课题一 编写一个小型命令处理器 smallshell

一、目标

设计并实现一个简单的命令处理程序，名字为 smallshell。要求具备以下基本功能：

1.支持交互式的用户界面

2.支持 5 条以上的内置命令：cd，exit，ls，cp，pwd 等。

3.支持可执行文件的运行

4.支持输入输出重定向功能

5.支持管道功能

二、准备知识

1.交互式用户界面

在 smallshell 中，显示如下格式的交互界面：

［sername@servername：pathname］$

需要涉及以下函数：

```
＃include<uniste.h>
char *getlogin（void)；
/*getlogin 函数返回与当前用户关联的用户名*/

int gethostname (char *name,size_t namelen);
/*gethostname 函数把机器的网络名写到字符串 name 中，name 的长度为 namelen，因此该字符串的长度不得超过 namelen 个字符，函数成功返回 0，否则返回-1。*/

#include<unistd>
char *getcwd(char *name,size_t size);
/*getcwd 函数返回一个指向当前工作目录的指针，并将当前工作目录存于 name 中，如果该目录名长度超过 size 给出的长度，返回 NULL。若 size 为 0，返回-1。*/
```

例如，下面这段程序模仿了 pwd 命令：

```
#include<stdio.h>
#include<unistd.h>
#define VERYBIG 200
void my_pwd(void);
main()
{
my_pwd();
}
void my_pwd(void);
{
char dirname[VERYBIG];
if(getcwd(dirname ,VERYBIG)==NULL)
```

```
        printf("getcwd error");
    else
        printf("%s",dirname);
}
```

2.支持内置命令：cd，exit，设置搜索路径 path 等

如果用户输入内置命令，smallshell 根据命令名及参数进行相应处理。下面以 cd，exit，path 命令为例，分别说明其实现所涉及的函数。

（1）cd 命令

该命令用于切换当前目录，可以通过 chdir（）函数实现。chdir（）使当前目录变为 path 所指向的目录，该函数的用法如下：

```
#include<unistd.h>
int chdir(const char *path);
```

chdir 在失败的情况下返回-1。需要注意的是，chdir 只影响调用进程，并不会影响启动这个进程的 shell 进程。

例如，使用 chdir（"/usr/bin"）；可切换到/usr/bin 目录。

（2）exit 命令

该命令用于退出 smallshell，需要调用 exit 函数实现， exit 函数用法如下：

```
#include<stdlib.h>
void exit(int status);
```

(3) path 命令

该命令用于设置搜索路径。可以设置一个全局变量 gpath，实现对搜索路径的更新。

3.支持可执行文件的运行

如果用户输入的不是内置命令，而是一个可执行文件名，则需要在设置的路径中搜索该命令，并在此环境中执行。可以通过 access（）函数先对该命令进行测试：

```
#include<unistd.h>
int access(const char *pathname, int mode);
```

该函数根据用户 id 测定进程是否具有访问某个特定文件的权限。参数 pathname 表示的是文件的名字，参数 mode 有三种可能的取值，它们的定义在<unistd.h>中可以找到：

R_OK　调用进程是否具有读访问权限

W_OK　调用进程是否具有写访问权限

X_OK　调用进程是否具有执行权限

access 函数返回 0，表示用户对文件具有访问权限，返回-1 表示不具有此种访问权限。

如果命令可执行，下一步是创建 smallshell 的子进程，之后让子进程执行这个可执行文件。因此需要用到 fork 和 exec 类函数。fork 函数的使用请参考课程实验指导书，exec 类函数有 6 个，这里可以用其中的 execve（）函数，用法如下：

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

pathname 参数是子进程要执行的文件的路径名，argv 是一个参数字符串列表，envp 是一个

环境变量字符串和值的列表。execve 成功不会返回，失败返回-1.
例：
cmd=”helloworld”; //helloworld 是一个可执行文件
char * envp[]={“PATH=/home/stu1”,0};
execve(cmd,NULL,envp); //执行 helloworld

4.支持输入输出重定向功能

每个进程的描述符中包含一个 file_struct 结构，记录用户的文件打开情况，这个结构称为用户打开文件表。并且用户打开文件表的信息在 fork（）和 exec（）调用后将仍然保持和继承。
进程的打开文件表记录进程已经打开的文件描述符，即指向文件对象的指针。通常系统为正在执行的进程自动打开三个文件：标准输入（键盘），标准输出（显示器），标准错误。通常对应文件描述符 0，1，2 。因此，输入输出重定向实际就是将标准输入和标准输出的文件描述符指向用户指定的文件。可以使用下列函数实现：
＃include<unistd.h>
fid=open(const char *pathname, int flags);
int dup2(int oldfd, int newfd);
open 函数打开一个文件，返回该文件的描述符。参数 pathname 是文件的路径名，flag 指定文件存取模式，取值为：O_RDONLY,O_WRONLY,O_RDWR,O_CREATE 等。
dup2 函数将老的文件描述符 oldfd 复制到新的文件描述符 newfd，即创建一个 oldfd 的拷贝。
例如：

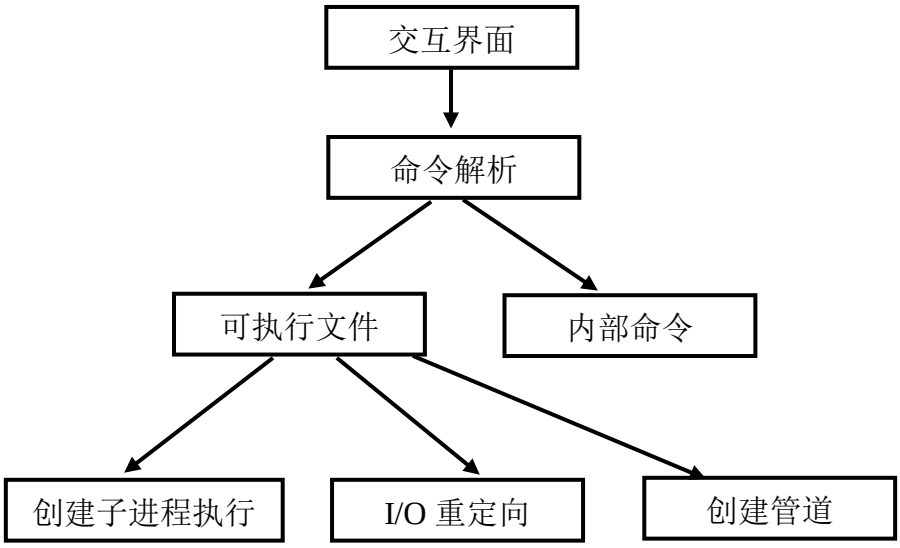
```
    fid=open(filename,O_WRONLY|O_CREAT);
 //以写的方式创建并打开文件 filename
      dup2(fid,1); //将文件描述符复制到标准输出，即输出时便输出到新建文件中，
                   //从而实现了输出重定向。
```

5.支持管道功能

shell 中的管道功能是将一个命令的输出作为另一个命令的输入。可以用 pipe 函数创建无名管道。返回的 pipe 文件描述符只能被同一家族的父子进程使用， 因此可以用 fork 函数和 exec 函数创建两个不同的进程并使其具有父子关系，之后使一个进程的输出重定向到管道，另一个进程的输入重定向到管道，从而实现管道功能。程序框架如下：

```
int   fd[2];
pipe(fd);
if(fork>0)   {       //父进程
                    close(fd[0]);   //关闭用于读文件的描述符
                     dup2(fd[1],1); //父进程实现输出重定向
                    ......
                     }
 else {         //子进程
          close(fd[1]); //关闭用于写的文件描述符
          dup2（fd[0],0); //子进程实现输入重定向
          ......
     }
```

三、smallshell 总体结构



课题二　使用 IPC 机制实现“生产者-过滤者-消费者”问题

一、 目标

学习 Linux 进程间通信机制，使用信号量和共享内存实现进程同步问题“生产者-过滤者-消费者”问题。具体要求：

1.创建信号量集，实现同步互斥信号量。

2.创建共享内存，模拟存放产品的公共缓冲池。

3.创建并发进程，实现进程对共享缓冲池的并发操作。生产者生产产品后放入缓冲池，过滤者取出产品，对产品过滤（可自己设计，如对产品值进行+1）再放入缓冲池，消费者将过滤后的产品取走。可创建一个或两个缓冲池。

二、准备知识

1.信号量集

（1）创建信号量集

＃include <sys/sem.h>

int semget(key_t key,int nsems, int permflags);

该调用与文件的 open 或 creat 相似，参数 key 是一个标识信号量集的数，参数 nsems 是信号量集中包含的信号量的个数,元素索引从 0 到 nsems-1，permflags 是 semget 函数将实现的操作，有两个值供选择，在头文件<sys/ipc.h>有定义，这两个值可以单独使用，也可用“或”位运算符连起来使用：

IPC_CREAT :创建一个 key 所代表的新信号量集，类似于 creat。

IPC_EXCL:同 IPC_CREAT 都设置时，若 key 对应信号量集存在，则返回-1，并将错误指示变量 errno 置 EEXIST。

例如：semid=semget((key_t)0010,1, 0600|IPC_CREAT|IPC_EXCL);

注意：这里 0600 表示对该信号量集的存取权限，设置的方式同文件存取权限的设置一致。0600 表示对信号量集的创建者具有读写权限，对同组用户和其他用户无任何权限。

（2）信号量控制

#include<sys/sem.h>

int semctl(int semid,int sem_num,int command,union semun ctl_arg);

该函数可以对 semid 表示的信号量集中的第 sem_num 号信号量执行 command 指示的操作。sem_num 是信号量的索引号，注意，索引号从 0 开始。command 给出要完成何种功能，功能可分为种：标准 IPC 功能、只对单个信号量起作用的功能、影响整个信号量集的功能。

标准 IPC 函数

IPC_STAT：把状态信息放入 ctl_arg.buf

IPC_SET:用 stl_arg.buf 中的值设置信号量的所有权/许可权

IPC_RMID：从系统中删除信号量集（此时 sem_num 参数忽略）

单信号量操作

GETVAL：返回信号量的值

SETVAL：设置信号量的值，写入 ctl_arg.val

GETPID：返回最后一个对 sem_num 号信号量执行 semop 操作的进程 id

GETNCNT：返回等待 sem_num 号信号量值增加的进程数，即在该信号量上等待的进程数
GETZCNT：返回等待 sem_num 号信号量值变为 0 的进程数。
全信号量操作
GETALL：返回所有信号量的值，结果保存在 ctl_arg.array,忽略参数 sem_num
SETALL：通过 ctl_arg.array 更新所有信号量的值。
参数 sem_num 与上述第二类功能选项一起使用，以针对某一个信号量。参数 ctl_arg 是调用该函数的进程必须事先定义的一个联合体，定义如下；

```
union semun {
int val;
struct semid_ds *buf;
unsigned short   *array
};
union semun ctl_arg;
```

联合体中的三个成员分别对应 semctl 的三个功能，val 是针对信号量值的，例如赋初值。结构体 semid_ds 是 IPC_STAT 和 IPC_SET 的缓冲，array 针对全信号量操作。
（3）信号量操作

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

该函数能够实现信号量的 p、v 操作。参数 semid 是信号量集的标识符，op_array 指向 sembuf 结构体类型的数组，参数 num_ops 是操作的个数。下面给出 sembuf 的结构：

```
struct sembuf {
unsigned short sem_num; //  存放信号量在信号量集合中的索引
        short sem_op;   //1，-1 分别可表示加一和减一操作，相当与 V、P 操作
        short sem_flg; //标志，设置 SEM_UNDO 表示进程退出时自动还原
}
```

例如，下面这段程序能够实现针对一个信号量的 P 操作：

```
int P(int semid)
        // semid 是信号量集的标识符
{ struct sembuf p_buf;

    p_buf.sem_num=0;

    p_buf.sem_op= -1;

    p_buf.sem_flg=SEM_UNDO;

 if(semop(semid,&p_buf,1)==-1)              //第三个参数是 p_buf 指向的 sembuf 数组的大小

    { perror("p(semid) failed");

      exit(1);

    }
```

return(0);

}

2.共享存储

共享存储操作适得两个或两个以上的进程可以共用一段物理内存（一般情况下，两个进程的数据区是完全独立的，父进程用 fork 创建子进程后，子进程会复制父进程数据到自己的数据区）。

（1）创建共享内存

#include<sys/shm.h>

int shmget(key_t key,size_t size, int permflags);

参数 key 是共享内存的标识，size 是共享内存段的最小字节数，permflags 是访问权限，值的设置同 semget 一样。

（2）共享内存的控制

#include<sys/shm.h>

int shmctl(int shmid, int command, struct shmid_ds *shm_stat);

该函数 semctl 相似，command 可设为 IPC_STAT,IPC_SET,IPC_RMID。参数 shm_stat 指向存放属性的结构体，具体内容请参考手册。

（3）共享内存的附接和断开

＃include<sys/shm.h>

void *shmat(int shmid, const void *addr, int shmflags);

int shmdt(const void *addr);

由于两个函数需指出进程地址空间中的地址，因此比较复杂。简化的方法是将 shmat 中的地址设为 NULL。

三、程序框架

```
// semaphore header file
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<errno.h>
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>

typedef union _semun {
        //定义联合体
.....

}semun;

struct databuf{          //定义“生产者-消费者”问题中存放数据的 buffer
```

```
.....
};

static int shmid,semid;
void initshm(struct databuf **p)        //创建共享存储区，并附接到调用进程

{

   .....
}
int initsem( )
      //创建信号量集，并初始化
{

  .....
}

void remobj(void)
        //删除 IPC 对象
{
     if(shmctl(shmid,IPC_RMID,NULL)==-1) printf("shmctl error");
   if(semctl(semid,IPC_RMID,0)==-1) printf("\nsemctl remobj error\n");

}

int P( )
   //定义 p 操作
{
.....
      }

int V( )   //定义 v 操作
{
  .....
}

void producer(    )       //生产者
{
   .....
}
void filter（）//  过滤者
{
}
```

```
void consumer( )
   //消费者
{
  ......
}

main()   //主程序
{
  key_t semkey，shmkey;
  struct databuf *buf;          //产品缓冲池

......
        initshm(&buf);              //创建共享存储区，并附接到 buf 指针指示的位置
        semid=initsem(semkey);   //创建信号量集
       ......                          //创建若干并发进程，实现 producer 与 consumer 的并发执行
        remobj();                        //删除 IPC 对象
}
```

课题三　使用 IPC 机制实现公平的“过桥”问题

一、 目标

学习进程间通信机制，使用信号量和共享内存实现进程同步问题：“过桥”问题。具体要求：
1.创建信号量集，实现互斥信号量。
2.创建共享内存，将共享的计数值保存其中。
3.创建并发进程，实现南北两个方向的车的并发操作。具体为：由南往北的车和由北往南的车不能同时在桥上，但同方向的车可以同时在桥上通行。

二、准备知识

见课题二

三、程序框架

```
// semaphore header file
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<errno.h>
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>

typedef union _semun {
            //定义联合体
......

}semun;

static int shmid,semid;

void initshm(struct databuf **p)        //创建共享存储区，并附接到调用进程
{
   ......
}

int initsem( )
      //创建信号量集，并初始化
{
 ......
}
```

```
void remobj(void)
        //删除 IPC 对象
{
  if(shmctl(shmid,IPC_RMID,NULL)==-1) printf("shmctl error");
  if(semctl(semid,IPC_RMID,0)==-1) printf("\nsemctl remobj error\n");
}

int P( )
   //定义 p 操作
{
......
       }

int V( )   //定义 v 操作
{
  ......
}

void NtoS(    )       //由北往南的车
{
    ......
}

void StoN( )   //由南往北的车
{
  ......
}

main()         //主程序
{
  key_t semkey，shmkey;

    int    *Ncount       //计数值
    int    *Scount       //计数值
......
    initshm(&Ncount);         //创建共享存储区，并附接到 Ncount 指针指示的位置
    semid=initsem(semkey);    //创建信号量集
        ......                          //创建若干并发进程，实现 producer 与 consumer 的并发执行
        remobj();                       //删除 IPC 对象
}
```

课题四　利用信号量和多线程机制实现“哲学家进餐”问题

一、 目标

学习多线程编程，使用线程的同步机制实现“哲学家进餐”问题。具体要求：

1.创建 POSIX 线程，实现多线程的并发执行，验证多线程共享进程资源的特性。

2.使用互斥量和条件变量，或使用信号量实现线程的同步互斥。

3. 验证 “ 哲学家进餐”问题中的死锁情况，并加以解决。

二、准备知识

首先，请自学课程教材中关于线程同步的内容。

POSIX 线程函数库（pthread）为多线程编程提供了一套支持函数，这些函数在头文件 pthread.h 中定义，所有多线程程序都必须包含这个头文件。由于 pthread 库不是 Linux 系统的库，所以编译时，需要用-lpthread 选项来链接线程库。

1.创建线程

```
int pthread_create(    pthread_t *thread,       //  线程 id 指针
                        pthread_attr_t *attr,    //线程属性指针
                       void *(start_routine)(void *),   //返回 void 类型的指针函数
                       void *arg);                      //start_routine 的形参
```

该函数成功返回 0，失败返回非 0 值。

例：

```
/* the example file is createthread.c       */
#include<pthread.h>
#include<stdio.h>
void *create(void *arg)            //新线程执行的函数
{int i;
for (i=0;i<10;i++)
printf("%d new thread created %c\n",i,*(char *)arg);
pthread_exit(NULL);                //退出线程
}
int main( int argc, char *argv[]) //程序从主函数开始
{
pthread_t p_thread;                //线程 ID
int    ret;
char a='A';
char b='B';

ret=pthread_create(&p_thread,NULL,create,(void*)&a);
                                   //创建新线程执行 create 函数，传递参数 a
create((void*)&b);                 //主线程执行此函数，传递参数 b
}
```

编译时使用命令：gcc -lpthread -o createthread createthread.c

2. 等待子线程结束

```
#include<pthread.h>
int pthread_join(pthread_t p_thread,void **status);
```

该函数的第一个参数是线程的 ID，第二个参数为子线程返回的值，该函数返回 0 表示成功，非 0 为失败。下面给出“观察者-报告者”线程并发的例子，体会多线程共享数据可能造成的问题。

```
#include<pthread.h>
#include<stdio.h>
static int    count=10;

 void *observer(void *arg)            //观察者执行的函数
{int i;
for (i=0;i<100 ;i++)
   printf("observer %d count=%d\n", i,(*(int *)arg)++);
pthread_exit(NULL);                  //退出线程
}
void *reporter(void *arg)            //报告者执行的函数
{int i;
for (i=0;i<100;i++)
{
   printf("%d count=%d    \n",i,*(int *)arg);
   *(int*)arg=0;
}
pthread_exit(NULL);                  //退出线程
}
int main( int argc, char *argv[])    //  主函数
{
pthread_t p1_thread,p2_thread;               //线程 ID
pthread_create(&p1_thread,NULL,observer,(void*)&count); //创建新线程
pthread_create(&p2_thread,NULL,reporter,(void**)&count);
pthread_join(p1_thread,NULL);     //主线程等待子线程结束
pthread_join(p2_thread,NULL);
}
```

3.互斥锁

互斥锁可以解决线程之间互斥访问资源的问题。当一个线程锁定一个互斥量后，可以执行临界区，而此时另一个线程若要锁定互斥量，将被挂起，直到前一个线程解锁该互斥量后，后一个线程才能再次锁定互斥量，并进入临界区访问。创建互斥量前，必须先声明一个类型为 pthread_mutex_t 的变量，并对它初始化，例如：

```
pthread_mutex_t=PTHREAD_MUTEX_INITIALIZER;   //该初始化可创建一个快速互斥量
```

下面三个函数分别用来锁定互斥量，解锁互斥量以及销毁互斥量

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

int pthread_mutex_destroy(pthread_mutex_t *mutex);
三个函数执行正确时，返回 0。
4.条件变量
通常互斥量与条件变量同时使用，用互斥量实现线程互斥，用条件变量实现线程间的同步。条件变量是一种使线程等待某些事件发生的机制。线程等待一个条件变量，直到另一个线程给该条件变量发送一个信号将该线程唤醒。
（1）创建条件变量
创建条件变量时，必须先声明一个类型为 pthread_cond_t 类型的变量，并进行初始化。可以简单地使用一个宏 PTHREAD_COND_INITIALIZER 或调用 pthread_cond_init()函数。
例如：pthread_cond_t got_request=PTHREAD_COND_INITIALIZER
需要注意的是，PTHREAD_COND_INITIALIZER 是一个结构，只能在条件变量声明时对它进行初始化，运行时对条件变量初始化必须使用 pthread_cond_init()函数。
（2）向条件变量发信号
向条件变量发出信号，可以唤醒等待这个条件变量的一个线程或所有线程（广播方式），使用下面两个函数可以实现：
int pthread_cond_signal(&got_request); // got_request 是一个已经正确初始化的条件变量
int pthread_cond_broadcast(&got_request); //广播函数
成功时返回 0，失败时返回非 0 值。
（3）守候条件变量
pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex) ;
pthread_cond_timewait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime) ;
这两个函数将使调用线程阻塞在条件变量上,并解锁互斥量（为避免竞争使用条件变量，在守候条件变量前必须加锁互斥量），第二个函数中，允许用户给定一个超时间隔，过了时间间隔，函数就返回，返回值为 ETIMEDOUT。而第一个函数若没有收到信号将一直等下去。
进一步说明上述两个守候条件变量的函数：在守候条件变量前，先对互斥量加锁，然后调用守候函数，这时，线程阻塞去等候条件，并解锁互斥量，之后，当某个线程向这个条件变量发出信号便会唤醒等候的线程。
（4）销毁条件变量
int pthread_cond_destory(pthread_cond_t *cond);
成功时返回 0，失败时返回非 0 值。

三、程序框架

```
#include<pthread.h>
#include<stdio.h>
#include<errno.h>
 ......              //声明条件变量和互斥量，以及线程共享的数据
void P( )
{
    pthread_mutex_lock();
     ......
    pthread_cond_wait( );     //等待条件变量
    ......
```

```
    pthread_mutex_unlock();
    ......
}
void V()
{pthread_mutex_lock( );
   ......
 pthread_cond_signal( );
 pthread_mutex_unlock( );
}
void *create(void *arg)            //新线程执行的函数
{   int ph_no;   //哲学家编号
    ph_no=*(int *)arg;
   ......
 P(ph_no);
 P((ph_no+1)%5);
 printf("the no %d philosopher dining\n",ph_no);
 V(ph_no);
 V((ph_no+1)%5);
 pthread_exit(NULL);
}
main()
{
pthread_t p_thread[5];              //线程 ID
int   ph_no[5],ret,i;                //ph_no 为哲学家编号

    pthread_create( );    ///创建 5 个哲学家线程
    ......
 }
}
```

课题五 基于 IPC 机制实现进程间通信

一、 目标

创建服务进程，负责资源分配，创建客户进程，提出资源请求，通过进程间高级通信实现客户进程和服务进程的通信，具体如下：

1.创建多个客户进程，以及一个服务进程。

2.创建两个公共消息队列，一个用于客户进程向服务进程发送请求，一个用于服务进程向客户进程返回结果。

3. 服务进程初始化系统资源列表，客户进程发送资源请求，询问是否可以申请资源，并阻塞接收结果，服务进程接收资源请求，使用银行家算法进行计算并回答“同意”或“拒绝”，将应答返回给客户进程。

二、准备知识

所涉及的函数如下：

1．建立消息队列

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
int msqid=msgget(key_t key,int msgflg);
```

其中，key 是消息队列的关键字，是通信双方约定的一个长整型数。key 有以下两个取值：

（1）IPC_PRIVATE:表示建立一个私有消息队列。

（2）正整数：表示建立一个公共的消息队列。

Msgflg 是消息队列的创建方式，有两个取值，这两个值可以单独使用，也可用“或”位运算符连起来使用：

（1）IPC_CREAT :创建一个 key 所代表的新消息队列，类似于 create。

（2）IPC_EXCL:同 IPC_CREAT 同时使用时，若 key 对应消息队列存在，则返回-1，并将错误指示变量 errno 置 EEXIST，单独使用时无意义。

此外，还可以设置消息队列的存取权限。调用该消息队列系统调用时，若 key=0，则创建一个消息队列，若 key≠0，在消息队列中查找关键字是 key 的消息队列，如果找到，且有权访问，则返回消息队列标识，如果无权访问则出错返回，如果没有找到，且 msgflg 含有 IPC_CREAT 标志，则创建一个消息队列，并返回消息队列标识。

例如：msqid=msgget(123, 0600|IPC_CREAT|IPC_EXCL);

注意：这里 0600 表示对该消息队列的存取权限，设置的方式同文件存取权限的设置一致。0600 表示对消息队列的创建者具有读写权限，对同组用户和其他用户无任何权限。

2．向消息队列发送消息

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

其中，msqid 是消息队列标识，msgp 是指向用户区要发送的消息的指针，msgsz 是要发送的消息正文的字节数，msgflg 是同步标志，当发送消息的某个条件不满足时，若 msgflg 中的 IPC_NOWAIT 标志未设置（即=0），发送进程阻塞等待，直到将消息挂入消息队列，若

IPC_NOWAIT 标志已设置，发送进程立即返回。
该函数，成功时返回实际发送的字节数，错误时返回-1。
3. 接收消息

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

其中，msqid 是消息队列标识，msgp 是指向用户接收消息的指针，msgsz 是要接收的消息正文的字节数，msgflg 是同步标志，当接收消息的某个条件不满足时，若 msgflg 中的 IPC_NOWAIT 标志未设置（即=0），接收进程阻塞等待，直到接收到消息为止，若 IPC_NOWAIT 标志已设置，接收进程立即返回。Msgtyp 是要接收的消息类型，msgtyp=0，表示接收消息队列中的第一个消息，若 msgtyp＞0，则接收消息队列中与 msgtyp 相同的第一个消息，若 msgtyp＜0，则接收消息队列中类型值小于 msgtyp 绝对值且类型值最小的消息。
该函数成功时，返回接收消息的正文长度，错误时，返回-1。

三、程序框架

系统要求最少设置三个客户进程，一个服务进程。具体是：客户进程将请求，提交给服务进程，服务进程从消息队列 1 接收后进行服务（通过银行家算法，判断在当前资源情况下，是否可以分配资源给该客户进程），将结果发送到消息队列 2，客户进程从消息队列 2 接收属于自己的结果。

```
 //client1.c
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>

#define SDKEY 75  //SDKEY是发送请求的消息队列关键字，是一个公共消息队列
#define RVKEY 76 //RVKEY是接收结果的消息队列关键字，也是一个公共消息队列

struct msgform{     //消息的格式
long mtype;         /*消息类型 */
char mtext[N];      /*消息正文 */
}

……

//client2.c
……
//client3.c
……
//server.c
……
```

课题六　基于 IPC 和线程机制模拟实现客户机通信

一、 目标

学习 Linux 的进程间通信机制，使用消息队列通信实现两个客户机和一个服务器之间的通信。具体要求：

1.创建两个客户进程，以及一个服务进程。

2.在每个客户进程中创建一个用于接收消息的私有消息队列。并将消息队列的 id 号发送给服务器进行“注册”。

3.在服务器端创建一个公共消息队列，用于接收客户机发来的私有消息队列 id 号，服务器记录下两个客户机的消息队列 id 号后，负责将 id 号传递给对方客户机（如 A 客户机的 id 号传给 B 客户机，B 客户机的 id 号传给 A 客户机），以便实现两客户机的通信。

4.每个客户机从自己的私有队列接收服务器发来的另一客户机私有消息队列 id 号，从而可以通过对方的消息队列 id 号，向对方发送消息，客户机之间通过各自的私有消息队列接收消息。客户机设置两个线程，一个负责发送消息，一个负责接收消息。从而实现两客户机的通信。

二、准备知识

1．建立消息队列

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
int msqid=msgget(key_t key,int msgflg);
```

其中，key 是消息队列的关键字，是通信双方约定的一个长整型数。key 有以下两个取值：

（1）IPC_PRIVATE:表示建立一个私有消息队列。

（2）正整数：表示建立一个公共的消息队列。

Msgflg 是消息队列的创建方式，有两个取值，这两个值可以单独使用，也可用“或”位运算符连起来使用：

（1）IPC_CREAT :创建一个 key 所代表的新消息队列，类似于 create。

（2）IPC_EXCL:同 IPC_CREAT 同时使用时，若 key 对应消息队列存在，则返回-1，并将错误指示变量 errno 置 EEXIST，单独使用时无意义。

此外，还可以设置消息队列的存取权限。调用该消息队列系统调用时，若 key=0，则创建一个消息队列，若 key≠0，在消息队列中查找关键字是 key 的消息队列，如果找到，且有权访问，则返回消息队列标识，如果无权访问则出错返回，如果没有找到，且 msgflg 含有 IPC_CREAT 标志，则创建一个消息队列，并返回消息队列标识。

例如：msqid=msgget(123, 0600|IPC_CREAT|IPC_EXCL);

注意：这里 0600 表示对该消息队列的存取权限，设置的方式同文件存取权限的设置一致。0600 表示对消息队列的创建者具有读写权限，对同组用户和其他用户无任何权限。

2．向消息队列发送消息

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

其中，msqid 是消息队列标识，msgp 是指向用户区要发送的消息的指针，msgsz 是要发送的

消息正文的字节数，msgflg 是同步标志，当发送消息的某个条件不满足时，若 msgflg 中的 IPC_NOWAIT 标志未设置（即=0），发送进程阻塞等待，直到将消息挂入消息队列，若 IPC_NOWAIT 标志已设置，发送进程立即返回。
该函数，成功时返回实际发送的字节数，错误时返回-1。

3．接收消息

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

其中，msqid 是消息队列标识，msgp 是指向用户接收消息的指针，msgsz 是要接收的消息正文的字节数，msgflg 是同步标志，当接收消息的某个条件不满足时，若 msgflg 中的 IPC_NOWAIT 标志未设置（即=0），接收进程阻塞等待，直到接收到消息为止，若 IPC_NOWAIT 标志已设置，接收进程立即返回。Msgtyp 是要接收的消息类型，msgtyp=0，表示接收消息队列中的第一个消息，若 msgtyp＞0，则接收消息队列中与 msgtyp 相同的第一个消息，若 msgtyp＜0，则接收消息队列中类型值小于 msgtyp 绝对值且类型值最小的消息。
该函数成功时，返回接收消息的正文长度，错误时，返回-1。

4.创建线程

```
int pthread_create(    pthread_t *thread,       // 线程 id 指针
                       pthread_attr_t *attr,    //线程属性指针
                      void *(start_routine)(void *),   //返回 void 类型的指针函数
                      void *arg);                      //start_routine 的形参
```

该函数成功返回 0，失败返回非 0 值。
例：

```
/* the example file is createthread.c       */
#include<pthread.h>
#include<stdio.h>
void *create(void *arg)           //新线程执行的函数
{int i;
for (i=0;i<10;i++)
printf("%d new thread created %c\n",i,*(char *)arg);
pthread_exit(NULL);                  //退出线程
}
int main( int argc, char *argv[]) //程序从主函数开始
{
pthread_t p_thread;               //线程 ID
int    ret;
char a='A';
char b='B';

ret=pthread_create(&p_thread,NULL,create,(void*)&a);
                                  //创建新线程执行 create 函数，传递参数 a
create((void*)&b);                //主线程执行此函数，传递参数 b
}
```

编译时使用命令：gcc -lpthread -o createthread createthread.c

2. 等待子线程结束

```
#include<pthread.h>
int pthread_join(pthread_t p_thread,void **status);
```
该函数的第一个参数是线程的 ID，第二个参数为子线程返回的值，该函数返回 0 表示成功，非 0 为失败。下面给出“观察者-报告者”线程并发的例子，体会多线程共享数据可能造成的问题。

```
#include<pthread.h>
#include<stdio.h>
static int    count=10;

 void *observer(void *arg)            //观察者执行的函数
{int i;
for (i=0;i<100 ;i++)
   printf("observer %d count=%d\n", i,(*(int *)arg)++);
pthread_exit(NULL);                  //退出线程
}
void *reporter(void *arg)            //报告者执行的函数
{int i;
for (i=0;i<100;i++)
{
   printf("%d count=%d    \n",i,*(int *)arg);
   *(int*)arg=0;
}
pthread_exit(NULL);                  //退出线程
}
int main( int argc, char *argv[])   //  主函数
{
pthread_t p1_thread,p2_thread;               //线程 ID
pthread_create(&p1_thread,NULL,observer,(void*)&count); //创建新线程
pthread_create(&p2_thread,NULL,reporter,(void**)&count);
pthread_join(p1_thread,NULL);     //主线程等待子线程结束
pthread_join(p2_thread,NULL);}
```

三、程序框架

服务进程一直处于工作状态，客户进程则可以设计一定的规则完成通信后撤销。

```
//server.c
```
创建公共消息队列；
接收消息；
两个客户机的消息队列id都收到，则记录并传递；

```
//client1.c
```
创建私有消息队列；
获取公共消息队列id并发送私有队列id；
接收client2的私有队列id

创建线程，想client2发送消息，并接收自己的消息，实现两客户机的对话
判断是否结束对话（如收到了bye），收到则结束进程。

课题七 添加系统调用

一、目标

1. 自定义一个系统调用，功能是实现关于进程资源使用状况的检测。
2. 编译内核，并测试该系统调用。

二、准备知识

不同 Linux 内核版本在编译内核和添加系统调用的方法上不尽相同，请查阅相应资料，找到适合当前内核版本的添加系统调用的方法。

系统调用的机制是：将系统调用功能号保存于寄存器，之后跳转到系统调用总入口 system_call，再查找系统调用系统调用表 sys_call_table，调用内核函数后返回。

添加系统调用时，需先定义调用函数；之后在系统中登记调用号，修改系统调用表 sys_call_table；最后进行内核编译。 在新的内核中进行系统调用的测试。

三、要求

1. 添加新的系统调用，并修改相关内核代码。其中，关于进程资源使用状况检测的系统调用所对应的用户级函数原型可为：

int get_process_usage(pid_t zPID, struct zgs_rusage* zRU);

//第一个参数用来传入指定进程的进程标识符

//第二个参数用来返回指定进程的资源使用状况信息

而关于进程资源使用状况信息的数据结构类型struct zgs_rusage的定义如下：

struct zgs_rusage {

struct timeval ru_utime; // 进程在用户态的执行时间（以秒和微秒为单位）

struct timeval ru_stime; // 进程在系统态的执行时间（以秒和微秒为单位）

long ru_majflt; // 需要物理输入输出操作的页面错误次数

long ru_minflt; // 无需物理输入输出操作的页面错误次数2

long ru_nswap; // 进程置换出内存的次数

};

建议上述类型名称中的zgs替代为你自己姓名拼音的字母缩写。

需要指出的是，该系统调用是 Linux 内核自身的系统调用 getrusage()的一个变种，所以建议首先研读和分析 getrusage()及 sys_getrusage()等相关源代码，以便在此基础上形成自己的系统调用函数设计方案。

2. 完成内核编译。
3. 新的系统调用能够通过用户态的调用测试。

课题八　守护进程

一、目标

1. 设计一个计时器守护进程，定时（1 分钟）向日志文件输出当前时间。
2. 设计一个用户交互进程，可以启动、撤销守护进程，还可以实现用户定制闹钟，到指定时间提示用户。

二、准备知识

1.守护进程的创建

守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。Linux 的大多数服务器就是用守护进程实现的。创建步骤如下：

（1）在后台运行。

方法：调用 fork()创建子进程，之后父进程退出。使子进程脱离父进程，由 init 进程领养。

（2）脱离控制终端，登录会话和进程组

方法：调用 setsid()使进程成为会话组长。

（3）禁止进程重新打开控制终端

方法：再调用 fork()创建子进程，使子进程不再成为会话组长来禁止子进程重新打开控制终端，父进程退出。

（4）关闭所有文件描述符

方法：for(i=0;i< NOFILE;++i)　//关闭打开的文件描述符

```
close(i);
```

（5）改变当前工作目录

方法：chdir("/tmp") ;　//改变到“/” 或者“/tmp”

（6）将文件权限掩码设为 0

方法：umask(0);

2. 关于信号和定时器相关知识，请参考实验指导书的实验三进程通信部分。

三、要求

1. 创建守护进程和交互进程，由交互进程给出操作菜单选项，包括启动守护进程，设置闹钟时间，终止守护进程等。

2. 设计守护进程和交互进程之间的通信，可以采用消息队列或者管道，由守护进程将自己的 pid 发送给交互进程，交互进程通过 kill()发送终止信号终止守护进程。用户输入的闹钟时间由交互进程发送给守护进程，守护进程设置定时器，时间到则通知交互进程，由交互进程在终端告知用户。

课题九 模拟设计磁盘文件的链接存储结构

一、目标

1．了解磁盘文件的管理方式。

2．掌握文件物理结构和存取方式。

3．明晰文件的逻辑地址和物理地址。

二、要求

1.磁盘文件的管理采用显式链接结构，将文件占用的物理块号和链接指针记录在一张文件分配表（FAT）中。文件第一块的块号记录在索引结点中。文件目录只记录文件名和索引结点的编号。索引结点的结构如下：

索引结点编号	文 件 属性	创 建 时间	文件第一块块号	文件占用盘块数	备用

2．假定磁盘存储空间共有 100 个物理块用于存放数据， 目录文件和索引结点可直接访问，不存放在这 100 个块中。

3．一个物理块可存放 3 个文件逻辑记录，并且假设文件逻辑记录定长。

4．要求用键盘输入的方式模拟用户读写请求，菜单格式建议如下：

1 Create（filename）

2 Write（filename， text， logical_record_no）

3 Read (filename, logical_record_no)

4 Delete（filename）

其中 filename 是要读写的文件名，text 是写入的内容，logical_record_no 是逻辑记录号。Create、Write、Read、Delete 分别表示创建一个文件，向文件的某个逻辑记录写，从文件的某个逻辑记录读，删除一个文件。

5．文件存储空间管理采用位示图（位示图为 7 行，16 列）的方式。

6．要求程序可以打印 FAT 以及位示图的情况。

课题十 进程监控

一、目标

1. 掌握 ptrace 系统调用的使用。

2. 了解进程监控和调试的一般过程。

二、准备知识

1. 通过上网查找资料、查看手册了解 ptrace 系统调用及其参数。

2. 了解进程监控的一般过程，即：父进程 fork() 出子进程，子进程调用 exec() 之前，先调用 ptrace()，以 PTRACE_TRACEME 为参数表明同意被 traced，当子进程执行后会进入暂停状态，把控制权转给它的父进程(SIG_CHLD 信号)。 而父进程在 fork()之后，调用 wait() 等子进程停下来，当 wait() 返回后，父进程就可以去查看子进程的寄存器包括系统调用使用的寄存器，从而追踪子进程的运行过程。

三、要求

实现父进程对子进程所使用的所有系统调用的追踪。

课题十一　自制最简操作系统

一、目标

探索、认识和理解操作系统的引导及自启动过程，设计和实现一个简单的操作系统内核及基于二次加载的操作系统内核自启动过程，并在虚拟机平台上加以验证

二、准备知识

学习慕课课程的 1-4 节启动模块及自装入机制，查找资料研究系统的加载和软盘镜像文件的读写。

三、要求

分析、设计与实现一个简单的操作系统内核以及相应的引导加载模块，将编译生成的可执行映像写入软盘（或软盘镜像文件）中，并在虚拟机平台上启动加载运行和测试验证。

简单操作系统内核的功能设计要求包括：

（1）引导加载模块主要承担自制简单操作系统内核的加载任务，其存放在软盘（或软盘镜像文件）的引导扇区，在计算机加电启动时，率先被加载到内存空间和执行处理；

（2）引导加载模块通过调用基本输入输出系统的中断服务例程来实现“系统启动…”等屏幕信息显示，以及把自制简单操作系统内核从软盘（或软盘镜像文件）读入和加载装入到内存空间；

（3）自制简单操作系统内核主要显示系统名称、版本、研发人员及单位等版权信息，并提供命令提示符显示以及系统时间显示、系统日期显示、系统时间设置、系统日期设置等内部命令解析执行的功能；

（4）将编译生成的可执行映像写入软盘（或软盘镜像文件）中，并在虚拟机平台上启动加载运行和测试验证。

课题十二　Linux 设备驱动程序设计与实现

一、目标

探索、分析、理解并掌握 Linux 设备驱动的设计模型、实现机制和编程要旨。

二、准备知识

查找资料，学习驱动程序编写的步骤。

三、要求

本实验课题功能设计要求如下：

（1）Linux设备驱动程序的设计与实现（包括内核模块初始化/退出函数以及设备各

类操作功能函数）；
（2）相应的设备驱动测试例程的设计实现，或设备驱动测试所用的现有应用程序的相关功能关联分析说明；
（3）设备驱动的测试验证，包括设备驱动加载和卸载操作、设备基本信息显示、设备驱动运行全过程内核信息输出以及测试例程自身运行的结果截屏。