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

Temi d'esame del primo modulo risolti

1. 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 execl sia il nome del programma stesso, depurato degli altri elementi del pathname. Tale estrazion enon 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 potrerre aver creato altri processi
    continue;
return status;
}
```

2. 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.

Soluzione

Una soluzione per il programma *corsa* è la seguente: il testo no è direttamente compilabile perché mancano i file di intestazione (*.h) e alcuni tipi sono impostati in modo semplificato: ad esempio, id come valore di ritorno da fork() è formalmente di tipo pid_t che è implementato come int

```
int main (int argc, const char * argv[]) {
int const N = ..., T = ...;
int id, id_giro[N];                                 // process_id dei concorrenti al giro corrente
int giri[N];                                        // numero di giri di ogni concorrente [1..N]
char buffer[5];
int concorrente, stato, i;

for (i = 0; i < N; i++)                                     // inizializzazione
    giri[i] = 0;

for (i = 0; i < N; i++) {                                   // crea i processi per i concorrenti
    if ((id_giro[i] = fork()) == 0) {                       // memorizza il process_id
        sprintf(buffer, "%d", i+1);                 // prepara la lista parametri
        execl("./corridore","corridore", buffer, (char *)0);  // e esegue il processo creato
        }
    }

while (1) {                                                 // esegue la corsa
    id = wait(&stato);                              // attende l'arrivo di un concorrente

    for (i = 0; i < N; i++) {
        if (id == id_giro[i]) {                             // vede che concorrente è
            concorrente = i;
            giri[i]++;                                      // incrementa il numero di giri e scrive
            printf("Il concorrente %d ha terminato il giro %d\n", i+1, giri[i]);

            if (giri[i] == T) {                                     // se è l'ultimo giro il concorrente ha
vinto
                printf("Il concorrente %d ha vinto la corsa\n",i+1);
                for (i = 0; i < N-1; i++)                           // aspetta che gli altri concorrenti
                    id = wait(&stato);                                      // finiscano il loro giro
                exit(0);                                                    // e termina
                }
            break;
            }
        }
                                                            // non è l'ultimo giro
    if ((id_giro[concorrente] = fork()) == 0) {     // crea un nuovo processo per il concorrente
        sprintf(buffer, "%d", concorrente+1);               // e lo esegue, come sopra
        execl("./corridore","corridore", buffer, (char *)0);
        }
    }
}
```

3. 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  timeout
         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);
}
```

4. 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);
    }
}
```

5. In un sistema a memoria virtuale paginata un processo in esecuzione ha uno spazio di indirizzamento contiguo di 384 Kbyte. La memoria fisica del sistema ha una dimensione non specificata ed è organizzata in frame di 64 Kbyte ciascuno. Al processo vengono assegnati quattro frame con una strategia di allocazione fissa in ambito locale. La tabella delle pagine del processo è la seguente, dove tutte le numerazioni sono decimali, i tempi sono espressi in unità convenzionali e indicano il tempo trascorso dal caricamento del frame o dall'ultimo accesso al suo contenuto:

# pag virtuale	*valida*	*modificata*	*# frame*	*tempo dal caricamento*	*tempo dall' ultimo accesso*
0	1	1	3	330	160
1	0	-	-	-	-
2	1	1	4	190	170
3	1	0	1	180	180
4	1	1	2	300	150
5	0	-	-	-	-

Si supponga che il processo acceda consecutivamente ai seguenti indirizzi virtuali: 100.000, 236.000. Si considerino gli algoritmi di sostituzione delle pagine FIFO e LRU, e per ciascuno di essi si dica:

1. se la sequenza dei due accessi causa, per uno solo o per entrambi gli accessi, un page fault;
2. se sì, qual è il frame che viene utilizzato per caricare la pagina mancante, e se il suo contenuto deve essere scritto su disco prima del caricamento;
3. qual è l'indirizzo fisico corrispondente ad ogni indirizzo virtuale (eventualmente dopo la risoluzione del page fault).

Soluzione

L'indirizzo 100.000 è nella pagina virtuale 1 (la seconda) all'offset 34.464; l'indirizzo 236.000 è nella pagina virtuale 3 (la quarta) all'offset 39.392.

La pagina virtuale 1 non è valida, l'accesso all'indirizzo 100.000 causa un page fault.

Algoritmo FIFO
Il frame residente in memoria da più tempo è il frame 3, che contiene la pagina virtuale 0; il frame è modificato quindi deve essere scritto su disco. Dopo la sostituzione l'indirizzo fisico corrispondente all'indirizzo virtuale 100.000 è 3 * 65.536 + 34.464 = 231.072.

La pagina virtuale 3 che contiene l'indirizzo virtuale 236.000 è valida, quindi non si ha page fault. l'indirizzo fisico corrispondente è 1 * 65.536 + 39.392 = 104.928.

Algoritmo LRU
Il frame non utilizzato da più tempo è il frame 1, che contiene la pagina virtuale 3; il frame non è modificato quindi non deve essere scritto su disco. Dopo la sostituzione l'indirizzo fisico corrispondente all'indirizzo virtuale 100.000 è 1 * 65.536 + 34.464 = 100.000.

La pagina virtuale 3 che contiene l'indirizzo virtuale 236.000 non è più valida, quindi si ha un altro page fault. Il frame non utilizzato da più tempo è il frame 4, che contiene la pagina virtuale 2; il frame è modificato quindi deve essere scritto su disco. Dopo la sostituzione l'indirizzo fisico corrispondente all'indirizzo virtuale 100.000 è 4 * 65.536 + 34.464 = 296.608.

6. Un processo in esecuzione ha uno spazio di indirizzamento contiguo di 64 Kbyte; la memoria fisica del sistema è organizzata in frame di 16Kbyte ciascuno; al processo vengono assegnati tre frame con una strategia locale di sostituzione delle pagine.

La tabella delle pagine del processo è la seguente, dove tutte le numerazioni sono decimali, i numeri di pagina logica e frame fisico partono da zero, i tempi sono espressi in unità convenzionali e indicano il tempo trascorso dal caricamento del frame o dall'ultimo accesso al suo contenuto:

# pagina virtuale	*valida*	*modificata*	*# frame*	*tempo dal caricamento*	*tempo dall' ultimo accesso*
0	1	1	4	250	140
1	1	0	7	160	160
2	1	0	2	190	150
3	0	-	-	-	-

Il programma accede consecutivamente agli indirizzi virtuali 58.912 e 24.192. Si considerino gli algoritmi di sostituzione delle pagine FIFO e LRU, e per ciascuno di essi si dica:

- se la sequenza dei due accessi causa, per uno o entrambi gli accessi, un page fault;
- se sì, qual è il frame utilizzato per la sostituzione, e se il suo contenuto deve essere salvato su disco prima della sostituzione;
- qual è l'indirizzo fisico corrispondente, eventualmente dopo la risoluzione del page fault.

Soluzione

L'indirizzo virtuale 58.912 è nella pagina virtuale 3 all'offset 9.760
L'indirizzo virtuale 24.192 è nella pagina virtuale 1 all'offset 7.798

Algoritmo di sostituzione FIFO

La pagina virtuale 3 non è valida, quindi l'accesso al primo indirizzo genera un page fault. Il frame vittima è quello da più tempo residente in memoria (quello con il *tempo dal caricamento* più elevato), cioè il frame 4, occupato dalla pagina virtuale 0. Il frame è modificato quindi deve essere salvato su disco. Dopo la risoluzione del page fault l'indirizzo virtuale 58.912 corrisponde all'indirizzo fisico 75.296 (4 * 16384 + 9.760)

Il successivo accesso alla pagina virtuale 1 non genera page fault, l'indirizzo fisico corrispondente all'indirizzo virtuale 24.192 è 122.486 (7 * 16384 + 7.798)

Algoritmo di sostituzione LRU

Anche in questo caso il primo accesso genera un page fault, ma il frame vittima è il frame da più tempo non usato (quello con il *tempo dall'ultimo accesso* più elevato), cioè il frame 7, occupato dalla pagina virtuale 1. Il frame noon è modificato quidni non deve essere salvato su disco. Dopo la risoluzione del page fault l'indirizzo virtuale 58.912 corrisponde all'indirizzo fisico 124.448 (7 * 16384 + 9.760)

Il successivo accesso alla pagina virtuale 1 genera un page fault, perché la pagina è stata rimossa dalla memoria fisica durante il passo precedente. In questo caso il frame utilizzato per la sostituzione è quello con il secondo valore di tempo più elevato, cioè il frame 2. Il frame non è modificato quindi non deve essere salvato su disco. Dopo la risoluzione del page fault l'indirizzo virtuale corrispondente è 40.566 (2 * 16384 + 7.798).

7. Si consideri il file system indicizzato multilivello di Unix, con dimensione del blocco di allocazione di 512 byte e puntatori di 4 byte. Si dica, motivando la risposta, quanti accessi a disco sono necessari per leggere sequenzialmente gli ultimi 10 blocchi di un file nei seguenti casi:

1. il file ha una dimensione di 10 Kbyte
2. il file ha una dimensione di 100 Kbyte
3. il file ha una dimensione di 1 Mbyte

Si assuma che l'i-node del file sia già in memoria e che dopo il primo accesso venga conservato in cache un blocco indice per ogni livello di indice.

Nota: Kbyte = 2^{10} byte, Mbyte = 2^{20} byte

Soluzione

Ogni blocco indice contiene 128 puntatori. Un file viene indicizzato in questo modo:
- blocchi da 1 a 12: accesso diretto;
- blocchi da 13 a 140: un livello di indice;
- blocchi da 141 a 16524 (140+128*128): due livelli di indice
- oltre: tre livelli di indice

Gli ultimi 10 blocchi hanno questa numerazione (partendo da 1) e un corrispondente numero di accessi:
- per il file da 10Kbyte (20 blocchi) da 11 a 20, con accesso diretto per i blocchi 11 e 12, con un livello di indice per i seguenti; totale = 10 + 1 = 11 accessi;
- per il file da 100Kbyte (200 blocchi) da 191 a 200, tutti indirizzati dallo stesso blocco indice di secondo livello (che indicizza i blocchi da 141 a 268); totale = 10 + 1 (indice di primo livello) + 1 (indice di secondo livello) = 12 accessi;
- per il file da 1 Mbyte (2048 blocchi): da 2039 a 2048, tutti indicizzati dallo stesso blocco indice di secondo livello (il quindicesimo, che indicizza i blocchi da 1933 a 2060); totale = 10 + 1 (indice di primo livello) + 1 (indice di secondo livello) = 12 accessi.

8. Si faccia riferimento al file system indicizzato multilivello di Unix, con dimensione del blocco di disco di 4Kbyte e puntatori di 4 byte.

Rispondere alle seguenti domande:

1) Escludendo l'i-node, quanti blocchi di disco (contenenti gli indici e i dati) servono per allocare un file di 32 Mbyte? e per un file di 320 Mbyte?

2) Attraverso quali e quanti blocchi di indice (indicare gli index block e gli offset dei puntatori interessati) si accede al 200° blocco di un generico file (assumendo che il file sia di dimensione >= 200 blocchi)? e al 2.000° blocco (sempre assumendo che il file sia di dimensione adeguata)?

3) In quale blocco si trova il byte di indirizzo 5.000.000, e quanti accessi a disco sono complessivamente necessari per leggerne il valore (assumendo che la dimensione del file sia >= 5.000.000 byte e che non vi sia alcuna informazione in cache oltre all'i-node)?

Nota: 1Kbyte = 1.024 byte, 1Mbyte = 1.024 ´ 1.024 byte.

Soluzione

Se il blocco logico è di 4096 byte e gli indirizzi di 4 byte si hanno 4096 / 4 = 1024 indirizzi per blocco

Un file di 32 Mbyte occupa 32M / 4K = 8K = 8192 blocchi

8.192 blocchi richiedono 8192 indici di cui 12 nell'i-node, 1024 nell'indice di primo livello e i rimanenti 7156 in 7 blocchi di 2° livello (a loro volta indicizzati da un indice di primo livello).

Complessivamente il file occupa 8192 + 7 (indici di 2° livello) + 2 (indici di 1° livello) = 8201 blocchi.

Un file di 320 Mbyte occupa 81.920 blocchi per i dati, 1 blocco di indice di primo livello, 79 blocchi di indice di secondo livello indicizzati da un ulteriore blocco indice di primo livello, per un totale di 82.001 blocchi

Il blocco n. 200 è indicizzato dal blocco indice di 1° livello (che indicizza i blocchi da 13 a 1035), l'offset del puntatore di accesso è 188 * 4 = 752.

Il blocco n. 2000 è indicizzato dal primo blocco indice di 2° livello (che indicizza i blocchi da 1036 a 2059), l'offset del puntatore di accesso è 964 * 4 = 3856.

Il byte 5.000.000 del file occupa il blocco 5.000.000 / 4096 = 1220 (contando da 0, quindi il 1221-esimo), e si trova all'offset (5.000.000) mod 4096 = 2880. Il blocco 1220 è indicizzato dal primo blocco indice di secondo livello (che indicizza i blocchi da 1036 a 2059) all'offset 184 * 4 = 736. Per accedervi sono necessari tre accessi al disco: indice di primo livello, indice di secondo livello e blocco dati

9. Si consideri, nel file system di Unix, un file di nome **A** che contiene il testo “Io sono il file A”. Data questa sequenza di comandi:

```
$ ln A C
$ ln -s C D
$ ln -s A E
$ mv A B
```

descrivere che relazioni si stabiliscono dopo l’esecuzione della sequenza tra i file denominati A, B, C, D ed E, e che cosa succede eseguendo i seguenti comandi, motivando la risposta:

```
$ cat A
$ cat B
$ cat C
$ cat D
$ cat E
```

si ricorda che:

`$ ln [-s] X Y`	crea un link di nome Y, hard oppure simbolico se è specificata l’opzione -s, al file X
`$ mv X Y`	rinomina il file X dandogli nome Y
`$ cat X`	visualizza il contenuto del file X

Soluzione

I comandi di link stabiliscono le seguenti relazioni:

`$ ln A C`	il pathname C identifica lo stesso file identificato dal pathname A
`$ ln -s C D`	il pathname D è un link simbolico al file C che identifica lo stesso file identificato da A
`$ ln -s A E`	il pathname E è un link simbolico al file A
`$ mv A B`	il file precedentemente identificato dal file A viene ora identificato dal pathname B

I comandi cat sopra riportati hanno quindi questo esito

`$ cat A`	
`cat: A: No such file or directory`	perché il file A è stato ridenominato con il pathname B
`$ cat B`	
`Io sono A`	perché il pathname B identifica il file che in origine si chiamava A
`$ cat C`	
`Io sono A`	perché il pathname C identifica il file che in origine si chiamava A
`$ cat D`	
`Io sono A`	perché il file D è un link simblico al file C
`$ cat E`	
`cat: E: No such file or directory`	perché il file E è un link simbolico al file A che non esiste più con questo nome