



## **Práctica No. 3**

# LLAMADAS AL SISTEMA Y PROGRAMACIÓN DE HILOS

### **Integrantes:**

Gómez Luna Alejandro  
Velarde Valencia Paulina

### **Grupo: 2**

### **Profesor:**

M. I. Edgar Tista García

### **Semestre:**

2021-2

### **Fecha:**

27/junio/2021

## Objetivo

- Conocer el funcionamiento básico de las llamadas al sistema en el sistema operativo Linux
- Poner en práctica los conceptos básicos de programación de Hilos y ver las diferencias entre hilos en POSIX e hilos en java

## Desarrollo

### PARTE 1 - LLAMADAS AL SISTEMA


#### *1.- Investiga en qué archivo del sistema linux se almacenan las llamadas al sistema*

Las llamadas al sistema se almacenan directamente en el núcleo de Linux en la “Tabla de llamadas al sistema”. Cada entrada de esta tabla tiene asignados un número único y una función específica que se ejecutará en el modo núcleo. Para ejecutar cualquier llamada al sistema Linux, se carga el número en la memoria de la CPU y después se carga mediante una interrupción de software 128 (llamada a una subfunción del sistema operativo que interrumpe la ejecución del programa en el modo de usuario).

La llamada al sistema es la interfaz fundamental entre una aplicación y el núcleo de Linux. A partir de la versión 2.4.17 de Linux, hay 1100 llamadas al sistemas listadas en **/usr/include/asm-generic/unistd.h**.

```
paulina@pc-pvv:/usr/include/asm-generic$ dir
auxvec.h      int-ll64.h   param.h      siginfo.h    termbits.h
bitsperlong.h ioctl.h      poll.h       signal-defs.h termios.h
bpf_perf_event.h ioctls.h    posix_types.h signal.h      types.h
errno-base.h  ipcbuf.h    resource.h   socket.h     ucontext.h
errno.h       kvm_para.h  sembuf.h    sockios.h    unistd.h
fcntl.h       mman-common.h setup.h      statfs.h
hugetlb_encode.h mman.h      shmbuf.h    stat.h
int-l64.h     msgbuf.h    shmparam.h  swab.h
```

Contenido del archivo:

```
Open  unistd.h [Read-Only]
/usr/include/asm-generic

/* SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note */
#include <asm/bitsperlong.h>

/*
 * This file contains the system call numbers, based on the
 * layout of the x86-64 architecture, which embeds the
 * pointer to the syscall in the table.
 *
 * As a basic principle, no duplication of functionality
 * should be added, e.g. we don't use lseek when llseek
 * is present. New architectures should use this file
 * and implement the less feature-full calls in user space.
 */

#ifndef __SYSCALL
#define __SYSCALL(x, y)
#endif

#if __BITS_PER_LONG == 32 || defined(__SYSCALL_COMPAT)
#define __SC_3264(nr, _32, _64) __SYSCALL(nr, _32)
#else
#define __SC_3264(nr, _32, _64) __SYSCALL(nr, _64)
#endif

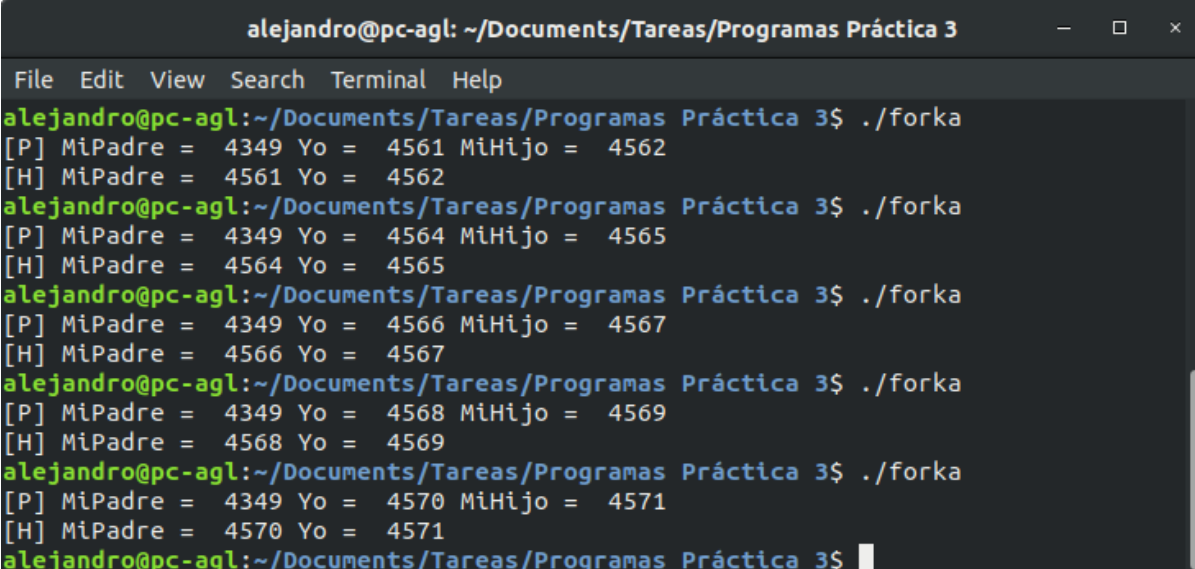
#ifdef __SYSCALL_COMPAT
#define __SC_COMP(nr, _sys, _comp) __SYSCALL(nr, _comp)
#define __SC_COMP_3264(nr, _32, _64, _comp) __SYSCALL(nr, _comp)
#else
#define __SC_COMP(nr, _sys, _comp) __SYSCALL(nr, _sys)
#define __SC_COMP_3264(nr, _32, _64, _comp) __SC_3264(nr, _32, _64)
#endif

#define __NR_io_setup 0
__SC_COMP(__NR_io_setup, sys_io_setup, compat_sys_io_setup)
#define __NR_io_destroy 1
__SYSCALL(__NR_io_destroy, sys_io_destroy)
#define __NR_io_submit 2
__SC_COMP(__NR_io_submit, sys_io_submit, compat_sys_io_submit)
#define __NR_io_cancel 3
__SYSCALL(__NR_io_cancel, sys_io_cancel)
#define __NR_io_getevents 4
__SC_COMP(__NR_io_getevents, sys_io_getevents, compat_sys_io_getevents)

/* fs/xattr.c */
#define __NR_setxattr 5
__SYSCALL(__NR_setxattr, sys_setxattr)
#define __NR_lsetxattr 6
__SYSCALL(__NR_lsetxattr, sys_lsetxattr)
#define __NR_fsetxattr 7
__SYSCALL(__NR_fsetxattr, sys_fsetxattr)
#define __NR_getxattr 8
__SYSCALL(__NR_getxattr, sys_getxattr)
#define __NR_lgetxattr 9
__SYSCALL(__NR_lgetxattr, sys_lgetxattr)
```

## 1.1 fork()

a)



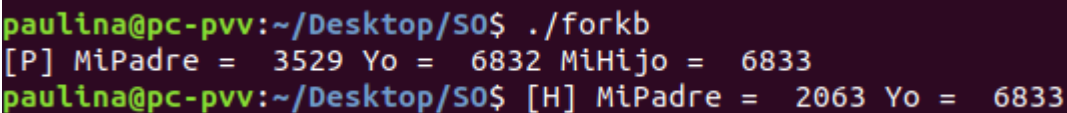
```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forka
[P] MiPadre = 4349 Yo = 4561 MiHijo = 4562
[H] MiPadre = 4561 Yo = 4562
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forka
[P] MiPadre = 4349 Yo = 4564 MiHijo = 4565
[H] MiPadre = 4564 Yo = 4565
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forka
[P] MiPadre = 4349 Yo = 4566 MiHijo = 4567
[H] MiPadre = 4566 Yo = 4567
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forka
[P] MiPadre = 4349 Yo = 4568 MiHijo = 4569
[H] MiPadre = 4568 Yo = 4569
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forka
[P] MiPadre = 4349 Yo = 4570 MiHijo = 4571
[H] MiPadre = 4570 Yo = 4571
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$
```

Cómo se puede observar, conforme se realizaron diversas ejecuciones, al momento de crear nuevos procesos mediante la llamada al sistema `fork()`, el PID de estos iba variando.

Con el código proporcionado se crea un proceso hijo que se ejecuta de manera concurrente con su proceso padre, el cual es un proceso que siempre se crea al ejecutar un programa en C. Entonces, a partir de la llamada al sistema `fork()`, se tienen dos procesos; el proceso padre y el proceso hijo. La función `fork()` va a retornar 0 en el proceso hijo y un número entero positivo en el proceso padre, por lo que, la estructura de control `if` se ejecutará solamente por el proceso hijo y la estructura de control `else` se ejecutará solamente por el proceso padre, lo cual se comprueba a la salida.

Asimismo, en la captura se observa que los PID del proceso padre e hijo corresponden mutuamente y que, el PID del proceso padre del proceso padre no varía, pues este ya no depende de la ejecución del programa.

b)



```
paulina@pc-pvv:~/Desktop/50$ ./forkb
[P] MiPadre = 3529 Yo = 6832 MiHijo = 6833
paulina@pc-pvv:~/Desktop/50$ [H] MiPadre = 2063 Yo = 6833
```

Al ejecutar el programa se crea un proceso (PID = 6832) y al momento de llamar a la función `fork()` se crea a partir de él un proceso hijo (PID = 6833). El valor que retorna esta función indicará el proceso en el que se encuentra, en este caso en un inicio `fork()` retorna el PID del proceso hijo (6833) con lo que se imprime lo que se encuentra dentro del bloque `else` indicando los PID's del proceso actual, de su padre y su hijo, terminando así la ejecución de éste proceso padre (PID = 6832).

Después de lo anterior, ahora se encuentra en el proceso hijo (PID = 6833) con lo que se imprime lo correspondiente al bloque dentro de if. En donde, se encuentra con la instrucción `sleep(1)` que hace que el sistema espere 1 segundo antes de ejecutar la siguiente instrucción, después de este lapso se imprimen los PID's del proceso actual y su proceso padre.

Se observa que el proceso actual en ese momento es PID = 6833 pero ahora su padre tiene un PID diferente, esto comprueba lo mencionado anteriormente. Es diferente ya que el padre (PID = 6832) ya había terminado su ejecución con lo que el proceso hijo (PID = 6833) tuvo que ser asociado a otro proceso (PID = 2063).

c)

```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forkc
[P] MiPadre = 4349 Yo = 5846 MiHijo = 5847
[H] MiPadre = 5846 Yo = 5847
[P]: Termino el hijo 5847 con estado 0
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forkc
[P] MiPadre = 4349 Yo = 5848 MiHijo = 5849
[H] MiPadre = 5848 Yo = 5849
[P]: Termino el hijo 5849 con estado 0
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forkc
[P] MiPadre = 4349 Yo = 5853 MiHijo = 5854
[H] MiPadre = 5853 Yo = 5854
[P]: Termino el hijo 5854 con estado 0
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forkc
[P] MiPadre = 4349 Yo = 5858 MiHijo = 5859
[H] MiPadre = 5858 Yo = 5859
[P]: Termino el hijo 5859 con estado 0
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$
```

Para este caso, se observa que los resultados son parecidos a las ejecuciones anteriores, con la distinción de que ahora el proceso padre muestra cuando finaliza la ejecución de su proceso hijo y el valor que retorna al terminar.

La llamada al sistema `wait()` solamente funciona para procesos padre y sirve para que el proceso espere a que termine el funcionamiento de sus procesos hijo. Si se le pasa un parámetro a la llamada al sistema `wait()`, en ese parámetro se guardará el valor que retorna el proceso hijo al finalizar. Esto se observa en el código al utilizar paso por referencia en la función, ya que se necesita almacenar este valor de retorno para posteriormente imprimirlo en pantalla, cómo se puede observar en las distintas ejecuciones.

El valor de retorno cero nos indica que la ejecución ha concluido con éxito.

d)

[illegible]

La salida de la ejecución de este código nos muestra que tanto el proceso padre como el hijo se ejecutan de manera concurrente. En un inicio, se crea un proceso hijo con la función `fork()` y se retorna el PID del proceso hijo por lo que se realiza lo que se encuentra dentro del bloque `else`, se imprime una cierta letra (P o H) varias veces, la cual indica que proceso es el que se encuentra ejecutándose en ese momento, dentro de la misma función `imprimirLetras` se hace una especie de conteo que hace que se consuma el tiempo dedicado a la ejecución del proceso actual y de paso al otro proceso para que se ejecute. Es por ello que cada cierto tiempo se imprime una letra y luego otra, intercambiándose según el tiempo del proceso que le corresponde, así hasta que ambos terminan de ejecutarse. En este caso específico, se puede ver que el proceso hijo termina primero que el proceso padre.

e)

```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$ ./forke
>forkc
[P] MiPadre = 3535 Yo = 3539 MiHijo = 3540
[H] MiPadre = 3539 Yo = 3540
[P]: Termino el hijo 3540 con estado 0
>forka
[P] MiPadre = 3535 Yo = 3541 MiHijo = 3542
[H] MiPadre = 3541 Yo = 3542
>forkb
[P] MiPadre = 3535 Yo = 3543 MiHijo = 3544
>[H] MiPadre = 1884 Yo = 3544
forkc
[P] MiPadre = 3535 Yo = 3546 MiHijo = 3547
[H] MiPadre = 3546 Yo = 3547
[P]: Termino el hijo 3547 con estado 0
>forke
>forka
[P] MiPadre = 3548 Yo = 3549 MiHijo = 3550
[H] MiPadre = 3549 Yo = 3550
>0
>0
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3$
```

Para este programa tenemos la llamada al sistema `execl()`, la cual nos permite ejecutar algún programa, el cual recibe cómo parámetros la ruta absoluta o relativa del archivo, el nombre del archivo y un carácter que será el delimitador que marcará el final de los argumentos, respectivamente.

Se creará un proceso hijo que se encargará de realizar la función `execl()`. El programa se mantendrá ejecutando hasta que no se encuentre el archivo especificado, en cuyo caso mostrará un mensaje de error y finalizará el proceso hijo, o cuando se ingrese un 0.

Cómo se puede observar, se ingresaron varios programas que se encontraban en la misma carpeta, inclusive se ejecutó el mismo programa, y todos finalizaron correctamente.

## 1.2 `open()/write()`

a), b) y c)

Con BLOQUE = 4050:

```
paulina@pc-pvv:~/Desktop/S0/P3$ time ./copiador archivo2 copiado
real    0m0,009s
user    0m0,000s
sys     0m0,009s
```

Con BLOQUE = 1:

```
paulina@pc-pvv:~/Desktop/S0/P3$ time ./copiador archivo2 copiado2
real    0m6,464s
user    0m0,332s
sys     0m6,130s
```

Cuando se cambia el tamaño de Bloque a 1, el tiempo de ejecución es mucho mayor que cuando el Bloque = 4050. Esto se debe a que el tamaño de BLOQUE indica el número de bytes que desean leer y escribir en cada iteración. Si el BLOQUE es 1, se indica que se leerá en cada iteración un byte por byte. Es por ello que el tiempo de ejecución es mayor y aun mas por que se trata de un archivo grande.

Las llamadas al sistema que se utilizan en este caso es para el manejo de archivos. Primero se hace uso de la función `open()` dos veces que abre el archivo que se quiere copiar y se crea el que en el cual se realizará la copia. La llamada retorna un valor que será el descriptor de archivo que se utilizará cada vez que se desee realizar una operación con el archivo.

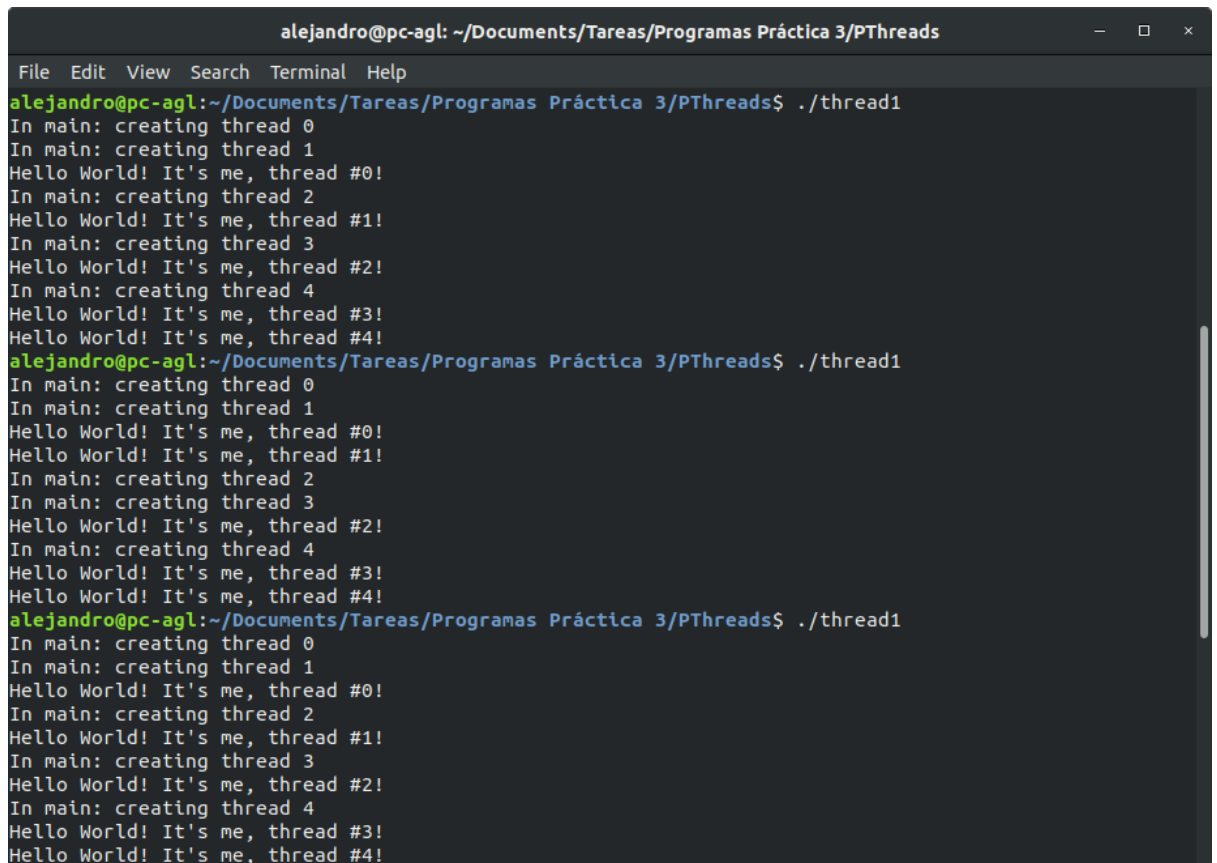
Para leer el archivo se utiliza la llamada `read()` con la cual se obtiene un número determinado de bytes leídos para enseguida escribirlos dentro de el segundo archivo abierto con la llamada `write()`, esto se repite hasta que ya se ha copiado todo el contenido del primer archivo.

Finalmente se utiliza la llamada `close()` para cerrar ambos archivos y se liberen los espacios de memoria que se le fueron asignados.

## PARTE 2 - HILOS

### 2.1 Hilos en Posix

#### a) thread1.c



```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/PThreads
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread1
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread1
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread1
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```



```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/PThreads
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread1
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread1
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #2!
Hello World! It's me, thread #1!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$
```

Para este programa se utiliza la llamada al sistema `pthread_create()`, la cual sirve para crear nuevos hilos. Esta llamada al sistema recibe como parámetros un búfer en donde se almacenarán los ID de los hilos creados, un apuntador hacia la estructura de datos que define los atributos iniciales de los hilos (en caso de ser NULL los hilos se crean con atributos por default), una secuencia inicial de ejecución y un parámetro para dicha secuencia, respectivamente. Si el hilo se creó con éxito, se retorna un valor cero. En caso de fallo, se retorna un número de error.

En este caso, se tiene un for que crea un máximo de 5 hilos, los cuales tienen una secuencia de ejecución inicial definida por la función `PrintHello()`, la cual se observa que imprime un mensaje en pantalla y especifica el número de hilo que lo imprimió. Se puede observar que, mientras el hilo main sigue creando más hilos y lo muestra en pantalla, estos mensajes se verán intercalados por las secuencias iniciales de los hilos que se van creando, en donde se aprecia que está intercalación varía entre un momento de ejecución a otro.

b) thread2.c

```

paulina@pc-pvv:~/Desktop/S0$ ./thread2
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!
Thread 3: Klingon: Nuq neH!
Thread 1: French: Bonjour, le monde!
Thread 5: Russian: Zdravstvuyte, mir!
Thread 6: Japan: Sekai e konnichiwa!
Thread 4: German: Guten Tag, Welt!
Thread 7: Latin: Orbis, te saluto!
Thread 2: Spanish: Hola mundo
paulina@pc-pvv:~/Desktop/S0$ ./thread2
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 7: Latin: Orbis, te saluto!
Thread 5: Russian: Zdravstvuyte, mir!
Thread 2: Spanish: Hola mundo
Thread 3: Klingon: Nuq neH!
Thread 6: Japan: Sekai e konnichiwa!
Thread 4: German: Guten Tag, Welt!
paulina@pc-pvv:~/Desktop/S0$ ./thread2
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 1: French: Bonjour, le monde!
Thread 0: English: Hello World!
Thread 3: Klingon: Nuq neH!
Thread 2: Spanish: Hola mundo
Thread 4: German: Guten Tag, Welt!
Thread 5: Russian: Zdravstvuyte, mir!
Thread 7: Latin: Orbis, te saluto!
Thread 6: Japan: Sekai e konnichiwa!

```

La función utilizada es `pthread_create` para crear un total de 8 hilos, los cuales realizarán la misma tarea, imprimir un determinado mensaje tomado de un arreglo, el mensaje que cada uno imprimirá dependerá del número de hilo del que se trate.

Cada que se crea un hilo, éste comienza a ejecutar la función para imprimir su mensaje pero como dentro de ésta se encuentra la función `sleep(1)` hace que el hilo creado se detenga por un segundo, dando paso a que mientras se cree otro hilo, esto sucede con los 8 hilos. Es por esto que primero se muestra que se han creado todos los hilos y luego se imprimen los mensajes que a cada hilo le corresponde y no necesariamente siguiendo el mismo orden como con el que fueron creados. Con esto se puede ver que cada hilo tiene su propio bloque de datos, cada uno tiene un valor diferente para cada una de las variables que se utilizan.

c) thread3.c

```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/PThreads
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread3
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World! Sum=0
Thread 1: French: Bonjour, le monde! Sum=1
Thread 2: Spanish: Hola al mundo Sum=3
Thread 3: Klingon: Nuq neH! Sum=6
Thread 6: Japan: Sekai e konnichiwa! Sum=21
Thread 4: German: Guten Tag, Welt! Sum=10
Thread 5: Russian: Zdravstvyye, mir! Sum=15
Thread 7: Latin: Orbis, te saluto! Sum=28
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread3
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World! Sum=0
Thread 1: French: Bonjour, le monde! Sum=1
Thread 2: Spanish: Hola al mundo Sum=3
Thread 4: German: Guten Tag, Welt! Sum=10
Thread 3: Klingon: Nuq neH! Sum=6
Thread 5: Russian: Zdravstvyye, mir! Sum=15
Thread 6: Japan: Sekai e konnichiwa! Sum=21
Thread 7: Latin: Orbis, te saluto! Sum=28
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/PThreads
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread3
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World! Sum=0
Thread 1: French: Bonjour, le monde! Sum=1
Thread 2: Spanish: Hola al mundo Sum=3
Thread 4: German: Guten Tag, Welt! Sum=10
Thread 5: Russian: Zdravstvyye, mir! Sum=15
Thread 6: Japan: Sekai e konnichiwa! Sum=21
Thread 3: Klingon: Nuq neH! Sum=6
Thread 7: Latin: Orbis, te saluto! Sum=28
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread3
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World! Sum=0
Thread 1: French: Bonjour, le monde! Sum=1
Thread 2: Spanish: Hola al mundo Sum=3
Thread 5: Russian: Zdravstvyye, mir! Sum=15
Thread 3: Klingon: Nuq neH! Sum=6
Thread 7: Latin: Orbis, te saluto! Sum=28
Thread 6: Japan: Sekai e konnichiwa! Sum=21
Thread 4: German: Guten Tag, Welt! Sum=10
```

```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/PThreads
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread3
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World! Sum=0
Thread 2: Spanish: Hola al mundo Sum=3
Thread 4: German: Guten Tag, Welt! Sum=10
Thread 1: French: Bonjour, le monde! Sum=1
Thread 6: Japan: Sekai e konnichiwa! Sum=21
Thread 3: Klingon: Nuq neH! Sum=6
Thread 7: Latin: Orbis, te saluto! Sum=28
Thread 5: Russian: Zdravstvuyte, mir! Sum=15
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$
```

Para este programa se utiliza nuevamente la llamada al sistema `pthread_create()`, para crear un total de 8 hilos.

En el código se observa que a cada uno de estos hilos, se le pasa una estructura a su secuencia de ejecución inicial `PrintHello()`, la cual define elementos como su identificador, un respectivo saludo en algún idioma como inglés, español, japonés, etc y un valor que irá incrementando conforme se creen más hilos.

Asimismo, en esta secuencia, antes de imprimir en pantalla su respectivo saludo, junto con el número de hilo y valor, se utiliza la función `sleep()` con parámetro 1. Esto con la finalidad de que todos los hilos pausen su ejecución durante 1 segundo y así el hilo `main` pueda crear todos los hilos.

En las ejecuciones se observa que efectivamente, primero siempre se crean todos los hilos, y luego estos irán imprimiendo en pantalla sus respectivos mensajes, observando que el orden de impresión en pantalla varía conforme a la ejecución.

d) thread4.c

```
paulina@pc-pvv:~/Desktop/S0$ ./thread4
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 1 done. Result = 4.071737e+13
Thread 2 done. Result = 4.071737e+13
Thread 0 done. Result = 4.071737e+13
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Main: completed join with thread 2 having a status of 2
Thread 3 done. Result = 4.071737e+13
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
paulina@pc-pvv:~/Desktop/S0$ ./thread4
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 2 done. Result = 4.071737e+13
Thread 0 done. Result = 4.071737e+13
Thread 1 done. Result = 4.071737e+13
Main: completed join with thread 0 having a status of 0
Main: completed join with thread 1 having a status of 1
Main: completed join with thread 2 having a status of 2
Thread 3 done. Result = 4.071737e+13
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
paulina@pc-pvv:~/Desktop/S0$ ./thread4
Main: creating thread 0
Main: creating thread 1
Main: creating thread 2
Thread 0 starting...
Thread 1 starting...
Thread 2 starting...
Main: creating thread 3
Thread 3 starting...
Thread 2 done. Result = 4.071737e+13
Thread 0 done. Result = 4.071737e+13
Main: completed join with thread 0 having a status of 0
Thread 1 done. Result = 4.071737e+13
Main: completed join with thread 1 having a status of 1
Main: completed join with thread 2 having a status of 2
Thread 3 done. Result = 4.071737e+13
Main: completed join with thread 3 having a status of 3
Main: program completed. Exiting.
```

Este código crea cuatro hilos, pero esta vez se crean con atributos específicos.

Las funciones que utiliza antes de crear los hilos son:

- `pthread_attr_init()`: Inicializa y define el set de valores por defecto para los hilos, se utiliza en todos los hilos creados para que tengan las mismas características sin que se tengan que definir uno por uno.
- `pthread_attr_setdetachstate(&atributos, PTHREAD_CREATE_JOINABLE);` : Con la especificación `PTHREAD_CREATE_JOINABLE` se indica que se esperará por el hilo, para que cuando este termine su ejecución se mantenga el estado y resultados obtenidos durante su ejecución y pueda así continuar con el flujo.

Esto hace que cuando un hilo es creado, éste comienza a realizar las tareas especificadas en la función dada (`BussyWork`), se observa que cada uno de los cuatro hilos ejecutan las instrucciones de manera “paralela” por lo que cada una imprime una línea diferente sin seguir el orden en como fueron creados pero únicamente lo hace con las instrucciones del primer bloque `for` ya que después de este bloque se encuentra un segundo en donde se utiliza la función `pthread_Join()`. Este segundo bloque `for` se realizará por cada hilo únicamente cuando el hilo ya haya acabado con todas sus tareas del primer bloque `for`.

e) `thread5.c`

```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/PThreads
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread5
Thread 1 did 100000 to 200000: mysum=100000.000000 global sum=100000.000000
Thread 0 did 0 to 100000: mysum=100000.000000 global sum=200000.000000
Thread 2 did 200000 to 300000: mysum=100000.000000 global sum=300000.000000
Thread 3 did 300000 to 400000: mysum=100000.000000 global sum=400000.000000
Sum = 400000.000000
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread5
Thread 0 did 0 to 100000: mysum=100000.000000 global sum=100000.000000
Thread 3 did 300000 to 400000: mysum=100000.000000 global sum=200000.000000
Thread 1 did 100000 to 200000: mysum=100000.000000 global sum=300000.000000
Thread 2 did 200000 to 300000: mysum=100000.000000 global sum=400000.000000
Sum = 400000.000000
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread5
Thread 3 did 300000 to 400000: mysum=100000.000000 global sum=100000.000000
Thread 1 did 100000 to 200000: mysum=100000.000000 global sum=200000.000000
Thread 2 did 200000 to 300000: mysum=100000.000000 global sum=300000.000000
Thread 0 did 0 to 100000: mysum=100000.000000 global sum=400000.000000
Sum = 400000.000000
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$ ./thread5
Thread 3 did 300000 to 400000: mysum=100000.000000 global sum=100000.000000
Thread 0 did 0 to 100000: mysum=100000.000000 global sum=200000.000000
Thread 1 did 100000 to 200000: mysum=100000.000000 global sum=300000.000000
Thread 2 did 200000 to 300000: mysum=100000.000000 global sum=400000.000000
Sum = 400000.000000
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/PThreads$
```



Para este programa se realiza el producto punto entre dos vectores de longitud 400000, ambos llenados únicamente con puros unos.

En el código se observan varias cuestiones que se ocuparon para poder realizar este producto punto de manera paralela, en donde para cada 100000 localidades de memoria, un hilo se encargaba de realizar el producto punto en dicha región.

Primero, se utilizan las llamadas al sistema `pthread_attr_init()` y `pthread_attr_setdetachstate()`, los cuales permiten inicializar los atributos de los hilos que se crearán y su estado de acoplo/desacoplo, el cual se especifica para que los hilos se puedan unir posteriormente en la ejecución del programa, respectivamente.

Posteriormente se crean los respectivos hilos, utilizando `pthread_create()`, y realizan el respectivo producto punto de la región que les corresponde del vector. Para determinar la región que trabajará cada hilo, se utiliza su ID. Una vez que han acabado de realizar el producto punto de dicha región, se utiliza una estructura mutex (mutua exclusión) para que solamente un hilo pueda actualizar la variable global de suma simultáneamente, que es donde se almacena la suma total del producto punto, (se utiliza `pthread_mutex_destroy()`) y, una vez que el hilo ha terminado de actualizarla, libera el bloqueo para que otro hilo pueda actualizar la variable (se utiliza `pthread_mutex_unlock()`). Esta estructura mutex se crea utilizando `pthread_mutex_init()` y se destruye con `pthread_mutex_destroy()`.

Finalmente se utilizan las llamadas al sistema `pthread_attr_destroy()` y `pthread_join()` para destruir los atributos de los hilos creados y unirlos nuevamente al hilo main principal, respectivamente. El hilo main se finaliza con `pthread_exit()`.

Cómo se puede observar en las ejecuciones, el resultado del producto punto nunca varía, así cómo las regiones que calcula cada hilo, lo que cambia entre cada ejecución es qué hilo obtiene primero el candado de la estructura mutex, resultando en que actualice primero a la variable global suma.

f) thread6.c

```
paulina@pc-pvv:~/Desktop/SO$ ./thread6
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 Count= 2. Going into wait...
inc_count(): thread 2, count = 3, unlocking mutex
inc_count(): thread 3, count = 4, unlocking mutex
inc_count(): thread 2, count = 5, unlocking mutex
inc_count(): thread 3, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 2, count = 11, unlocking mutex
inc_count(): thread 3, count = 12 Threshold reached. Just sent signal.
inc_count(): thread 3, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received. Count= 12
watch_count(): thread 1 Updating the value of count...
watch_count(): thread 1 count now = 137.
watch_count(): thread 1 Unlocking mutex.
inc_count(): thread 2, count = 138, unlocking mutex
inc_count(): thread 3, count = 139, unlocking mutex
inc_count(): thread 2, count = 140, unlocking mutex
inc_count(): thread 3, count = 141, unlocking mutex
inc_count(): thread 2, count = 142, unlocking mutex
inc_count(): thread 3, count = 143, unlocking mutex
inc_count(): thread 2, count = 144, unlocking mutex
inc_count(): thread 3, count = 145, unlocking mutex
Main(): Waited and joined with 3 threads. Final value of count = 145. Done.
```

Se hace uso de funciones que incluyen un mecanismo que permite la sincronización de los hilos en donde se establecen condiciones de acceso y señalizaciones de su cumplimiento en una parte específica del código, en este caso en ambas funciones `inc_mutex()` y `watch_mutex()`, en donde los hilos creados pueden acceder y modificar a una misma variable `count`.

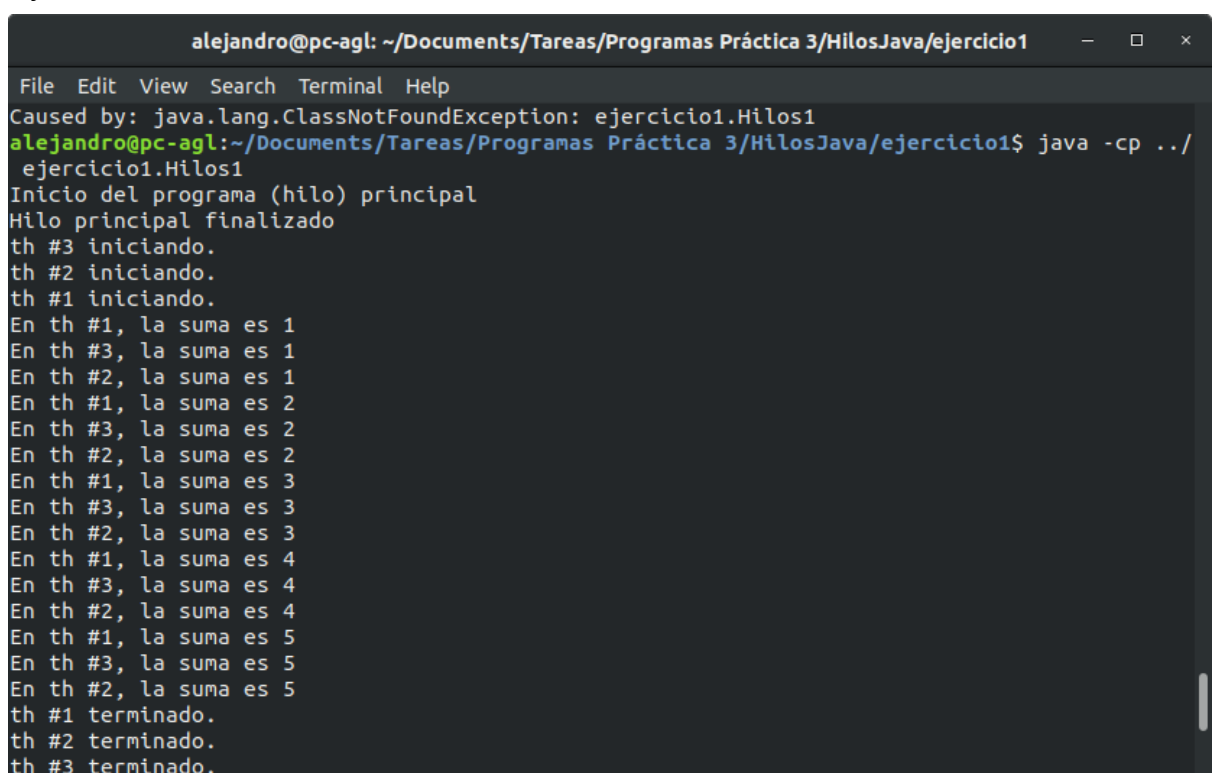
Se crean tres hilos. El hilo 1 realiza la función `watch_count()`, en donde se hace un bloqueo de la parte de código de las siguientes líneas con `pthread_mutex_lock()` y con la función `pthread_cond_wait()` entra en un estado de espera hasta que otro hilo lo active de nuevo, de manera concurrente los hilos 2 y 3 realizan la función `inc_count()`. El primer hilo que comience con la ejecución de esta función bloquea el mutex con `pthread_mutex_lock()`, haciendo que otro hilo quede en espera y no accede a esta parte. El primer hilo entonces comienza el conteo aumentando en uno el valor de la variable `count`, luego desbloquea el mutex con `pthread_mutex_unlock()` para que el otro hilo pueda acceder a esta misma parte y continúe con el conteo, esto se repite entre los dos hilos, se alternan, bloquean y ceden el paso al otro para aumentar `count`, así hasta que `count` tenga un valor de 12. Cuando esto sucede, se indica al hilo 1 que puede activarse a través de la función `pthread_cond_signal()`. Entonces, el hilo 1 continúa con la siguiente línea del código en donde se quedó por última vez. Aquí el hilo 1 suma 125 a la variable `count`, dándole un valor de 137, se imprimen las acciones realizadas y desbloquea el mutex con `pthread_mutex_unlock()`.

Con ello los hilos 2 y 3, continúan incrementando `count` hasta que se tiene que la condición del ciclo `for` se cumpla.

Con la función `pthread_join()`, el hilo principal espera a que los tres hilos creados terminen su ejecución. Después de ellos, se imprime el valor final de la variable `count`. Finalmente, se liberan los hilos y la condición del mutex mediante las funciones `pthread_attr_destroy()`, `pthread_mutex_destroy()`, `pthread_cond_destroy()` y `pthread_exit()`.

## 2.2 Hilos en Java

### a) ejercicio1



```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio1
File Edit View Search Terminal Help
Caused by: java.lang.ClassNotFoundException: ejercicio1.Hilos1
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio1$ java -cp ../
ejercicio1.Hilos1
Inicio del programa (hilo) principal
Hilo principal finalizado
th #3 iniciando.
th #2 iniciando.
th #1 iniciando.
En th #1, la suma es 1
En th #3, la suma es 1
En th #2, la suma es 1
En th #1, la suma es 2
En th #3, la suma es 2
En th #2, la suma es 2
En th #1, la suma es 3
En th #3, la suma es 3
En th #2, la suma es 3
En th #1, la suma es 4
En th #3, la suma es 4
En th #2, la suma es 4
En th #1, la suma es 5
En th #3, la suma es 5
En th #2, la suma es 5
th #1 terminado.
th #2 terminado.
th #3 terminado.
```



```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio1
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio1$ java -cp ../ejercicio1.Hilos1
Inicio del programa (hilo) principal
Hilo principal finalizado
th #2 iniciando.
th #1 iniciando.
th #3 iniciando.
En th #3, la suma es 1
En th #1, la suma es 1
En th #2, la suma es 1
En th #3, la suma es 2
En th #1, la suma es 2
En th #2, la suma es 2
En th #3, la suma es 3
En th #1, la suma es 3
En th #2, la suma es 3
En th #3, la suma es 4
En th #1, la suma es 4
En th #2, la suma es 4
En th #3, la suma es 5
En th #2, la suma es 5
En th #1, la suma es 5
th #1 terminado.
th #2 terminado.
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio1$
```

Para este programa tenemos dos clases; Hilos1 y MiHilo2.

La clase Hilos1 es la principal, y únicamente se utiliza el método crearYComenzar() de tres instancias creadas a partir de la clase MiHilo2, y se imprime en pantalla el inicio como el término del hilo principal.

Posteriormente se prosigue con la ejecución de los hilos creados a partir de la clase MiHilo2.

Esta clase implementa la interfaz Runnable, con la finalidad de poder utilizar hilos en Java.

La clase MiHilo2 cuenta con un método constructor en el cual se inicializa el atributo hilo mediante una instancia de la clase Thread. También tiene los métodos crearYComenzar(), así como la sobrecarga del método run(), el cual se lleva a cabo siempre que el hilo comienza su ejecución.

El método crearYComenzar() inicializa el hilo actual y comienza a ejecutarlo utilizando el método start().

El método run() que se ejecuta al utilizar el método start() empieza por mostrar en pantalla el hilo que se ha iniciado. Después tenemos un for, en el cual en cada iteración un hilo se dormirá por 0.8 segundos, para luego mostrar en pantalla el valor de su atributo cuenta, el cual irá incrementando en uno su valor.

En las ejecuciones se observa que el orden en el cual los hilos incrementan el valor de su atributo cuenta y lo muestran en pantalla varía.

b) ejercicio2

```
En Prioridad Baja
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Alta
En Prioridad Normal #1
En Prioridad Normal #3
En Prioridad Normal #3
En Prioridad Normal #3
En Prioridad Normal #2
En Prioridad Normal #2
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Alta
En Prioridad Alta
En Prioridad Baja
En Prioridad Alta
En Prioridad Normal #2
En Prioridad Alta
En Prioridad Normal #3
En Prioridad Alta
En Prioridad Normal #2
En Prioridad Normal #2
En Prioridad Normal #2
En Prioridad Baja
En Prioridad Normal #2
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Normal #1
En Prioridad Alta
En Prioridad Alta
En Prioridad Alta
En Prioridad Normal #3
En Prioridad Normal #1
En Prioridad Normal #2
En Prioridad Baja

Prioridad Baja terminado.

Prioridad Normal #1 terminado.

Prioridad Normal #3 terminado.

Prioridad Normal #2 terminado.

Prioridad Alta terminado.

Hilo de alta prioridad contó hasta      1000000000
Hilo de baja prioridad contó hasta      98004476
Hilo de normal prioridad #1 contó hasta 84129237
Hilo de normal prioridad #2 contó hasta 78233230
Hilo de normal prioridad #3 contó hasta 76454923
```

En este caso, se tienen dos clases: Hilos2 y PrioridadHilos.

La clase Hilos2 es la principal en donde se crean cinco hilos a los cuales se les da un nombre específico que indica que tipo de prioridad se le da (Alta, Baja y Normal) llamando a la clase PrioridadHilos.

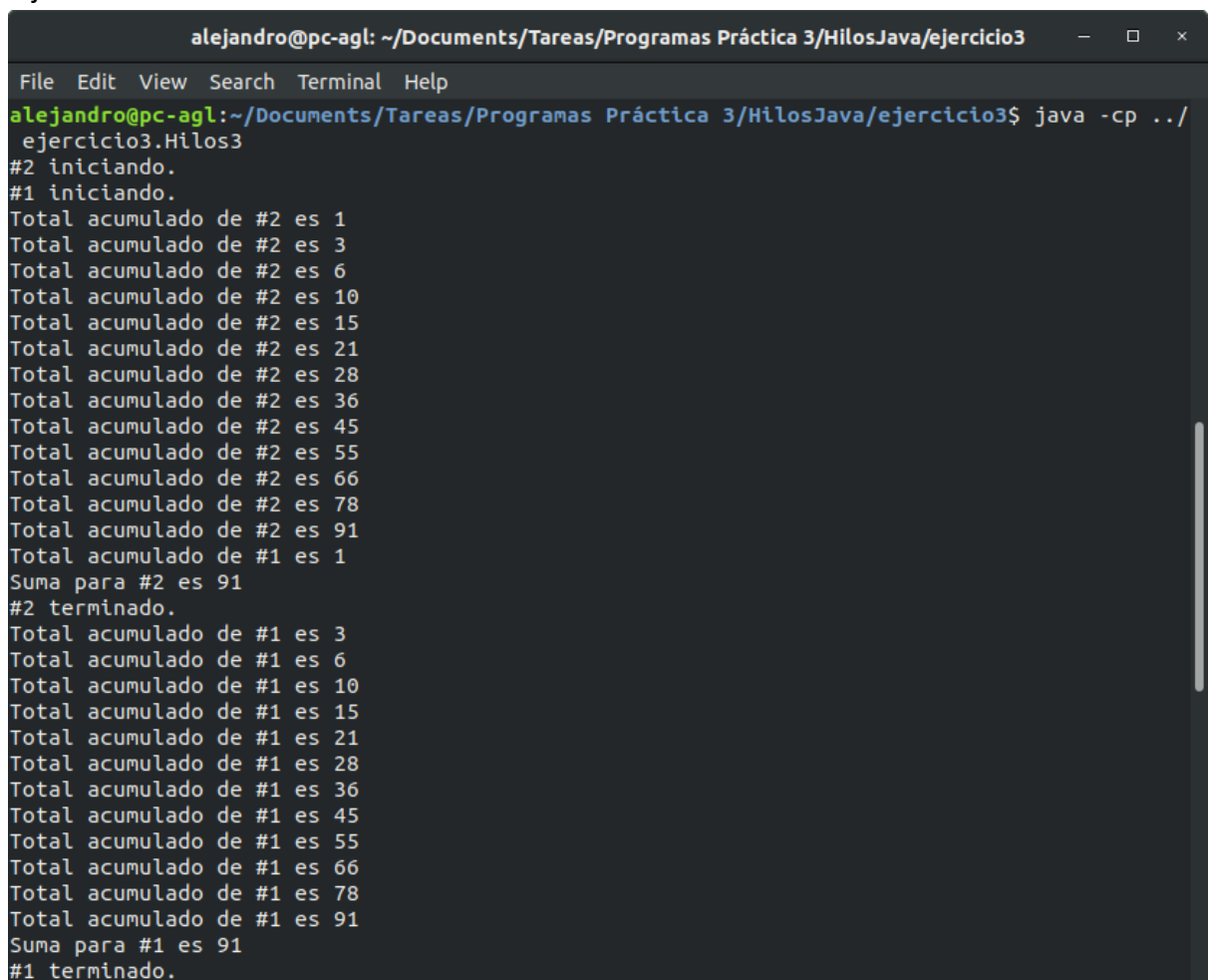
Cada uno se inicia con la función start() y comienza a ejecutar las instrucciones especificadas dentro de la función run() alternándose cada uno de los hilos y llevando su propia cuenta. El sistema le asigna un determinado tiempo dependiendo de la prioridad que este tenga por lo que la cuenta que lleva cada uno es diferente.

Cuando uno de ellos acaba primero, llega a un conteo de 1000000, hace que los demás hilos suspendan las iteraciones y con ello su propio conteo.

La función join() hace que el hilo principal espere a cada uno de los hilos hijos creados, por lo que las impresiones de conteo de cada uno no se realizan hasta que ya todos han suspendido sus iteraciones por el término de las de uno de los hilos hijo.

En cada ejecución del programa los conteos de cada hilo, así como el momento de término es diferente y no siempre el hilo de prioridad alta es el que termina primero su ejecución.

c) ejercicio3



```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio3
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio3$ java -cp ../ejercicio3.Hilos3
#2 iniciando.
#1 iniciando.
Total acumulado de #2 es 1
Total acumulado de #2 es 3
Total acumulado de #2 es 6
Total acumulado de #2 es 10
Total acumulado de #2 es 15
Total acumulado de #2 es 21
Total acumulado de #2 es 28
Total acumulado de #2 es 36
Total acumulado de #2 es 45
Total acumulado de #2 es 55
Total acumulado de #2 es 66
Total acumulado de #2 es 78
Total acumulado de #2 es 91
Total acumulado de #1 es 1
Suma para #2 es 91
#2 terminado.
Total acumulado de #1 es 3
Total acumulado de #1 es 6
Total acumulado de #1 es 10
Total acumulado de #1 es 15
Total acumulado de #1 es 21
Total acumulado de #1 es 28
Total acumulado de #1 es 36
Total acumulado de #1 es 45
Total acumulado de #1 es 55
Total acumulado de #1 es 66
Total acumulado de #1 es 78
Total acumulado de #1 es 91
Suma para #1 es 91
#1 terminado.
```

```
alejandro@pc-agl: ~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio3
File Edit View Search Terminal Help
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio3$ java -cp ../ejercicio3.Hilos3
#1 iniciando.
#2 iniciando.
Total acumulado de #1 es 1
Total acumulado de #1 es 3
Total acumulado de #1 es 6
Total acumulado de #1 es 10
Total acumulado de #1 es 15
Total acumulado de #1 es 21
Total acumulado de #1 es 28
Total acumulado de #1 es 36
Total acumulado de #1 es 45
Total acumulado de #1 es 55
Total acumulado de #1 es 66
Total acumulado de #1 es 78
Total acumulado de #1 es 91
Total acumulado de #2 es 1
Suma para #1 es 91
#1 terminado.
Total acumulado de #2 es 3
Total acumulado de #2 es 6
Total acumulado de #2 es 10
Total acumulado de #2 es 15
Total acumulado de #2 es 21
Total acumulado de #2 es 28
Total acumulado de #2 es 36
Total acumulado de #2 es 45
Total acumulado de #2 es 55
Total acumulado de #2 es 66
Total acumulado de #2 es 78
Total acumulado de #2 es 91
Suma para #2 es 91
#2 terminado.
alejandro@pc-agl:~/Documents/Tareas/Programas Práctica 3/HilosJava/ejercicio3$
```

Para este programa se tienen tres clases; Hilos3, MiHilo y Suma.

La clase Hilos3 es la principal, y únicamente se crea un arreglo que contiene los números del 1 al 13, y utiliza el método creaElInicio() de dos instancias creadas a partir de la clase MiHilo. También se utiliza el método join(), el cual hará que el hilo principal creado por el método main() espera a que finalice la ejecución de los hilos que acaba de crear.

Posteriormente se prosigue con la ejecución de los hilos creados a partir de la clase MiHilo.

La clase MiHilo cuenta con un método constructor en el cual se inicializa el atributo hilo mediante una instancia de la clase Thread. También tiene los métodos creaElInicio(), así como la sobrecarga del método run(), el cual se lleva a cabo siempre que el hilo comienza su ejecución.

El método creaElInicio() inicializa el hilo actual y comienza a ejecutarlo utilizando el método start().

El método run() que se ejecuta al utilizar el método start() empieza por mostrar en pantalla el hilo que se ha iniciado. Después se utiliza un bloque synchronized sobre el arreglo definido en la clase principal, esto con la finalidad de que solamente un hilo pueda acceder a dicho arreglo a la vez.

Ya que se ha accedido a este arreglo se utiliza el método `sumArray()` mediante una instancia de la clase `Suma`. Este método se va a encargar de ir sumando los elementos que se encontraban en el arreglo y, cada que adicione un nuevo elemento, hará que el hilo actual se duerma durante 10 milisegundos, para que el otro hilo pueda acceder al bloque `synchronized`. Esto continúa hasta que alguno de los hilos termine de sumar todos los elementos del arreglo.

Se muestra en pantalla cuando alguno de los hilos ha terminado.

En las capturas de pantalla se puede mostrar que primero puede acceder el hilo 1 o 2 al arreglo, sin embargo, no se aprecia que el otro hilo acceda al arreglo cuando el hilo que lo está ocupando se encuentra dormido, ya que este último no libera el recurso aún cuando duerme.

## Conclusiones

- *Alejandro*: Se logró observar el funcionamiento de algunas llamadas al sistema en Linux, utilizando el lenguaje de programación C. Asimismo, se puso en práctica la creación y manipulación de hilos en POSIX y Java.

Las llamadas al sistema que se utilizaron para las actividades de esta práctica nos permiten comprender mejor su importancia y funcionamiento visto en teoría. Dentro de las llamadas al sistema que encontramos está; `fork()`, la cual nos permite crear un proceso hijo a partir de un proceso padre; `wait()` para que el proceso padre espere a que finalicen los procesos hijos que haya creado; `execl()` para ejecutar algún archivo; `open()` para abrir algún archivo; `write()` para escribir en algún archivo; `pthread_create()` para crear hilos; entre otras llamadas al sistema. Para estas diversas llamadas al sistema se tiene que algunas necesitan parámetros obligatorios, como `execl()`, la cual requiere conocer el archivo que se va a ejecutar, su ruta y un carácter delimitador. Por otro lado, existen llamadas al sistema como `fork()`, la cual no requiere de parámetros, pues en caso de querer configurar al nuevo proceso creado, se utilizan otras llamadas al sistema para realizarlo.

En cuanto a la manipulación de hilos, se pudo realizar tanto en el lenguaje de programación C, como en el lenguaje de programación Java. Para el lenguaje C, se tuvieron que utilizar llamadas al sistema, mientras que en el lenguaje Java se utilizó la interfaz `Runnable` y la clase `Thread`. Para ambos casos se observó que, dependiendo la ejecución, podía cambiar el orden en que los hilos realizaban una tarea, comenzaban a ejecutarse, entre otras cuestiones. También es importante remarcar que, cuando se compartía algún recurso entre todos los hilos creados, debía existir alguna estructura de control, para el caso de los programas en C se utilizó una estructura `mutex`, y para el caso de los programas en Java, este implementa de manera automática el uso de monitores.

Para esta práctica se pudo observar el funcionamiento de las llamadas al sistema, las cuales permiten a los procesos y al usuario comunicarse con el kernel, y el uso de hilos, los cuales son creados por distintos procesos o el mismo SO. A través de los distintos programas proporcionados se pudo ver más explícito el funcionamiento de las llamadas al sistema, ya que muchas veces no se puede observar todo lo que está realizando nuestro SO cuando nosotros damos click en un programa. De igual forma se observó que el uso de hilos requiere de un estricto control para que no existan conflictos al acceder a un recurso compartido o para que no se ejecuten acciones en un momento no deseado.

Es así como la práctica permite que se comprenda de mejor manera las llamadas al sistema y el uso de hilos.

- *Paulina:* Con ayuda de los ejercicios fue posible comprender mejor cómo es el manejo tanto de procesos como hilos ya en un sistema operativo real. La ejecución de cada programa fue más fácil de comprender debido a lo que teóricamente se ha revisado en clase, así como también conocer cómo es que funcionan las funciones específicas junto con sus parámetros para su manejo en C y Java.

Por un lado, la creación de procesos utilizando la función `fork()` hace que los procesos hijos sean una copia del padre en cuanto al espacio de código, variables, memoria, archivos etc. y es posible saber qué proceso se encuentra ejecutándose, dependiendo del valor que la función retorne, esto hace que se pueda establecer instrucciones específicas para cada proceso mediante estructuras de control, además se puede confirmar que cada vez que el proceso padre o hijo se ejecuta y modifica partes de memoria o valor de variables, no afectan directamente a las del otro proceso ya que cada uno guarda su propio estado y registro. Incluso cuando el proceso padre termina su ejecución, pero el hijo no se le puede asignar uno nuevo.

Por el otro lado, para el manejo de hilos, se revisan implementados en Java y C. En Java es utilizada la interfaz `Thread` y la redefinición del método `run()` que indique la tarea que el hilo realizará.

En ambos casos se puede ver que la ejecución y tiempo asignado para cada hilo es diferente en cada vez que se ejecuta el mismo programa pero que mantiene sus datos propios. Además de funciones como `sleep()` y `wait()` que hacen que el proceso o hilo esperen un determinado tiempo deteniendo su ejecución y dando paso a otro.

## Referencias

- <https://www.geeksforgeeks.org/> consultado por última vez el 19/junio/2021 a las 18:48
- <https://man7.org/linux/man-pages/man3/> consultado por última vez el el 18/junio/2021 a las 20:07
- [https://w3.ual.es/~acorral/DSO/Practicas\\_Linux\\_2007.pdf](https://w3.ual.es/~acorral/DSO/Practicas_Linux_2007.pdf) consultado por última vez el el 21/junio/2021 a las 17:35
- [https://www.fceia.unr.edu.ar/~diegob/so/material/Llamadas\\_al\\_Sistema.pdf](https://www.fceia.unr.edu.ar/~diegob/so/material/Llamadas_al_Sistema.pdf) consultado por última vez el el 21/junio/2021 a las 17:50