

CSCI376 Multicore and GPU Programming

**Lab 4 - Synchronization  
Objective: Using synchronization methods in programs with multiple threads or processes**

**Submission Due: May 17th, Sunday 23:59 ( Lab\_04.at9htlkvc2t4txhr@u.box.com )**

Race conditions are cases where different threads or processes try to access data at the same time, but they do so in a way that causes undesirable outcomes. For instance, if two threads wish to increment the value of a variable, which is made up of a read and a write instruction, they must do so after each other.

The following code simply adds every element in the array to a shared global count for each thread. Running it with 1 thread gives you the sum, running it with 2 threads gives you twice the sum etc. On executing the program with more than one thread, you will notice that the result is wrong. This is because the threads run into race conditions when attempting to modify the shared count variable.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #define ARRAY\_SIZE 1000000  // Global variable  int count;  int \*arr;  // Thread method to add every element in the array to the count  void\* addElements(void\* lparams){  for(int i=0; i<ARRAY\_SIZE; i++){  count += arr[i];  }  }  int main(){  // Initialize the array to 1 from start to finish  arr = (int\*)malloc(ARRAY\_SIZE \* sizeof(int));  for(int i = 0; i < ARRAY\_SIZE; i++){  arr[i] = 1;  }  pthread\_t thr1, thr2;  pthread\_create(&thr1, NULL, addElements, NULL);  pthread\_create(&thr2, NULL, addElements, NULL);  pthread\_join(thr1, NULL);  pthread\_join(thr2, NULL);    printf("Content of count is: %d\n", count);  free(arr);  } |

Places in the code where race conditions can occur are called critical sections. To make sure that your result is consistent and correct, you need to coordinate between threads and make sure they do not interrupt a critical section. This can be achieved through synchronization, specifically mutexes when working with threads.  
  
Mutexes act like a lock which you can lock and unlock. Only one thread can hold a mutex at the time, and if another thread wants to lock it, it must wait till the first thread releases or unlocks it. Locking too much of the code could lead to your application becoming sequential. Additionally, locking and unlocking comes with extra overhead, which may slow down your application if you do it too often.

The previous code snipped is modified to add mutex locking and make the output correct.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #define ARRAY\_SIZE 1000000  // Global variable  pthread\_mutex\_t myMutex;  int count;  int \*arr;  // Thread method to add every element in the array to the count  void\* addElements(void\* lparams){  for(int i=0; i<ARRAY\_SIZE; i++){  pthread\_mutex\_lock(&myMutex); // Lock  count += arr[i];  pthread\_mutex\_unlock(&myMutex); // Unlock  }  }  int main(){  // Initialize the array to 1 from start to finish  arr = (int\*)malloc(ARRAY\_SIZE \* sizeof(int));  for(int i = 0; i < ARRAY\_SIZE; i++){  arr[i] = 1;  }  // Initializing a mutex is necessary before using it  // Alternatively use PTHREAD\_MUTEX\_INITIALIZER at declaration  pthread\_mutex\_init(&myMutex, NULL);  pthread\_t thr1, thr2;  pthread\_create(&thr1, NULL, addElements, NULL);  pthread\_create(&thr2, NULL, addElements, NULL);  pthread\_join(thr1, NULL);  pthread\_join(thr2, NULL);    // Destroying the mutex will deallocate its resources  pthread\_mutex\_destroy(&myMutex);  printf("Content of count is: %d\n", count);  free(arr);  } |

Mutex locks only work within the context of one process, meaning that they only work between threads and not processes. To do synchronization between processes, you can make use of named semaphores, which are persistent and managed by the kernel. This allows them to work between processes.  
  
Just like mutex locks, semaphores need to be created, locked, unlocked and destroyed, but the process is called open, wait, post, close and unlink. The following shows semaphores while writing to files.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  #include <fcntl.h>  #include <semaphore.h>  #define ARRAY\_SIZE 1000  #define CHILD\_COUNT 2  // Global variable  sem\_t \*mySem;  int count;  int \*arr;  int main(){  // Opening a file that the processes will print into  FILE\* file = fopen("list.txt", "w");  // Initialize the array to 1 from start to finish  arr = (int\*)malloc(ARRAY\_SIZE \* sizeof(int));  for(int i = 0; i < ARRAY\_SIZE; i++){  arr[i] = 1;  }  // Opening a semaphore is necessary before using it  // O\_CREAT means create a new semaphore if one doesn't exist  mySem = sem\_open("/semName", O\_CREAT, 0666, 1);  pid\_t children[CHILD\_COUNT];  memset(children, 0, CHILD\_COUNT \* sizeof(pid\_t));  for(int i=0; i<CHILD\_COUNT; i++){  children[i] = fork();    // If the ID is 0, it means a child process  if(children[i] == 0){  for(int j=0; j<ARRAY\_SIZE; j++){  sem\_wait(mySem); // Lock  fprintf(file, "Proc %d - Elem %d - Value %d\n", i, j, arr[j]);  fflush(file);  sem\_post(mySem); // Unlock  }  return 0;  }  }    // Closing and Unlinking the semaphore are both necessary to free it  sem\_close(mySem);  sem\_unlink("/semName");  fclose(file);  free(arr);  } |

Semaphores are persistent entities in the system. If you forget to close and unlink it, it remains in the system and you might not be able to open it again. The following code shows how to close a semaphore if it was left in the system.

|  |
| --- |
| // Clears a previously created semaphore  void clear\_semaphore(const char\* name){  // Test if the semaphore can be opened  sem\_t\* sem = sem\_open(name, 0);    // If the semaphore failed to open, it's closed or doesn't exist  if(sem == SEM\_FAILED){  printf("There is no semaphore with name \"%s\"\n", name);  } else {  // If the semaphore does exist, then  // Close the semaphore  if(sem\_close(sem) == -1){  printf("Failed to close semaphore \"%s\"\n", name);  exit(EXIT\_FAILURE);  }  // Unlink the semaphore  if(sem\_unlink(name) == -1){  printf("Failed to unlink semaphore \"%s\"\n", name);  exit(EXIT\_FAILURE);  }  printf("Successfully cleared semaphore \"%s\"\n", name);  }  } |

Lab Tasks

1) Threads and Mutexes

You are given 8 values in an array that represent money deposits to a user's bank account. Deposit these values using multiple threads. **You must use the fetchBalance and updateBalance functions and not directly modify the balance variable.**

1. Complete the deposit function to update the balance.
2. Partition the values and start at least 2 threads to work on the array.
3. Add any additional requirements that are necessary

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <unistd.h>  #include <pthread.h>  #define THREADS 4  #define ARRAY\_SIZE 8  double balance = 0;  double fetchBalance(){  sleep(1);  return balance;  }  double updateBalance(double value){  sleep(1);  balance = value;  }  void\* deposit(void\* lparam){  // [1] Implement the function  }  int main(){  pthread\_t threads[THREADS];  double deposits[ARRAY\_SIZE] = {  72.5, 13.75, 50.5, 82.25,  20.5, 64.75, 10.5, 199.25  };    // [2] Partition and start threads  printf("Balance: %f\n", balance);  } |

2) Forks and Semaphores

You are given a file list.txt with 1,000 numbers. You need to filter it and only keep prime numbers.

1. Create at least 2 child processes
2. Each child process must read from the input file list.txt one number at a time
3. If the number is prime, print it to a file "result.txt"
4. All child processes must write to the same file.
5. Add any additional requirements to make the program run correctly.

3) Dependent Operations

1. Create an array of 50,000 integers
2. Initialize the array to a series of 1, 2, 3 ... n
3. Find the average of the integers by summing up the elements using multiple threads
4. Find the standard deviation of the using multiple threads.   
   