

# 操作系统 实验指导书

山东大学  
计算机科学与技术学院  
计算机系统教研室  
操作系统教研组  
2021 年 10 月修订

# 目 录

目 录 .....	I
实验 1 进程控制 .....	1
1.1 实验目的 .....	1
1.2 实验说明 .....	1
1.2.1 与进程创建、执行有关的系统调用说明 .....	1
1.2.2 与进程控制有关的系统调用说明 .....	4
1.3 示例实验 .....	5
1.4 独立实验 .....	9
1.5. 实验要求 .....	9
实验 2 线程和管道通信 .....	9
2.1 实验目的 .....	9
2.2 实验说明 .....	10
2.2.1 线程有关的函数或系统调用 .....	10
2.2.2 线程属性结构体 pthread_attr_t .....	12
2.2.3 管道通信机制 .....	17
2.3 示例实验 .....	18
2.4 独立实验 .....	23
2.5 实验要求 .....	23
实验 3 进程综合实验 (Shell) .....	23
3.1 实验目的 .....	23
3.2 实验说明 .....	23
3.2.1 将程序在后台运行 .....	24
3.2.2 文件描述符与 I/O 重定向 .....	26
3.2.3 系统调用 dup() .....	26
3.2.4 系统调用 dup2() .....	27
3.2.5 重定向与管道命令 .....	28
3.3 简单的 shell 框架 .....	29
3.4 独立实验 .....	32
3.5 实验要求 .....	32
实验 4 进程同步 .....	32
4.1 实验目的 .....	32
4.2 实验说明 .....	33
4.2.1 共享内存 .....	33
4.2.2 信号量 .....	39
4.2.3 消息队列 .....	46
4.2.4 常用 IPC 命令 .....	52
4.3 示例实验 .....	54
4.4 独立实验 .....	64
4.5 实验要求 .....	64

实验 5 进程互斥 .....	64
5.1 实验目的 .....	64
5.2 实验说明 .....	64
5.3 示例实验 .....	65
5.4 独立实验 .....	73
5.5 实验要求 .....	74
实验 6 死锁问题 .....	74
6.1 实验目的 .....	74
6.2 实验说明 .....	74
6.3 示例实验 .....	74
6.4 独立实验 .....	84
6.5 实验要求 .....	84
实验 7 存储管理 .....	85
7.1 实验目的 .....	85
7.3 实验说明 .....	85

# 实验 1 进程控制

## 1.1 实验目的

加深对于进程并发执行概念的理解。实践并发进程/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 系统中进程/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

### 1.2.1 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副本：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、已打开的文件描述符、工作目录和资源限制等。子进程可以通过系统调用族 `exec()` 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。

#### 1、`fork()`系统调用

```
#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。

范例

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
int main() {
```

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

```
    if(pid < 0) { //执行不成功，返回-1
```

```
        printf("fork failed\n");
```

```
    } else if (pid == 0) { //子进程执行代码
```

```
        printf("This is the child process\n");
```

```
    } else { //父进程执行代码
```

```
        printf("This is the parent process\n");
```

```
    }
```

```
    return 0;
```

```
}
```

*/\* 由于父子进程并发执行，因此，两个字符串的输出顺序取决于操作系统对父子进程的调度顺序，执行结果不唯一*

*下面是一种可能的输出：*

*This is the parent process*

*This is the child process*  
\*/

## 2、exec()系统调用

系统调用 exec()有一组 6 个不同名称、不同格式的函数，其中后面的示例程序中引用了 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。由于 exec() 用新的程序覆盖了原进程的地址空间，因此，执行 exec()后，不会再返回到调用进程，即不会再返回到调用 exec()的下条语句继续执行。如果 exec 调用失败，它会返回-1。

相比于系统调用 execve(), execvp()更加方便使用。

execvp() 系统调用语法：

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

- path: 要装入的新的可执行文件名字符串。
- argv[]: 要传递给新执行程序的完整的命令参数列表(可以为空)。

execvp() 从系统的 PATH 环境变量所定义的目录中查找可执行文件 file，找到后便执行该文件，然后将第二个参数 argv 传给 file。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
*/
```

## 3、wait()、waitpid()系统调用

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

pid\_t waitpid(pid\_t pid,int \*status,int option);

参数说明:

status 用于保留子进程的退出状态

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

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。

wait() /waitpid()会阻塞当前执行的进程, 直到有信号来到或子进程结束。如果在调用 wait() /waitpid()时子进程已经结束, 则 wait() /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起返回。如果不关心结束状态值, 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;
}
/* 输出结果唯一, 如果父进程先执行 wait 则会被阻塞, 切换子进程执行
This is the child process
This is the parent process
*/
```

#### 4、getpid()、getppid()系统调用

```
#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。

### 1.2.2 与进程控制有关的系统调用说明

可以通过信号（signal）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

注：默认操作是内核态调用 exit() 退出。

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 (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：利用 shell 的 kill 命令，以及在程序中利用系统调用 kill。kill 能够发送除杀死一个进程 (SIGKILL、SIGTERM、SIGQUIT) 之外的其它信号，例如键盘中断 (Ctrl+C) 信号 SIGINT，进程暂停 (Ctrl+Z) 信号 SIGTSTP 等。

系统调用 pause 会挂起（阻塞）调用进程，直到一个任意信号到来后再继续运行。

系统调用 sleep 会挂起（阻塞）调用进程，睡眠指定的秒数后，或一个它可以响应的信号到来后继续执行。

每个进程都可以使用 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。

#### 1、kill 系统调用

```
#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。

#### 2、pause 系统调用

```
#include <unistd.h>
```

```
int pause(void);
```

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

### 3、sleep 系统调用

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

seconds: 指定进程睡眠的秒数。  
如果时间到了指定的秒数，sleep 返回 0。

### 4、signal 系统调用

```
#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。

### 5、perror 系统调用

```
#include <stdio.h>
#include <errno.h>
void perror(const char *s) ;
```

perror 主要用于跟踪一个系统调用是否成功执行，在调试程序时使用。

若 perror 之前最近的一个系统调用成功执行，标准设备 (stderr) 将输出参数 s 所给出的字符串；如果该系统调用没有成功执行，标准设备 (stderr) 首先输出参数 s 所指定的字符串，然后输出该系统调用没有成功执行的原因，并输出错误的原因。该错误原因依照全局变量 errno 的值来决定要输出的字符串。

在库函数中有个全局变量 errno，每个 errno 值对应着以字符串表示的相应错误类型。当某些系统调用没有成功执行时，该函数重新设置了 errno 的值。

## 1.3 示例实验

以下实验示例程序应实现一个类似于 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，然后和父进程并发执行。

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) 2022 by
 * 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}; //子进程要缺省执行的命令

signal(SIGINT,(sighandler_t)sigcat); //注册一个本进程处理键盘中断的函数
perror("SIGINT"); //如果系统调用 signal 成功执行，输出”SIGINT”，否则，
                  //输出”SIGINT”及出错原因
pid=fork(); //建立子进程
if(pid<0) // 建立子进程失败?
{
    printf("Create Process fail!\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\nMy father is %d\n",getpid(),getppid());
    pause(); //暂停，等待键盘中断信号唤醒
    //子进程被键盘中断信号唤醒继续执行
    printf("%d child will Running: \n",getpid()); //
    if(argv[1] != NULL){
        //如果运行该程序是用命令行参数指定了子进程要执行的命令
        //则执行输入行参数中给定的程序
        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{
        //如果命令行参数没给出子进程要执行的程序，即运行程序时没有命令行参数
        //则执行缺省的命令，即 ls -a
        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
{
    printf("\nI am Parent process %d\n",getpid()); //报告父进程进程号
    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("%d Waiting for child done.\n\n",getpid());
        waitpid(pid,&status,0);    //等待子进程结束
        printf("\nMy child exit! status = %d\n\n",status);
    }
    else{
        //如果在命令行上没输入子进程要执行的命令
        //唤醒子进程，与子进程并发执行不等待子进程执行结束
        sleep(5); //思考：如果去掉这条语句，可能会出现什么现象
        if(kill(pid,SIGINT) >= 0)
            printf("%d Wakeup %d child.\n",getpid(),pid) ;
        printf("%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. 建立以下项目管理文件 **Makefile**

```

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

```

$ gmake （注：有的系统用 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 Child process 4113
My father is 4112

I am Parent process 4112
Wakeup 4113 child.
4112 don't Wait for child done.

4113 Process continue
4113 child will Running: /bin/ls -a
... Makefile pctl pctl.c pctl.h pctl.o
$

```

以上程序的输出说明父进程 4112 创建了一个子进程 4113，子进程执行被暂停。父进程向子进程发出键盘中断信号唤醒子进程并与子进程并发执行。父进程并没有等待子进程的结束继续执行先行结束了 (此时的子进程成为了孤儿进程，不会有父进程为它清理退出状态了)。而子进程继续执行，它变成了列出当前目录所有文件名的命令 `ls -a`。在完成了列出文件名命令之后，子进程的执行也结束了。此时子进程的退出状态将有初始化进程为它清理。

## 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 Child process 4223 My father is 4222

I am Parent process 4222
4222 Waiting for child done.

```

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  TTYTIME  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
4223 child will 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! 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）最后父进程也结束执行。

## 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。父进程每隔 3 秒重复建立两个子进程，首先创建的子进程让其执行 `ls` 命令，之后创建的子进程让其执行 `ps` 命令，并控制 `ps` 命令总在 `ls` 命令之前执行。

## 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，说明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？在真实的操作系统中它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 2 线程和管道通信

## 2.1 实验目的

通过 Linux 系统中线程和管道通信机制的实验，熟悉 `pthread` 线程库的使用，加深对于线程控制和管道通信概念的理解，观察和体验并发线程间的通信和协作的效果，练习基于 `pthread` 线程库、利用无名管道进行线程通信的编程和调试技术。

## 2.2 实验说明

### 2.2.1 线程有关的函数或系统调用

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、文件描述符和信号量，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 `pthread` 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。`Pthread` 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。`pthread` 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，同时继续执行其它处理。
- 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，可以认为是系统调用，也可以认为是 `pthread` 库的函数调用。

#### 1、`pthread_attr_init()`

```
#include <pthread.h>
```

```
int pthread_attr_init (pthread_attr_t* attr);
```

功能：创建线程之前，通常先调用 `pthread_attr_init()` 来初始化线程属性变量。

传入参数：attr：线程属性结构体

返回值：成功：0

失败：-1

其中，线程属性 attr 是一个结构体 `pthread_attr_t` 类型，详见 2.2.2 节。

#### 2、`pthread_create()`

```
#include <pthread.h>
```

```
int pthread_create ( pthread_t *thread,  
                    pthread_attr_t *attr,
```

```
void *(*start_routine) (void *),  
void *arg);
```

功能：pthread\_create 函数创建一个新的线程。

传入参数：thread 保存新线程的标识符

attr 决定了线程应用哪种线程属性，若将 attr 设置为 NULL，  
则使用默认的属性设置（详见 2.2.2 节）

start\_routine 是一个指向新建线程中要执行的函数的指针

arg 是新线程函数携带的参数

返回值：执行成功返回 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。

### 3、pthread\_join()

```
#include <pthread.h>
```

```
int pthread_join(pthread_t tid, void **thread_return);
```

函数 pthread\_join() 阻塞当前线程（调用 pthread\_join 函数的线程，一般是父线程），直到被等待的线程（一般是子线程）tid 结束为止，当 pthread\_join() 函数返回时，被等待线程（子线程）的资源被收回。如果线程已经结束，该函数会立即返回。

第一个参数 tid 为被等待的线程（子线程）标识符，第二个参数 thread\_return 为一个用户定义的指针，它可以用来存储被等待线程（子线程）的返回值。这个函数是一个线程阻塞的函数，调用它的线程（父线程）将阻塞自己，一直等待到被等待的线程（子线程）结束为止，当函数返回时，被等待线程（子线程）tid 的资源被收回。

关于 pthread\_join 的几点注记：

（1）在调用 pthread\_create() 函数创建线程时，如果第二个参数 attr 的值为 NULL，则所创建的子线程采用默认的属性，其中的 detach 属性默认是 PTHREAD\_CREATE\_JOINABLE，简称为 joinable。

对于 joinable 类型的线程，当其运行结束后，线程所占用的资源不会被自动释放，也就不会被重新分配给其它线程，需要父线程调用 pthread\_join() 将其收回。

父线程调用 pthread\_join() 函数时，将阻塞自己，等待子线程结束。当子线程结束时，父线程被唤醒，收回子线程所占的资源。

如果设置了子线程的 detach 属性是 PTHREAD\_CREATE\_DETACHED，简称为 detached，则子线程执行结束后，自动释放资源，父线程不需要调用 pthread\_join() 函数等待子线程结束。如果父线程中调用了 pthread\_join() 函数也不起作用，父线程也不会阻塞自己等待子线程结束。（详见 2.2.2 节）

(2) 一个线程不能被多个线程等待。也就是说，对于一个要等待的线程 `tid`，只能有一个线程调用一次 `pthread_join`，等待线程 `tid` 结束。如果多个线程调用了 `pthread_join` 等待同一个线程结束，或一个线程中调用了多次 `pthread_join` 等待同一个线程结束，则只有一个 `pthread_join` 能正确返回，其它的调用将返回 `ESRCH` 错误（理论上）。

从上可以看出，该函数的功能类似于进程的系统调用 `waitpid`。

#### 4、`pthread_exit()`

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

一个线程使用 `pthread_exit` 函数退出或终止自身。`pthread_exit` 函数的实现中，`pthread_exit` 调用了清除处理函数 `pthread_cleanup_push()`，然后中止当前进程的执行，返回值保持到 `retval` 中。`retval` 可以由父线程或其它线程通过 `pthread_join` 来检索。

结束一个线程的执行有两种途径：一种是在线程中调用函数 `pthread_exit()` 结束自身，另一种是线程的之行体运行结束，即创建线程时指定其运行的函数 `start_routine` 执行完毕。

#### 2.2.2 线程属性结构体 `pthread_attr_t`

头文件 `pthread.h` 中声明了一个线程的属性结构体 `pthread_attr_t`，其成员变量是要创建线程的各种属性值。在创建线程之前，使用 `pthread_attr_init()` 初始化线程的属性值，然后，可以使用一组函数 `pthread_attr_getxxx()` 和 `pthread_attr_setxxx()` 获取或设置相应的属性值。

该结构体声明如下：

```
typedef struct
{
    int            detachstate; //线程分离属性
    int            schedpolicy; //线程调度策略
    struct sched_param schedparam; //线程的优先级
    int            inheritsched; //线程的继承性
    int            scope;        //线程的作用域
    size_t         gurdsize;     //线程栈尾部的警戒缓冲区大小
    int            stackaddr_set; //线程栈地址集
    void           *stackaddr;   //线程栈位置
    size_t         stacksize;    //线程栈大小
} pthread_attr_t;
```

在调用 `pthread_create` 创建线程时，如果把第二个参数（即 `pthread_attr_t *attr`，）设置为 `NULL` 的，新建的子线程将采用结构体 `pthread_attr_t` 中默认的属性配置。

常用的几个属性介绍如下：

（1）**detach 属性**：\_\_detachstate。主要包括 `PTHREAD_CREATE_JOINABLE` 和 `PTHREAD_CREATE_DETACHED` 两种属性，缺省为 **`PTHREAD_CREATE_JOINABLE`**（该宏的值为 0）。

对于 `PTHREAD_CREATE_JOINABLE` 属性，当子线程执行结束后，不会自动释放其所占用栈与线程描述符等资源（大约 8KB），也就没有真正退出。需要创建它的父线程调用 `pthread_join` 来等待子线程结束，返回子线程的状态，并释放子线程所占资源。（参见 `pthread_join` 函数）。

如果设置 detach 属性为 `PTHREAD_CREATE_DETACHED`（该宏的值为 1），则子线程在退出时自行释放所占用的资源，主线程中不需要调用 `pthread_join` 来同步子线程。

创建子线程之前，父线程可通过函数 `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)` 获取为子线程设置的 detach 属性，成功返回 0，若失败返回 -1。其中第二个参数 `*detachstate=0` 表示 joinable，1 表示 detached。

可通过下述三种方法设置线程的 detach 属性为 detached：

（a）父线程调用 `pthread_create` 创建一个子线程之前，利用 `pthread_attr_setdetachstate` 函数设置。

例如考察如下父线程中的程序片段：

```
int detachstate;
pthread_attr_t attr; //获取默认属性
pthread_attr_getdetachstate(&attr,&detachstate); //获取默认的 detach 属性
//设置要创建的线程的 detach 属性为 detached;
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, runner, param); //创建线程
.....
//pthread_join(tid,NULL); //不再需要该语句
```

注：大多数资料介绍，设置子线程的 detach 属性为 detached 后，父线程不能再使用 `pthread_join()` 同步子线程。

将子线程的 detach 属性由默认的 JOINABLE 修改为 DETACHED，主线程执行到 `pthread_join(tid,NULL)` 时，不会再阻塞自己（也就不需要该语句）。

函数 `pthread_attr_getdetachstate()` 与 `pthread_attr_setdetachstate()` 的原型为：

```
#include <pthread.h>
```



```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

(b) 父线程调用 `pthread_create()` 创建线程后，父线程可调用 `pthread_detach(thread_id)` 函数将线程标识 `thread_id` 所指定的子线程的 `detach` 属性设置为 `PTHREAD_CREATE_DETACHED`。

例如父线程中的程序片段：

```
pthread_attr_init(&attr);
pthread_create(&tid, &attr, runner, param);
pthread_detach(tid); //设置线程 tid 为 PTHREAD_CREATE_DETACHED
.....
//pthread_join(tid,NULL); //不再需要该语句
```

(c) 一般在子线程的第一条语句调用函数 `pthread_detach(pthread_self())`，将子线程设置为 `PTHREAD_CREATE_DETACHED`。

其中，函数 `pthread_self()` 的原型为：`pthread_t pthread_self(void)`，其作用是获得线程自身的 ID。（`pthread_t` 的类型为 `unsigned long int`，输出格式符：`%lu`）。

需要注意的是，一旦将一个线程设置为 `PTHREAD_CREATE_DETACH`，则不能再将其恢复到 `PTHREAD_CREATE_JOINABLE`。

应用场景：例如在 Web 服务器中，当主线程为每个新来的链接请求创建一个子线程进行处理时，因为主线程还要继续处理之后到来的链接，因此主线程并不希望因为调用 `pthread_join` 因等待子线程而阻塞自己，这时可以利用上述三种方法之一将子线程设置为 `PTHREAD_CREATE_DETACHED`。

**(2) policy 属性：**`__schedpolicy`。表示新线程的调度策略，主要包括 `SCHED_OTHER`（正常、非实时）`SCHED_FIFO`（实时、先入先出）、和 `SCHED_RR`（实时、轮转法）三种，**缺省为 `SCHED_OTHER`**。一般情况下，其它两种调度策略仅对超级用户有效。运行时可以利用 `pthread_attr_getschedpolicy()` 与 `pthread_attr_setschedpolicy()` 来获取或改变线程的调度策略。

三个宏 `SCHED_OTHER`、`SCHED_FIFO` 与 `SCHED_RR` 的值分别对应 0,1,2、

两个函数的原型为：

```
#include <pthread.h>
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

例如考察如下父线程程序片段：

```
pthread_attr_init(&attr); //获取默认属性
```

```

int policy;
pthread_attr_getschedpolicy(&attr, &policy); //获取默认的调度策略
printf("policy=%d\n",policy); //默认调度策略为 SCHED_OTHER, 输出 0
pthread_attr_setschedpolicy(&attr,SCHED_FIFO ); //修改调度策略
pthread_create(&tid, &attr, runner, param); //创建线程
pthread_attr_setschedpolicy(&attr, SCHED_RR ); //修改调度策略

```

(3) 优先级: `__schedparam`。是一个 `struct sched_param` 结构, 目前仅有一个 `sched_priority` 整型变量表示线程的优先级。这个参数仅当调度策略为实时 (即 `SCHED_RR` 或 `SCHED_FIFO`) 时才有效, 并可以在运行时通过 `pthread_setschedparam()` 函数来改变, **缺省为 0**。

函数 `pthread_attr_getschedparam()` 获取线程优先级, 原型为:

```

#include <pthread.h>
int pthread_attr_getschedparam (pthread_attr_t* attr, struct sched_param* param);

```

函数传入值: `attr`: 线程属性;  
                   `param`: 线程优先级;

函数 `pthread_attr_setschedparam` 设置线程 `schedparam` 属性, 即调用的优先级, 原型为:

```

#include <pthread.h>
int pthread_attr_setschedparam (pthread_attr_t* attr, struct sched_param* param);

```

函数传入值: `attr`: 线程属性。  
                   `param`: 线程优先级。

头文件 `pthread.h` 中声明结构体 `sched_param` 如下:

```

struct sched_param {
    int sched_priority;
}

```

考察下述父线程的程序片段:

```

#include <pthread.h>
pthread_attr_init(&attr); //获取默认属性
//声明优先级结构体变量, 仅含有一个成员变量 int sched_priority;
struct sched_param param;

pthread_attr_getschedparam (&attr, &param); //获取默认优先级
printf("sched_priority=%d\n",param.sched_priority); //默认优先级, 输出 0
pthread_attr_setschedpolicy(&attr,SCHED_FIFO ); //设置调度策略

param.sched_priority=2;
pthread_attr_setschedparam (&attr, &param); //设置优先级为 2

pthread_attr_getschedparam (&attr, &param);

```

```
printf("\nsched_priority=%d\n",param.sched_priority); //输出 2

/* create the thread*/
pthread_create(&tid, &attr, runner, argv[1]);

pthread_attr_setschedpolicy(&attr,SCHED_RR ); //修改调度策略

param.sched_priority=5;
pthread_attr_setschedparam (&attr, &param); //设置优先级为 5

pthread_attr_getschedparam (&attr, &param);
printf("\nsched_priority=%d\n",param.sched_priority); //输出 5
```

**(4) 继承属性：**\_\_inheritsched。有两种值可供选择：  
PTHREAD\_EXPLICIT\_SCHED 和 PTHREAD\_INHERIT\_SCHED，前者表示新线程使用显式指定调度策略和调度参数（即 attr 中的值），而后者表示继承调用者线程的值。通过 pthread\_attr\_setinheritsched 函数设置，**缺省为 PTHREAD\_EXPLICIT\_SCHED。**

**(5) scope 属性：**\_\_scope。表示线程之间竞争 CPU 的范围，也就是说线程优先级的有效范围。POSIX 的标准中定义了两个值：PTHREAD\_SCOPE\_SYSTEM 和 PTHREAD\_SCOPE\_PROCESS，前者表示该线程与系统中所有线程一起竞争 CPU 时间，后者表示该线程仅与同一进程中的线程竞争 CPU。可通过函数 pthread\_attr\_setscope 和 pthread\_attr\_getscope 设置和获取线程的作用域。目前大多数的 pthread 中仅支持 PTHREAD\_SCOPE\_SYSTEM。

两个函数的原型声明如下：

```
#include <pthread.h>
int pthread_attr_getscope (pthread_attr_t* attr, int* scope);
```

函数传入值：

attr: 线程属性。

\*scope: 0, 表示 PTHREAD\_SCOPE\_SYSTEM，该线程与系统中所有线程一起竞争 CPU 时间

1, 表示 PTHREAD\_SCOPE\_PROCESS，该线程仅与同进程中的线程竞争 CPU

```
#include <pthread.h>
int pthread_attr_setscope (pthread_attr_t* attr, int scope);
```

函数传入值：

attr: 线程属性。

scope: PTHREAD\_SCOPE\_SYSTEM，值为 0

PTHREAD\_SCOPE\_PROCESS, 值为 1

考察父线程中的如下程序片段:

```
printf("PTHREAD_SCOPE_SYSTEM=%d\n",PTHREAD_SCOPE_SYSTEM); //0
printf("PTHREAD_SCOPE_PROCESS=%d\n",PTHREAD_SCOPE_PROCESS); //1

int scope;
pthread_attr_getscope(&attr, &scope); //默认为 PTHREAD_SCOPE_SYSTEM
printf("\nscope=%d\n",scope); //输出 0

//设置欲创建的线程仅与同进程的线程竞争 CPU
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
//尽管下述函数成功执行, 但目前 pthread 不支持 PTHREAD_SCOPE_PROCESS
perror("pthread_attr_setscope");

pthread_attr_getscope(&attr, &scope);
printf("\nscope=%d\n",scope); //仍然输出 0
```

### 2.2.3 管道通信机制

管道 pipe 是实现进程间通信的一种机制。在内存中临时建立的管道称为无名管道, 在磁盘上建立的管道实质上是一个特殊类型的文件, 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。如果一个进程通过管道的一端向管道写入数据, 另一进程可以在管道的另一端从管道中读取数据, 则实现了两个进程之间的通信。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, 读管道数据的进程保留读出端, 关闭管道写入端; 而写管道进程保留写入端, 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, 其中管道默认采用同步工作方式, 采用了生产者/消费者模型的同步工作机制。

#### 1、pipe 系统调用

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

pipe 建立一个无名管道, pipe\_id[0]中和 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。其中, 管道标识符 pipe\_id[0]用于 read 管道, pipe\_id[1]则用于 write 管道。

#### 2、管道读/写的系统调用

```
#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 是写入/读出管道的数据缓冲区首地址，count 说明数据缓冲区以 sizeof 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（字节数）。

注意管道的读写默认的通信方式为同步读写方式，采用生产者/消费者模型，即如果管道为空，则读进程阻塞，直到数据到达（写进程写入数据），反之如果管道已满，则写进程阻塞，直到数据被读进程读走。

## 2.3 示例实验

1) 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) 2022 by
 *      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]; //存放第两个无名管道标号
pthread_t thrd1,thrd2; //存放第两个线程标识

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，等待线程 thrd2 结束，并回收其资源
pthread_join(thrd2,NULL);
//挂起当前线程，等待线程 thrd1 结束，并回收其资源
pthread_join(thrd1,NULL);
//思考与测试：如果去掉上述两个pthread_join 的函数调用，会出现什么现象
exit(EXIT_SUCCESS);
}
//线程1 执行函数，它首先向管道1 写，然后从管道2 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{
        write(pipe1[1],&x,sizeof(int));
        read(pipe2[0],&x,sizeof(int));
        printf("thread%d read: %d\n",*num,x++);
    }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1 读，然后向管道2 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

```

注：使用 pthread 编程时，需要链接参数 **-lpthread**，或 **-pthread**，因为 pthread 并非 Linux 系统的默认库，而是 POSIX 线程库。在 Linux 中将其作为一个库来使用，因此加上 **-lpthread**（或 **-pthread**）以显式链接该库。函数在执行错误时的错误信息将作为返回值返回，并不修改系统全局变量 `errno`，当然也无法使用 `perror()` 打印错误信息。

## (3) 使用 make 命令编译连接生成可执行文件 tpipe:

```

$ gmake
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) 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>
#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,
            //所以关掉管道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:

```

$ gmake
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：

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

请基于无名管道，利用 pthread 线程库编程建立 3 个并发协作线程，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，说明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它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 3 进程综合实验（Shell）

## 3.1 实验目的

掌握操作系统 shell 的工作机制与实现过程，练习 Linux 系统中进程创建与控制有关的编程和调试技术。

## 3.2 实验说明

Shell 有时称为命令解释器，用于解释、执行用户命令，是一个用户使用操作系统的交互界面。在 shell 程序提供的界面中，用户输入要执行的命令，shell 负责执行这个命令。

Linux shell 是用户与 Linux 系统之间的一个接口程序，用户通过 shell 可以执行操作系统提供的命令、调用 Linux 工具，还可以自己编写自己的 shell 脚本程序。例如 Ubuntu 中的终端或命令窗口，是 shell 程序的运行界面。

Unix 图形化用户界面，注入了 GNOME、KDE 和 Xfce 等图像化元素，有时也被称为“可视 shell”或“图形 shell”。更多的情况下，shell 术语通常与命令行相关联。

在 Linux 中，有多中不同形式的 shell 可供用户选择使用，例如：

#### (1) /bin/sh Bourne shell

它是 Unix 的默认 Shell，也是其它 Shell 的开发基础。Bourne Shell 在编程方面相当优秀，但在处理与用户的交互方面不如其它几种 Shell。

#### (2) /bin/csh C shell

它提供了 Bourne Shell 所不能处理的用户交互特征，如命令补全、命令别名、历史命令替换等。但是，C Shell 与 Bourne Shell 并不兼容。

#### (3) /bin/ksh Korn shell

它集合了 C Shell 和 Bourne Shell 的优点，并且与 Bourne Shell 向下完全兼容。Korn Shell 的效率很高，其命令交互界面和编程交互界面都很好。

#### (4) /bin/bash Bourne Again Shell

它是 Linux 系统中一个默认的 Shell。Bash 不但与 Bourne Shell 兼容，还继承了 C Shell、Korn Shell 等优点。

### 3.2.1 将程序在后台运行

Shell 的工作过程一般是读取用户命令行的输入，如用户输入 ./a.out，Shell 创建一个子进程来执行命令 a.out，Shell 调用 wait() 等待该子进程执行结束并回收该子进程。当命令行的最后一个字符是“&”，如用户输入 ./a.out& (./a.out &)，即要求执行 a.out 的子进程在后台运行，则父进程不需要等待子进程执行，而是继续执行，此时后台运行的子进程不接收键盘的输入。

#### 1. 将程序在后台运行

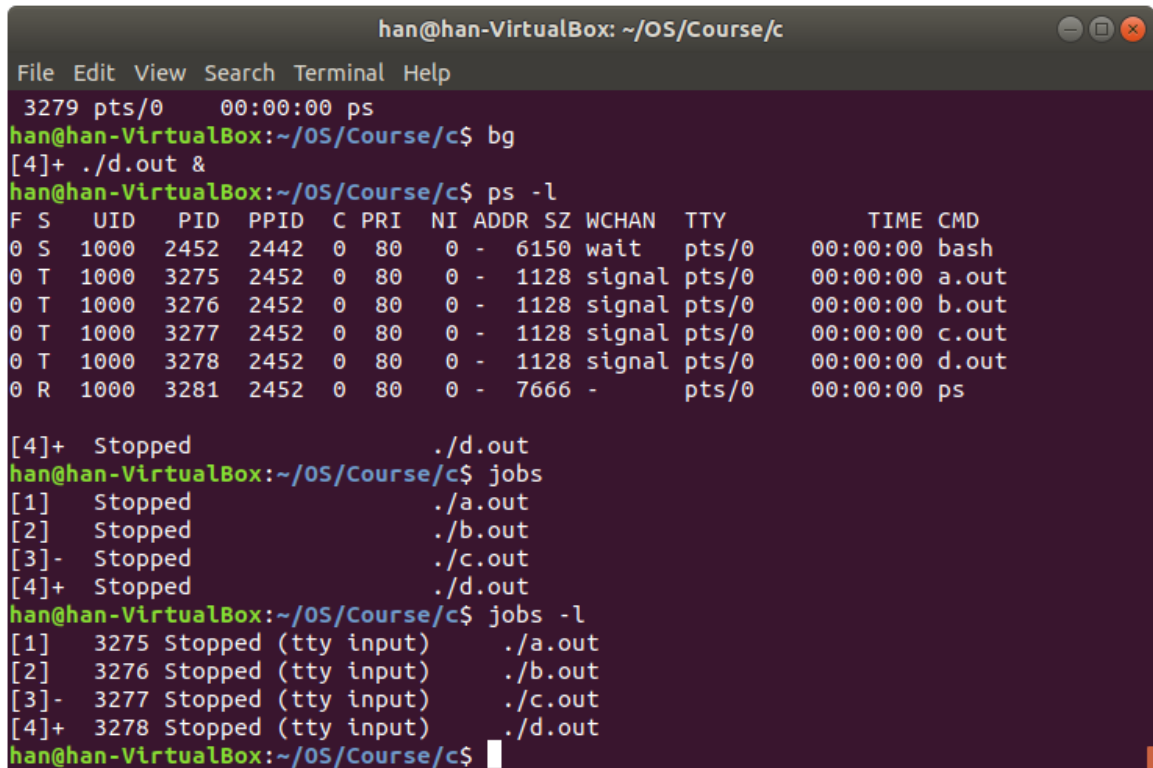
在命令行尾部添加符号&运行一个程序，如 ./a.out&，则系统将程序 a.out 在后台运行。在 a.out 在后台执行过程中，不会接收键盘、鼠标等输入。此时 Shell 可继续接收键盘命令，可以执行其它程序。

Ctrl+Z 可以将一个正在前台运行的程序切换到后台，并暂停该程序的执行。

#### 2. 查看后台运行的程序

可以利用作业控制命令 `jobs` 查看后台运行的程序。例如运行命令 `./a.out&`、`./b.out&`、`./c.out&`与`./d.out&`后，命令 `ps -l`、`jobs` 与 `jobs -l` 的输出结果如图所示。

在 `jobs` 命令的输出结果中，每个后台程序最左边对应的[1]、[2]、[3]与[4]是后台运行的作业号，其中作业[4]+中的“+”号表示该作业是当前要操作的作业，[3]-中的“-”号表示该作业是下一个要操作的作业。例如，如图所示中，由于作业[4]后有“+”号，执行 `fg`，会将作业[4]切换到前台运行，



```
han@han-VirtualBox: ~/OS/Course/c
File Edit View Search Terminal Help
3279 pts/0    00:00:00 ps
han@han-VirtualBox:~/OS/Course/c$ bg
[4]+ ./d.out &
han@han-VirtualBox:~/OS/Course/c$ ps -l
 F S      UID      PID   PPID    C  PRI   NI     ADDR  SZ  WCHAN    TTY          TIME CMD
 0 S      1000     2452    2442    0   80    0      -   6150  wait   pts/0        00:00:00 bash
 0 T      1000     3275    2452    0   80    0      -   1128  signal pts/0        00:00:00 a.out
 0 T      1000     3276    2452    0   80    0      -   1128  signal pts/0        00:00:00 b.out
 0 T      1000     3277    2452    0   80    0      -   1128  signal pts/0        00:00:00 c.out
 0 T      1000     3278    2452    0   80    0      -   1128  signal pts/0        00:00:00 d.out
 0 R      1000     3281    2452    0   80    0      -   7666  -      pts/0        00:00:00 ps

[4]+  Stopped                  ./d.out
han@han-VirtualBox:~/OS/Course/c$ jobs
[1]  Stopped                  ./a.out
[2]  Stopped                  ./b.out
[3]- Stopped                  ./c.out
[4]+ Stopped                  ./d.out
han@han-VirtualBox:~/OS/Course/c$ jobs -l
[1]  3275 Stopped (tty input)  ./a.out
[2]  3276 Stopped (tty input)  ./b.out
[3]- 3277 Stopped (tty input)  ./c.out
[4]+ 3278 Stopped (tty input)  ./d.out
han@han-VirtualBox:~/OS/Course/c$
```

### 3. 将后台程序切换号前台运行

命令 `fg`，将当前作业（作业号后有+号，如[4]+）切换到前台运行。

命令 `fg n` 或 `fg %n`，将第 `n` 个作业切换到前台运行。

命令 `fg -`或`fg %-`，将第下一个作业切换到前台运行（作业号后有-号，如[3]-）

### 4. 继续运行后台暂停的程序

命令 `bg`，继续在后台运行暂停的当前作业（作业号后有+号，如[4]+）。该程序仍然在后台执行，不能接收键盘、鼠标等输入。

命令 `bg n` 或 `bg %n`，继续在后台运行暂停的第 `n` 个作业。

命令 `bg -`或`fg %-`，继续在后台运行暂停的下一个作业（作业号后有-号，如[3]-）

### 5. 关于命令 `./a.out&` 的实现

如果执行命令 `./a.out`，则 Shell 创建子进程，在子进程中调用 `exec` 系统调用执行 `a.out`，并调用 `wait()` 等待子进程结束。等子进程执行结束后，继续接收键盘输入，运行新的命令。

如果执行命令 `./a.out&`，则 Shell 创建子进程，在子进程中调用 `exec` 系统调用执行 `a.out`，不调用 `wait()` 等待子进程结束，而是继续接收键盘的输入，运行新的命令。

### 3.2.2 文件描述符与 I/O 重定向

在进程的 PCB 中，有一部分表项保存该进程打开的文件描述符（有的资料称为文件句柄），称为用户文件描述符表（UNIX 中称为 U 区，U area）。每个表项（UNIX 中称为 slot）对应一个索引值，该索引值从 0 开始编号，依次为 0、1、2、...

操作系统系统创建进程时，会为每个进程自动打开三个标准设备：`stdin`、`stdout` 和 `stderr`，并为其分配三个文件标识符 0、1、2。之后当用户通过系统调用 `fd=open(...)` 和 `fd=creat(...)` 打开或创建文件时，则会从 3 开始分配文件描述符并返回给 `fd`。

需要注意的是，系统在为打开或创建的文件分配文件描述符时，会在文件描述符表项中从索引值 0 开始顺序查找，将搜索到的第一个未被分配的文件描述符表项分配给该文件，然后将该表项的索引值返回给 `fd`，即系统选择索引值最小的空文件描述符表项予以分配。

这意味着，如果你利用系统调用 `close(1)` 关闭了标准输出设备 `stdout`（显示器），然后再使用 `open()` 打开一个文件，如 `OutFile`，这时系统会将该进程的标准输出重定向到文件 `OutFile`，即你输出到 `stdout` 的信息会重定向到文件 `OutFile` 中。例如，如果你用 C 语言编程，函数 `printf()` 的输出会写入到 `OutFile` 中，而不再输出到屏幕上。

同样，如果你利用系统调用 `close(0)` 关闭了标准输入设备 `stdin`（键盘），然后再使用 `open` 打开一个文件，如 `InputFile`，这时系统会将该进程的标准输入重定向到文件 `InputFile`，即当你在程序中利用输入语句期望从键盘输入信息时，程序会读取文件 `InputFile` 中的内容作为输入，而不再接收键盘键入的信息。例如，如果你用 C 语言编程，函数 `scanf()` 不再等待你从键盘输入信息，而直接从文件 `InputFile` 读取内容作为键盘的输入。

### 3.2.3 系统调用 dup()

利用系统调用 `dup()` 可很方便地实现 I/O 重定向。

`dup()` 系统调用语法：

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

系统调用 `dup()` 将一个文件描述符复制到该用户文件描述符表的第一个空的表项中，当复制成功时，返回索引值最小且尚未使用的新的文件描述符。若有错误则返回 -1，`errno` 会存放错误代码。`dup` 适用于所有的文件类型。

执行成功后，新的文件描述符与复制的文件描述符实质上是指向了同一个文件。

例如，当执行系统调用 `int newfd=dup(oldfd)` 时，系统会将索引值 `oldfd` 对应的文件描述符表项中的信息复制到一个索引值最小、未被分配的用户文件描述符表项中，并将该表项的索引值返回给 `newfd`。此时，`newfd` 与 `oldfd` 就指向了同一个文件表项，也就指向了同一个文件。通过 `newfd` 与 `oldfd` 访问文件时，就可以共享该文件所有的锁定、读写位置和各项权限。

利用系统调用 `dup()` 很容易实现输入/输出重定向。例如考察下述程序片段：

```
int fd=open("outFile.txt");
close(1); // close(stdout); 关闭标准输出设备 stdout
dup(fd);
```

系统将文件 `outFile.txt` 对应的文件描述符复制到原标准输出设备（`stdout`）的文件描述符中，其后的标准输出将重定向到文件 `outFile.txt`。

注：该段代码与上段代码完成相同的功能，只是习惯上采用上段代码

```
close(1); //close(stdout); 关闭标准输出设备 stdout
int fd=open("outFile.txt");
```

基于上述讨论，容易将标准输入/输出通过 `dup` 重定向到管道中，然后用函数 `scanf()`/`printf()` 从管道中读写数据。

```
void redirect(int k, int pipe_id[2]) {
    // k 表示文件描述符，0 代表 stdin，1 代表 stdout，2 代表 stderr
    if(!pipe_id) return -1;
    close(k);
    dup(pipe_id[k]);
    close(pipe_id[0]);
    close(pipe_id[1]);
}
```

### 3.2.4 系统调用 `dup2()`

`dup2()` 系统调用语法：

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

将 oldfd 复制到 newfd 的文件描述符表项中，并返回 newfd。如果 newfd 已经被打开，dup2()会先将其关闭，然后复制 oldfd 到 newfd 中，并返回 newfd。如果 oldfd 等于 newfd，不会关闭 oldfd，返回 newfd，

同样，系统调用 dup2()返回的描述符 newfd 与 oldfd 描述符所指向的文件共享同一文件表项。

### 3.2.5 重定向与管道命令

一般的操作系统的 Shell 均支持如下的几个 I/O 重定向命令：>、>>、<、<<及管道命令“|”。

输入重定向：

- (1) 命令 < 文件：将文件作为命令的标准输入
- (2) 命令 << 分界符：从标准输入中读入，直到遇到分界符停止

输出重定向：

(1) 命令 > 文件，或命令 1>文件：将标准输出重定向到文件中（清除原有文件中的数据）。这里的 1 表示文件描述符 1，即标准输出。

(2) 命令 2> 文件：将错误输出重定向到文件中（清除原有文件中的数据）。这里的 2 表示文件描述符 2，即标准错误输出

注：命令 &>文件，将标准输出与错误输出都重定向到文件中（清除原有文件中的数据）

(3) 命令 >> 文件，或命令 1>>文件：将标准输出重定向到文件中（在原有的内容后追加）

(4) 命令 2>> 文件：将错误输出重定向到文件中（在原有内容后面追加）

注：命令 &>> 文件：标准输出和错误输出都写入文件（在原有内容后追加）

这些重定向命令可组合使用，如：

命令 < 文件 1 >文件 2：将文件 1 作为命令的标准输入并将标准输出到文件 2

例如，在 shell 界面中输入命令 echo hello，你会看到在标准输出中打印了 hello 这个单词；如果你输入命令 echo hello > file.txt，你会发现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。

在命令 echo hello > file.txt 中，echo 是一个可执行文件，你可以在路径 /bin 下找到它。当 Shell 执行这个命令的时候会创建一个子进程来运行这个可执行文件，hello 是 echo 的一个参数。

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 中可以使用管道命令符 "|" 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。

管道的实现机制是系统建立了一个临时管道，并将左边命令的输出重定向到管道的写端，将右边命令的输入重定向到管道的读端。

注：关于标准输出与错误输出

考察下述程序：

```
#include <stdio.h>
int main()
{
    int n=10;
    for (int i=0;;i++)
        printf("%d ",i);
    printf(stderr,"%s\n","error"); /*或 fprintf(stderr,"error\n"); */
}
```

其中，`printf()`默认输出到 1 号设备即标准输出 `stdout`。由于 `fprintf` 中指定输出到 `stderr`，即 2 号设备，因此其结果将输出到标准错误输出 `stderr`。

假设将上述程序编译成可执行程序 `a.out`，则执行 `./a.out`，系统会将屏幕作为 `stdout` 与 `stderr`，`printf()`与 `fprintf()`都将结果输出到屏幕上。

若执行 `./a.out 1>t.txt`，则 `printf()`将结果输出到文件 `t.txt` 中，`fprintf()`将结果输出到屏幕上。

若执行 `./a.out 2>t.txt`，则 `printf()`将结果输出到屏幕上，`fprintf()`将结果输出到文件 `t.txt` 中。

若执行 `./a.out &>t.txt`，则 `printf()`与 `fprintf()`都将结果输出到文件 `t.txt` 中。

### 3.3 简单的 shell 框架

实现 Shell 接口的一种方法是，父进程给出用户命令提示符，等待用户输入命令行。Shell 读用户命令行的输入，然后创建一个子进程来执行这个命令。如果命令行的最后一个字符不是“&”，即不要求执行该命令的子进程在后台运行，则父



进程应该等待子进程执行结束。反之，如果命令行的最后一个字符是“&”，即要求执行该命令的子进程在后台运行，则父进程不需要等待子进程执行，而是继续执行，父子进程即可并发执行。

一个简单 shell 的 C 程序结构大致如下：

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 100 /* 限定用户输入的命令行长度不超过 100 字符*/

int main(void)
{
    int background; /* 如果命令行的最后字符是&，则 background=1 */
    while (1) {
        /* 父进程自定义信号 SIGINT 的处理程序，当按下 ctrl-c 后，
        * 不终止父进程（Shell），只终止其子进程。
        * 设置其它信号，当父进程（shell）收到该信号后，结束执行，退出
        */
        background=0;
        printf("COMMAND->"); /* 命令提示符*/
        /* 读用户输入的命令行，并分析命令行 */
        if (命令行含有&) /* 后台运行用户输入的命令*/
            background = 1;
        if (fork()= =0) /* 处理执行命令行输入的命令 */
            /* 这里可以考虑使用 vfork() 替代 fork()，效率更高*/
            {
                /* 设置信号 SIGINT 的处理方式为默认 SIG_DFL，
                * 按下 ctrl-c 后，终止子进程 */
                /* 命令行中是否有 I/O 重定向，即是否有符号 >, < */
                if (输出重定向)
                { /* 重定向标准输出到文件 outfile */
                    fd=creat(outfile,fmask);
                    close(1); /* close(stdout) */
                    dup(fd);
                    close(fd);
                    /*或
                    fd=creat(outfile,fmask);
                    dup2(fd,1);
                    close(fd); */
                } /* if (输出重定向) */
                if (输入重定向)
                { /* 重定向文件 inputfile 到标准输入 */
```

```

    fd=open(inputfile,fmask);
    close(0); /* close(stdin); */
    dup(fd);
    close(fd);
    /*或
    fd=open(inputfile,fmask);
    dup2(fd,0);
    close(fd); */

} /* if (管道) */
/* 是否有形如 command1|command2 的管道 */
if (需要建立管道 "|")
{
    pipe(fildes); /* 创建无名管道 fildes */
                    /* fildes[0] 用于读管道, fildes[1] 用于写管道 */
    if (fork()==0) /* 处理管道左部分 command1 */
    {
        /* 标准输出到管道 */
        close(1); // close(stdout);
        dup(fildes[1]); //标准输出定向到管道的写端
        close(fildes[1]); //
        close(fildes[0]);
        /* 执行管道左部分命令 */
        execve(command1, ...);
    } /* 处理管道左部分 */
    /* 管道右部分 command2 */
    /* 标准输入来自管道 */
    close(0); // close(stdin);
    dup(fildes[0]);
    close(fildes[0]);
    close(fildes[1]);
} /* if (需要建立管道 "|") */
/* 如果输入的命令行有形如 command1|command2 的管道,
* 这里就执行管道右部分命令。
* 如果输入的命令行中没有管道符,
* 则 command2 是用户在命令行中输入的命令
*/
execve(command2,...);
} /* if (fork() == 0) 执行命令行输入的命令 */
/*对于形如 command1|command2|command3 多级管道, 需要同时重定
* 向 command2 的标准设备 0 与 1 */

/* 父进程 (即 shell) 从此处执行 */

```

```

/*
 * 如果不要求后台运行输入的命令，父进程等待子进程结束，
 * 否则，父进程不需要等待，继续执行，读入并处理用户输入
 */
if (background==0)
    retid=wait(&status);
} /* while (1) */
} /* main */

```

### 3.4 独立实验

设计与实现一个简单的 shell，至少具备以下功能：

- (1) 执行用户输入的合法命令，允许命令携带参数，如 `ls -la`；
- (2) 在命令执行期间，允许用户按下一个给定的组合键结束命令的执行；
- (3) 支持 I/O 重定向 (`>`、`<`) 与管道 (`|`)；
- (4) 支持命令的后台运行；
- (5) 保存用户最近输入的 30 个命令，可利用上下方向键进行选择；
- (6) 若用户输入非法命令，或找不到命令要执行的文件等，给出错误提示；

运行示例：

```

# 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
###

```

### 3.5 实验要求

对比一个真实的 Linux Shell，你所实现的 Shell 需要做哪些改进？实现这些改进的思路如何？根据实验程序、调试过程和结果分析写出实验报告。

## 实验 4 进程同步

### 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、信号量数组、消息队列，来解决协作并发进程间的同步与互斥的问题。

### 4.2.1 共享内存

共享内存是操作系统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，然后可以对该段内存进行读写等操作。如果其中一个进程更新了其中的数据，其它进程立即会看到这一更新，从而实现了进程之间的通信。

多个进程共享内存时，需要经历下述 5 个步骤：

（1）首先需要其中的一个进程调用 `shmget()` 创建一段共享内存区，其它要共享该内存区的进程调用 `shmget()` 获取该段内存的描述符。调用时给出对共享内存区的访问权限（e.g read、write 权限）。

（2）进程调用 `shmat()` 将该段共享内存附接到进程的地址空间上。

由于（1）中创建的共享内存区具有相对独立的地址空间，不在这些进程各自的地址空间范围内，由于存储保护的原因，这些进程无法访问该共享内存段，因此需要这些进程将该共享内存段附加到它们各自的地址空间中，然后才能访问，否则会产生地址越界错误。

（3）进程访问该段共享内存

要共享该段内存的进程将其附接到自己的地址空间后，每个进程可以根据对该共享内存段所具有的权限访问（读/写）该段内存，实现了进程之间的共享。

（4）进程调用 `shmdt()` 将共享内存区从进程的地址空间上分离

当共享结束后，需要其中的一个进程删除该共享内存段。在删除之前，需要共享该内存段的每个进程先将该共享内存段分别从它们的地址空间中剥离（分离）出去，然后才能删除。

（5）进程调用 `shmctl()` 删除共享内存区

## 1、与信号量相关的系统调用

下面介绍的这些系统调用，如果调用不成功，可以通过 `perror`(“系统调用名”) 获取错误原因。

### (1) 创建共享内存区系统调用

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

功能：创建一个新的键值为 `key` 的共享存储区，或获取一个已经存在的键值为 `key` 的共享存储区标识。

传入参数：

- `key`：共享内存的键值，可以为 `IPC_PRIVATE`，也可以指定一个整数；
- `size`：共享内存字节长度；
- `flags`：共享内存权限位，可以用 `IPC_CREAT`、`IPC_EXCL`、`S_IRUSR`、`S_IWUSR` 等宏与访问权限的组合；

返回值：`shmget` 调用成功后，如果 `key` 用新整数指定（`key` 对应的共享存储区不存在），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 `shm`。如果指定的 `key` 已存在则返回与 `key` 关联的标识符 `shm`。

如果调用不成功，返回-1，错误原因存于 `error` 中

错误代码：

- `EINVAL`：参数 `size` 小于 `SHMMIN` 或大于 `SHMMAX`
- `EEXIST`：预建立 `key` 所指的共享内存，但已经存在
- `EIDRM`：参数 `key` 所指的共享内存已经删除
- `ENOSPC`：超过了系统允许建立的共享内存的最大值(`SHMALL`)
- `ENOENT`：参数 `key` 所指的共享内存不存在，而参数 `shmflg` 未设 `IPC_CREAT` 位
- `EACCES`：没有权限
- `ENOMEM`：核心内存不足

对于上述系统调用，及后续介绍的几个系统调用，当调用不成功返回-1时，可以通过返回值查阅错误代码，也可以简单地在系统调用之后通过 `perror`(“系统调用名”)获取具体的错误原因。如 `perror`(“shmget”)。

**注 1：**关于第一个参数 `key`。键值 `key` 是该段共享内存存在系统中的唯一标识，类似于一个文件的文件名。一个进程基于该 `key` 通过系统调用 `shmget`()创建一段共享，如果创建成功，`shmget`()返回该段内存的一个共享内存段标识符，类似于打开一个文件是系统返回的文件描述符。

其它要共享该段内存的进程需要基于该 key 通过系统调用 shmget()获取该共享内存段的共享内存段标识符，然后基于该标识符访问该共享段。只获取共享内存段时，参数 size 可以指定为 0。

我们可以自己指定一个整数作为 key，也可以通过函数 ftok()获取一个 key。

函数 ftok()在头文件<sys/ipc.h>中定义，原型为：

```
#include <sys/ipc.h>
key_t ftok( const char * fname, int id )
```

其中，fname 是一个已存在的文件名。在 UNIX 的实现中，一般是将文件 fname 的索引节点号的前面加上第二个参数 id 的低 8 位，作为 ftok()的返回值。

例如 key\_t key = ftok(".", random()); //一般文件名使用当前目录

**注 2：**关于第三个参数 msgflg。参数 msgflg 可以是 IPC\_CREAT、IPC\_EXCL 与访问权限控制符的组合，其中的访问权限控制符的意义和文件系统中的权限控制符是类似的。关于一些常用的访问权限控制符可参见表 4-1（定义在头文件 <sys/stat.h>中）。IPC\_CREAT 与 IPC\_EXCL 在头文件<sys/ipc.h>中定义。

- **IPC\_CREAT**：调用 msgget()时，如果共享存储区对象不存在，则创建之，并返回一个新建的共享存储区对象的标识符；如果共享存储区对象已经存在，则返回一个已存在的共享存储区对象的标识符。

- **IPC\_EXCL**：一般与 IPC\_CREAT 一起使用。例如，IPC\_CREAT|IPC\_EXCL，表示调用 msgget()时，如果共享存储区对象不存在则创建之，如果已经存在，则产生一个错误并返回-1。

IPC\_EXCL 标志本身并没有太大的意义，但和 IPC\_CREAT 标志一起使用可以用来保证所得的共享存储区对象是新创建的而不是打开的已有的对象。

msgflg 一般是 IPC\_CREAT、IPC\_EXCL 与访问权限控制符的组合。例如要创建一个或打开一个所有进程都能读写的共享存储区，msgflg 可以取：IPC\_CREAT|S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP|S\_IROTH|S\_IWOTH，或 IPC\_CREAT|0666。

访问权限 0666 的含义：八进制数 0666 对应二进制是 110 110 110，最高三位 110 对应的创建共享存储区用户的访问权限，中间三位是同组用户的访问权限，最低三位对应其它用户的访问权限。

每一组的三位从左往右分别对应读(r)、写(w)、执行(x)的权限。

以下是几个系统调用 shmget()的例子：

(a) int shmid = shmget(0x8888, 1024, IPC\_CREAT|0666);

创建了一个键值为 0x8888，大小为 1024 字节，访问权限为 0666 的共享内存区，返回共享存储区标识符 shmid。

(b) `int shmid = shmget(key, IPC_CREAT | IPC_EXCL | 0666);`

如果 `key` 对应的共享存储区已经存在，返回错误（-1），如果 `key` 对应的共享存储区不存在，创建之，返回共享内存段标识符 `shmid`。

(c) `int shmid = shmget(IPC_PRIVATE, S_IRUSR|S_IWUSR);`

创建仅在该进程内使用的共享存储区，访问权限为 0600。

关于几个共享内存区的访问权限的宏如表 4-1 所示，这些宏在头文件 `<sys/stat.h>` 中定义。可以看出，关于用户（进程）对共享内存的访问权限的概念，与文件的访问权限是一致的。这也体现 Linux 中“一切皆文件”的思想。该思想同样适用于信号量和消息队列。

表 4-1 几个访问权限的定义

st_mode 屏蔽	含义
S_IRUSR S_IWUSR S_IXUSR	用户读 用户写 用户执行
S_IRGRP S_IWGRP S_IXGRP	组读 组写 组执行
S_IROTH S_IWOTH S_IXOTH	其他读 其他写 其他执行

## (2) 将一段共享内存附加到调用进程中的系统调用

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

功能：将 `shmid` 标识的共享内存区对象映射到调用进程的地址空间上，随后进程可像访问本地空间一样访问该段内存。

传入参数：

`shmid`：由系统调用 `shmget()` 创建的共享内存的标识符

**shmaddr:** 一般总为 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

返回值:

调用成功后, 返回附加的共享内存的首地址 (附接到进程地址空间上的首地址)

如果调用不成功, 返回-1, 错误原因存于 **error** 中

错误代码:

**EACCES:** 无权限以指定方式连接共享内存  
**EINVAL:** 无效的参数 **shmid** 或 **shmaddr**  
**ENOMEM:** 核心内存不足

例如:

```
shmaddr = (char *)shmat( shmid, NULL, 0 )
```

### (3) 对共享内存段读写

利用系统调用 **shmget()**得到的共享内存段标识符。对共享内存段进行读写。需要注意的是, 该共享内存区需要进程之间互斥访问, 如果读写进程之间符合生产者-消费者模型, 需要在进程中利用信号量编写相应的代码, 实现进程之间的互斥与同步。

如:

```
strcpy( shmaddr, "Hi, I am child process!\n" );  
scanf("%[^\\n]", shmaddr); //字符串的结束符为回车  
sprintf(shmaddr,"Hi, there!");
```

### (4) 将一段共享内存从到调用进程中分离出去的系统调用

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

功能: 与 **shma()**相反, 将共享内存段从进程的地址空间中分离出去, 此后本进程将被禁止访问此段共享内存。

传入参数:

**shmadr:** 系统调用 **shmat()**返回的指向附加共享内存的指针 (地址)

返回值:

**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，错误原因存于 `error` 中

错误代码：

`EINVAL`：无效的参数 `shmaddr`

例如：`int ret=shmdt( shmaddr )`

## （5）删除共享内存段

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shm_id *buf)
```

功能：该系统调用完成对共享内存的控制，可以用来删除内存共享段。

传入参数：

`shmid`：由系统调用 `shmget()` 创建的共享内存的标识符  
`cmd`：

`IPC_STAT`：得到共享内存的状态，把共享内存的 `shm_id` 结构体复制到 `buf` 中

`IPC_SET`：改变共享内存的状态，把 `buf` 所指的 `shm_id` 结构中的 `uid`、`gid`、`mode` 复制到共享内存的 `shm_id` 结构体内

`IPC_RMID`：删除这段共享内存

`buf`：共享内存管理结构体

返回值：调用成功，返回 0。调用不成功，返回-1，错误原因存于 `error` 中。

错误代码：

`EACCESS`：参数 `cmd` 为 `IPC_STAT`，确无权限读取该共享内存

`EFAULT`：参数 `buf` 指向无效的内存地址

`EIDRM`：标识符为 `msqid` 的共享内存已被删除

`EINVAL`：无效的参数 `cmd` 或 `shmid`

`EPERM`：参数 `cmd` 为 `IPC_SET` 或 `IPC_RMID`，却无足够的权限执行

## 2、利用命令 `ipcs -m` 观察共享内存情况

例如：

```
$ 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
```

其中,

key: 共享内存关键值  
shmid: 共享内存标识  
owner: 共享内存所有者(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(共享内存的进程数)  
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。

### 4.2.2 信号量

信号量数组是操作系统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。

多个进程利用信号量实现互斥与同步时,需要经历下述几个步骤:

(1) 首先需要其中的一个进程调用 semget ()创建一个信号量集(信号量数组),其它进程调用 semget ()获取该信号量集的描述符。

(2) 进程调用 semctl()初始化信号量(如赋初值)

一般情况下,由创建信号量集的进程对其进行初始化。理论上,只要具备相应的访问权限,使用该信号量集的进程均可对其完成初始化操作。

(3) 进程调用 semop ()操作信号量,实现相应的 P、V 操作。

(4) 进程调用 semctl ()删除信号量集

#### 1、与信号量相关的系统调用

下面介绍的这些系统调用,如果调用不成功,可以通过 perror(“系统调用名”)获取错误原因。

##### (1) 创建一个信号量数组的系统调用

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

功能: 创建一个新的键值为 key 的信号量数组,或获取一个已经存在的键值为 key 的信号量数组(信号量集)标识。

传入参数:

- **key**: 信号量数组的键值, 可以为 **IPC\_PRIVATE**, 也可以指定一个整数
- **nsems**: 信号量数组中信号量的个数
- **flags**: 信号量数组权限位, 含义同共享内存权限位。

返回值:

调用成功: 如果 **key** 用新整数指定 (**key** 对应的信号量集不存在), 且 **flags** 中设置了 **IPC\_CREAT** 位, 则返回一个新建立的信号量数组标识符 **semid**。如果指定的整数 **key** 已存在, 则返回与 **key** 关联的标识符 **semid**。

调用不成功, 返回-1, 错误原因存于 **error** 中

错误代码:

- **EACCES**: 没有访问该信号量集的权限
- **EEXIST**: 信号量集已经存在, 无法创建
- **EINVAL**: 参数 **nsems** 的值小于 0 或者大于该信号量集的限制; 或者是该 **key** 关联的信号量集已存在, 并且 **nsems** 大于该信号量集的信号量数
- **ENOENT**: 信号量集不存在, 同时没有使用 **IPC\_CREAT**
- **ENOMEM**: 没有足够的内存创建新的信号量集
- **ENOSPC**: 超出系统限制

注: 其中的键值 **key** 是该信号量集数组 (信号量集) 在系统中的唯一标识。一个进程基于该 **key** 通过系统调用 **semget()** 创建一个信号量集, 如果创建成功, **semget()** 返回该信号量集的标识符, 其它要访问该信号量集的进程需要基于该键值 **key** 通过系统调用 **semget()** 获取该信号量集的标识符, 然后基于该标识符访问该信号量集。

## (2) 操作信号量数组的系统调用

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

功能: 操作一个或一组信号量。

传入参数:

- semid**: 由 **semget()** 创建或获取的信号量数组的标识符
- sops**: 指向存储信号量操作的结构体数组指针
- nsops**: 信号量数组元素的个数。

返回值: 调用成功返回 0, 不成功返回-1, 错误原因存于 **error** 中。

错误代码:

- E2BIG**: 一次对信号量的操作数超出系统的限制

EACCES: 调用进程没有权能执行请求的操作, 并且不具有 CAP\_IPC\_OWNER 权能  
EAGAIN: 信号量操作暂时不能满足, 需要重试  
EFAULT: sops 或 timeout 指针指向的空间不可访问  
EFBIG: sem\_num 指定的值无效  
EIDRM: 信号量集已被移除  
EINTR: 系统调用阻塞时, 被信号量中断  
EINVAL: 参数无效  
ENOMEM: 内存不足  
ERANGE: 信号量所允许的值越界

系统调用 semop() 的第二个参数 sops, 指向一个类型为 sembuf 的结构体数组, 每个 sembuf 结构体对应一个特定信号量的操作。要对信号量进行操作, 必须熟悉该数据结构, 在头文件 linux/sem.h 中声明的结构体如下:

```
struct sembuf{
    unsigned short  sem_num; /* semaphore index in array */
    short           sem_op;   /* semaphore operation */
    short           sem_flg;  /* operation flags */
};
```

其中,

**sem\_num**: 信号量在信号量集中的编号 (索引), 第一个信号的编号是 0, 第二个信号的编号是 1, 以此类推。当使用单个信号量时, 取值为 0;

**sem\_op**: 信号量增量, 根据其值为正、为负、0, 分为下述三种情况:

如果  $\text{sem\_op} > 0$ , 表示进程将释放  $\text{sem\_op}$  个该信号量对应的资源, 信号量的值就加上  $\text{sem\_op}$ , 表示 V 操作。

如果  $\text{sem\_op} < 0$ , 表示进程申请资源的使用权, 信号量的值就减去  $\text{sem\_op}$  的绝对值。相当于 P 操作。如果操作之后, 信号量的值小于 0, 进程是否进入阻塞状态, 取决于第三个成员变量  $\text{sem\_flg}$  的取值。若操作之后信号量的值小于 0, 且  $\text{sem\_flg}$  没有被设置成 IPC\_NOWAIT (例如被设置成 SEM\_UNDO), 则调用进程阻塞, 直到信号量的值大于等于 0 被唤醒; 若  $\text{sem\_flg}$  被设置成 IPC\_NOWAIT, 进程不会睡眠, 而直接返回错误代码 EAGAIN

如果  $\text{sem\_op} = 0$ , 如果  $\text{sem\_flg}$  没有被设置 IPC\_NOWAIT (例如被设置成 SEM\_UNDO), 则调用进程进入睡眠状态, 直到信号量的值变为 0; 否则, 如果  $\text{sem\_flg}$  被设置 IPC\_NOWAIT, 则进程不会睡眠, 直接返回 EAGAIN 错误。

**sem\_flg**: 信号操作标志, 可能的选择一般有两种:

IPC\_NOWAIT: 即使信号量值为负时 (目前无可用资源), 进程调用 semop() 时不会阻塞, 立即返回, 同时设定错误信息。

**SEM\_UNDO**：选项会让内核记录一个与调用进程相关的 UNDO 记录，当该进程正常结束或异常退出时，内核会根据该进程的 UNDO 记录自动重设为 semop()调用前的值，以防止进程在异常情况下结束时，未将锁定的资源解锁，导致资源不可用。

例如定义信号量标识 **semid** 所指定的单个信号量的 P、V 操作如下：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semid=semget(...); /*创建一个信号量，或获取一个已创建信号量的描述符*/
union semun sem_arg; /*该结构体见：（3）控制信号量数组的系统调用 */
sem_arg.val=1; /* 信号量的初值*/
semctl(sem_id,0,SETVAL,sem_arg); /* 初始化信号量的值为 1 */
struct sembuf buf;
int P(int semid) /* P()操作，或 wait()操作
{
    buf.sem_num = 0; /* 对索引值为 0 的信号量施加 P 操作*/
    buf.sem_op = -1; /* 调用 semop()时，信号量将减 1 */
    buf.sem_flg = SEM_UNDO; /*若操作后信号量 0 的值不为正，进程等待*/
                                /*且进程退出时自动重设信号量为 semop()*/
                                /*调用前的值 */
    if((semop(semid,&buf,1)) < 0) /* */
    {
        perror("P error ");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}

int V(int semid) //V()操作，或 signal()操作
{
    struct sembuf buf;
    buf.sem_num = 0; /* 对索引值为 0 的信号量施加 V 操作*/
    buf.sem_op = 1; /* 调用 semop()时，信号量将+1 */
    buf.sem_flg = SEM_UNDO;
    if((semop(semid,&buf,1)) < 0)
    {
        perror("V error ");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

还有一个与此类似的系统调用 **semtimedop()**，其原型为：

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

```
int semtimedop(int semid, struct sembuf *sops, unsigned nsops, struct timespec
*timeout);
```

其使用方法请自行查阅相关资料。实验中主要使用 `semop()`。

### (3) 控制信号量数组的系统调用

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

功能：在 `semid` 标识的信号量数组，或者该数组的第 `semnum` 个信号量上执行 `cmd` 所指定的控制命令。

可以利用该调用初始化信号量的值、获取当前信号量的值，以及删除已存在的信号量。

传入参数：

- `semid`：由 `semget()` 创建或获取的信号量数组的标识符
- `semnum`：该信号量数组中的第几个信号量（从 0 开始）
- `cmd`：对信号量发出的控制命令  
如 `GETVAL` 返回当前信号量的值，`SETVAL` 设置信号量的值，`IPC_RMID` 删除标识为 `semid` 的信号量  
关于 `cmd` 的命令的使用详见注 1。
- `arg`：保存信号量状态的联合体，包括信号量的值等。详见注 2。

返回值：执行成功，根据 `cmd` 的命令返回相应的值；执行不成功，返回 -1。

例如，若执行成功，`cmd` 取 `IPC_STAT`、`SETVAL`、`IPC_RMID`，返回 0；若 `cmd` 取 `GETVAL`，则返回信号量的当前值。

**注 1：**第三个参数 `cmd` 可以使用如下命令：

- `IPC_STAT`：读取一个数组的数据结构 `semid_ds`，并将其存储在 `semun` 中的 `buf` 参数中，调用进程应有对信号量数组的读权限。
- `IPC_SET`：设置信号量集的数据结构 `semid_ds` 中的元素 `ipc_perm`，其值取自 `semun` 中的 `buf` 参数。
- `IPC_RMID`：将信号量集从内存中删除，并唤醒因调用 `semop()` 而阻塞的进程。
- `GETALL`：用于读取信号量集中的所有信号量的值。
- `GETNCNT`：返回正在等待资源的进程数目。
- `GETPID`：返回最后一个执行 `semop` 操作的进程的 PID，又称为 `sempid`。

- **GETVAL**: 返回信号量集中的一个单独的信号量的值。
- **GETZCNT**: 返回正在等待完全空闲的资源的进程数目。
- **SETALL**: 设置信号量集中的所有的信号量的值。
- **SETVAL**: 设置信号量集中的一个单独的信号量的值。
- **SEM\_STAT**: 返回信号集标识 swmid
- **SEM\_INFO**: 同 **IPC\_INFO**
- **IPC\_INFO**: 返回系统范围内的信号量的限制和参数，保存到第 4 个参数的成员 **struct seminfo** 所定义的缓存中。

**注 2:** 第四个参数 **union semun arg** 保存信号量状态，是一个联合体，其结构定义如下（在 **bits/sem.h** 或 **linux/sem.h** 中定义）：

```
#include <sys/sem.h>
union semun {
    int val;                /* value for SETVAL , 即信号量的值*/
    struct semid_ds *buf;   /*buffer for IPC_STAT &IPC_SET*/
    unsigned short *array; /*array for GETAL& SETALL*/
    struct seminfo *__buf;  /*buffer for IPC_INFO */
    void *__pad;           /*for system use */
};
```

需要说明的是，有的系统中，例如 **CentOs 6.5**, **Ubuntu** 等，在头文件 **sys/sem.h** 中声明了该联合体，有的系统中可能没有对其定义，编程时需要自己声明。

简单编程时主要使用其中的成员 **semun.val**，即信号量的值。其它几个成员这里给出其定义，仅供参考。

结构体 **semid\_ds** 在 **<linux/sem.h>** 中定义，原型如下：

```
struct semid_ds {
    struct ipc_perm sem_perm; /* Ownership and permissions see ipc.h */
    time_t sem_otime;        /* Last semop time */
    time_t sem_ctime;        /* create/last semctl() time */
    struct sem *sem_base;    /* ptr to first semaphore in array */
    struct sem_queue sem_pending; /* pending operations to be processed */
    struct sem_queue **sem_pending_last; /* last pending operation */

    struct sem_undo *undo;    /* undo requests on this array */
    unsigned long sem_nsems; /* No. of semaphores in set */
};
```

其中，全局结构体 **ipc\_perm** 在 **<sys/ipc.h>** 中定义，原型如下

```
struct ipc_perm {
    key_t key;    /* Key supplied to semget() */
    gid_t uid;    /* Effective UID of owner */
    gid_t gid;    /* Effective GID of owner */
};
```



```

uid_t cuid;      /* Effective UID of creator */
gid_t cgid;      /* Effective GID of creator */
unsigned short mode; /* Permissions */
unsigned short seq; /* Sequence number */
};

```

系统中因等待信号量而睡眠的进程，都分别对应一个 sem\_queue 结构，其原型如下：

```

struct sem_queue {
    struct sem_queue* next; /* next entry in the queue */
    struct sem_queue** prev; /* previous entry in the queue, *(q->prev=q) */
    struct task_struct* sleeper; /* this process , struct task_struct 即 PCB*/
    struct sem_undo* undo; /* undo structure*/
    int pid; /* process pid of requesting process*/
    int status; /* completion status of operation */
    struct semid_ds* sma; /*semaphore semid_ds array operations */
    int id; /* internal sem id*/
    struct sembuf* sops; /*array of pending operations */
    int nsops; /* numbers of operations*/
    int alter; /* operation will alter semaphore */
}

```

操作系统对信号量操作的一些限制条件，如信号量的最大值、信号量的最大描述符、系统中信号量的最大数量、每个 semop()调用能够操作的信号量的最大值等，在结构体 seminfo 中进行了定义，详情参见<linux/sem.h>中的说明。原型如下：

```

struct seminfo {
    int semmap; /* # of entries in semaphore map */
    int semmni; /* max # of semaphore identifiers */
    int semmns; /* max # of semaphores in system */
    int semmnu; /* num of undo structures system wide */
    int semmsl; /* max num of semaphores per id */
    int semopm; /* max num of ops per semop call */
    int semume; /* max num of undo entries per process */
    int semusz; /* sizeof struct sem_undo */
    int semvmx; /* semaphore maximum value */
    int semaem; /* adjust on exit max value */
};

```

表 4-2 给出了几个常见系统中有关信号量限制的参数。

表 4-2 信号量限制参数



说明	典型值			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
任一信号量的最大值	32 767	32 767	32 767	65 535
任一信号量的最大退出时的调整值	16 384	32 767	16 384	32 767
系统中信号量集的最大数量	10	128	87 381	128
系统中信号量的最大数量	60	32 000	87 381	导出的
每个信号量集中的信号量的最大数量	60	250	87 381	512
系统中 undo 结构的最大数量	30	32 000	87 381	导出的
每个 undo 结构中 undo 项的最大数量	10	无限制	10	导出的
每个 semop 调用中操作的最大数量	100	32	5	512

## 2、利用命令 `ipcs -s` 观察信号量数组的情况

例如

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

其中，

`semid` 信号量的标识号

`nsems` 信号量的个数

其它字段意义同共享内存所述。

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

### 4.2.3 消息队列

消息队列是操作系统内核控制并发进程间共享资源的另一种进程间通信机制，系统对于消息队列的管理，符合生产者-消费者模型。

进程之间利用消息队列进行通信时，需要经过下面几个步骤：

(1) 首先需要其中的一个进程调用 `msgget()` 创建一个消息队列，其它进程调用 `msgget()` 获取该消息队列的描述符。

(2) 进程调用 `msgsnd()` 给消息队列发送消息，或调用 `msgrcv ()` 从消息队列接收消息。

(3) 进程调用 `msgctl()` 删除消息队列

## 1、消息队列有关的系统调用

下面介绍的这些系统调用，如果调用不成功，可以通过 `perror`(“系统调用名”) 获取错误原因。

### (1) 创建消息队列的系统调用

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

功能：创建一个新的键值为 `key` 的消息队列，或获取一个已经存在的键值为 `key` 的消息队列标识。

传入参数：

- `key`：消息队列的键值，可以为 `IPC_PRIVATE`，也可以指定一个整数。
- `flags`：消息队列权限位，含义同共享内存权限位和信号量权限位。

返回值：

调用成功：如果 `key` 用新整数指定（`key` 对应的消息队列不存在），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 `msgid`。如果指定的整数 `key` 已存在则返回与 `key` 关联的标识符 `msgid`。

调用不成功，返回-1，错误原因存于 `error` 中。

错误代码：

- `EACCES`：指定的消息队列已存在，但调用进程没有权限访问它，而且不拥有 `CAP_IPC_OWNER` 权能
- `EEXIST`：`key` 指定的消息队列已存在，而 `msgflg` 中同时指定 `IPC_CREAT` 和 `IPC_EXCL` 标志
- `ENOENT`：`key` 指定的消息队列不存在，同时 `msgflg` 中没有指定 `IPC_CREAT` 标志
- `ENOMEM`：需要建立消息队列，但内存不足
- `ENOSPC`：需要建立消息队列，但已达到系统的最大消息队列容量

注：其中的键值 `key` 是该消息队列在系统中的唯一标识。一个进程基于该 `key` 通过系统调用 `msgget` () 创建一个消息队列，如果创建成功，`msgget` () 返回该消息队列的标识符，其它要访问该消息队列的进程需要基于该键值 `key` 通过系统调用 `msgget` () 获取该消息队列的标识符，然后基于该标识符访问该消息队列。

### (2) 发送一条新消息到消息队列的系统调用

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

功能：将消息缓冲区 `msgp` 中，数据长度为 `msgsz` 的消息发送到 `msqid` 所指定的消息队列中。当消息队列已满时，发送进程是否阻塞，由 `msgflg` 的设置决定。系统对于消息队列的管理，符合生产者-消费者模型。

传入参数：

- `msqid` : `msgget()`返回的消息队列的标识符
- `msgp`: 消息缓冲区指针，暂存将要发送的消息，是一个结构体。
- `msgsz`: 消息数据的长度 */\*纯消息数据的长度，不包括消息类型\*/*
- `msgflg`: 设置为 0，表示阻塞方式  
设置为 `IPC_NOWAIT`，表示非阻塞方式  
设置为 `MSG_NOERROR`，表示截取消息数据，不返回错误

返回值：调用成功返回 0，不成功返回-1，错误原因存于 `error` 中。

错误代码：

- EACCES: 调用进程在消息队列上没有写权能，同时没有 `CAP_IPC_OWNER` 权能
- EAGAIN: 消息队列已满，且 `msgflg` 中指定 `IPC_NOWAIT` 标志，消息不能被发送
- EFAULT: `msgp` 指针指向的内存空间不可访问
- EIDRM: 消息队列已被删除
- EINTR: 消息队列已满，进程等待时被信号中断
- EINVAL: 参数无效
- ENOMEM: 系统内存不足，无法将 `msgp` 指向的消息拷贝进来

注 1: 关于第二个参数 `msgp`

`msgp` 指向一个消息缓冲区，存放一条将要发送的消息。消息缓冲区的结构为（在头文件 `<sys/msg.h>` 中定义）：

```
struct msgbuf {  
    long mtype; /* 消息类型，必须>0 */  
                /* msgrcv()根据该值接收相应类型的消息 */  
    char mtext[1]; /* 消息数据，长度应于 msgsz 声明的一致 */  
};
```

编程时，可根据具体的消息数据大小，使用 `malloc()`函数申请相应大小的消息缓冲区。

注 2: 关于第四个参数 `msgflg`

一般 `msgflg` 可取：0，`IPC_NOWAIT`，`MSG_NOERROR`

如果设置 `msgflg=0`，则采用阻塞方式。即当消息队列满的时候，`msgsnd()`将阻塞发送进程，直到接收进程取走消息时将其唤醒。

如果设置 `msgflg=IPC_NOWAI`，则采用非阻塞方式。即如果消息队列已满，`msgsnd()`不阻塞发送进程，而是直接返回到调用进程，并返回错误 `EAGAIN`。

如果设置 `msgflg=MSG_NOERROR`，当要发送的消息数据的字节数大于 `msgsnd()`中给出的消息字节数（由参数 `msgsz` 指定），则截取消息数据的前 `msgsz` 个字节，其余部分将被丢弃，且不通知发送进程。

可用命令 `ipcs -l` 查看系统设定的消息队列最大长度、每个消息最大字节数、消息队列最大占用的字节数等限制信息。

### （3）从消息队列中结束（读出）一条消息的系统调用

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

功能：从 `msqid` 所标识的消息队列中，取走由 `msgtype` 所指定类型的消息，存放到 `msgp` 所指向的消息缓冲区中，如果 `msgsz` 小于消息的数据长度，则截取 `msgsz` 个字节的消息数据，存放到消息缓冲区 `msgp` 中。若消息队列中没有 `msgtype` 类型的消息可以接收，是否阻塞接收进程由 `msgflg` 的设置决定。

当接收到一个符合接收类型的消息时，`msgrcv()`会将该消息从消息队列中删除。

传入参数：

- `msqid` 由消息队列的标识符
- `msgp` 消息缓冲区指针。
- `msgsz`：消息数据的长度
- `msgtype` 决定从队列中接收哪条消息，分为三种情况：
  - `msgtype = 0`：接收消息队列中第一条消息
  - `msgtype > 0`：接收消息队列中等于 `msgtype` 类型的第一条消息。
  - `msgtype < 0`：接收类型等于或小于 `msgtype` 绝对值，且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

返回值：调用成功，返回实际读到的消息数据长度

调用不成功，返回 -1，错误原因存于 `error` 中。

错误代码：

- `E2BIG`：消息数据长度大于 `msgsz` 且 `msgflg` 没有设置 `IPC_NOERROR`
- `EIDRM`：标识符为 `msqid` 的消息队列已被删除
- `EACCESS`：无权限读取该消息队列

- EFAULT: 参数 msgp 指向无效的内存地址
- ENOMSG: 参数 msgflg 设为 IPC\_NOWAIT, 而消息队列中无消息可读
- EINTR: 等待读取队列内的消息情况下被信号中断

注 1: 关于第二个参数 msgp

msgp 指向一个消息缓冲区, 存放一条从消息队列中接收到的消息。消息缓冲区的结构为 (在头文件 <sys/msg.h> 中定义):

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

注 2: 关于第五个参数 msgflg

一般 msgflg 可取: 0, IPC\_NOWAIT, MSG\_NOERROR

GNU 版本还实现了: MSG\_EXCEPT、MSG\_COPY

如果设置 msgflg=0, 则采用阻塞方式。当消息队列中没有指定类型的消息可接收时 (参数 msgtype 指定了可以接收的消息类型), msgrcv() 将阻塞接收进程, 直到能够取到该类型的消息为止。

如果设置 msgflg= IPC\_NOWAI, 则采用非阻塞方式。当消息队列中没有 msgtype 所指定类型的消息可接收时, msgsnd() 不阻塞接收进程, 而是直接返回到调用进程, 并返回错误 ENOMSG。

如果设置 msgflg= MSG\_NOERROR, 当消息队列中有 msgtype 所指定类型的消息可接收, 但消息的数据大于所请求的字节数 (参数 msgsz 指定了要接收消息数据的字节数), 则截取消息数据的前 msgsz 字节, 其余部分将被丢弃, 且不通知接收进程。

GNU 版本还可以在 msgflg 中设置 MSG\_EXCEPT 和 MSG\_COPY, 目前 Ubuntu 和 CentOS 中尚未实现。

其中, IPC\_EXCEPT 与 msgtype 配合, 表示接收消息队列中第一个类型不为 msgtype 的消息。

MSG\_COPY 表示当接收到指定类型的消息后, 不将该消息从消息队列中删除, 只是复制该消息到消息缓冲区中。

#### (4) 控制消息队列的系统调用

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

功能：根据 cmd 中给出的命令，对 msqid 标识的消息队列施加相应的操作。  
通常用来删除一个消息队列。

传入参数：

- msqid：消息队列的标识符
- cmd：一组 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，读取消息队列的 msgid\_ds 结构，写到 buf 中  
IPC\_SET 改变消息队列的 UID、GID、访问权限等设置
- buf：指向存储结构体 msgid\_ds 的缓冲区指针

返回值：调用成功返回 0，不成功返回-1，错误原因存于 error 中。

错误代码：

- EACCES：cmd 使用 IPC\_STAT，进程对消息队列没有读权限
- EFAULT：cmd 使用 IPC\_SET 与 IPC\_STAT，buf 指向的地址无效
- EIDRM：在读取消息队列时，队列被删除
- EINVAL：msgid 无效, 或者 msgsz 小于 0
- EPERM：cmd 使用命令 IPC\_SET 或 IPC\_RMID，但进程没有写的权限

注：关于参数 cmd 和 msgid\_ds

若 cmd 取 IPC\_RMID，如 msgctl( msqid, IPC\_RMID, 0)，删除 msqid 标识的消息队列。

若 cmd 取 IPC\_STAT，则 msgctl( msqid, IPC\_STAT, buf)读取消息队列的数据结构 msgid\_ds，并将其存储在 buf 指定的缓冲区中。

结构体 msgid\_ds 在头文件<sys/msg.h>中声明，具体在<asm/msgbuf.h>中定义，如下所示：

```
struct msgid_ds {
    struct ipc_perm msg_perm; /* Ownership and permissions */
    time_t msg_stime;        /* last msgsnd time */
    time_t msg_rtime;        /* last msgrcv time */
    time_t msg_ctime;        /* last change time */
    ulong_t msg_cbytes;      /* current number of bytes on queue */
    ulong_t msg_qnum;        /* number of messages in queue */
    ulong_t msg_qbytes;      /* max number of bytes on queue */
    pid_t msg_lspid;         /* pid of last msgsnd */
    pid_t msg_lrpid;         /* last receive pid */
    ulong_t __unused4;
    ulong_t __unused5;
};
```

其中，全局结构体 `ipc_perm` 在 `<sys/ipc.h>` 中定义，原型如下

```
struct ipc_perm {
    key_t key;      /* Key supplied to msgget() */
    gid_t uid;      /* Effective UID of owner */
    gid_t gid;      /* Effective GID of owner */
    uid_t cuid;     /* Effective UID of creator */
    gid_t cgid;     /* Effective GID of creator */
    unsigned short mode; /* Permissions */
    unsigned short seq; /* Sequence number */
};
```

若 `cmd` 取 `IPC_SET`，首先需要在 `buf` 定义一个结构体 `msqid_ds` 变量，对其 `ipc_perm` 中的成员设置相应的值，则 `msgctl( msqid, IPC_SET, buf)` 会根据你的设置，修改消息队列的数据结构 `msqid_ds` 中的 `ipc_perm` 元素的值。主要用来改变消息队列的 UID、GID 以及访问权限等设置。

## 2、linux 中可用命令 `ipcs -q` 观察消息队列的情况。

例如：

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

其中，

- `msgmid`: 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

其它字段意义与共享存储区、信号量两种机制所述相同。

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。

### 4.2.4 常用 IPC 命令

前面已经提到，可以利用命令 `ipcs -m` 观察共享内存情况，`ipcs -s` 观察信号量数组的情况，`ipcs -q` 观察消息队列的情况。我们还可以通过一些命令删除 IPC 对象，也可以访问操作系统为 IPC 对象临时创建的几个虚拟文件查看它们的有关信息。

1、`ipcs`，或 `ipcs -a`，查看目前系统的所有的 ipc 对象资源

2、`ipcrm` 命令

在权限允许的情况下可以使用 `ipcrm` 命令删除系统当前存在的 IPC 对象中的一个对象。

#### (1) 通过 IPC 对象的 id 号删除相应的 IPC 对象

格式: `ipcrm -[m,s,q] id`, 删除 id 号标识的 IPC 对象。例如,

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

#### (2) 通过 IPC 对象的键值 key 删除相应的 IPC 对象

格式: `ipcrm -[M,S,Q] key`, 删除键值 key 标识的 IPC 对象。例如,

- `ipcrm -m 66666`, 删除键值为 0x66666 的共享内存。
- `ipcrm -s 77777`, 删除键值为 0x77777 的信号量数组。
- `ipcrm -q 88888`, 删除键值为 0x88888 的消息队列。

### 3、临时虚拟文件

linux 在运行时, 在根目录下临时创建了一个临时目录 `/proc`。 `/proc` 是一个伪文件系统, 它只存在内存当中, 而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。

用户和应用程序可以通过访问 `proc` 中相关的文件得到系统的信息, 并可以改变内核的某些参数。由于系统的信息, 如进程, 是动态改变的, 所以用户或应用程序读取 `proc` 文件时, `proc` 文件系统是动态从系统内核读出所需信息并提交的。

例如, 命令 `procinfo` 可以获取一些系统信息; `/proc/cmdline` 文件给出了内核启动的命令行; `/proc/cpuinfo` 文件提供了有关系统 CPU 的有关参数; `/proc/devices` 文件列出字符和块设备的主设备号, 以及分配到这些设备号的设备名称; `/proc/filesystems` 文件列出可供使用的文件系统类型; `/proc/interrupts` 文件给出一些中断信息; `/proc/meminfo` 文件给出了内存状态; `/proc/mounts` 文件给出当前系统所安装的文件系统信息; `/proc/pci` 文件给出 PCI 设备的信息; `/proc/stat` 文件包含的信息有 CPU 利用率, 磁盘, 内存页, 内存对换, 全部中断, 接触开关以及上次启动时间 (自 1970 年 1 月 1 日起的秒数); `/proc/net` 子目录下的文件描述了一些网络行为; `/proc/sys` 子目录中的许多项都可以用来调整系统的性能。总之, `/proc` 文件系统包含了大量的有关当前系统状态的信息。

其中, `/proc/sysvipc` 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



可以在命令窗口中访问这三个文件，或者在编程时访问这三个文件，获取有关 IPC 对象的当前的实时信息。

### 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，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。

基于消息队列的进程间通信，由于系统已经实现了发送进程与接收进程之间的同步控制，符合生产者-消费者模型，实现上比较简单，同学可课后自行练习。

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) 2022 by
 *      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);

/*信号量控制用的共同体*/
typedef union semuns {
    int val;
} Sem_uns;
```

```

/* 消息结构体*/
typedef struct msgbuf {
    long mtype;
    char mtext[1];
} Msg_buf;

//生产者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],column[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] != ' ') column[j++] = line[i++];
        column[j] = '\0';
        if(atoi(column) != key) continue;
        j=0;
        while(line[i] == ' ') i++;
        while(line[i] != ' ') column[j++] = line[i++];
        column[j] = '\0';
        i = atoi(column);
        fclose(pf);
        return i;
    }
    fclose(pf);
    return -1;
}

/*
 *      信号量上的down/up 操作
 *      semid: 信号量数组标识符
 *      semnum: 信号量数组下标
 *      buf: 操作信号量的结构
 */
int down(int sem_id) //P 操作
{
    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) //V 操作
{
    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 个信号量的信号量
 *      如果建立成功, 返回 一个信号量数组的标识符 sem_id
 *      输入参数:
 *      sem_key 信号量数组的键值
 *      sem_val 信号量数组中信号量的个数
 *      sem_flag 信号量数组的存取权限
 */
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 标识的信号量数组是否已经建立
    if((sem_id = get_ipc_id("/proc/sysvipc/sem",sem_key)) < 0 )
    {
        //semget 新建一个信号量,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 * 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; //生产者同步信号量键值
    pmtx_key = 202; //生产者互斥信号量键值
    cons_key = 301; //消费者同步信号量键值
    cmtx_key = 302; //消费者互斥信号量键值
    sem_flg = IPC_CREAT | 0644;

```

```

//生产者同步信号量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量, 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, 同步信号量初值设为0
sem_val = 0;

//获取消费者同步信号量, 引用标识存 cons_sem
cons_sem = set_sem(cons_key,sem_val,sem_flg);
//生产者互斥信号量初值为1
sem_val = 1;
//获取生产者互斥信号量, 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;    //生产者同步信号量键值
    pmtx_key = 202;    //生产者互斥信号量键值
    cons_key = 301;    //消费者同步信号量键值
    cmtx_key = 302;    //消费者互斥信号量键值
    sem_flg = IPC_CREAT | 0644; //信号量操作权限
    //生产者同步信号量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量，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，同步信号量初值设为0
    sem_val = 0;
    //获取消费者同步信号量，引用标识存 cons_sem
    cons_sem = set_sem(cons_key,sem_val,sem_flg);
    //消费者互斥信号量初值为1
    sem_val = 1;
    //获取消费者互斥信号量，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，生产者重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，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？对应教材中信号量的定义，说明信号量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物理意义是什么？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 5 进程互斥

## 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,它应能实现一下功能:

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 0 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) 2022 by  
  
 *   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  
/*信号量控制用的共同体*/  
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
*/
#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<STRSZ; 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;
                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 = READERREQUEST;
    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;s
}

```

## 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：

```
$gmake
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: AAAAAAAAAA
```

```
.....
```

```
$/reader 1
```

```
3904 reader quest
```

```
3904 reading: AAAAAAAAAA
```

```
3904 reader quest
```

```
3904 reading: AAAAAAAAAA
```

```
3904 reader quest
```

```
3904 reading: AAAAAAAAAA
```

```
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。

13) 如果不采用上述机制，如何在读者-写者问题中添加一个信号量，实现写者优先？

## 5.4 独立实验

睡眠理发师问题：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。

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，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？为什么会出现进程饥饿现象？本实验的饥饿现象是怎样表现的？怎样解决并发进程间发生的饥饿现象？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 6 死锁问题

### 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) 2022  
 * Function : 声明 IPC 机制的函数原型和哲学家管程类  
 */  
  
#include <iostream.h>  
#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>
```

```

/*信号量控制用的共同体*/
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() { }

/*
 * 信号量上的 down/up 操作
 * semid: 信号量数组标识符
 * semnum: 信号量数组下标
 * buf: 操作信号量的结构
 */
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 {
        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，
 * 将其从饥饿中唤醒。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 个信号量的信号量
 * 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 信号量中信号量的个数
 *   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 新建一个信号量,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;

    //为防止两个相邻的哲学家同时就餐, 导致共有同一只筷子, 允许5个中只有1个
    // 在就餐, 建立一个初值为1的用于锁的信号量
    if ((sem_id = set_sem(sem_key++, 1, ipc_flg)) < 0) {
        perror("Semaphore 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量
    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
    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
}

```

```

    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

```
$ gmake
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 中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。请模拟实现两个方向行车，而不会出现撞车或长时间等待的情况。请构造一个管程来解决该问题。

## 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。分析示例实验是否真正模拟了哲学家就餐问题？为什么示例程序不会产生死锁？为什么会出现进程死锁和饥饿现象？怎样利用实验造成和表现死锁和饥饿现象？管程能避免死锁和饥饿的机理是什么？您对于管程概念有哪些新的理解和认识？条件变量和信号量有何不同？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？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？你的独立实验程序是怎样解决单行道问题的？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。

## 实验 7 存储管理

### 7.1 实验目的

通过获取进程空间，理解 Linux 中进程的存储管理方式与进程地址空间的组成。

### 7.3 实验说明

查阅资料，至少实现 linux 命令 `pmap [-x -X]` 的功能，其它功能自选。