Unix fork()

Overview

Unix fork() is a system call of the kernel that a process creates **a copy of itself.** Fork is the primary method process creation on **Unix** and its workalikes operating systems.

MAIN

In linux, the first process created is called *init*. All the other processes create to make the adequate system call is by fork(). The fork() create the child process and do not disturb the parent process work properly. Then these two processes will run at the same time, and follow the codes following the fork() system call. A child process uses same pc(program counter), same CPU registers, same open files which use in parent process.

Let's dive more into some details of how it works. One of the very basic fork() operation is as following, which clearly illustrated what fork() can do.

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

int main(int argc, char *argv[]){
  printf("hello world (pid:%d)\n",(int) getpid());//PID: Process Identifier
  int rc = fork();

  if(rc < 0){
  // fork failed
  fprintf(stderr, “fork failed\n”);
  exit(1);
  }
  else if (rc == 0){
  // child (new process)
  Printf("hello, I am child (pid:%d)\n", (int) getpid());
  else{
      //parent goes down this path(main)
      printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
      return 0;
}
```

Fork() takes no parameters. And the fork() system call creates a new process, which is a copy of the current process except for the returned value. When fork() returns a negative number, it failed to create a child process. When fork() returns zero, it goes to a newly created child process. If it is a positive value(the child PID), it goes to its parent process. If it failed, it will not print parent and child but failed only(though failed but still return a negative value). And by using return PID, it is easy to distinguish between parent process and child process.

Both processes will start their execution at the next statement following the fork() call. Although parent process and child process are running the same process, it does not mean that they are totally the same. OS allocate different data and states for these two processes, and the control flow of these processes can be different. In our program example, because the child process is created successfully, it will follow the instruction after the codes two times by both parent process and the child process. That is why the outcome of the code above will be "hello world (pid:141)" “hello, I am parent of 142 (pid: 141)” "hello, I am child (pid:142)". Here, the later two can be in different order because the parent process and child process are running concurrently. So we don't know whether the OS will first give control to the parent process or the child process.

There are many different systems calls. Using Waitpid, parent process can wait until the child process finish. Using pause(), you can suspend the caller until the next signal. Other interesting system calls are in book page 737.

Different address spaces

Not only can we control the different printf outcomes by the return value of fork(), we can also change the value of "same" variable. Unix make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces. Thus, any variable change in parent process will only effect the parent process. The child process will stay the same.

If a certain file was open in the parent before fork(), it will continue to be open in both parent and the child afterward. Changes made to the file will be visible to the other. These changes are also visibe to any unrelated process that opens the file.

Ps. **exec()** is the system call to replace the current process with a new program.

Drawbacks

Fork() is not perfect. Because Fork() creates an exact copy of the address space of the caller; with a large address space, making such a copy is *slow* and *data intensive*. Even worse, most of the address space is immediately over-written by a subsequent call to exec(), which overlays the calling process’s address space with that of the soon-to-be-exec’d program.

Improvement

By instead performing a copy-on-write fork(), the OS avoids much of the needless copying and thus retains the correct semantics while improving performance. In modern UNIX that follow the virtual memory model, the physical memory need not be actually copied. Instead, virtual memory pages in both processes may refer to the same pages of physical memory until one of them writes to such a page: then it is copies. Using MMU, it will only copy 4k at a time. Using copy-on-write model, fork() not only highly improve its efficiency, but also improve its security.