Theory

In this lab, we explore basic process management. A process is simply a program that is being executed. It could be on one or multiple threads. There are many states to a process. The new state is where a process is created. The running state is where instructions are executed. The waiting state is where the process pauses until an event occurs. The ready state is where a process is waiting to be assigned to another process. The terminated state is where a process has finished. An orphan process is a child process whose parent has been terminated. It is then adopted by the process dispatcher. A zombie process occurs when a child is terminated but the parent is not notified.

The fork function creates a new process that is a copy of the current process. The two processes are the same except that the child runs in its own memory space and it has its own PID. The fork function returns the child's PID in the parent process. It returns 0 if it is running in the child process. It returns -1 if there is an error.

The exit function terminates a process. It returns the value of status which is an input parameter. It will flush all stdio streams as well. The variable passed into is not protected so it is not thread safe.

The wait function waits for a process to change states. A state change could be a child terminated or a child that has been stopped or resumed by a signal. The wait function return the PID of the terminated child.

Results and Observations

Exercise 1

The output for program 1 is below.

```
The value of val is 2. The PID is 5000
The value of val is 5. The PID is 4999
```

The child simply adds two to val and then completes. Then the parent adds five to val and completes.

Exercise 2

The output for program 2 is below.

```
The parent PID is 7535
I am child 1.
The result is 3 and the PID is 7536
I am child 2.
The result is 4 and the PID is 7537
I am child 1
The result is 12 and the PID is 7536
The parent has finished executing.
```

The parent starts by printing its PID. It creates a child 1 which calls the subtract function and then creates child 2 which calls the add function. Child 1 waits for child 2 to finish before executing before executing the multiply function. The parent waits for the child processes to finish before printing that it is done to the screen.

Exercise 3

The output for program 2 is below.

```
austinsbrown@DESKTOP-O0AMQ3N:/mnt/c/Users/austi/OneDrive/School/cpe434/lab2$ ./program3 4
The parent's PID is 179

The child's PID is 180

The child's PID is 181

The child's PID is 182
```

The program first checks to make sure that the input is even. If it is not, then the program prints an error message and exits. If you pass the error check, Then the program goes into a loop. The loop creates a child process and then exits.

Exercise 4

Below is an example of an orphan process. The parent has been terminated so this process has been adopted by the system. You can tell because it has a PID of 1.

```
1 S  1000  1357    10  0  80   0 -   623 hrtime pts/1    00:00:00 orphan
```

Below is an example of a sleeping beauty process. You can tell because it has an s as its status.

```
0 S  1000  1653   104  0  80   0 -   590 hrtime pts/1    00:00:00 sleeping
```

Below is an example of a zombie process. The child is terminated while the parent is asleep. This means that the parent isnt notified that the child has been terminated.

```
0 S  1000  2059   104  0  80   0 -   623 hrtime pts/1    00:00:00 zombie
1 Z  1000  2060  2059  0  80   0 -     0 -      pts/1    00:00:00 zombie <defunct>
```

Conclusion

In this lab, we gained experience with process management. We used to fork function to create new processes. We used the wait function to force parents to wait until their children are done, and we used the exit function to force a process to terminate. We also learned about orphan and zombie processes.

Appendix

```c
// Program 1

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

int main()
{
    pid_t pid;
    int val = 0;

    pid = fork();
    if(pid == 0)
    {
        val += 2;
        printf("The value of val is %d. The PID is %d\n", val, getpid
());
        return 0;
    }

    wait(NULL);
    val += 5;
    printf("The value of val is %d. The PID is %d\n", val, getpid());
    return 0;
}
```

```c
// Program 2

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int sub(int, int);
int add(int, int);
int mul(int, int);
```

```c
int main()
{
    pid_t pPID, cPID;
    int result;
    int status = 0;
    printf("The parent PID is %d\n", getpid());
    pPID = fork();

    if(pPID == 0)
    {
        result = sub(5,2);
        printf("I am child 1.\n");
        printf("The result is %d and the PID is %d\n", result, getpid
());
        cPID = fork();
        if(cPID == 0)
        {
            result = add(2,2);
            printf("I am child 2.\n");
            printf("The result is %d and the PID is %d\n", result, ge
tpid());
            exit(0);
        }
        while ((pPID = wait(&status)) > 0);
   // wait for the child to finish
        result = mul(2,6);
        printf("I am child 1\n");
        printf("The result is %d and the PID is %d\n", result, getpid
());
        exit(0);
    }
    while ((pPID = wait(&status)) > 0);                                 /
/ wait for the child to finish
    printf("The parent has finished executing.\n");
    return 0;
}


int sub(int arg1, int arg2)
```

```
{
    int result = arg1 - arg2;
    return result;
}

int add(int arg1, int arg2)
{
    int result = arg1 + arg2;
    return result;
}

int mul(int arg1, int arg2)
{
    int result = arg1 * arg2;
    return result;
}
```

```
// Program 3

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

int main(int argc, char **argv)
{
    if(argc % 2 != 0)
    {
        printf("The input must be an even number!\n");
        return 0;
    }

    int n = atoi(argv[1]);
    pid_t pPID, cPID;

    pPID = getpid();                                                    /
/ get the parents pid
    printf("The parent's PID is %d\n\n", getpid());
    for(int i=0; i<n-1; i++)
    {
```

```
        cPID = fork();
        if(pPID == getpid())
        {
            printf("The child's PID is %d\n\n", cPID);
        }

        else if(cPID == 0)
            exit(0);
    }
    return 0;
}
```

```
// Orphan

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main()
{
    int pid = fork();
    if(pid == 0)
    {
        printf("This is the child. My PID is %d\n", getpid());
        sleep(5);
        printf("This is the child. My PID is %d\n", getpid());
        exit(0);
    }

    printf("I am the parent. My PID is %d\n\n", getpid());
    return 0;
}
```

```
// Sleeping Beauty

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

```c
#include <stdlib.h>

int main()
{
    sleep(60);
    printf("I am awake\n");
    return 0;
}
```

```c
// Zombie

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main()
{
    if (fork () > 0)
    {
    printf("I am the parent.\n");
    sleep(50);
    }
    return 0;
}
```