quand on a besoin de workload: faudra mettre dans simulation.c sauf si sjf

INFO0940-1: Operating Systems (Groups of 2)
Project 1: CPU Scheduler Simulator

Prof. L. Mathy - B. Knott - G. Gain

Submission before Friday, April 13, 2024 (23:59:59)

1 Introduction

The CPU scheduler is the part of the operating system that decides which process to execute next. It is a key component that has a significant impact on the overall system performance.

Programming a real CPU scheduler is a complex task. In this project, you will not implement a real CPU scheduler but a simplified simulator. The simulator will read a file containing a list of processes and simulate their execution according to different scheduling algorithms.

The goal of this project is to understand the basic concepts of CPU scheduling and to learn how the different resources interact with each other in an operating system. Among these, we can mention:

- The scheduler, which is the part of the operating system that decides which process to execute next.
- The CPU, which is the part of the computer that executes the instructions of a program.
- The disk, as process execution consists of a cycle of CPU execution and I/O wait.

2 The CPU scheduler simulator: Launch options, behavior and system assumptions

This section describes the particularities of the simulator, the CPU scheduler and the system it is running on.

Note that the simulator uses time units to represent time. The time unit serves to depict the progression of the simulation but may not always align with real-time. For instance, while the duration of a context switch is typically insignificant compared to the total duration of a process, we intentionally extended it in our simulation to facilitate clear visualization on a graph.

2.1 Launch options

Input file

The simulator will read a file containing a list of processes. The file will contain one process per line. Each line will contain the following information:

- The process ID (PID), which is a unique identifier for the process.
- The start time, which is the time at which the process arrives in the system.
- The duration, which is the time the process needs to execute on the CPU **and on the disk**.
- The priority, which is the priority of the process. The lower the number, the higher the priority.
- The list of CPU and I/O events of the process. A timestamp is given for each event, indicating when the event occurs with respect to the start time of the process.

A minimal input file would look like this:

```
# pid, start_time, duration, priority, [list of timestamps and events] (IO, CPU)
1, 0, 20, 1, [(0, CPU), (10, IO), (12, CPU)]
```

Here, the process with PID 1 arrives at time 0 and has priority 1. This process starts on the CPU (as any process does) and then goes to the disk after 10 time units of execution. The disk operation takes 2 time units. Then, the process goes back to the CPU for 8 more time units.

Launch options

The simulator can be launched with the following options:

```
Usage: ./cpuScheduler INPUT_FILE -c NB_CORES -q NB_READY_QUEUES \
                  ALGORITHM_OPTIONS_QUEUE_0 [ALGORITHM_OPTIONS_QUEUE_1] [...]
ALGORITHM_OPTIONS: --algorithm=ALGORITHM
                [--RRSlice=RRSLICE_LIMIT]
                [--limit=TIME_LIMIT]
                [--age=AGE_LIMIT]
```

- `INPUT_FILE` is the file containing the list of processes described above.
- `-c NB_CORES` is the number of cores of the CPU.
- `-q NB_READY_QUEUES` is the number of ready queues.
- `ALGORITHM_OPTIONS_QUEUE_N` is the scheduling algorithm and its options for the `N`-th queue. These options are:
  - `--algorithm=ALGORITHM` is the scheduling algorithm to use. The possible values for are `FCFS`, `SJF`, `RR` or `PRIORITY`.
  - `--RRSlice=RRSLICE_LIMIT` is the time slice for the Round-Robin algorithm. This option is mandatory if the algorithm is Round-Robin and has no effect otherwise.
  - `--limit=TIME_LIMIT` is the execution time limit on this queue.
  - `--age=AGE_LIMIT` is the age limit on this queue.

The order of the parameters is important, and only parameters surrounded by brackets are optional. Here is an example of a launch command:

```
./cpuScheduler input.txt -c 2 -q 2 --algorithm=FCFS --limit=20 \
                          --algorithm=RR --RRSlice=5 --age=30
```

This command will launch the simulator with the input file `input.txt`, 2 cores and 2 ready queues. The first queue will use the First-Come-First-Served algorithm with an execution time limit of 20 time units. The second queue will use the Round-Robin algorithm with a time slice of 5 time units and an age limit of 30 time units.

2.2 Scheduling Features

+ mettre static

Scheduling algorithms

Examples for most of what is talked about in this section can be found in the appendix.

The simulator will use different scheduling algorithms to decide which process to execute next. The possible algorithms are:

SchedulingAlgorithmType + SchedulingAlgorithm_t

- **First-Come-First-Served (FCFS)**: The process that arrives first in the ready queue is executed first.
- **Shortest-Job-First (SJF)**: The process with the shortest execution time is executed first.
- **Round-Robin (RR)**: Each process is executed for a time slice, and then goes to the end of the queue.
- **PRIORITY**: The process with the highest priority is executed first.

=> si = regarder au pid

Please be careful that these algorithms only apply to the ready queue(s) of the CPU and apply to the next CPU burst of each process (not the whole process' execution). In particular:

– With **FCFS**, the process that arrives first in the ready queue is executed first, but if this process goes to the disk, it will be interrupted and the next process in the queue will be executed. When the initial process comes back from the disk, it will be put at the end of the queue.
– With **SJF**, it is not the process with the *total* shortest execution time that is executed first, but the process with the shortest remaining execution time of its *current CPU burst* (either until the next IO event or until the end of its execution).

Also note that **PRIORITY** and **SJF** are preemptive algorithms. If a process arrives on the ready queue with a higher priority (resp. a shorter remaining execution time), the current process will be interrupted and the new process will be executed.

Concerning the **RR** algorithm, if a process has finished its time slice but no other process is ready to be executed, the process will start a new time slice without passing through the ready queue. If on the other hand, a process is preempted by another process at the middle of its time slice (this can happen when there is multiple queues, see next Section), when the process is put back on the CPU, it will start a new time slice from 0 (it does not try to finish its previous time slice).

Multilevel feedback queue scheduling reutiliser ceux d'au-dessus

The simulator will use a multilevel feedback queue scheduling to manage the ready queues (except if the `-q 1` option is used, i.e. there is only one ready queue). This means that multiple queues will be used to hold the ready processes. Each queue has a different priority and a different scheduling algorithm.

In our simulator, every process starts on queue 0. Queue 0 is the queue which has the highest priority, then the queue 1, and so on. For a process to move to the next queue, it must have been executing in the current queue for a certain amount of time (the `--limit` argument). To avoid starvation, a process that has been waiting for a certain amount of time[1] in the current queue will be moved to the next queue (the `--age` argument).

previous

Multicore

The simulator will use a multicore CPU. This means that the CPU will be able to execute multiple processes at the same time. The number of cores is given as an argument to the simulator.

Context Switches

Lastly, the simulator will also represent context switches. In an operating system, a context switch is the process of storing and restoring the state of a process into its process control block (PCB) so that execution can be resumed from the same state at a later time. In our simulator, the switch in time and the switch out time are represented with macros. By default, the switch in time is 1 time unit and the switch out time is 2 time units.

Whenever a process starts executing on a core, there is a switch in time. Even if it is the first time the process is executed, the switch in time is used to represent the time it takes to "create" the process (including the creation of the PCB).

However, when a process has finished its execution, there is no switch out time. The process is simply removed from the core.

2.3 Disk Management

wait queue du scheduler + queue du disk

Examples for most of what is talked about in this section can be found in the appendix.

As previously mentioned, the process execution consists of a cycle of CPU execution and IO wait. To mimic this behavior, the simulator will use a disk to manage the IO events of the processes. No other IO devices are used in this simulator.

The scheduler will maintain an IO wait queue with the processes that are waiting for the completion of their IO operations. When the disk is ready to perform a new IO operation, the IO operation of the first process of the wait queue should be handled by the disk. In a real system, it is not be the

[1] It is the `accumulated` waiting time in the queue that is considered. If the process starts executing for a moment and then goes back to the queue, the waiting time is not reset.

1) en fct des evenements les preparer pour l'etape 2, si on voit qu'il faudra mettre sur CPU ca sera a l'etape 2 qu'il le fera
2) assign: mettre sur le core et disk : scheduler qui decide si il met sur le CPU (simplification pour le projet)

scheduler's job to put the IO operation on the disk, but only to maintain the wait queue. However, for simplicity, in our simulator the scheduler is allowed to directly manipulate the disk structure (as well as the CPU structure).

Once the IO operation is complete, the disk controller needs to send an interrupt to the CPU to inform it that the IO operation has complete. The CPU will then execute the interrupt handler. And finally, the process that was waiting for the IO operation to complete will be put back on the ready queue. -> handler deja coder dans computer.c, faut juste l'appeler au bon moment

The interrupt handler will use some execution time on the CPU, therefore the process that was executing on the CPU should not continue its execution during the interrupt handling.

2.4 System Assumptions

The system running the CPU scheduler is simplified. In particular, the following assumptions are made:

- The system has a single disk.
- The system uses DMA (Direct Memory Access) to handle IO operations. This means that the CPU does not need to be involved in the IO operations, except for the interrupt handling at the end of the IO operation.
- All IO operations are blocking among themselves, meaning that only one IO operation can be handled by the disk at a given time.
- The interrupt related to IO operation is the only interrupt that the CPU will handle.
- There cannot be more than one interrupt at a time (which is guaranteed thanks to the four previous conditions).
- Processes that trigger an IO operation will always wait for the IO operation to complete before they can continue their execution.
- The system has a single set of ready queues that it uses for all the cores.

3 Stats and graph

In addition to programming the CPU scheduler, you are asked to compute some statistics and make a graph of the execution of the processes. The graph will show the state of the processes and of the disk at each time unit. Examples are in given in a file named `graph_stats_example.c`.
The states[2] of a process are:

- **READY**: The process is ready to be executed, represented by a dash (`-`).
- **RUNNING**: The process is currently being executed, represented by a digit indicating on which core it is running (`0` to `9`[3]). = ProcessState
- **WAITING**: The process is waiting for an IO operation to complete, represented by a dot (`.`).
- **TERMINATED**: The process has finished its execution, also represented by a blank space ( ).

The graph is mainly used for representation purposes, therefore some freedom has been taken:

- When there is a context switch, the state of the process is represented by a dash (`-`), although the process is not really in the **READY** state at this time.
- When an interrupt handler is running on a core that was executing a process, the state of the process is, once again, represented by a dash (`-`), even though the process is not really in the **READY** state at this time.
- When an IO operation is being handled by the disk, the process' ID is being represented on the disk line. This does not mean that the process is on the disk, but that the disk is handling the IO operation of this process.

[2] There exists also the NEW state. However as a new process is directly put in a ready queue, we omit this state.
[3] If there are more than 10 cores, letters will be used for labeling. This is automatically managed when displaying the graph.

The states of the disk are either idle (represented by a blank space) or busy (represented by a digit representing the PID of the process which initiated the IO operation)[4].

The stats that you are asked to compute are:

- The arrival time (the first time unit where the process is in the system);
- The finish time (the first time unit where the process is terminated);
- The turnaround time; = amount of time to execute a particular process
- The CPU time (the time the process has spent on the CPU); = ProcessStats_t
- The waiting time (the time the process has spent in the *ready* queue);
- The mean response time (the average time it takes for a process to be assigned to a core from the moment it is ready);
- The number of context switches;

4 Skeleton

A program skeleton is provided and already contains some functions and structures that you will need to use. Your task is mainly to add the logic of the main simulation loop, which actually simulates the CPU scheduler in action. The structures and functions that are provided may need some additional fields or logic.

Let us first explain the structure of the skeleton.

Given files

The code consists of the following files:

- `main.c` is the main program. It parses the arguments, initializes some structures, launches the simulation, and finally, it prints the stats and graph.
- `simulation.{h|c}` contains the Workload structure, as well as the simulation code. a modifier
  The simulation is responsible for: main loop
  - initialising and deleting the different computation resources (scheduler, disk and CPU);
  - running the main simulation loop, where it will call the different computation resources to get some job done (for example ask the scheduler which is the next process to be executed);

  To do that, it will need to keep track of the different processes creation time, the processes' event times (CPU or IO) and the processes deletion time. The Workload structure is used for this purpose and is initialised by reading the input file.
- `schedulingLogic.{h|c}` contains the Scheduler structure, as well as the scheduling logic code. a modifier
- `computer.{h|c}` contains the CPU and Disk structures and code, as well as a Computer structure that contains the Scheduler, CPU and Disk. (a modifier)
- `schedulingAlgorithms.{h|c}` contains the SchedulingAlgorithm structure which holds different scheduling algorithms informations. = info sur quel info a utiliser: a ne pas modifier
- `process.h` contains the PCB (Process Control Block) structure.
- `graph.{h|c}` contains the Graph structure and code.
- `stats.{h|c}` contains the Stats structure and code.
- `utils.{h|c}` contains some utility functions and structures that you may need to use.

Figure 1 shows the structure of the main skeleton files.

[4] same remark as footnote 2

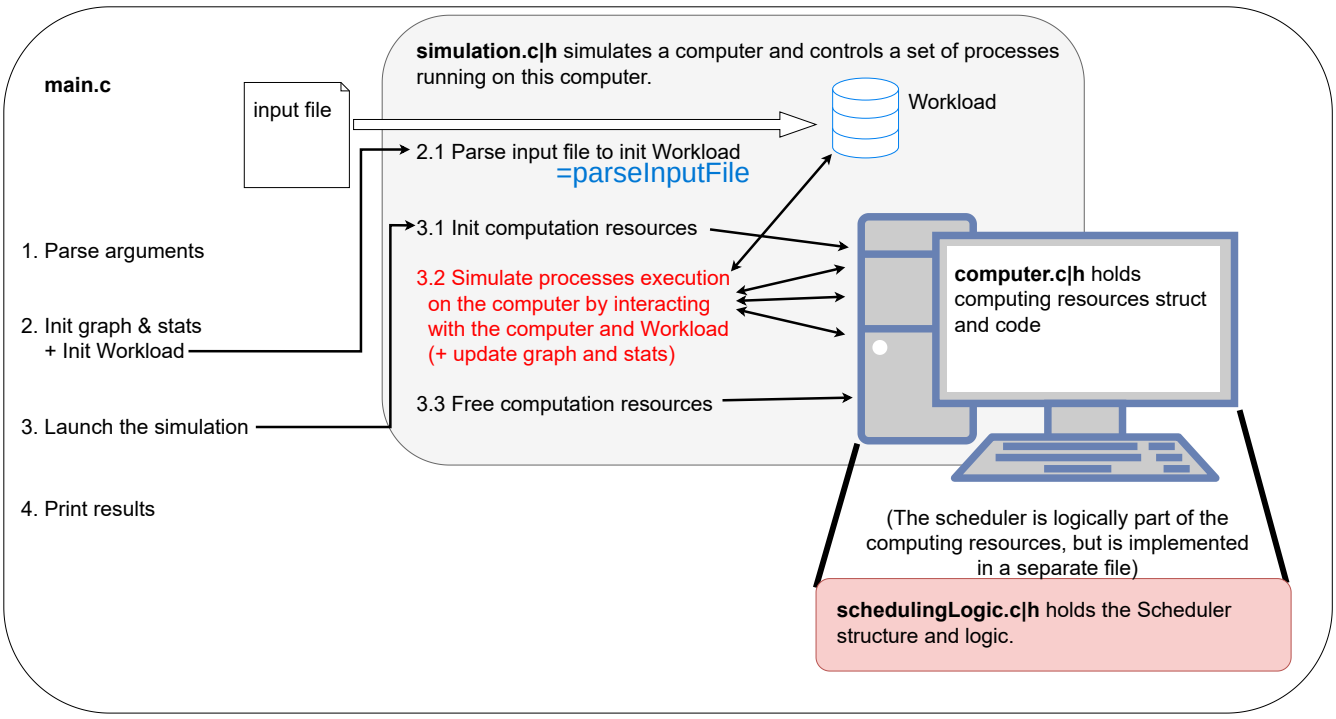

Figure 1: Skeleton structure

What is already given

Steps 1, 2, 3 and 4 from Figure 1 are already implemented for you. You should not have to modify these parts of the code. Parts of steps 3.1 and 3.3 as well as the CPU and disk structures are also given to you, but you will need to complete them if needed.

The implementation of these steps are scattered in the different files, and you should not have to modify them (even though you can[5]). Basically, you should only have to modify the `simulation.{h|c}`, `schedulingLogic.{h|c}` and `computer.{h|c}` files (and possibly the `utils.{h|c}` files).

You can consider our code inside the other C files as black boxes. You should not have to care about the implementation of the different functions that are already given to you and you should not take care of error cases that should be handled by our code. In particular:

- You can use functions provided in header files (such as `addCoreEventToGraph`) or getter functions (such as `getProcessDuration`) as much as you want, without having to care about the performance of these functions.
- Once the code has reached the main simulation loop, here are the assumptions that you can make about the different structures and variables available:
  - The graph, stats and workload structures are correctly initialised with the right number of processes and they can be used without any error. They will be freed automatically at the end of the simulation. The stats structure in particular has some default values that you can modify in `addAllProcessesToStats` if you want.
  - The elements of `algorithms` have been correctly initialised and all the values inside the structure are correct and make sense (aging is positive or equal to `NO_LIMIT` etc). It has been placed inside the scheduler structure and will be automatically freed inside `freeScheduler` (except if you modify it).
  - The Computer structure is correctly initialised and contains a pointer to the CPU, disk and scheduler. It will be freed automatically at the end of the simulation.

[5]But if you modify the format of the output, you will fail all our tests :)

- The CPU and disk are correctly initialised and will be freed automatically at the end of the simulation with the computer. They are initially idle.
- The minimal scheduler structure's fields are correctly initialised and will be freed automatically at the end of the simulation with the computer. You will have to add new fields, initialise them in `initScheduler` and free them in `freeScheduler`.

The already defined structures that you will have to manipulate are:

- The Workload structure in the `simulation.{h|c}` files, which contains the information of the input file as well as the processes advancement in the simulation.
- The ProcessGraph structure in the `graph.{h|c}` files.
- The AllStats structure in the `stats.{h|c}` files.
- The SchedulingAlgorithm structure in the `schedulingAlgorithms.{h|c}` files, which contain the informations of the arguments (except the number of cores and the input file).

Some getter and setter functions are provided and documented in the code for most structures. The `AllStats` and `SchedulingAlgorithm` struct fields can be accessed and modified directly.

What can/must be modified

Your main task will be to implement the simulation loop, which is represented by step 3.2 in Figure 1. The simulation loop will be in the `simulation.c` file, but needs to use different structures and functions located in other files. In particular, it should make many calls to the `schedulingLogic.{h|c}` files, therefore, most of your implementation will be in `schedulingLogic.c`.

You will also have to define the Scheduler structure in the `schedulingLogic.c` file (it should be opaque).You will have to complete the CPU and Disk structures in the `computer.{h|c}` files. As steps 3.1 and 3.3 depend on these structures, you will have to modify these steps also (which correspond to the init and free functions for these different structures).

As you can see, the code structure tries to isolate the different parts of the system, and you must respect this isolation in your code. The three main properties that you must respect are:

1. Everything related to scheduling that does not require knowing the future (i.e., the next events of the processes) should be in the `schedulingLogic.{h|c}` files.
2. When an interrupt is triggered, it is the CPU that should handle it. So it must be in the `computer.{h|c}` files.[6]
3. Everything else should be in the `simulation.{h|c}` files.

comment faire pour processArrived

There is only one exception to these properties: The **SJF** algorithm logic should be in `schedulingLogic.{h|c}` files, even though it needs to know the future (the next events of the processes).

You can also define some additional functions not related to the simulation in the `utils.{h|c}`.

Suggestion for the logic of the main loop

You are free to implement the main loop as you want, but a suggestion is given here.

The main loop would consist of 4 main steps, that would be executed for the current time unit:

1. Handle event(s): The simulator and the scheduler check if an event is triggered at the current time unit and handle it if so. For example, if a process arrives in the system, the simulator will call the scheduler to put the process in the ready queue. The events mentioned here could also be scheduling events, such as a process needing to move to an upper queue because of aging, or hardware events, such as the triggering of an interrupt.

+ fin d'interrupt

[6] It is enough to simply launch the "interrupt event" inside `computer.{h|c}`, for example with a timer. The scheduler code can then simply check if this timer is over and, if it indeed is over, deduct that the IO operation of the first process of the IO wait queue has completed.

2. Assign processes to resources: This is the step where the main scheduling decisions are made: choosing what processes to execute next. The scheduler will check if a process is ready to be executed and will choose what core it should put it on (or not). The scheduler could also put a process on the disk if it is idle.
3. Get next event time: Here, the simulator and scheduler will simply check what is the time unit of the next event in order to jump directly to it in the next step.
4. Advance time to the next event: This is where the progression of time is simulated. The simulator will update the advancement of the processes in the workload and the scheduler will update its timers. Then the current time can be updated too in order to deal with the next event at the next iteration of the loop.

An example of an execution of this loop can be found in the Appendix.

Note that it is possible to remove the third step (and possibly even merge step 2 and 4) if you want to by simply iterating over all time units. It might even be simpler, but be careful to do everything you need in every iteration. The advantage of the third step is that it could indicate that some events still need to be dealt with at the current time unit and deal with it in the next iteration.

You will also have to update the graph and stats in the main loop. Do not worry too much about performance, it is okay to update the graph at every iteration of the loop.

5 Additional requirements

The following requirements must be satisfied:

- Your code must respect the given skeleton.
- You may use static allocation only when appropriate, i.e. for small data structures whose length is known at compile time. For any other data structure, you **must** use dynamic allocation, i.e use `malloc()` and `free()` to manage memory. Use valgrind[7] to detect memory leaks.
- Your code must be readable. Use common naming conventions for variable names and comments.
- Your code must be robust and must not crash. In addition, errors handling must be managed in a clean way. Only fatal errors such as failed memory allocation will stop the simulator. Other errors must be displayed on `stderr`. As previously mentioned, some assumptions can be made on the input file and the arguments (see Section 4).

6 Joker

As the project touches many different concepts, you have the possibility to choose a joker (i.e., a part of the project that you do not want to implement). You have two joker choices:

- Implement everything except the multilevel feedback queue scheduling. (i.e. the scheduler will have only one ready queue and the aging and limit options will not be used)
- Implement everything except the multicore CPU. (i.e. the CPU will have only one core)

And if you choose to implement everything, you will get a bonus of 2 points on the final mark.
Be careful, it is **mandatory** to implement either the multilevel feedback queue scheduling or the multicore CPU, otherwise you will loose points. You cannot have two jokers.

Hint: It is a good idea to start by implementing your code (in particular your structures) in the most general way rather than, for example, implementing it only for one core and adding support for multiple cores later on.

[7] http://www.valgrind.org

7 Report

You are asked to write a very short report (**max 1 page**). A template of the report is provided on eCampus. The report should contain the following information:

- The names and the student IDs of the authors.
- A short overview of your structure/code. For instance, if you did not use the suggested structure of the main loop, a description of the structure you used.
- What you chose for the joker (multilevel feedback queue scheduling, multicore CPU or none of them).
- Answers to the following questions:
  - Assuming that there is no context switch time (set to 0) and all processes are equally important, what do you think is the best scheduling algorithm among the four proposed? Why?
  - Do you think this algorithm is implemented in most operating systems? Why?
- A feedback on the assignment. In particular:
  - Was it too easy or too difficult compared to other projects?
  - Was it too short or too long compared to other projects?
  - If you have some remarks (positive or negative), you are free to write them.

8 Evaluation and tests

Your program can be tested on the submission platform. A set of automatic tests will allow you to check if your program satisfies the requirements. Depending on the tests, a **temporary** mark will be attributed to your work. Note that this mark does not represent the final mark. Indeed, another criteria such as the structure of your code, the memory management, the respect of the defined skeleton, the correctness and your report will also be considered. You are however **reminded** that the platform is a **submission** platform, not a test platform.

9 Submission

Projects must be submitted before the deadline. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^N - 1$ marks (where N is the number of started days after the deadline).

Your submission will include your code (all the required {`.c`|`.h`} files in a directory named `src`) along with a Makefile which produces the binary file `cpuScheduler`. A template of the Makefile is provided on eCampus. You are free to add new files and modify the Makefile, but it is probably a bad idea (see Section 4). Please also add your report (`.pdf`) in the `src` directory.

Submissions will be made as a `group_{your_group_ID}.tar.gz` archive (for example `group_3.tar.gz`), on the submission system. Failure to compile will result in an awarded mark of 0.
Bon travail...

Appendix A: Graph examples

In the following examples, the graph represents processes running on core X with an X (where X is the core id), processes waiting on the ready queue with "–" and processes waiting for IO completion with ".". The "switch in" lasts 1 time unit, the "switch out" lasts 2 time units and the interrupt handling lasts 1 time unit.

Context switch example

The following representations are only intended to understand scheduler concepts. Real context switches would not take so much time with respect to a process execution time. We use the following input file:

```
# pid, start_time, duration, priority, [list of timestamps and events] (IO, CPU)
1, 0, 10, 1, [(0, CPU)]
2, 0, 15, 3, [(0, CPU)]
3, 0,  5, 2, [(0, CPU)]
```

Listing 1: Example input file

```
Process    0    5   10   15   20   25   30   35
      1    -00000---------------00000
      2    --------00000---------0000000000
      3    ----------------00000
```

Listing 2: Context switch example using RR with time slice of 5. Green dashes represent switch in and red dashes represent switch out. Note that during the context switches, the process are in the ready state.

=/= en cours (photo):
en 0: il y a -
priority n'a rien a voir

Algorithm examples

The same input file as in the previous section can be used (Listing 1). In addition, all these examples have context switches.

```
Process    0    5   10   15   20   25   30
      1    -0000000000  cpu: 10 - waiting: 0
      2    ----------000000000000000      cpu: 15 - waiting:11
      3    ----------------------------00000  cpu: 5 - waiting: 27
```

Listing 3: Example with FCFS

```
Process    0    5   10   15   20   25   30
      1    ------0000000000
      2    ----------------000000000000000
      3    -00000
```

Listing 4: Example with SJF

```
Process    0    5   10   15   20   25   30   35
      1    -00000---------------00000
      2    --------00000---------0000000000
      3    ----------------00000
```

Listing 5: Example with RR and a time slice of 5

```
Process    0    5   10   15   20   25   30   35
      1    -00000---------------00000
      2    --------00000---------0000000000
      3    ----------------00000
```

Process 0 5 10 15 20 25 30
1 -0000000000
2 ----------000000000000000
3 ----------00000

Listing 6: Example with PRIORITY

IO example

This input file is now used:

```
# pid, start_time, duration, priority, [list of timestamps and events] (IO, CPU)
1, 0, 10, 1, [(0, CPU), (2, IO), (4, CPU)]
2, 0, 15, 3, [(0, CPU)]
3, 0,  5, 2, [(0, CPU)]
```

Listing 7: Example input file where process 1 has an IO at time 2 which lasts 2 time units

The two following examples will be clearer without context switch time, therefore they are temporarily set to 0.

avec CS

```
Process    0    5   10   15   20   25
      1    00...------------------000000   cpu: 8 - waiting: 18
      2    --00-0000000000000   cpu: 15 - waiting:2
      3    ------------------00000   cpu: 5 - waiting: 18

   Disk       11
```

```
Process   0   5   10   15   20   25   30   35
      1   -00...----------------------000000      cpu: 8 - waiting:22
      2   -------000000000000000                  cpu:15 - waiting: 6
      3   ----------------------00000             cpu:5 - wiating: 22

   Disk      11
```

Listing 8: Example with IO using FCFS and with no switch time. The disk handles the IO of process 1 at time 3 and 4. During this time, process 2 can start executing. The interrupt handler is on core 0 at time 4, thus process 1 is only put back at the back of the ready queue at time 5.

The same example but using SJF:

```
Process    0    5   10   15   20   25
      1    00...---000000
      2    --------------000000000000000
      3    --00-000

   Disk       11
```

Listing 9: Example with IO using SJF and with no switch time. The disk handles the IO of process 1 at time 3 and 4. During this time, process 3 can start executing. The interrupt handler is on core 0 at time 4, thus process 1 is only put back on the ready queue at time 5.

Multilevel feedback queue scheduling

For example, with the following launch command:

```
./cpuScheduler input.txt -c 1 -q 3 --algorithm=RR --RRSlice=2 --limit=4 \
                         --algorithm=RR --RRSlice=3 --limit=9 \
                         --algorithm=FCFS
```

And this input file:

```
1, 0, 25 , 1, [(0, CPU)]
2, 0, 15, 1, [(0, CPU)]
3, 22, 10, 1, [(0, CPU)]
```

The graph would look like this:

```
Process    0    5   10   15   20   25   30   35   40   45   50   55   60   65   70   75   80   85   90
      1    -00--------00--------0--------------000--------------000--------------00--------000000000000
      2    -------00--------00--------------000--------------000--------------000--------00
      3                                --0000--------------000--------------000
```

Listing 10: Example of multilevel feedback queue scheduling using RR for the two first ready queues and FCFS for the last. The two first queues have a maximum execution time limit. At time 22, process 1 is preempted by process 3 because it is on a lower queue. Once process 1 gets back on the CPU, it restarts a new time slice (it does not try to finish its previous time slice)

Multicore

For example, with the following launch command:

```
./cpuScheduler input.txt -c 2 -q 1 --algorithm=FCFS
```

using the example input file in Listing 1 would yield:

```
Process    0    5   10   15
      1    -0000000000
      2    -111111111111111
      3    ------------00000
```

cpu: 10 - waiting: 0
cpu: 15 - wiating: 0
cpu: 5 - waiting: 11

Listing 11: Example of multiple cores using FCFS

Appendix B: Example of an iteration of the main loop

Here is an example of an iteration of the main loop. This example could work for example with the **FCFS** algorithm with a single core and a single ready queue. The iteration is at time unit 10.

detect event
+ prepare for step 2

faire un tableau avec les process sur chaque core + val par defaut si aucun

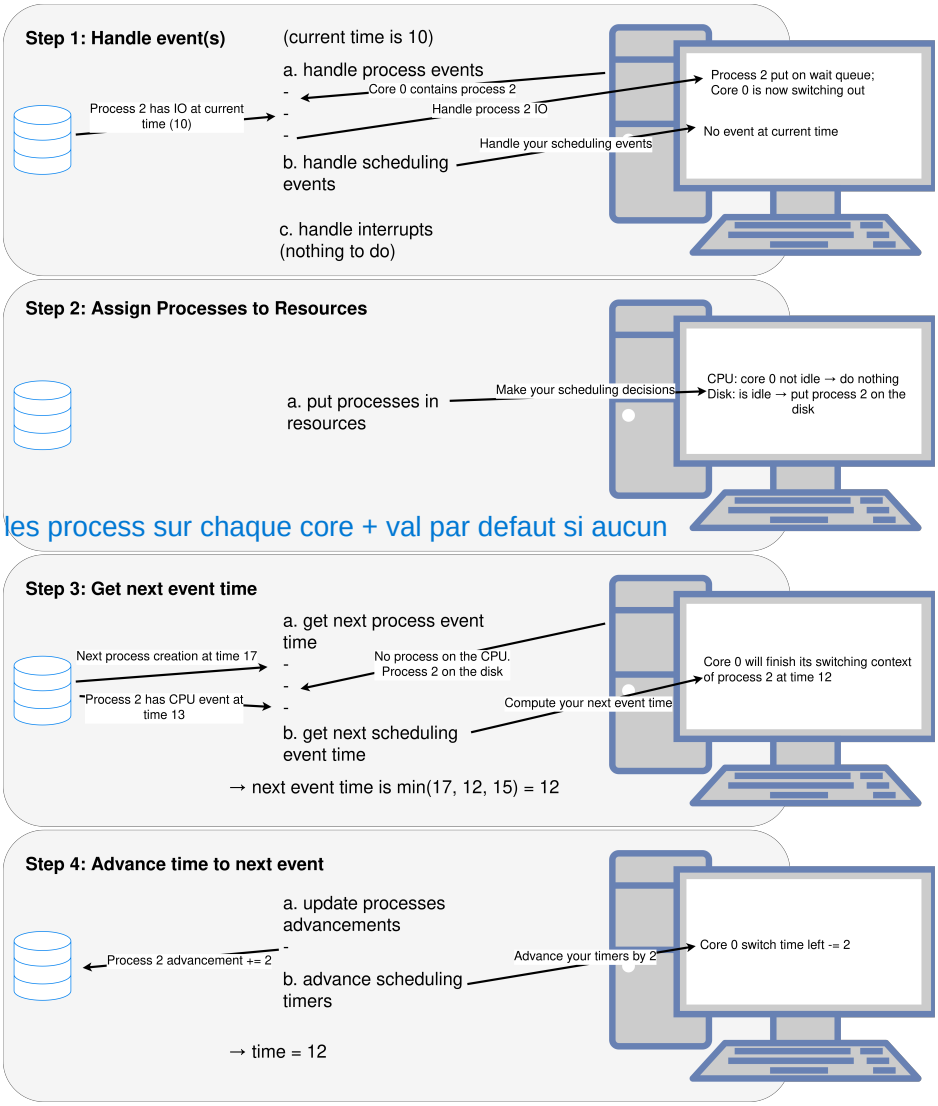

Figure 2: Example of an iteration of the main loop at time 10. A column running on core 0 has an IO event at time 10, the scheduler thus puts it on the disk and switch it out of the core. The switch out time is 2 time units and the end of the switch out will be the next event to handle at time 12 (at the next iteration).