COSC1112/1114 Operating Systems Principles

Semester 2, 2019

Assignment 2 – Multithreaded Allocator

A technique that has become much more important over the past few years, especially with the end to increases in speed in single core programming (single core programs run about as fast as they did a decade or so ago), is the technique of multithreaded programming.

You need to modify an assignment 1 solution (either yours or ours) using the posix threads library (higher level libraries won't be accepted for this assignment, such as c++11 threads as it would allow you to avoid some of the things we want you to consider).

As with the first assignment we will allow you to develop your solutions in either C or C++ as the libraries used will be the same.

Academic Integrity

All work you submit for this course must be your own work unless otherwise credited. Even so, we cannot give you marks for work that is not your even if credited. It is a condition of submission of assignments that you agree with RMIT's Assessment Declaration available here: https://www.rmit.edu.au/students/student-essentials/assessment-and-exams/assessment/assessment-declaration

All submissions will be checked for academic integrity.

Programming Task

Consider the memory allocator to be used for a C or a C++ program (they are not the same language) as a shared resource as it definitely is. We want to make our memory allocator thread safe (as malloc and new are out of the box). What this means is that we must implement locking mechanisms on the data structures we implemented for assignment 1 (just the allocator – nothing else that is changeable should be shared).

Solution Design

Just as with the first assignment, you are expected to come up with a design to make this work. You may consider splitting the data structures that are part of the allocator into several files or data types if you think this is appropriate. On the other hand you might just put the required additional code in the same files as before. You will need to explain and defend this in your report.

Locking

You should investigate the locking functions available in the posix threads library (pthread.h) and how you might use these for your solution. In particular I found pthread_mutex_* (for implementing mututal exclusion. I also found the pthread_cond_* functions useful for implementing a condition variable for getting the read and write locks to work well together. The only functions you might use to implement locking outside these libraries are the posix semaphore libraries ie, in sys/sem.h or semaphore.h. If you are unsure about allowed libraries please ask on the discussion board.

Read Locks

Whenever accessing data structures such as the free and alloc lists, you must lock the whole datastructure for reading – what that means is any new readers can also read from that list but once a writer requests access, all current readers must be allowed to complete but any new readers must wait.

Write Locks

Whenever a thread wants to modify the alloc and free lists, it must gain exclusive access to the list by using a semaphore / mutex to lock it. No other threads may modify the lists while the current thread has them locked for writing.

Maximise Concurrency

It is important that you maximise concurrency in your implementation. Not every lock of the alloc list requires a lock on the free list or vice versa. A read lock should not lock out other readers.

No Race Conditions

You should avoid any race conditions in your code. A race condition is where given the same set of inputs different outcomes are possible. Your code must be deterministic – given a certain set of events in a certain order, the same outcomes must result.

Deadlock-free

Your code must avoid deadlocks using the techniques discussed in this course.

Code For Your Experiment

As with assignment 1, you will need to develop code to test your new mutli-threaded allocator. Consider the properties of your allocator such as parallelism and speed. You should modify your test data from assignment 1 to adequately test the new allocator so you can make an adequate comparison with the allocator from assignment 1 (either yours or ours).

Make File

As with your first assignment, you must construct a Makefile that separately builds each source file and then links together these files along with any external libraries you have used.

Coding Standards

Your code should comply with good coding standards. Your code should be a maximum of 80 columns, should use meaningful variable names, avoid magic numbers, should be broken down into reasonable components and be well commented.

Please note that a submission which cannot be compiled will attain a fail mark.

Your Report

This section is expected to be in a well formatted report. You should provide a cover page and table of contents and clear heading, and well formatted tables and graphs if you feel you need them.

Provide a description of the method and reasoning behind your choices of experimental data.

Based on your experiments, provide data on the performance of the various memory allocation strategies.

We are particularly interested in any performance differences with each of the three allocation strategies when compared with the program for the first assignment. Does the allocator's performance greatly benefit from concurrency or not at all? How much concurrency is there? How does this relate to Amdahl's law in terms of the maximum benefit of this concurrency?

Which memory allocation strategy was best? Has concurrency changed this?

Marking Guide

Please note that a program that cannot be compiled results in a fail mark. So please develop and test in the target environment and provide adequate instructions on how to build and run your application.

The code

- Demonstration in week 10 or 11 (5 marks) – demonstrate the use of read and write locks on the two lists. You don't need to have race free or deadlock free code at this point.
- Solution Design (10 marks)
- Read locks (10 marks)
- Write locks (9 marks)
- No Race Conditions (8 marks)
- Deadlocks (8 marks)
- Experiment code (10 marks)
- Makefile (5 marks)
- Code standards (5 marks)

Submit your code in a zip file including your Makefile. Submitting in another archive format will be penalised.

The Report

Please note that a report that lacks experimental evidence but is based on solid research cannot get above 50% of the mark for the report. All materials outside of provided course materials that are accessed must be referenced and a bibliography provided. You must submit your report via the provided turnitin submission link

- Report formatting (5 marks)
- evidence presentation (5 marks)
- Discussion of Results (5 marks)
- Particular discussion about concurrency(5 marks)
- Comparison with results in part 1 (5 marks)
- Conclusions (5 marks)