《操作系统》
实验指导书

南昌大学软件学院

2022 年

实验 1： Linux 进程创建

1.1 实验概述

实验目的：掌握进程的概念，明确进程和程序的区别。

实验目标：创建进程、控制进程。

实验要求：通过验证实验内容掌握进程创建，解决实验设计问题，提交报告。

实验语言：c

实验环境：linux、gcc

1.2 实验内容

1.2.1 fork()系统调用

系统调用 fork()用于创建新进程，仔细看下面这段代码（p1.c），亲自键入并运行！

```c
#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) { // 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)
        printf("hello, I am parent of %d (pid:%d)\n",
            rc, (int) getpid());
    }

    return 0;
}
```

运行结果可能如下所示：

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

或

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

执行解析：

1. 当它刚开始运行时，进程输出一条 hello world 信息，以及自己的进程描述符（process identifier，PID）。该进程的 PID 是 29146。在 Linux 系统中，如果要操作某个进程（如终止进程)，就要通过 PID 来指明。
2. 进程调用了 fork()系统调用，这是操作系统提供的创建新进程的方法。新创建的进程几乎与调用进程完全一样，对操作系统来说，这时看起来有两个完全一样的 p1 程序在运行，并都从 fork()系统调用中返回。新创建的进程称为子进程（child），原来的进程称为父进程（parent）。子进程不会从 main()函数开始执行（因此 hello world 信息只输出了一次），而是直接从 fork()系统调用返回，就好像是它自己调用了 fork()。
3. 子进程并不是完全拷贝了父进程。具体来说，虽然它拥有自己的地址空间（即拥有自己的私有内存）、寄存器、程序计数器等，但是它从 fork()返回的值是不同的。父进程获得的返回值是新创建子进程的 PID，而子进程获得的返回值是 0。
4. 假设我们在单个 CPU 的系统上运行，在子进程被创建后，系统中的这两个活动进程（子进程和父进程）在执行输出不是确定的，如上图所示，可能父进程先运行并输出信息，也可能子进程可能先运行。CPU 调度程序(scheduler）决定了某个时刻哪个进程被执行。

1.2.2 wait()系统调用

有时候父进程需要等待子进程执行完毕，这项任务由 wait()系统调用（或者更完整的兄弟接口 waitpid()）来实现。

在 p1.c 的基础上进行修改[1]，如下 p2.c 所示：

[1] 在 linux 中，可以使用 shell 命令完成文件复制：
prompt>cp p1.c p2.c

```c
#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) {  // 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 wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }

    return 0;
}
```

在 p2.c 中，父进程调用 wait()，延迟自己的执行，直到子进程执行完毕。当子进程结束时，wait()才返回父进程。因此输出结果也变得确定了：

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

有可能子进程只是碰巧先运行，像以前一样，因此先于父进程输出结果。但是，如果父进程碰巧先运行，它会马上调用 wait()。该系统调用会在子进程运行结束后才返回。因此，即使父进程先运行，它也会等待子进程运行完毕，然后 wait()返回，接着父进程才输出自己的信息。

试试使用 waitpid()实现同样的功能！

1.2.3 exec()系统调用

使用 exec()这个系统调用可以让子进程执行与父进程不同的程序。

比如，子进程调用 execvp()[2]来运行字符计数程序 wc[3]。它针对源代码文件 p3.c

[2] execvp()是 exec()的变体，其它的还有 execl()、execle()、execlp()、execv()、execvpe()、

[3] Linux 系统中的 wc 命令的功能为统计指定文件中的字节数、字数、行数，并将统计结果显示输出。

运行 wc，从而告诉我们该文件有多少行、多少单词，以及多少字节。

执行结果如下所示：

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

如实现此功能，同样可以复制 c 文件为 p3.c，修改为：

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.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());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {  // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }

    return 0;
}
```

给定可执行程序的名称（如 wc）及需要的参数（如 p3.c）后，exec()会从可执行程序中加载代码和静态数据，并用它覆写自己的代码段（以及静态数据），堆、栈及其他内存空间也会被重新初始化。然后操作系统就执行该程序，将参数通过 argv 传递给该进程。因此，它并没有创建新进程，而是直接将当前运行的

程序（以前的 p3）替换为不同的运行程序（wc）。子进程成功执行 exec()之后不会再返回，所以后面的 printf 语句不会输出。

1.2.4 子进程输出重定向

Linux 的 shell 也是一个用户程序，它首先显示一个提示符[4]（比如 prompt），然后等待用户输入。你可以向它输入一个命令（一个可执行程序的名称及需要的参数），大多数情况下，shell 可以在文件系统中找到这个可执行程序，调用 fork()创建新进程，并调用 exec()的某个变体来执行这个可执行程序，调用 wait()等待该命令完成。子进程执行结束后，shell 从 wait()返回并再次输出一个提示符，等待用户输入下一条命令。

fork()和 exec()的分离，让 shell 可以方便地实现很多有用的功能。比如：

prompt> wc p3.c > newfile.txt

在上面的例子中，wc 的输出结果被重定向(redirect)到文件 newfile.txt 中(通过 newfile.txt 之前的大于号来指明重定向)。shell 实现结果重定向的方式也很简单，当完成子进程的创建后，shell 在调用 exec()之前先关闭了标准输出(standard output)，打开了文件 newfile.txt。这样，即将运行的程序 wc 的输出结果就被发送到该文件，而不是打印在屏幕上。

下面的代码 p4.c 展示了这样做的一个程序。重定向的工作原理，是基于对操作系统管理文件描述符方式的假设。具体来说，Linux 系统从 0 开始寻找可以使用的文件描述符。在这个例子中，STDOUT_FILENO 将成为第一个可用的文件描述符，因此在 open()被调用时，得到赋值。然后子进程向标准输出文件描述符的写入（例如通过 printf()这样的函数），都会被透明地转向新打开的文件，而不是屏幕。

下面是运行 p4.c 的结果：

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

首先，当运行 p4 程序后，好像什么也没有发生。shell 只是打印了命令提示符，等待用户的下一个命令。但事实并非如此，p4 确实调用了 fork 来创建新的子进程，之后调用 execvp()来执行 wc。屏幕上没有看到输出，是由于结果被重定向到文件 p4.output。其次，当用 cat[5]命令打印输出文件时，能看到运行 wc 的所有预期输出。

p4.c 的代码如下所示：

[4] 不同的 Linux 系统可能风格不同，用户也可设置自己的格式

[5] 也可以使用 more 命令来查看

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.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("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {   // parent goes down this path (main)
        int rc_wait = wait(NULL);
    }

    return 0;
}
```

UNIX 管道也是用类似的方式实现的，但用的是 pipe()系统调用。在这种情况下，一个进程的输出被链接到了一个内核管道(pipe)上(队列)，另一个进程的输入也被连接到了同一个管道上。因此，前一个进程的输出无缝地作为后一个进程的输入，许多命令何以用这种方式串联在一起，共同完成某项任务。比如通过将 grep、wc 命令用管道连接可以完成从一个文件中查找某个词，并统计其出现次数的功能：grep -o foo file | wc -l。

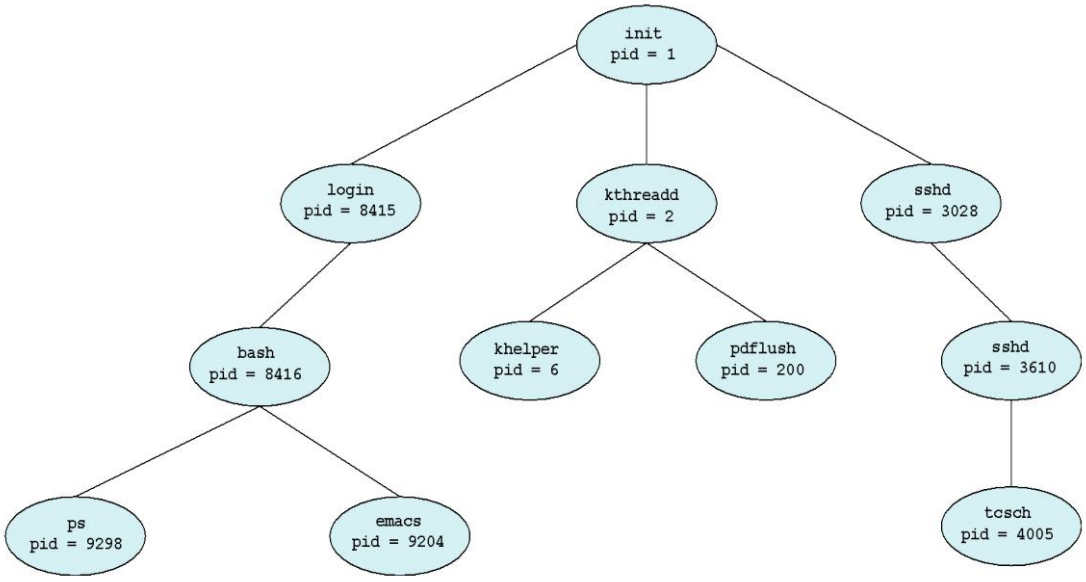
1.2.5 进程树

在 Linux 系统，我们可以通过 ps 命令得到一个进程列表。例如，命令：

ps -el

可以列出系统中的所有当前活动进程的完整信息。通过递归跟踪父进程一直到进程 init[6]，可以轻松构造类似下图所示的进程树。（此外，Linux 系统还提供了 pstree 命令，该命令将显示包含系统中所有进程的树。）

[6] 对于新版本的 Linux，init 已经被 systemd 替代，但作用类同。



现在我们生成 ABCD 进程，请根据代码及运行结果画出其进程树的结构。

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

int main(int argc, char *argv[]) {
    printf("hello, I am A1 (pid:%d)\n", (int) getpid());

    int rc_b, rc_d;
    rc_b = fork();
    if (rc_b < 0) {
        fprintf(stderr, "A1 fork failed\n");
        exit(1);
    } else if (rc_b == 0) {
        printf("hello, I am B1 (pid:%d), parent is (pid:%d)\n",
            (int) getpid(), (int) getppid());

        int rc = fork();
        if(rc < 0) {
            fprintf(stderr, "B fork failed\n");
            exit(1);
        } else if(rc == 0) {
            sleep(2);
            printf("hello, I am C (pid:%d), parent is (pid:%d)\n",
                (int) getpid(), (int) getppid());
        } else {
            wait(NULL);
            printf("hello, I am B2 (pid:%d), parent is (pid:%d)\n",
                (int) getpid(), (int) getppid());
        }
```

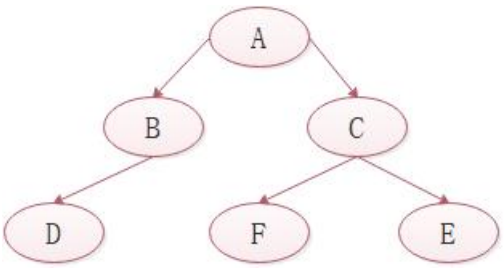
```
30      } else {
31          rc_d = fork();
32          if(rc_d < 0) {
33              fprintf(stderr, "A2 fork failed\n");
34              exit(1);
35          } else if(rc_d == 0) {
36              sleep(1);
37              printf("hello, I am D (pid:%d), parent is (pid:%d)\n",
38                  (int) getpid(), (int) getppid());
39          } else {
40              int st;
41              waitpid(rc_b, &st, 0);
42              waitpid(rc_d, &st, 0);
43              printf("hello, I am A2 (pid:%d)\n", (int) getpid());
44         }
45      }
46
47      return 0;
48  }
```

1.3 实验设计

1、编写一个调用 fork()的程序。在调用 fork()之前，让主进程访问一个变量（例如 x）并将其值设置为某个值（例如 100）。子进程中的变量有什么值？当子进程和父进程都改变 x 的值时，变量会发生什么？请编程给出结论。

2、使用 fork()编写一个程序。子进程应打印“hello”，父进程应打印“goodbye”。你应该尝试确保子进程始终先打印。你能否不在父进程调用 wait()或 waitpid()而做到这一点呢？

3、使用 fork()编写一个程序 A，创建下图所示的进程树，在每个进程中显示当前进程标识和父进程标识。



4、编写一个程序，创建两个子进程，并使用 pipe()系统调用，将一个子进程的标准输出连接到另一个子进程的标准输入。