Esercizi

1. Si consideri la seguente situazione, dove P_0, P_1, P_2 sono tre processi in esecuzione, C è la matrice delle risorse correntemente allocate, Max è la matrice del numero massimo di risorse assegnabili ad ogni processo e A è il vettore delle risorse disponibili:

	C			Max		
	A	B	C	A	B	C
P_0	0	2	0	0	5	1
P_1	0	0	0	2	4	3
P_2	2	3	0	2	4	2

Available (A)		
A	B	C
1	5	x

   (a) Calcolare la matrice R delle richieste.

   (b) Determinare il minimo valore di x tale che il sistema si trovi in uno stato sicuro.

   Soluzione:

   (a) La matrice R delle richieste è data dalla differenza $Max - C$:

R		
A	B	C
0	3	1
2	4	3
0	1	2

   (b) Se $x = 0$, allora non esiste nessuna riga R_i tale che $R_i \leq A$; quindi il sistema si trova in uno stato di deadlock. Se $x = 1$, allora l'unica riga di R minore o uguale a A è la prima. Quindi possiamo eseguire P_0 che, una volta terminato, restituisce le risorse ad esso allocate aggiornando A al valore $(1, 7, 1)$. A questo punto non esiste alcuna riga di R minore o uguale al vettore A e quindi il sistema è in stato di deadlock. Analogamente, se $x = 2$, dopo aver eseguito P_0, il valore di A è $(1, 7, 2)$. A questo punto si può eseguire P_2 aggiornando A al valore $(3, 10, 2)$: dato che l'unica riga di R rimasta da considerare non è minore o uguale al valore corrente di A, il sistema è in stato di deadlock.

   Il valore minimo di x per cui lo stato risulta sicuro è 3; infatti in questo caso esiste la sequenza sicura $\langle P_0, P_2, P_1 \rangle$. Dapprima si esegue P_0, generando il valore $(1, 7, 3)$ di A, poi si esegue P_2 portando A al valore $(3, 10, 3)$. A questo punto si conclude la sequenza eseguendo P_1 e generando il valore finale di A, ovvero, $(3, 10, 3)$.

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

$$C = \begin{bmatrix} 3 & 0 & 1 \\ 0 & 2 & 4 \\ 1 & 5 & 2 \end{bmatrix} \qquad Max = \begin{bmatrix} 3 & 2 & 2 \\ 1 & 2 & 5 \\ 2 & 6 & 3 \end{bmatrix}$$

Se il vettore delle risorse disponibili è $A = (2, 1, 1)$, lo stato attuale è sicuro (safe). Si supponga che arrivi la richiesta $(0, 1, 1)$ relativa al processo P_A: può essere soddisfatta subito? Ovvero, il nuovo stato generato soddisfando la richiesta è sicuro? Perchè?

Soluzione: supponendo di accettare la richiesta per P_A, la matrice C diventa come segue (bisogna aggiungere la richiesta alla prima riga, relativa alle risorse allocate al processo P_A):

$$C = \begin{bmatrix} 3 & 1 & 2 \\ 0 & 2 & 4 \\ 1 & 5 & 2 \end{bmatrix}$$

A questo punto la matrice delle richieste $R = Max - C$ risulta essere la seguente:

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

Non esistendo quindi nessuna riga di R minore od uguale al nuovo valore di $A = (2, 0, 0)$ (al vecchio valore di A bisogna sottrarre le risorse allocate a P_A in base alla sua richiesta), lo stato non risulta sicuro. Di conseguenza la richiesta di risorse da parte di P_A deve essere rifiutata dal sistema.

3. 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.

4. 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:

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

5. 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?

   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 $p = 5$.

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

   Soluzione: 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).

7. 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.

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. 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;
```

12. 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. L'implementazione della primitiva `signal` in questo caso deve essere tale da risvegliare tutti i thread/processi in attesa sulla condition variable `bridgeFree`. In questo modo tutti i thread che erano in attesa possono ricontrollare (uno per volta) la condizione a guardia del `while` ed accedere al ponte o tornare a sospendersi in attesa di un'altra segnalazione

dell'evento `bridgeFree`. Quindi tutte le auto che erano in coda e che hanno il turno a loro favorevole possono passare.

13. Si supponga che in una struttura vi sia un solo bagno e che sia necessario disciplinarne l'accesso tenendo presente quanto segue:

    - non deve mai capitare che nel bagno siano presenti contemporaneamente uomini e donne;
    - un uomo (risp. una donna) prima di entrare deve controllare che nel bagno non siano presenti delle donne (risp. degli uomini): se la condizione non è verificata deve attendere.

    Si completi la specifica del seguente monitor in modo da soddisfare i requisiti menzionati.

```
monitor Bathroom
    ...
    integer nmen, nwomen;

    procedure manEnter();
        begin
          while(nwomen>0) do ...
          ...
        end;

    procedure manExit();
        begin
          nmen := nmen-1;
          if(...) then ...
        end;

    procedure womanEnter();
        begin
          ...
          ...
        end;

    procedure womanExit();
        begin
          ...
          ...
        end;

    nmen := 0;
    nwomen := 0;
end monitor;
```

Soluzione:

```
monitor Bathroom
    condition bathroomFree
```

```
    integer nmen, nwomen;

    procedure manEnter();
        begin
            while(nwomen>0) do wait(bathroomFree);
            nmen := nmen + 1;
        end;

    procedure manExit();
        begin
            nmen := nmen-1;
            if(nmen=0) then signal(bathroomFree);
        end;

    procedure womanEnter();
        begin
            while(nmen>0) do wait(bathroomFree);
            nwomen := nwomen + 1;
        end;

    procedure womanExit();
        begin
            nwomen := nwomen-1;
            if(nwomen=0) then signal(bathroomFree);
        end;

    nmen := 0;
    nwomen := 0;
end monitor;
```

14. Ipotizzando che l'istruzione `signal()` possa comparire solo per ultima nelle procedure dei monitor, semplificare la seguente implementazione dei monitor tramite semafori:

```
down(&mutex);
...
procedura del monitor
...

if(prossimo_contatore > 0)
  up(&prossimo);
else
  up(&mutex);
```

dove `mutex` è il semaforo binario associato al monitor e `prossimo` è un semaforo su cui i processi che eseguono una `signal` possono autosospendersi, nell'attesa che il processo risvegliato si metta nuovamente in attesa o lasci il monitor.

Data una condition variable `x`, associamo ad essa una variabile contatore `x_contatore` che rappresenta il numero di processi/thread sospesi su di essa ed un semaforo `x_sem`, con le seguenti implementazioni di `wait` e `signal`:

```
wait(x):                          signal(x):

x_contatore++;                    if(x_contatore > 0) {
                                    prossimo_contatore++;
if(prossimo_contatore > 0)          up(&x_sem);
  up(&prossimo);                    down(&prossimo);
else                                prossimo_contatore--;
  up(&mutex);                     }

down(&x_sem);
x_contatore--;
```

Soluzione: sotto le ipotesi summenzionate, l'implementazione del monitor tramite semafori diventa la seguente:

```
down(&mutex);
...
procedura del monitor
...
up(&mutex);
```

con le seguenti implementazioni di `wait` e `signal`:

```
wait(x):                          signal(x):

x_contatore++;                    if(x_contatore > 0)
up(&mutex);                         up(&x_sem);
down(&x_sem);
down(&mutex);
x_contatore--;
```

Infatti, se la `signal` viene sempre eseguita come ultima istruzione, non c'è bisogno di sospendere il processo che la esegue sul semaforo `prossimo` (che diventa così inutile), dato che il blocco sul monitor verrà rilasciato subito dopo. Si noti invece che nella nuova implementazione della `wait` è necessario riacquisire il blocco sul monitor (`down(&mutex)`) dopo il "risveglio" in seguito alla `down(&x_sem)`.

15. 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.

16. Si consideri l'algoritmo di scheduling nel sistema Unix tradizionale. Supponendo che un quanto = 5 tic = 100 msec e che all'istante 0 si trovino in coda ready 3 processi nuovi, nell'ordine A, B, C, tutti con priorità uguale a 0 e CPU burst di 150, 100 e 200 tic rispettivamente, simulare l'esecuzione dei 3 processi nell'intervallo di tempo 0–2 secondi.

    Soluzione: la simulazione dell'esecuzione è data dalla seguente tabella (si ricordi che l'algoritmo di scheduling nel sistema Unix tradizionale ricalcola i tempi di CPU e le priorità ogni secondo secondo le equazioni CPU_j=$\text{CPU}_j/2$ e Pr_j=CPU_j+nice_j, dove nice_j in questo caso vale 0):

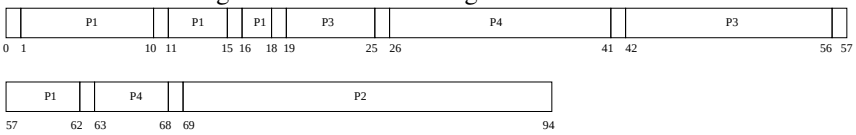
	Pr_A	CPU_A	Pr_B	CPU_B	Pr_C	CPU_C
0 sec	0	0	0	0	0	0
	0	5	0	0	0	0
	0	5	0	5	0	0
	0	5	0	5	0	5
	0	10	0	5	0	5
	0	10	0	10	0	5
	0	10	0	10	0	10
	0	15	0	10	0	10
	0	15	0	15	0	10
	0	15	0	15	0	15
	0	20	0	15	0	15
1 sec	10	10	7	7	7	7
	10	10	7	12	7	7
	10	10	7	12	7	12
	10	10	7	17	7	12
	10	10	7	17	7	17
	10	10	7	22	7	17
	10	10	7	22	7	22
	10	10	7	27	7	22
	10	10	7	27	7	27
	10	10	7	32	7	27
	10	10	7	32	7	32
2 sec	5	5	16	16	16	16

17. Si consideri un sistema con scheduling a priorità con tre code, A, B, C, di priorità decrescente, con prelazione tra code. Le code A e B sono round robin con quanto di 15 e 20 ms, rispettivamente; la coda C è FCFS. Se un processo nella coda A o B consuma il suo quanto di tempo, viene spostato in fondo alla coda B o C, rispettivamente. Si supponga che i processi P_1, P_2, P_3, P_4 arrivino rispettivamente nelle code A, C, B, A, con CPU burst e tempi indicati nella tabella seguente (si assuma pari a 1 ms il tempo di latenza del kernel):

	arrivo	burst
P_1	0	20ms
P_2	10	25ms
P_3	15	20ms
P_4	25	20ms

Si determini il diagramma di Gantt relativo all'esecuzione dei quattro processi ed i tempi medi di attesa e di reazione.

Soluzione: il diagramma di Gantt è il seguente:



Il tempo medio di attesa è $\frac{42+59+21+23}{4} = 36,25$, mentre quello medio di reazione è $\frac{1+59+4+1}{4} = 16,25$.

18. In coda ready arrivano i processi P_1, P_2, P_3, P_4, con CPU burst e istanti di arrivo specificati in tabella:

	arrivo	burst
P_1	0	15ms
P_2	5	10ms
P_3	15	15ms
P_4	20	5ms

(a) Se i processi terminano nell'ordine P_2, P_1, P_4, P_3, quale può essere l'algoritmo di scheduling? (Si trascuri il tempo di latenza del kernel e, nel caso di processi che arrivano in coda ready nello stesso istante, si dia priorità maggiore al processo con indice inferiore.)

i. RR q=10ms
ii. RR q=5ms
iii. Scheduling non-preemptive
iv. SJF
v. Nessuno dei precedenti

(b) (*) Si consideri ora tempo di latenza pari a 1ms e un algoritmo di scheduling SRTF. Si determini per i processi P_1, P_2, P_3, P_4 della tabella sopra:

i) il diagramma di GANTT relativo all'esecuzione dei quattro processi;

ii) il tempo di attesa medio;

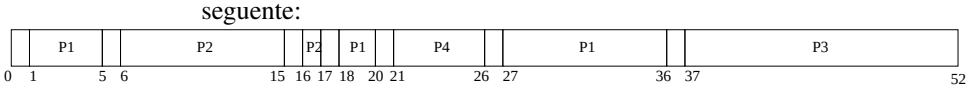
iii) il tempo di turnaround medio.

Soluzione:

(a) i).

(b) Considerando un tempo di latenza pari a 1ms e un algoritmo di scheduling SRTF, abbiamo quanto segue:

i) il diagramma di GANTT relativo all'esecuzione dei quattro processi è il seguente:



ii) il tempo di attesa medio è $\frac{21+2+22+1}{4} = \frac{46}{4} = 11,5ms$,

iii) il tempo di turnaround medio è: $\frac{36+12+37+6}{4} = \frac{91}{4} = 22,75ms$.