UNIVERSIDAD DE CÓRDOBA

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

ASIGNATURA ***SISTEMAS OPERATIVOS***

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 2

Hilos y comunicación entre hilos. Mecanismos de exclusión y sincronización. Semáforos.

Profesorado: Juan Carlos Fernández Caballero

Enrique García Salcines

Javier Pérez Rodríguez

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue familiarizar al alumnado con la creación y gestión de hilos en UNIX, también conocidos como procesos ligeros o hebras. En una primera parte se dará una introducción teórica sobre hilos, siendo en la segunda parte de la misma cuando, mediante programación en C, se practicarán los conceptos aprendidos, utilizando las rutinas de interfaz del sistema que proporciona a los programadores la librería *pthread* basada en el estándar POSIX.

2 Recomendaciones

El lector debe completar las nociones dadas en las siguientes secciones con consultas bibliográficas, tanto en la Web como en la biblioteca de la Universidad, ya que unos de los objetivos de las prácticas es potenciar su capacidad autodidacta y su capacidad de análisis de un problema. Es recomendable que, aparte de los ejercicios prácticos que se proponen, pruebe y modifique otros que se encuentren en la Web (se dispone de una gran cantidad de problemas resueltos en C sobre esta temática), ya que al final de curso deberá acometer un examen práctico en ordenador como parte de la evaluación de la asignatura.

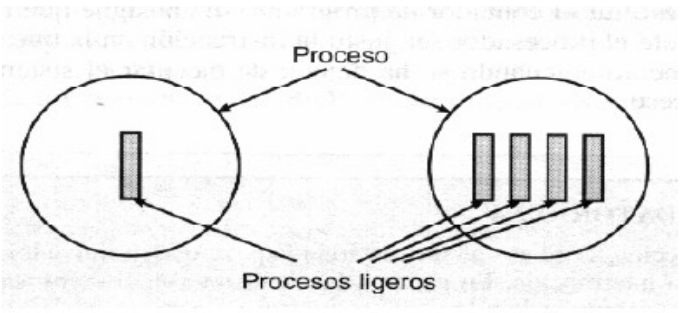
No olvide que debe consultar siempre que lo necesite el estándar POSIX http://pubs.opengroup.org/onlinepubs/9699919799/ y la librería GNU C (*glibc*) en http://www.gnu.org/software/libc/libc.html.

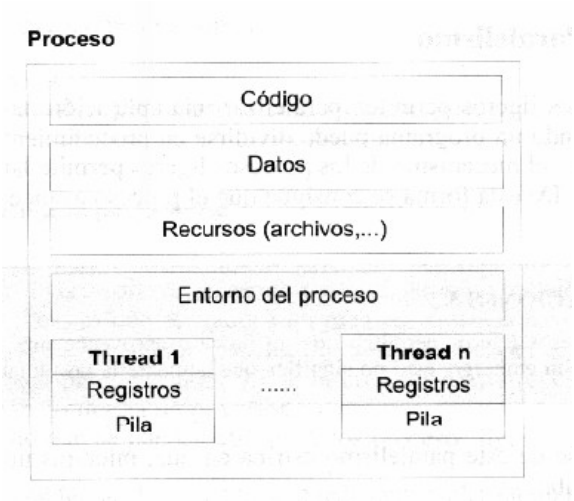
3 Conceptos teóricos

3.1 Diferencia entre procesos y hebras (threads, hilos o procesos ligeros)

Un **proceso** es un programa en ejecución que se ejecuta en secuencia, no más de una instrucción a la vez. Como se vio en la Práctica 1, se pueden crear procesos nuevos mediante la llamada a *fork()*. Estos nuevos procesos son idénticos al proceso padre que hizo la llamada (ya que son una copia del mismo), excepto en que se alojan en zonas o espacios de memoria distintos. Al ser copias tienen las mismas variables con los mismos nombres, pero distintas instancias, por lo que si un proceso hijo realiza una modificación en una variable, ésta no se ve afectada en el proceso padre u otros procesos. Por tanto, los procesos no comparten memoria ni se comunican entre si a no ser que utilicemos mecanismos de intercomunicación de procesos (IPC – *InterProcess Communication*) específicos como las colas de mensajes, la memoria compartida, pipelines, señales, o sockets.

Una **hebra**, **hilo** o **proceso ligero** (**thread** en inglés) es un flujo de control perteneciente a un proceso, que tiene su propia pila. A diferencia de un proceso, un *thread* comparte memoria con otros *threads*. Un proceso puede tener una o varias hebras, tal y como se refleja en la siguiente figura. Desde el punto de vista de la programación, una hebra se define como una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario o proceso ligero primario se correspondería con el *main()*.



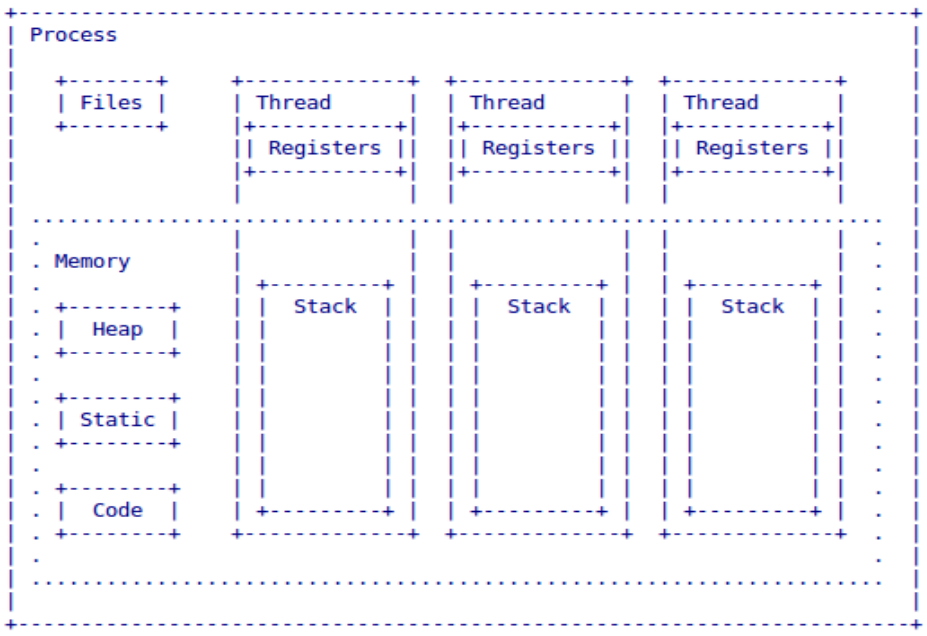


Cada hebra dentro de un proceso tiene determinada información propia que **no comparte** con el resto de hebras que nacen del mismo proceso:

- Tienen su propio estado (ejecutando, listo, bloqueado).
- Tienen su propia pila o stack.
- Tienen sus propios valores de los registros que usa del procesador (contexto).
- Propio contador de programa.
- *Errno*. Cada hilo tiene su propia variable *errno* para cuando se produzca errores en llamadas al sistema. El programa principal no tiene acceso directo al *errno* de un hilo, de modo que si necesita esta información se deberá devolver a través de la función *pthread_join()* que estudiará en las siguientes secciones.

Con respecto a la información que se **comparte** con otras hebras procedentes del mismo proceso son:

- **Mismo espacio de memoria.** El espacio de memoria incluye: Código, datos y pilas del conjunto de hebras (ver figuras anteriores). El espacio de memoria corresponde al proceso y a todas las hebras que engloba ese proceso, por lo que no hay una protección de memoria como ocurre con los procesos. Esto hace imprescindible el uso de semáforos o mutex (EXclusión MUTua) para evitar que dos *threads* accedan a la vez a la misma estructura de datos (se estudiará en las siguientes prácticas). También hace que si un hilo "se equivoca" y corrompe una zona de memoria, todos los demás hilos del mismo proceso vean la memoria corrompida. Un fallo en un hilo puede hacer fallar a todos los demás hilos del mismo proceso.
- **Variables globales.** Las hebras de un mismo proceso se pueden comunicar (entre una de las maneras posibles) mediante variables globales, pero hay que tener extremado cuidado con su programación.
- **Archivos abiertos.**
- **Temporizadores.**
- **Señales y Semáforos** (se estudiará más adelante).
- **Entorno de trabajo.**



La ventajas principales (debido a que comparten el mismo espacio de memoria) de usar un grupo de *threads* en vez de un grupo de procesos son:

- Crear o terminar una hebra es mucho más rápido que crear o terminar un proceso, ya que las hebras comparten determinados recursos del padre. Cuando se termina un proceso se debe eliminar el BCP del mismo, mientras que en un hilo se elimina solo su contexto y pila.
- El cambio de contexto entre *threads* es realizado mucho más rápidamente que el cambio de contexto entre procesos, mejorando el rendimiento del sistema. Al cambiar de un proceso a otro el sistema operativo (mediante el *dispatcher*) genera lo que se conoce como *overhead*, que es tiempo desperdiciado por el procesador para realizar un cambio de contexto, por ejemplo pasar del estado de ejecución del proceso actual al estado de espera o bloqueo y colocar un nuevo proceso en ejecución. En los hilos, como pertenecen a un mismo proceso, al realizar un cambio de hilo el tiempo perdido es casi despreciable.
- Si se necesitase comunicación entre hebras también sería mucho más rápido que intercomunicar procesos, ya que los datos están inmediatamente habilitados y disponibles entre hebras. En la mayoría de los sistemas, en la comunicación entre procesos, debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre hilos pueden comunicarse entre sí sin la invocación al núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de procesos separados.

Así como podemos tener múltiples procesos ejecutando en un PC, también podemos tener múltiples *threads*. Como dentro de un proceso puede haber varios hilos de ejecución (varios threads), en el caso de que tuviéramos más de un procesador o un procesador con varios núcleos, un proceso podría estar haciendo varias cosas "a la vez", pero de manera más rápida que si lo hiciéramos con procesos puros. Así, cada hebra podría asignarse a un núcleo o a un procesador, y si un hilo se interrumpe o bloquea los demás pueden seguir ejecutando. Estos son los procesos a nivel de núcleo o KLT que estudiará en clases teóricas.

Resumiendo, en el caso de UNIX, mediante *fork()* creamos procesos independientes entre si, de forma que sea imposible que un proceso se entremezcle por equivocación en la zona de memoria de otro proceso, haciendo que el sistema sea fiable. Por otro lado, las hebras pueden ser útiles para programar aplicaciones que deben hacer tareas simultáneamente y/o queremos que haya bastante intercomunicación entre ellas. Dependiendo de la aplicación optaremos por una solución u otra, aunque eso es también un aspecto que debe elegir el analista-programador en base a su experiencia.

Algunos ejemplos de usos de hebras, cuyo uso permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente, pueden ser:

- Procesador de textos: utilizar a una hebra por cada tarea que realiza.
  - ✔ Interacción con el usuario.
  - ✔ Corrector ortográfico/gramatical.
  - ✔ Guardar automáticamente y/o en segundo plano.
  - ✔ Mostrar resultado final en pantalla.
- Hoja de cálculo:
  - ✔ Interacción con el usuario.
  - ✔ Actualización en segundo plano.
- Servidor web: dos tipos de hebras.
  - ✔ Interacción con los clientes (navegadores).
  - ✔ Gestión del caché de páginas.
- Navegador: varios tipos de hebras.
  - ✔ Interacción con el usuario.
  - ✔ Interacción con los servidores (web, ftp, ...).
  - ✔ Dibujo de la página (1 hebra por pestaña).

3.2 Biblioteca de C para el uso de hebras y normas de compilación

Al igual que con los procesos en sistemas UNIX, las hebras también tienen una especificación en el estándar POSIX IEEE Std 1003.1-2008, concretamente en la **biblioteca *pthread***. Para crear programas que hagan uso de la biblioteca *pthread* necesitamos, en primer lugar, la biblioteca en sí. Ésta viene en la mayoría de distribuciones Linux, y seguramente se instale al mismo tiempo que los paquetes incluidos para el desarrollo de aplicaciones. Si no es así, o usa un sistema que no sea Linux pero se base en POSIX, la biblioteca no debería ser difícil de encontrar en la red, porque es bastante conocida y usada. Una vez tenemos la biblioteca instalada, y hemos creado nuestro programa, deberemos compilarlo y "linkarlo" con la misma. El fichero de cabecera <*pthread.h*>*w* debe estar incluido en nuestras implementaciones. La forma más usual de compilar si estamos usando *gcc* es:

```
gcc programa_con_pthreads.c -o programa_con_pthreads -lpthread
```

3.3 Servicios POSIX para la gestión de hebras

A continuación se expondrán las funciones o llamadas al sistema para la gestión de hebras que implementa la librería *pthread* al seguir el estándar POSIX especificado en la IEEE (Institute of Electrical and Electronics Engineers, Instituto de Ingenieros Eléctricos y Electrónicos).

3.3.1 Creación y ejecución de una hebra (pthread_create())

Para crear un *thread* nos valdremos de la función *pthread_create()* de la biblioteca *pthread*, y de la estructura *pthread_t*, la cual identifica cada *thread* diferenciándola de las demás y conteniendo todos sus datos.

El prototipo de la función es el siguiente[1]. Consulte en la Web los posibles valores que puede devolver esta llamada para el tratamiento de errores (EAGAIN, EPERM):

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t *attr, void * (*start_routine) (void *),
                   void *arg)
```

- *thread*: Es una variable del tipo *pthread_t* que contendrá los datos del *thread* y que nos servirá para identificar un *thread* en concreto (ID).
- *attr*: Es un parámetro del tipo *pthread_attr_t* y que se debe inicializar previamente con los atributos que queramos que tenga el *thread*. Si pasamos como parámetro NULL la biblioteca le asignará al *thread* los atributos por defecto. La biblioteca *pthread* admite implementar hilos a nivel de usuario y a nivel de núcleo. Por defecto se hace a nivel de núcleo, de tal manera que cada hebra se pueda tratar como un proceso independiente. Si queremos manejarlas a nivel de usuario y establecerles prioridad y algoritmo de planificación habría que indicarlo en el momento de crearla, cambiando alguno de los atributos por defecto. También se pueden indicar cosas como la cantidad de reserva de memoria utilizada en la pila de la hebra, si una hebra debe o no esperar a la finalización de otra y algunas más que puede consultar en la Web. Los atributos de una hebra no se pueden modificar durante su ejecución.
- *start_routine*: Aquí pondremos la dirección de memoria de la función que queremos que ejecute el *thread*. La función debe devolver un puntero genérico (*void* *) como resultado, y debe tener como único parámetro otro puntero genérico. La ventaja de que estos dos punteros sean genéricos es que podremos devolver cualquier cosa que se nos ocurra mediante los *castings* de tipos necesarios. Si necesitamos pasar o devolver más de un parámetro a la vez, se puede crear una estructura y meter allí dentro todo lo que necesitemos. Luego pasaremos o devolveremos la dirección de esta estructura como único parámetro.
- *arg*: Es un puntero al parámetro que se le pasará a la función *start_routine* (parámetro *start_routine*). Puede ser NULL si no queremos pasar nada a la función. **Cualquier argumento pasado a la hebra se debe pasar por referencia y hacerle un *casting* a *void ***.

En caso de que todo haya ido bien, la función devuelve un 0, o un valor distinto de 0 en caso de que hubiera algún error.

Tenga cuidado al reutilizar variables que se pasan por referencia a los hilos cuando estos se crean, podría suceder que el hilo creado no se programara para ejecutarse a tiempo a fin de utilizar los valores antes de que se sobreescriban. Tenga en cuenta que está utilizando punteros y puede que al hilo no le de tiempo a copiar el parámetro en su memoria local. Consulte el ejemplo "hello_arg3.c" disponible en Moodle.

1 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_create.html

Una vez hemos llamado a esta función, ya tenemos a nuestro(s) *thread*(s) funcionando, pero ahora tenemos dos opciones: esperar a que terminen los *threads*, en el caso de que nos interese recoger algún resultado, o simplemente decirle a la biblioteca *pthread* que cuando termine la ejecución de la función del *thread* elimine todos los datos de sus tablas internas. Para ello, disponemos de dos funciones: *pthread_join()* y *pthread_detach()*.

3.3.2 Espera a la finalización de una hebra (pthread_join())

Esta función hace que el hilo invocador espere a que termine el hilo especificado. Entiéndase que por hilo podemos referirnos también al propio *main()*. Supongamos que varios hilos están realizando un cálculo y es necesario el resultado de todos ellos para obtener el resultado total por parte del *main()* o hilo principal. Para ello el hilo encargado del resultado total debe esperar a que todos los demás hilos terminen.

La función *pthread_join()* tiene el siguiente prototipo[2]. Consulte en la Web el tratamiento de errores de esta función (EDEADLK):

```
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return)
```

Esta función suspende el *thread* llamante (el que invoca a esta función) hasta que termine la ejecución del *thread* indicado por *th*. Además, una vez éste último termina, pone en *thread_return* el resultado devuelto y que estamos esperando.

- *th:* Es el identificador del thread que queremos esperar, y es el mismo que usamos al invocar a *pthread_create()*.
- *thread_return*: Es un puntero a puntero que apunta (valga la redundancia) al resultado devuelto por el *thread* que estamos esperando cuando terminó su ejecución. Ese *thread* al que esperamos devolverá un valor usando *return()* o *pthread_exit()*. Si el parámetro parámetro *thread_return* es NULL, le estamos indicando a la biblioteca que no nos importa el resultado de la hebra a la que estamos esperando.

Esta función devuelve 0 en caso de que todo esté correcto, o valor diferente de 0 si hubo algún error.

Puede utilizar la función *pthread_join()* en cualquier momento, ya que aunque una hebra haya terminado, el estado y los resultados de la misma se guardan hasta que en un determinada zona del código se llame a *pthread_join()*. Cuando se ha recibido una hebra con *pthread_join()*, sus recursos que queden en el sistema, como el estado y valor devuelto por la misma, se liberan.

pthread_join() es bloqueante, pero s*i cuando se hace esta llamada* la hebra a la que esperamos ya ha terminado, se devuelve el control a la hebra llamante en ese mismo momento, pudiéndose recoger el estado de la hebra y su posible valor devuelto mediante *pthread_exit()*, que se estudiará a continuación.

Si se hace una creación de hilos desde el *main()* es necesario poner en dicho *main()* un *pthread_join()* para que no se termine nuestro programa y se desapile de memoria antes de que terminen los hilos creados. Por defecto un *thread* es *joinable*, es decir, requiere que el proceso creador utilice *pthread_join()* para recoger su finalización. En caso contrario no tendremos control

2 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_join.html

sobre nuestro programa y depende del planificador el que nuestras hebras tengan “la suerte” o no de ejecutarse antes de que termine el proceso principal. Si una hebra *joinable* no se recibe con un *pthread_join()*, ésta termina y el proceso principal aún no ha finalizado, su estado y determinados recursos del sistema asociados a la misma seguirán estando ocupados hasta que el hilo principal termine.

Por último, si hace un *pthread_join()* de una hebra que ya está siendo esperada por otro *ptread_join()* obtendrá un error (si hace el tratamiento de errores de *pthread_join()*). Es responsabilidad del programador el hacer un solo *pthread_join()* por hebra.

El programa **demo1.c** crea dos hebras a las que se le pasa una estructura con un mensaje. Estúdielo y pruébelo. Después elimine los pthread_join y observe los resultados.

En **demo2.c** dispone de otro ejemplo del que sacar conclusiones. Estúdielo y pruébelo.

3.3.3 Finalizar una hebra y devolver resultados (pthread_exit())

La terminación de un hilo se produce cuando la función asignada al mismo y que está ejecutando termina, cuando ejecuta un *return()* o cuando se llama a *pthread_exit()*. Si la función asignada a la hebra no ejecuta ni *return()* ni *pthread_exit()* a su finalización, automáticamente y de manera transparente para el programador, se ejecuta un *pthread_exit(NULL)*. **Por tanto, *pthread_exit()* hace que el hilo invocador termine de manera normal sin causar que todo el proceso llegue a su fin.**

La función *pthread_exit()* invoca controladores de terminación de hilos, cosa que *return()* no realiza, por tanto, intente terminar sus hilos con *pthread_exit()* en vez de con *return()*. Otra diferencia de *pthread_exit()* con respecto a *return()*, es que la primera se puede llamar desde cualquier subrutina que invoque la función de la hebra, haciendo que ésta termine. Para estudiar con más profundidad las diferencias entre ambas llamadas es necesario que consulte e investigue en la Web.

Hay que tener en cuenta que *pthread_exit()* libera los recursos asociados a una hebra, pero no elimina por completo el estado en que terminó está y su valor devuelto, el cual se puede recoger posteriormente con *pthread_join()*.

El prototipo de *pthread_exit()* es el siguiente[3].

```
#include <pthread.h>
void pthread_exit (void *retval)
```

- *retval*: Es un puntero genérico a los datos que queremos devolver como resultado. Estos datos serán recogidos cuando alguien haga un *pthread_join()* con el identificador de *thread*. El parámetro es el valor que se devolverá al hilo que espera. Como es un *void **, puede ser un puntero a cualquier cosa. Según están implementadas las hebras, el valor devuelto no debe estar localizado en la pila de la hebra. **Si necesitamos que cada hebra tenga su propio grupo de datos sobre los que operar y devolver necesitaremos utilizar la función *malloc()* de C. Eso es porque el valor devuelto es un puntero, por lo que el dato debe estar en una zona de memoria localizable después de que termine la hebra, no puede ser una variable local a la hebra, ya que la variable se pierde al terminar su ejecución.**

3 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_exit.html

Estudie los programas “sampleFAIL.c”, “sampleOK.c”, “sampleOK2.c” y “sampleOK3.c” dispuestos en Moodle, observe sus resultados y obtenga conclusiones.

Por razones de portabilidad y para que un valor entero no coincida con la macro PTHREAD_CANCELED (definida como un valor entero con *casting* a void que devuelven las hebras que se cancelan), las hebras no deberían devolver número enteros, por lo que es mejor utilizar en estos casos un “*long*”. En la documentación de Moodle y en la Web puede encontrar más información sobre este tema.

Es importante que note que si en una hebra usásemos *exit()*, se terminaría el proceso completo (con todas sus hebras). Y por consiguiente, también podemos afirmar que si en un determinado momento terminamos un proceso, automáticamente también se terminarán todas las hebras asociadas al mismo.

En **demo3.c** puede consultar un ejemplo en el que se espera a un hilo al que se le pasa como parámetro un vector de enteros y no devuelve nada. Estúdielo y ejecútelo.

Modifique **demo3.c** cambiando el *pthread_exit(NULL)* por un *exit(0)*, observe sus resultados y saque sus conclusiones.

En **demo4.c** tiene otro ejemplo donde trabajar los conceptos asociados a *pthread_exit()* y return en las hebras.

A continuación en **demo5.c, demo6.c, demo7.c, demo8.c** encontrará ejemplos de paso y retorno de parámetros en la función que ejecuta el hilo.

3.3.4 Obtener la información de una hebra (pthread_self())

Para obtener la información de un hebra (entre otras su ID) utilizaremos la función pthread_self()[4]:

```
#include <pthread.h>
pthread_t pthread_self(void)
```

Esta función devuelve al *thread* que la llama su identificación, en forma de variable del tipo *pthread_t*. Se puede hacer un *casting (unsigned int) pthread_t* para imprimir el ID, busque algún ejemplo en la Web.

En **demo9.c** dispone de un pequeño ejemplo de su uso.

4 Comunicación entre procesos ligeros y sincronización

4.1 Concurrencia

Un sistema operativo multitarea permite que coexistan varios procesos activos a la vez (a nivel de proceso o a nivel de hilo), es decir, varios procesos que se están ejecutando de forma concurrente, paralela. En el caso de que exista un solo procesador con un solo núcleo, el sistema operativo se encarga de ir repartiendo el tiempo del procesador entre los distintos procesos, intercalando la ejecución de los mismos, dando así una apariencia de ejecución simultánea. Cuando existen varios procesadores o un procesador con varios núcleos, los procesos no sólo pueden intercalar su

4 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_self.html

ejecución sino también superponerla (paralelizarla). En este caso sí existe una verdadera ejecución simultánea de procesos, pudiendo ejecutar cada procesador o cada núcleo un proceso o hilo diferente.

En ambos casos (uniprocesador y multiprocesador), el sistema operativo mediante un algoritmo de planificación, otorga a cada proceso un tiempo de procesador para no dejar a otros procesos del sistema sin su uso. La finalización del tiempo de reloj ("rodaja de tiempo") otorgado a los procesos del sistema puede dar lugar a problemas de concurrencia. Estos problemas de concurrencia hacen que se produzcan inconsistencias en los recursos compartidos por 2 o más procesos (o hilos indistintamente). Veamos más sobre ello en las siguientes secciones.

4.2 Condiciones de carrera

Cuando decidimos trabajar con programas concurrentes uno de los mayores problemas con los que nos podremos encontrar, y que es inherente a la concurrencia, es el acceso a variables y/o estructuras compartidas o globales. Cuando dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, a esto se le conoce como **condiciones de carrera** . Veamos un ejemplo:

```
Hilo 1
void *funcion_hilo_1(void *arg)
{
    int resultado;
    ...
    if (i == valor_cualquiera) {
         ...
         resultado = i *
(int)*arg;
         ...
    }
     pthread_exit(&resultado);
}
```

```
Hilo 2
void *funcion_hilo_2(void *arg)
{
    int otro_resultado;
    ...
    if (funcion_sobre_arg(*arg) ==
0) {
         ...
         i = *arg;
         ...
    }
   pthread_exit(&otro_resultado);
}
```

Este código, que tiene la variable *i* como global, aparentemente es inofensivo, pero nos puede traer muchos problemas si se dan ciertas condiciones. Supongamos que el hilo 1 se empieza a ejecutar antes que el hilo 2, y que casualmente se produce un cambio de contexto (el sistema operativo suspende la tarea o hilo actual y pasa a ejecutar la siguiente) justo después de la línea que dice "*if (i == valor_cualquiera)*". La entrada en ese *if* se producirá si se cumple la condición, que suponemos que sí. Pero justo después de ese momento el sistema hace el cambio de contexto comentado anteriormente y pone a ejecutar al hilo2, que se ejecuta el tiempo suficiente como para ejecutar la línea "*i = *arg*". Al poco rato hilo 2 deja de ejecutarse y el planificador vuelve a optar por ejecutar el hilo 1, entonces, ¿qué valor tiene ahora *i*?. El hilo 1 está "suponiendo" que tiene el valor de *i* que comprobó al entrar en el *if*, pero *i* ha tomado el valor que le asignó hilo 2, con lo que el resultado que devolverá el hilo 1 después de sus cálculos posiblemente será totalmente inconsistente e inesperado.

Todo esto puede que no sucediese si el sistema tuviera muy pocos procesos en ese momento, con lo cual el sistema podría optar por que cada proceso se ejecute por más tiempo, si el procesador fuera muy rápido o y si el código del hilo 1 fuera lo suficientemente corto como para no sufrir ningún cambio de contexto en medio de su ejecución. Pero NUNCA deberemos hacer suposiciones de éste tipo, porque no sabremos hasta dónde y cuándo se van a ejecutar nuestros programas.

El problema que tienen estos *bugs* es que son difíciles de detectar en el caso de que no nos fijemos a la hora de implementar nuestro código en qué parte puede haber condiciones de carrera. Puede que a veces vaya todo a la perfección y que en otras ocasiones salga todo mal debido a las condiciones de carrera no controladas.

4.3 Sección crítica y exclusión mutua

¿Cómo evitamos las condiciones de carrera?. La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucra la memoria compartida, los archivos compartidos y cualquier otra cosa que se comparta en el sistema, es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo o mediante multiplexaciones no controladas. Esa parte del programa en la que se accede a un recurso compartido de le conoce como **región crítica** o **sección crítica**. Por tanto, un proceso está en su sección critica cuando accede a datos compartidos modificables.

Dicho esto, lo que necesitamos es **exclusión mutua** (nunca más de un proceso ejecutando la sección crítica), es decir, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo (se evita la carrera).

La siguiente figura ilustra el mecanismo de exclusión mutua en términos abstractos. Hay *n* procesos para ser ejecutados concurrentemente. Cada proceso incluye (1) una sección crítica que opera sobre algún recurso *Ra*, y (2) código adicional que precede y sucede a la sección crítica y que no involucra acceso a *Ra*. Dado que todos los procesos acceden al mismo recurso *Ra*, se desea que sólo un proceso esté en su sección crítica al mismo tiempo. Para aplicar exclusión mutua se proporcionan dos funciones o mecanismos abstractos: *entrarcritica()* y *salircritica()*. A cualquier proceso que intente entrar en su sección crítica mientras otro proceso está en su sección crítica, por el mismo recurso, se le hace esperar.

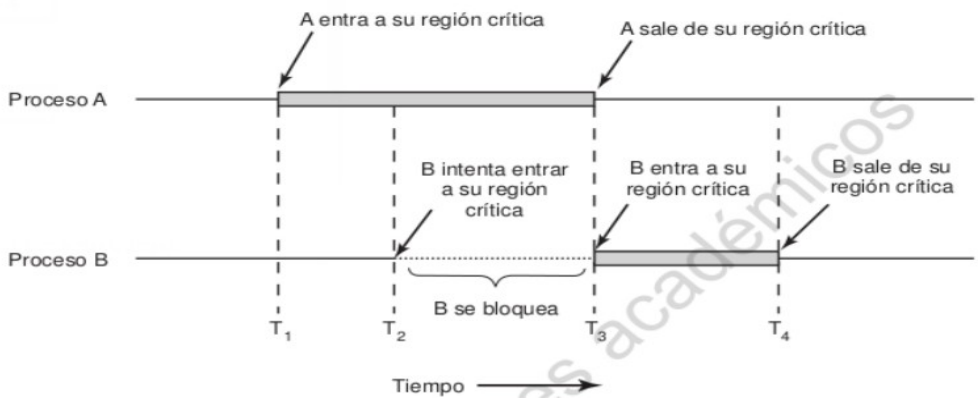
```
/* PROCESO 1 */
void P1
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

```
/* PROCESO 2 */
void P2
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

. . .

```
/* PROCESO n */
void Pn
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

Faltaría examinar mecanismos específicos para proporcionar los mecanismos *entrarcritica()* y *salircritica()*. Así, si dos procesos A y B intentasen entrar en una sección crítica, la solución proporcionada debería permitir que la exclusión mutua reflejada en la siguiente figura se hiciera correctamente:



4.4 Cerrojos o barreras mediante semáforos binarios o mutex

La biblioteca de hilos *pthread* proporciona una serie de funciones o llamadas al sistema basadas en el estándar POSIX para la resolución de problemas de exclusión mutua, lo cual hace su uso relativamente más sencillo y eficiente con respecto a los algoritmos clásicos (Dekker, Peterson, Lamport, etc). Estos mecanismos de *pthread* se llaman *mutexs* y variables de condición aunque también se conocen como semáforos binarios.

Un *mutex* (*Mutual Exclusion*) se asemeja a un semáforo porque puede tener dos estados, abierto o cerrado, los cuales servirán para proteger el acceso a una sección crítica. Cuando un semáforo está abierto (verde), al primer *thread* que pide entrar en una sección crítica se le permite el acceso y no se deja pasar a nadie más por el semáforo. Mientras que si el semáforo está cerrado (rojo) (algún *thread* ya lo usó y accedió a sección crítica poniéndolo en rojo), el *thread* que lo pide parará su ejecución hasta que se vuelva a abrir el semáforo (puesta en verde). Dicho esto, solo podrá haber un *thread* poseyendo el bloqueo de un semáforo binario o barrera, mientras que puede haber más de un *thread* esperando para entrar en una sección crítica.

Con estos conceptos se puede implementar el acceso a una sección crítica: Se pide el bloqueo del semáforo antes de entrar, éste es otorgado al primero que llega, mientras que los demás se quedan bloqueados esperando en una cola (suele ser FIFO) a que el que entró primero libere el bloqueo. Una vez el que entró en la sección crítica sale de ella debe notificarlo a la biblioteca *pthread* para que mire si había algún otro *thread* esperando para entrar en la cola y dejarlo entrar.

La siguiente figura muestra una definición de semáforo binario o *mutex*:

```
struct binary_semaphore {
    enum {cero, uno} valor;
    queueType cola;
};
void semWaitB(binary_semaphore s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
        {
            poner este proceso en s.cola;
            bloquear este proceso;
        }
}
void semSignalB(binary_semaphore s)
{
    if (esta_vacia(s.cola))
        s.valor = 1;
    else
    {
            extraer un proceso P de s.cola;
            poner el proceso P en la lista de listos;
        }
}
```

Figura 5.4. Una definición de las primitivas del semáforo binario.

A continuación se expondrán brevemente las funciones más utilizadas y que ofrece la librería *pthread* para llevar a cabo exclusión mutua. Posteriormente se expondrá algún ejemplo de su uso. El alumno deberá hacer un estudio más profundo en la Web y en otros recursos bibliográficos de los que dispone en la biblioteca de la Universidad.

4.4.1 Inicialización de un mutex *(pthread_mutex_init)*

Esta función inicializa un *mutex*[5]. Hay que llamarla antes de usar cualquiera de las funciones que trabajan con *mutex*.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

- mutex: Es un puntero a un parámetro del tipo *pthread_mutex_t*, que es el tipo de datos que usa la biblioteca *pthread* para controlar los mutex.
- attr: Es un puntero a una estructura del tipo *pthread_mutexattr_t*[6], y sirve para definir qué tipo de *mutex* queremos:

  -Normal (PTHREAD_MUTEX_NORMAL).

  -Recursivo (PTHREAD_MUTEX_RECURSIVE).

  -Errorcheck (PTHREAD_MUTEX_ERRORCHECK).

  Si el valor de attr es NULL, la biblioteca le asignará un valor por defecto, concretamente PTHREAD_MUTEX_NORMAL.

  Para crear un mutex diferente al usado por defecto, tendremos que indicarlo previamente a *pthread_mutex_init()*. Busque en la Web información sobre los tipos de *mutex* anteriormente citados.

pthread_mutex_init() devuelve 0 si se pudo crear el *mutex* o distinto de cero si hubo algún error. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en OpenGroup ([EAGAIN], [ENOMEM], [EPERM], [EINVAL]).

También podemos inicializar un *mutex* sin usar la función *pthread_mutex_init()*, basta con declararlo de esta manera (inicializa un *mutex* por defecto). Es como se hacía en antiguas implementaciones de POSIX pero todavía se permite utilizar:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

4.4.2 Petición de bloqueo de un mutex *(pthread_mutex_lock)*

Esta función[7] pide el bloqueo para entrar en una sección crítica. Si queremos implementar una sección crítica, todos los *threads* tendrán que pedir el bloqueo sobre el mismo *mutex*.

5 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_init.html
6 https://www.sourceware.org/pthreads-win32/manual/pthread_mutexattr_init.html
7 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_lock.html

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- mutex: Es un puntero al *mutex* sobre el cual queremos pedir el bloqueo o sobre el que nos bloquearemos en caso de que ya haya alguna hebra dentro de la sección crítica. Si la sección crítica ya se encuentra ocupada, es decir, alguna otra hebra bloqueó el *mutex* previamente, entonces el sistema operativo bloquea a la hebra actual que hace la invocación de *pthread_mutex_lock()* hasta que el *mutex* se libere.

Como resultado, devuelve 0 si no hubo error, o diferente de 0 si lo hubo. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en *OpenGroup*.

4.4.3 Petición de desbloqueo de un mutex *(pthread_mutex_unlock)*

Esta es la función[8] contraria a la anterior. Libera el bloqueo que tuviéramos sobre un mutex.

```
#include <pthread.h>
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

- mutex: Es el semáforo donde tenemos el bloqueo y queremos liberarlo. Al liberarlo, la sección crítica ya queda disponible para otra hebra.

Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en OpenGroup.

Cuando estamos utilizando *mutex*, es responsabilidad del programador el asegurarse de que cada hebra que necesite utilizar un *mutex* lo haga, es decir, si por ejemplo 4 hebras están actualizando los mismos datos pero solo una de ellas usa un *mutex* para hacerlo, los datos podrían corromperse.

Otro punto importante del que el programador se debe responsabilizar es que si una hebra utiliza un *pthread_mutex_lock()* bajo un determinado *mutex* para proteger su sección crítica, esa misma hebra que lo adquirió es la que debe desbloquear ese *mutex* mediante la invocación a *pthread_mutex_unlock()*. Es decir, no debemos permitir que un hilo adquiera un candado y otro hilo diferente sea el que lo libere, excepto cuando se usen variables de condición, las cuales se estudiarán en las siguientes secciones.

Si un hilo intenta desbloquear un mutex que no está bloqueado, OpenGroup no define el comportamiento de nuestro programa.

8 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_trylock.html

4.4.4 Destrucción de un mutex *(pthread_mutex_destroy)*

Esta función[9] le dice a la biblioteca que el *mutex* que le estamos indicando no lo vamos a usar más (ninguna hebra lo va a bloquear más en el futuro), y que puede liberar toda la memoria ocupada en sus estructuras internas por ese *mutex*. Esa destrucción se debe producir inmediatamente después de que se libera el *mutex* o barrera por alguna hebra. **Si se intenta destruir un *mutex* que está bloqueado por una hebra, el comportamiento de nuestro programa no está definido.** Si se quiere reutilizar un *mutex* destruido debe volver a ser reinicializado con *pthread_mutex_init()*, de lo contrario los resultados que se pueden obtener después de que ha sido destruido no están definidos.

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- mutex: El mutex que queremos destruir.

La función, como siempre, devuelve 0 si no hubo error, o distinto de 0 si lo hubo. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en OpenGroup.

4.4.5 Ejemplos de uso de mutexs

Veamos como se reescribe el código de la sección 3.2 usando *mutexs*. En color azul han sido añadidas las líneas que antes no estaban. Como se puede observar, lo que hay que hacer es inicializar el *mutex*, pedir el bloqueo antes de la sección crítica y liberarlo después de salir de la misma. Mientras más pequeñas hagamos las secciones críticas, más concurrentes serán nuestros programas, porque tendrán que esperar menos tiempo en el caso de que haya bloqueos.

9 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_destroy.html

```
Variables globales:

pthread_mutex_t mutex_acceso;
int i;
```

```
Hilo 1 (Versión correcta)
void *funcion_hilo_1(void *arg)
{
    int resultado;
    ...

pthread_mutex_lock(&mutex_acceso);
    if (i == valor_cualquiera) {
        ... <
        resultado = i * (int)*arg;
        ... <
    }
pthread_mutex_unlock(&mutex_acceso);
     pthread_exit(&resultado);
}
```

```
Hilo 2 (Versión correcta)
void *funcion_hilo_2(void *arg)
{
    int otro_resultado;
    ...
    if (funcion_sobre_arg(*arg) == 0)
      {
        ...

pthread_mutex_lock(&mutex_acceso);
        i = *arg;
pthread_mutex_unlock(&mutex_acceso);
    }
   pthread_exit(&otro_resultado);
}
```

```
int main(void)
{
     ...
     pthread_mutex_init(&mutex_acceso, NULL);
     ...
}
```

En la demo10.c se muestra otro ejemplo del uso de *mutex* para proteger una variable global que se incrementa de forma concurrente por dos hebras. Puede ejecutarlo por ejemplo con "./a.out 5", donde "5" indica el número de "*loops*" a realizar en el bucle de la función que se le asignan a las hebras implicadas, dos en este caso.

En la demo11.c se realiza un producto escalar entre dos vectores y luego muestra la suma resultante de los elementos del vector producto que se obtendría. El número de hebras usadas y la longitud de vector que utilizará cada una viene definido en las macros NUMTHRDS y VECLEN del ejemplo. Estúdielo, ejecútelo y observe sus resultados.

4.5 Semáforos generales

Un **semáforo general** es un objeto o estructura con un valor entero al que se le puede asignar un valor inicial no negativo, con una cola de procesos, y al que sólo se puede acceder utilizando dos operaciones atómicas: wait() y signal (). Su filosofía es parecida a los mutexs que estudió en el apartado anterior, ya que se basan en que dos o más procesos pueden cooperar por medio de simples señales, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya

recibido una señal específica de otro.

De manera abstracta (veremos más adelante la implementación POSIX), las operaciones *wait()* y *signal()* de un semáforo general o con contador (se entiende de aquí en adelante) son las siguientes:

```
wait (s) /* Operación atómica wait() para un semáforo "s" */
{
        s = s –1;
        if (s < 0)
        {
                Poner proceso en cola;
                Bloquear al proceso;
        }
}
```

La operación *wait()* decrementa el valor del semáforo "s". Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *wait()* se bloquea. Cuando al decrementar "s" el valor es cero o positivo, el proceso entra en la sección crítica. El semáforo se suele inicializar a 1, para que el primer proceso que llegue, A, lo decremente a cero y entre en su sección crítica. Si algún otro proceso llega después, B, y A todavía está en la sección crítica, B se bloquea porque al decrementar el contador pasa a ser negativo.

Si el valor inicial del semáforo fuera, por ejemplo, 2, entonces dos procesos podrían ejecutar la llamada *wait()* sin bloquearse y por tanto se permitiría que ambos ejecutaran de forma simultánea dentro de la sección crítica. Algo peligroso si se realizan operaciones de modificación en la sección crítica y no se controlan.

```
signal (s) /* Operación atómica signal() para un semáforo "s" */
{
        s = s + 1;
        if ( s <= 0 )
        {
                Extraer un proceso de la cola bloqueado en la operación wait();
                Poner el proceso listo para ejecutar;
        }
}
```

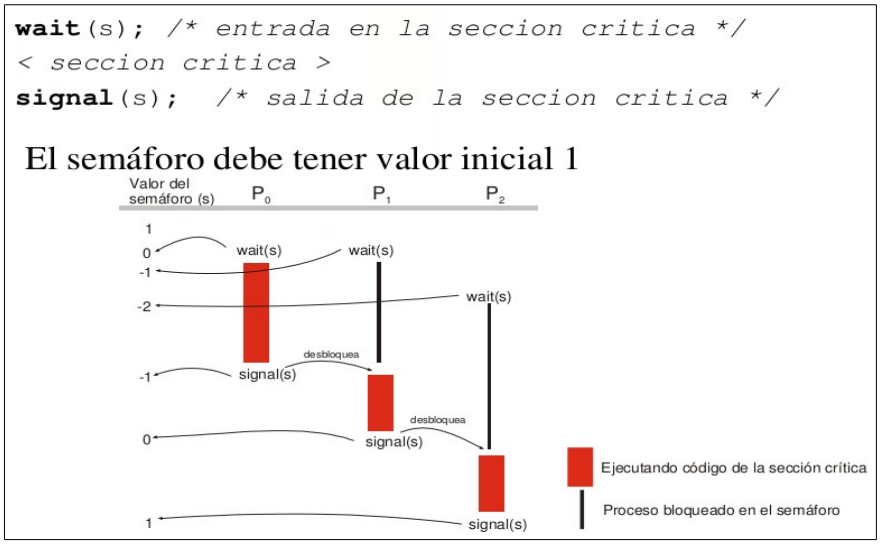
La operación *signal()* incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *wait()*.

El número de procesos, que en un instante determinado se encuentran bloqueados en una operación *wait()*, viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación *signal()*, el valor del semáforo se incrementa, y en el caso de que haya algún proceso bloqueado en una operación *wait()* anterior, se desbloqueará a un solo proceso.

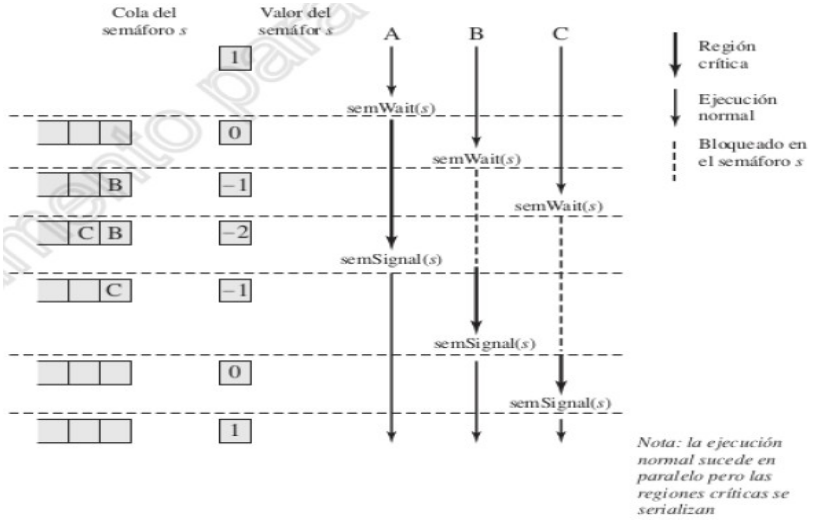
Para resolver el problema de la sección crítica utilizando semáforos debemos proteger el código que

constituye la sección crítica de la siguiente forma (ver figura):

```
wait(s); /* entrada en la seccion critica */
< seccion critica >
signal(s);  /* salida de la seccion critica */
```



La Figura siguiente figura muestra una posible secuencia de tres procesos usando la disciplina de exclusión mutua explicada anteriormente. En este ejemplo, tres procesos (A, B, C) acceden a un recurso compartido protegido por el semáforo "s". El proceso A ejecuta semWait(), y dado que el semáforo tiene el valor 1 en el momento de la invocación, A puede entrar inmediatamente en su sección crítica y el semáforo toma el valor 0. Mientras A está en su sección crítica, ambos B y C realizan una operación semWait() y son bloqueados, pendientes de la disponibilidad del semáforo. Cuando A salga de su sección crítica y realice semSignal(), B, que fue el primer proceso en la cola, podría entonces entrar en su sección crítica. Cuando B haga otro semSignal() y salga de la sección crítica, entonces C podrá entrar en la misma.



El estándar POSIX especifica una interfaz para los semáforos generales (los binarios son los mutexs). Esta interfaz no es parte de la librería *pthread*, es decir, *pthread* no proporciona funciones para el manejo de semáforos generales. A pesar de ello, la mayoría de los sistemas Unix actuales que implementan hilos también proporcionan semáforos generales a partir de otra librería, concretamente la librería *semaphore*, definida en "*semaphore.h*". Esta librería se basa en el estándar POSIX.

Existe otra implementación de semáforos, conocida como “semáforos en *Unix System V*”, que es la implementación tradicional de Unix, pero son más complejos de utilizar y no aportan más funcionalidad. Se define en *<sys/sem.h>*. A partir de la versión de kernel 2.6 de los sistemas Linux actuales, los semáforos POSIX se soportan por el sistema operativo y se puede utilizar sin problema *semaphore.h*. Si alguna vez tuviéramos que usar semáforos en máquinas con sistemas operativos muy antiguos, sería más conveniente usar los semáforos *System V*, ya que son más portables a equipos viejos.

En POSIX un semáforo se identifica mediante una variable del tipo *sem_t*. El estándar POSIX define dos tipos de semáforos:

• **Semáforos sin nombre.** Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso, o a los procesos que heredan el semáforo a partir de una llamada *fork()* (padre e hijo). En este segundo caso habría que utilizar técnicas de memoria compartida entre procesos para el acceso a la sección crítica, por lo que en esta práctica nos ceñiremos al uso de hilos.

• **Semáforos con nombre**. En este segundo tipo de semáforos, éstos llevan asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada *fork()*, en este caso los procesos tienen que abrir el semáforo con el mismo nombre. El estándar tiene en cuenta la posibilidad de su utilización para la sincronización de procesos (además de hilos), pero esta posibilidad no está soportada en todas las implementaciones y por ello, en esta asignatura, sólo serán utilizados los semáforos sin nombre.

4.5.1 Inicialización de un semáforo (sem_init())

Los semáforos deben inicializarse antes de usarlos. Para ello utilizaremos la siguiente función[10]:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, int value);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.
- pshared: Entero que determina si el semáforo sólo puede ser utilizado por hilos del mismo proceso que inicia el semáforo, en cuyo caso pondremos como parámetro el valor 0, o se puede utilizar para sincronizar procesos que lo hereden por medio de una llamada a *fork()* (! =0). En esta práctica no se contempla el último caso.
- value: Entero que representa el valor que se asigna inicialmente al semáforo.

Esta función devuelve 0 en caso de que que pueda inicializar el semáforo o -1 en caso contrario estableciendo el valor del error en *errno (*[EINVAL], [ENOSPC], [EPERM]*)*. Consulte *OpenGroup*.

4.5.2 Petición de bloqueo o bajada del semáforo (sem_wait())

Para entrar en la sección crítica por parte de un proceso antes se debe consultar el semáforo con

10 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_init.html

sem_wait()[11]:

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.

Como se comento al principio de la práctica, al realizar esta llamada, el contador del semáforo se decrementa. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *sem_wait()* se bloquea.

Esta función devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*.

Otra función similar es *sem_trywait()*:

```
#include <semaphore.h>
int sem_trywait(sem_t *sem);
```

La función *sem_trywait()* es igual que *sem_wait()* pero no bloqueante, es decir, si el valor del semáforo al decrementarse llega a negativo el hilo invocador de *sem_trywait()* no se bloquea sino que continua con su ejecución. En este caso la función *sem_trywait()* devuelve el valor -1, estableciendo *errno* el valor EAGAIN. Consulte *OpenGroup*. En caso de que al invocarse el decremento no llegase a negativo la función *sem_trywait()* devuelve 0 y el proceso entra en sección crítica. Por tanto es una función que permite a un proceso seguir con la siguiente sentencia de código despues de *sem_trywait()* en caso de que la sección crítica no hay sido desbloqueada. En función de eso quizás nos interese que nuestro hilo haga otras tareas alternativas.

4.5.3 Petición de desbloqueo o subida del semáforo (sem_post())

Cuando un proceso va a salir de la sección crítica es necesario que envíe una señal indicando que ésta queda libre, y eso lo hace con la función *sem_post()*[12] (equivalente al *semSignal()* teórico comentado al principio de la práctica):

```
#include <semaphore.h>
int sem_post(sem_t *sem);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.

La operación *sem_post()* incrementa el valor del semáforo. Si al hacer el incremento el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *sem_wait()*, normalmente se suele emplear para ello un esquema FIFO (First In First Out). Si al hacer el incremento el valor es > 0, entonces es que no hay hilos bloqueados y pueden entrar en sección crítica tanto hilos como valor tenga el número positivo. Es responsabilidad del programador el controlar posibles *sem_post()* que incrementen un semáforo a un valor positivo > 1.

11 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_wait.html
12 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_post.html

Esta función devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup*.

4.5.4 Examinar el valor de un semáforo (sem_getvalue())

Si necesitamos saber cuál es el valor de un semáforo podemos consultarlo con la siguiente función[13]:

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.
- sval: Puntero a variable de tipo entero en la que se depositará el valor del semáforo.

El uso de *sem_getvalue()* no altera el valor del semáforo. *sval* mantiene el valor que tenía el semáforo en algún momento no especificado durante la llamada, pero no necesariamente el valor en el momento de la devolución. Tenga en cuenta que puede haber más de un hilo intentando acceder al semáforo indicado como primer parámetro, o simplemente, por ejemplo, antes de que *sem_getvalue()* nos devuelva algo un hilo puede haber salido de la sección crítica.

Si el semáforo está bloqueado, *sval* contendrá el valor cero o un valor negativo que indica, en valor absoluto, el número de subprocesos en espera.

Esta función devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup*

4.5.5 Destrucción de un semáforo (sem_destroy())

Para destruir un semáforo previamente creado con *sem_init()* se utiliza la siguiente función[14]:

```
#include <semaphore.h>
int sem_destroy(sem_t *sem);
```

- sem: Puntero al semáforo a destruir para liberar recursos.

Esta función devuelve 0 en caso de que que pueda destruir el semáforo o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup* para el tratamiento de errores.

4.5.6 Ejemplos

La **demo12.c** crea dos hilos. Cada hilo hace exactamente lo mismo, incrementar una variable global un numero de veces. El resultado final debe ser el doble del numero de veces que se incrementa la variable, ya que cada hilo la incrementa el mismo número de veces. Si no hubiera semáforo general, el resultado podría ser inconsistente. Pruebe a ejecutarlo con semáforos y quitando los semáforos.

13 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_getvalue.html
14 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_destroy.html

La **demo13.c** es un programa de sincronización entre hilos que suma los números impares entre 0 y 20, es decir, los números 1+3+5+7+9+11+13+15+17+19 = 100. El primer hilo comprueba que el número es impar, si lo es deja que el segundo hilo lo sume, sino comprueba el siguiente número. Cópielo y ejecútelo para comprobar su salida. Haga una traza en papel de su funcionamiento teniendo en cuenta varios casos en los que el procesador da rodaja de tiempo al hilo P1 o al hilo P2.

La **demo14.c** es la implementación del problema lectores-escritores donde hay 2 lectores y 2 escritores (prioridad a los lectores). Un buffer con un solo dato que se va incrementando. Compílelo, ejecútelo y observe su salida. Haga varias trazas del comportamiento del programa para poder entender los semáforos.

5 Ejercicios prácticos

5.1 Ejercicio1

Implemente un programa que cree dos hebras. Cada hebra ejecutará una función a la que se le pasará como parámetro una cadena, concretamente a la primera hebra se le pasará la cadena “hola” y a la segunda “mundo”. La función que deben ejecutar ambas debe imprimir carácter a carácter la cadena recibida, haciendo un *sleep(1)* entre cada impresión de carácter. Observe los resultados obtenidos.

Repita lo mismo pero recogiendo las dos cadenas por la linea de argumentos.

5.2 Ejercicio2

Implemente un programa que cree un número *N* de hebras. Cada hebra creará 2 números aleatorios (consulte la web para la generación de aleatorios) y guardará su suma en una variable para ello, que será devuelta a la hebra llamadora (la que invocó *pthread_create()*). La hebra principal ira recogiendo los valores devueltos por las *N* hebras y los ira sumando en una variable no global cuyo resultado mostrará al final por pantalla. Para ver que los resultados finales son los que usted espera, muestre los números que va creando cada hebra y su suma, de forma que pueda comparar esas sumas parciales con la suma final de todos los números creados por todas las hebras. Utilice macros definidas y comprobación de errores en sus programas (*errno* y comprobación de valores devueltos en cada llamada), será valorado en el examen final de la asignatura.

5.3 Ejercicio3

Implementar un programa para realizar la suma en forma paralela de los valores de un vector de 10 números enteros que van de 0 a 9 (puede probar con aleatorios). Utilice una cantidad de hilos indicada como parámetro de entrada por la linea de argumentos y reparta la suma del vector entre ellos (como considere oportuno). La suma debe ser el subtotal devuelto por cada hilo. Haga comprobación de errores en su programa.

5.4 Ejercicio4

Implemente un programa que cuente las líneas de los ficheros de texto que se le pasen como parámetros y al final muestre también el número de líneas totales (contando las de todos los ficheros juntos). Ejemplo de llamada: ./a.out fichero1 fichero2 fichero3

Debe crear un hilo por fichero obtenido por linea de argumentos, de forma que todos los ficheros se cuenten de manera paralela.

5.5 Ejercicio5

Realice la multiplicación en paralelo de una matriz de 3x3 por un vector de 3x1. Para ello cree tres hebras que se repartan las filas de la matriz y del vector. Cada hijo debe imprimir la fila que le ha tocado y el resultado final de la multiplicación la cual además envía al padre. El padre debe esperar por la terminación de cada hijo y mostrar el vector resultante.

5.6 Ejercicio6

Como se comentó en la práctica 1, los procesos a diferencia de los hilos, no comparten el mismo espacio de memoria, por lo que si queremos que accedan a las mismas variables en memoria, estos deben compartirla. Realice un programa que expanda N procesos hijos. Cada hijo debe compartir una variable denominada contador, que debe estar inicializada a cero. Esta variable debe ser incrementada por cada hilo 100 veces. Imprima la variable una vez finalicen los hilos y analice el resultado obtenido. Un resultado previsible para 3 procesos sería 300, así ha sido?

5.7 Ejercicio7

Una tienda que vende camisetas, guarda en la base de datos (buffer) las cantidades de camisetas según el modelo, (camisetas[5], indica que son 5 modelos de camisetas y cada elemento de este buffer guarda las cantidades iniciales de cada una). Realizar un programa que genere N clientes y M proveedores (lo misma cantidad de proveedores que modelos de camiseta).

Para **simular una compra**, cada hilo Cliente debe generar un valor aleatorio para el modelo de camiseta y otro para la cantidad a comprar. Esta cantidad debe decrementar el stock de la camiseta en cuestión.

Para **simular un suministro**, cada Proveedor debe hacer lo mismo que el Cliente pero en este caso, incrementando el stock de la camiseta en cuestión.

Utilice semáforos binarios para resolver este problema de concurrencia imprimiendo el buffer antes de generar los hilos y al final del programa para comprobar que se ha ejecutado correctamente.

5.8 Ejercicio8

En concurrencia, el problema del productor-consumidor[15] es un problema típico, y su enunciado general es el siguiente:

Hay un proceso generando algún tipo de datos (registros, caracteres, aumento de variables, modificaciones en arrays, modificación de ficheros, etc) y poniéndolos en un *buffer*. Hay un consumidor que está extrayendo datos de dicho *buffer* de uno en uno.

El sistema está obligado a impedir la superposición de las operaciones sobre los datos, es decir, sólo un agente (productor o consumidor) puede acceder al *buffer* en un momento dado (así el productor no sobrescribe un elemento que esté leyendo el consumidor, y viceversa). Estaríamos hablando de

15 http://es.wikipedia.org/wiki/Problema_Productor-Consumidor

la sección crítica.

Si suponemos que el *buffer* es limitado y está completo, el productor debe esperar hasta que el consumidor lea al menos un elemento para así poder seguir almacenando datos. En el caso de que el *buffer* esté vacío el consumidor debe esperar a que se escriba información nueva por parte del productor.

Existen variaciones del problema productor-consumidor, como por ejemplo en el caso de que el *buffer* sea ilimitado. Consulte la Web si desea obtener más información sobre ello.

Una vez haya estudiado más detenidamente el problema del productor-consumidor realice las siguientes tareas:

Implemente el problema para hilos teniendo en cuenta que la sección crítica va a ser un *array* de enteros con una capacidad de 5 elementos. Haga una implementación usando mutexs.

5.9 Ejercicio9

Resuelva el problema del productor-consumidor con un buffer circular y acotado usando semáforos generales en vez de semáforos binarios. Haga su programa genérico para que se pueda indicar el tamaño del buffer y la cantidad de datos a producir-consumir.