CIT 5950 Recitation 6 - Synchronization, Locks, and Scheduling

Welcome back to recitation! We're glad that you're here :)

Exercise 1 - Synchronization & mutex locks

It's payday! It's time for Penn to pay each of the 595 TAs their monthly salary. Each of the TA's bank account is inside the `bank_accounts[]` array and the person who is in charge of paying the TAs is a 595 student and decided to use pthreads to pay the TAs by adding 1000 into each bank account. Here is the program the student wrote:

```
// Assume all necessary libraries and header files are included
const int NUM_TAS = 7;

static int bank_accounts[NUM_TAS];
static pthread_mutex_t sum_lock;

void *thread_main(void *arg) {
  int *TA_index = reinterpret_cast<int*>(arg);

  pthread_mutex_lock(&sum_lock);
  bank_accounts[*TA_index] += 1000;
  pthread_mutex_unlock(&sum_lock);

  delete TA_index;
  return nullptr;
}

int main(int argc, char** argv) {
  pthread_t thds[NUM_TAS];
  pthread_mutex_init(&sum_lock, NULL);

  for (int i = 0; i < NUM_TAS; i++) {
    int *num = new int(i);
    if (pthread_create(&thds[i], nullptr, &thread_main, num) != 0) {
      /*report error*/
    }
  }

  for (int i = 0; i < NUM_TAS; i++) {
    cout << bank_accounts[i] << endl;
  }

  pthread_mutex_destroy(&sum_lock);
  return 0;
}
```

a) Does the program increase the TAs' bank accounts correctly? Why or why not?

No its not correct. It needs to use pthread_join to wait for each thread to finish before exiting the main program. pthread_exit() might not be the best solution here. You want to check the return value of join to make sure the transaction applied rather than just exiting and trusting the threads to finish successfully. Gotta get those TA dolla's.

b) Could we implement this program using processes instead of threads? Why would or why wouldn't we want to do this?

We could, but doing so would require some way for the processes to communicate with each other so that the data structure can be "shared" (remember that inter-process communication can be difficult and time consuming). It is much easier to just use threads since each thread could directly access the data structure.

c) Assume that all the problems, if any, are now fixed. The student discovers that the program they wrote is kinda slow even though its a multithreaded program. Why might it be the case? And how would you fix that?

Because there is a lock over the entire bank account array, so only one thread can increase the value of one account at a time and there is no difference from incrementing each account sequentially. To fix this, we can have one lock per account so that multiple threads can increment the account at the same time. (With the current setup, we could also just not use a lock since we know that no thread will have a conflicting TA_index. For a more generalized program, it would be better to use the first answer.)

Exercise 2 - Condition Variables & Deadlock

The 5950 Staff is having troubles again with writing programs for getting milk. In this case, instead of having two threads that are roommates, we have a thread that delivers milk and two threads that receive milk. This is sort of like having a milkman come to people's house to deliver milk.

We write a program to model this by using a global integer `milk_count` to mark the number of milk delivered, and have a `pthread_mutex_t milk_lock` associated with the milk. One complication is that the milk can only be consumed if there is milk delivered (e.g. `milk_count > 0`). The program we wrote is below but doesn't work as expected.

```
#include <iostream>
#include <cstdlib>
#include <unistd.h>
#include <pthread.h>

using std::endl;
using std::cout;
using std::cerr;

pthread_mutex_t milk_lock;
int milk_count = 0;

void* milk_delivery(void* arg) {
  int* num_deliveries = (int*) arg;

  for (int i = 0; i < *num_deliveries; i++) {
     pthread_mutex_lock(&milk_lock);

     milk_count++;

     pthread_mutex_unlock(&milk_lock);
  }

  delete num_deliveries;
  return nullptr;
}
```

```
void* milk_consume(void* arg) {
  int* num_consume = (int*) arg;

  for (int i = 0; i < *num_consume; i++) {
     pthread_mutex_lock(&milk_lock);
     // can only use milk if there is milk to use

     while (milk_count <= 0) {
       // if there is no milk, sleep for a bit
       // and check again

       sleep(1);

     }

     milk_count--;
     cout << "I Got milk! I Like Milk :)" << endl;

     pthread_mutex_unlock(&milk_lock);
  }

  delete num_consume;
  return nullptr;
}

int main() {
  pthread_t consumer1;
  pthread_t consumer2;
  pthread_t milk_deliverer;

  pthread_mutex_init(&milk_lock, nullptr);

  pthread_create(&consumer1, nullptr, milk_consume, new int(3));
  pthread_create(&consumer2, nullptr, milk_consume, new int(7));
  pthread_create(&milk_deliverer, nullptr, milk_delivery,
                                                      new int(10));

  pthread_join(consumer1, nullptr);
  pthread_join(consumer2, nullptr);
  pthread_join(milk_deliverer, nullptr);

  pthread_mutex_destroy(&milk_lock);

  return EXIT_SUCCESS;
}
```

a) The program doesn't finish and not everyone gets all the milk they want. Why is that the case?

A consumer thread can acquire the milk_lock when the milk_count is zero. The consumer thread will continuously run the while loop waiting to receive milk. However, the milk deliverer will not be able to acquire the milk_lock to increment the milk counter and so no progress can be made.

b) How can we solve this problem without introducing any new locks or condition variables? The program should also stay multithreaded and concurrent.

We can add the following:

```
void* milk_consume(void* arg) {
  int* num_consume = (int*) arg;

  for (int i = 0; i < *num_consume; i++) {
     pthread_mutex_lock(&milk_lock);
     // can only use milk if there is milk to use

     while (milk_count <= 0) {

       // if there is no milk, sleep for a bit
       // and check again
       pthread_mutex_unlock(&milk_lock);
       sleep(1);
       pthread_mutex_lock(&milk_lock);

     }

     milk_count--;
     cout << "I Got milk! I Like Milk :)" << endl;

     pthread_mutex_unlock(&milk_lock);
  }

  delete num_consume;
  return nullptr;
}
```

Instead of locking on the whole while loop in milk_consume(), we unlock right before sleeping so that the milk deliverer can acquire the milk lock. In this way, the milk_count can be incremented.

c) Another way to solve this problem is to involve the use of a condition variable. How could we change the code to work properly while using a condition variable.

```
pthread_mutex_t milk_lock;
pthread_cond_t milk_cond; // Add condition variable
int milk_count = 0;

void* milk_delivery(void* arg) {
  int* num_deliveries = (int*) arg;

  for (int i = 0; i < *num_deliveries; i++) {
     pthread_mutex_lock(&milk_lock);

     Milk_count++;
     // signal to consumer thread that it can wake up
     // the consumer thread will have the milk_lock
     pthread_cond_signal(&milk_cond);
     pthread_mutex_unlock(&milk_lock);
  }

  delete num_deliveries;
  return nullptr;
}

void* milk_consume(void* arg) {
  int* num_consume = (int*) arg;

  for (int i = 0; i < *num_consume; i++) {
     pthread_mutex_lock(&milk_lock);
     // can only use milk if there is milk to use

     while (milk_count <= 0) {
       // if there is no milk, sleep for a bit
       // and check again

       // sleep(1); No longer needed
       // release lock and put thread to sleep
       pthread_cond_wait(&milk_cond, &milk_lock);

     }
     milk_count--;
     cout << "I Got milk! I Like Milk :)" << endl;
     pthread_mutex_unlock(&milk_lock);
  }
  delete num_consume;
  return nullptr;
}
```

d) Using a condition variable is usually considered to make better use of the computer’s resource when compared to the type of solution used in part b. Why might this be the case?

In part b, we had to use “spinning” in order to prevent a deadlock from occurring. In cases where the milk count reaches 0, the consumer thread will continually loop and switch between acquiring and releasing a lock.

Condition variables make better use of a computer’s resources since the while loop in a consumer thread is not executing continuously. Instead, the consumer threads are put to sleep and wait for a signal from the producer thread. Once a signal is sent, one or more consumer threads will wake up and continue execution.