Assignment 2 – Threads

CMPT 300 – Operating Systems – Instructor: Nate Payne

Part 1 - POSIX Threads

```
Please submit a zip folder with the following naming conventions to canvas:

LastName_FirstName_StudentNumber_Assig2

This file should include all code, and a text file called answers.txt that
includes answers to all questions. There may be questions for both part 1 and
part 2, so make sure that you address all questions.

Instructions
============

This program sorts strings using "enzymes".  An enzyme is a function that
sorts two consecutive characters.  We define one enzyme per pair of
consecutive characters; these enzymes working together in parallel can sort
the entire string.  We use pthreads to instantiate and parallelize the
enzymes. Unfortunately, this program doesn't seem to be working correctly.
That's where you come in.

Before you edit the code, read through it, and answer these questions:

  1) Briefly explain why this application would be difficult to write using
     multiple processes instead of threads.

  2) What is the significance of 'workperformed'?  How is it used?

  3) Explain exactly what is the type of 'fp' in the following declaration:
       void *(*fp)(void *)

Part II
=======

Now, to fix the program:

  1) The function run_enzyme() needs to be created. Please see the notes
inside enzyme.c.

  2) The function 'make_enzyme_threads' has a memory bug. Fix this by simply
     re-ordering the lines in this function.  It is simple fix but may take a
     while for you to find it.

  3) The function 'join_on_enzymes' is incomplete. Read the relevant man
pages and figure out how the function is supposed to work. Then insert the
     correct code snippets into 'whatgoeshere'.

  4) Your programming work can be considered complete when you have completed
     the above and all of the tests pass.

Testing
```

```
-------

make test
./enzyme -test -f0 all
Running tests...
 1.make                ::pass
 2.sort                ::pass
 3.pleasequit1         ::pass
 4.pleasequit0         ::pass
 5.swap1               ::pass
 6.swap2               ::pass
 7.swap3               ::pass
 8.run_enzyme          ::pass
 9.join                ::pass
 10.cancel             ::pass

You may also want to experiment with the cancel function -
./enzyme Cba
./enzyme CBA

Questions
=========

  1) Why do we not detach any of the enzyme threads? Would the program
function if we detached the sleeper thread?

  2) Why does the program use sched_yield? What happens if this is not used?
     Will the swap counts always be identical?

  3) Threads are cancelled if the string contains a 'C' e.g. "Cherub".
     Why do we not include cancelled threads when adding up the total number
     of swaps?

  4) What happens when a thread tries to join itself?
     (You may need to create a test program to try this)
     Does it deadlock? Or does it generate an error?

  5) Briefly explain how the sleeper thread is implemented.

  6) Briefly explain why PTHREAD_CANCEL_ASYNCHRONOUS is used in this MP.

  7) Briefly explain the bug in Part II, #2 above.

Submit all code for this part of the assignment.

Reminder: Do not copy or plagiarize any code from any other student in the
course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so
will receive a 0 for the assignment portion of the course. My goal is to
maximize your learning, so please focus on that!
```

Part 2 – Multi-thread Calculator

In this assignment, you will be implementing a simple multithreaded calculator. Our calculator will accept expressions as infix notation text

strings consisting only of non-negative integers and three operators: grouping "()", addition "+", and multiplication "*". For example, given the string "4*(6+3)" as input, the expected output is the string "36".

Multiple expressions are read from stdin, seperated by newlines, and evaluated concurrently. A period '.' as the first character of a line indicates end of input. Then the program can stop once all of the expressions already entered have been processed.

There should be no whitespace, negative numbers, fractions, etc. You should not implement precedence rules for operations; the test input will disambiguate the precedence order using the grouping operator, i.e., we will test "4+(6*3)" but never "4+6*3". In general, unless specifically asked in one of the steps below, you do not need to worry about the validity of the input.

A skeleton implementation of the calculator is provided in calc.c. The buffer[] array holds the inputted expressions, seperated by semicolons ';' (these are used internally only, and are never input by the user or output by the program!).

The first expression in the buffer is the one that is actively being reduced by the ungrouping, addition, and multiplication threads. When this expression has been reduced to a single number (such that the buffer looks something like "36;2+5;..."), a sentinel thread detects this condition and prints that number to stout, followed by a newline, and removes the expression from the buffer, so work can start on the next expression, e.g., "36;2+5;" becomes "2+5;".

The calculator implementation encompasses five threads, aside from the main thread, all working concurrently:

1. reader_thread. This thread reads lines from stdin and appends them to the buffer. If there is not enough space remaining in the buffer[], reader blocks until there is.

2&3. adder_thread and multiplier_thread. These look for "+" and "*" signs, respectively, surrounded by two "naked" numbers, e.g., "4+3" or "2*6", but NOT something "9+(2)" or "(5*(2))*8". The corresponding operation on the two numbers is performed, replacing the subexpression with the result, e.g., "4+3" becomes "7".

4. degrouper. It looks for a single number surrounded by parentheses, e.g. "(8)", and removes said parentheses.

5. sentinel. This thread checks the buffer to see if the currently processed expression has been reduced to a single number; if it has, it prints the number to stdout and removes it from the buffer as described above.

Read through the provided code before beginning the remainder of this assignment. Your primary tasks in this MP are implementing the adder, multiplier, and degrouper threads, using sychronization to protect critical sections, and improving performance via blocks and yields.

Step 1: Getting Started

Compile and run the program:
> make
> ./calc

You'll notice it doesn't do much, exiting immediately after printing that zero operations were performed.

There's a problem with the code--it quits as soon as any input is typed, or possibly before, depending on your thread scheduler. The reason is that the main thread (thread running in the smp3_main() function) is falling through to the end prematurely.

You need to join onto one, and only one, of the five subthreads so that the program will only be able to terminate once all input has been processed and all results output. Identify the correct thread and fix this problem.

Step 2: Implement adder, multiplier, and degrouper

Write the code to implement these functions. They should only consider the current expression, which is the first expression in the buffer, or, equivilently, everything up until the first semicolon. Do not implement synchronization and mutual exclusion yet.

Tip: See sentinel() for an example invocation of strcpy() that shifts the characters in a string to the left.

Tip: If you are not certain which stdlib functions to use for string/number manipulation, feel free to use the provided utility functions (string2int, int2string, isNumeric).

Pseudocode for adder/multiplier:
 a. Scan through current expression looking for a number.
 b. Check each number to see if it is followed by a +/*, and then a numeric character (indicating the start of another number).
 c. If it is, add/multiply the two numbers, and replace the addition/multiplication subexpression with the result, e.g., "34+22" becomes "56".

Pseudocode for degrouper:
 a. Scan through current expression looking for a '('.
 b. Check if the next character is numeric (indicating the start of a number).
 c. If it is, check if the number is immediately followed by a ')'.
 d. If so, we have something like "(32432)". Now remove the '(' and ')' we've just located from the expression.

The above pseudocode is only a suggestion; your code may work differently. For example, you could hunt for + or * in the expression, then check that there are "naked" numbers to the left and right.

Q1: At this point, your solution does not contain any synchronization or mutual exclusion. Give an example of and explain a possible synchronization error that could occur in this code. Be specific.

Q2: Suppose we implement correct synchronization and mutual exclusion for all of the threads. If our three functions were to operate on all expression in the buffer at once (not just the first expression), would the program generate incorrect output? Why or why not?

Step 3: Critical Sections

Identify and protect the critical sections of the adder, multiplier and degrouper functions with a POSIX mutex. Try to keep your critical sections as small as possible.

Tip: man pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_init, ...

Check the return values of these functions for errors. Print a brief error message on stderr and exit your program with EXIT_FAILURE if one of them fails. Use the provided function printErrorAndExit().

Next, identify and protect the critical sections of the reader and sentinel functions, as well. Your code should now be immune to synchronization errors (e.g., race conditions, data corruption).

Q3: For this step, what specific data structure(s) need(s) protection? Why?

Q4: What would happen if you had a busy-wait within one of your critical sections? What if it is a loop with sched_yield()?

Q5: Why is it sometimes necessary to use the non-blocking pthread_mutex_trylock() instead of the blocking pthread_mutex_lock()? Think for example of a program that needs to acquire multiple mutexes at the same time.

Step 4: Accounting

Store the number of operations performed by your calculator in the variable num_ops, which is printed out before successful program termination. Only addition, multiplication and degrouping should be counted as operations, not reading or printing. Make sure access to this variable is free from race conditions.

Q6: Is a new mutex, separate from what you have in Step 3, required to correctly implement this behavior? Why or why not?

Step 5: Performance

Identify where the program is spin-waiting, that is looping while (implicitly or explicitly) waiting for something to change. Add sched_yield() calls at the appropriate place inside these loops.

Q7: Why is it important, even on single-processor machines, to keep the critical sections as small as possible?

Q8: Why is spin-waiting without yielding usually inefficient?

Q9: When might spin-waiting without yielding or blocking actually be *more* efficient?

Step 6: Monitoring Progress

In the current form, it is possible for the program to deadlock if incorrect input, e.g., "5++3", is given. All threads would be waiting for someone else to change the buffer to something they can handle, and no progress would occur.

Change adder, multiplier, degrouper, and sentinel so that the sentinel thread can detect when the current expression cannot be reduced due to an improper input sequence. You still do not need to worry about whitespace, fractions, negative numbers, etc.; only incorrect permutations of numbers and our three operations.

Do not check the input directly, but rather detect that the buffer is unaltered after the adder, multiplier and degrouper have all run. You will need an additional shared data structure to indicate this condition and a mutex to protect it.

If the sentinel finds that no progress can be made, it should output to stdout the exact string "No progress can be made\n" and immediately exit the program with EXIT_FAILURE. Be very careful that you do not admit any false positives, as this may have a severe adverse effect on your autograder results!

Step 7: Semaphores

Mutexes are easy. Ordinary. Boring. Change your code to use a semaphore instead of a mutex for the previous step. As it is generally somewhat easier to mess up when semaphores as opposed to mutexes, make sure everything works with mutexes before undertaking this step.

Tip: Read up on POSIX semaphores: man sem_init, sem_wait, sem_post, ...

Again, check the return values of these functions for errors. Print a brief error message on stderr and exit your program with EXIT_FAILURE if one of them fails.

Q10: You have to supply an initial value when creating a semaphore. Which value should you use to mimic the default functionality of a mutex? What would happen if you specified a larger value?

Testing

A test suite is provided for self-evaluation. Use it to gauge your progress:

```
> make test
or
> ./calc -test <test name>
```

Submit all code for this part of the assignment.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!