Università Ca' Foscari Venezia - Corso di Laurea in Informatica
Sistemi Operativi (modulo 1) – prof. Augusto Celentano

La gestione dei processi – Esercizi e temi d'esame risolti

1) Si consideri l'esecuzione di questo frammento di programma C nel sistema operativo Unix, supponendo che il processo che lo esegue abbia *process-id* uguale a 999 e che i processi creati successivamente abbiano *process-id* consecutivi e crescenti:

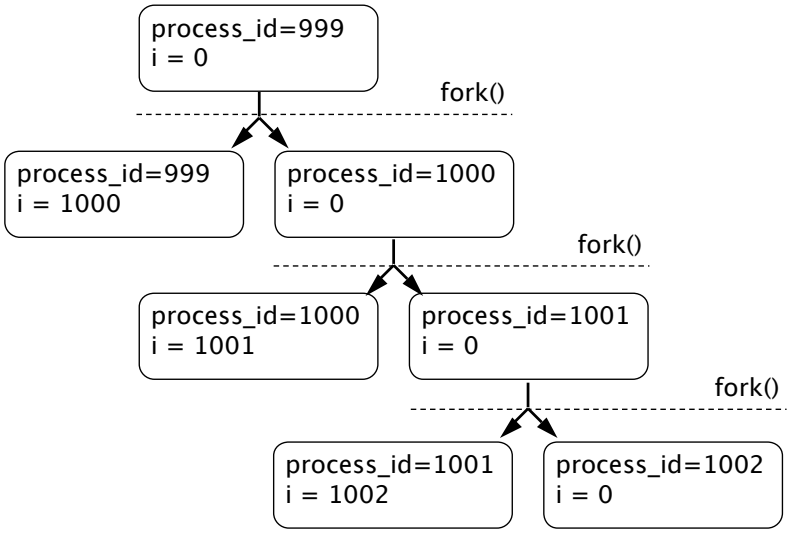
```
...
for (i = 0; i < 10; i++)
    i = fork();
...
```

Dire come evolve il sistema durante l'esecuzione, cioè quanti processi vengono creati, da chi vengono creati, che *process_id* hanno, e come prosegue la loro esecuzione.

Soluzione

Alla prima esecuzione del ciclo il processo crea una copia cui viene assegnato *process-id* = 1000. Nel processo creante (processo *padre*) la funzione *fork* restituisce il valore 1000, che viene assegnato alla variabile *i*. La variabile *i* viene incrementata e il ciclo for termina poiché *i* non è < 10; il processo prosegue con l'elaborazione successiva al ciclo.

Il processo creato (processo figlio) riceve dalla fork il valore 0 e lo assegna alla variabile *i*. Dopo l'incremento si ripete il ciclo, riproducendo il comportamento appena descritto. Il processo esegue la fork, crea un processo con *process-id* = 1001, valore che viene assegnato alla variabile **i**, causando la terminazione del ciclo for. Il processo creato riprende il ciclo con *i* = 1, e così all'infinito, o fino ad esaurimento delle risorse del sistema se i processi non terminano immediatamente l'esecuzione.



2) Si consideri l'esecuzione di questo programma nel sistema operativo Unix, supponendo che il processo che lo esegue abbia *process-id* uguale a 1001 e che i processi creati successivamente abbiano *process-id* consecutivi e crescenti:

```
main()
{ int i, j;
  i = fork();  j = fork();
  if (i == j)
      visualizza sul terminale "Caso i = j: i, j = " seguito dal valore di i;
  else if (i > j)
      visualizza sul terminale "Caso i > j: i, j = " seguito dai valori di i e j;
  else
      visualizza sul terminale "Caso i < j: i, j = " seguito dai valori di i e j;
}
```

Illustrare come evolve il sistema durante l'esecuzione: quali processi sono creati, che *process-id* hanno, che gerarchie padre-figlio si stabiliscono e che cosa viene visualizzato sul terminale da parte di quale processo. Si assuma che non si verificano errori.

Nota. In genere l'ordine di esecuzione dei processi generati dalla system call *fork* non è definito a priori: si ipotizzi un ordine di esecuzione qualunque, purché plausibile, e si proceda nell'analisi dell'esecuzione in modo coerente.

Soluzione

Si identifichino i processi con i nomi A, B, etc., e sia A il processo iniziale. Si assume che l'ordine di esecuzione dei processi corrisponda con l'ordine dei *process-id*. Dopo l'istruzione i = fork() si hanno i seguenti processi:

A (processo iniziale):	process-id = 1001, i = 1002, j = indefinito
B (creato da A):	process-id = 1002, i = 0, j = indefinito

Dopo l'istruzione j = fork(), che viene eseguita da entrambi i processi, la situazione è la seguente:

A (processo iniziale):	process-id = 1001, i = 1002, j = 1003
B (creato da A):	process-id = 1002, i = 0, j = 1004
C (creato da A):	process-id = 1003, i = 1002, j = 0
D (creato da B):	process-id = 1004, i = 0, j = 0

Sul terminale vengono quindi visualizzate le seguenti informazioni:

processo A:	Caso i < j: i, j = 1002, 1003
processo B:	Caso i < j: i, j = 0, 1004
processo C:	Caso i > j: i, j = 1002, 0
processo D:	Caso i = j; i, j = 0

3) Si consideri l'esecuzione di questo frammento di programma nel sistema operativo Unix, supponendo che il processo che lo esegue abbia l'identificatore di processo 1000, che i processi creati successivamente abbiano identificatori consecutivi e crescenti e che non si verifichino errori (nota: getpid() è una system call che restituisce l'identificatore del processo che la invoca):

```
...
while (fork() == fork())
    printf("Processo %d nel ciclo while\n", getpid());
printf("Processo %d fuori dal ciclo while\n", getpid());
...
```

Illustrare l'evoluzione del sistema durante l'esecuzione: che processi vengono creati e con quali identificatori di processo, che gerarchie padre-figlio si stabiliscono, che cosa viene visualizzato sul terminale, e come prosegue la loro esecuzione, assumendo che il codice non dettagliato non sia rilevante rispetto alle relazioni tra i processi.

Soluzione

Indicando i processi con i loro process_id (P1000, P1001, etc.), e le due fork con i nomi fork1 e fork2 per distinguerle, si ha questo comportamento:

1. Il processo P1000 esegue fork1 e genera il processo P1001; in P1000 fork1 restituisce 1001, in P1001 restituisce 0; entrambi i processi proseguono e eseguono fork2

2. Il processo P1000 esegue fork2 e genera il processo P1002; in P1000 fork2 restituisce 1002, in P1002 restituisce 0;

3. Anche il processo P1001 esegue fork2 e genera il processo P1003; in P1001 fork2 restituisce 1003, in P1003 restituisce 0;

4. Considerando i due valori restituiti dalle due fork nei tre processi (si ricordi che un processo alla creazione è una copia del proceso creante, quindi eredita l'area dati al momento della creazione) si ha questo schema:

in P1000 fork1 ha restituito 1001, fork2 ha restituito 1002
in P1001 fork1 ha restituito 0, fork2 ha restituito 1003
in P1002 fork1 ha restituito 1001, fork2 ha restituito 0
in P1003 sia fork1 sia fork2 hanno restituito 0

5. Quindi solo nel processo P1003 le due fork hanno restituito valori uguali. Sul terminale compare

Processo 1003 nel ciclo while

e il processo rientra in ciclo, ripetendo la sequenza sopra descritta all'infinito (in teoria). Gli altri processi visualizzano sul terminale i seguenti messaggi:

Processo 1000 fuori dal ciclo while
Processo 1001 fuori dal ciclo while
Processo 1002 fuori dal ciclo while

L'ordine con cui vengono visualizzati questi messaggi non è deterministico perché dipende dalla sequenza di esecuzione, che a sua volta dipende dallo scheduler.

4) E' dato il seguente frammento di programma, eseguito nel sistema operativo Unix

```
id = 0;                 // programma "A"
if (fork() == 0)
    id = fork();

if (id == 0)
    execl("B",...);
else
    execl("C",...);
...
```

descrivere l'evoluzione del sistema durante l'esecuzione, cioè descrivere quali processi vengono creati, quali programmi eseguono e quale è il contenuto della memoria del sistema (aree codice e aree dati) durante e al termine della esecuzione del frammento, nell'ipotesi che non si verifichino errori.

Soluzione

Chiamiamo per semplicità A il processo che esegue il programma A, e A1, A2, etc., i processi creati a seguito dell'esecuzione della funzione fork(). L'esecuzione della istruzione

```
if (fork() == 0)
```

causa la creazione di un processo figlio A1, cui la fork restituisce il valore 0. Al processo A viene restituito un valore > 0 che coincide con l'identificatore di processo del processo A1. Il processo figlio A1 quindi vede soddisfatta la condizione ed esegue l'istruzione

```
    id = fork();
```

che causa la creazione del processo A2. Nel processo A1 la variabile **id** riceve un valore > 0, mentre nel processo A2 la variabile **id** riceve il valore 0.
Nel processo A la variabile id conserva il valore 0 assegnato dall'istruzione

```
id = 0;
```

poiché il processo A non è entrato all'interno della struttura condizionale if.
Si ha quindi la seguente situazione:

- nel processo A id = 0,
- nel processo A1 id > 0,
- nel processo A2 id = 0.

I processi A e A2 proseguiranno quindi eseguendo il codice del programma B, mentre il processo A1 proseguirà eseguendo il codice del programma C.

5) Si consideri l'esecuzione di questo programma nel sistema operativo Unix, supponendo che il processo che lo esegue abbia process-id uguale a 1000 e che i processi creati successivamente abbiano process-id consecutivi e crescenti:

```
main()
{ int i;
  while ((i = fork()) > 0)
      printf ("i = %d\n", i);
}
```

Dire come evolve il sistema durante l'esecuzione, cioè se vengono creati altri processi, in caso affermativo che gerarchie padre-figlio si stabiliscono, e che cosa viene visualizzato sul terminale.

Soluzione

Il processo chiama la funzione fork() che crea un processo con process_id = 1001, restituisce al processo creato il valore 0 e al processo padre il valore 1001. Il processo creato quindi non esegue il ciclo while e termina.

Il processo padre chiama la funzione printf che scrive su terminale

i = 1001

Rientra quindi in ciclo ed esegue nuovamente la funzione fork(), che crea un processo con process_id = 1002. Come prima il processo creato riceve dalla funzione fork() il valore 0, quindi non entra nel ciclo while e termina, mentre il processo padre scrive sul terminale "i = 1002", e così via in un ciclo teoricamente infinito.

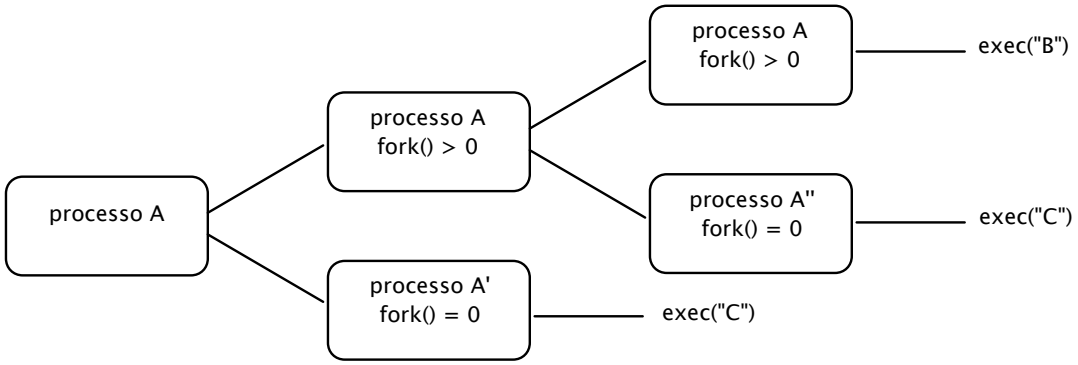
6) Dato il seguente frammento di programma, eseguito nel sistema operativo Unix, descrivere l'evoluzione del sistema durante l'esecuzione, cioè descrivere quali processi vengono creati, quali programmi eseguono e quale è il contenuto della memoria del sistema (aree codice e aree dati) durante e al termine della esecuzione del frammento, nell'ipotesi che non si verifichino errori.

```
...                          // programma "A"
if (fork() && fork())
    exec("B");
else
    exec("C");
...
```

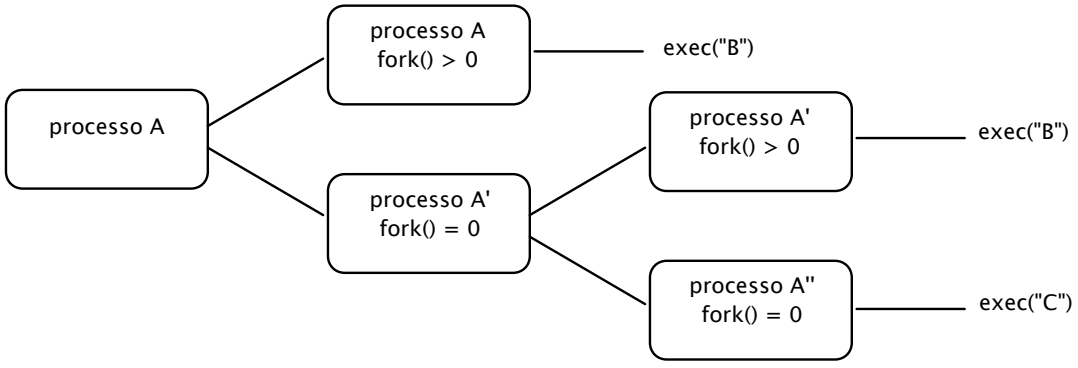
Come cambia l'esecuzione se l'operatore && è sostituito dall'operatore || ?

Soluzione

Ricordiamo che l'operatore && provoca la valutazione del suo secondo operando solo se il primo ha valore logico vero. La prima fork() crea un processo cui restituisce il valore 0, cioè falso. Pertanto nel processo figlio non viene valutata il secondo operando nella clausola if, che risulta complessivamente falsa. Il processo figlio prosegue quindi eseguendo il programma "C".
Al processo chiamante la fork restituisce l'identificatore del figlio, maggiore di zero e quindi corrispondente al valore logico vero. Viene quindi valutato il secondo operando della clausola if, causando una ulteriore duplicazione del processo che esegue il programma "A". Come prima, nel processo figlio così creato fork() restituisce valore zero, assegnando complessivamente il valore logico falso all'intera clausola if; il processo figlio esegue quindi il programma "C". il padre, ricevendo l'identificatore di processo del figlio, maggiore di zero e quindi corrispondente al valore logico vero, vede soddisfatta l'intera clausola logica dell'istruzione if, quindi esegue il programma "B".



Se l'operatore && è sostituito dall'operatore || la valutazione della clausola if cambia, e il comportamento complessivo è descritto dalla figura seguente:



7) Descrivere, utilizzando il linguaggio C, un frammento di programma che, utilizzando le system call dedicate alla gestione dei processi del sistema operativo Unix, realizzi quanto segue:

a. crei un nuovo processo che esegue il programma contenuto nel file /usr/bin/prog passandogli la sequenza di parametri "param1 param2", segnalando eventuali errori nella creazione;
b. se il processo è stato correttamente creato, ne attenda la fine prima di procedere con la propria esecuzione;
c. visualizzi sul terminale il codice con cui termina il processo creato
d. continui quindi con altre istruzioni non dettagliate.

Soluzione

Sono possibili piccole varianti di stile ma la sostanza della soluzione è la seguente:

```
...
int pid, status;
...
pid = fork();
if (pid < 0)                  // creazione del processo non riuscita
  { printf("Errore nella creazione del processo, esito = %d\n", pid);
     ... eventuale trattamento dell'errore ...
  }
else if (pid == 0)            // processo figlio
  { execl("/usr/bin/prog", "/usr/bin/prog", "param1", "param2",(char *) 0);
     printf("Errore nella esecuzione di /usr/bin/prog param1 param2\n");
     ... eventuale trattamento dell'errore ...
  }
else                          // processo padre
    pid = wait(&status);      // si assume che ci sia solo un processo figlio
     printf("Esecuzione terminata con codice %d\n",status);
  }
... segue codice non specificato ...
```

Se il processo in questione ha creato altri processi in precedenza, non è più valida l'ipotesi che ci sia un solo processo figlio,quindi il codice deve modificarsi come segue (dalla quartultima riga):

```
else                              // processo padre
  { while (wait(&status) != pid)  // attende che termini il processo controllandone l'id
    continue;
     printf("Esecuzione terminata con codice %d\n",status);
  }
... segue codice non specificato ...
```

8) Progettare un frammento di programma che utilizzi le funzioni *fork* , *exec* e *wait* del sistema Unix per realizzare il seguente comportamento:

Il programma *p1* deve mandare in esecuzione i programmi *p2* e *p3* creando per essi due processi distinti. I programmi *p2* e *p3* devono quindi evolvere indipendentemente, mentre *p1* deve aspettare che entrambi i processi creati finiscano prima di proseguire con l'esecuzione. *p1* prosegue poi chiamando la funzione *f2* se è terminato prima *p2*, o la funzione *f3* se è terminato prima *p3*.

Soluzione

```
i = fork();                  // duplica il processo che esegue p1
if (i == 0)                  // è il processo figlio?
  { execl("prog2",...);          // sì, esegue prog2
    ...gestione errori...
  }
else                             // no, il padre memorizza l'id
    id2 = i;                     // del processo che esegue prog2

i = fork();                      // come sopra per prog3
if (i == 0)
  { execl("prog3",...);
    ...gestione errori...
  }

fine1 = wait(&stato);        // p1 attende la fine dei due figli
fine2 = wait(&stato);        // e ne memorizza l'id

if (fine1 == id2)            // è finito prima prog2?
    f2(...);                     // sì, esegue la funzione f2
else
    f3(...);                     // no, esegue la funzione f3
```

9) Progettare un frammento di programma che utilizzi le funzioni di gestione dei processi di Unix per realizzare il seguente comportamento:

Il programma *A* manda in esecuzione concorrente due processi che eseguono rispettivamente i programmi *B* e *C*, e ne attende la terminazione. I processi terminano rispettivamente con codici di completamento *cB* e *cC*. Quindi *A* verifica che il processo che termina per primo abbia il codice di completamento maggiore dell'altro, e in tal caso termina con codice di completamento 1, altrimenti termina con codice di completamento 0.

Soluzione

```
if (fork() == 0)                  // A crea un processo che eseguirà il programma B
  { execl("B",...);
    ...gestione errori...
  }

if (fork() == 0)                  // idem per il programma C
  { execl("C",...);
    ... gestione errori ...
  }

wait(&stato1);                  // aspetta la fine del primo processo
                                // e ne rileva lo stato
wait(&stato2);                  // idem con il secondo

if (stato1 > stato2)                // verifica la relazione tra gli stati
    exit(1);                        // e termina di conseguenza
else
    exit(0);
```

10) Utilizzando le system call di Unix per la gestione dei processi, progettare e codificare in linguaggio C una funzione

int xprogram (const char * pathname, const char* argument);

che realizza il seguente comportamento:

1. verifica che pathname non sia una stringa vuota, altrimenti emette un messaggio diagnostico e termina con valore -1;

2. crea un nuovo processo che esegue il programma il cui nome è contenuto nel parametro pathname, passandogli come argomento la stringa contenuta nell'unico parametro argument;

3. se si verificano errori nella creazione del processo o nell'esecuzione del programma, emette un appropriato messaggio diagnostico e termina con valore -1;

4. altrimenti attende finché il processo creato non termina la sua esecuzione, quindi termina a sua volta restituendo come valore il codice di terminazione del processo creato.

Soluzione

La soluzione non richiede grandi commenti. La variabile command serve perché è una convenzione Unix che il primo parametro passato a un programma eseguito con la funzione execl sia il nome del programma stesso, depurato degli altri elementi del pathname. Tale estrazione non è qui dettagliata ma condensata in un commento.

```
int xprogram (const char * pathname, const char* argument) {
int id, status;
char command[256];

if (pathname == (char*) 0 || pathname[0] = '\0'){
    printf("Manca il nome del programma\n");
    exit(-1);
    }

if ((id = fork()) == 0) {

//      copia in command il nome del programma estratto da pathname
//      . . .

    execl(pathname,command,argument,(char *)0);
    printf("Errore nell'esecuzione del programma %s\n",pathname);
    exit(-1);
    }
else if (id < 0) {
    printf("Errore nella creazione del processo per il programma %s\n",pathname);
    exit(-1);
    }

while (wait(&status) != id)    // il programma potrebbe aver creato altri processi
    continue;
return status;
}
```

11) Progettare in linguaggio C e commentare adeguatamente un frammento di programma che simula una corsa su pista con *N* concorrenti. Il programma, denominato *corsa*, deve utilizzare le funzioni *fork*, *exec*, *exit* e *wait* del sistema operativo Unix per realizzare il comportamento qui descritto.

Il programma *corsa* crea *N* nuovi processi che eseguono in modalità concorrente tra di loro uno stesso programma contenuto nel file di pathname relativo *corridore*, che corrisponde al completamento di un giro della pista da parte di un concorrente. Il programma *corridore* (il cui codice non deve essere progettato) riceve come argomento un numero (1..*N*) che identifica il concorrente. Per semplicità indichiamo con i nomi *corsa* e *corridore* (*corridore*$_1$, *corridore*$_2$, *corridore*$_N$) sia i programmi sia i processi che li eseguono.

Dopo aver creato gli *N* processi *corridore*$_i$ il processo *corsa* si mette in attesa. Quando un processo *corridore*$_i$ termina, il processo *corsa* scrive sul terminale il messaggio “Il concorrente … ha terminato il giro … ”, specificando sia il numero del concorrente sia il numero del giro completato. Il processo *corsa* rimanda quindi in esecuzione il programma *corridore*, creando un nuovo processo, e passandogli come argomento il numero dello stesso concorrente: *corsa* quindi si rimette in attesa e questo comportamento si ripete finché non termina il processo *corridore*$_i$ che ha completato per primo *T* giri. In questo caso *corsa* scrive sul terminale “Il concorrente … ha vinto la corsa” specificando il numero del concorrente vincitore, aspetta che tutti gli altri concorrenti abbiamo terminato il loro giro e termina a sua volta.

Si supponga per semplicità che non si verifichino errori nella creazione dei processi e nell’esecuzione dei programmi, e si assuma che il processo che esegue *corsa* non abbia creato in precedenza e non crei durante l’esecuzione altri processi all’infuori di quelli sopra descritti. Per il dimensionamento delle strutture dati si assuma che *N* e *T* siano costanti simboliche.

La soluzione di questo esercizio è lasciata allo studente, verrà discussa a lezione

12) Progettare in linguaggio C e commentare adeguatamente un frammento di programma denominato *esperimento* che utilizzi le funzioni *fork* , *exec* , *exit* e *wait* del sistema operativo Unix per realizzare il seguente comportamento:

Il programma *esperimento* crea due nuovi processi che eseguono in modalità concorrente due programmi contenuti nei file di pathname relativo *timeout* e *test* (che non hanno parametri in ingresso e il cui codice non è rilevante). Per semplicità chiamiamo *esperimento*, *timeout* e *test* sia i programmi sia i processi che li eseguono.

Successivamente il processo *esperimento* attende che uno degli altri due processi termini, quindi scrive sul terminale il messaggio “Programma ... terminato con codice ...”, specificando sia il nome del programma eseguito dal processo che è terminato, sia il suo codice di uscita.

Se è terminato il processo *test* e il processo *timeout* è ancora in esecuzione, *esperimento* rimanda in esecuzione il programma *test,* creando un nuovo processo, e rimettendosi poi in attesa. Ripete quindi questo comportamento finché non termina il processo *timeout*. Quando il processo *timeout* termina, *esperimento* scrive il messaggio di cui sopra (ora riferito al processo *timeout*) e termina immediatamente con codice di uscita uguale al numero di esecuzioni del programma *test*.

Si supponga per semplicità che non si verifichino errori nella creazione dei processi e nell’esecuzione dei programmi, e si assuma che il processo che esegue *esperimento* non abbia creato in precedenza altri processi.

Soluzione

Una soluzione per il programma *esperimento* è la seguente, che comprende anche le dichiarazioni dei file di intestazione e la rilevazione degli errori nella gestione dei processi.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, const char * argv[]) {
    pid_t id, id_test, id_timeout;
    int contatore_test, stato;

    if ((id_timeout = fork()) == 0) {            // crea un processo per timeout
        execl("./timeout","timeout", (char *)0);                // e lo esegue
        printf("Errore nell'esecuzione del programma timeout\n"); // salvo errori
        exit(-1);
        }
    else if (id_timeout < 0) {
        printf("Errore nella creazione del processo per timeout\n");
        exit(-1);
        }

    if ((id_test = fork()) == 0) {                      // idem per il programma  test
        execl("./test","test", (char *)0);
        printf("Errore nell'esecuzione del programma test\n");
        exit(-1);
    }
    else if (id_test < 0) {
        printf("Errore nella creazione del processo per test\n");
        exit(-1);
        }


    contatore_test = 1;
    while (1) {                          // esegue ripetutamente
        id = wait(&stato);                      // attende fine processo
        if (id == id_timeout) {                 // se è il programma timeout scrive msg
            printf("Programma timeout terminato con codice %d\n", stato);
            exit(contatore_test);                       // esce con # esecuzioni di test
            }
        else {                                  // è il programma test
            printf("Programma test terminato con codice %d\n", stato);
            if ((id_test = fork()) == 0) {              // crea un nuovo processo per test
                execl("./test","test", (char *)0);      // e lo esegue
                exit(-1);                               // a meno di errori
                }
            contatore_test++;
        }
    }
    exit(contatore_test);
}
```

13. Progettare in linguaggio C un frammento di programma che utilizzi le funzioni fork , exec e wait del sistema operativo Unix per realizzare il seguente comportamento:

Il programma “P1” deve mandare in esecuzione il programma contenuto nei file di pathname relativo “P2” (il cui codice non è rilevante) creando per esso un nuovo processo. Se si verificano errori, P1 deve emettere un messaggio diagnostico che identifica l’errore (creazione del processo o esecuzione del programma “P2”) e deve terminare la propria esecuzione con codice di terminazione -999.

L’esecuzione deve continuare con questo comportamento: quando il processo creato termina, P1 scrive sul terminale il messaggio “P2 terminato con codice ...” seguito dal codice di terminazione del processo.

Se il codice di terminazione è positivo o nullo, P1 rimanda in esecuzione lo stesso programma (creando un nuovo processo); se il codice di terminazione è negativo, P1 termina a sua volta con codice di terminazione uguale all’ultimo codice di terminazione del processo che esegue “P2”.

Si assuma che il processo che esegue P1 non abbia creato in precedenza altri processi.

Soluzione

Una traccia per il programma P1 è la seguente, dove la funzione exec è indicata in forma generica.

```
main() {
int id_p2, stato;

while (1) {                              // esegue ripetutamente
    if ((id_p2 = fork()) == 0) {                 // crea un processo per “P2”
        exec(“P2”, ...);                         // e lo esegue
        printf(“Errore nell’esecuzione del programma P2\n”);  // salvo errori
        exit(-999);
        }

    else if (id_p2 < 0) {                // errore nella creazione del processo
        printf(“Errore nella creazione del processo per P2\n”);
        exit(-999);
        }

    id_p2 == wait(&stato);               // attende P2 e controlla esito
    if (stato < 0)                       // fine di P2 non OK?
        exit(stato);                             // termina anche p1
    else                                 // fine di P2 OK, msg e torna in ciclo
        printf(“P2 terminato con codice %d\n”, stato);
    }
}
```

14) Progettare in linguaggio C e commentare adeguatamente un frammento di programma denominato *ambiente* che utilizzi le funzioni *fork* , *exec* , *exit* e *wait* del sistema operativo Unix per realizzare le seguenti operazioni relative al monitoraggio di variabili ambientali.

Il programma *ambiente* crea due nuovi processi che eseguono in modalità concorrente due programmi contenuti nei file di pathname relativo *temperatura* e *umidità*, senza parametri in ingresso. Per semplicità chiamiamo *ambiente*, *temperatura* e *umidità* sia i programmi sia i processi che li eseguono.

I processi *temperatura* e *umidità* (il cui codice non deve essere progettato) eseguono ciascuno delle misure per un tempo limitato non precisato e terminano con un codice di uscita uguale alla media delle misure effettuate, oppure con il codice -1 se si verificano errori di misura.

Il processo *ambiente* dopo aver creato i due processi e eseguito i due programmi attende che uno dei due termini, quindi scrive sul terminale il messaggio “Temperatura media rilevata ...” oppure “Umidità media rilevata ...”, a seconda di quale processo sia terminato, seguito dal codice di uscita del processo. Rimanda quindi in esecuzione il programma terminato creando un nuovo processo; questo ciclo si ripete finché uno dei due processi non termina con codice -1. Il processo *ambiente*, in questo caso, attende che anche l’altro processo termini la sua esecuzione, quindi termina a sua volta con codice di uscita 1 se gli errori di misura (segnalati dal codice di terminazione -1) riguardano la temperatura, 2 altrimenti.

Si supponga per semplicità che non si verifichino errori nella creazione dei processi e nell’esecuzione dei programmi, e si assuma che il processo *ambiente* non abbia creato in precedenza altri processi.

Soluzione
Tralasciando le intestazioni del programma e i controlli sull'esito delle operazioni fork e execl, uno schema di soluzione è il seguente:

```
int main (int argc, const char * argv[]) {
    pid_t id, id_ok, id_temp, id_umid;
    int stato;

    if ((id_temp = fork()) == 0)                                // crea un processo per temperatura
        execl("./temperatura","temperatura", (char *)0);        // e lo esegue

    if ((id_umid = fork()) == 0)                                // idem per il programma  timeout
        execl("./umidita","umidita", (char *)0);

    while (1) {                                                 // esegue ripetutamente
        id = wait(&stato);                                      // attende fine processo
        if (stato == -1) {                                      // c'è stato un errore di misura?
            id_ok = wait(&stato);                               // aspetta l'altro processo
            if (id == id_temp)                                  // l'errore riguarda la temperatura
                exit(1);
            else                                                // l'errore riguarda l'umidità
                exit(2);
            }

        if (id == id_temp) {                                    // misura OK
            printf("Temperatura media rilevata %d\n", stato);       // scrive il valore di temperatura
            if ((id_temp = fork()) == 0)                        // e ricrea il processo
                execl("./temperatura","temperatura", (char *)0);
            }
        else {
            printf("Umidità media rilevata %d\n", stato);       // idem per umidità
            if ((id_umid = fork()) == 0)
                execl("./umidita","umidita", (char *)0);
            }
        }                                                       // fine while (1)
}
```