Práctica 3

Procesos e hilos: planificación y sincronización

3. Procesos e hilos: planificación y sincronización **1**
- 3.1. Procesos . . . 2
- 3.2. Hilos (*threads*). Biblioteca de hilos Posix, *pthread* . . . 4
- 3.3. Mecanismos de sincronización . . . 5
  - 3.3.1. Semáforos POSIX . . . 5
  - 3.3.2. Cerrojos para hilos . . . 6
  - 3.3.3. Variables de condición . . . 6
- 3.4. Manual de uso del simulador . . . 7
  - 3.4.1. Ejecución y generación de diagramas . . . 7
  - 3.4.2. Descripción del perfil de ejecución de las tareas . . . 8
- 3.5. Modelado del simulador . . . 10
  - 3.5.1. Equilibrado de carga . . . 11
  - 3.5.2. Estructuras de datos relevantes . . . 11
  - 3.5.3. Uso de listas doblemente enlazadas . . . 13
  - 3.5.4. Implementación de un nuevo planificador . . . 14
- 3.6. Parte obligatoria . . . 16

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;
```

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

```
int stat;

 while (....) {
   child_pid = fork ();
   if (child_pid != 0) {
  .....
   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
```

[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.

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_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 el simulador de planificadores que se ha implementado. Antes de comenzar la descripción del simulador y de las modificaciones propuestas en esta práctica, vamos a comentar brevemente cómo usarlo y sus principales funciones. En su estado actual (tal y como se entrega al alumno), el entorno es funcional y permite:

- Especificar el número de CPUs que serán simuladas
- Escoger qué planificador se usará. *RR (round-robin)* y *SJF* (primero el trabajo más corto) ya están implementados.
- Decidir si queremos la versión *preemptive* del planificador seleccionado
- Para el planificador *round-robin*, especificar el *quanto* de tiempo que se asignará a las tareas
- Seleccionar el periodo con el que se ejecutará el equilibrador de carga (además, se ejecutará siempre que una CPU esté ociosa)
- Generar gráficas de la planificación realizada en cada procesador, incluyendo información del tiempo de ejecución, tiempo bloqueado, tiempo en espera....

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:

```
$ ./schedsim -h
```

que devolverá un listado parecido al siguiente:

```
Usage: ./schedsim -i <input-file> [options]

List of options:
        -h: Displays this help message
        -n <cpus>: Sets number of CPUs for the simulato (default 1)r
        -m <nsteps>: Sets the maximum number of simulation steps (default 50)
        -s <scheduler>: Selects the scheduler for the simulation (default RR)
        -d: Turns on debug mode (default OFF)
        -p: Selects the preemptive version of SJF or PRIO (only if they are selected
            with -s)
        -t <msecs>: Selects the tick delay for the simulator (default 250)
        -q <quantum>: Set up the timeslice or quantum for the RR algorithm (default 3)
        -l <period>: Set up the load balancing period (specified in simulation steps,
            default 5)
        -L: List available scheduling algorithms
```

Muchas de las opciones son autoexplicativas, y coinciden con las funcionalidades enumeradas anteriormente. A continuación, podemos realizar una primera simulación:

```
$ ./schedsim -i examples/example1.txt
```

La ejecución imprimirá por pantalla estadísticas de la ejecución de cada una de las tareas especificadas en el fichero *examples/example1.txt* y creará un fichero `CPU_0.log` que usaremos para generar un diagrama de la planificación. Para ello, usaremos la herramienta `generate_gantt_chart` en el directorio `gantt-gplot`:

```
$ cd ../gantt-gplot
$ ./generate_gantt_chart ../schedsim/CPU_0.log
```

Si todo ha ido bien en el directorio `schedsim` se habrá generado `CPU_0.eps`. La figura 3.1 muestra el resultado para ese ejemplo concreto. Vemos la progresión de cada una de las cuatro tareas que forman este ejemplo. Las partes azules indican tiempo de ejecución en CPU (de ahí que nunca solapen), la parte amarilla es tiempo de bloqueo (p.ej., simulando entrada/salida) y las zonas grises representan el tiempo en que una tarea estaba preparada para su ejecución pero la CPU estaba ocupada con otra tarea.

Podemos ejecutar el mismo ejemplo pero definiendo un sistema con 2 CPUs:

```
$ ./schedsim -i examples/example1.txt -n 2
```

La figura 3.2 muestra el resultado obtenido tras generar las diagramas.

Como podemos comprobar, al usar dos CPUs el tiempo total de ejecución pasa de 18 unidades de tiempo a 11. El reparto inicial de tareas a CPUs se hace de forma circular en función del número de la tarea: la tarea `P1` a la CPU 0, la tarea `P2` a la CPU 1, la tarea `P3` a la CPU 0.... En este ejemplo el planificador no ha llevado a cabo ninguna migración, por lo que cada tarea finaliza en la misma CPU en la que comenzó su ejecución.

3.4.2. Descripción del perfil de ejecución de las tareas

Cada fila del fichero representa una nueva tarea que se simulará en el sistema. La primera columna representa el nombre de la tarea. La segunda columna, su prioridad. Posteriormente se indica el tiempo de llegada de la tarea al sistema (si es 0, se indica que la tarea está disponible desde el comienzo de la simulación). A partir de ahí encontramos el perfil de ejecución de cada tarea: el primer número indica el tiempo de ejecución de la primera ráfaga de CPU; luego, tiempo de espera (por E/S o similar). Y el patrón se repite hasta que la tarea finaliza (es decir, el mismo formato que se usa en la hoja de ejercicios).

Por ejemplo, en `example1.txt` se especifica que se crearán 4 tareas. La primera tarea, llamada `P1` comienza su ejecución al principio de la simulación y tiene prioridad 1. Al comenzar, tratará de usar la CPU durante una unidad de tiempo. Posteriormente, se bloqueará durante 5 unidades de tiempo. Finalmente, volverá a requerir la CPU durante 4 unidades de tiempo y finalizará su ejecución.

Ejercicio 3: escribe un fichero de entrada que simule el conjunto de tareas del ejercicio 7 de la hoja de problemas. Ejecuta el simulador para resolver los apartados *b*,*c* y *d* de dicho problema.

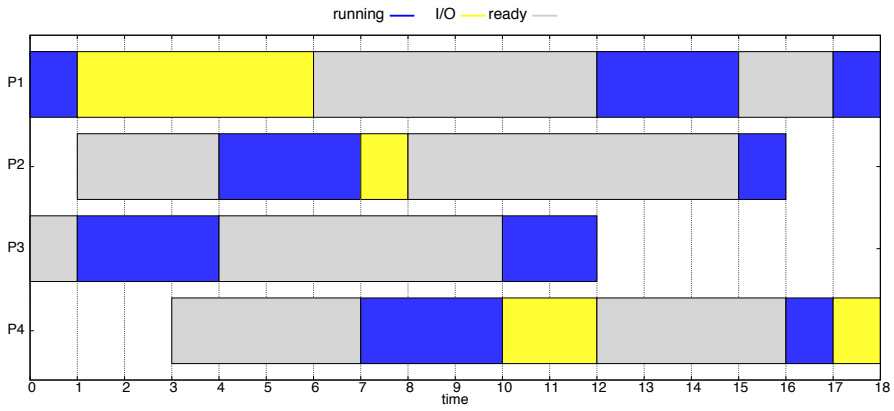
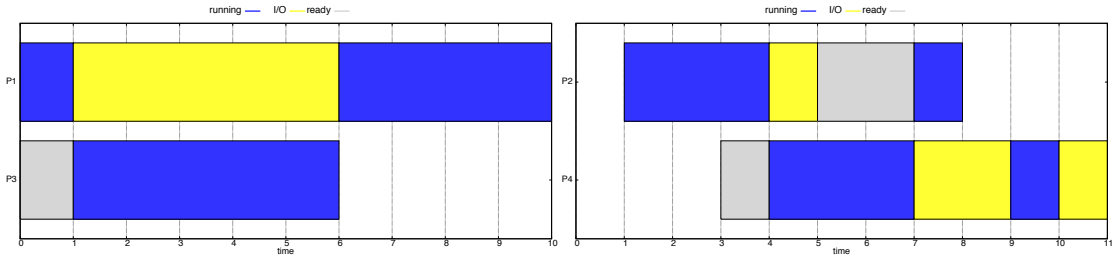


Figura 3.1: Resultado de simular el fichero example1.txt con una sola CPU



(a) Planificación en CPU 0 (b) Planificación en CPU 1

Figura 3.2: Simulación de example1.txt con 2 CPUs

3.5. Modelado del simulador

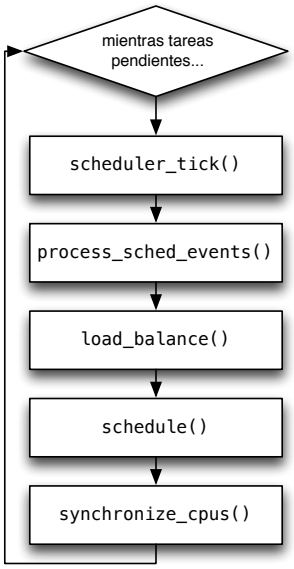


Figura 3.3: Diagrama de flujo del simulador

Hemos desarrollado un simulador de políticas de planificación basado en eventos. El código entregado facilita enormemente el desarrollo de nuevos planificadores, pues se encarga de la gestión de los eventos que van llegando al sistema (creación de tareas, migraciones, finalización de situación de bloqueo) e invoca al código de un planificador específico a través de una API muy sencilla. Por tanto, el trabajo del alumno se limitará a implementar dicha interfaz, que se detallará en la sección 3.5.4.

El simulador comienza creando un hilo por cada una de las CPUs que formen parte del sistema simulado. Tras una primera fase de inicialización de diferentes estructuras, cada hilo comienza a ejecutar la función `sched_cpu()`. La figura 3.3 ilustra el bucle de simulación presente en dicha función, que se ejecutará una y otra vez mientras queden tareas pendientes. El objetivo es simular fielmente lo que ocurriría en el planificador real en cada *tick* del reloj del sistema.

Como se observa en la figura, en cada paso de simulación (una iteración del bucle) se invocan cinco funciones, todas ellas ya implementadas en el fichero `sched.c`. En un planificador real, la llegada de una interrupción del *TIMER* del sistema (un *tick*) desencadenaría la ejecución de estas funciones.

En primer lugar se ejecuta la función **`scheduler_tick()`**. Esta función invoca a la función `task_tick()` del planificador en uso. Posteriormente comprueba si la tarea en ejecución termina en este ciclo su ráfaga de CPU (es decir, si se va a bloquear por E/S o finaliza completamente su ejecución). Si es así, pasa la tarea al estado `SLEEP` para que no sea considerada en la siguiente decisión de planificación o informa de la finalización de la tarea (si es el caso).

Posteriormente se invoca la función **`process_sched_events()`** que procesa los eventos pendientes en esta CPU (un evento es la entrada de una nueva tarea en el sistema o una tarea saliendo de su situación de bloqueo). En caso de que haya un evento, se invoca a la función `enqueue_task()` del planificador usado para que introduzca la tarea correspondiente en la *run queue* (cola de tareas listas para ejecutar asociadas a una CPU).

En tercer lugar, se llama a la función **`load_balance()`** que comprobará si es necesario realizar un equilibrado de carga. Si es así, comenzará una migración de tareas de la CPU con más carga a la CPU con menos carga. Este proceso requiere una sincronización *delicada* entre hilos, por lo que postergamos su discusión a la sección 3.5.1.

Tras el balanceo de carga, se invoca a la función **`schedule()`** que, en caso de que sea necesario tomar una decisión de planificación (esto es, si la llamada a `scheduler_tick()` marcó la cola de ejecución para replanificación) procede a escoger la tarea que pasará a ocupar la CPU. Para ello, invoca la función `pick_next_task()` del planificador en uso, que devolverá la tarea escogida de acuerdo con el criterio de planificación. En ese instante, se realiza el cambio de contexto.

Finalmente se produce la llamada a **`synchronize_cpus()`**, en la que el hilo de cada CPU comprueba si ya ha terminado el trabajo que tenía asignado. Nótese que, dado que es posible realizar migraciones de tareas, todos los hilos que simulan cada CPU deben continuar iterando hasta que todas las CPUs hayan terminado su trabajo; es posible que la CPU *i-ésima* no tenga ninguna tarea asignada en el instante *t*, pero que se produzca un equilibrado de carga que le asigne una (o más) tareas en el instante *t+1*. Esto exige una sincronización entre todos los hilos del simulador para determinar cuándo se ha finalizado realmente la simulación. Esta sincronización se realiza mediante una *barrera*. En el código entregado se utiliza una barrera POSIX (tipo `pthread_barrier_t`), que se basa en el uso de la función `pthread_barrier_wait()`. Uno de los puntos de la parte obligatoria será implementar este mecanismo utilizando cerrojos y variables condicionales.

3.5.1. Equilibrado de carga

En un ciclo de simulación sin equilibrado de carga, cada CPU simulada (esto es, cada hilo de ejecución en nuestro simulador) consultará (y modificará si procede) únicamente su propia *run queue* a la hora de tomar una decisión de planificación. Sin embargo, durante el proceso de equilibrado de carga, es posible que el hilo de la CPU0 deba acceder a la *run queue* de la CPU1, lo que convierte a estas colas en secciones críticas que debemos proteger con un cerrojo.

Examinando con detalle el código de la función `load_balance()`, comprobamos que puede haber dos hilos (el de la CPU con mayor carga y el de la CPU con menor carga) que van a tratar de conseguir los cerrojos de sus respectivas *run queues*. Dependiendo del orden en que se realice la adquisición de dichos cerrojos, podemos encontrarnos con una situación de interbloqueo.

Para evitar esa situación, se establece un orden en el que deben adquirirse los cerrojos. Esa estrategia está implementada en la función `double_lock_rq`: hay que empezar por adquirir los cerrojos de las CPUs con IDs mayores. En algún caso, eso puede exigir liberar temporalmente el cerrojo propio para volver a adquirirlos en el orden establecido.

Ejercicio 4: La solución adoptada para evitar interbloqueos en el equilibrador de carga se puede utilizar para resolver el problema de *los filósofos*. Implementa un código que modele dicho problema 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.

3.5.2. Estructuras de datos relevantes

Cada una de las CPUs simuladas dispone de su propia *run queue*, una cola donde almacena las tareas asignadas a esa CPU y que irán ejecutándose siguiendo la estrategia de planificación concreta. El tipo de datos empleado para una *run queue* es:

```
typedef struct{
    slist_t tasks;      /* runnable task queue */
    task_t* cur_task;   /* Pointer to the task in the CPU. It may be the idle task*/
    task_t idle_task;   /* This CPU's idle task */
    bool need_resched;  /* Flag activated when a user preemption must take place */
```

```
    int nr_runnable;     /* Keeps track of the number of runnable task in this CPU
                            -> Note that current is not on the RQ */
    int next_load_balancing;     /* Timestamp of the next simulation step
                                    where load_balancing will take place */
    void* rq_cs_data;    /* Pointer enabling a scheduling class to store
                                 private data if needed */
    pthread_mutex_t lock;        /* Runqueue lock*/
}runqueue_t;
```

El primer campo, `tasks`, es la lista de tareas en espera de poder usar la CPU. En la sección 3.5.3 se explica el API para gestionar esta lista. El campo `cur_task` es un puntero a la tarea que actualmente está en ejecución en la CPU asociada a esta *run queue*. Es importante tener en cuenta que esta tarea (la actual) no está incluida en la lista de tareas del campo anterior.

Cada *run queue* tiene su propia `idle_task` que tomará la CPU cuando no haya ninguna tarea *runnable*, esto es, en disposición de ejecutarse. El flag `need_resched` deberá ponerse a `TRUE` cuando se deba proceder a replanificar (elegir una nueva tarea para su ejecución). `nr_runnable` es el contador del número de tareas actualmente en la cola.
`next_load_balancing` especifica en qué instante de tiempo (del futuro) debe estudiarse un nuevo equilibrado de carga. El campo `rq_cs_data` es un puntero genérico (void*) que permite que el algoritmo de planificación escogido apunte a una estructura privada con los campos que desee. En la versión del simulador que se proporciona, ninguna estrategia de planificación hace uso de datos privados en la *run queue*, por lo que este campo se pone a `NULL`. Por último, el campo `lock` es el cerrojo que usaremos para acceder en exclusiva a los diferentes campos de esta estructura.

Otro tipo de datos relevante para la implementación de nuevos algoritmos de planificación es el de una tarea[4]: `task_t`.

```
typedef struct{
    int task_id;                    /* Internal ID for the task*/
    char task_name[MAX_TASK_NAME];
    exec_profile_t task_profile; /* Task behavior */
    int prio;
    task_state_t state;
    int last_cpu;               /* CPU where the task ran last time */
    int last_time_enqueued;     /* Last simulation step where the task was enqueued */
    int runnable_ticks_left;    /* Number of ticks the application has to
                                       complete till blocking or exiting */
    list_node_t ts_links;        /* Node for the global task list */
    list_node_t rq_links;       /* Node for the RQ list */
    bool on_rq;                 /* Marker to check if the task is on the rq or not */
    unsigned long flags;        /* generic flags field */
    void* tcs_data;     /* Pointer enabling a scheduling class to store private
                                                    data if needed */
/* Global statistics */
    int user_time;              /* CPU time */
    int real_time;              /* Elapsed time since the application entered the
        system */
    int sys_time;               /* For now this time reflects the time the thread
                                 spends doing IO */
    slist_t sched_regs;         /* Linked list to keep track of the sched log registers
                                       (track state changes for later use) */
}task_t;
```

Muchos de los campos de esta estructura tienen un carácter estadístico y son autoexplicativos. Asimismo, la mayoría de los campos se utilizan únicamente en el fichero `sched.c` que

[4]Durante esta práctica usamos siempre el término tarea para referirnos a las entidades que serán simuladas; usamos *hilo* para referirnos a hilos reales que usamos para implementar el simulador. El tipo de datos `task_t` es parte de lo que podría contener el *Bloque de Control de Hilo* de un sistema operativo real.

se da completamente implementado. Los campos más reseñables de cara a la realización de la práctica son:

- `prio` es la prioridad de la tarea, que puede usarse para determinar cuál es la siguiente tarea que debe acceder a la CPU.
- `runnable_ticks_left` indica cuántos *ticks* restan para que finalice la ráfaga de CPU actual. La inicialización de este campo se realiza en `sched.c`. Este campo puede necesitarse para escoger la siguiente tarea a ejecutar: así ocurre en el algoritmo SJF.
- `on_rq` es un booleano que el planificador pone a `TRUE` cuando la tarea esté en la *run queue* (es decir, no está en ejecución ni bloqueada).
- `flags` puede usarse para anotar cualquier situación que consideremos oportuna para el planificador. Actualmente hay definidos dos *flags* (en el fichero `sched.h`):
  - `TF_IDLE_TASK` que nos permite saber rápidamente si una tarea es la tarea *idle*
  - `TF_INSERT_FRONT` que utiliza la política SJF para introducir una tarea al principio de la lista en lugar de al final en determinadas situaciones.
- `tcs_data` es un puntero genérico (void*) que permite que el algoritmo de planificacion activo apunte a una estructura con campos privados de la tarea para uso interno del algoritmo de planificación. En la versión inicial del simulador, solo el algoritmo RR hace uso de este puntero (ver fichero `sched_rr.c`); para el resto de políticas de planificación este campo se pone a `NULL`.

3.5.3. Uso de listas doblemente enlazadas

En el fichero `slist.h` se define el tipo `slist_t` que permite la creación de listas enlazadas de cualquier tipo de elementos. Como se indicó anteriormente, una *run queue* posee, entre otras cosas, una lista de tareas (estructuras `task_t`) que el planificador deberá gestionar. El API proporcionado para interaccionar con estas listas es el siguiente:

```
void init_slist (slist_t* slist, size_t node_offset);
void insert_slist ( slist_t* slist,void* elem);
void insert_slist_head ( slist_t* slist,void* elem);
void remove_slist ( slist_t* slist,void* elem);
void* head_slist ( slist_t* slist);
void* tail_slist ( slist_t* slist);
void* next_slist ( slist_t* slist,void* elem);
void* prev_slist ( slist_t* slist,void* elem);
void insert_after_slist(slist_t* slist, void *object, void *nobject);
void insert_before_slist(slist_t* slist, void *object, void *nobject);
int is_empty_slist(slist_t* slist);
int size_slist(slist_t* slist);
void sorted_insert_slist(slist_t* slist, void* object, int ascending, int (*compare)(
    void*,void*));
void sorted_insert_slist_front(slist_t* slist, void* object, int ascending, int (*
    compare)(void*,void*));
void sort_slist(slist_t* slist, int ascending, int (*compare)(void*,void*));
```

Nuevamente, la mayoría de las funciones son autoexplicativas. A modo de ejemplo comentamos algunas de las funciones más utilizadas:

- `head_slist()` devuelve el primer elemento de la lista (pero no lo saca de la lista).

- `tail_slist()` devuelve el último elemento de la lista (pero no lo saca de la lista).
- `remove_slist()` elimina el elemento `elem` de la lista.
- `insert_slist()` inserta el elemento `elem` al final de la lista.
- `sorted_insert_slist()` realiza una inserción ordenada del elemento `object` al final de la lista. La función de comparación que se empleará es el cuarto argumento de la función. Se puede consultar un uso de esta función en el fichero `sched_sjf.c`.

3.5.4. Implementación de un nuevo planificador

Ya estamos en disposición de implementar un nuevo planificador. Para ello basta implementar las funciones de la interfaz de un planificador (estructura `sched_class_t` definida en `sched.h`). Dicha interfaz consta de las siguientes operaciones:

```
typedef struct sched_class {
    int (*sched_init)(void);
    void (*sched_destroy)(void);
    int (*task_new)(task_t* t);
    void (*task_free)(task_t* t);
    task_t* (*pick_next_task)(runqueue_t* rq);
    void (*enqueue_task)(task_t* t,runqueue_t* rq, int runnable);
    void (*task_tick)(runqueue_t* rq);
    task_t* (*steal_task)(runqueue_t* rq);
}sched_class_t;
```

Las operaciones `pick_next_task()`, `enqueue_task()` y `steal_task()` se han de implementar siempre en todos los algoritmos de planificación. El resto de operaciones sólo se implementan en algunos casos que detallaremos a continuación. Describimos ahora sucintamente qué deben hacer estas funciones:

- `sched_init()` y `sched_destroy()` son respectivamente las operaciones de inicialización y destrucción de un planificador y se invocan una sola vez durante la simulación. Estas operaciones han de implementarse solamente en el caso de que el planificador necesite almacenar información extra en la *run queue* de cada CPU, es decir, usando campos que no estén definidos en `runqueue_t`. En tal caso, el planificador hará uso del puntero genérico `rq_cs_data`, campo de `runqueue_t`, que se inicializará adecuadamente en `sched_init()` y cuya memoria se liberará en `sched_destroy()`. En la versión básica del simulador que se proporciona ninguna política de planificación implementa estas funciones, al no precisar de campos extra en la *run queue*. No obstante, se podría hacer uso de esta característica en alguna ampliación propuesta como parte extra de la práctica.
- `task_new()` y `task_free()` se invocan al crear una tarea y al destruirla, respectivamente. Estas operaciones han de implementarse solamente si el planificador necesita mantener campos extra de cada tarea; es decir, campos no definidos en la estructura `task_t`. El planificador RR es el único algoritmo del simulador que implementa estas operaciones, ya que reserva una estructura con campos extra por cada tarea en `task_new()` (consultar fichero `sched_rr.c`). Como se comentó previamente, el puntero genérico `tcs_data` de `task_t` sirve para apuntar a la estructura con campos extra. Esto permite recuperar esta estructura en las operaciones de la interfaz del planificador.
- `pick_next_task()` se invocará cuando se deba escoger una nueva tarea para ejecutar en la CPU. Seleccionará una tarea de la *run queue*, la eliminará de la *run queue* y devolverá dicha tarea. Es la función que realmente lleva a cabo la política de planificación deseada.

- `enqueue_task()` se invocará cada vez que debamos encolar una tarea en la *run queue* de una CPU, bien porque acaba de crearse, porque sale de un bloqueo o porque se ha migrado desde otra *run queue*.

- `task_tick()` se invocará en cada *tick* de simulación (iteración del bucle principal). En políticas de planificación expropiativas, como RR, se usa esta operación para actualizar alguna métrica de la tarea, que resulta relevante para determinar cuándo se debe producir una replanificación.

- `steal_task()` se invocará cuando se lleve a cabo una migración que exija *robar* una tarea de la *run queue* `rq`. La tarea escogida se sacará de la *run queue* y se devolverá como argumento de salida de la función.

Para implementar un nuevo algoritmo de planificación debemos implementar el subconjunto de operaciones necesarias de la interfaz `sched_class_t` en un nuevo fichero `.c`. Por convenio, el nombre de la función que implementa cada operación de la interfaz contendrá un sufijo con el nombre en minúscula del algoritmo de planificación precedido por "_". Así por ejemplo, la función `pick_next_task_rr()` implementa la operación `pick_next_task` del algoritmo RR.

Una vez se hayan definido estas operaciones, se ha de especificar en el mismo fichero `.c` la relación entre cada operación de la interfaz y la función que la implementa. Para ello, basta con definir una variable de tipo `sched_class_t` inicializada adecuadamente. Por ejemplo, la variable `rr_sched`, definida en `sched_rr.c`, define el planificador RR:

```
sched_class_t rr_sched={
    .task_new=task_new_rr,
    .task_free=task_free_rr,
    .pick_next_task=pick_next_task_rr,
    .enqueue_task=enqueue_task_rr,
    .task_tick=task_tick_rr,
    .steal_task=steal_task_rr
};
```

Por último, para acabar con la definición del nuevo planificador, será preciso añadir una nueva entrada en la estructura `available_schedulers`, (`sched.h`), así como en el tipo enumerado definido justo antes de dicha variable. El contenido actual de la variable es:

```
static const sched_choice_t available_schedulers[NR_AVAILABLE_SCHEDULERS]={
    {RR_SCHED,"RR",&rr_sched},
    {SJF_SCHED,"SJF",&sjf_sched},
};
```

Para registrar un nuevo planificador en el simulador bastaría añadir una nueva línea indicando la variable de tipo `sched_class_t` creada en el nuevo fichero `.c` con la implementación del planificador.

En las secciones anteriores se mostraba el contenido completo de dos estructuras de datos relevantes en el simulador. Sin embargo, a la hora de implementar una nueva política de planificación, los únicos campos de las estructuras `runqueue_t` y `task_t` que se deben modificar son `rq->tasks`, `rq->need_resched` y `tsk->flags`. Es importante notar que el cerrojo de la *run queue* no está en esta lista: en el momento de invocar a las funciones de un planificador específico, el cerrojo de la *run queue* ya se ha adquirido si era necesario, por lo que no debemos preocuparnos de ese aspecto durante la implementación.

3.6. Parte obligatoria

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

- Crear un planificador FCFS (*First Come First Served*): el trabajo que lleve más tiempo esperando será el siguiente en ocupar la CPU. A la hora de *ceder* tareas a otra *run queue* se escogerá la última tarea de la cola (la que lleva menos tiempo esperando). Crear un fichero `sched_fcfs.c` en el que se implementarán todas las operaciones de la interfaz `sched_class_t` que el alumno considere necesarias para construir el planificador FCFS.

- Crear un planificador **expropiativo** basado en prioridades. El único criterio de planificación será la prioridad asignada a cada tarea de forma estática en el fichero de configuración. Para la implementación de este planificador es aconsejable fijarse en la estrategia de implementación del algoritmo SJF.[5] En caso de migración, se cederá la tarea menos prioritaria de las que estén en la cola. Crear un fichero `sched_prio.c` en el que se implementarán todas las operaciones necesarias de la interfaz `sched_class_t`.

- 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 dos datos:

  - Qué fichero de ejemplo desea simular. 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
  - Número máximo de CPUs que se desean usar en la simulación. El número introducido no deberá ser mayor que 8. Si es mayor, se informará al usuario del error y se le volverá a preguntar el número de CPUs.

  A continuación, se creará un directorio `resultados` y se ejecutará el simulador para cada uno de los 4 planificadores disponibles, y para todas las CPUs entre 1 y el número escogido por el usuario. Todos 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.

[5] Notese que la variable global `preemptive_scheduler` que se usa en la implementación de SJF se pone a 1 al especificar la opción `-p` de la línea de comando al invocar el simulador.

El pseudo-código de la parte central del *script* sería:

```
maxCPUs = valor introducido por usuario

foreach nameSched in  listaDeSchedulersDisponibles
do
    for cpus = 1 to maxCPUs
    do
      ./sched-sim -n cpus -i .......
      for i=1 to cpus
      do
        mover CPU_$i.log a  results/nameSched-CPU-$i.log
        generar gráfica
      done
    done
done
```