Caselle riservate

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

Sistemi Operativi

Compito d'esame

03 Settembre 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 descriva l'utilizzo dei segnali nel sistema operativo Unix/Linux con relativi vantaggi e svantaggi. Si descrivano in particolare le system call `signal`, `kill`, `pause` e `alarm`, riportando un esempio completo di utilizzo delle prime due.

   Un segnale è un interrupt software, ovvero un evento di sistema inviato a un processo. I segnali permettono di gestire eventi asincroni ma non sono degli interrupt perchè gli interrupt sono inviati dall'hardware al sistema operativo mentre 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.).

   Sono generati quando il processo sorgente effettua l'evento necessario e consegnati quando il processo destinatario assume le azioni richieste dal segnale. Un segnale non consegnato risulta pendente. Un segnale ha un tempo di vita che va dalla sua generazione alla sua consegna. I sengali possono venire utilizzati per la comunicazione tra i processi. Quando un segnale si presenta é possibile ignorare esplicitamente il segnale, lasciare che si verifichi il comportamento di default, catturare il segnale.

   - `void (*signal (int sig, void (*func)(int)))(int);` consente di settare il signal handler `func` alla ricezione del segnale `sig`.
   - `int kill (pid_t pid, int sig);` invia un segnale a un processo o a un gruppo di processi.
   - `int pause (void);` sospende il processo chiamante sino all'arrivo di un signale.
   - `unsigned int alarm (unsigned int seconds);` alarm invia SIGALARM dopo `seconds` second.

   Esempio di utilizzo:

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

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

```
main () {
  pid_t pid;
  pid = fork();
  switch (pid) {
    case -1: /* error /*
      printf ("fork failed");
      exit (1);
    case 0: /* child */
      sleep(5);
      kill (getppid(), SIGALRM);
      exit(0);
  }
  /* father */
  (void) signal (SIGALRM, myAlarm);
  pause ();
  exit (0);
}
```

2. Si riportino tre possibili soluzioni software al problema delle sezioni critiche con due processi denominati P_i e P_j, motivandole l'erroneità oppure la correttezza nonché vantaggi e svantaggi.

Progresso non assicurato: P_i e P_j devono entrare nella SC ma in maniera alternata. `turn` inizializzata a 0 o 1.

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

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

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

La soluzione con attesa non definita (P_i e P_j possono rimanere bloccati per sempre) è simile.

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

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

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

3. Si definisca che cosa si intende per "stato sicuro" indicandone i concetti e gli algoritmi principali. Si riporti inoltre il grafico planare di processo congiunto nel caso di due processi che possano o meno terminare in una condizione di deaclock. Se ne illustri il significato e l'utilizzo.

   I principali algoritmi di prevenzione si basano sul concetto di stato sicuro.

   Uno stato si dice sicuro se:

   - il sistema è in grado di allocare risorse e impedire il verificarsi di uno stallo.
   - esiste una sequenza sicura, ovvero una sequenza di processi $P_1, P_2, \ldots, P_n$ tale che per ogni P_i le richieste che esso può ancora effettuare possono essere soddisfatte impiegando le risorse attualmente disponibili più le risorse liberate dai processi P_j con $j < i$.

   Uno stato si dice non sicuro in caso contrario.

   Algoritmicamente con risorse aventi istanze unitarie si può lavorare sul grafo di assegnazione delle risorse, mentre coon risorse aventi istanze multiple si utilizza l'algoritmo del banchiere.

   Eventuale breve descrizione dell'algoritmo del banchiere ...

   Soprattutto grafico del progresso congiunto di due processi e relativa spiegazione (vedere il testo Andrew S. Tanenbaum oppure i lucidi).

4. Si scriva un tratto di codice C che, ricevuto un valore intero n sulla riga di comando, sia in grado di generare esattamente n figli (numerati da 0 a $n-1$) tali che:
   - i figli "pari" (0, 2, 4, etc.) sostituiscano la loro immagine con quella di un processo analogo a quello iniziale ma con parametro uguale a $n-1$
   - i figli "dispari" (1, 3, 5, etc.) visualizzino a video il loro numero d'ordine (1, 3, 5, etc.) seguito dal loro process identifier utilizzando la funzione `system`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main (
  int argc,
  char *argv[]
  )
{
  char str[100];
  int pid, i;
  fprintf (stdout, "Proc pid=%d running with parameter %s\n", getpid (), argv[1]);
  fflush (stdout);
  for (i=0; i<atoi(argv[1]); i++) {
    pid = fork ();
    if (pid == 0) {
      /* child */
      if (i%2 == 0) {
        fprintf (stdout, "even - run <pgrm> %d\n", i-1);
        fflush (stdout);
        sprintf (str, "%d", i-1);
        if (execlp ("./a", "a", str, (char *) 0) < 0)
          printf ("error exec\n");
      } else {
        fprintf (stdout, "odd - run echo pid=%d - i=%d\n", getpid(), i);
        fflush (stdout);
        sprintf (str, "echo odd - pid=%d - i=%d", getpid (), i);
        system (str);
      }
      exit (1);
    }
  }
 return 0;
}
```

5. Realizzare uno script di shell che utilizzando SED esegua quanto segue.

   Lo script riceve una stringa `str` sulla riga di comando e quindi ricerca nell'albero di direttori di nome `str` tutti i file aventi dimensione minore di 100Kbyte che contengono almeno una riga che incomincia con la stringa "BEGIN" oppure termina con la stringa "END".

   Lo script visualizzi (a video) tutte le righe dei file con tali caratteristiche, private delle stringhe "BEGIN" e "END".

```
#!/bin/bash
if [ $# -ne 1 ]
then
  echo "usage $0 dirsName"
  exit 0
fi
for d in ‘find . -type d -name "$1"‘
do
  find $d -type f -size -100k -exec sed -n -e ’/^BEGIN /p’ -e ’/END$/p’ ’{}’ \; | sed -e ’s/BEGIN //g’ -e "s/END//g"
done
```

6. Un file contiene un testo che include solo stringhe alfanumeriche (parole) e numeri interi.

   Scrivere uno script AWK che, ricevuto il nome di tale file sulla riga di comando, visualizzi su standard output tutti i numeri contenuti nel file, nonchè il valore minore e maggiore tra di essi e la loro media aritmetica.

```
#!/usr/bin/awk -f
BEGIN {
  sum = 0;
  count = 0;
}
{
  for (i=1; i<=NF; i++) {
    if (match($i, "[0-9]+")) {
      sum = sum + $i;
      count++;
      if (count==1) {
        min = max = $i;
      } else {
        if ($i>max)
          max = $i;
        if ($i<min)
          min = $i;
      }
      print $i
    }
  }
}
END {
  if (count > 0) {
    average = sum/count;
    print "average " average;
    print "min " min;
    print "max " max;
  }
}
```