Paul Kummer

Dr. Hanku Lee

Operating Systems CSIS 430-01

09/09/2021

Lab 01: Examining fork()

Purpose of Lab

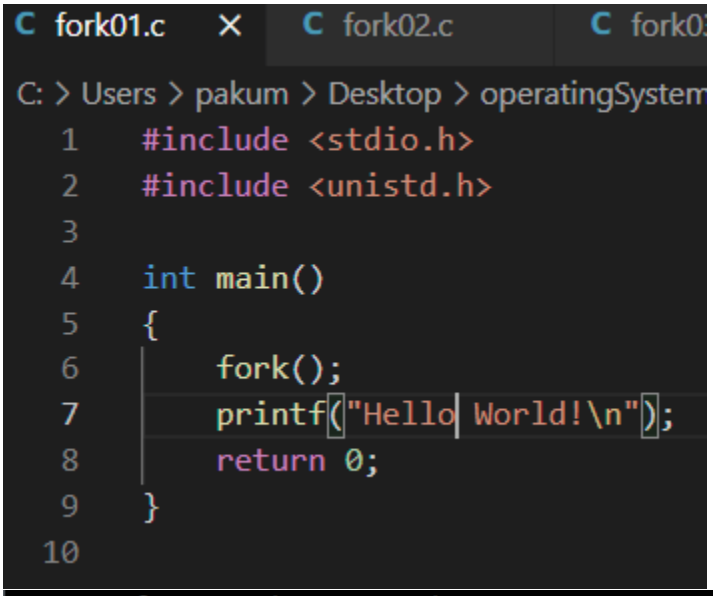
The main goal of lab 01 is explain and show how the fork function works in the C language. By examining the fork function in the lab, the way operating systems can duplicate code and create new processes will be shown.

When a new process is created with fork(), the computer will create a duplicate child process that runs concurrently with the parent process. The fork function will return a negative value if the child process was unsuccessful. If the return value is zero, then the process is a child process. If the return value is positive, then the process is the parent and is its pid.

Example Code

fork01.c

This example shows how the print function gets duplicated using fork().




```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ g++ fork01.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
Hello World!
Hello World!
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

fork02.c

Like the first example, the print function is duplicated. Additionally, as the fork function is called, other fork functions are also duplicated. This creates exponentially more processes and a hierarchy of parent and child processes.

fork01.c | fork02.c | fork03

> Users > pakum > Desktop > operatingSystems

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

int main()
{
    fork();
    fork();
    fork();
    printf("Hello World!\n");
    return 0;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ g++ fork02.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

fork03.c

In this example, the return values of fork are demonstrated. When the process identifier is zero, then the process is a child process, and “Hello from child” is displayed. When the process is positive, “hello from parent” is displayed. Since there is only one fork call, only two process will exist.

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

void fork_ex()
{
    int pid = fork();
    if (pid==0)
    {
        printf("Hello from child\n");
    }
    else
    {
        printf("hello from parent\n");
    }
}

int main()
{
    [redacted]
    fork_ex();
    [redacted]
    return 0;
}
```

```
g++ fork03.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
hello from parent
Hello from child
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

fork03.1.c

This example is the same as the previous example, however it shows the value returned by the fork call. When the process is a child, the pid will always be zero and it will be a positive number for all parent processes.

fork01.c | fork02.c | fork03.1.c | fork03.c

C: > Users > pakum > Desktop > operatingSystems > lab01 > C fork03.

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

void fork_ex()
{
    int pid = fork();
    if (pid==0)
    {
        printf("Hello from child %d\n", pid);
    }
    else
    {
        printf("hello from parent %d\n", pid);
    }
}

int main()
{
    [redacted]
    fork_ex();
    [redacted]
    return 0;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ g++ fork03.1.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
hello from parent 18935
Hello from child 0
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

fork04.c

Similar to the last example, what the output of a process is dependent on whether a process is a parent or child. However, in this example, if the process is a child, x will be incremented. Otherwise, x will be incremented. This shows that each process has its own address space for the variable x because the x value will always be the same for the child and the parent.

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

void fork_ex()
{
    int x=1;
    int pid = fork();

    if (pid==0)
    {
        printf("Child x = %d\n", ++x);
    }
    else
    {
        printf("Parent x = %d\n", --x);
    }
}

int main()
{
    fork_ex();

    return 0;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ g++ fork04.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
Parent x = 0
Child x = 2
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

fork05.c

In this example, multiple processes are running concurrently. When the results are outputted, the values could be zero, showing a child's output, or a positive number, indicating a parent. The order that they appear could be in any order, depending on which processes finish first. The parent process seem to finish faster than the childeren.

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

int main()
{
    int pid1 = fork();
    int pid2 = fork();

    printf("Hello World %d\n", pid1);
    printf("Hello World %d\n", pid2);

    return 0;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ g++ fork05.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
Hello World 19418
Hello World 19419
Hello World 19418
Hello World 0
Hello World 0
Hello World 19420
Hello World 0
Hello World 0
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

fork06.c

In the final example, the importance of writing to a buffer is shown when working with fork functions. This is because that when a process forks, the child process can run at the same time as the parent. They could both be writing to the shell, which could cause the output to get mixed together. However, if both process write to a buffer, the results of a process will be contigious.

```c
#include <stdio.h>
#include <unistd.h>
#include <string.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));
    }

    return 0;
}
```

```
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ g++ fork06.c
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$ ./a.out
this line is from pid 19576, value = 1
this line is from pid 19576, value = 2
this line is from pid 19576, value = 3
this line is from pid 19576, value = 4
this line is from pid 19576, value = 5
this line is from pid 19576, value = 6
this line is from pid 19576, value = 7
this line is from pid 19576, value = 8
this line is from pid 19576, value = 9
this line is from pid 19576, value = 10
this line is from pid 19576, value = 11
this line is from pid 19576, value = 12
this line is from pid 19576, value = 13
this line is from pid 19576, value = 14
this line is from pid 19576, value = 15
this line is from pid 19576, value = 16
this line is from pid 19576, value = 17
this line is from pid 19576, value = 18
this line is from pid 19576, value = 19
this line is from pid 19576, value = 20
this line is from pid 19577, value = 1
this line is from pid 19577, value = 2
this line is from pid 19577, value = 3
this line is from pid 19577, value = 4
this line is from pid 19577, value = 5
this line is from pid 19577, value = 6
this line is from pid 19577, value = 7
this line is from pid 19577, value = 8
this line is from pid 19577, value = 9
this line is from pid 19577, value = 10
this line is from pid 19577, value = 11
this line is from pid 19577, value = 12
this line is from pid 19577, value = 13
this line is from pid 19577, value = 14
this line is from pid 19577, value = 15
this line is from pid 19577, value = 16
this line is from pid 19577, value = 17
this line is from pid 19577, value = 18
this line is from pid 19577, value = 19
this line is from pid 19577, value = 20
cx3645kg@smaug:~/Documents/CSIS430_operatingSys/lab01$
```

Conclusion

This lab taught me how forking can be used to create new processes that duplicate the unprocessed code after the fork. Also, I learned some of the implications of using fork. When multiple forks are done consecutively, the number of processes can grow drastically. Also, because the processes run independently of each other. The order that the processes finish can vary, so if information needs to be output in a certain order, buffers must be used. Lastly, since each fork is independent, they create their own address space, copying any variables. This makes any transformations contained to each process. Forking is useful if code needs to branch of and perform a task while the other process still continues to run.