Práctica 3

Procesos e hilos: planificación y sincronización

Objetivos

En esta práctica se implementarán distintos algoritmos de planificación en un simulador que refleja con bastante exactitud la estructura de un planificador real. En la construcción del planificador, se emplearán los recursos que el sistema operativo proporciona para la multiprogramación; en concreto, crearemos hilos que se sincronizarán a través de cerrojos y variables condicionales.

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 esperar a finalizacion 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 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 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);
```

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.

[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.

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_terrchkmutex=PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

Obviously, the freshly initialized *mutex* is not acquired by any thread initially.

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. Manual de uso del simulador

El objetivo de esta práctica es completar un simulador de un **puente estrecho de una dirección**. Dicho puente únicamente dispone de un único carril y por lo tanto solamente pueden circular vehículos en un solo sentido. Para este simulador se considera que la longitud del

puente es de tamaño `LENGTH=3` lo que significa que pueden están simultaneamente hasta tres vehículos en el puente y que se tarda en cruzarlo 3 ciclos de reloj. Dicho simulador emulará el comportamiento de un vehículo mediante la ejecución de un hilo o thread, permitiendo la ejecución de varios hilos simultáneamente que simularán el comportamiento de varios vehículos a la hora de cruzar el puente de acuerdo a las siguientes reglas:

- Si llega un coche y encuentra el puente ocupado por otros coches cruzando en dirección contraria, deberá de esperar a que el puente esté vacío.
- Si llega un coche y encuentra que hay menos de tres coches accediendo al puente en su mismo sentido, deberá de cruzar inmediatamente.
- Si llega un coche y encuentra que hay tres coches accediendo al puente en su mismo sentido, deberá de esperar a que el primero de ellos abandone el puente para cruzar.

3.4.1. Ejecución y generación de diagramas

Una vez compilado (usando el *Makefile* entregado), podemos proceder a ejecutar el simulador. Para ver todas las opciones disponibles haremos lo siguiente:

```
$ ./sim
```

que devolverá los parámetros necesarios para la ejecución, parecido a la siguiente salida:

```
./exec inputfile outfile
```

donde el fichero *inputfile* corresponde a un fichero donde describe el comportamiento de los vehículos:

- El identificador del vehículo, descrito con un número entero único.
- Tipo de vehículo descrito con un carácter (c: coche, v: furgoneta y m: monovolumen)
- El momento de llegada al puente, descrito con un número entero .
- La dirección del vehículo descrita con un 0: este/oeste o 1: oeste/este

La ejecución escribirá en un fichero las estadísticas correspondiente a la simulador (llegada y cruce de cada vehículo) de acuerdo a alguno de los ficheros de entrada del simulador que se encuentran en el directorio *examples/*. La simulación genera un fichero de log de salida `sim.log` que usaremos para generar un diagrama de la planificación. Para ello, usaremos la herramienta `generate_gantt_chart` en el directorio `gantt`:

```
$ ./sim examples/example1.txt example1.log
local_time(0)=0
local_time(2)=1
local_time(1)=2
local_time(5)=3
local_time(6)=4
local_time_extra(6)=7
local_time(4)=7
local_time(3)=8

$ cd ../gantt
$ ./generate_gantt_chart ../sim/example1.log
```

La figura 3.1 muestra el resultado para el ejemplo concreto `examples/example1.txt`. Vemos la progresión de cada uno de los vehículos. Las partes azules indican el cruce en sentido este/oeste, la parte amarilla es tiempo de cruce de oeste/este y las zonas grises representan el tiempo en que un vehículo está preparado pero no podía cruzar porque el puente estaba ocupado en ese momento.

3.4.2. Estructuras de datos relevantes

La información referida a cada vehículo se almacene en una estructura del tipo *tcar* cuyos campos están relacionados con:

- *id*: identificador del vehículo, descrito con un número entero único.
- *type*: tipo de vehículo descrito con un carácter (c: coche, v: furgoneta y m: monovolumen)
- *my_direction*: dirección del vehículo descrita con un 0: este/oeste o 1: oeste/este.
- *t_arrival*: momento de llegada al puente.
- *t_cross_in*: momento en el que comienza a cruzar el puente.
- *t_cross_out*: momento en el que acaba de a cruzar el puente.

```
typedef struct {
  int id;
  char type; // C=car, M=minivan, V=van
  int my_direction; // 0, 1
  int t_arrival;
  int t_cross_in;
  int t_cross_out;
} tcar;
```

La información relacionada con el puente así como los mecanismos de sincronización se encuentran descritos en la estructura del tipo *tbridge* cuyos campos corresponden a:

- *mtx*: cerrojo que permite acceder a los campos de la estructura de forma exclusiva.
- *VCs*: dos variables condicionales que implementan las colas donde los vehículos esperan cuando el puente está ocupado (una para la dirección este/oeste y otra para la oeste/este).
- *cars_on_bridge*: número de coches que están accediendo al puente, es decir cruzándolo.
- *cur_direction*: sentido de circulación actual del puente: este/oeste o viceversa.
- *cars_waiting*: número de coches esperando en ambos sentidos.

```
#define EMPTY -1

typedef struct {
        pthread_mutex_t mtx;
        pthread_cond_t VCs[2];
        int cars_on_bridge;
        int cur_direction;
        int cars_waiting[2];
} tbridge;
```

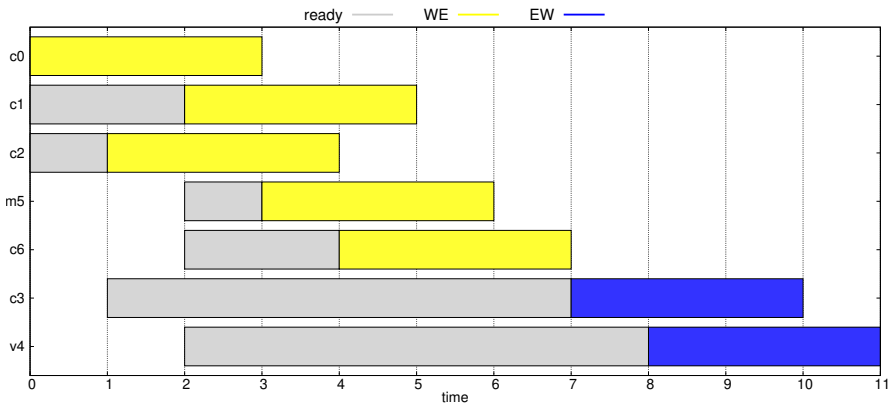

Figura 3.1: Resultado de simular el fichero example1.txt

3.5. Parte obligatoria

Como parte obligatoria de esta práctica se deberán realizar las siguientes modificaciones en el simulador:

- Desarrollar la funcionalidad del puente tal y como se ha descrito a lo largo de la práctica, rellenando las funciones *bridge_in(tcar *dcar)* y *bridge_out(tcar *dcar)*.
- Implementar una barrera de sincronización usando cerrojos y variables condicionales. Completar el fichero `barrier.c` (funciones `sys_barrier_init()`, `sys_barrier_destroy()` y `sys_barrier_wait()` de la rama `#else`). Para probar esta funcionalidad, hay que modificar el *Makefile* para evitar que se declare la macro `POSIX_BARRIER`.
- Escribir un *script* shell que no reciba ningún argumento, pero que pregunte al usuario por el fichero que desea emular. Se comprobará que el fichero existe y es un fichero regular. En caso contrario, se informará al usuario y se volverá a preguntar por el nombre. A continuación, se creará un directorio `resultados`. Los resultados se irán almacenando (sin sobre-escribirse) en el directorio `resultados`. Finalmente, se generarán las gráficas para todos los ficheros de salida, almacenándose también en el mismo directorio. r