Caselle riservate

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

Sistemi Operativi

Compito d'esame

31 Agosto 2015

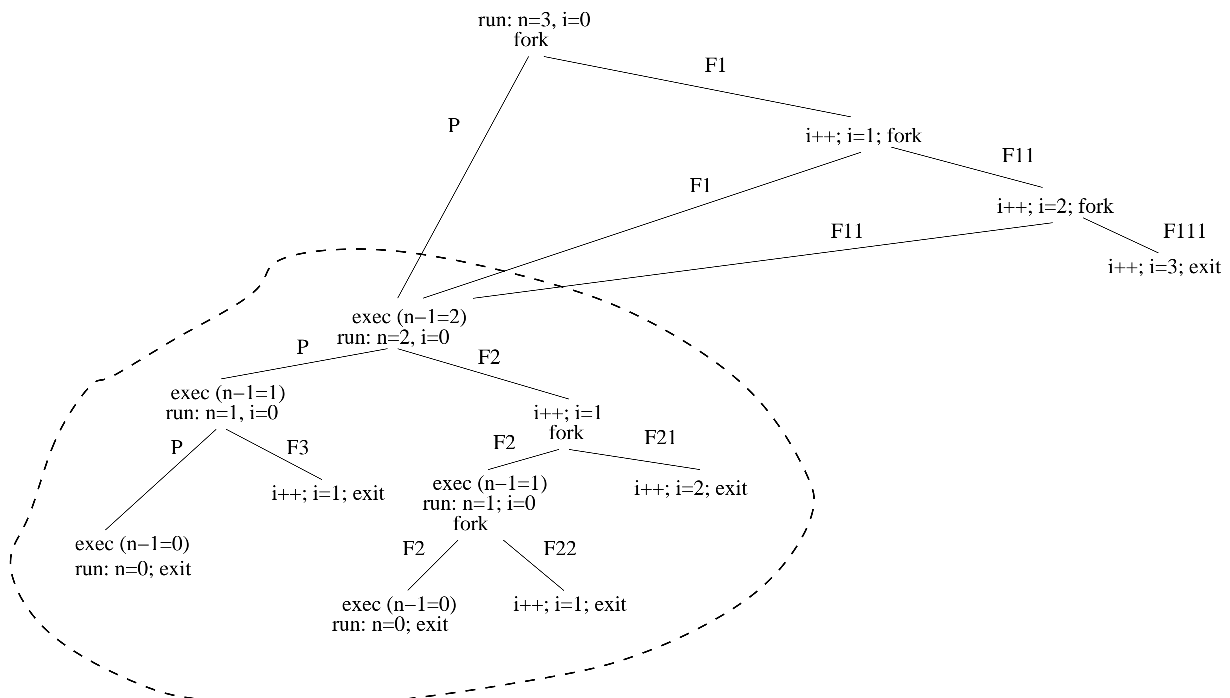
Matricola ______________ Cognome ______________________ Nome ______________________

Non si possono consultare testi, appunti o calcolatrici. Riportare i passaggi principali. L'ordine sarà oggetto di valutazione.
Durata della prova: 100 **minuti.**

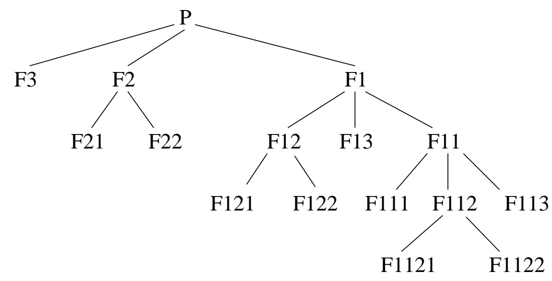
1. Si riporti l'albero di generazione dei processi a seguito dell'esecuzione del seguente tratto di codice C. Si supponga che il programma venga eseguito con un unico parametro, il valore intero 3, sulla riga di comando. Si indichi inoltre che cosa esso produce su video e per quale motivo.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/wait.h>
6
7  int main (int argc, char *argv[]) {
8    int i, n;
9    char str[50];
10
11   n = atoi (argv[1]);
12
13   printf ("run with n=%d\n", n); fflush (stdout);
14
15   for (i=0; i<n; i++) {
16     if (fork () > 0) {
17       sprintf (str, "%d", n-1);
18       execlp (argv[0], argv[0], str, NULL);
19     }
20   }
21
22   exit (0);
23 }
```

CFG:



Albero di generazione dei processi:



Output prodotto:

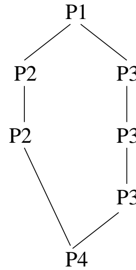
```
run with n=3 - compare 1 volta
run with n=2 - compare 3 volta
run with n=1 - compare 6 volta
run with n=0 - compare 6 volta
```

con 16 stampe in tutto.

2. Un programma concorrente è costituito da 4 processi (P_1, P_2, P_3, P_4) **aciclici**, **non** ricorsivi, che **non** si richiamano a vicenda e di cui sono eventualmente presenti diverse istanze in esecuzione contemporanea. I processi sono tali per cui:

   - P_1 viene eseguito per primo.
   - al termine di P_1, P_2 deve essere eseguito due volte in sequenza e P_3 tre volte in sequenza. Le varie esecuzioni di P_2 e di P_3 possono essere effettuate in parallelo.
   - una volta che P_2 e P_3 hanno terminato viene eseguito P_4.

   Si rappresenti il grafo di precedenza del sistema e si scriva il programma (in pseudo-codice) illustrandone i meccanismi di sincronizzazione utilizzando il minimo numero di semafori possibile.



Prima dell'esecuzione:

```
sem_t s2, s3, s4;
init (s2, 0);
init (s3, 0);
init (s4, 0);
int n2=0, n3=0;
```

Si esegue una istanza di P_1 e di P_4, due di P_2 e tre di P_3.

```
P1
printf ("P1\n");
signal (s2);
signal (s3);
```

```
P2
wait (s2);
n2++;
printf ("P2\n");
if (n2<2)
  signal (s2);
else
  signal (s4);
```

```
P3
wait (s3);
n3++;
printf ("P3\n");
if (n2<3)
  signal (s3);
else
  signal (s4);
```

```
P4
wait (s4);
wait (s4);
printf ("P4\n");
```

Al termine dell'esecuzione:

```
destroy (s2);
destroy (s3);
destroy (s4);
```

3. Si illustri il problema dei "*Readers e Writers*". Se ne riporti la soluzione mediante primitive semaforiche nel caso di precedenza ai Readers. Si indichi esattamente che cosa si intende per "precedenza ai Readers"?

   Partendo dal problema precedente, si realizzi il seguente schema di sincronizzazione. Si supponga di avere due gruppi di Readers, denominati R_1 e R_2. Ogni membro di R_1 (R_2) può accedere alla sezione critica insieme a altri membri di R_1 (R_2), però membri di R_1 e di R_2 devono accedere alla sezione critica in mutua esclusione. A quale altro schema di sincronizzazione ci si può ricondurre per la soluzione?

   Lo schema di sincronizzazione con due insieme di Readers è identico a quello del "Tunnel a senso alternato". Vedere lucidi e relative spiegazioni oppure i testi consigliati.

4. Realizzare un script BASH che ricevi due stringhe sulla riga di comando: il path assoluto di un direttorio e una estensione di file. Lo script deve elaborare tutti i file del direttorio di estensione indicata, cambiando tutte le occorrenze della stringa “bash” in “BASH” e ridenominando ciascun file con lo stesso nome ma con estensione “new”.

```
1  #!/bin/bash
2
3  if [ $# -ne 2 ]
4  then
5    echo "usage " $0 " dir ext"
6    exit 1
7  fi
8
9  for f in `find $1 -name "*.$2" -type f`
10 do
11   # Cancella path e estensione file
12   fo=`basename $f $2`
13   # Cancella file pre-esistenti
14   echo -n > $1/$fo"new"
15
16   while read line
17   do
18
19   # tr sostituisce anche i caratteri singoli
20   # echo $line | tr -t bash BASH >> $1/$fo"new"
21   for word in `echo $line`
22   do
23     if [ $word = "bash" ]
24     then
25       echo -n "BASH " >> $1/$fo"new"
26     else
27       echo -n "$word " >> $1/$fo"new"
28     fi
29   done
30   # Va a capo
31   echo >> $1/$fo"new"
32
33   done < $f
34
35 done
```

5. Uno script AWK riceve sulla riga di comando $n + 1$ parametri. I primi n parametri sono nomi di file di ingresso. L'ultimo è il nome di un file di uscita, ovvero generato dallo script. Lo script deve scrivere nel file di uscita, con formato libero e in ordine qualsiasi, tutte le parole che compaiono in tutti i file di ingresso, mentre deve eliminare ogni parola che **non** compaia in tutti i file di ingresso.

   Suggerimento.

   Per comprendere quale file di ingresso si sta parsificando si consiglia di procedere come segue. Mentre la variabile predefinita `NR` indica il numero di righe lette complessivamente, la variabile predefinita `FNR` indica il numero di righe lette ri-azzerandosi all'inizio di ciascun file parsificato. In altre parole alla variabile predefinita `FNR` viene assegnato il valore 1 ogni volta che viene parsificata la prima riga di ciascun file di ingresso.

   Soluzione A:

```
1  #!/bin/awk
2
3  BEGIN {
4    outFile = ARGV[ARGC-1]
5    ARGV[ARGC-1] = ""
6    fileId = 0;
7    print "" > outFile
8  }
9
10 {
11   if (FNR == 1){
12     fileId++;
13   }
14   for (i=1; i<=NF; i++) {
15     if (lastOccurrence[$i] == fileId-1)
16       lastOccurrence[$i]++
17   }
18 }
19
20 END {
21   for (parola in lastOccurrence){
22     if (lastOccurrence[parola] == fileId){
23       printf("%s\n", parola) >> outFile
24     }
25   }
26 }
```

   Soluzione B:

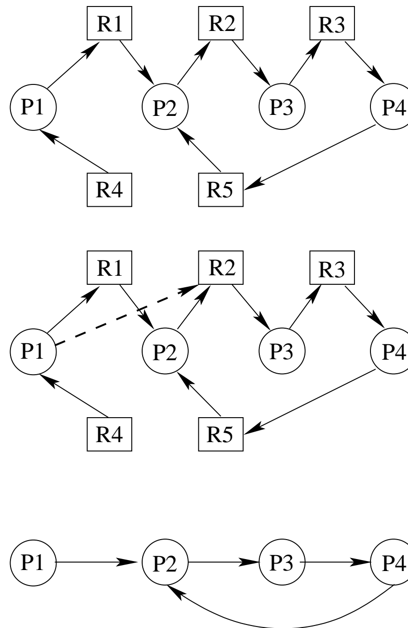
```
1  BEGIN {
2    fileout = ARGV[ARGC-1]
3    ARGV[ARGC-1] = ""
4    print "" > fileout
5    nfile = 0
6  }
7
8  {
9    if (FNR==1) {
10     nfile++
11   }
12
13   for (i=1; i<=NF; i++) {
14     M[nfile][$i]=1
15   }
16 }
17
18 END {
19   for (f in M){
20     if (isarray(M[f])) {
21       for (w in M[f]){
22         vett[w]+=M[f][w]
23       }
24     }
25   }
26
27   for (w in vett) {
28     if (vett[w] == ARGC-2)
29       print w >> fileout
30   }
31 }
```

6. In un sistema concorrente il sistema operativo deve gestire 5 risorse (R_1, R_2, R_3, R_4, R_5) con istanze unitarie. In un certo istante sono in esecuzione sul sistema i seguenti quattro processi:

- Il processo P_1, che ha ottenuto l'assegnazione della risorsa R_4, e ha richiesto la risorsa R_1.
- Il processo P_2, che ha ottenuto l'assegnazione delle risorse R_1, e R_5 e ha richiesto la risorsa R_2.
- Il processo P_3, che ha ottenuto l'assegnazione della risorsa R_2, e ha richiesto la risorsa R_3.
- Il processo P_4, che ha ottenuto l'assegnazione della risorsa R_3 e ha richiesto la risorsa R_5.

In futuro il processo P_1 richiederà anche una istanza di R_2. Si rappresentino il grafo di allocazione delle risorse, il grafo di attesa e il grafo di rivendicazione. Si indichi come è possibile rilevare e ripristinare una situazione di deadlock utilizzando il grafo di allocazione delle risorse.

Il grafo di allocazione delle risorse, il grafo di rivendicazione e il grafo di attesa sono illustrati in figura.



Rilevazione: si effettua tramite la determinazione dei cicli nel caso di risorse con istanze unitarie. La presenza di cicli diventa condizione necessaria, ma non sufficiente con istanze multiple. In questo caso occorre utilizzare l'algoritmo del banchiere.

Ripristino: terminare tutti i processi in stallo, terminare un processo alla volta sino a ripristinare una condizione corretta, prelazionare risorse a un processo una alla volta.