The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

COMPUTER SYSTEM PROGRAMMING
ECE454 – FALL 2023
ASSIGNMENT-4

Dues are defined at the end of this guideline.

Objectives

- Understanding/Implementing Parallel Processing through Multi-Threading
- Understanding/Implementing Locking Process

Note- This assignment has two parts (1 & 2) and the following is Part-2. The guideline for the Part-1 was already posted on Quercus. Any clarification and revisions to this assignment will be posted on Quercus,

4.2 Implementing Concurrency and Locking (A4 – Part-2)

4.2.1-Coarse-grained and Fine-grained Locking

Data structures are often larger than a single memory location. So how can an entire data structure be protected while it is being accessed by several requests at the same time?
If there is one Mutex protecting an entire data structure in a program, then it can be implemented using Coarse-grained Locking process. And if there are many Mutexes, like one per node in a data structure in a program, needs to be read or written, then it can be implemented using Fine-grained Locking process.

Both Coarse-grained and Fine-grained locking processes have their own advantages and dis-advantages, such as: Coarse-grained locking is easier in implementation by providing only one lock, but it has lower performance as there is not much concurrency. Fine-grained locking can reduce conflicts and maximizing parallelism in operations on shared data structures, but can also increase the code complexity, errors and execution overhead.

In this section, you will implement a specific data structure and then try to access a shared part of it through applying several operations at the same time. To protect your data structure during those simultaneous operations, you will implement various locking techniques of Coarse-grained and Fine-grained as follows: [15 Marks]

1) Implement a single-linked-list with sorted values using the *Node* and *List* structures below. Insert these values in the linked-list: 40, 50, 100, 120, 160, 180 using a regular insertion process.
2) Try to have two more insertions for values 65 and 77 but this time simultaneously. Both of these nodes should be inserted at the same time between the nodes 50 and 100 and so two threads should be considered for these operations, one for each node to be inserted. These two threads will try to operate on the same links at the same time and so one of the insertions may be lost. To protect your linked-list, provide the following two Locking algorithms:

   a) Implement a single Global-Lock on the whole linked-list data structure. Apply the lock before each insertion/deletion (before the operation is started) and unlock it after each insertion/deletion (after the operation is completed) where/when it is needed.
   b) Implement a Fine-grained Lock on the individual nodes of the linked-list data structure. Apply the locks (on the nodes before and after that specific node to be inserted/deleted) when/where needed.

3) Measure the time for insertion of the nodes (both nodes at the same time) in step2 above for each algorithm (a) and (b). For each a&b measure the time in 5 different runs, and find the average. Record the results in a proper table.

4) Compare the processes in (a) and (b), and provide the advantages and disadvantages of each of them.

Note1-Your program should be written using C++ object-oriented language.
Note2-The following header files and structures should be used in your program:

```
#include <iostream>
#include <thread>
#include <mutex>
…
struct Node {
       int value;
       Node* next;
};

struct List {
       Node* head;   // point to head-node of the whole linked-list
       m1 lock_st;   // lock status field for the whole linked-list
};
```

// Implement the locks using C++ class std::mutex:
lock( ) and unlock( )

4.2.2-Spin-Lock vs other Locks

The purpose of a Spin-Lock is to prevent multiple threads from concurrently accessing a shared data structure. In Spin-Lock, the threads will busy-wait and waste CPU cycles instead of yielding the CPU to another thread.
So "*It is not recommended to use Spin-Locks unless the consequences can be identified and recognized certainly and clearly in advance.*"

The main aim in Spin-Lock is to make sure that one access to the shared data structure always strictly "happens before" another one. And the usage of acquire/release in lock/unlock operations are required and sufficient to guarantee this ordering. To emphasize the details about the locking process in both Mutex-Locking and Spin-Locking, here are some points for a quick review:

-When a new thread tries to lock a Mutex and the Mutex was already locked, then the system will move that thread to sleep state. And when the Mutex is unlocked by the thread which currently holds lock on it, then the other thread can be awakened from its sleep state. In case the system puts a thread to sleep state, it will also create a context switch and a kind-of performance degrading for that thread.

-When a thread tries to lock a Spin-Lock and fails, it will continuously try to lock it until it succeeds. This means that during this time, the system does not allow another thread to take its place and will not move the thread to sleep state. During this time the operating system may also switch to another thread when the cpu time for the first thread has been exceeded its run time quota, but will avoid creating a performance degrading to be recorded for that thread as it is in the technique above (Mutex-Locking).
Also, since the thread will be busily spinning on a blocked spinlock, it makes no difference as if the CPU cycles are spent for exchange or acquire or other operations. The reason is because in any of those operations if there are no memory-synchronizing instructions or tasks, then they may cause to dominate the memory subsystem and degrade the performance of other system components too.

In this section, you will implement a specific data structure and then try to access a shared part of it through applying several operations at the same time. To protect your data structure during those simultaneous operations, you will implement Spin-Locking technique as follows: [15 Marks]

1) Implement a class named "Spinlock" containing: two methods "lock()" and "unlock()", and an attribute with atomic type to represent the status of a shared part of memory: if it is in the acquired mode (locked) or in the released mode (unlock). Also define a vector of threads (max 5 elements).

Note-Feel free to add any other attributes or methods required in this class. You can also define extra classes and variables in your program, if required.

2) Simulate the Spin-Lock process while those threads (#5) trying to have access to a shared function Task() in the program. For each thread if it cannot lock the Task() because it was already acquired by another thread, it should go to the spinning mode. In this mode, a related counter to that thread (could be a simple counter variable) will start counting incrementally from zero until the time that thread is out of the spinning mode. Once that thread is out of the spinning mode or the Task() is released from its previous lock, that thread can acquire the Task() this time and lock it.
What is returned from this spinning process, is the related counter to each thread (created during its spinning mode). There could be other messages or information printed during this process according to your algorithm. Consider the function Task() with all read/write and other instructions to take 5 ms execution time. You can set the time for this function to be just a fixed delay.

3) Measure the time for the spinning mode of each thread, starting at the time the spinning starts or its related counter starts to count until the time the spinning is done. Record this time for each thread in 5 different runs and find the average for each of them. Record also the results for all threads in a proper table.
Note-The execution time should be calculated inside the code and not from the compilation options' results.

4) Also measure the total time for each thread's working, from being activated and then going to spinning mode, to working with function Task() and being done/inactive. Record this time for each thread in 5 different runs and find the average for each of them. Record also the results for all threads in a proper table.
Note-The execution time should be calculated inside the code and not from the compilation options' results.

Note1-Your program should be written using C++ object-oriented language.
Note2-The following header files and structures should be used in your program:

```
#include <iostream>
#include <thread>
#include <atomic>
#include <vector>
…
//Implement the locks using std::memory_order_ defined in header <atomic> for:
   memory_order_acquire,
   memory_order_release
```

Deliverables & Submission for Assignment-4 (Part-2) [Total 30 Marks = weight 5%]
-The teams should submit their final report through the submission folder on Quercus and then present their work in the labs (on the same day) to be evaluated by the TAs through Q/A by Thursday Nov23, at 8:59am [30 Marks]
-Your final report should include: all the source codes with proper comments (in plain text), all the results' tables with proper definition and discussion (in word or pdf), a screen-shot of each result.
Note-*This should be noted that the evaluation for this assignment A4-Part2 will be done* ***competitively*** *meaning that the best submission will get the best mark and the other submissions will get the marks relatively compared to the best one.*
-Make sure to add the names and student numbers of all Team members, and the Lab section your team attends at the top part of your final report. Only ONE final submission is required from each team.
-The teams should save all their answers in a final report and submit it under this naming rule: "PRAxx_Tyy_Az.txt" where "xx" is the Lab section (01 for 12-3pm, and 02 for 9am-12pm), "yy is the Team number and "z" is the Assignment number, e.g. "PRA01_T06_A3.txt" is the name for submission from Team06 for A3 working in the Lab on Thursday 12-3pm.

Good Luck.
Useful Resources:
https://en.cppreference.com/w/cpp/atomic/memory_order
https://thispointer.com/c11-multithreading-part-2-joining-and-detaching-threads/
https://www.geeksforgeeks.org/multithreading-in-cpp
https://stackoverflow.com/questions/58443465/stdscoped-lock-or-stdunique-lock-or-stdlock-guard