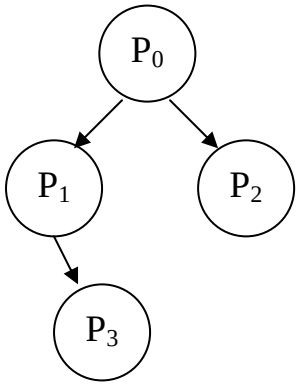
Sistemi operativi 2/ed
Paolo Ancilotti, Maurelio Boari, Anna Ciampolini, Giuseppe Lipari

SOLUZIONI DEI PROBLEMI DEL CAPITOLO 8

1 Nel programma compare una sequenza di due system call `fork()`. In particolare, la seconda `fork` viene invocata incondizionatamente; pertanto essa viene eseguita anche dal processo generato dalla prima `fork()`.
Quindi, in assenza di eccezioni, l'esecuzione del programma crea la gerarchia di 4 processi illustrata dal grafo seguente:



Infatti, l'esecuzione provoca i seguenti effetti:

- il processo iniziale P_0 con la prima `fork()` genera il processo figlio P_1;
- sia P_0 che P_1 eseguono la seconda `fork()`, creando rispettivamente i processi P_2 (figlio di P_0 e fratello di P_1) e P_3 (figlio di P_1 e nipote di P_0).

La condizione (`pid1==0`) viene quindi valutata da tutti i processi e risulterà soddisfatta soltanto per P_1 e P_3: il messaggio "ciao!" verrà quindi stampato 2 volte, poiché la prima `printf` verrà eseguita solo da P_1 e P_3. Diversamente, l'ultima `printf` verrà eseguita da tutti e quattro i processi e quindi il messaggio "addio!" verrà stampato 4 volte. E' importante osservare che l'ordine di visualizzazione dei messaggi dipende dallo scheduling effettivo dei processi.

2. Una possibile soluzione è la seguente:

```
#include  <stdio.h>

main(int argc, char *argv[])
{  int pid[3], status, i, p;
if (argc!=4)
{  printf("Sintassi sbagliata!\n");
   exit(1);
}

for (i=0; i<3; i++)
{  pid[i]=fork();
if (pid[i]==0)
   {execlp(argv[i+1], argv[i+1], (char *)0);
    printf("exec fallita!\n");
    exit(2);
}
else if (pid[i]<0)
```

```
        printf("fork fallita!\n");
}
 for (i=0; i<3; i++)
    p=wait(&status);
}
```

Dopo una verifica sulla corretta invocazione del comando, il processo iniziale P_0, esegue il ciclo di creazione dei 3 processi figli: ogni iterazione provoca la creazione (tramite `fork`) di un nuovo figlio `pid[i]` (`i`=0, ..2), che passa ad eseguire (tramite `execlp`) il comando specificato mediante l'argomento `argv[i+1]`.
Il ciclo successivo, eseguito soltanto da P_0 , attende la terminazione dei tre figli.

3. Il comando sfrutta i meccanismi di piping e ridirezione dello shell di Unix, descritti nell'appendice C.
Il primo comando della pipeline (`ls-l anna*`) produce come output l'elenco di tutti i file con relativi attributi contenuti del direttorio corrente il cui nome inizia con la stringa `anna`. Ad esempio:

```
-rw-r--r--  1 anna  anna  61 20 Mar 05:50 anna.old
-r--rw-r--  1 anna  anna  39 20 Mar 15:50 anna1
--wx--x--x  1 anna  anna  45 20 Mar 12:53 anna2
```

L'output prodotto da questo comando viene ridiretto nel canale di standard input del secondo comando della pipeline (`grep rw`) che seleziona, all'interno dello stream di input, le linee che contengono la stringa "rw". Facendo riferimento all'esempio di output precedente, il risultato di questo comando sarebbe quindi:

```
-rw-r--r--  1 anna  anna  61 20 Mar 05:50 anna.old
-r--rw-r--  1 anna  anna  39 20 Mar 15:50 anna1
```

Poiché l'output del secondo comando è ridiretto sul file `ris`, il risultato finale ottenuto dall'esecuzione della sequenza di comandi data è la scrittura tale file dell'output prodotto dal secondo comando.

La soluzione proposta richiede la conoscenza dei contenuti dell'appendice C:

```
#! /bin/bash

while true
do
     sleep 5s
     ls $1
     if  ! test -f $1
     then
          echo OK!
     else
          echo file $1 cancellato
          exit
     fi
```

```
done
```

Il file comandi ha una struttura ciclica, costituita da un ciclo `while`. La temporizzazione richiesta viene ottenuta mediante l'invocazione del comando `sleep`.
Per verificare la presenza del file ($1) dato come unico argomento, si usa il comando `test` con opzione `–f`.
Nel caso in cui il file dato non esista, viene eseguito il ramo else del comando `if`, che provoca la stampa del messaggio di errore e la terminazione forzata (mediante `exit`) del processo.

5. Una possibile soluzione è la seguente:

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

#define MAXT 10


void *thread_Giocatore(void * arg) /*codice del giocatore*/
{    int id, risultato;
     id=*((int )arg); /* id è l'identificatore del thread */
     risultato=rand()%6+1; /* lancio del dado */
     printf("Giocatore %d: risultato ottenuto %d !\n", id,
risultato);
     pthread_exit((void *)risultato);
}

main ()
{
  pthread_t T[10];
  int A[MAXT], i, ris, max=0;


srand(getpid());

/* Ciclo di creazione  dei thread: */
for (i=0; i<MAXT; i++)
{    A[i]=i;
     pthread_create (&T[i], NULL, thread_Giocatore, &A[i]);
}

for (i=0; i<MAXT; i++)
{   pthread_join(T[i], (void *)&A[i]);
    printf("il figlio %d e` morto restituendo %d\n", i,
A[i]);
    max=(A[i]>A[max]?i:max);
}

printf("\n\nIl Giocatore vincente e` il n. %d con risultato
%d\n\n", max, A[max]);
```

```
    return 0;
}
```

Il croupier è rappresentato dal thread iniziale (che esegue la funzione `main`), il quale crea i thread giocatori, passando ad ognuno come argomento il proprio identificatore (un intero da 0 a 9) tramite il quarto parametro della `pthread_create`.
Ogni giocatore lancia il dado ottenendo un valore aleatorio ottenuto con la funzione standard `rand()`, e termina restituendo tale valore al croupier attraverso il paramentro della `pthread_exit`.
Quando tutti i giocatori sono terminati, il thread croupier, avendo raccolto i risultati con `pthread_join`, può finalmente dichiarare, tramite l'ultima printf, il vincitore.