Operating Systems

2024 - 2025

2nd Laboratory Exercise

Laboratory Exercise

Process Scheduler

**1. First Phase**

* 1. **Introduction**

You are asked to implement a scheduling environment on a Unix operating system. Specifically, a scheduler must be implemented that takes as input the processes it needs to execute by reading a file with their names, inserts them into an appropriate data structure (list) and then schedules them by applying one of the following policies: FCFS (First Come First Serve) and RR (Round Robin).

The implementation of the scheduler must be carried out in such a way that its basic data structures and internal functions are as compact and independent as possible of the policy it applies. For example, policy could change dynamically at execution time. This section includes a general description of the key elements of scheduler implementation that you should consider.

**1.2 Application description**

Each application to be run by the scheduler will be described with an appropriate data structure. Such a structure will be bound and initialized appropriately for each application read from the input file and to be executed by the scheduler. It will then be entered into the run queue. This structure will include fields that describe process attributes, such as executable file name, pid, execution status, time to enter the execution queue, etc.

Possible execution states of an application (process) are:

1. **NEW:** The application has just been queued for execution. Applications run through the scheduler, which creates the process to run an application (*with fork-exec*).
2. **RUNNING**: The process – process is active.
3. **STOPPED:** The process has stopped running.
4. **EXITED:** The process has terminated.

**1.3 Execution queue**

The data structure containing the process descriptors must be a double-linked list (queue). Inserting processes occur at the end of the list while deleting from its beginning.

**1.4 Process Scheduler**

The scheduler selects the next application to run each time, and if there is an application, it performs the appropriate actions otherwise it shuts down.

Initially, we believe that the applications to be launched include only calculations and no input-output commands.

Therefore, in the case of static scheduling policy (FCFS), the scheduler will be activated only when an application – process shuts down.

In contrast, in the case of dynamic routing policy (RR), the scheduler will be activated at regular intervals, as determined by the scheduler quantum that is a parameter of the program. The router activation mechanism will be implemented using an appropriate timer (e.g. using **the nanosleep** call).

When implementing the Round Robin policy, the execution of applications – processes, i.e. the suspension and continuation of their execution, will be managed using the **SIGSTOP** and **SIGCONT signals**.

Finally, when an application – process shuts down, the scheduler should be updated, having set an signal handler for the **SIGCHLD signal,** and by performing the necessary actions for its proper operation.

In any case, when an application shuts down, its total execution time should be calculated and printed, starting from the time it entered the run list until the time it finished running. For simplicity, all processes are entered into the system, i.e. in the execution queue, before the execution of the routing policy starts (i.e. at the corresponding time "0" according to the theory exercises).

**1.5 Useful hints**

1. The scheduler should receive the **SIGCHLD** signal only when a child application shuts down, not when the child's application is suspended (see **sigaction**)
2. It is useful for the scheduler to hold a pointer to a variable in the descriptor of the process that is running at a given time.
3. When terminating a process, the scheduler should display information to run the corresponding application. Alternatively, the information can be printed before the router is finished. The scheduler should also display the total runtime of all applications.
4. For the development of the scheduler, you can use simple test programs that allow to check its correct operation.
5. The code of the above test programs, characteristic input files, a script with examples of running the scheduler, an indicative example of printing results, as well as an indicative code of the scheduler, which you need to complete, are given along with the pronunciation of the exercise.
6. The FCFS routing policy can also be implemented as a sub-case of running the RR routing policy by setting the time quantum to a very large value.

**1.6. Use**

According to the above, the scheduler program will be used as follows:   
**scheduler <policy> [<quantum>] <input\_filename>,** where

1. **Scheduler:** The executable of the router that you will implement.
2. **Policy:** The routing policy by which applications will run. Possible values are FCFS, RR.
3. **quantum:** The routing quantum in msec. Required only for the RR routing policy.
4. **input\_filename:**  The name of the file that contains the workload to be performed through the router. Each line of the file includes the name of an application's executable.

Examples of use:

*$ scheduler FCFS homogeneous.txt*

*$ scheduler FCFS reverse.txt*

*$ scheduler RR 1000 homogeneous.txt*

*$ scheduler RR 1000 reverse.txt*

**2. Second Phase**

**2.1 Extention of Scheduler**

So far we have assumed that the processes to be launched do not include I/O instructions and therefore do not grant the processor until they terminate or expire the time quantum in which they are selected to run. This question asks you to extend your scheduler to take into account applications that are going to execute I/O commands. The simulation of the above case is done using appropriate signals. An application that is a in execution mode calls a **perform\_io MSEC** routine that does the following:

1. Informs the scheduler that it is about to start I/O by sending it the signal **SIGUSR1**

2. "He sleeps" (see **nanosleep**) for a certain period of time, as defined by the routine argument.

3. Sends the scheduler a second signal (**SIGUSR2**) informing it that it has completed the I/O and can continue its execution.

4. It sends itself the **SIGSTOP**  signal so that it is activated again by the scheduler, when it selects it to execute.

It is clear that the scheduler code must be extended appropriately so that in any case it performs the appropriate actions (e.g. scheduling another application, entering the queue of an application that has completed the I/O commands, etc.). The list of states of a process can also be expanded.

For this extension, the scheduler code will be implemented in the **scheduler\_io.c file.**The work5x2\_io **test program**, created by compiling the **work\_io.c**  code, uses the I/O emulation mechanism presented above. The **mixed.txt**  execution file also includes a workload that can be used to evaluate the correctness of your implementation.

Note: for this query, it is sufficient to implement only one of the two routing policies on the extended scheduler. If you implement both correctly, then an extra point will be given to the grading of the assignment.

The use of the extended router remains as before, according to the following examples:

*$ scheduler FCFS mixed.txt*

*$ scheduler RR 1000 mixed.txt*

**3. Summary and delivery of work**

**3.1 Requested**

You are asked to complete the requested schedulers, the executables of which you will name **scheduler**  **(**basic version) and **scheduler\_io** (extended version).

Analytically:

1. Implement the structures and functions of the basic timer.
2. Implement the FCFS scheduling policy.
3. Implement the RR scheduling policy.
4. Run and evaluate the above policies for router execution examples as defined in the **run.sh file**.
5. Implement the timer extension to handle instances of applications performing I/O as described
6. Use the **mixed.txt**  file to demonstrate the correct operation of the extended timer.
7. Write the report according to the instructions and present the results from your experiments.

You will be graded separately for all the above requests. If you haven't implemented all routing policies, make sure that your code prints the appropriate message for them and runs correctly for all other policies.

**Comments**:

1. Include in the zip file with your code, the structure (directories and files) of the original code that has been given to you. That is, it must contain the two directories, the corresponding Makefile files, the code of the utilities and the completed code of the two implementations of the router.
2. Don't include executable files in the zip file with your code.
3. You can make any changes to the incomplete code of the scheduling , but you will have to follow the interface (i.e. the execution parameters) defined in the pronunciation and used in the execution examples.
4. The code you deliver should be able to compile and execute scripts like run.sh. The first stage of evaluating the code of your work will be carried out in exactly this way.
5. Executables for utilities and scheduler can be created by running the make command in the terminal and while you are in the corresponding directory.
6. The runtime of utilities (load1, load2, ...) can be adjusted by setting an appropriate value to the DELAY constant, as shown in the Makefile.
7. An implementation of the task has been tested on Linux systems (see diogenis, falcon) without problems.
8. If you run your experiments on a system that is being used by multiple users simultaneously (e.g. diogenis), then the runtime of utilities is expected to vary each time, depending on the overall system load.

**Good luck!**