Hilos

Sistemas Operativos (SO)
Grado en Ingeniería de Datos e Inteligencia Artificial

Universidad de León

Resumen

El objetivo de esta práctica utilizar llamadas al sistema relacionadas con la identificación, creación y terminación de threads; utilizar llamadas al sistema para terminar threads y para conocer el estado en el que finalizan otros threads; escribir programas que den lugar a la ejecución de múltiples threads de manera concurrente; utilizar mecanismos de gestión de la concurrencia (Semáforos y variables de condición); y escribir programas que den lugar a la ejecución de múltiples threads de manera concurrente gestionando de manera ordenada su interacción

Índice

1. Concepto de hilo

Un thread (hilo) es flujo de ejecución que pertenece a un proceso, es decir, los hilos se ejecutan de forma concurrente y comparten el mismo espacio de direcciones. Un hilo y un proceso son por tanto conceptos similares, indicados para resolver situaciones en las que un programa deba realizar varias tareas de manera simultánea (por ejemplo: mostrar

una ventana y realizar un cálculo). Tanto procesos, como hilos podrían resolver este tipo situaciones, sin embargo el uso de hilos implica una menor carga para la maquina ya que la creación, sincronización y comunicación de hilos implica menos consumo de recursos (CPU, memoria). Cada hilo compartirá con el resto de hilos del proceso:

- Espacio de memoria: Lo que implica que ejecuta el mismo código, en la misma zona de memoria. Esta es una característica de especial importancia ya que cualquier puntero hará referencia a la misma posición de memoria.
- Variables globales: La modificación de una variable global en un hilo, la modifica para el resto de los hilos.
- Ficheros abiertos.
- Procesos hijos.
- Señales y Temporizadores.

Por otra parte, cada hilo tiene:

- Sus propios registros (SP, PC, etc.): Los registros son independientes entre cada uno de los hilos.
- Su propia pila: Las variables locales se almacenan en la pila, por tanto los hilos tendrán variables locales independientes. A este respecto hay que tener en cuenta que en aunque la pila esta separada, sigue perteneciendo al mismo espacio de direcciones de todo el proceso.
- Su estado.

Cada proceso podrá tener uno o más hilos de ejecución. Cada hilo de ejecución tendrá su propio identificador (tid) que sólo será valido para los hilos del mismo proceso. A diferencia de los identificadores de proceso pid que son únicos para todos los procesos.

1.1. Planificación de threads. Tipos de threads

Cada hilo dispone de su propia política de planificación. Dependiendo del tipo de hilo, la planificación será gestionada por el sistema operativo, o por el planificador de hilos del proceso:

- Hilos de Usuario: El sistema operativo no tiene constancia de este hilo, será planificado dentro del proceso.
- Hilos de Núcleo: El sistema operativo es el encargado de planificar el hilo.

Dependiendo de la forma de hacer corresponder los hilos de usuario en los hilos del núcleo, existen tres tipos diferentes:

- Muchos-a-uno (Many-to-one): Varios hilos de usuario se implementan como un solo hilo de núcleo.
  - Ejemplo: Biblioteca light weighted processes del sistema operativo SunOS.
- Uno-a-uno (One-o-one): Cada hilo de usuario tiene correspondencia con un hilo del núcleo.
  - Ejemplo: Linux y Windows.
- Muchos-a-muchos (Many-to-many): Varios hilos de usuario tienen correspondencia con varios hilos de núcleo.
  - Ejemplo: Tru64 (antiguo Digital UNIX).

2. Biblioteca pthreads.h

El uso de hilos en Linux se realiza por medio de interfaz POSIX pthreads. Este interfaz esta descrito en la biblioteca pthreads.h de C.

2.1. Creación y finalización de threads

Para crear un nuevo hilo se invoca la función pthread_create():

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

con los siguientes argumentos:

- thread: Esta función retorna el identificador del hilo (tid) en la variable.
- attr: Si indica las características de hilo que se va a crear (NULL para utilizar los atributos por defecto).

- start: Un puntero a la función que ejecutara el hilo.
- arg: Argumentos de la función.

Para finalizar un hilo se invoca a la función pthread_exit():

```
void pthread_exit(void *retval);
```

con los siguientes argumentos:

- retval: Valor de retorno.

Todas las funciones de hilos necesitan incluir la biblioteca pthread.h. Es importante resaltar que cuando se crea un nuevo hilo, no se duplica toda la información del proceso, como sucede cuando se invoca la función fork(). Es decir, toda la información que esta almacenada el PCB (Process Control Block) señala a la del proceso que creo el hilo. El siguiente programa crea dos hilos, que comienzan utilizando la misma función, pero con argumentos diferentes:

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

void *Hilo(void *arg) {
  printf ("%s\n",(char *)arg);
  pthread_exit (NULL);
}

int main() {
  pthread_t t1, t2;
  pthread_create (&t1, NULL, Hilo, "Ejecuta un hilo");
  pthread_create (&t2, NULL, Hilo, "Ejecuta otro hilo");
  printf("Fin del thread principal\n");
  return 0;
}
```

Para compilar utilizando la biblioteca pthread.h se le debe indicar al compilador que utilice esta biblioteca de la siguiente manera:

```
$ gcc -o ejemplo1 ejemplo1.c -lpthread
```

2.2. Atributos de un thread

Cuando se crea un hilo mediante la función pthread_create() se pueden definir varios atributos referentes al nuevo hilo:

- Devolución de los recursos.
- Ámbito: Usuario o núcleo.
- Otros: Tamaño de la pila , ubicación de la pila, etc.

Los atributos se indicarán en el segundo argumento de pthread_create(). Antes de invocar esta función se debe inicializar los atributos con que deseamos que utilice este hilo (crear los atributos). Una vez que se haya creado el hilo se deben destruir los atributos (eliminar los atributos). Esto se realiza con las funciones pthread_attr_init() y pthread_attr_destroy():

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

pthread_attr_init() inicializa los atributos (attr), para ser posteriormente utilizados en pthread_create(); devuelve 0 si no se ha producido un error. pthread_attr_destroy() destruye los atributos.

2.3. Devolución de recursos. Threads detached vs. threads joinable

Cuando un hilo finaliza, puede devolver los recursos que esta utilizando de dos formas:

- Detached: El hilo es autónomo, una vez que finaliza devuelve todos los recursos que esta utilizando (tid, pila, etc.) .
- Joinable: Cuando finaliza, retiene todos los recursos hasta que un hilo invoca la función pthread_join().

Para inicializar la devolución de recursos se utilizara la función pthread_attr_getdetachstate():

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

con los argumentos:

- attr: Atributo a modificar.
- detachstate: Tipo de devolución de recursos:
  - PTHREAD_CREATE_JOINABLE
  - PTHREAD_CREATE_DETACHED

Al invocar la función pthread_join() el hilo que la invoca se queda detenido hasta que el hilo indicado finalice:

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

donde:

- thread: Es el identificador del thread al que se espera.
- retval: Es el valor retornado por pthread_exit().

y donde pthread_exit()sería:

```
void pthread_exit(void *value_ptr);
```

donde:

- value_ptr: Es el valor que se recupera con pthread_join().

El siguiente ejemplo muestra como la función pthread_exit() envía retorno a pthread_join():

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

int return_value=10;

void *Hilo2(void *arg) {
  printf("%s\n", (char *)arg);
  sleep(10);
  pthread_exit(NULL);
}

void *Hilo1(void *arg) {
  printf ("%s\n", (char *)arg);
  pthread_exit((void *)return_value);
}

int main() {
  pthread_t t1, t2;
  pthread_attr_t attr1, attr2;
  int ret1;

  // Inicializacion del attributo
  if (pthread_attr_init(&attr1) != 0) {
    perror("Error en la creación de la estructura de los atributos.");
    exit(-1);
  }
  if (pthread_attr_init(&attr2) != 0) {
    perror("Error en la creación de la estructura de los atributos.");
    exit(-1);
  }

  pthread_attr_setdetachstate(&attr1, PTHREAD_CREATE_JOINABLE);
  pthread_attr_setdetachstate(&attr2, PTHREAD_CREATE_DETACHED);
  pthread_create (&t1, &attr1, Hilo1, "Soy joinable");
  pthread_create (&t2, &attr2, Hilo2, "Soy detached");
  pthread_join(t1, (void **)&ret1);
  printf("El thread joinable terminó con retval='%d'\n", ret1);
  return 0;
}
```

2.4. Ámbito. Threads de usuario vs. Threads de núcleo

Los hilos se pueden ejecutar como hilos de usuario, donde será la propia biblioteca pthread.h la encargada de planificar los hilos. De esta manera, el sistema operativo desconoce la existencia de un nuevo hilo. Para el kernel, será simplemente un solo proceso. O bien, se pueden ejecutar como hilos de núcleo. En este modo es el kernel del sistema operativo el que se encarga de planificar los hilos. Los hilos de usuario son mas "ligeros" ya que cada vez que se conmuta de un hilo a otro no es necesario que intervenga el kernel del sistema operativo. Para inicializar el ámbito de un hilo utilizará la función pthread_attr_setscope ():

```
int pthread_attr_setscope (const pthread_attr_t *attr, int *scope);
```

con los argumentos:

- attr: Atributo a modificar:
- scope: Tipo de hilo:
  - PTHREAD_SCOPE_SYSTEM: Para hilos de núcleo.
  - PTHREAD_SCOPE_PROCESS: Para hilos de usuario.

Las threads de Linux (pthread.h) sólo soportan los hilos de núcleo. Si intentamos poner ámbito de usuario en Linux, la función anterior nos devolverá un código de error. Sin embargo la norma POSIX lo permite.

2.5. Identificación de threads

Para obtener la identificación de hilo (tid) se dispone de la función pthread_t pthread_self():

```
pthread_t pthread_self(void);
```

El valor que retorna es únicamente valido dentro del mismo proceso. Este valor no coincide con el valor que muestra el comando `ps`[1]. Si lo que deseamos es mostrar el valor del identificador de thread que muestra el comando `ps` en LINUX, debemos utilizar la siguiente función:

```
pid_t gettid(void) {
  return syscall(__NR_gettid);
}
/* Requiere #include <sys/syscall.h> *
* Esta función no esta implementada en ninguna biblioteca *
* por lo que debemos codificarla en cada programa. */
```

Lamentablemente esta función no es POSIX (no esta incluida en ninguna biblioteca), ya que hace una llamada al sistema operativo para obtener el valor SPID. Por tanto, si se desea hacer un programa portable POSIX no se debe utilizar.

Ejemplo:

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

pid_t gettid(void) {
  return syscall(__NR_gettid);
}

void *pruebaSpid(void *ptr) {
  printf("Thread secundario: PID=%d, SPID=%d\n", getpid(), gettid());
  sleep(10);
  pthread_exit(NULL);
}

int main() {
  pthread_t thread1;
  pthread_create(&thread1, NULL, pruebaSpid, NULL);
  printf("Thread principal: PID=%d, SPID=%d\n", getpid(), gettid());
  sleep(20);
  return 0;
}
```

2.6. Terminación de threads

Para obligar a un hilo a que termine se invoca la función pthread_cancel() pasando como argumento el tid del hilo que se desea cancelar:

```
int pthread_cancel(pthread_t thread);
```

Ejemplo:

[1] Con el comando process status (ps) se pueden listar los hilos de cada proceso: `ps -T`. Este comando muestra el pid y el identificador de hilo spid. Se puede combinar con la opción *–p* para mostrar los hilos de un proceso concreto. Por ejemplo, `ps -p 2345 -T`, muestra el listado de todos los hilos del proceso 2345.

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

void *Hilo (void *arg) {
  while(1) {
    printf ("%s\n", (char *)arg);
    sleep(1);
  }
}

int main() {
  pthread_t th1;
  pthread_create(&th1, NULL, Hilo,"Looped thread");
  printf("Espero 10 segundos antes de cancelar este thread.\n");
  sleep(10);
  pthread_cancel(th1);
  printf("Cancelado.\n");
  return 0;
}
```

2.7. Ejercicios

2.7.1. Espacio de direcciones en threads vs. espacio de direcciones en procesos

Examinemos el siguiente programa:

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

int a=0;

void *Hilo1(void *arg) {
  printf ("EL valor de a en el hilo es %d\n", a);
  a++;
  printf ("Nuevo valor de a en el hilo es %d\n", a);
  pthread_exit(NULL);
}

int main() {
  pthread_t t1;
  pthread_create(&t1, NULL, Hilo1, "Soy joinable\n");
  pthread_join(t1, NULL);
  printf("El valor después de ejecutar el thread es %d\n", a);
  return 0;
}
```

Este programa crea un thread que modifica la variable a y termina.

- ¿Qué valor muestra el thread principal?
- ¿Qué conclusión extrae sobre el espacio de direcciones que utilizan los threads?

El siguiente código es similar al anterior, pero utiliza procesos en lugar de threads:

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

int a=0;
int main() {
  pid_t hijo=fork();
  if (hijo==-1) {
    perror("Error en la llamada a fork()");
    exit(-1);
  } else if (hijo==0) {
    printf("El valor de a en el proceso hijo es %d\n", a);
    a++;
    printf("Nuevo valor de en el proceso hijo es %d\n", a);
  } else {
    hijo=wait(NULL);
    printf("El valor después de crear/terminar el proceso hijo es %d\n", a);
  }
  return 0;
}
```

Ahora es un proceso el que modifica una variable global:

- ¿Qué valor muestra el proceso hijo?.

- Una vez finalizado este proceso, ¿Qué valor muestra el proceso padre?
- ¿Qué conclusión extraemos sobre el espacio de direcciones que utilizan los procesos?

2.7.2. Terminación de procesos con threads

Analiza el siguiente código:

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

void *hiloColgado(void *ptr) {
  while(1) {
    printf("%s: PID=%d\n", (char *)ptr, getpid());
    sleep(1);
  }
}

void *hiloAsesino(void *ptr) {
  printf("%s: PID=%d. Voy a esperar 10 segundos y termino todo este lío.\n", (char *)ptr, getpid());
  sleep(10);
  exit(0);
}

int main() {
  pthread_t thread1, thread2;
  char *message1 = "Hilo colgado";
  char *message2 = "Hilo asesino de proceso";
  pthread_create(&thread1, NULL, hiloColgado, (void*)message1);
  pthread_create(&thread2, NULL, hiloAsesino, (void*)message2);
  while (1){
    printf("Hilo principal: PID=%d\n", getpid());
    sleep(1);
  }
  return 0;
}
```

El único hilo que termina es el que invoca la función hiloAsesino, entonces, ¿Por qué terminan el resto de los hilos?

2.7.3. Terminación de hilos

Analiza el siguiente código:

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

void *hilo(void *ptr) {
  int i, x=*(int*)ptr;
  for (i=0; i<x; i++) {
    printf("Voy a repetir %d y estoy en la iteración %d\n", x, i);
    sleep(1);
  }
}

int main() {
  pthread_t thread1, thread2;
  int n1=5, n2=10;
  pthread_create(&thread1, NULL, hilo, (void*)&n1);
  pthread_create(&thread2, NULL, hilo, (void*)&n2);
  return 0;
}
```

Los hilos del thread1 y thread2 no se ejecutan completamente. ¿Qué es lo que sucede? Modifica la función main para que los hilos se ejecuten correctamente.

2.7.4. Sincronización de hilos

Analiza el siguiente código:

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

```c
#define N 15.0

void *tareaHilo(void *ptr) {
  int x;
  printf("COMIENZO TAREA HILO %d\n", *(int *)ptr);
  srand ((int)pthread_self());
  x = 1+(int)(N*rand()/RAND_MAX+1.0); // X es un número aleatorio entre 1 y N
  sleep(x);
  printf("FIN TAREA HILO %d\n",*(int *)ptr);
}

int main() {
  pthread_t thread1;
  int hilo = 1;
  pthread_create(&thread1, NULL, tareaHilo, &hilo);
  sleep(1);
  hilo++;
  tareaHilo(&hilo);
  pthread_exit(NULL);
  return 0;
}
```

El resultado de ejecutar el programa es:

```
COMIENZO TAREA HILO 1
COMIENZO TAREA HILO 2
FIN TAREA HILO 2
FIN TAREA HILO 2
```

¿Como explicas que no aparezca FIN TAREA 1? Cree un programa que sincronice los dos hilos utilizando pthread_join(), de tal manera que el programa primero realice la tarea del hilo 1 y después la del hilo 2 (hilo principal). Por tanto, la salida del programa será:

```
COMIENZO TAREA HILO 1
FIN TAREA HILO 1
COMIENZO TAREA HILO 2
FIN TAREA HILO 2
```

Compara esta sincronización con la que realizaremos en el siguiente punto.

2.7.5. Sincronización de varios hilos

Analiza el siguiente código:

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

#define N 5.0

void *hilo(void *ptr) {
  int x;
  printf("COMIENZO TAREA HILO\n");
  srand((int)pthread_self());
  x=1+(int)(N*rand()/ RAND_MAX +1.0); // X es un número aleatorio entre 1 y N
  sleep(x);
  printf("FIN TAREA HILO: Tiempo ejecucion %d\n",x);
}

int main() {
  pthread_t t1, t2, t3, t4, t5;
  pthread_create(&t1, NULL, hilo, NULL);
  pthread_create(&t2, NULL, hilo, NULL);
  pthread_create(&t3, NULL, hilo, NULL);
  pthread_create(&t4, NULL, hilo, NULL);
  pthread_create(&t5, NULL, hilo, NULL);
  printf("Van a comenzar los hilos :\n");
  pthread_exit(NULL);
  return 0;
}
```

Todos los hilos están sin sincronizar, es decir, comienzan y terminan sin esperar unos a otros. Modifica el programa para queden sincronizados. Para hacerlo, NO puedes utilizar la función pthread_join() en la función main(). Todos los hilos serán ejecutados de forma concurrente, de tal manera que el hilo t1, espere a que finalice el hilo principal, el t2 al t1, el t3 al t2, etc. Por tanto el programa mostrará por pantalla:

```
Van a comenzar los hilos:
COMIENZO TAREA HILO
```

```
FIN TAREA HILO : Tiempo ejecución 2
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 6
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 5
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 4
COMIENZO TAREA HILO
FIN TAREA HILO : Tiempo ejecución 3
```

3. Mecanismos de sincronización con threads en LINUX

3.1. Exclusión mutua. Semáforos (mutex)

El mutex es un semáforo binario con dos operaciones atómicas:

- lock(): Intenta bloquear un mutex. Si está bloqueado el hilo se suspende hasta que otro hilo realice una operación unlock().
- unlock(): Desbloquea el mutex. Si existen hilos bloqueados se desbloquea a uno de ellos.

Para utilizar los mutex, primero se deben crear mediante la función pthread_mutex_init():

```
int pthread_mutex_init(pthread_mutex_t * mutex, const pthread_mutexattr_t *attr);
```

con los siguientes argumentos:

- mutex: mutex que se inicializa.
- attr: Parámetros de inicialización. Para inicializar con parámetros por defecto utilizamos NULL, en ese caso el mutex se inicializa desbloqueado.

pthread_mutex_init() retorna 0 si no se ha producido un error. Si se produce un error retorna el código de error.
Para eliminar un mutex se utiliza la función pthread_mutex_destroy():

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

pthread_mutex_destroy() retorna 0, si no se ha producido un error. Si se produce un error retorna el código de error.
Para bloquear un mutex se utiliza la función pthread_mutex_lock():

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

pthread_mutex_lock() retorna 0, si no se ha producido un error. Si se produce un error retorna el código de error.
Para desbloquear un mutex se invoca a la función pthread_mutex_unlock():

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

pthread_mutex_unlock() retorna 0, si no se ha producido un error. Si se produce un error retorna el código de error.
El siguiente código muestra como realizar una exclusión mutua mediante semáforos:

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


void *Hilo1(void *arg) {
  pthread_mutex_lock(arg);
  printf("Comienza un hilo\n");
  sleep(5);
  printf("Finaliza un hilo\n");
  pthread_mutex_unlock(arg);
  pthread_exit(NULL);
} /* Fin de Hilo */

int main() {
  pthread_t t1,t2;
  pthread_mutex_t misemaforo;
  // Creamos un semaforo
  if (pthread_mutex_init(&misemaforo, NULL)!=0)
    exit(-1);
  pthread_create(&t1, NULL, Hilo1, &misemaforo);
  pthread_create(&t2, NULL, Hilo1, &misemaforo);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  if (pthread_mutex_destroy(&misemaforo)!=0)
    exit(-1);
  return 0;
}
```

El siguiente código resuelve (parcialmente) la cuestión 5 de la práctica anterior utilizando semáforos:

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

#define N 5.0
pthread_mutex_t mutex;
int i[5]={1,2,3,4,5};

void *hilo(void *ptr) {
  int x;
  pthread_mutex_lock(&mutex);
  printf("COMIENZO TAREA HILO %d\n",*(int *)ptr);
  srand((int)pthread_self());
  x=1+(int)(N*rand()/RAND_MAX+1.0); // X es un número aleatorio entre 1 y N
  sleep(x);
  printf("FIN TAREA HILO %d: Tiempo ejecución %d\n", *(int *)ptr, x);
  pthread_mutex_unlock(&mutex);
}

int main() {
  pthread_t t1, t2, t3, t4, t5;
  pthread_mutex_init(&mutex, NULL);
  pthread_create(&t1, NULL, hilo, (void *)&i[0]);
  pthread_create(&t2, NULL, hilo, (void *)&i[1]);
  pthread_create(&t3, NULL, hilo, (void *)&i[2]);
  pthread_create(&t4, NULL, hilo, (void *)&i[3]);
  pthread_create(&t5, NULL, hilo, (void *)&i[4]);
  pthread_exit(NULL);
  return 0;
}
```

Este código no resuelve totalmente la cuestión 5, ya que pedía que los hilos se sincronizasen en orden. Es decir, el primer hilo que tendría que entrar en la exclusión mutua sería el t1, el segundo t2, etc. En el código anterior, si bien garantizamos la exclusión mutua, dejamos en manos del mutex el orden de entrada a esta zona crítica.

3.2. Variables de condición

Una variable de condición es un objeto de sincronización que permite bloquear a un hilo hasta que otro decide reactivarlo. Una variable de condición siempre está asociada a un mutex. Las operaciones atómicas que realiza son:

- Esperar una condición (wait):
  - Esta condición se debe realizar con el mutex cerrado (lock).
  - El hilo se suspende hasta que otro señaliza la condición y el mutex asociado se desbloquea.
- Señalizar una condición (signal) :
  - Se señaliza esta condición.
  - Si no existen hilos suspendido por esta operación no tiene ninguna consecuencia.
  - Si existe un hilo, o más, suspendido/s, se activa/n y pasa a competir por el mutex asociado.

Para esperar una condición se utiliza la función pthread_cond_wait():

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

con los siguientes argumentos:

- cond: Condición.
- mutex: mutex asociado a esta señal.

pthread_cond_wait() retorna 0 si no se ha producido ningún error.
Para señalizar una condición se utiliza la función pthread_cond_signal():

```
int pthread_cond_signal(pthread_cond_t *cond);
```

con los siguientes argumentos:

- cond: Condición.

pthread_cond_signal() retorna 0 si no se ha producido ningún error. El contenido de la variable de condición es opaco para el programador. Es decir, las variables declaradas pthread_cond_t sólo se utilizan para realizar operaciones wait y signal, el contenido de estas variables carece de significado para el programador.

Antes de utilizar un mutex este se debe inicializar mediante la función pthread_cond_init():

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

con los siguientes argumentos:

- cond: Variable de condición que se va a inicializar.
- attr: Atributos de la variable de condición. Usaremos NULL para las opciones por defecto.

pthread_cond_init() retorna 0 si no se ha producido ningún error.

Una vez que no se vaya a utilizar más una variable de condición, se destruiye para liberar recursos mediante la función pthread_cond_destroy():

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

pthread_cond_destroy() retorna 0 si no se ha producido ningún error.

Con las variables de condición podemos resolver la cuestión 5 de la práctica anterior de manera que se especifique el orden de ejecución de los hilos:

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

#define N 5.0

pthread_mutex_t mutex;

int i[5]={1,2,3,4,5};

pthread_cond_t condiciones[6];
/*
Cada condición expresa la finalización de un hilo
condiciones[0] fin del hilo main
condiciones[1] fin del hilo t1
condiciones[2] fin del hilo t2
condiciones[3] fin del hilo t3
condiciones[4] fin del hilo t4
condiciones[5] fin del hilo t5
*/

pthread_cond_t condicion; // han entrado los cinco threads

int M=0; // Numero de threads que han entrado

void *hilo(void *ptr) {
  int x;
  pthread_mutex_lock(&mutex);

  // Enviar señal al thread main cuando los 5 threads estén en el semáforo
  if ((++M)>=5) pthread_cond_signal(&condicion);

  // Comprobamos si ha finalizado su hilo precedente
  pthread_cond_wait(&condiciones[*(int *)ptr-1], &mutex);
  printf("COMIENZO TAREA HILO %d\n", *(int *)ptr);
  srand((int)pthread_self());
  x=1+(int)(N*rand()/RAND_MAX+1.0); // X es un número aleatorio entre 1 y N
  sleep(x);
  printf("FIN TAREA HILO %d: Tiempo ejecucion %d\n", *(int *)ptr, x);

  // enviamos señal de finalización del hilo
  pthread_cond_signal(&condiciones[*(int *)ptr]);
  pthread_mutex_unlock(&mutex);
}

int main() {
  pthread_t t1, t2, t3, t4, t5;
  pthread_attr_t tattr;
  pthread_mutex_init(&mutex, NULL);
  int x;

  // inicializamos las condiciones de espera de los hilos
  for(x=0; x<=5; x++)
    if (pthread_cond_init(&condiciones[x], NULL)!=0)
```

```
        exit(-1);

    // inicializamos la condicion de espera del hilo principal
    if (pthread_cond_init(&condicion, NULL)!=0) exit(-1);
    pthread_mutex_lock(&mutex);
    pthread_create(&t1, NULL, hilo, (void *)&i[0]);
    pthread_create(&t2, NULL, hilo, (void *)&i[1]);
    pthread_create(&t3, NULL, hilo, (void *)&i[2]);
    pthread_create(&t4, NULL, hilo, (void *)&i[3]);
    pthread_create(&t5, NULL, hilo, (void *)&i[4]);

    // Esperamos a que los 5 threads queden detenidos en el semáforo
    pthread_cond_wait(&condicion, &mutex);
    printf("Van a comenzar los hilos\n");

    //Enviamos la señal de finalización del hilo principal
    pthread_cond_signal(&condiciones[0]);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
    return 0;
}
```

4. Ejercicio: fila única

El objetivo de la práctica es realizar un programa que simule el sistema de fila única para el cobro a clientes en un pequeño supermercado. La práctica tiene una parte básica donde se define el funcionamiento básico del sistema de fila única y que es obligatorio implementar (supondrá como mucho el 80 % de la nota) y, además, se proponen una serie de mejoras opcionales para aumentar la nota (como mucho supondrá el 20 % de la nota).

4.1. Parte Básica (80 % de la nota)

El sistema de fila única para el cobro a clientes del supermercado funciona de la siguiente forma:

- El supermercado cuenta con 3 cajeros para atender a sus clientes. 3 clientes pueden ser atendidos a la vez, el resto esperará formando una fila única.
- Cuando los clientes terminan sus compras se colocan en la fila única. A cada cliente se le asignará un identificador único y secuencial (cliente_1, cliente_2, ..., cliente_N). El número máximo de clientes que pueden ponerse a la cola es de 20.
- Los clientes pueden cansarse de esperar y abandonar la cola dejando sus compras.
- El supermercado cuenta con 1 reponedor al que los cajeros pueden llamar para consultar un precio.

Cada cliente será atendido por un cajero que:

- Tiene un identificador único (cajero_1, cajero_2 y cajero_3).
- Solamente cuando haya terminado con un cliente pasará a atender al siguiente.
- Cada vez que un cajero atiende a 10 clientes se toma un descanso (duerme 20 segundos), y no atiende al siguiente hasta que vuelve.
- De los clientes atendidos el 70 % no tiene problemas. El 25 % tiene algún problema con el precio de alguno de los productos que ha comprado y es necesario que el reponedor vaya a comprobarlo, si está ocupado cliente y cajero deberán esperar. El último 5 % no puede realizar la compra por algún motivo (no tiene dinero, no funciona su tarjeta, etc.).

Toda la actividad quedará registrada en un fichero plano de texto llamado `registroCaja.log`:

- Cada vez que se atiende a un cliente se registrará en el log las horas de inicio y final de la atención al mismo.
- Se registrará el precio total de la compra.
- Se registrará si hay algún problema con algún precio o si el cliente no finalizó la compra.
- Se registrará la entrada y salida del descanso por parte del cajero.
- Al finalizar el programa se debe registrar el número de clientes atendidos por cada cajero.

Consideraciones prácticas:

- Simularemos la llegada de clientes a la fila única mediante la señal SIGUSR1. Se enviarán desde fuera del programa mediante el comando *kill* (`kill -SIGUSR1 PID`).
  - Los clientes pueden irse sin ser atendidos tras 10 segundos de espera. Se simulará esta posibilidad haciendo que el 10 % lo hagan.
- Simularemos el tiempo de atención de cada cliente con un valor aleatorio entre 1 y 5 segundos.
- El precio total de cada compra será un número aleatorio entre 1 y 100.
- Todas las entradas que se hagan en el log deben tener el siguiente formato: `[YYYY-MM-DD HH:MI:SS] identificador: mensaje.`, donde `identificador` puede ser el identificador del cliente, el identificador del cajero, el identificador el reponedor *o el identificador del departamento de atención al cliente* y `mensaje` es una breve descripción del evento ocurrido.
- Las entradas del log deben quedar escritas en orden cronológico.
- Hay que controlar el acceso a los recursos compartidos (al menos la cola de clientes y al fichero de log) utilizando semáforos (*mutex*).
- Para comunicar a los cajeros con el reponedor, utilizaremos una variable de condición. Los cajeros señalizarán la condición cuando necesiten que el reponedor consulte un precio. El reponedor estará a la espera de que esto ocurra.
- Simularemos el tiempo de atención del reponedor durmiendo un valor aleatorio entre 1 y 5 segundos.

4.2. Partes opcionales (20 % de la nota)

- Asignación estática de recursos (10 %):
  - Modifica el programa para que el número de clientes que pueda acceder al la fila única sea un parámetro que reciba el programa al ser ejecutado desde la línea de comandos.
  - Modifica el programa para que el número de cajeros que atienden sea un parámetro que reciba el programa al ser ejecutado desde la línea de comandos.
- Asignación dinámica de recursos I (5 %):
  - Modifica el programa para que el número de clientes que puedan acceder a la fila única pueda modificarse en tiempo de ejecución.
  - Solamente es necesario contemplar un incremento en el número de clientes. No es necesario contemplar la reducción.
  - Cada vez que se cambie el número de clientes tiene que reflejarse en el log.
- Asignación dinámica de recursos II (5 %):
  - Modifica el programa para que el número de cajeros que atiende se pueda modificar en tiempo de ejecución.
  - Solamente es necesario contemplar un incremento en el número de cajeros. No es necesario contemplar la reducción.
  - Cada vez que se produce un cambio en este sentido debe quedar reflejado en el log.

4.3. Diseño

En los siguientes apartados se detalla un posible diseño para la práctica. las zonas coloreadas en rojo, azul y verde son zonas de exclusión mutua que deben ser controladas. En las zonas verdes se deben usar variables condición.

4.3.1. Variables globales

Para implementar el diseño propuesto, el programa debería contar, al menos, con las siguientes variables globales:

- Ruta al archivo de log.
- Archivo de log (FILE *logFile).
- Lista de 20 clientes:
  - id
  - estado (0=esperado, 1=en cajero, 2=atendido)

- Semáforo para controlar el acceso al archivo de log.
- Semáforo para controlar el acceso a la lista de clientes.
- Variable de condición para interactuar con el reponedor.
- Semáforo asociado a la variable anterior.

4.3.2. Función main

A continuación, se enumeran la secuencia de acciones que se deberían realizar en el hilo principal.

1. `sigaction` de SIGUSR1 para crear clientes cuando se reciba la señal.
2. Inicializar recursos (¡Ojo! Inicializar no es lo mismo que declarar):
   - Semáforos.
   - Variables de condición.
   - Creamos el fichero de log.
   - Lista de clientes.
3. Creamos 3 hilos cajero.
4. Creamos 1 hilo reponedor.
5. Esperamos la recepción de señales (`pause`).

4.3.3. Llegada de nuevos clientes

Cada vez que se reciba la señal SIGUSR1 deberían realizarse las siguientes acciones. Estas tienen que implementarse en la función manejadora de la señal.

1. Buscamos una posición vacía en la lista de clientes.
2. Si hay hueco en la lista de clientes.
   - Creamos un nuevo hilo cliente.
   - Rellenamos los datos del cliente (id, estado=0).
3. Si no lo hay ignoramos la llamada.

4.3.4. Cajeros

Cada hilo cajero debería realizar la siguiente secuencia de acciones en bucle:

1. Buscamos el cliente que más tiempo lleva en la cola (el de menor id).
2. Si hay un cliente que atender:
   a) Cambiamos su estado (1).
   b) Calculamos el tiempo de atención.
   c) Escribimos la hora de la atención de la compra.
   d) Esperamos el tiempo de atención.
   e) Generamos un aleatorio de 1 a 100:
      - Si está entre 71 y 95, avisamos al reponedor y esperamos a que vuelva.
      - Si está entre 96 y 100, el cliente tiene algún problema (no tiene dinero suficiente, no funciona su tarjeta, etc.) y no puede realizar la compra.
   f) Escribimos la hora de finalización de la compra, si está ha terminado correctamente –en ese caso registramos también el importe– o ha habido algún problema.
   g) Cambiamos el estado del cliente (2).
   h) Si ha atendido a 10 clientes, el cajero descansa 20 segundos.

4.3.5. Clientes

Cada hilo cliente debería realizar la siguiente secuencia de acciones:

1. Escribimos la hora de llegada del cliente.
2. Calculamos el tiempo de espera máximo del cliente (aleatorio).
3. Esperamos a que expire el tiempo de espera o a que un agente nos atienda.
4. Esperamos si un agente nos ha atendido, esperamos a que termine (comprobar estado).
5. Guardamos en el log la hora de finalización.
6. Borramos la información del cliente de la lista.

4.3.6. Reponedor

El hilo reponedor debería realizar la siguiente secuencia de acciones en bucle:

1. Esperamos a que algún cajero me avise.
2. Calculamos el tiempo de trabajo (aleatorio).
3. Esperamos el tiempo.
4. Avisamos de que ha terminado el reponedor.

4.4. Escritura de mensajes en log

Es recomendable utilizar una función parecida a esta para evitar repetir líneas de código. Recibe como parámetros dos cadenas de caracteres, una para el identificador y otra para el mensaje (la fecha la calcula la propia función):

```
void writeLogMessage(char *id, char *msg) {
  // Calculamos la hora actual
  time_t now = time(0);
  struct tm *tlocal = localtime(&now);
  char stnow[25];
  strftime(stnow, 25, "%d/%m/%y %H:%M:%S", tlocal);

  // Escribimos en el log
  logFile = fopen(logFileName, "a");
  fprintf(logFile, "[%s] %s: %s\n", stnow, id, msg);
  fclose(logFile);
}
```