实验一、进程控制实验

1.1 实验目的

加深对于进程并发执行概念的理解。实践并发进/线程的创建和控制方法。观察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习 Linux 系统中进/线程创建与控制有关的系统调用的编程和调试技术。

1.2 实验说明

1）与进程创建、执行有关的系统调用说明

进程可以通过系统调用 fork() 创建子进程并和其子进程并发执行。子进程初始的执行映像是父进程的一个复本。子进程可以通过 exec() 系统调用族装入一个新的执行程序。父进程可以使用 wait() 或 waitpid() 系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

fork() 系统调用语法:

```
#include <unistd.h>
pid_t fork(void);
```

fork 成功创建子进程后将返回子进程的进程号，不成功会返回 -1。

exec() 系统调用

有一组 6 个函数,其中示例实验中引用了 execve 系统调用语法：

```
#include <unistd.h>
int execve(const char *path, const char *argv[], const char * envp[]);
```

- path 要装入的新的执行文件的绝对路径名字符串。
- argv[] 要传递给新执行程序的完整的命令参数列表(可以为空)。
- envp[] 要传递给新执行程序的完整的环境变量参数列表(可以为空)。

Exec 执行成功后将用一个新的程序代替原进程，但进程号不变，它绝不会再返回到调用进程了。如果 exec 调用失败，它会返回 -1。

wait() 系统调用语法:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid,int *status,int option);
```

1. status 用于保留子进程的退出状态

2. pid 可以为以下可能值:

- -1 等待所有 PGID 等于 PID 的绝对值的子进程
- 1 等待所有子进程
- 0 等待所有 PGID 等于调用进程的子进程
- > 0 等待 PID 等于 pid 的子进程

3. option 规定了调用 waitpid 进程的行为:

- WNOHANG 没有子进程时立即返回
- WUNTRACED 没有报告状态的进程时返回

wait 和 waitpid 执行成功将返回终止的子进程的进程号，不成功返回 -1。

getpid() 系统调用语法:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

getpid 返回当前进程的进程号，getppid 返回当前进程父进程的进程号。

2）与进程控制有关的系统调用说明

可以通过信号向一个进程发送消息以控制进程的行为。信号是由中断或异常事件引发的，如: 键盘中断、定时器中断、非法内存引用等。信号的名字都以 SIG 开头，例如 SIGTERM、SIGHUP。可以使用 `kill -l` 命令查看系统当前的信号集合。 信号可在任何时间发生，接收信号的进程可以对接收到的信号采取 3 种处理措施之一

- 忽略这个信号
- 执行系统默认的处理
- 捕捉这个信号做自定义的处理

信号从产生到被处理所经过的过程：产生 (generate) -> 挂起 (pending) -> 派送 (deliver) -> 部署 (disposition) 或忽略 (igore）。

一个信号集合是一个 C 语言的 sigset_t 数据类型的对象，sigset_t 数据类 型定义在 signal.h 中。被一个进程忽略的所有信号的集合称为一个信号掩 码(mask)。

从程序中向一个进程发送信号有两种方法:调用 shell 的 kill 命令，调用 kill 系统调用函数。kill 能够发送除杀死一个进程 (SIGKILL、SIGTERM、 SIGQUIT) 之外的其他信号，例如键盘中断 (Ctrl+C) 信号 SIGINT，进程暂停 (Ctrl+Z) 信号 SIGTSTP 等等。

调用 pause 函数会令调用进程的执行挂起直到一个任意信号到来后再继续运行。

调用 sleep 函数会令调用进程的执行挂起睡眠指定的秒数或一个它可以响应的信号到来后继续执行。

每个进程都能使用 signal 函数定义自己的信号处理函数，捕捉并自行处理接收的除 SIGSTOP 和 SIGKILL 之外的信号。以下是有关的系统调用的语法说明。

kill() 系统调用语法:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

pid 接收信号的进程号，signal 要发送的信号。kill 发送成功返回接收者的进程号，失败返回 -1。

pause() 系统调用语法:

```
#include <unistd.h>
int pause(void);
```

pause 挂起调用它的进程直到有任何信号到达。调用进程但不自定义处理方法，则进行信号的默认处理。只有进程自定义了信号处理方法，捕获并处理了一个信号后，pause 才会返回调进程。pause 总是返回 -1，并设置系统系量 errno 为 EINTR。

sleep() 系统调用语法:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

seconds 指定进程睡眠的秒数。如果指定的秒数到，sleep 返回 0。

signal() 系统调用语法为:

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

signum 要捕捉的信号，handler 进程中自定义的信号处理函数名。signal 调用成功会返回信号处理函数的返回值，不成功返回 -1，并设置系统变量 errno 为 SIG_ERR。

1.3 进程示例实验

以下实验示例程序应实现一个类似子 shell 子命令的功能,它可以从执行程序中启动另一个新的子进程并执行一个新的命令和其并发执行。

1. 打开一终端命令行窗体，新建一个文件夹，在该文件夹中建立以下名为 pctl.c 的 C 语言程序:

```
/*
* Filename: pctl.c
* copyright : (C) 2006 by zhanghonglie
* Function: 父子进程的并发执行
*/
#include "pctl.h"
int main(int argc, char *argv[]) {
```

```
    // 如果在命令行上没输入子进程要执行的命令，则执行缺省的命令
    int i;
    int pid; // 存放子进程号
    int status; // 存放子进程返回状态
    char *args[] = {"/bin/ls", "-a", NULL}; // 子进程执行缺省的命令

    signal(SIGINT, (sighandler_t)sigcat); // 注册一个本进程处理中断信号的函数
    pid = fork() ; // 建立子进程，成功则返回子进程号，否则返回 -1
    if (pid < 0) { // 建立子进程失败
        printf("Create Process failed!\n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) { // 子进程执行代码段
        // 报告父子进程进程号
        printf("I am Child process %d, my Father is %d\n", getpid(), getppid());
        // 唤醒父进程
        if (argv[1] == NULL && kill(getppid(), SIGINT) >= 0)
            printf("Child %d Wakeup Father %d\n", getpid(), getppid());
        // 暂停，等待中断信号唤醒
        pause();
        // 子进程被键盘中断信号唤醒继续执行
        printf("Child %d is Running: \n", getpid());
        if (argv[1] != NULL) {
            // 如果在命令行上输入了子进程要执行的命令，则执行输入的命令
            for (i = 1; argv[i] != NULL; i++)
                printf("%s ", argv[i]); printf("\n");
            // 装入并执行新的程序
            status = execve(argv[1], &argv[1], NULL);
        } else {
            // 如果在命令行上没输入子进程要执行的命令，则执行缺省的命令
            for (i = 0; args[i] != NULL; i++)
                printf("%s ", args[i]); printf("\n");
            // 装入并执行新的程序
            status = execve(args[0], args, NULL);
        }
    } else { // 父进程执行代码段
        // 报告父进程进程号
        printf("I am Parent process %d\n", getpid());
        if (argv[1] != NULL) {
            // 如果在命令行上输入了子进程要执行的命令，则父进程等待子进程执行结束
            printf("Parent %d is Waiting for Child done\n\n", getpid());
            // 等待子进程结束
            waitpid(pid, &status, 0);
            printf("\nMy Child Exit with status = %d\n\n", status);
        } else {
            // 暂停，等待中断信号唤醒
            pause();
            // 唤醒子进程，与子进程并发执行不等待子进程执行结束
            if (kill(pid, SIGINT) >= 0)
```

```
                printf("\nParent %d Wakeup Child %d\n", getpid(), pid);
            printf("Parent %d don't Wait for Child done\n\n", getpid());
        }
    }
    return EXIT_SUCCESS;
}
```

2. 再建立以下名为 pctl.h 的 C 语言头文件:

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h> // 进程自定义的键盘中断信号处理函数
typedef void (*sighandler_t) (int);
void sigcat() {
  printf("%d Process continue\n", getpid());
}
```

3. 建立 Makefile 文件

```
head = pctl.h
srcs = pctl.c
objs = pctl.o
opts = -g -c
all: pctl
pctl: $(objs)
    gcc $(objs) -o pctl
pctl.o: $(srcs) $(head)
    gcc $(opts) $(srcs)
clean:
    rm pctl *.o
```

4. 输入 make 命令编译连接生成可执行的 pctl 程序

```
$ make
gcc -g -c pctl.c
gcc pctl.o -o pctl
```

5. 执行 pctl 程序（注意进程号是动态产生的，每次执行都不相同）

```
$ ./pctl
I am Parent process 5842
I am Child process 5843, my Father is 5842
Child 5843 Wakeup Father 5842
5842 Process continue
```

```
Parent 5842 Wakeup Child 5843
Parent 5842 don't Wait for Child done

5843 Process continue
Child 5843 is Running:
/bin/ls -a
. .. Makefile pctl pctl.c pctl.h pctl.o
$
```

根据上述程序输出，可以得知如下执行过程：

1. 父进程 5842 创建了一个子进程 5843
2. 父进程执行被暂停
3. 子进程唤醒父进程
4. 子进程执行被暂停
5. 父进程唤醒子进程
6. 父进程继续执行并先行结束（此时的子进程成为了孤儿进程，不会再有父进程为其清理退出状态）
7. 子进程继续执行，并执行命令 `ls -a`
8. 完成 `ls -a` 命令后，子进程执行结束

由于子进程和父进程是并发执行的，因此并非只有上述一种运行结果。若遇到其他运行结果，可以观察其输出分析其执行过程。

6. 再次执行带有子进程指定执行命令的 pctl 程序：

```
$ ./pctl /bin/ls -l
I am Parent process 4222
Parent 4222 is Waiting for Child done.

I am Child process 4223, my Father is 4222
```

可以看到这一次子进程仍然被挂起，而父进程则在等待子进程的完成。为了检测父子进程是否都在并发执行，请输入 ctrl+z 将当前进程放入后台并输入 ps 命令查看当前系统进程信息，显示如下：

```
[1]+  Stopped ./pctl /bin/ls -l
$ ps -l
F S    UID PID    PPID    C PRI    NI ADDR SZ WCHAN    TTY  TIME  CMD
0 S 0 4085 4083 0 76  0 - 1413 wait pts/1 00:00:00 bash
0 T 0 4222 4085 0 76  0 - 360 finish  pts/1 00:00:00 pctl
1 T 0 4223 4222 0 76  0 - 360 finish  pts/1 00:00:00 pctl
0 R 0 4231 4085 0 78  0 - 1302 -  pts/1 00:00:00 ps
```

可以看到当前系统中同时有两个叫 pctl 的进程，它们的进程号分别是 4222 和 4223。它们的状态都为 T，说明当前都被挂起。4223 的父进程是 4222， 而 4222 的父进程是 4085，也就是 bash-shell。为了让 pctl 父子进程继续执行，请输入 fg 命令让 pctl 再次返回前台，显示如下：

```
$ fg
./pctl /bin/ls -l
```

现在pctl父子进程从新返回前台。我们可以通过键盘发键盘中断信号来唤醒pctl 父子进程继续执行，输入ctrl+c，将会显示：

```
4222 Process continue
4223 Process continue
Child 4223 is Running:
/bin/ls -l
total 1708
-rw-r--r-- 1 root root  176 May 8 11:11 Makefile
-rwxr-xr-x 1 root root  8095 May  8 14:08 pctl
-rw-r--r-- 1 root root  2171 May  8 14:08 pctl.c
-rw-r--r-- 1 root root  269 May 8 11:10 pctl.h
-rw-r--r-- 1 root root  4156 May  8 14:08 pctl.o

My Child Exit with status = 0
```

以上输出说明了子进程在捕捉到键盘中断信号后继续执行了指定的命令，按我们要求的长格式列出了当前目录中的文件名，父进程在接收到子进程执行结束的信号后将清理子进程的退出状态并继续执行，它报告了子进程的退出编码（0 表示子进程正常结束），然后父进程结束执行。

1.4 独立实验

参考以上示例程序中建立并发进程的方法，编写一个多进程并发执行程序。父进程建立两个子进程，首先创建的让其执行 ls 命令，之后创建执行让其执行 ps 命令，并控制 ps 命令总在 ls 命令之前执行。要求父进程等待两个子进程结束。

1.5 实验要求

根据实验中观察和记录的信息结合示例实验和独立实验程序，思考以下问题：

- 它们反映出操作系统教材中进程及处理机管理一节讲解的进程的哪些特征和功能?
- 真实的操作系统是怎样实现和反映出教材中讲解的进程的生命期、进程的实体和进程状态控制的?
- 你对于进程概念和并发概念有哪些新的理解和认识?
- 子进程是如何创建和执行新程序的?
- 信号的机理是什么?
- 怎样利用信号实现进程控制?

根据实验程序、调试过程和结果分析写出实验报告。

实验二、线程和管道通信实验

2.1 实验目的

通过 Linux 系统中线程和管道通信机制的实验，加深对于线程控制和管道通信概念的理解，观察和体验并发进 / 线程间的通信和协作的效果 ，练习利用无名管道进行进 / 线程程通信的编程和调试技术。

2.2 实验说明

1）与线程创建、执行有关的系统调用说明

线程是在共享内存中并发执行的多道执行路径，它们共享一个进程的资源，如进程程序段、文件描述符和信号等，但有各自的执行路径和堆栈。线程的创建无需像进程那样重新申请系统资源，线程在上下文切换时也无需像进程那样更换内存映像。多线程的并发执行即避免了多进程并发的上下文切换的开销又可以提高并发处理的效率。

Linux 利用了特有的内核函数 clone 实现了一个叫 phread 的线程库，clone 是 fork 函数的替代函数，通过更多的控制父子进程共享哪些资源而实现了线程。phread 是一个标准化模型，用它可把一个程序分成一组能够并发执行的多个任务。phread 线程库是 POSIX 线程标准的实现，它提供了 C 函数的线程调用接口和数据结构。

线程可能的应用场合包括：

- 在返回前阻塞的 I/O 任务能够使用一个线程处理 I/O，同时继续执行其他处理。
- 需要及时响应多个前台用户界面操作同时后台处理的多任务场合。
- 在一个或多个任务受不确定事件影响时能够处理异步事件同时继续进行正常处理。
- 如果某些程序功能比其他功能更重要，可以使用线程以保证所有功能都出现，但那些时间密集型的功能具有更高优先级。

下面介绍 pthread 库中最基本的调用。

pthread_create() 系统调用语法：

```
#include <pthread.h>
Int pthread_create(pthread_t *thread,pthread_attr_t *attr,
void *(*start_routine)(void *) Void *arg);
```

pthread_create 函数创建一个新的线程。

pthread_create 在 thread 中保存新线程的标识符。attr 决定了线程应用那种线程属性。使用默认可给定参数 NULL；start_routine 是一个指向新线程中要执行的函数的指针，最后一个参数 arg 是新线程函数携带的参数。

Pthread_create 执行成功会返回 0 并在 thread 中保存线程标识符，执行失败则返回一个非 0 的出错代码。

pthread_exit() 系统调用语法：

```
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_exit 函数使用函数 pthread_cleanup_push 调用任何用于该线程的清除处理函数，然后中止当前进程的执行，返回 retval。retval 可以由父线程或其他线程通过 pthread_join 来检索。一个线程也可以简单地通过从其初始化函数返回来终止。

pthread_join() 系统调用语法：

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
int pthread_detach(pthread_t th);
```

函数 pthread_join 用于挂起当前线程，直到 th 指定的线程终止运行为止。

另一个线程的返回值如果不为 NULL，则保存在 thread_return 指向的地址中。一个线程所使用的内存资源在对该线程应用 pthread_join 调用之前不会被重新分配。因而对于每个可切入的线程（默认的）必须调用一次 pthread_join 函数。线程必须是可切入的而不是被分离的状态，并且其他线程不能对同一线程再应用 pthread_join 调用。

通过在 pthread_create 调用创建一个线程时使用 PTHREAD_CREATE_DETACHED 属性或者使用 pthread_detach 可以让线程处于被分离状态。

注意不像由 fork 创建的进程可以使用众多 wait 等待子进程退出，在 pthread 多线程中似乎没有等待某个线程退出的方法。

2）管道通信机制

管道 pipe 是进程间通信最基本的一种机制。在内存中建立的管道称为无名管道， 在磁盘上建立的管道称为有名管道。无名管道随着进程的撤消而消失，有名管道则可以长久保存，shell 命令符 | 建立的就是无名管道，而 shell 命令 mkfifo 建立的是有名管道。两个进程可以通过管道一个在管道一端向管道发送其输出,给另一进程可以在管道的另一端从管道得到其输入。

管道以半双工方式工作，即它的数据流是单方向的。因此使用一个管道一般的规则是读管道数据的进程关闭管道写入端，而写管道进程关闭其读出端。管道既可以采用同步方式工作也可以采用异步方式工作。

pipe() 系统调用的语法为:

```
#include <unistd.h>
int pipe(int pipe_id[2]);
```

pipe 建立一个无名管道，pipe_id[0] 和 pipe_id[1] 中将放入管道两端的描述符。如果 pipe 执行成功返回 0，出错返回 -1。

管道读写的系统调用语法为：

```
#include <unistd.h>
ssize_t read(int pipe_id,const void *buf,size_t count);
ssize_t write(int pipe_id,const void *buf,size_t count);
```

read 和 write 分别在管道的两端进行读和写。

pipe_id 是 pipe 系统调用返回的管道描述符。Buf 是数据缓冲区首地址，count 说明数据缓冲区以 size_t 为单位的长度。read 和 write 的返回值为它们实际读写的数据单位。

注意管道的读写默认的通信方式为同步读写方式，即如果管道读端无数据则读者阻塞直到数据到达，反之如果管道写端缓冲区数据已满则写者阻塞直到数据被读走。

2.3 示例实验

1）两个并发线程

以下示例实验程序实现并发的两个线程合作将整数 X 的值从 1 加到 10 的功能。它们通过管道相互将计算结果发给对方。

1. 在新建文件夹中建立以下名为 tpipe.c 的 C 语言程序

```
/*
* main.c  :description
* copyright : (C) by zhanghonglie
* Function  : 利用管道实现在在线程间传递整数
*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
void task1(int *); // 线程 1 执行函数原型
void task2(int *); // 线程 2 执行函数原型
int pipe1[2], pipe2[2]; // 存放的两个无名管道标号
pthread_t thrd1, thrd2; // 存放的两个线程标识

int main(int argc, char *arg[]) {
    int ret;
    int num1, num2;
    // 使用 pipe() 系统调用建立两个无名管道。建立不成功程序退出，执行终止
    if (pipe(pipe1) < 0) {
        perror("pipe1 not create");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipe2) < 0) {
        perror("pipe2 not create");
        exit(EXIT_FAILURE);
    }
    // 使用 pthread_create 系统调用建立两个线程。建立不成功程序退出，执行终止
    num1 = 1;
    ret = pthread_create(&thrd1, NULL, (void *) task1, (void *) &num1);
    if (ret) {
        perror("pthread_create: task1");
        exit(EXIT_FAILURE);
    }

    num2 = 2;
    ret = pthread_create(&thrd2, NULL, (void *) task2, (void *) &num2);
    if (ret) {
```

```
        perror("pthread_create: task2"); exit(EXIT_FAILURE);
    }
    // 挂起当前线程切换到 thrd2 线程
    pthread_join(thrd2, NULL);
    // 挂起当前线程切换到 thrd1 线程
    pthread_join(thrd1, NULL);
    exit(EXIT_SUCCESS);
}
// 线程 1 执行函数，它首先向管道写，然后从管道读
void task1(int *num) {
    int x = 1;
    // 每次循环向管道 1 的 1 端写入变量 X 的值
    // 并从管道 2 的 0 端读一整数写入 X 再对 X 加 1，直到 X 大于 10
    do {
        printf("thread%d read: %d\n", *num, x++);
        write(pipe1[1], &x, sizeof(int));
        read(pipe2[0], &x, sizeof(int));
    } while (x <= 9);

    // 读写完成后，关闭管道
    close(pipe1[1]);
    close(pipe2[0]);
}
// 线程 2 执行函数，它首先从管道读，然后向管道写
void task2(int *num) {
    int x;
    // 每次循环从管道 1 的 0 端读一个整数放入变量 X 中
    // 并对 X 加 1 后写入管道 2 的 1 端，直到 X 大于 10
    do {
        read(pipe1[0], &x, sizeof(int));
        printf("thread2 read: %d\n", x++);
        write(pipe2[1], &x, sizeof(int));
    } while (x <= 9);

    // 读写完成后，关闭管道
    close(pipe1[0]);
    close(pipe2[1]);
}
```

2. 再建立程序的 Makefile 文件：

```
src = tpipe.c
obj = tpipe.o
opt = -g -c

all:  tpipe
tpipe:  $(obj)
        gcc $(obj) -l pthread -o tpipe
tpipe.o:  $(src)
        gcc $(opt) $(src)
clean:
          rm tpipe  *.o
```

3. 使用 make 命令编译连接生成可执行文件 tpipe:

```
$ make
gcc -g -c tpipe.c
gcc tpipe.o -l pthread -o tpipe
```

4. 编译成功后执行 tpipe:命令:

```
$ ./tpipe
thread1 read: 1
thread2 read: 2
thread1 read: 3
thread2 read: 4
thread1 read: 5
thread2 read: 6
thread1 read: 7
thread2 read: 8
thread1 read: 9
thread2 read: 10
```

可以看到以上程序的执行中线程 1 和线程 2 交替的将整数X 的值从 1 加到了 10。

2）并发父子线程

以下示例实验程序实现并发的父子进程合作将整数 X 的值从 1 加到 10 的功能。它们通过管道相互将计算结果发给对方。

1. 在新建文件夹中建立以下名为 ppipe.c 的 C 语言程序

```
/*
* Filename  : ppipe.c
* copyright : (C) 2006 by zhanghonglie
* Function  : 利用管道实现在父子进程间传递整数
*/
#include <stdio.h>
```

```c
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int pid;  // 进程号
    int pipe1[2];   // 存放第一个无名管道标号
    int pipe2[2];   // 存放第二个无名管道标号
    int x;  // 存放要传递的整数
    // 使用 pipe() 系统调用建立两个无名管道。建立不成功程序退出，执行终止
    if (pipe(pipe1) < 0) {
        perror("pipe not create");
        exit(EXIT_FAILURE);
    }
    if (pipe(pipe2) < 0) {
        perror("pipe not create");
        exit(EXIT_FAILURE);
    }
    // 使用 fork() 系统调用建立子进程，建立不成功程序退出，执行终止
    if ((pid = fork()) < 0) {
        perror("process not create");
        exit(EXIT_FAILURE);
    }
    // 子进程号等于 0 表示子进程在执行
    else if (pid == 0) {
        // 子进程负责从管道 1 的 0 端读，管道 2 的 1 端写
        // 因此关掉管道 1 的 1 端和管道 2 的 0 端
        close(pipe1[1]);
        close(pipe2[0]);
        // 每次循环从管道 1 的 0 端读一个整数放入变量 X 中
        // 并对 X 加 1 后写入管道 2 的 1 端，直到 X 大于 10
        do {
            read(pipe1[0], &x, sizeof(int));
            printf("child %d read: %d\n", getpid(), x++);
            write(pipe2[1], &x, sizeof(int));
        } while (x <= 9);
        // 读写完成后，关闭管道
        close(pipe1[0]);
        close(pipe2[1]);
        // 子进程执行结束
        exit(EXIT_SUCCESS);
    }
    // 子进程号大于 0 表示父进程在执行，
    else {
        // 父进程负责从管道 2 的 0 端读，管道 1 的 1 端写
        // 所以关掉管道 1 的 0 端和管道 2 的 1 端
        close(pipe1[0]);
        close(pipe2[1]);
        x = 1;
        // 每次循环向管道 1 的 1 端写入变量 X 的值
        // 并从管道 2 的 0 端读一整数写入 X 再对 X 加 1，直到 X 大于 10
```

```
        do {
            write(pipe1[1], &x, sizeof(int));
            read(pipe2[0], &x, sizeof(int));
            printf("parent %d read: %d\n", getpid(), x++);
        } while (x <= 9);
        // 读写完成后，关闭管道
        close(pipe1[1]);
        close(pipe2[0]);
    }
    // 父进程执行结束
    return EXIT_SUCCESS;
}
```

2. 在当前目录中建立以下Makefile文件：

```
srcs =  ppipe.c
objs =  ppipe.o
opts =  -g -c

all:  ppipe
ppipe:  $(objs)
    gcc $(objs) -o ppipe
ppipe.o:  $(srcs)
    gcc $(opts) $(srcs)
clean:
    rm ppipe *.o
```

3. 使用make命令编译连接生成可执行文件ppipe:

```
$ make
gcc -g -c ppipe.c
gcc ppipe.o -o ppipe
```

4. 编译成功后执行ppipe:命令:

```
$ ./ppipe
child 8697 read: 1
parent 8696 read: 2
child 8697 read: 3
parent 8696 read: 4
child 8697 read: 5
parent 8696 read: 6
child 8697 read: 7
parent 8696 read: 8
child 8697 read: 9
parent 8696 read: 10
```

可以看到以上程序的执行中父子进程交替的将整数X 的值从 1 加到了 10。

2.4 独立实验

设有二元函数 f(x, y) = f(x) + f(y)

其中：

- f(x) = f(x-1) * x (x >1)
- f(x) = 1 (x=1)
- f(y) = f(y-1) + f(y-2) (y > 2)
- f(y) = 1 (y=1, 2)

请编程建立 3 个并发协作进程或线程，它们分别完成 f(x, y)、f(x)、f(y)

2.5 实验要求

根据示例实验程序和独立实验程序观察和记录的调试和运行的信息，思考以下问题：

- 它们反映出操作系统教材中讲解的进 / 线程协作和进 / 线程通信概念的哪些特征和功能?
- 在真实的操作系统中是怎样实现和反映出教材中进 / 线程通信概念的?
- 你对于进 / 线程协作和进 / 线程通信的概念和实现有哪些新的理解和认识?
- 管道机制的机理是什么?
- 怎样利用管道完成进 / 线程间的协作和通信?

根据实验程序、调试过程和结果分析写出实验报告。

实验三、Shell实验（MSH）

3.1 实验说明

这一部分的实验将会揭示 UNIX 系统中的 shell 是如何使用基础系统调用的。说到 Shell，你可能并不陌生，在你的 Ubuntu 系统中打开终端（命令行这个名字可能对你来说更熟悉一些），里面运行着的就是一个 Shell —— 它的名字叫 Bash。当然，你也可以选择使用其他的 Shell，比如 Zsh、Fish 等等。如果你感兴趣的话可以到网上搜一下。

在你打开的这个 Shell 中，你可以执行很多命令，例如你输入 `echo hello`，你会看到在标准输出中打印了 hello 这个单词；如果你输入 `echo hello > file.txt` 你会看到标准输出中并没有打印 hello 这个词，反而在当前目录下出现了一个名为 `file.txt` 的文件，这个文件中存储了一个单词 hello。

在这行命令中，echo 是一个可执行文件，你可以在路径 `/bin` 下找到它，**你并不需要手动实现这些命令**。当 Shell 执行命令的时候会创建一个进程来运行这个可执行文件，而 hello 为 echo 的一个参数（当一个程序执行时，可以传递 0 或多个参数，在 C 语言中，参数的数量会被传递给 main 函数的第一个参数 `argc`，而参数列表会被传递给 main 函数的第二个参数 `argv`）。

命令符 ">" 是标准输出的重定向符，它会将 echo 的输出重定向到文件 `file.txt` 中，而不是 `stdout`，相反，命令符 "<" 则是标准输入的重定向符，它会将输出重定向到文件中，而不是 `stdin`。两者可以结合。

在 Linux 的 Shell 中可以使用命令符 "|" 来建立双向的无名管道，它的语法为 `command1 | command2`。这个命令符会通过管道将左侧程序的输出作为输入传递给右侧程序。管道支持多级连接，即支持语法 `command1 | command2 | command3 | ... | commandN`，左侧的标准输出内容会被一次传递给右侧的下一级，作为它们的标准输入。

3.2 独立实验

你的任务是实现一个在 Linux 系统下运行的简单 Shell，它能够实现如下功能：

1. 运行带参数的命令
2. 重定向输入输出（仅实现 `<` 与 `>` 即可）
3. 建立多级连接的双向管道，实现 `|` 左侧的标准输出传递给右侧作为其标准输入
4. 能够响应 Linux 中的信号，即当执行命令的进程收到 `SIGINT` 时会被杀死，而 Shell 本身不会被杀死

其余的 Shell 特性不需要实现。以下是示例（这里的 `#` 是命令提示符，类似于 bash 中的 `$`）。

```
# echo hello there
hello there
# echo something > file.txt
# cat file.txt
something
# cat file.txt | wc
     1       1      10
# echo echo hello | ./msh
hello
# # #
```

3.3 实验提示

在这个实验中，你可能会用到的系统调用为 fork / execvp / wait / signal / pipe / dup，无需使用除此以外其他的系统调用。下面将对这些系统调用作简要的介绍。(**PS**：这一部分的内容与前两个实验的介绍高度重合，请选择性参考)

fork()

进程可以通过系统调用 fork() 创建子进程并和其子进程并发执行。子进程初始的执行映像是父进程的一个复本：子进程会复制父进程的数据与堆栈空间, 并继承父进程的用户代码、组代码、环境变量、已打开的文件代码、工作目录和资源限制等。子进程可以通过 exec() 系统调用族装入一个新的执行程序。父进程可以使用 wait() 或 waitpid() 系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

fork() 系统调用语法：

```
#include <unistd.h>
pid_t fork(void);
```

fork 成功创建子进程后将返回子进程的进程号,不成功会返回-1。**不保证父子进程先后的调用顺序**

范例

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int pid = fork();
    if(pid < 0) {
        printf("fork failed\n");
    } else if(pid == 0) {
        printf("This is the child process\n");
    } else {
        printf("This is the parent process\n");
    }
  return 0;
}
/* 可能的一种输出
This is the parent process
This is the child process
*/
```

execvp()

exec 系统调用有一组 6 个函数，其中 execvp() 系统调用语法：

```
#include <unistd.h>
int execvp(const char *file, const char *argv[]);
```

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。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。

范例

```
#include <unistd.h>
int main() {
    char * argv[] = {"ls", "-al", "/etc/passwd", NULL};
    execvp("ls", argv);
}
/* 输出
-rw-r--r-- 1 root root 705 Sep 3 13 :52 /etc/passwd
*/
```

wait()

wait() 系统调用语法：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int * status);
```

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。

范例

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    int pid = fork();
    if(pid < 0) {
        printf("fork failed\n");
    } else if(pid == 0) {
        printf("This is the child process\n");
    } else {
        wait(NULL);
        printf("This is the parent process\n");
    }
  return 0;
}
/* 输出唯一，如果父进程先执行 wait 则会被挂起，切换子进程执行
This is the child process
This is the parent process
*/
```

signal()

可以通过信号向一个进程发送消息以控制进程的行为。信号是由中断或异常事件引发的，如：键盘中断、定时器中断、非法内存引用等。信号的名字都以 SIG 开头，例如 SIGTERM、SIGHUP。可以使用 kill -l 命令查看系统当前的信号集合。 信号可在任何时间发生，接收信号的进程可以对接收到的信号采取 3 种处理 措施之一：

1. 忽略这个信号
2. 执行系统默认的处理
3. 捕捉这个信号做自定义的处理

信号从产生到被处理所经过的过程： 产生 (generate) -> 挂起 (pending) -> 送 (deliver) -> 部署 (disposition) 或忽略 (ignore)

一个信号集合是一个 C 语言的 sigset_t 数据类型的对象，sigset_t 数据类型定义在 <signal.h> 中。被一个进程忽略的所有信号的集合称为一个信号掩码 (mask)。从程序中向一个进程发送信号有两种方法：调用 shell 的 kill 命令，调用 kill 系统调用函数。kill 能够发送除杀死一个进程 (SIGKILL、SIGTERM、SIGQUIT) 之外的其他信号，例如键盘中断 (Ctrl+C) 信号 SIGINT，进程暂停 (Ctrl+Z) 信号 SIGTSTP 等等。每个进程都能使用 signal 函数定义自己的信号处理函数，捕捉并自行处理接收的除 SIGSTOP 和 SIGKILL 之外的信号。

singal() 系统调用语法：

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

其中 signum 是要捕捉的信号，handler 是自定义的信号处理函数名。

pipe()

管道 pipe 是进程间通信最基本的一种机制。在内存中建立的管道称为无名管道， 在磁盘上建立的管道称为有名管道。无名管道随着进程的撤消而消失，有名管道则可以长久保存，Shell 命令符 “|” 建立的就是无名管道，而 shell 命令 mkfifo 建立的是有名管道。两个进程可以通过管道一个在管道一端向管道发送其输出，给另一进程可以在管道的另一端从管道得到其输入 。管道以半双工方式工作，即它的数据流是单方向的。因此使用一个管道一般的规则是读管道数据的进程关闭管道写入端，而写管道进程关闭其读出端。管道既可以采用同步方式工作也可以采用异步方式工作。

pipe() 系统调用语法：

```
#include <unistd.h>
int pipe(int pipe_id[2]);
```

pipe() 建立一个无名管道，pipe_id[0] 和 pipe_id[1] 将放入管道两端的描述符。如果 pipe 执行成功返回 0，出错返回 -1。

管道读写的系统调用语法为：

```
#include <unistd.h>
size_t read(int pipe_id, const void *buf, size_t count);
size_t write(int pipe_id, const void *buf, size_t count);
```

read 和 write 分别在管道的两端进行读和写。pipe_id 是 pipe 系统调用返回的管道描述符。Buf 是数据缓冲区首地址，count 说明数据缓冲区以 size_t 为单位的长度。read 和 write 的返回值为它们实际读写的数据单位。注意管道的读写默认的通信方式为同步读写方式，即如果管道读端无数据则读者阻塞直到数据到达，反之如果管道写端有数据则写者阻塞直到数据被读走。

dup()

在每个进程被创建的时候，操作系统会为其自动打开三个文件描述符：0、1、2，它们分别代表 stdin、stdout 和 stderr，之后如果进行文件打开操作，则会从 4 开始分配文件描述符。在分配文件描述符的时候会选择最小的未被分配的描述符。这意味着，如果你通过 close 将 1 号文件描述符 (stdout) 关闭，然后再使用 open 打开一个文件，这时你的 printf 输出会输出到这个文件中。

dup() 系统调用语法：

```
#include <unistd.h>
int dup(int oldfd);
```

dup() 用来复制参数 oldfd 所指的文件描述符，并将它返回。此新的文件描述符和参数 oldfd 指的是同一个文件，共享所有的锁定、读写位置和各项权限。当复制成功时，则返回最小及尚未使用的文件描述符。若有错误则返回 -1，errno 会存放错误代码。

我们以下述代码为例，进一步介绍该系统调用的使用方法。

```
// stdin、stdout、stderr 文件描述符分别为 0、1、2
int pipe_id[2];
if (pipe(pipe_id) < 0) {          // 建立管道，其管道读、写的描述符分别为 3、4
  perror("pipe is not created");
  exit(EXIT_FAILURE);
}
close(0);          // 关闭 0 位置处的 stdin 文件描述符，即该位置的文件描述符失效
dup(pipe_id[0]);   // 将管道读的描述符复制到位置 0 处，此时位置 0 与位置 3 均为管道读的描述符
close(pipe_id[0]); // 关闭位置 3 的管道读的描述符
```

根据上述代码，我们即可实现将「标准输入」 重定向到「管道读」的效果。此时如果执行 scanf，则不再从标准输入中进行数据读取，而是从管道中进行读取。

实验四、进程同步实验

4.1 实验目的

加深对并发协作进程同步与互斥概念的理解，观察和体验并发进程同步与互斥操作的效果，分析与研究经典进程同步与互斥问题的实际解决方案。了解 Linux 系统中 IPC 进程同步工具的用法，练习并发协作进程的同步与互斥操作的编程与调试技术。

4.2 实验说明

在 linux 系统中可以利用进程间通信（interprocess communication ）IPC 中的 3 个对象：共享内存、信号灯数组、消息队列，来解决协作并发进程间的同步与互斥的问题。

1. 共享内存是 OS 内核为并发进程间交换数据而提供的一块内存区（段）。如果段的权限设置恰当，每个要访问该段内存的进程都可以把它映射到自己私有的地址空间中。如果一进程更新了段中数据，那么其他进程立即会看到这一更新。进程创建的段也可由另一进程读写。

linux 中可用命令 ipcs -m 观察共享内存情况。

```
$ ipcs -m
------ Shared Memory Segments --------
key shmid owner perms bytes nattch  status
0x00000000 327682 student 600 393216  2 dest
0x00000000 360451 student 600 196608  2 dest
0x00000000 393220 student 600 196608  2 dest
```

key 共享内存关键值

- shmid 共享内存标识
- owner 共享内存所由者（本例为 student）
- perm 共享内存使用权限（本例为 student 可读可写）
- byte 共享内存字节数
- nattch 共享内存使用计数
- status 共享内存状态

上例说明系统当前已由 student 建立了一些共享内存，每个都有两个进程在共享。

2. 信号灯数组是 OS 内核控制并发进程间共享资源的一种进程同步与互斥机制。

linux 中可用命令 ipcs -s 观察信号灯数组的情况。

```
$ ipcs -s
------ Semaphore Arrays --------
key semid   owner   perms   nsems
0000000 163844  apache  600   1
0x4d00f259 294920 beagleind 600 8
0x00000159 425995 student 644 1
```

- semid 信号灯的标识号
- nsems 信号灯的个数

其他字段意义同以上共享内存所述。

上例说明当前系统中已经建立多个信号灯。其中最后一个标号为 425996 是由 student 建立的，它的使用权限为 644，信号灯数组中信号灯个数为 1 个。

3. 消息队列是 OS 内核控制并发进程间共享资源的另一种进程同步机制。 linux 中可用命令 `ipcs -q` 观察消息队列的情况。

```
$ipcs -q
------ Message Queues --------
key msqid   owner   perms   used-bytes  messages
0x000001c8 0  root  644 8     1
```

- msqid 消息队列的标识号
- used-bytes 消息的字节长度
- messages 消息队列中的消息条数

其他字段意义与以上两种机制所述相同。

上例说明当前系统中有一条建立消息队列，标号为 0，为 root 所建立，使用权限为 644，每条消息 8 个字节，现有一条消息。

4. 在权限允许的情况下您可以使用 ipcrm 命令**删除系统当前存在的 IPC 对象中的任一个对象。**

- ipcrm -m 21482 删除标号为 21482 的共享内存。
- ipcrm -s 32673 删除标号为 32673 的信号灯数组。
- ipcrm -q 18465 删除标号为 18465 的消息队列。

5. 在 linux 的 proc 文件系统中有 3 个虚拟文件动态记录了由以上 ipcs 命令显示的

当前 IPC 对象的信息，它们分别是：

- /proc/sysvipc/shm 共享内存
- /proc/sysvipc/sem 信 号 量
- /proc/sysvipc/msg 消息队列

我们可以利用它们在程序执行时获取有关 IPC 对象的当前信息。

6. IPC 对象有关的系统调用函数原型都声明在以下的头文件中：

```
#include <sys/types.h>
#include <sys/ipc.h>
```

创建一段共享内存系统调用语法：

```
#include <sys/shm.h>
int shmget(key_t key,int size,int flags);
```

- key 共享内存的键值
  - 可以为 IPC_PRIVATE，也可以用整数指定一个 size 共享内存字节长度。
- flags 共享内存权限位
- shmget 调用成功后，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的共享内存段标识符。 如果指定的 key 已存在则返回与 key 关联的标识符，不成功返回 -1。

令一段共享内存附加到调用进程中的系统调用语法：

```
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int flags)
```

- shmid 由 shmget 创建的共享内存的标识符
- shmaddr 总为 0，表示用调用者指定的指针指向共享段。
- flags 共享内存权限位
- shmat 调用成功后返回附加的共享内存首地址

令一段共享内存从到调用进程中分离出去的系统调用语法：

```
#include <sys/shm.h>
int shmdt(char *shmadr);
```

- shmadr 进程中指向附加共享内存的指针
- shmdt 调用成功将递减附加计数，当计数为 0，将删除共享内存。调用不成功返回 -1。

创建一个信号灯数组的系统调用有语法:

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flags);
```

- key 信号灯数组的键值
  - 可以为 IPC_PRIVATE，也可以用整数指定一个 nsems 信号灯数组中信号灯的个数。
- flags 信号灯数组权限位
  - 如果 key 用整数指定，应设置 IPC_CREAT 位。
- semget 调用成功，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的信号等数组标识符。 如果指定的整数 key 已存在则返回与 key 关联的标识符，不成功返回 -1。

操作信号灯数组的系统调用语法:

```
 #include <sys/sem.h>
int semop(int semid, struct sembuf *semop, unsigned nops);
```

- semid 由 semget 创建的信号灯数组的标识符
- semop 指向 sembuf 结构体数组的指针，表示对 semid 指定的信号灯数组的操作
- nops 信号灯数组元素的个数。
- semop 调用成功返回 0，不成功返回 -1。

结构体 `struct sembuf` 包含三个 short 类型字段

- sem_num 表示该操作的目标信号灯在数组中的下标
- sem_op 表示对信号灯的操作，有以下三种情况
  - 如果 sem_op 的值是负数。存在以下三种可能。
    - 如果相应信号灯的值大于等于 sem_op 的绝对值。在 semop 函数执行以后，相应信号灯的值会被减去 sem_op 的绝对值。
    - 如果相应信号灯的值小于 sem_op 的绝对值并且 (sem_flg & IPC_NOWAIT) 非0，即 struct sembuf 的第三个参数设置了 IPC_NOWAIT，那么 semop 函数将会直接返回。
    - 如果相应信号灯的值小于 sem_op 的绝对值并且 (sem_flg & IPC_NOWAIT) 是0，即 struct sembuf 的第三个参数没有设置 IPC_NOWAIT，那么在 semop 函数执行以后进程将会被挂起。当信号灯的值大于等于 sem_op 的绝对值后或者相应的信号灯被删除（即不存在）或者进程接收到信号的时候被挂起的进程会被唤醒。
  - 如果 sem_op 的值是正数。在 semop 函数执行以后，相应信号灯的值会被加上 sem_op。
  - 如果 sem_op 的值是0。这种情况我们不在这里进行讨论，如果感兴趣可以咨询查阅相关材料。

- sem_flg 表示该操作的标志位。这里我们仅仅简单地讨论一下 SEM_UNDO 的使用。SEM_UNDO 这个标志是用来防止死锁的发生。在调用 semop 函数是，如果 sem_flg 位设置了 SEM_UNDO，那么在这个进程正常退出（调用 exit 退出或 main 执行完）或异常退出（如段异常、除 0 异常、收到 KILL 信号等）时将修改的信号灯的值归还给信号灯。例如一个信号灯的初值是 20，进程以 SEM_UNDO 方式操作信号量减 2，减 5，加 1；在进程未退出时，信号量变成 20-2-5+1=14；在进程退出时，将修改的值归还给信号量，信号量变成 14+2+5-1=20。

控制信号灯数组的系统调用语法:

```
#include <sys/sem.h>
int semctl(int semid,int semnum,int cmd, union semun arg);
```

- semid 由 semget 创建的信号灯数组的标识符
- semnum 该信号灯数组中的第几个信号灯
- cmd 对信号灯发出的控制命令
    - 例如：GETVAL 返回当前信号灯状态 - SETVAL 设置信号灯状态。
- IPC_RMD 删除标号为 semid 的信号灯
- arg 保存信号灯状态的联合体
    - 信号灯的值是其中一个基本成员。

```
union semun {
    int val;  /* value for SETVAL */
    ......
};
```

semctl 执行不成功返回 -1，否则返回指定的 cmd 的值。

创建消息队列的系统调用语法:

```
#include<sys/msg.h>
int msgget(key_t key,int flags)
```

- key 消息队列的键值
    - 可以为 IPC_PRIVATE，也可以用整数指定一个。
- flags 消息队列权限位
- msgget 调用成功，如果 key 用新整数指定，且 flags 中设置了 IPC_CREAT 位，则返回一个新建立的消息队列标识符。如果指定的整数 key 已存在则返回与 key 关联的标识符，否则返回 -1。

追加一条新消息到消息队列的系统调用语法：

```
#include <sys.msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

- msqid 由消息队列的标识符
- msgp 消息缓冲区指针。消息缓冲区结构为：

```
struct msgbuf {
    long mtype; /* 消息类型，必须大于 0 */
    char mtext[1];  /* 消息数据，长度应与 msgsz 声明的一致 */
}
```

- msgsz 消息数据的长度
- msgflg 为 0 表示阻塞方式，设置 IPC_NOWAIT 表示非阻塞方式

msgsnd 调用成功返回 0，不成功返回 -1。

从到消息队列中读出一条新消息的系统调用语法：

```
#include <sys.msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg);
```

- msqid 由消息队列的标识符
- msgp 消息缓冲区指针。消息缓冲区结构为：

```
struct msgbuf {
    long mtype; /* 消息类型，必须大于 0 */
    char mtext[1];  /* 消息数据，长度应于msgsz 声明的一致*/
}
```

- msgsz 消息数据的长度
- msgtype 决定从队列中返回哪条消息：
    1. `= 0` 返回消息队列中第一条消息
    2. `> 0` 返回消息队列中等于 mtype 类型的第一条消息
    3. `< 0` 返回 mtype <= type 绝对值最小值的第一条消息
- msgflg 为 0 表示阻塞方式，设置 IPC_NOWAIT 表示非阻塞方式

msgrcv 调用成功返回 0，不成功返回 -1。

删除消息队列的系统调用语法：

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- msqid 由消息队列的标识符
- cmd 控制命令，常用的有：
    - IPC_RMID 删除 msgid 标识的消息队列
    - IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲 buf
    - IPC_SET 改变队列的 UID、GID，访问模式和最大字节数。

msgctl 调用成功返回 0，不成功返回 -1。

4.3 示例实验

以下示例实验程序应能模拟多个生产 / 消费者在有界缓冲上正确的操作。它利用 N 个字节的共享内存作为有界循环缓冲区，利用写一字符模拟放一个产品，利用读一字符模拟消费一个产品。当缓冲区空时消费者应阻塞睡眠，而当缓冲区满时生产者应当阻塞睡眠。一旦缓冲区中有空单元，生产者进程就向空单元中入写字符, 并报告写的内容和位置。一旦缓冲区中有未读过的字符，消费者进程就从该单元中读出字符，并报告读取位置。生产者不能向同一单元中连续写两次以上相同的字符，消费者也不能从同一单元中连续读两次以上相同的字符。

1. 在当前新建文件夹中建立以下名为 ipc.h 的 C 程序的头文件，该文件中定义了生产者 / 消费者共用的 IPC 函数的原型和变量：

```
/*
* Filename  : ipc.h
* copyright : (C) 2006 by zhonghonglie
* Function  : 声明 IPC 机制的函数原型和全局变量
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>

#define BUFSZ 256
//建立或获取 ipc 的一组函数的原型说明
int get_ipc_id(char *proc_file,key_t key);

char *set_shm(key_t shm_key,int shm_num,int shm_flag);
int set_msq(key_t msq_key,int msq_flag);
int set_sem(key_t sem_key,int sem_val,int sem_flag);
int down(int sem_id);
int up(int sem_id);

/*信号灯控制用的共同体*/
typedef union semuns {
      int val;
} Sem_uns;

/* 消 息 结 构 体 */
typedef struct msgbuf {
      long mtype;
      char mtext[1];
} Msg_buf;

//生产消费者共享缓冲区即其有关的变量
key_t buff_key;
int buff_num;
char *buff_ptr;
```

```
//生产者放产品位置的共享指针
key_t pput_key;
int pput_num;
int *pput_ptr;

//消费者取产品位置的共享指针
key_t cget_key;
int cget_num;
int *cget_ptr;

//生产者有关的信号量
key_t prod_key;
key_t pmtx_key;
int prod_sem;
int pmtx_sem;

//消费者有关的信号量
key_t cons_key;
key_t cmtx_key;
int cons_sem;
int cmtx_sem;

int sem_val;
int sem_flg;
int shm_flg;
```

2. 在当前新建文件夹中建立以下名为 ipc.c 的 C 程序，该程序中定义了生产者/消费者共用的 IPC 函数：

```
/*
* Filename  : ipc.c
* copyright : (C) 2006 by zhonghonglie
* Function  : 一组建立 IPC 机制的函数
*/
#include "ipc.h"
/*
* get_ipc_id() 从/proc/sysvipc/文件系统中获取 IPC 的 id 号
* pfile: 对应/proc/sysvipc/目录中的 IPC 文件分别为
* msg-消息队列,sem-信号量,shm-共享内存
* key:  对应要获取的 IPC 的 id 号的键值
*/
int get_ipc_id(char *proc_file, key_t key) {
    FILE *pf; int i, j;
    char line[BUFSZ], colum[BUFSZ];

    if ((pf = fopen(proc_file, "r")) == NULL) {
        perror("Proc file not open");
        exit(EXIT_FAILURE);
```

```
    }
    fgets(line, BUFSZ, pf);
    while (!feof(pf)) {
        i = j = 0;
        fgets(line, BUFSZ, pf);
        while (line[i] == ' ') i++;
        while (line[i] != ' ') colum[j++] = line[i++];
        colum[j] = '\0';
        if (atoi(colum) != key) continue;
        j = 0;
        while (line[i] == ' ') i++;
        while (line[i] != ' ') colum[j++] = line[i++];
        colum[j] = '\0';
        i = atoi(colum);
        fclose(pf);
        return i;
    }
    fclose(pf);
    return -1;
}

/*
* 信号灯上的down/up 操作
* semid:信号灯数组标识符
* semnum:信号灯数组下标
* buf:操作信号灯的结构
*/
int down(int sem_id) {
    struct sembuf buf;
    buf.sem_op = -1;
    buf.sem_num = 0;
    buf.sem_flg = SEM_UNDO;
    if ((semop(sem_id, &buf, 1)) < 0) {
        perror("down error ");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}

int up(int sem_id) {
    struct sembuf buf;
    buf.sem_op = 1;
    buf.sem_num = 0;
    buf.sem_flg = SEM_UNDO;
    if ((semop(sem_id, &buf, 1)) < 0) {
        perror("up error ");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
```

```
}

/*
* set_sem 函数建立一个具有 n 个信号灯的信号量
* 如果建立成功，返回 一个信号灯数组的标识符 sem_id
* 输入参数：
* sem_key 信号灯数组的键值
* sem_val 信号灯数组中信号灯的个数
* sem_flag 信号等数组的存取权限
*/
int set_sem(key_t sem_key, int sem_val, int sem_flg) {
    int sem_id;
    Sem_uns sem_arg;
    //测试由 sem_key 标识的信号灯数组是否已经建立
    if ((sem_id = get_ipc_id("/proc/sysvipc/sem", sem_key)) < 0 ) {
        //semget 新建一个信号灯,其标号返回到 sem_id
        if ((sem_id = semget(sem_key, 1, sem_flg)) < 0) {
            perror("semaphore create error");
            exit(EXIT_FAILURE);
        }
        //设置信号灯的初值
        sem_arg.val = sem_val;
        if (semctl(sem_id, 0, SETVAL, sem_arg) < 0) {
            perror("semaphore set error");
            exit(EXIT_FAILURE);
        }
    }
    return sem_id;
}

/*
* set_shm 函数建立一个具有 n 个字节 的共享内存区
* 如果建立成功，返回 一个指向该内存区首地址的指针 shm_buf
* 输入参数：
* shm_key 共享内存的键值
* shm_val 共享内存字节的长度
* shm_flag 共享内存的存取权限
*/
char *set_shm(key_t shm_key, int shm_num, int shm_flg) {
    int i, shm_id;
    char *shm_buf;
    //测试由 shm_key 标识的共享内存区是否已经建立
    if ((shm_id = get_ipc_id("/proc/sysvipc/shm", shm_key)) < 0 ) {
        //shmget 新建 一个长度为 shm_num 字节的共享内存,其标号返回shm_id
        if ((shm_id = shmget(shm_key, shm_num, shm_flg)) < 0) {
            perror("shareMemory set error"); exit(EXIT_FAILURE);
        }
        //shmat 将由 shm_id 标识的共享内存附加给指针 shm_buf
        if ((shm_buf = (char *)shmat(shm_id, 0, 0)) < (char *)0) {
```

```
            perror("get shareMemory error"); exit(EXIT_FAILURE);
        }
        for (i = 0; i < shm_num; i++) shm_buf[i] = 0; //初始为 0
    }
    //shm_key 标识的共享内存区已经建立,将由 shm_id 标识的共享内存附加给指针 shm_buf
    if ((shm_buf = (char *)shmat(shm_id, 0, 0)) < (char *)0) {
        perror("get shareMemory error");
        exit(EXIT_FAILURE);
    }

    return shm_buf;
}

/*
* set_msq 函数建立一个消息队列
* 如果建立成功，返回 一个消息队列的标识符 msq_id
* 输入参数:
* msq_key 消息队列的键值
* msq_flag 消息队列的存取权限
*/
int set_msq(key_t msq_key, int msq_flg) {
    int msq_id;
    //测试由 msq_key 标识的消息队列是否已经建立
    if ((msq_id = get_ipc_id("/proc/sysvipc/msg", msq_key)) < 0 ) {
        //msgget 新建一个消息队列,其标号返回到 msq_id
        if ((msq_id = msgget(msq_key, msq_flg)) < 0) {
            perror("messageQueue set error"); exit(EXIT_FAILURE);
        }
    }
    return msq_id;
}
```

3. 在当前新文件夹中建立生产者程序 producer.c

```
/*
* Filename  : producer.c
* copyright : (C) 2006 by zhonghonglie
* Function  :  建立并模拟生产者进程
*/

#include "ipc.h"

int main(int argc, char *argv[]) {
    int rate;
    //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
    if (argv[1] != NULL) rate = atoi(argv[1]);
    else rate = 3;  //不指定为 3 秒
    //共享内存使用的变量
```

```
    buff_key = 101;//缓冲区任给的键值
    buff_num = 8;//缓冲区任给的长度
    pput_key = 102;//生产者放产品指针的键值
    pput_num = 1; //指针数
    shm_flg = IPC_CREAT | 0644;//共享内存读写权限

    //获取缓冲区使用的共享内存，buff_ptr 指向缓冲区首地址
    buff_ptr = (char *)set_shm(buff_key, buff_num, shm_flg);
    //获取生产者放产品位置指针 pput_ptr
    pput_ptr = (int *)set_shm(pput_key, pput_num, shm_flg);

    //信号量使用的变量
    prod_key = 201;//生产者同步信号灯键值
    pmtx_key = 202;//生产者互斥信号灯键值
    cons_key = 301;//消费者同步信号灯键值
    cmtx_key = 302;//消费者互斥信号灯键值
    sem_flg = IPC_CREAT | 0644;

    //生产者同步信号灯初值设为缓冲区最大可用量
    sem_val = buff_num;
    //获取生产者同步信号灯，引用标识存 prod_sem
    prod_sem = set_sem(prod_key, sem_val, sem_flg);
    //消费者初始无产品可取，同步信号灯初值设为 0
    sem_val = 0;

    //获取消费者同步信号灯，引用标识存 cons_sem
    cons_sem = set_sem(cons_key, sem_val, sem_flg);
    //生产者互斥信号灯初值为 1
    sem_val = 1;
    //获取生产者互斥信号灯，引用标识存 pmtx_sem
    pmtx_sem = set_sem(pmtx_key, sem_val, sem_flg);

    //循环执行模拟生产者不断放产品
    while (1) {
        //如果缓冲区满则生产者阻塞
        down(prod_sem);
        //如果另一生产者正在放产品，本生产者阻塞
        down(pmtx_sem);

        //用写一字符的形式模拟生产者放产品，报告本进程号和放入的字符及存放的位置
        buff_ptr[*pput_ptr] = 'A' + *pput_ptr;
        sleep(rate);
        printf("%d producer put: %c to  Buffer[%d]\n", getpid(), buff_ptr[*pput_ptr],
*pput_ptr);
        //存放位置循环下移
        *pput_ptr = (*pput_ptr + 1) % buff_num;

        //唤醒阻塞的生产者
        up(pmtx_sem);
```

```
        //唤醒阻塞的消费者
        up(cons_sem);
    }

    return EXIT_SUCCESS;
}
```

4. 在当前新文件夹中建立消费者程序 consumer.c

```
/*
Filename  : consumer.c
copyright : (C) by zhanghonglie
Function  : 建立并模拟消费者进程
*/

#include "ipc.h"

int main(int argc, char *argv[]) {
    int rate;
    //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
    if (argv[1] != NULL) rate = atoi(argv[1]);
    else rate = 3;  //不指定为 3 秒
    //共享内存 使用的变量
    buff_key = 101; //缓冲区任给的键值
    buff_num = 8; //缓冲区任给的长度
    cget_key = 103; //消费者取产品指针的键值
    cget_num = 1; //指针数
    shm_flg = IPC_CREAT | 0644; //共享内存读写权限
    //获取缓冲区使用的共享内存，buff_ptr 指向缓冲区首地址
    buff_ptr = (char *)set_shm(buff_key, buff_num, shm_flg);
    //获取消费者取产品指针，cget_ptr 指向索引地址
    cget_ptr = (int *)set_shm(cget_key, cget_num, shm_flg);
    //信号量使用的变量
    prod_key = 201; //生产者同步信号灯键值
    pmtx_key = 202;   //生产者互斥信号灯键值
    cons_key = 301; //消费者同步信号灯键值
    cmtx_key = 302;   //消费者互斥信号灯键值
    sem_flg = IPC_CREAT | 0644; //信号灯操作权限
    //生产者同步信号灯初值设为缓冲区最大可用量
    sem_val = buff_num;
    //获取生产者同步信号灯，引用标识存 prod_sem
    prod_sem = set_sem(prod_key, sem_val, sem_flg);
    //消费者初始无产品可取，同步信号灯初值设为 0
    sem_val = 0;
    //获取消费者同步信号灯，引用标识存 cons_sem
    cons_sem = set_sem(cons_key, sem_val, sem_flg);
    //消费者互斥信号灯初值为 1
    sem_val = 1;
```

```
    //获取消费者互斥信号灯，引用标识存 pmtx_sem
    cmtx_sem = set_sem(cmtx_key, sem_val, sem_flg);

    //循环执行模拟消费者不断取产品
    while (1) {
        //如果无产品消费者阻塞
        down(cons_sem);
        //如果另一消费者正在取产品，本消费者阻塞
        down(cmtx_sem);

        //用读一字符的形式模拟消费者取产品，报告本进程号和获取的字符及读取的位置
        sleep(rate);
        printf("%d  consumer  get:  %c  from Buffer[%d]\n", getpid(),
buff_ptr[*cget_ptr], *cget_ptr);
        //读取位置循环下移
        *cget_ptr = (*cget_ptr + 1) % buff_num;

        //唤醒阻塞的消费者
        up(cmtx_sem);
        //唤醒阻塞的生产者
        up(prod_sem);
    }

    return EXIT_SUCCESS;
}
```

5. 在当前文件夹中建立 Makefile 项目管理文件

```
hdrs = ipc.h
opts = -g -c
c_src = consumer.c ipc.c
c_obj = consumer.o ipc.o
p_src = producer.c ipc.c
p_obj = producer.o ipc.o
all:  producer consumer
consumer: $(c_obj)
            gcc $(c_obj) -o consumer
consumer.o: $(c_src) $(hdrs)
            gcc $(opts) $(c_src)

producer: $(p_obj)
            gcc $(p_obj) -o producer
producer.o: $(p_src) $(hdrs)
            gcc $(opts) $(p_src)
clean:
          rm consumer producer *.o
```

6. 使用 make 命令编译连接生成可执行的生产者、消费者程序

```
$ make
gcc -g -c producer.c ipc.c
gcc producer.o ipc.o -o producer
gcc -g -c consumer.c ipc.c
gcc consumer.o ipc.o -o consumer
```

7. 在当前终端窗体中启动执行速率为 1 秒的一个生产者进程

```
$./producer 1
12263 producer put: A to Buffer[0]
12263 producer put: B to Buffer[1]
12263 producer put: C to Buffer[2]
12263 producer put: D to Buffer[3]
12263 producer put: E to Buffer[4]
12263 producer put: F to Buffer[5]
12263 producer put: G to Buffer[6]
12263 producer put: H to Buffer[7]
```

可以看到 12263 号进程在向共享内存中连续写入了 8 个字符后因为缓冲区满而阻塞。

8. 打开另一终端窗体，进入当前工作目录，从中再启动另一执行速率为 3 的生产者进程: $ ./producer 3
可以看到该生产者进程因为缓冲区已满而立即阻塞。

9. 再打开另外两个终端窗体，进入当前工作目录，从中启动执行速率为 2 和 4 的两个消费者进程:

```
$ ./consumer 2
12528 consumer get: B from Buffer[1]
12528 consumer get: D from Buffer[3]
12528 consumer get: F from Buffer[5]
12528 consumer get: H from Buffer[7]
……
```

```
$./consumer 4
12529 consumer get: A from Buffer[0]
12529 consumer get: C from Buffer[2]
12529 consumer get: E from Buffer[4]
12529 consumer get: G from Buffer[6]
……
```

在第一个生产者窗体中生产者 1 被再此唤醒输出:

```
12263 producer put: B to Buffer[1]
12263 producer put: D to Buffer[3]
12263 producer put: F to Buffer[5]
12263 producer put: H to Buffer[7]
……
```

在第二个生产者窗体中生产者 2 也被再此被唤醒输出

```
12264 producer put: A to Buffer[0]
12264 producer put: C to Buffer[2]
12264 producer put: E to Buffer[4]
12264 producer put: G to Buffer[6]
```

可以看到由于消费者进程读出了写入缓冲区的字符，生产者从新被唤醒继续向读过的缓冲区单元中同步的写入字符。

请用 Ctrl+c 将两生产者进程打断，观察两消费者进程是否在读空缓冲区后而阻塞。反之，请用 Ctrl+c 将两消费者进程打断，观察两生产者进程是否在写满缓冲区后而阻塞。

4.4 独立实验

抽烟者问题。假设一个系统中有三个抽烟者进程，每个抽烟者不断地卷烟并抽烟。抽烟者卷起并抽掉一颗烟需要有三种材料：烟草、纸和胶水。一个抽烟者有烟草，一个有纸，另一个有胶水。系统中还有两个供应者进程，它们无限地供应所有三种材料，但每次仅轮流提供三种材料中的两种。得到缺失的两种材料的抽烟者在卷起并抽掉一颗烟后会发信号通知供应者，让它继续提供另外的两种材料。这一过程重复进行。请用以上介绍的 IPC 同步机制编程，实现该问题要求的功能。

4.5 实验要求

根据示例实验程序和独立实验程序中观察和记录的信息，结合生产者 / 消费者问题和抽烟者问题的算法的原理，思考以下问题：

1. 说明真实操作系统中提供的并发进程同步机制是怎样实现和解决同步问题的？它们是怎样应用操作系统教材中讲解的进程同步原理的?
2. 对应教材中信号灯的定义，说明信号灯机制是怎样完成进程的互斥和同步的？其中信号量的初值和其值的变化的物理意义是什么?
3. 使用多于 4 个的生产者和消费者，以各种不同的启动顺序、不同的执行速率检测以上示例程序和独立实验程序是否都能满足同步的要求。

根据实验程序、调试过程和结果分析写出实验报告。

实验五、进程互斥实验

5.1 实验目的

进一步研究和实践操作系统中关于并发进程同步与互斥操作的一些经典问题的解法，加深对于非对称性互斥问题有关概念的理解。观察和体验非对称性互斥问题的并发控制方法。进一步了解 Linux 系统中 IPC 进程同步工具的用法，训练解决对该类问题的实际编程、调试和分析问题的能力。

5.2 实验说明

以下示例实验程序应能模拟一个读者 / 写者问题，它应能实现以下功能：

1. 任意多个读者可以同时读；
2. 任意时刻只能有一个写者写；
3. 如果写者正在写，那么读者就必须等待；
4. 如果读者正在读，那么写者也必须等待；
5. 允许写者优先；
6. 防止读者或写者发生饥饿。

为了能够体验 IPC 机制的消息队列的用法，本示例程序采用了 Theaker & Brookes 提出的消息传递算法。该算法中有一控制进程，带有 3 个不同类型的消息信箱，它们分别是：读请求信箱、写请求信箱和操作完成信箱。

读者需要访问临界资源时首先要向控制进程发送读请求消息，写者需要访问临界资源时也要先向控制进程发送写请求消息，在得到控制进程的允许消息后方可进入临界区读或写。读或写者在完成对临界资源的访问后还要向控制进程发送操作完成消息。

控制进程使用一个变量 count 控制读写者互斥的访问临界资源并允许写者优先。count 的初值需要一个比最大读者数还要大的数，本例取值为 100。当 count 大于 0 时说明没有新的读写请求，控制进程接收读写者新的请求，如果收到读者完成消息，对 count 的值加 1，如果收到写者请求消息， count 的值减 100，如果收到读者请求消息，对 count 的值减 1。 当 count 等于 0 时说明写者正在写，控制进程等待写者完成后再次令 count 的值等于 100。当 count 小于 100 时说明读者正在读，控制进程等待读者完成后对 count 的值加 1。

5.3 示例实验

我们可以利用上节实验中介绍的 IPC 机制中的消息队列来实验一下以上使用消息传递算法的读写者问题的解法，看其是否能够满足我们的要求。仍采用共享内存模拟要读写的对象，一写者向共享内存中写入一串字符后，多个读者可同时从共享内存中读出该串字符。

1. 在新建的文件夹中建立以下 ipc.h 头文件

```
/*
* Filename  : ipc.h
* copyright : (C) 2006 by zhonghonglie
* Function  : 声明 IPC 机制的函数原型和全局变量
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
```

```
#include <sys/msg.h>
#define BUFSZ 256
#define MAXVAL  100
#define STRSIZ  8
#define WRITERQUEST   1 //写请求标识
#define READERQUEST   2 //读请求标识
#define FINISHED  3 //读写完成标识
/*信号灯控制用的共同体*/
typedef union semuns {
    int val;
} Sem_uns;

/* 消 息 结 构 体 */
typedef struct msgbuf {
    long mtype;
    int mid;
} Msg_buf;
key_t buff_key;
int buff_num;
char *buff_ptr;
int shm_flg;

int quest_flg;
key_t quest_key;
int quest_id;

int respond_flg;
key_t respond_key;
int respond_id;

int get_ipc_id(char *proc_file, key_t key);

char *set_shm(key_t shm_key, int shm_num, int shm_flag);
int set_msq(key_t msq_key, int msq_flag);
int set_sem(key_t sem_key, int sem_val, int sem_flag);
int down(int sem_id);

int up(int sem_id);
```

2. 将消费者生产者问题实验中 ipc.c 文件拷贝到当前目录中。

3. 在当前目录中建立如下的控制者程序 control.c

```
/*
*
* Filename  : control.c
* copyright : (C) 2006 by zhonghonglie
* Function  : 建立并模拟控制者进程
*/
```

```c
#include "ipc.h"
int main(int argc, char *argv[]) {
    int i; int rate;
    int w_mid;
    int count = MAXVAL;
    Msg_buf  msg_arg;
    struct msqid_ds msg_inf;
    //建立一个共享内存先写入一串 A 字符模拟要读写的内容
    buff_key = 101;
    buff_num = STRSIZ + 1;
    shm_flg = IPC_CREAT | 0644;
    buff_ptr = (char *)set_shm(buff_key, buff_num, shm_flg);
    for (i = 0; i < STRSIZ; i++) buff_ptr[i] = 'A';
    buff_ptr[i] = '\0';

    //建立一条请求消息队列
    quest_flg = IPC_CREAT | 0644;
    quest_key = 201;
    quest_id = set_msq(quest_key, quest_flg);

    //建立一条响应消息队列
    respond_flg = IPC_CREAT | 0644;
    respond_key = 202;
    respond_id = set_msq(respond_key, respond_flg);

    //控制进程准备接收和响应读写者的消息
    printf("Wait quest \n");
    while (1) {
        //当 count 大于 0 时说明没有新的读写请求，查询是否有任何新请求
        if (count > 0) {
            quest_flg = IPC_NOWAIT; //以非阻塞方式接收请求消息
            if (msgrcv(quest_id, &msg_arg, sizeof(msg_arg), FINISHED, quest_flg) >= 0)
{
                //有读者完成
                count++;
                printf("%d reader finished\n", msg_arg.mid);
            } else if (msgrcv(quest_id, &msg_arg, sizeof(msg_arg), READERQUEST,
quest_flg) >= 0) {
                //有读者请求，允许读者读
                count --;
                msg_arg.mtype = msg_arg.mid;
                msgsnd(respond_id, &msg_arg, sizeof(msg_arg), 0);
                printf("%d quest read\n", msg_arg.mid);
            } else if (msgrcv(quest_id, &msg_arg, sizeof(msg_arg), WRITERQUEST,
quest_flg) >= 0) {
                //有写者请求
                w_mid = msg_arg.mid;
                count -= MAXVAL;
```

```
                //有读者正在读，则等待所有读者读完
                while(count < 0) {
                    //以阻塞方式接收读完成消息
                    msgrcv(quest_id, &msg_arg, sizeof(msg_arg), FINISHED, 0);
                    count ++;
                    printf("%d reader finish\n", msg_arg.mid);
                }

                //允许写者写
                msg_arg.mtype = w_mid;
                //恢复被读者覆盖的 mid
                msg_arg.mid = w_mid;
                msgsnd(respond_id, &msg_arg, sizeof(msg_arg), 0);
                printf("%d quest write \n", msg_arg.mid);
      }
        }
        //当 count 等于 0 时说明写者正在写，等待写完成
        if (count == 0) {
            //以阻塞方式接收消息.
            msgrcv(quest_id, &msg_arg, sizeof(msg_arg), FINISHED, 0);
            count = MAXVAL;
            printf("%d write finished\n", msg_arg.mid);
            if (msgrcv(quest_id, &msg_arg, sizeof(msg_arg), READERQUEST, quest_flg) >=
0) {
                //有读者请求，允许读者读
                count --;
                msg_arg.mtype = msg_arg.mid;
                msgsnd(respond_id, &msg_arg, sizeof(msg_arg), 0);
                printf("%d quest read\n", msg_arg.mid);
            }
        }
    }
    return EXIT_SUCCESS;
}
```

4. 在当前目录中建立如下的读者程序 reader.c

```
/*
* Filename  : reader.c
* copyright : (C) 2006 by zhonghonglie
* Function  : 建立并模拟读者进程
*/
#include "ipc.h"

int main(int argc, char *argv[]) {
    int i; int rate;
    Msg_buf msg_arg;
```

```
    //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
    if (argv[1] != NULL) rate = atoi(argv[1]);
    else rate = 3;
    //附加一个要读内容的共享内存
    buff_key = 101;
    buff_num = STRSIZ + 1;
    shm_flg = IPC_CREAT | 0644;
    buff_ptr = (char *)set_shm(buff_key, buff_num, shm_flg);
    //联系一个请求消息队列
    quest_flg = IPC_CREAT | 0644;
    quest_key = 201;
    quest_id =  set_msq(quest_key, quest_flg);
    //联系一个响应消息队列
    respond_flg = IPC_CREAT | 0644;
    respond_key = 202;
    respond_id =  set_msq(respond_key, respond_flg);
    //循环请求读
    msg_arg.mid = getpid();
    while (1) {
        //发读请求消息
        msg_arg.mtype = READERQUEST;
        msgsnd(quest_id, &msg_arg, sizeof(msg_arg), 0);
        printf("%d reader quest\n", msg_arg.mid);
        //等待允许读消息
        msgrcv(respond_id, &msg_arg, sizeof(msg_arg), msg_arg.mid, 0);
        printf("%d reading: %s\n", msg_arg.mid, buff_ptr); sleep(rate);
        //发读完成消息
        msg_arg.mtype = FINISHED;
        msgsnd(quest_id, &msg_arg, sizeof(msg_arg), quest_flg);
    }
    return EXIT_SUCCESS;
}
```

5. 在当前目录中建立如下的写者程序 writer.c

```
/*
* Filename  : writer.c
* copyright : (C) 2006 by zhonghonglie
* Function  : 建立并模拟写者进程
*/
#include "ipc.h"

int main(int argc, char *argv[]) {
    int i, j = 0; int rate;
    Msg_buf msg_arg;

    //可在在命令行第一参数指定一个进程睡眠秒数，以调解进程执行速度
    if (argv[1] != NULL) rate = atoi(argv[1]);
```

```
    else rate = 3;
    //附加一个要读内容的共享内存
    buff_key = 101;
    buff_num = STRSIZ + 1;
    shm_flg = IPC_CREAT | 0644;
    buff_ptr = (char *)set_shm(buff_key, buff_num, shm_flg);
    //联系一个请求消息队列
    quest_flg = IPC_CREAT | 0644;
    quest_key = 201;
    quest_id =  set_msq(quest_key, quest_flg);
    //联系一个响应消息队列
    respond_flg = IPC_CREAT | 0644;
    respond_key = 202;
    respond_id =  set_msq(respond_key, respond_flg);
    //循环请求写
    msg_arg.mid = getpid();
    while (1) {
        //发写请求消息
        msg_arg.mtype = WRITERQUEST;
        msgsnd(quest_id, &msg_arg, sizeof(msg_arg), 0);
        printf("%d writer quest\n", msg_arg.mid);
        //等待允许写消息
        msgrcv(respond_id, &msg_arg, sizeof(msg_arg), msg_arg.mid, 0);
        //写入 STRSIZ 个相同的字符
        for (i = 0; i < STRSIZ; i++) buff_ptr[i] = 'A' + j;
        j = (j + 1) % STRSIZ ; //按 STRSIZ 循环变换字符
        printf("%d writing: %s\n", msg_arg.mid, buff_ptr);
        sleep(rate);
        //发写完成消息
        msg_arg.mtype = FINISHED;
        msgsnd(quest_id, &msg_arg, sizeof(msg_arg), 0);
    }
    return EXIT_SUCCESS;
}
```

6. 在当前目录中建立如下 Makefile 文件

```
hdrs = ipc.h
c_src = control.c ipc.c
c_obj = control.o ipc.o
r_src = reader.c ipc.c
r_obj = reader.o ipc.o
w_src = writer.c ipc.c
w_obj = writer.o ipc.o
opts  = -g -c
all:  control reader writer
control: $(c_obj)
          gcc $(c_obj) -o control
```

```
control.o: $(c_src) $(hdrs)
          gcc $(opts) $(c_src)

reader: $(r_obj)
            gcc $(r_obj) -o reader
reader.o: $(r_src) $(hdrs)
            gcc $(opts) $(r_src)

writer: $(w_obj)
            gcc $(w_obj) -o writer
writer.o:$(w_src) $(hdrs)
            gcc $(opts) $(w_src)

clean:
        rm control reader writer *.o
```

7. 在当前目录中执行 make 命令编译连接，生成读写者，控制者程序：

```
$ make
gcc -g -c control.c ipc.c
gcc control.o ipc.o -o control
gcc -g -c reader.c ipc.c
gcc reader.o ipc.o -o reader
gcc -g -c writer.c ipc.c
gcc writer.o ipc.o -o writer
```

8. 可打开四个以上的终端模命令窗体，都将进入当前工作目录。先在一窗体中启动./control 程序：

```
$ ./control
Wait quest
```

现在控制进程已经在等待读写者的请求。

9. 再在另两不同的窗体中启动两个读者，一个让它以 1 秒的延迟快一些读，一个让它以 10 秒的延迟慢一些读：

```
$ ./reader 10

3903 reader quest

3903 reading: AAAAAAAA

......
```

```
$ ./reader 1
```

```
3904 reader quest

3904 reading: AAAAAAAA

3904 reader quest

3904 reading: AAAAAAAA

3904 reader quest

3904 reading: AAAAAAAA

3904 reader quest

......
```

现在可以看到控制进程开始响应读者请求，让多个读者同时进入临界区读：

```
Wait quest

3903 quest read

3904 quest read

3904 quest read

3904 reader finished

3904 reader finished

3904 quest read

3904 reader finished

3903 reader finished

......
```

10. 再在另一终端窗体中启动一个延迟时间为 8 秒的写者进程：

```
$ ./writer 8

3906 writer quest

3906 writing: AAAAAAAA

3906 writer quest
```

```
3906 writing: BBBBBBBB

3906 writer quest

3906 writing: CCCCCCCC

......
```

此时可以看到控制进程在最后一个读者读完后首先响应写者请求：

```
3906 quest write

3904 reader finish

3903 reader finish

3906 write finished

......
```

在写者写完后两个读者也同时读到了新写入的内容：

```
3903 reader quest

3903 reading: BBBBBBBB

3903 reader quest

3903 reading: CCCCCCCC

......

3904 reading: BBBBBBBB

3904 reader quest

3904 reading: CCCCCCCC

......
```

请仔细观察各读者和写者的执行顺序：可以看出在写者写时不会有读者进入，在有读者读时不会有写者进入，但一旦读者全部退出写者会首先进入。分析以上输出可以看出该算法实现了我们要求的读写者问题的功能。

11. 请按与以上不同的启动顺序、不同的延迟时间，启动更多的读写者。观察和分析是否仍能满足我们要求的读写者问题的功能。

12. 请修改以上程序，制造一个读者或写者的饥饿现象。观察什么是饥饿现象，说明为什么会发生这种现象。

5.4 独立实验

理发店问题：假设理发店的理发室中有 3 个理发椅子和 3 个理发师，有一个可容纳 4 个顾客坐等理发的沙发。此外还有一间等候室，可容纳 13 位顾客等候进入理发室。顾客如果发现理发店中顾客已满（超过 20 人），就不进入理发店，即直接离开。

在理发店内，理发师一旦有空就为坐在沙发上等待时间最长的顾客理发，同时空出的沙发让在等候室中等待时间最长的顾客就坐。顾客理完发后，可向任何一位理发师付款。但理发店只有一本现金登记册，在任一时刻只能记录一个顾客的付款。理发师在没有顾客的时候就坐在理发椅子上睡眠。理发师的时间就用在理发、收款、睡眠上。

请利用 linux 系统提供的 IPC 进程通信机制实验并实现理发店问题的一个解法。

5.5 实验要求

总结和分析示例实验和独立实验中观察到的调试和运行信息，思考以下问题：

1. 说明您对与解决非对称性互斥操作的算法有哪些新的理解和认识?
2. 为什么会出现进程饥饿现象?
3. 本实验的饥饿现象是怎样表现的?
4. 怎样解决并发进程间发生的饥饿现象?
5. 您对于并发进程间使用消息传递解决进程通信问题有哪些新的理解和认识?

根据实验程序、调试过程和结果分析写出实验报告。

实验六、死锁问题实验

6.1 实验目的

通过本实验观察死锁产生的现象，考虑解决死锁问题的方法。从而进一步加深对于死锁问题的理解。掌握解决死锁问题的几种算法的编程和调试技术。练习怎样构造管程和条件变量，利用管程机制来避免死锁和饥饿问题的发生。

6.2 实验说明

以下示例实验程序采用经典的管程概念模拟和实现了哲学家就餐问题。其中仍使用以上介绍的 IPC 机制实现进程的同步与互斥操作。为了利用管程解决死锁问题，本示例程序利用了 C++ 语言的类机制构造了哲学家管程，管程中的条件变量的构造利用了 linux 的 IPC 的信号量机制，利用了共享内存表示每个哲学家的当前状态。

6.3 示例实验

1. 在新建的文件夹中建立以下 dp.h 头文件

```
/*
    * Filename : dp.h
    * copyright : (C) 2006 by zhonghonglie
    * Function : 声明 IPC 机制的函数原型和哲学家管程类
*/

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/msg.h>
#include <sys/wait.h>

/*信号灯控制用的共同体*/
typedef union semuns { int val; } Sem_uns;

//哲学家的 3 个状态（思考、饥饿、就餐）
enum State { thinking, hungry, eating };

//哲学家管程中使用的信号量
class Sema {
public:
    Sema(int id);
    ~Sema();
    int down(); //信号量加 1
    int up();   //信号量减 1
private:
    int sem_id; //信号量标识符
};

//哲学家管程中使用的锁
class Lock {
public:
    Lock(Sema *lock);
    ~Lock();
    void close_lock();
    void open_lock();
private:
    Sema *sema; //锁使用的信号量
};

//哲学家管程中使用的条件变量
class Condition {
```

```
public:
    Condition(char *st[], Sema *sm);
    ~Condition();
    void Wait(Lock *lock, int i); //条件变量阻塞操作
    void Signal(int i); //条件变量唤醒操作
private:
    Sema *sema; //哲学家信号量
    char **state;//哲学家当前的状态
};

//哲学家管程的定义
class dp {
public:
    dp(int rate); //管程构造函数
    ~dp();
    void pickup(int i); //获取筷子
    void putdown(int i); //放下筷子
    //建立或获取 ipc 信号量的一组函数的原型说明
    int get_ipc_id(char *proc_file, key_t key);
    int set_sem(key_t sem_key, int sem_val, int sem_flag);
    //创建共享内存，放哲学家状态
    char *set_shm(key_t shm_key, int shm_num, int shm_flag);
private:
    int rate;      //控制执行速度
    Lock *lock;    //控制互斥进入管程的锁
    char *state[5];      //5 个哲学家当前的状态
    Condition *self[5]; //控制 5 个哲学家状态的条件变量
};
```

2. 在当前目录中建立如下的哲学家就餐程序 dp.cc

```
/*
    * Filename : dp.cc
    * copyright : (C) 2006 by zhonghonglie
    * Function : 哲学家就餐问题的模拟程序
*/

#include "dp.h"
Sema::Sema(int id) {
    sem_id = id;
}

Sema::~Sema() { }

/*
    * 信号灯上的 down/up 操作
    * semid:信号灯数组标识符
    * semnum:信号灯数组下标
    * buf:操作信号灯的结构
```

```
*/
int Sema::down() {
    struct sembuf buf;
    buf.sem_op = -1;
    buf.sem_num = 0;
    buf.sem_flg = SEM_UNDO;
    if ((semop(sem_id, &buf, 1)) < 0) {
        perror("down error ");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}

int Sema::up() {
    Sem_uns arg;
    struct sembuf buf;
    buf.sem_op = 1;
    buf.sem_num = 0;
    buf.sem_flg = SEM_UNDO;
    if ((semop(sem_id, &buf, 1)) < 0) {
        perror("up error ");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}

/* * 用于哲学家管程的互斥执行 */
Lock::Lock(Sema *s) {
    sema = s;
}
Lock::~Lock() {}

//上锁
void Lock::close_lock() {
    sema->down();
}
//开锁
void Lock::open_lock() {
    sema->up();
}

//用于哲学家就餐问题的条件变量
Condition::Condition(char *st[], Sema *sm) {
    state = st;
    sema = sm;
}

/*
    * 左右邻居不在就餐，条件成立，状态变为就餐
```

```
    * 否则睡眠，等待条件成立
*/
void Condition::Wait(Lock *lock, int i) {
    if ((*state[(i + 4) % 5] != eating) &&
            (*state[i] == hungry) &&
            (*state[(i + 1) % 5] != eating)) {
        *state[i] = eating;//拿到筷子，进就餐态
    } else {
        std::cout << "p" << i + 1 << ":" << getpid() << " hungry\n";
        lock->open_lock();//开锁
        sema->down();//没拿到，以饥饿态等待
        lock->close_lock();//上锁
    }
}

/*
    *左右邻居不在就餐，则置其状态为就餐，
    *将其从饥俄中唤醒。否则什么也不作。
*/
void Condition::Signal(int i) {
    if ((*state[(i + 4) % 5] != eating) &&
            (*state[i] == hungry) &&
            (*state[(i + 1) % 5] != eating)) {
        //可拿到筷子，从饥饿态唤醒进就餐态
        sema->up();
        *state[i] = eating;
    }
}

/*
    * get_ipc_id() 从/proc/sysvipc/文件系统中获取 IPC 的 id 号
    * pfile: 对应/proc/sysvipc/目录中的 IPC 文件分别为
    * msg-消息队列,sem-信号量,shm-共享内存
    * key: 对应要获取的 IPC 的 id 号的键值
*/
int dp::get_ipc_id(char *proc_file, key_t key) {
    #define BUFSZ 256
    FILE *pf;
    int i, j;
    char line[BUFSZ], colum[BUFSZ];
    if ((pf = fopen(proc_file, "r")) == NULL) {
        perror("Proc file not open");
        exit(EXIT_FAILURE);
    }
    fgets(line, BUFSZ, pf);
    while (!feof(pf)) {
        i = j = 0;
        fgets(line, BUFSZ, pf);
        while (line[i] == ' ') i++;
```

```
        while (line[i] != ' ') colum[j++] = line[i++];
        colum[j] = '\0';
        if (atoi(colum) != key)
            continue;
        j = 0;
        while (line[i] == ' ') i++;
        while (line[i] != ' ') colum[j++] = line[i++];
        colum[j] = '\0';
        i = atoi(colum);
        fclose(pf);
        return i;
    }
    fclose(pf);
    return -1;
}

/*
    * set_sem 函数建立一个具有 n 个信号灯的信号量
    * 如果建立成功，返回 一个信号量的标识符 sem_id
    * 输入参数:
        * sem_key 信号量的键值
        * sem_val 信号量中信号灯的个数
        * sem_flag 信号量的存取权限
*/
int dp::set_sem(key_t sem_key, int sem_val, int sem_flg) {
    int sem_id;
    Sem_uns sem_arg;
    //测试由 sem_key 标识的信号量是否已经建立
    if ((sem_id = get_ipc_id("/proc/sysvipc/sem", sem_key)) < 0 ) {
        //semget 新建一个信号灯,其标号返回到sem_id
        if ((sem_id = semget(sem_key, 1, sem_flg)) < 0) {
            perror("semaphore create error");
            exit(EXIT_FAILURE);
        }
    }
    //设置信号量的初值
    sem_arg.val = sem_val;
    if (semctl(sem_id, 0, SETVAL, sem_arg) < 0) {
        perror("semaphore set error");
        exit(EXIT_FAILURE);
    }
    return sem_id;
}

/*
  * set_shm 函数建立一个具有 n 个字节 的共享内存区
  * 如果建立成功，返回 一个指向该内存区首地址的指针 shm_buf
  * 输入参数:
  * shm_key 共享内存的键值
```

```
  * shm_val 共享内存字节的长度
  * shm_flag 共享内存的存取权限
*/
char *dp::set_shm(key_t shm_key, int shm_num, int shm_flg) {

    int i, shm_id;
    char *shm_buf;

    //测试由 shm_key 标识的共享内存区是否已经建立
    if ((shm_id = get_ipc_id("/proc/sysvipc/shm", shm_key)) < 0) {
        //shmget 新建 一个长度为 shm_num 字节的共享内存
        if ((shm_id = shmget(shm_key, shm_num, shm_flg)) < 0) {
            perror("shareMemory set error");
            exit(EXIT_FAILURE);
        }
        //shmat 将由 shm_id 标识的共享内存附加给指针 shm_buf
        if ((shm_buf = (char *)shmat(shm_id, 0, 0)) < (char *)0) {
            perror("get shareMemory error");
            exit(EXIT_FAILURE);
        }
        for (i = 0; i < shm_num; i++)
            shm_buf[i] = 0; //初始为 0
    }
    //共享内存区已经建立,将由 shm_id 标识的共享内存附加给指针 shm_buf
    if ((shm_buf = (char *)shmat(shm_id, 0, 0)) < (char *)0) {
        perror("get shareMemory error");
        exit(EXIT_FAILURE);
    }
    return shm_buf;
}

//哲学家就餐问题管程构造函数
dp::dp(int r) {
    int ipc_flg = IPC_CREAT | 0644;
    int shm_key = 220;
    int shm_num = 1;
    int sem_key = 120;
    int sem_val = 0;
    int sem_id;
    Sema *sema;
    rate = r;

    //5 个中同时可以有 2 个在就餐，建立一个初值为 2 的用于锁的信号灯
    if ((sem_id = set_sem(sem_key++, 2, ipc_flg)) < 0) {
        perror("Semaphor create error");
        exit(EXIT_FAILURE);
    }
    sema = new Sema(sem_id);
    lock = new Lock(sema);
```

```
    for (int i = 0; i < 5; i++) {
        //为每个哲学家建立一个条件变量和可共享的状态 //初始状态都为思考
        if ((state[i] = (char *)set_shm(shm_key++, shm_num, ipc_flg)) == NULL) {
            perror("Share memory create error");
            exit(EXIT_FAILURE);
        }
        *state[i] = thinking;
        //为每个哲学家建立初值为 0 的用于条件变量的信号灯
        if ((sem_id = set_sem(sem_key++, sem_val, ipc_flg)) < 0) {
            perror("Semaphor create error");
            exit(EXIT_FAILURE);
        }
        sema = new Sema(sem_id);
        self[i] = new Condition(state, sema);
    }
}

//获取筷子的操作
//如果左右邻居都在就餐，则以饥饿状态阻塞
//否则可以进入就餐状态
void dp::pickup(int i) {
    lock->close_lock();//进入管程，上锁
    *state[i] = hungry; //进饥饿态
    self[i]->Wait(lock, i); //测试是否能拿到两只筷子
    std::cout << "p" << i + 1 << ":" << getpid() << " eating\n";
    sleep(rate); //拿到，吃 rate 秒
    lock->open_lock();//离开管程，开锁
}

//放下筷子的操作
//状态改变为思考，如左右邻居有阻塞者则唤醒它
void dp::putdown(int i) {
    int j;
    lock->close_lock();//进入管程，上锁
    *state[i] = thinking; //进思考态
    j = (i + 4) % 5;
    self[j]->Signal(j); //唤醒左邻居
    j = (i + 1) % 5;
    self[j]->Signal(j); //唤醒右邻居
    lock->open_lock();//离开管程，开锁
    std::cout << "p" << i + 1 << ":" << getpid() << " thinking\n";
    sleep(rate); //思考 rate 秒
}

dp::~dp() { }

// 哲学家就餐问题并发执行的入口
```

```
int main(int argc, char *argv[]) {
    dp *tdp; //哲学家就餐管程对象的指针
    int pid[5]; //5 个哲学家进程的进程号
    int rate;

    rate = (argc > 1) ? atoi(argv[1]) : 3 ;

    tdp = new dp(rate); //建立一个哲学家就餐的管程对象

    pid[0] = fork(); //建立第一个哲学家进程
    if (pid[0] < 0) {
        perror("p1 create error");
        exit(EXIT_FAILURE);
    } else if (pid[0] == 0) { //利用管程模拟第一个哲学家就餐的过程
        while (1) {
            tdp->pickup(0);//拿起筷子
            tdp->putdown(0);//放下筷子
        }
    }
    pid[1] = fork(); //建立第二个哲学家进程
    if (pid[1] < 0) {
        perror("p2 create error");
        exit(EXIT_FAILURE);
    } else if (pid[1] == 0) { //利用管程模拟第二个哲学家就餐的过程
        while (1) {
            tdp->pickup(1);//拿起筷子
            tdp->putdown(1);//放下筷子
        }
    }
    pid[2] = fork(); //建立第三个哲学家进程
    if (pid[2] < 0) {
        perror("p3 create error");
        exit(EXIT_FAILURE);
    } else if (pid[2] == 0) { //利用管程模拟第三个哲学家就餐的过程
        while (1) {
            tdp->pickup(2);//拿起筷子
            tdp->putdown(2);//放下筷子
        }
    }
    pid[3] = fork(); //建立第四个哲学家进程
    if (pid[3] < 0) {
        perror("p4 create error");
        exit(EXIT_FAILURE);
    } else if (pid[3] == 0) { //利用管程模拟第四个哲学家就餐的过程
        while (1) {
            tdp->pickup(3);//拿起筷子
            tdp->putdown(3);//放下筷子
        }
    }
```

```
    pid[4] = fork(); //建立第五个哲学家进程
    if (pid[4] < 0) {
        perror("p5 create error");
        exit(EXIT_FAILURE);
    } else if (pid[4] == 0) { //利用管程模拟第五个哲学家就餐的过程
        while (1) {
            tdp->pickup(4);//拿起筷子
            tdp->putdown(4);//放下筷子
        }
    }
    return 0;
}
```

3. 在新建文件夹中建立以下 Makefile 文件

```
head = dp.h
srcs = dp.cc
objs = dp.o
opts = -w -g -c
all: dp
dp: $(objs)
  g++ $(objs) -o dp
dp.o: $(srcs) $(head)
  g++ $(opts) $(srcs)
clean:
  rm dp *.o
```

4. 在新建文件夹中执行 make 命令编译连接生成可执行的哲学家就餐程序

```
$ make
g++ -w -g -c dp.cc
g++ dp.o -o dp
```

5. 执行的哲学家就餐程序 dp

```
$ ./dp 1
p1:4524 eating
p2:4525 hungry
p3:4526 eating
p4:4527 hungry
p5:4528 hungry
p1:4524 thinking
p5:4528 eating
p3:4526 thinking
p2:4525 eating
p1:4524 hungry
p5:4528 thinking
p4:4527 eating
```

```
p3:4526 hungry
p1:4524 eating
p2:4525 thinking
p5:4528 hungry
……
```

可以看到 5 个哲学家进程在 3 种状态中不断的轮流变换，且连续的 5 个输出中不应有多于 2 个的状态为 eating，同一进程号不应有两个连续的输出。您可用不同的执行速率长时间的让它们执行，观察是否会发生死锁或饥饿现象。如果始终没有产生死锁和饥饿现象，可用 kill 按其各自的进程号终止它们的执行。

6. 请修改以上 dp.cc 程序，制造出几种不同的的死锁现象和饥饿现象，记录并分析各种死锁、饥饿现象和产生死锁、饥饿现象的原因。

6.4 独立实验

在两个城市南北方向之间存在一条铁路，多列火车可以分别从两个城市的车站排队等待进入车道向对方城市行驶，该铁路在同一时间，只能允许在同一方向上行车，如果同时有相向的火车行驶将会撞车。

请构造一个管程，模拟实现两个方向行车，并满足以下要求：

1. 可以设置铁路同方向最多行车数
2. 不会出现撞车
3. 不会出现某一方向火车长时间等待

6.5 实验要求

总结和分析示例实验和独立实验中观察到的调试和运行信息，思考以下问题：

1. 分析示例实验是否真正模拟了哲学家就餐问题?
2. 为什么示例程序不会产生死锁?
3. 为什么会出现进程死锁和饥饿现象?
4. 怎样利用实验造成和表现死锁和饥饿现象?
5. 管程能避免死锁和饥饿 的机理是什么?
6. 您对于管程概念有哪些新的理解和认识?
7. 条件变量和信号量有何不同?
8. 为什么在管程中要使用条件变量而不直接使用信号量来达到进程同步的目的?
9. 示例实验中构造的管程中的条件变量是一种什么样式的？其中的锁起了什么样的作用?
10. 你的独立实验程序是怎样解决单行道问题的?
11. 您是怎样构造管程对象的?

根据实验程序、调试过程和结果分析写出实验报告。