Fonaments dels Sistemes Operatius

Departament d'Informàtica de Sistemes i Computadores (DISCA)
Universitat Politècnica de València

fSO

Pràctica 5

Versió 5.1
Creació de fils d'execució i avaluació de prestacions

1. Objectius

L'objectiu principal de la pràctica és **adquirir experiència en el maneig de les funcions de l'estàndard POSIX per a la creació i l'espera de fils**; treballar amb un escenari on es produïsquen operacions concurrents. En concret, veurem un exemple amb operacions concurrents sobre una matriu de dades per avaluar la millora que, en termes de temps d'execució, s'obté per l'ús dels fils en un processador multinucli.

2. Creació de fils

El codi de la figura 1 constitueix l'esquelet bàsic d'una funció que utilitza fils en la implementació.

```
/**
* Programa d'exemple "Hola món" amb pthreads.
* Per a compilar teclegeu:
*      gcc hola.c -lpthread -o hola
*/
#include <stdio.h>
#include <pthread.h>

void *Imprimeix( void *ptr )
  {
    char *missatge;
    missatge=(char*)ptr;

    //EXERCICI1.b
    write(1,missatge,strlen(missatge));
  }
int main()
  {
    pthread_attr_t atrib;
    pthread_t file1, fil2;

    pthread_attr_init( &atrib );

    pthread_create( &fil1, &atrib, Imprimeix, "Hola \n");
    pthread_create( &fil2, &atrib, Imprimeix, "món \n");

   //EXERCICI1.a
    pthread_join( fil1, NULL);
    pthread_join( fil2, NULL);

  }
```

Figura 1: Esquelet bàsic d'una programa amb fils POSIX

Creeu un arxiu "hola.c" que continga aquest codi, compileu-lo i executeu-lo des de la línia d'ordres.

```
$ gcc hola.c -lpthread -o hola
```

Com s'observa en el codi de la figura 1, les novetats que introdueix el maneig de fils van acompanyades de les funcions que calen per a inicialitzar-los; només hem fet ús de les més bàsiques o imprescindibles d'aquestes.

- Tipus **`pthread_t`** i **`pthread_attr_t`**, a què s'accedeix des de l'arxiu de capçalera `pthread.h`.

```
#include <pthread.h>

pthread_t th;
pthread_attr_t attr;
```

- Funció **pthread_attr_init**, encarregada d'assignar uns valors per defecte als elements de l'estructura d'atributs d'un fil. AVÍS! Si no s'inicialitzen els atributs, el fil no es pot crear.

```
#include <pthread.h >

int pthread_attr_init(pthread_attr_t *attr)
```

- Funció **pthread_create**, encarregada de crear un fil.

```
#include <pthread.h >

int pthread_create(pthread_t   *thread,const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Paràmetres de* pthread_create*:

thread*:* és el primer paràmetre d'aquesta funció, *thread,* i conté l'identificador del fil.

attr*:* l'argument *attr* especifica els atributs del fil. Pot prendre el valor NULL, i en aquest cas rep també valors per defecte: "*the created thread is joinable (not detached) and has default (non real-time) scheduling policy*".

start_routine: el comportament del fil que es crea ve definit per la funció que s'hi passa com a tercer paràmetre, *start_routine*, i a la qual es passarà com a argument el punter *arg*.

Valor de tornada de la funció pthread_create():

Retorna 0 si la funció s'executa amb èxit. En cas d'error, la funció retorna un valor diferent de zero.

- Funció **pthread_join**. Té l'efecte de suspendre el fil que la invoca fins que el fil que s'hi especifica com a paràmetre acabe. Aquest comportament és necessari, ja que quan el fil principal *acaba*, destrueix el procés i, per tant, obliga a la terminació de tots els fils que s'hagen creat.

```
#include <pthread.h >

int pthread_join(pthread_t  *thread, void **exit_status,);
```

Paràmetres de pthread_join:

`thread`: paràmetre que identifica al fil que cal esperar.

`exit_status`: conté el valor que el fil acabat comunica al fil que invoca a `pthread_join`.

- Funció **pthread_exit** Permet a un fil terminar voluntariament la seua execució. En acabar l'últim fil d'un procés termina el procés mateix. Mitjançant el paràmetre `exit_status` pot comunicar un valor de terminació a un altre fil que estiguera esperant-ne la finalització.

```
#include <pthread.h >

int pthread_exit(void *exit_status);
```

2.1 Exercici 1: treballar amb *pthread_join* i *pthread_exit*

Comproveu el comportament de la crida `pthread_join()` fent les modificacions següents al codi del programa `hola.c` que s'ha mostrat anteriorment.

Qüestions Exercici 1:

Elimineu (o comenteu) les crides `pthread_join` del fil principal.

- ¿Qué hi passa?¿Per què?

Substituïu les crides `pthread_join` per una crida `pthread_exit(0)`, prop del punt del programa marcat com //EXERCICI1.a)

- ¿Completa ara el programa la seua execució correctament?¿Perquè?

Elimineu (o comenteu) qualsevol crida a `pthread_join` o `pthread_exit` (prop del comentari //EXERCICI1.a) e introduiu en aquest mateix punt un retard d'1 segon (fent servir `usleep(…)`)

```
#include <unistd.h>
  void usleep(unsigned long usec);  // usec en microsegons
```

- ¿Què hi passa després de fer les modificacions proposades?

Introduïu ara un retard de 2 segons prop del comentari `//EXERCICI1.b`

- ¿Què hi passa ara?¿Perquè?

3 Seqüencialitat *versus* concurrència

Cal comprovar la diferència entre executar seqüencialment un conjunt d'accions i executar-les concurrent usant fils d'execució, en tots dos casos sobre un processador multinucli.
Per a fer-ho considerem un conjunt de NUMROWS vectors de dimensió DIMROW, de manera que DIMROW és molt major que NUMROWS. El programa executa una operació `AddRow()` per cada vector o fila. Aquesta operació té un cost computacional elevat, ja que DIMROW és gran. En l'exemple, l'operació `AddRow()` suma, per a tots els elements del vector, el valor d'una funció, i emmagatzema el resultat en el camp `suma` de la fila corresponent. El programa principal realitza NUMROWS crides a la funció `AddRow()` de forma seqüencial. Les estructures de dades i l'esquema de les crides que fa el fil principal es mostren en la figura 2.

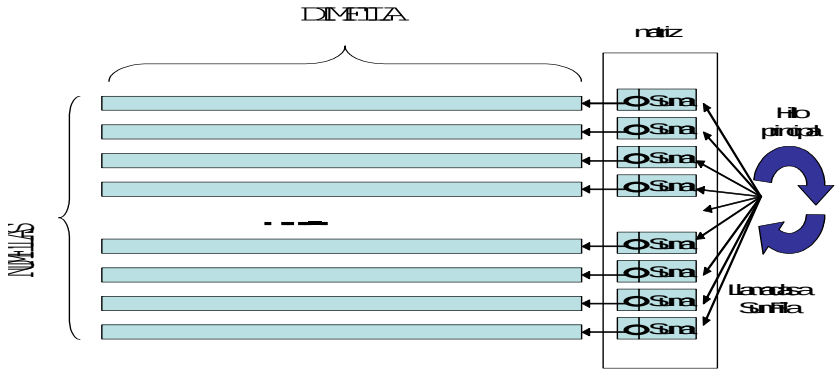

Figura 2. NUMFILES vectors de grandària DIMFILA

El codi corresponent a aquest exemple és el mostrat en la figura 3, on apareix la definició de `struct row` i la declaració de la variable global `matrix`. En el programa principal, després del bucle de crides seqüencials a `AddRow()`, se sumen tots els resultats parcials i es mostra en pantalla la suma total, que en aquest cas ha de ser **`DIMROW*NUMROWS`**, atès que tots els elements dels vectors emmagatzemen el valor 1.

```c
// Program SequentialAdd.c
// To compile do:
//     gcc SequentialAdd.c  -o SeqAdd -lm

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

#define DIMROW 1000000
#define NUMROWS 20

typedef struct row{
        int vector[DIMROW];
        long addition;
} row;

struct row matrix[NUMROWS];

void *AddRow( void *ptr )
{
    int k;
    row *fi;
    fi = (row *)ptr;

    fi->addition=0;
    for(k=0;k<DIMROW;k++) {
        fi->addition += exp((k*(fi->vector[k])+
          (k+1)*(fi->vector[k]))/(fi->vector[k]+2*k))/2;
    }
 }

int main()
{
    int i,j;
    long total_addition=0;
    pthread_t  threads[NUMROWS];
    pthread_attr_t atrib;

    // Vector elements are initialized to 1
    for(i=0;i<NUMROWS;i++) {
        for(j=0;j<DIMROW;j++) {
            matrix[i].vector[j]=1;
        }
    }
    // Thread attributes initialization
    pthread_attr_init( &atrib );

    // EXERCISE 2.a
    for(i=0;i<NUMROWS;i++)
    {
        AddRow(&matriz[i]);
    }
    // EXERCISE 2.b

    for(i=0;i<NUMROWS;i++)
        total_addition += matrix[i].addition;
    printf("Total addition is: %ld \n", total_addition);

}
```

Figura 3: codi del programa *SequentialAdd.c*. Per a augmentar el temps d'execució en cada iteració per a sumar un u cada vegada s'utilitza l'expressió:

"exp((k*(fi->vector[k])+(k+1)*(fi->vector[k]))/(fi->vector[k]+2*k))/2"

3.1 Exercici 2: sumar les fileres seqüencialment: *SumaSequencial.c*

Comentario [AMB1]: Si en alguns contextos no és possible posar dièresi en aquest mot, escriviu "SumaSequencial" en els contextos esmentats i "SumaSeqüencial" en la resta.

Compileu SequentialAdd.c i executeu el codi resultant. Comproveu que el resultat és correcte:

```
$ gcc SequentialAdd.c -o SeqAdd –lm
$ ./SeqAdd
Total addition is 20000000
```

3.2 Exercici 3: sumar les fileres concurrentment *ThreadsAdd.c*

Modifiqueu el codi proposat de manera que les crides a la funció `AddRow()` siguen concurrents. La figura 4 mostra un esquema de la distribució de fils resultant.

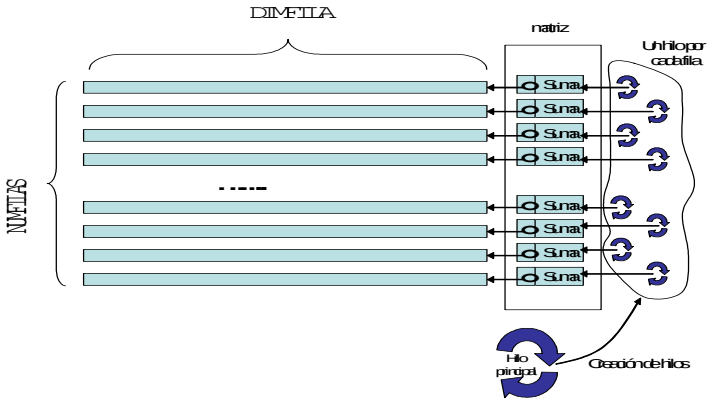

Figura 4: esquema que cal implementar en *ThreadsAdd.c*

Per a fer aquest exercici seguiu els passos següents:

- Copieu *SequentialAdd.c.c* en *ThreadsAdd.c*. Feu les modificacions sobre *ThreadsAdd.c*
- Canvieu les línies de codi que apareixen entre els comentaris `//EXERCICI2.a` i `//EXERCICI2.b` de manera que:
  - Hi haja un bucle de crides `pthread_create` per a crear les activitats concurrents. Com a primer paràmetre de la funció, feu servir el vector *fils* definit en *main*. No oblideu passar a cada thread el punter a la filera corresponent de la matriu que ha de sumar (el quart paràmetre de *pthread_create*)
  - Com que la suma total no pot calcular-se mentre no acaben tots els fils, introduïu un bucle de crides `pthread_join`.
- Compileu i executeu el codi. Comproveu que el resultat de la *suma total* siga el mateix que en el cas seqüencial.

```
$ gcc ThreadsAdd.c -o ThAdd –lm -lpthread
$ ./ThAdd
Total addition is 20000000
```

3.3 Exercici 4: comparar el temps d'execució; ordre *time*

Per a comprovar els temps d'execució de les dues versions del programa (*SeqAdd* i *ThAdd*) executeu l'ordre del *shell* `time`. L'ordre *time* executa l'ordre que hom li passa com a paràmetre i, després de l'execució d'aquesta darrera, informa sobre el **temps real d'execució**, el **temps** que el procés ha estat **executant instruccions en mode usuari** i el **temps en** què ha estat **en mode sistema**. Si en voleu més informació, consulteu les pàgines de manual de *time* .

```
$ man time
$ time  ./SeqAdd
$ time  ./ThAdd
```

- Comproveu els temps d'execució de *SeqAdd* i *ThAdd* i empleneu la taula

Sumant files	*SeqAdd*	*ThAdd*
Temps real d'execució		
Temps d'execució en mode usuari		
Temps d'execució en mode sistema		

Qüestió de l'exercici 4:

- Observeu les diferències i similituds dels resultats sobre temps d'execució pel que fa a l'execució seqüencial. Intenteu justificar el comportament observat.

3.4 Exercici 5: optimitzar en funció del nombre de nuclis (*cores*)

Es proposa optimitzar el codi de l'exemple de manera que es minimitze la sobrecàrrega associada a la creació i terminació dels fils, alhora que s'aprofita al màxim la disponibilitat de nuclis del processador per a aconseguir la màxima velocitat d'execució. Per a assolir aquest objectiu, el nombre de fils a cada moment ha de coincidir amb el nombre de nuclis del processador, i a més la càrrega associada a cada fil ha de ser aproximadament la mateixa.

Per a averiguar el nombre de nuclis del processador podeu executar l'ordre *top* y polsar la tecla 1 per a veure desglosada la càrrega de cadascun dels nuclis. També podeu consultar l'arxiu `/proc/cpuinfo`. Alguns exemples:

```
$ grep processor /proc/cpuinfo | wc -l
$ cat /proc/cpuinfo | grep "cpu cores"
```

Per a fer l'exercici proposat cal seguir els passos següents:

- Copieu *ThreadAdd.c* en *ThreadsAdd2.c*, i feu les modificacions sobre la còpia *ThreadsAdd2.c*
- Canvieu de nou les línies de codi que apareixen entre els comentaris `//EXERCICI2.a` i `//EXERCICI2.b` de manera que:
    - Hi haja hi un bucle de crides `pthread_create` per a crear les activitats concurrents amb tantes iteracions com a nuclis tinga el processador.
    - Cada fil creat ha de processar una part proporcional de les crides a `AddRow`. Cal tenir en compte que el nombre total de crides a `AddRow` pot no ser un múltiple exacte del nombre de nuclis, per la qual cosa algun fil pot haver d'executar una crida més que d'altres.
    - El bucle de crides `pthread_join` ha de tenir tantes iteracions com nuclis tinga el

processador.

- Compileu i executeu el codi. Comproveu que el resultat de la *suma total* és el mateix que en el cas seqüencial (`SeqAdd`).
- Comproveu els temps d'execució de *ThAdd* i *ThAdd2* i empleneu la taula següent:

Sumant files amb un nombre de fils igual al nombre de nuclis	*ThAdd*	*ThAdd2*
Temps real d'execució		
Temps d'execució en mode usuari		
Temps d'execució en mode sistema		

Qüestió de l'exercici 5:

- Observeu les diferències i similituds dels resultats sobre temps d'execució d'aquesta versió respecte a crear un fil per cada fila.

4. Treballant amb fils periòdics

En moltes ocasions els fils aprofiten per a dur a terme tasques que s'han de repetir periòdicament, és a dir cada cert interval de temps. Una manera senzilla d'aconseguir-lo és que el codi del fil incorpore una crida d'espera `sleep()` dins d'un bucle. Seguint aquesta idea, heu de completar un programa que faça una animació que recorde l'efecte conegut com "pluja digital" on una sèrie de caràcters aleatoris van apareguent periòdicament formant columnes descendents.

El codi de partida "matrix_basic.c" apareix en la Figura 5 i es basa en la creació d'un fil "Dibuixador" (`DrawCol`) per a cadascuna de les columnes de la pantalla. Cadascun d'aquests fils dibuixadors va completant la columna que té assignada escrivint els seus caràcters en una matriu bidimensional `m` compartida. Al mateix temps, altre fil "`Refresh`" (únic) va traslladant periòdicament el contingut complet d'aquesta matriu a la pantalla per a visualitzar l'animació.

4.1 Exercici 6: Animació mitjançant fils. Elaboreu un programa "matrix_draw.c" a partir del programa "matrix_basic.c", afegint a la seua funció main les crides necessàries per a crear un fil dibuixador per cadascuna de les columnes. Feu ús de la constant `COLUMNS` i de les variables globals que teniu ja definides per a albergar els identificadors dels fils. Feu servir la funció `DrawCol` com a cos del seus fils. Aquesta funció ha de rebre com argument (per valor) el número de columna assignat al fil (de 0 a `COLUMNS`–1). Creeu també un fil refrescador (la funció `Refresh` no utiliza el valor que rep com argument). Assegureu-vos a més a més d'esperar la finalització de tots els fils dibuixadors, de forma que el programa termine quan s'hi haja fet el dibuix de totes les columnes (observareu que algunes d'elles no s'hi dibuixen per a imitar l'efecte original).

Qüestió Exercici 6:

- ¿Ha d'esperar també la finalització del fil `Refresh`?

4.2 Exercici 7: Fils que creen altres fils. Examineu el codi de la funció `DrawCol` i afegiu-hi el que calga per aconseguir que cada fil dibuixador cree un fil "`EraseCol`" quan el seu bucle de fileres arribe a la meitat del seu recorregut. Cada dibuixador ha de passar-li com argument al seu esborrador corresponent el número de columna que té assignada. El fil esborrador, ja elaborat, s'encarregarà d'anar fent desaparèixer el contingut de la columna des de la part superior, al mateix temps que el dibuixador termina de completar-la. Asegureu-vos de que cada dibuixador espera la finalització del seu company esborrador en el moment adient. D'aquesta forma, el programa ha de finalitzar quan es complete el dibuixat i l'esborrament de totes les columnes (la pantalla ha d'haver quedat completament esborrada). Guardeu el nou programa amb el nom "matrix_erase.c".

Qüestió Exercici 7:

- ¿Cal que la funció *main* espere també la finalització dels fils esborradors per a obtenir el comportament demanat? ¿Per què?

4.3 Exercici 8. Tots els fils es creen al mateix nivell: "germans". Obriu un nou terminal i mentre en un s'executa l'exercici 7 comproveu en l'altre, amb l'ordre $ps -alT que tots els fils són qermans, sense cap relació jeràrquica entre ells, amb independència de si han sigut creats des del fil principal o no.

4.4. Exercici 9. Completant l'animació. A partir de l'exercici 7, modifiqueu la funció del fil `DrawCol` de forma que es repeteixca indefinidament la seqüència Dibuixar-Esborrar per aconseguir un efecte de "pluja digital" complet. Guardeu el vostre programa amb el nom `matrix_complete.c`.

Qüestió Exercici 9:

- Els fils dibuixadors ara no terminen mai. Elimineu per tant el codi d'espera de finalización d'aquests fils en la funció main. ¿Per quin altre codi caldrà substituir les línies eliminades perquè l'animació no acabe abruptament?

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

#define COLUMNS 80
#define ROWS 25

char m[ROWS][COLUMNS];
long delay[COLUMNS];
int row_b[COLUMNS];

pthread_attr_t attrib;
pthread_t  draw_thread[COLUMNS];
pthread_t  erase_thread[COLUMNS];
pthread_t  refresh_thread;


void *EraseCol(void *ptr) {
  int row, col=(int)(long)ptr;

  for (row=0; row<ROWS; row++) {
    m[row][col]= ' '; // Write space
    usleep(delay[col]); // Wait before the following erase
  }
}

void *DrawCol(void *ptr) {
  int row, col=(int)(long)ptr;

  delay[col]= 50000+rand()%450000; // Random delay: 0,05s to 0,5s
  if (rand()%10 > 4) { // Sometimes do not draw column
    usleep(delay[col]*ROWS); // Wait without drawing
  } else {
    for (row=0; row<ROWS; row++) {
      row_b[col] = row;
      m[row][col] = 32+rand() % 94; // Write random char
      usleep(delay[col]); // Wait before next char
    }
  }
}

void *Refresh (void *ptr) {
  int row, col;
  char order[20];

  while(1) {
    write(1,"\033[1;1f\033[1;40;32m",16); // Back to left-up corner, Green text
    for (row=0; row<ROWS; row++) {
      write(1,m[row],COLUMNS); write(1,"\n",1); // Refresh row
    }
    write(1,"\033[1;37m",7); // White text
    for (col=0; col<COLUMNS; col++) {
      sprintf(order,"\033[%d;%df%c",row_b[col]+1,col+1,m[row_b[col]][col]);
      // Rewrite in white the last character in column col
      if (row_b[col]<ROWS-1) write(1,order,strlen(order));
    }
    usleep(100000); // Wait 0,1s before refreshing again
  }
}
```

```
int main()
{
  int col;
  memset (m,' ', ROWS*COLUMNS); // Erase matrix m
  write(1,"\033[2J\033[?25l",10); // Clean screen and hide cursor

  pthread_attr_init(&attrib);

  // Create a drawing thread for every column

  // Create a screen refresh thread

  // Wait for drawing threads ending

  write(1,"\033[0m\033[?25h\r",11); // Reset usual text and cursor
}
```

Figura 5: Codi del programa matrix_basic.c

Anexe. Notes sobre matrix_basic.c

Ajuda per a la creació dels fils DrawCol y EraseCol

Cal destacar que les funcions de fil `DrawCol` i `EraseCol` esperen rebre per valor el nombre de columna sobre la qual han de treballar. Això obligarà a fer un casting de la variable de bucle de columnes `col` en l'últim paràmetre de la crida `pthread_create()` que haurà d'incloure en la funció main:

```
(void*)(long)col
```

Aquest doble casting converteix el valor de la variable `col` al tipus punter que la funció de fil necessita. La conversió intermitja `(long)` evita que el compilador mostre un avís sobre la diferència entre la grandària dels tipus origen i destí.

Sobre les seqüències d'escapament per al terminal

El programa fa ús de codis d'escapament de l'estándard ANSI X3.64 per a dur a terme operacions com l'esborrat de la pantalla, el moviment del cursor o el canvi de color del text.