操作系统实验二 进程管理

班级：2017211314 学号：2017213508 学生：蒋雪枫

一、知识储备

在 C 语言中，我们可以在 main 函数中创建子进程，若进程创建成功，我们的程序将从原来一个主进程分叉为父子两个进程。两个进程同时运行于操作系统中，它们之间可以通过一些方式进行进程通信。在本书中，主要介绍有三种方式。首先是共享内存方式，用户进程空间一般是独立的，它们不能直接相互访问，所以这种方式便给它们提供了一段可直接访问的空间实现信息交换，当然操作系统也会为它们提供控制读写的同步互斥工具。其次是消息传递方式，如果进程之间不存在可以共享的内存空间，它们就只能使用操作系统提供的消息传递方式进行通信了，该方式分为两种，其一是通过进程自身的消息缓冲队列传递，其二是通过某个中间实体实现消息传递(比如电子邮件系统)。而管道通信是借助管道实现通信的方式。所谓管道，是指存在于内存中，用于连接一个读与另一个写进程的共享文件，又名 pipe 文件。在 Linux 中，管道机制得到了广泛运用，也有自身的实现机制与使用规定。

Collatz 猜想：任意写出一个正整数 N，并且按照以下的规律进行变换：如果是个奇数，则下一步变成 3N+1；如果是个偶数，则下一步变成 N/2。无论 N 是怎样的一个数字，最终都无法逃脱回到谷底 1。例如：如果 N=35，则有序列 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1。

二、实验任务

本次实验的任务是编写 C 语言进程管理有关程序，结合实例在实践中研究父子进程的运行时情况、进程之间通过共享内存和管道这种特殊的内存文件实现通信。

三、实验环境

基于 Linux 的 Ubuntu 虚拟机映像环境、GCC

四、实验过程与分析

1.实验内容一：

采用系统调用 fork()，编写一个 C 程序，以便在子进程中生成这个序列。

要求：

（1）从命令行提供启动数字

实现方式：通过 int main(int argc,char** argv)从命令行输入启动数字 n，并通过 stdlib.h 的 atoi 函数实现字符串转整数。

（2）由子进程输出数字序列

实现方式：通过 pid 判断是否是子进程，如果是，则在在该程序段输出这一序列。

（3）父进程等子进程结束后再退出。

实现方式：wait(NULL)实现让父进程等待子进程结束后再退出，运行程序的结果也说明了这一点。(通过打印 pid 可知，第一个“end in here”来自子进程)

```
jxf@2BB3NCKF8U1RJ2A:~$ cd /mnt
jxf@2BB3NCKF8U1RJ2A:/mnt$ cd d
jxf@2BB3NCKF8U1RJ2A:/mnt/d$ cd zjj
jxf@2BB3NCKF8U1RJ2A:/mnt/d/zjj$ ls
exp1  exp1.c  exp2  exp2.c  exp3  exp3.c  hello  hello.c
jxf@2BB3NCKF8U1RJ2A:/mnt/d/zjj$ gcc exp1.c -o exp1
exp1.c: In function ‘main’:
exp1.c:30:9: warning: implicit declaration of function ‘wait’; did you mean ‘main’? [-Wim
plicit-function-declaration]
         wait(NULL);
         ^~~~
         main
jxf@2BB3NCKF8U1RJ2A:/mnt/d/zjj$ ./exp1 10
10
5
16
8
4
2
1
Main process ends in here
Main process ends in here
jxf@2BB3NCKF8U1RJ2A:/mnt/d/zjj$
```

```c
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(int argc,char** argv)
{
    //命令行输入参数n
    int n;
    n=atoi(argv[1]);
    pid_t pid;
    pid=fork();
    if(pid==0)//child process
    {
        while(n!=1)
        {
            printf("%d\n",n);
            if(n&1==1)
                n=3*n+1;
            else
                n=n/2;
        }
        printf("1\n");
    }
    else if (pid<0)
    { // error occurred
        printf("Fork failed\n");
    }
    else //father process
    {
        wait(NULL);
    }
    printf("Main process ends in here\n");
}
```

2.实验内容二：

以共享内存技术编程实现 Collatz 猜想。要求在父子进程之间建立一个共享内存对象，允许子进程将序列内容写入共享内存对象，当子进程完成时，父进程输出序列。父进程包括如下步骤：1）建立共享内存对象（shm_open(), ftruncate(), mmap()）2）建立子进程并等待他终止 3）输出共享内存的内容 4）删除共享内存对象。

另外，通过查阅有关材料可知，使用 system（“ipcs -m”）可以查看共享内存的情况。下面解释源代码中部分函数的原型与用途，运行结果和代码如下附上。

1、int shmget(key_t key, size_t size, int shmflg);

第一个参数，与信号量的 semget 函数一样，程序需要提供一个参数 key（非 0 整数），它有效地为共享内存段命名，shmget()函数成功时返回一个与 key 相关的共享内存标识符（非负整数）。第二个参数，size 以字节为单位指定需要共享的内存容量。第三个参数，shmflg 是权限标志，它的作用与 open 函数的 mode 参数一样。共享内存的权限标志与文件的读写权限一样，举例来说，0644,它表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。

2、perror(str) 用来将上一个函数发生错误的原因输出到标准设备(stderr)。

3、void *shmat(int shm_id, const void *shm_addr, int shmflg);

第一个参数，shm_id 是由 shmget()函数返回的共享内存标识。第二个参数，shm_addr 指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。第三个参数，shm_flg 是一组标志位，通常为 0。调用成功时返回一个指向共享内存第一个字节的指针，如果调用失败返回-1。

4、int shmctl(int shm_id, int command, struct shmid_ds *buf);

第一个参数，shm_id 是 shmget()函数返回的共享内存标识符。第二个参数，command 是要采取的操作，它可以取下面的三个值 ：IPC_STAT：把 shmid_ds 结构中的数据设置为共享内存的当前关联值，即用共享内存的当前关联值覆盖 shmid_ds 的值；IPC_SET：如果进程有足够的权限，就把共享内存的当前关联值设置为 shmid_ds 结构中给出的值；IPC_RMID：删除共享内存段。第三个参数，buf 是一个结构指针，它指向共享内存模式和访问权限的结构。

5、int shmdt(const void *shmaddr);

该函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。

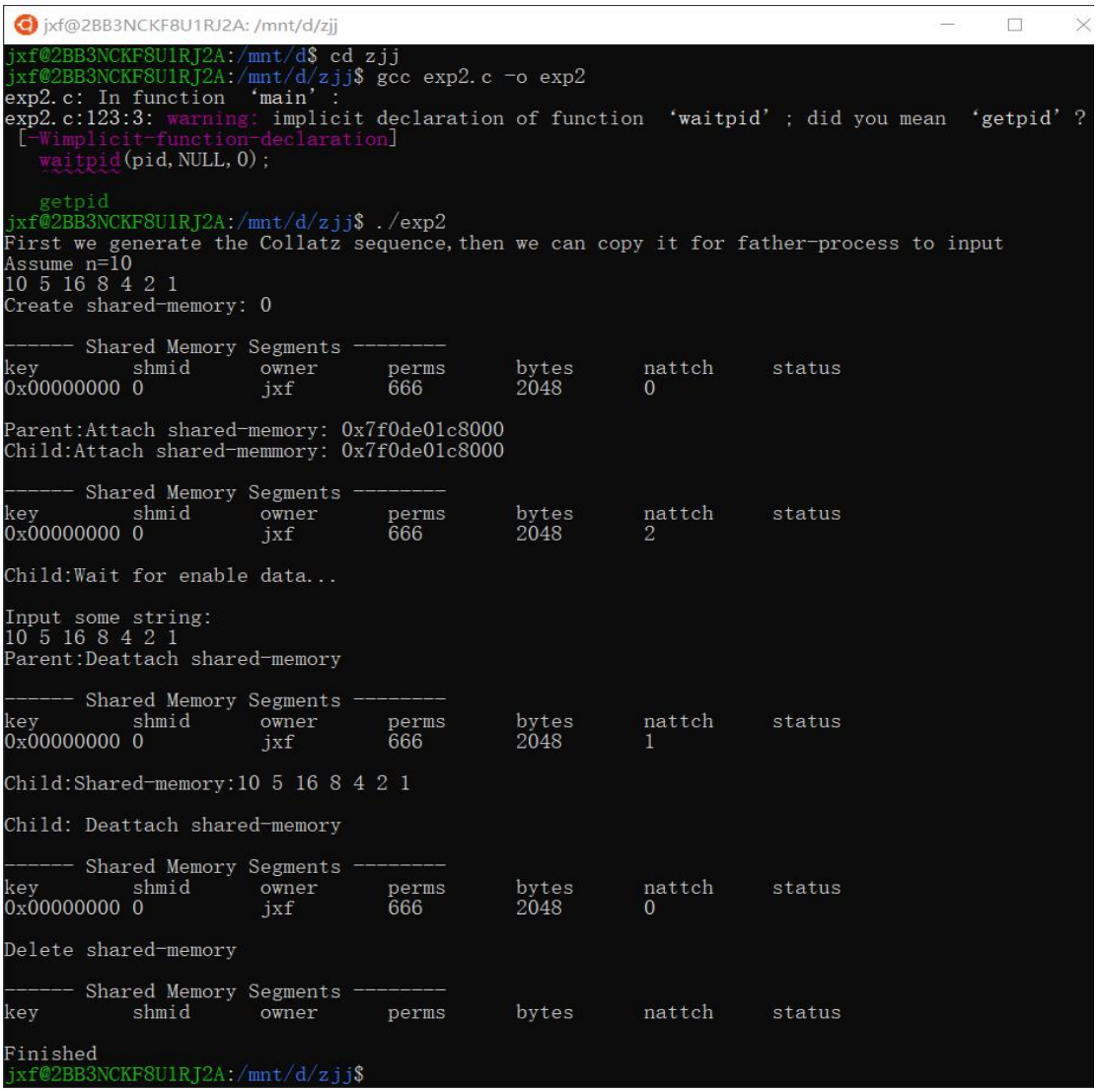
6、ipcs 的用法

ipcs -a 是默认的输出信息 打印出当前系统中所有的进程间通信方式的信息

ipcs -m 打印出使用共享内存进行进程间通信的信息

ipcs -q 打印出使用消息队列进行进程间通信的信息

ipcs -s 打印出使用信号进行进程间通信的信息




```c
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define BUFFER_SIZE 2048
int main()
{
    printf("First we generate the Collatz sequence,then we can copy it for father-process to input\nAssume n=10\n");
    printf("10 5 16 8 4 2 1\n");

    pid_t pid;
```

```c
int shmid;/*要映射的共享内存区标识符*/
char *shm_addr=NULL;
char flag[]="WROTE"; //标志字符串
char buff[20];//定义 20 字节大小

/*创建共享内存*/
if((shmid=shmget(IPC_PRIVATE,BUFFER_SIZE,0666))<0)
{
    perror("shmget");
    exit(1);
}
else
{
    printf("Create shared-memory: %d\n",shmid);//内存标识符
}
/*显示共享内存情况*/
system("ipcs -m");
pid=fork();//创建子进程
if(pid==-1)
{
    perror("fork");
    exit(1);
}
else if(pid==0) //子进程处理
{
    /*映射共享内存*/
    if((shm_addr=shmat(shmid,0,0))==(char *)-1)
    {
        perror("Child: shmat");
        exit(1);
    }
    else
    {
        printf("Child:Attach shared-memmory: %p\n",shm_addr);
    }
    system("ipcs -m"); //执行命令: ipcs -m
    /*通过检查在共享内存的头部是否有标志字符串“WROTE”来确认父进程
    已经向共享内存写入有效数据*/
    while(strncmp(shm_addr,flag,strlen(flag)))
    {
       printf("Child:Wait for enable data...\n");
       sleep(50);
    }
```

```
        /*共享内存的有效数据显示*/
        strcpy(buff,shm_addr+strlen(flag));
        printf("Child:Shared-memory:%s\n",buff);
        /*解除共享内存映射*/
        if((shmdt(shm_addr))<0)
        {
            perror("shmdt");
            exit(1);
        }
        else
        {
            printf("Child: Deattach shared-memory\n");
        }
        system("ipcs -m");//报告此时进程间的通信机制状态
        /*删除共享内存*/
        if(shmctl(shmid,IPC_RMID,NULL)==-1)
        {
            perror("Child:shmctl(IPC_RMID)\n");
            exit(1);
        }
        else
        {
            printf("Delete shared-memory\n");
        }
        system("ipcs -m");

    } /*子进程处理程序结束*/
    else /*父进程处理*/
    {
        /*映射共享内存*/
        if((shm_addr=shmat(shmid,0,0))==(char *)-1)
        {
            perror("Parent: shmat");
            exit(1);
        }
        else
        {
            printf("Parent:Attach shared-memory: %p\n",shm_addr);
        }
        sleep(1);

        printf("\nInput some string:\n");
        fgets(buff,BUFFER_SIZE,stdin);
        strncpy(shm_addr+strlen(flag),buff,strlen(buff));
```

```
        strncpy(shm_addr,flag,strlen(flag));

        /*解除共享内存映射*/
        if((shmdt(shm_addr))<0)
        {
            perror("Parent:shmdt");
            exit(1);
        }
        else
        {
            printf("Parent:Deattach shared-memory\n");
        }
        system("ipcs -m");
        waitpid(pid,NULL,0);
        printf("Finished\n");
    } /*父进程处理结束*/
    exit(0);
}
```

3.实验内容三：普通管道通信

设计一个程序，通过普通管道进行通信，让一个进程发送一个字符串消息给第二个进程，第二个进程收到此消息后，变更字母的大小写，然后再发送给第一个进程。比如，第一个进程发消息：“I am Here”，第二个进程收到后，将它改变为：“i AM hERE”之后，再发给第一个进程。

提示：

（1）需要创建子进程，父子进程之间通过普通管道进行通信。

（2）需要建立两个普通管道。

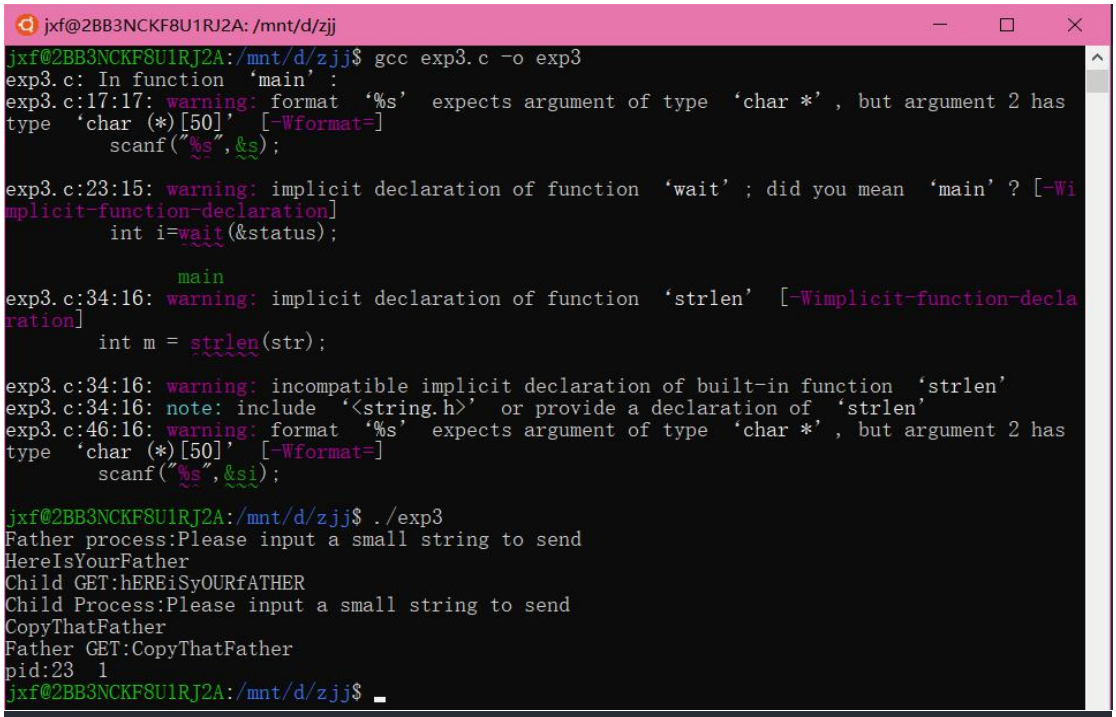
注意：

在这里创建字符串的时候，不可用 char* str=...，原因学生不是很清楚，但是如果这么处理运行后子进程最后在命令行全部会输出满屏幕的 0。

管道与 Linux C 的管道原型：

管道是由内核管理的一个缓冲区，相当于我们放入内存中的一个纸条。管道的一端连接一个进程的输出。这个进程会向管道中放入信息。管道的另一端连接一个进程的输入，这个进程取出被放入管道的信息。一个缓冲区不需要很大，它被设计成为环形的数据结构，以便管道可以被循环利用。当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试放入信息的进程会等待，直到另一端的进程取出信息。读写是阻塞的。当两个进程都终结的时候，管道也自动消失。

创建管道需要用到#include<unistd.h>里的 int pipe(int filedes[2]);语句。其返回值：成功，返回 0，否则返回-1。参数数组包含 pipe 使用的两个文件的描述符。fd[0]:读管道，fd[1]:写管道。该头文件下，write()会把参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。read()会把参数 fd 所指的文件传送 count 个字节到 buf 指针所指的内存中。




```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[])
{
    int fd1[2],fd2[2];
    pipe(fd1);
    pipe(fd2);
    int pid=fork();
    if(pid>0)
{
    close(fd1[0]);
    close(fd2[1]);
    char str[50];
    printf("Father process:Please input a small string to send\n");
    char s[50];
    scanf("%s",&s);
    write(fd1[1],s,50);
    read(fd2[0],str,50);
    printf("Father GET:%s\n",str);
    int status;
    int i=wait(&status);
    printf("pid:%d\t%d\n",i,WIFEXITED(status));
    exit(0);
}
    else if(pid==0)
{
```

```
    close(fd1[1]);
    close(fd2[0]);
    char str[50];
    read(fd1[0],str,50);
//process BIG-SMALL casefold
    int m = strlen(str);
    for(int i = 0; i < m; i++)
    {
    if(str[i]>='A'&&str[i]<='Z')
    str[i] = str[i]+32;
    else if(str[i]>='a'&&str[i]<='z')
    str[i] = str[i]-32;
    }
    //printf
    printf("Child GET:%s\n",str);
    printf("Child Process:Please input a small string to send\n");
    char si[50];
    scanf("%s",&si);
    write(fd2[1],si,50);
    }
    return 0;
}
```