Andrew Ma

Lab 2

CPE 435

1/25/21

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. Unlike threads, memory in processes is not shared but is isolated with each process. An orphan process is where the parent exits before the child. This can happen when the child sleeps longer than the parent. As soon as the parent exits, the child is now an orphan, and it will be adopted by the process dispatcher. The orphan child's parent PID will be different from its original parent PID. A zombie process is where the child exits before the parent. This can happen when the parent sleeps longer than the child. The child process will now be a zombie process, and we can see a process is a zombie by typing 'ps -el' and seeing 'Z' in the second column.

The fork() function is used to create new child processes, which will run concurrently with the parent process starting from the instruction after the fork call. It is declared in uninstd.h. The child process will have the same state as the parent process up until the fork call, with the same registers, open files, and program counter. In the child process the return value of fork() is 0, and in the parent process the return value of fork() is the child process's PID. The different return value allows conditional and separate code for child and parent processes.

The exit(status) function is used by the child process upon completion to terminate itself. It is declared in stdlib.h. It does not return anything, but it takes an integer parameter that is the status value returned to the parent process. If the status value is 0, then it is a successful termination. Any value other than 0 is an unsuccessful termination.

The wait() function is used by the parent to wait for a child process to terminate. It returns the PID of the child process that terminated and caused the wait() to wake up, and it takes in an int pointer parameter that holds the status value of the child process if NULL or 0 is not passed in. It is declared in sys/wait.h.

Observations

Assignment 1

```
PARENT:  VAL: 5, PID: 11342
CHILD:  VAL: 2, PID: 11343
```

The val 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 val 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 val variable was incremented to 5. The parent's message was printed first with the parent's PID = 11342 and the val = 5. The child process's val variable was incremented to 2. The child's message was printed with the child's PID =11343. Since the child process was allocated after the parent, its PID is greater than the parent's PID. Since the val variable didn't add up to 7, we can conclude that val is isolated in separate processes.

Assignment 2

```
PARENT: parent's PID: 2511
CHILD1: 5, child1's PID: 2512
CHILD2: 15, child2's PID: 2513
CHILD1: 50
```

Fork is called to create a child1 process with PID 11364 from the parent process with PID 11363. The parent's PID is printed out, and it waits for child1 to finish. The first number used for the calculations is 10, and the second number is 5. In child1, it subtracts the two numbers and prints 5. Child1 is forked to create child2 with PID 11365. The code for child2 is nested in the child1 code to only fork from child1 and not the original parent. In child2, it adds the two numbers and prints the sum of the two numbers = 15. In order to make sure child2 runs before child1's multiplication step, child1 must wait() for child2 to complete and terminate. Child1 multiplies the two numbers = 50 and exits, returning control to the parent process which then exits successfully.

Assignment 3

```
5
Number is odd!
```

This is the output if the command line argument is an odd number.

```
10
PARENT: 3269
CHILD 1: 3270
CHILD 2: 3271
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_02/cmake-build-wsl_debug$ CHILD 3: 3272
CHILD 4: 3273
CHILD 5: 3274
CHILD 6: 3275
CHILD 7: 3276
CHILD 8: 3277
CHILD 9: 3278
```

This is the output if the command line argument is even. Here the n = 10, so it has one parent, and 9 children.

9 children are forked. I used a For loop with 9 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.

Assignment 4

1. **Orphan Process**

```
About to create an orphan process that is still sleeping when the parent ends
Original Parent 4215
Child 4216
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_02/cmake-build-wsl_debug$ After sleep Parent 2178
```

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 -   260 -      ?        00:00:10 init
1 S     0     8     1  0  80   0 -   314 -      ?        00:00:00 init
5 S     0    30     8  0  80   0 -  3045 -      ?        00:00:00 sshd
4 S     0    31    30  0  80   0 -  3475 -      ?        00:00:00 sshd
5 S     0    33     1  0  80   0 -   314 -      ?        00:00:00 init
1 S     0    34    33  0  80   0 -   316 -      ?        00:00:00 init
4 S  1000    35    34  0  80   0 -  1702 core_s pts/1    00:00:00 fsnotifier-wsl
5 S  1000   372    31  0  80   0 -  3475 -      ?        00:00:00 sshd
5 S     0  2177     1  0  80   0 -  3255 -      ?        00:00:00 init
1 S     0  2178  2177  0  80   0 -  3211 -      ?        00:00:01 init
4 S  1000  2179  2178  0  80   0 -  3401 core_s pts/2    00:00:00 bash
0 S  1000  3838  3179  0  80   0 -  3399 do_wai pts/3    00:00:00 bash
1 S  1000  4216  2178  0  80   0 -   623 hrtime pts/2    00:00:00 ex4
0 R  1000  4217  3838  0  80   0 -  2635 -      pts/3    00:00:00 ps
```


An orphan process is where the parent terminates before the child. In the process table, the parent process PID is not seen because it ended. The child process PID is in the process table and the child process is sleeping. Note that after waking up, the parent PID has changed. The orphan child process is adopted by the process dispatcher.

2. Zombie Process

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  1  80   0 -   298 -      ?        00:08:40 init
0 S  1000  1735 30756  0  80   0 -  3399 do_wai pts/5    00:00:00 bash
0 R  1000  2052  1735  0  80   0 -  2635 -      pts/5    00:00:00 ps
0 S  1000  2785 30756  0  80   0 -  1488 -      ?        00:00:00 sftp-server
5 S     0  6766     1  0  80   0 -   363 -      ?        00:00:00 init
1 S     0  6767  6766  0  80   0 -   363 -      ?        00:00:00 init
4 S  1000  6768  6767  0  80   0 -  1702 core_s pts/2    00:00:00 fsnotifier-wsl
5 S     0 11950     1  0  80   0 -   259 -      ?        00:00:00 init
1 S     0 11951 11950  0  80   0 -   259 -      ?        00:00:00 init
4 S  1000 11952 11951  0  80   0 -  2510 core_s pts/0    00:00:00 bash
0 S  1000 22287 30756  0  80   0 -  2982 core_s pts/4    00:00:00 gdbserver
0 t  1000 22291 22287  0  80   0 -  1470 ptrace pts/4    00:00:00 Lab_02
1 S     0 29275     1  0  80   0 -   363 -      ?        00:00:00 init
5 S     0 29295 29275  0  80   0 -  3045 -      ?        00:00:00 sshd
4 S     0 30686 29295  0  80   0 -  3475 -      ?        00:00:00 sshd
5 S  1000 30756 30686  0  80   0 -  3475 -      ?        00:00:03 sshd
0 S  1000 32635 30756  0  80   0 -  3399 core_s pts/3    00:00:00 bash
```



```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  1  80   0 -   363 -      ?        00:08:41 init
0 S  1000  1735 30756  0  80   0 -  3399 do_wai pts/5    00:00:00 bash
0 S  1000  2058 32635  0  80   0 -   623 hrtime pts/3    00:00:00 ex4
1 Z  1000  2059  2058  0  80   0 -     0 -      pts/3    00:00:00 ex4 <defunct>
0 R  1000  2061  1735  0  80   0 -  2635 -      pts/5    00:00:00 ps
0 S  1000  2785 30756  0  80   0 -  1488 -      ?        00:00:00 sftp-server
5 S     0  6766     1  0  80   0 -   363 -      ?        00:00:00 init
1 S     0  6767  6766  0  80   0 -   363 -      ?        00:00:00 init
4 S  1000  6768  6767  0  80   0 -  1702 core_s pts/2    00:00:00 fsnotifier-wsl
5 S     0 11950     1  0  80   0 -   259 -      ?        00:00:00 init
1 S     0 11951 11950  0  80   0 -   259 -      ?        00:00:00 init
4 S  1000 11952 11951  0  80   0 -  2510 core_s pts/0    00:00:00 bash
0 S  1000 22287 30756  0  80   0 -  2982 core_s pts/4    00:00:00 gdbserver
0 t  1000 22291 22287  0  80   0 -  1470 ptrace pts/4    00:00:00 Lab_02
1 S     0 29275     1  0  80   0 -   363 -      ?        00:00:00 init
5 S     0 29295 29275  0  80   0 -  3045 -      ?        00:00:00 sshd
4 S     0 30686 29295  0  80   0 -  3475 -      ?        00:00:00 sshd
5 S  1000 30756 30686  0  80   0 -  3475 -      ?        00:00:03 sshd
0 S  1000 32635 30756  0  80   0 -  3399 do_wai pts/3    00:00:01 bash
```

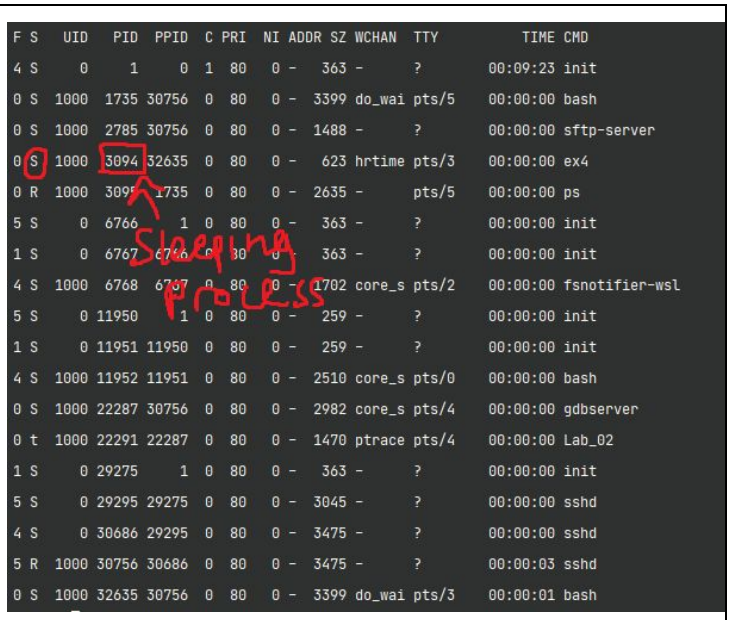
Zombie process is when the child terminates before the parent without the parent knowing. If the parent is sleeping and the child terminates, the parent has no way of knowing, and the child becomes a zombie. We can see a process is a zombie by typing ‘ps -el’ and seeing ‘Z’ in the second column.

3. Sleeping Beauty Process

```
About to Sleep process: 3094

Awake:  3094
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	1	0	1	80	0	-	341	-	?	00:09:23	init
0	S	1000	1735	30756	0	80	0	-	3399	core_s	pts/5	00:00:00	bash
0	S	1000	2785	30756	0	80	0	-	1488	-	?	00:00:00	sftp-server
0	R	1000	3086	32635	0	80	0	-	2635	-	pts/3	00:00:00	ps
5	S	0	6766	1	0	80	0	-	363	-	?	00:00:00	init
1	S	0	6767	6766	0	80	0	-	363	-	?	00:00:00	init
4	S	1000	6768	6767	0	80	0	-	1702	core_s	pts/2	00:00:00	fsnotifier-ws
5	S	0	11950	1	0	80	0	-	259	-	?	00:00:00	init
1	S	0	11951	11950	0	80	0	-	259	-	?	00:00:00	init
4	S	1000	11952	11951	0	80	0	-	2510	core_s	pts/0	00:00:00	bash
0	S	1000	22287	30756	0	80	0	-	2982	core_s	pts/4	00:00:00	gdbserver
0	t	1000	22291	22287	0	80	0	-	1470	ptrace	pts/4	00:00:00	Lab_02
1	S	0	29275	1	0	80	0	-	363	-	?	00:00:00	init
5	S	0	29295	29275	0	80	0	-	3045	-	?	00:00:00	sshd
4	S	0	30686	29295	0	80	0	-	3475	-	?	00:00:00	sshd
5	D	1000	30756	30686	0	80	0	-	3475	-	?	00:00:03	sshd
0	S	1000	32635	30756	0	80	0	-	3399	do_wai	pts/3	00:00:01	bash



A sleeping beauty process is when a process is sleeping and waiting for an internal event to awaken it. In the process table we can see the process has “S” in the second column.

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

```
#include <cstdio>
#include <unistd.h>

int main() {
   int val = 0;
   pid_t PID = fork();
   if (PID == 0) {
       //child process
       val += 2;
       printf("CHILD:  VAL: %d, PID: %d\n", val, getpid());

   } else {
       val += 5;
       printf("PARENT:  VAL: %d, PID: %d\n", val, getpid());

   }
}
// val is isolated between processes and ends up being 2 in the child and 5
in the parent.
```

```
#include <cstdio>
#include <cstdlib>
#include <sys/wait.h>
#include <unistd.h>


int subtract(int num1, int num2) {
   return num1 - num2;
}

int main() {
   int num1 = 10;
```

```
    int num2 = 5;

    //child1
    pid_t PID = fork();
    if (PID == 0) {
        //child1
        int subtractResult = subtract(num1, num2);
        printf("CHILD1: %d, child1's PID: %d\n", subtractResult, getpid());

        //child2
        PID = fork();
        if (PID == 0) {
            //child2
            printf("CHILD2: %d, child2's PID: %d\n", num1 + num2, getpid());
//add two numbers
            exit(0);
        } else {
            //child1
            wait(nullptr); // child1 waits for child2 to die
            printf("CHILD1: %d\n", num1 * num2); //multiply 2 numbers
            exit(0);                             //child1 terminate
        }
    } else {
        //parent
        printf("PARENT: parent's PID: %d\n", getpid()); // print parent
process's id
        wait(nullptr);                                          //wait for
child1 to die
        exit(0);                                        // resume and
terminate program
    }
    fflush(stdout);
}
```

```
#include <cstdio>
#include <cstdlib>
#include <unistd.h>
#include <string>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Need one argument that is number of child processes  %d.\n",
argc);
        exit(1);
    }

    std::string arg1_s = argv[1];
    int n = stoi(arg1_s);
    printf("%d\n", n);

    if (n % 2 != 0) {
        printf("Number is odd!\n");
        exit(1);
    }
```

```
    printf("PARENT: %d\n", getpid());

    for (int i = 0; i < n - 1; i++) {
        pid_t PID = fork();
        if (PID == 0) {
            printf("CHILD %d: %d\n", i+1, getpid());
            fflush(stdout);
        }
        else {
            break;
        }
    }
}
```

```
#include <cstdio>
#include <cstdlib>
#include <unistd.h>

int main() {
    // orphan process - parent ends before child
    // notice how the parent PID changes after the child sleeps
    // this is because as soon as the parent dies and the child becomes an
orphan it is adopted by the process dispatcher
    printf("About to create an orphan process that is still sleeping when the
parent ends\n");
    pid_t PID = fork();
    if (PID == 0) {
        printf("Child %d\n", getpid());
        sleep(20);
        printf("After sleep Parent %d\n", getppid());


        // zombie process - child ends before parent without parent knowing
        // when the parent is sleeping, the child ends and becomes a zombie
        // we can see a process is a zombie by typing 'ps -el' and seeing 'Z'
in the second column
        printf("About to create a child that ends when the parent is still
sleeping\n");
        if (fork() != 0) {
            // parent
            printf("Parent\n");
            sleep(10);
        } else {
            // child does nothing
        }



        // sleeping beauty process - process sleeps
        printf("About to Sleep process: %d\n", getpid());
        sleep(10);
        printf("Awake:  %d\n", getpid());
    } else {
        printf("Original Parent %d\n", getpid());
    }
```

```
}
```