CPSC 457 - Assignment 3

Due date: **Friday, Oct 26, 2018 at 11:30pm**.
Individual assignment. No group work allowed.
Weight: 6% of the final grade.

Q1 – Written question (2 marks)

Suppose a program spends 75% of time doing I/O. Calculate the CPU utilization when running 8 copies of such a program at the same time. Assume one single-core CPU, and all I/O operations are executed in parallel.

Q2 – Written question (4 marks)

Consider the following set of processes (all times given in seconds):

Process	Arrival Time	Burst Time
P1	0	12
P2	2	1
P3	3	3
P4	5	1
P5	9	5

Draw a Gantt chart to illustrate the execution of these processes using the SRTN scheduling algorithm. Also calculate the average wait time.

In case of ties, use FCFS to break break them. This means that an older job in the system has priority over a newly arrived job. For example, if an older job needs to be inserted into the ready queue at precisely the same time as some newly arrived job, the older job will be inserted first.

Q3 – Written question (4 marks)

For the same set of processes as in Q2, draw a Gantt chart to illustrate the execution of these processes using the RR scheduling algorithm with 1sec quantum. Use FCFS to break ties. How many context switches are there?

Q4 – Written question (4 marks)

For the same set of processes as in Question 2, assume that P1 and P5 are high priority processes, P2 and P4 are medium priority processes, and P3 is a low priority process. Draw the Gantt chart to illustrate the execution of these processes using a multi-level feedback queue scheduling algorithm using 3 queues: the high priority level queue Q1 uses RR scheduling with 2s quantum, the medium priority level queue Q2 uses RR scheduling with 4s quantum, and the low priority level queue Q3 uses FCFS scheduling.

Notes:

- When P1 and P5 arrive, they go straight to Q1. When P2 and P4 arrive, they go straight to Q2. P3 goes straight to Q3.

- Processes can move down in priority if they exceed the corresponding time slice - for example P1 starts in Q1, but it could eventually end up in Q3.
- In general, processes can move both up and down in priority queues. However, for this question, processes cannot move up.
- In general, a running process belonging to a lower priority queue could be preempted by a process arriving into a higher priority queue. For this question you will ignore such source of preemption.

Q5 - Programming question – multithreaded (30 marks)

In Appendix 1 you will find a C program called `countPrimes.c`, which reads numbers from the standard input, and counts how many of them are prime. To compile this program, use:

```
$ gcc countPrimes.c –O2 -o count -lm
```

You can also use a C++ compiler:

```
$ g++ countPrimes.c –O2 -o count -lm
```

After you run the program you can type in the numbers. To indicate EOF you press <ctrl-d>:

```
$ ./count
Counting primes using 1 threads.
1 3 <enter>
5 <enter>
<ctrl-d>
Found 2 primes.
```

Notice that the input can have multiple numbers per line. To simplify testing, you can prepare different input files, and then use standard input redirect '<' to feed a file to the program. For example, you can store 4 numbers in a file `test.txt`:

```
123 321 13
11
```

To run the code on the file, instead of having to type in the numbers yourself, you can:

```
$ ./count < test.txt
Counting primes using 1 threads.
Found 2 primes.
```

In the above example the program found 2 primes (13 and 11). You can find additional test files in Appendix 2.

Your job is to improve the execution time of `countPrimes.c` by making it multi-threaded, using pthreads. Your program should take a single command line argument 'N', which will determine how many threads your program will be allowed to have running at any given time. Ideally, if your program takes time T to complete a test using 1 thread, then using N threads the program should only take T/N time to complete. For example, if it takes 10 seconds for a run with 1 thread, then it should take 2.5 seconds to run it with 4 threads. In order to achieve this goal, you will need to design your program so that:

- each thread is doing roughly equal amount of work;
- the synchronization mechanisms are efficient; and
- running time of your solution with 1 thread is the same as running time of the original non-threaded program.

The output of your program should match the output of the original program exactly. You may assume that there will be no more than 10,000 numbers in the input file, and that all numbers will be in the range [0 .. 2^63-2].

Please note that the purpose of this question is NOT to find a more efficient prime test algorithm. The purpose of this assigment is to parallelize the existing solution, using the exact same prime testing algorithm. You are free to use any parts of the `countPrimes.c` from the Appendix 1 in your program. However, you are **not allowed** to use any other code in your solution that you did not write yourself (unless it was explicitly provided for you by the intstructor or your TA).

Q6 – Written question (5 marks)

Time the original program as well as your solution on three files from Appendix 2: `medium.txt`, `hard.txt` and `hard2.txt`. For each input file you will run your solution 6 times, using different number of threads: 1, 2, 3, 4, 8 and 16. You will record your results in 3 tables, each looking like this:

Test file: `medium.txt`			
# threads	**Observed timing**	**Observed speedup compared to original**	**Expected speedup**
original program		1.0	1.0
1			1.0
2			2.0
3			3.0
4			4.0
8			8.0
16			16.0

The ‘Observed timing’ column will contain the raw timing results of your runs. The ‘Observed speedup’ column will be calculated as a ratio of your raw timing with respect to the timing of the original program. Once you have created the tables, eplain the results you obtained. Are the timings what you expected them to be? If not, explain why they differ.

Q7 – Programming question – single threaded (20 marks)

Write a program that simulates the execution of processes using two different scheduling algorithms: non-preemptive shortest-job-first and preemptive round-robin. Your program will start by reading the description of the processes from a cofiguration file, and then run the simulation. During the simulation your program will output the state of the processes at ever simulated time step. The scheduling algorithm that will be used in your simulation will be specified on the command line. You may make the following assumptions:

- There is only one CPU, i.e., only one process can run at a time.
- The processes in the configuration files 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 CPU-bound, i.e., a process will never be in the WAITING state.
- There will be between 0 and 30 processes in the input configuration file.
- Process arrival time will be in the range [0,100], and process burst in the range [1,100].

A sample configuration file `config.txt` is below:

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

This file contains information about 3 processes, each process on a separate line. Each line in the configuration file contains exactly 2 integers: the first one denotes the arrival time of the process, and the second one the burst length. For example, the 2nd line "3 5" means that process P1 arrives at time 3 and it has a burst of 5 seconds.

Your program will accept 3 command-line arguments: (1) the name of the configuration file, (2) the name of the scheduling algorithm ('RR' for round-robin or 'SJF' for shortest job first), and (3) the time quantum for the round-robin scheduling algorithm. Sample command-line arguments for your program are given in this example:

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

Your program will schedule the jobs using the scheduling algorithm specified on the command-line arguments (round-robin with time-slice of 3 seconds in the above example). Your program must output the states of all processes every simulated time step to standard output.At the end, the program also reports the time that each process spent waiting in the ready queue as well as the overall average waiting time. Below is a sample output for the above command line arguments and the `config.txt` file. The first column in output.txt is the number of seconds into the execution of the program, and there is one column for each process. Use "+" to denote the READY state, "." to denote the RUNNING state, and a white space " " to denote a non-existing process or a terminated process. The output of your program should be nicely aligned:

```
$ ./scheduler config.txt RR 3
Time P0    P1    P2
------------------------------------------------------------
 0
 1   .
 2   .
 3   .     +
 4   +     .
 5   +     .     +
 6   +     .     +
 7   .     +     +
 8   .     +     +
 9   .     +     +
10   +     +     .
11   +     +     .
12   +     +     .
```

```
13   +     .
14   +     .
15   .
16   .
17   .
18   .
------------------------------------------------------------
P0  waited 8.000 sec.
P1  waited 7.000 sec.
P2  waited 5.000 sec.
Average waiting time = 6.667 sec.
```

Hints:

Recall that a process is always in one of 3 states: READY, RUNNING or WAITING, alghough in this question you will ignore the WAITING state. Your program should maintain a ready queue to keep track of the processes according to the scheduling algorithm given on the command line. When a process is picked by the scheduler, its state is changed from READY to RUNNING. If a process is preempted, its state is changed from RUNNING to READY, and it is placed back in the ready queue at the appropriate spot. If a process finishes its execution, it should be removed from your simulation.

Notes on the command line arguments:

Your program must accept some parameters on the command line. Please make sure that you error check this user input! If the user does not provide correct arguments, you should print out an informative error message and abort the program.

You must support the uppercase strings "RR" and "SJF". If you also want to accept lowercase "rr" and "sjf", that's OK.

It is up to you how you will handle the 4th command line argument for "SJF" scheduler. The sensible options are:

- If the user specifies time-slice for SJF, you report this as an error and abort (I suggest this option).
- If the user specifies time-slice for SJF, you ignore it, although I recommend adding a warning message if you decide on this option.

Submission

You should submit 3 files for this assignment:

- Answers to the written questions combined into a single file, called either `report.txt` or `report.pdf`. Do not use any other file formats.
- Your solution to Q5 called `count.c` or `count.cpp`.
- Your solution to Q7 called `scheduler.c` or `scheduler.cpp`.

Since D2L will be configured to accept only a single file, you will need to submit an archive, eg. `assignment3.tgz`. To create such an archive, you could use a command similar to this:

```
$ tar czvf assignment3.tgz report.pdf count.c scheduler.cpp
```

General information about all assignments:

- All assignments must be submitted before the due date listed on the assignment. Late assignments or components of assignments will not be accepted for marking without approval for an extension beforehand. What you have submitted in D2L as of the due date is what will be marked.
- Extensions may be granted for reasonable cases, but only by the course instructor, and only with the receipt of the appropriate documentation (e.g. a doctor's note). Typical examples of reasonable cases for an extension include: illness or a death in the family. Cases where extensions will not be granted include situations that are typical of student life, such as having multiple due dates, work commitments, etc. Forgetting to hand in your assignment on time is not a valid reason for getting an extension.
- After you submit your work to D2L, make sure that you check the content of your submission. It's your responsibility to do this, so make sure that you submit your assignment with enough time before it is due so that you can double-check your upload, and possibly re-upload the assignment.
- All assignments should include contact information, including full name, student ID and tutorial section, at the very top of each file submitted.
- Assignments must reflect individual work. Group work is not allowed in this class nor can you copy the work of others. 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.
- You can and should submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It’s better to submit incomplete work for a chance of getting partial marks, than not to submit anything.
- Only one file can be submitted per assignment. If you need to submit multiple files, you can put them into a single container. The container types supported will be ZIP and TAR. No other formats will be accepted.
- 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.

Appendix 1 – countPrimes.c for Q5 and Q6

```
/// counts number of primes from standard input
/// compile with:
///   $ gcc countPrimes.c -O2 -o count -lm
/// by Pavol Federl, for CPSC457 Spring 2017, University of Calgary
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

/// primality test, if n is prime, return 1, else return 0
int isPrime(int64_t n)
{
     if( n <= 1) return 0; // small numbers are not primes
     if( n <= 3) return 1; // 2 and 3 are prime
     if( n % 2 == 0 || n % 3 == 0) return 0; // multiples of 2 and
     int64_t i = 5;
     int64_t max = sqrt(n);
     while( i <= max) {
         if (n % i == 0 || n % (i+2) == 0) return 0;
         i += 6;
     }
     return 1;
}

int main( int argc, char ** argv)
{
    /// parse command line arguments
    int nThreads = 1;
    if( argc != 1 && argc != 2) {
        printf("Uasge: countPrimes [nThreads]\n");
        exit(-1);
    }
    if( argc == 2) nThreads = atoi( argv[1]);
    /// handle invalid arguments
    if( nThreads < 1 || nThreads > 256) {
        printf("Bad arguments. 1 <= nThreads <= 256!\n");
    }
    if( nThreads != 1) {
        printf("I am not multithreaded yet :-(\n");
        exit(-1);
    }
    /// count the primes
    printf("Counting primes using %d thread%s.\n",
           nThreads, nThreads == 1 ? "s" : "");
    int64_t count = 0;
    while( 1) {
        int64_t num;
        if( 1 != scanf("%ld", & num)) break;
        if( isPrime(num)) count ++;
    }
    /// report results
    printf("Found %ld primes.\n", count);
    return 0;
}
```

Appendix 2 – test files for Q5 and Q6

easy.txt

```
1 2 3 4 5 6 7 8 9 10
100 101 102 103 104 105
106 107
```

medium.txt

```
9007199254740997 8796451843471 2 900719925474105073 90071996392800899
4 90071992547410493 90071992547410613 900719925474105161 900719925474104987
1 90072054854210657 900719925474104927 9007199254741049 900720046819564759
90071992547410513 9007207298190743 9007206203142577 5 3 900719925474105013
9007199254740881 8796254359103 90071992547410663 9007199254741033
900719925474104977 90071992547410609 900720004735038979 8796093022151
```

hard.txt

```
1 2 3 4
9223372036854775783
5 6 7 8 9
```

hard2.txt

```
9223371873002223329 1 2 3 4 5 6 7 8
```