Lab 2 – Pthreads Mutex locks and Semaphores in C on Linux
Operating Systems SFWRENG 3SH3, Winter 2025
Prof. Neerja Mhaskar

1. You must show your working solution of parts I and II of this lab to the TA for a grade.
2. For Mac M1, M2, and M3 users (all Macs with 64-bit ARM CPUs), you need to install UTM for virtualization: https://mac.getutm.app/
3. The TA will check your solution and will quiz you on your work. After which they will enter your mark and feedback on Avenue.
4. If you do not show your work to your Lab TA, you will get a zero (unless you provide an MSAF, in which case this lab's weight will be moved to Assignment 2).
5. It is your responsibility to connect with your Lab TA to get a grade and ensure that your grade has indeed been posted on Avenue.

Outline:

In this section you will learn how to create threads using the Pthreads (POSIX standard) API.

1. The C program shown on the next page demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.
2. In a Pthreads program, separate threads begin execution in a specified function. In the code on the next page, this is the `runner()` function.
3. When this program begins, a single thread of control begins in `main()`.
4. After some initialization, main() creates a second thread that begins control in the `runner()` function. ***Both threads share the global data sum.***
5. All Pthreads programs must include the `pthread.h` header file.
6. The statement `pthread_t tid` declares the identifier for the thread we will create. Each thread has a set of attributes, including stack size and scheduling information. The `pthread_attr_t attr` declaration represents the attributes for the thread. We set the attributes in the function call `pthread_attr_init(&attr)`. Because we did not explicitly set any attributes, we use the default attributes provided.
7. A separate thread is created with the `pthread_create()` function call. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution—in this case, the `runner()` function. Last, we pass the integer parameter that was provided on the command line, `argv[1]`.

```c
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid;  /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
 pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

8. At this point, the program has two threads: the initial (or parent) thread in `main()` and the summation (or child) thread performing the summation operation in the `runner()` function. This program follows the thread *create/join* strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the `pthread_join()` function.
9. The summation thread will terminate when it calls the function `pthread_exit()`.
10. Once the summation thread has returned, the parent thread will output the value of the shared data `sum`.

In your programs you may need to create more than one thread. A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple `for` loop.

```
#define NUM THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

Lab question

Given a list of size 20 consisting of natural numbers, write a multithreaded C program for adding all the numbers in the list as follows: The list of numbers is divided into two smaller lists of equal size. Two separate threads (which we will term as ***summing threads***) add numbers in each sublist.

Because global data are shared across all threads, the easiest way to set up the data is to create a global array. Each ***summing thread*** will work on one half of this array. This lab will require passing parameters to each of the **summing threads**. It will be necessary to identify the starting index and ending index of the sublist in which each thread is to begin adding numbers. The parent thread will output the sum once all summing threads have exited.

You are to write a C program using `Pthreads` that contains the entire solution for this question. In particular your program needs to do the following:

1. To be able to create threads in your C program you need to include the `pthread.h` header file.
2. Each thread has a unique thread ID. To create thread IDs for your threads in your program you should use the `pthread_t` data type.
3. Thread attributes should be created/modified using `pthread_attr_t` structure.
4. Declare and code the function in which the thread begins control. For an example see the `runner()` function on the previous page.

5. To be able to identify the starting index and ending index of the sublist in which each thread begins adding numbers you can do the following:
    a. Create a structure to store the starting index and ending index of the sublist. For example:

```
typedef struct { int from_index;
int to_index; } parameters;
```

6. Since threads can share heap, you can simply create a variable of type `parameters`, allocate memory for it on the heap, and assign values to its members as follows:

```
parameters *data =
(parameters *) malloc (sizeof(parameters));
data->from_index = 0; data->to_index = (SIZE/2) - 1;
```

7. To create threads use `pthread_create()` function and pass in the necessary parameters.
8. For the parent thread to output the sum after all summing threads have exited, it is important that you use the `pthread_join()` function.
9. Whenever you dynamically allocate memory on the heap, it is important that you deallocate/free this memory when it is not required by the program using the `free()` function.
10. To compile your program, you need to use the `-pthread` option as follows:

```
gcc -pthread -o PLthreads PLthreads.c
```

Sample Output for the following list:

```
List={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}
./PLthreads
---------------------------------------------------------------------------------------------------
Sum of numbers in the list is: 210
```

Outline: PART I

Banking System Problem

Consider a simple banking system that maintains bank accounts for its users. Every bank account has a balance (represented by an integer variable **`amount`**). The bank allows deposits and withdrawals from these bank accounts, represented by the two functions: **`deposit`** and **`withdraw`**. These functions are passed an integer value that is to be deposited or withdrawn from the bank account. Assume that a husband and wife share a bank account. The husband only withdraws from the account and the wife only deposits into the account using the **`withdraw`** and **`deposit`** functions respectively. Race condition is possible when the shared data (**`amount`**) is accessed by these two functions concurrently.

In this lab you are to write a C program that provides a critical section solution to the Banking System Problem using mutex locks provided by the POSIX `Pthreads` API. In particular, your solution needs to do the following:

1. Take **two command line arguments**. First argument is the amount to be deposited (an integer value) and the second argument is the amount to be withdrawn (an integer value).
2. Create a total of 6 threads that run concurrently using the `Pthreads` API.
    a. 3 of the 6 threads call the `deposit()` function, and
    b. 3 of the 6 threads call the `withdraw()` function.
3. Create the threads calling the `deposit()` function using the `pthread_Create()` function. While creating these threads you need to pass the thread identifier, the attributes for the thread, `deposit()` function, and the first integer command line argument `argv[1]` (which is the amount to be deposited).
4. Similarly, create the threads calling the `withdraw()` function.
5. To achieve mutual exclusion use mutex locks provided by the `Pthreads` API.
6. You are to provide print statements that output an error message if an error occurs while creating threads, mutex locks etc.
7. Your program should print the value of the shared variable `amount`, whenever it is modified.
8. Finally, the parent thread should output the final `amount` value after all threads finish their execution.

Make sure you use pthread_join() for all the threads created. This will ensure that the parent thread waits for all the threads to finish, and the final amount reported by main is correct for every execution of the program.

Notes:

1. See lecture slides on Chapters 6&7 for using mutex locks provided by the `pthreads` API.
2. You may see that the amount is negative. This could happen if the threads calling the withdraw function are scheduled to run on the CPU before the deposit function. This is acceptable for PART-I of this lab.

Sample Output: ./PLmutex 100 50

Withdrawal amount = -50
Withdrawal amount = -100
Withdrawal amount = -150
Deposit amount = -50
Deposit amount = 50
Deposit amount = 150
Final amount = 150

Part II

In this part you are to modify your C program created for Part I to ensure the following conditions are met:

1. Withdrawals don't take place if `amount <=0`.
2. Deposits don't take place if `amount >=400`.
3. The amount of money deposited or withdrawn at a given time is 100.
4. Your program should create a total of **10** threads that run concurrently. **7 of 10** threads call the `deposit()` function and **3 of 10** threads call the `withdraw()` function.

In particular, your solution needs to do the following:

1. Take **one command line argument**. Since the amount of money withdrawn/ deposited at a given time is 100, the value of the command line argument is 100.
2. You are to use mutex locks provided by the Pthreads API to achieve mutual exclusion as explained in PART-I.
3. You are to use **two semaphores** to ensure **condition 1 and condition 2** are met.
4. Additionally, you need to set the initial value of these semaphores correctly for your solution to work.
5. You are to provide print statements that output an error message if an error occurs while creating threads, mutex locks semaphores etc.
6. You are to provide print statements in the deposit and withdraw functions that output the value of the shared variable '`amount`' after each modification.
7. You are to provide print statements at the beginning of the `deposit()` and `withdraw()` function. This way you can see (through the print statements) when threads beginning their execution in their functions.
8. Finally, the parent thread should output the final amount value after all threads finish their execution. Since deposit function gets executed 7 times and withdraw () function gets executed 3 times the correct final amount = 7*100 – 3*100 = 700-300 = 400.

Make sure you use pthread_join() for all the threads created. This will ensure that the parent thread waits for all the threads to finish, and the final amount reported by this thread is correct for every execution of the program.

Notes:

1. See lecture slides on Chapters 6&7 to use mutex locks provided by the `pthreads` API and to use semaphores provided by the `POSIX SEM extension`.
2. **If you see negative amount values, your solution is incorrect.**

Sample Output: ./PLsem 100

Executing deposit function
Amount after deposit = 100
Executing Withdraw function

Amount after Withdrawal = 0
Executing Withdraw function
Executing Withdraw function
Executing deposit function
Amount after deposit = 100
Amount after Withdrawal = 0
Executing deposit function
Amount after deposit = 100
Amount after Withdrawal = 0
Executing deposit function
Amount after deposit = 100
Executing deposit function
Amount after deposit = 200
Executing deposit function
Amount after deposit = 300
Executing deposit function
Amount after deposit = 400
Final amount = 400