Esercitazioni di Sistemi Operativi

5.3.2002

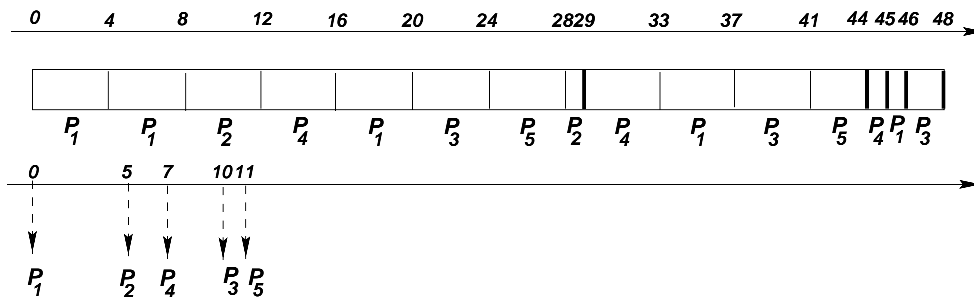
- **Esercizio 1**

Si consideri un insieme di cinque processi P_1, P_2, P_3, P_4, P_5 con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

processo	*tempo di arrivo*	*tempo di esecuzione*
P_1	0	17
P_2	5	5
P_3	10	10
P_4	7	9
P_5	11	7

Assegnare l'insieme di processi ad un processore in base alla politica Round Robin considerando un quanto di tempo di 4 millisecondi. Calcolare il valor medio del tempo di attesa ed il valor medio del tempo di turnaround dei processi.

Soluzione



$$t_a = \frac{29 + 19 + 28 + 29 + 26}{5} = 26.2 \text{ msec}$$

$$t_{tr} = \frac{46 + 24 + 38 + 38 + 33}{5} = 35.8 \text{ msec}$$

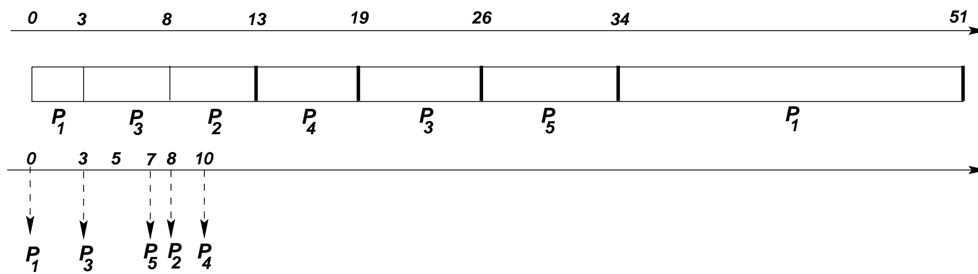
- **Esercizio 2**

Si consideri un insieme di cinque processi P_1, P_2, P_3, P_4, P_5 con i seguenti tempi di arrivo e tempi di esecuzione in millisecondi:

processo	*tempo di arrivo*	*tempo di esecuzione*
P_1	0	20
P_2	8	5
P_3	3	12
P_4	10	6
P_5	7	8

Assegnare l'insieme di processi ad un processore in base alla politica Shortest Job First preemptive e calcolare il valor medio del tempo di attesa e di turnaround.

Soluzione



$$t_a = \frac{31 + 0 + 11 + 3 + 19}{5} = 12.8 \text{ msec}$$

$$t_{tr} = \frac{51 + 5 + 23 + 9 + 27}{5} = 23 \text{ msec}$$

- **Esercizio 3**
Si vuole modellare un sistema composto da 1 macchinetta distributrice di aranciata, 1 fornitore di aranciata e N=20 consumatori di aranciata.
Requisiti:

1. La macchinetta fornisce due operazioni:
   - caricamento, che permette al fornitore di rifornire la macchinetta con M=10 aranciate;
   - prelievo, che permette ad un consumatore di prelevare una aranciata.
2. Il fornitore inizialmente carica la macchinetta, poi si sospende in attesa di essere chiamato a rifornire nuovamente la macchinetta vuota.
3. Ogni consumatore ha accesso alla macchinetta per prelevare aranciate.
4. Il primo consumatore che trova la macchinetta scarica deve chiamare il fornitore e attendere una nuova disponibilità di aranciata. Finché la macchinetta è vuota, i consumatori si devono sospendere in attesa di aranciata.

Scrivere le procedure *carica* e *preleva* che simulano la situazione utilizzando il costrutto semaforico.

Soluzione

```
type A = array[1..10] of orangine;
var pieno: semaphore(initial 1);
    vuoto: semaphore(initial 0);
    mutex: semaphore(initial 1);
    index: 1..10;
    aranciate: A;

procedure carica(x:A)
begin
    wait(pieno);
    aranciate := x;
    index := 10;
    signal(vuoto);
end;
```

```
procedure preleva_i(var y: orangine)
begin
    wait(mutex);
    wait(vuoto);
    y := aranciate[index];
    index := index − 1;
    if index = 0
        then signal(pieno)
        else signal(vuoto);
    signal(mutex);
end;
```

- **Esercizio 4**

  Utilizzare le primitive **wait** e **signal** definite sui semafori per realizzare un semplice programma concorrente per risolvere il problema dei lettori/scrittori che possa causare l'attesa indefinita (starvation) ma non lo stallo (deadlock) dei processi componenti.

  Soluzione

  Nel problema dei processi lettori/scrittori può avvenire il fenomeno di attesa indefinita; infatti, se mentre è presente un processo lettore nella sezione critica, continuano ad arrivare altri lettori, tutti i processi scrittori dovranno aspettare (attesa indefinita) l'uscita dell'ultimo lettore dalla sua sezione critica, rilasciando così la mutua esclusione sul semaforo db.

```
semaphore mutex=1, db=1;
int rc=0;

void Lettore()
  {
    wait(mutex);
    rc++;
    if (rc == 1) wait(db);
    signal(mutex);
    LetturaDB();
    wait(mutex);
    rc−−;
    if (rc == 0) signal(db);
    signal(mutex);
  }

void Scrittore()
  {
    wait(db);
    ScritturaDB();
    signal(db);
  }
```

- **Esercizio 5**

  Lungo una strada a due corsie e doppio senso di marcia, in direzione nord–sud, vi è un ponte ad una sola corsia, percorribile a senso unico alternato. Il ponte ha la capacità di N automobili. Le A_i dirette verso nord (sud) possono attraversare il ponte solo se non è attraversato da nessuna A_j diretta verso sud (nord). Scrivere il programma relativo ai processi "A_i *diretta a nord*" e "A_j *diretta a sud*" utilizzando i semafori.

Soluzione
Per risolvere il problema occorre:

1. implementare due liste di attesa, una per le automobili provenineti da nord e l'altra per le automobili provenienti da sud, ed implementarle sfruttando i semafori privati (ciascuna automobile deve essere dotata di un semaforo privato);
2. realizzare la mutua esclusione per l'accesso alle variabili che governano il ponte e le liste di attesa.

Le modalità relizzative sono le seguenti:

– Ciascuna nuova macchina che arriva in prossimità del ponte deve controllare tre condizioni:
  1. se ci sono macchine che stanno attraversando il ponte nell'altro senso;
  2. se ci sono N automobili provenienti dallo stesso verso che stanno già attraversando il ponte;
  3. se ci sono automobili che stanno attraversando il ponte nello stesso senso e ci sono altre automobili che sono in attesa dall'altra parte del ponte.

  In tutti e tre i casi l'automobile deve sostare ad un semaforo privato.
– Quando un'automobile A_i sta attraversando il ponte, all'uscita da questo deve controllare se è l'ultima: in tal caso, se ci sono automobili in attesa in senso opposto, la prima di queste dovrà iniziare l'attraversamento (per evitare l'attesa indefinita); altrimenti il ponte verrà attraversato da un'altra macchina proveniente dallo stesso senso di marcia. Viceversa, se A_i non è l'ultima automobile ad uscire dal ponte, si deve controllare che non ci siano automobili in attesa alla fine del ponte, nel qual caso si devono risvegliare k=N−r automobili provenienti dallo stesso senso (essendo N la portata del ponte, ed r il numero di auto rimanenti sul ponte una volta che A_i ne è uscita).

```
#define boolean int
#define false 0
#define true !false

boolean SulPonteDaSud = false;
int AutomobiliSulPonte = 0;
/****************** DA NORD ******************/
void AccessoAutomobiliProvenientiDaNord()
  {
    int i;
    i = DeterminaProprioIndice();
    wait(mutex); /* Ottiene la mutua esclusione */
    if (SulPonteDaSud || !Vuota(ListaAttesaDaSud) || AutomobiliSulPonte == N)
    {
        Push(ListaAttesaDaNord,i); /* L'automobile va messa in lista d'attesa */
    }
    else
    {
        SulPonteDaNord = true;
        AutomobiliSulPonte++;
        signal(V(i)); /* NON vogliamo che l'automobile si addormenti ... zzz */
    }
    signal(mutex); /* Rilascia la mutua esclusione */
    wait(V(i)); /* Addormenta l'automobile */
  }
```

```
void UscitaAutomobiliProvenientiDaNord()
{
    int j;
    wait(mutex); /* Ottiene la mutua esclusione */
    AutomobiliSulPonte−−;
    if (AutomobiliSulPonte == 0) /* Se è l'ultima automobile... */
    {
        /* ...allora si deve controllare che ce non ne siano dall'altra parte */
        if (Vuota(ListaAttesaDaSud))
        {
            /* Sveglia al più N automobili in attesa da nord */
            while (!Vuota(ListaAttesaDaNord) && AutomobiliSulPonte < N)
            {
                AutomobiliSulPonte++;
                j = Pop(ListaAttesaDaNord);
                signal(V(j));
            }
        }
        else
        {
            /* Sveglia al più N automobili in attesa da sud */
            SulPonteDaNord = false;
            SulPonteDaSud = true;
            while (!Vuota(ListaAttesaDaSud) && AutomobiliSulPonte < N)
            {
                AutomobiliSulPonte++;
                j = Pop(ListaAttesaDaSud);
                signal(V(j));
            }
        }
    }
    else
    {
        if (Vuota(ListaAttesaDaSud) && !Vuota(ListaAttesaDaNord))
        {
            AutomobiliSulPonte++;
            j = Pop(ListaAttesaDaNord);
            signal(V(j));
        }
        else
        {
            SulPonteDaNord = false;
        }
    }
    signal(mutex); /* Rilascia la mutua esclusione */
}
```

```
/****************** DA SUD ******************/
void AccessoAutomobiliProvenientiDaSud()
  {
    int i;
    i = DeterminaProprioIndice();
    wait(mutex); /* Ottiene la mutua esclusione */
    if (SulPonteDaNord) || !Vuota(ListaAttesaDaNord) || AutomobiliSulPonte == N)
    {
      Push(ListaAttesaDaSud,i); /* L'automobile va messa in lista d'attesa */
    }
    else
    {
      SulPonteDaSud = true;
      AutomobiliSulPonte++;
      signal(V(i)); /* NON vogliamo che l'automobile si addormenti ... zzz */
    }
    signal(mutex); /* Rilascia la mutua esclusione */
    wait(V(i)); /* Addormenta l'automobile */
  }

void UscitaAutomobiliProvenientiDaSud()
  {
    int j;
    wait(mutex); /* Ottiene la mutua esclusione */
    AutomobiliSulPonte--;
    if (AutomobiliSulPonte == 0) /* Se è l'ultima automobile... */
    {
      /* ...allora si deve controllare che ce non ne siano dall'altra parte */
      if (Vuota(ListaAttesaDaNord))
      {
        /* sveglia al più N automobili in attesa da sud */
        while (!Vuota(ListaAttesaDaSud) && AutomobiliSulPonte < N)
        {
          AutomobiliSulPonte++;
          j = Pop(ListaAttesaDaSud);
          signal(V(j));
        }
      }
      else
      {
        /* Sveglia al più N automobili in attesa da nord */
        SulPonteDaSud = false;
        SulPonteDaNord = true;
        while (!Vuota(ListaAttesaDaNord) && AutomobiliSulPonte < N)
        {
          AutomobiliSulPonte++;
          j = Pop(ListaAttesaDaNord);
          signal(V(j));
        }
```

```
        }
    }
    else
    {
        if (Vuota(ListaAttesaDaNord) && !Vuota(ListaAttesaDaSud))
        {
            AutomobiliSulPonte++;
            j = Pop(ListaAttesaDaSud);
            signal(V(j));
        }
        else
        {
            SulPonteDaSud = false;
        }
    }
    signal(mutex); /* Rilascia la mutua esclusione */
}
```

- **Esercizio 6**

  Un processo mittente P ed N processi riceventi (R_1, ..., R_N) comunicano tramite un buffer circolare di capacità B messaggi. P deposita i messaggi nel buffer e gli R_i li leggono. Ciascun messaggio deve essere letto da tutti gli R_i e inoltre ciascun R_i deve leggere i messaggi nell'ordine in cui questi sono stati spediti. Tuttavia gli R_i possono leggere uno stesso messaggio in tempi diversi: in un dato istante R_k può aver letto fino a B messaggi più di R_j. Scrivere un monitor per implementare questo modello di comunicazione.

  Soluzione

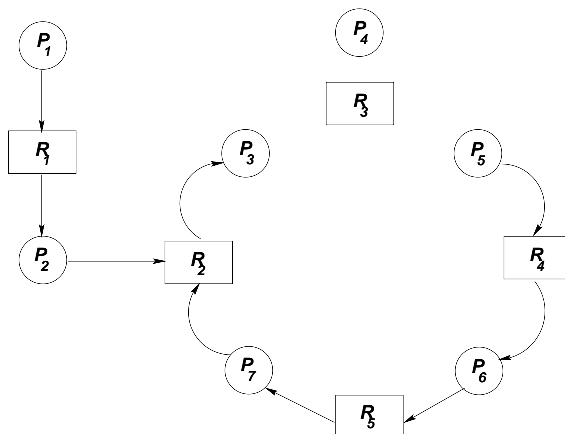
```
type slot = record
        dati: msg; quanti: 0..N;
        end;
        buffer = array[0..B − 1] of slot;
        atmu = monitor;
var buff: buffer;
      disp: 0..B;
      in: 0..B − 1;
      adest: 0..N;
      dove: array[1..N] of 0..B − 1;
      mitt, dest: condition;
procedure entry send (M: msg)
begin
    if disp = 0 then mitt.wait;
    disp := disp − 1;
    buff[in].dati := M;
    buff[in].quanti := 0;
    in := (in + 1) mod B;
    while adest > 0 do
        begin
            adest := adest − 1;
            dest.signal
        end;
end;
```

```
procedure entry receive (i: 1..N; var M: msg)
begin
    if (disp = B or dove[i] = in) then
        begin
            adest := adest + 1;
            dest.wait
        end;
    M := buff[dove[i]].dati;
    buff[dove[i]].quanti := buff[dove[i]].quanti + 1;
    if buff[dove[i]].quanti = N then
        begin
            disp := disp + 1;
            mitt.signal
        end;
    dove[i] := (dove[i] + 1) mod B;
end;

begin {inizializzazione}
    disp := B − 1;
    in := 0;
    for adest := 1 to N do dove[adest] := 0;
    adest := 0;
end;
```

- **Esercizio 7**

Si consideri il seguente grafo di allocazione risorse:



e sia Max tale che

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

Se la tecnica prescelta per gestire i deadlock consiste nell'*evitarli*, una richiesta della risorsa R_3 da parte di P_5 può essere soddisfatta? Motivare la risposta.

Soluzione
La richiesta di P_5 non può essere soddisfatta perché altrimenti lo stato del sistema diventa non sicuro. Infatti, la matrice Max suggerisce che anche il processo P_3 potrebbe richiedere R_3 e, in questo caso, si verificherebbe una attesa circolare (corrispondente ad un ciclo nel grafo di allocazione delle risorse). I processi coinvolti nel deadlock sarebbero P_3, P_5, P_6 e P_7 e le risorse R_2, R_3, R_4 ed R_5.

- **Esercizio 8**
Si consideri un sistema costituito da 5 processi, P_0, P_1, P_2, P_3, P_4, e dalle 4 classi di risorse di tipo A, B, C, D, descritto dalle seguenti matrici

$$Allocation = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 \\ 1 & 3 & 5 & 4 \\ 0 & 6 & 3 & 2 \\ 0 & 0 & 1 & 4 \end{bmatrix}, \quad Max = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 7 & 5 & 0 \\ 2 & 3 & 5 & 6 \\ 0 & 6 & 5 & 2 \\ 0 & 6 & 5 & 6 \end{bmatrix}, \quad Available = \begin{bmatrix} 1 & 5 & 2 & 0 \end{bmatrix}.$$

Utilizzando l'*algoritmo del Banchiere* si stabilisca se il sistema è in stato sicuro e, in caso affermativo, se lo stato si mantiene sicuro a fronte di una ulteriore richiesta $[0, 4, 2, 0]$, da parte del processo P_1.

Soluzione
In questo caso, la matrice $Need = Max - Allocation$ risulta:

$$Need = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix}.$$

Pertanto la successione $< P_0,\ P_2,\ P_1,\ P_3,\ P_4 >$ ($Available = [1,5,3,2] \rightarrow [2,8,8,6] \rightarrow [3,8,8,6] \rightarrow [3,14,11,8] \rightarrow [3,14,12,12]$) è una successione sicura e tale è anche lo stato descritto dalle matrici *Available*, Max e dal vettore *Available*.
Nel caso in cui venga accordata al processo P_1 l'ulteriore richiesta di risorse $[0, 4, 2, 0]$ le matrici *Allocation* e $Need$ divengono:

$$Allocation = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 4 & 2 & 0 \\ 1 & 3 & 5 & 4 \\ 0 & 6 & 3 & 2 \\ 0 & 0 & 1 & 4 \end{bmatrix}, \quad Need = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 3 & 0 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 6 & 4 & 2 \end{bmatrix},$$

mentre il vettore delle risorse disponibili risulta $Available = [1, 1, 0, 0]$. Può quindi essere ancora una volta servito il processo P_0 che restituisce le proprie risorse, fornendo $Available = [1, 1, 1, 2]$. Di nuovo, può essere servito il processo P_2, che produce $Available = [2, 4, 6, 6]$. In seguito, vengono serviti i processi P_1 ($Available = [3, 8, 8, 6]$), P_3 ($Available = [3, 14, 11, 8]$) e P_4 ($Available = [3, 14, 12, 12]$). Pertanto, la stessa sequenza $< P_0,\ P_2,\ P_1,\ P_3,\ P_4 >$ è sicura e tale è anche lo stato del sistema.

- **Esercizio 9**

  Si supponga che la lista delle partizioni libere di memoria, tenuta in ordine di indirizzo, consista di cinque nodi N_1, N_2, N_3, N_4, N_5 le cui dimensioni e gli indirizzi sono riportati nella tabella seguente:

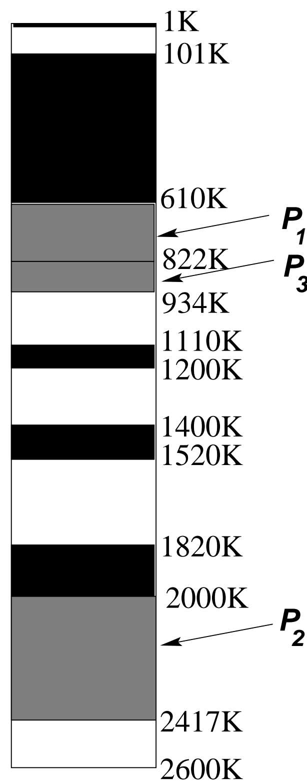
Nodi	*Dimensioni*	*Indirizzo*
N_1	100K	1K
N_2	500K	610K
N_3	200K	1200K
N_4	300K	1520K
N_5	600K	2000K

  Dovendo inserire P_1, P_2, P_3, P_4 rispettivamente di 212K, 417K, 112K e 426K (in questa successione), come e dove saranno memorizzati usando **First–fit** (facendo partire la ricerca dalla partizione successiva a quella cui si era arrivati, quand'anche fosse rimasto nella stessa un buco di dimensione sufficiente, ed iniziando dal nodo N_1) e **Best–fit**.

  Soluzione

  Con l'algoritmo **First–Fit** avremo la situazione seguente:

Processi	*Indirizzi*
P_1	610K
P_2	2000K
P_3	822K
P_4	non può essere allocato

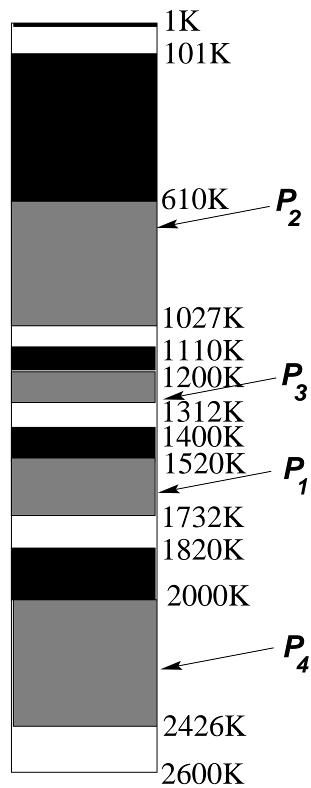


con la seguente lista dei blocchi liberi

Nodi	*Dimensioni*	*Indirizzo*
N_1	100K	1K
N_2	176K	934K
N_3	200K	1200K
N_4	300K	1520K
N_5	183K	2417K

Con **Best–Fit** la situazione sarà invece la seguente:

Processi	*Indirizzi*
P_1	1520K
P_2	610K
P_3	1200K
P_4	2000K



con relativa lista libera:

Nodi	*Dimensioni*	*Indirizzo*
N_1	100K	1K
N_2	83K	1027K
N_3	88K	1312K
N_4	88K	1732K
N_5	174K	2426K

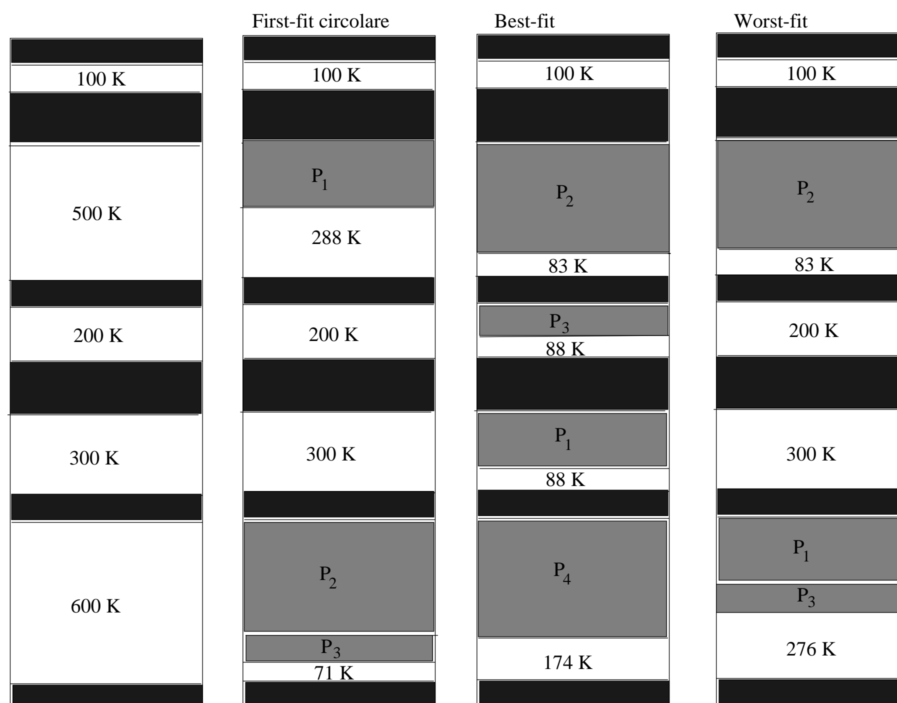
- **Esercizio 10**

a) Si consideri una memoria centrale gestita con il metodo delle partizioni multiple; sono presenti le aree libere di dimensione 100K, 500K, 200K, 300K, 600K (elencate in ordine di indirizzo crescente). Si consideri una sequenza di quattro processi che necessitano, rispettivamente, 212K, 417K, 112K, 426K, e si descriva come vengono allocati in memoria dagli algoritmi **First–fit**, **Best–fit** ed **Worst–fit**. Si confrontino le situazioni risultanti.

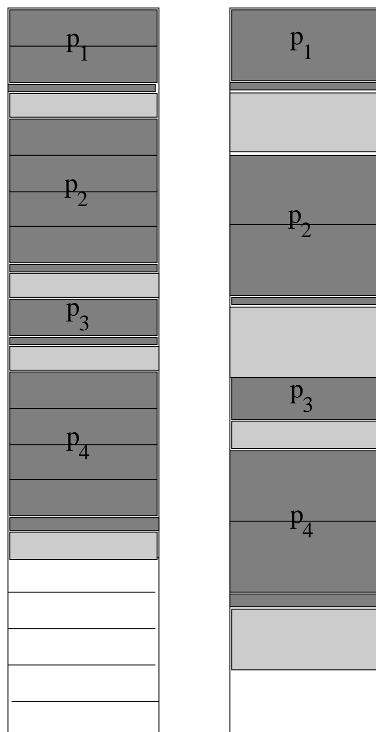
b) Si supponga ora di gestire una memoria di 2000K per mezzo di paginazione, si considerino dimensioni di pagine di 100K e 200K. Si descriva l'allocazione di memoria conseguente, a fronte della sequenza di processi sopra indicata.

Soluzione

a) Sia nel caso di **First–fit** circolare che nel caso di **Worst–fit**, si riescono ad allocare solo tre processi su quattro (vedi figura), a causa del fenomeno della *frammentazione esterna* (959K per **First–fit**, 533K per **Best–fit** e 959K per **Worst–fit**). Pertanto, in entrambi i casi, o si procede ad un compattamento della memoria libera o si deve attendere che qualche processo termini (ad esempio, P_1, nel primo caso, e P_2 nel secondo).



b) Nel caso dell'utilizzo della tecnica di paginazione, con pagine di dimesione 100K e 200K rispettivamente, si ha la situazione riportata in figura:



Si nota che la *frammentazione interna* è maggiore nel secondo caso (633K) rispetto al primo (333K). La memoria libera residua è quindi costituita da 5 frame (500K) nel primo caso e da un solo frame (200K) nel secondo.

- **Esercizio 11**

Si consideri la seguente tabella dei segmenti:

Segmento	*Base*	*Lunghezza*
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

e si calcolino gli indirizzi fisici corrispondenti agli indirizzi logici $a = (0, 430)$, $b = (1, 10)$, $c = (2, 500)$, $d = (3, 400)$ ed $e = (4, 112)$.

Soluzione

Poiché il primo numero di ciascuna coppia identifica il segmento, mentre il secondo indica l'offset all'interno dello stesso, gli indirizzi fisici relativi alle coppie sopra descritte sono:

$$a = 219 + 430 = 649,$$
$$b = 2300 + 10 = 2310,$$
$$c = \text{indefinito } 500 > 100,$$
$$d = 1327 + 400 = 1727,$$
$$e = \text{indefinito } 112 > 96.$$