CS232L Operating Systems Lab
Lab 11: Synchronization between threads

CS Program
Habib University

Fall 2019

1 Introduction

When multiple threads work together, especially when they share data between them, it is important that they access shared resources in a synchronized manner i.e., one thread does not access data when another is modifying it.

The operating systems provide different mechanisms for synchronization between threads and processes. POSIX libraries also provides locks and other structures to perform synchronizations.

In this lab we will learn:

1. sharing global variables between threads
2. using locks to synchronize access to shared variables

2 Race conditions

One advantage of threads is that it is very easy to share data among threads by using golbal variables. But this comes at a price, namely, the programmer must make sure that multiple threads do not modify the same variable at the same time.

```c
/* race.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <pthread.h>

#define INC_SIZE 1000

static volatile int glob = 0;

void * access_global (void * args){

  char *p = (char *)args;
  int loc = 0;

  printf("%s: thread started\n", p);
  for (int i=0; i<INC_SIZE; i++) {
    loc = glob;
    loc++;
    glob = loc;
  }

  printf("%s: thread ending\n", p);
  return NULL;
}
```

```c

int main(int argc, char * argv[]){

  pthread_t t1, t2;  //thread identifier

  //create a new thread that runs hello_arg with argument hello
  printf("main: creating threads\n");
  pthread_create(&t1, NULL, access_global, "T0");
  pthread_create(&t2, NULL, access_global, "T1");
  printf("main: creatred threads\n");

  //wait until the thread completes
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("main: joined threads\n");
  printf("main: glob = %d\n", glob);

  return 0;
}
```

Listing 1: Accessing global data (thread_incr.c)

Listing 1 [1]
on page 1 shows a code that accesses a global variable *glob* and launches two threads which try to modify this global variable. Try guessing the output of the code and then compile and run the program to see whether you got it correct.

Now increase the size of the constant INC_SIZE to 10 million and run the same experiment again. Did any thing change?

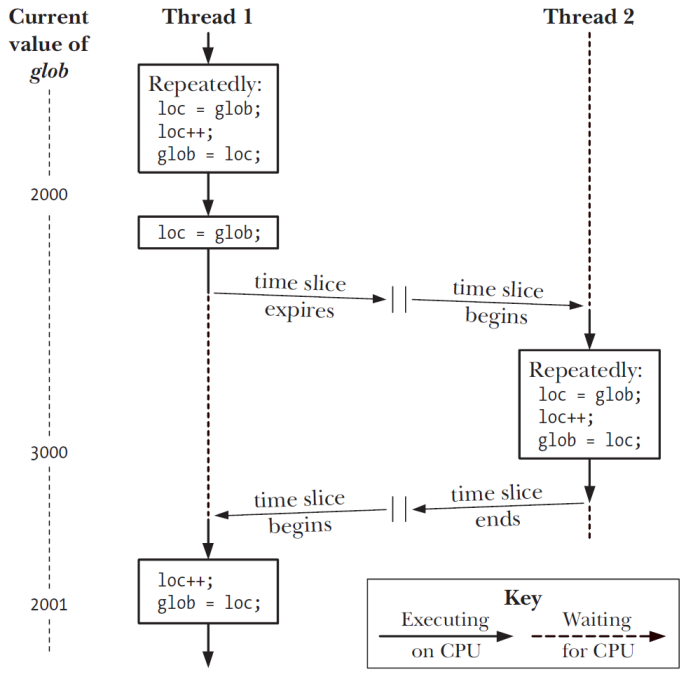

Figure 1: Race condition

The output can be explained by the fact that since a thread is an independent unit of execution, its scheduling is performed by the OS scheduler and is therefore beyond our control.

[1] The three instructions for updating *glob* inside the for loop are there to imitate the *load*, *update*, and *store* behaviour of RISC processors. Replacing them with a single *glob++* would not mitigate the race condiiton.

Therefore it is possible that multiple threads access and modify the global variable simultaneously. On a single core processor this problem is manifested when a thread *Thread 1* which is trying to update a global variable is interrupted before it can finish its update and a second thread *Thread 2* is scheduled which accesses and updates the same global variable. The previous thread *Thread 1* when rescheduled tries to finish its update, which was interrupted, and overwrites the last value updated by *Thread 2* with the old value that it had stored in itd context. This situation is graphically explained in the Figure 1.

The problem arose because both threads tried to modify the shared variable *concurrently*. [2] The section of code that accesses a shared variable is called *critical section* and should be executed *atomically* otherwise we would have on our hands what is called a *race condition*.

3 Mutexes

Another related concept is *mutual exclusion* i.e. we make sure that only on thread can enter its critical section at any given time. If *Thread 1* is in its critical section and gets interrupted and *Thread 2* is scheduled and tries to enter its critical section, it should not be able to do so and shall block until *Thread 1* has finished updating its shared variables and is out of its critical section.

Mutexes, also known as locks, are the primitive most OSes provide to achieve mutual exclusion.

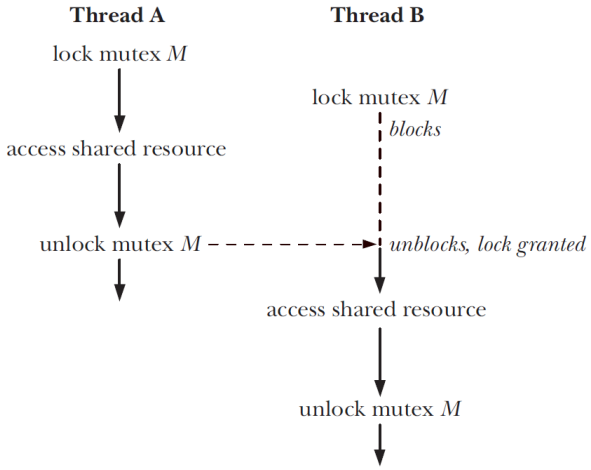

Figure 2: mutual exclusion

A mutex has two states *locked* and *unlocked*. At any given moment at most one thread may hold the lock on the mutex. If another thread tries to get the lock on the same mutex at this time, it would block. It would unblock only after the first thread has unlocked that mutex. This is shown in Figure 2 on page 3.

In general a separate mutex is associated with each shared resource or with each critical section. Each thread then follows the following protocol when accessing shared resources:

- lock the mutex for *that* shared resource

[2]The concurrency problem happens for different reasons on uniprocessors and multiprocessors. On a multiprocessor system multiple threads are running at the same time and it is possible that critical sections of two or more threads execute simultaneously and try to modify the shared variable at the same time.
On a uniprocessor system while multiple threads cannot execute simultaneously, the same affects result if the execution of a critical section of a thread is interrupted and the critical section of another thread is exectued before the original thread gets a chance to complete its critical section.

- use the shared resource
- unlock the mutex

This ensures that only one thread can access enter the critical section, i.e., access the shared resource, at any given time.

```c
/* race.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <pthread.h>

#define INC_SIZE 10000000

static volatile int glob = 0;
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void * access_global (void * args){

  char *p = (char *)args;
  int loc = 0;

  printf("%s: thread started\n", p);
  for (int i=0; i<INC_SIZE; i++) {
    pthread_mutex_lock(&m);
    loc = glob;
    loc++;
    glob = loc;
    pthread_mutex_unlock(&m);
  }

  printf("%s: thread ending\n", p);
  return NULL;
}

int main(int argc, char * argv[]){

  pthread_t t1, t2;  //thread identifier

  //create a new thread that runs hello_arg with argument hello
  printf("main: creating threads\n");
  pthread_create(&t1, NULL, access_global, "T0");
  pthread_create(&t2, NULL, access_global, "T1");
  printf("main: creatred threads\n");

  //wait until the thread completes
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("main: joined threads\n");
  printf("main: glob = %d\n", glob);

  return 0;
}
```

Listing 2: Accessing global data with mutual exclusion (thread_incr_mutex.c)

Listing 2 on page 4 shows the modified code using POSIX mutexes. The PTHREAD library provides functions to initialize, lock and unlock mutexes.

The function *pthread_mutex_lock()* would lock a To lock a mutex, we specify the mutex in a call to *pthread_mutex_lock()*. If the mutex is currently unlocked, this call locks the mutex and returns immediately. If the mutex is currently locked by another thread, then *pthread_mutex_lock()* blocks until the mutex is unlocked, at which point it locks the mutex and returns.

What do you think would happen if a thread having locked a mutex calls the *pthread_mutex_lock()* again on the same mutex? [3].

The *pthread_mutex_unlock()* function unlocks a mutex previously locked by the calling thread. It is an error to unlock a mutex that is not currently locked, or to unlock a mutex that is locked by another thread.

If more than one other thread is waiting to acquire the mutex unlocked by a call to *pthread_mutex_unlock()*, it is indeterminate which thread will succeed in acquiring it [4] .

3.1 todo: try other functions

Read the documentation of the functions provided in the footnote and try rewriting the code in listing 2 using those functions.

4 Performance

4.1 todo: performacne

Try to measure the run times for both your codes i.e. with and without mutexes. Do you see any difference? How would you explain it?

5 Deadlocks

Thread A	**Thread B**
1. *pthread_mutex_lock(mutex1);*	1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex2);* blocks	2. *pthread_mutex_lock(mutex1);* blocks

Figure 3: Example of a deadlock.

A deadlock is a situation of a circular wait which makes the progression of threads impossible. Imagine two threads which are using two mutexes to synchronize access to two different shared resources. *Thread A* succeeds in locking *mutex1* and tries to lock *mutex2*. But before it does so, it gets interrupts and *Thread B* gets scheduled, locks *mutex2* successfully and issues a call to lock *mutex1*. Since *mutex1* is being held by *Thread A*, it will get blocked and uptimately descheduled. *Thread A* will be scheduled again and try to lock *mutex2*. Since that particular mutex is being held by *Thread 2*, it will block too, resulting in a deadlock. The situation is depicted in the Figure 3 on page 5.

If we are not careful in our synchronizations, deadlocks may result and our programs can be blocked *for ever*.

This two thread deadlock logic can be extened to more than two threads as well as a single thread.

[3] hint: google the term deadlock.

[4] Read the man pages for pthread mutexes. The library provides many more functions e.g. pthread_mutex_trylock() and pthread_mutex_timedlock() as well as pthread_mutex_init() and pthread_mutex_destroy().

6 Todo: Linked list

Make functions in a program that initialize, insert elements in, delete elements from, and count the elements in a linked list of integers. Declare a global linked list initialized to NULL.

6.1 single threaded

Test the functions by calling adding 100000 elements in the list, deleting 1000 of them and then counting the remaining elements of the list. All in a single threaded program.

6.2 multiple threaded

Modify your program to create multiple threads that each adds 100000 elements to the list. The main thread should wait for all threads to finish and then print the number of elements in the list.