**实验二 进程创建与管道通信**

**一、实验描述**

本实验共分成三个部分

1. **学习使用fork()系统调用创建子进程，体会父子进程之间的并发关系。**

请编写一段程序，使用系统调用fork()创建两个子进程，实现当此程序运行时，在系统中有一个父进程和两个子进程在活动。父进程的功能是输出一个字符“a”；两个子进程的功能是分别输出一个个字符“b”和一个字符“c”。 多次运行这个程序，试观察记录屏幕上的显示结果，并分析原因。 另外，为了更好地展示进程之间的父子关系，大家可以使用getpid()系统调用来获取当前进程的PID，并用getppid()用于获取当前进程的父进程的PID。

1. **继续体会进程之间的并发关系**

修改刚才的程序，将每一个进程输出一个字符改为用一个循环输出1000个字符（父进程输出1000个“a”，子进程分别输出1000个“b”和“c”），再观察程序执行时屏幕上出现的现象，并分析原因。

1. **进程的管道通信**

编写程序实现进程的管道通信。父进程使用系统调用pipe()创建一个无名管道，二个子进程分别向管道各写一句话：

Child 1 is sending a message!

Child 2 is sending a message!

父进程从管道中读出二个来自子进程的信息并显示出来。 补充材料中给出了管道通信实现过程中需要使用的系统调用的说明，请仔细阅读。

**二、实验过程描述**

**2.1 进程间并发关系**

使用fork()函数进行新进程的创建时，其在子进程中的返回值为0。基于这一特点，我们可以根据当前pid的数值进行执行区域的确定。

然而在实验中，我们发现，题目要求中所说的“创建两个子进程”是无法直接通过使用两次fork()函数直接得到的。fork()函数将当前环境中的变量复制给另外一个进程，同时，其下方的代码块会被父进程和子进程执行。这就带来一个问题：首先执行的的fork()函数生成的两个子进程均会执行下方的fork()函数，这样就会导致无论是父进程或子进程都生成了自己的子进程，也就是共有四个进程。

为了避免这种情况发生，我们应该考虑进程函数fork()的使用位置：应该在子进程执行期间再使用fork()函数，避免父进程再生成子进程。

使用fork()函数进行新进程的创建时，其在子进程中的返回值为0。基于这一特点，我们可以根据当前pid的数值进行执行区域的确定。同时通过查阅文献，当前进程的getpid()数值根据进程创建顺序的不同而顺序给出。所以根据以上事实，我们有如下**共识**：

1. 进程创建的先后顺序决定了getpid()值：父进程 < 子进程 < 第二子进程；
2. 用fork()函数创建新进程时，子进程的返回值为0，父进程的返回值不为零；

给予这样的考虑，我们有如下代码实现，其执行结果也一并附上。

----------------------------

*代码2-1 使用fork()函数进行两个子进程的创建*

#include <stdio.h>

#include <unistd.h>

#include <string.h>

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

/\* Here describes the father process

\* Use getpid() to get current pid #.\*/

printf("MainProcess's getpid() = %d\n", getpid());

/\* Use fork() function to establish a new process.\*/

int pid\_1 = fork();

/\* What is known:

\* 1. When a new process is forked, pid\_1 of son's

\* process equals 0, which is false

\* 2. Sub-process(son) is processed first\*/

if(pid\_1){

/\* Activity declaration of Father's process\*/

wait(pid\_1);

printf("Parent's getpid() = %d\n", getpid());

printf("Parent's pid\_1 = %d\n", pid\_1);

}

else{

int pid\_2 = fork(); // Another process is established.

/\* Here declares what sons are doing \*/

if(pid\_2 == 0){

/\* Declaration of Grandson\*/

printf("Grandson's getpid = %d\n", getpid());

printf("Grandson's pid\_1 = %d\n", pid\_1);

printf("Grandson's pid\_2 = %d\n", pid\_2);

}else{

/\* Declaration of Son process\*/

printf("Son's getpid() = %d\n", getpid());

printf("Son's pid\_1 = %d\n", pid\_1);

printf("Son's pid\_2 = %d\n", pid\_2);

}

}

return 0;

}

/\*

\* Conclusion:

\* Son is executed first

\* son's pid = 0;

\* Father's pid is greater than 0;

\* Son's pid != getpid()

\*/

*代码2-1 结束*

---------------------

其运行结果如下所示：

*---------------------*

*代码2-1-1 使用fork()函数进行子进程创建运行结果*

ubuntu@VM-0-6-ubuntu:~/os/02$ ubuntu@VM-0-6-ubuntu:~/os/02$ gcc fork.c

fork.c: In function ‘main’:

fork.c:26:3: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declaration]

wait(pid\_1);

^

ubuntu@VM-0-6-ubuntu:~/os/02$ ./a.out

MainProcess's getpid() = 27534

Grandson's getpid = 27536

Grandson's pid\_1 = 0

Grandson's pid\_2 = 0

Son's getpid() = 27535

Son's pid\_1 = 0

Son's pid\_2 = 27536

Parent's getpid() = 27534

Parent's pid\_1 = 27535

Child 2 sends a message.////////

ubuntu@VM-0-6-ubuntu:~/os/02$

*代码2-1-1 结束*

根据上面提到的关于pid, getpid()数值关系我们可以确定父进程、子进程、第二子进程的对应关系正确。**我们使用了系统调用wait()函数来保证区域代码执行的完整性。同时发现，运行过程中总是先执行新进程，后执行父进程**。

**2.2 继续体会进程间的并发关系**

我们将上面的代码进行一些修改，得到了如下所示的代码。

-------------------------

*代码2-2 修改后带有循环输出的子进程创建代码*

#include<stdio.h>

#include<unistd.h>

#include<string.h>

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

/\* Use fork() function to establish a new process.\*/

int pid\_1 = fork();

if(pid\_1){

/\* Activity declaration of Father's process\*/

// wait(pid\_1);

/\* print loop here.\*/

for(int i = 0; i < 1000; i++){

if(i % 10 == 0){

printf("a ");

}

}

printf("\n");

}

else{

int pid\_2 = fork(); // Another process is established.

/\* Here declares what sons are doing \*/

if(pid\_2){

/\* Declaration of son process \*/

wait(pid\_2);

for(int i = 0; i < 1000; i++){

if(i % 10 == 0){

printf("b ");

}

printf("\n");

}

}

else{

/\* Declaration of Grandson\*/

/\* print loop here.\*/

for(int i = 0; i < 1000; i++){

if(i % 10 == 0){

printf("a ");

}

}

printf("\n");

}

}

return 0;

}

*代码2-3 运行结束*

-----------------------------

上面代码中三个进程均有一共同特点，为运行时间长的批处理任务。得到的运行结果如下所示：

-------------------------

*代码2-3 运行结果*

ubuntu@VM-0-6-ubuntu:~/os/02$ gcc fork\_loop.c

fork\_loop.c: In function 憁ain?

fork\_loop.c:25:13: warning: implicit declaration of function 憌ait?[-Wimplicit-function-declaration]

wait(pid\_2);

^

ubuntu@VM-0-6-ubuntu:~/os/02$ ./a.out

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

ubuntu@VM-0-6-ubuntu:~/os/02$ c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c

b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b

*代码2-3 运行结果结束*

------------------------------

我们可以看到，尽管在同一个代码块当中，a的循环输出并不紧接着信息打印后进行，而是在第一子进程的信息打印完成后才进行。等到b打印时候，整个程序已经结束，出现了继续输入命令的提示符。而在第二子进程进行的时候，甚至都没有进行完全便关闭了控制台的输出，可以看出，系统调度进程采用了**时间片轮转**的方法，等到父进程结束，子进程自动结束。而无法预知父进程何时结束，有可能在子进程还没有完成时，父进程便退出了，所以我们应该控制进程的执行顺序。

**解决办法**：利用系统调用wait()函数，使得进程按照合理的顺序执行。将29行、14行处的wait()函数恢复，得到改进后的带有循环输出的进程创建代码，结果如下所示，可以看到程序可以按照既定的顺序执行，从而保证了执行的完整性。

-----------------------

*代码2-4 改进后的代码2-3的运行结果*

ubuntu@VM-0-6-ubuntu:~/os/02$ gcc fork\_loop\_opti.c

fork\_loop\_opti.c: In function 憁ain?

fork\_loop\_opti.c:10:9: warning: implicit declaration of function 憌ait?[-Wimplicit-function-declaration]

wait(pid\_1);

^

ubuntu@VM-0-6-ubuntu:~/os/02$ ./a.out

c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c

b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b

a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

ubuntu@VM-0-6-ubuntu:~/os/02$

*代码2-4 改进后的代码2-3运行结果结束*

------------------------------

**2.3 进程的管道通信**

进程的管道通信是通过一个长度为2的数组进行串行通信，使用read()函数进行读取，write()函数进行写入，从而新型进程间通信。实现的代码如下所示，代码中的注释详细说明了每部分的作用。

------------------------

*代码2-5 管道通信代码*

#include<stdio.h>

#include<unistd.h>

#include<string.h>

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

/\* Description of establishment of pipe.

\* In this scope, field[0] is opened for reading

\* while field[1] is opened for writing\*/

int field[2]; // pipe field[2] declaration

pipe(field); // pipe() for initialization

char buf\_read[200];

/\* Use fork() function to establish a new process.\*/

int pid\_1 = fork();

if(pid\_1){

/\* Activity declaration of Father's process\*/

wait(pid\_1);

/\* Father process reading from field[0] requires

\* close() function on field[1].\*/

close(field[1]);

read(field[0], buf\_read, 24);

printf("%s\n", buf\_read);

}

else{

int pid\_2 = fork();// Another process is established.

/\* Here declares what sons are doing \*/

if(pid\_2){

/\* Declaration of son

\* Son process reading from field[1] requires

\* close() function on field[0].\*/

close(field[0]);

char \* info = "Child 1 sends a message.";

write(field[1], info, strlen(info));

}

else{

/\* Declaration of Grandson

\* Grandson process reading from field[0] requires

\* close() function on field[0].\*/

close(field[0]);

char \* info = "Child 2 sends a message.";

write(field[1], info, strlen(info));

}

}

return 0;

}

*代码2-5 结束*

------------------------------

其运行结果如下所示：

*------------------------*

*代码2-6 管道通信运行结果*

ubuntu@VM-0-6-ubuntu:~/os/02$ gcc pipe.c

pipe.c: In function ‘main’:

pipe.c:18:3: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declaration]

wait(pid\_1);

^

ubuntu@VM-0-6-ubuntu:~/os/02$ ./a.out

Child 2 sends a message.

Child 1 sends a message.

ubuntu@VM-0-6-ubuntu:~/os/02$

**三、实验结果及其分析**

* 在不牵扯到批处理任务时候，孩子进程总是首先被执行，然后再执行父进程；
* pid数值和getpid()函数返回的数值不一定相同。前者为fork()函数的返回值，在子进程中为0，而在父进程中与子进程的getpid()函数的返回值相同，为子进程进程号。
* 在进行批处理任务时候，由于采用了时间片轮转处理方法对进程进行调度，所以很有可能在子进程还未结束时候父进程就已经结束。所以对于这种问题，我们可以采用wait(pid)的方式控制进程执行的次序，进而控制进程的完成。
* 对于管道的初始化，使用pipe()函数进行操作。Write()表示写入，read()表示读出。

=====END=====