Andrew Ma

Lab 1

CPE 435

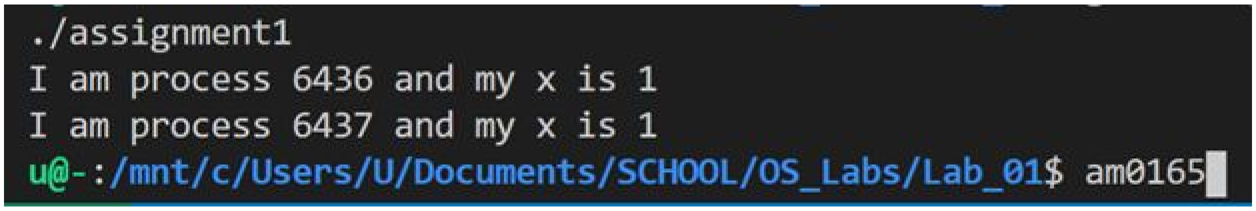
1/15/21

Theory

A process is a program in execution, and it is the basic active entity in an OS. Processes are executed sequentially, and they have these states: new, running, waiting, ready, and terminated. Each process stores information like process state, process number, program counter, registers, memory limits, and list of open files. The CPU can switch between processes.

Observations

Assignment 1

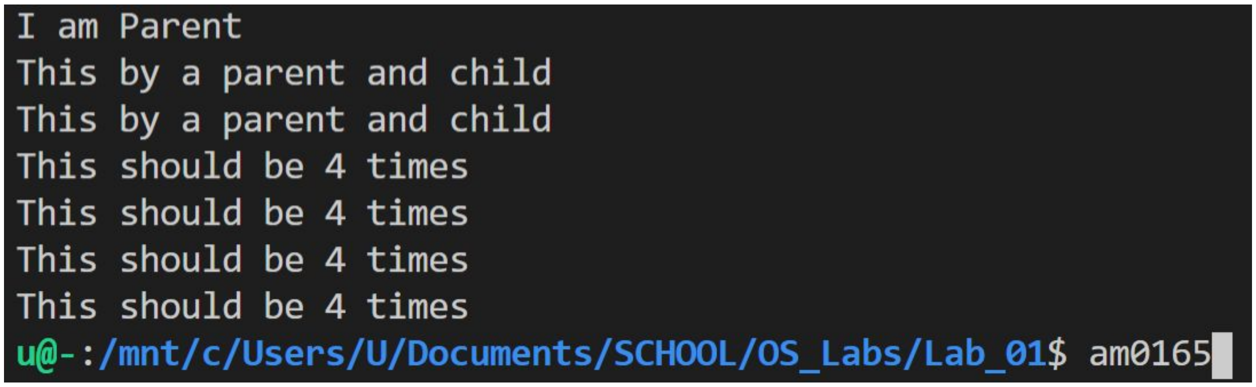


```
./assignment1
I am process 6436 and my x is 1
I am process 6437 and my x is 1
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ am0165
```

The x variable was set to 0. After fork() was run, there were now 2 processes (the parent process and a new child process). The child process is a copy of the parent process with its own copies of the memory and stack, so the child's x variable is also 0. The fork() call will return 0 to the child process and the child PID to the parent process, and the child process will begin executing after the fork() call..

The parent's x variable was incremented to 1. The parent's message was printed first with the parent's PID = 6346 and the x = 1. The child process's x variable was also incremented to 1. The child's message was printed with the child's PID = 6437. Since the child process was allocated after the parent, its PID is greater than the parent's PID.

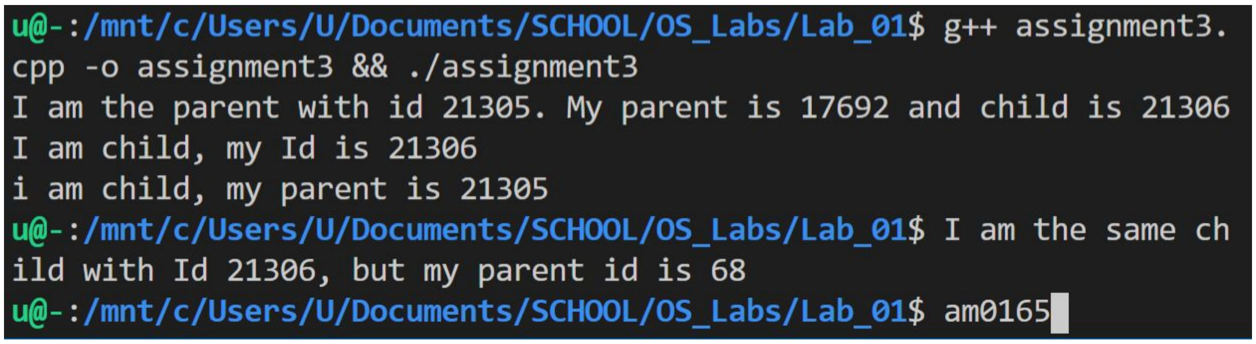
Assignment 2



```
I am Parent
This by a parent and child
This by a parent and child
This should be 4 times
This should be 4 times
This should be 4 times
This should be 4 times
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ am0165
```

4 processes are created. There were 2 fork() calls and each fork call keeps the parent process and creates a new child process, so the number of processes created = 2^(number of fork() calls) = 2^2 = 4. There was the original parent, the original's first child, the original's second child, and the original's first child's child (grandchild).

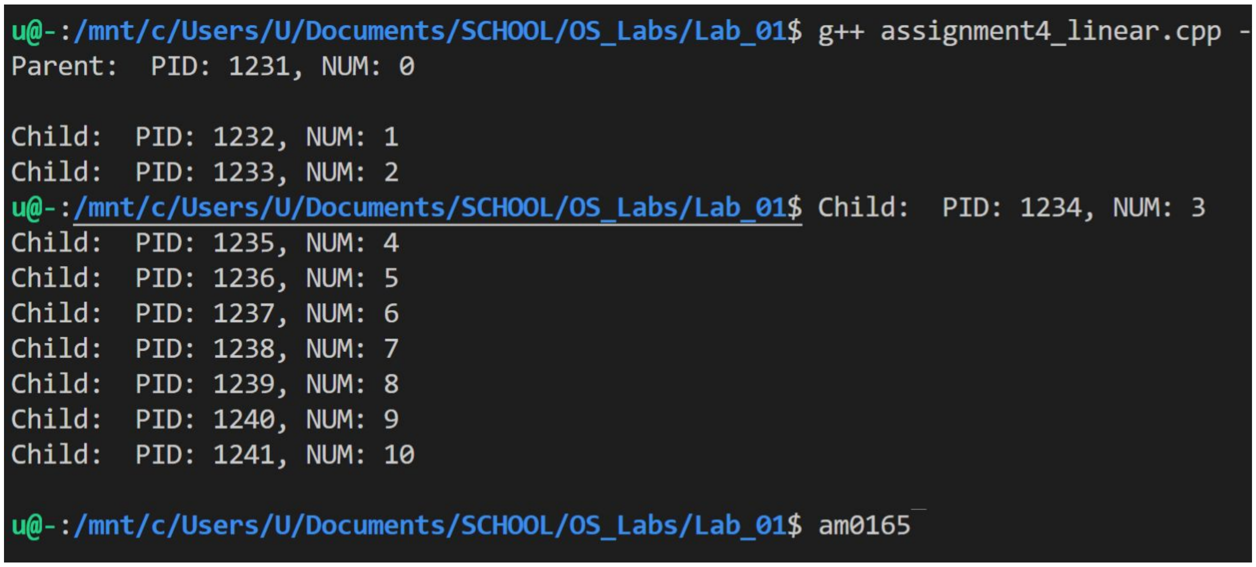
Assignment 3



```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ g++ assignment3.
cpp -o assignment3 && ./assignment3
I am the parent with id 21305. My parent is 17692 and child is 21306
I am child, my Id is 21306
i am child, my parent is 21305
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ I am the same ch
ild with Id 21306, but my parent id is 68
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ am0165
```

An orphan process is a child process whose parent process is dead, but it is still alive. The child process PID=21305 is an orphan process. We know it is an orphan process because its original parent PID=17692 but it changed to 68 (adopted by process dispatcher).

Assignment 4

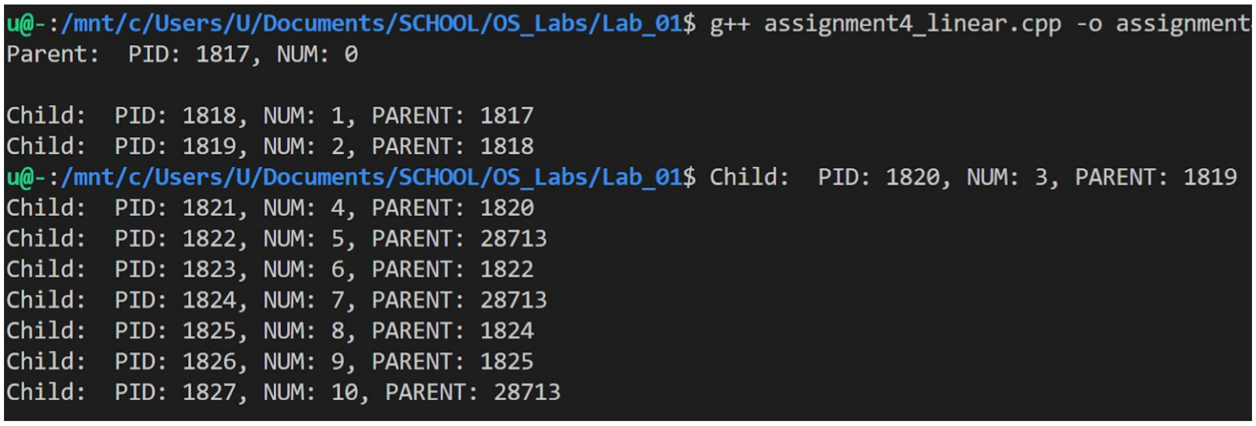


```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ g++ assignment4_linear.cpp -
Parent:  PID: 1231, NUM: 0

Child:  PID: 1232, NUM: 1
Child:  PID: 1233, NUM: 2
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ Child:  PID: 1234, NUM: 3
Child:  PID: 1235, NUM: 4
Child:  PID: 1236, NUM: 5
Child:  PID: 1237, NUM: 6
Child:  PID: 1238, NUM: 7
Child:  PID: 1239, NUM: 8
Child:  PID: 1240, NUM: 9
Child:  PID: 1241, NUM: 10

u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ am0165
```

10 children are forked. Because we need to keep the child process numbers in order, I used a For loop with 10 iterations and called fork() on the newest created child process (when PID == 0) each iteration. This creates more of a line structure instead of a tree because each process is only forked once.



```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ g++ assignment4_linear.cpp -o assignment
Parent:  PID: 1817, NUM: 0

Child:  PID: 1818, NUM: 1, PARENT: 1817
Child:  PID: 1819, NUM: 2, PARENT: 1818
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_01$ Child:  PID: 1820, NUM: 3, PARENT: 1819
Child:  PID: 1821, NUM: 4, PARENT: 1820
Child:  PID: 1822, NUM: 5, PARENT: 28713
Child:  PID: 1823, NUM: 6, PARENT: 1822
Child:  PID: 1824, NUM: 7, PARENT: 28713
Child:  PID: 1825, NUM: 8, PARENT: 1824
Child:  PID: 1826, NUM: 9, PARENT: 1825
Child:  PID: 1827, NUM: 10, PARENT: 28713
```

Conclusion

I learned how processes are forked to create child processes, and how each nested fork will double the number of processes (so 3 calls to fork() will create 2*2*2=8 processes). Forked child processes will create a copy of their parent's memory and stack at their time of creation, so variables are isolated from other processes. Orphan processes are child processes whose parent processes have terminated while they are still alive, and they will be adopted by the process dispatcher (and seen when the parent PID of the child changes numbers).

Source Code

```cpp
// Assignment 1
#include <stdio.h>
#include <iostream>
#include <unistd.h>

using namespace std;
int main() {
    int x = 0;
    fork();
    x++;
    cout << "I am process " << getpid() << " and my x is " << x << endl;
    return 0;

}
```

```cpp
//Assignment 2
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <unistd.h>

using namespace std;

int main() {
    printf("I am Parent\n");
    fork();
    printf("This by a parent and child\n");
    fork();
    printf("This should be 4 times\n");
    return 0;
}
```

```
// Assignment 3
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

    if (pid==0) {
        //child
        printf("I am child, my Id is %d\n", getpid());
        printf("i am child, my parent is %d\n", getppid());
        sleep(10);
        printf("I am the same child with Id %d, but my parent id is %d",
getpid(), getppid());
    }
    else {
        printf("I am the parent with id %d. My parent is %d and child is
%d\n", getpid(), getppid(), pid);
        sleep(5);
    }

}
```

```
// Assignment 4
// Forks 10 children, each which will print PID and number
#include <stdio.h>
#include <unistd.h>
```

```
int main() {

    printf("Parent:  PID: %d, NUM: %d\n\n", getpid(), 0);
    for (int i=1; i <= 10; i++) {
        pid_t PID = fork();
        if (PID == 0) {
            // child process
            printf("Child:  PID: %d, NUM: %d\n", getpid(), i);
        } else {
            break;
        }
    }
}
```