6 进程间通信

虚拟CPU和虚拟内存的引入保证了进程的一个重要特性就是隔离，一个进程在执行过程中总是认为自己占用了所有的CPU和内存，但是实际在底层，操作系统和硬件完成了很多工作才实现了隔离的特性（比如内核和时钟设备配合实现进程调度）。在多个进程之间，如果需要进行通信的话，隔离特性会造成一些通信的障碍。所以我们需要一些手段来跨越隔离，实现进程间通信（InterProcess Communication，IPC）。

6.1 管道

多进程之间一种最自然的通信方式就是依赖文件系统，一个进程打开文件并读写信息，另一个进程也可以打开文件来获取信息。显然，这种通信方式要依赖磁盘文件，效率十分低下。为了提升效率，**有名管道**（named pipe，FIFO）就被设计出来了，有名管道是文件系统中一种专门用来读写的文件，但是通过有名管道进行通信的时候实际上并没有经过磁盘，而是经过内核的管道缓冲区进行数据传递。

如果对于拥有亲缘关系的进程而言，它们之间可以使用另一种**匿名管道**。匿名管道又可以直接被称为管道，它不需要在文件系统创建单独的文件，相反它是进程在执行过程中动态创建和销毁的。

6.1.1 popen和pclose

管道一种常见的应用场景就创建一个连接到另一个进程管道，然后向管道中写入数据或者从管道中读取。`popen` 函数可以使用 `fork` 创建一个子进程，然后子进程可以执行一个shell命令。根据模式的不同，子进程可以从管道中读取数据或者是往管道中写入数据。

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

`popen` 库函数会首先使用 `fork` 创建一个子进程，并且在父子进程之间创建一根管道用于通信，然后调用 `exec` 执行command。command表示子进程要执行的shell命令。

管道在子进程的读/写端会根据模式重定向到标准输出或者标准输入。使用type表示模式：`“r”` 表示返回的文件指针连接到command的标准输出，也就是返回文件流可读；`“w”` 表示返回的文件指针连接到command进程的标准输入缓冲区，即返回文件流可写。

`pclose` 函数关闭文件流，并返回command的终止状态。

```
//print.c
#include <func.h>
int main()
{
    puts("I am print");
    return 0;
}
```

```c
//popen_r.c
#include <func.h>
int main()
{
    FILE *fp = popen("./print", "r");
    ERROR_CHECK(fp,NULL,"popen");
    char buf[128] = {0};
    fread(buf,1,sizeof(buf),fp);
    printf("read from pipe %s\n",buf);
    pclose(fp);
    return 0;
}
```

使用`popen`的`"w"`模式可以实现一个online judge的基本模型。考生将自己代码上传服务器进行编译，服务器的处理进程会使用`popen`的`"w"`模式打开子进程（使用`exec`执行考生的程序）和管道，然后将测试数据写入子进程中，最后再比对子进程的输出结果和标准答案是否一致。

```c
//add.c
#include <func.h>
int main()
{
    int i,j;
    scanf("%d%d",&i,&j);
    printf("sum = %d\n",i+j);
    return 0;
}
```

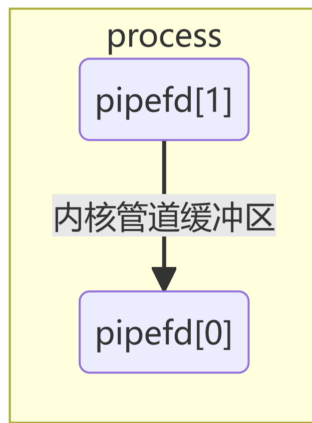
```c
//popen_w.c
#include <func.h>
int main()
{
    FILE *fp = popen("./add", "w");
    ERROR_CHECK(fp,NULL,"popen");
    fwrite("3 4",1,3,fp);
    pclose(fp);
    return 0;
}
```

6.1.2 pipe

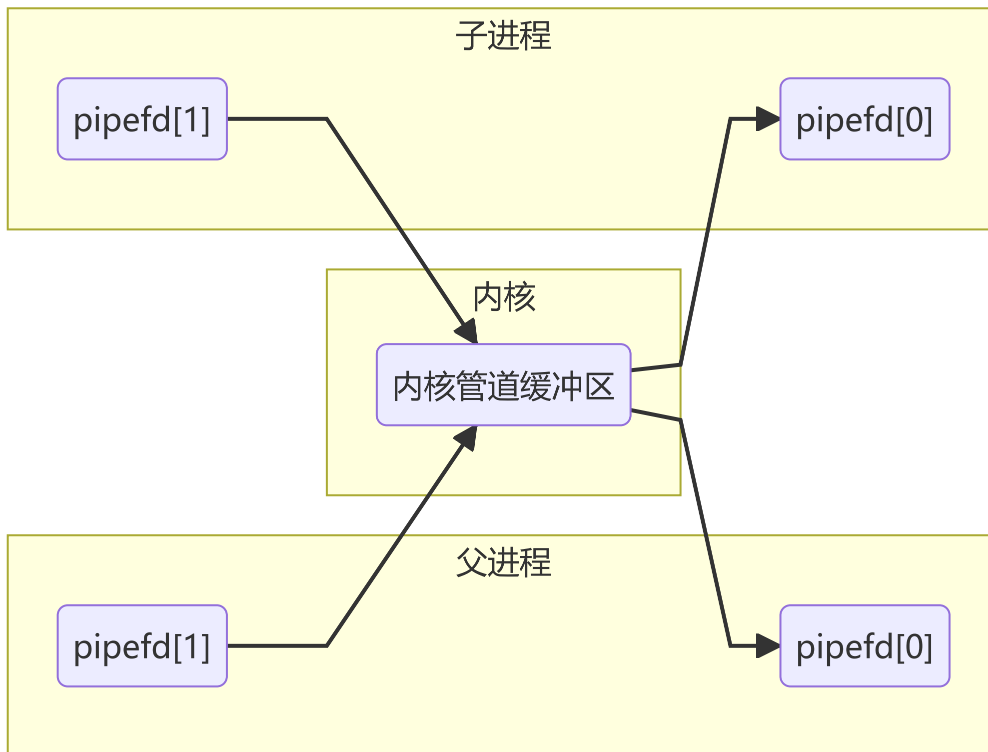
使用系统调用`pipe`可以创建匿名管道，为了支持可移植性，管道是**半双工**的，所以一般同时使用两条管道来实现全双工通信。除此以外，管道只使用于存在亲缘关系的进程之间进行通信。通常而言，在一个进程中可以创建一个管道，然后再利用`fork`就可以实现进程间通信了。

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

使用`pipe`之前，需要首先创建一个大小为2的整型数组，用于存储文件描述符。但系统调用执行完成之后，pipefd[0]是读端的文件描述符，pipefd[1]是写端的文件描述符。



显然，单个进程的管道的是没有什么价值的，所以需要再之后使用`fork`来创建一个新的子进程。内核管道缓冲区在`fork`之后是采用`dup`机制进行拷贝的，而用户态数据则是拷贝了一份副本。




```c
#include <func.h>
int main(){
    int fds[2];
  pipe(fds);//注意这里需要填写数组名
  if(!fork()){
    close(fds[1]);//关闭子进程写端
    char buf[128] = {0};
    read(fds[0],buf,sizeof(buf));
    puts(buf);
    exit(0);
  }
  else{
    close(fds[0]);//关闭父进程读端
    write(fds[1],"hello",5);
    wait(NULL);
    exit(0);
  }
```

```
}
```

如果要实现父子进程之间全双工通信，需要调用`pipe`两次来创建两条管道。一个值得关注的事实是，`fork`的次数不会影响管道的数量，每次使用`pipe`才会在内核创建一个管道缓冲区。

6.1.3 FIFO

使用匿名管道可以不需要在文件系统创建一个管道文件，进程退出时也不需要删除管道文件，但是匿名管道只能在存在亲缘关系的进程之间通信。而有名管道则刚好相反，使用`mkfifo`可以创建管道文件，使用`unlink`则可以删除所有文件包括管道文件。

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
#include <unistd.h>
int unlink(const char *path);
```

```
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int ret = mkfifo(argv[1],0600);
    ERROR_CHECK(ret,-1,"mkfifo");
    int fd = open(argv[1],O_WRONLY);
    write(fd,"hello",5);
    unlink(argv[1]);//使用unlink可以删除管道文件
    return 0;
}
```

除了删除文件以外，还可以移动/重命名文件，使用`rename`接口即可。使用`link`接口可以给文件建立硬连接。

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
//注意这里的newpath必须是文件名而不是目录名
```

6.2 共享内存

6.2.1 使用共享内存的原因

在进程的数量比较少的时候，使用管道进行通信是比较自然的思路。如果需要在任意两个进程之间使用管道，它需要调用两次`pipe`系统调用，但是随着进程数量增加，`pipe`的使用次数将会急剧增加。除此以外，使用管道通信的时候，数据要从写端拷贝到内核管道缓冲区，在从缓冲区拷贝到读端，总共要进行两次拷贝，性能比较差。为了提升进程间通信的效率，**共享内存**（也有翻译成共享存储）的方式就诞生了。

对于进程而言，代码中所使用的地址都是虚拟地址，所操作的地址空间是虚拟地址空间。当进程执行的时候，如果发生访问内存的操作，操作系统和硬件需要能保证虚拟地址能够映射到物理内存上面，和这种地址转换相关的设备被称为**内存管理单元（MMU）**。因此，如果两个不同的进程使用相同的虚拟地址，它们所对应的物理内存地址是不一样的。共享内存就允许两个或者多个进程共享一个给定的物理存储区域。当然为了实现共享，内核会专门维护一个用来存储共享内存信息的数据结构，这用不同的进程就可以通过共享内存进行通信了。

共享内存的思想实际上是非常普遍的。对于使用C语言书写的程序，使用C标准库是非常频繁的。对于每一个使用C标准库的进程，如果都去打开磁盘中的动态库文件，那么就会消耗巨大的存储资源去存储重复数据。所以使用共享内存的方法在内存中常驻这些经常使用的文件的思路是非常自然的。

```
$lsof
# 使用lsof命令可以列出所有进程所打开的文件
# 可以发现非常多进程都使用C标准库文件
```

6.2.2 system V 版本的共享内存

接下来，我们将会以system V版本的共享内存接口来介绍共享内存的使用方法。内核使用一个非负整数**键**来区分不同的共享内存区域（或者是信号量或消息队列）。服务端进程和客户端进程可以使用同一个键来定位共享内存段进行通信。键可以手动指定，也可以使用接口`ftok`生成。`ftok`需要根据一个已存在的文件和一个项目ID（0～255的整数）来生成一个键。

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

```
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    key_t key = ftok(argv[1],1);
    ERROR_CHECK(key,-1,"ftok");
    printf("key = %d\n", key);
    return 0;
}
```

共享内存、信号量和消息队列一旦创建以后，即使进程已经终止，这些IPC并不会释放在内核中的数据结构，可以用使用命令`ipcs`来查看这些IPC的信息。

```
$ipcs
# key          shmid       owner       perms       bytes       nattch      status
# 键                  描述符      拥有者      权限          占据空间     连接数       状态
$ipcs -l
# 查看各个IPC的限制
$ipcrm -m shmid
# 手动删除
```

6.2.3 创建/获取共享内存

使用`shmget`接口可以根据键来获取一个共享内存段。无论是创建新共享内存段，还是引用现存的内存段，都可以使用`shmget`函数。创建的共享内存段的所有字节会被初始化为0。key参数表示传入的键，键的取值可以是一个正数或者是宏`IPC_PRIVATE`。size表示共享内存的大小，其取值应当是页大小的整数倍。shmflg表示共享内存的属性，其最低9位表示各个用户对其的读/写/执行权限（当然IPC的执行权限事实上是无用的）。`shmget`的返回值表示共享内存段的描述符，以供后续使用。

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

```
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);//key为1000 大小为4096 创建一个0600的共享内存
    //如果希望重复创建的时候提示错误，需要或上IPC_EXCL
    ERROR_CHECK(shmid,-1,"shmget");
    return 0;
}//多次重复执行该程序的时候，不会新建新的共享内存，也不会修改大小
```

使用`shmat`接口根据一个指定描述来建立连接。shmaddr参数一般设置为空指针，表示在堆空间中自动分配区域映射共享内存段。shmflg表示权限（事实上这个权限是多余的），一般就是0。

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

```
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);//如果key为1000的共享内存已经存在，则找到它的描述符
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(char *)-1,"shmat");
    while(1);
    return 0;
}//当进程正在运行的时候，nattch为1，终止进程以后，nattch为0
```

6.2.4 使用共享内存进行进程间通信

共享内存可以在两个互不关联的进程之间进行通信，只需要彼此之间知道共享内存的键就好了。

```
//shm_w
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);//key为1000 大小为4096 创建一个0600的共享内存
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(char *)-1,"shmat");
    strcpy(p,"I love you");
    return 0;
}
```

```
//shm_r
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);//key为1000 大小为4096 创建一个0600的共享内存
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(char *)-1,"shmat");
    puts(p);
    return 0;
}
```

先后执行shm_w和shm_r程序，后一个进程可以读取共享内存中前一个进程写入的信息。

6.2.5 两个进程同时对共享内存写入

共享内存可以实现多个进程同时对同一个数据进行访问和修改，这种并发的操作往往会造成预期外的错误。

```
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);
    ERROR_CHECK(shmid,-1,"shmget");
    int *p = (int *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(int *)-1,"shmat");
    p[0] = 0;
    pid_t pid = fork();
    if(pid == 0){
        for(int i=0; i < 10000000; ++i){
            ++p[0];
        }
        exit(0);
    }
    else{
        for(int i=0; i < 10000000; ++i){
            ++p[0];
        }
        wait(NULL);
        printf("total = %d\n",p[0]);
```

```
    }
    return 0;
}
```

执行上述程序，所得到的结果和预期的20000000并不一致。进程的切换是造成结果不一致的原因：当A进程尝试对共享资源进行访问的时候，比如将p[0]从共享内存取出到寄存器当中时，此时A进程有可能被切换到B进程，而B进程会修改p[0]的数值并写回到共享内存中（修改共享内存当然不会触发写时复制），当重新切换回A进程的时候，A进程会继续刚才的指令继续运行，就会将寄存器当中p[0]的内容写回内存中，这样的话B进程刚刚的修改就丢失掉了。这个多个进程同时写入造成结果出错误的情况被称为**竞态条件**。

6.2.6 解除共享内存映射

使用`shmdt`可以解除堆空间到共享内存段的映射。具体的使用方法和`free`差不多。

```
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);//key为1000 大小为4096 创建一个0600的共享内存
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(void *)-1,"shmat");
    int ret = shmdt(p);
    ERROR_CHECK(ret,-1,"shmdt");
    while(1);
    return 0;
}
```

*6.2.7 修改共享内存属性

使用`shmctl`可以用于对共享内存段执行多种操作。根据cmd参数的不同，可以执行不同的操作：IPC_STAT可以用来获取存储共享内存段信息的数据结构；IPC_SET可以用来修改共享内存段的所有者、所在组和权限； IPC_RMID可以用来从内核删除共享内存段，当删除时，无论此时有多少进程映射到共享内存段，它都会被标记为待删除，一旦被标记以后，就无法再建立映射了。当最后一个映射解除时，共享内存段就真正被移除。

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
struct shmid_ds {
               struct ipc_perm shm_perm;    /* Ownership and permissions */
               size_t          shm_segsz;   /* Size of segment (bytes) */
               time_t          shm_atime;   /* Last attach time */
               time_t          shm_dtime;   /* Last detach time */
               time_t          shm_ctime;   /* Last change time */
               pid_t           shm_cpid;    /* PID of creator */
               pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2) */
               shmatt_t        shm_nattch;  /* No. of current attaches */
               ...
};
struct ipc_perm {
```

```
        key_t          __key;    /* Key supplied to shmget(2) */
        uid_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 + SHM_DEST and
                                    SHM_LOCKED flags */
        unsigned short __seq;    /* Sequence number */
    };
```

```
//获取共享内存段信息
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);//key为1000 大小为4096 创建一个0600的共享内存
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(char *)-1,"shmat");
    struct shmid_ds stat;
    int ret = shmctl(shmid,IPC_STAT,&stat);
    ERROR_CHECK(ret,-1,"shmctl");
    printf("cuid = %d perm = %o size= %ld nattch = %ld\n",stat.shm_perm.cuid,stat.shm_perm.mode,stat.shm_segsz,stat.shm_nattch);
    return 0;
}
```

```
//删除共享内存
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(char *)-1,"shmat");
    int ret = shmctl(shmid,IPC_RMID,NULL);
    ERROR_CHECK(ret,-1,"shmctl");
    return 0;
}
```

```
//修改共享内存段的权限
#include <func.h>
int main()
{
    int shmid = shmget(1000,4096,0600|IPC_CREAT);
    ERROR_CHECK(shmid,-1,"shmget");
    char *p = (char *)shmat(shmid,NULL,0);
    ERROR_CHECK(p,(char *)-1,"shmat");
    struct shmid_ds stat;
    int ret = shmctl(shmid,IPC_STAT,&stat);
    ERROR_CHECK(ret,-1,"shmctl");
    printf("cuid = %d perm = %o size= %ld nattch = %ld\n",stat.shm_perm.cuid,stat.shm_perm.mode,stat.shm_segsz,stat.shm_nattch);
    stat.shm_perm.mode = 0666;
```

```
    ret = shmctl(shmid,IPC_SET,&stat);
    ERROR_CHECK(ret,-1,"shmctl");
    return 0;
}
```

6.2.8 私有共享内存

使用`shmget`接口时，如果参数key的取值是宏`IPC_PRIVATE`。那么创建的共享内存段是私有的。如果利用私有共享内存进行通信，那么进程之间必须存在亲缘关系。

```
#include <func.h>
int main()
{
    int shmid = shmget(IPC_PRIVATE,4096,0600|IPC_CREAT);
    ERROR_CHECK(shmid,-1,"shmget");
    pid_t pid = fork();
    if(pid == 0){
        char *p = (char *)shmat(shmid,NULL,0);
        strcpy(p,"hello");
        shmdt(p);
        exit(0);
    }
    else{
        char *p = (char *)shmat(shmid,NULL,0);
        wait(NULL);
        puts(p);
        shmdt(p);
        //shmctl(shmid,IPC_RMID,NULL);
        //如果每次使用掉以后不删除，那么每次执行都会产生shmid不同的共享内存。
    }
    return 0;
}
```

*6.2.9 虚拟地址和物理地址转换机制

在虚拟内存机制出现之前，程序员编写程序是非常痛苦的，他们需要小心翼翼地使用物理内存，预防各种类型的异常，时刻担心会与其他程序发生冲突。引入虚拟内存机制以后，除了做内核开发或者是驱动开发的少量程序员以外，绝大多数程序员都迎来了一个非常用户友好的内存模型——虚拟内存。在虚拟内存中，地址空间是广袤无际的，地址的范围可以从0到最大值，并且可以连续地使用内存而不需要考虑底层细节。

为了实现虚拟内存机制，操作系统需要和硬件MMU配合一起实现虚拟地址到物理地址的映射。通常来说，内存管理机制是整个操作系统内核中最为复杂的部分。接下来的内容要以x86架构为实例，展示内存寻址这一简短而又重要的机制。

硬件分页机制

分页单元的主要作用就是把虚拟地址转换成物理地址，其中的一个关键任务就是检查本次内存访问是否无效。虚拟地址空间和物理内存都被分成固定长度的**页**（常见的页大小是4096字节）。在一个虚拟页中，虚拟地址是连续的，其映射的物理地址也是连续的（物理页，也叫页框）。在内核访问内存时，每次访问都是以一页为单位的。

以32位的虚拟地址、4096字节为例子，因为4096等于2的12次方，所以虚拟地址的最低12位描述了该字节在自己页内部的**偏移量**。所以分页机制只需要找到虚拟地址的高20位和物理地址的高位对应关系即可。我们可以使用一个数组来存储这种映射关系，其中数组的索引就虚拟地址的高20位，数组的元素就是物理地址的高位和一些补充位，补充位可以描述一些其他信息，比如这个物理页是否已经分配。这个数组可以被称为**页表**，页表是存储在物理内存中，索引可以被称为**虚拟页号**，物理地址的高位可以被称为**物理页号**，数组元素被称为**页表项**。

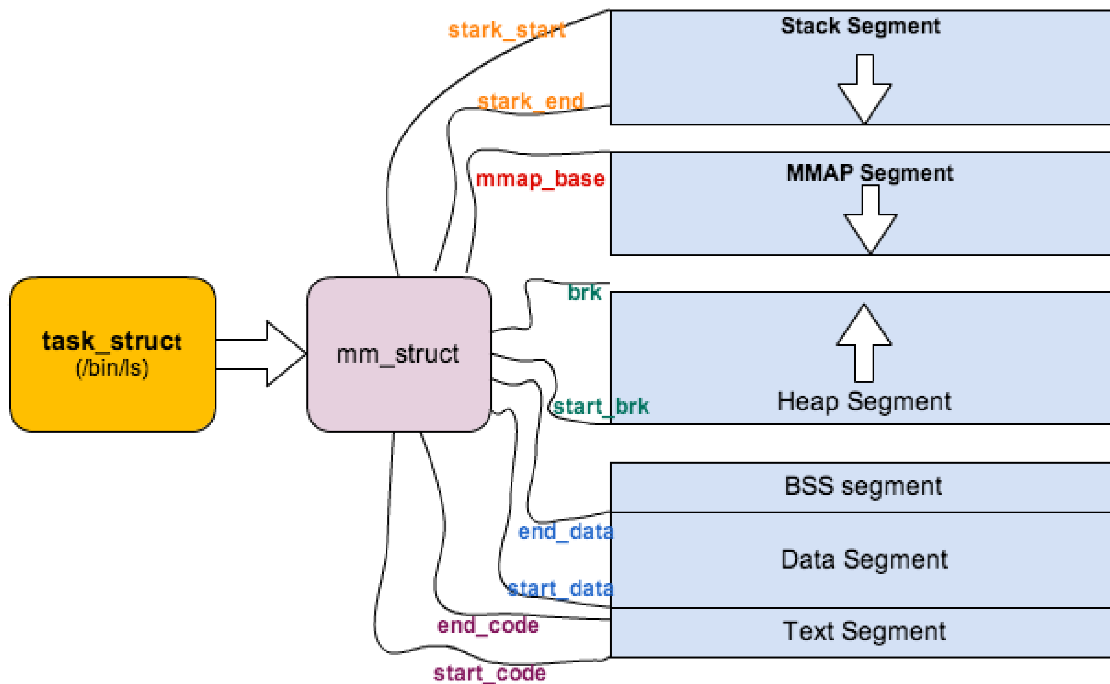
如果虚拟页号直接采用虚拟地址的高20位的话，那么页表中就要存储2的20次方（大约一百多万）个成员，要花费巨大的空间（如果页表项是4个字节，那就需要4MB）去存储页表。一种降低存储空间代价的方法就是采用**二级页表**。首先，创建一个小页表，页表中存在1024个页表项（对应虚拟地址的中间10位），如果每个页表项需要4个字节存储，那么这个页表需要占据4096字节也就是一页的空间，这样整个页表可以映射1024个页面；然后，创建一个**页目录表**，页目录表当中存储每个页表所在页的页表项，即页目录表项，共需要1024项（对应虚拟地址的高10位）。虽然如果把所有的页表都存储起来依然要消耗很大的存储空间，但是实际工作的时候，完全可以只分配需要的页表，一旦访问到未分配页表区域时，可以从页目录表中得到信息，按需分配。

- 如果64位的虚拟地址怎么办？如果单页大小依然是4096字节对应12位，剩余52位可以用来划分页目录和页表。当然，无论怎么划分，页目录表的大小都太大了。所以x86_64的解决方案是采用4级页表，每级页表用来寻址9位，合计48位，虚拟地址中的剩余16位是不参与寻址的。（即使是48位一般也足够进程使用了）
- CPU的频率比内存快太多了，解决速度差异问题的方法是引入高速缓存，高速缓存的读写速度比较快，它包括一个存储设备，里面存储了内存中的少量数据，另外还有一个控制器，用来判断高速缓存是否命中。
- 如果每次转换地址都要访问内存中的页表，速度也太缓慢了。TLB是一种缓存设备，可以用来缓存虚拟地址和物理地址的映射结果，这样如果遇到重复的虚拟页号，可以直接访问TLB。
- 物理内存的某些区域是永久地分配给内核的，用来存储内核代码和静态内核数据结构。

Linux的页管理

对于每个用户进程，它都拥有自己独立的页目录表和页表集合，这样不同进程的相同虚拟地址对应的物理地址就是不一样的。用户进程的进程控制块中`mm`和`active_mm`成员就是本进程的内存描述符，用来管理地址空间的信息。内存描述符存储了这样一些信息（不完全统计）：

- 所有已经分配的虚拟地址集合，使用链表和红黑树进行管理
- 页目录表，总共使用的页表的个数
- 代码段、数据段、堆、栈、环境变量、命令行参数的起止地址
- 总共使用的物理页数量，栈、共享内存、文件映射使用的页数量



当进程需要分配虚拟内存空间（比如使用`malloc`或者`mmap`之类的）时，首先会修改其内存描述符里面内容，但是此时并不会去实际分配物理页。当第一次访问到未分配物理的虚拟地址时，此时会触发**缺页异常**。如果虚拟地址合理有效，那么内核就会真正地分配物理页，并且给虚拟页建立和物理页的映射关系。多个连续的虚拟页所映射的物理页可能不是连续的。

再探写时复制

当父进程使用`fork`创建了一个子进程以后，操作系统需要为子进程拷贝一份父进程的页表到内存当中（基本上这是`fork`系统调用最令人烦恼的开销）。当子进程的某个可以写入的页被第一次写入的时候，内核会意识到对应的物理页是属于父进程的，此时会为子进程分配一个新的物理页，如果要分配的物理页不在主存储器中，此时会触发一个异常名为**缺页异常**，当异常处理完成以后，子进程会被分配一个新的物理页，并且物理页内容会是原来页的拷贝。这就是所谓的写时复制。

* 6.3 信号量

信号量是一种用于进程间同步的IPC。信号量本身是一个计数器，表示有多少个共享资源可以共享使用。当进程访问共享资源时，首先需要需要检查信号量的数值，如果信号量为正，那么进程就可以使用该资源，并且信号量数值减1，也就是操作系统所述的**P操作**；如果信号量的数值为0，那么就进程就休眠直到信号量大于0。当进程不再访问共享资源的时候，信号量加1，并且唤醒休眠的进程，这就是操作系统所述的**V操作**。为了在进程切换的时候也能保证信号量的正确性，信号量值的测试和减1操作是**原子操作**，也被称为**原语**，原子这里代表不可分割的含义。如果信号量的值初始为1，那么就可以被称为**二元信号量**。

我们所使用的信号量是SystemV版本的信号量。信号量一经创建以后是分配在内核区，而信号量不再是一个简单的计数器，而是多个信号量值的集合，这样就可以用一个信号量来管理比较复杂的共享情况，但是在创建时必须要指定集合的大小。`semget`就是创建信号量的接口。

当创建信号量以后，还需要使用`semctl`去初始化信号量的各个信号量值。和共享内存资源类似，当进程终止以后，也还是需要使用`semctl`主动删除信号量，否则这个信号量可以被其他进程使用。使用`semop`配合恰当的参数可以实现P操作和V操作。

信号量在操作系统的查看信息`ipcs`和手动删除`ipcrm`的操作和共享内存是一致的。

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

```c
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...);
int semop(int semid, struct sembuf *sops, size_t nsops);
struct sembuf{
           unsigned short sem_num;   /* semaphore number */
           short          sem_op;    /* semaphore operation */
           short          sem_flg;   /* operation flags */
}
//这个由用户自己声明
union semun{
  int val; //val for SETVAL
  struct semid_ds *buf; //buffer for IPC_STAT,IPC_SET
  unsigned short *arry; //Array for GETALL,SETALL
}
```

在函数`semget`当中，nsems参数表示集合中总共有多少个信号量值，semflg参数和`shmget`函数一致。

在函数`semctl`中，semid参数是信号量的标识符，就是`semget`的返回值，semnum表示某个信号量值在信号量集合中的索引（范围从0开始），cmd参数表示要执行的操作：IPC_STAT表示要获取信号量状态，IPC_SET表示要设置信号量状态，可变参数要设置为状态结构体的指针；IPC_RMID表示要删除信号量，不需要设置可变参数，注意和共享内存的删除不同，信号量是立即删除的；GETVAL表示获取置某个信号量值，SETVAL表示设置某个信号量值，可变参数传入数值；GETALL和SETALL表示获取和设置信号量集合，可变参数传入一个短整型数组或者不写。

在函数`semop`中，semid参数是信号量的标识符，sops是描述操作的结构体，sem_num成员表示信号量值的索引，sem_op表示信号量数值的变化，sem_flg表示一些其他控制信息。

6.3.1 使用信号量保护共享资源

使用信号量保护共享资源以后，并发写入共享资源的操作结果就一定是正确的了。

```c
#include <func.h>
#define N 10000000
int main()
{
    int semArrid = semget(1000,1,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    int ret = semctl(semArrid,0,SETVAL,1);
    ERROR_CHECK(ret,-1,"semctl");
    struct sembuf P;
    P.sem_num = 0;//信号量值在集合中的下标
    P.sem_op = -1;//信号量值修改
    P.sem_flg = SEM_UNDO;//如果P操作被终止，会自动恢复
    struct sembuf V;
    V.sem_num = 0;
    V.sem_op = 1;
    V.sem_flg = SEM_UNDO;
    int shmid = shmget(1000,4096,IPC_CREAT|0600);
    int *p = (int *)shmat(shmid,NULL,0);
    p[0]=0;
    if(!fork()){
        for(int i =0; i < N; ++i){
            semop(semArrid,&P,1);
            ++p[0];
```

```
            semop(semArrid,&V,1);
        }
        exit(0);
    }
    else{
        for(int i =0; i < N; ++i){
            semop(semArrid,&P,1);
            ++p[0];
            semop(semArrid,&V,1);
        }
        wait(NULL);
        printf("p[0] = %d\n", p[0]);
    }
    return 0;
}//如果程序的执行速度太慢，可以使用top检查进程执行状态
```

6.3.2 信号量的性能

从上述进程的执行情况可以得知，使用信号量的情况下，消耗的时间大大提升了。多消耗的时间自然是信号量导致的，在使用P操作的时候，如果信号量为非正数，那么进程就需要将自己的状态调整成睡眠状态，直到信号量值为正时，才会调整成就绪状态。因此，大量的时间浪费在进程之间的切换和等待上。使用函数`gettimeofday`可以较为精确地统计消耗的时间。

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
struct timeval {
               time_t      tv_sec;     /* seconds */
               suseconds_t tv_usec;    /* microseconds */
};
```

使用`gettimeofday`的时候，主要是关心第一个参数tv，它描述距离1970年1月1日经过了多少秒和微秒，tz参数描述了时区和修正信息，对我们统计时间没有什么用处。需要特别注意的是，tv_usec成员的数值总是小于10的6次方，描述了精确的微秒数值。

```
//如果使用加解锁来累加20000000次
#include <func.h>
#define N 10000000
int main()
{
    int semArrid = semget(1000,1,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    int ret = semctl(semArrid,0,SETVAL,1);
    ERROR_CHECK(ret,-1,"semctl");
    struct sembuf P;
    P.sem_num = 0;//信号量值在集合中的下标
    P.sem_op = -1;//信号量值修改
    P.sem_flg = SEM_UNDO;//如果P操作被终止，会自动恢复
    struct sembuf V;
    V.sem_num = 0;
    V.sem_op = 1;
    V.sem_flg = SEM_UNDO;
    int shmid = shmget(1000,4096,IPC_CREAT|0600);
    int *p = (int *)shmat(shmid,NULL,0);
    p[0]=0;
```

```
    struct timeval beg,end;
    gettimeofday(&beg,NULL);
    if(!fork()){
        for(int i =0; i < N; ++i){
            semop(semArrid,&P,1);
            ++p[0];
            semop(semArrid,&V,1);
        }
        exit(0);
    }
    else{
        for(int i =0; i < N; ++i){
            semop(semArrid,&P,1);
            ++p[0];
            semop(semArrid,&V,1);
        }
        wait(NULL);
        gettimeofday(&end,NULL);
        time_t s = end.tv_sec - beg.tv_sec;
        time_t us = end.tv_usec - beg.tv_usec;
        if(us < 0){
            us = us + 1000000;
            --s;
        }
        printf("total time = %ld s %ld us\n", s, us);
        printf("p[0] = %d\n", p[0]);
    }
    return 0;
}
```

```
//如果不使用加锁来累加20000000次
#include <func.h>
#define N 10000000
int main()
{
    int p[1] = {0};
    struct timeval beg,end;
    gettimeofday(&beg,NULL);
    if(!fork()){
        for(int i =0; i < 2*N; ++i){
            ++p[0];
        }
        exit(0);
    }
    else{
        wait(NULL);
        gettimeofday(&end,NULL);
        time_t s = end.tv_sec - beg.tv_sec;
        time_t us = end.tv_usec - beg.tv_usec;
        if(us < 0){
            us = us + 1000000;
            --s;
        }
        printf("total time = %ld s %ld us\n", s, us);
        printf("p[0] = %d\n", p[0]);
```

```
    }
    return 0;
}
```

在测试环境中（使用命令`$cat /proc/cpuinfo`可以查看CPU信息），在CPU为单核、主频为2.5GHz的情况下，上述两个程序运行结果相差了接近30秒左右，其中使用信号量的程序加锁解锁共合计4000000次，所以平均每次加锁、解锁所消耗的时间是0.75微秒左右。

6.3.3 semget的注意事项

使用接口`semget`创建信号量集合的时候，如果不是采用私有的方式创建信号量集合，那么多个进程传入同一个key来重复使用`semget`时，必须是不能修改信号量集合的大小的。

```
//如果执行程序之前已经存在了一个大小为1的信号量集合
//使用ipcs命令可以查看信号量集合
#include <func.h>
int main()
{
    int semArrid = semget(1000,2,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    //直接执行进程，会发现提示错误
    return 0;
}
```

使用`ipcrm`命令可以删除指定的信号量集合。

```
$ipcrm -s semid
```

6.3.4 GETVAL

在接口`semctl`中，cmd参数如果传入的是GETVAL，那么就可以用来获取信号量的值。下面是示例：

```
#include <func.h>
int main()
{
    int semArrid = semget(1000,1,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    int ret = semctl(semArrid,0,SETVAL,5);
    ERROR_CHECK(ret,-1,"semctl");
    int val = semctl(semArrid,0,GETVAL);//一定要拥有写权限
    ERROR_CHECK(val,-1,"semctl");
    printf("val = %d\n",val);
    semctl(semArrid,0,IPC_RMID);
    return 0;
}
```

6.3.5 SETALL和GETALL

除了可以单独给某个信号量值初始化，使用SETALL参数可以对信号量集合中的所有信号量值进行初始化。首先需要先创建一个元素类型为unsigned short的数组，数组中的值是各个信号量值的初始值。

```
#include <func.h>
```

```
int main()
{
    int semArrid = semget(1000,2,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    unsigned short arr[2] = {3,5};
    int ret = semctl(semArrid,0,SETALL,arr);//第二参数是无效的
    ERROR_CHECK(ret,-1,"semctl SETALL");
    //如何得知信号量集合的大小？通过semArrid获取信号量集合信息
    //arr的大小应当和信号量集合大小相等，如果arr太小会触发数组越界
    unsigned short retArr[2] = {0};
    ret = semctl(semArrid,0,GETALL,retArr);
    ERROR_CHECK(ret,-1,"semctl GETALL");
    for(int i = 0; i < 2; ++i){
        printf("retArr[%d] = %d\n", i, retArr[i]);
    }
    return 0;
}
```

6.3.6 IPC_STAT和IPC_SET

使用参数IPC_STAT可以获取信号量集合的状态，使用IPC_SET可以设置信号量集合的状态，需要特别注意的是，能修改的信息是非常有限的，主要使用拥有者、组和权限。

```
#include <func.h>
int main()
{
    int semArrid = semget(1000,2,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    struct semid_ds stat;
    semctl(semArrid,0,IPC_STAT,&stat);
    printf("cuid = %d perm = %o nsems=
%ld\n",stat.sem_perm.cuid,stat.sem_perm.mode,stat.sem_nsems);
    stat.sem_perm.mode = 0666;
    semctl(semArrid,0,IPC_SET,&stat);
    printf("cuid = %d perm = %o nsems=
%ld\n",stat.sem_perm.cuid,stat.sem_perm.mode,stat.sem_nsems);
    return 0;
}
```

6.3.7 生产者-消费者问题

生产者消费者问题是最经典的进程间通信问题之一。这个问题是这样的，存在一个仓库的格子上限为N，每个格子可以存储一个商品，消费者可以从仓库中取出商品，生产者可以放入商品，但是执行的过程中，商品数量不能为负数，也不能超过N个。

使用信号量的方法解决这个问题有两个思路。一种是采用一个二元信号量，在取出和放入商品前先根据当前商品数量来设置信号量数值，从而确定此操作是否可以进行：

```
#include <func.h>
int main()
{
    int semArrid = semget(1000,1,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    int ret = semctl(semArrid,0,SETVAL,1);
```

```
    ERROR_CHECK(ret,-1,"semctl");
    struct sembuf P;
    P.sem_num = 0;//信号量值在集合中的下标
    P.sem_op = -1;//信号量值修改
    P.sem_flg = SEM_UNDO;//如果P操作被终止，会自动恢复
    struct sembuf V;
    V.sem_num = 0;
    V.sem_op = 1;
    V.sem_flg = SEM_UNDO;
    int shmid = shmget(1000,4096,IPC_CREAT|0600);
    int * p = (int *)shmat(shmid,NULL,0);
    p[0] = 0;
    p[1] = 10;
    if(!fork()){
        //子进程是消费者
        while(1){
            sleep(2);
            printf("I am consumer, before consume, product = %d, space =
%d\n",p[0],p[1]);
            semop(semArrid,&P,1);
            if(p[0] > 0){
                --p[0];
                printf("Buy a product\n");
                ++p[1];
            }
            semop(semArrid,&V,1);
            printf("I am consumer, after consume, product = %d, space =
%d\n",p[0],p[1]);
        }
        exit(0);
    }
    else{
        //父进程是生产者
        while(1){
            sleep(1);
            printf("I am producer, before produce, product = %d, space =
%d\n",p[0],p[1]);
            semop(semArrid,&P,1);
            if(p[1] > 0){
                --p[1];
                printf("Produce a product\n");
                ++p[0];
            }
            semop(semArrid,&V,1);
            printf("I am producer, after produce, product = %d, space =
%d\n",p[0],p[1]);
        }
        wait(NULL);
    }
    return 0;
}
```

另一种方法使用采用计数信号量，将商品个数和剩余格子的个数都设为信号量，然后使用信号量集合来管理这两个信号量值。

```c
#include <func.h>
int main()
{
    int semArrid = semget(1000,2,IPC_CREAT|0600);
    ERROR_CHECK(semArrid,-1,"semget");
    unsigned short arr[2] = {0,10};
    int ret = semctl(semArrid,0,SETALL,arr);
    ERROR_CHECK(ret,-1,"semctl");
    if(!fork()){
        //子进程是消费者
        struct sembuf consume[2];
        consume[0].sem_num = 0; //0表示商品 1表示空格
        consume[0].sem_op = -1;//商品减1
        consume[0].sem_flg = SEM_UNDO;
        consume[1].sem_num = 1;
        consume[1].sem_op = 1;//空格加1
        consume[1].sem_flg = SEM_UNDO;
        while(1){
            sleep(2);
            printf("I am consumer, before consume, product = %d, space = %d\n",
                   semctl(semArrid,0,GETVAL),semctl(semArrid,1,GETVAL));
            //semop(semArrid,consume,2); 这种写法显示太快了
            semop(semArrid,&consume[0],1);
            printf("Buy a product\n");
            semop(semArrid,&consume[1],1);
            printf("I am consumer, after consume, product = %d, space = %d\n",
                   semctl(semArrid,0,GETVAL),semctl(semArrid,1,GETVAL));
        }
        exit(0);
    }
    else{
        //父进程是生产者
        struct sembuf produce[2];
        produce[0].sem_num = 0; //0表示商品 1表示空格
        produce[0].sem_op = 1;//商品加1
        produce[0].sem_flg = SEM_UNDO;
        produce[1].sem_num = 1;
        produce[1].sem_op = -1;//空格减1
        produce[1].sem_flg = SEM_UNDO;
        while(1){
            sleep(1);
            printf("I am producer, before produce, product = %d, space = %d\n",
                   semctl(semArrid,0,GETVAL),semctl(semArrid,1,GETVAL));
            //semop(semArrid,produce,2);
            semop(semArrid,&produce[1],1);
            printf("Produce a product\n");
            semop(semArrid,&produce[0],1);
            printf("I am producer, after produce, product = %d, space = %d\n",
                   semctl(semArrid,0,GETVAL),semctl(semArrid,1,GETVAL));
        }
        wait(NULL);
    }
    return 0;
}
```

使用计数信号量的优势就是能减少加锁的区域。如果将二元信号量的条件判断放在加锁区域以外，出现错误（就是格子/商品数量超出限制范围）。

6.3.7 不使用SEM_UNDO的后果

如果不使用SEM_UNDO，处于加锁状态的代码依然又可能会触发报错导致进程终止，并且信号量的数值并不会恢复，导致死锁，而使用SEM_UNDO的时候，如果进程在加锁状态下终止，那么信号量的数值会直接根据加锁次数回退，并且最小值为0。

```
$kill -9 父进程pid
```

* 6.4 消息队列

广义的消息队列指的是一种用于在多个进程（通常是跨机器的）之间的通信机制，它的设计目标是：将不同进程的功能进行解耦；维持高峰值的请求数据；支持一定的一致性，并且能够实现请求广播和流量控制。我们这里所描述的是狭义的消息队列，它是一种进程间通信的手段，只适用于在本地的多进程进行通信。区别于管道的流式结构，消息队列最显著的特点就是它维持一个**消息的链表**，并且可以使用先进先出的方式来进行取出消息。除此以外，消息队列使用的时候不需要先打开读端写端，可以直接往其中写入数据。

`msgget` 可以用来创建一个消息队列。

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

```
#include <func.h>
int main()
{
    int msgid = msgget(1000,IPC_CREAT|0600);
    ERROR_CHECK(msgid,-1,"msgget");
    return 0;
}
```

使用 `ipcs` 可以查看消息队列创建情况。

`msgsnd` 将新消息加入到队列末尾，每个消息当中应当包含一个正的长整型字段、一个非负的长度和实际数据字节数。这里使用一种变长结构体的设计，首先使用 `malloc` 申请一个足够大的堆空间来存储结构体，如果需要访问成员，就通过指针偏移，然后修改基类型进行间接访问。

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

msgp参数总是指向一个消息数据，它的长度应当小于系统规定的上限，第一个成员必须是长整型表示消息类型。这个结构体类型必须要重新声明：

```
struct mymesg{
    long mtype;
  char mtext[1];
};
```

msgsz参数表示的是消息中mtext数组的大小。参数flag的值可以设置为IPC_NOWAIT表示非阻塞方式发送数据。

```
#include <func.h>
struct mymesg{
    long mtype;
    char mtext[64];
};
int main()
{
    int msgid = msgget(1000,IPC_CREAT|0600);
    ERROR_CHECK(msgid,-1,"msgget");
    struct mymesg buf;
    buf.mtype = 1;
    strcpy(buf.mtext,"hello");
    int ret = msgsnd(msgid,&buf,5,0);
    ERROR_CHECK(ret,-1,"msgsnd");
    return 0;
}//使用ipcs命令可以查看消息队列的情况
```

`msgrcv` 可以用来读取消息，参数msgtyp可以用指定要读取的数据的类型，类型匹配的消息会按先进先出的方式取出，如果该参数为0，则会无视类型取出第一个消息。

```
#include <func.h>
struct mymesg{
    long mtype;
    char mtext[64];
};
int main()
{
    int msgid = msgget(1000,IPC_CREAT|0600);
    ERROR_CHECK(msgid,-1,"msgget");
    struct mymesg buf;
    memset(&buf,0,sizeof(buf));
    int ret = msgrcv(msgid,&buf,sizeof(buf.mtext),0,0);
    ERROR_CHECK(ret,-1,"msgrcv");
    printf("type = %ld, msg = %s\n",buf.mtype, buf.mtext);
    return 0;
}
```

`msgctl` 可以用来删除消息队列。

```
#include <func.h>
int main()
{
    int msgid = msgget(1000,IPC_CREAT|0600);
    ERROR_CHECK(msgid,-1,"msgget");
    msgctl(msgid,IPC_RMID,NULL);//删除是即时的
    return 0;
}
//在shell中使用ipcrm -q可以实现一样的效果
```

6.4.1 proc目录

Linux系统上的/proc目录是一种文件系统，即proc文件系统。与其它常见的文件系统不同的是，/proc是一种伪文件系统（也即虚拟文件系统），存储的是当前内核运行状态的一系列特殊文件，用户可以通过这些文件查看有关系统硬件及当前正在运行进程的信息，甚至可以通过更改其中某些文件来改变内核的运行状态。/proc中的文件常被称作虚拟文件，并具有一些独特的特点。例如，其中有些文件虽然使用查看命令查看时会返回大量信息，但文件本身的大小却会显示为0字节。

为了查看及使用上的方便，这些文件通常会按照相关性进行分类存储于不同的目录甚至子目录中，如/proc/scsi目录中存储的就是当前系统上所有SCSI设备的相关信息，/proc/N中存储的则是系统当前正在运行的进程的相关信息，其中N为正在运行的进程（可以想象得到，在某进程结束后其相关目录则会消失）。

```
$cat /proc/sys/kernel/shm*
$cat /proc/sys/kernel/sem*
$cat /proc/sys/kernel/msg*
#这里存储了IPC的限制信息
```