Caselle riservate

Ex. 1	
Ex. 2	
Ex. 3	
Ex. 4	
Ex. 5	
Ex. 6	
Tot.	

Sistemi Operativi

Compito d'esame

26 Giugno 2013

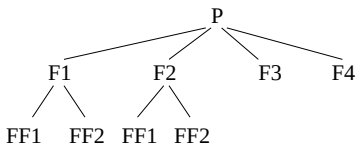
Matricola _____ Cognome _____ Nome _____

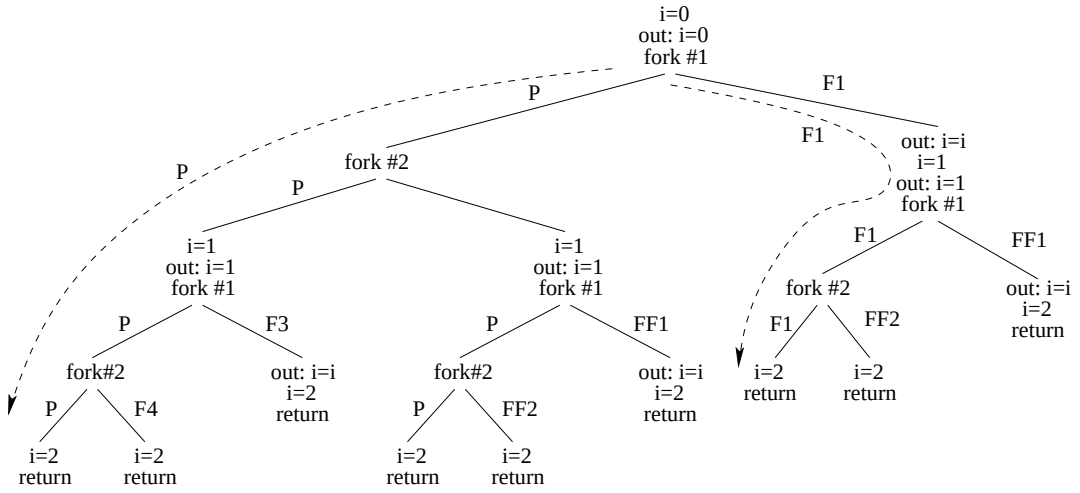
Docente: ◯ Laface ◯ Quer

Non si possono consultare testi, appunti o calcolatrici. Riportare i passaggi principali. L'ordine sarà oggetto di valutazione.
Durata della prova: 75 **minuti.**

1. Si riporti l'albero di generazione dei processi e si indichi che cosa produce su video il seguente programma e per quale motivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main () {
  int i;
  for (i=0; i<2 ; i++) {
    printf ("i=%d\n", i);
    if (fork())             /* call #1 */
      fork();               /* call #2 */
    else
      system ("echo i=i");  /* call #3 */
  }
}
```






```
Output:
  i=0
  i=1
  i=1
  i=i
  i=i
  i=i
  i=i
```

Schematica descrizione di fork e system ... L'ordine dell'output dipende dallo scheduler.

2. Si illustrino le caratteristiche delle *pipe* per la comunicazione e la sincronizzazione tra processi. Se ne illustri l'utilizzo con un esempio descritto utilizzando codice in linguaggio C.

Ogni processo che condivide dati con altri processi è un processo cooperante. Processi cooperanti necessitano di meccanismi che consentano la condivisione di dati. Ci sono due modelli di comunicazione.

- Modello a memoria condivisa
- Modello basato sul passaggio di messaggi

In quest'ultimo caso la comunicazione avviene mediante lo scambio di messaggi.

Una pipe permette di stabilire un flusso dati tra due processi. Ciascun processo, attraverso un file descriptor, accede a uno degli estremi della pipe. Può essere utilizzata per la comunicazione tra processi con un parente comune. Il flusso di dati half-duplex, i.e., i dati fluiscono solo in una direzione. Lettura e scrittura da e su pipe vengono effettuate mediante `read` e `write`. Esse son o bloccanti per pipe vouta o piena, rispettivamente.

Il seguente è un esempio di utilizzo:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main() {
  int n;
  int file[2];
  char cR = 'X';
  char cW;
  pid_t pid;
  if (pipe(file) == 0) {
    pid = fork ();
    if (pid == -1) {
      fprintf(stderr, "Fork failure");
      exit(EXIT_FAILURE);
    }
    if (pid == 0) {
      // child reads
      close (file[1]);
      n = read (file[0], &cR, 1);
      printf("Read %d bytes: %c\n", n, cR);
      exit(EXIT_SUCCESS);
    } else {
      // parent writes
      close (file[0]);
      n = write (file[1], &cW, 1);
      printf ("Wrote %d bytes: %c\n", n, cW);
    }
  }
  exit(EXIT_SUCCESS);
}
```

Descrizione del codice (pipe, clse, read e write) ...

3. Si illustri il significato e l'utilizzo dei "grafi di allocazione delle risorse" nel caso di risorse semplici e multiple. Se ne esemplifichi l'utilizzo in presenza e in assenza di deadlock (si richiedono almeno due esempi).

Una *condizione di stallo* (*deadlock*) si verifica quando un processo richiede una risorsa che non è disponibile ed entra in uno stato di attesa che non termina mai.

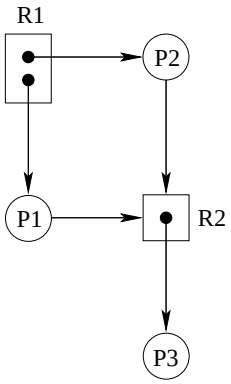
Il modello di gestione prevede quanto segue la creazione di un grafo $G = (V, E)$ in cui:

- I vertici V sono divisi tra
  - Risorse, a loro volta suddivisi in classi (tipi) $R = \{R_1, R_2, \ldots, R_m\}$. Ogni risorsa di tipo R_i ha W_i istanze. Tutte le istanze di una classe sono "indentiche", ovvero una qualsiasi istanza soddisfa una richiesta per quel tipo di risorsa.
  - Processi $P = \{P_1, P_2, \ldots, P_n\}$ che utilizzano le risorsa tramite un protocollo di richiesta, utilizzo e rilascio.
- Gli archi E sono divisi in
  - Archi di richiesta $P_i \to R_j$, i.e., da processo a risorsa
  - Archi di assegnazione $R_j \to P_i$, i.e., da risorsa a processo

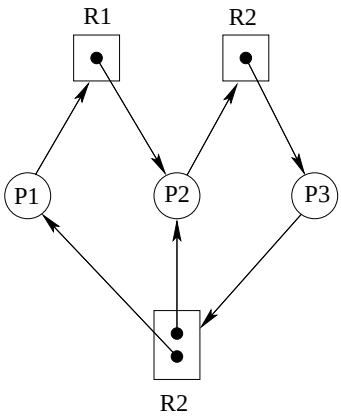
Dato un grafo di assegnazione è possibile verificare la presenza di uno stallo:

- Se il grafo non contiene cicli allora non c'è deadlock.
- Se il grafo contiene uno o più cicli allora
  - Se esiste solo un'istanza per tipo di risorsa, c'è sicuramente deadlock.
  - Se esiste più di un'istanza per tipo di risorsa, c'è la possibilità di deadlock.

La figura successiva ...

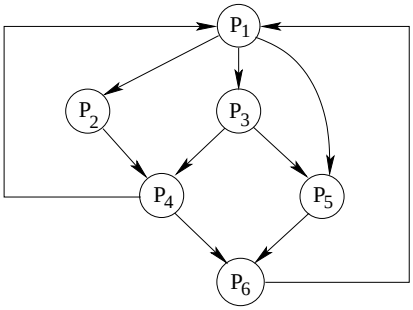


No cicli; no stallo



Due cicli; stallo

4. Dato il seguente grafo di precedenza, realizzarlo utilizzando il **minimo** numero possibile di semafori. I processi rappresentati devono essere processi ciclici (con corpo del tipo `while(1)`). Si utilizzino le primitive `init`, `signal` e `wait`. Riportare il corpo dei processi $(P_1, \ldots, P_6)$ e l'inizializzazione dei semafori.



Gli archi P_1–P_3 e P_4–P_1 sono superflui.

```
Inizializzazione:
sem_t s1, s2, s3, s4, s5, s6;

init (s1, 1);
init (s2, 0);
...
init (s6, 0);
```

```
Terminazione:
destroy (s1);
...
destroy (s6);
```

```
P1
while (1) {
  wait (s1);
  printf ("P1\n");
  signal (s2);
  signal (s3);
}
```

```
P2
while (1) {
  wait (s2);
  printf ("P2\n");
  signal (s4);
}
```

```
P3
while (1) {
  wait (s3);
  printf ("P3\n");
  signal (s4);
  signal (s5);
}
```

```
P4
while (1) {
  wait (s4);
  wait (s4);
  printf ("P4\n");
  signal (s6);
}
```

```
P5
while (1) {
  wait (s5;)
  printf ("P5\n");
  signal (s6);
}
```

```
P6
while (1) {
  wait (s6);
  wait (s6);
  printf ("P6\n");
  signal (s1);
}
```

5. Uno script di shell riceve come parametro sulla riga di comando il nome di una directory.

   Lo script deve:

   - cercare all'interno dell'albero di direttori, specificato dal parametro, tutti i file di testo di estensione ‘‘.txt’’, che contengono almeno una riga che incomincia con una cifra numerica ($0 \div 9$).
   - cancellare tali righe dai rispettivi file.
   - ricopiare i file stessi (senza le righe cancellate) in file con lo stesso nome ma estensione ‘‘.mod’’.

   Non è consentito utilizzare né AWK né SED.

```
#!/bin/bash
if [ $# -ne 1 ]
then
  echo usage $0 dir
  exit 1
fi
for f in ‘find $1 -name "*.txt" -type f ‘
do
grep "^[0-9]." $f > "temp.txt"
if [ $? -eq 0 ]
then
  #file that has a line starting with a digit [0-9] found
  fo=‘basename $f .txt‘
  grep -v "^[0-9]." $f > $fo’.mod’
fi
done
```

6. I risultati di un programma nella gestione di diversi progetti sono memorizzati su file. Per ogni riga del file vengono indicati il nome del progetto e i tempi di esecuzione del programma durante tre fasi successive, come di seguito indicato:

```
progA  12.34  26.45 123.99
progB  32.45  16.45  23.23
ex001  56.34   6.45 343.99
pdtsw 112.84 265.45  56.82
```

Si scriva uno script AWK in grado di:

- ricevere il nome di due file del tipo indicato sulla riga di comando.
- visualizzare (a video) l'elenco di tutti i progetti contenuti nei due file, comprensivo del nome del progetto, dei tre tempi di esecuzione, della somma di tali tempi e di una stringa che indica se il progetto compare in entrambi i file oppure in uno solo. I progetti che compaiono in un solo file devono essere individuati dalla stringa "ONE". I progetti che compaiono in entrambi i file devono essere individuati dalla stringa "BEST". In quest'ultimo caso i tempi visualizzati (e la relativa somma) devono essere quelli relativi al (i.e., presenti nel) file per cui la somma dei tempi è minore.

Si osservi che non tutti i progetti sono riportati nei due file e che in ogni casi l'ordine dei progetti nei file non è identico.

```
#!/usr/bin/awk -f
{
   check_both[$1]++;
   sum1[$1]=$2+$3+$4;
   t11[$1]=$2;
   t12[$1]=$3;
   t13[$1]=$4;
}
END {
  while ((getline < ARGV[2])) {
    sum2[$1]=$2+$3+$4;
    check_both[$1]--;
    t21[$1]=$2;
    t22[$1]=$3;
    t23[$1]=$4;
  }
  for (i in check_both) {
    if (check_both[i]==0) {
      if (sum1[i]<sum2[i])
        print i " " t11[i] " " t12[i] " " t13[i] " " sum1[i] " BEST"
      else
        print i " " t21[i] " " t22[i] " " t23[i] " " sum2[i] " BEST"
    } else {
      if (check_both[i]>0)
        print i " " t11[i] " " t12[i] " " t13[i] " " sum1[i] " ONE"
      else
        print i " " t21[i] " " t22[i] " " t23[i] " " sum2[i] " ONE"
    }
  }
}
```