Caselle riservate

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

Sistemi Operativi

Compito d'esame

06 Febbraio 2017

Matricola ______________ Cognome ____________________ Nome ____________________

Docente: ◯ Quer ◯ Sterpone

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

1. Si riporti il grafo di controllo del flusso e 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 <unistd.h>
4   #include <sys/wait.h>
5
6   int main (int argc, char *argv[]) {
7     char str[20];
8     int n = atoi (argv[1]);
9
10    if (n>=0 && fork()) {
11      if (fork()>0) {
12        sprintf (str, "%d", n);
13        execlp ("echo", "echo", "n=", str, (char *) 0);
14        if (fork()>0) {
15          execlp (argv[0], argv[0], n-1, (char *) 0);
16        }
17      } else {
18        sprintf (str, "%s %d", argv[0], n-1);
19        printf ("%s\n", str);
20        if (fork()>0) {
21          system (str);
22        }
23      }
24    }
25
26    return (1);
27  }
```

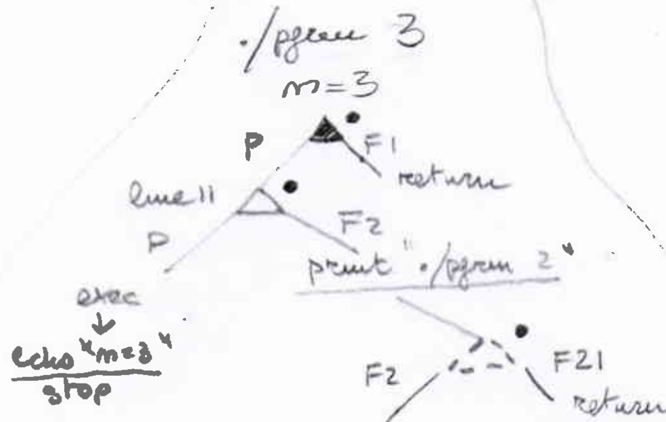
①

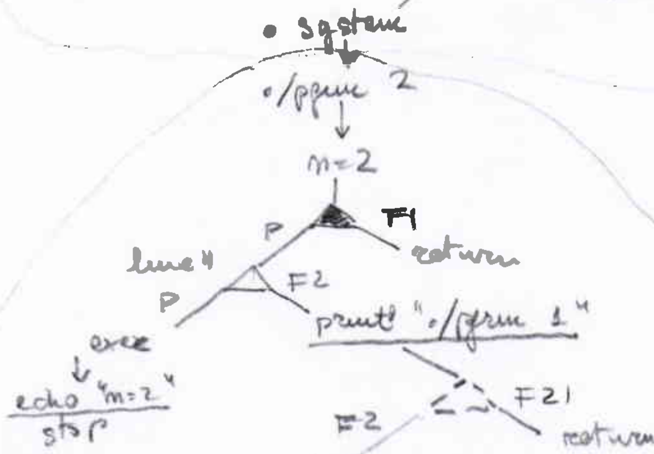
line 10: fork inutile
P continua
F esce

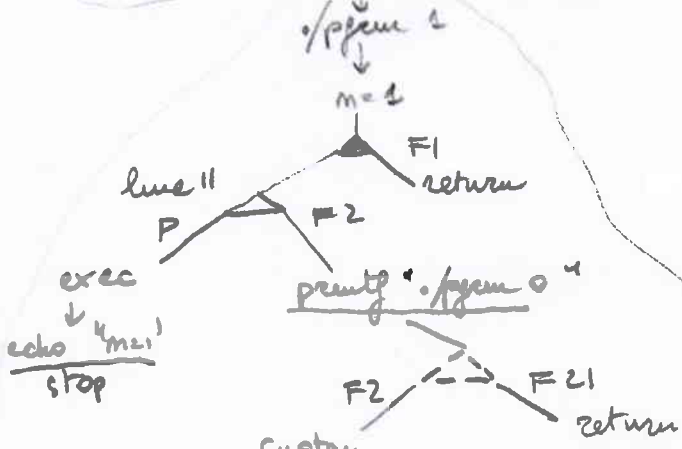
line 20: fork inutile
P fa system
F esce

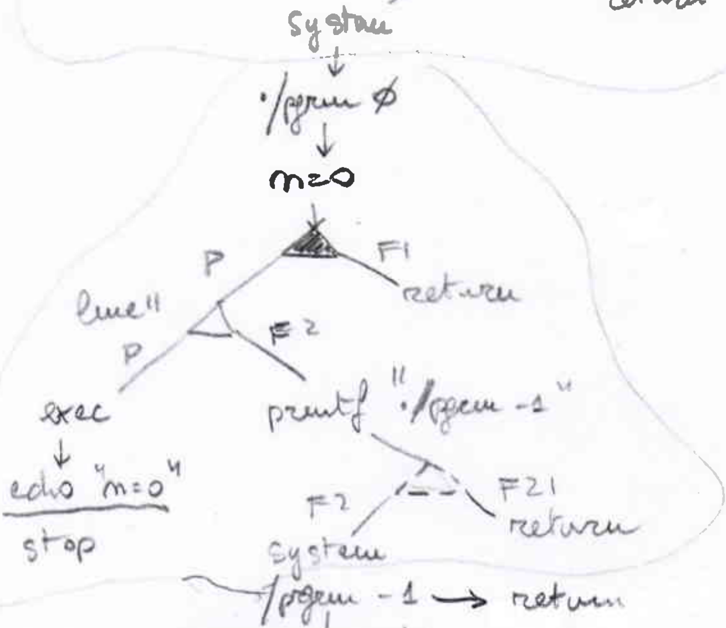
line 14÷16: mai eseguite (prima de la exec)

①

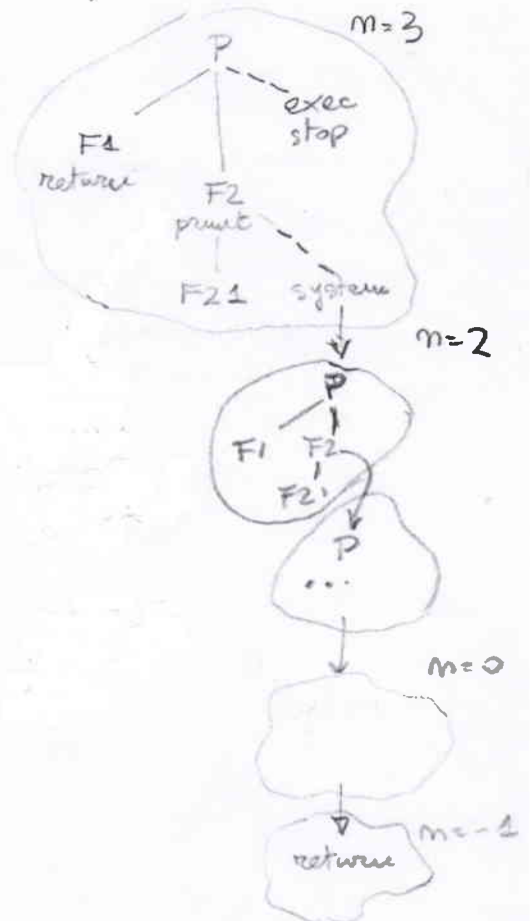








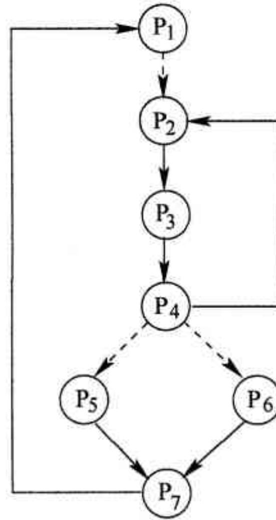
②



③

OUT:
./pgrm 2
n=~~4~~ 3
./pgrm 1
n=~~4~~ 2
./pgrm 0
n= 1
./pgrm -1
n= 0

2. 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)`). Inoltre il ciclo interno (processi 2, 3 e 4) deve essere eseguito 3 volte per ogni iterazione del ciclo esterno: la prima esecuzione incomincia una volta terminato P_1, mentre le successive due incominciano una volta terminato P_4; inoltre, i processi P_5 e P_6 devono essere eseguiti solo una volta terminata la terza iterazione. Utilizzare le primitive `init`, `signal`, `wait` e `destroy`. Indicare gli eventuali archi superflui, riportare il corpo dei processi ($P_1, \ldots, P_7$) e l'inizializzazione dei semafori.




```
int n=0;
sem_t S1,S2,S3,S4,S5,S6,S7;
init (S1,1);
init (S2,0);
 ...
init (S7,0);
```

```
destroy(S1);
...
destroy(S7);
```

```
P1()
  while(1) {
     wait (S1)
     PRINT("P1")
     signal(S2)
  }
```

```
P2()
  while(1) {
     wait(S2)
     PRINT("P2")
     signal(S3)
  }
```

```
P3()
  while(1) {
     wait(S3)
     PRINT("P3")
     signal(S4)
  }
```

```
P4()
  while(1) {
     wait(S4)
     PRINT("P4")
     if(n<2)
          n++; signal(S2);
     else
          n=0; signal(S5), signal(S6);
  }
```

```
P5()
  while(1) {
     wait (S5)
     PRINT("P5")
     signal (S7)
  }
```

```
P6()
  while(1) {
     wait (S6)
     PRINT (P6)
     signal (S7)
  }
```

```
P7()
  while(1) {
     wait (S7)
     wait (S7)
     PRINT (P7)
     signal (S1)
  }
```

3. Con riferimento alle soluzioni software e hardware al problema della sincronizzazione si descrivano la strategia di Peterson e quella basata sulla procedura di swap. Per entrambe le strategie, si riporti il codice e se ne illustrino le funzionalità, il protocollo di utilizzo, i pregi e i difetti.

① SW

```
Var globali :   int turn = i ;
                int flag [2] = { F, F } ;

Pi :  while (T) {
          flag [i] = T ;
          turn = J ;
          while (flag [J] && turn = J) ;
          SC
          flag [i] = F ;
          SC (barrato)
      }

PJ :  while (T) {
          flag [J] = T ;
          turn = i ;
          while (flag [i] && turn = i) ;
          SC
          flag [J] = F ;
          SC (barrato)
      }
```

Soddisfa ME, progresso (no deadlock), attesa definita (no starvation) e simmetria. Non generalizzabile facilmente a N processi
Richiede busy waiting su spin lock.

② HW

```
Var globali :   char lock = F ;

void swap (char *v1, char *v2) {
    char tmp ;
    tmp = *v1 ;
    *v1 = *v2 ;
    *v2 = tmp ;
    return ;
}

while (T) {
    key = T ;
    while (key == T)
        swap (&lock, &key) ;
    SC
    lock = F ;
    SC (barrato)
}
```

Tecniche con sistemi con diritto di prelazione.
Si basano su "lock" e operazioni "atomiche". Soddisfa ME, progresso, simmetria ma NON attesa definita.
Estendibili a N processi.

4. Due file contengono esclusivamente numeri interi, separati da un o più spazi e disposti con formato libero, ovvero in numero indefinito su ciascuna riga del file. Scrivere uno script BASH in grado di ricevere il nome di tali file sulla riga di comando, di verificare il loro corretto passaggio e l'esistenza dei file, e di visualizzare solo i numeri che **non** compaiono in entrambi i file.

```bash
#!/bin/bash

#
# SOLUTION A
# Command concatenation, temporary files to store out of intermediate steps. #
#

# Check correct number of parameters
if [ $# -ne 2 ]; then
    echo "Usage: es4.sh file1 file2"
    exit 1
fi

# Check if files are valid
if [ ! -f $1 ] || [ ! -f $2 ]; then
    echo "Invalid file or files!"
    exit 1
fi

# Streamline file format and remove duplicates
cat $1 | tr -s " \n" "\n" | sort | uniq > "tmp1.txt"
cat $2 | tr -s " \n" "\n" | sort | uniq > "tmp2.txt"

# Filter out shared numbers
cat "tmp1.txt" "tmp2.txt" | sort | uniq -u

# Remove temporary files
rm "tmp1.txt" "tmp2.txt"
```

```bash
#!/bin/bash

#
# SOLUTION B
# Iterates one time over each file, uses arrays to store unique numbers
#

# Check correct number of parameters
if [ $# -ne 2 ]; then
    echo "Usage: es4.sh file1 file2"
    exit 1
fi

# Check if files are valid
if [ ! -f $1 -o  ! -f $2 ]; then
    echo "Invalid file or files!"
    exit 1
fi

# Read numbers from the first file and store them in an array.
# Note that duplicate numbers are filtered through the array representation.
for num in $(cat $1); do
    vet1[$num]=$num
done

# Remove numbers that are shared with the second file and store
# unique numbers of the second file into a second array.
# Note that unique numbers of the second file cannot be stored in the first array
# direclty as duplicate numbers appearing only in such file would be marked as
removed.
# Duplicate numbers of the second file are filtered through the array
representation.
for num in $(cat $2); do
    if [ "${vet1[$num]}" != "" ]; then
        vet1[$num]="R"
    elif [ "${vet1[$num]}" != "R" ]; then
        vet2[$num]=$num
    fi
done

# Print all unique numbers
for num in ${vet1[*]} ${vet2[*]}; do
    if [ "$num" != "R" ]; then
        echo $num
    fi
done
```

```
#!/bin/bash

#
# SOLUTION C
# Iterates two time over each file, uses arrays to store unique numbers.
#
# Check correct number of parameters
if [ $# -ne 2 ]; then
    echo "Usage: es4.sh file1 file2"
    exit 1
fi

# Check if files are valid
if [ ! -f $1 -o  ! -f $2 ]; then
    echo "Invalid file or files!"
    exit 1
fi

# Read numbers from the first file and store them into an array
while read line; do
    for num in $line; do
        vet1[$num]=$num
    done
done < $1

# Remove numbers shared with the second file
while read line; do
    for num in $line; do
        vet1[$num]=""
    done
done < $2

# Read numbers from the second file and store them into an array
while read line; do
    for num in $line; do
        vet2[$num]=$num
    done
done < $2

# Remove numbers shared with the first file
while read line; do
    for num in $line; do
        vet2[$num]=""
    done
done < $1

# Print all unique numbers
for num in ${vet1[*]} ${vet2[*]}; do
    echo $num
done
```

```bash
#!/bin/bash

#
# SOLUTION D
# Iterates one time over each file, uses grep to test
#

# Check correct number of parameters
if [ $# -ne 2 ]; then
    echo "Usage: es4.sh file1 file2"
    exit 1
fi

# Check if files are valid
if [ ! -f $1 -o  ! -f $2 ]; then
    echo "Invalid file or files!"
    exit 1
fi

# Read numbers from the first file
# check if they are contained in the second file and store them in an array.
# Note that duplicate numbers are filtered through the array representation.
for num in $(cat $1); do
    grep -e "\<$num\>" $2 > /dev/null        #Or --quiet
    if [ $? -eq 1 ]; then
        vet[$num]=$num
    fi
done

# Read numbers from the second file
# check if they are contained in the first file and store them in an array.
# Note that duplicate numbers are filtered through the array representation.
for num in $(cat $2); do
    grep -e "\<$num\>" $1 > /dev/null        #Or --quiet
    if [ $? -eq 1 ]; then
        vet[$num]=$num
    fi
done

# Print all unique numbers
for num in ${vet[*]}; do
    echo $num
done
```

5. Scrivere uno script AWK in grado di manipolare l'output di un riconoscitore vocale secondo le seguenti specifiche.

   Il riconoscitore vocale genera un file con le seguenti caratteristiche:

```
pronounced word : one          utterance n. :     1
recognized word : one          prob. : -0.15530174E+02
pronounced word : one          utterance n. :     2
recognized word : one          prob. : -0.92723360E+01
pronounced word : one          utterance n. :     3
recognized word : one          prob. : -0.95529728E+01
pronounced word : one          utterance n. :     4
recognized word : eight        prob. : -0.11542629E+02
pronounced word : one          utterance n. :     5
recognized word : one          prob. : -0.10308277E+02
pronounced word : two          utterance n. :     6
recognized word : two          prob. : -0.15609604E+02
pronounced word : two          utterance n. :     7
recognized word : two          prob. : -0.81284819E+01
pronounced word : two          utterance n. :     8
recognized word : two          prob. : -0.92986431E+01
pronounced word : two          utterance n. :     9
recognized word : zero         prob. : -0.12929784E+02
```

   in cui diverse parole pronunciate possono essere riconosciute correttamente o meno con una determinata probabilità. Il nome di tale file viene ricevuto dallo script sulla riga di comando.

   Lo script deve quindi parsificare tale file e produrre in uscita la cosiddetta "matrice di confusione" con il seguente formato:

```
        one   zero    two  eight    tot  percent_correct
  one     4      0      0      1      5   80.00%
 zero     0      0      0      0      0
  two     0      1      3      0      4   75.00%
eight     0      0      0      0      0
```

   In essa le parole pronunciate e/o riconosciute sono riportate lungo le righe e le colonne e, in base al numero di riconoscimenti indicati negli elementi stessi della matrice, vengono calcolate le percentuali di correttezza (e.g., $4/5 = 0.8$ ovvero 80%, $3/4 = 0.75$ ovvero 75%, etc.).

```
#!/bin/gawk

#
# SOLUTION A - Single words vector.
#

{
    # Get pronounced and recognized word
    p = $4
    getline
    r = $4

    # Record pronounced-recognized association in the matrix
    mat[p,r]++;

    # Update words array
    words[p]++
    words[r]++
}

END {

    # Print column header
    printf("\t");
    for(w in words){
        printf("%s\t", w);
    }
    printf("tot\tpercent_correct\n");

    # Print table rows
    # Note that matrices in AWK are associative array
    # Thus, we cannot iterate over mat as we would iterate over a C-like matrix
    #
    # for(x in mat){
    #   for(y in mat[y]){
    #       ...
    #   }
    # }
    for(w1 in words){

        # Print row header
        printf("%s\t", w1);

        # Print row entries while computing total and correct counters
        tot = 0;
        for(w2 in words){
            n = mat[w1,w2];
            printf("%d\t", n);
            tot += n;
        }

        # Print total
        printf("%d\t", tot);

        # Print correctness percentage
        if(tot > 0) {
            printf("%.2f%", mat[w1,w1]/tot*100);
        }
        printf("\n")
    }
}
```

```
#!/bin/gawk

#
# SOLUTION B - Separate vectors for pronounced and recognized words.
#

{
    # Get pronounced and recognized word
    p = $4
    getline
    r = $4

    # Record pronounced-recognized association in the matrix
    mat[p,r]++;

    # Update words arrays
    pronounced[p]++
    recognized[r]++
}

END {
    # Print column header
    printf("\t");
    for(w in recognized){
        printf("%s\t", w);
    }
    printf("tot\tpercent_correct\n");

    # Print table rows
    # Note that matrices in AWK are associative array
    # Thus, we cannot iterate over mat as we would iterate over a C-like matrix
    #
    # for(x in mat){
    #   for(y in mat[y]){
    #       ...
    #   }
    # }
    for(p in pronounced){

        # Print row header
        printf("%s\t", p);

        # Print row entries while computing total and correct counters
        tot = 0;
        for(r in recognized){
            n = mat[p,r];
            printf("%d\t", n);
            tot += n;
        }

        # Print total
        printf("%d\t", tot);

        # Print correctness percentage
        if(tot > 0) {
            printf("%.2f%", mat[p,p]/tot*100);
        }
        printf("\n")
    }
}
```

```
#!/bin/gawk

#
# SOLUTION C
# Two parallel vectors of pronounced and recognized words.
#

BEGIN {
    i=0;
}

{
    # Get pronounced and recognized word
    p = $4
    getline
    r = $4
    # Record pronounced-recognized association in two parallel vectors
    pronounced[i] = p
    recognized[i] = r
    i++
    # Update words arrays
    words[p] = 1
    words[r] = 1
}

END {
    # Print column header
    printf("\t");
    for(w in words){
        printf("%s\t", w);
    }
    printf("tot\tpercent_correct\n");
    #Print table rows
    for(p in words){
        # Print row header
        printf("%s\t", p);
        # Print recognized words counts
        tot = 0
        correct = 0
        for(r in words){
            count = 0
            for(j=0; j<i; j++){
                if(pronounced[j] == p && recognized[j] == r){
                    count++
                }
            }
            printf("%d\t", count);
            tot += count;
            if(p == r){
                correct += count
            }
        }

        # Print total
        printf("%d\t", tot);

        # Print correctness percentage
        if(tot > 0) {
            printf("%.2f%", correct/tot*100);
        }
        printf("\n")
    }
}
```

6. Si supponga che un disco rigido sia costituito da 25 blocchi, che i blocchi liberi siano indicati con 0 e quelli occupati con 1, e che la situazione iniziale del disco sia la seguente:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	0	1	1	0	0	0	0	0	1	0	0	1	1	1	1	0	0	0	0	1	0	0	1

Indicare le principali caratteristiche delle metodologie di allocazione di file contigua, concatenata, FAT e indicizzata. Utilizzando tali tipi di allocazione e il disco precedentemente descritto, indicare come possono essere allocati in sequenza i seguenti file: `File1`, `File2` e `File3` ciascuno di dimensione uguale a 4 blocchi.

① CONTIGUA: ogni file occupa insieme contiguo di blocchi

File	Blocchi Occupati	Dir Entry Start	Dir Entry length
File1	5, 6, 7, 8	5	4
File2	17, 18, 19, 20	17	4
File3	NON ALLOCABILE		

Semplice per accessi sequenziali e diretti.
Occorre politica di allocazione. Produce frammentazione esterna.

② CONCATENATA: lista concatenata di blocchi. I ↑ sono memorizzati nei blocchi su disco.

File	Blocchi	Dir Entry Start	Dir Entry End
File 1	0 → 1 → 2 → 5	0	5
File 2	6 → 7 → 8 → 9	6	9
File 3	11 → 12 → 17 → 18	11	18

Permetta allocazione dinamica dei file. Efficace solo per accessi sequenziali. Memorizza ↑ (spazio e criticità).

③ FAT: Tabella con 1 elemento ∀ blocco del disco
Come la precedente MA i ↑ sono memorizzati nella FAT (File Allocation Table)

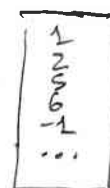
File	Dir Entry
File1	0
File2	6
File3	11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	2	5			-1	7	8	9	-1		12	17					18	-1						

FAT

④ INDICIZZATA: Uso blocco indice. 1 ∀ file

File	Dir Entry	Blocco Indice
File1	0	1, 2, 5, 6, -1, ...
File2	7	8, 9, 11, 12, -1, ...
File3	17	18, 19, 20, 22, -1, ...



Elimina frammentazione esterna ed è OK per allocazione dinamica. Richiede blocco indice (dimensione?)