



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
 AND SCHOOL OF ENGINEERING  
 DEPARTAMENTO DE CIENCIA DE LA COMPUTACION

## IIC2333 — Operating Systems and Networks — 1/2024 Task 0

Delivery Date: Monday, April 8, 2024 at 9:00 p.m.  
 Composition: Task in pairs

### Goals

- Implement process life cycle management using *syscalls*.
- Establish a communication mechanism between processes using signals.
- Model the execution of an algorithm *scheduling* of processes.

### Part I: Processes

#### 1. Program functionality **runner**

You will need to create a program called **runner**, which must be able to execute multiple programs concurrently by creating processes, and control that these do not exceed a certain execution time. The tasks that your program must accomplish are:

1. Receive the arguments delivered to your program through the command line.
2. Read a file, which will contain instructions on executing a set of programs.
3. Run the set of external programs in parallel form by creating multiple processes. The number of child processes in memory cannot exceed a value *<amount>*, so, if reached, you must wait for one of the processes to finish to continue creating new processes.
4. Capture signals sent by the user and handle them.
5. Once a time is up *<max>* since the execution of **runner**, this must send a signal **SIGINT** to all running child processes. In the event that the child processes do not finish their execution after 10 seconds, the signal must be sent to them **SIGTERM**.
6. Collect statistics of the external programs executed and deliver them in a file in .csv.

#### 2. Execution

The main program will be executed through command lines with the following syntax:

```
./runner <input> <output> <amount> [<max>]
```

Where:

- *<input>* is the path of an input file, which has the list of all external programs to execute.

- <output> is the path of a text file with the execution statistics, in .csv, which should be written by your program<sup>1</sup>.
- <amount> corresponds to the maximum number of external programs that can be running concurrently.
- <max> is a parameter optional, which is an integer that indicates the maximum number of seconds that a program can take to run before it is terminated. If this parameter is not given, the programs should be considered to have unlimited time to execute.

For example, some executions could be the following:

```
./runner input.txt output.csv 5
./runner hello/input.txt output.csv 5 10
```

### 3. Input file (*input*)

The input file (*input*) will be a text file, which starts with a line with an integer *N*, which indicates the total number of instructions. Next, there are *N* lines of instructions where each one corresponds to one of the following alternatives:

1. External executables = These lines will contain the instructions of a program to be executed, separated by a space:

- Number of arguments
- Name of the executable (external program)
- Arguments for the executable

For example, the following is a valid line that invokes `/bin/python3` with 1 argument:

```
1 /bin/python3 hello.py
```

2. Command **wait all** = This line describes the special command `wait all`. This command should block<sup>2</sup> to the main program until the running processes have finished. The line is as follows:

```
- 1 wait all timeout
```

Where <timeout> is the maximum waiting time in seconds since the command is processed. If a process does not terminate after <timeout> seconds, the signal must be sent `SIGKILL` to the corresponding processes.

In case the time <max> is fulfilled before <timeout>, the program must follow the instructions described in point 5 of the section **Program functionality runner**.

The following is an example of *input* valid:

---

```
5
2 ./servir_cliente 'choripan' 5 2 python3
play_cuecas.py 15
- 1 wait_all 6
1 python3 compra_pan.py 0 ./
concretar_sale
```

---

<sup>1</sup>The output of your process has to be the file entered, otherwise it will not be counted as correct.

<sup>2</sup>So it should not process any more input during this period.

Note that:

- An executable can exist with the number of arguments equal to zero. The
- number of arguments indicated in each line will always be correct.

## 4. Output

The program must terminate after completing the execution of all instructions, or if it receives an interruption from the user. At the end, a file must be written.csv with the final results. This *output* must contain *N* lines, one for each external program indicated in the *input*, and where each line must follow strictly the following format:

---

executable_name	execution_time	status
-----------------	----------------	--------

---

Where:

- executable name is the name of the external program executed.
- execution time refers to the execution time of the external program. If it was interrupted, it is the time it managed to execute.
- **status** refers to the state of the process, which depends on the signals emitted. In case the process ends successfully, consider the *exit code*. For example, if a signal is emitted **SIGINT**, the state would be 2.

## Interruptions

In addition to running programs concurrently, the main program must be able to capture the signal SIGTSTP, the one generated by pressing Ctrl+Z. In the context of this task, upon receiving this signal the main program should not terminate, but must terminate the functioning of the currently instantiated children, that is, it must interrupt all external programs that are being executed concurrently at that moment. In the event that the child processes do not finish their execution after 10 seconds, they must be interrupted with the signal SIGKILL. After this, the main program must write the statistics to the output file and finish its execution. It is important to note that, naturally, this will affect both the time statistics as in the number of cases abnormal in the execution.

## Part II: *Scheduler*

### 1. *Schedules*

The scheduler that must be implemented is about a simulation which will be in charge of managing shifts on a single CPU of several processes grouped according to the P.I.D. of the process, whose parent is the OS. It is considered the P.I.D. of the SW as 0, so the P.I.D. of the processes start in 1. A variation of the LIFO scheme will be used<sup>3</sup> within each process group, and the general groups will be located according to the order of process creation.

Each group of processes is represented by an input line, and each cycle that is active is assigned certain work units (CPU time, *burst*) for your processes to run. The order in which the processes run within a group is the same as that of the input, that is, if a line indicates an action that requires work units, This action must complete all of its units before moving on to the next.

The order of execution of each group depends solely on when they reached the SW. Is he SW the one who assigns them the P.I.D. to each process and it assigns it according to its order of arrival. Several process groups will be delivered and each one will contain the information shown in the section **5.1 Input file (*input*)**.

### 2. Process

A process contains the following information:

- PID, the unique ID of each process, are assigned by the SW in order of process creation (when the IC).
- PPID, the ID of the process that created the current process. If there is no parent process, it is said to have been created by SW, so the value of the PPID used in which case is 0.
- GID, the ID of the process group. If he PPID of the process is 0, the is used P.I.D. of process. Otherwise, the GID of the parent of the process.
- State, What can be READY, RUNNING, WAITING, FINISHED.
- Plus, he could have a son who is being executed.

### 3. Flow

#### 3.1. Work of a process

Be  $q$  the amount of work units that a group currently owns. A process in the group executes the minimum of  $q$  and what is left to work on. In case the process has been executed for a different number of 0, the status of this must be changed to RUNNING and report the RUN from the process to SW. Note that if the entire value of  $q$  available, the group to which the process belongs continues using the work units remaining.

#### 3.2. Considerations

It will be understood that a group You can work if you need work units to continue with the current step. Otherwise, the group is considered can't work.

Algorithm:

Be  $q_0$  the work units who assigned him the SW to a group and  $q$  the amount of work units that is left to the group, assigning  $q = q_0$ , the following operations are done in *loop* while  $q > 0$  OR NOT You can work the group.

1. It is checked if the group has not finished all its processes. In case all its processes have finished, it terminates the *loop*.

---

<sup>3</sup>Last in, first out

2. If the group does not work:

- If the process finished running, then this process is added to the list of completed processes. You
- advance to the next step.

Otherwise, it works and the SW time worked advances.

3. Finally it is added to the output the information on the changes that occurred in this iteration of the loop.

Once the loop is finished, the  $q$  of the group as follows:

$$q_{Yo+Yo} := \max\{q_{Yo} - q_{delta}, q_{min}\}$$

with  $q_{Yo}$  working time  $q$  that the group had before starting the loop, and  $q_{delta}$ ,  $q_{min}$  parameters present in the input file

### 3.3. SO Flow

The Scheduler simulation does the following loop to simulate SW:

1. Check the groups you have:

- If there are groups that have not yet started the execution and that should have started according to their  $Y_{OU}$ , he SW reviews which ones you can add and adds them to the system based on your  $Y_{OU}$ <sup>4</sup>. Then, in case there are no groups currently active, the SW enters state IDLE and moves forward until the next group arrives.

2. The system iterates through its active groups (those who have been cared for SW and they are not finished yet) and assigns the work units  $q_{Yo}$  to each group and making them work in order of arrival. In the event that a group ends, that group leaves the set of active groups and is discarded.

3. If all the groups have already entered and finished, the simulation is finished.

It is worth mentioning that the time of SW starts in 0.

## 4. Simulation Execution

The simulator will be executed via the command line with the following instruction:

```
./schedulesly <file> <output>
```

And these elements represent the following values:

- <file> will correspond to a file name that should be read as *input*.
- <output> will correspond to the path of a file TXT with the simulation statistics, which must be written by your process.<sup>5</sup>

## 5. Files

### 5.1. Input file (*input*)

The simulation data is delivered as input in a text file, where the first line indicates the quantity  $K$  which corresponds to the lines that follow. The next line indicates three values:  $q_{start}$  the work units that he SW initially assigned to each process group,  $q_{delta}$  the value of how much the work unit of the group each iteration of the SW by all groups, and  $q_{min}$  which is the minimum number of work units that can be assigned to a group in each iteration. The following  $K-1$  lines follow the following format:

---

<sup>4</sup>As a tiebreaker, it uses which line of the input they appeared on, giving preference to the smaller lines.

<sup>5</sup>The output of your process has to be the file entered, otherwise it will not be counted as correct.

---

TI CI NH CI\_1 NH\_1 ... CF\_1 CE CI\_2 NH\_2 ... CF\_2 CE CI\_3 NH\_3 ... CF\_3 ... CF

---

The format reads as follows (parentheses are added to separate what is not about the initial process)

---

TI CI NH (CI\_1 NH\_1 ... CF\_1) CE (CI\_2 NH\_2 ... CF\_2) CE (CI\_3 NH\_3 ... CF\_3) ... CF

---

Where:

- $Y_{OU}$  is the time in seconds of the group's arrival at the queue. Consider that  $Y_{OU} \geq 0$ . Note that this value only appears once per line.
- $I_C$  is the amount of time in seconds that the process will run before possibly creating your child. Consider  $I_C \geq 1$ .
- $N.H.$  is the number of processes the process will create.
- $E_C$  is the amount of time in seconds that the process will run between child creations. Note that this value will appear between each pair of children and may not always be the same. Consider  $E_C \geq 1$ .
- $C.F.$  is the amount of time in seconds that the process will run after creating all of its children. If the process has children,  $C.F. \geq 1$ , and if it has no children the process does not execute after creating children so  $C.F. = 0$ .

You can use the following assumptions:

- Each number is a non-negative integer that does not exceed the maximum value of a int32 bit.
- There will be at least one process described in the file.

The following example illustrates what a possible input file would look like:

---

```
3
30 10 12
3 5 2 4 0 0 4 20 0 0 8
7 10 1 10 1 10 1 10 0 0 3 3 3
```

---

Example group input line

---

7 8 3 4 0 0 1 5 0 0 2 6 0 0 9

---

Explanation:

7 8 3 4 0 0 1 5 1 10 0 0 2 2 6 0 0 9

- $I_T = 7$
- For him **first process**:
  - $I_C = 8$
  - $NH = 3$
  - $E_C = 1$  when you are between your first and second child
  - $E_C = 2$  when you are between the second and third child
  - $C.F. = 9$

- For the first child (second process)
  - IC =4
  - NH =0
  - CF =0 (when a process has no children, this value will always be 0 but anyway it is delivered in the input)
- For the second child (third process)
  - IC =5
  - NH =1
  - CF =2
- For the son of the second son (fourth process)
  - IC =10
  - NH =0
  - CF =0
- For the third child (fifth process)
  - IC =6
  - NH =0
  - CF =0

Note that the P.I.D. assigned by the OS (if this is the only group in the input) will be 1 for the father, and then 2 for the first child, 3 for the second, 4 for the son of the second and 5 for the third.

## 5.2. Output file (*Output*)

Within each loop, the events reported to the user will be recorded. SW, these are:

- IDLE: Report the time spent in IDLE on SW.
 

```
IDLE <IDLE_TIME>
IDLE 2
```
- ENTER: Reports that a process enters for the first time SW.
 

```
ENTER 1 0 1 TIME 7 LINE 1 ARG 1
ENTER <PID> <PPID> <GID> TIME <SO_CURRENT_TIME> LINE <LINE_NUM> ARG <ARGUMENT_NUM>
```
- RUN: Report the time the process worked.
 

```
RUN 1 8
RUN <PID> <WORK_TIME>
```
- WAIT: Reports that the process entered status WAITING.
 

```
WAIT <PID>
WAIT 1
```
- SUMMARIZES: Reports that a process returns to state READY.
 

```
RESUME <PID>
SUMMARY 1
```

- END: Reports that a process ends its execution.

```
END <PID> TIME <SO_CURRENT_TIME> END
2 TIME 19
```

At the end of each loop of theSW,heSWreports general statistics of each process. These are in order of arrival of each group to theSW,and then the completed processes are reported (for these we useGID=0).<sup>6</sup>

1. The text REPORT START is reported.

```
REPORT START
```

2. The text TIME is reported followed by the current time of theSW.

```
TIME <CURRENT_TIME_SO>
TIME 20
```

3. It is reported for each group in order:

- A line indicating the text GRUP, GID, number of processes in the group.

```
GROUP <GID> <NUM_PROGRAMS_GRUPÖ>
GROUP 1 2
```

- For each process in the group, according to its order of entry into the group, the following is reported: PROGRAM text, PID, PPID, GID, process status, amount of CPU it has used.

```
PROGRAM <PID> <PPID> <GID> <PROCESS_STATUS> <TOTAL_CPU_USED>
PROGRAM 1 0 1 WAITING 9
```

4. The text REPORT END is reported.

```
REPORT END
```

Important: You may notice that, basically, aTXTwhere in each row each data is presentedwith only a single space in between.

## Formalities

Each student was assigned (or will be assigned in the very near future) a username and password for the course server<sup>7</sup>. To submit your assignment you must create a folder calledT0in the home directory of your personal folder and upload your assignment to that folder. in this folderYou should only include the source codenecessary to compile your task and aMakefile.The contents of said folder will be reviewed on Monday, April 8, 2024 at 9:00 p.m. Only one member of each group must submit in their folder. You will choose the couples and if someone does not have a partner, you can try searching through the course forum or the Telegram chat.

Before the end of the task delivery period, a form will be sent where the couples can register along with the username of the person who will make the delivery on the server.

- You should NOT upload your assignment to a public repository. Otherwise, the maximum grade will be 4.0.
- You should NOT include binary files in your delivery. Otherwise, you will have a discount of 0.3 points on your final grade.
- If you register your group incorrectly, you will have a discount of 0.3 points.

---

<sup>6</sup>Within each group, it is reported according to when they began to be executed  
<sup>7</sup>iic2333.ing.puc.cl





- Your task will need to compile using the command `make` in the folder `T0`, and generate the executables `runner` and `scheduler` in this same one. If your program does not have a `Makefile` or the folder name is not correct, will not be corrected and you will have a discount of 0.5 points on the re-correction.
- If you use the *plugin* of SSH for VSCode or a similar one, that installs software on the server, will have a 1 point discount on their task.
- If your program does not compile either it does not work (*segmentation fault*), they will obtain the minimum grade, being able to go through modifying lines of code with a discount of one tenth for every four lines modified, with a maximum of 20 lines to modify.

Failure to comply with formalities or an extremely disordered code could lead to additional discounts at the discretion of the correcting assistant. It is recommended to modularize, use functions and use explanatory variable names.

## Assessment

Each part of this task will be evaluated with a grade between 1 and 7. Thus, the final grade will be the average between the 2 parts of the task. The score of each of the parts will follow the following distribution:

- 6.0 pts. Part I: Processes.
  - 0.8 pts. Correct implementation of `wait` all.
  - 0.5 pts. Correct implementation of `time` (`max`).
  - 1.5 pts. Correct implementation of multiple parallel processes.
  - 1.5 pts. Communication between processes through signals, implementation of the term by `CtrlZ`.  
  - 1.7 pts. *Output* correct.
  - Discount: Memory Management It will be discounted up to 1.0 pts. Yeahvalgrind does not report in its code 0 *leaks* and 0 memory errors in all use cases<sup>8</sup>.
- 6.0 pts. Part II: Scheduler.
  - 1.0 pt. Tests with groups that have a single program.
  - 2.0 pts. Tests with groups that have up to 3 programs.
  - 3.0 pts. Tests with groups that have an arbitrary number of programs.
  - Discount: Memory Management It will be discounted up to 1.0 pts. Yeahvalgrind does not report in its code 0 *leaks* and 0 memory errors in every use case.

## Important:

With the objective that both parts of the task are carried out and none is left out, in the event that one of the two parts of the task has a score lower than 2.5 pts., The grade for the assignment will be assigned using the lowest score weighted by 0.8 and the highest score weighted by 0.2.

$$\text{Score obtained} = \min\{\text{Score Part I}, \text{Score Part II}\} \cdot 0.8 + \max\{\text{Score Part I}, \text{Score Part II}\} \cdot 0.2$$

The task must be performed in the programming language `c`. Any assignment written in another programming language will not be reviewed and will be evaluated with a grade of 1. In addition, your work will be reviewed using Moss, a plagiarism and code similarity detection software, so any code that is not your own that they use in their tasks must be properly referenced. This excludes the code base given to students.

<sup>8</sup>That is, it must report 0 *leaks* and 0 errors for everything *test*, regardless of whether it ends normally or through an interruption.

## Questions

Any questions ask through [official forum](#).

## UC Academic Integrity Policy and Honor Code

Students of the School of Engineering of the Pontificia Universidad Católica de Chile must maintain behavior in accordance with the University's Honor Code:

*"As a member of the community of the Pontificia Universidad Católica de Chile, I am committed to respecting the principles and regulations that govern it. Likewise, I promise to act with rectitude and honesty in relationships with other members of the community and in the performance of all work, particularly in those activities linked to teaching, learning and the creation, dissemination and transfer of knowledge. In addition, I will ensure the integrity of people and take care of the assets of the University."*