

Assignment 4

CPSC 457 Winter 2023

Due date is posted on D2L.

Individual assignment. Group work is NOT allowed.

Weight: 22% of the final grade.

There are two C++ programming questions in this assignment. Do not use threads and do not create additional processes for either question. While both questions have the same weight, I suggest you start with Q1, as it is simpler of the two.

Q1 – Deadlock Detector (50 marks)

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. toposort). If a deadlock is detected after processing an edge, your algorithm will stop processing any more 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 find_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.

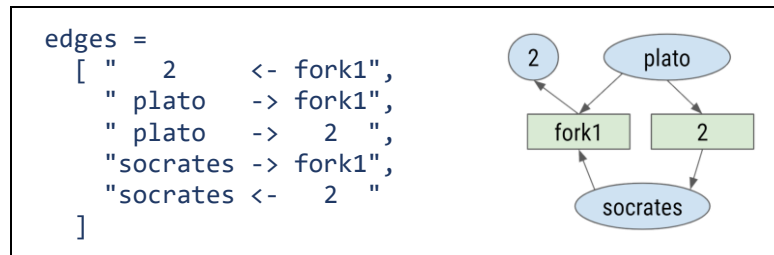
Your function will start with an empty system state – by initializing an empty graph data structure. For each string `edges[i]` it will parse it to determine if it represents an assignment edge or request edge and update the graph accordingly. After inserting each edge into the graph, the function will 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 process names 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 an edge. A request edge will have the format "`process -> resource`", and assignment edge will be of the form "`process <- resource`", where `process` and `resource` are the names of the process and resource, respectively. Here is a sample input, and its graphical representation:



The input above represents a system with three processes: "plato", "socrates" and "2", and two resources: "fork1" and "2". The first line "2 <- fork1" represents an assignment edge, and it denotes process "2" currently holding resource "fork1". The second line "plato -> fork1" is a request edge, meaning that process "plato" is requesting resource "fork1". The resource allocation graph on the right is a graphical representation of the input. Process and resource names are independent from each other, and it is therefore possible for processes and resources to have the same names.

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

Starter code

Start by downloading the following starter code:

```

$ git clone https://gitlab.com/cpsc457w23/deadlock.git
$ cd deadlock
$ make

```

You need to implement `find_deadlock()` function by modifying the `find_deadlock.cpp` file. The rest of the starter code contains a driver code (`main.cpp`) and some convenience functions you may use in your implementation (`common.cpp`). **Only modify** file `find_deadlock.cpp`, and **do not** modify any other files.

The included driver will read edge descriptions from standard input. It parses them into an array of strings, calls your `find_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 find_deadlock()...

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

```



```

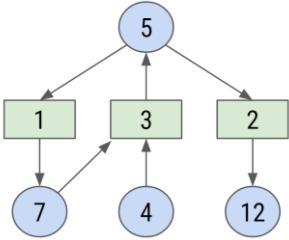
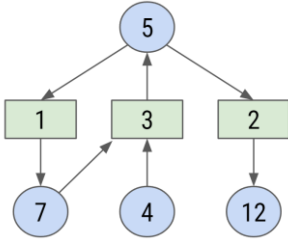
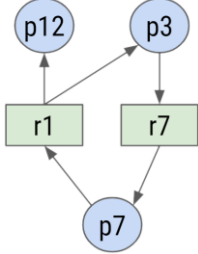
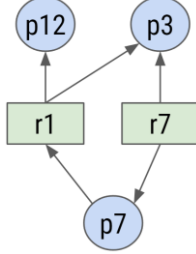
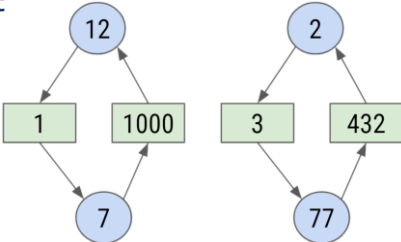
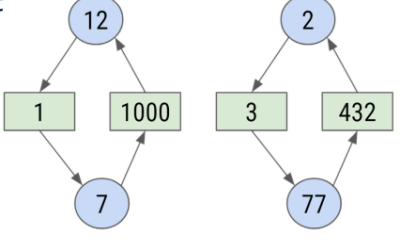
$ ./deadlock < test1.txt
Reading in lines from stdin...
Running find_deadlock()...

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:

<pre> \$ cat test2a.txt 5 -> 1 5 <- 3 5 -> 2 7 <- 1 12 <- 2 4 -> 3 7 -> 3 \$./deadlock < test2a.txt index: 6 procs: [4,7,5] </pre> 	<pre> \$ cat test2b.txt 5 -> 1 5 <- 3 5 -> 2 7 <- 1 12 <- 2 4 -> 3 7 -> 3 \$./deadlock < test2b.txt index: 5 procs: [7,5] </pre> 
<pre> \$ cat test3a.txt p7 <- r7 p7 -> r1 p3 -> r7 p3 <- r1 p12 <- r1 \$./deadlock < test3a.txt index: 3 procs: [p7,p3] </pre> 	<pre> \$ cat test3b.txt p7 <- r7 p7 -> r1 p3 <- r7 p3 <- r1 p12 <- r1 \$./deadlock < test3b.txt index: -1 procs: [] </pre> 
<pre> \$ cat test4.txt 12 -> 1 12 <- 1000 7 -> 1000 7 <- 1 2 -> 3 2 <- 432 77 -> 432 77 <- 3 \$./deadlock < test4.txt index: 3 procs: [12,7] </pre> 	<pre> \$ cat test5.txt 12 -> 1 12 <- 1000 7 -> 1000 7 <- 1 2 -> 3 2 <- 432 77 -> 432 77 <- 3 \$./deadlock < test5.txt index: 6 procs: [2,77] </pre> 

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 ... 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.

Marking

Your submission will be marked both on correctness and speed for several test files. To get full marks, your program will need to finish under 10s on all inputs during marking.

About 80% of the marks will be based on tests with less than 10,000 edges, which should be easy to achieve. For example, your program should be able to finish `test6.txt` under 10s on `linuxlab` machines. The remaining 20% will be awarded only to submissions that can finish under 10s for ~30,000 edges, which is more difficult (e.g. `test7.txt` file).

Hints

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

```
result = empty Result
g = initialize empty graph
for i = 0 to edges.size():
    e = parse edge in edges[i]
    insert e into g
    run toposort on g
    if toposort failed to finish:
        result.procs = nodes that toposort did not remove that represent processes
        result.index = i
        return result
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 processes that are involved in a deadlock.

I suggest you start with the following data structures to represent a graph. These data structures should be good enough to finish under 10s on medium sized files, such as `test6.txt`.

```
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 (pseudo-code):

```
out = out_counts # copy out_counts so that we can modify it
zeros[] = find all nodes in graph with outdegree == 0
while zeros is not empty:
    n = remove one entry from zeros[]
    for every n2 in adj_list[n]:
        out[n2] --
        if out[n2] == 0:
            append n2 to zeros[]
processes involved in deadlock are nodes n that represent a process and out[n]>0
```

To get run times under 10s on large files, such as `test7.txt`, you can use the same algorithm as above, but you will need to switch to data structures that are more efficient than hash tables. The problem with hash tables is that they spend considerable time on calculating hashes on strings, and also on resolving collisions. To avoid this overhead, you can pre-convert all strings (process and resource names) into consecutive integers, and then use fast dynamic arrays instead of hash tables to store the adjacency list and outdegree counts:

```
class FastGraph {
    std::vector<std::vector<int>> adj_list;
    std::vector<int> out_counts;
    ...
} graph;
```

You can use the provided `Word2Int` class to help you with converting strings to unique consecutive integers. The topological sort algorithm would remain the same as above. Do not forget to convert the integers back to strings at the end, to correctly populate `procs[]`.

Q2 – CPU Scheduler Simulation (50 marks)

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

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

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 processes, where struct `Process` is defined in `scheduler.h` as:

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

The fields `id`, `arrival` and `burst` for each 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 process with computed values. You must also report the 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 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, 1, 2, 2, 2]`, then the condensed equivalent would be `[-1, 1, 2, 1, 2]`.

Starter code

Download the starter code and compile it:

```
$ git clone https://gitlab.com/cpsc457w23/rrsched.git
$ cd rrsched
$ make
```

You need to implement the `simulate_rr()` function in `scheduler.cpp`. The rest of the starter code contains a driver code (`main.cpp`) and some convenience functions you may use in your implementation (`common.cpp`). **Do not modify** any files except `scheduler.cpp`.

Using the driver

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

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

The 2nd line "3 5" means that process P1 arrives at time 3 and it has a CPU burst of 5 seconds.

Once the driver parses the command line and standard input, it calls your simulator, and then 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 make the following assumptions about the inputs:

- 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.
- All processes are 100% CPU-bound, i.e., a process will never be in the waiting state.
- There will be between 0 and 30 processes.
- Time slice and CPU bursts will be integers in the range $[1 \dots 2^{62}]$
- Process arrival times will be integers in the range $[0 \dots 2^{62}]$
- The finish time of every process will fit into `int64_t`.

The git repository includes some test files and the [README.md](#) contains several sample results. Please remember to also design your own test data to make sure your program works correctly and efficiently for all of the above limits.

Marking

Your submission will be marked both on correctness and speed for a number of different test files. To get full marks, your program will need to finish under 10s on all test cases. About half of the test cases will include inputs with small values for arrival times and CPU bursts. But there will be some test cases with very large arrival times, and/or CPU bursts, and very small time-slices.

Start with a simulation loop that increments current time by 1. This should make your simulator work fast enough for small arrival times and bursts. Once you are convinced that it works correctly, you can try to improve it to run fast for large burst and large arrival numbers. To do this, you will need to determine the optimal value of the amount by which you increment the current time for each simulation loop iteration.

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
 - 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

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.

Submission

Submit 2 files to D2L for this assignment.

<code>find_deadlock.cpp</code>	solution to Q1
<code>scheduler.cpp</code>	solution to Q2

Submit individual files. Do not submit a ZIP file.

General information about all assignments:

- All assignments are due on the date listed on D2L. Late submissions will not be marked.
- Extensions may be granted only by the course instructor.
- After you submit your work to D2L, verify your submission by re-downloading it.
- You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
- Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
- All programs you submit must run on linuxlab.cpsc.ucalgary.ca. If we are unable to run your code on the [linuxlab](http://linuxlab.cpsc.ucalgary.ca) machines, you will receive 0 marks.
- **Assignments must reflect individual work.** For further information on plagiarism, cheating and other academic misconduct, check the information at this link: <http://www.ucalgary.ca/pubs/calendar/current/k-5.html>.
- Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions with anyone else; you are not allowed to sell or purchase a solution. This list is not exclusive.
- We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.