【OS】Day8(2)

【Ch2】Process API(2)

5.3 The exec() System Call

The `exec()` system call is useful when you want to run a program that is different from the calling program.

For example, calling fork() in p2.c is only useful if you want to keep running copies of the same program.

However, often you want to run a different program; `exec()` does just that.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.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) {
    fprintf(stderr,"Fork failed\n");
    exit(1);
  } else if(rc == 0) {
    printf("Hello, I am child (pid: %d)\n", (int)getpid());
    char *args[3];
    args[0] = strdup("wc");
    args[1] = strdup("prac.c");
    args[2] = NULL;
    execvp(args[0], args);
  } else {
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (wait: %d) (pid: %d)\n", rc,rc_wait,  (int)getpid());
  }
  return 0;
}
```

In this example, the child process calls `execvp()` in order to run the program `wc`, which is the word counting program. In fact, it runs `wc` on the source file `p3.c`, thus telling us how many lines, words, and bytes are found in the file:

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

What `exec()` does: given the name of an executable(e.g. `wc`), and some arguments(e.g. `p3.c`) it loads code (and static data) from that executable and overwrites it current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.

Thus, the OS simply runs that program, passing in any arguments as the `argv` of that process.

Thus, it does not create a new process; rather, it transforms the currently running program(formerly `p3`) into a different running program(`wc`). After the `exec()` in the child, it is almost as if `p3.c` never ran; a successful call to `exec()` never returns.

5.4 Why? Motivating The API

Why would we build such an odd interface to what should be the simple act of creating a new process?

The separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code after the call to `fork()` but before the call to `exec()`; this code can alter the enviornment of the about-to-be-run program, and thus enables a variesty of interesting features to be readily built.

The shell is just a user program. It shows you a prompt and then wait for you to type something into it.

You then type a command(i.e. the name of an executable program, plus any arguments) into it; in most cases, the shell then figures out where in the file system the executable resides, calls `fork()` to create a new child process to run the command, calls some variant of `exec()` to run the command, and then waits for the command to complete by calling `wait()`.

When the child completes, the shell returns from `wait()` and prints out a prompt again, ready for your next command.

Let's look at an example:

```
prompt> wc p3.c > newfile.txt
```

In the example above, the output of the program `wc` is redirected into the output file `newfile.txt` (the greater-than sign is how said redirection is indicated).

The way the shell accomplishes this taks is quite simple: when the child is created, before calling `exec()`, the shell closes standard output and opens the file newfile.txt. By doing so, any output from the soon-to-be-running program wc are sent to the file instead of the screen.

The following program does exactly this:

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

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4 . c)**

The reason this redirection works is due to an assumption about how the operating system mangaes file descriptors.

Specifically, UNIX systems start looking for free file descriptors at zero. In this case, STDOUT_FILENO will be the first availabe one and thus get assigned when `open()` is called.

Subsequent writes by the child process to the standard output file descriptor, for example by rountines such as `printf()`, will then be routed transparently to the newly-opened file instead of the screen.

Here is the output:

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```