Caselle riservate

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

Sistemi Operativi

Compito d'esame

27 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 descriva l'utilizzo dei segnali nel sistema operativo Unix/Linux con relativi vantaggi e svantaggi. Si descrivano in particolare le system call `signal` e `kill`.

   Si riportino inoltre due esempi di gestione dei segnali. Il primo illustri come la system call `alarm` possa essere implementata tramite le system call `fork`, `signal`, `kill` e `pause`. Il secondo descriva come la system call `alarm` possa essere utilizzata da un processo che desideri continuare a svolgere i suoi compiti ma visualizzare dopo 10 secondi un preciso messaggio su standard output.

Un segnale è un interrupt software, i.e., un evento di sistema inviato a un processo. I segnali permettono di gestire eventi asincroni. Non sono interrupt perchè gli interrupt sono inviati dallhardware al sistema operativo. I segnali sono inviati a processi da altri processi.
Sono utilizzati per notificare il verificarsi di eventi particolari: Condizioni di errore, violazioni di segmentazione di memoria, errori floating point o istruzioni illegali, etc.
Per esempio `ctrl-C` invia un segnale `INT` (`SIGINT`) e `ctrl-Z` invia un segnale `TSTP` (`SIGTSTP`).

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

Il file `signal.h` definisce i nomi dei segnali (i.e., `SIGABORT` Process abort, `SIGALARM` Alarm clock, etc.).
La `signal` consente di settare un signal handler. Ha due parametri: `sig` che specifica il segnale da intercettare e `func` che indica la funzione da invocare. `func` ha un solo parametro di tipo int (il segnale ricevuto).
Quando un segnale si presenta è possibile: (a) Ignorare esplicitamente il segnale (tale comportamento è impossibile per i segnali SIGKILL e SIGSTOP che non possono essere ignorati), (b) Lasciare che si verifichi il comportamento di default, (c) Catturare il segnale.

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

Invia un segnale a un processo o a un gruppo di processi.
È possibile spedire segnali solo a processi con lo stesso UID.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

Permette di attivare un timer (count-down) che terminerà a un preciso istante futuro. Quando il count-down termina il segnale `SIGALARM` viene generato. Il parametro `seconds` specifica il numero di secondi dopo i quali viene generato il segnale. Il valore di ritorno è pari al numero di secondi rimasti prima dellinvio del segnale come definito da chiamate precedenti. Il valore 0 indica il caso non ci siano state chiamate precedenti.

Esempio 1:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void myAlarm (int sig) {
  printf ("Alarm\n");
}

int main (void) {
  pid_t pid;
  (void) signal (SIGALRM, myAlarm);
  if (fork() == 0) {
    sleep(5);
    kill (getppid(), SIGALRM);
    exit(0);
  }
  /* father */
  pause ();
  exit (0);
}
```

Esempio 2:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void myAlarm (int sig) {
  printf ("Alarm Ended\n");
}

...
(void) signal (SIGALRM, myAlarm);
alarm (10);
...
// do what it has to be done
...
}
```

2. Si descrivano le caratteristiche principali delle *pipe* per la comunicazione e la sincronizzazione tra processi e se ne illustri l'utilizzo realizzando il seguente programma in linguaggio C.

   Un processo P, dopo aver generato un figlio F, effettua le seguenti operazioni:

   (a) Genera un numero casuale intero r incluso nell'intervallo $[1, 3]$.

   (b) Se $r = 1$ legge da standard input un numero intero n e una stringa s, trasmette al figlio F su una pipe il carattere `F`, l'intero n, e la stringa s, e ritorna al punto (a).

   (c) Se $r = 2$ trasmette al figlio F il carattere `X` e ritorna dal punto (a).

   (d) Se $r = 3$ termina.

   Il processo figlio F effettua le seguenti operazioni:

   (a) Legge i dati trasmessi sulla pipe (un carattere `F` seguito da altri dati, oppure un carattere `X` singolo).

   (b) Se il primo carattere ricevuto è una `F`, genera a sua volta un figlio e ritorna al punto (a). Il figlio (del processo `F`) si occupa di visualizzare n volte a distanza di 1 secondo la stringa s e termina ritornando come valore di terminazione il valore di n.

   (c) Se il primo carattere ricevuto è una `X`, raccoglie e visualizza su standard output i codici di terminazione di tutti i suoi figli (generati sino a quel momento) e termina.

   La stringa s abbia lunghezza costante uguale a 20 caratteri e non contenga spazi. Si ricorda che il codice di terminazione di un figlio può essere catturato con la macro `WEXITSTATUS`. Si utilizzi l'espressione `((int)(rand()%3)+1)` per generare un numero casuale nell'intervallo $[1, 3]$.

   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 una pipe vuota o piena, rispettivamente.

   Eventuale descrizione delle system call `pipe`, `close`, `read` e `write`.

```
1   #include <unistd.h>
2   #include <sys/wait.h>
3   #include <stdlib.h>
4   #include <stdio.h>
5   #include <string.h>
6
7   #define SIZE 20
8
9   void child (int []);
10  void parent (int []);
11
12  int main() {
13    int file[2];
14
15    if (pipe(file) == 0) {
16      if (fork () == 0) {
17        child (file);
18      } else {
19        parent (file);
20      }
21    }
22    exit(EXIT_SUCCESS);
23  }
24
25  void parent (int file[2]) {
26    int rn, n;
27    char f = 'F', x = 'X', str[SIZE];
28
29    // parent writes
30    close (file[0]);
31    while (1) {
32      rn = (int) ((rand() % 3) + 1);
33      if (rn==1) {
34        printf ("N and STR: ");
35        scanf ("%d%s", &n, str);
36        write (file[1], &f, sizeof (char));
37        write (file[1], &n, sizeof (int));
38        write (file[1], str, SIZE * sizeof (char));
39      } else
40      if (rn==2) {
41        write (file[1], &x, sizeof (char));
42      } else {
43        return;
44      }
45    }
46  }
47
48  void child (int file [2]) {
49    char c, str[SIZE];
50    int i, ns, nc, stat;
51
52    // child reads
53    close (file[1]);
54
55    nc = 0;
56    do {
57      read (file[0], &c, sizeof (char));
58      switch (c) {
59        case 'F':
60          read (file[0], &ns, sizeof (int));
61          read (file[0], str, SIZE * sizeof (char));
62          if (fork()==0) {
63            for (i=0; i<ns; i++) {
64              sleep (1);
65              fprintf (stdout, "%s\n", str);
66            }
67          }
68          exit (ns);
69          break;
70        case 'X':
71          for (i=0; i<nc; i++) {
72            wait (&stat);
73            fprintf (stdout, "Return Value: %d\n", WEXITSTATUS (stat));
74          }
75          break;
76      }
77    } while (c!='X');
78  }
```

3. Nello studio delle situazioni di stallo (deadlock), si chiarisca che cosa si intende per "condizioni necessarie", illustrando singolarmente tali condizioni. Si dimostri inoltre (in maniera formale) come sia possibile evitare l'*attesa circolare*.

Siano dati infine i tre processi successivi utilizzanti i semafori $A, \ldots, F$ tutti inizializzati a 1. Esiste una sequenza temporale di esecuzione di P_1, P_2 e P_3 tale per cui si crei una situazione di stallo? Motivare la risposta.

```
P1: while (1) {
      wait (D);
      wait (E);
      wait (B);
      printf ("P1\n");
      signal (D);
      signal (E);
      signal (B);
    }
```

```
P2: while (1) {
      wait (C);
      wait (F);
      wait (D);
      printf ("P2\n");
      signal (C);
      signal (F);
      signal (D);
    }
```

```
P3: while (1) {
      wait (A);
      wait (B);
      wait (C);
      printf ("P3\n");
      signal (A);
      signal (B);
      signal (C);
    }
```

Condizioni necessarie ma non sufficienti :

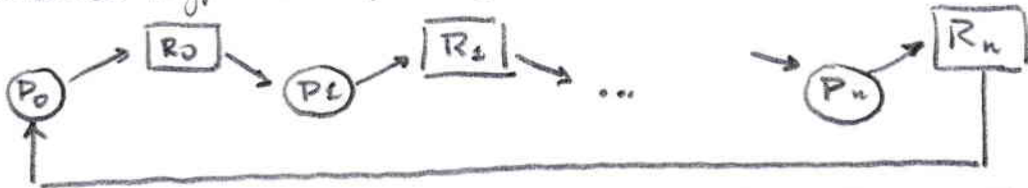
- Mutua esclusione : ...
- Possesso e attesa : ...
- No prelazione : ...
- Attesa circolare : ...

F = funzione che impone ordine univoco tra risorse R_i

Ogni processo P chiede risorse in ordine $F(R_{new}) > F(R_{old})$

(altrimenti la risorsa NON viene concessa)

Condizione sufficiente per evitare attesa circolare :

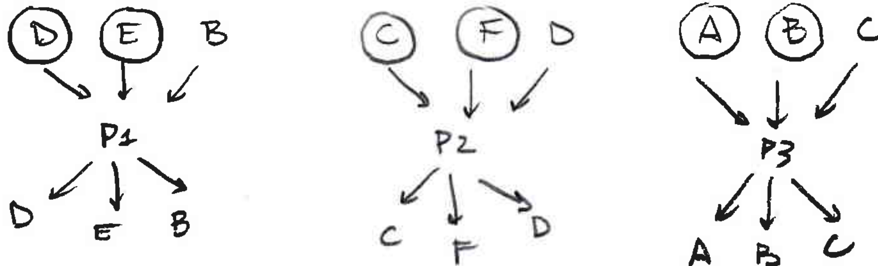


L'ordine di richiesta IMPONE $F(R_k) < F(R_{k+1})$ cioè

$F(R_0) < F(R_1) < \ldots < F(R_n) < F(R_0)$

assurdo

Esempio :



Ogni P acquisisce 2 risorse ≡ ◯ (risorse acquisite).

B, D, C sono già acquisite (da P_3, P_1, P_2) ⇒ STALLO !

4. Per ottimizzare lo spazio su disco, si scriva uno script BASH in grado di:
   - Ricevere 3 parametri sulla riga di comando, i.e., il nome di un utente, un numero intero e il nome di un file di log, verificandone la corretta ricezione.
   - Elencare ricorsivamente tutti i file presenti nella home directory dell'utente di dimensioni maggiori al numero specificato (valore in MBytes) e per ognuno di essi chiedere all'utente se desidera:
     - `D`: (delete) cancellare il file.
     - `Z`: (gzip) comprimere il file (cancellando poi il file originale).

     Lo script deve effettuare le operazioni richieste e, per ognuna di esse, appendere nel file di log una riga del tipo:
     `type user date originalSize currentSize path`
     in cui il campo:
     - `type` è `D` nel caso di un file cancellato oppure `Z` nel caso di un file compresso.
     - `user` indica il nome dell'utente proprietario del file.
     - `date` indica la data di cancellazione e/o compressione del file (il comando `date` fornisce su standard output la data corrente in formato aaaa-mm-gg, e.g. 2017-02-27).
     - `originalSize` (e `currentSize`) indicano la dimensione (in bytes) dei file prima (e dopo) la cancellazione (compressione).
     - `path` è il percorso assoluto del file rimosso (o compresso).

   Per comprimere il file si utilizzi il comando `gzip <nomeFile>` che produce una versione del file con stesso nome e ulteriore estensione `gz`. Si ricorda che il comando `ls -l` produce in output righe del tipo:

   `permissions num_links owner_group owner_user bytesize last_access name`

```
1  #!/bin/bash
2
3  # Check correct number of parameters
4  if [ $# -ne 3 ]; then
5      echo "Usage: es4.sh user numMbytes logfile"
6      exit 1
7  fi
8
9  # Set user home directory
10 user_home="/home/$1/"
11
12 # Scan files with size greater than $2 Mbytes
13 for file in $(find $user_home -size "+$2M"); do
14
15     # Ask user action
16     echo "$file"
17     echo -n "Type D (delete) or Z (zip): "
18     read action
19
20     # Gather file info
21     owner=$(ls -l $file | cut -d " " -f 4)
22     original_size=$(ls -l $file | cut -d " " -f 5)
23     data=$(date +"%Y-%m-%d")
24
25     # Perform requested action with logging
26     if [ $action == "D" ]; then
27         rm $file
28         current_size=0
29         echo "D $owner $data $original_size $current_size $file" >> $3
30     elif [ $action == "Z" ]; then
31         gzip $file
32         current_size=$(ls -l "$file.gz" | cut -d " " -f 5)
33         echo "Z $owner $data $original_size $current_size $file" >> $3
34     fi
35 done
```

5. Scrivere uno script AWK che processi un file con lo stesso formato descritto nell'esercizio precedente, ovvero costituito da righe del tipo:

   ```
   type user date originalSize currentSize path
   ```

   Lo script deve calcolare la quantità di spazio su disco risparmiata in totale e per i singoli utenti. In particolare, per ogni utente, lo script deve fornire in output la quantità di spazio liberato (in Mbytes) mediante cancellazione, la quantità di spazio liberato (in Mbytes) mediante compressione e la quantità di spazio liberata (in Mbytes) totale. Inoltre lo script deve stampare le quantità di spazio liberate a livello di intero sistema, considerando la cancellazione e la compressione dei file di tutti gli utenti. Ad esempio, dato il seguente file

   ```
   Z gianni 2017-01-16  739239936 89229201 /home/gianni/Video/vid.avi
   D gianni 2017-01-16 1468033024        0 /home/gianni/dvd.iso
   D paolo  2017-01-18 2990078103        0 /home/paolo/folder/archive.zip
   Z paolo  2017-01-18  693704050 46246936 /home/paolo/Video/vid.mp4
   ```

   lo script deve fornire un output del tipo:

   ```
   gianni:   deleted=1400  compressed= 619  total=2019
   paolo:    deleted=2851  compressed= 617  total=3468
   overall:  deleted=4251  compressed=1236  total=5487
   ```

```
1   {
2     users[$2] = $2
3     if($1 == "D"){
4       deleted[$2] += $4/(1024*1024);
5     } else if($1 == "Z") {
6       compressed[$2] += ($4 - $5)/(1024*1024);
7     }
8   }
9
10  END {
11    totdel = 0;
12    totcomp = 0;
13    for(user in users){
14      printf ("%s:\t\tdeleted=%d\tcompressed=%d\ttotal=%d\n",
15              user, deleted[user], compressed[user], deleted[user] + compressed[user]);
16      totdel += deleted[user];
17      totcomp += compressed[user];
18    }
19    printf ("overall:\tdeleted=%d\tcompressed=%d\ttotal=%d\n",
20            totdel, totcomp, totdel + totcomp);
21  }
```

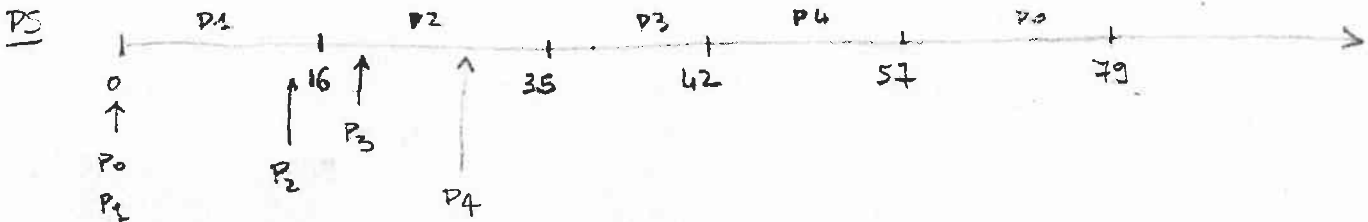
6. Si consideri il seguente insieme di processi:

Processo	Tempo arrivo	Burst Time	Priorità
P_0	0	22	5
P_1	0	16	2
P_2	15	19	4
P_3	17	7	1
P_4	25	15	3

Rappresentare mediante diagramma di Gantt l'esecuzione di tali processi utilizzando gli algoritmi di scheduling PS (Priority Scheduling), RR (Round Robin) e SRTF (Shortest Remaining Time First). Calcolare inoltre il tempo di attesa medio per ciascun algoritmo.

Si osservi che la priorità maggiore è associata al valore di priorità inferiore e che per la strategia Round Robin il quantum temporale è di 10 unità di tempo.

PS

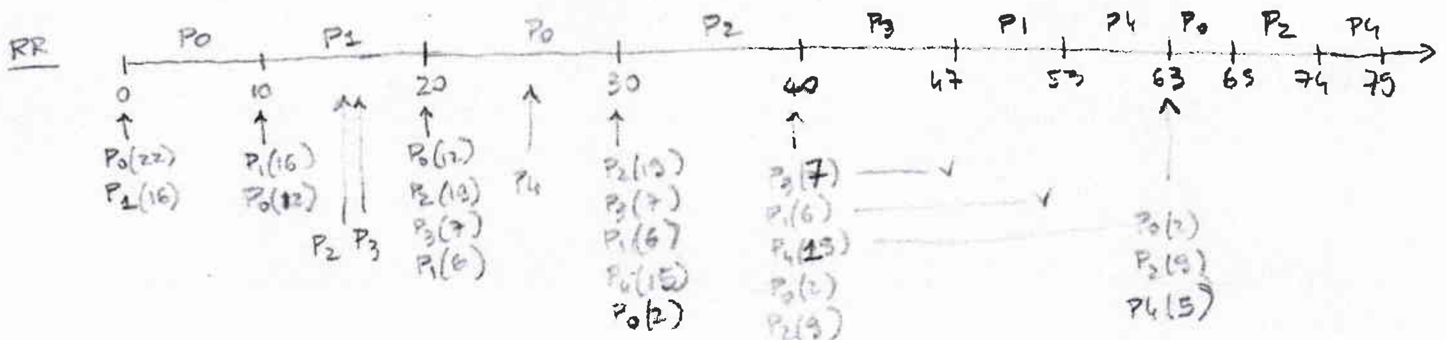


Tempi Attesa:

P0 57 − 0 = 57
P1 0 − 0 = 0
P2 16 − 15 = 1
P3 35 − 17 = 18
P4 42 − 25 = 17

$\overline{T}_a = \frac{57+0+1+18+17}{5} = \frac{93}{5} = 18.6$

RR

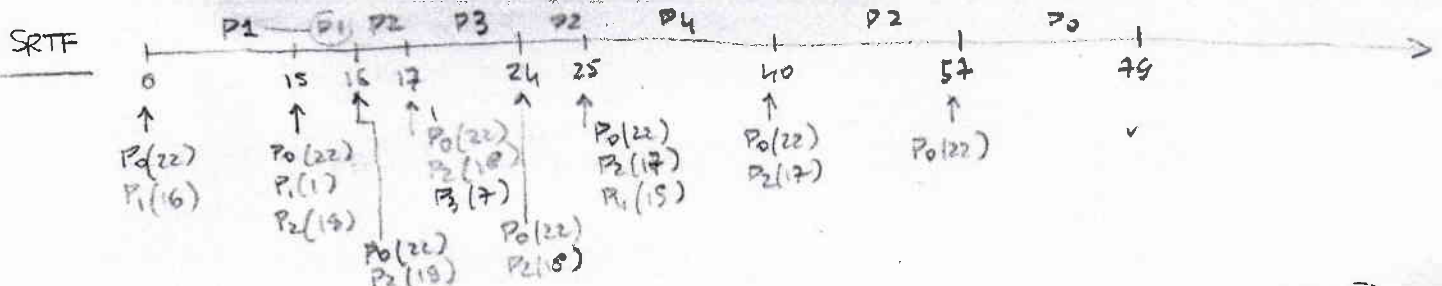


Tempi Attesa:

P0 (0−0)+(20−10)+(63−30) = 43
P1 (10−0)+(47−20) = 37
P2 (30−15)+(65−40) = 40
P3 (40−17) = 23
P4 (53−25)+(74−63) = 39

$\overline{T}_a = \frac{43+37+40+23+39}{5} = \frac{182}{5} = 36.4$

SRTF



Tempi Attesa:

P0 57 − 0 = 57
P1 (0 − 0) + (15 − 15) = 0
P2 (16 − 15) + (24 − 17) + (40 − 25) = 23
P3 (17 − 17) = 0
P4 (25 − 25) = 0

$\overline{T}_a = \frac{57+0+23+0+0}{5} = \frac{80}{5} = 16.0$