Laboratorio di Sistemi Operativi
19 giugno 2014
Compito

Si risponda ai seguenti quesiti, giustificando le risposte.

1. (3 punti) Si consideri il seguente output prodotto dal comando `ls -l` eseguito nella home directory dell'utente `pippo`:

   ```
   ...
   -rwxr--r--   1 pippo    users    1045 Jan 25 18:05 f1
   ...
   ```

   Si supponga che l'utente `pippo` esegua il comando `chmod go+x ~/f1`. Quali saranno i nuovi permessi del file `f1`? Se invece del comando precedente l'utente `pippo` eseguisse il comando `chmod go=x ~/f1`, quali sarebbero i nuovi permessi del file `f1`?

   Soluzione:

   Nel caso in cui l'utente `pippo` esegua il comando `chmod go+x ~/f1`, ai permessi correnti del gruppo e del resto degli utenti verrà aggiunto il permesso di esecuzione. Quindi i permessi finali del file `f1` saranno `rwxr-xr-x`.

   Se invece l'utente `pippo` eseguisse il comando `chmod go=x ~/f1`, l'operatore `=` avrebbe un effetto diverso da `+` in quanto assegnerebbe *esattamente* e *solamente* i permessi specificati (in questo caso al gruppo ed al resto degli utenti). Quindi i nuovi permessi del file `f1` sarebbero `rwx--x--x`.

2. (3 punti) Scrivere un unico comando (facendo uso degli opportuni operatori di combinazione di comandi) che compili il programma C contenuto nel file `prog.c` nella directory corrente e ne lanci in esecuzione l'eseguibile prodotto soltanto in caso di successo della fase di compilazione.

   Soluzione:

   Un modo per ottenere l'effetto richiesto è quello di utilizzare l'operatore condizionale `&&` di combinazione dei comandi:

   ```
   gcc prog.c && ./a.out
   ```

   in tal modo il prodotto della compilazione `a.out` verrà lanciato soltanto se la compilazione di `prog.c` sarà andata a buon fine.

3. (3 punti) Qual è l'effetto dei seguenti comandi (nella sequenza fornita)?

   ```
   cd
   ps -ef > log.txt
   cat log.txt | grep '^pippo' | wc -l
   ```

   Soluzione:

   Il primo comando (`cd`) porta l'utente nella sua home directory. Il secondo comando della sequenza (`ps -ef > log.txt`) redirige l'output di `ps -ef` (ovvero il full listing di tutti i processi in esecuzione collegati o meno ad un terminale) nel file `log.txt` nella directory corrente (sostituendone il contenuto precedente nel caso in cui il file esistesse già). Infine il terzo comando (`cat log.txt | grep '^pippo' | wc -l`) legge il contenuto del file `log.txt` per darlo in pasto (tramite la pipeline) al filtro `grep` che passa in input (sempre tramite la pipeline) al comando `wc` soltanto le linee che iniziano con la stringa `pippo`; quest'ultimo (`wc`) conta il numero di tali linee e lo emette in output.

   L'effetto finale è che viene riportato sullo schermo del terminale il numero dei processi dell'utente `pippo` correntemente in esecuzione (infatti il full listing del comando `ps` produce come prima informazione su ogni linea il nome dell'utente che ha lanciato il processo).

4. (3 punti) Si predisponga uno script della shell che prenda in input sulla linea di comando un percorso di una directory e cancelli ricorsivamente a partire da quest'ultima tutti i file aventi un nome terminante in `.bak`. Si ignori la gestione degli eventuali errori.

Soluzione:

```
find $1 -name "*.bak" -type f -exec rm {} \;
```

5. (3 punti) Spiegare qual è l'effetto delle seguenti dichiarazioni in C:

   1. ```
      struct nodo_lista {
        int val;
        struct nodo_lista *prossimo;
      };
      ```
   2. `struct nodo_lista l1;`
   3. `struct nodo_lista *l2;`

Soluzione:

1. Viene dichiarata la struttura ricorsiva `struct nodo_lista` con un membro `val` di tipo `int` ed un membro `prossimo` di tipo puntatore alla struttura stessa.
2. Viene dichiarata la variabile `l1` di tipo `struct nodo_lista`.
3. Viene dichiarato il puntatore `l2` ad una struttura di tipo `struct nodo_lista`.

6. In base alle dichiarazioni dell'esercizio 5, dire se i seguenti frammenti di codice C sono corretti o errati (spiegando il motivo):
   (a) (1 punto) `l1.val=50;`
   (b) (1 punto) `l1->val=50;`
   (c) (1 punto)
   ```
   l2=(struct nodo_lista *)malloc(sizeof(struct nodo_lista));
   l2.val=20;
   ```
   (d) (1 punto)
   ```
   l2=(struct nodo_lista *)malloc(sizeof(struct nodo_lista));
   l2->val=20;
   ```
   (e) (1 punto)
   ```
   l2=(struct nodo_lista *)malloc(sizeof(struct nodo_lista));
   (*l2).val=20;
   ```

Soluzione:

(a) Corretto: il punto è l'operatore di selezione dei campi di una struttura.

(b) Errato: l'operatore `->` si può utilizzare soltanto con i puntatori ad una struttura.

(c) Errato: nonostante venga correttamente allocata la memoria per un elemento di tipo `struct nodo_lista` ed il relativo indirizzo venga assegnato a `l2`, essendo quest'ultimo un puntatore, non si può utilizzare direttamente il punto per selezionare il membro `val` della struttura puntata.

(d) Corretto: viene allocata la memoria per un elemento di tipo `struct nodo_lista` ed il relativo indirizzo viene assegnato a `l2`; quindi l'operatore `->` viene utilizzato correttamente per selezionare il membro `val` della struttura puntata ai fini dell'assegnamento.
   ```

(e) Corretto: viene allocata la memoria per un elemento di tipo `struct nodo_lista` ed il relativo indirizzo viene assegnato a `l2`; quindi l'operatore di deriferimento `*` viene utilizzato per accedere alla struttura puntata ed il punto permette di selezionarne correttamente il membro `val` per l'assegnamento.

7. (4 punti) Utilizzando la struttura `nodo_lista` dichiarata nell'esercizio 5, scrivere il codice C per allocare in memoria una lista formata da 3 nodi, il primo con valore (campo `val`) `100`, il secondo con valore `200` ed il terzo con valore `300`. Si memorizzi il puntatore al primo nodo della lista nella variabile `testa`.

Soluzione:

```
struct nodo_lista *testa;
testa=(struct nodo_lista *)malloc(sizeof(struct nodo_lista));
testa->val=100;
testa->prossimo=(struct nodo_lista *)malloc(sizeof(struct nodo_lista));
testa->prossimo->val=200;
testa->prossimo->prossimo=(struct nodo_lista *)malloc(sizeof(struct nodo_lista));
testa->prossimo->prossimo->val=300;
testa->prossimo->prossimo->prossimo=NULL;
```

8. Si consideri il seguente programma `triangle.c`:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i,j,n=atoi(argv[1]);
  for(i=0; i<n; i++) {
    for(j=0; j<i+1; j++)
      printf("*");
    printf("\n");
  }
}
```

Una volta compilato, l'eseguibile accetta un intero sulla linea di comando e stampa un triangolo di asterischi nel modo seguente:

```
> ./triangle 5
*
**
***
****
*****
```

Si vuole ottenere lo stesso effetto con un altro programma `triangle_fork.c` che genera `n` processi figli (dove `n` è il valore dell'intero passato sulla linea di comando) facendo stampare una linea ad ogni figlio:

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

void print_row(int l) { // funzione che stampa una linea di l asterischi
  int i;
```

```
  for(i=0;i<l;i++)
    printf("*");
  printf("\n");
}

int main(int argc, char *argv[]) {
  int i,j,n=atoi(argv[1]), *pid;
  pid=(int *)calloc(n,sizeof(int)); // alloco memoria per un array di n interi

  for(i=0; i<n; i++) {
    pid[i]=fork();          // creazione del figlio i-esimo
    switch(pid[i]) {
      case -1:              // fallimento della fork
        perror("fork failed!\n");
        exit(1);
      case 0:
        print_row(i+1);     // il figlio i-esimo stampa la sua riga...
        exit(0);            // ... e termina la sua esecuzione
    }
  }

  for(i=0; i<n; i++) {
    waitpid(pid[i],NULL,0); // attendo la terminazione del figlio i-esimo
  }

  free(pid);       // libero la memoria allocata per il vettore di interi
  return 0;
}
```

(a) (3 punti) il programma `triangle_fork.c` funzionerà correttamente, ovvero, produrrà in output il triangolo di asterischi? Perché?

(b) (5 punti) Se si è risposto negativamente alla precedente domanda, modificare il programma in modo che rispetti la consegna.

Soluzione:

(a) Il programma `triangle_fork.c` non è corretto, in quanto non è affatto detto che la sequenza di esecuzione dei processi figli sia la stessa della loro creazione. Quindi è possibile che le righe di asterischi vengano stampate fuori ordine.

(b) Ci sono vari modi per correggere il programma: il più semplice è quello di aggiungere il caso di default allo `switch` in questo modo:

```
...
default:
    waitpid(pid[i],NULL,0);
```

eliminando poi il ciclo `for` finale con i `waitpid`. In questo modo il padre (che eseguirà per l'appunto il codice relativo al caso di default) attenderà la terminazione del processo figlio i-esimo (che stamperà la sua linea di asterischi) prima di lanciare il processo figlio (i+1)-esimo.