- Che cos'è un sistema time-sharing? Si può avere time-sharing in un sistema con singola CPU?

  Risposta: Un sistema time-sharing è una variante della multiprogrammazione in cui il/i processore/i esegue più processi commutando le loro esecuzioni con una frequenza sufficientemente elevata da permettere a ciascun utente di interagire con il proprio programma durante la sua esecuzione. Il time-sharing si può avere anche in un sistema con una singola CPU.

- Che cos'è lo spooling?

  Risposta: Lo spooling (da Simultaneous Peripheral Operation On Line) è una tecnica per gestire autonomamente ed in modo efficiente molte richieste concorrenti a dispositivi di I/O che non possono accettare per loro natura flussi di dati intercalati. Ad esempio le applicazioni che vogliono stampare riversano i loro dati in un file che viene memorizzato in una directory di spool; più tardi un demone od un thread del kernel apposito provvederanno a leggere il contenuto dei file della directory e ad inviarlo in modo ordinato alla stampante.

- Quali sono gli svantaggi di un sistema operativo basato su thread rispetto ad uno basato su processi?

  Risposta: Gli svantaggi di un sistema operativo basato su thread rispetto ad uno basato su processi si riassumono sostanzialmente in una maggiore complessità di progettazione e programmazione. Infatti i processi devono essere "pensati" ed organizzati in attività concorrenti, c'è un minor *information hiding* in quanto lo spazio di memoria fra i thread di uno stesso processo è condiviso. Inoltre bisogna porre estrema cura nella sincronizzazione dei thread e nel loro scheduling (che spesso è demandato all'utente). I thread infine risultano particolarmente inadatti a situazioni in cui è necessario proteggere i dati.

- Si descriva l'algoritmo di scheduling del sistema Linux.

  Risposta: Innanzitutto lo scheduling di Linux è basato sui thread e non sui processi. A tal proposito le classi considerate sono tre:

  1. real-time FIFO,
  2. real-time round robin,
  3. timesharing.

  I thread real-time FIFO hanno la massima priorità e non sono soggetti a prerilascio se non da parte di un thread real-time FIFO inserito nella coda ready. I thread real-time round robin invece possono essere prelazionati dall'orologio del sistema. Ogni thread ha una priorità di scheduling di default pari a 20. Tramite la system call `nice` si può alterare tale valore a 20 - *valore*, dove *valore* è un intero compreso fra -20 e 19. La priorità quindi può variare da 1 a 40 con l'idea che a valori maggiori corrisponda un servizio migliore (tempi di risposta più veloci e una maggior percentuale del tempo di CPU). Inoltre ogni thread ha associato un *quantum* (numero di tick dell'orologio per cui un thread può continuare l'esecuzione). L'algoritmo di scheduling in base a queste grandezze calcola la *goodness* di un thread come segue:

```
if (class == real_time) goodness = 1000 + priority;
if (class == timesharing && quantum > 0) goodness = quantum + priority;
if (class == timesharing && quantum == 0) goodness = 0;
```

Quando deve essere presa una decisione di scheduling, l'algoritmo seleziona il thread con il valore di goodness più alto. Man mano che il thread esegue il suo quantum viene decrementato di uno ad ogni tick dell'orologio. Un thread viene quindi prelazionato se si verifica una delle seguenti condizioni:

1. il quantum diventa zero,
2. il thread si sospende per un'operazione di I/O o un altro evento,
3. nella cosa ready entra un thread precedentemente bloccato con una goodness maggiore.

Quando i quanta di tutti i processi in coda ready raggiungono lo zero, lo scheduler ricalcola i loro valori per tutti i thread del sistema (ready e waiting) in accordo alla regola seguente:

```
quantum = quantum / 2 + priority;
```

In questo modo tutti i thread I/O bound (i.e., interattivi) vengono favoriti dato che di solito si sospendono prima di esaurire il proprio quantum.

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

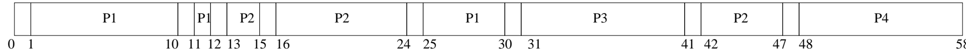
	arrivo	burst
P_1	0	15ms
P_2	10	15ms
P_3	15	10ms
P_4	30	10ms

1. Se i processi terminano nell'ordine P_1, P_3, P_2, P_4, quale può essere l'algoritmo di scheduling? (Si trascuri il tempo di latenza del kernel.)
   (a) RR q=20ms
   (b) RR q=10ms
   (c) Scheduling preemptive
   (d) SRTF
   (e) Nessuno dei precedenti
2. (*) Si consideri ora tempo di latenza pari a 1ms e un algoritmo di scheduling RR con q=10ms. 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 reazione medio;
   iii) il tempo di turnaround medio.

Risposta:

1. b), c), d).
2. Considerando un tempo di latenza pari a 1ms e un algoritmo di scheduling RR con q=10ms, abbiamo quanto segue:

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



   ii) il tempo di reazione medio è $\frac{1+(13-10)+(31-15)+(48-30)}{4} = \frac{1+3+16+18}{4} = 38/4 = 9,5ms$;

   iii) il tempo di turnaround medio è $\frac{30+(47-10)+(41-15)+(58-30)}{4} = \frac{30+37+26+28}{4} = 121/4 = 30,25ms$.

- Si dica se la seguente affermazione è vera o falsa, motivando la risposta:

  > Se le quattro condizioni di Coffman sono verificate, allora *sicuramente* nel sistema si verificherà un deadlock.

  Risposta: l'affermazione è falsa; infatti le quattro condizioni di Coffman sono condizioni *necessarie*, ma non sufficienti affinché si verifichi un deadlock. Quindi se un deadlock si verifica, allora sono verificate le condizioni di Coffman. Viceversa se le condizioni sono verificate, allora *può* verificarsi un deadlock (ma non è detto che ciò avvenga).

- Si illustrino brevemente le primitive necessarie per implementare un sistema di scambio di messaggi per la sincronizzazione di processi. Tale sistema può essere applicato anche a sistemi distribuiti? Motivare la risposta.

  Risposta: le due primitive necessarie per l'implementazione di un sistema di scambio di messaggi sono le seguenti:

  – `send(destination, &message);`
  – `receive(source, &message);`

  La prima invia il messaggio puntato da `&message` alla destinazione specificata, mentre la seconda permette a chi la esegue di ricevere un messaggio dal mittente specificato, memorizzandolo nel buffer puntato da `&message`. Nel caso in cui non sia disponibile nessun messaggio al momento della chiamata, il processo che ha eseguito la primitiva `receive` può bloccarsi (in attesa dell'arrivo di un messaggio), oppure ritornare con un opportuno codice d'errore. Questo sistema può essere applicato anche a sistemi distribuiti in quanto non richiede la presenza di una memoria condivisa.

- Si consideri la seguente situazione, dove P_0, P_1, P_2, P_3, P_4 sono cinque 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	D	A	B	C	D
P_0	0	2	0	2	0	3	1	2
P_1	0	0	0	0	2	7	5	0
P_2	2	3	5	4	2	3	7	6
P_3	0	4	3	2	0	4	5	2
P_4	0	0	1	5	0	6	5	5

Available (A)			
A	B	C	D
1	5	3	0

1. Calcolare la matrice R delle richieste.
2. Il sistema è in uno stato sicuro (safe)?
3. Nel caso arrivi la richiesta di allocazione $(1, 4, 2, 0)$ per il processo P_1, quest'ultima può essere soddisfatta, ovvero, il sistema rimane in uno stato sicuro?

Risposta:

1. La matrice R delle richieste è data dalla differenza $Max - C$:

R

A	B	C	D
0	1	1	0
2	7	5	0
0	0	2	2
0	0	2	0
0	6	4	0

2. Sì il sistema è in uno stato sicuro in quanto esiste la sequenza sicura $\langle P_0, P_2, P_1, P_3, P_4 \rangle$. Infatti, dapprima si esegue P_0 in quanto $R_0 \leq A$ ed A diventa quindi $(1, 7, 3, 2)$. A questo punto $R_2 \leq A$ e quindi si esegue P_2 generando il nuovo valore di A: $(3, 10, 8, 6)$. Quindi si può mandare in esecuzione P_1 dato che $R_1 \leq A$ lasciando inalterato A, dato che $C_1 = (0, 0, 0, 0)$. In seguito può essere eseguito P_3 ($R_3 \leq A$) aggiornando A al valore $(3, 14, 11, 8)$. Infine viene eseguito P_4 ($R_4 \leq A$) generando il valore finale di $A = (3, 14, 12, 13)$.
3. Nel caso venga soddisfatta la richiesta $(1, 4, 2, 0)$ per il processo P_1 i nuovi valori di C, R e A sono i seguenti:

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

Il sistema non si trova più in uno stato sicuro; infatti l'unica riga di R minore od uguale al valore di A è la prima, relativa al processo P_0. Quindi, mandando in esecuzione P_0, si ottiene il nuovo valore di $A = (0, 3, 1, 2)$. A questo punto non esiste nessuna riga della matrice R minore od uguale al valore di A; quindi il sistema è in uno stato di deadlock.

Il punteggio attribuito ai quesiti è il seguente: 3, 3, 3, 3, 3, 6, 3, 3, 2, 2, 2 (totale: 33).