Caselle riservate

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

Sistemi Operativi

Compito d'esame

19 Giugno 2014

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 la struttura denominata `stat` per una directory entry. Si scriva inoltre un programma, in linguaggio C, in grado di ricevere due parametri sulla riga di comando e di copiare l'albero di direttori specificato dal primo parametro nell'albero di direttori specificato dal secondo parametro.

```
#include <sys/types.h>
#include <sys/stat.h>

int stat (const char *path, struct stat *sb);
int lstat (const char *path, struct stat *sb);
int fstat (int fd, struct stat *sb);
```

La funzione `stat` restituisce la struttura `stat` per il file (o riferimento) passato come parametro. Le funzioni `lstat` e `fstat` restituiscono informazioni simili. I valori di ritorno sono: 0 se l'operazione ha avuto successo, -1 se cè un errore.

```
struct stat {
mode_t st_mode;
ino_t st_ino;
dev_t st_dev;
dev_t st_rdev;
...
};
```

Il secondo argomento di `stat` è il puntatore alla struttura. Il campo `st_mode` codifica il tipo di file. Alcune macro permettono di capire di quale file si tratti: `S_ISREG` regular file, `S_ISDIR` directory, `S_ISBLK` block special file, `S_ISCHR` character special file, `S_ISFIFO` FIFO, `S_ISSOCK` socket, `S_ISLNK` symbolic link.

```c
void
visitDirRecur (
  char *fullnameR,
  char *fullnameW,
  int level
  )
{
  DIR *dp;
  struct stat statbuf;
  struct dirent *dirp;
  char nameR[N], nameW[N];
  fprintf (stdout, "Entering %s; level %d\n", fullnameR, level);
  if (lstat(fullnameR, &statbuf) < 0 ) {
    fprintf (stderr, "Error Running lstat.\n");
    exit (1);
  }
  if (S_ISDIR(statbuf.st_mode) == 0) {
    return;
  }
  /* Create New Directory */
  fprintf (stdout, "Creating %s; level %d\n", fullnameW, level);
  mkdir (fullnameW, statbuf.st_mode);
  /* Visit Old Directory */
  if ( (dp = opendir(fullnameR)) == NULL) {
    fprintf (stderr, "Error Opening Dir.\n");
    exit (1);
  }
  while ( (dirp = readdir(dp)) != NULL) {
    sprintf (nameR, "%s/%s", fullnameR, dirp->d_name);
    sprintf (nameW, "%s/%s", fullnameW, dirp->d_name);
    if (lstat(nameR, &statbuf) < 0 ) {
      fprintf (stderr, "Error Running lstat.\n");
      exit (1);
    }
    if (S_ISDIR(statbuf.st_mode) == 0) {
      /* File */
      fprintf (stdout, "Reading %s; level %d\n", nameR, level+1);
      fprintf (stdout, "Copying %s to %s; level %d\n",
        nameR, nameW, level+1);
      myCopy (nameR, nameW, statbuf.st_mode);
    } else {
      /* Directory */
      if (strcmp(dirp->d_name, ".") == 0  ||
        strcmp(dirp->d_name, "..") == 0)
        continue;
      visitDirRecur (nameR, nameW, level+1);
    }
  }
  if (closedir(dp) < 0) {
    fprintf (stderr, "Error.\n");
    exit (1);
  }
  return;
}
void
myCopy (
  char *nameR,
  char *nameW,
  mode_t mode
  )
{
  int nR, nW, fdR, fdW;
  char buf[BUFFSIZE];
  fdR = open (nameR, O_RDONLY);
  // fdW = open (nameW, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  fdW = open (nameW, mode | O_WRONLY | O_CREAT);
  if (fdR==(-1) || fdW==(-1)) {
    fprintf (stdout, "File Open Error (R=%d)(W=%d).\n", fdR, fdW);
    exit (1);
  }
  while ((nR = read (fdR, buf, BUFFSIZE)) > 0) {
    /* Write on stdout */
    /* write (1, buf, nR); */
    nW = write (fdW, buf, nR);
    if (nR != nW)
      fprintf (stderr, "Write Error (read %d, write %d).\n", nR, nW);
  }
  if (nW < 0)
    fprintf (stderr, "Write Error.\n");
  close (fdR);
  close (fdW);
  return;
}
```

2. 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. Indicare le differenze tra `alarm` e `sleep`.

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.

Una race condition avviene quando il comportamento di più processi che lavorano su dati comuni dipende dallordine di esecuzione. La segnalazione tra processi può generare race conditions che portano il programma a non funzionare nel modo desiderato.

```
#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>
int pause (void);
```

Sospende il processo chiamante sino all'arrivo di un signale. Ritorna solo quando viene eseguito un gestore di segnali e al termine della sua esecuzione questo ritorna. Nel caso ritorno la funzione restituisce valore -1.

```
#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.

La system call `signal (SIGCHLD, SIG_IGN)` permette a un processo di ignorare (costante `SIG_IGN`) i segnali `SIGCHLD` inviati dai figli che terminano. Questo è anche il comportamento di default. Se un processo vuole essere notificato dalla terminazione dei figli (ed evitare che diventino "zombie") deve utilizzare la system call `wait`.

La `alarm` genera un `SIGALRM` dopo un certo tempo. La `sleep` dorme per un certo tempo o sino a catturare un segnale, ma non lo genera.

Esempio ...

3. Si indichino le principali differenze tra *processi* e *thread*. Si descriva la gestione di tali entità da parte del sistema operativo (struttura in memoria, etc.). Se ne indichino le caratteristiche principali e i relativi vantaggi e svantaggi.

   Si indichino inoltre le principali differenze tra thread a livello utente e thread a livello kernel, descrivendo i vari modelli di thread normalmente disponibili.

   Un processo (heavyweight process) corrisponde a un task con un solo thread.

   La specifica POSIX 1003.1c introduce il concetto di thread o processo "leggero" (lightweight process). Un thread è una sezione di un processo che viene schedulata e eseguita indipendentemente dal processo (thread) che lha generata. Un thread condivide con gli altri thread che appartengono allo stesso processo il codice, i dati (variabili, descrittori di file, etc.), le risorse del sistema operativo (e.g., segnali). I thread consentono di avere tempi di risposta ridotti (i.e., sganciare un thread è $10-100$ volte più veloce che generare un processo), condividere e economizzare le risorse, ottenere maggiore scalabilità.

   Si osservi che non esiste protezione tra i thread di uno stesso processo in quanto tutti sono eseguiti nello stesso spazio degli indirizzi.

   Esistono tre modelli di programmazione multi-thread:

   - Utilizzo di soli thread a livello utente (user-level thread). In questo caso il pacchetto dei thread è inserito completamente nello spazio dell'utente. Sono supportati mediante un insieme di chiamate, a livello utente, a librerie di sistema. Ogni processo ha una tabella personale dei thread in esecuzione. Il kernel non è a conoscenza dellesistenza dei thread e gestisce solo processi standard.
   - Utilizzo di soli thread a livello kernel (kernel-level thread). I thread sono gestiti dal kernel e le informazioni sui thread in esecuzione sono le stesse mantenute nel caso della gestione di thread utente. Uno degli svantaggi è avere un context switch più costoso. Inotre è limitato il numero massimo di thread per task/sistema.
   - Utilizzo di user e kernel thread (implementazione mista o ibrida). Limplementazione mista tenta di cobinare i vantaggi di entrambi gli approcci. Lutente decide quanti thread utente eseguire e su quanti thread kernel mapparli. Il kernel è a conoscenza solo dei thread kernel e gestisce solo tali thread. Ogni thread kernel può essere utilizzato a turno da diversi thread utente.

4. Si illustri il significato e l'utilizzo dei "grafi di allocazione delle risorse" nel caso di risorse con istanze semplici e multiple. Se ne esemplifichi l'utilizzo in presenza e in assenza di deadlock (si richiedono almeno due esempi).

   Si indichi inoltre che cosa si intende per "grafo di attesa" e per "arco di reclamo" e per quali motivi sono stati introdotti.

   Vedere lucidi e relative spiegazioni oppure i testi consigliati.

5. Uno script di shell riceve sulla riga di comando due parametri: il nome di un file e quello di un direttorio. Il file include un insieme di righe ciascuna contenente due stringhe, il nome di un direttorio e il nome di un file, separate da spazi. Il seguente è un esempio corretto di file:

   ```
   /home/bin/ dos2unix.sh
   /usr/bin/ old.c
   /usr/bin/ new.c
   ...
   ```

   Lo script deve ricercare tra i file elencati (e.g., `/home/bin/dos2unix.sh`, `/usr/bin/old.c`, etc.) quelli che contengono la stringa `main` (si osservi che la stringa `main` deve essere presente nel contenuto del file *non* nel suo nome) e per ognuno di questi:

   - Chiedere all'utente se vuole effettuarne una copia.
   - Leggere la risposta dell'utente da tastiera (del tipo `si/no`).
   - In caso affermativo, copiare tale file nel direttorio specificato sulla riga di comando. In caso negativo, procedere con il file successivo.

   Si osservi che all'interno di un ciclo di lettura da file tramite comando `read`, una ulteriore lettura da tastiera con lo stesso comando, può essere effettuata come `read nomeVariabile </dev/tty`.

```
#!/bin/bash
while read line
do
  file_to_copy=$(echo $line | tr -d ' ')
  if [ -e $file_to_copy ]
  then
    cat $file_to_copy | grep "main"
    if [ $? -eq 0 ]
      then
        echo "Copy file "$file_to_copy" in "$2"?(yes/no)"
        read a </dev/tty
        if [ $a == "yes" ]
        then
          echo "copying $file_to_copy ......"
          cp $file_to_copy $2
        fi
      fi
  fi
done < $1
```

6. In un programma concorrente tutti i thread producono il proprio output sullo stesso file. Al testo memorizzato dal processo principale, ogni thread secondario aggiunge il proprio inserendo il testo tra parentesi graffe e il proprio identificatore di thread tra parentesi quadre. Il successivo è un esempio di output generato in tale contesto:

```
... testo 1 (processo principale) ...
[123]{testo prodotto dal thread 123}
... testo 2 (processo principale)
ancora il processo principale ...
[245]{testo prodotto dal thread 245}
... testo 3 (processo principale) ...
[123]{...}
```

Si desidera scrivere uno script AWK in grado di separare l'output dei vari thread da quello del processo principale. Il nome del file che occorre parsificare viene passato allo script sulla riga di comando. Il file di nome "tid.txt" contiene l'elenco dei thread e il nome dei file in cui il relativo output deve essere memorizzato, con il seguente formato:

```
tid1 fileThread1
tid2 fileThread2
...
```

dove `tid1`, `tid2`, etc., sono numeri interi (e.g., `123`, `245`, etc.). L'output del processo principale deve essere memorizzato in un file con lo stesso nome di quello di ingresso ma con estensione `''log''`.

Soluzione 1:

```
BEGIN {
  #reading tid.txt file
  while ((getline line < "tid.txt") > 0) {
    split(line,v," ")
    tid=v[1]
    filename=v[2]
    vett_tid[tid]=filename
  }
}
#parsing....
!/^\[[0-9]*\]/ {
  print $0 >> "log.txt"
}
/^\[[0-9]*\]/ {
  for (tid in vett_tid){
    if ($0 ~ tid)
      print $0 >> vett_tid[tid]
    }
}
```

Soluzione 2:

```
BEGIN {
  while(getline < "tid.txt"){
    tids[$1] = $2;
  }
}
{
  if(match($0, /\[([0-9]+)\]\{([^\n\}]*)\}/, res)){
    print res[2] > tids[res[1]]
  } else {
    print $0 > FILENAME".log"
  }
}
```