

## Assignment 3

# Processes and threads: scheduling and synchronization

<b>3</b>	<b>Processes and threads: scheduling and synchronization</b>	<b>1</b>
3.1	Processes . . . . .	2
3.2	Threads. POSIX Threads library. . . . .	4
3.3	Synchronization Mechanisms . . . . .	5
3.3.1	POSIX semaphores . . . . .	5
3.3.2	Mutexes or Locks . . . . .	6
3.3.3	Condition variables . . . . .	6
3.4	Getting started with the simulator . . . . .	7
3.4.1	Running the simulator and generating diagrams . . . . .	7
3.4.2	Specifying tasks' properties . . . . .	8
3.5	Design of the scheduling simulator . . . . .	10
3.5.1	Load balancing . . . . .	11
3.5.2	Relevant data structures . . . . .	12
3.5.3	Doubly-linked lists . . . . .	14
3.5.4	Implementing a new scheduling algorithm . . . . .	15
3.6	Mandatory part . . . . .	16

## Objectives

In this lab assignment different scheduling algorithms will be implemented on top of a simulator. The design of simulator is inspired by the internal structure of schedulers in modern operating systems such as GNU/Linux or Solaris.

Because the simulated scheduling infrastructure is multi-threaded and runs at user space, its implementation relies on the resources that the operating system provides for the creation of concurrent programs. Specifically, the simulator consists of several threads that synchronize with each other by means of locks and condition variables.

We now provide an overview of the process and thread-related concepts presented in class as well as enumerate the most relevant associated system calls and library functions. Moreover, several exercises are proposed to help students accomplish the goals of this lab assignment.

### 3.1 Processes

Essentially, a process is the instance of a program in execution. In this lab assignment, students will get familiar with the most relevant system calls for process management. Nevertheless, the assignment itself focuses on working with threads at user space.

To create a new process, as a clone of the current process, the following system call must be used:

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

Fork() triggers the creation of a new process in the operating system. The new process (*child* process) is an identical copy of the calling process (*parent* process). This means that all memory regions inside the process are COPIED, so parent and child process do not share memory.

The following code snippet illustrates how to use `fork()`:

```
int main ()
{
    pid_t child_pid;

    child_pid = fork ();
    if (child_pid != 0) {
        // This code is executed by the PARENT process ONLY
        .....
    } else {
        // This code is executed by the CHILD process ONLY
        .....
    }
    // BOTH the parent and the child will run this code
    // (unless the process invoked exec, exit, or returned from the main function.)
    .....
}
```

Usually, the newly created child process may wish to execute a different program stored in a given executable file. This entails destroying the old process memory image (inherited from the parent upon `fork()`) and creating a brand new one based on the information in the executable file. To make this possible, a function from the *execxx* family must be used:

```
#include <unistd.h>

int execl(const char *path, const char *arg, ... );
int execlp(const char *file, const char *arg, ... );
int execlx(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[]);
```

To terminate the execution of a process, the `main()` function in the associated program must complete (e.g., by using `return`). Alternatively, the process termination can be enforced by invoking the following function at any point in the process's code:

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

Finally, the OS offers a set of system calls to make it possible for a parent process to wait for the termination of a child process:

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

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

Notably, a common use case of the aforementioned functions and system calls is as follows:

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

    while (....) {
        child_pid = fork ();
        if (child_pid != 0) {
            ....
        }
        else {
            execv(...);
            // Just in case execv() fails...
            exit(-1);
        }
        // Parent waits until child process completes
        if (wait(&stat) == -1) {
            ....
        }
        if (WIFEXITED(stat)) {
            ....
        }
    }
}
```

**Exercise 1:** Analyze the source code of the program *fork\_example.c* and answer the following questions:

- How many processes are created? Draw the resulting process tree.
- What is the maximum number of processes running simultaneously?
- During the program's execution, Is it possible that any process enters the *zombie* state? To figure this out, calls to the *sleep()* function can be introduced in the code. Use the *top* tool during the execution if necessary.
- Would the program behavior be the same if the memory region pointed by the *p\_heap* variable were not allocated in the heap but instead was defined statically as an *integer* global variable?
- How would the program behavior be affected in the event the invocation to *open()* was performed separately by each child process instead of how it is done in the original program?
- Is it possible that any child process created by the program eventually becomes a child of *init* (PID=1)? Use the *top* tool during the execution to try to figure this out. Note that the code can be changed to make it possible to find an answer to this question.

Lastly, before continuing with the discussion of the thread-related functions, it is worth recalling which memory regions and resources are shared between a child and a parent process (after *fork()*) and between threads of the same process:

Memory region/Resource	Parent-Child process	Threads in the same process
Shared variables (.bss, .data)	NO	YES
Local variables (stack)	NO	NO
Dynamically allocated memory (heap)	NO	YES
Open file descriptor table	Each process features a private table (Duplicated after fork)	Shared

### 3.2 Threads. POSIX Threads library.

The POSIX Threads library (`libpthread`) available on Linux and on most Unix and Unix-like OSes, makes it possible to create additional threads in a process. To use functions of this library in a C program, we must include the following preprocessor statement at the beginning of C files:

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

We must also build the program as follows:

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

When a process starts its life-cycle, right after being created with `fork()`, the process features a single thread. Additional threads can be created with the following function:

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

When invoking `pthread_create()`, the programmer may specify a set of attributes via the `attr` parameter, which the OS will take into consideration when creating the thread. To use the default settings when creating a thread, a NULL pointer can be passed as the second argument of the function. By contrast, to customize thread properties upon creation, the programmer must declare an attribute set (`attr_t` variable) and then initialize it by using the following functions:

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

where XXX denotes a specific attribute that can be defined for a thread, such as `scope`, `detachstate`, `schedpolicy` and `inheritsched`.

To retrieve attribute values associated with a given thread, the following function can be used:

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

A thread can terminate its execution under different circumstances:

- Its associated `func()` function returns.
- The process associated with this thread exits or terminates abruptly (e.g., due a segmentation fault).
- Any thread in the process invokes `exec()`.
- The thread invokes the following function:

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

Once a thread is created, another thread in the process can wait for this thread to finish by using `pthread_join()`:

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


A thread can obtain its *thread ID* (`tid`) by using the following function:

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

More information about the POSIX threads API can be obtained by typing the following command: `man 7 pthreads`. Recall that all threads in the same process share the same memory address space as well as most resources in the process (e.g. open files). As such, threads share global variables and the heap memory region. Nevertheless, each thread features a separate stack (for local variables).

**Example 1. Partial sum:** The program `partial_sum.c` creates two threads that cooperate to evaluate the following formula in parallel:

$$total\_sum = \sum_{i=1}^{10000} i$$

When running the program multiple times, we can observe that the final result (wrong in most cases) varies from run to run. What is wrong with this solution? 

### 3.3 Synchronization Mechanisms

This section reviews the synchronization mechanisms that will be used in the lab assignment. As such, special attention will be paid to the mechanisms enabling to synchronize threads.

#### 3.3.1 POSIX semaphores

GNU/Linux is equipped with an implementation of general semaphores compliant with the POSIX standard. Specifically, the *unnamed* variant of POSIX semaphores is well-suited to synchronization between threads of the same process.<sup>1</sup> Typically, the *main* thread of the process is in charge of creating the necessary *unnamed* semaphores. The remaining threads in the process can then access these semaphores to enforce synchronization when needed.

The available functions enabling to perform operations with unnamed POSIX semaphores are as follows:

```
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);
```

More information about these functions can be found in section 3 of Linux man pages.

**Exercise 2:** Use an unnamed POSIX semaphore in the program of Example 1 (Partial Sum) to ensure the program always provides the correct result. In addition, modify the main function to enable to pass two command-line arguments to the program, enabling to specify the upper limit

<sup>1</sup> Note that POSIX semaphores can be also used to synchronize threads from different processes. Nevertheless, *named* semaphores (not discussed here) constitute a more flexible approach than *unnamed* semaphores for this task.

of the summation and the number of threads used to perform the associated calculation in parallel. For example, the command “./partial\_sum 50000 5” will calculate the summation of numbers ranging between 1 and 50000 using 5 threads.

### 3.3.2 Mutexes or Locks

A mutex or lock is a mechanism specifically tailored for synchronization between threads. Its main purpose is to enforce mutual exclusion when accessing shared resources in critical sections. When a mutex is held by a specific thread, this thread is the current owner of the mutex. Note that only the owner can release the mutex. The POSIX threads library on GNU/Linux is equipped with mutexes. The associated operations are implemented by the following functions:

```
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);
```

A mutex can be created by declaring a `pthread_mutex_t` variable that is later initialized via the `pthread_mutex_init()` function shown earlier. Alternatively, the initialization can be done in a declarative way as follows:

```
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;
```

### 3.3.3 Condition variables

A condition variable is a synchronization mechanism used to block a thread until a certain condition in the program becomes true. The condition is typically expressed in terms of the state of the variables shared among threads. Unlike other synchronization mechanisms, condition variables are not standalone. Instead, a common mutex, typically shared among the set of condition variables used in the concurrent program, is necessary to control the invocation of blocking operations on condition variables.

In the implementation available on Linux’s POSIX threads library, condition variables follow the Lamport-Redell semantics, in which the signalling thread has a higher priority than the signalled thread. As such, the signalling thread does not grant the mutex directly to the signalled thread, but instead releases the mutex shortly after signalling. Hence, the signalled thread may have to compete for the mutex with other threads in the concurrent program. As a result, the signalled thread must verify the condition associated with the condition variable once it wakes up and acquires the mutex.

The functions enabling to perform operations on condition variables (man `pthread`) are as follows:

```
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);
```

As mutexes in the POSIX Threads library, condition variables can be initialized with a function (`pthread_cond_init()`) or in a declarative way using a macro, as shown below:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

## 3.4 Getting started with the simulator

This lab assignment entails extending the functionality of a scheduling simulator. Before getting started with a detailed description of the simulator and enumerating the required changes in the code, we will briefly discuss how to use it and describe its main functions. In its current state (as it is provided to students), the simulator is already functional and enables to:

- Specify the number of CPUs to be simulated
- Pick a scheduling algorithm to use. The *RR* (*round-robin*) and *SJF* (shortest job first) algorithms are implemented already.
- Decide whether the *preemptive* version of the selected scheduler will be used or not (in the event it is available).
- Indicate the *quantum* (or time slice) used for the *round-robin* scheduler.
- Specify how often (in *ticks*) the per-CPU load balancer will be activated. Note that the load balancer also gets activated on idle CPUs every *tick*.
- Display information about the simulation including the completion time, user (CPU) time and I/O time associated with each simulated task under the selected scheduling algorithm.
- Generate charts that show the state (running, waiting, ready) of every simulated task over time under a particular scheduling algorithm.

### 3.4.1 Running the simulator and generating diagrams

After building the simulator (by using the available *Makefile*), we can proceed to execute it. To display the available command-line options supported by the simulator we can invoke it as follows:

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

this will display the following information on the screen:

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

List of options:

```
-h: Displays this help message
-n <cpus>: Sets number of CPUs for the simulator (default 1)
-m <nsteps>: Sets the maximum number of simulation steps (default 50)
-s <scheduler>: Selects the scheduler for the simulation (default RR)
```

## 8 ASSIGNMENT 3. PROCESSES AND THREADS: SCHEDULING AND SYNCHRONIZATION

```
-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
```

Many of the options are self-explanatory, and match the features listed above. We can now perform our first simulation:

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

The simulator will print on the screen various statistics associated with the simulation. The “-i” switch makes it possible indicate the pathname of a file that contains the features of the various tasks used during the simulation. Because the “-s” option is not specified in the command line, the default algorithm will be selected for the simulation (RR with quantum=3). Similarly, the default CPU count (1) will be used, as we did not use the “-n” option in the command line.

As a result of running the aforementioned command, the simulator will create a file named `CPU_0.log`. We will use this file to generate a scheduling diagram by using the `generate_gantt_chart` command-line tool found in the `gantt-gplot` directory. Specifically, we can generate the diagram as follows:

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

If everything went well, we will find the `CPU_0.eps` file in the `schedsim` directory. Figure 3.1 shows the scheduling diagram associated with that file. The figure depicts the state of each task over time for the corresponding simulation. As evident, four tasks (P1, P2, P3 and P4) are used in this simulation, as displayed on the y axis. Because we are using one simulated CPU, only one process can be in the running state (blue regions) at a time. Yellow regions, by contrast, indicate that a task is going through and I/O phase (waiting state). Finally, gray regions are used to represent a process in the ready state, thus sitting on a scheduler run queue until it has a chance to run.

We can use the same input file to perform a simulation with 2 CPUs as follows:

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

Figure 3.2 depicts the resulting diagrams associated with the simulation, one for CPU 0 and another for CPU 1. As evident, in using two CPUs, the total execution time is reduced from 18 time units to 11. The initial task-to-CPU mapping is performed on a round-robin basis (regardless of the scheduling algorithm selected) taking the task number into consideration as follows: task P1 is mapped to CPU 0, task P2 is mapped to CPU 1, task P3 is mapped to CPU 0, and so on ... In this example the load balancer in the scheduler did not trigger any migration. As such, each task spends its entire execution mapped to the same CPU.

### 3.4.2 Specifying tasks' properties

In a simulation input file (`-i <file>`), each row represents a new task to be simulated in the system. The first column represents the name of the task. The second one, the task's priority. The third column indicates the arrival time of the task (when it enters the system). For example, in the event the arrival time is 0, then the task is available from the beginning of the simulation.



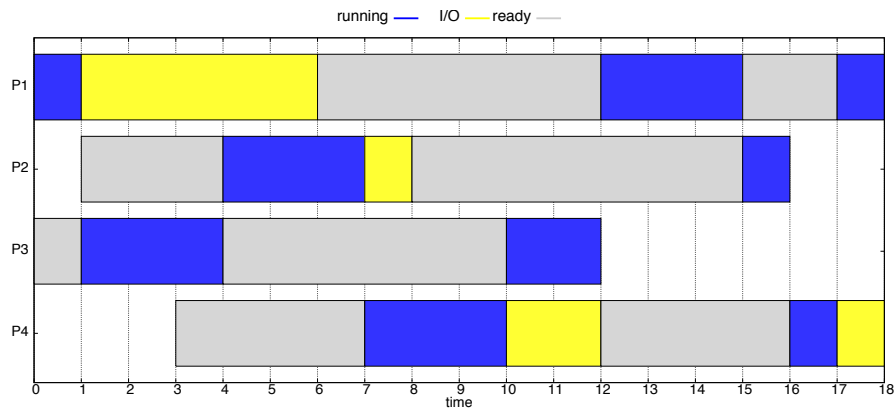
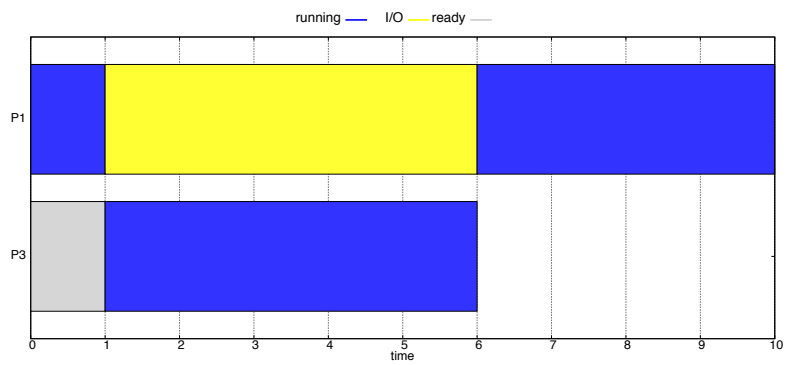
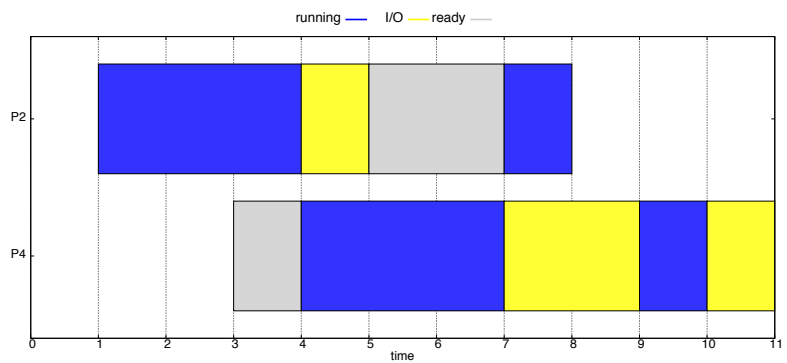


Figure 3.1: Simulation of tasks in the example1.txt file with a single CPU



(a) Schedule for CPU 0



(b) Schedule for CPU 1

Figure 3.2: Simulation of tasks in the example1.txt file with 2 CPUs

Subsequent columns specify the task's execution profile, which is represented as a sequence of interspersed CPU bursts and I/O burst (set of numbers). Specifically the first number indicates the length of the first CPU burst followed by the length of the subsequent I/O phase, followed by the length of another CPU phase, and so on. As such, the execution of the job completes when the last burst (CPU or I/O) indicated in the corresponding row of the file is completed. Note that this format to represent tasks' profiles matches the one used in the exercise sheet.

For instance, the `example1.txt` file describes the behavior of 4 tasks. The first task (P1) enters the system at  $t = 0$  and its priority is 1. When mapped to a CPU for the first time, it will run for 1 time unit and then it will block for 5 time units. After that it will enter the "Ready" state again and, when mapped to a CPU, it will run 4 time units (not necessarily without being preempted) before completing.

**Exercise 3:** write an input file to simulate the set of tasks described in Exercise #7 of the exercise sheet. Use the simulator to obtain the solution for parts *b*, *c* and *d* in that exercise.

### 3.5 Design of the scheduling simulator

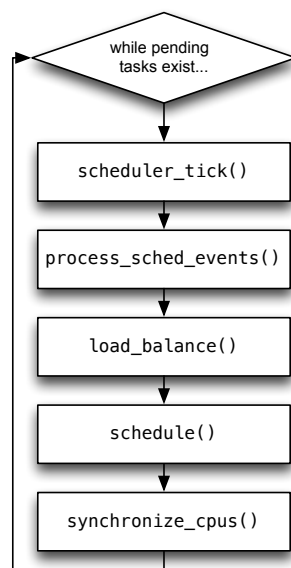


Figure 3.3: Simulator's flow diagram

We have designed and implemented an event-driven scheduling simulator. The source code of the simulator, provided as supplementary material of this lab assignment, greatly simplifies the implementation of new scheduling algorithms on it, as the core part of the simulator already takes care of dealing with scheduling-related events (task creation, task migrations, waking up tasks,...) and invokes the code of the *active* scheduling algorithm, implemented by means of a simple API. As such, students will create new scheduling algorithms by implementing a well-defined interface of operations presented in Section 3.5.4.

The simulator starts the execution by creating a thread for each simulated CPU.<sup>2</sup> After initializing various data structures, each thread will run the `sched_cpu()` function. Figure 3.3 depicts the workflow of the simulation loop found in `sched_cpu()`. The loop will be invoked until all tasks in the simulation have finished. The goal of the

`sched_cpu()` function is to simulate accurately what would actually happen on a CPU of the system every clock *tick* when using a real scheduling algorithm

As shown in the figure, five functions (already implemented in the `sched.c` file) are invoked on each simulation step (i.e., an iteration of the loop). In an actual OS scheduler, an

<sup>2</sup>Recall that the CPU count during the simulation can be specified via the `-n` option in the command line.

interrupt of the *system timer* would actually trigger the invocation of many of these functions.

The `scheduler_tick()` function is invoked first. In turn, this function invokes the `task_tick()` operation of the active scheduling algorithm. Then, `scheduler_tick()` checks whether the task running on that CPU will block due to I/O when completing this simulation cycle. If so, it sets the state of the task to the `SLEEP` value (*waiting state*), so that it is not selected to run by the scheduler later on. Otherwise the function checks if this cycle constitutes the last execution cycle of this task. If this is the case, the function reports the termination of the task.

The second step of the simulation cycle is performed by invoking `process_sched_events()`. This function processes the scheduling-related events that take place on this simulation cycle, such as when a new task enters the system or when a task wakes up upon completion of an I/O operation. In case an event leads a task to enter the *ready* state, the `enqueue_task()` operation of the active scheduling algorithm is invoked, enabling it to add a new runnable task to the *run queue* associated with that CPU.

Third, the `load_balance()` will be invoked to check whether a load balancing operation should be performed. If so, it will migrate tasks from CPUs with the highest load to CPUs with a lower load. This process requires synchronization between threads. We will elaborate on this aspects in Section 3.5.1.

Fourth, the simulation thread will invoke `schedule()`. In the event that the *resched* flag in the *run queue* associated with this CPU has been enabled during the processing that takes place in `scheduler_tick()`, the `schedule()` function will perform a set of actions. First it will select a task to run on that CPU by invoking the `pick_next_task()` operation of the active scheduling algorithm. If the selected task is not running already on the CPU, the scheduler will perform a context switch.

Finally, the simulation thread invokes the `synchronize_cpus()` function, used by the thread to verify if it has completed all the work assigned to it (tasks assigned to the CPU). Because task migrations may be triggered, every per-CPU thread in the simulator should proceed with a simulation step until all CPUs have completed their work. Note that a CPU may become idle in an iteration (no tasks assigned to it), but it can be assigned more work to do in the next iteration if a task is migrated to the associated run queue. Due to this issue, global synchronization among per-CPU threads is required. This is accomplished by using a *barrier*. The provided simulator code uses the implementation of barriers found in the POSIX threads library (`pthread_barrier_t` type) which relies on the `pthread_barrier_wait()` function. As part of this lab assignment, students will have to create an alternative implementation for this barrier by using mutexes and condition variables.

### 3.5.1 Load balancing

In a simulation cycle without load balancing, each simulated CPU (simulation thread) will access its own *run queue* when making scheduling decisions. However during load balancing, the simulation thread associated with CPU 0 may have to access the *run queue* of CPU 1. Thus, run queues constitute shared resources among threads, so they must be protected by means of locks.

By analyzing the implementation of the `load_balance()` function, we can observe that two threads (from the highest loaded CPU and the lowest loaded CPU) will attempt to acquire the locks associated with both run queues. If the order followed by the threads when acquiring the locks is not carefully controlled, we can run into a deadlock in this scenario.

To overcome this problem, a locking protocol is defined. This protocol is implemented in the `double_lock_rq` function. This function always ensures that locks associated with CPUs with a greater ID are acquired first. Note that, in some scenarios, the lock associated with the

run queue of the current CPU may have to be released first and reacquired later to follow the locking protocol.

**Exercise 4:** The locking protocol used to prevent deadlocks can be employed to solve the *dining philosophers* problem as well.

Complete the implementation of the `philosophers.c` program. In this program, each philosopher is emulated by means of a separate thread. Hence, the program features 5 threads, created with `pthread_create()`. Each philosopher repeatedly thinks and eats. Specifically, *thinking* is simulated via the `sleep()` function with a random time period. Before eating, the philosopher must grab its left and right fork (not necessarily in that order...). After eating, the philosopher will put the forks back on the table, and will then sleep for some time before thinking again. In implementing the program, grabbing a fork will be done by acquiring a lock. Similarly, putting a fork back on the table implies releasing a lock.

### 3.5.2 Relevant data structures

Each simulated CPU has a *run queue* associated with it. Each run queue features an actual task queue consisting of tasks in the “ready” state assigned to that CPU. The `runqueue_t` data type represents a per-CPU run queue and is defined as follows:

```
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;
```

The first field, `tasks`, is a doubly-linked list with the tasks in the “ready” state assigned to that CPU. Section 3.5.3 presents the API to work with this type of doubly-linked lists. The `cur_task` field is a pointer to the *current task*, namely, the one that is actually running on the CPU. Note that the **current task is not included in the task list.**

Each *run queue* has its own idle task associated with it (`idle_task` field). This special task will be selected to run only in the event that there is no other task in the *ready* state on that CPU (i.e., when the task list is empty). The `need_resched` flag must be set to `TRUE` when a given scheduling algorithm detects that a preemption must be performed: a new task must be selected to run on the current CPU. The `nr_runnable` field is the number of runnable task on the CPU; this count includes both the running task as well as tasks in the *ready* state found in the task list associated with that CPU.

`next_load_balancing` indicates the future simulation step in which the periodic load balancer will be invoked. The `rq_cs_data` is a generic pointer (`void*`) enabling the active scheduling algorithm to keep a pointer to a structure of its choice. This field can be used to add more fields (in an indirect fashion) to the run queue as need by a given scheduling algorithm. Note, however that none of the scheduling algorithms implemented in the provided

source code uses this field. As such `rq_cs_data` is set to `NULL`. Lastly, the `lock` field is the lock associated with the *run queue*. This lock must be held when accessing fields in the *run queue*.

Another relevant data type for scheduling-algorithm implementers is the structure describing a task:<sup>3</sup>

```
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;
```

Many of the fields in this structure are employed to maintain statistics in the simulator; these fields are largely self-explanatory. Moreover, most of the fields in the structure are used only in the core scheduler code (`sched.c`) which is fully implemented already. The most relevant fields in the context of this lab assignment are as follows:

- `prio` is the task's priority, which can be used to determine which task must be selected next to run on a CPU.
- `runnable_ticks_left` indicates how many *ticks* are left to complete the task's current CPU burst. The initialization of this field can be found in the `sched.c` file. When implementing a new scheduling algorithm, this field must be accessed to determine if the currently running task will leave the CPU at that *tick*. If that is the case, the `nr_runnable` field in the associated *run queue* must be decremented (for more information, have a look at the implementation of the RR and SJF algorithms). Furthermore, in some algorithms such as SJF, the `runnable_ticks_left` field is used to decide which task runs next.
- `on_rq` is a boolean that must be set to `TRUE` when the task is in the *run queue* (i.e., the task is in the *ready* state).
- `flags` can be used to annotate any exceptional situation for a task. Typically, this field is used to simplify the implementation of scheduling algorithms. Currently, just the following two *flags* (bitmasks) are defined (`sched.h`):

---

<sup>3</sup>In this lab assignment we will always use the term *task* to refer to the schedulable entities that will be simulated. By contrast, we will use the term *thread* to refer to the actual (per-CPU) simulation threads. The `task_t` data type resembles a *Process Control Block (PCB)* in a modern OS.

- `TF_IDLE_TASK` enables us to know if a task is actually an idle task.
- `TF_INSERT_FRONT` is used by the SJF policy in several cases to indicate that the task must be inserted at the beginning of the task list rather than at the end.
- `tcs_data` is a generic pointer (`void*`) enabling the active scheduling algorithm to point to a private data structure. Specifically, the private structure is used in the event the scheduling algorithm wants to define new task-specific fields. In any case, the definition and initialization of that private data structure is up to the implementation of the algorithm in question. In the current version of the simulator, only the RR algorithm uses this feature (see file `sched_rr.c`); for the remaining scheduling algorithms this field is set to `NULL`.

### 3.5.3 Doubly-linked lists

In the `slist.h` file, the `slist_t` type is defined. This data type represents a generic implementation of a doubly-linked list. As stated earlier, a *run queue includes, among other things, a list of tasks in the “ready” state*. The implementation of a scheduling algorithm must deal with the task list, which contains elements of the `task_t` type. The set of operations enabling to manipulate that doubly-linked list is as follows:

```
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*));
```

Again, most of the functions are self-explanatory. As such, we only describe the most relevant ones:

- `head_slist()` returns the first element of the list (but it does not remove it from the list).
- `tail_slist()` returns the last element of the list (but it does not remove it from the list).
- `remove_slist()` removes the `elem` item from the list.
- `insert_slist()` inserts the `elem` item at the end of the list
- `sorted_insert_slist()` inserts the `elem` item into a sorted list. In order to perform the insertion, a comparison function must be passed as the third parameter. An example of invocation of the `sorted_insert_slist()` function can be found in the `sched_sjf.c` file.

### 3.5.4 Implementing a new scheduling algorithm

We are now ready to implement a new scheduling algorithm. This entails implementing the operations of the scheduling class's interface (`sched_class_t` type defined in `sched.h`). This interface consists of the following operations:

```
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, int cpu);
    void (*enqueue_task)(task_t* t, int cpu, int runnable);
    void (*task_tick)(runqueue_t* rq, int cpu);
    task_t* (*steal_task)(runqueue_t* rq, int cpu);
} sched_class_t;
```

The last four operations in the interface must be implemented in all scheduling algorithms. The first four operations are implemented in some cases, as explained later. We now describe the set of actions that must be performed by each operation.

- `sched_init()` and `sched_destroy()` enable to initialize a scheduler and free up the scheduler-specific resources respectively. These operations are invoked once during the simulation, and must be implemented just in the event that the scheduler stores extra information in per-CPU *run queues*, (by using the private field `tcs_data`) or uses other private structures not included in the run queue. If that is the case, the necessary resources must be allocated and initialized in `sched_init()` function (including the `tcs_data` field of the run queue) and will be then freed up in `sched_destroy()`. Currently, none of the implemented scheduling algorithms uses these functions. Nevertheless, students may use this feature in the extra part of the assignment.
- `task_new()` and `task_free()` are invoked when a task is created and when destroyed, respectively. These operations must be implemented if the scheduling algorithm defines and maintains extra task-specific fields by means of the `tcs_data` field in the task structure. Note that the RR scheduler is the only algorithm in the simulator that uses this field. So, for each task it allocates a new data structure in `task_new()`; the pointer to that data structure is stored in the `tcs_data` field. This data structure is then freed up in the associated `task_free()` operation. (Have a look at the `sched_rr.c` file).
- `pick_next_task()` will be invoked to select a new task to run on a CPU. This function will pick one task from the task list of the *run queue*, will then remove the selected task from the list and will return the selected task. Obviously, the mechanism to select a task is specific to each scheduling algorithm.
- `enqueue_task()` gets invoked every time that a task must be enqueued on a given run queue. Three situations may trigger the invocation of this function (1) a task enters the system, (2) a task awakes upon completion of an I/O operation and (3) the task was migrated onto this CPU from a different CPU.
- `task_tick()` gets invoked every *tick* of the simulation (each iteration of the main loop). When the currently running task on the CPU is about to complete its current CPU burst (one tick left), this function decrements the number of runnable tasks in the run queue (`nr_runnable` field). In some scheduling policies, such as RR, the `task_tick()` operation also updates task-specific fields to aid in deciding when to preempt the currently running task.

- `steal_task()` will be invoked when the load balancer needs to *steal* a task from this CPU's run queue (passed as a parameter). This function is responsible for selecting the most suitable task to be migrated to another CPU. As such the function removes the selected task from the run queue and returns the selected task, so that the load balancer can complete the migration.

Implementing a new scheduling algorithm boils down to implementing the necessary subset of operations of the `sched_class_t` interface in a new separate `.c` file. By convention, the name of the function that implements an operation of the interface contains a suffix with the name of the scheduling algorithm (lowercase version) prepended by “\_”. So, for example, the `pick_next_task_rr()` function implements the `pick_next_task` operation of the RR scheduler.

Once the operations have been implemented, the relationship between each operation and the associated function must be stated in the same `.c` file where the functions are defined. To this end, a global variable with type `sched_class_t` must be defined and initialized accordingly. For example, the `rr_sched` variable defined in `sched_rr.c`, instantiates the interface of operations for the RR scheduler as follows:

```
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
};
```

To finish off with the definition of the new scheduling algorithm, we must add a new entry in the `available_schedulers` array (defined in `sched.h`) as well as in the enumerate type defined right before that structure. The current definition of the array is shown below:

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

To register a new scheduler in the simulator, we must add a new line (array entry) indicating the name of the variable of type `sched_class_t` defined in the newly created `.c` file with the scheduler's implementation.

In the previous section, the various fields in the `runqueue_t` and `task_t` data structures were described. Notably, in implementing a new scheduling algorithm, just a few fields in both structures must be modified: `rq->tasks`, `rq->need_resched`, `rq->nr_runnable`, `tsk->on_rq` and `tsk->flags`. We should also highlight that the lock of the run queue is not on the list. Specifically, when any operation of the scheduling algorithm is invoked, the caller function has already acquired this lock if necessary. As such, the developer does not have to worry about locks when implementing a scheduling algorithm in the simulator.

### 3.6 Mandatory part

As the mandatory part of the assignment, the scheduling simulator must be extended in the following ways:

- Create the FCFS (*First Come First Served*) scheduler. When the *stealing* operation of the scheduling algorithm gets invoked, by convention, FCFS will pick the last task in the *run*



*queue*. The implementation of this algorithm must be included in a new `sched_fcfs.c` file. The file must define a `sched_class_t` structure as well as the necessary functions for the FCFS scheduling policy.

- Create a *preemptive* priority-based scheduler (PRIO). This scheduler will make decisions based on each task's static priority value defined in the input file passed as a command-line argument to the simulator (`-i` option). Students are strongly encouraged to pay special attention to the implementation of the SJF<sup>4</sup> policy, because it is somewhat similar to that of the priority-based scheduler. In the event the *stealing* operation of the scheduling algorithm gets invoked, this scheduler will select the lowest priority task found in the *run queue*. The implementation of this algorithm must be included in a new `sched_prio.c` file. The file must define a `sched_class_t` structure as well as the necessary functions for the PRIO scheduling policy.
- Implement a synchronization barrier by using locks and condition variables. Complete the barrier implementation in the `barrier.c` file (functions `sys_barrier_init()`, `sys_barrier_destroy()` and `sys_barrier_wait()` of the `#else` code path). Activating this barrier in the scheduler entails modifying the provided *Makefile* so as to ensure that the `POSIX_BARRIER` preprocessor symbol is not defined.
- Create a shell *script* that does not accept arguments, but instead asks the user to enter the following information interactively:
  - *The input task file to be simulated*. The script will verify that the file exists and is regular. If that is not the case, the script will display an error message and will ask the user to enter a file name again.
  - *Maximum number of CPUs to use during the simulation*. This number should not be greater than 8. If the number is greater than 8, the script will display an error message and will ask the user to enter a new CPU count.

Then, a `results` directory will be created and the simulator will be executed for each available scheduling algorithm and for each CPU count from 1 to the maximum CPU count provided by the user. The script will ensure that all simulation results are safely backed up (without overwriting files) in the `results` directory. Finally, for each file associated with the various simulations, a different chart will be generated. The charts will be stored in the `results` directory as well.

A possible pseudocode representation of the core processing of the *script* is as follows:

```
maxCPUs = value provided by the user

foreach nameSched in listOfAvailableSchedulers
do
  for cpus = 1 to maxCPUs
  do
    ./sched-sim -n cpus -i .....
    for i=1 to cpus
    do
      move CPU_${i}.log to results/nameSched-CPU-${i}.log
    done
    generate chart
  done
done
```

---

<sup>4</sup>Note that the `preemptive_scheduler` global variable used in the implementation of SJF is set to one only in the event that the `-p` option is specified in the command line when launching the simulator.