Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

09/23/2021

Lab 03: Multiple Threads and Locks

Purpose of Lab

The purpose of the lab was to show the importance controlling what and when a thread can access data in different parts of a program. Since threads operate concurrently, race conditions can exist very easily if the programmer is not careful. This lab demonstrates how to implement ways of protecting variables from being changed before they should, preventing a race condition. One way this is done is with mutual exclusion. Some code may not execute at all if the main process doesn't include a join, making the process wait for the threads to finish their execution. The programmer must be very cognizant of what a thread can manipulate, and when it is manipulating something. If the programmer isn't aware, the code can have a variety of undesired results.

Example Code

print_num.c

This example shows how the use of a global variable can affect what a function will output. Even though the global value is incremented in main, the results of the variable “i” being incremented changes the values of the thread displaying the value of “i”. This is because the thread is in the same address space of main.

C: > Users > pakum > Desktop > operatingSystems > lab03 > code > C print_num.c > .

```c
// File print_num.c
// gcc print_num.c -o print_num -l pthread

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int i; //global variable to share data

//This function will run concurrently.
void *print_num(void *ptr)
{
    while(1)
    {
        sleep(1);
        printf("While in print_num: %d\n", i);
    }
}

int main()
{
    pthread_t t1;
    i = 1;
    int iret1 = pthread_create(&t1, NULL, print_num, NULL);
    while(1)
    {
        sleep(4);
        printf("While in main: %d\n", i);
        i++;
    }
    exit(0);
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ gcc print_num.c -o print_num -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_num
While in print_num: 1
While in print_num: 1
While in print_num: 1
While in main: 1
While in print_num: 2
While in print_num: 2
While in print_num: 2
While in print_num: 2
While in main: 2
While in print_num: 3
While in print_num: 3
While in print_num: 3
While in print_num: 3
While in main: 3
While in print_num: 4
While in print_num: 4
While in print_num: 4
```

print_random.c

This example shows that the thread is not guaranteed to execute its function completely, even though the function is called in the main function. This is because the main function does not wait for the thread to complete its task unless there is a join statement. Therefore, depending on how long the main function takes to execute, there is a chance that the function “print_random” will begin to display to the terminal.

C: > Users > pakum > Desktop > operatingSystems > lab03 > code > C print_random.c > ...

```c
// File print_random.c
// gcc print_random.c -o print_random -l pthread

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

//This func will run concurrently
void *print_random(void *ptr)
{
    printf("a\n");
    printf("b\n");
}

int main()
{
    pthread_t t1;
    int i = 1;
    int iret1 = pthread_create(&t1, NULL, print_random, NULL);
    //pthread_join(t1, NULL);
    printf("c\n");
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./print_random
c
a
b
b
```

p_unsafe.c

In this example, the dangers of using a global variable are shown again. A thread and the main function both call the function “last_char”, which uses the global character pointer “c”. Both the main process and the thread will access the variable “c” at differing times because they have different sleep intervals. At some periods of time, “c” is set to NULL and the thread will attempt to access “c” when it is null, causing a segmentation fault because “c” isn’t a real char value. There are many ways this problem can be solved, which could be not using a global variable, not setting the global variable to NULL, using join to wait for the thread before accessing “c”, or using a mutex lock.

: > Users > pakum > Desktop > operatingSystems > lab03 > code > C p_unsafe.c > ...

```c
// File: p_unsafe.c
// gcc p_unsafe.c -o p_unsafe -lpthread
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

char s1[] = "abcdefg" ;
char s2[] = "abc" ;
char* c ;

void last_char(char* a, int i)
{
  printf("last letter a is %s and i is %d\n", a, i) ;
  sleep(i) ;
  //c = NULL ;
  sleep(i) ;
  c = a ;
  sleep(i) ;
  while (*(c))
  {
  c++ ;
  printf("%c\n", *(c-1)) ;
  }
}

// This function will run concurrently.
void *call_last_char(void *ptr)
{
  last_char(s2, 2) ;
}

int main()
{
  pthread_t t1 ;
  int iret1 = pthread_create(&t1, NULL, call_last_char, NULL) ;
  last_char(s1, 5) ;
  sleep(10) ;
  printf("Ended nicely this time\n") ;
  exit(0) ;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ gcc p_unsafe.c -o p_unsafe -lpthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./p_unsafe
last letter a is abcdefg and i is 5
last letter a is abc and i is 2
a
b
c
a
b
c
d
e
f
g
Ended nicely this time
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ gcc p_unsafe.c -o p_unsafe -lpthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./p_unsafe
last letter a is abcdefg and i is 5
last letter a is abc and i is 2
Segmentation fault (core dumped)
```

p_mutex.c

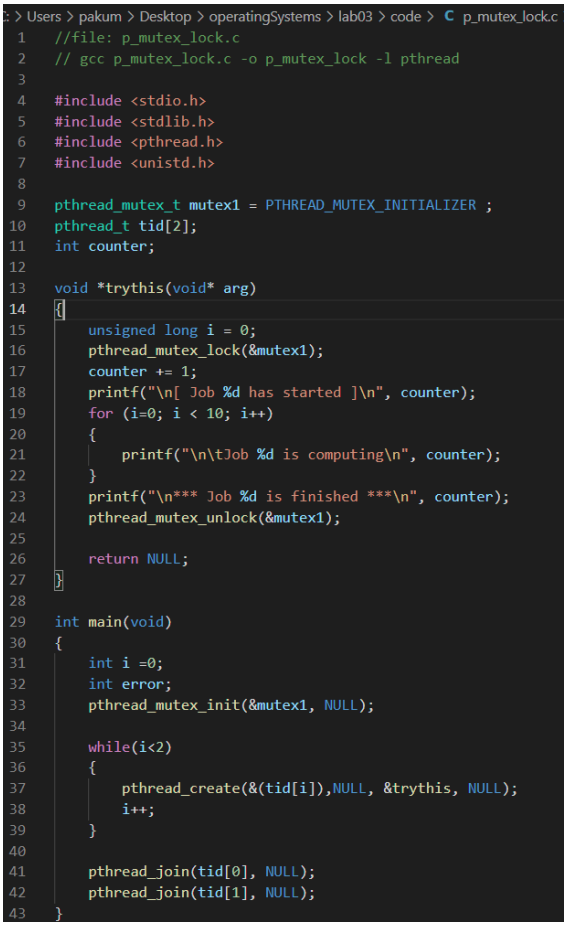
This example shows how to use mutual exclusion in order to only allow one thread into a critical region at a given time. This is important if there is a variable that both threads manipulate and require that variable not to change while some other processing occurs. In this instance, the order that text is displayed is protected so that each thread will print when it is in the critical region instead of both printing at the same time because they are in what is supposed to be the critical region. To setup the critical region, a mutex lock is used to only allow one thread to execute code until it is unlocked.

```
: > Users > pakum > Desktop > operatingSystems > lab03 > code > C p_mutex.c >
1   // File: p_mutex.c
2   // gcc p_mutex.c -o p_mutex -lpthread
3   //
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <pthread.h>
7   #include <unistd.h>
8   pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER ;
9   void print_critical_region(char* a, char* b)
10  {
11    pthread_mutex_lock(&mutex1);
12    printf("1: %s\n", a) ;
13    sleep(1);
14    printf("2: %s\n", b) ;
15    pthread_mutex_unlock(&mutex1);
16  }
17  // These two functions will run concurrently.
18  void* print_i(void *ptr)
19  {
20    print_critical_region("I am", " in i") ;
21  }
22  void* print_j(void *ptr)
23  {
24    print_critical_region("I am", " in j") ;
25  }
26  int main()
27  {
28    pthread_t t1, t2 ;
29    pthread_mutex_init(&mutex1, NULL);
30    int iret1 = pthread_create(&t1, NULL, print_i, NULL) ;
31    int iret2 = pthread_create(&t2, NULL, print_j, NULL) ;
32    while(1)
33    {}
34    exit(0) ;
35  }
```

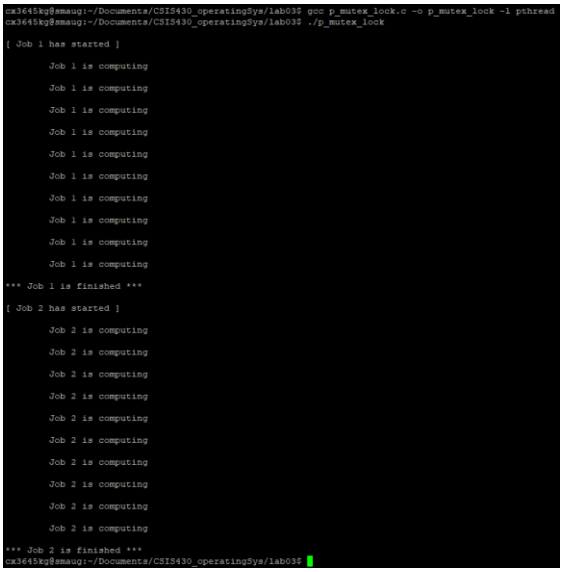
```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ gcc p_mutex.c -o p_mutex -lpthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./p_mutex
1: I am
2:  in i
1: I am
2:  in j
^C
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ gcc p_mutex.c -o p_mutex -lpthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./p_mutex
1: I am
1: I am
2:  in i
2:  in j
```

p_mutex_lock.c

This example is very similar to the previous example, but there is an actual global variable being used called “counter”. If two threads both call the “trythis” function at the same time, the “counter” variable is incremented twice, so the first thread will be displaying incorrect information because “counter” should be one less than what it is. To prevent this, a mutex lock is setup between all the code that should only be executed by one thread at a time. Then one thread will complete its code before the next thread can execute the same code in the critical region.




```c
//file: p_mutex_lock.c
// gcc p_mutex_lock.c -o p_mutex_lock -l pthread

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER ;
pthread_t tid[2];
int counter;

void *trythis(void* arg)
{
    unsigned long i = 0;
    pthread_mutex_lock(&mutex1);
    counter += 1;
    printf("\n[ Job %d has started ]\n", counter);
    for (i=0; i < 10; i++)
    {
        printf("\n\tJob %d is computing\n", counter);
    }
    printf("\n*** Job %d is finished ***\n", counter);
    pthread_mutex_unlock(&mutex1);

    return NULL;
}

int main(void)
{
    int i =0;
    int error;
    pthread_mutex_init(&mutex1, NULL);

    while(i<2)
    {
        pthread_create(&(tid[i]),NULL, &trythis, NULL);
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```


cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ gcc p_mutex_lock.c -o p_mutex_lock -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$ ./p_mutex_lock
[Job 1 has started]
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
Job 1 is computing
*** Job 1 is finished ***
[Job 2 has started]
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
Job 2 is computing
*** Job 2 is finished ***
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab03$

Conclusion

This lab was useful in showing the practical implications of using threads. The dangers of global variables were obvious, as one thread can easily be using the wrong value if another thread or process changes the global variable. The use of mutex locks is one way to prevent global values from being manipulated while a thread is still using the them, which is considered being in a critical region. Also, this lab showed the importance of using join statements with the creation of threads, otherwise the code that the thread is supposed to execute may never be executed completely or some code might be executed in the wrong order without a join statement. Knowing what values threads use, when the values are used, and when threads are executed are all very important when working with threads and the programmer must be aware of how and when threads execute to control them.