CIT 595 Section 6

Synchronization, Locks

Review - locks & synchronization

Exercise 1 - mutex locks

Exercise 1

```
// 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, nullptr);

  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;
}
```

Exercise 1

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 its a multithreaded program. Why might it be the case? And how would you fix that?

Exercise 1

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

No, it's 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.

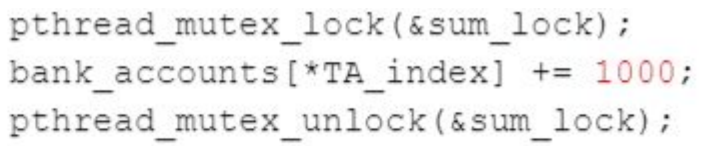
Exercise 1

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.

Exercise 1

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?



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

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

Exercise 2

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

b. How can we solve this problem without introducing 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?

Exercise 2

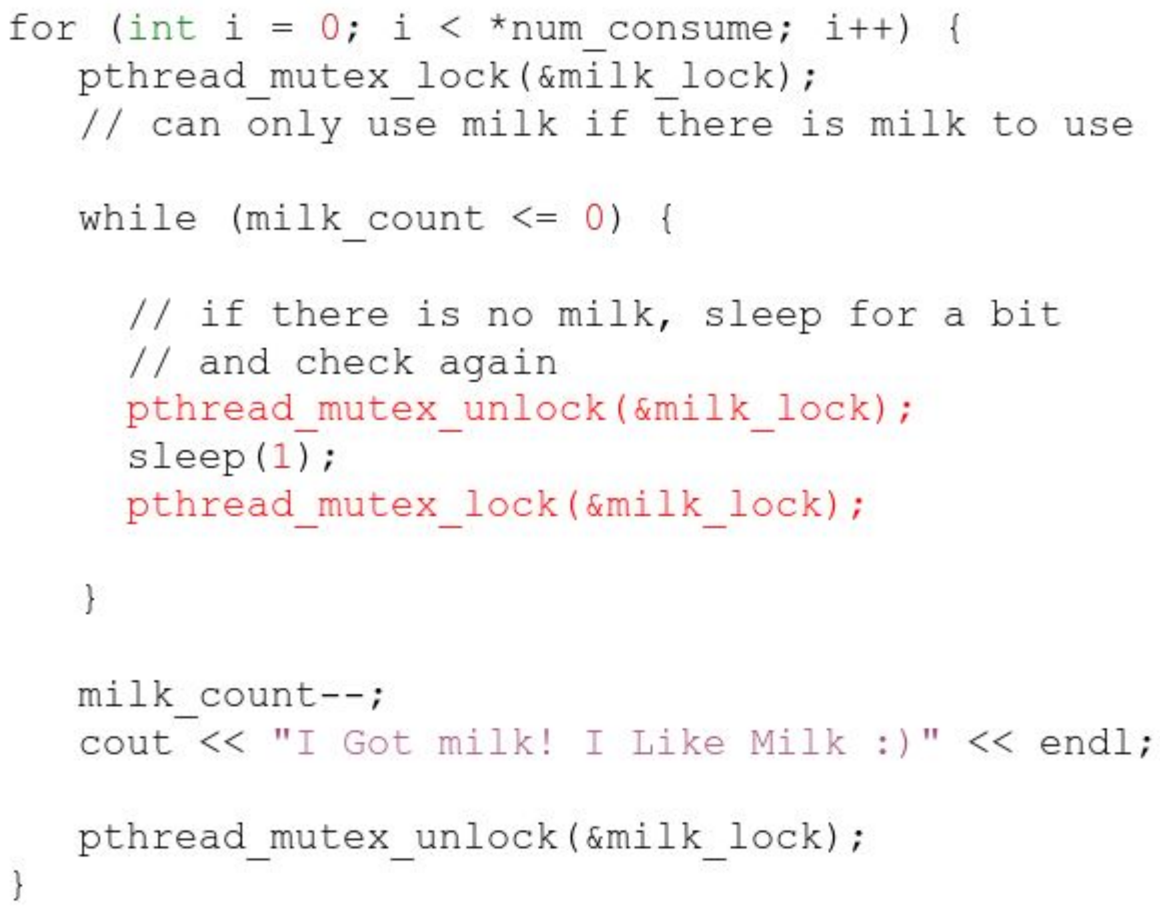
a) The program doesn’t finish and not everyone gets all the milk that they want. Why is that the case?

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.

Exercise 2

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

Exercise 2



```
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);
}
```

Exercise 2

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.

Exercise 2

```
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;
}
```

Exercise 2

```
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;
}
```

Exercise 2

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

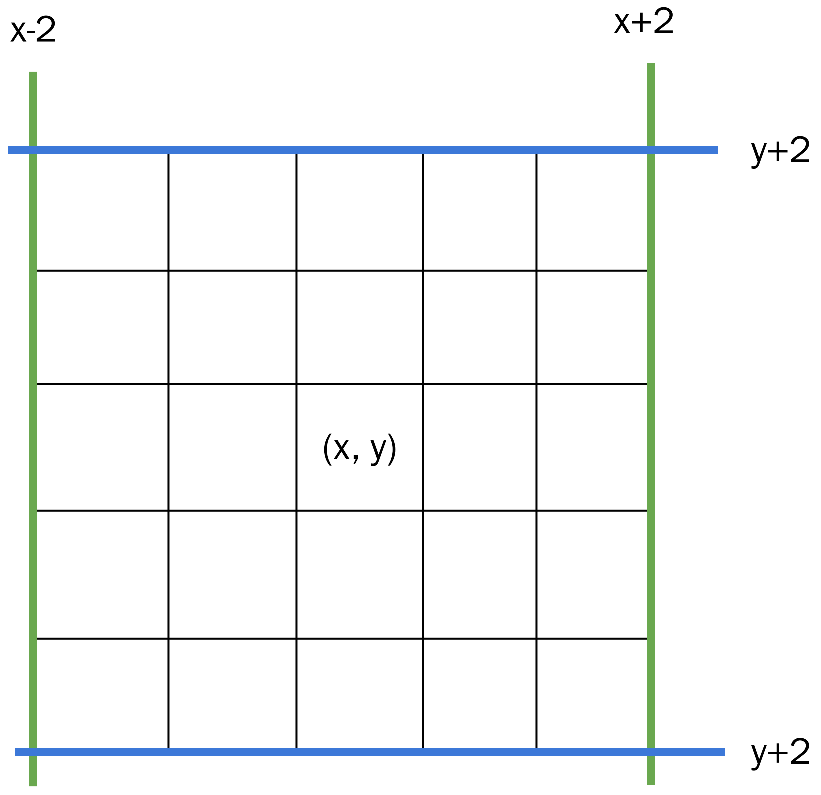
Homework 3 Overview

TL:DR

- Blur Doge.bmp sequentially
- Blur Doge.bmp concurrently

- Implement a thread-safe DoubleQueue
- Implement a short program that receives input from the keyboard, performs some basic calculations, and outputs results.

How to Blur



Box-size = 2

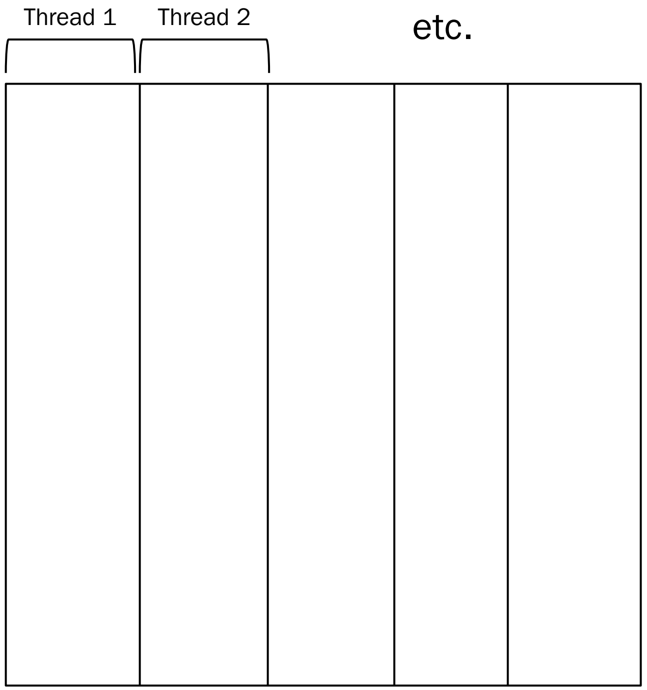
Blur → Average color of all pixel within this grid. Including the center pixel.

$$\text{Average Color} = \frac{\text{Sum of all colors}}{\text{\# of grid pixels}}$$

Make sure to handle out of bounds coordinates.

Default to 0, image height, or image width.

How To Pass Arguments to Thread Functions



N = 5

Given N threads, divide the images in N roughly equally parts. Each thread will blur a corresponding part of the image.

What information does each thread need to blur it's chunk of the image?

Store it in a struct and pass as argument to pthread_create

```
pthread_create (Addr. to store threadId,
                Special attributes,          // nullptr
                Thread function,
                Addr. of thread function args)
```

Are locks required for this blurring program?

Why or why not.

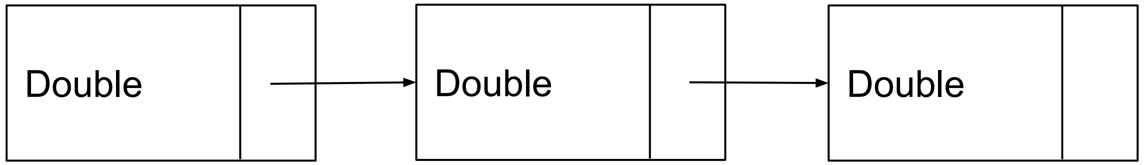
DoubleQueue

Essentially a LinkedList…but make it thread-safe. (Look back at HW0)

Hint: You'll probably wants to introduce some member variables that are locks.

remove() removes node from the head.
Similar to LinkedList_Pop.

add() adds node to the tail.
Similar to LinkedList_Append.



Difference Between remove and wait_remove?

	remove(double* val)	**wait_remove()**
DoubleQueue is empty	Return false.	Spin until DoubleQueue is not empty. Then…
DoubleQueue is not empty	Remove Node from head. Store double value in return parameter. Return true.	Remove Node from head. Return double value.

Child Threads in numbers.cc

Reader	Writer
Read input from keyboard (see numbers_sequential.cc). Add to double Queue. Sleep for 1 second.	Get value from DoubleQueue. (remove or wait_remove?) Calculate and output (see numbers_sequential.cc).

Please pay attention to the instructions!
You only need to find the min, max, and average of the most recent 5 values that were input.

I would say 70% of the code is already given in numbers_sequential.cc.
So pull from there whenever you can.