

Esercizio 1

Si consideri un sistema con due risorse di tipo R1 e una risorsa di tipo R2. Sono presenti tre processi: P1, P2 e P3. Il processo P3 ha già assegnata una risorsa di tipo R1 e il processo P1 una di tipo R2. Le frecce tratteggiate indicano le potenziali richieste future dei processi.

1. Completare con le opportune frecce il grafo di assegnazione dopo ogni richiesta, assumendo che il sistema utilizzi l'algorithm del banchiere. Spiegare nel riquadro corrispondente.

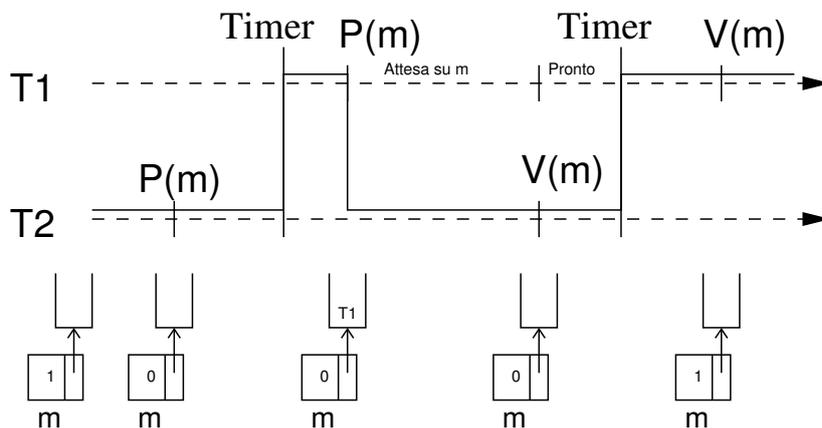
Grafo iniziale	P3 chiede R1	P2 chiede R1	P1 chiede R1	P1 rilascia le risorse e termina
	Spiegazione: Simula l'assegnamento: restano 0 risorse, non c'è una sequenza di terminazione. Lo stato è insicuro. P3 attende.	Spiegazione: Simula l'assegnamento: restano 0 risorse, non c'è una sequenza di terminazione. Lo stato è insicuro. P2 attende.	Spiegazione: Simula l'assegnamento: Sequenza di terminazione <P1,P3, P2>. Lo stato è sicuro. La risorsa è assegnata.	Spiegazione: Simula l'assegnamento a P3 (primo in coda): Seq. di terminazione <P3,P2>. Lo stato è sicuro. La risorsa è assegnata.

2. Si verifica una situazione di deadlock? Perché?

Non si verifica una situazione di deadlock, in quanto l'algorithm del Banchiere è una tecnica di controllo a run-time che impedisce sempre la formazione di stalli: ogni assegnamento viene simulato ed effettuato solo se lo stato risultante non porta a uno stallo nemmeno quando tutti i processi chiedono le risorse massime dichiarate. Viene infatti verificata l'esistenza di una sequenza di terminazione.

Esercizio 2

I thread $T1$ e $T2$ sono in esecuzione su un singolo processore in *time-sharing* e utilizzano un semaforo binario m con coda FIFO, inizializzato ad 1, per proteggere una sezione critica. Indicare, nello schema sottostante, il valore e la coda del semaforo m , in corrispondenza delle varie operazioni P e V . Indicare, inoltre, il thread in esecuzione tramite una linea continua; lasciare invece la linea tratteggiata nel caso il thread sia pronto o in attesa (in quest'ultimo caso scrivere che il thread è in attesa). Il thread inizialmente in esecuzione è $T2$; in corrispondenza dell'interrupt del timer il sistema effettuerà un cambio di contesto in favore dell'altro thread, se possibile.



Spiegazione:
 La prima $P(m)$, effettuata da $T2$, rende il semaforo 'rosso'. Il timer passa l'esecuzione a $T1$. Quando $T1$ esegue $P(m)$ viene accodato sul semaforo e l'esecuzione torna a $T2$. Al momento della $V(m)$ $T1$ torna ad essere pronto e al timer successivo ottiene nuovamente l'uso della CPU. L'ultima $V(m)$ fa tornare il semaforo 'verde'.

Esercizio 3

Considerare N produttori e M consumatori che eseguono il seguente codice:

```

Produttore {
  while(true) {
    <produci d>
    buffer.produci(d);
  }
}

Consumatore {
  while(true) {
    d = buffer.consuma();
    <consuma d>
  }
}

```

1. Scrivere il codice di un monitor `buffer`, opportunamente commentato, che implementi la funzioni `produci(d)` e `consuma()` per ricevere, rispettivamente, un dato `d` da un Produttore e per consegnare un dato a un Consumatore. Il monitor può memorizzare al più `MAX` dati. Implementare opportunamente la sincronizzazione in caso di buffer pieno o buffer vuoto.

```

// soluzione basata su *signal*

Monitor buffer {
  t_dati dati[MAX]; // array per memorizzare i dati
  n_dati=0; // numero dati nel buffer
  int i_v=0; // primo elemento vuoto
  int i_p=0; // primo elemento pieno
  condition pieno,vuoto; // variabili condition per la sincronizzazione

  produci(t_dati d) {
    if (n_dati == MAX)
      pieno.wait; // se e' pieno attende
    dati[i_v]=d; // memorizza d
    i_v = (i_v + 1)%MAX; // incrementa l'indice
    n_dati = n_dati+1; // incrementa il numero di dati
    vuoto.signal; // sveglia il primo consumatore in attesa
    // (n_dati e' sicuramente > 0)
  }

  t_dati consuma() {
    if (n_dati == 0)
      vuoto.wait; // se e' vuoto attende
    d=dati[i_p]; // legge il primo elemento pieno
    i_p = (i_p + 1)%MAX; // incrementa l'indice
    n_dati = n_dati-1 ; // decrementa il numero di dati
    pieno.signal; // sveglia il primo produttore in attesa
    // (n_dati e' sicuramente < MAX)
    return d;
  }
}

```

2. Discutere la possibilità di starvation (attesa indefinita) nella soluzione proposta, indicando eventuali soluzioni a tale problema:

Avendo utilizzato `signal` non c'è possibilità di starvation: la `signal` dà subito l'esecuzione al thread sbloccato sospendendo nella coda urgente il process che la esegue. Assumendo che le code sulle `condition` siano gestite con politica FIFO, non può quindi esserci starvation.

NOTA: Differente sarebbe utilizzando la `notify` che non garantisce l'esecuzione immediata del thread risvegliato: per questa ragione è bene racchiudere la `wait` in un ciclo `while` che ri-verifica la condizione di bloccaggio. Se però un altro thread entra nel monitor e 'ruba' la cella piena o vuota al thread appena sbloccato, si può verificare starvation.