

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

# Sistemi Operativi

## Compito d'esame

29 Giugno 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 descrivano le caratteristiche principali delle *pipe* per la comunicazione e la sincronizzazione tra processi. Se ne illustri l'utilizzo realizzando il seguente programma in linguaggio C. Un processo  $P$  genera un figlio  $F_1$  che a sua volta genera un figlio  $F_2$ .  $F_2$  legge da tastiera un intero, la trasmette a  $F_1$  su una prima pipe e termina.  $F_1$  trasforma il numero nel suo opposto e lo trasmette a  $P$  su una seconda pipe e termina.  $P$  visualizza il numero ricevuto su standard output e termina.

I processi concorrenti possono essere indipendenti oppure cooperanti. Un processo è indipendente se non può essere influenzato dagli altri processi e non può influenzare l'esecuzione di altri processi. Un processo è cooperante in caso contrario. La cooperazione può avvenire tramite lo scambio oppure la condivisione di dati. Scambio e condivisione di dati richiedono l'implementazione di meccanismi opportuni.

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 sono bloccanti per pipe vuota o piena, rispettivamente.

Il seguente è un esempio di utilizzo:

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/wait.h>
6
7 void P (int []);
8 void F1(int []);
9 void F2(int []);
10
11 int
12 main (
13     int argc,
14     char ** argv
15 )
16 {
17     int fd1[2];
18     // Create pipe
19     pipe (fd1);
20
21     if (fork()!=0) {
22         P(fd1);
23     } else {
24         F1(fd1);
25     }
26
27     return (0);
28 }
29
30 void P(int fd1[2]) {
31     int n;
32     close (fd1[1]);
33     read (fd1[0], &n, sizeof (int));
34     printf ("P (pid=%d): %d\n", getpid(), n);
35     exit (0);
36 }
37
38 void F1(int fd1[2]) {
39     int n, fd2[2];
40     close (fd1[0]);
41     pipe (fd2);
42     if (fork()!=0) {
43         close (fd2[1]);
44         read (fd2[0], &n, sizeof (int));
45         printf ("F2 (pid=%d): receives %d\n", getpid (), n);
46         n = (-n);
47         printf ("F2 (pid=%d): transmit %d\n", getpid (), n);
48         write (fd1[1], &n, sizeof (int));
49         exit (0);
50     } else {
51         F2(fd2);
52     }
53 }
54
55 void F2(int fd2[2]) {
56     int n;
57     close (fd2[0]);
58     printf ("F2 (pid=%d): ", getpid ());
59     scanf ("%d", &n);
60     write (fd2[1], &n, sizeof (int));
61     exit (0);
62 }
63

```

2. Con riferimento allo scheletro di una shell UNIX/Linux si scriva un programma, in linguaggio C, in grado di:

- leggere da tastiera dei comandi Linux con eventuali parametri sulla riga di comando (ad esempio “cp file1 file2”, “ls -laR”, etc.)
- eseguire tali comandi in background, procedendo (in foreground) alla lettura del comando successivo.
- terminare il procedimento, ovvero la lettura di nuovi comandi Linux una volta ricevuto da tastiera il comando “end”.

Si osservi che, a scelta del candidato, la stringa “end” può essere utilizzata anche per delimitare la fine di ciascun comando letto da tastiera.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <time.h>
5 #include <string.h>
6 #include <sys/wait.h>
7
8 #define OPZIONI 10
9 #define LEN 10
10 #define MAX 80
11
12 int
13 main (
14     int argc,
15     char* argv[]
16 )
17 {
18     char tmp[MAX], **mat, *tmpP;
19     FILE *fp;
20     int i, nString;
21     pid_t pid;
22
23     // Alloco a priori ... anche se sarebbe meglio essere piu' accurati
24     mat = (char **) malloc (OPZIONI * sizeof (char *));
25     for (i=0; i<OPZIONI; i++) {
26         mat[i] = (char *) malloc (LEN * sizeof (char));
27     }
28
29     /*
30      * System Call: Exec
31      */
32
33     fp = fopen(argv[1], "r");
34     if (fp==NULL) {
35         fprintf (stderr, "Error opening file.");
36         exit (1);
37     }
38
39     nString = 0;
40     while (fscanf(fp, "%s", tmp) != EOF) {
41         if (strcmp(tmp,"end") != 0) {
42             strcpy (mat[nString], tmp);
43             nString++;
44         } else {
45             printf ("Comando: ");
46             for (i=0; i<nString; i++) {
47                 printf ("%s ", mat[i]);
48             }
49             printf ("\n");
50             tmpP = mat[nString];
51             mat[nString] = NULL;
52             pid = fork();
53             if (pid==0) {
54                 execvp (mat[0], mat);
55             } else {
56                 sleep (3);
57             }
58             mat[nString] = tmpP;
59             nString = 0;
60         }
61     }
62     fclose(fp);
63
64     return (1);
65 }
```

3. Si riportino quattro possibili soluzioni software al problema delle sezioni critiche con due processi denominati  $P_i$  e  $P_j$ . Si motivi l'erroneità oppure la correttezza di ciascuna soluzione.

Starvation:  $P_i$  e  $P_j$  devono entrare nella SC ma in maniera alternata. turn inizializzata a 0 o 1.

```
Pi                                Pj
while (TRUE) {                    while (TRUE) {
  while (turn==j);                 while (turn==i);
  SC di Pi                         SC di Pj
  turn = j;                        turn = i;
  sezione non critica            sezione non critica
}
```

Mutua esclusioni non assicurata:  $P_i$  e  $P_j$  possono entrare entrambi nella SC. flag di 2 elementi inizializzati a 0.

```
Pi                                Pj
while (TRUE) {                    while (TRUE) {
  while (flag[j]);                 while (flag[i]);
  flag[i] = TRUE;                  flag[j] = TRUE;
  SC di Pi                         SC di Pj
  flag[i] = FALSE;                 flag[j] = FALSE;
  sezione non critica            sezione non critica
}
```

Deadlock:  $P_i$  e  $P_j$  possono rimanere bloccati per sempre. flag di 2 elementi inizializzati a 0.

```
Pi                                Pj
while (TRUE) {                    while (TRUE) {
  flag[i] = TRUE;                  flag[j] = TRUE;
  while (flag[j]);                 while (flag[i]);
  SC di Pi                         SC di Pj
  flag[i] = FALSE;                 flag[j] = FALSE;
  sezione non critica            sezione non critica
}
```

Algoritmo corretto (Peterson). turn e flag definite e inizializzate come in precedenza.

```
Pi                                Pj
while (TRUE) {                    while (TRUE) {
  flag[i] = TRUE;                  flag[j] = TRUE;
  turn = j;                        turn = i;
  while (flag[j] && turn==j);      while (flag[i] && turn==i);
  SC di Pi                         SC di Pj
  flag[i] = FALSE;                 flag[j] = FALSE;
  sezione non critica            sezione non critica
}
```

4. Si riportino i comandi UNIX per effettuare le operazioni indicate, utilizzando eventuali ridirezioni e pipe:

- Nel direttorio “/home/foo” cercare i file con il nome che inizia con il carattere “L” e estensione “txt”. In questi file ricercare al presenza della stringa “laib”. Visualizzare il nome del file e l'intera riga in cui tale stringa viene rintracciata.
- Trovare tutti i file di estensione “txt” nel direttorio “/home” memorizzati tra il livello di profondità 3 (incluso) e il livello di profondità 5 (incluso) dell'albero dei direttori e che siano leggibili. Di questi modificare il proprietario in “ugo”.
- Per ogni file di estensione “txt” presente nel direttorio corrente ricavare il nome e il numero di caratteri presenti nel file. L'elenco venga ordinato in base al numero di caratteri in ordine numerico inverso e memorizzato nel file “stat.txt”.
- Fare un esempio di utilizzo di ciascuno dei tre filtri cut, tr, uniq, spiegando l'effetto di ciascun comando.
- Una applicazione C è formata dai file main.c, f1.c, f2.c e main.h. Scrivere un Makefile con due target. Il primo sia in grado di compilare l'applicazione denominando l'eseguibile myapp. Il secondo rimuova eventuali file temporanei e sposti l'eseguibile nel direttorio “/user/bin”.

```
find /home/foo -name "L*.txt" -exec grep -H "laib" '{}' \;
```

```
find /home -mindepth 3 -maxdepth 5 -name "*.txt" -readable \  
-exec chown "ugo" '{}' \;
```

```
find . -name "*.txt" -exec wc -c '{}' \; | sort -rn -k 1,1 > stat.txt
```

```
cut -c2 test.txt
```

Visualizza il secondo carattere di tutte le righe del file test.txt.

```
echo ciao | tr a-z A-Z
```

Converte minuscole in maiuscole.

```
uniq -d test.txt
```

Stampa solo le linee duplicate del file test.txt

Makefile:

```
compile: main.c f1.c f2.c  
    gcc -o myapp main.c f1.c f2.c  
install:  
    rm *.tmp  
    cp myapp /user/bin
```

5. Scrivere un script AWK in grado di effettuare sostituzioni multiple in un testo soddisfacendo alle specifiche successive.

Un primo file include su righe successive coppie di stringhe, nel formato:

```
stringa1 stringa2
```

Lo script deve sostituire ogni comparsa della `stringa1` con la `stringa2` in un file di ingresso, generando il corrispondente file di uscita. Si osservi che le stringhe possono degenerare in caratteri singoli e devono essere rintracciate e sostituite nel testo in ingresso anche come sotto-stringhe di stringhe di lunghezza maggiore. Si supponga inoltre che tutte le sostituzioni siano indipendenti e possano essere applicate in qualsiasi ordine. Lo script riceve tre stringhe sulla riga di comando. Tali stringhe individuano il file di conversione, quello di ingresso e quello di uscita da generare.

### Esempio

Si supponga lo script venga eseguito come segue:

```
myScript file.txt inFile.txt outFile.txt
```

e che i file di nome `file.txt` e `inFile.txt` abbiano il contenuto di seguito indicato. Lo script deve generare il file `outFile.txt` riportato.

file.txt	inFile.txt	outFile.txt
awk Awk	ScriptawkInIngresso	ScriptAwkInIngressO
unix UNIX	Contenente unix e UNIX	COntenente UNIX e UNIX
o O	e testo vario	e testO vario

```
1 #!/usr/bin/awk -f
2
3 BEGIN {
4
5     #Fix arguments
6     convFile = ARGV[1]
7     inFile = ARGV[2]
8     outFile = ARGV[3]
9     ARGV[1] = inFile
10    delete ARGV[2]
11    delete ARGV[3]
12
13    #Load conversions
14    while(getline < convFile){
15        conv[$1] = $2
16    }
17 }
18
19
20
21 {
22     for (word in conv){
23         gsub (word, conv[word])
24     }
25     print ($0) > outFile
26 }
```

**6. Per i candidati iscritti al corso nell'anno accademico 2014–2015.**

Indicare le principali caratteristiche delle tipologie di allocazione di file (contigua, concatenata, indicizzata). Che cosa si intende per FAT? Che cosa si intende per collegamento o link?

**Per i candidati iscritti al corso negli anni accademici 2012–2013 o 2013–2014.**

Si illustri la struttura di un file system UNIX con particolare riferimento alla memorizzazione di file e direttori. Che cosa si intende per collegamento o link?

Vedere lucidi e relative spiegazioni oppure i testi consigliati.