Laboratorio di Sistemi Operativi
7 luglio 2016
Compito

Si risponda ai seguenti quesiti, giustificando le risposte.

1. (a) Si specifichi il comando per creare un link **hard** di nome `link1` nella directory corrente al file `/home/pippo/documento.txt`
   (b) Si specifichi il comando per creare un link **simbolico** di nome `link2` nella directory corrente al file `/home/pippo/documento.txt`
   (c) A livello dell'implementazione nel filesystem, c'è differenza fra `link1` e `link2` creati nei due punti precedenti? Se sì, quale?
   (d) È possibile creare un link simbolico ad un file risiedente su un dispositivo fisico diverso da quello in cui risiede il link? Ed un link hard? Motivare le risposte.

Soluzione:

(a) ln /home/pippo/documento.txt link1

(b) ln −s /home/pippo/documento.txt link2

(c) Sì, c'è differenza fra `link1` e `link2` creati nei due punti precedenti: nel primo caso infatti si tratta di un alias per lo stesso numero di inode, ovvero, sia `documento.txt` che `link1` puntano allo stesso elemento dello stesso vettore di inode. Invece nel secondo caso `link2` punta ad un numero di inode diverso rispetto a quello puntato da `documento.txt`; infatti `link2` altro non è che un file di testo (trattato in modo speciale dal sistema operativo) contenente il pathname del file `documento.txt`.

(d) Sì, è possibile creare un link simbolico ad un file risiedente su un dispositivo fisico diverso da quello in cui risiede il link in quanto, come detto precedentemente, un link simbolico è un file di testo contenente un pathname. Quindi è sufficiente seguire il percorso per arrivare al file puntato. Al contrario non è possibile creare un link hard ad un file risiedente su un dispositivo fisico diverso da quello in cui risiede il link in quanto ogni dispositivo fisico ha un proprio vettore di inode (e non è possibile puntare ad inode di un dispositivo diverso da quello di origine).

2. Si scrivano i comandi necessari per risolvere ognuno dei task seguenti:
   (a) fornire il conteggio dei processi dell'utente `pippo`;
   (b) fornire la dimensione in byte (caratteri) del file `/etc/passwd` e memorizzarla nella variabile `len`;
   (c) estrarre il primo (user name) ed il terzo (user ID) campo dal file `/etc/passwd` e salvarli nel file `estratto.txt` in modo che i campi corrispondenti su ogni linea siano separati dal carattere underscore (_). Quindi l'output finale deve essere del tipo seguente:

```
root_0
daemon_1
bin_2
...
```

Soluzione:

(a) ps −u pippo −−no−headers | wc −l

(b) len=‘wc −c < /etc/passwd‘

(c)
```
cat /etc/passwd | cut -d: -f1 > f1.txt
cat /etc/passwd | cut -d: -f3 > f3.txt
paste -d_ f1.txt f3.txt > estratto.txt
```

3. Si scriva il codice di uno **script della shell** che prenda in input come parametro della linea di comando il percorso di un file e ne stampi a video la linea più lunga.

   Ad esempio (se `longest` è il nome dello script eseguibile):

```
> ./longest /etc/passwd
```

deve produrre in output la linea più lunga del file `/etc/passwd`. Si gestiscano gli eventuali errori (numero di argomenti errato, file non leggibile).

Soluzione:

Esempio di soluzione:

```
if test $# -ne 1
then
    echo "utilizzo: $0 <file>"
    exit 1
fi

if ! test -e $1 -a -r $1
then
    echo "il file $1 non e' accessibile"
    exit 2
fi

linee=`wc -l < $1`
i=0
max=0
lunga=''

while test $i -lt $linee
do
    linea_corrente=`cat $1 | tail -n +$i | head -1`
    num_car=`echo $linea_corrente | wc -c`
    if test $num_car -gt $max
    then
        max=$num_car
        lunga=$linea_corrente
    fi
    i=$[$i+1]
done

echo $lunga
```

4. Si consideri il seguente programma `logger_server.c` (per semplicità sono state omesse le direttive di inclusione) che implementa un server multithread che accoda nel file `log.txt` i messaggi inviati dai client a lui connessi. Ogni connessione viene gestita da un nuovo thread che esegue la funzione `logger()`, dove avviene la ricezione del messaggio ed il suo accodamento nel file `log.txt`.

```
#define SERVER_PORT 8888  /* porta di ascolto del server */
#define LINESIZE 255      /* dimensione dei buffer */

struct channel {
  int fd;  /* file descriptor del canale di comunicazione */
  int num; /* numero intero identificante il thread */
};

void *logger(void *c) {
  char inputline[LINESIZE];
  char buffer[2*LINESIZE];
  int  len;
  int fd=((struct channel *)c)->fd;   /* recupero il socket file descriptor */
  int num=((struct channel *)c)->num; /* recupero il numero del thread */
  ...     /* se necessario, è possibile aggiungere altre variabili */

  while ((len = recv(fd, inputline, LINESIZE-1, 0)) > 0) {
    ...  /* completare (1) */
  }
```

```
20    close(fd);    /* chiudo la connessione con il client */
21  }
22
23  int main() {
24    int sock, client_len, fd, i=0;
25    struct sockaddr_in server, client;
26    pthread_t t;
27    struct channel ch;
28
29    if((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
30      perror("chiamata alla system call socket fallita");
31      exit(1);
32    }
33    server.sin_family = ...;                                  /* completare (2) */
34    server.sin_addr.s_addr = ...;                             /* completare (3) */
35    server.sin_port = ...;                                    /* completare (4) */
36
37    /* binding dell'indirizzo al transport end point */
38    if (bind(...) == -1) {                                    /* completare (5) */
39      perror("chiamata alla system call bind fallita");
40      exit(2);
41    }
42
43    /* impostiamo il server in modo che possa gestire 15 richieste
44       contemporaneamente */
45    ...;                                                      /* completare (6) */
46
47    while (1) {
48      client_len = sizeof(client);
49      if ((fd = accept(sock, (struct sockaddr *)&client, &client_len)) < 0) {
50        perror("accepting connection");
51        exit(3);
52      }
53
54      /* ogni volta che il server accetta una nuova connessione,
55       * quest'ultima viene gestita da un nuovo thread
56       */
57      ch.fd=fd;
58      ch.num=++i;
59
60      if(pthread_create(&t,NULL,logger,(void *)&ch)!=0) {
61        fprintf(stderr,"Errore nella creazione del thread n. %d!",i);
62        return 1;
63      }
64    }
65  }
```

(a) Si completi il codice relativo alle system call delle socket (dove compaiono i puntini).

(b) Si completi il codice della funzione `logger()` in modo che la linea accodata nel file `log.txt` abbia il formato seguente:

`Thread n. i: <stringa>`

dove `i` è l'intero identificante il thread (memorizzato nel campo `num` di `struct channel`) e `<stringa>` è il messaggio ricevuto dal client (si faccia attenzione al fatto che `recv` **non** mette nel buffer in cui scrive il carattere terminatore: va messo a mano).

(c) Una volta completato il punto precedente, si modifichi la funzione `logger()` ed eventualmente il programma in generale (dove necessario) per garantire che il server non registri in modo disordinato i messaggi provenienti dai client, ma faccia in modo che le linee di ogni client appaiano contigue. Per esempio, nel caso di tre client connessi, il file `log.txt` non deve contenere una sequenza del genere:

```
Thread n. 1: abc
Thread n. 2: bla bla bla
Thread n. 1: def
Thread n. 3: 1111
Thread n. 1: ghi
thread n. 3: 2222
```

Deve invece fare in modo che la successione sia la seguente:

```
Thread n. 1: abc
Thread n. 1: def
Thread n. 1: ghi
Thread n. 2: bla bla bla
Thread n. 3: 1111
thread n. 3: 2222
```

facendo eventualmente attendere gli altri thread, mentre il thread correntemente in esecuzione sta scrivendo nel file.

Soluzione:

(a) Codice relativo alle system call delle socket:

Linea 33: server.sin_family = AF_INET;

Linea 34: server.sin_addr.s_addr = inet_addr("127.0.0.1");
oppure
server.sin_addr.s_addr = INADDR_ANY;

Linea 35: server.sin_port = htons(SERVER_PORT);

Linea 38: **if** (bind(sock, (**struct** sockaddr *)&server, **sizeof** server) == −1){

Linea 45: listen(sock, 15);

(b) Codice della funzione logger:

```
1  void *logger(void *c) {
2    char inputline[LINESIZE];
3    char buffer[2*LINESIZE];
4    int  len;
5    int fd=((struct channel *)c)->fd;
6    int num=((struct channel *)c)->num;
7    FILE *logfd;
8
9    while ((len = recv(fd, inputline, LINESIZE-1, 0)) > 0) {
10     inputline[len]='\0';
11     sprintf(buffer,"Thread n. %d: ",num);
12     strcat(buffer,inputline);
13     logfd=fopen("log.txt","a");    /* apro il file log.txt in append */
14     fputs(buffer,logfd);           /* scrivo nel file */
15     fclose(logfd);                 /* chiudo il file log.txt */
16   }
17
18   close(fd);
19 }
```

(c) Vi sono vari modi di risolvere il problema. Ad esempio si può fare in modo che il padre attenda la terminazione di ogni thread (mediante una `pthread_join()`) prima di creare il successivo. Oppure è possibile aggiungere un mutex, dichiarandolo ed inizializzandolo dopo le direttive `#define`:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

ed utilizzarlo per realizzare una sezione critica all'interno della funzione `logger()`:

```
1  void *logger(void *c) {
2    char inputline[LINESIZE];
3    char buffer[2*LINESIZE];
4    int  len;
5    int fd=((struct channel *)c)->fd;
6    int num=((struct channel *)c)->num;
7    FILE *logfd;
8
9    pthread_mutex_lock(&mutex);        /* inizio sezione critica */
```

```
10
11    while ((len = recv(fd, inputline, LINESIZE-1, 0)) > 0) {
12      inputline[len]='\0';
13      sprintf(buffer,"Thread n. %d: ",num);
14      strcat(buffer,inputline);
15      logfd=fopen("log.txt","a");
16      fputs(buffer,logfd);
17      fclose(logfd);
18    }
19
20    close(fd);
21    pthread_mutex_unlock(&mutex);       /* fine sezione critica */
22  }
```