantaggi/svantaggi dell'introduzione dell'algoritmo di deadlock avoidance.

    Risposta:

    - Il sistema sarà ancora in grado di far "girare" i 5.000 job in quanto il tempo di CPU salirà dal 70 % al 77 %.
    - Per quanto riguarda i vantaggi/svantaggi dell'introduzione dell'algoritmo di deadlock avoidance, facendo una valutazione puramente economica, abbiamo quanto segue:
      - senza deadlock avoidance, il costo del sistema è dato dalla somma di $4.980 \times 2\$$ (costo dei job eseguiti "normalmente"), di $20 \times 1\$$ (costo dell'esecuzione "a metà" dei 20 job che bisogna far ripartire) e di $20 \times 2\$$ (costo dei 20 job rieseguiti): il totale ammonta a 10.020$;
      - con l'algoritmo di deadlock avoidance non ci sono job bloccati da rieseguire ed il costo è dato semplicemente da $5.000 \times 2,2\$ =$ 11.000$.

Quindi c'è una spesa aggiuntiva di 980$ introdotta dall'algoritmo di deadlock avoidance che farebbe propendere per restare alla situazione precedente.

11. Si consideri la seguente situazione:

	Allocation				Max			
	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2
P_1	1	0	0	0	1	7	5	0
P_2	1	3	5	4	2	3	5	6
P_3	0	6	3	2	0	6	5	2
P_4	0	0	1	4	0	6	5	6

Available (A)			
A	B	C	D
1	5	2	0

- Calcolare la matrice R delle richieste.
- Il sistema è in uno stato sicuro (safe)?
- Nel caso arrivi la richiesta di allocazione $(0, 4, 2, 0)$ per il processo P_1, quest'ultima può essere soddisfatta?

Soluzione:

- La matrice R delle richieste si ottiene per differenza (Max - Allocation):

	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

- Il sistema è in uno stato sicuro; infatti esiste una sequenza di esecuzione che permette a tutti i processi di concludere il loro lavoro anche nel caso in cui richiedano immediatamente tutte le risorse di cui necessitano:
  - P_0 esegue per primo; valore aggiornato di A: $(1, 5, 3, 2)$;
  - P_2 esegue per primo; valore aggiornato di A: $(2, 8, 8, 6)$;
  - P_1 esegue per primo; valore aggiornato di A: $(3, 8, 8, 6)$;
  - P_3 esegue per primo; valore aggiornato di A: $(3, 14, 11, 8)$;
  - P_4 esegue per primo; valore aggiornato di A: $(3, 14, 12, 12)$.
- Sì, la richiesta per P_1 può essere soddisfatta immediatamente. Essa porta i seguenti aggiornamenti per quanto riguarda Allocation, R e Available:

	Allocation				R			
	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	0	0
P_1	1	4	2	0	0	3	3	0
P_2	1	3	5	4	1	0	0	2
P_3	0	6	3	2	0	0	2	0
P_4	0	0	1	4	0	6	4	2

Available (A)			
A	B	C	D
1	1	0	0

Il nuovo stato è ancora safe; a riprova di quanto detto esiste la seguente sequenza di esecuzione:

- P_0 esegue per primo; valore aggiornato di A: $(1, 1, 1, 2)$;
- P_2 esegue per primo; valore aggiornato di A: $(2, 4, 6, 6)$;
- P_1 esegue per primo; valore aggiornato di A: $(3, 8, 8, 6)$;
- P_3 esegue per primo; valore aggiornato di A: $(3, 14, 11, 8)$;
- P_4 esegue per primo; valore aggiornato di A: $(3, 14, 12, 12)$.

12. Il gestore di un parcheggio vuole automatizzare il sistema di ingresso/uscita in modo da impedire l'accesso quando il parcheggio è pieno (i.e., la sbarra d'ingresso deve sollevarsi solo se ci sono posti disponibili) e da consentire l'uscita solo in caso di parcheggio non vuoto (i.e., la sbarra d'uscita deve rimanere abbassata in caso di parcheggio vuoto). Scrivere un monitor che permetta di gestire l'accesso al parcheggio.

    Soluzione:

```
monitor CarPark
    condition notfull,notempty;
    integer spaces, capacity;

    procedure enter();
        begin
            while(spaces=0) do wait(notfull);
            spaces := spaces-1;
            signal(notempty);
        end;

    procedure exit();
        begin
            while(spaces=capacity) do wait(notempty);
            spaces := spaces+1;
            signal(notfull);
        end;

    spaces := N;
    capacity := N;
end monitor;
```

13. Un ponte che collega due sponde di un fiume è così stretto che permette il passaggio delle auto su un'unica corsia a senso alternato (quindi le macchine possono muoversi concorrentemente sul ponte solo se vanno nella stessa direzione).

    - Per semplicità ci si riferisca alle macchine che procedono da sinistra a destra con l'espressione "RedCars" e a quelle che procedono da destra a sinistra con l'espressione "BlueCars".
    - Si scriva il codice di gestione dell'accesso al ponte in modo da evitare incidenti fra "RedCars" e "BlueCars", usando i monitor.
    - C'è possibilità di starvation? In caso di risposta affermativa riscrivere il codice in modo da evitare questo problema.

    Soluzione:

```
monitor OneWayBridge
    condition bridgeFree;
    integer nred, nblue;

    procedure redEnter();
        begin
            while(nblue>0) do wait(bridgeFree);
            nred := nred+1;
        end;

    procedure redExit();
        begin
            nred := nred-1;
            if(nred=0) then signal(bridgeFree);
        end;

    procedure blueEnter();
        begin
            while(nred>0) do wait(bridgeFree);
            nblue := nblue+1;
        end;

    procedure blueExit();
        begin
            nblue := nblue-1;
            if(nblue=0) then signal(bridgeFree);
        end;

    nred := 0;
    nblue := 0;
end monitor;
```

L'invariante soddisfatta dal monitor è la seguente:

$$\texttt{nred} \geq 0 \wedge \texttt{nblue} \geq 0 \wedge \neg(\texttt{nred} > 0 \wedge \texttt{nblue} > 0)$$

Quindi non ci possono essere incidenti fra "RedCars" e "BlueCars". Tuttavia può esserci *starvation*; infatti supponiamo che alle due estremità del ponte arrivino una "RedCar" ed una "BlueCar" e che la prima ottenga il lock del monitor entrando sul ponte (eseguendo il metodo `redEnter()`). A questo punto la "BlueCar" si sospende in attesa dell'evento `bridgeFree`. Nel frattempo, in presenza di un flusso costante di "RedCars", queste ultime riusciranno ad accedere al ponte mantenendo il valore di `nred` strettamente maggiore di zero. La conseguenza di tutto ciò è che la "BlueCar" non riuscirà ad accedere al ponte (starvation).

Una variante che risolve questo problema, introducendo il meccanismo di *turno*, è la seguente:

```
monitor FairBridge
    condition bridgeFree;
```

```
    integer nred, nblue, waitred, waitblue;
    boolean blueturn;

    procedure redEnter();
        begin
            waitred := waitred+1;
            while(nblue>0 or (waitblue>0 and blueturn)) do wait(bridgeFree);
            waitred := waitred-1;
            nred := nred +1;
        end;

    procedure redExit();
        begin
            nred := nred-1;
            blueturn := true;
            if(nred=0) then signal(bridgeFree);
        end;

    procedure blueEnter();
        begin
            waitblue := waitblue+1;
            while(nred>0 or (waitred>0 and not(blueturn))) do wait(bridgeFree);
            waitblue := waitblue-1;
            nblue := nblue +1;
        end;

    procedure blueExit();
        begin
            nblue := nblue-1;
            blueturn := false;
            if(nblue=0) then signal(bridgeFree);
        end;

    nred := 0;
    nblue := 0;
    waitred := 0;
    waitblue := 0;
    blueturn := true;
end monitor;
```

In questo modo ogni "RedCar", uscendo dal ponte, imposta il turno per le "BlueCar" (`blueturn := true`) e, viceversa, ogni "BlueCar", uscendo dal ponte, imposta il turno per le "RedCar" (`blueturn := false`). Un'auto, prima di accedere al ponte, controlla che non ci siano macchine che marciano nella direzione opposta sul ponte e che non ci siano macchine in attesa con il turno di passaggio a loro favorevole.

Modern Operating Systems Simulators

Al seguente URI

`http://www.ontko.com/moss/`

si possono scaricare dei simulatori (scritti in Java) che illustrano i concetti chiave (scheduling, gestione della memoria, deadlock ecc.) presentati nel testo
Andrew S. Tanenbaum
Modern Operating System
Second Edition (Prentice-Hall, 2001)

In particolare si suggerisce di programmare il simulatore di **deadlock** per verificare le differenti situazioni di contesa delle risorse viste a lezione:

`http://www.ontko.com/moss/#deadlock`
`http://www.ontko.com/moss/deadlock/user_guide.html`