Report

By: Dylan Silva-Rivas

# Project Programming language

I will be using C for this project.

# Environment

My environment is Ubuntu 22.04.4 LTS x86\_64. This environment is not virtualized, I am running it as its own OS on my dual-booted machine.

# Phase 1

## Objective

The main objective for this phase was to successfully create two threads that represent different banking transactions that a user can perform on a account that has a certain amount of money. But I took a slightly different approach. Instead of creating two threads that represent withdrawals and deposits, I created an array of size NUM\_THREADS that contains elements that can be either a withdrawal or a deposit. After that array was created, I made a struct that represents a user, which was used to create an array of users of size NUM\_THREADS. To simulate a user choosing to withdrawal or deposit from the account, I created another array called arrayOfChoices with a size of NUM\_THREADS. The arrayOfChoices can be filled with either a 1 or 0 representing a withdrawal or a deposit. To complete this phase there’s a for loop that runs from 0 to NUM\_THREADS. Each iteration there is a random value that is generated which represents how much money a user has and each user will get their own ID. The value that the for loop accesses in arrayOfChoices, will determine if the user will deposit or withdrawal, which is controlled by an if-else statement. The threads will only access two functions: withdrawalFunc and depositFunc. Both of these function would take in a void pointer, which would be set equal to a userData pointer. After casting, there are two variables used to obtain the data from the userData pointer variable: UserID and randWithDrawal.

## Challenges

I was struggling to understand void pointers and properly casting that pointer. In this case, it was casted to struct userData. I also was struggling to understand how to obtain certain values from the struct after setting the void pointer equal to a userData pointer. There is different syntax used to point to certain values in a struct pointer variable.

## Solutions

I used stackoverflow to find an answer to my challenge:

<https://stackoverflow.com/questions/14946344/casting-a-void-pointer-to-a-struct>

and I also used a geeksForGeeks article talking about declaring a pointer to a struct:

<https://www.geeksforgeeks.org/how-to-declare-pointer-to-struct-in-c/>

## Outcome

Learned about threads, void pointers, structs and how to use them efficiently.

# phase 2

## Objective

The main objective was to implement mutexes to prevent multiple accounts from accessing the account at the same time, not implementing this would lead to an inaccurate bank account value. Since I created an array of threads that represent banking functions I also had to create an array of mutexes to accurately manage lock and unlocking threads. The size of the mutex array is of size NUM\_THREADS. In the withdrawal and deposit functions mutex one is locked by accessing mutex[userID%NUM\_THREADS] and mutex two is locked by accessing mutex[(userID + 1) % NUM\_THREADS]. After locking, the functions perform their actions. The withdrawal function will withdrawal money obtained from the userData struct that is passed into the function. The deposit function does the same thing but deposits money into the account. Then, mutexes are unlocked the same way they are locked.

## Challenges

Figuring out that an array of mutexes was necessary instead of creating two mutexes. If using two mutex variables will cause a deadlock and can’t be resolved even with timeout functionality.

## Solution

The solution was to create a mutex element for each thread element.

## Outcome

Learned about mutexes and their relationship with threads.

# phase 3

## Objective

The main objective for this phase was to cause a deadlock. I looked at this phase as a tiny extension of phase 2. To really cause a deadlock depends on how you organize your mutex locks and unlocks. The way I caused deadlock was to lock mutexOne then mutexTwo. After more code execution mutexTwo is unlocked first then mutexOne is unlocked. In a random case one thread could lock mutexOne and another thread could lock mutexTwo which leads to a deadlock.

## Challenges

The challenge here was figuring out how to cause a deadlock.

## Solutions

I watched this video: <https://www.youtube.com/watch?v=LjWug2tvSBU>.

## Outcome

Successfully caused a deadlock.

# phase 4

## Objective

The objective for phase 4 was to implement a timeout function that successfully breaks a deadlock. To implement a timeout function I had to use the signal function. Before creating a signal function, I had to create a volatile variable. A volatile variable lets the computer know that a value could be changed outside of the control of the program. After creating that variable I created a timeout\_handler. The struct alters the volatile timeoutFlag and unlocks all mutexes. In the threadSimluation function, signal function takes in SIGALRM, and timeout\_handler as parameters. The alarm function takes 3 seconds which terminated receiving processes. This sends the process an asynchronous notification.

## Challenges

Figuring out how to create a timeout function in C. There were a lot of sources there took different approaches to creating timeout until I found a more “reliable” way of doing it.

## Solutions

I used this source to learn about signals in C: [https://thelinuxcode.com/sigalarm\_alarm\_c\_language/#google\_vignette](https://thelinuxcode.com/sigalarm_alarm_c_language/" \l "google_vignette)

## Outcome

Timeout\_handler successfully manages deadlocks and lets the user know that a timeout occurred.

# Reflection

This project was very beneficial to my understanding of threading and the C programming language. I have a stronger understanding of creating programs in C. Hopefully this knowledge will be useful for OS development if I ever decide on doing that :)