Bahir Dar University

Cseg 2154 Operating Systems

Lab # 8

Threads Synchronization (Mutex)

Objective:

When multiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities.

Threads need to synchronize their activities to effectively interact. This includes:

- Implicit communication through the modification of shared data
- Explicit communication by informing each other of events that have occurred.

This lab describes the synchronization types available with threads (the Linux/Unix library called pthreads) and discusses when and how to use synchronization. There are a few possible methods of synchronizing threads:

- Mutual Exclusion (Mutex) Locks
- Condition Variables

Here we will discuss Mutexes only.

Example:

The following program is the thread demonstration program which you saw in the previous class. Almost all of you were surprised when you run the program because it didn't give you the output you expected. Please write down what you expect the output of the program would be before you run the code.

The output of the program I expected and why I expected that output is given bellow the code (In case you need some help)

/*Lab2.c*/

```
#include <stdio.h>
#include <pthread.h>
int glob_data = 5 ;

void *kidfunc(void *p) {
        printf ("Kid here. Global data was %d.\n", glob_data) ;
        glob_data = 15;
        printf ("Kid Again. Global data was now %d.\n", glob_data) ;
}

void main ( ) {
        pthread_t kid ;
        pthread_create (&kid, NULL, kidfunc, NULL);
        printf ("Parent here. Global data = %d\n", glob_data);
        glob_data = 10;
        pthread_join (kid, NULL);
        printf ("End of program. Global data = %d\n", glob_data);
}
```

Here is the output of the program I expect.

```
Parent here. Global data = 5
Kid here. Global data was = 10
Kid Again. Global data was now = 15
End of program. Global data = 15
```

Explanation about the output I expected:

One may think that first the main thread starts execution and executes the statements:

```
        printf ("Parent here. Global data = %d\n", glob_data);
        glob_data = 10;
        pthread_join (kid, NULL);
```

and since the last statement thread_join function tells the scheduler to stop executing the current thread and first finish executing the other thread (kid). The scheduler context switches to the second thread and executes the statements

```
printf ("Kid here. Global data was %d.\n", glob_data);
```

```
glob_data = 15;

printf ("Kid Again. Global data was now %d.\n", glob_data);
```

Since the kid thread is finished it now switches back to the main thread (which was waiting for the kid thread to finish). Then the main thread executes the following statement and finishes.

```
printf ("End of program. Global data = %d\n", glob_data);
```

Therefore, if the program is executed the way it is explained above (Just writing down the statements in printf() and replacing the value of glob_data with the actual value) the output would look like:

```
Parent here. Global data = 5
Kid here. Global data was = 10
Kid Again. Global data was now = 15
End of program. Global data = 15
```

compile the program as:

gcc hellothread.c –o prog -**lpthread**

Then run your program as:

./prog

Actual Output of Lab2.c

The actual output of the program looked like this in my computer.

```
Parent here. Global data = 5
Kid here. Global data was = 5
Kid Again. Global data was now = 15
End of program. Global data = 15
```

Explanation:

After the statement pthread_create there are two threads. Therefore the scheduler may pick anyone of them for execution. Therefore, there is no predefined order of execution. (Meaning no one can accurately define the

output of the program, since it depends on which thread the scheduler picks to run and for how long.)

For example, here is a sample execution sequence of the two threads in my PC.

The scheduler executes the statement

```
printf ("Parent here. Global data = %d\n", glob_data);
```

and context switches to execute kid thread. then it executes the statement

```
printf ("Kid here. Global data was %d.\n", glob_data) ;
```

and context switches to execute main thread and executes the statements

```
glob_data = 10;
```

```
pthread_join (kid, NULL);
```

the second pthread_join statement makes the main thread wait for the kid thread. (It is just saying to the scheduler; "I don't want you to execute any more statements unless the kid thread executes). Therefore, the scheduler goes ahead and executes kid thread until it finishes before coming to the main thread. Therefore, it executes the following statements from the kid thread

```
glob_data = 15;
```

```
printf ("Kid Again. Global data was now %d.\n", glob_data);
```

Since the kid thread is finished it now switches back to the main thread (which was waiting for the kid thread to finish). Then the main thread executes the following statement and finishes.

```
printf ("End of program. Global data = %d\n", glob_data);
```

Therefore, if the program is executed the way it is explained above (Just writing down the statements in printf() and replacing the value of glob_data with the actual value) the output would look like:

```
Parent here. Global data = 5
Kid here. Global data was 5
```

Kid Again. Global data was now 15
End of program. Global data = 15

What we do we need to make the actual output of the program similar to the expected one? We should make sure that the main thread will not be interrupted when executing the following statemets.

```
printf ("Parent here. Global data = %d\n", glob_data);
glob_data = 10;
```

Therefore, the above statements are critical section. Also the kid thread shouldn't be interrupted while executing the following statements:

```
printf ("Kid here. Global data was %d.\n", glob_data) ;
glob_data = 15;
printf ("Kid Again. Global data was now %d.\n", glob_data) ;
```

This is the critical section of the second thread.

In this lab session we are going to try to achieve mutual execlusion (no interruption when executing critical regions) using Mutexes.

Mutex variables must be of type pthread_mutex_t.

The function pthread_mutex_init(&mutex, NULL) creates and initializes a new mutex mutex object, and sets its attributes to NULL (i.e no attributes. You don't need to care about this attribute thing. It hope it will be covered indepth in your system programming class).

The function pthread_mutex_lock (mutex) locks the mutex.
pthread_mutex_trylock (pthread_mutex_t mutex)
pthread_mutex_unlock (pthread_mutex_t mutex)

The function pthread_mutex_lock(&mutex) routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, the call will block the calling thread until the mutex is unlocked.

The function pthread_mutex_unlock( ) will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data.

Example 2. Implementing the critical sections we defined earlier using

mutexes.

```
/*Lab2.c*/
#include <stdio.h>
#include <pthread.h>
int glob_data = 5 ;

/* This is the lock for thread synchronization */
pthread_mutex_t    my_sync;


void *kidfunc(void *p) {
        /* Lock the mutex when its our turn */

        pthread_mutex_lock(&my_sync);

        /*This is a critical section*/
        printf ("Kid here. Global data was %d.\n", glob_data) ;
        glob_data = 15;
        printf ("Kid Again. Global data was now %d.\n", glob_data) ;
        /*End of the critical section*/
        pthread_mutex_unlock(&my_sync);

}

void main ( ) {
        pthread_t kid ;
        pthread_create (&kid, NULL, kidfunc, NULL);

        pthread_mutex_init (&my_sync,NULL);

        /* Lock the mutex when its our turn */

        pthread_mutex_lock(&my_sync);

        /*This is a critical section*/
        printf ("Parent here. Global data = %d\n", glob_data);
        glob_data = 10;
        /*End of the critical section*/
        pthread_mutex_unlock(&my_sync);
```

```
        pthread_join (kid, NULL);
        printf ("End of program. Global data = %d\n", glob_data);
}
```