操 作 系 统
实 验 报 告

实验名称：实验二 进程间通信和命令解释器

姓名：王迎旭

学号：16340226

实验名称：**进程间通信和命令解释器**

一、实验目的：

1. 进程间共享内存实验，初步了解这种进程间通讯

2. 实现简单的 shell 命令解释器：了解程序运行。

二、实验要求：

1. 进程间共享内存实验：完成课本第三章的练习 3.10 的程序

2. shell 的要求：参考书上第三章项目实现 shell。除此之外还需要实现程序的后台运行

三、实验过程：

（1）进程间共享内存实验

3.10 在习题 3.6 中，由于父进程和子进程都有它们自己的数据副本，子进程必须输出 Fibonacci 序列。设计此程序的另一个方法是在父进程和子进程之间建立一个共享内存段。此方法允许子进程将 Fibonacci 序列的内容写入共享内存段，当子进程完成时，父进程输出此序列。由于内存是共享的，每个子进程的变化都会影响到共享内存，也会影响到父进程。

这个程序将采用 3.5.1 小节介绍的 POSIX 共享内存方法来构建。程序首先需要创建共享内存段的数据结构，这可以通过利用 struct 来完成。此数据结构包括两项：（1）长度为 MAX_SEQUENCE 的固定长度数组，它保存 Fibonacci 的值；（2）子进程生成的序列的大小——sequence_size，其中 sequence_size≤MAX_SEQUENCE。这些项可表示成如下形式：

```
#define MAX_SEQUENCE 10

typedef struct {
```

```
        long fib_sequence [MAX_SEQUENCE];
        int sequence_size;
    } shared_data;
```

父进程将会按下列步骤进行：

a. 接受命令行上传递的参数，执行错误检查以保证参数不大于 MAX_SEQUENCE。
b. 创建一个大小为 shared_data 的共享内存段。
c. 将共享内存段附加到地址空间。
d. 在命令行将命令行参数值赋予 shared_data。
e. 创建子进程，调用系统调用 wait()等待子进程结束。
f. 输出共享内存段中 Fibonacci 序列的值。
g. 释放并删除共享内存段。

由于子进程是父进程的一个副本，共享内存区域也将被附加到子进程的地址空间。然后，子进程将会把 Fibonacci 序列写入共享内存并在最后释放此区域。

采用协作进程的一个问题涉及同步问题。在这个练习中，父进程和子进程必须是同步的，以使在子进程完成生成序列之前，父进程不会输出 Fibonacci 序列。采用系统调用 wait()，这两个进程将会同步。父进程将调用 wait()，这将使其被挂起，直到子进程退出。

1. 明确目标程序的需求：

①使用共享内存来为解决父进程与子进程中数据存放的问题

②使用子进程完成斐波拉契数列各项的计算，使用父进程输出斐波拉契数列各项的具体数值

2. 搜索题目所涉及相关资料：

Ⅰ、**了解 int main(int argc,char* argv[])函数中参数 argc 与*argv[]的用法 ：**

① argc：命令行总的参数的个数,即 argv 中元素的格式。

② * argv[ ]：字符串数组,用来存放指向你的字符串参数的指针数组，　每一个元素指向一个参数

Ⅱ、**了解共享内存模块相关资料**

① 创建或打开共享存储区(**shmget**)：依据用户给出的整数值 key，创建新区或打开现有区，返回一个共享存储区 ID。

② 连接共享存储区(**shmat**)：连接共享存储区到本进程的地址空间，返回共享存储区首地址。父进程已连接的共享存储区可被 fork 创建的子进程继承。

③ 拆除共享存储区连接(**shmdt**)：拆除共享存储区与本进程地址空间的连接。

④ 共享存储区控制(**shmctl**)：对共享存储区进行控制。如：共享存

储区的删除需要显式调用 shmctl(shmid, IPC_RMID, 0);

⑤ 用系统调用 **ftok** 给出 IPC 键值 key:保证同一个用户的两个进程获得相同的 IPC 键值 key

Ⅲ、了解 wait()函数以及 waitpid()函数的用法

① wait()

A、函数介绍:

wait()会暂时停止目前进程的执行,直到有信号来到或子进程结束。如果在调用 wait()时子进程已经结束，则 wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数 status 可以设成 NULL。子进程的结束状态值请参考 waitpid()。

B、返回值类型

如果执行成功则返回子进程识别码（PID），如果有错误发生则返回-1。失败原因存于 errno 中。

② waitpid()

A、函数介绍:

waitpid()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 waitpid()时子进程已经结束，则 waitpid()会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一快返回。如果不在意结束状态值，则参数 status 可以设成 NULL。

B、返回值类型

如果执行成功则返回子进程识别码（PID），如果有错误发生则返回-1。失败原因存于 errno 中。

3.斐波拉契数列的程序设计与运行

Ⅰ、针对斐波拉契数列数列理清设计思路

① 使用结构体构建共享内存模块

```c
#define  MAX_SEQUENCE  10
typedef  struct{
    long  fib_sequence[MAX_SEQUENCE];//存储斐波拉契数列的具体数值

    int  sequence_size;//限定子进程生成序列的大小

}shared_data;
```

② 创建共享内存块并开辟指针空间进行连接

```c
//创建共享进程编号
int segment_id ;
//求出共享内存的内存大小
int segment_size = sizeof(shared_data);
//创建共享内存指针
shared_data *shared_memory ;
//分配一个共享内存块
segment_id = shmget(IPC_PRIVATE, segment_size, S_IRUSR | S_IWUSR);
//连接一个共享内存块
shared_memory = (shared_data *) shmat(segment_id, NULL, 0);
//把运行程序时输入的参数赋给所申请的共享内存块
shared_memory -> sequence_size = atoi(argv[1]);
```

③ 创建子进程

```c
int pid;
pid = fork() ;
```

④ 父进程中设置 wait()函数等待子进程结束输出求和序列，子进程中完成对斐波拉契数列求和

```c
//在子进程中完成对斐波拉契数列的求和
if(pid == 0)
{
    //使用指针访问斐波拉契内存块
    shared_memory -> fib_sequence[0] = 0 ;
    shared_memory -> fib_sequence[1] = 1 ;
    if(atoi(argv[1]) > 2)
    {
        for(int i = 2 ; i < atoi(argv[1]) ; i ++ )
        {
            shared_memory ->  fib_sequence[i]  = shared_memory ->  fi
        }
    }
}
```

```
//在父进程中完成对斐波拉契数列的输出
else
{
    //父进程中调用wait函数等待程序结束
    wait(0) ;
    printf("The child process finished\n");
    printf("The Fibonacci sequence is:");
    for(int i = 0 ; i < atoi(argv[1]) ; i ++ )
    {
        printf("%d ", shared_memory -> fib_sequence[i] );
    }
    printf("\n");
}
```

⑤ 共享内存块的解除连接与释放空间

```
//解除内存块的连接
shmdt(shared_memory);
//清楚共享内存块
shmctl(segment_id, IPC_RMID, NULL);
```

Ⅱ、编译运行并输出结果

① 编译运行

```
[dell@localhost Desktop]$ gcc question1.c -o question1
[dell@localhost Desktop]$ ./question1 10
```

② 结果显示

```
Succeed to create the shared memory segment 1146891
The child process finished
The Fibonacci sequence is:0 1 1 2 3 5 8 13 21 34
[dell@localhost Desktop]$
```

分析:

A、在运行时候使用./question1 10 指令,这里是为了使 main 函数中的参数 argc = 2 以及 argv[1] = 10 ，这么做也是在程序运行之前直接给其传入运行参数，而非程序运行之后传入参数。

B、创建共享内存块 id 为 1146891 的共享内存块;随后父进程进入 wait()函数中，等待子进程完成；当子进程中赋值过程完成之后，子进程会 printf 一句话“The child process finished”，并结束；父进程完成对斐波拉契数列的输出；最后整个程序取消共享内存块的连接，释放共享内存。

(2) Shell 的实现

1、明确目标程序的需求：

① 通过 C 程序实现一个具备基本功能的 shell

② 实现程序的后台运行

2、搜索题目所涉及相关资料：

Ⅰ、execvp()函数

A、函数介绍

execvp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个参数 argv 传给该欲执行的文件。

B、返回值

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

Ⅱ、setup()函数

A、函数介绍

setup() 用于读入下一行输入的命令，并将它分成没有空格的命令和参数存于数组 args[]中，用 NULL 作为数组结束的标志。

B、返回值类型

无返回值

Ⅲ、信号捕捉函数 signal(int signum, sighandler_t handler)

A、函数介绍

设置对某一信号的捕捉。接收到一个类型为 sig 的信号时，就执行 handler 所指定的函数。（int）signum 是传递给它的唯一参数。执行了 signal()调用后，进程只要接收到类型为 sig 的信号，不管其正在执行程序的哪一部分，就立即执行 handler 函数。当 handler 函数执行结

束后，控制权返回进程被中断的那一点继续执行。

函数第一个参数 signum：指明了所要处理的信号类型，它可以取除了 SIGKILL 和 SIGSTOP 外的任何一种信号。

函数第二个参数 handler：描述了与信号关联的动作，它可以取以下三种值：

1)SIG_IGN:表示忽略该信号。

2)SIG_DFL:表示恢复对信号的系统默认处理,不写此处理函数默认也是执行系统默认操作。

3)sighandler_t 类型的函数指针，接收到 sig 信号之后进行函数调用

B、返回值：

返回先前的信号处理函数指针，如果有错误则返回 SIG_ERR(-1)。

Ⅲ、编写程序实现 shell

① 首先限定命令行最多能读入的字符个数

```
/*限定命令行最多80个字符*/
#define MAX_LINE 80
```

② 参考 PPT，实现 setup()函数（部分截图）

```
void setup(char inputBuffer[], char *args[],int *background)
{
    int length, /* 命令的字符数目 */
    i,      /* 循环变量 */
    start,  /* 命令的第一个字符位置 */
    ct;     /* 下一个参数存入args[]的位置 */

    ct = 0;
    start = -1;
    /* 读入命令行字符，存入inputBuffer */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);
```

③ 实现 main 主函数

注：由于一般程序使用 ctrl+c 就能强制程序退出，但是 shell 中需要使用 ctrl + d 指令完成程序退出功能，所以这里需要使用信

号捕捉函数对 ctrl + c 指令进行处理，让程序读到 ctrl + c 指令之后什么都不做。

具体步骤：

Ⅰ、设置 main 函数运行参数

```
char inputBuffer[MAX_LINE]; /* 这个缓存用来存放输入的命令*/
int background;             /* ==1时，表示在后台运行命令，即在命令后加上'&' */
char *args[MAX_LINE/2+1];/* 命令最多40个参数 */
```

Ⅱ、完成指令输入部分，并设置信号捕捉函数，处理 ctrl + c 导致进程结束的问题

```
background = 0;
printf(" COMMAND->"); //输出提示符，没有换行，仅将字符串送入输出缓存
fflush(stdout);       //若要输出输出缓存内容用fflush(stdout);头文件stdio.h
setup(inputBuffer,args,&background);       /* 获取下一个输入的命令 */
/*处理产生的ctrl c信号*/
signal(SIGINT,SIG_IGN);
```

Ⅲ、创建子进程，完成程序

```
pid_t fid = fork() ;
/*判断进程是否创建成功*/
if(pid < 0)
{
    printf("Create-Process Failed");
}
else if(pid == 0)
{
    execvp(args[0],args);
    exit(0);
}
else
{
    if(background == 0)
    {
        wait(NULL);
    }
    else
    {
        setup(inputBuffer,args,&background);
    }
}
```

Ⅳ、shell 程序的编译及运行

A、编译运行

```
[dell@localhost Desktop]$ gcc question2.c -o question2
[dell@localhost Desktop]$ ./question2
COMMAND->
```

B、运行结果显示

1) ls 指令：

```
COMMAND->ls
antstudy                    eclipse-installer   question2
Ant教程详解.pdf             HelloWorld          question2.c
a.text                      Junit               Sublime_text
chromium-browser.desktop    lxterminal.desktop  vi编辑器的使用.pdf
```

2) ctrl + c 指令

```
COMMAND->^C
COMMAND->
```

3) rm 指令

```
COMMAND->rm a.text
COMMAND->ls
antstudy                    HelloWorld          question2.c
Ant教程详解.pdf             Junit               Sublime_text
chromium-browser.desktop    lxterminal.desktop  vi编辑器的使用.pdf
eclipse-installer           question2
COMMAND->
```

分析：

① 通过使用 ls 指令，成功显示出 Desktop 上保留的文件夹

② 同时 ctrl + c 信号量也被成功捕捉，并且并没有造成程序退出

③ 使用 rm 指令之后，发现 a.text 被成功删除，证明 shell 程序的完整性

四、实验心得：

1.关于共享内存块的创建于使用部分，在参考了 PPT 所给的资料之后，完成起来并没有遇到太大的阻力，但是在使用指针的时候，还是有点不太熟练，就导致报错很多，但是语法上的错误还是慢慢的更正了过来。

2.关于创建 shell 并完成相应功能部分，这个部分应该是问题比较大的，因为初次使用 PPT 中所给的函数，遇到了一定的困难，比如 setup 函数的运行机制以及 execvp 函数在 shell 程序中的用途；同时在参考书本设置信号捕捉量也是遇到了一定的问题，比如如何设计指令让程序可以避免读到 ctrl + c 这个中断；虽然说问题比较多，但是还是花了时间把这些问题一一解决了，这次的实验也是让我对程序的进程运行有了更深的了解。