**实验二 进程创建与管道通信**

**1 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。

**2 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*

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

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**。

**结论：**

* 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。

**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2-1-2 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2-1-3：代码2-1-2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2-1-3 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—1-4 改进后的代码2—1-3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2-1-4 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2-1-5 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2-1-5 结束*

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

其运行结果如下所示：

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

*代码2-1-6 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。

**实验三 线程管理**

**一、实验内容概述**

本实验主要训练编写Linux环境下的多线程程序。掌握最常用的三个函数pthread\_create，pthread\_join和pthread\_exit的用法，以及撞我向线程传递参数的方法

1. 创建线程

在这个任务中，需要在主程序中创建两个线程myThread1和myThread2，每个线程打印一句话。提示：先定义每个线程的执行体，然后在主函数中使用pthread\_create负责创建两个线程。整个程序等待子线程结束后再退出。

1. 向线程传递参数

在上一个程序的基础上，分别向两个线程传递一个字符和一个整数，并让线程负责将两个参数的值打印出来。

1. 使用两个线程实现数组排序

主程序中用数组data[10]保存10个整数型数据，创建两个线程，一个线程将这个数组中的数据从大到小排列输出；另一个线程将这个数组中的数据从小到大排列输出。

下面我们在实验的详细过程部分对上面的各实验进行详细的说明

**二、实验详细设计说明**

Linux下的多线程遵循POSIX线程接口，称为pthread。编写Linux下的多线程程序，需要使用头文件pthread.h，以保可以使用pthread库。

需要注意的是，由于pthread库不属于linux系统库，所以在进行程序编译时要加上-lpthread选项，否则编译不会通过。

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

*代码3-1 pthread的使用*

#include <stdio.h>

#include <pthread.h>

void myThread1(){

printf("This line is printed by thread 1\n");

**return**;

}

void myThread2(){

printf("This is printed by thread 2\n");

**return**;

}

int main(){

pthread\_t id\_1;

pthread\_t id\_2;

pthread\_create(**&**id\_1, NULL, (void \*)myThread1, NULL);

pthread\_create(**&**id\_2, NULL, (void \*)myThread2, NULL);

pthread\_join(id\_1, NULL);

pthread\_join(id\_2, NULL);

printf("This is printed by main thread\n");

**return** 0;

}

*代码3-2-1 结束*

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

线程创建中使用到了pthread\_create()函数用来创建线程以及明确这个线程的执行函数、传递参数。pthread()函数用来等待一个线程的结束。我们这里将第二个线程的创建放置在第一线程结束执行之前，以保证在一段时间内线程1和线程2是“并行执行”的。上方所述的代码只是进行了两个并发进程的创建，而并没有参数的传递。其运行结果如下所示：

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

*代码3-2-2：代码3-2-1运行结果*

ubuntu@VM-0-6-ubuntu:~/os/03$ ./thread

This is printed by thread 2

This line is printed by thread 1

This is printed by main thread

ubuntu@VM-0-6-ubuntu:~/os/03$

*代码3-2-2结束*

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

**2.2 向线程传递参数**

我们创建了一个新的源程序文件名为task\_2.c，其源程序代码如下所示：

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

*代码3-2-3 向线程传递参数task\_2.c*

#include <stdio.h>

#include <pthread.h>

void \* myThread1(void \* chr){

char \* temp **=** (char \*)chr;

printf("%c\n", **\*** temp);

**return** (void \*)0;

}

void \* myThread2(void \* num){

int \* temp **=** (int \*)num;

printf("%d\n", **\*** temp);

**return** (void \*)0;

}

*/\* Main function here*

*\* Threads establishment and*

*\* factors declaration.\*/*

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

pthread\_t id\_1;

pthread\_t id\_2;

int num **=** 65536;

char chr **=** 96;

int ret\_1 **=** pthread\_create(**&**id\_1, NULL, (void \*)myThread1, (void \*)**&**chr);

pthread\_join(id\_1, NULL);

int ret\_2 **=** pthread\_create(**&**id\_2, NULL, (void \*)myThread2, (void \*)**&**num);

pthread\_join(id\_2, NULL);

printf("This is printed by main thread\n");

**return** 0;

}

*代码3-2-3结束*

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

我们通过多次失败以及查阅文献得知，**线程的参数传递只有一个位置**，而且必须通过(void \*)进行传递。在进行多个参数传递时，需要将所有的参数加工成结构体，通过将结构体指针cast成(void \*)类型的指针后再进行传递。于是我们得到了上方所述的代码。

在主函数部分，我们仿照任务1中的结构创建了两个线程，同时声明两个需要被传递的参数

int num **=** 65526;

char chr **=** 96;

这两个参数的传递过程中，按照上方的信息，我们将这两个参数的地址cast为void \*，进行传递。在传递进入各自的线程动作描述函数后，在重新转变为各自的类型，并打印出来。于是我们得到上述代码的运行结果如下所示：

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

*代码3-2-4 带参数传递的多线程建立task\_2.c的运行结果*

ubuntu@VM-0-6-ubuntu:~/os/03$ gcc task\_2.c -lpthread

ubuntu@VM-0-6-ubuntu:~/os/03$ ./a.out

This line is printed by thread 1

`

65536

This is printed by main thread

ubuntu@VM-0-6-ubuntu:~/os/03$

*代码3-2-4 结束*

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

从结果中我们看到参数均被成功地传递到线程中并且成功执行。

**2.3 使用两个线程实现数组的排序**

主程序中用数组data[10]保存10个整数型数据，创建两个线程，一个线程将这个数组中的数据从大到小排列输出；另一个线程将这个数组中的数据从小到大排列输出。

应该明确的是，两个线程不可以同时在一个数组内进行内排序。两个线程同时按照不同的标准进行数据交换，所得的最终的排序结果必然是混乱的。**所以内排序算法的实现不可以在各自的线程内对同一个数组进行。**

程序设计是应该考虑的非常重要的一点是**复杂度**。两个线程并发执行，但是实际上也是单线进行的。所以基于这一点考虑，我们有如下设计思路：

1. **采用gcc::quick\_sort()进行排序**
2. **只进行一次排序**
3. **对于题目中要求的两种方式输出我们可以按照不同方向进行遍历，从而得到各自的结果**
4. **结果的输出不可以同时进行**

基于这样的设计思路，我们有如下的代码实现

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

*代码3-2-5 两个线程数组排序*

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

void \* myThread1(void \* var){

int \* temp **=** (int \*)var;

**for**(int i **=** 0; i **<** 10; i**++**){

printf("%d\t", temp[i]);

}

printf("\n");

**return** (void \*)0;

}

void \* myThread2(void \* var){

int \* temp **=** (int \*)var;

**for**(int i **=** 0; i **<** 10; i**++**){

printf("%d\t", temp[9 **-** i]);

}

printf("\n");

**return** (void \*)0;

}

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

*/\* Two threads' tid declaration \*/*

pthread\_t id\_1;

pthread\_t id\_2;

*/\* An array with random values.\*/*

int data[10] **=** {879,894651,23,7894563,32156,64351,87964531,535,894,3465};

*/\* Sort the array \*/*

qsort(data, 10, **sizeof**(data[0]), cmp);

int ret\_1 **=** pthread\_create(**&**id\_1, NULL, (void \*)myThread1, (void \*)data);

pthread\_join(id\_1, NULL);

*/\* Output cannot be made simultaneously.\*/*

int ret\_2 **=** pthread\_create(**&**id\_2, NULL, (void \*)myThread2, (void \*)data);

pthread\_join(id\_2, NULL);

**return** 0;

}

*代码3-2-5结束*

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

其运行结果如下所示：

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

*代码3-2-6：代码3-2-5运行结果*

ubuntu@VM-0-6-ubuntu:~/os/03$ gcc task\_3.c -lpthread

ubuntu@VM-0-6-ubuntu:~/os/03$ ./a.out

23 535 879 894 3465 32156 64351 894651 7894563 87964531

87964531 7894563 894651 64351 32156 3465 894 879 535 23

ubuntu@VM-0-6-ubuntu:~/os/03$

*代码3-2-6结束*

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

从结果中我们可以看到，程序能够良好地运行

**三、实验结果分析**

* 实验中创建进程可以使用pthread()进行进程的创建，同时需要明确传递的指针以及开始、并发、结束的区间
* 对于内排序不可以在同一个数组内有两个线程进行排序
* 只能有一个参数以void\*的形式传递进入。如果有多个参数，可以通过结构体指针的方式传递

**实验四 利用信号量实现线程互斥与同步**

**一、实验描述**

**1.1 生产者消费者问题**

有数据文件1.dat和2.dat分别存放了10个整数。创建4个线程，其中两个线程read1和read2负责分别从文件1.dat和2.dat中读取一个整数到公共的缓冲区，另两个线程operate1和operate2分别从缓冲区读取数据作加运算和乘运算。使用信号量控制这些线程的执行，保证缓冲区中的数据只能被计算一次（加或者乘），计算完成之后才能继续进行数据的读取工作。 提示：这是一个缓冲区长度为2的生产者消费者问题。read1和read2是生产者，operate1和operate2是消费者。需要互斥使用的缓冲区的长度为2。在进行读取的时候不见得一定是read1读一个数然后read2读一个数，允许read1读得快（或者read2读得快），所以缓冲区中当前的数据都来自于1.dat或者2.dat是被允许的。 例如，事先编辑好数据文件1.dat和2.dat的内容分别为1 2 3 4 5 6 7 8 9 10和 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10，那么运行你编写的程序，可能得到如下类似的结果：

1+-1=0

-2\*2=-4

3+4=7

-3\*-4=12

5+-5=0

6\*-6=-36

…

**1.2 严格控制数据来源**

如果严格限制加和乘的两个操作数必须分别来自1.dat和2.dat，应该如何修改上面的程序？

**二、实验分析**

在任务1中，要求运算、取数的进程并发随机执行，然而需要明确一下限制：

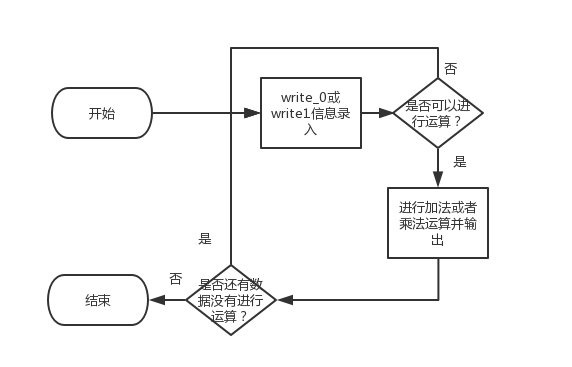
1. 取数的函数必须在运算函数之后执行，即，必须在**数组中的元素已经使用过**或者在临时存储数组中**没有其他元素**的情况下进行操作。**不可以**在数组中元素还没有使用的情况下在取数到数组中。
2. 运算的函数必须在**数组中元素全部为新元素的**情况下进行，即，数组中的元素不能全部或者有一部分被使用，也不能为空
3. 任务1中要求的取数函数随机发生，同时，允许一次运算当中的数据来源为统一文件。这便要求**不能重复重复使用数据写入的位置**
4. 任务2中需要有额外限定：在明确了以上限制的情况下，我们认为需要设定严格限制加和乘的两个操作数必须分别来自1.dat和2.dat。

在明确了以上限定后，我们将开始实验的设计

**三、实验设计分析**

**3.1 无数据来源限定情况下四进程并发设计**

根据上述讨论，我们可以确定工作模式以及逻辑顺序。总体的程序流程图如下所示：



*Fig. 3-1 无数据来源限定情况下工作流程图*

程序开始后，首先I/O线程将数据添加到数组当中。待两个位置均被沾满后，运算线程相应地进行运算，打印信息，并在数据取走后将信号量复位。

然而在实际的运行过程中，我们发现了一个问题：一**个进程成功地进行数据录入后，另外一个录入进程无法得知当前应该录入的位置。**

为了解决这一个问题，我们应该有**一个模为2的计数器确定每次信息录入位置的索引**。所以，在每次要录入信息之前，我们应该检查当前计数器的数值，从而得到录入索引位置。这样的话，我们应该在上述限定中增加上一个限定：**只能有一个线程在同一时刻访问当前的索引数值**。

为了满足上述的限定，我们将使用信号量来进行进程的同步控制。详细的信号量的描述如下表格所示：

**表格3-3-1 任务1的信号量使用描述**

| **信号量名称** | **信号量初始值** | **作用** |
| --- | --- | --- |
| Write | 2 | 控制写入的同步性 |
| read | 0 | 控制数据读取的同步性以及运行顺序 |
| check\_idx | 1 | 控制检查录入索引的信号量 |

在明确了如上限定以及所有流程后，我们得到最终的流程图如下所示。通过以上的设计，我们得到了如下的代码：

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

*代码4-3-1 无数据来源限定情况下四进程并发设计源代码*

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <semaphore.h>

*/\* int [] for the buffer of temporary integer holder \*/*

int buf[2];

*/\* semaphores declaration\*/*

sem\_t Write;

sem\_t take;

sem\_t check\_idx;

int insert\_index;

*/\* Thread of info writing*

*\* data inserted by this thread are all negative. \*/*

void \* write\_0(){

int data\_set[] **=** {

**-**1, **-**2, **-**3, **-**4, **-**5, **-**6, **-**7, **-**8, **-**9, **-**10

};

*/\* Passing values towards buffer array\*/*

**for**(int i **=** 0; i **<** 10; i**++**){

sem\_wait(**&**Write);

buf[insert\_index] **=** data\_set[i];

insert\_index **=** (insert\_index **+** 1) **%** 2;

*//printf("%d is inserted\n", data\_set[i]);*

*/\* Check whether to post signal "take"\*/*

sem\_wait(**&**check\_idx);

**if**(insert\_index **==** 0){

sem\_post(**&**take);

*// printf("write0: ready to take\n");*

}

sem\_post(**&**check\_idx);

}

**return** (void \*)0;

}

*/\* Thread of info writing*

*\* data inserted by this thread are all positive. \*/*

void \* write\_1(){

int data\_set[] **=** {

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

};

**for** (int i **=** 0; i **<** 10; i**++**){

sem\_wait(**&**Write);

buf[insert\_index] **=** data\_set[i];

insert\_index **=** (insert\_index **+** 1) **%** 2;

*//printf("%d is inserted\n", data\_set[i]);*

sem\_wait(**&**check\_idx); *// Make sure only 1 thread check idx*

**if**(insert\_index **==** 0){

sem\_post(**&**take);

*//printf("write1: ready to take\n");*

}

sem\_post(**&**check\_idx);

}

**return** (void \*)0;

}

*/\* Thread of info calculation.*

*\* 2 numbers from buf[2] will be output after addition. \*/*

void \* AddFun(){

**for**(int i **=** 0; i **<** 10; i**++**){

sem\_wait(**&**take);

int temp\_0 **=** buf[0];

int temp\_1 **=** buf[1];

printf("%d + %d = %d\n", temp\_0, temp\_1, temp\_0 **+** temp\_1);

sem\_post(**&**Write);

sem\_post(**&**Write);

}

**return** (void \*)0;

}

*/\* Thread of info calculation.*

*\* 2 numbers from buf[2] will be output after multiplication. \*/*

void \* MulFun(){

**for**(int i **=** 0; i **<** 10; i**++**){

sem\_wait(**&**take);

int temp\_0 **=** buf[0];

int temp\_1 **=** buf[1];

printf("%d \* %d = %d\n", temp\_0, temp\_1, temp\_0 **\*** temp\_1);

sem\_post(**&**Write);

sem\_post(**&**Write);

}

**return** (void\*) 0;

}

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

*/\* semaphores' initialization\*/*

sem\_init(**&**Write, 0, 2);

sem\_init(**&**take, 0, 0);

sem\_init(**&**check\_idx, 0, 1);

*/\* index of inserting value initialzation\*/*

insert\_index **=** 0;

*/\* Threads declaration \*/*

pthread\_t id\_write\_0;

pthread\_t id\_write\_1;

pthread\_t id\_AddFun;

pthread\_t id\_MulFun;

*/\* Threads initialization \*/*

pthread\_create(**&**id\_write\_0, NULL, (void \*)write\_0, NULL);

pthread\_create(**&**id\_write\_1, NULL, (void \*)write\_1, NULL);

pthread\_create(**&**id\_AddFun, NULL, (void \*)AddFun, NULL);

pthread\_create(**&**id\_MulFun, NULL, (void \*)MulFun, NULL);

*/\* Threads finish \*/*

pthread\_join(id\_AddFun, NULL);

pthread\_join(id\_MulFun, NULL);

pthread\_join(id\_write\_0, NULL);

pthread\_join(id\_write\_1, NULL);

**return** 0;

}

​

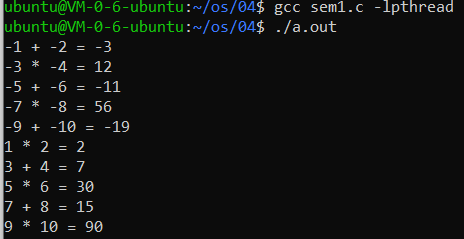
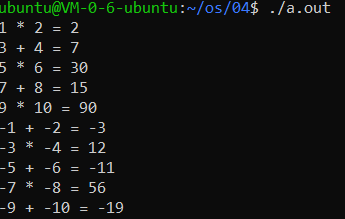
*代码4-3-1结束*

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

运行结果如下所示：

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

*代码4-3-2： 代码4-3-1运行结果*

**

*代码4-3-2 结束*

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

由于没有控制读取的顺序，所以在最终运行的时候，很有可能出现不同的运行结果，这些都是正常的。上方展示了两种典型的不同运算结果。

通过实验，我们发现，使用信号量进行进程间的同步是非常有效的。任务2中将会对两个数据的来源进行控制，我们将对此进行进一步讨论。

**3.2 有数据来源限定情况下四进程并发设计**

题目中要求，对于运算中两个数据的来源，我们需要通过信号量的方法对数据的来源进行限定。通过分析，我们发现任务2与任务1相似，同样为取得数据后进行运算。而不同的是，我们在本次任务中只需要加上对于数据来源限定的相关代码即可。下面我们将本次任务中使用的信号量进行说明

| **信号量名称** | **信号量初始值** | **作用** |
| --- | --- | --- |
| write\_0 | 1 | 控制写入第一个位置的同步性 |
| write\_1 | 1 | 控制写入第二个位置的同步性 |
| read\_0 | 0 | 控制0号位置数据读取 |
| read\_1 | 0 | 控制1号位置数据读取 |

在下方展示了我们这次任务的代码。

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

*代码4-3-3： 有数据来源限定情况下四进程并发代码*

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <semaphore.h>

*/\* Global variables, including buffers and semaphores \*/*

int buf[2];

sem\_t write\_buf\_0;

sem\_t write\_buf\_1;

sem\_t take\_buf\_0;

sem\_t take\_buf\_1;

*/\* Thread of info writing*

*\* data inserted by this thread are all positive. \*/*

void \*write\_0(){

int data\_set[] **=** {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

**for** (int i **=** 0; i **<** 10; i**++**){

int temp **=** data\_set[i];

sem\_wait(**&**write\_buf\_0);

buf[0] **=** temp;

sem\_post(**&**take\_buf\_0);

}

**return** (void \*)0;

}

*/\* Thread of info writing*

*\* data inserted by this thread are all negative. \*/*

void \*write\_1(){

int data\_set[] **=** {**-**1, **-**2, **-**3, **-**4, **-**5, **-**6, **-**7, **-**8, **-**9, **-**10};

**for** (int i **=** 0; i **<** 10; i**++**){

int temp **=** data\_set[i];

sem\_wait(**&**write\_buf\_1);

buf[1] **=** temp;

sem\_post(**&**take\_buf\_1);

}

**return** (void \*)0;

}

*/\* Thread of info calculation.*

*\* 2 numbers from buf[2] will be output after addition. \*/*

void \*AddFun(){

**while** (1){

sem\_wait(**&**take\_buf\_0);

sem\_wait(**&**take\_buf\_1);

int temp\_0 **=** buf[0];

int temp\_1 **=** buf[1];

printf("%d + %d = %d\n", temp\_0, temp\_1, temp\_0 **+** temp\_1);

sem\_post(**&**write\_buf\_0);

sem\_post(**&**write\_buf\_1);

}

**return** (void \*)0;

}

*/\* Thread of info calculation.*

*\* 2 numbers from buf[2] will be output after multiplication. \*/*

void \*MulFun(){

**while** (1){

sem\_wait(**&**take\_buf\_0);

sem\_wait(**&**take\_buf\_1);

int temp\_0 **=** buf[0];

int temp\_1 **=** buf[1];

printf("%d \* %d = %d\n", temp\_0, temp\_1, temp\_0 **\*** temp\_1);

sem\_post(**&**write\_buf\_0);

sem\_post(**&**write\_buf\_1);

}

**return** (void \*)0;

}

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

sem\_init(**&**write\_buf\_0, 0, 1);

sem\_init(**&**write\_buf\_1, 0, 1);

sem\_init(**&**take\_buf\_0, 0, 0);

sem\_init(**&**take\_buf\_1, 0, 0);

printf("ready\_for\_cal: %d\n", get\_sem\_val(**&**ready\_for\_cal));

pthread\_t id\_write\_0;

pthread\_t id\_write\_1;

pthread\_t id\_AddFun;

pthread\_t id\_MulFun;

pthread\_create(**&**id\_write\_0, NULL, (void \*)write\_0, NULL);

pthread\_create(**&**id\_write\_1, NULL, (void \*)write\_1, NULL);

pthread\_create(**&**id\_AddFun, NULL, (void \*)AddFun, NULL);

pthread\_create(**&**id\_MulFun,- NULL, (void \*)MulFun, NULL);

pthread\_join(id\_write\_0, NULL);

pthread\_join(id\_write\_1, NULL);

pthread\_join(id\_AddFun, NULL);

pthread\_join(id\_MulFun, NULL);

**return** 0;

}

*代码4-3-3结束*

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

**四、结果分析**

* 对于任务中可能产生的竞争情况，应该使用信号量进行控制
* 将共同资源区域代码包含在信号量控制区块里面，以保证线程同步的正确
* 对于顺序存储的情况，可以使用模计数器进行存储索引的控制

# 实验五 基于消息队列和共享内存的进程间通信

## 一、实验描述

### 1. 消息的创建、发送和接受

主函数中使用fork()系统调用创建两个子进程sender和receiver。sender负责接收用户输入的一个字符串，并构造成一条消息，传送给receiver。Receiver在接收到消息后在屏幕上显示出来。此过程一直继续直到用户输入exit为止。在程序设计过程中使用msgget()、msgsnd()、msgrcv()、msgctl()。

### 2. 共享存储取得创建、附接和断接

主函数中使用fork()系统调用创建两个子进程sender和receiver。Sender创建共享内存区域并从用户输入得到一个整数放入共享内存区域，receiver负责取出此数并将此整数的平方计算出来。要求程序可以计算10个整数的平方值。在程序设计过程中使用shmget()、shmat()、shmctl()。

## 二、实验分析

本实验秩序按照补充材料中的内容认真研读后实现即可，没有逻辑上的难点。

## 三、实验过程分析

### 3.1 消息队列

使用消息队列进行进程间的消息传送需要牵扯到消息队列的建立、数据的发送、数据接收以及消息队列的回收这四部分。这四部分当中需要使用到的头文件如下所示。

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

代码5-3-1 消息队列模拟msg\_queue.c需要用到的头文件

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

\**代码5-3-1 结束*

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

本实验中需要注意一点的是，**无论是接受方还是发送方，只要是存在于两个队列当中，必须进行消息队列的创建**。由于系统的内部机制，在消息队列存在的情况下，系统可以直接获得消息队列，而无需关心消息队列重复建立产生内存浪费的情况。消息队列的建立代码如下所示:

int msg\_id **=** msgget(MSG\_KEY, IPC\_CREAT **|** 0666);

其中

key：共享储存区描述符。

size：共享储存区的大小。

shmflg：建立标志和储存权限，取值如下，且可通过or组合在一起：

IPC\_CREAT：建立新的共享区段。

IPC\_EXCL：和IPC\_CREAT标志一起使用，如果共享区段已存在失败返回。

SHM\_HUGETLB：使用"huge pages"来分配共享区段。

SHM\_NORESERVE：不要为共享区段保留交换空间

消息发送msgsnd的功能是发送一条消息到指定的消息队列。消息接收msgrcv其功能是从指定消息队列接收一条消息 他们的原型是：

int msgsnd(msgid，msgp，size，flag);

ssize\_t msgrcv(msgid，msgp，size，type，flag);

其中

msgid：消息队列描述符。

msgp：用来存放接收到的消息的用户数据结构的地址。

size：指示由msgp 指向的数据结构中字符数组的长度，即消息的长度。

type：用户要读的消息类型：

type为0 ：接收该队列中的全部消息；

type 为正：接收队列中第一个类型为type的消息；

type 为负：接收小于或等于type绝对值的最小的类型的第一个消息。

flag：用来指明在队列没有数据的情况下msgrcv()所应采取的行动。如果是IPC\_NOWAIT，那么如果队列中没有可读的消息，则不等待，直接返回ENOMSG错误。如果是MSG\_NOERROR，则当消息超过了size的时候会被截断而不会报错。当msgflg为MSG\_EXCEPT时，如果type的值大于0，则接收类型不等于type的第一条消息。

msgrcv返回收到的消息正文的字节数。

基于上述情况，我们给出基于消息队列的进程间通信代码以及运行结果

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

代码5-3-2 基于消息队列的进程间通信代码

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/msg.h>

#include <string.h>

#define MSG\_KEY 1223

*// message queue establising: int msgget(key\_t key, int msgflg);*

*// message send: int msgsnd(msgid, msgp, size, flag);*

*/\* Define a set of message as a strcture.\*/*

**struct** mybuf{

long mtype;

char text[256];

};

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

int pid **=** fork();

**if**(pid **>** 0){

**while**(1){

**struct** mybuf buf;

*/\* Message queue establishment\*/*

int msg\_id **=** msgget(MSG\_KEY, IPC\_CREAT **|** 0666);

*/\* Receive message from queue and cp . buf\*/*

msgrcv(msg\_id, **&**buf, **sizeof**(buf.text), 0, MSG\_NOERROR);

*/\* Check if the message received === 'exit' \*/*

**if**(buf.text[0] **==** 'e' **&&** buf.text[1] **==** 'x' **&&** buf.text[2] **==** 'i' **&&** buf.text[3] **==** 't'){

**break**;

}

*/\* Print the message \*/*

printf("Message received: %s\n", buf.text);

}

}

**else**{

**while**(1){

**struct** mybuf buf;

*/\* Message queue establishment\*/*

int msg\_id **=** msgget(MSG\_KEY, IPC\_CREAT **|** 0666);

*/\* Receive message from console and print \*/*

printf("Please input your message:\n");

fgets(buf.text, **sizeof**(buf.text), stdin);

buf.mtype **=** 1000;

*/\* Send the message \*/*

msgsnd(msg\_id, **&**buf, **sizeof**(buf.text), IPC\_NOWAIT);

*/\* Check if the message received === 'exit' \*/*

**if**(buf.text[0] **==** 'e' **&&** buf.text[1] **==** 'x' **&&** buf.text[2] **==** 'i' **&&** buf.text[3] **==** 't'){

printf("Goodbye\n");

abort **=** 1;

**break**;

}

*/\* Put the current process into READY QUEUE \*/*

sleep(0);

}

}

**return** 0;

}

代码5-3-2 结束

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

运行结果如下所示：

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

代码5-3-1：代码5-3-2运行结果

ubuntu@VM-0-6-ubuntu:~/os/05$ gcc msg\_queue.c -lpthread

msg\_queue.c:41:5: warning: built-in function ‘abort’ declared as non-function

int abort = 0;

^

msg\_queue.c: In function ‘main’:

msg\_queue.c:50:12: warning: implicit declaration of function ‘fork’ [-Wimplicit-function-declaration]

int pid = fork();

^

msg\_queue.c:85:4: warning: implicit declaration of function ‘sleep’ [-Wimplicit-function-declaration]

sleep(0);

^

ubuntu@VM-0-6-ubuntu:~/os/05$ ./a.out

Please input your message:

456

Message received: 456

Please input your message:

**exit**

ubuntu@VM-0-6-ubuntu:~/os/05$ Goodbye

*代码5-3-2结束*

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

可以看到，程序能够按照要求识别出exit和正常信息，达到信息传送的目的。

### 3.2 共享内存

与消息队列相似的是，无论是发送方或者是接收方，都需要对共享队列进行附接。下面给出基于共享内存的消息传递源代码。具体代码作用已经在注释中给出，再次不过多叙述。

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

*代码5-3-3 基于共享内存的消息传递*

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

#include <math.h>

#define SHM\_KEY 1221

#define SIZE 1024

int main(){

int pid **=** fork();

**if**(pid **>** 0){

*/\* 共享存储区的建立，得到存储区的描述符，开通读写权限\*/*

int shid **=** shmget(SHM\_KEY, SIZE, IPC\_CREAT **|** 0666);

*/\* 共享存储区的附接，得到存储区的地址\*/*

int \* addr **=** shmat(shid, 0, 0);

*/\*在子进程进行完毕后在进行后续操作 \*/*

wait(pid);

printf("Number reveived are:\n");

*/\* 获取存储区内部信息 \*/*

**for**(int i **=** 0; i **<** 10; i**++**){

printf("%d\n", **\***(addr **+** i **\*** **sizeof**(int)));

}

*/\*打印\*/*

printf("Their second power values are:\n");

**for** (int i **=** 0; i **<** 10; i**++**){

printf("%ld\n",(long int)(**\***(addr **+** i **\*** **sizeof**(int)) **\*** (**\***(addr **+** i **\*** **sizeof**(int)))));

}

*/\* 共享存储区断开 \*/*

shmdt(addr);

*/\* 共享存储区释放 \*/*

shmctl(shid, IPC\_RMID, 0);

}

**else**{

*/\* 共享存储区的建立，得到存储区的描述符，开通读写权限\*/*

int shid **=** shmget(SHM\_KEY, SIZE, IPC\_CREAT **|** 0666);

*/\* 共享存储区的附接，得到存储区的地址\*/*

int \* addr **=** shmat(shid, 0, 0);

*/\* 信息录入\*/*

**for** (int i **=** 0; i **<** 10; i**++**){

printf("Please input the %d number", i);

scanf("%d", addr **+** i **\*** **sizeof**(int));

}

*/\* 将当前进程放回READY队列\*/*

sleep(0);

}

**return** 0;

}

*代码5-3-3结束*

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

运行结果如下所示

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

*代码5-3-4:代码5-3-3运行结果*

ubuntu@VM-0-6-ubuntu:~/os/05$ ./shm

Please input the 0 number5

Please input the 1 number23

Please input the 2 number46

Please input the 3 number78

Please input the 4 number321

Please input the 5 number321

Please input the 6 number4656

Please input the 7 number123

Please input the 8 number846

Please input the 9 number312

Number reveived are:

5

23

46

78

321

321

4656

123

846

312

Their second power values are:

25

529

2116

6084

103041

103041

21678336

15129

715716

97344

*代码5-3-4结束*

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

# 实验六 使用信号进行进程间通信

## 一、实验内容

1、学习任务。

学习补充材料中的UNIX信号及其描述。

2、使用信号进行进程间通信。

编写一个程序，完成下列功能：实现一个SIGINT、用户自定义信号的处理程序，注册该信号处理程序。主函数中创建一个子进程，令父子进程都进入等待状态。SIGINT、 用户自定义信号的处理程序完成的任务包括：（1）打印接收到的信号的编号（2）打印进程PID。编译并运行该程序，然后在键盘上敲Ctrl + C，观察出现的现象，并解释其含义。

## 二、实验过程

信号（signal）是UNIX提供的进程间通信与同步机制之一，用于通知进程发生了某个异步事件。信号与硬件中断相似，但不使用优先级。即，认为所有信号是平等的；同一时刻发生的多个信号，每次向进程提供一个，不会进行特别排序。 进程可以相互发送信号，内核也可以发出信号。信号的发送通过更新信号接收进程的进程表的特定域而实现。由于每个信号作为一个二进制位代表，同一类型的信号不能排队等待处理。 下面给出源代码

代码6-1 使用信号进行进程间通信

#include <stdio.h>

#include <signal.h>

*/\* Definition of action taken after SIGINT receives\*/*

**static** void sig\_int(int no){

printf("recieved SIGINT signal:%d\n", no);

printf("pid of this process is %d\n", getpid());

**return**;

}

*/\* Definition of action taken after SIGINT receives\*/*

**static** void sig\_usr(int no){

printf("received SIGUSR1 signal:%d\n", no);

printf("pid of this process is %d\n", getpid());

**return**;

}

int main(){

int pid **=** fork();

**if**(pid **==** 0){

*/\* This is the child\*/*

printf("Child's pid %d\n", getpid());

signal(SIGINT, sig\_int);

**for**(;;);

}

**else**{

printf("Father's pid %d\n", getpid());

signal(SIGUSR1, sig\_usr);

**for**(;;);

}

**return** 0;

}

代码6-1 结束

代码中使用到的signal可以指定接受到信号后执行的函数，其中前者为信号类型，后者为动作函数。运行结果如下所示：

ubuntu@VM-0-6-ubuntu:~/os/06$ ./a.out &

[1] 23735

ubuntu@VM-0-6-ubuntu:~/os/06$ Father's pid 23735

Child's pid 23736

ubuntu@VM-0-6-ubuntu:~/os/06$ kill -SIGUSR1 23735

ubuntu@VM-0-6-ubuntu:~/os/06$ received SIGUSR1 signal:10

pid of this process is 23735

kill -SIGINT 23736

ubuntu@VM-0-6-ubuntu:~/os/06$ recieved SIGINT signal:2

pid of this process is 23736

程序能够按照要求正确运行