Northeastern University
College of Engineering
Department of Electrical & Computer Engineering

EECE7376: Operating Systems: Interface and Implementation

Homework 5

Instructions

- For programming Problems:
  1. Your code must be well commented by explaining what the lines of your program do. Have at least one comment for every 4 lines of code.
  2. At the beginning of your source code files write your full name, students ID, and any special compiling/running instruction (if any).
  3. Before submitting the source code file(s), your code must compile and tested with a standard GCC compiler (available in the CoE Linux server).
  4. Do not submit any compiled object or executable files.

- **Deliverables**: Submit the following to the homework assignment page on Canvas:
  1. Your homework report developed by a word processor and submitted as <u>one</u> PDF file. For answers that require drawing and if it is difficult on you to use a drawing application, which is preferred, you can neatly hand draw the answer, scan it, and include it into your report. The report includes the following (depending on the assignment contents):
     a. Answers to the non-programming Problems that show all the details of the steps you follow to reach these answers.
     b. A summary of your approach to solve the programming Problems.
     c. The **screen shots** of the **sample run(s)** of your program(s)
  2. The well-commented source code files (i.e., the .c or .h files) that you add or modify.

  Do NOT submit any files (e.g., the PDF report file and the source code files) as a compressed (zipped) package. Rather, upload each file individually.

<u>Note</u>: You can submit multiple attempts for this homework, however, only what you submit in the last attempt will be graded. This means all required files must be included in this last submission attempt.

Problem 1 (40 Points)

In this problem you will examine the Michael & Scott Concurrent Queue code presented in the course slides. In your code main function create one queue and initialize it. Also, in the main function create at least two threads to run the **Queue_Enqueue** function and at least two more threads to run the **Queue_Dequeue** function.

a) Test your program without using any locks. Simulate a sequence of enqueue and dequeue operations. Comments on the results and highlights the data race problem that is expected to occur.
b) Compare the performance of your program by using two locks (as in the slides) as opposed to using only one lock for both the enqueue and dequeue operations. Simulate a large number of overlapped (not sequential) enqueues and dequeues operations by all threads and use the system time (this is for simplicity as it is not the best way) to compare the performance of these two approaches. Comments on the results.
c) Why do we need a dummy node in case two locks are deployed? Support your answer by testing the code without using a dummy node.

Attach your programs as three files named `h5-p1-nolock.c`, `h5-p1-onelock.c`,and `h5-p1-twolocks.c`

Problem 2 (40 Points)

The goal of this problem is designing a multithreaded reduction algorithm that calculates the sum of all elements in a vector, providing a single scalar value as a result. Use the following guidelines for your implementation:

a) Use a global variable `sum` to accumulate the partial sums calculated by each individual thread. Declare a global mutex called `sum_mutex`, which should be locked by each thread before accessing variable `sum`. Initialize the global mutex as explained in the course slides.
b) Declare a data type called `struct runner_args_t` that contains the set of arguments passed to each thread when the thread is created. The structure has the following fields:
    - Field `v`, representing the base address of the array of integers to work on.
    - Field `index`, representing the index of the element in array `v` that a thread will start summing from.
    - Field `size`, indicating the number of elements that a thread is responsible for summing.

c) Write a `runner()` function containing the code to be executed by each thread. The function should perform the following actions:
   - Cast its generic input argument of type `void *` into a pointer to a structure of type `struct runner_args_t *` and obtain its input arguments from the structure fields.
   - Traverse `size` elements of vector `v` starting at position `index` and calculate their sum as a partial result in a local variable.
   - Accumulate the partial result into global variable `sum`. Lock the mutex before accessing this global variable then unlock it.
   - Free the thread's arguments structure as it should be allocated dynamically as explained below.

d) Write a `main()` program that creates a vector, computes its sum in parallel, and prints the result. The program should run the following intermediate actions:
   - Read the number of desired elements (`num_elements`) and the number of desired threads (`num_threads`) from two arguments passed by the user when running the program. If the program was not executed with exactly two arguments, print an error and exit.
   - Dynamically allocate a vector with the desired number of elements and initialize each element of the vector to a value equal to its index (i.e., `v[0]` = 0, `v[1]` = 1, etc.) This initialization guarantees predictable and verifiable results.
   - Spawn the desired number of threads. For each thread, allocate a new `struct runner_args_t` dynamically, and initialize it with the appropriate arguments for that thread. The computation burden should be evenly distributed among threads.
   - Wait for all child threads to complete.
   - Free the vector and print the result.

This is an example of the execution of the program, in which a vector with 10,000 elements is created, and its sum is calculated with five concurrent threads:

```
$ ./main 10000 5
Sum = 49995000
```

Test your program with more examples that cover different scenarios of the possible inputs. Attach the full program in a file named `h5-p2.c`

Problem 3 (20 Points)

The following table represents the current state of a system in which four resources *A*, *B*, *C* and *D* are needed by the shown four processes. The system contains the following total instances of each resource: 6 of *A*, 4 of *B*, 4 of *C*, 2 of *D*.

Processes	*Allocation*				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0	2	0	1	1	3	2	1	1								
P_1	1	1	0	0	1	2	0	2								
P_2	1	0	1	0	3	2	1	0								
P_3	0	1	0	1	2	1	0	1								

Answer the following questions using the Banker's algorithm:

a) Fill the missing entries in columns *Need* and *Available* in the above table.
b) Is the system in a safe state? Explain why.
c) Can a request from P_2 of (2,2,0,0) be granted? Explain why.