Programació amb fils (1a part)

Lluís Garrido – lluis.garrido@ub.edu

Novembre 2014

Resum

La interfície POSIX-1.2001, també coneguda amb el com de "POSIX *threads*" o simplement "*pthreads*" defineix la interfície estàndard per a manipulació de fils[1]. Els fils ens permeten realitzar múltiples tasques a la vegada dintre de l'entorn d'un mateix procés. En aquesta fitxa ens centrarem només en els aspectes de creació i acabament de fils. La sincronització de múltiples fils per realitzar una tasca comuna s'explicarà a la segona part d'aquesta fitxa.

Índex

[1] POSIX (*Portable Operating System Interface*) fa referència a una família d'estàndards desenvolupats per l'IEEE (*Institute of Electrical and Electronics Engineers*) amb l'objectiu de promoure la portabilitat d'aplicacions entre sistemes UNIX. L'estàndard defineix els serveis que un sistema operatiu ha d'implementar perquè es pugui dir que compleix amb l'estàndard POSIX. Una de les característiques d'aquest estàndard és que només defineix la interfície a implementar, però no indica com s'ha d'implementar.

1 Fonaments dels fils

1.1 La llibreria pthreads

Per poder utilitzar la llibreria *pthreads* hem d'incloure al codi C la següent instrucció

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

A l'hora de compilar, ho hem de fer així

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

Observeu l'opció "`-lpthread`" que indica al compilador que ha d'enllaçar amb la llibreria de fils.

1.2 Identificació de fils

Cada fil té el seu propi identificador, de la mateixa forma que cada procés té el seu propi identificador. Però mentre l'identificador de procés té sentit en el context del sistema operatiu, l'identificador de fil només té sentit dintre del procés al qual pertany.

En un procés l'identificador de procés és un tipus *pid_t*, un sencer no negatiu. Un identificador de fil es representa amb el tipus *pthread_t*, que no necessàriament ha de ser un sencer sinó que pot ser una estructura. És per això que per fer aplicacions portables no podem suposar que el tipus *pthread_t* correspon a un sencer.

Per això es fa necessari utilitzar funcions per comparar l'identificador de dos fils.

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

La funció retorna un valor no nul si els identificadors són iguals, i zero en cas contrari. Una conseqüència que *pthread_t* pugui ser una estructura es que no hi ha cap forma portable d'imprimir per pantalla el seu valor. A vegades és útil imprimir els identificadors de fil per analitzar l'execució dels fils (per fer un *debugging*), però generalment no hi ha aquesta necessitat.

Un fil pot obtenir el seu propi identificador si crida a la funció *pthread_self*.

```
pthread_t pthread_self(void)
```

1.3 Creació de fils

Quan s'executa un programa el sistema operatiu crea un procés amb un sol fil d'execució. Es poden crear fils addicionals amb la instrucció *pthread_create*.

```
int pthread_create(pthread_t *tidp, pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

La funció retorna zero si el fil es pot crear satisfactòriament, i un sencer no nul (que identifica l'error) en cas que no s'hagi pogut crear el fil. Vegem-ne un exemple abans d'explicar els paràmetres de la funció *pthread_create*.

Aquest exemple correspon al codi `creacio_fils.c`, veure Figura 1. Per compilar aquest exemple cal executar aquesta instrucció

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

pthread_t ntid;

void printids(const char *s)
{
  pid_t pid;
  pthread_t tid;

  pid = getpid();
  tid = pthread_self();
  printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
      (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
  printids("nou fil: ");
  return((void *)0);
}

int main(void)
{
  int err;

  err = pthread_create(&ntid, NULL, thr_fn, NULL);
  if (err != 0) {
    printf("no puc crear el fil.\n");
    exit(1);
  }
  printids("fil principal:");
  sleep(1);
  exit(0);
}

```

Figura 1: Codi `creacio_fils.c`, veure secció 1.3.

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

Analitzem a continuació els arguments de la crida *pthread_ create*. El primer argument és un punter a l'identificador del fil. En cas que la funció *pthread_ create* retorni de forma satisfactòria *ntid* contindrà l'identificador del fil que s'ha creat. El segon paràmetre es pot utilitzar per personalitzar diversos atributs del fil. Aquí el posarem a *NULL* per crear un fil amb els atributs per defecte.

El nou fil comença a executar-se a la funció especificada al tercer argument de *pthread_ create*, que en aquest cas és *thr_ fn*. Direm que *thr_ fn* és la funció d'entrada al fil. Observeu que el tercer argument és un punter a una funció (vegeu la fitxa 1, darrera secció, per recordar el que és un punter una funció). La funció d'entrada a un fil només admet un únic argument, *arg*, que s'especifica al quart argument de la funció *pthread_ create* i que és un punter sense tipus associat (vegeu a la fitxa 1 l'exemple del *quicksort*). Veure la secció 2.1 i 2.2 per veure com es pot passar un sencer i una estructura, respectivament, a cada fil.

Quan s'executa un nou fil no hi ha cap garantia sobre quin fil s'executarà primer: el nou fil que s'acaba de crear o bé el fil que ha fet la crida a *pthread_ create* (vegeu secció 2.1 per veure un exemple). A continuació es mostra la sortida d'executar dos cops seguits l'anterior programa.

```
$ ./creacio_fils
fil principal: pid 7962 tid 405145344 (0x18260700)
nou fil:  pid 7962 tid 397264640 (0x17adc700)
$ ./creacio_fils
fil principal: pid 7964 tid 3568469760 (0xd4b28700)
nou fil:  pid 7964 tid 3560589056 (0xd43a4700)
```

Observeu que els dos fils tenen el mateix identificador de procés (*pid*) però identificadors de fil (*tid*) diferents. Depenent d'on s'executi aquesta aplicació (Mac, Solaris, FreeBSD) el rang de valors que s'imprimeixen pot ser diferent.

Hi ha uns detalls que cal comentar respecte l'exemple anterior i en què entrarem a la següent secció: el primer és el fet la funció *exit* fa que el procés finalitzi (incloent tots els seus fils) i per això és necessari posar a dormir el fil principal durant un segon, per assegurar que el fil que s'acaba de crear tingui temps d'executar-se. El segon detall és el fet que el fil que s'ha creat ha de cridar a *thread_ self* per saber quin és el seu identificador tot i que aquest identificador s'emmagatzema a la variable global *ntid*. La raó és que el nou fil pot començar a executar-se abans que la funció *pthread_ create* retorni i en aquest cas el nou fil podria veure la variable *ntid* no inicialitzada. Finalment, observeu que la funció d'entrada al fil retorna un punter al tipus genèric *void*. Això no vol dir que la funció no pugui retornar cap informació sinó que la funció pot retornar un punter a una estructura qualsevol d'informació. A les seccions 2.3 i 2.4 veurem dos exemples de com fer-ho.

1.4 Acabament de fils

Si un fil qualsevol crida a la funció *exit* aleshores el procés sencer finalitza (encara que els altres fils no hagin finalitzat). A més, cal tenir en compte que retornar un nombre sencer de la funció *main* és equivalent a cridar la funció *exit* amb el mateix valor. Per tant, des de la funció *main* és equivalent fer

```
exit(0);
```

o bé

```
return(0);
```

on el sencer zero és el codi de sortida del procés[2]. Això significa que el procés finalitza (incloent tots els seus fils que estiguin executant en aquell moment) així que el fil principal retorna de la funció *main*.

Un fil pot finalitzar de tres formes diferents sense finalitzar el procés.

1. El fil simplement retorna de la funció des de la qual ha començat a executar-se. El valor que retorna la funció és el codi de sortida del fil i que pot ser capturat mitjançant la funció *pthread_join*.

2. El fil pot ser cancel·lat per un altre fil del mateix procés.

3. El fil pot cridar a *pthread_exit*.

   ```
   void pthread_exit(void *rval_ptr);
   ```

   L'argument *rval_ptr* és un punter sense tipus associat, similar a l'argument que es passa a la funció d'entrada d'un fil.

Un fil pot esperar a que un altre fil finalitzi amb la funció *pthread_join*.

```
int pthread_join(pthread_t tid, void **rval_ptr);
```

La funció retorna zero si s'executa satisfactòriament, i un valor no nul en cas de fallada. El fil que fa la crida a *pthread_join* quedarà bloquejat fins que el fil amb identificador *tid* crida a *pthread_exit*, retorna de la seva funció d'entrada o és cancel·lat per un altre fil. En cas que el fil cridi a *pthread_exit*, el valor *rval_ptr* retornat a *pthread_join* apuntarà al *rval_ptr* especificat com a argument a *pthread_exit*. Si el fil retorna de la seva funció d'entrada amb un determinat valor, *rval_ptr* contindrà el aquest valor retornat. Si el fil amb identificador *tid* ha estat cancel·lat, la direcció de memòria especificada per *rval_ptr* contindrà el valor *PTHREAD_CANCELED*.

En cas que no estiguem interessats en el valor d'acabament del fil podem posar l'argument *rval_ptr* de *pthread_join* a *NULL*. En aquest cas el fil que fa la crida a *pthread_join* esperarà que el fil especificat al primer argument finalitzi, però no podrà recuperar el valor d'acabament del fil.

Tot i que no sembla obligatori l'ús d'aquesta funció, és recomana utilitzar-la sempre. D'una banda, evita sorpreses desagradables i d'altra sembla que la funció *pthread_join* allibera memòria que és reservada per *pthread_create*. A la secció 2.3 i 2.4 trobareu dos exemples que us mostren com es retorna informació des d'un fil.

Un fil pot demanar a un altre fil del mateix procés que es cancel·li (és a dir, finalitzi) cridant a la funció *pthread_cancel*.

```
int pthread_cancel(pthread_t tid);
```

[2]El valor sencer retornat per la funció *main* pot ser llegit pel procés pare per saber, per exemple, si el procés s'ha executat correctament.

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

pthread_t ntid[10];

void *thr_fn(void *arg)
{
  int i = (int) arg;
  printf("El fil amb ID %d te assignat el sencer %d\n", pthread_self(), i);
  return((void *)i);
}

int main(void)
{
  int i, err;
  void *tret;

  for(i = 0; i < 10; i++) {
    err = pthread_create(ntid+i, NULL, thr_fn, (void *) i);
    if (err != 0) {
      printf("no puc crear el fil.\n");
      exit(1);
    }
  }

  for(i = 0; i < 10; i++) {
    err = pthread_join(ntid[i], &tret);
    if (err != 0) {
      printf("error pthread_join amb fil %d\n", i);
      exit(1);
    }
    printf("El fil %d m'ha retornat el numero %d\n", i, (int) tret);
  }

  return 0;
}

```

Figura 2: Codi `passar_sencers_fils.c`, veure secció 2.1.

Per defecte la funció *pthread_ cancel* farà que el fil especificat amb identificador *tid* finalitzi com si hagués cridat a *pthread_ exit* amb argument *PTHREAD_ CANCELED*. El fil que es cancel·la pot ignorar aquesta petició (depèn dels paràmetres associats al fil) o bé controlar com se'l cancel·la (per exemple, per alliberar memòria dinàmica abans de finalitzar realment el fil). És important mencionar que la funció *pthread_ cancel* no fa que el fil que es cancel·la finalitzi en el moment de fer la crida. El fil que fa la crida a *pthread_ cancel* tampoc es bloqueja fins que el fil especificat finalitzi, sinó que simplement posa la petició de cancel·lació en una cua i continua executant. A la bibliografia trobareu més informació al respecte.

2 Exemples

2.1 Passar un paràmetre escalar a cada fil

Hem vist a la secció 1.2 que l'estàndard POSIX utilitza el tipus *pthread_ t* per representar l'identificador d'un fil. Aquest identificador no es pot tractar com a un sencer, ja que cada sistema operatiu (sigui Linux, FreeBSD o MAC, per exemple) pot tenir un tipus diferent associat a *pthread_ t*, que pot ser una estructura.

En dissenyar una aplicació multifil pot ser necessari tenir un identificador sencer associat al fil.

Aquest sencer va des de 0 fins a $N-1$, on N és el nombre de fils que es creen. L'identificador retornat per la funció *pthread_ self()* (vegeu secció 1.2) no ens serà útil. Al següent codi es mostra com crear deu fils i passar a cadascun d'ells un sencer diferent del 0 al 9. En concret, el fil principal crea deu fils passant per paràmetre el seu identificador sencer. Un cop s'han creat els 10 fils el fil principal espera que aquests acabin. El codi s'anomena `passar_sencers_fils.c` i es pot veure a la Figura 2.

Observeu com es passa un sencer a la funció d'entrada del fil: gràcies al *casting* ho podem aconseguir. Per passar el sencer a la funció *thr_fn* es fa *(void *) i*. Un cop a la funció *thr_fn* es recupera el sencer fent *(int) arg*. Cal anar molt amb compte a l'hora de fer un *casting* ja que forcem canviar d'un tipus (en aquest cas, un sencer) a un altre (un punter genèric). En arquitectures de 32 bits un punter té una mida de 32 bits i un sencer també té una mida de 32 bits. Per tant, aquí no hi ha problema a l'hora de fer els *castings*. En arquitectures de 64 bits un punter té una mida de 64 bits mentre que un sencer té una mida de 32 bits. En aquest cas tampoc hi ha problema en realitzar l'assignació d'un sencer a un punter.

Aquí hi ha un exemple d'execució del programa. Observeu que els fils poden executar en qualsevol ordre.

```
$ ./passar_sencers_fils
El fil amb ID 1267054336 te assignat el sencer 3
El fil amb ID 1275447040 te assignat el sencer 2
El fil amb ID 1292232448 te assignat el sencer 0
El fil amb ID 1258661632 te assignat el sencer 4
El fil amb ID 1283839744 te assignat el sencer 1
El fil amb ID 1233483520 te assignat el sencer 7
El fil amb ID 1225090816 te assignat el sencer 8
El fil 0 m'ha retornat el numero 0
El fil 1 m'ha retornat el numero 1
El fil 2 m'ha retornat el numero 2
El fil 3 m'ha retornat el numero 3
El fil 4 m'ha retornat el numero 4
El fil amb ID 1241876224 te assignat el sencer 6
El fil amb ID 1216698112 te assignat el sencer 9
El fil amb ID 1250268928 te assignat el sencer 5
El fil 5 m'ha retornat el numero 5
El fil 6 m'ha retornat el numero 6
El fil 7 m'ha retornat el numero 7
El fil 8 m'ha retornat el numero 8
El fil 9 m'ha retornat el numero 9
```

2.2 Passar una estructura a cada fil

Aquest exemple mostra com es pot passar una estructura amb dades diferents a cada fil. L'exemple és el mateix que abans (secció 2.1) només que ara es passa una estructura. L'exemple correspon al codi `passar_estructura_fils.c` i es pot veure a la Figura 3.

Observar que de nou es fa ús el *casting* per tal de convertir un punter a l'estructura a un punter tipus *void*. Observar que aquesta operació és correcte: només s'està fent una assignació de punters.

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

pthread_t ntid[10];

struct parametres {
  int i;
  int j;
  char *str;
};

void *thr_fn(void *arg)
{
  struct parametres *par = (struct parametres *) arg;
  printf("El fil amb sencer %d te assignat la j = %d i str = %s\n",
                par->i, par->j, par->str);
  return NULL;
}

int main(void)
{
  int i, err;
  void *tret;
  struct parametres *par;

  for(i = 0; i < 10; i++) {
    par = malloc(sizeof(struct parametres));
    par->i = i;
    par->j = 10 - i;
    par->str = malloc(sizeof(char) * 10);
    sprintf(par->str, "Hola %d", i);
    err = pthread_create(ntid+i, NULL, thr_fn, (void *) par);
  }

  for(i = 0; i < 10; i++) {
    err = pthread_join(ntid[i], &tret);
  }

  return 0;
}

```

Figura 3: Codi `passar_estructura_fils.c`, veure secció 2.2.

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

void *thr_fn(void *arg)
{
  int valor = 1;
  printf("fil 1 retorna 1\n");
  return((void *)valor);
}

int main(void)
{
  int err;
  pthread_t tid1;
  void *tret;

  err = pthread_create(&tid1, NULL, thr_fn, NULL);
  if (err != 0) {
    printf("no puc crear fil 1.\n");
    exit(1);
  }

  err = pthread_join(tid1, &tret);
  if (err != 0) {
    printf("error pthread_join al fil 1.\n");
    exit(1);
  }
  printf("codi de sortida del fil 1: %d\n", (int)tret);

  return 0;
}

```

Figura 4: Codi retornar_sencer.c, veure secció 2.3.

No seria vàlid intentar passar com a argument alguna variable amb una mida més gran que un punter tipus *void*. Hem de suposar el pitjor cas que implica un punter de 32 bits. En aquest exemple l'estructura *parametres* té una mida superior a 32 bits, però cal fer notar de nou que s'està passant com a argument un punter a l'estructura, no l'estructura en sí.

La sortida de l'execució es mostra a continuació.

```
$ ./passar_estructura_fils
El fil amb sencer 0 te assignat la j = 10 i str = Hola 0
El fil amb sencer 2 te assignat la j = 8 i str = Hola 2
El fil amb sencer 1 te assignat la j = 9 i str = Hola 1
El fil amb sencer 6 te assignat la j = 4 i str = Hola 6
El fil amb sencer 7 te assignat la j = 3 i str = Hola 7
El fil amb sencer 8 te assignat la j = 2 i str = Hola 8
El fil amb sencer 5 te assignat la j = 5 i str = Hola 5
El fil amb sencer 9 te assignat la j = 1 i str = Hola 9
El fil amb sencer 3 te assignat la j = 7 i str = Hola 3
El fil amb sencer 4 te assignat la j = 6 i str = Hola 4
```

2.3 Retornar sencers des d'un fil

En el següent exemple es mostra com es pot capturar el valor d'acabament d'un fil amb *pthread_join*. Correspon al codi `retornar_sencer.c`, veure Figura 4. El procediment és similar a l'utilitzat per

passar un sencer al fil: en concret, en retornar del punt d'entrada d'un fil es pot retornar un punter. Podem utilitzar l'emmagatzematge del punter per guardar-hi un sencer.

2.4 Retornar estructures des d'un fil

Hem de tenir en compte que quan un fil surt de la seva funció d'entrada s'elimina la pila associada al fil. S'ha d'anar en compte doncs a l'hora de retornar dades al fil principal (el fil que espera al *pthread_join*) ja que ens hem d'assegurar que la memòria utilitzada per l'estructura és vàlida quan es finalitzi la crida a la funció. Si l'estructura s'emmagatzema a la pila d'un fil que finalitza, el contingut de la memòria pot haver-se modificat en el moment en què s'intenta accedir a l'estructura. Al codi de l'exemple de la figura 5 veiem un codi en què un fil "reserva" memòria per a una estructura a la seva pila i retorna un punter d'aquesta estructura. La pila del fil que finalitza és destruïda i, per tant, la memòria associada a l'estructura pot estar sent utilitzada per una altra cosa en el moment en què el que el fil principal intenta accedir-hi.

En el codi el fil principal crea un primer fil que "reserva" a la pila memòria per a una estructura *foo* i imprimeix el contingut per pantalla. Aleshores el primer finalitza retornant com un punter a l'estructura *foo*. El fil principal recupera aquest punter amb *pthread_join* a la variable *fp*. A continuació el fil principal crea un segon fil, s'espera que acabi i el fil principal mostra per pantalla el contingut de l'estructura *fp*, que idealment hauria de mostrar el contingut de *foo*.

En executar aquest programa obtenim

```
$ ./retonar_estructura_problema
fil 1:   estructura ubicada a la direccio 0x155d7ed0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
El fil principal crea un segon fil.
fil 2 amb identificador 358450944
fil principal:   estructura ubicada a la direccio 0x155d7ed0
  foo.a = 4196958
  foo.b = 0
  foo.c = 0
  foo.d = 0
```

Com es pot veure, el contingut de l'estructura (ubicada a la pila del primer fil) ha canviat en el moment en què el fil principal intenta accedir a l'estructura. En aquest cas la pila del segon fil ha sobreescrit els continguts de la pila del primer fil.

Per resoldre aquest problema es pot utilitzar una estructura global o bé reservar memòria per a l'estructura amb *malloc*. A la Figura 6 es presenta la forma correcta de la funció *thr_fn1* que utilitza *malloc*. Proveu de compilar el codi utilitzant aquesta funció i veureu que el resultat és correcte.

2.5 Computar el temps d'execució d'un programa

A moltes aplicacions interessa calcular el temps d'execució d'una aplicació. En aquesta secció es mostra una forma per fer-ho.

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

struct foo {
  int a, b, c, d;
};

void printfoo(const char *s, const struct foo *fp)
{
  printf(s);
  printf("  estructura ubicada a la direccio 0x%x\n",
            (unsigned)fp);
  printf("  foo.a = %d\n", fp->a);
  printf("  foo.b = %d\n", fp->b);
  printf("  foo.c = %d\n", fp->c);
  printf("  foo.d = %d\n", fp->d);
}

void *thr_fn1(void *arg)
{
  struct foo foo = {1, 2, 3, 4};

  printfoo("fil 1:\n", &foo);
  return ((void *)&foo);
}

void *thr_fn2(void *arg)
{
  printf("fil 2 amb identificador %d\n", pthread_self());
  return ((void *)0);
}

int main(void)
{
  int err;
  pthread_t tid1, tid2;
  struct foo *fp;

  err = pthread_create(&tid1, NULL, thr_fn1, NULL);
  err = pthread_join(tid1, (void *)&fp);

  sleep(1);

  printf("El fil principal crea un segon fil.\n");
  err = pthread_create(&tid2, NULL, thr_fn2, NULL);
  sleep(1);
  printfoo("fil principal:\n", fp);
  return 0;
}

```

Figura 5: Codi `retornar_estructura_problema.c`, veure secció 2.4.

```c
void *thr_fn1(void *arg)
{
  struct foo *foo;

  foo = malloc(sizeof(struct foo));

  foo->a = 1; foo->b = 2; foo->c = 3; foo->d = 4;

  printfoo("fil 1:\n", foo);
  return((void *)foo);
}
```

Figura 6: Codi correcte per retornar una estructura des d'un fil.

Hi ha dues mesures que podem prendre

- La primera, és el temps de CPU utilitzat per l'aplicació en executar. Suposem que tenim una aplicació d'un sol fil que triga dos segons en executar-se en una sola CPU. Després de fer l'aplicació, aquesta triga 1 segon fent servir dues CPUs. El temps total que triga l'aplicació multifil continua sent 2 segons, ja que ocupa 1 segon a cada CPU. Aquest temps de CPU es pot mesurar amb la funció *clock* de la lliberia C.
- La segona, és el temps cronològic que triga l'aplicació en executar. Suposem de nou que tenim una aplicació multifil que s'executa en 1 segon fent servir dues CPUs. El temps cronològic d'execució és d'un segon. Aquest temps es pot mesurar fent servir la funció *gettimeofday* de la llibreria C.

Anem a veure a continuació un exemple d'ús d'aquestes dues funcions amb l'exemple de càlcul del valor de π vist a classe de teoria. L'exemple de la Figura 7 mostra com es pot mesurar el temps de CPU així com el temps cronològic d'execució d'un programa. Per a l'exemple tractat l'execució resulta en

```
$ ./calcul_pi_fils
Valor de pi: 3.141593
Temps de CPU: 1.640000 seconds
Temps cronologic = 0.844294 seconds
```

Observar que el temps cronològic d'execució es aproximadament la meitat del de CPU. En aquest cas, però, el temps cronològic té sentit ja que no hi ha cap altre procés a l'ordinador que consumeixi molts recursos de CPU.

3 Bibliografia

1. Rochkind, M.J. Advanced Unix Programming. Addison Wesley, 2004.
2. Stevens, W.R. Advanced Programming in the Unix Environment. Addison Wesley, 2005.

```c
#define NUM_FILS  2
#define NUM_RECTS 100000000

double result[NUM_FILS];

double integral(int id)
{
  int i;
  double mid, height, width, sum = 0.0;
  double area;

  width = 1.0 / (double) NUM_RECTS;
  for(i = id; i < NUM_RECTS; i += NUM_FILS) {
    mid = (i + 0.5) * width;
    height = 4.0 / (1.0 + mid * mid);
    sum += height;
  }
  area  = width * sum;
  return area;
}

void *thread_fn(void *arg)
{
  int i = (int) arg;
  result[i] = integral(i);
  return ((void *)0);
}

int main(void)
{
  struct timeval tv1, tv2; // Cronologic
  clock_t t1, t2; // CPU
  pthread_t ntid[NUM_FILS];

  double valor_pi;
  int i;

  gettimeofday(&tv1, NULL);
  t1 = clock();

  for(i = 0; i < NUM_FILS; i++)
     pthread_create(&(ntid[i]), NULL, thread_fn, (void *) i);

  for(i = 0; i < NUM_FILS; i++)
     pthread_join(ntid[i], NULL);

  valor_pi = 0.0;
  for(i = 0; i < NUM_FILS; i++)
    valor_pi += result[i];

  gettimeofday(&tv2, NULL);
  t2 = clock();

  printf("Valor de pi: %f\n", valor_pi);
  printf("Temps de CPU: %f seconds\n",
         (double)(t2 - t1) / (double) CLOCKS_PER_SEC);
  printf("Temps cronologic = %f seconds\n",
         (double) (tv2.tv_usec - tv1.tv_usec) / 1000000 +
         (double) (tv2.tv_sec - tv1.tv_sec));
}

```

Figura 7: Codi `calcul_pi_fils.c`, veure secció 2.5.