Lab 04 – Scheduling ⏱️

Instructor: Lorenzo De Carli, University of Calgary (lorenzo.decarli@ucalgary.ca)
Slides by Lorenzo De Carli, based on material by Robert Walls (WPI)

In this lab

- Build a **scheduler simulator**
  - Receive as input a **sequence of jobs** in a file
  - **Simulate execution** of those jobs
  - Output an **execution trace** with:
    - Job start time
    - Job end time
    - Policy analysis

Deliverable

A scheduler executable

- You will be provided with a **template** (including tests and Makefile)
- Compiler should produce a **scheduler.out** file accepting **3 parameters**:
  - A flag (**0** or **1**) detailing whether or not to perform **policy analysis**
    - …more details about this later
  - Name of the scheduling policy (**FIFO/SJF/STCF/RR/LT**)
  - Length of each **time-slice** (used for RR; ignored otherwise)
  - Name of input job file (e.g. **jobs.txt**)

Example

Running the SJF policy w/ no analysis from jobs.txt

./scheduler.out 0 SJF 2 jobs.txt

- You can assume **all parameters** are **always specified**
- Here **0** means “no policy analysis”
- **SJF** means to run the SJF scheduling policy
- 2 means a timeslice of 2 time units (meaningless for SJF)
- **jobs.txt** means to read the list of jobs from the file “jobs.txt”

Input file format

- Each **workload** is defined in a **workload file**.
- Each line of the workload file represents a **different job** in the workload
- Each line consists of **two comma-separated numbers**:
  - the **arrival time**, and
  - the **total amount of simulated time** that job needs to run.

Input file format - example

```
0,1
2,5
3,11
5,4
7,1
25,4
```

Each line includes **arrival time** and **duration**

You can assume there are **no extra spaces**, and that **every line ends with a newline**

So… what’s about these jobs?

- To be clear, those are **not actual jobs**
- The scheduler should **simulate** that sequence of jobs by appropriately **computing when they start and end**
- There is no need to actually run anything (except the **scheduler calculations**)

Job list data structure

- The scheduler uses the job file to initialize a **job list data structure**
- In practice, this should be a **linked list**
- Each job should be assigned an **id** based on the **line number in the file**
- The job on the **first line** should be assigned an id of **0**; the job on the **second line** should be assigned an id of **1**; and so on

Job list data structure /2

This is just an example…

```c
struct job {
    int id;
    int arrival;
    int length;
    // other meta data
    struct job *next;
};
```

Some more things…

- Your scheduler should account for periods when there are **no jobs** to run
  - That is, all the arrived jobs **have completed** before the new jobs arrive
  - In other words, **the CPU can be idle**
- For pre-emptive scheduling policies: also note that a job (or the remaining duration of a job) may last **less than the duration of a time slice**

Implementing policies

Implementing FIFO

- The **FIFO policy** is one of the simplest scheduling policies
  - Good starting point! 🙂
- The FIFO policy states that **jobs are scheduled in order of their arrival**
- Each job **runs to completion**
- To be clear: there is **no preemption** for this FIFO policy.

Example scheduler output…

…when running FIFO

```
$ ./scheduler.out 0 FIFO 2 tests/3.in
Execution trace with FIFO:
t=0: [Job 0] arrived at [0], ran for: [20]
t=20: [Job 1] arrived at [0], ran for: [19]
t=39: [Job 2] arrived at [1], ran for: [18]
t=57: [Job 3] arrived at [1], ran for: [17]
t=74: [Job 4] arrived at [2], ran for: [16]
t=90: [Job 5] arrived at [3], ran for: [15]
t=105: [Job 6] arrived at [4], ran for: [14]
End of execution with FIFO.
```

Implementing SJF

(aka “Shortest Job First”)

- **SJF** always picks the job with the **shortest runtime** to run next
- We again assume that a job will run to completion **before the next is started**
- If two jobs need the same amount of time, SJF breaks the tie by favoring **the job that arrived earlier**
- Your SJF scheduler should account for periods when there are **no jobs** to run
    - That is, all the arrived jobs **have completed** before the new jobs arrive
    - In other words, **the CPU can be idle**

Example scheduler output…

…when running SJF

```
$ ./scheduler 0 SJF 2 tests/8.in
Execution trace with SJF:
t=0: [Job 0] arrived at [0], ran for: [1]
t=2: [Job 1] arrived at [2], ran for: [5]
t=7: [Job 4] arrived at [7], ran for: [1]
t=8: [Job 3] arrived at [5], ran for: [4]
t=12: [Job 2] arrived at [3], ran for: [11]
t=25: [Job 5] arrived at [25], ran for: [4]
End of execution with SJF.
```

Note that the CPU was **idle** for 1 tick here

Implementing STCF

(aka “Shortest Time to Completion”)

- The **STCF policy** is a preemptive version of SJF
- STCF only makes scheduling decisions when jobs arrive/complete
- When a new job arrives, STCF must run find out which jobs has the shortest amount of time left, and run that one
- When a job complete, the same process is also followed

Example scheduler output…

…when running STCF

```
Execution trace with STCF:
t=100: [Job 0] arrived at [100], ran for: [10]
t=110: [Job 1] arrived at [110], ran for: [10]
t=120: [Job 2] arrived at [110], ran for: [10]
t=130: [Job 0] arrived at [100], ran for: [90]
t=220: [Job 3] arrived at [220], ran for: [20]
t=240: [Job 4] arrived at [220], ran for: [30]
End of execution with STCF.
```

Implementing RR

(aka “Round-Robin”)

- **RR** runs each job in turn for the duration of the time slice S
- Note that not all jobs may arrive at the same time!
  - If a scheduling decision is being made at time T, only jobs arrived at or before T should be considered
- Once a job has been run for S ticks, its duration must be diminished by S

Example scheduler output…

…when running RR

```
Execution trace with RR:
t=0: [Job 0] arrived at [0], ran for: [2]
t=2: [Job 0] arrived at [0], ran for: [2]
t=4: [Job 0] arrived at [0], ran for: [1]
t=5: [Job 1] arrived at [5], ran for: [2]
t=7: [Job 2] arrived at [5], ran for: [2]
t=9: [Job 2] arrived at [5], ran for: [1]
t=14: [Job 3] arrived at [14], ran for: [1]
End of execution with RR.
```

Implementing LT

(aka “Lottery Scheduling”)

- Strictly speaking, a **lottery scheduler** does not have to be preemptive…
- …however, here we are going to implement it as such
- The lottery scheduler assigns a number of **tickets** to each job. Then:
  - **Extract ticket** T and run the job J_T to which the T belongs to for S ticks
  - Reduce the duration of J_T by S
  - Look at all jobs and **run the lottery again**

Extract ticket?

- **Simple implementation:** Use a **linked list** of jobs and the allotted number of tickets.
- Extract the winning number using a **random number generator**
- **Traverse the list** and use a simple **counter** and stop when that counter exceeds the winning number.
- See lecture slides for more details…

How do I assign tickets to jobs?

We are going to keep it simple

- Assign 100 tickets to the first job that arrives
- 200 tickets to the next
- And so on…

A note on randomness

- In order to ensure compliance with the test, you need to pre-initialize the random number generator

  Look at code template: `srand(42);`

- Later, every time the scheduler makes a decision you can select the winning ticket using: `int winning_ticket = rand() % total_tickets;`

- … and then, use the linked list approach discussed in class to pick the winning process

- We’ll be somewhat flexible w/ compliance with the tests for the LT part

Example scheduler output…

…when running LT

```
Execution trace with LT:
t=0: [Job 2] arrived at [0], ran for: [10]
t=10: [Job 1] arrived at [0], ran for: [10]
t=20: [Job 0] arrived at [0], ran for: [10]
t=30: [Job 2] arrived at [0], ran for: [10]
t=40: [Job 2] arrived at [0], ran for: [10]
t=50: [Job 1] arrived at [0], ran for: [10]
t=60: [Job 2] arrived at [0], ran for: [10]
t=70: [Job 1] arrived at [0], ran for: [10]
t=80: [Job 2] arrived at [0], ran for: [10]
t=90: [Job 2] arrived at [0], ran for: [10]
t=100: [Job 2] arrived at [0], ran for: [10]
End of execution with LT.
```

Policy analysis

Policy analysis?

- In this part of this project, you will add code to the scheduler to help it evaluate the **performance** of the previously implemented **policies**
- Your code will measure **three metrics**:
    - Response time
    - Turnaround time
    - Wait time

Metric definitions

- Assume:
  - T_a is the job **arrival time**
  - T_s is the job **start time**
  - T_c is the job **completion time**
- Then:
  - **Response time** is $T_s - T_a$
  - **Turnaround time** is $T_c - T_a$

Wait time

- Wait time is the total accumulated amount of time a job spends waiting while other jobs run
- If a job arrives at 0, starts at **6**, runs for 2 ticks, then wait for **4** ticks, then run for 2 ticks, is overall wait time is **6 + 4 = 10**
- Note: for non-pre-emptive policies, wait time and response time are the same!

Scheduler policy analysis

- The modified scheduler should output, for each metric:
  - The **per-job value** of the metric
  - The **average value** of the metric **across all jobs**

Example FIFO scheduler output…

…with metrics

```
$ ./scheduler 1 FIFO 2 tests/3.in
Execution trace with FIFO:
t=0: [Job 0] arrived at [0], ran for: [20]
t=20: [Job 1] arrived at [0], ran for: [19]
t=39: [Job 2] arrived at [1], ran for: [18]
t=57: [Job 3] arrived at [1], ran for: [17]
t=74: [Job 4] arrived at [2], ran for: [16]
t=90: [Job 5] arrived at [3], ran for: [15]
t=105: [Job 6] arrived at [4], ran for: [14]
End of execution with FIFO.
Begin analyzing FIFO:
Job 0 -- Response time: 0 Turnaround: 20 Wait: 0
Job 1 -- Response time: 20 Turnaround: 39 Wait: 20
Job 2 -- Response time: 38 Turnaround: 56 Wait: 38
Job 3 -- Response time: 56 Turnaround: 73 Wait: 56
Job 4 -- Response time: 72 Turnaround: 88 Wait: 72
Job 5 -- Response time: 87 Turnaround: 102 Wait: 87
Job 6 -- Response time: 101 Turnaround: 115 Wait: 101
Average -- Response: 53.43 Turnaround 70.43 Wait 53.43
End analyzing FIFO.
```

Example SJF scheduler output…

…with metrics

```
$ ./scheduler 1 SJF 2 tests/8.in
Execution trace with SJF:
t=0: [Job 0] arrived at [0], ran for: [1]
t=2: [Job 1] arrived at [2], ran for: [5]
t=7: [Job 4] arrived at [7], ran for: [1]
t=8: [Job 3] arrived at [5], ran for: [4]
t=12: [Job 2] arrived at [3], ran for: [11]
t=25: [Job 5] arrived at [25], ran for: [4]
End of execution with SJF.
Begin analyzing SJF:
Job 0 -- Response time: 0 Turnaround: 1 Wait: 0
Job 1 -- Response time: 0 Turnaround: 5 Wait: 0
Job 2 -- Response time: 9 Turnaround: 20 Wait: 9
Job 3 -- Response time: 3 Turnaround: 7 Wait: 3
Job 4 -- Response time: 0 Turnaround: 1 Wait: 0
Job 5 -- Response time: 0 Turnaround: 4 Wait: 0
Average -- Response: 2.00 Turnaround 6.33 Wait 2.00
End analyzing SJF.
```

Example RR scheduler output…

…with metrics

```
Execution trace with RR:
t=0: [Job 0] arrived at [0], ran for: [1]
t=1: [Job 1] arrived at [1], ran for: [2]
t=3: [Job 2] arrived at [3], ran for: [2]
t=5: [Job 1] arrived at [1], ran for: [2]
t=7: [Job 2] arrived at [3], ran for: [2]
t=9: [Job 1] arrived at [1], ran for: [1]
End of execution with RR.
Begin analyzing RR:
Job 0 -- Response time: 0  Turnaround: 1  Wait: 0
Job 1 -- Response time: 0  Turnaround: 9  Wait: 4
Job 2 -- Response time: 0  Turnaround: 6  Wait: 2
Average -- Response: 0.00  Turnaround 5.33  Wait 2.00
End analyzing RR.
```

Code template

- **scheduler.c:** template file to complete to implement the scheduler
- To complete the lab, you must extend the template to implement the required functionality
- As in lab 3, you must only submit the *scheduler.c* file, nothing else!

There are tests!

Look in the code template folder

- `test_fifo/:` FIFO policy tests
- `test_sjf/:` SJF policy tests
- `test_stcf/:` STCF policy tests
- `test_rr/:` RR policy tests
- `test_lt/:` LT policy tests
- Each test folder includes tests for runs with and without analysis
- We will also use these tests for grading

How to run tests?

- You can run individual tests by doing (e.g., for feature 1):

```
cd test_fifo
python3 test_fifo.py
```

- From the output, it will be clear which tests passed and which ones do not
- Look into the `.in` files in each test directory to understand what specifically is being tested (the corresponding expected output is in the `.out` files)
- To run all tests, run `python3 test_all.py`
- Note, the tests assume that there exists a `scheduler.out` executable in the main project directory (remember to run `make` before running the tests!)

Grading rubric

…the part everyone cares about!

- You'll have until **11:59PM of the day before the next lab** to upload the **complete solution**. That will be graded as follows:
    - Correct **FIFO implementation**: 2 pts
    - Correct **SJF implementation**: 2 pts
    - Correct **STCF analysis**: 2 pts
    - Correct **RR analysis**: 2 pts
    - Correct **LT analysis**: 2pts

Submission instructions

- You need to submit **a single file to the D2L "Lab 3" dropbox**
  - You have until **11:59 PM on the day before the next lab** to submit your completed lab solution, in a single *scheduler.c* file
  - **Please do not compress the file!**
- **THE SUBMISSION MUST ONLY CONSIST OF scheduler.c, NOTHING ELSE**
  - NO tests
  - NO readme
- This submission will be graded **10 Pts** according to the rubric

That’s all!