操作系统实验一 进程管理 实验报告

一、实验目的

1．加深对进程概念的理解，明确进程和程序的区别；

2．进一步认识并发执行的实质；

3．分析进程争用资源的现象，学习解决进程互斥的方法；

4．了解 Linux 系统中进程通信的基本原理。

二、实验环境

Archlinux 2009.08

Linux Kernel 2.6.31-ARCH

zsh 4.3.10

gcc 4.4.1-1

gdb 7.0-2

vim 7.2.266-1

二、实验预备内容

1．阅读 Linux 的 sched.h 源码文件，加深对进程管理概念的理解

include/sched.h 中定义了调度相关的结构体和函数，比如进程描述符 task_struct，线程描述符 thread_info，以及 CPU 调度函数 schedule()，IO 调度函数 io_schedule()，还有一些给调度提供支持的结构和函数，如信号结构 signal_struct，计时函数 do_timer()，CPU 计时结构 task_cputime 等。

CPU 调度主要算法在 schedule()函数中，此函数定义在 kernel/sched.c 中。

2．阅读 Linux 的 fork()源码文件，分析进程的创建过程

kernel/fork.c 中定义了大部分进程创建相关的函数。

do_fork()函数是 fork()的主要过程，它会调用 copy_process()，根据参数 clone_flags 复制或者不复制父进程的资源，如：堆栈、页表、寄存器、打开文件的描述符等等。分析代码得 copy_proces()函数流程如下：

- 调用 dup_task_struct()为新进程创建一个内核栈、thread_info 结构和 task_struct，完全和父进程的相同。
- 检查当前用户进程数目是否超过限制。
- 重新设置进程描述符的一些域，将子进程和父进程区别开来。
- 设置子进程状态为 TASK_UNINTERRUPTIBLE 以保证它不被运行。
- copy_porcess()调用 copy_flags()以更新 task_struct 的 flags 成员。
- 调用 get_pid()为子进程申请一个有效的 PID。

- copy_porcess()根据 clone_flags 拷贝或者共享打开的文件、文件系统信息、信号处理函数、进程地址空间和命名空间等。
- 让父进程和子进程平分剩余的时间片，返回一个指向子进程的指针。
- 回到 do_fork()函数，新创建的子进程被唤醒执行。

三、实验内容

1.进程的创建

题目

编写一段程序，使用系统调用 fork() 创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符“a”，子进程分别显示字符“b”和“c”。试观察记录屏幕上的显示结果，并分析原因。

编译并运行十次

```
$ gcc fork.c -o fork
$ for ((i=0; i<10; i++)); do ./fork; printf ';'; done;
```

按 10 次 ctrl+c，运行结果

```
bca;bca;bca;bca;bca;bca;bca;bac;bca;cba;
```

输出结果为 8 次“bca”、一次“bac”以及一次“cba”。

原因分析

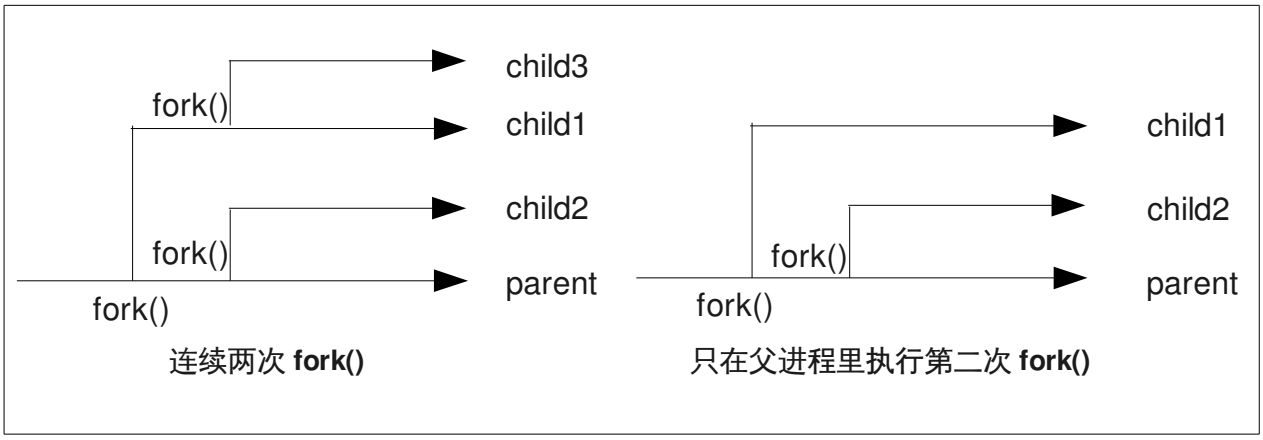
Linux 调度器尽量让 fork()后的子进程先得到调度。因为子进程并不是真正的复制了父进程资源，而是采用了 copy-on-write 技术，fork()产生的子进程往往会立即调用 exec()，所以先调度子进程可以避免父进程的写操作带来的不必要的页拷贝。

遇到的问题

连续调用两次 fork()得到的不是两个子进程，而是三个。因为在第一个 fork()之后，子进程和父进程同时执行第二个 fork()，将会再产生两个进程，这样一共就有 4 个进程在运行，不符合题意。

解决方法

在第一次 fork()后，判断 pid，只在父进程（pid != 0）的区域里执行 fork()这样就只有一个父进程和两个子进程。



代码结构如下(本次实验的所有代码基于下面的结构)

```
01   if ((pid1 = fork()) < 0) {
02       fprintf(stderr, "%s\n", strerror(errno));
03       return -1;
04   }
05   else if (pid1 == 0) {
06       /* action in child one */
07   }
08   else {
09       if ((pid2 = fork()) < 0) {
10           fprintf(stderr, "%s\n", strerror(errno));
11           return -1;
12       }
13
14       if (pid2 == 0) {
15           /* action in child two */
16       } else {
17           /* action in parent */
18       }
19   }
```

2.进程的控制

题目 A

修改已经编写的程序，将每个进程输出一个字符改为每个进程输出一句话，再观察程序执行时屏幕上出现的现象，并分析原因。

主要代码(基于第一题的代码框架)

```
01  /* action in child one */
02      printf("process 2.\nprocess 2.\n");
03  /* action in child two */
04      printf("process 3.\nprocess 3.\n");
05  /* action in parent */
06      printf("process 1.\nprocess 1.\n");
```

编译并运行十次

```
$ gcc fork2.c -o fork2
$ for ((i=0; i<10; i++)); do ./fork2; done;
```

运行结果

	process 3. process 3. process 1. process 1. process 2. process 2.		process 3. process 3. process 1. process 1. process 2. process 2.		process 3. process 3. process 1. process 2. process 1. process 2.
1		**2**		**3**	
4	process 3. process 3. process 1.	**5**	process 2. process 2. process 1.	**6**	process 2. process 2. process 3.

7	process 1. process 2. process 2. process 2. process 2. process 1. process 1. process 3. process 3.	**8**	process 1. process 3. process 3. process 3. process 3. process 1. process 1. process 2. process 2.	**9**	process 3. process 1. process 1. process 3. process 3. process 1. process 1. process 2. process 2.			
10	process 2. process 2. process 3. process 1. process 3. process 1.							

结果分析

与第一个程序相同，三个进程的执行顺序是随机的。第 3 组和第 10 组输出显示，一个 printf 函数可能被另一个进程的 printf 打断，并且是按行打断。

遇到的问题

原先每个进程都用一句“print(“prccess x\n”);”输出，输出中不会出现两个进程的输出混合的情况。

解决方法

问题的原因是 printf 函数是“按行缓冲”的，也就是说只有遇到‘\n’，printf 才会输出到 stdout。所以改成“printf(“prccess x\nprocess x\n”);”，在一个 printf 中输出两行，这样 printf 会分两次占用 stdin 输出两行，从而被令一个并发的 printf 抢占 stdin。

题目 B

如果在程序中使用系统调用 lockf() 来给每一个进程加锁，可以实现进程之间的互斥，观察并分析出现的现象。

主要代码(基于第一题的代码框架)

```
01  /* action in child one */
02      lockf(1, F_LOCK, 0);
03      printf("process 2.\nprocess 2.\n");
04      lockf(1, F_ULOCK, 0);
05  /* action in child two */
06      lockf(1, F_LOCK, 0);
07      printf("process 3.\nprocess 3.\n");
08      lockf(1, F_ULOCK, 0);
09  /* action in parent */
10      lockf(1, F_LOCK, 0);
11      printf("process 1.\nprocess 1.\n");
12      lockf(1, F_ULOCK, 0);
```

编译并运行十次

```
$ gcc lockf.c -o lockf
$ for ((i=0; i<10; i++)); do ./lockf; done;
```

运行结果

1	process 3. process 3. process 1. process 1. process 2. process 2.	**2**	process 3. process 3. process 1. process 1. process 2. process 2.	**3**	process 3. process 3. process 1. process 1. process 2. process 2.
4	process 1. process 1. process 2. process 2. process 3. process 3.	**5**	process 2. process 2. process 1. process 1. process 3. process 3.	**6**	process 2. process 2. process 3. process 3. process 1. process 1.
7	process 2. process 2. process 1. process 1. process 3. process 3.	**8**	process 3. process 3. process 1. process 1. process 2. process 2.	**9**	process 2. process 2. process 1. process 1. process 3. process 3.
10	process 3. process 3. process 2. process 2. process 1. process 1.				

结果分析

与没有加 lockf 的程序比较，输出里没有出现两个进程的输出混合在一起的情况，说明通过 lockf 给 stdout 加锁，实现了三个进程对 stdout 的互斥使用。

3.进程间信号通信

a）编写一段程序，使其实现进程的软中断通信。
要求：使用系统调用 fork() 创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上来的中断信号（即按 DEL 键）；当捕捉到中断信号后，父进程用系统调用 kill()向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

```
Child Process 1 is killed by Parent!
Child Process 2 is killed by Parent!
```

父进程等待两个子进程终止后，输出如下的信息后终止：

```
Parent Process is killed!
```

主要代码(基于第一题的代码框架)

```
01 /* global */
02 int lock = 1;
03 void unlock(){ lock = 0; } // release the lock
04 void busy_wait() { while (lock); }
05 /* action in child one */
06     signal(SIGINT, SIG_IGN);  // Ignore SIGINT signal
07     signal(SIGUSR1, unlock);
08     busy_wait(); // wait until signal SIGUSR1 captured.
09     lockf(1, F_LOCK, 0);
10     printf("Child process 1 killed by parrent process.\n");
11     lockf(1, F_ULOCK, 0);
12     return 0;
13 /* action in child two */
14     signal(SIGINT, SIG_IGN);  // Ignore SIGINT signal
15     signal(SIGUSR1, unlock);
16     busy_wait(); // wait until signal SIGUSR1 captured.
17     lockf(1, F_LOCK, 0);
18     printf("Child process 1 killed by parrent process.\n");
19     lockf(1, F_ULOCK, 0);
20     return 0;
21 /* action in parent */
22     signal(SIGINT, unlock);
23     busy_wait(); // block until SIG_INT come
24     kill(pid1, SIGUSR1);
25     kill(pid2, SIGUSR2);
26     wait(pid1);
27     wait(pid2); // wait two child process return
28     lockf(1, F_LOCK, 0);
29     printf("Parrent process exit.\n");
30     lockf(1, F_ULOCK, 0);
```

编译并运行十次

```
$ gcc signal.c -o signal
$ for ((i=0; i<10; i++)); do ./signal; done;
```

按 **10** 次 **ctrl+c**，运行结果

1	Child process 2 killed by parrent process. Child process 1 killed by parrent process. Parrent process exit.
2	Child process 2 killed by parrent process. Child process 1 killed by parrent process. Parrent process exit.
3	Child process 1 killed by parrent process. Child process 2 killed by parrent process. Parrent process exit.
4	Child process 1 killed by parrent process. Child process 2 killed by parrent process. Parrent process exit.
5	Child process 2 killed by parrent process. Child process 1 killed by parrent process. Parrent process exit.

6	Child process 1 killed by parrent process. Child process 2 killed by parrent process. Parrent process exit.
7	Child process 2 killed by parrent process. Child process 1 killed by parrent process. Parrent process exit.
8	Child process 1 killed by parrent process. Child process 2 killed by parrent process. Parrent process exit.
9	Child process 1 killed by parrent process. Child process 2 killed by parrent process. Parrent process exit.
10	Child process 2 killed by parrent process. Child process 1 killed by parrent process. Parrent process exit.

结果分析

其中有 5 次 child1 先输出，另 5 次 child2 先输出，可见 signal 发出后，哪个子进程先接收到信号是不定的，其取决于内核 CPU 调度器先调度到哪个进程。

遇到的问题

1. signal()函数是异步函数，执行注册后，程序并不等待信号的到来，而是继续执行，所以需要在执行 signal 后，阻塞进程，直到接受信号后，恢复进程的执行。
2. 按下 Ctrl+C 以后，只有父进程输出了，子进程没有输出。

解决方法

1. 执行 signal()函数后，用一个 while 循环阻塞进程，用一个全局变量做循环变量，初始化为 1；当父进程信号到来后，在信号处理函数中将循环变量设置成 0，这样进程会跳出循环继续执行。
2. 因为在终端下按 Ctrl+C 产生的 SIG_INT 信号会发给所有从终端执行的进程，包括父进程和两个子进程，在子进程接受到父进程的信号之前，已经被 SIG_INT 的默认处理函数结束，所以子进程在输出之前就已经结束。解决方法是在两个子进程里添加 signal(SIGINT, SIG_IGN)，忽略终端发出的 SIGINT 信号。

题目 B

在上面的程序中增加语句 signal(SIGINT, SIG_IGN) 和 signal(SIGQUIT, SIG_IGN)，观察执行结果，并分析原因。

执行结果

在父进程中增加 signal(SIGINT, SIG_IGN) 和 signal(SIGQUIT, SIG_IGN)，进程将忽略 SIGINT 和 SIGQUIT 信号，也就是键盘 ctrl+c 和 ctrl+\，进程将无法正常退出，只能用 kill 将其杀死。

4.进程的管道通信

题目

编制一段程序，实现进程的管道通信。
使用系统调用 pipe() 建立一条管道线；两个子进程 P1 和 P2 分别向管道各写一句话：

```
Child 1 is sending a message!
Child 2 is sending a message!
```

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。
要求父进程先接收子进程 P1 发来的消息，然后再接收子进程 P2 发来的消息。

主要代码(基于第一题的代码框架)

```
01  /* action in child one */
02      close(fd1[0]);
03      write(fd1[1], "Child 2 is sending a message!\n", 31);
04  /* action in child two */
05      close(fd2[0]);
06      write(fd2[1], "Child 2 is sending a message!\n", 31);
07  /* action in parent */
08      cnt = read(fd1[0], msg, MAX);
09      write(1, msg, cnt);
10      cnt = read(fd2[0], msg, MAX);
11      write(1, msg, cnt);
```

编译并运行

```
$ gcc pipe.c -o pipe
$ ./pipe
```

运行结果

```
Child 1 is sending a message!
Child 2 is sending a message!
```

结果分析

read 管道是阻塞的，直到有数据写入管道，程序才会继续执行。所以 read 的在程序中的先后，就能决定先接受哪个管道来的消息，而不需要其他同步手段。

四、实验总结

1. fork()函数用法

```
#include <unistd.h>
pid_t fork(void);
```

- fork()函数执行一次返回两次，在父进程中返回子进程的 PID，在子进程里返回 0；
- fork()函数正常返回后，子进程和父进程继续执行 fork()后面的指令；
- fork()返回后，子进程和父进程的执行顺序是不一定的，取决于调度算法。

2. signal()函数用法

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

signal 函数用于设定一个特定消息的处理方法。

参数说明：signo 为 Unix 信号，比如 SIGINT，SIGSEGV 等；func 的值是常

SIG_IGN、SIG_DFL 或接到此信号要调用的函数的地址。如果指定 SIG_IGN，则向内核表示忽略此信号。如果指定 SIG_DFL，则表示接到信号后动作时系统默认动作。当指定函数地址时，则在信号发生时，调用该函数（信号处理函数）。SIG_KILL 和 SIG_STOP 两个信号不能被忽略。

3. kill()函数用法

```
#include <signal.h>
int kill(pid_t pid, int signo);
```

kill 的 pid 参数有四种情况：

pid > 0	将该信号发送给进程 ID 为 pid 的进程
pid == 0	将该信号发送给与发送进程属于同一进程组的所有进程，而且发送进程具有向这些进程发送信号的权限。
pid < 0	将该信号发送给其进程组 ID 等于 pid 的绝对值，而且发送进程具有向其发送信号的权限。
pid == -1	将该信号发送给发送进程有权限向它们发送信号的系统上的所有进程

4. lockf()函数用法

```
#include <sys/file.h>
int lockf(int fd, int cmd, off_t len);
```

lockf 用于锁定或打开锁定一个共享文件.返回 1 表示调用 lockf 成功.

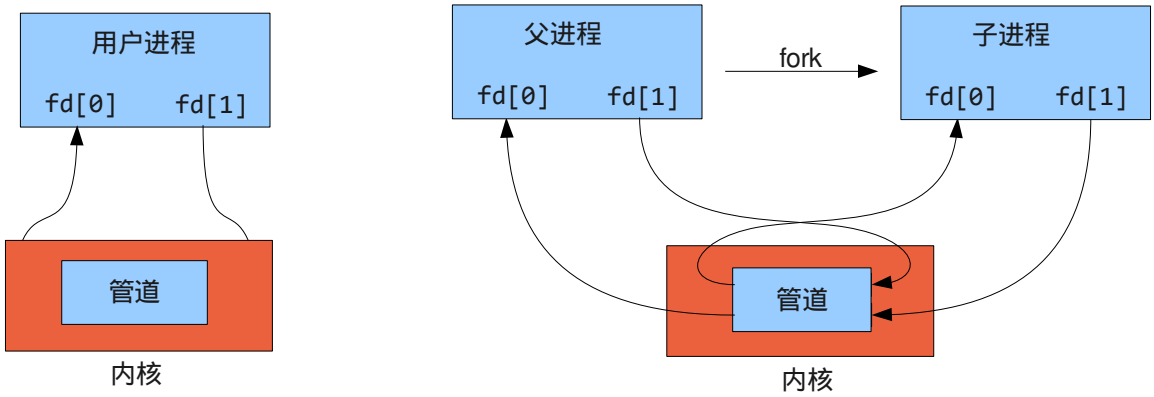
参数说明：fd 为目标文件的文件描述符（0 为 stdin，1 为 stdout，2 为 stderr）；cmd 为执行的操作，F_LOCK(锁定)，F_ULOCK(打开锁定)，F_TLOCK，F_TEST。len 为文件偏移量。当文件被一个进程锁定后，其他 lockf 函数将阻塞，直到该文件解锁。

5. pipe 函数用法

```
#include <unistd.h>
int pipe(int filedes[2]);
```

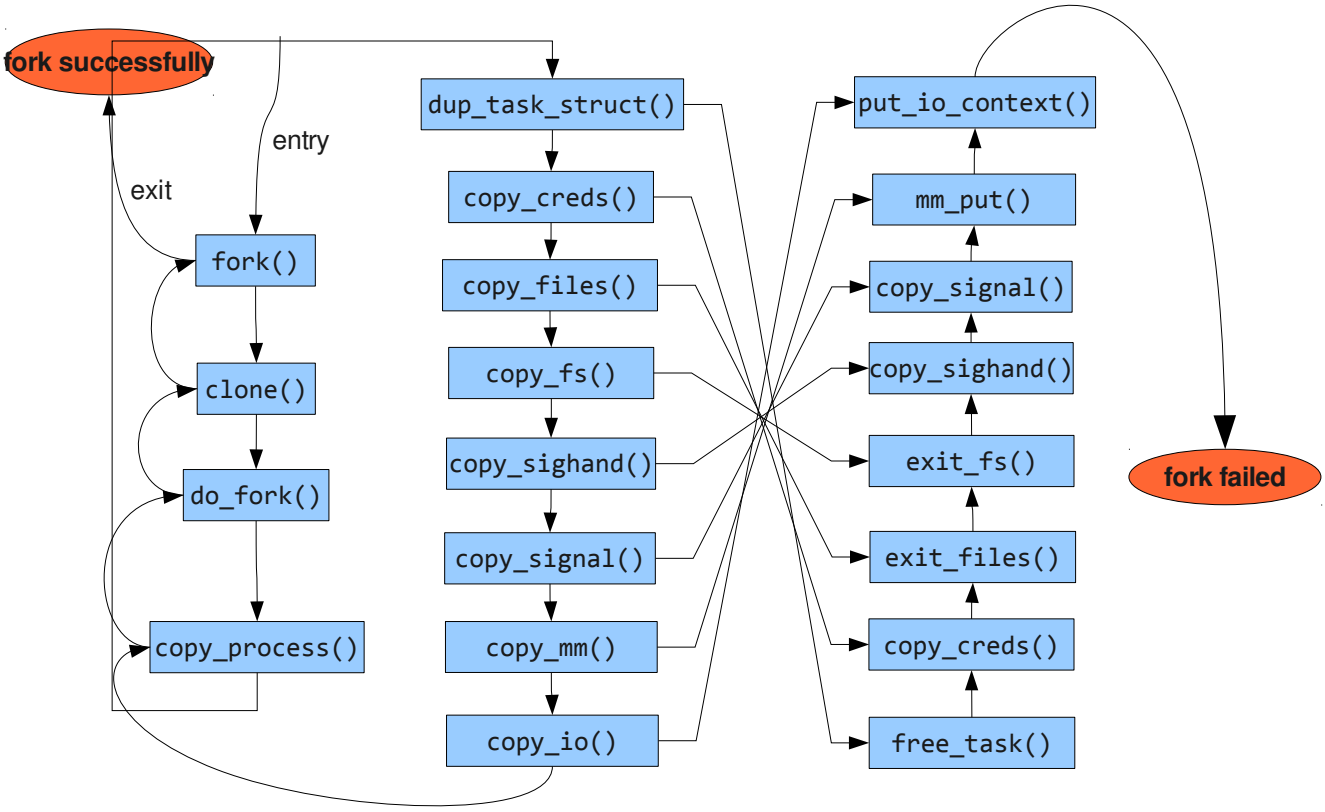
参数说明：经由参数 filedes 返回两个文件描述符：filedes[0]为读而打开，filedes[1]为写而打开。filedes[1]的输出是 filedes[0]的输入。

通常，调用 pipe 的进程接着调用 fork，这样就创建了从父进程到子进程的 IPC 通道。



五、思考

1. 系统是怎样创建进程的？



详见实验准备。

2．可执行文件加载时进行了哪些处理？

Linux 用 exec 系统调用加载可执行文件。其中有很多 architecture-specific 的汇编代码，尚未看明白。

3．当首次调用新创建进程时，其入口在哪里？

新进程从 fork()函数后开始执行，并且 fork()在子进程中返回 0。

4．进程通信有什么特点？

Linux 进程有一共有以下几种方法

1． 管道通信：只能用于具有公共祖先的进程间，半双工的通信方式。

2． 命名管道：允许无亲缘关系进程通信，半双工的通信方式。

3． XSI IPC

1） 消息队列： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

2） 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

3） 共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

4. 信号：信号机制提供了一种异步处理事件的方法，避免了 busy waiting，用于通知接收进程某个事件已经发生。

5. 网络 IPC－套接字：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。

六、参考资料

[1] W. Richard Stevens, Stephen A. Rago. *Advanced Programming in the Unix Environment, Second Edition.* Addison-Wesley.

[2] Robert Love. *Linux Kernel Development, Second Edition.* Novell Press.