【OS】Day8

【Ch2】Process API

In this section, we discuss process creation in UNIX systems.

UNIX uses the following system calls to create a new process: `fork()` and `exec()`.

A third routine, `wait()`, can be used by a process wishing to wait for a process it has created to complete.

5.1 The fork() System Call

The `fork()` system call is used to create a new process.

Look at the following code:

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


int main(int argc, char *argv[]) {
  printf("Hello world (pid:%d)\n", (int)getpid());
  int rc = fork();
  if(rc < 0) {
    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 {
    printf("Hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
  }
  return 0;
}
```

It will give the following output:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Let us know understand what happened in `p1.c`

1. When it first started running, the process prints out a hello world message; included in that message is its process identifier, also known as a PID.

   The process has a PID of 29146; in UNIX systems, the PID is used to name the process if one wants to do something with the process, such as (for exasmple) stop it from running.

2. Now the process calss the `fork()` system call, which the OS provides as a way to create a new process.

   The odd part: the process that is created is an (almost) exact copy of the calling process. That means that to the OS, it now looks like there are two copies of the program `p1` running, and both are about to return from the `fork()` system call.

   The newly-created process(called the child, in contrast to the creating parent) doesn't start running at `main()`; rather it just comes into life as if it had called `fork()` itself

3. You might have noticed: *the child isn't an exact copy*. Specifically, although it now has its own copy of the address space(i.e. its own private memory), its own registers, itw own PC, and so forth, the value it returns to the caller of `fork()` is different.

   Specifically, while the parent receives the PID of the newly-created child, the child receives a return code of zero.

4. The output(of `p1.c`) is not deterministic.

   When the child process is created, there are now two active processes in the system that we care about: the parent and the child.

   Assuming we are running on a system with a single CPU, then either the child or the parent might run at that point. In our example(above), the parent did and thus

printed out its message first. In other cases, the opposite might happen:

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

The CPU scheduler determines which process runs at a given moment in time; because the scheduler is complex, we cannot usually make strong assumptions about what it will choose to do, and hence which process will run first.

This non-determinism, as it turns out, leads to some intersting problems, particularly in multi-threaded programs; hence, we'll see a lot more nondeterminism when we study concurrency.

5.2 The wait() System Call

Sometimes, it is quite useful for a parent to wait for a child process to finish what it has been doing.

This task is accomplished with the `wait()` system call(or its more complete sibling `waitpid()`).

See the following code for an example:

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

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    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)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

In this example, the parent process calls `wait()` to delay its execution until the child finishes executing.

When the child is done, `wait()` returns to the parent.

Adding a `wait()` call to the code above makes the output deterministic.

Here is the output:

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

With this code, we now know that the child will always print first.

It might simply run first, as before, and thus print before the parent.

However, if the parent does happen to run first, it will immediately call `wait()`; this sytem call won't return until the child has run and exited.

Thus, even when the parent runs first, it politely waits for the child to finish running, then `wait()` returns, and then the parent prints its message.