# CPSC 457 - Assignment 4

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

## Q1 – Programming question (20 marks)

For this question you will write a program `deadlock.[c|cpp]` that detects a deadlock in a system with a single instance per resource type.

The system state will be fed into your program as a sequence of request and assignment edges. Your program will start with an empty state (empty resource allocation graph). For each edge, your program will incrementally update the resource allocation graph and run a deadlock detection algorithm. As soon as a deadlock is detected, your program will output the edge that caused the deadlock, and all processes involved in the deadlock. Your program will terminate as soon as a deadlock is detected, or if there are no more edges to process. If your program does not detect any deadlocks, it will print an appropriate message. The formatting off the messages is illustrated in the examples below.

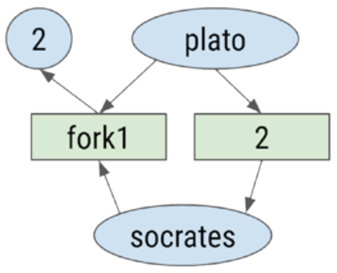
### Command line

Your program will take no command line arguments.

### Input

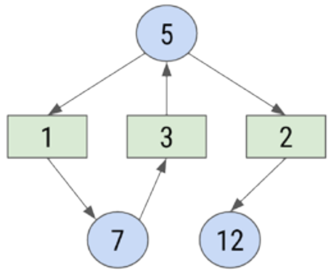
Your program will read the edges from standard input. Each edge will be specified on a separate line. A request edge will have the format `"(P) -> (R)"`, and assignment edge will be of the form `"(P) <- (R)"`, where `(P)` and `(R)` are the names of the process and resource, respectively. Here is a sample input and output:

```
$ cat test1.txt
2 <- fork1
plato -> fork1
plato -> 2
socrates -> fork1
socrates <- 2

$ ./a.out < test1.txt
No deadlock.
```

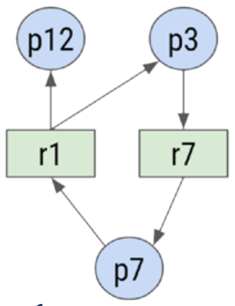


The input above represents a system with three processes: `"plato"`, `"socrates"` and `"2"`, and two resources: `"fork1"` and `"2"`. The first line `"2 <- fork1"` represents an assignment edge, and it denotes process `"2"` currently holding resource `"fork1"`. The second line `"plato -> fork1"` is a request edge, meaning that process `"plato"` is waiting for resource `"fork1"`. The resource allocation graph on the right is a graphical representation of the entire input. Process and resource names are independent from each other, and it is therefore possible for a process and a resource to share the same name. There is no deadlock in the system state above since there is no loop in the graph. Here are 4 more examples:

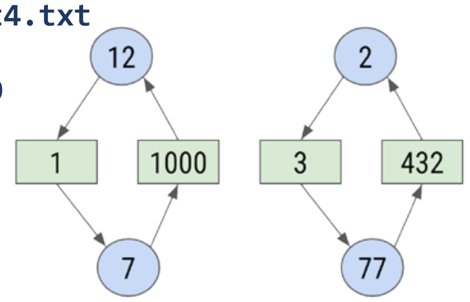
```
$ cat test2.txt
5 -> 1
5 <- 3
5 -> 2
7 <- 1
12 <- 2
7 -> 3

$ ./a.out < test2.txt
Deadlock on edge: 7 -> 3
Deadlocked processes: 5, 7
```



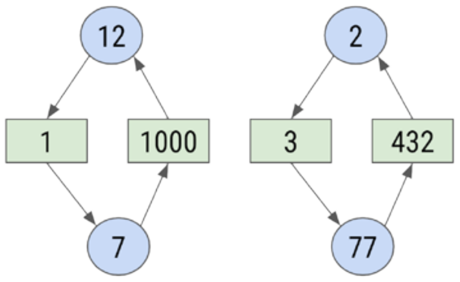
```
$ cat test3.txt
p7 <- r7
p7 -> r1
p3 -> r7
p3 <- r1
p12 <- r1

$ ./a.out < test3.txt
Deadlock on edge: p3 <- r1
Deadlocked processes: p3, p7
```



```
$ cat test4.txt
12 -> 1
12 <- 1000
7 -> 1000
7 <- 1
2 -> 3
2 <- 432
77 -> 432
77 <- 3

$ ./a.out < test4.txt
Deadlock on edge: 7 <- 1
Deadlocked processes: 7, 12
```



```
$ cat test5.txt
12 -> 1
12 <- 1000
7 -> 1000
2 -> 3
2 <- 432
77 -> 432
77 <- 3
7 <- 1

$ ./a.out < test5.txt
Deadlock on edge: 77 <- 3
Deadlocked processes: 77, 2
```



Please note that the resource allocation graphs above represent the entire input, but your program will stop reading the input as soon as it detects a deadlock.

**Output**

As soon as your program reads in an edge that causes a deadlock it needs to print out two lines, 1st indicating the edge on which the deadlock occurred, and the 2nd one listing all processes that are deadlocked, and then your program must stop. If your program does not detect a deadlock, it will output one line indicating so. Your output should match the sample output above exactly.

**Limits:**

You may assume the following limits on input:

- both process and resource names will contain at most 40 alphanumeric characters and can contain upper- and lower-case letters.
- number of edges will be in range [0 ... 10000]

Your solution should be efficient enough to be able run on any input within the above limits in less than 10 seconds. Hint: you should implement an efficient cycle-detection algorithm.

Link to sample input and useful starter code here: https://gitlab.com/cpsc457/public/deadlock-detect .

## Q2 – Scheduler simulation (20 marks)

Write a program `scheduler.[c|cpp]` that simulates two different scheduling algorithms: shortest job first and round-robin.

### Command line arguments

Your program will accept 2-3 command-line arguments:

1. The name of the configuration file. Your program will read the configuration file to obtain the description of processes.
2. The type of the scheduling algorithm: ‘RR’ for round-robin or ‘SJF’ for shortest job first.
3. The time quantum for the RR scheduling algorithm. If 2nd argument is SJF, the 3rd argument will not be specified.

For example, the command line below should invoke your simulator on file `config.txt` using RR scheduler and time slice of 3:

```
$ ./scheduler config.txt RR 3
```

To run the simulation on the same file using SJF scheduler, you would start your simulator like this:

```
$ ./scheduler config.txt SJF
```

If the user does not provide correct arguments, your simulator will print out an informative error message and stop. For example, if the user specifies time-slice for SJF, you should report this as an error and abort the program:

```
$ ./scheduler config.txt SJF 3
SJF does not accept timeslice.
```

You must support the uppercase strings "RR" and "SJF", but if you wish you can also support lower case versions.

### Configuration file

The configuration file contains descriptions of processes that your simulator will schedule, each on a separate line. Each line contains 2 integers: the first one denotes the arrival time of the process, and the second one the CPU burst length. A sample configuration file `config.txt` is below:

```
1 10
3 5
5 3
```

This file contains information about 3 processes: P0, P1 and P2. The 2nd line “3 5” means that process P1 arrives at time 3 and it has a CPU burst of 5 seconds.

### Simulation

You can use the simulation loop pseudocode presented during lectures to write your solution. Your simulation should start at time 0. Please notice that in order to efficiently handle cases with large CPU bursts, arrival times and/or time slices, it will be impossible to advance your simulation by 1 time unit at a time.

### Simulation output

At the end of your simulation your program will output a compressed execution sequence of all processes. Any time the CPU is idle, your simulator should output "-" in the execution sequence. Make sure your program output matches these examples:

```
$ cat test1.txt
1 10
3 5
5 3
$ ./a.out test1.txt RR 3
Seq: -,P0,P1,P0,P2,P1,P0
```

```
$ cat test2.txt
$ ./a.out test2.txt SJF
Seq:
```

```
$ cat test3.txt
100000000000 1000000000000
300000000000  500000000000
500000000000  300000000000
$ ./a.out test1.txt RR 300000000000
Seq: -,P0,P1,P0,P2,P1,P0
```

```
$ cat test4.txt
5 10
6 6
14 1
50 17
$ ./a.out test4.txt SJF
Seq: -,P0,P2,P1,-,P3
```

### Limits

You may make the following assumptions about the configuration file:

- The processes are sorted by their arrival time, in ascending order.
- For output purposes you need to give the processes names in the format 'Px', where x is the process ID. Assign IDs consecutively starting from 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 compressed execution sequence will not have more than 100 entries.

### Hints

Start by making your simulator increment current time by 1. This should make your simulator work for small numbers. Then you can try to improve it to run fast for large numbers – by incrementing current time by an optimal value. SJF simulation will be easier to improve than RR, so start with that.

Link to sample input: https://gitlab.com/cpsc457/public/scheduler .

## Submission

Submit 2 files to D2L for this assignment:

| | |
|---|---|
| `deadlock.[c\|cpp]` | solution to Q1 |
| `scheduler.[c\|cpp]` | solution to Q2 |

## General information about all assignments:

1. All assignments are due on the date listed on D2L. Late submissions will be not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. 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.
5. 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, you can contact your instructor.
6. All programs you submit must run on `linux.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.
7. **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.
8. 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.
9. 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.