CIT 5950 Recitation 5 - 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 5950 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 5950 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 = 8;

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?

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

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

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 deliver 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?

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

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.

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?