The Unix ”Fork” Command

Yun-Ping Du
yd2263@nyu.edu

***Abstract*—This document is a one page summary of what the Unix ”fork” command does and what activities we would expect to see in the operating system after calling this function from a program. Also, we would examine the various states that a process goes through and explain which states we would expect each process to be in and why.**

I. Overview

In multitasking operating systems, sometimes a process needs to create new processes (to run different programs). The *fork* system call is the primary method of process creation on Unix operating systems.

The process calling the fork command, which is referred to as the *parent* process, will first create a copy of itself, called the *child* process. The fork operation creates a separate *address space* for the child process; the child process has an exact copy of memory segments of the parent process.

The fork system call takes no arguments and return integer values, which has three possibilities. If the process creation fails, -1 is returned in the parent. Otherwise (successfully created), the process ID (PID) of the child is returned in the parent, and 0 is returned in the child.

After the fork operation, both processes will resume the exact instruction right after the fork system call, and they can determine their status by the return value of the call to act accordingly.

In practice, the child process may perform the *exec* system call to overlay itself with the other program. In this case, the child only performs few actions after the fork operation and the complete copy of the parent's memory segments is thus inefficient. In modern Unix variants that follow the virtual memory model, *copy-on-write* is implemented to alleviate the adverse effect.

The following example (in C programming language) demonstrates the fork system call:

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

    if (pid == -1) {
        printf("failed");
    }
    else if (pid == 0) {
        printf("child");
    }
    else {
        printf("parent");
    }
}
```

II. Process State Transition

In the following analysis we assume that our computer system has only one processor. First of all, the parent process is clearly in the **running** state to perform the fork system call. Subsequently the child process is in the **new** state during the creation and then enters the **ready** state and is waiting to be executed. After the creation, both processes are in the **ready** state and resume the same instruction right after the fork system call. However, since there is only one processor, we don't know whether the OS will first give control to the parent process or the child process.