Lab 05 - More 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

- Extend your **scheduler simulator** developed in lab 4
- Add pre-emptive scheduling policies:
  - STCF
  - Round-Robin
  - Lottery Scheduling
- Including **analysis**!

Deliverable

Another scheduler executable

- You will need to provide a **Makefile** to compile your code
- Compiler should produce a **scheduler.out** file accepting **4 parameters**:
  - A flag (**0** or **1**) detailing whether or not to perform **policy analysis**
    - …more details about this later
  - Name of the scheduling policy (**STCF, RR,** or **LT**)
  - Length of each **time-slice**

Example

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

./scheduler.out 0 RR jobs.txt 5

- You can assume **all parameters** are **always specified**
- Here **0** means “no policy analysis”
- **RR** means to run the Round-Robin scheduling policy
- **jobs.txt** means to read the list of jobs from the file “jobs.txt”
- **5** means use time-slices of 5 ticks in size

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**

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**
- Also note that a job (or the remaining duration of a job) may last **less than the duration of a time slice**

Implementing policies

Exercise 1: Implementing STCF

(aka “Shortest Time to Completion”)

- The **STCF policy** is a preemptive version of SJF
- Your scheduler should receive as parameter a **time slice size** S
- Every S ticks, the scheduler considers which job J_L has the shortest remaining duration
  - Run J_L for S ticks
  - Reduce the duration of J_L by S
  - Look at all jobs and decide again

Example scheduler output…

…when running STCF

Execution trace with STCF:
t=0: [Job 0] arrived at [0], ran for: [10]
t=10: [Job 1] arrived at [10], ran for: [10]
t=20: [Job 2] arrived at [10], ran for: [10]
t=30: [Job 0] arrived at [0], ran for: [10]
t=40: [Job 0] arrived at [0], ran for: [10]
t=50: [Job 0] arrived at [0], ran for: [10]
t=60: [Job 0] arrived at [0], ran for: [10]
t=70: [Job 0] arrived at [0], ran for: [10]
t=80: [Job 0] arrived at [0], ran for: [10]
t=90: [Job 0] arrived at [0], ran for: [10]
t=100: [Job 0] arrived at [0], ran for: [10]
t=110: [Job 0] arrived at [0], ran for: [10]
End of execution with STCF.

Exercise 2: 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: [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.

Exercise 3: 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 09 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…

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

Exercise 4: 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**

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 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

File-by-file description

- **scheduler.c:** template file to complete to implement the scheduler
- **example_stcf.in:** example input to test the STCF scheduler
- **example_stcf_analysis.out:** expected output when running **./scheduler.out 1 STCF example_stcf.in 10**
- **example_rr.in:** example input to test the RR scheduler
- **example_rr_analysis.out:** expected output when running **./scheduler.out 1 RR example_rr.in 2**
- **example_lt.in:** example input to test the LT scheduler
- **example_lt_analysis.out:** example output when running **./scheduler.out 1 LT example_lt.in 10**

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 (1 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 **STCF implementation**: 1.5 pts
  - Correct **RR implementation**: 1.5 pts
  - Correct **LT implementation:** 2 pts
  - Correct **STCF analysis**: 0.5 pts
  - Correct **RR analysis**: 0.5 pts
  - Correct **LT analysis:** 1 pt