Práctica 3

Procesos e hilos: gestión y sincronización

Objetivos

En esta práctica se estudiarán diversos ejemplos en los que se hace uso de las llamadas al sistemas relacionadas con la gestión de procesos e hilos, y las primitivas de comunicación y sincronización.

A continuación enumeramos los conceptos estudiados en clase y las llamadas al sistema relevantes. Para ampliar la información de cualquiera de ellas, consulta el manual del sistema. Asimismo, proponemos ejercicios que ayudarán en la consecución de los objetivos de la práctica.

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) {
            .....
```

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

```
        } else {
            execv(...);
            // Por si exec falla...
            exit(-1);
        }
        // Padre espera a finalización de hijos
        if (wait(&stat) == -1) {
            ....
        }
        if (WIFEXITED(stat)) {
            ....
        }
    }
}
```

Ejercicio 1: Estudia el código del fichero *fork_example.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:

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

y compilarlo con:

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

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 saber cuál es su *tid* con:

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

Consultar *man 7 pthreads* para más información. 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 *partial_sum1.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 caso de no ser así, utiliza el ejemplo codificado en *partial_sum2.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 solo 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);
```

[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.

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

Ejercicio 2: Añadir un semáforo sin nombre al **Ejemplo 1 (*partial_sum1.c*)** 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:

```
./partial_sum1 5 50000
```

sumará los números 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);
```

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_t errchkmutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

Un *mutex* recién inicializado no pertenece a ningún hilo.

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 tienen 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*):

```
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);
```

También se puede indicar la inicialización mediante una declaración del modo siguiente:

```
pthread_cond_t cond = pthread_cond_initializer;
```

3.4. Entregable 3.1. Utilidad *my_system*

La función *system()* de la biblioteca estándar de C utiliza la llamada al sistema *fork()* y la función *execl()* para crear un nuevo proceso shell hijo que invoca el comando que se pasa como parámetro a la función:

```
int system(const char *command);
```

El proceso de usuario que invoca *system()* se queda bloqueado hasta que el comando que se pasa como argumento finalice. Como salida, *system()* devuelve el código de salida (*status*) del shell.

En este primer entregable de la práctica se debe implementar una nueva utilidad *my_system*, cuya funcionalidad sea semejante a la llamada *system()*. Como argumento de esta utilidad se pasará los comandos que se deseen ejecutar.

Nota: cuando se crea un shell se puede pasar como argumento una cadena de comandos con la opción *-c <comando>*. Por ejemplo, para ejecutar el comando *ls -l*, se puede invocar a *bash* con los argumentos:

```
/bin/bash -c 'ls␣-l'
```

3.5. Entregable 3.2. Problema de los filósofos comensales con *pthreads*

Implementa un código que modele el problema de los filósofos comensales creando 5 hilos que representarán a los 5 filósofos. Cada uno de ellos se limitará a *pensar* (que se simulará mediante a una llamada a *sleep()* con un tiempo aleatorio) y posteriormente *comer*. Antes de comer, deberán coger dos tenedores: el de su derecha y el de su izquierda (no necesariamente

en ese orden...). Posteriormente, dejará los tenedores, dormirá y volverá a pensar. El acto de coger un tenedor se modelará con el intento de adquirir un cerrojo.

3.6. Entregable 3.3. Simulador concurrente con *pthreads*

Se desea simular el funcionamiento de un autobús urbano que realiza una ruta circular con N_PARADAS paradas. En cada parada dicho autobús deberá de esperar a que las personas suban y bajen antes de continuar con la ruta.

El funcionamiento viene definido por el siguiente código:

```
#define N_PARADAS  5   // número de paradas de la ruta
#define EN_RUTA    0   // autobús en ruta
#define EN_PARADA  1   // autobús en la parada
#define MAX_USUARIOS 40  // capacidad del autobús
#define USUARIOS   4   // numero de usuarios
// estado inicial
int estado        = EN_RUTA;
int parada_actual = 0;     // parada en la que se encuentra el autobus
int n_ocupantes   = 0;     // ocupantes que tiene el autobús

// personas que desean subir en cada parada
int esperando_parada[N_PARADAS]; //= {0,0,...0};

// personas que desean bajar en cada parada
int esperando_bajar[N_PARADAS];  //= {0,0,...0};

// Otras definiciones globales (comunicación y sincronización)

void * thread_autobus(void * args) {
    while (/*condicion*/) {
        // esperar a que los viajeros suban y bajen
        Autobus_En_Parada();
        // conducir hasta siguiente parada
        Conducir_Hasta_Siguiente_Parada();
    }
}

void * thread_usuario(void * arg) {
    int id_usuario;
    // obtener el id del usario
    while (/*condicion*/) {
        a=rand() % N_PARADAS;
        do{
            b=rand() % N_PARADAS;
        } while(a==b);
        Usuario(id_usuario,a,b);
    }
}

void Usuario(int id_usuario, int origen, int destino) {
    // Esperar a que el autobus esté en parada origen para subir
    Subir_Autobus(id_usuario, origen);
    // Bajarme en estación destino
    Bajar_Autobus(id_usuario, destino);

}
```

```
int main(int argc, char *argv[]) {
    int i;
    // Definición de variables locales a main
    // Opcional: obtener de los argumentos del programa la capacidad del
    // autobus, el numero de usuarios y el numero de paradas

    // Crear el thread Autobus

    for (i = 0; ...){
        // Crear thread para el usuario i

        // Esperar terminación de los hilos
        return 0;
    }
}
```

Se puede asumir que el autobús está circulando continuamente, tiene capacidad para un máximo de *MAX_USUARIOS* viajeros y que este valor es mayor que el numero de *USUARIOS*). También se puede asumir que los usuarios cuando acaban su trayecto vuelven a iniciar un nuevo viaje. Opcionalmente se puede fijar via argumentos del simulador el número de usuarios, la capacidad del autobús y el número de paradas, y establecer condiciones de terminación para los hilos autobús y usuarios. La funcionalidad de las funciones auxiliares a implementar es la siguiente:

```
void Autobus_En_Parada(){
/*  Ajustar el estado y bloquear al autobús hasta que no haya pasajeros que
    quieran bajar y/o subir la parada actual. Después se pone en marcha */
}

void Conducir_Hasta_Siguiente_Parada(){
/* Establecer un Retardo que simule el trayecto y actualizar numero de parada */
}

void Subir_Autobus(int id_usuario, int origen){
/*  El usuario indicará que quiere subir en la parada 'origen', esperará a que
el autobús se pare en dicha parada y subirá. El id_usuario puede utilizarse para
proporcionar información de depuración */
}

void Bajar_Autobus(int id_usuario, int destino){
/*  El usuario indicará que quiere bajar en la parada 'destino', esperará a que
el autobús se pare en dicha parada y bajará. El id_usuario puede utilizarse para
proporcionar información de depuración */
}
```