Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

09/16/2021

Lab 02: Process and Threads

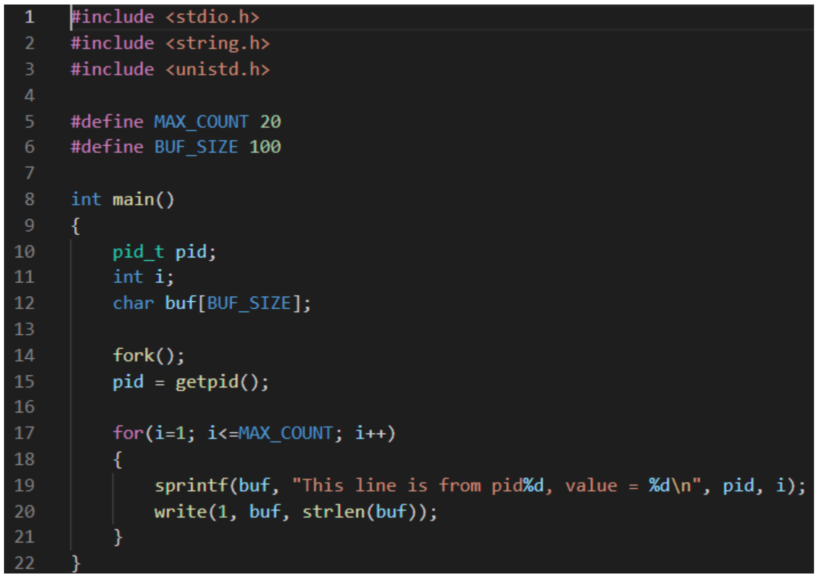
Purpose of Lab

The purpose of the lab is to show how processes and threads can be implemented in programming. Some of the properties of the strings and processes are shown, like how they can work independently. This allows parts of the code to finish at different times. However, the programmer can still manipulate threads and processes with wait, join, sleep, and program structuring. Furthermore, the dangers of global variables is demonstrated with threads.

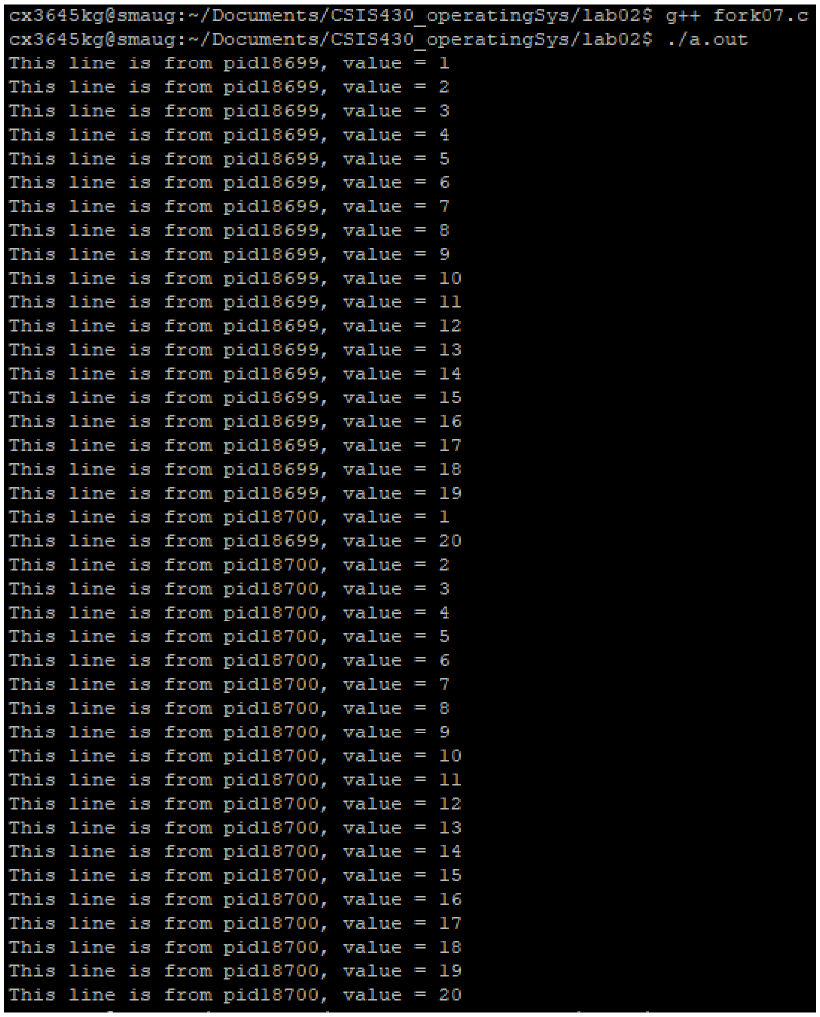
Example Code

fork07.c

In this code, a process is forked, duplicating the code. Then, because each process has its own address space, they create their own character buffer that stores the results of a for loop. The buffer is used so that the results to not get mixed together since each process works independently.



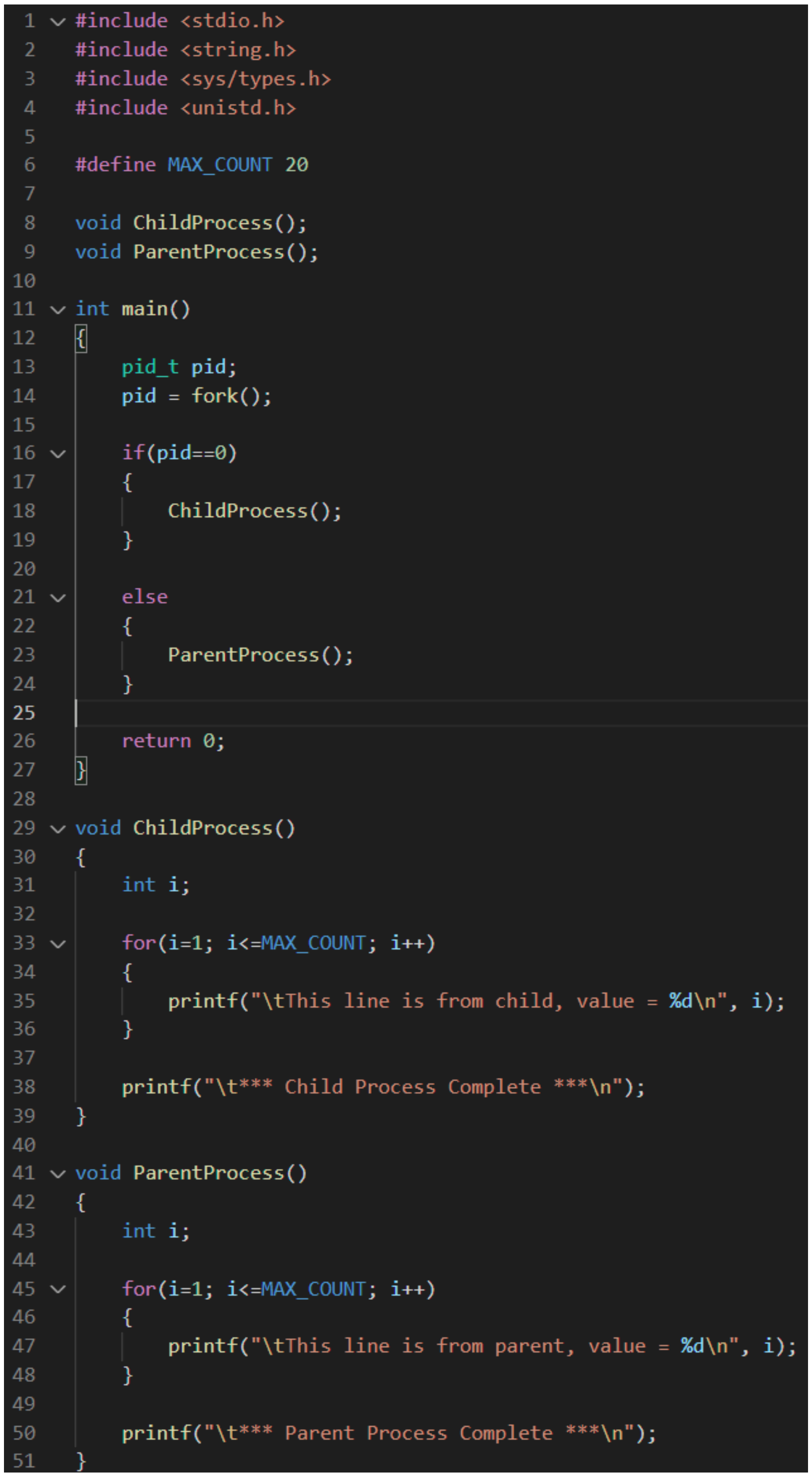
```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define MAX_COUNT 20
#define BUF_SIZE 100

int main()
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];

    fork();
    pid = getpid();

    for(i=1; i<=MAX_COUNT; i++)
    {
        sprintf(buf, "This line is from pid%d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```



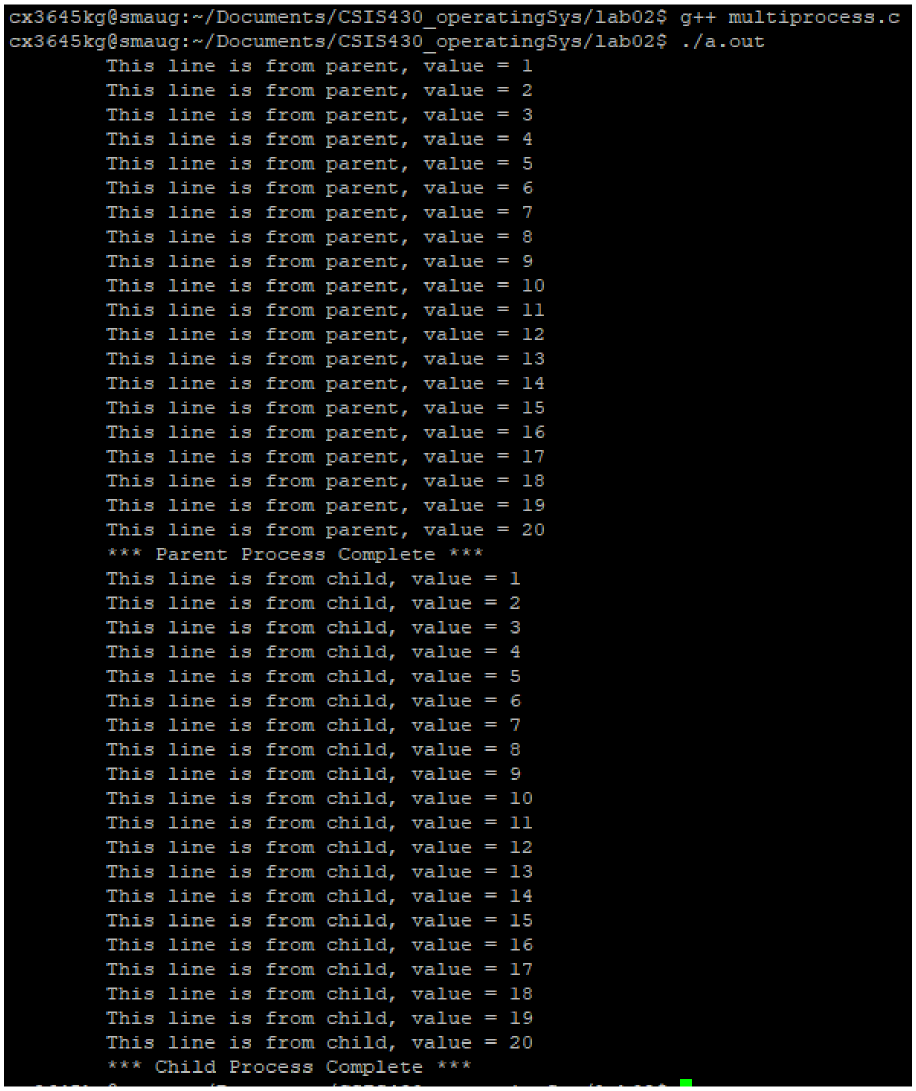
```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ fork07.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
This line is from pid18699, value = 1
This line is from pid18699, value = 2
This line is from pid18699, value = 3
This line is from pid18699, value = 4
This line is from pid18699, value = 5
This line is from pid18699, value = 6
This line is from pid18699, value = 7
This line is from pid18699, value = 8
This line is from pid18699, value = 9
This line is from pid18699, value = 10
This line is from pid18699, value = 11
This line is from pid18699, value = 12
This line is from pid18699, value = 13
This line is from pid18699, value = 14
This line is from pid18699, value = 15
This line is from pid18699, value = 16
This line is from pid18699, value = 17
This line is from pid18699, value = 18
This line is from pid18699, value = 19
This line is from pid18700, value = 1
This line is from pid18699, value = 20
This line is from pid18700, value = 2
This line is from pid18700, value = 3
This line is from pid18700, value = 4
This line is from pid18700, value = 5
This line is from pid18700, value = 6
This line is from pid18700, value = 7
This line is from pid18700, value = 8
This line is from pid18700, value = 9
This line is from pid18700, value = 10
This line is from pid18700, value = 11
This line is from pid18700, value = 12
This line is from pid18700, value = 13
This line is from pid18700, value = 14
This line is from pid18700, value = 15
This line is from pid18700, value = 16
This line is from pid18700, value = 17
This line is from pid18700, value = 18
This line is from pid18700, value = 19
This line is from pid18700, value = 20
```

multiprocess.c

This code shows how a process can be forked, giving it a new process id. If the process id is zero, then the process is a child. In this code, based on the process id, the parent and child process will execute different code.



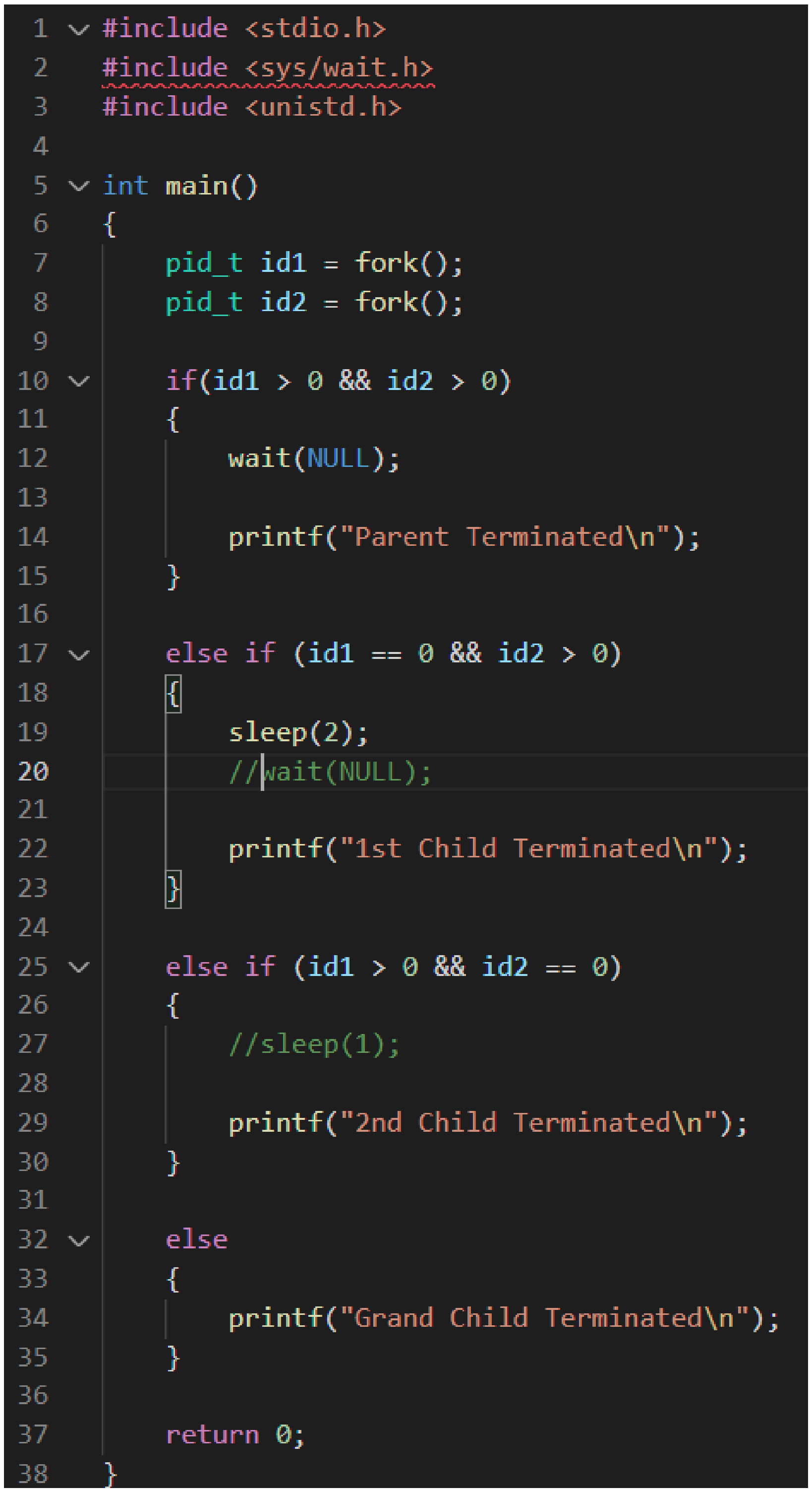
```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX_COUNT 20

void ChildProcess();
void ParentProcess();

int main()
{
    pid_t pid;
    pid = fork();

    if(pid==0)
    {
        ChildProcess();
    }

    else
    {
        ParentProcess();
    }

    return 0;
}

void ChildProcess()
{
    int i;

    for(i=1; i<=MAX_COUNT; i++)
    {
        printf("\tThis line is from child, value = %d\n", i);
    }

    printf("\t*** Child Process Complete ***\n");
}

void ParentProcess()
{
    int i;

    for(i=1; i<=MAX_COUNT; i++)
    {
        printf("\tThis line is from parent, value = %d\n", i);
    }

    printf("\t*** Parent Process Complete ***\n");
}
```



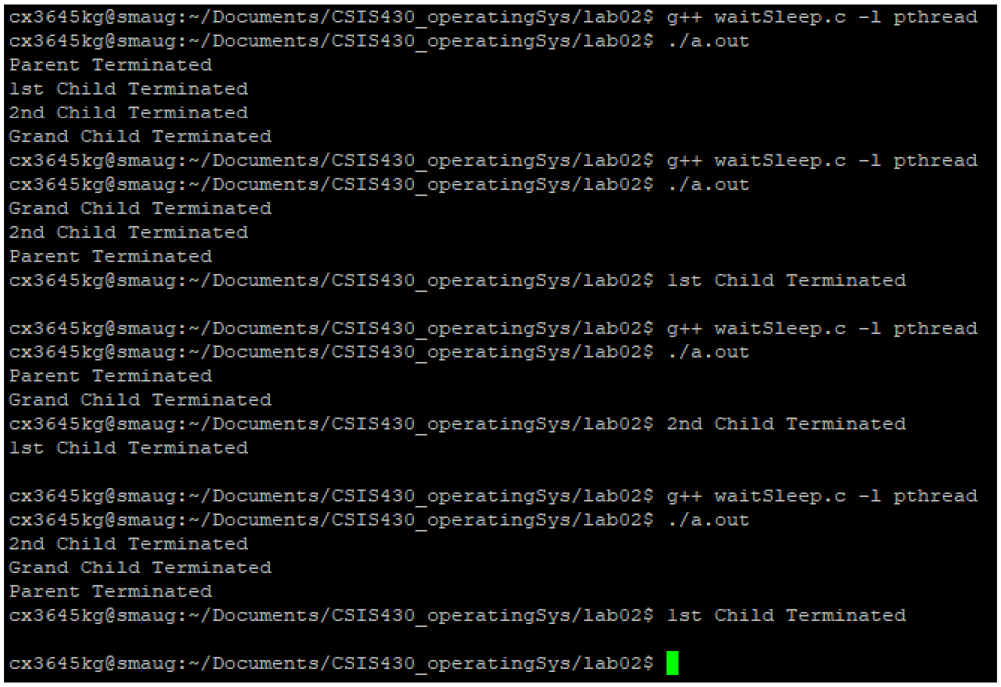
```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ multiprocess.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
        This line is from parent, value = 1
        This line is from parent, value = 2
        This line is from parent, value = 3
        This line is from parent, value = 4
        This line is from parent, value = 5
        This line is from parent, value = 6
        This line is from parent, value = 7
        This line is from parent, value = 8
        This line is from parent, value = 9
        This line is from parent, value = 10
        This line is from parent, value = 11
        This line is from parent, value = 12
        This line is from parent, value = 13
        This line is from parent, value = 14
        This line is from parent, value = 15
        This line is from parent, value = 16
        This line is from parent, value = 17
        This line is from parent, value = 18
        This line is from parent, value = 19
        This line is from parent, value = 20
        *** Parent Process Complete ***
        This line is from child, value = 1
        This line is from child, value = 2
        This line is from child, value = 3
        This line is from child, value = 4
        This line is from child, value = 5
        This line is from child, value = 6
        This line is from child, value = 7
        This line is from child, value = 8
        This line is from child, value = 9
        This line is from child, value = 10
        This line is from child, value = 11
        This line is from child, value = 12
        This line is from child, value = 13
        This line is from child, value = 14
        This line is from child, value = 15
        This line is from child, value = 16
        This line is from child, value = 17
        This line is from child, value = 18
        This line is from child, value = 19
        This line is from child, value = 20
        *** Child Process Complete ***
```

waitSleep.c

This code shows how wait and sleep perform differently. Sleep will pause a process for a specified duration wait will halt a process until its child is done. Wait can be used to get a return value from a child process.



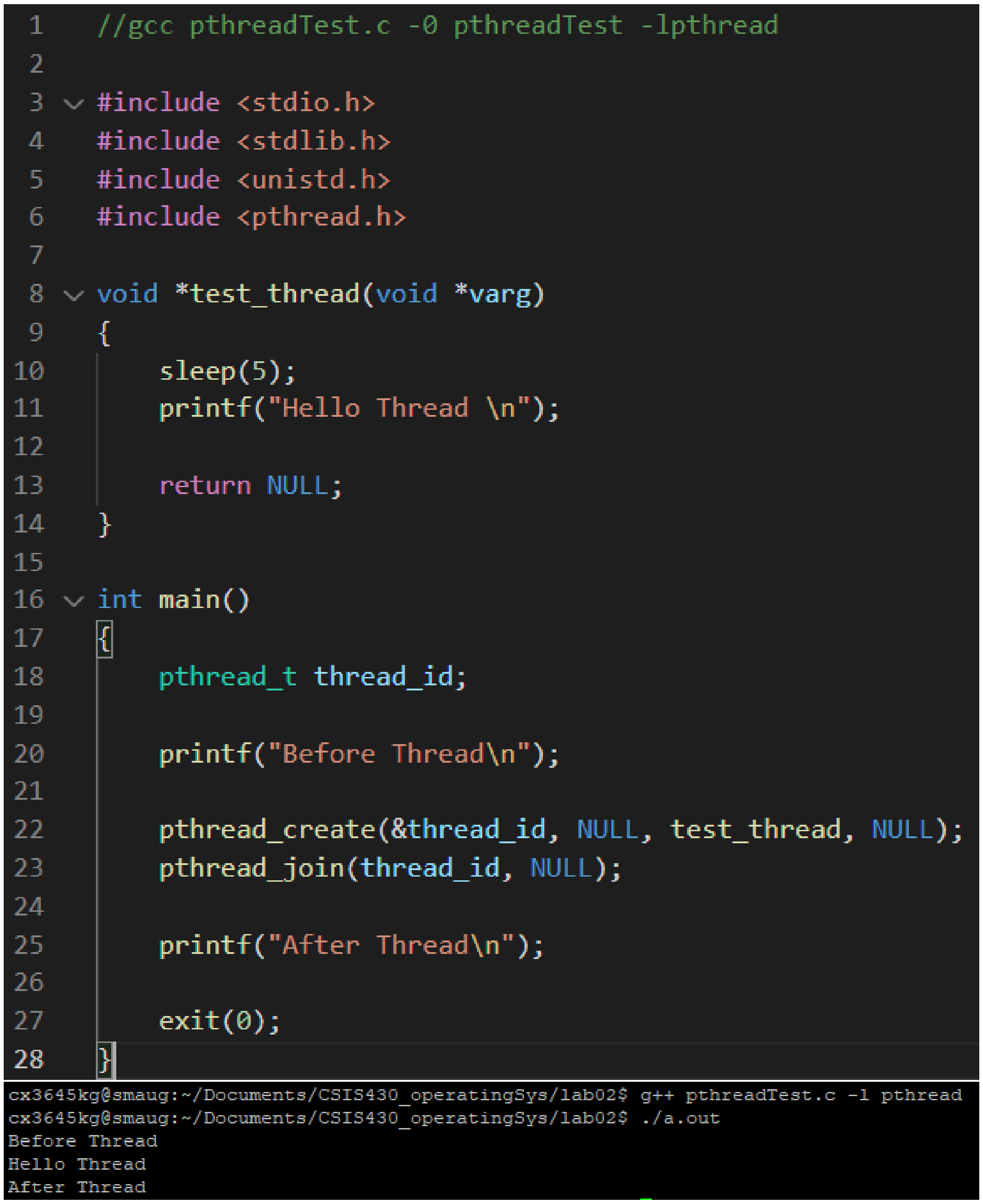
```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    pid_t id1 = fork();
    pid_t id2 = fork();

    if(id1 > 0 && id2 > 0)
    {
        wait(NULL);

        printf("Parent Terminated\n");
    }

    else if (id1 == 0 && id2 > 0)
    {
        sleep(2);
        //wait(NULL);

        printf("1st Child Terminated\n");
    }

    else if (id1 > 0 && id2 == 0)
    {
        //sleep(1);

        printf("2nd Child Terminated\n");
    }

    else
    {
        printf("Grand Child Terminated\n");
    }

    return 0;
}
```



```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ waitSleep.c -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
Parent Terminated
1st Child Terminated
2nd Child Terminated
Grand Child Terminated
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ waitSleep.c -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
Grand Child Terminated
2nd Child Terminated
Parent Terminated
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ 1st Child Terminated

cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ waitSleep.c -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
Parent Terminated
Grand Child Terminated
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ 2nd Child Terminated
1st Child Terminated

cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ waitSleep.c -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
2nd Child Terminated
Grand Child Terminated
Parent Terminated
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ 1st Child Terminated

cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ 
```

pthreadTest.c

This example shows how a thread can be created within a process. Additionally, it shows how a thread can wait for another thread to finish with the use of join.

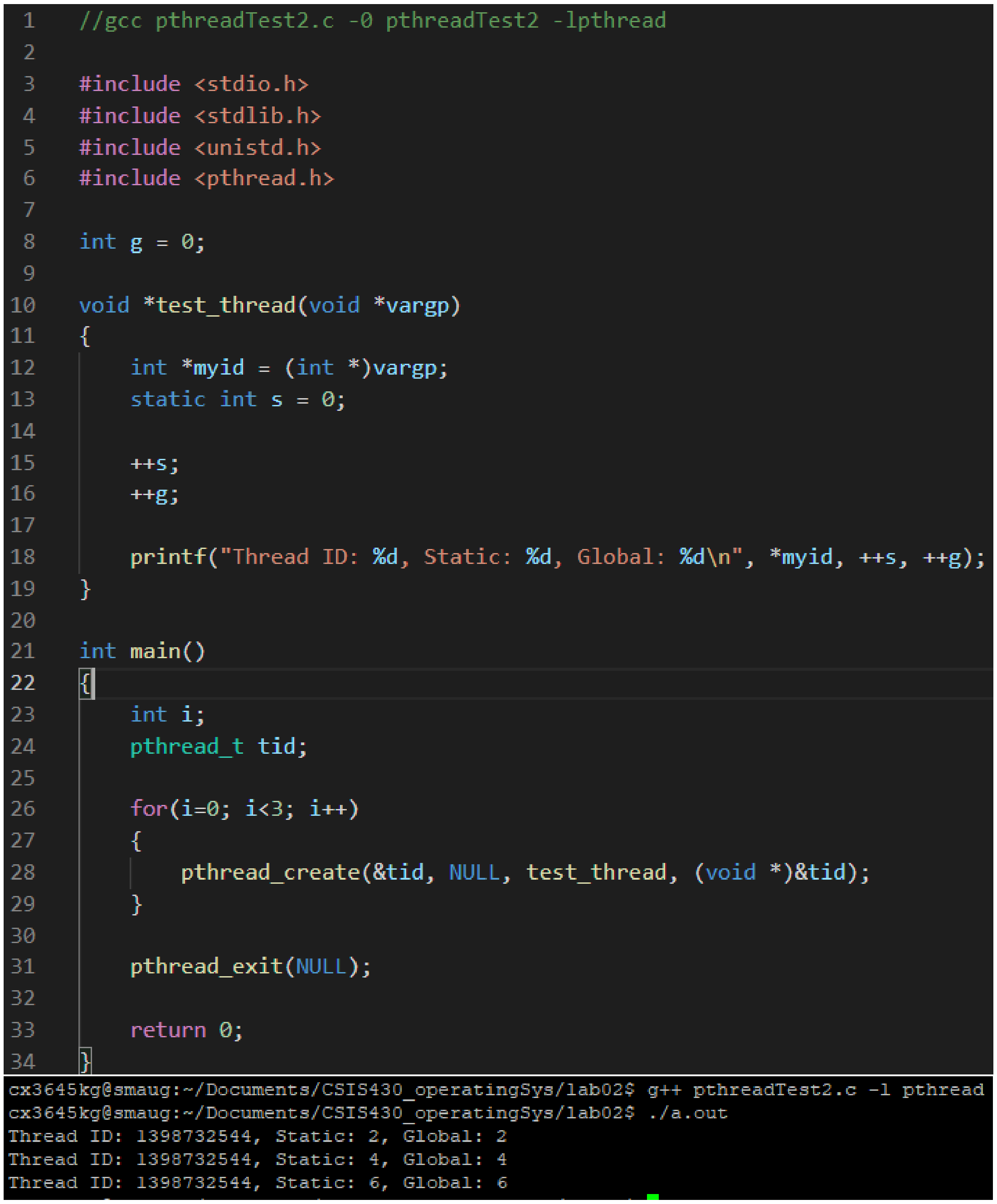


```c
//gcc pthreadTest.c -0 pthreadTest -lpthread

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

void *test_thread(void *varg)
{
    sleep(5);
    printf("Hello Thread \n");

    return NULL;
}

int main()
{
    pthread_t thread_id;

    printf("Before Thread\n");

    pthread_create(&thread_id, NULL, test_thread, NULL);
    pthread_join(thread_id, NULL);

    printf("After Thread\n");

    exit(0);
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ pthreadTest.c -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
Before Thread
Hello Thread
After Thread
```

pthreadTest2.c

In this example, it is shown how a static variable, which shouldn't change, does in fact change with the use of threads. Since the threads share a common address space, the static variable is incremented with every new thread. If the thread is to be confined to its own space it must not be static.



```c
//gcc pthreadTest2.c -0 pthreadTest2 -lpthread

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

int g = 0;

void *test_thread(void *vargp)
{
    int *myid = (int *)vargp;
    static int s = 0;

    ++s;
    ++g;

    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}

int main()
{
    int i;
    pthread_t tid;

    for(i=0; i<3; i++)
    {
        pthread_create(&tid, NULL, test_thread, (void *)&tid);
    }

    pthread_exit(NULL);

    return 0;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ g++ pthreadTest2.c -l pthread
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab02$ ./a.out
Thread ID: 1398732544, Static: 2, Global: 2
Thread ID: 1398732544, Static: 4, Global: 4
Thread ID: 1398732544, Static: 6, Global: 6
```

Conclusion

This lab was very helpful for working with threads and processes in practice. It allowed me to see how processes work independently in their own address space and threads work in a shared address space. I was also able to see how controlling when and how processes and threads execute can be critical for getting output in the correct order. To get information in the correct order, buffers may be used or methods like wait, sleep, or join can be called. One important takeaway I got from this lab is how cautious I should be when working with global variables and threads.