CPSC 457 - Assignment 4

Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 22% of the final grade.

There are two programming questions in this assignment. Question 1 should be quite straightforward to implement, and Question 2 will require more effort. Please do not use threads or additional processes for either question.

Q1 – Deadlock Detector (50 marks)

For this question you will write a C++ function that detects a deadlock in a system state with a single instance per resource type. The system state will be provided to this function as an ordered sequence of request- and assignment-edges. Your function will start by initializing an empty system state. Then, your function will process the edges one at a time. For each edge, your function will update the system state, and run a deadlock detection algorithm. If a deadlock is detected after processing an edge, your function will stop processing any more edges and return results immediately.

Below is the signature of the `detect_deadlock` function you need to implement:

```
struct Result {
    int edge_index;
    std::vector<std::string> dl_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 in `edges[]` it will parse it to determine if it represents an assignment edge or request edge, and update the graph accordingly. After inserting each edge, 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`:

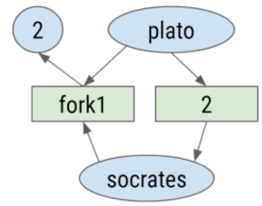
- with `edge_index` set to the index of the edge in `edges[]` that caused the deadlock; and
- with `dl_procs[]` containing process names that are involved in a deadlock. Order is arbitrary.

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

Edge description

Your function will be given the edges as a vector of strings, where each edge will represent and edge. A request edge will have the format `"(P) -> (R)"`, and assignment edge will be of the form `"(P) <- (R)"`, where `(P)` and `(R)` are the names of the process and resource, respectively. Here is a sample input, and its graphical representation:

```
edges =
  [ "   2     <- fork1",
    " plato   -> fork1",
    " plato   ->   2  ",
    "socrates -> fork1",
    "socrates <-   2  "
  ]
```



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 waiting for 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 spaces. Feel free to use the provided `simplify()` and/or `split()` functions from `common.cpp` to help you parse these strings.

Starter code

Start by downloading the following starter code:

```
$ git clone https://gitlab.com/cpsc457f21/deadlock-detect.git
$ cd deadlock-detect
$ make
```

You need to implement `detect_deadlock()` function my modifying `deadlock_detector.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`). **Only modify** file `deadlock_detector.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, and then calls your `detect_deadlock()`, and afterwards 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()...

edge_index : 6
dl_procs   : [12,7,7]
real time  : 0.0000s
```

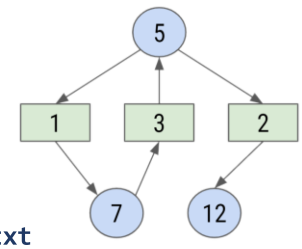
```
$ ./deadlock < test1.txt
Reading in lines from stdin...
Running detect_deadlock()...

edge_index : -1
dl_procs   : []
real time  : 0.0001s
```

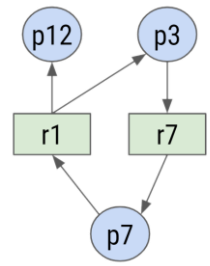
Please note that the above output on the left is incorrect because the starter code has an incomplete implementation of the deadlock detector that returns hard-coded values. Once implemented correctly, the output of the program will look like the one on the right. The correct output indicates no deadlock because there is no loop in the graph.

Few more examples:

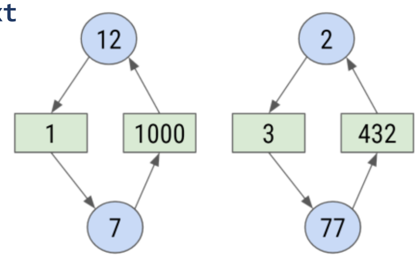
```
$ cat test2.txt
5 -> 1
5 <- 3
5 -> 2
7 <- 1
12 <- 2
7 -> 3
$ ./deadlock < test2.txt
edge_index : 5
dl_procs   : [5,7]
```



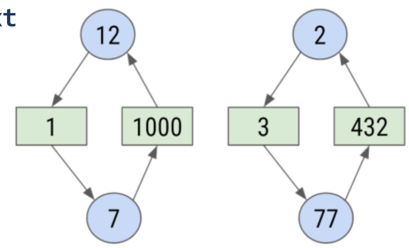

```
$ cat test3.txt
p7 <- r7
p7 -> r1
p3 -> r7
p3 <- r1
p12 <- r1
$ ./deadlock < test3.txt
edge_index : 3
dl_procs   : [p7,p3]
```




```
$ cat test4.txt
12 -> 1
12 <- 1000
7 -> 1000
7 <- 1
2 -> 3
2 <- 432
77 -> 432
77 <- 3
$ ./deadlock < test4.txt
edge_index : 3
dl_procs   : [12,7]
```




```
$ cat test5.txt
12 -> 1
12 <- 1000
7 -> 1000
2 -> 3
2 <- 432
77 -> 432
77 <- 3
7 <- 1
$ ./deadlock < test5.txt
edge_index : 6
dl_procs   : [2,77]
```



Limits

You may assume the following limits on input:

- Both process and resource names will contain at most 40 alphanumeric characters (digits, upper- and lower-case letters).
- 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 cycle-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 cases.

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. You can test your code on the large `test7.txt` file to see if it is fast enough to get full marks.

Q2 – Scheduler simulation (50 marks)

For this question you will implement a round-robin CPU scheduling simulator for 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_time, burst;
    int64_t start_time, finish_time;
};
```

The fields `id`, `arrival_time` and `burst` are the inputs to your simulator. Do not modify these.

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 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

Download the starter code and compile it:

```
$ git clone https://gitlab.com/cpsc457f21/scheduler.git
$ cd scheduler
$ 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 on external files

The starter code includes a driver 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 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.
- 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 … 2^{62}]
- Process arrival times will be integers in the range [0 … 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 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. There are more detailed hints in the appendix on how this can be achieved.

Submission

Submit 2 files to D2L for this assignment. **Do not submit a ZIP file. Submit individual files.**

`deadlock_detector.cpp`	solution to Q1
`scheduler.cpp`	solution to Q2

Please note – you need to **submit all files every time** you make a submission, as the previous submission will be overwritten.

Appendix 1 – hints for Q1

I suggest you start with the following data structures to represent a graph. These data structures should be good enough to get run times 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 a simple yet 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 of adj_list[n]:
   out[n2] --
   if out[n2] == 0:
     append n2 to zeros[]
dl_procs[] = all 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 more efficient data structures than hash tables. The problem with hash tables is they spend considerable time on calculating hashes on strings, and also on 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 `dl_procs[]`.

Appendix 2 – hints for Q2

Here are some suggestions for keeping track of the current state of the simulation:

- The current time
    - A 64-bit integer should do the trick, e.g. `int64_t curr_time =0;`
- The remaining time slice

    - e.g. `int64_t remaining_slice;`
- Currently executing process
    - an index into processes[] should suffice, where '-1' could represent idle CPU
    - e.g. `int cpu = -1;`
- Contents of the Ready Queue (RQ) and Job Queue (JQ)
    - I suggest std::vector of indices into processes[]
    - e.g. `std::vector<int> rq, jq;`
- You also probably need to keep track of the remaining bursts for all processes
    - e.g. `std::vector<int64_t> remaining_bursts;`

I suggest you start with an implementation where you increment current time by 1. Once you are done with this basic implementation, you can implement additional optimizations.

Here are some ideas for simple optimizations:

- see if you can adjust your simulation loop so that at the top of the loop, your CPU is always idle;
- if you start executing a process, you can execute full time slice, just make sure that before you put the process back into RQ, you check to see if any processes arrived during that time and if they did, put them into RQ before
- any time you start executing a process, and the process will finish before it exceeds the quantum, you can finish the process and advance current time to the end time of the process. If any processes arrive during this time, insert those into RQ.
- if you are executing last process (i.e., RQ is empty, and JQ is empty), then you can finish executing the current process, and end simulation.
- if RQ is empty and CPU is idle, then you can skip current time to the arrival of the next process.
- if RQ is empty but CPU has a process, then you can execute multiple time slices as long as you don't increment current time past the next arrival time.

The hardest part is to optimize the case when CPU is idle, but RQ is not empty, and there are processes still arriving in the future. This is especially needed when the time slice is tiny but arrival times and/or burst times are very large. Here are some hints for an optimization that you can do after you have already populated the execution sequence, and after every process in RQ has a recorded start time:

- ask yourself this question: could you execute one quantum for every process in RQ without any processes finishing, and without going past the arrival time of the next process? If you could, that 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.

- next question is: how many times could you do the above step? i.e. what is the maximum safe `N` such 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.

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 your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
- **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.