CSC 246 Spring 2019 Homework 3

Due: February 18 2019, 11:55PM

Due for Homework 3.1: March 4 2019, 11:55PM

ChangeLog

02/25: Homework 3.1 online.

02/12: deleting "but we will be using 1+ instead" in Problem 3. adding one more hint on task orgnizaitons.
02/05: removing two questions for Problem 1, and it is now final.
02/05: homework 3 online. We will have our first midterm on February 19. Please plan accordingly.

Instructions

(a) You can work on any current Linux installation. We will use the EOS Linux machines (one of the Linux desktop machines in EB 3 or Daniels, or via remote-linux.eos.ncsu.edu) to do the grading. If you are using Linux, type "ssh [unity_id]@remote-linux.eos.ncsu.edu" to access the machines remotely. For Windows, use "putty" to ssh into remote-linux.eos.ncsu.edu. Use your campus password to get through.

(b) Makefile and README: Use Makefile to compile all source files of your code. Use gcc as your compiler. Also write a README explaining what your code does and how to compile and execute it. The README is in addition to the comments in your source files at critical points in your code explaining what does that piece of code do. You can refer to the lab tutorial chapter of the OSTEP book for some help on Makefile.

(c) File names: For programming problems, name your source files as prob_x.c and header files as prob_x.h, where x is the question number here. Try to just use one .c and one .h file for each problem. For non-programming problems, put the anwser to one single ASCII format text file and name it problems.txt. Please submit ASCII format text file ONLY for non-programming problems. You may be penalized for incorrect filenames or formats.

(d) The first line of each file: You MUST put your name and unity id on the top of each file you submit, formatted as follows:
/* unity-id FirstName MiddleInitial LastName */
Don't leave any field blank.

(e) Before you submit: *** IMPORTANT*** Please make sure your programs compile and execute normally on the specified VCL image, you will receive 0 for programs that do not compile or do not execute. You will receive partial credit for programs that do not produce correct output.

Problem 1

We have a program prob_1.c, it can be compiled fine with:

```
gcc prob_1.c -o prob_1 -g -lpthread
```

What outputs are possible for this program?

Put your answers in problems.txt (ASCII file).

Problem 2

Give the following pseudo code:

```
// Transfer from account acc_from to account acc_to,
// where each account has a mutex field and an amount field
bool Transfer (amount, acc_from, acc_to) {
  pthread_mutex_lock(&acc_from.mutex);
  pthread_mutex_lock(&acc_to.mutex);
  if (acc_from.balance < amount)
    return ERROR;
  acc_from.balance -= amount;
  acc_to.balance += amount;
  pthread_mutex_unlock(&acc_from.mutex);
  pthread_mutex_unlock(&acc_to.mutex);
  return SUCCESS;
}

Client1() {
  Transfer (amount1, account1, account2);
}

Client2() {
  Transfer (amount2, account2, account1);
}

int main(){
  // Some code to initialize two global accounts: account1 and account2
  ...

  // The two threads
  pthread_create(&tid1, NULL, Client1, NULL);
  pthread_create(&tid2, NULL, Client2, NULL);
}
```

1. Find and describe a potential deadlock in the pseudo code.
2. Now you are given a function Order(pthread_mutex_t *m1, pthread_mutex_t *m2), which returns 0 if m1 and m2 are the same and returns 1 or -1 if m1 and m2 are different. This Order() function has the following properties: if Order(&mutexA, &mutexB) returns value X (where X is not equal to 0), then Order(&mutexB, &mutexA) returns value -X; if Order(&mutexA, &mutexB) and Order(&mutexB, &mutexC) return the same value X, then Order(&mutexA, &mutexC) also return value X. With such a function, how would you fix the potential deadlock. Please describe.

Put your answers in problems.txt (ASCII file).

Problem 3

Our goal here is to write a multithreaded program that can calculate the maximum difference for a list of numbers stored in a file, where a number takes one line in the file and the file name is passed in through a command line option. For a file containing the following numbers:

```
1
2
3
4
5
```

your program should print out 4. We will use a single manager thread and multiple worker threads to collaborate on solving the problem. We will first start worker threads and then use the main thread as the manager thread. The manager uses a task queue to post tasks if a slot is available in the task queue. The workers fetch tasks from the task queue when they are idle if there are tasks posted in the queue. The manager thread can read in all numbers before starting worker threads, but a new task can be put into the queue only if there is an empty slot.

If all task slots have already been occupied, the manager will have to wait for worker threads to complete more tasks before posting more work. If all tasks have already been assigned, workers will have to wait for the main thread to put in more tasks, until the manager informs the worker threads that that all numbers have already been processed. Once all numbers have been worked on, a single reduction task needs to be put in the task queue, and one of the worker threads will take the reduction task and calculate a global maximum.

Essentially, this is a producer/consumer problem, where the queue is a task queue. A skeleton, prob_3.c, will help you get started, and the program takes two parameters:

```
./proc_3 <number_of_worker_threads> input_long
```

In the given skeleton, the maximum number of worker threads and the task queue size have been defined as 8 and 16, respectively.

Basically, there are two type of tasks: many to find the local maximum difference for each worker and one to find the global maximum difference among workers. The total number of tasks is equal to 1+N, where N is the number of numbers in the input file~~, but we will be using 1+<number of worker threads> instead~~.

You need to compute the difference of all pairs of numbers, so be careful when your manager is assigning tasks. The difference computation between the newly read number and all the older ones is a good granularity level for task assignment. Each of your worker can store its local maximum difference to a global structure, based on which the final reduction to find the overall maximum difference can be conducted. One possible design of data structure is using an array to store the local maximum difference for number i to numbers 0...i-1.

One way to orgnize tasks: for a file containing

```
1
```

2
3
4
5

The first task for "1" does not compute anything; the second task for "2" compares "2" with "1", getting "1" as the local max; the third task for "3" compares "3" with "2" and "1", getting "2" as the local max; (omitting the fourth task); the fifth task for "5" compares "5" with "1", "2", "3", and "4", geeting "4" as the local max; the last task compute the global max from local maximums "NA", "1", "2", "3", "4".

You need to make sure the global reduction task is only worked on after the local ones have been completed, i.e., you somehow need to synchronize between these phase. In the example above, you need to make sure the last task starts after "4" is calculated by the fifth task.

Since they are multithreaded programs, you need to be careful while testing and debugging. Make sure you reason about all possible interleavings and use necessary synchronization operations in your code.

As always, if your code catch certain error conditions, print meaningful error messages in your own format. After you finish, turn in your source files prob_3.c, Makefile, and README.

Homework 3.1

Since we have not covered much on memory management yet and this producer-consumer queue programming assignment can use some extra time, you can use another week to reengineer the program. Each of you can submit a new version of your program and help others on their programs. Below, we describe the policy, and we assume 100 as the maximum points for the programming assignment.

If you choose to submit a new version, your final grade for this programming assignment will be adjusted. If your original grade is X with a penalty of Y, which is 0 or 25%, and your new grade is Z, (Z-X) will be scaled and added to X*(1-Y). (Z-X) will be scaled only if it is larger than 20, and points in between 20+10i will be halved for i times, e.g., 26 will be scaled to 23, 34 will be scaled to 26, and 48 will be scaled to 28.5. You need to minimize your code changes. As a result, (Z-X) after scaling with further be divided by your code change factor: number_of_all_lines_in_new_file/number_of_reused_lines. Please still change lines as usual and do not put all code into one line.

When you are working on your new version, you can ask for your fellow classmates for help. Suppose you ask two classmates for help, (Z-X)*scalefactor/changefactor will be given to them as credits, and you three need to decide how to distribute the credit. Note that you will still get (Z-X)*scalefactor/changefactor for yourself. You can decide according your own situation by specifying a percentage for each and the two percentage numbers should be less than 1 when they add up. You can decide whom you want to ask for help, and you do not necessarily need to ask for two classmates for help. If you prefer, you can also work in groups.

Everyone can provide and offer help. The maximum number of points that can be added to X*(1-Y) will be 25, and the maximum final points is 115.

After you finish, turn in your new source files prob_3.c, Makefile, and README. There will be a new submission folder, but please use the original discussion forum. In your README, please CLEARLY DESCRIBE what changes have been made and why they are made. If you do not, your credits will be further scaled. If you provide

or receive help, please briefly describe such activities and specify how the credits should be distributed.

Lastly, if you find holes in the policy above, please let the instructor know. Thanks in advance for helping each other master the producer-consumer queue.