Universidad de Antioquia
Facultad de Ingeniería
Sistemas Operativos y Laboratorio

UNIVERSIDAD DE ANTIOQUIA
Facultad de Ingeniería

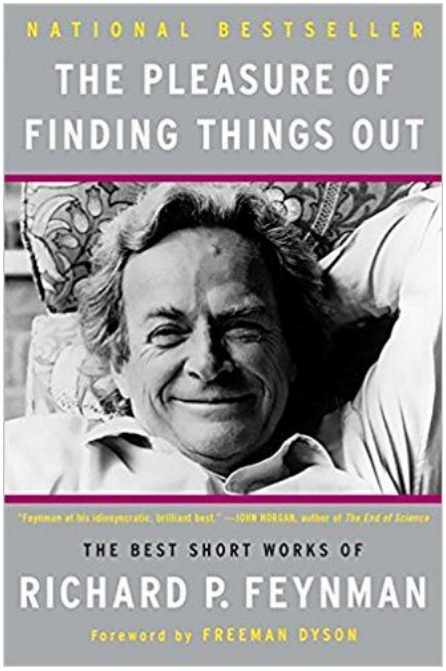
Laboratorio 4 - Hilos

Objetivos

- Explorar el uso de hilos como herramienta básica para aprovechar la concurrencia en los procesos.
- Conocer y usar las principales herramientas para la gestión de hilos en la librería pthread.

0. Motivación

What happened to the prejudice of two years ago, which was that the parallel programming is difficult? It turns out that what was difficult, and almost impossible, is to take an ordinary program and automatically figure out how to use the parallel computation effectively on that program. Instead, one must start all over again with the problem, appreciating that we have the possibility of parallel calculation, and rewrite the program completely with a new [understanding of] what is inside the machine. It is not possible to effectively use the old programs. They must be rewritten. That is a great disadvantage to most industrial applications and has met with considerable resistance. But the big programs usually belong to scientists or other, unofficial, intelligent programmers who love computer science and are willing to start all over again and rewrite the program if they can make it more efficient. So what's going to happen is that the hard programs, vast big ones, will be the first to be re-programmed by experts in the new way, and then gradually everybody will have to come around, and more and more programs will be programmed that way, and programmers will just have to learn how to do it[1].



[1] **Capítulo 2 del libro:** The Pleasure of Finding Things Out

1. Hilos

Los hilos son el medio por el cual un proceso puede implementar concurrencia. Éstos permiten que un proceso posea varios caminos de ejecución y, a su vez, compartan cierta región de memoria de manera directa. Por otro lado, en un sistema con KLT (Kernel Level Thread) y con multiprocesamiento o multithreading (por ejemplo, el hyperthreading de Intel), los hilos aprovechan al máximo los recursos de hardware disponibles y aumentan considerablemente el desempeño de una aplicación.

1.2. Librería *pthread* para la implementación de Hilos en C bajo Linux

Cada sistema operativo tiene, generalmente, una forma diferente de implementar los hilos a bajo nivel, sin embargo cuando estamos interactuando con un Lenguaje de Alto Nivel (LAN), éstos poseen o usan librerías que permiten abstraer aún más el problema. Cómo este curso está basado en herramientas libres y el sistema Linux, nosotros en la presente guía de laboratorio trabajaremos la implementación de hilos a través del lenguaje de programación C, la librería ***pthread*** y bajo el sistema operativo Linux. Recomendamos que el estudiante explore otros lenguajes, librerías y SO y se sugiere el siguiente material ([2]).

2. Ejercicios

Compile los siguientes ejercicios, analice el código y la salida. Para la compilar el código con la librería pthread es necesario notificarle al compilador el uso de la librería pthread, mediante la opción **–lpthread**, como se muestra este ejemplo:

```
gcc codigo.c –o ejecutable -lpthread
```

2.1. Creación de un hilo

Para la creación de un hilo se emplea la función **`pthread_create`** cuya descripción se encuentra con más detalle en el apéndice de la guía. Respecto al caso de los procesos existe una similitud (obviamente sin olvidar los detalles y aspectos conceptuales de fondo) entre las

[2] [] Java2 API – Thread. Available online: http://download.oracle.com/javase/1.3/docs/api/java/lang/Thread.html. Last visited: 22/09/11

funciones **`pthread_create`** y **`fork`** tal y como se muestra en el siguiente gráfico:

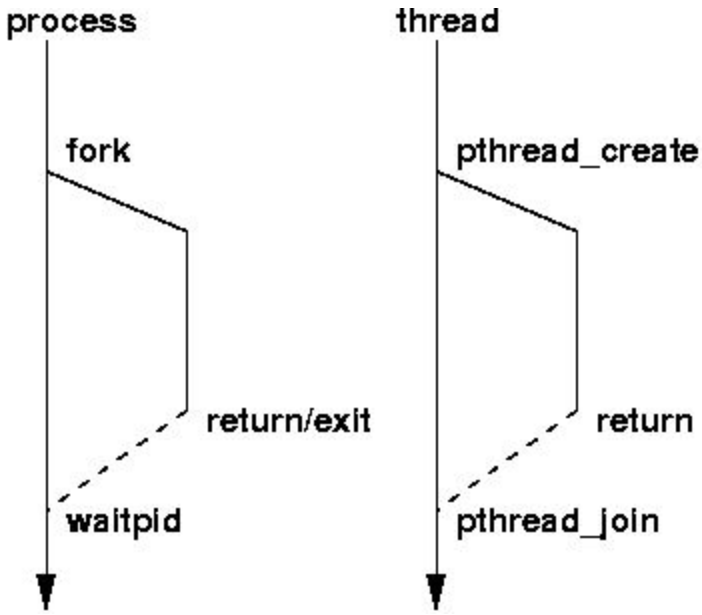


Figura 1. Similitudes proceso e hilo[3].

A continuación se muestra un código ejemplo.

Ejemplo 1: El siguiente código emplea la función **`pthread_create`** para mostrar la creación de hilos.

```
/**************************************************************************
* Code listing from "Advanced Linux Programming," by CodeSourcery LLC  *
* Copyright (C) 2001 by New Riders Publishing                           *
* See COPYRIGHT for license information.                                *
**************************************************************************/

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

/* Prints x's to stderr.  The parameter is unused.  Does not return.  */

void* print_xs (void* unused)
```

[3] Imagen tomada de http://www.cs.fsu.edu/~baker/realtime/restricted/notes/pthreads.html

```
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread_id;
  /* Create a new thread.  The new thread will run the print_xs
     function.  */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Print o's continuously to stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

Nota: Puede finalizar la ejecución con **<Ctrl + C>**.

Preguntas:

1. ¿Cuáles son y para qué sirven los argumentos de la función pthread_create? ¿Cómo funcionan estos para este ejemplo en especial?[4]
2. ¿Cómo es la salida en pantalla? ¿Cuál es la razón para este tipo de salida?

2.2. Varios Hilos y modo de ejecución

Tal y como en secciones previas del curso, al igual que en los procesos es posible crear varios hilos pues la función **pthread_create** puede ser llamada tantas veces como se desee dentro del código (en el cual se encuentra el hilo principal). La siguiente gráfica aclara esto:

[4] Puede guiarse del anexo pero la descripción tiene que ser con sus palabras.

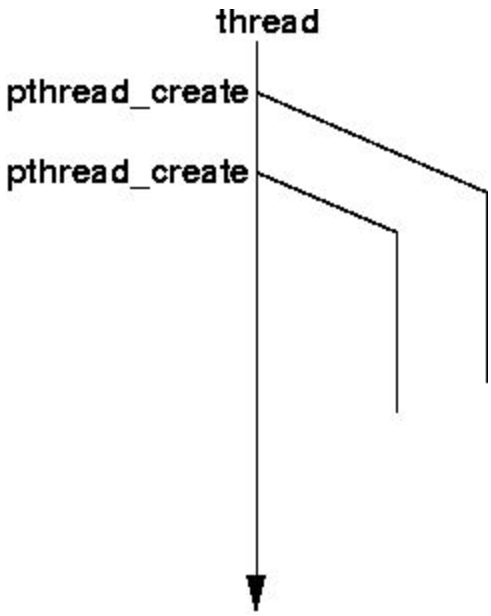


Figura 1. Creación de varios hilos[5].

El siguiente ejemplo muestra este caso.

Ejemplo 2: En el siguiente ejemplo se crean dos hilos a partir de un hilo principal.

```
/**************************************************************************
* Code listing from "Advanced Linux Programming," by CodeSourcery LLC  *
* Copyright (C) 2001 by New Riders Publishing                          *
* See COPYRIGHT for license information.                               *
**************************************************************************/

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

/* Parameters to print_function.  */

struct char_print_parms
{
  /* The character to print.  */
  char character;
  /* The number of times to print it.  */
  int count;
};

/* Prints a number of characters to stderr, as given by PARAMETERS,
   which is a pointer to a struct char_print_parms.  */

void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
```

[5] Imagen tomada de http://www.cs.fsu.edu/~baker/realtime/restricted/notes/pthreads.html

```
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Create a new thread to print 30000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /*-------INSERTAR AQUÍ-------*/

  return 0;
}
```

Preguntas:

1. ¿Cuál es el resultado de la ejecución? ¿Usted esperaba este resultado? ¿Por qué?
2. ¿Para qué se usa un apuntador a un tipo de dato void? ¿Qué sentido tiene hacer esto?

2.3. Conectando hilos (Join)

Cuando un hilo, después de haber creado un hilo previamente invoca la función **`pthread_join`** esperará a que el hilo creado (hijo) culmine su tarea antes de proseguir, esta función es bastante similar a la función **`waitpid`** empleada para los procesos tal y como se muestra en la figura 1.

Ejemplo 3: Empleo de la función **`pthread_join`**.

Modifique el código anterior justo después de la marca resaltada en amarillo

```
...
thread2_args.character = 'o';
thread2_args.count = 20000;
pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
```

```
/*-------INSERTAR AQUÍ-------*/
```

inserte las siguientes líneas:

```
pthread_join (thread1_id, NULL);
pthread_join (thread2_id, NULL);
```

Preguntas:

1. ¿Qué sucede ahora con la ejecución de este código? ¿Por qué?
2. ¿Cuál es la funcionalidad de hacer un join en los hilos?
3. ¿Cuáles son los parámetros de la función pthread_join? ¿Cuál es su uso?

2.4. PID e hilos

Recordemos que todo proceso se caracteriza por tener un PID asociado, pero ¿Qué pasa cuando se crea un hilo?, ¿será que el PID del hilo hijo cambia en relación con el del padre? En la siguiente sección, el objetivo es que usted descubra esto por cuenta propia al analizar el siguiente código fuente:

Ejemplo 4: PID e hilos

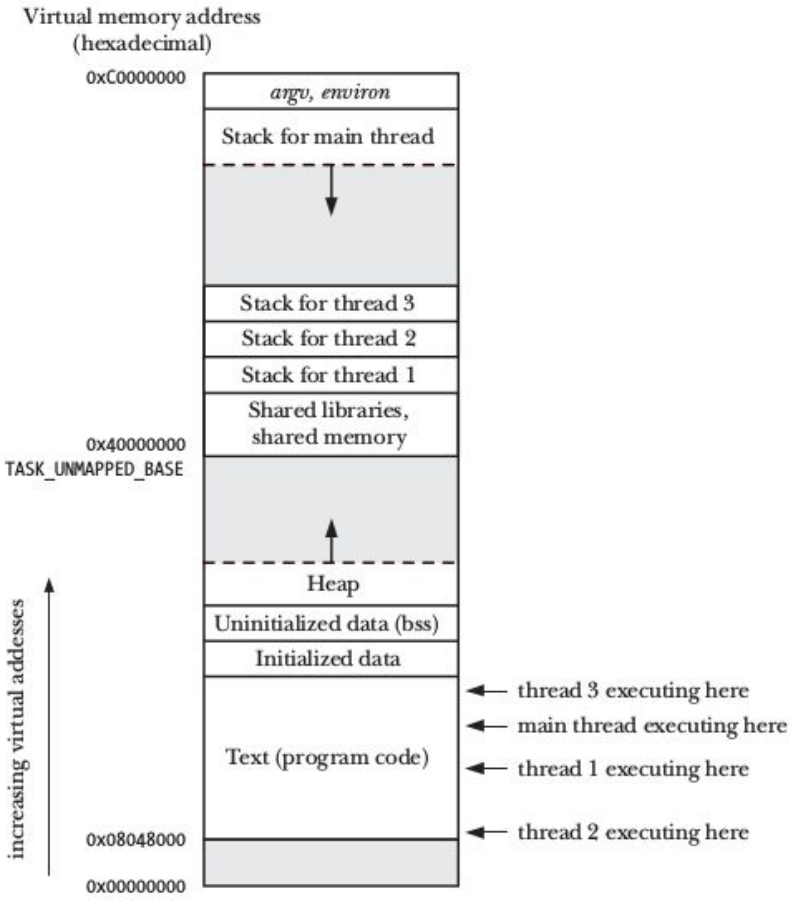
```
/**************************************************************************
* Code listing from "Advanced Linux Programming," by CodeSourcery LLC  *
* Copyright (C) 2001 by New Riders Publishing                           *
* See COPYRIGHT for license information.                                *
**************************************************************************/

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

void* thread_function (void* arg)
{
  fprintf (stderr, "child thread pid is %d\n", (int) getpid ());
  /* Spin forever.  */
  while (1);
  return NULL;
}

int main ()
{
  pthread_t thread;
  fprintf (stderr, "main thread pid is %d\n", (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* Spin forever.  */
  while (1);
  return 0;
```

```
}
```

2.4. La misma operación con el mismo dato

Tal y como hemos visto a lo largo de la guia un solo proceso puede contener múltiples hilos los cuales ejecutan de manera independiente partes del mismo programa. Visto en términos de hilos un proceso está dividido en dos partes:

- **Parte 1**: Esta contiene los recursos usados por el programa completo tales como las instrucciones del programa y los datos globales.
- **Parte 2**: Información relacionada con el estado de ejecución, tal como el program counter y el stack. Esta parte es la que se refiere al hilo propiamente.



Lo anterior hace importante distinguir en el momento de paralelizar a nivel de código que será individual (asociado a cada hilo únicamente) y que será compartido (asociado a todos los hilos). La siguiente tabla muestra esto:

Parte 1 - Compartido	Parte 2 - Individual

• global memory, including the initialized data and uninitialized data • heap segments • process ID and parent process ID • process group ID and session ID • controlling terminal • process credentials (user and group IDs) • open file descriptors • record locks created using **`fcntl()`** • signal dispositions • file system–related information: umask, current working directory, and root directory • interval timers (**`setitimer()`**) and POSIX timers (**`timer_create()`**) • System V semaphore undo (**`semadj`**) values • resource limits • CPU time consumed (as returned by **`times()`**) • resources consumed (as returned by **`getrusage()`**) • nice value (set by **`setpriority()`** and **`nice()`**).	• thread ID • signal mask • thread-specific data • alternate signal stack (**`signaltstack()`**) • the errno variable • floating-point environment • realtime scheduling policy and priority • CPU affinity (Linux-specific) • capabilities (Linux-specific) • stack (local variables and function call linkage information).

En el siguiente ejemplo sencillo se muestran las implicaciones de lo que se mencionó con anterioridad.

Ejemplo 5: Compartiendo variables entre hilos.

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

/* Global variable */
int x;

void fd(void);

int main(void) {
  pthread_t threads_ids[4];
  int i;
  for(i = 0; i < 4; i++) {
    pthread_create(&threads_ids[i],NULL,(void *)fd, NULL);
    printf("Iniciando hilo: %d\n", i + 1);
  }
  for(i = 0; i < 4; i++) {
    pthread_join(threads_ids[i],NULL);
  }
}
void fd(void) {
```

```
  int i;
  printf("Thread PID: %lu \n-> x = %d (before to be incremented 1000 times for this
thread)\n", (unsigned long)pthread_self(),x);
  for(i = 1;i <=1000;i++) {
    x++;
  }
}
```

Preguntas:

1. Analice el código del presente ejemplo. ¿Qué hace?
2. ¿Cuántos hilos tiene el proceso?
3. Ejecute el código muchas veces. ¿Los hilos se ejecutan en el mismo orden siempre? ¿esto es lo que usted esperaba?
4. ¿Qué opina al respecto de los identificadores para los hilos? ¿Cuál es la razón para tener este valor?

2.5. Retornar un valor desde el hilo

Cuando el segundo argumento que se pasa a pthread_join no es nulo, el valor de retorno del hilo será colocado en la la localización apuntada por dicho argumento. En el siguiente código se resalta esto:

Ejemplo 6: Retornando valores desde el hilo.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/* Funcion para calcular primo.*/
void* calcular_primo (void* arg);

int main(){
  pthread_t id_hilo;
  int cual_primo = 5000;
  int* primo;
  // Inicia el hilo, se requiere el 5000-iesimo numero primo
  pthread_create(&id_hilo, NULL, &calcular_primo, &cual_primo);
  // Puedo hacer algo mientras... si quiero
  // Espero que el número sea calculado y retornado
  pthread_join(id_hilo, (void *) &primo);
  // Imprimo el número entregado
  printf("El %d-esimo número primo es %d\n", cual_primo, *primo);
  free(primo);
  return 0;
}

/* Calcula los numeros primos sucesivamente. retorna el n-esimo numero primo donde n es el
valor apuntado por arg*/
void* calcular_primo (void* arg){
  int candidato = 2;
  int n = *((int*)arg);
  int factor;
```

```
  int es_primo;
  while(1){
    es_primo = 1;
    for(factor = 2; factor < candidato; factor++)
      if(candidato % factor == 0){
        es_primo = 0;
        break;
      }
    if(es_primo){
      if (--n == 0){
        int* p_c = malloc(sizeof(int));
        *p_c = candidato;
        return p_c;
      }
    }
    candidato++;
  }
  return NULL;
}
```

Preguntas:

1. ¿Cómo es el funcionamiento del código presentado?
2. Según lo visto en este ejercicio ¿Cuál es la forma de retornar valores desde un hilo? Explique de manera clara y en sus propias palabras.
3. Modifique el programa del punto anterior, de manera que pueda obtener el tiempo que demora la ejecución del hilo (Investigue sobre la función **gettimeofday**).

4. Ejemplos de refuerzo

1. El siguiente ejemplo[6] muestra un programa que ejecuta 3 tareas empleando un solo hilo.

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

extern int
main(void)
{
  do_one_thing(&r1);
  do_another_thing(&r2);
  do_wrap_up(r1, r2);
  return 0;
}

void do_one_thing(int *pnum_times)
{
```

[6] Tomado de: PThreads Programming

```
  int i, j, x;

  for (i = 0;    i  <  4;  i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }
}

void do_another_thing(int *pnum_times)
{
  int i, j, x;

  for (i = 0;    i  <  4;  i++) {
    printf("doing another \n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }
}

void do_wrap_up(int one_times, int another_times)
{
 int total;

 total = one_times + another_times;
 printf("wrap up: one thing %d, another %d, total %d\n",

 one_times, another_times, total);
}
```

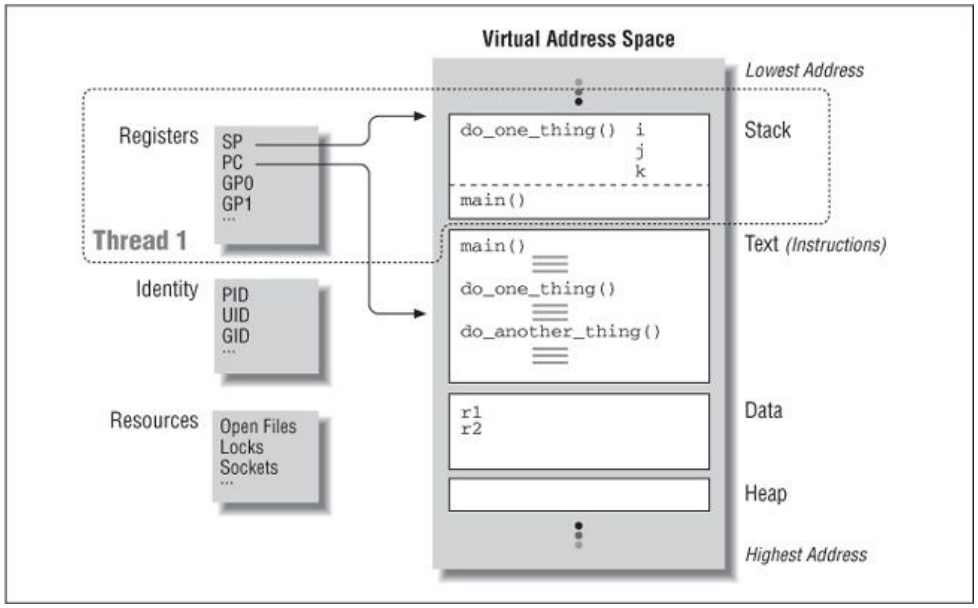
Compilando el programa con el siguiente comando se tiene:

```
gcc simple.c -o simple.out -lpthread
```

Al ejecutarlo la salida será la siguiente:

```
s_guia4/parte1/ejemplos_de_refuerzo$ ./simple.out
doing one thing
doing one thing
doing one thing
doing one thing
doing another
doing another
doing another
doing another
wrap up: one thing 4, another 4, total 8
```

Nótese que cada una de las tareas se hace de manera secuencial. La siguiente tarea resalta este caso esto en el mapa de memoria asociado al proceso (de esta aplicación) que ejecuta un solo hilo.



2. El siguiente ejemplo[7] ejecuta las mismas 3 tareas, sin embargo, mediante el uso de hilos el código es paralelizado mediante la creación de dos hilos (para un total de tres hilos con el principal). Para el caso, los hilos hijos ejecutan 2 de las 3 tareas mientras que el hilo padre ejecuta la otra tarea.

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

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

extern int main(void)
{
  pthread_t       thread1,  thread2;

  pthread_create(&thread1,
          NULL,
          (void *) do_one_thing,
          (void *) &r1);

  pthread_create(&thread2,
          NULL,
          (void *) do_another_thing,
          (void *) &r2);

  pthread_join(thread1,  NULL);
  pthread_join(thread2,  NULL);

  do_wrap_up(r1, r2);
```

[7] Tomado de: PThreads Programming

```
  return 0;
}

void do_one_thing(int *pnum_times)
{
  int i, j, x;

  for (i = 0;    i  <  4;  i++) {
    printf("doing one thing\n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }
}

void do_another_thing(int *pnum_times)
{
  int i, j, x;

  for (i = 0;    i  <  4;  i++) {
    printf("doing another \n");
    for (j = 0; j < 10000; j++) x = x + i;
    (*pnum_times)++;
  }
}

void do_wrap_up(int one_times, int another_times)
{
 int total;

 total = one_times + another_times;
 printf("wrap up: one thing %d, another %d, total %d\n",

 one_times, another_times, total);
}
```

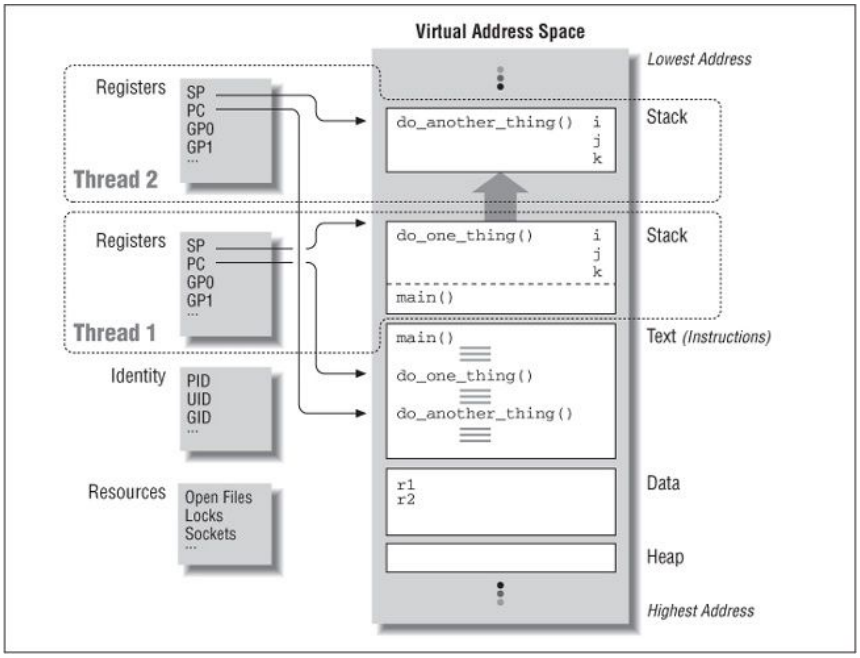
Compilando el programa con el siguiente comando se tiene:

```
gcc simple_threads.c -o simple_threads.out -lpthread
```

Al ejecutarlo la salida será la siguiente:

```
s_guia4/parte1/ejemplos_de_refuerzo$ ./simple_threads.out
doing one thing
doing one thing
doing one thing
doing one thing
doing another
doing another
doing another
doing another
wrap up: one thing 4, another 4, total 8
```

Nótese que a diferencia del ejemplo 1, cada uno de los hilos hijos tiene su propio contexto, sin embargo hay otras cosas que se comparten como las instrucciones asociadas a codigo (text), las variables globales (Data) y el heap entre otros.



3. El siguiente ejemplo[8] presenta la misma salida de los dos ejemplos anteriores pero para el caso, la solución se hace empleando procesos. Note que el comportamiento es similar que el ejemplo anterior, sin embargo el desarrollo de las tareas no es llevada a cabo por hilos sino por procesos.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int   shared_mem_id;
int   *shared_mem_ptr;
int   *r1p;
int   *r2p;
extern int
```

[8] Tomado de: PThreads Programming

```
main(void)
{
  pid_t  child1_pid, child2_pid;
  int  status;

  /* initialize shared memory segment */
  shared_mem_id = shmget(IPC_PRIVATE, 2*sizeof(int), 0660);
  shared_mem_ptr = (int *)shmat(shared_mem_id, (void *)0, 0);
  r1p = shared_mem_ptr;
  r2p = (shared_mem_ptr + 1);

  *r1p = 0;
  *r2p = 0;

  if ((child1_pid = fork()) == 0) {
    /* first child */
    do_one_thing(r1p);
    exit(0);
  }

  /* parent */
  if ((child2_pid = fork()) == 0) {
    /* second child */
    do_another_thing(r2p);
    exit(0);
  }

  /* parent */
  waitpid(child1_pid, &status, 0);
  waitpid(child2_pid, &status, 0);

  do_wrap_up(*r1p, *r2p);
  return 0;

}
```

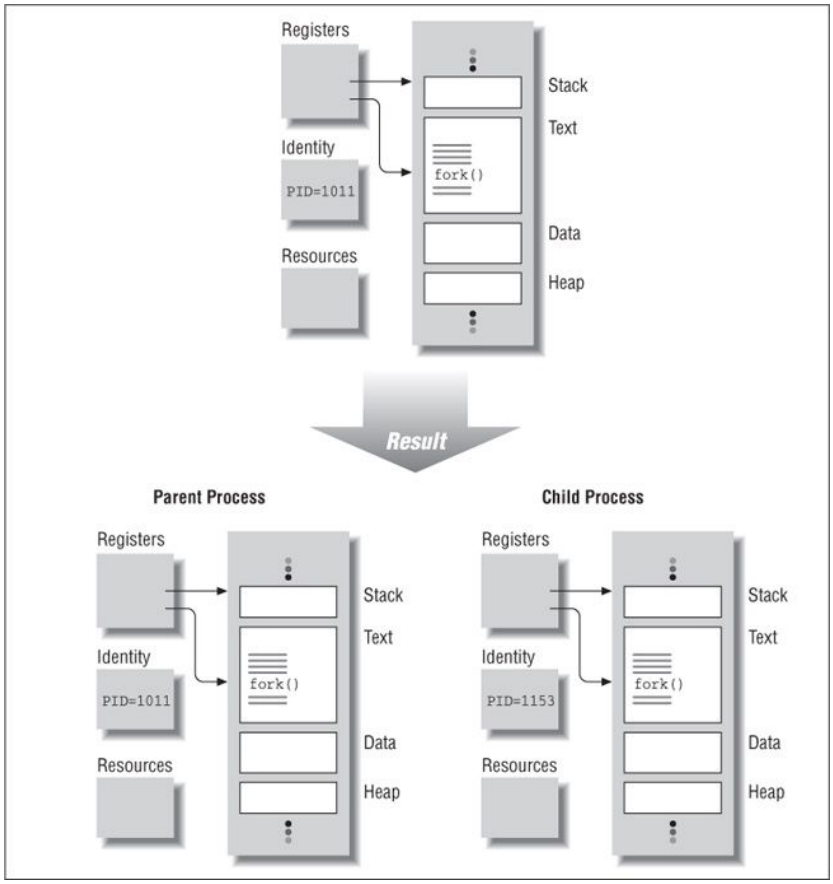
El comando de compilación para el caso:

```
gcc simple_process.c -o simple_process.out
```

Finalmente la salida:

```
./simple_process.out
doing one thing
doing one thing
doing one thing
doing one thing
doing another
doing another
doing another
doing another
wrap up: one thing 4, another 4, total 8
```

Nótese que una vez se lleva a cabo el fork, se crea un nuevo proceso completamente independiente con su propio espacio de memoria virtual y con las variables completamente independientes entre sí (y por ende no compartidas). Esta es la principal diferencia con el caso de los hilos.



4. El siguiente ejemplo tomado de la página [PTHREAD TUTORIAL – SIMPLIFIED] muestra un ejemplo en el cual un hilo hace tareas relacionadas con manejo de archivos. Entenderlo puede ser de utilidad para el desarrollo de la práctica de laboratorio.

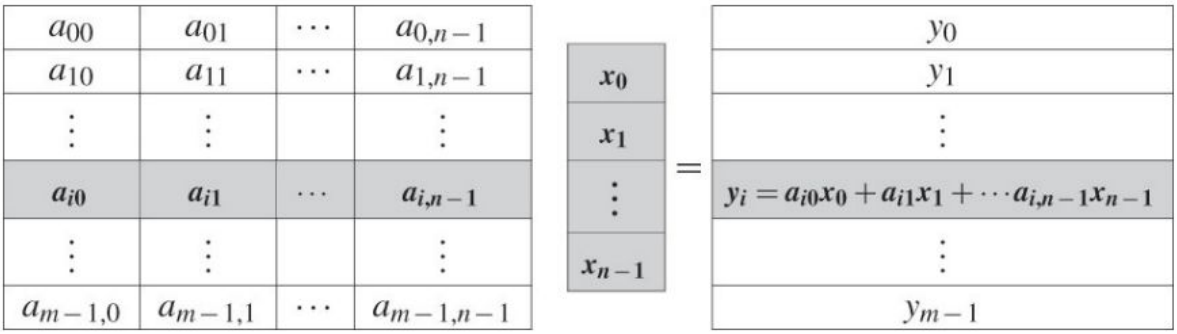
```
/*
Ejemplo tomado y levemente (casi nada) adaptado de:
-> https://vcansimplify.wordpress.com/2013/03/08/pthread-tutorial-simplified
*/

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

void* say_hello(void* data)
{
    char *str;
    str = (char*)data;
    while(1)
```

```
    {
        printf("%s\n",str);
        usleep(1);
    }
}

void* open_file(void* data)
{
    char *str;
    str = (char*)data;
    printf("Opening File\n");
    FILE* f = fopen(str,"w");
    for(int i = 0; i < 100; i++) {
      fprintf(f,"%d\n",i);
    }
    fclose(f);
    printf("Closing File\n");
}

void main()
{
    pthread_t t1,t2;

    pthread_create(&t1,NULL,open_file,"hello.txt");
    pthread_create(&t2,NULL,say_hello,"The CPU is mine now :D");
    pthread_join(t1,NULL);
}
```

Para más información e ideas de como trabajar puede apoyarse en foros, uno de tantos se muestra en la siguiente página del siempre exitoso stackoverflow.

5. El siguiente ejemplo está tomado del libro An Introduction to Parallel Programming de Peter Pacheco (**Quiere cacao**) y pretende ser un abrebocas al curso de **Computación paralela** ofrecido como una electiva en el departamento. El objetivo del ejemplo consiste en multiplicar una matrix **A** de **mxn** por un vector **x** de **n** elementos.

   Para este caso en particular, la siguiente figura muestra el resultado al multiplicar la i-ésima fila de la matrix A por el vector x para obtener el i-ésimo elemento del vector resultante y.



Solución secuencial:
La expresión de arriba para el i-ésimo componente del vector resultante y está dada por la siguiente expresión:

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

El pseudocódigo que emplea la expresión anterior para calcular todos loe elementos del vector resultante está dado por:

```
/* Para cada fila de A */
for(i = 0; i < m; i++) {
  y[i] = 0.0;
  for (j = 0; j < n; j++) {
    y[i] += A[i][j]*x[j];
  }
}
```

En mat_vect_mult.c[9] se encuentra el código solución. Inicialmente vamos a realizar un caso de test en un shell online de python[10]. Para ello introduzca los siguientes comandos:

```
>>> import numpy as np
>>> A = [ [1, 2, 3], [1, 10, 13], [-2, -10, 4], [5, 6, 9] ]
>>> B = [ -7, 4, 12 ]
>>> C = np.mult(A,B)
>>> C
array([ 37, 189,  22,  97])
```

[9] Nótese lo que es un código bien documentado al ver este ejemplo.
[10] Una de tantas posibilidades puede ser: https://www.python.org/shell/

Solo resta probar nuestro ejemplo y comparar resultados:

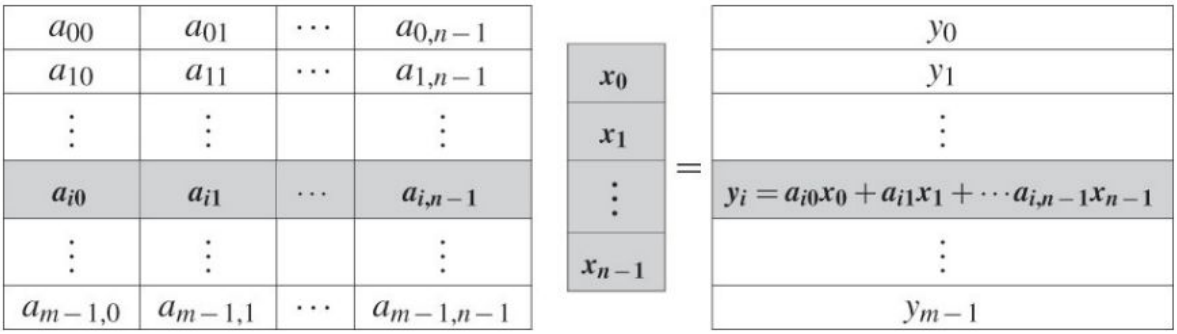
```
gcc -g -Wall -o mat_vect_mult mat_vect_mult.c
```

Ejecutando la aplicación empleando los mismos datos de test del caso anterior tenemos:

```
./mat_vect_mult
Enter the number of rows
4
Enter the number of columns
3
Enter the matrix A
1
2
3
1
10
13
-2
-10
4
5
6
9
Enter the vector x
-7
4
12

The vector y
37.000000 189.000000 22.000000 97.000000
```

Solución paralela:
Recordemos que para el caso paralelo es bueno formular una serie de preguntas antes de empezar a plantear el algoritmo, estas básicamente se resumen en responder los siguientes cuestionamientos generales los cuales apuntan a definir los parámetros que se emplearán en la función **`pthread_create`** :

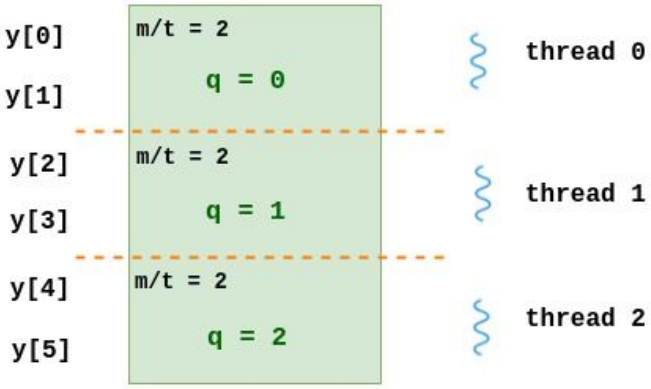
1. ¿Como se puede paralelizar del problema?
2. ¿Que necesita el hilo?
3. ¿Qué hará el hilo? el cual plantea el desarrollo de la parte algorítmica.



Para este problema particular de la matrix el tratamiento de estas preguntas se resume a continuación:

1. **¿Como se puede paralelizar el problema?** → Teniendo en cuenta que cada componente del vector respuesta es la multiplicación de una fila de la matrix por el vector de entrada y que el cálculo de un componente no necesita esperar el cálculo de otro, el problema se presta para que la matrix se divida en varias submatrices a la entrada las cuales aportaran algunos de los componentes del resultado general de modo que cada hilo de los creados será el responsable de una submatrix.
2. **¿Que necesita el hilo?**
    a. Elementos involucrados de manera general: Matrix A, vector de entrada x y vector de salida y.
    b. Entradas a cada hilo: Cada submatrix (definida por los índices final e inicial de cada fila involucrada en el calculo), el vector de entrada x. Estas se manejarán como variables globales.
    c. Salidas de cada hilo: Componentes resultantes de la multiplicación de cada submatrix por el vector de entrada, también se maneja como variable global.
3. **¿Qué hará el hilo?**
    a. Aca va el pseudocódigo del algoritmo, este será asociado a la función start asociada que se corre cuando se crea e hilo.
    b. Restricciones: El número de elementos del vector resultante debe ser múltiplo del número de hilos que la salida sea válida.

La siguiente figura muestra una descripción de la solución del problema cuando se usan 3 hilos:

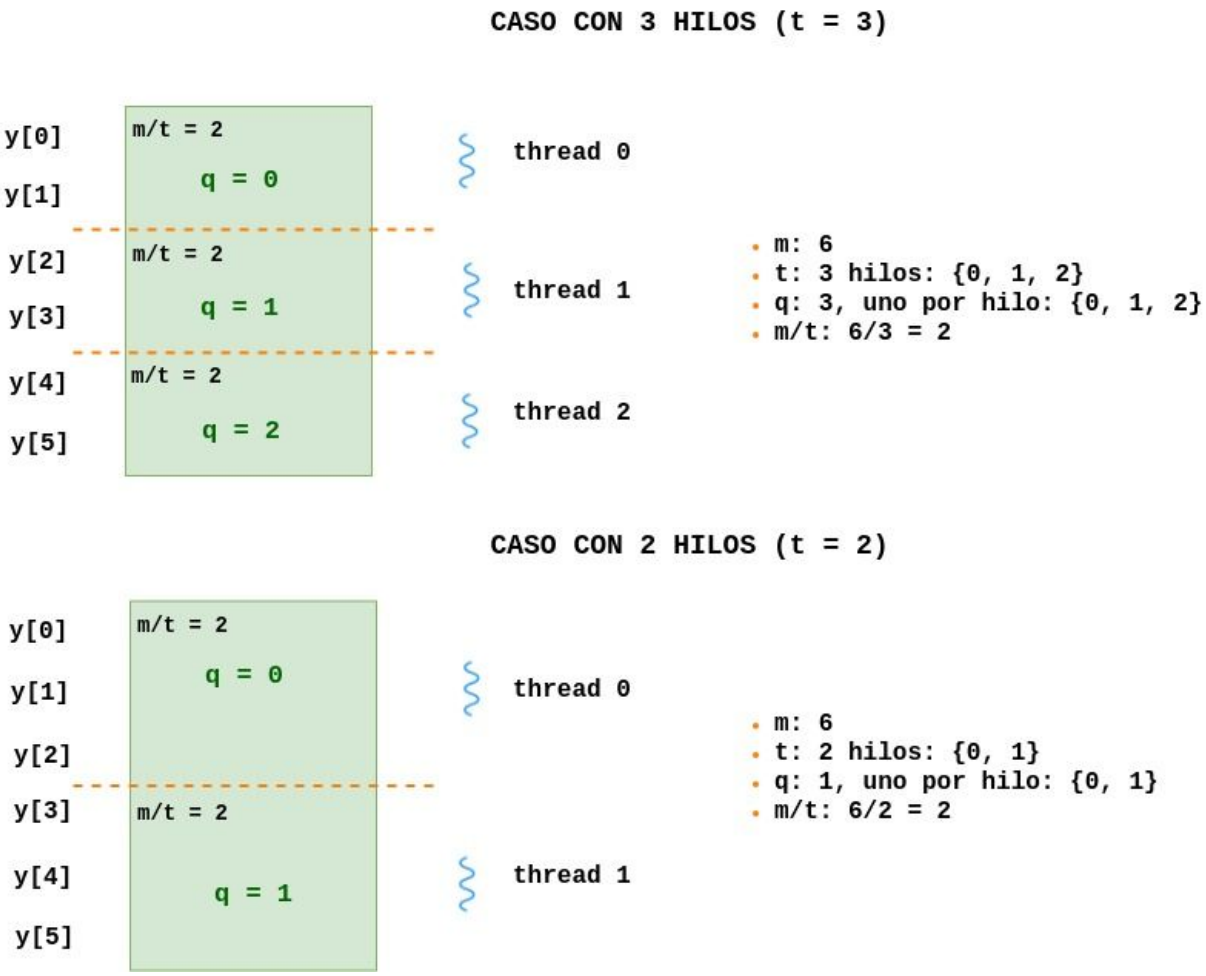


Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

- m: 6
- t: 3 hilos: {0, 1, 2}
- q: 3, uno por hilo: {0, 1, 2}
- m/t: 6/3 = 2

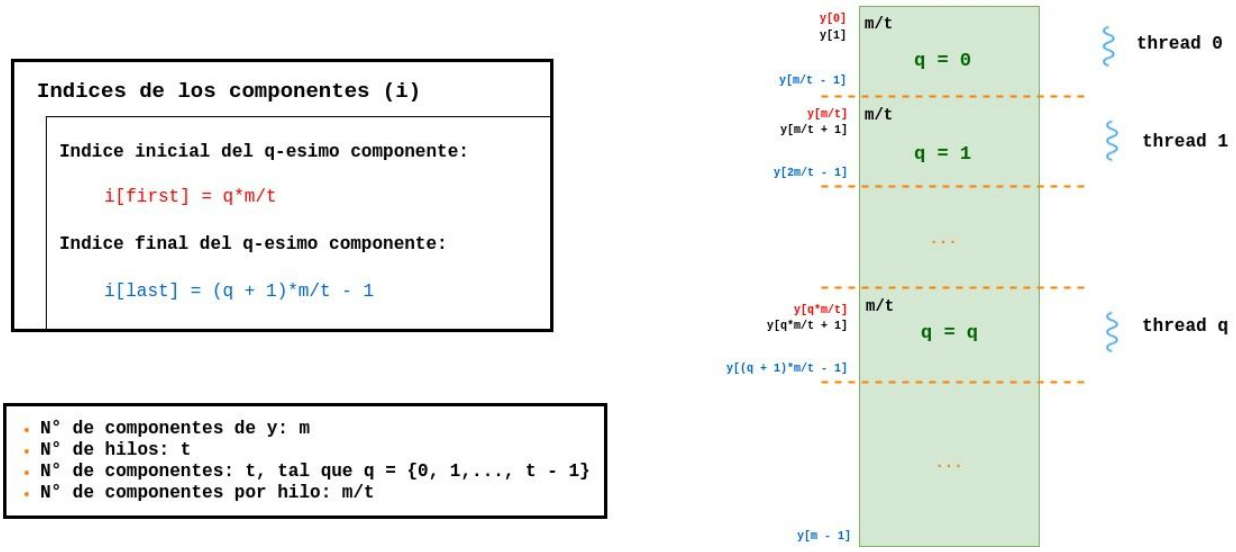
- m: N° componentes de y (o filas de A)
- t: N° de hilos
- q: Componente asociado al i-esimo hilo
- m/t: Numero de filas por hilo

Nótese bien los componentes involucrados por hilo, la siguiente figura compara los casos con 2 y 3 hilos:



Nótese que el número de componentes cambia. Finalmente el caso genérico y a partir del cual se definirá el pseudocódigo del algoritmo de la función de start será el siguiente:

CASO CON t HILOS



El pseudocódigo asociado se muestra a continuación:

```
/*--------------------------------------------------------------------
 * Function:       Pth_mat_vect
 * Purpose:        Multiply an mxn matrix by an nx1 column vector
 * In arg:         rank
 * Global in vars: A, x, m, n, thread_count
 * Global out var: y
 */
void* Pth_mat_vect(void* rank) {
  long my_rank = (long) rank;
  int i, j;
  int local_m = m/thread_count;
  int my_first_row = my_rank*local_m;
  int my_last_row = (my_rank+1)*local_m - 1;
  for (i = my_first_row; i <= my_last_row; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
      y[i] += A[i][j]*x[j];
    }
  }
  return NULL;
} /* Pth_mat_vect */
```

El código solución proporcionado por Pacheco se colocó en el siguiente enlace. Ejecute el programa con las mismas entradas del caso de uso anterior de la siguiente manera:

```
...

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   if (argc != 2) Usage(argv[0]);
   thread_count = atoi(argv[1]);
   thread_handles = malloc(thread_count*sizeof(pthread_t));

   ...

   A = malloc(m*n*sizeof(double));
   x = malloc(n*sizeof(double));
   y = malloc(m*sizeof(double));

   ...

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
         Pth_mat_vect, (void*) thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   ...

   free(A);
   free(x);
   free(y);

   return 0;
}  /* main */

...


void* Pth_mat_vect(void* rank) {
  long my_rank = (long) rank;
  int i, j;
  int local_m = m/thread_count;
  int my_first_row = my_rank*local_m;
  int my_last_row = (my_rank+1)*local_m - 1;
  for (i = my_first_row; i <= my_last_row; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++) {
      y[i] += A[i][j]*x[j];
    }
  }
  return NULL;
} /* Pth_mat_vect */
```

Cabe anotar tal y como lo dice el autor, el manejo de variables globales en el problema anterior no es buena idea. Una forma alternativa sería emplear alguna estructura como entrada a la

función de start del hilo. Lo animamos a que lo intente (Para ello puede basarse en el **ejemplo 2** de la **sección 2.2** de la guia).

Número de hilos	**Salida**
1	
2	
3	

4. Tips y recomendaciones

Cuando trabaje con hilos tener en cuenta las siguientes claves puede sacarlo de más de un apuro:

1. Tipo sobre el número de hilos dependiendo del tipo de aplicación.
2. Tipo sobre la forma como se van a reservar los recursos (cuando hacer la reserva de memoria, cuáles van a ser las preguntas claves para empezar a paralelizar, que se debe tener en cuenta).
3. Las variables globales pueden introducir bugs confusos y sutiles por lo tanto se recomienda su uso sólo a situaciones en las cuales son realmente necesarias, por ejemplo cuando se hace necesario el uso de variables compartidas.

n our example, the user specifies the number of threads to start by typing in a command-line argument. The main thread then creates all of the “subsidiary” threads. While the threads are running, the main thread prints a message, and then waits for the other threads to terminate. This approach to threaded programming is very similar to our approach to MPI programming, in which the MPI system starts a collection of processes and waits for them to complete.

There is, however, a very different approach to the design of multithreaded programs. In this approach, subsidiary threads are only started as the need arises. As an example, imagine a Web server that handles requests for information about highway traffic in the San Francisco Bay Area. Suppose that the main thread receives the requests and subsidiary threads actually fulfill the requests. At 1 o’clock on a typical Tuesday morning, there will probably be very few requests, while at 5 o’clock on a typical Tuesday evening, there will probably be thousands of requests. Thus, a natural approach to the design of this Web server is to have the main thread start subsidiary

threads when it receives requests.

5. Ejercicios propuestos

1. Analice, entienda y pruebe los siguientes dos códigos en los que se paraleliza la multiplicación de matrices:
    a. **Código 1**: Enlace
    b. **Código 2**: Enlace

2. Se requiere un programa que reciba un vector de números a través de un archivo de texto. La idea es que el programa sume todos los números del vector. Implemente el programa de dos maneras, la primera de una forma estrictamente secuencial. La segunda forma es creando dos hilos, de manera que cada uno de ellos realice la sumatoria de la mitad de los componentes del vector. El hilo 1 sumará los primeros datos del vector y el hilo 2 los últimos. Luego cuando los dos hilos finalicen muestre en pantalla el resultado.

    a. Realice el programa de manera genérica, de tal forma que sea posible ingresar vectores de cualquier tamaño.
    b. Mida el tiempo de ejecución de ambas implementaciones para varios tamaños del vector.
    c. ¿El resultado obtenido es acorde a lo que usted esperaba?
    d. Describa la técnica que usó para realizar la medición del tiempo. ¿Cuáles son las debilidades de esta técnica? ¿Existe otra forma de medir el tiempo de ejecución de un programa?

2. ***Medida de Dispersión:*** el profesor de un curso desea un programa en lenguaje C que calcule la desviación estándar (símbolo **σ** o **s**) de las notas obtenidas por sus estudiantes en el curso.

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \qquad \overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

Requisitos:

    a. El número de notas es variable (se debe usar memoria dinámica).
    b. El programa debe crear tantos hilos como se especifique en el parámetro de entrada **cantidad_hilos**, se debe ejecutar así:

```
$./nombre_ejecutable fichero_notas.csv cantidad_hilos
```

c. Se debe calcular la desviación estándar, implemente la función:

`calculate_standard_deviation().`

6. Anexo - Resumen de funciones

<table>
<tr><th>Funcion</th><th>Sintaxis</th></tr>
<tr><td rowspan="4">Thread Creation</td><td>pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code</td></tr>
<tr><td>

```
#include <pthread.h>

int pthread_create(
                    pthread_t * thread,                 /* out */
                    const pthread_attr_t * attr,        /* in */
                    void *(* start )(void *),           /* in */
                    void * arg                          /* in */
                   );
```

</td></tr>
<tr><td>Parámetros de la función:
thread : Es un apuntador (por ende debe ser previamente inicializado) en el cual se almacena el ID del thread recién creado.
attr : Apuntador que puede ser usado para asignar atributos al hilo. Si se asigna NULL, los atributos con los que se inicializará el hilo serán los atributos por defecto.
start_routine: Es la rutina (función) que el hilo ejecutará una vez que es creado.
arg: Unico argumento que puede ser pasado a la start_routine. Este deberá ser pasado por referencia haciendo un casting a un apuntador tipo void. Sí NULL es empleado significa que ningún argumento será pasado.</td></tr>
<tr><td>Retorna:
0: Si la creación del hilo exitosa.
Número positivo: En caso de error.</td></tr>
<tr><td></td><td>is used to explicitly exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist.</td></tr>
<tr><td rowspan="2">Thread Termination</td><td>

```
include <pthread.h>

void pthread_exit(void * retval );
```

</td></tr>
<tr><td></td></tr>
</table>

<table>
<tr><td rowspan="2">Thread IDs</td><td>

```
include <pthread.h>

pthread_t pthread_self(void);
```

</td></tr>
<tr><td>Returns the thread ID of the calling thread</td></tr>
<tr><td rowspan="2">check whether two thread IDs are the same</td><td>

```
include <pthread.h>

int pthread_equal(pthread_t t1 ,
                  pthread_t t2 );
```

</td></tr>
<tr><td>Returns nonzero value if t1 and t2 are equal, otherwise 0</td></tr>
<tr><td></td><td></td></tr>
<tr><td rowspan="4">Joining with a Terminated Thread</td><td>Llamada que hace que el hilo que el llamado espere hasta que el hilo hijo se termine</td></tr>
<tr><td>

```
include <pthread.h>

int pthread_join(
                 pthread_t thread ,            /* in */
                 void ** retval                /* out */
                );
```

</td></tr>
<tr><td>Parámetros de la función:
thread : Objeto tipo pthread_t asociado con el hilo hijo.
retval : Parámetro que puede ser usado para recibir cualquier valor calculado por el hilo. Este se usa comúnmente para almacenar el valor retornado por la función llamada en el pthread_create.</td></tr>
<tr><td>Retorna:
0: En caso de éxito.
Número positivo: En caso de error.</td></tr>
</table>

7. Referencias

[1] Silberschatz, A., Cagne, G., Galvin, P. Operating system concepts. Wiley, 2005.

[2] Java2 API – Thread. Available online: http://download.oracle.com/javase/1.3/docs/api/java/lang/Thread.html. Last visited: 22/09/11

[3] https://computing.llnl.gov/tutorials/pthreads/

[4] https://hpc.llnl.gov/training/tutorials#training_materials

[5] https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

[6] https://bootlin.com/doc/legacy/posix/posix-api.pdf

[7] https://bootlin.com/doc/legacy/posix/posix-api-labs.pdf

[8] http://www.cs.fsu.edu/~baker/realtime/restricted/notes/pthreads.html

[9] http://profesores.elo.utfsm.cl/~agv/elo330/2s08/lectures/POSIX_Threads.html

[10] https://users.cs.cf.ac.uk/Dave.Marshall/C/node29.html

[11] https://www.geeksforgeeks.org/pthread_self-c-example/

[12] http://dis.um.es/~ginesgm/medidas.html

[13] https://users.cs.cf.ac.uk/Dave.Marshall/C/node32.html

[14] https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

[15] https://www.geeksforgeeks.org/multithreading-c-2/

[16] https://www.safaribooksonline.com/library/view/pthreads-programming/9781449364724/ch01.html

https://github.com/NatiiDC/SistemasParalelos2017

https://github.com/MariaBamundo/Learning-C/blob/master/p_threads/pthread_pi.c

https://github.com/angrave/SystemProgramming/wiki/Pthreads,-Part-1:-Introduction

https://www.geeksforgeeks.org/multiplication-matrix-using-pthreads/

https://github.com/imsure/parallel-programming/blob/master/matrix-multiplication/matrix-mul-pthread.c

https://github.com/koswesley/Pthread-Matrix-Multiplication

http://www.par.tuwien.ac.at/teaching/2017w/184.710.html

http://www-users.cs.umn.edu/~karypis/parbook/

http://booksite.elsevier.com/9780123742605/