University of Calgary
Faculty of Science
Department of Computer Science

CPSC 457 - Principles Of Operating Systems
Assignment 3
Deadlock Detection, CPU Scheduler
Fall 2023

1 Objective

The primary objective of this assignment is to deepen students' understanding of Deadlock problem and to introduce a popular algorithm in CPU time scheduling (Round Robin)

The first section delves into the concept of Deadlocks. Students are expected to gain a thorough understanding of deadlock conditions and strategies to handle them. This encompasses recognizing situations where processes may become permanently blocked as they wait for resources, detecting the conditions that lead to deadlocks, and learning how to apply deadlock prevention and resolution techniques.

The second section explores the area of CPU scheduling, using practical example about how a round-robin scheduler operates. Through this example, students will learn about the challenges in scheduling CPU time among multiple processes, such as the need for fair process rotation and avoiding process starvation. Additionally, students will be tasked with implementing scheduling algorithms, ensuring that all processes are given CPU time in an orderly and efficient manner.

Section 1: Deadlock Detection (50 marks)

You're the manager of a large e-commerce distribution center where robots are deployed to fetch items from shelves and bring them to the packing. During busy times, many robots might be tasked with fetching the same popular item from the same shelf. However, for efficiency and safety, only one robot is allowed access to a particular shelf at any given time. Also, each shelf has a set of tools required for picking different types of items (e.g., a scanner, a special gripper, or a magnetic pad). A robot needs to request these tools before accessing the shelf. After picking an item, a robot might need to put back a tool it used.

This leads to a situation where Robot A might be waiting for a scanner that Robot B is currently using, while Robot B is waiting for a magnetic pad that Robot A has. If not

managed properly, they could end up blocking each other indefinitely.

You, as the warehouse manager, realize the need for a system that can predict and prevent such situations. Hence, implementing a Deadlock Detector becomes essential to ensure timely deliveries and maintain the company's reputation.

Description

For this question you will write a deadlock detection algorithm for a system state with a single instance per resource type. The input will be an ordered sequence of request and assignment edges. Your algorithm will start by initializing an empty system state (e.g. empty graph), and then process the edges one at a time. For each edge, your algorithm will update the system state (e.g. insert the edge into a graph), and then run a deadlock detection algorithm (e.g. topological sort). If a deadlock is detected after processing an edge, your algorithm will stop processing further edges and return results immediately. Below is the signature of the find-deadlock function you need to implement:

```
struct Result {
    int index;
    std::vector<std::string> procs;
};

Result detect_deadlock(const std::vector<std::string> &edges);
```

The parameter `edges[]` is an ordered list of strings, each representing an edge. The function returns an instance of Result containing two fields as described below.

Your function will start with an empty system state – by initializing an empty graph data structure. For each string `edges[i]` it will parse it to determine if it represents an assignment edge or request edge and update the graph accordingly. After inserting each edge into the graph, the function will run an algorithm that will look for a deadlock (by detecting if a cycle is present in the graph). If deadlock is detected, your function will stop processing any more edges and immediately return `Result`:

- with `index=i`, where i indicates which `edges[i]` caused the deadlock; and
- with `procs[]` containing robot names (processes) that are involved in a deadlock, in arbitrary order.

If no deadlock is detected after processing all edges, your function will indicate this by returning `Result` with `index=-1` and an empty `procs[]`.

Edge description

Your function will be given the edges as a vector of strings, where each string will represent and edge. A request edge will have the format `"robot -> resource"`, and assignment edge will be of the form `"robot <- resource"`, where `robot` and `resource` are represntations of

the process and resource, respectively. Here is a sample input, and its graphical representation:

```
edges =
[
    robot1 <- scanner,
    robot2 -> scanner,
    robot2 -> gripper,
    robot3 -> scanner,
    robot3 <- gripper
]
```

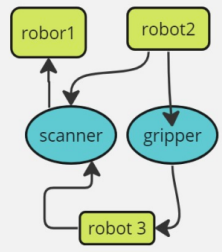


Figure 1: Graphical representation of resource allocation

The input above represents a system consists of three robots (processes) `("robot1", "robot2", "robot3")` and two resources `("scanner", "gripper")`. The first line, `"robot1 <- scanner"` represents an assignment edge, and it denotes `"robot1"` currently holding resource `"scanner"`. The second line `"robot 2 -> scanner"` demonstrates a request edge, denoting that `"robot 2"` is waiting for resource `"scanner"`. The allocation and requests are illustrated in the graph.

Notice that each individual string representing an edge may contain an arbitrary number of white spaces. Feel free to use the provided `split()` function to help you parse these strings, as it correctly deals with white spaces.

Starter Code

Start by downloading the starter code posted on D2L:

```
$ cd Deadlock-Detector-Started-code
$ make
```

You need to implement `detect_deadlock()` function by modifying the deadlock_detector.cpp file. **Only modify** file `deadlock_detector.cpp`, and **do not modify any other files**. Additionally, While you may write `C` code, the file must retain a `.cpp` extension, as all the `C` code is compatible with `C++` files.

The included driver (`main.cpp`) will read edge descriptions from standard input. It parses them into an array of strings, calls your `detect_deadlock()` and finally prints out the results. The driver will ensure that the input passed to your function is syntactically valid, i.e., every string in `edges[]` will contain a valid edge. Here is how you run it on file `test1.txt`:

```
$ ./deadlock ¡ test1.txt
Reading in lines from stdin...
Running detect_deadlock()...

index     : 6
procs     : [12,7,7]
real time : 0.0000s
```

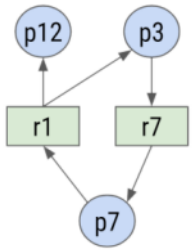
```
$ ./deadlock ¡ test1.txt
Reading in lines from stdin...
Running detect_deadlock()...

index     : -1
procs     : []
real time : 0.0001s
```

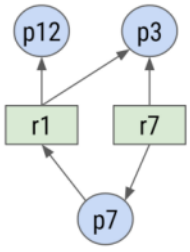
If you run the starter code (with an incomplete implementation), you will get the output on the left, which is obviously incorrect. Once implemented correctly, the output of your program will look like the one on the right, indicating no deadlock.

Few more examples:

```
$ cat test3a.txt
p7 <- r7
p7 -> r1
p3 -> r7
p3 <- r1
p12 <- r1
$ ./deadlock < test3.txt
index: 3
procs: [p7,p3]
```




```
$ cat test3b.txt
p7 <- r7
p7 -> r1
p3 <- r7
p3 <- r1
p12 <- r1
$ ./deadlock < test3b.txt
index: -1
procs: []
```



Limits

You may assume the following limits on input:

- Both process and resource names will only contain alphanumeric characters and will be at most 40 characters long.
- Number of edges will be in the range $[0 \ldots 30,000]$.

Your solution should be efficient enough to run on any input within the above limits in less than 10s, which means you should implement an efficient deadlock-detection algorithm (see appendix for hints). Remember, you are responsible for designing your own test cases.

Hints

I suggest using the following pseudocode for your implementation of `detect_deadlocks()`:

```
result ← empty Result
g ← initialize empty graph
for i = 0 to edges.size() do
  e ← parse edge in edges[i]
  insert e into g
  if toposort failed then
    result.procs ← nodes that toposort did not remove that represent robots
    result.index ← i
    return result
  end if
end for
result.index ← −1
return result
```

The above uses topological sort to detect whether a graph has cycles. Please note that toposort identifies any nodes that are directly or indirectly involved in a cycle, which is perfect for this assignment, as you need to report any robots that are involved in a deadlock.

Data Structure

I suggest you start with the following data structures to represent a graph:

```
class Graph {
    std::unordered_map<std::string, std::vector<std::string>>
        ↪ adj_list;
    std::unordered_map<std::string, int> out_counts;
    // ...
}graph;
```

The field `adj_list` is a hash table of dynamic arrays, representing an adjacency list. Insert nodes into it so that `adj_list["node"]` will contain a list of all nodes with edges pointing towards "node". The `out_counts` field is a hash table of integers, representing the number of outgoing edges for every node (outdegrees). Populate it so that `out_counts["node"]` contains the number of edges pointing out from "node".

With these data structures you can implement efficient topological sort (pseudocode):

out ← *out_counts*
zeros ← all nodes in graph with outdegree == 0
while zeros is not empty **do**
  n ← remove one entry from zeros[]
  for every $n2$ in adj_list[n] **do**
    $out[n2] \leftarrow out[n2] - 1$
    if $out[n2] == 0$ **then**
      add $n2$ to zeros[]
    end if
  end for
  robots involved in deadlock are nodes n that represent a robot and $out[n] > 0$
end while

Section 2: CPU Scheduler (50 marks)

Imagine a city hospital's emergency room (ER). Patients arrive at different times with varying degrees of health issues, similar to "processes" in a computer. The ER uses a scheduling system, much like the Round Robin method, to determine the order patients are seen. In this system, each patient gets a fixed time with the doctor. The hospital needs a well-planned system to ensure all patients are treated in time, critical cases are handled immediately, and waiting times are short. Write a program that allows hospital staff to effectively schedule the health care services for the patients to meet these objectives.

Defining the Elements:

- Process: Each patient in the ER.

- id: Patient ID number.
- arrival: Time the patient arrives.
- burst: Estimated time with the doctor.
- start-time: Actual time the patient starts seeing the doctor.
- finish-time: Time the patient finishes seeing the doctor.
- Quantum (time slice): Fixed time each patient is initially allotted with the doctor.

Description

To solve this question you will implement a round-robin CPU scheduling simulator. The input to your simulator will be a set of patients (processes) and an estimated time with the doctor (time slice). Each patient will be described by an id, arrival time and an Estimated time with the doctor (CPU burst). Your simulator will simulate RR scheduling on these patients and for each patient it will calculate actual time the patient starts seeing the doctor (start time) and ime the patient finishes seeing the doctor (finish time). Your simulator will also compute a condensed execution sequence of all patients. You will implement your simulator as a function `simulate_rr()` with the following signature:

Listing 1: Function signature and Process structure

```
void simulate_rr(
    int64_t quantum,
    int64_t max_seq_len,
    std::vector<Process> & processes,
    std::vector<int> & seq);

struct Process {
    int id;
    int64_t arrival, burst;
    int64_t start_time, finish_time;
};
```

The parameter `quantum` will contain a positive integer describing the length of the time slice and `max_seq_len` will contain the maximum length of execution order to be reported. The array `processes` will contain the description of pateients, where struct `Process` is defined in `scheduler.h` as above.

The fields `id`, `arrival` and `burst` for each patient (process) are the inputs to your simulator, and you should not modify these. However, you must populate the `start_time` and `finish_time` for each patient (process) with computed values. You must also report the condensed order of serving patients (condensed execution sequence of the processes) via the output parameter `seq[]`. You need to make sure the reported order contains at most the first `max_seq_len` entries. The entries in `seq[]` will contain either patient (process) ids, or -1 to denote idle CPU.

A condensed execution sequence is similar to a regular execution sequence, except consecutive repeated numbers are condensed to a single value. For example, if a regular non-

condensed sequence was `[-1,-1,-1,1,1,2,1,2,2,2]`, then the condensed equivalent would be `[-1,1,2,1,2]`.

Starter Code

Start by downloading the starter code posted on D2L and compile it:

```
$ cd Scheduler-Simulation-Starter-code
$ make
```

You need to implement the `simulate_rr()` function in scheduler.cpp. Do not modify any files except scheduler.cpp.

Using the driver

The starter code includes a driver (`main.cpp`) that parses command lines arguments to obtain the time slice and the maximum execution sequence length. It then parses standard input for the description of patients (processes), where each patient is specified on a separate line. Each input line contains 2 integers: the first one denotes the arrival time of the patient, and the second one denotes the CPU burst length. For example, the file test1.txt contains information about 3 patients: P0, P1 and P2:

```
$ cat test1.txt
1 10
3 5
5 3
```

The 2nd line "3 5" means that patient P1 arrives at time 3 and the stimated time with the doctor is 5 time unit. After parsing the inputs, the driver calls your `simulate_rr()`, and afterwards prints out the results. For example, to run your simulator with `quantum=3` and `max_seq_len=20` on a file `test1.txt`, you would invoke the driver like this:

```
$ ./scheduler 3 20 < test1.txt
Reading in lines from stdin...
Running simulate_rr(q=3, maxs=20, procs=[3])
Elapsed time    : 0.0000s

seq = [0,1,2]
+----+---------+-------+-------+--------+
| Id | Arrival | Burst | Start | Finish |
+----+---------+-------+-------+--------+
|  0 |       1 |    10 |     1 |     11 |
|  1 |       3 |     5 |     3 |      8 |
|  2 |       5 |     3 |     5 |      8 |
+----+---------+-------+-------+--------+
```

Please note that the output above is incorrect, as the starter code contains an incomplete implementation of the scheduling algorithm. The correct results should look like this:

```
seq = [-1,0,1,0,2,1,0]
+----+---------+-------+-------+--------+
| Id | Arrival | Burst | Start | Finish |
+----+---------+-------+-------+--------+
|  0 |       1 |    10 |     1 |     19 |
|  1 |       3 |     5 |     4 |     15 |
|  2 |       5 |     3 |    10 |     13 |
+----+---------+-------+-------+--------+
```

Important - If your simulation detects that an existing process exceeds its time slice at the same time as a new process arrives, you need to insert the existing process into the ready queue before inserting the newly arriving process.

Limits

You may assume the following limits on input:

- The processes are sorted by their arrival time, in ascending order. Processes arriving at the same time must be inserted into the ready queue in the order they are listed.
- All arrival times will be non-negative.
- All burst times will be greater than 0.
- Process IDs will be consecutively numbered starting with 0.

Hints for a simple solution

You can start with this solution, or immediately go to the "better solution". This simple solution increments the current time in the simulation loop by at most 1 time unit, and many students will find it the easiest to debug.

Please refer to the lecture slides for ideas on how to structure your simulation loop. Here are some suggestions for data structures for keeping track of the current state:

- The current time, e.g., `int64_t curr_time`
- The remaining time slice of the currently executing process, e.g., `int64_t remaining_slice`
- Currently executing process, e.g., `int cpu`, so that `cpu` is an index into `processes[]`, and `cpu=-1` represents idle CPU
- Ready Queue (RQ) and Job Queue (JQ), e.g., `std::vector<int> rq, jq`
  - the integers stored in `jq` and `rq` would be indices into `processes[]`, just like `cpu`
  - initialize JQ with all processes, and remove them from JQ as they 'arrive'

- You will need to keep track of the remaining bursts for all processes
  - Since you cannot modify `processes[]`, you need to keep track of this in your own data structure, e.g., `std::vector<int64_t> remaining_bursts;`

Hints for a better solution

This solution increments the current time by a value up to quantum, whenever possible.

Adjust your simulation loop so that at the top of the loop, your CPU is always idle. This way you no longer need keep track of what's currently on the CPU nor the remaining slice. In other words, you only need to keep track of `curr_time`, `remaining_bursts`, `jq` and `rq`. There are 4 cases you need to consider in your simulation loop:

- both JQ and RQ are empty:
  - this means your simulation is done
- only RQ is empty:
  - skip current time to the arrival of the next process in JQ
- only JQ is empty:
  - As we usually do, execute full time slice from the next process in RQ, unless the process would finish during this time
  - if the process did not finish, re-insert the process into RQ
  - adjust current time accordingly
- both JQ and RQ are not empty
  - the implementation here is similar to the case where JQ is empty, but with one important difference:
  - before you re-insert a process back into RQ, you must check to see if any processes arrived during the quantum, and if they did, put them into RQ before you re-insert the current process into RQ

Bonus (Extra 20 marks)

Hints for the best solution

This solution increments the current time by large multiples of quantum.

This builds on top of the "better solution" above, by adding additional optimizations steps into the simulation loop. These optimizations will essentially result in incrementing

the current simulation time by very large multiples of quantum. I suggest you implement this in two steps:

Step 1:

There are situations where you could safely execute one quantum for every process in RQ without any processes finishing, and without going past the arrival time of the next process. This would not change the order of processes in the RQ, and you would not miss the end time for any processes, and you would not miss arrival time for any process. In other words, you could increment current time by `rq.size() * quantum` without changing the state of the system, as long as you update the remaining time for each process in the RQ by subtracting quantum.

To make your life easy, only perform this optimization if you have the execution sequence completed (i.e. once you collected `max_seq_len` entries), and when all process in RQ already have their start time recorded.

Step 2:

This is similar to step 1, but this time making the increment to current time even larger. Instead of executing a single quantum for each process in RQ, you will execute multiple time slices for each process. In other words, you need to find the largest possible N so that you can increment current time by `N * rq.size() * quantum`. The N needs to be as big as possible, but small enough that no processes will finish during this time, and no processes will arrive during this time.

Grading

The assignment is worth a total of 100 points, divided equally between two sections, each worth 50 points. For the first problem, you can earn 50 points by successfully coding the 'detect_deadlock' method, using a topological sort algorithm that identifies and returns the index of the edge causing the deadlock. In the second problem, a basic implementation of the RR algorithm will earn you 25 points. An improved solution will grant you the remaining 25 points. Moreover, if you implement the best solution for the second problem, the CPU Scheduler, you will receive an additional 20 points.

Deadline

The assignment is due by 1st December at 11:59 PM, after that the deduction policy of 20% daily (part of the day is considered as a full day) will be applied. The first 15 minutes of the penalty day is a grace period and will not result in a deduction.

Submission

To submit, create a new folder called "assigment3" and place all files you wish to submit in this folder. The Zip File name should be the student name+ID (example: Name = John

Doe, Student ID = 12345678, FILENAME = john_doe_12345678). Ensure that you have added all the files while zipping.