RMIT UNIVERSITY

COSC 1114 Operating Systems Principles
Semester 2, 2021
Programming Assignment 1

Assessment Type	Individual assignment. Submit online to Canvas → Assignments → PrgAsg1 . Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
Submission Due Date	End of week 7.
Marks	100 (30% of semester)

1. Overview

In this assignment, you must write C/C++ code to implement a classic concurrency problem. Concurrency means "at the same time" so each of these problems involve some level of simultaneous processing. The code must use system calls to implement multi-threading to code the problems.

2. Learning Outcomes

This assessment relates to all the learning outcomes of the course which are:

- Summarise the full range of considerations in the design of file systems, summarise techniques for achieving synchronisation in an operation system,
- Compare and contrast the common algorithms used for both pre-emptive and non-pre-emptive scheduling of tasks in operating systems, such a priority, performance comparison, and fair-share schemes. Contrast kernel and user mode in an operating system

3. Assessment details

You must implement two of five classic concurrency problems. While these problems have strange names like "Dining Philosophers" and "Sleeping Barber" they represent real problems you may face in your programming career that you may need to solve. You will learn from implementing a solution how to solve problems of concurrency.

In the process, you will come to understand the problems of debugging multi-threaded code.

Your solution to the problem you choose must not use busy waiting, which is where a program sits in a tight loop until some condition is met as this wastes CPU time. Instead, you must have each thread sleep until a condition is met (use a condition variable), wake up and do some processing and then perhaps signal other threads that the job has been done. Please ensure sufficient output so that it is clear when your program performs any actions such as adding to an array, locking a lock, etc.

You should ensure in your design of the program that you only share between threads the minimum state (variables) possible as the more information that is shared the more likely there is to be a conflict (deadlocks, race conditions, etc). Please see the courseware material for more detail on these problems. If your algorithm requires randomness, you should ensure you seed the random number generator correctly.

To ease marking of your assignment, you must call your executable "simulation" and there must not be any subfolders in the submitted zip file. All files must be in the root of the zip file.

Makefile

Your solution must be written in C / c++ and you must supply a makefile that builds your solution incrementally and then links it together. If you are feeling a big rusty about Makefiles or have not used them before, we recommend going through the following tutorial:

https://www.tutorialspoint.com/makefile/index.htm

Compiler Settings

Please note that as a minimum you must use the -Wall flag to generate warnings. You may use any supported c++ compiler on the server - if you wish to use a standard above c++11 on the server with g++ you will need to use the scl command, e.g.:

```
scl enable devtoolset-9 bash
```

Note: if you use a shell other than bash you can replace bash with your own preference.

Graceful Exit

Regardless of the problem you choose to implement, you will need to gracefully exit your simulation at the end of ten seconds. Use the following method to do this: once you have launched all the required threads from main, call the sleep function, to specify sleep for ten seconds. Once the sleep function finishes, change the value of a global variable to indicate that all the threads should exit.

Valgrind

The solution you submit will ideally be as bug-free as possible including free of memory bugs. Your markers will mark your submission on the titan/jupiter/saturn servers provided by the university and we will use the tool "valgrind" to test that your program is memory correct (no invalid access to memory and all memory freed) and as such you should check your program on titan before submission with this tool. The following command:
valgrind --track-origins=yes --leak-check=full --show-leak-kinds=all ./simulation
Will run your program with valgrind. Please note that in this case the name of the executable produced by compilation of your program is 'executable', but it can be named whatever you like - just make it a sensible name, given the task you are implementing.
Allowed Concurrency Functions

For the concurrency part of the assignment, you must limit yourself to the following functions:

- pthread_create
- pthread_join
- pthread_detach
- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_trylock
- pthread_mutex_lock
- pthread_cond_wait
- pthread_cond_signal
- pthread_cond_init
- you may also need the pthread_mutexattr_* functions

This list is not exhaustive and so if you are unsure, please ask on the discussion board about the function you wish to use.

Please note that in practice beyond this course, you would use higher-level c++ functions that call these functions but part of what you are learning here is the underlying calls that are made as part of managing concurrency. That is, part of what you are learning is to apply the theory that you learn in tutorials to practical problems.

4. The Problems

Below you will find a selection of five concurrency problems to choose from. You must select two problems as follows.

- Select A or B, and one of C,D,E to implement for your submission.

In each case, the simulations should run for ten seconds and terminate cleanly - no crashes, no deadlocks, no race conditions. You should also print a line of text for each action in your program such as adding an element to an array.

All normal messages should be printed to stdout (use cout) and error messages should be printed to stderr (use cerr). This will allow your marker to capture normal output and error output separately if they wish to.

A: The Producer-Consumer Problem

Create an array of size 10 which are "buckets" that hold items that have been produced - e.g., random numbers, random strings, etc. Have five threads that are concurrently producing items to fill the available buckets and five threads that are concurrently consuming those items - say, printing them out to the screen would be fine. Have the program run for 10 seconds and then clean-up and exit without crashes, race conditions or deadlocks.

B: The Readers and Writers Problem

Consider a single resource which has multiple threads which need to read from or write to that resource. We can only allow a single writer to write data to that resource. However, it is fine to have multiple readers reading at the same time, but no writer is allowed to write while there are readers reading.

Have a single resource (an integer is fine) with 5 reader threads and 5 writer threads. The rules for access to the resource must be as outlined above. Provide sufficient output so it is clear the decisions that are made by your program. As with other implementations, kill it off after ten seconds and your program should exit gracefully with no race conditions or deadlocks.

C: The Dining Philosophers' Problem

This problem models needing two of a particular resource in order to complete a task.

A suggested solution is as follows:

- create an array of 'forks' - say, five fork objects (for this exercise, these should be mutex locks).
- create an array of threads, the same amount as the number of forks.
- for the required duration of the program, say ten seconds:
    - sleep ('think') for a random number of milliseconds and try to grab the first fork (C1 for the first philosopher, C2 for the second philosopher, etc). If it is not available, sleep for a random number of milliseconds and try again.
    - sleep ('think') for a random number of milliseconds and try to grab the second fork (for thread 0, that would be lock 4 (wrap around to the largest lock), for thread 1 that would be lock 0, etc).
    - Once both locks have been attained, sleep ('eat') for a random number of milliseconds and then release both locks.

DI: The above does NOT resolve deadlock in all cases. Find and Fix it so that it does.
HD: The algorithm is not always fait in that some philosophers can get to eat more than others due to timing and sequence. Make it fair.

D: The Sleeping Barbers Problem

In this problem, have an array of 5 slots. Have a barber thread that is the one that will be servicing the other threads. Use a condition variable to send the barber to sleep until a customer arrives.

At random time intervals (in milliseconds), a customer will enter. If there is a seat available, they will take up that seat and if there is no space, it will leave. When a customer arrives, and the barber is asleep it should be woken up (via the condition variable).
Once the barber is awake, it will serve each customer, in order. Serving a customer in this case can be simulated by sleeping for a random number of milliseconds. Once a customer has been serviced, they should be removed from the array to make space for another customer and the barber will move on to the next customer available. When there are no customers left to be served, the barber should go back to sleep until a new customer arrives.

CR: Have more than 1 seat (say, 4 seats) in the waiting room).
DI: Have more than 1 barber, customers check for full waiting rooms.
HD: Make it fair. Ensure all barbers get an equal amount of work for a given (large) set of customers

E: <u>The Cigarette Smoker's Problem</u>

Three smokers possess a large supply of one of three ingredients for cigarettes – paper, tobacco and matches. A non-smoking agent takes an item from two of the smokers at random, and places them on the table. Each smoker checks the items on the table one at a time to see if it is their turn. The third smoker combines the items on the table with his own, and smokes. When finished, the agent places a new, possibly different pair of items on the table.

This is a variation of the dining philosopher problem, as it uses an agent. Explain how thisworks.
DI: Demonstrate how to avoid deadlocks
HD: Make it fair. Each smoker gets to smoke.

5. Report and GitHub

You are to use Github as your project space. Make it a private project, and include your tutorial tutor as a collaborator.

In additional to the program solution, you are required to write a DOCX/PDF report describing the following.

1. Identify yourself using student ID, Name, and Github project name/url
2. Describe any issues and limitations of your implementation.
3. The program solution implements two algorithms.
    a. One from A,B and one from C,D,E
4. For each algorithm
    a. Describe in detail how each algorithm works, and how its design avoids things like deadlock, starvation, etc.
        i. Ensure that all variables have meaning, and describe these meaningful variables here.
    b. Describe in great detail, 2 real-world industrial / business scenarios, where each algorithm applies.
        i. Describe how your variables should translate to such an environment. For example, what would the Barber become?

6. Submission

Submit a ZIP file containing

1. Makefile
2. a complete set of source files needed to build your solution
3. A Readme file explaining what needs to be done other than just calling 'make' to run your program.

A separate Report in DOCX/PDF format as per previous section

Students are advised and expected to make 3 submissions of the ZIP file, in week 5,6,7, with 7 being the final submission. Only the final submission will be used for marking, but the others are needed to demonstrate progress.

Assessment declaration:

When you submit work electronically, you agree to the assessment declaration:

https://www.rmit.edu.au/students/student-essentials/assessment-and-exams/assessment/assessment-declaration

7. Academic integrity and plagiarism (standard warning)

It is your responsibility to ensure that all files you submit are your own work. We will check your submission against other submissions using automated software to check for plagiarism. You must agree with the RMIT Assessment declaration available here:

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarized, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods,

- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites.

If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviors, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity

8. Rough Marking Guidelines

A rough marking guide is described below. A more detail marking guide will be attached to the assignment spec, and it will be the final guide.

Report,docx **(8 marks)**
As per report section.

***Makefile* and version control** **(2 marks)**
You must submit a Makefile to compile your assignment. It must build each source file into a compiled object file and then link them all together. Project must be on Github and there must be evidence of version control.

Compilation ***(2 marks)***
Your program must compile without errors or warnings (compiler option -Wall must be used).

Programming Style **(2 marks)**
You must follow good programming style - avoid use of "magic numbers", name your constants, variables and functions meaningfully, etc.

Synchronization Primitives **(4 marks):**
use the correct objects and methods from the pthread library to implement threads, locking and sleeping.

Basic Implementation **(3 marks)**
provide a solution that is well designed and modularised that solves problems A or B.

Further implementations **(3 / 6 / 9 marks)**
provide a solution that is well designed and modularised that solves the problems that you have chosen for problems C, D or E. The level CR / DI / HD also indicates the extra parts that are needed for the extra marks.

In each case, half the marks will be for your first algorithm and half for the second algorithm. For each of the criterion, there are 4 possible marks:

- **Excellent: the solution you have provided is correct as per assignment requirements.**
- **Good: the solution you have provided is good but there are one or two minor issues with your implementation. Your marker will detail the problems in the provided implementation.**
- **Not so good: you gave it a go but it's a long way from the required implementation.**
- **No marks: no implementation provided.**

Late Submission Policy: 10% of the available marks will be deducted for every day late. Please note that this includes days on the weekend.

Note that the published rubric for this assignment will apply, even if there are small differences with the marking described herein.