Práctica 3

Procesos, hilos y sincronización

Objetivos

En esta práctica se emplearán los recursos que el sistema operativo proporciona para la multiprogramación; en concreto, crearemos hilos que se sincronizarán a través de semáforos, cerrojos y variables condicionales.

A continuación se proporciona un resumen de los conceptos estudiados en clase, así como un catálogo de llamadas al sistema y funciones de librería que resultarán útiles para el desarrollo de la práctica. Para ampliar la información de cualquiera de ellas, consulta el manual del sistema Recuerda que debes realizar los ejercicios que se proponen a lo largo de las secciones del guión.

3.1. Procesos

Como ya sabemos, un proceso es una instancia de un programa en ejecución. En esta práctica vamos a conocer las llamadas al sistema más relevantes para la creación y gestión de procesos, si bien centraremos el desarrollo de la práctica en el uso de hilos.

Para crear un nuevo proceso, una réplica del proceso actual, usaremos la llamada al sistema:

```
#include <unistd.h>
pid_t   fork(void);
```

Como ya se ha estudiado, esta llamada crea un nuevo proceso (proceso *hijo*) que es un duplicado del padre[1]. Un ejemplo de uso sencillo es el siguiente:

```
int main ()
{
pid_t child_pid;

 child_pid = fork ();
 if (child_pid != 0) {
   // ESTE codigo lo ejecuta SOLO el padre
   .....
 else {
   // Este codigo lo  ejecuta SOLO el hijo
   ....
}
 // En un principio, este codigo lo ejecutan AMBOS PROCESOS
 // (salvo que alguno haya hecho un return, exit,execxx...)
}
```

Sin embargo, lo más habitual es que el nuevo proceso quiera cambiar completamente su mapa de memoria ejecutando una nueva aplicación (es decir, cargando un nuevo código en memoria desde un fichero ejecutable). Para ello, se hace uso de la familia de llamadas *execxx*:

```
#include <unistd.h>

int execl(const char *path, const char *arg, ... );
int execlp(const char *file, const char *arg, ... );
int execle(const char *path, const char *arg, ... );
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvP(const char *file, const char *search_path, char *const argv[]);
```

Para finalizar un proceso, su código debe terminar su función *main* (ejecutando un *return*) o puede invocar la siguiente función:

```
#include <stdlib.h>
void exit(int status);
```

Por último, existen llamadas para que un padre espere a que finalice la ejecución de un hijo:

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

De ese modo, una estructura habitual en el uso de estas funciones, podría ser la siguiente:

```
int main ()
{
pid_t child_pid;
int stat;

 while (....) {
   child_pid = fork ();
   if (child_pid != 0) {
          .....
   else {
               execv(...);
               // Por si exec falla...
               exit(-1);
```

[1]Duplicado implica que todas las regiones de memoria se COPIAN, por lo que padre e hijo no comparten memoria por defecto

```
        }
        // Padre esperar a finalizacion de hijos
        if (wait(&stat) == -1) {
                ....
        }
        if (WIFEXITED(stat)) {
                ....
        }
 }
}
```

Ejercicio 1: Estudia el código del fichero *ejemploFork.c* y responde a las siguientes preguntas:

- ¿Cuántos procesos se crean? Dibuja el árbol de procesos generado
- ¿Cuántos procesos hay como máximo simultáneamente activos?
- Durante la ejecución del código, ¿es posible que algún proceso quede en estado *zombi*? Intenta visualizar esa situación usando la herramienta *top* e introduciendo llamadas a *sleep()* en el código donde consideres oportuno.
- ¿Cómo cambia el comportamiento si la variable *p_heap* no se emplaza en el *heap* mediante una llamada a *malloc()* sino que se declara como una variable global de tipo *int*?
- ¿Cómo cambia el comportamiento si la llamada a *open* la realiza cada proceso en lugar de una sola vez el proceso original?
- En el código original,¿es posible que alguno de los procesos creados acabe siendo hijo del proceso *init* (PID=1)? Intenta visualizar esa situación mediante *top*, modificando el código proporcionado si es preciso.

Por último, antes de comenzar con las llamadas relativas a hilos, vamos a recordar qué zonas de memoria se comparte tras un *fork()* y cuáles entre hilos:

Zona de memoria	*Procesos padre-hijo*	*Hilos de un mismo proceso*
Variables globales (.bss, .data)	NO	Sí
Variables locales (pila)	NO	NO
Memoria dinámica (heap)	NO	Sí
Tabla de descriptores de ficheros	Cada proceso la suya (se duplica)	Compartida

3.2. Hilos (*threads*). Biblioteca de hilos Posix, *pthread*

Dentro de un proceso GNU/Linux pueden definirse varios hilos de ejecución con ayuda de la biblioteca `libpthread`, que permite usar un conjunto de funciones que siguen el estándar POSIX.

Para usar funciones de hilos POSIX en un programa en C debemos incluir al comienzo del código las sentencias:

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

y compilarlo con:

```
gcc ... -pthread
```

`_REENTRANT` indica que la versión reentrante de las librerías estándar es la que se debe de usar en el proceso de enlazado. Un código es reentrante si es seguro llamarlo concurrentemente desde varios hilos. Es importante que todas las funciones invocadas desde un hilo sean reentrantes, si no, el programa puede (o puede no hacerlo) mostrar un comportamiento inapropiado.

Todo proceso contiene un hilo inicial (`main`) cuando comienza a ejecutarse. A continuación pueden crearse nuevos hilos con:

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *arg);
```

Si se usan los atributos de hilo por defecto basta usar `NULL` como segundo argumento de `pthread_create()`, pero si se quieren otras especificaciones hay que declarar un objeto atributo, establecer en él las especificaciones deseadas y crear el hilo con tal atributo. Para ello se usa:

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_setXXX(pthread_attr_t *attr, int XXX);
```

donde `XXX` designa la especificación que se desea dar al atributo (principalmente `scope`, `detachstate`, `schedpolicy` e `inheritsched`).

Se pueden conocer los atributos de un *thread* con:

```
int pthread_attr_getXXX(const pthread_attr_t *attr, int XXX);
```

Un hilo puede terminar por distintas circunstancias:

- Su `func()` asociada se completa.
- El proceso que lo incluye termina.
- Algún hilo de su mismo proceso llama a `exec()`.
- Explícitamente llama a la función:

```
void pthread_exit(void *status);
```

Una vez creado un hilo, se puede esperar explícitamente su terminación desde otro hilo con:

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

Un hilo puede puede saber cuál es su `tid` con:

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

Recuérdese que todos los hilos de un mismo proceso comparten el mismo espacio de direcciones (por tanto, comparten variables globales y heap, pero NO variables locales, que residen en la pila) y recursos (p.e. ficheros abiertos, sea cual sea el hilo que lo abrió).

Ejemplo 1. Suma: El código `suma_parcial1.c` lanza dos hilos encargados de colaborar en el cálculo del siguiente sumatorio:

$$suma\_total = \sum_{n=1}^{10000} n$$

Después de ejecutarlo varias veces observamos que no siempre ofrece el resultado correcto, ¿Por qué? En una máquina que posea varios núcleos resultar más sencillo observar este efecto, debido a que el kernel Linux implementa threads de tipo NTPL (Native POSIX Threads Library). En esta implementación, el planificador puede asignar cada hilo a un núcleo y ejecutar así el código en paralelo (hilos tipo kernel 1:1). Consultar `man 7 pthreads` para más información. En caso de no ser así, utiliza el ejemplo codificado en `suma_parcial2.c` y observa que nunca obtenemos el resultado correcto. ¿Por qué?

3.3. Mecanismos de sincronización

En esta sección haremos un pequeño repaso a los mecanismos de sincronización que usaremos en el desarrollo de la práctica. Si bien sólo usaremos dichos mecanismos para sincronizar hilos, la sección 3.3.1 presenta los semáforos POSIX que pueden ser usados tanto para sincronizar hilos como para sincronizar procesos[2].

3.3.1. Semáforos POSIX

GNU/Linux dispone de una implementación para semáforos generales que satisface el estándar POSIX y que es del tipo semáforo "sin nombre" o semáforo "anónimo", de aplicación a la coordinación exclusivamente entre hilos de un mismo proceso, por lo que típicamente son creados por el hilo inicial del proceso y utilizados por los restantes hilos de la aplicación de forma compartida[3].

Las llamadas aplicables son:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```

- `sem_init()`: crea un semáforo POSIX y le da un valor inicial positivo; devuelve, en su primer argumento, un identificador para el semáforo creado. Para semáforos compartidos entre hilos del mismo proceso, `pshared` debe valer 0.

[2] En ese caso, es más útil utilizar la variante *con nombre* y no los semáforos *sin nombre* aquí estudiados

[3] El estándar POSIX especifica posibilidades más extensas de estos semáforos que permitiría emplear semáforos "con nombre" y permitir que el semáforo sea aplicable a hilos pertenecientes a procesos diferentes; pero nuestra versión está limitada en los términos comentados, consultar `man sem_overview` para más información.

- `sem_wait()`: realiza la operación atómica "`P`" (*wait*) sobre `semáforo`, disminuyendo el valor del semáforo en uno, si es posible sin que pase a ser negativo, y bloqueando el hilo invocante en caso contrario.
- `sem_trywait()`: intenta realizar una operación "`P`" (*wait*) sobre el semáforo; pero si el hilo invocante fuese a quedar bloqueado, regresa indicando tal situación y permite al hilo continuar su ejecución.
- `sem_post()`: realiza la operación atómica "`V`" (*signal*) sobre semáforo, despertando a un hilo bloqueado en el semáforo designado si lo hay (con disciplina FIFO) o incrementando el valor del semáforo en uno en caso contrario.
- `sem_getvalue()`: devuelve en su segundo argumento el valor actual del semáforo (sólo es indicativo; el valor puede haber sido alterado por operaciones concurrentes).
- `sem_destroy()`: destruye un semáforo POSIX previamente creado con la llamada `sem_init()`.

Para más información sobre estas llamadas puede consultarse la sección 3 del manual.

Ejercicio 2: Incluir un semáforo sin nombre al **Ejemplo 1 (Suma)** de forma que siempre dé el resultado correcto. Incluir además la opción de especificar, mediante línea de comando, el valor final del sumatorio y el número de hilos que deberán repartirse la tarea. Ejemplo:

```
./Suma1 5 50000
```

sumará los número entre 1 y 50000 empleando para ello 5 hilos.

3.3.2. Cerrojos para hilos

Los mutexes son semáforos binarios, con caracterización de propietario (es decir, un cerrojo sólo puede ser liberado por el hilo que lo tiene en ese momento), empleados para obtener acceso exclusivo a recursos compartidos y para asegurar la exclusión mutua de secciones críticas. La implementación usada en GNU/Linux es la incluída en la biblioteca de hilos `pthread` y sólo es aplicable a la coordinación de hilos dentro de un mismo proceso.

Las funciones aplicables son:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `pthread_mutex_init()`: crea un *mutex* con el atributo especificado que permite elegir entre tres comportamientos distintos cuando un mismo hilo intenta apropiarse más de una vez de un mismo *mutex*:
  - "rápido": el intento de tomar posesión de nuevo de un *mutex* ya ocupado produce bloqueo sea quien sea el hilo que lo intente.
  - "recursivo": el intento de tomar posesión de nuevo de un *mutex* ya ocupado tiene éxito si quien lo pide es el hilo-propietario actual. Se toma nota del número de veces que esto ocurre y se exige que el hilo devuelva la posesión del *mutex* el mismo número de veces.

- "chequeo de error": en el mismo caso anterior, la llamada regresa indicando situación de error (`EDEADLK`).

La inicialización también se puede hacer de forma declarativa, del siguiente modo:

```
pthread_mutex_t fastmutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex=PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_terrchkmutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

En cualquier caso, evidentemente, el *mutex* recién inicializado está libre.

- `pthread_mutex_lock()`: se corresponde con la operación "`P`" (*wait*) de semáforo; si el *mutex* está ocupado por otro hilo, el hilo invocante se bloquea. Si el hilo invocante es el hilo-propietario, el comportamiento depende del tipo de *mutex* ("rápido", "recursivo" o "chequeo de error")
- `pthread_mutex_unlock()`: se corresponde con la operación "`V`" (*signal*) del semáforo, con la salvedad de que esta operación sólo tiene éxito si quien la realiza es el hilo propietario actual del *mutex*.
- `pthread_mutex_destroy()`: destruye un *mutex* previamente creado.

3.3.3. Variables de condición

Una variable de condición es una variable de sincronización asociada a un *mutex* que se utiliza para bloquear un hilo hasta que ocurra alguna circunstancia representada por una expresión condicional. En la implementación contenida en la biblioteca `pthread` las variables de condición tiene el comportamiento propugnado por Lampson-Reddell, según el cual el hilo señalizador tiene preferencia de ejecución sobre el hilo señalizado, por lo cual éste último debe volver a comprobar la condición de bloqueo una vez despertado.

Las funciones aplicables son (extracto de `man pthread.h`):

```
NAME
       pthread.h - threads
SYNOPSIS
       #include <pthread.h>
DESCRIPTION
   int pthread_cond_init(pthread_cond_t *cond,
                         pthread_condattr_t *cond_attr);
   int pthread_cond_signal(pthread_cond_t *cond);
   int pthread_cond_broadcast(pthread_cond_t *cond);
   int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
   int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
                              *mutex, const struct timespec *abstime);
   int pthread_cond_destroy(pthread_cond_t *cond);
```

- `pthread_cond_init()`: Inicia una variable de tipo `pthread_cond_t` con los atributos especificados (dependiendo de la versión de librería `pthread` utilizada no se aceptan atributos, por lo que este argumento lo pondremos a `NULL`) . También se puede indicar mediante una declaración del modo siguiente:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- `pthread_cond_wait()`: corresponde a la operación "*waitc*" de una *condition* de un monitor; por tanto, siempre suspende al hilo invocante y libera "el monitor", es decir, el *mutex* referenciado como argumento. El código empleado suele ser:

```
/* Código del hilo_1 */
pthread_mutex_lock(&mutexA);
while (/* expresión condicional (E.C.) es falsa */) {
   pthread_cond_wait(&condA, &mutexA); /* Se bloquea el hilo_1 */
}
/* acciones deseadas si E.C. es cierta */
pthread_mutex_unlock(&mutexA);
```

- `pthread_cond_timedwait()`: variante de `pthread_cond_wait()` que limita el plazo temporal de espera. Si la función regresa con estado `0` (éxito) se deberá a que ha recibido una señal dentro de plazo; si regresa con estado de error `ETIMEDOUT` se deberá a que ha pasado el plazo de tiempo y no se ha recibido ninguna señal sobre la condición. La estructura `struct timespec` con la que se indica el plazo de tiempo tiene dos componentes: `tv_sec` y `tv_nsec` (segundos y nanosegundos, respectivamente).
- `pthread_cond_signal()`: corresponde con la operación "*signalc*" de una *condition* de un monitor; por tanto, desbloquea al hilo más antiguo suspendido en la variable de condición referida. Si no hay ningún hilo bloqueado, es equivalente a una "no operación". El código empleado suele ser:

```
/* Código del hilo_2 */
pthread_mutex_lock(&mutexA);
/* operaciones protegidas hasta hacer que
   la E.C. esperada sea cierta */
pthread_cond_signal(&condA);          /* Despierta al hilo_1 */
/* más operaciones protegidas */
pthread_mutex_unlock(&mutexA);        /* El hilo 1 puede proseguir */
```

- `pthread_cond_broadcast()`: desbloquea a todos los hilos suspendidos en una variable de condición.

3.4. Señales

Las señales son un mecanismo de notificación de eventos a procesos o hilos. En cierto modo, son la contrapartida software de las interrupciones hardware, ya que una señal tiene un carácter asíncrono (puede recibirse en cualquier momento) y su recepción supone la ejecución inmediata de un función de tratamiento de señal.

Del mismo modo que ocurre con las interrupciones, debemos especificar qué acción hay que realizar cuando se recibe una determinada señal. Asimismo, es posible enmascarar un conjunto de señales para ignorar su recepción.

Para especificar qué acción debe ejecutarse al recibir una determinada señal, debe utilizarse la llamada al sistema:

```
int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

Para establecer el conjunto de señales enmascaradas se dispone de las siguientes funciones:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Una vez establecido el conjunto de señales que se desean enmascarar (dicho conjunto quedará almacenado en una variabe de tipo *sigset_t*), se debe proceder a fijar dicha máscara para el proceso mediante la llamada al sistema:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Como se ha dicho anteriormente, generalmente las señales se envían a un proceso para notificar que ha ocurrido algún evento. Por ejemplo, SIGKILL o SIGTERM indican al proceso que debe terminar (SIGKILL no puede enmascararse ni tratarse, y SIGTERM sí). SIGINT notifica al proceso de una interrupción producida por el usuario desde el teclado (habitualmente por pulsar la combinación de teclas *Ctrl+C*). Pero también es posible enviar de forma explícita una señal a un proceso mediante la llamada al sistema:

```
int kill(pid_t pid, int sig);
```

Es muy importante recalcar que, dado el carácter asíncrono de las señales, no es *prudente* realizar determinadas acciones en la función de tratamiento de una señal. En la página de *signal* de la sección 7 del manual, se puede consultar una lista de funciones que sí pueden ser invocadas desde una función de tratamiento de señal.

En concreto, modificar variables en memoria compartida, realizar llamadas potencialmente bloqueantes (como *pthread_mutex_lock()*, *sem_wait()*...) está desaconsejado pues el comportamiento es indefinido y extremadamente complejo de depurar si no resulta correcto.

Es posible eliminar el carácter *asíncrono* de las señales si el proceso/hilo está dispuesto a quedarse bloqueado hasta que llegue una señal. De ese modo, no es posible que la llegada de una señal encuentre al proceso en mitad de una operación *delicada* que no deba ser interrumpida (y, en el peor de los casos, que la función de tratamiento de señal haga una nueva llamada a esa misma función, dejando el proceso en un estado indeterminado). Esto tiene sentido en caso de programación multi-hilo, como se comentará más adelante. La función que permite el bloqueo de un proceso/hilo hasta la llegada de una señal es:

```
int sigwait(const sigset_t *set, int *sig);
```

3.4.1. Hilos y señales

En Linux, la implementación de hilos no sigue completamente el estándar POSIX. En concreto, en lo referente a la gestión de señales, cuando se envía una señal a un proceso no es posible determinar qué hilo de dicho proceso gestionará la señal[4].

Sin embargo, se han incorporado algunas llamadas de tratamiento de señales específicas para hilos. En concreto, para definir las señales enmascaradas únicamente para el hilo[5]:

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

Para evitar el problema del indeterminismo del hilo que atenderá cada señal, es una buena práctica de programación dedicar un hilo (de un programa multi-hilo) a la gestión de señales. El hilo principal enmascarará todas las señales, y únicamente el hilo gestor desbloqueará las señales deseadas.

[4]Sí existe la garantía de que la señal se recibirá una sola vez; en ningún caso se recibirá la misma señal replicada para cada hilo

[5]Por defecto, el hilo hereda de su creador el conjunto de señales enmascaradas

3.5. Desarrollo de la práctica

El objetivo de esta práctica es implementar un sistema cliente/servidor. El hilo del servidor estará esperando la llegada de un mensaje a un buzón (cola de mensajes). Cuando lo reciba, realizará el servicio asociado al mensaje recibido y notificará al cliente correspondiente.

Los hilos cliente se comunican con **el hilo servidor** a través de un buzón (cola de mensajes). En el sistema podrán coexistir múltiples clientes simultáneamente. Cada cliente enviará un número determinado de mensajes de un mismo tipo –TIME ó RANDOM– al buzón (de uno en uno) y, después de enviar cada mensaje, quedará a la espera de la notificación por parte del servidor de la finalización del servicio demandado. La respuesta proporcionada por el servidor se almacenará en el campo `content` de la estructura del mensaje.

El hilo servidor estará esperando la llegada de un mensaje al buzón. Cuando lo reciba, realizará el servicio asociado al mensaje recibido y notificará al cliente correspondiente. A continuación seguirá procesando más mensajes del buzón.

Como se indicó previamente, la comunicación y sincronización entre el servidor y los clientes se realizará a través de un mecanismo de paso de mensajes que construiremos nosotros mismos. Para ello, en primer lugar se creará un tipo *variable condicional* (*mi_cond_var_t*) a partir de cerrojos y semáforos POSIX. Posteriormente, se implementará el tipo *mailbox* y la funcionalidad asociada haciendo uso de cerrojos POSIX y las variables condicional recién creadas.

El servidor implementa dos servicios:

- Generación de números aleatorios. Si el servidor recibe un mensaje de tipo *RANDOM*, generará un número aleatorio y se lo comunicará al hilo cliente que solicitó ese servicio.
- Solicitud de hora actual. Si el servidor recibe un mensaje de tipo *TIME*, el servidor consultará la hora actual y se la comunicará al hilo cliente que solicitó el servicio.

Todo el código del servidor está ya implementado en el fichero `server.c`

Asimismo, habrá dos tipos de clientes: clientes que solicitan servicios tipo *RANDOM* y los que solicitan servicios tipo *TIME*. En la función `main()` hay ejemplos de cómo crear clientes de cada tipo. El fichero `client.c` contiene prácticamente toda la implementación de los clientes, salvo la creación del hilo en sí, que se solicita como parte de esta práctica.

El hilo principal es el que ejecuta la función `main()`, y por tanto es creado automáticamente por el sistema operativo al crear el proceso. Este hilo es el encargado de (1) inicializar los recursos globales del programa y de sincronización entre los hilos, (2) crear el hilo servidor y el gestor de señales y (3) procesar y ejecutar los comandos que el usuario introduce por la entrada estándar. La tabla 3.1 describe los distintos comandos que soporta el hilo principal.

El gestor de señales es el encargado de recibir de todas la señales entrantes y realizar las acciones asociadas a cada señal. Más concretamente, el sistema debe capturar las señales SIGINT. La señal SIGINT se genera al teclear `Ctrl+C` en el terminal asociado al proceso; al recibir esta señal, el gestor de señales imprimirá un mensaje por pantalla. Esta funcionalidad ya está implementada en el código entregado.

El programa multihilo se invocará del siguiente modo:

```
$ ./practica3 [opciones]
```

El programa soporta las siguientes opciones:

- `-o <logfile>` Permite redirigir los mensajes que clientes y servidor imprimen por pantalla a un fichero especificado por `<logfile>` . Es recomendable utilizar esta opción para

Comando	Descripción
`client <tipo> <num_msgs> <num_segs>`	Crea un hilo cliente del tipo especificado que envía `<num_msgs>` de uno en uno al servidor con esperas de `<num_segs>` segundos entre cada envío. Los tipos de cliente soportados son 0 (RANDOM) ó 1 (TIME).
`exit`	Al recibir este comando el hilo principal deja de procesar más comandos, espera a que finalicen los clientes en curso, fuerza la finalización del servidor y termina la ejecución del programa.
`quit`	Este comando es equivalente a `exit`
`pid`	Imprime por pantalla el PID del proceso
`<parametro>=<valor entero>`	Este tipo de comando permite modificar los parámetros por defecto que se usarán al crear hilos cliente cuando el proceso reciba la señal SIGUSR1 (si se realiza la parte opcional). Actualmente `<parametro>` puede ser uno de los siguientes valores: ▪ `default_client_type` : tipo de cliente ▪ `nr_msgs`: número de mensajes ▪ `nr_segs`: número de segundos entre cada envío de mensaje.
`show_params`	Imprime por pantalla el valor de los parámetros de creación de clientes por defecto.
`! orden`	Crea un shell BASH para que ejecute `orden`, que ha de ser cualquier comando válido soportado por BASH. Este comando puede emplearse para enviar señales al proceso desde el mismo terminal desde donde se está ejecutando el programa. Por ejemplo, el comando `! kill -USR1 $PPID`, enviará la señal SIGUSR1 al propio proceso.

Cuadro 3.1: Comandos soportados por el sistema

poder interactuar más cómodamente con el hilo principal. Los mensajes de cliente y servidor pueden redirigirse a otra ventana de terminal abierta por el usuario. Para conocer el nombre de dispositivo asociado a un terminal debemos teclear el comando `tty` en el mismo. Este nombre de dispositivo (p.ej.: /dev/pts/6) puede pasarse como parámetro de la opción -o.

- `-n <mailbox_size>` Establece el tamaño máximo del buzón: número máximo de mensajes que pueden almacenarse en el buffer circular del buzón.
- `-s <default_nr_secs>` Establece el valor inicial del parámetro global `nr_segs`.
- `-m <default_nr_msgs>` Establece el valor inicial del parámetro global `nr_msgs`.
- `-c <default_client_type>` Establece el valor inicial del parámetro global `default_client_type`.
- `-h` Imprime por pantalla las opciones soportadas

El código completo de la práctica constará de los siguientes ficheros:

- `sem.c (.h)`. Implementación de nuestro propio tipo *semáforo general* mediante cerrojos (*locks*) y variables condicionales POSIX. Esto puede servir de *inspiración* para la implementación de nuestras propias variables condicionales. Para la realización de esta parte de la práctica, **NO es necesario modificar este fichero** (sí consultarlo).
- `varcond.c (.h)`. Implementación de nuestras propias variables condicionales a partir de cerrojos y semáforos POSIX. El tipo *mi_cond_var_t* está ya definido en el fichero `varcond.h` y no se debe modificar. Este **fichero sí deberá ser completado.**
- `message.c (.h)` Tipo de un mensaje y funciones para su manipulación. Para la realización de esta parte de la práctica, **NO es necesario modificar este fichero** (sí consultarlo)
- `cbuffer.c (.h)`: Tipo de datos `cbuffer_t` y operaciones asociadas que implementan un buffer circular acotado. Este **fichero sí deberá ser completado.**
- `mailbox.c (.h)` Tipo del buzón y funciones para su uso. La sincronización, del tipo productor/consumidor, se deberá hacer usando cerrojos POSIX y nuestra propia implementación de variables condicionales. La comunicación (el *buffer* en el que se almacenan los mensajes) se realizará por memoria compartida. Este **fichero sí deberá ser completado.**
- `server.c (.h)` Código del servidor que espera peticiones en el buzón y realiza el servicio asociado al mensaje. Para la realización de esta parte de la práctica, **NO es necesario modificar este fichero** (sí consultarlo)
- `clients.c (.h)` Código de los diferentes tipos de clientes que piden servicios a través del buzón. Este **fichero sí deberá ser completado.**
- `signal_handler.c (.h)` Código del hilo que se encarga de la gestión de señales. Para la realización de esta parte de la práctica, **NO es necesario modificar este fichero** (para el apartado opcional sí).
- `main.c` Código de la función principal que crea al servidor, clientes.... Para la realización de esta parte de la práctica, **NO es necesario modificar este fichero** (sí consultarlo).

El código proporcionado responde a la estructura anterior, pero está incompleto. Se deberán completar los tres ficheros referidos tal y como se indica en la siguiente sección.

3.5.1. Realización de la práctica

El objetivo es hacer funcionar el sistema, completando las funciones que no se entregan desarrolladas:

1. Completar la implementación del tipo *cbuffer_t*. En concreto, se deberán implementar las funciones:

   ```
   void remove_cbuffer_t ( cbuffer_t* cbuffer);
   void* head_cbuffer_t ( cbuffer_t* cbuffer );
   ```

   La función *remove* elimina el primer elemento del buffer circular y actualiza convenientemente los campos `head` y `size` de la estructura.

   La función *head* devuelve el primer elemento del buffer, pero no modifica el buffer en sí.

2. Completar la implementación del tipo *mi_cond_var_t*. El fichero `varcond.h` contiene la definición del tipo y no es necesario modificarlo. El fichero `varcond.c` contiene parte de la implementación, pero falta por desarrollar la función `var_cond_wait()`, es decir, la espera sobre una variable condicional. Debe seguir el comportamiento estudiado en clase, que se recuerda a continuación junto con el prototipo de la función:

   ```
   int var_cond_wait(struct  mi_cond_var *vc, pthread_mutex_t* m) {

    // 1. Incrementar el contador de hilos bloqueados en vc.
    //  (hay un campo para ese contador en el tipo de vc)

    // 2. Liberar el cerrojo m

    // 3. Bloquearse (usando el semáforo de vc)

    // 4. Volver a obtener el cerrojo m

    // 5. Decrementar el contador de hilos
   }
   ```

3. Implementar las funciones *fetch* y *post* del buzón (*mailbox*). Para la sincronización necesaria, se deberán emplear cerrojos POSIX y las variables condicionales recién creadas. Concretamente hay que completar las siguientes funciones del fichero *mailbox.c*:

   ```
   void  mbox_post( struct sys_mbox *mbox, void *msg);
   int mbox_fetch(struct sys_mbox *mbox, void **msg);
   ```

   El comportamiento de dichas funciones, que siguen el patrón productor/consumidor estudiado en clase, debe ser el siguiente:

   - *post* debe introducir un nuevo mensaje en el buzón. El hilo que la invoque quedará bloqueado en caso de que no haya espacio en el buzón para un nuevo mensaje, y se desbloqueará en cuanto haya espacio para el mensaje (es decir, es el *productor*)
   - *fetch* debe extraer un mensaje del buzón. Si el buzón está vacío, el hilo quedará bloqueado hasta que llegue un nuevo mensaje (es decir, es el *consumidor*).

4. Implementar la función `create_client()` del fichero `clients.c`. Dicha función se invocará cada vez que se quiera crear un nuevo cliente (podéis consultar la función `main()` para ver ejemplos de uso). Deberá crear un nuevo hilo que realizará el trabajo de ese cliente. Dado que la llamada para crear nuevos hilos exige que la función de entrada del hilo reciba un único argumento de tipo *void**, deberemos utilizar una variable de tipo *struct*

client_arg para indicarle toda la información necesaria: tipo de mensaje, número de mensajes, período y puntero al *mailbox*. El nuevo hilo de cliente deberá comenzar su ejecución en la función `client_thread(void*)`. Consultad su código para obtener más información sobre el paso de argumentos.

5. (Opcional). Añadir el tratamiento de la señal SIGUSR1 en el fichero `signal_handler.c` de modo que, cada vez que se reciba una señal SIGUSR1 se cree un nuevo cliente del tipo por defecto con un número de mensajes y período por defecto. En la función que tratará la señal, tras crear dicho hilo, debe llamarse a la función `my_resume()` para que el hilo gestor de señales vuelva a quedarse bloqueado en espera de nuevas señales.

Se recomienda hacer test unitarios para los dos primeros apartados, pues no será posible comprobar la funcionalidad completa del sistema hasta el final.