Lab 04 - Becoming a Scheduler ⏱️

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

Deliverable

A scheduler executable

- You will need to provide a **Makefile** to compile your code
- 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** or **SJF**)
  - Name of input job file (e.g. **jobs.txt**)

Example

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

./scheduler.out 0 SJF jobs.txt

- You can assume **all parameters** are **always specified**
- Here **0** means “no policy analysis”
- **SJF** means to run the SJF scheduling policy
- **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…

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

Implementing policies

Exercise 1: 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 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.
```

Exercise 2: 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 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

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 **two metrics**:
  - Response time
  - Turnaround 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 $T_c - T_a$

Exercise 3: 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 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 tests/8.in 0 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.

Code template

File-by-file description

- **scheduler.c:** template file to complete to implement the scheduler
- **example_fifo.in:** example input to test the FIFO scheduler
- **example_fifo.out:** expected output when running ./scheduler.out 0 FIFO example_fifo.in
- **example_fifo_analysis.out:** expected output when running **./scheduler.out 1 FIFO example_fifo.in**
- **example_sjf.in:** example input to test the FIFO scheduler
- **example_sjf.out:** expected output when running **./scheduler.out 0 SJF example_sjf.in**
- **example_sjf_analysis.out:** expected output when running **./scheduler.out 1 SJF example_sjf.in**

More on the code template

- The **readme.md** file (also in the template directory) contains useful information on **how to test your code**
- Most importantly, **don’t assume we are going to use the reference inputs/outputs when grading**.
- If your code works on the sample inputs/outputs but **fails on other ones**, you will still **lose points**
- See **readme.md** for some hints on how to **test your work efficiently**
- Finally, **remember to create a Makefile**

Grading rubric

…the part everyone cares about!

- As usual, you get **3 pts** for uploading a **partial solution** by **end of lab (4 PM)**
- Then, 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 **FIFO analysis**: 1.5 pts
  - Correct **SJF analysis**: 1.5 pts