实验四 进程管理实验

# 实验目的

通过实际上机调试和运行程序了解 Linux 系统中的进程基本编程：创建和终止，了解父进程和子进程的概念。

# 实验内容

进程（process 或 task）现在已经成为操作系统和并发程序设计中一个非常重要的概念。关于进程概念的讨论在操作系统中已经很详细。在 Linux 系统中，进程被认为是具有一定功能的程序关于一个数据集合的一次运行活动，是处于活动状态的计算机程序。进程在运行态、就绪态、等待态、stopped 和 zombie 这几种状态之间相互转化，但对用户是透明的。如果 Linux 系统中某个进程崩溃，不会影响到其它的进程。每个进程运行在各自的虚拟地址空间中，通过一定的通讯机制，它们之间才能发生联系。

Linux 系统中的最大进程数目缺省值一般为 512，每个进程都有一个在系统中唯一的进程标识号（process Identifiers），系统就根据这些进程 ID 来管理进程。创建一个新进程的唯一方法是由某个已经存在的进程调用 fork 函数（或 vfork 函数，本讲义不介绍），被创建的新进程是子进程，已经存在的进程是父进程。当然某些特殊进程并不需要通过这种方法来创建，如 swapper 进程、init 进

程等，它们是作为操作系统启动的一部分而被内核创建的。函数 fork（）是用来创建新进程的：

pid\_t fork（void）； // 数据类型 pid\_t 可认为就是 int 类型

函数 fork（）是一个单调用双返回的函数。具体的说，某个进程调用此函数后，若创建子进程成功，则这个函数在父进程中返回值是创建的子进程的进程标识号，而在子进程中返回值为零，否则（创建不成功）返回为-1。每个进程可以有多个子进程，但没有哪个函数调用是用来将子进程的进程标识号返回给父进程的；而每个进程至多有一个父进程，利用 getppid（）函数可以得到父进程的进程标识号。回此 fork（）函数将子进程的进程标识号返回给了父进程，便于父进程利用此进程标识号与子进程取得联系；而统一给子进程返回零。这个可以很简单地区分父进程和子进程。

子进程是父进程的一个拷贝。具体地说，子进程从父进程那里得到了数据段和堆栈段的拷贝，这些需要分配新的内存，而不是与父进程共享内存。函数 fork

（）返回后，子进程和父进程一样都是从调用 fork（）函数的语句的下一条开始执行。例 4-1 示范了 fork 函数的常见方法：

例 4-1 fork.c #include <stdio.h>

#include <sys/types.h> #include <unistd.h>

int main(void)

{

pid\_t pid;

printf("Process Creation Study\n"); pid = fork();

switch(pid) {

case 0:

printf("Child process is running,CurPid is %d, ParentPid is %d\n", pid, getppid());

break; case -1:

perror("Process creation failed\n"); break;

default:

printf("Parent process is running,ChildPid is %d, ParentPid is %d\n", pid, getpid());

break;

}

exit(0);

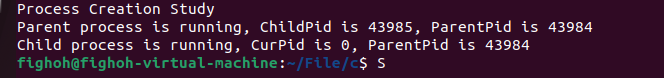
}

程序的编辑、整理、编译和运行可以参考以下步骤进行（以后的参考程序类似处理）：

1. 在 Linux 系统提示符下输入命令 gcc –o fork fork.c 进行编译。若有错误，gcc 将会给出一些提示信息，指出哪些行上的代码有问题，则要进行适当修改（还是用 vi 编辑器）。
2. 若能编译通过（但不能说明程序已经没有任何问题了），在 Linux 系统提示符下输入命令 ls 可看到在当前目录下多出一个名 fork.out 的文件，这就是 Linux 系统下的一种可执行文件。
3. 在 Linux 系统提示符下输入命令 ./fork.out 即可看到程序运行的结果

（输入 ./ 是告诉 Linux 系统我们要运行的程序在当前目录下）。

## 请回答，程序的运行结果是什么？并且分析程序输出这个结果的原因。



发现父进程的编号是43984，子进程的编号是43985。我们发现子进程此时对应的父进程编号是43984

如果再次运行此程序，系统分配给进程的 ID 一般会发生变化，两次运行的 结果均是子进程先运行。一般来说，fork 之后是父进程先执行还是子进程先执行 是不确定的，这取决于内核所使用的算法。将例 4-1 稍微修改一下就可以发现父 进程和子进程是交替执行的，因为操作系统一般让所有的进程都享有同等执行权，除非某个进程的优先级比其它的高，例 4-2 是对 4-1 的改进。

例 4-2 fork2.c #include <stdio.h> #include <sys/types.h> #include <unistd.h> int main(void)

{

pid\_t pid; char \* msg; int k;

printf("Process Creation Study\n"); pid = fork();

switch(pid) {

case 0:

msg = "Child process is running"; k = 3;

break; case -1:

perror("Process creation failed\n"); break;

default:

msg = "Parent process is running"; k=5;

break;

}

while(k > 0)

{

puts(msg); sleep(1);

k--;

}

exit(0);

}

## 请回答，程序的运行结果是什么？并且分析程序输出这个结果的原因。

## 

把k的值也输出了一下，可以发现子进程和父进程是交替运行的，同时在fork函数之后，应该是父进程先运行。

**执行新程序**

使用fork或vfork创建子进程后，子进程通常会调用exec函数来执行另外一个程序。系统调用exec用于执行一个可执行程序以代替当前进程的执行映像。需要注意的是exec调用并没有生成新进程。一个进程一旦调用exec函数，它本身就“死亡了”，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段和堆栈段，唯一保留的就是进程的ID。也就是说，对系统而言，还是同一个进程，不过执行的已经是另外一个程序了。下面通过例4-3演示exec函数的用法。

例4-3：程序processimage.c和exec.c程序processimage.c

#include <stdio.h> #include <sys/types.h> #include <unistd.h>

int main(int argc,char \*argv[],char \*\*environ)

{

int i;

printf("I am a process image!\n");

printf("My pid = %d, parentpid = %d\n", getpid(), getppid()); printf("uid = %d,gid = %d\n", getuid(), getgid());

for(i=0; i< argc; i++) printf("argv[%d]:%s\n",i ,argv[i]);

}

## 程序exec.c

#include <stdio.h> #include <sys/types.h> #include <unistd.h>

int main(int argc,char \* argv[],char \*\* environ)

{

pid\_t pid;

int stat\_val;

printf("Exec example!\n"); pid = fork();

switch(pid) {

case -1:

perror("Process Creation failed\n"); exit(1);

case 0:

printf("Child process is running\n");

printf("My pid = %d ,parentpid = %d\n",getpid(),getppid()); printf("uid = %d,gid =%d\n",getuid(),getgid()); execve("processimage",argv, environ);

printf("process never go to here!\n"); exit(0);

default:

printf("Parent process is running\n"); break;

}

wait(&stat\_val); exit(0);

}

**再编译第二个程序：gcc –o execve execve.c**

**先编译第一个程序，生成可执行文件：gcc –o processimage processimage.c**

## 回答：程序执行的结果是什么？请分析改程序的执行过程。

## 父进程fork之后，会打印“Parent process is running”然后结束。而子进程执行了execve（）函数，改变了它的内容，使得其里面的内容变成了processimage.c的对应代码，所以会先后打印两次不同的内容。但是它本身的id等信息没有发生变化。

## 

## 等待进程结束

当父进程先于子进程退出时，如果父进程没有调用wait和waitpid函数，子进程就会进入僵死状态。如果父进程调用了wait和waitpid函数，就不会使子进程变为僵尸进程，这两个函数声明如下：

#include<sys/types.h> #include<sys/wait.h> pid\_t wait(int \*statloc);

pid\_t waitpid(pid\_t pid, int \*statloc, int options);

wait函数使父进程暂停执行，直到它的一个子进程结束为止。该函数的返回值是终止运行的子进程的PID。参数statloc所指向的变量存放子进程的退出码，即从子进程的main函数返回的值或子进程中exit函数的参数。如果statloc不是一个空指针，状态信息将被写入它指向的变量。

检查wait和waitpid所返回的终止状态的宏

|  |  |
| --- | --- |
| 宏定义 | 说明 |
| WIFEXITED(stat\_val) | 若子进程是正常结束的，该宏返回一个非零值，表示真。若子进程异常结束，返回零，表示假 |
| WEXITSTATUS(stat\_val) | 若WIFEXITED返回值非零，它返回子进程中exit或exit参数的低8位 |
| WIFSIGNALED(stat\_val) | 若子进程异常终止，它就取得一个非零值，表示真 |
| WTERMSIG(stat\_val) | 如果宏WIFSIGNALED的值非零，该宏返回使子进程异常终止的信号编号 |
| WIFSTOPPED(stat\_val) | 若子进程暂停，它就取得一个非零值，表示真 |
| WSTOPSIG(stat\_val) | 若WIFSTOPPED非零，它返回使子进程暂停的信号编号 |

waitpid也用来等待子进程的结束，但它用于等待某个特定进程结束。参数pid指明要等待子进程的PID。pid值的意义见下表。参数statloc的含义与wait函数中的statloc相同。options参数允许用户改变waitpid的行为，若将该参数赋值为 WNOHANG，则使父进程不被挂起而立即返回并执行其后的代码。

aitpid函数参数pid不同取值的意义

|  |  |
| --- | --- |
| 取值 | 意义 |
| pid>0 | 等待其进程ID等于pid的子进程退出 |
| pid=0 | 等待其进程ID等于调用进程的组ID的任一子进程 |
| pid<-1 | 等待其组ID等于pid绝对值的任一进程 |
| pid=-1 | 等待任一子进程 |

如果想让父进程周期性地检查某个特定的子进程是否已经退出，可以按如下方式调用waitpid。

waitpid(child\_pid,(int \*) 0,WNOHANG);

如果子进程尚未退出，它将返回0；如果子进程已经结束，则返回child\_pid。调用失败时返回-1。失败的原因包括没有该子进程、参数不合法等。

注意：wait等待第一个终止的子进程，而waitpid则可以指定等待特定的子进程。waitpid提供了一个wait的非阻塞版本。有时希望取得一个子进程的状态，但不想使父进程阻塞，waitpid提供了一个这样的选项：WNOHANG，它可使调用者不阻塞。如果一个没有任何子进程的进程调用wait函数，会立即出错返回。例 4-4是等待进程的处理示例。

例4-4 wait.c #include <stdio.h>

#include <sys/types.h> #include <sys/wait.h> #include <unistd.h>

int main()

{

pid\_t pid;

char \*msg;

int k;

int exit\_code;

printf("Study how to get exit code\n"); pid = fork();

switch(pid) {

case 0:

msg = "Child process is running"; k = 5;

exit\_code = 37; break;

case -1:

perror("Process creation failed\n"); exit(1);

default:

exit\_code = 0; break;

}

/\* 父子进程都会执行以下这段代码子进程中pid值为0，父进程pid值为子进程的ID \*/

if(pid != 0) { // 父进程等待子进程结束

int stat\_val;

pid\_t child\_pid; child\_pid = wait(&stat\_val);

printf("Child procee has exited, pid = %d\n",child\_pid); if(WIFEXITED(stat\_val))

printf("Child exited with code %d\n",WEXITSTATUS(stat\_val));

else

printf("Child exited abnormally\n");

}

程状态

else { // 子进程暂停5秒，在这个过程中可以运行命令ps aux查看父进

while(k-->0) { puts(msg); sleep(1);

}

}

exit(exit\_code);

}

**等待子进程的。**

**回答：程序运行的结果是什么？请根据结果结合程序代码分析父进程是如何**