Paul Kummer

Dr. Hanku Lee

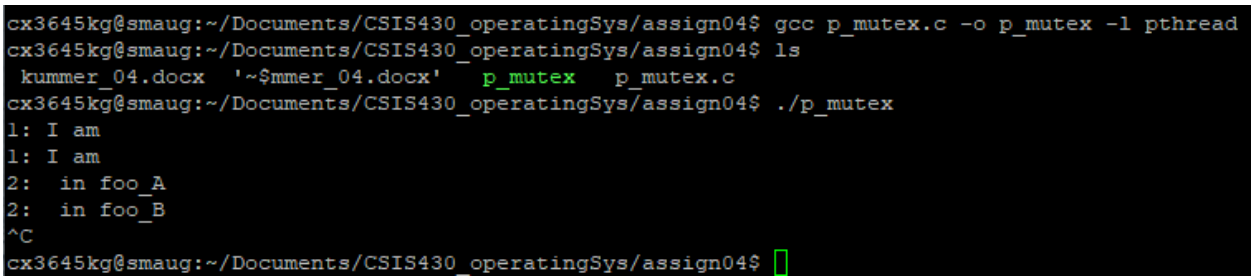
Operating Systems CSIS 430-01

10/01/2021

Mutex Exclusion Tests

p_mutex.c

With the original p_mutex.c code, the results from both threads are mixed in the output. This is because both threads call print_critical_region at very similar times and they are running concurrently. The first parameter from each function call is displayed almost immediately, which is “I am”. Then both thread’s function call will call the sleep function for one second before printing the caller’s second argument. The sleep function will pause the thread from executing any more statements until a specified amount of time has passed. Without the sleep call, both threads would print both parameters almost instantly. While all this is occurring, the main process will have already gone into an indefinite while loop because the condition is always true.



p_mutex_solution.c

C: > Users > pakum > Desktop > operatingSystems > assign04 > code > C p_mutex_solution.c

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void print_critical_region(char* a, char* b)
{
    pthread_mutex_lock(&mutex1);
    printf("1: %s\n", a);
    sleep(1);
    printf("2: %s\n", b);
    pthread_mutex_unlock(&mutex1);
}

// These two functions will run concurrently.
void* foo_A(void *ptr)
{
    print_critical_region("I am", " in foo_A");
}

void* foo_B(void *ptr)
{
    print_critical_region("I am", " in foo_B");
}

int main()
{
    pthread_t t1, t2;
    pthread_mutex_init(&mutex1, NULL);

    int iret1 = pthread_create(&t1, NULL, foo_A, NULL);
    int iret2 = pthread_create(&t2, NULL, foo_B, NULL);

    while(1){}

    return 0;
}
```

The solution code applies a mutually exclusive lock to parts of the code in print_critical_region. When this is done, only one thread can execute the code between the lock and unlock at a time. Until the first thread calls the mutex unlock, the second thread will wait to execute any further.

When the code is executed, the first thread will call foo_A, which in turn calls print_critical_region. Once print_critical_region is called by the first thread, a mutex lock is initiated on the code within that block. Meanwhile, the second thread has already been created and has already called print_critical_region as well. However, because the first thread started the mutex lock, the second thread waits for the first thread to finish executing the code between the lock and unlock. This will result in the ouput displaying the arguments from foo_A before foo_B.

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/assign04$ gcc p_mutex_solution.c -o p_mutex_solution -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/assign04$ ./p_mutex_solution
1: I am
2:  in foo_A
1: I am
2:  in foo_B
^C
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/assign04$
```

In the main process, the two threads are created, which each will respectively call their own functions that go into the critical region. While the threads are busy with function calls, the main

process will continue executing the code in the main block. The only other code left to execute after creating the threads is an indefinite while loop. The point of this loop is to keep the program running so the threads will have time to display their results. If the while loop is excluded, the program will complete before the threads can finish their function calls, as shown below.

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/assign04$ gcc p_mutex_solution.c -o p_mutex_solution -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/assign04$ ./p_mutex_solution
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/assign04$
```

Conclusion

This assignment was almost identical to the example p_mutex_lock from lab03. I didn't learn anything new that I didn't learn from that lab. I did get practice working with mutex locks and unlocks, which is good to practice. Also, this assignment help solidify the difference between threads and processes.