COSC 439

Operating Systems

Spring 2020

**Lab Exercise #5**

*I pulled a thread out of the air  
Golden and shining as thin as a hair  
I couldn't do anything but hold it and stare  
I don't know why it was hovering there*

* Phish

**Name: / 6**

**Goals:** The intention of this lab is to better understand how a Linux threads and the Pthread library works.

**Environment:** The GNU GCC C compiler on your Ubuntu virtual machine.

**Slack Channel:** All/any questions, problems and/or announcements for this lab should be directed to the course’s #lab5 Slack channel.

**Submission:** All files, including this document with the required screenshots, should be submitted via Slack/GitLab as indicated. ***Lab submissions not following this convention may not be graded.***

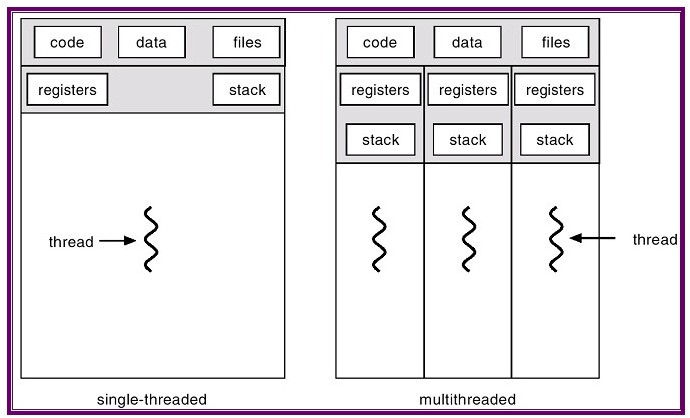
**Deadline:** Submitted via GitLab by 11:59pm on [Friday](https://www.youtube.com/watch?v=kfVsfOSbJY0), March 6, [2020](https://www.youtube.com/watch?v=AcCRhVwpqhg) with all source code, developed files and this document with screenshots checked into your GitLab repository. You should save the C Pthread programs to the same Lab 4 directory and repository as the previous lab.

**Pthreads** (6 points) The POSIX thread (Pthread) libraries are a standards-based thread API for C/C++ (see full Pthreads tutorial at <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>). It allows you to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor, thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on single processor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies, such as [MPI](https://searchenterprisedesktop.techtarget.com/definition/message-passing-interface-MPI) and [PVM](https://en.wikipedia.org/wiki/Parallel_Virtual_Machine), are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

# *Thread Basics*

Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction. Specifically:

* A thread does not maintain a list of created threads, nor does it know the thread that created it.
* All threads within a process share the same address space.
* Threads in the same process share (see Figure 1):
  + Process instructions
  + Most data
  + Open files (descriptors)
  + Signals and signal handlers
  + Current working directory
  + User and group id
* Each thread has a unique (see Figure 1):
  + Thread ID
  + Set of registers
  + Stack pointer
  + Stack for local variables and return addresses
  + Signal mask
  + Priority
  + Return value: errno (Pthread functions return "0" if OK)



**Figure 1.** Concept of multithreading

# To create a new thread, the pthread\_create function is used, as follows:

int pthread\_create(pthread\_t \* thread, const pthread\_attr\_t \* attr,

void \* (\*start\_routine)(void \*), void \*arg);

Arguments:

* *thread* - returns the thread id. (unsigned long int defined in bits/[pthreadtypes.h](https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html)).
* *attr* - Set to NULL if default thread attributes are used.
* *void \* (\*start\_routine)* - pointer to the function to be threaded. Function has a single argument: pointer to void.
* \**arg* - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

# To terminate a thread, the pthread\_exit function is used, as follows:

void pthread\_exit(void \*retval);

Arguments:

* *retval* - Return value of thread.

This routine kills the thread. The pthread\_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread\_join.

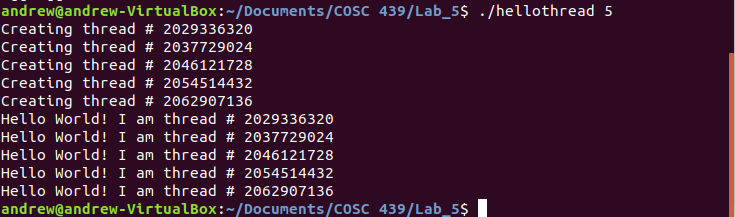
With this, **develop a C program, named hellothread.c, that takes an integer, *n,* at the command line that must be between 1-10 and uses Pthreads and implements 2 functions that do the following:**

* **main()**
  + **Checks the parameter to ensure the integer is between 1-10, if not it prints an error message and quits.**
  + **Creates *n* threads, where *n* is the passed in integer, and prints “Creating thread #<id>” where <id> is the thread id number.**
  + **For each thread, the printHelloThread() function is called.**
* **printHelloThread()**
  + **Receives a thread as a parameter and prints “Hello world! I am thread #<id>” where <id> is the thread id number.**

To implement this, don’t forget to terminate the threads after they are finished. To compile a C program using Pthreads, you need to use the -pthread flag. For example, to compile a hellothread.c program using Pthreads, you would issue the following at the command line:

gcc -pthread -o hellotread hellothread.c

**Include a screenshot of your output here.**



# *Thread Synchronization*

The threads library provides three synchronization mechanisms:

* *mutexes* - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
* *joins* - Make a thread wait till others are complete (terminated).
* *condition variables* - data type pthread\_cond\_t

# *Mutexes*

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes, as do semaphores. See [this](http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#SYNCHRONIZATION) tutorial for more information.

# *Joins*

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completion of the threads with a join.

int pthread\_join(pthread\_t thread, void \*\*retval);

The pthread\_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread\_join() returns immediately. The thread specified by thread must be joinable.

# *Mutex Functions*

int pthread\_mutex\_init (pthread\_mutex\_t \*mutex,   
const pthread\_mutexattr\_t \*attr);

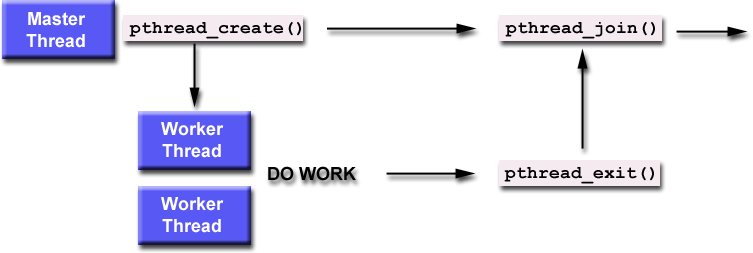
The pthread\_mutex\_init() function initializes the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);

The mutex object referenced by mutex is locked by calling pthread\_mutex\_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

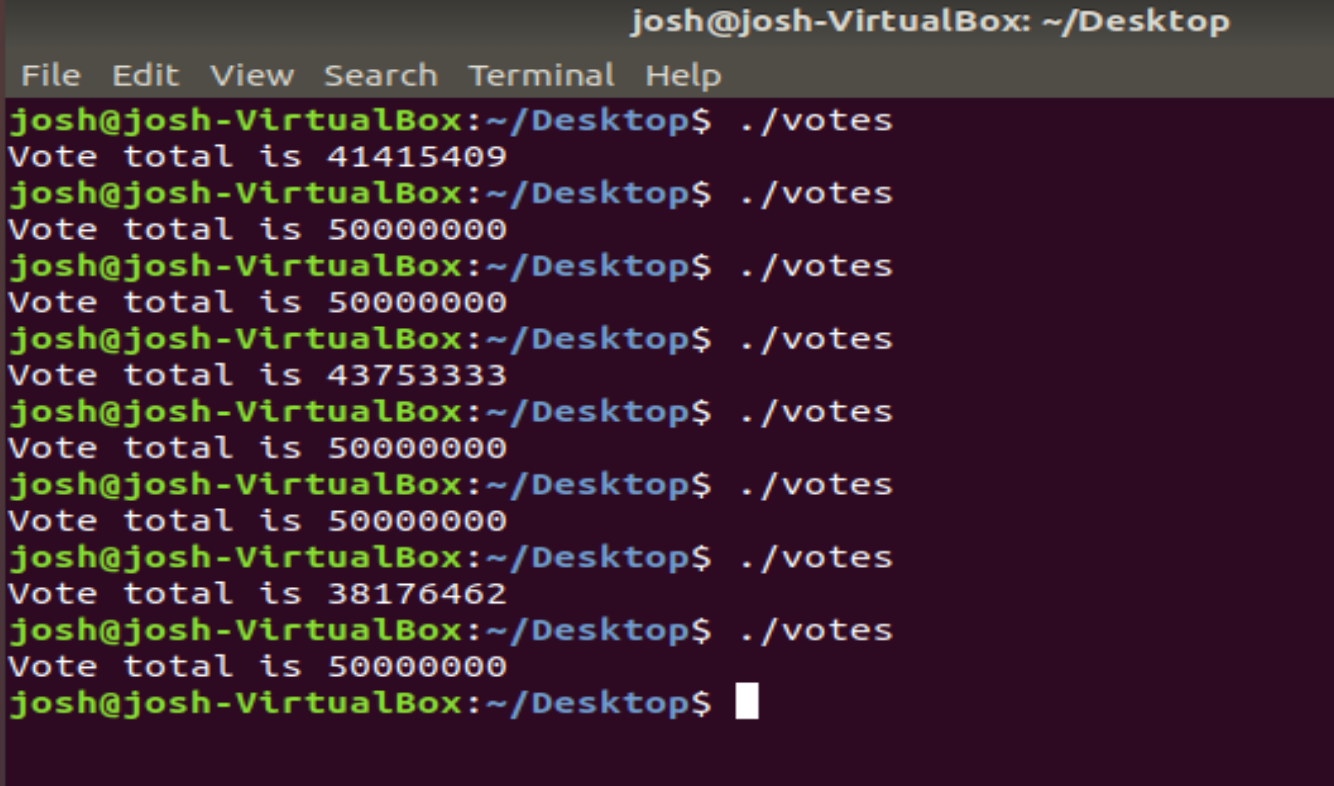
int pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);

The pthread\_mutex\_unlock() function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread\_mutex\_unlock() is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.



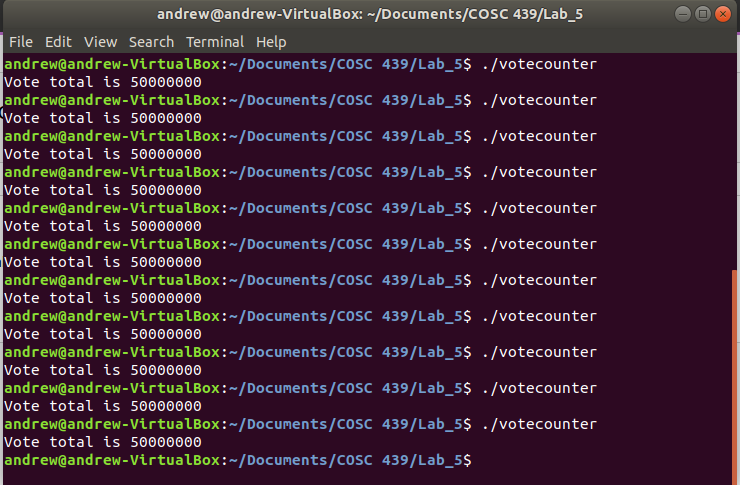
**Figure 2.** Pthread functions implementation.

With this, **compile and run (several times) the provided** [**votecounter**](https://bostonglobe-prod.cdn.arcpublishing.com/resizer/Q-IjWDFmFoq_7KcA8U7ksTgZNAE=/820x0/arc-anglerfish-arc2-prod-bostonglobe.s3.amazonaws.com/public/4DSKQCHGUQI6RGRY4CQIDD275I.jpg)**.c program. You should get an output that varies the vote count output (as shown below), even though it should always produce 5,000,000.**

****

The reason for this difference is that the 5 created threads share the global variable *votes* and can read/write/modify it unchecked – this is not a good idea for critical data (like counting votes)! **Using pthread mutexes, modify the votecounter.c program to ensure that the correct vote count is *always* produced by the program.**

Once you have it working, **include a screenshot of several runs of your output here and include the modified votecounter.c program in your *cpractice* directory.**



**Note:** There are at least 3 [Easter eggs](https://en.wikipedia.org/wiki/Easter_egg_(media)) (this isn’t one of them) in this lab. If you find one, Slack me and let me know!