第4章 System V IPC进程通信

实验目的

- 理解System V IPC通信机制工作原理
- 掌握和使用共享主存实现进程通信
- 掌握和使用消息队列实现进程通信
- 掌握和使用信号量实现进程同步

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

System V IPC概述

- **IPC资源**
  - 表示单独的消息队列、共享内存或是信号量集合
- **具有相同类型的接口函数**
  - 信号量实现与其他进程同步
  - 消息队列以异步方式为通信频繁、但数据量少的进程通信提供服务
  - 共享主存为数据量大的进程间通信提供服务
- **共同点**
  - 通过 System V IPC 对象通信时，需传递该对象的唯一IPC标识符
  - 访问System V IPC 对象时必须经过许可检验
  - System V IPC对象访问权限的设置由对象的创建者实现
  - IPC通信机制把IPC对象的IPC标识符作为对系统资源表的索引

System V IPC共有操作

- 操作函数（XXX代表msg、sem、shm三者之一）
  - XXXget(): 获得IPC标识符
  - XXXctl(): 控制IPC资源
- 操作模式
  - 先通过XXXget()创建一个IPC资源，返回值是该IPC资源ID
  - 随后操作均以IPC资源ID为参数以对相应的IPC资源进行操作
  - 其他进程可通过XXXget() 取得已有的IPC资源ID（权限允许的话），并对其操作

IPC机制的公共数据结构

❖ipc_ids结构[linux/ipc/util.h]

- 每一类IPC资源都有一个ipc_ids结构的全局变量，用于描述同一类资源的公有数据
- 三个全局变量分别是semid_ds，msgid_ds和shmid_ds

```
struct ipc_ids {
 int size;       /* entries数 组 的 大 小 */
 int in_use;     /* entries数 组 已 使 用 的 元 素 个 数 */
 int max_id;
 unsigned short seq;
 unsigned short seq_max;
 truct semaphore sem;  /*控 制 对 ipc_ids结 构 的 访 问 */
 spinlock_t ary;/*自 旋 锁 控 制 对 数 组 entries的 访 问 */
 struct ipc_id* entries;
};

struct ipc_id {
    struct kern_ipc_perm* p;
};
```

IPC机制的公共数据结构

❖ kern_ipc_perm结构[linux/ipc.h]

- 表示每一个IPC资源的属性，用来控制操作权限

```
struct kern_ipc_perm {
__kernel_key_t key;      /*IPC键 */
__kernel_uid_t uid;      /*资 源 所 有 者 的 UID*/
__kernel_gid_t gid;      /*资 源 所 有 者 的 GID*/
__kernel_uid_t cuid;     /*创 建 这 个 资 源 的 进 程 UID */
__kernel_gid_t cgid;     /*创 建 这 个 资 源 的 进 程 GID*/
__kernel_mode_t mode; /*文 件 系 统 类 型 的 权 限 */
unsigned short seq;      /*位 置 使 用 序 号 */
};
```

- cuid和cgid成员设置为创建者进程的有效用户ID和有效组ID，合称为创建者ID
- uid和gid成员设置为拥有者进程的有效用户ID和有效组ID，合称为属主ID
- key：为整型，由用户提供，用于申请一个IPC标识符
- mode：指该资源的所有者、组以及其他用户对资源的读、写访问权限
- Seq：表示位置使用序列号，在计算IPC标识符时使用

IPC对象标识符与IPC键

- **IPC键**
  - 是IPC对象的外部表示，可由程序员选择
  - 如果键是公用的，则系统中所有进程通过权限检查后，均可找到和访问相应IPC对象
  - 如果键是私有的，则键值为0
  - 每个进程都可建立一个键值为0的私有IPC对象
- **IPC标识符**
  - 由内核分配给IPC对象，在系统内部是唯一的
  - IPC对象标识符的获取：**XXXget()**
    - ✓ 将IPC键传递给以sys_打头的内核函数，并为用户分配一个与IPC对象相对应的数据结构。
    - ✓ 返回一个32位的IPC标识符，进程使用此标识符对这个资源进行访问。

IPC键的创建

- 创建函数
  - key_t ftok(char * filename, int id);
- 功能说明
  - 将一个已存在的文件名(该文件必须是存在而且可以访问的)和一个整数标识符id转换成一个key_t值
  - 在Linux系统实现中，调用该函数时，系统将文件的索引节点号取出，并在前面加上子序号，从而得到key_t的返回值

```
#define IPCKEY 0x111
char path[256];
sprintf( path, "%s/etc/config.ini", (char*)getenv("HOME") );
msgid=ftok( path, IPCKEY );
```

IPC对象获取的内核函数

- 函数原型
  - nt sys_semget(key_t key,int nsems,int oflag);
  - int sys_msgget(key_t key,int oflag);
  - int sys_shmget(key_t key,int size,int oflag);
- 参数说明
  - key: 可由ftok()函数产生或定义为IPC_PRIVATE常量
  - oflag: 包括读写权限，还可以包含IPC_CREATE和IPC_EXCL标志位，组合效果如下

oflag标志	IPC对象不存在	IPC对象已存在
无特殊标志	出错，errno=ENOENT	成功，引用已存在对象
IPC_CREAT	成功，创建新对象	成功，引用已存在对象
IPC_CREAT|IPC_EXCL	成功，创建新对象	出错，errno=EEXIST

IPC对象权限位的设定

- 八进制表示格式
  - 0400: 由用户(属主)读
  - 0200: 由用户(属主)写
  - 0040: 由(属)组成员读
  - 0020: 由(属)组成员写
  - 0004: 由其他用户读
  - 0002: 由其他用户写

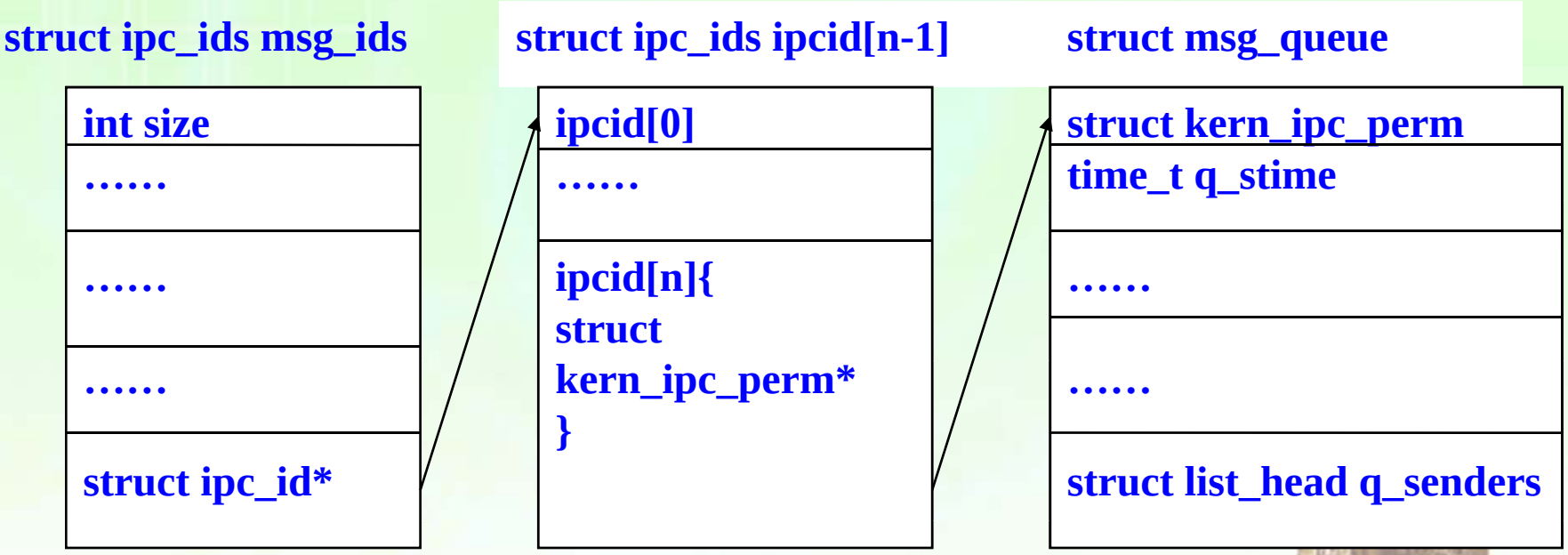
主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

消息队列

- 是一个格式化的可变长信息单位
- 消息机制允许一个进程向任何其他进程发送一个消息
- 当一个进程收到多个消息时，可将它们排成一个消息队列

消息队列全局数据结构

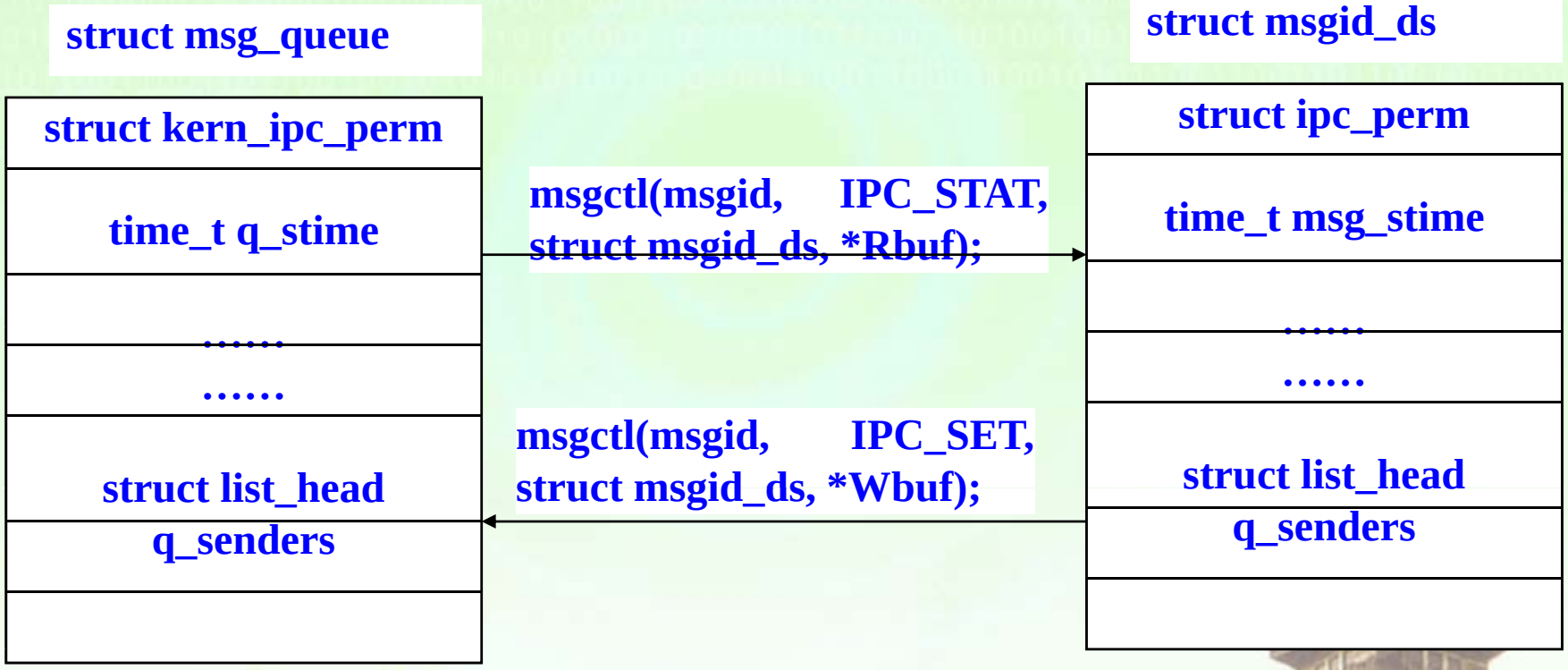

struct ipc_ids msg_ids
int size
……
……
……
struct ipc_id*
struct ipc_ids ipcid[n-1]
ipcid[0]
……
ipcid[n]{
struct
kern_ipc_perm*
}
struct msg_queue
struct kern_ipc_perm
time_t q_stime
……
……
struct list_head q_senders

msg_queue结构定义

- 定义位置：**linux/ipc/msg.c**
- 若**IPC_NOWAIT**未被设置
  - 消息队列容量已满时，发送进程进入睡眠状态并添加到相应的**q_senders**队列
  - 当消息队列中无合适消息时，接收进程进入睡眠状态并添加到相应的**q_receivers**队列

```
struct msg_queue {
 struct kern_ipc_perm q_perm;
 time_t q_stime;  /*最近一次msgsnd时间*/
 time_t q_rtime;  /*最近一次msgrcv 时间*/
 time_t q_ctime;  /* 最近的改变时间 */
 unsigned long q_cbytes; /* 队列中的字节数 */
 unsigned long q_qnum;  /* 队列中的消息数目 */
 unsigned long q_qbytes;/*队列中允许的最大字节数 */
 pid_t q_lspid;/*最近一次msgsnd()发送进程的pid */
 pid_t q_lrpid;/*最近一次msgrcv()接收进程的pid */
 struct list_head q_messages;   /*消息队列*/
 struct list_head q_receivers;/*待接收消息的睡眠进程队列*/
  struct list_head q_senders;  /*待发送消息的睡眠进程队列*/
};
```

Linux消息队列链表结构

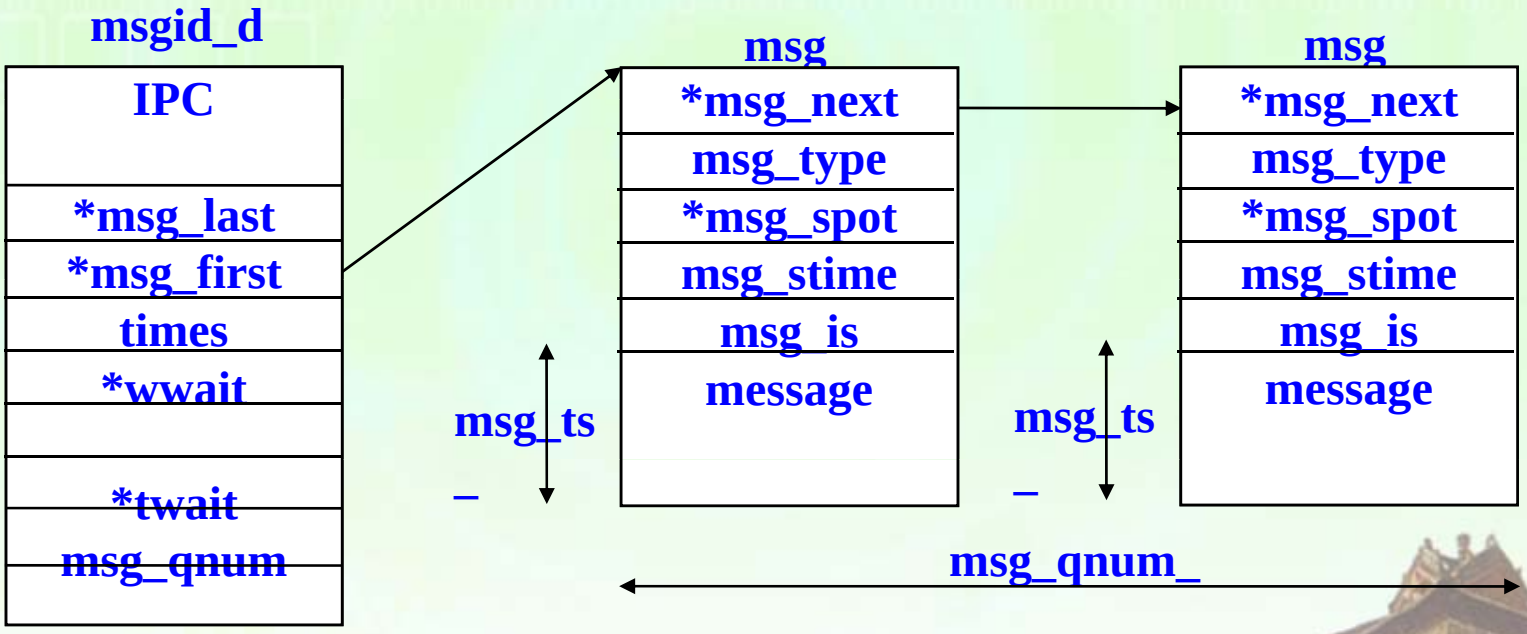


msgqid_ds结构

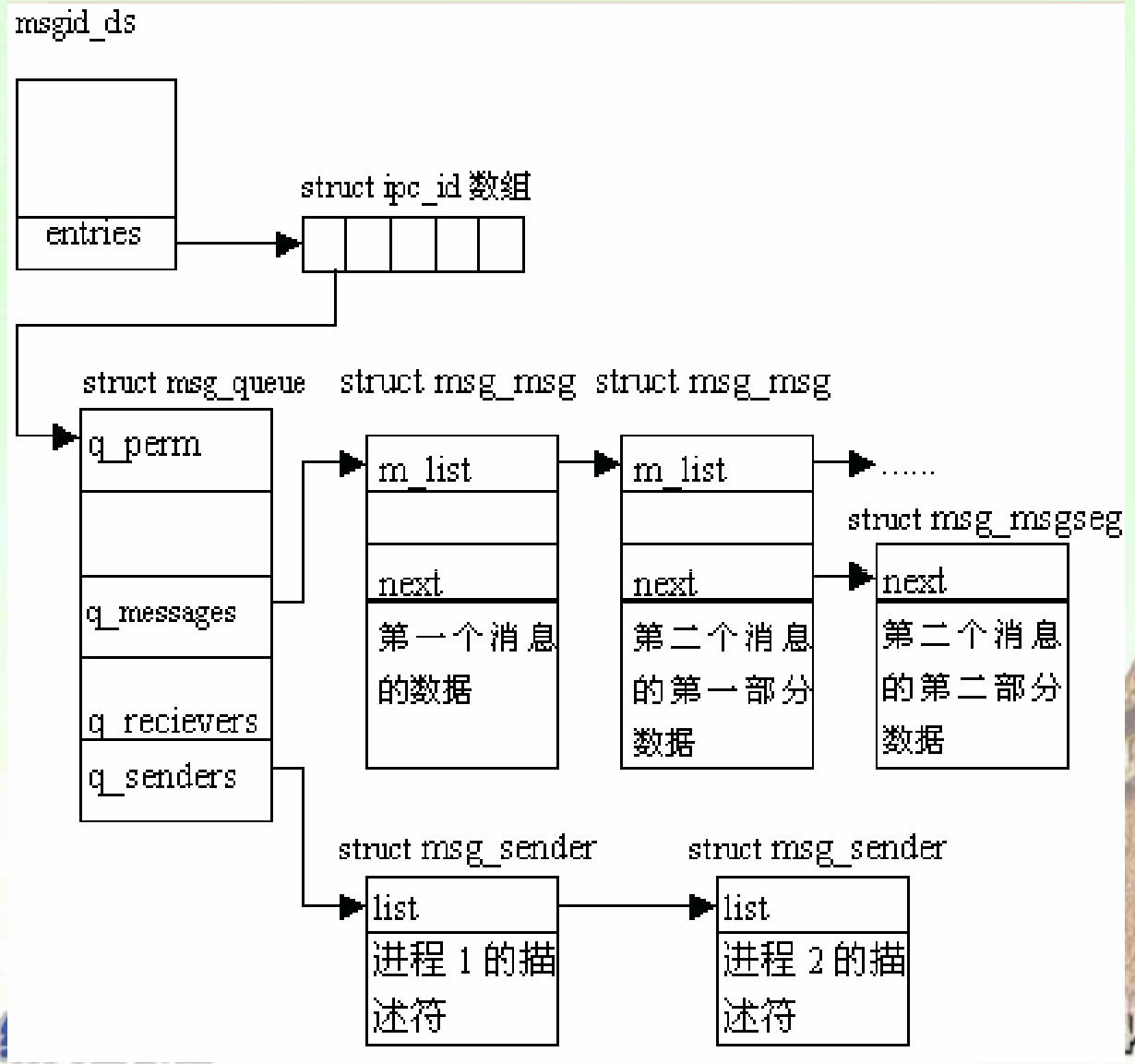
- 一旦一个新的消息队列被创建，在系统主存中会为其分配一个新的**msqid_ds**数据结构，并把它插入到数组中

```
struct  msgqid_ds {
    struct  ipc_perm  msg_perm;   /*许可权结构*/
    short   padl[7];      /*由系统使用*/
    ushort msg_qnum;      /*队列上消息数*/
    ushort msg_qbytes;    /*队列上最大字节数*/
    ushort msg_lspid;     /*最后发送消息的PID*/
    ushort msg_lrpid;     /*最后接收消息的PID*/
    time_t msg_stime;     /*最后发送消息的时间*/
    time_t msg_rtime;     /*最后接收消息的时间*/
    time_t msg_ctime;     /*最后更改时间*/
  };
```

消息队列结构


msgid_d
IPC
*msg_last
*msg_first
times
*wwait
*twait
msg_qnum
msg
*msg_next
msg_type
*msg_spot
msg_stime
msg_is
message
msg_ts_
msg
*msg_next
msg_type
*msg_spot
msg_stime
msg_is
message
msg_ts_
msg_qnum_

消息队列各数据结构之间的关系



消息队列的基本操作—msgget()

- 功能
  - 创建一个新消息队列或打开一个存在的队列
- 函数原型
  - **int msgget(key_t key, int flag);**
- 参数说明
  - **key:** 是创建/打开队列的键值
  - **msgflg:** 创建/打开方式，常取**msgflg=IPC_CREAT|IPC_EXCL|0666**
    - ✓ 若不存在key值的队列，则创建；否则，如果存在则打开队列
    - ✓ 0666表示与一般文件权限一样
- 返回值
  - 成功返回消息队列描述字，否则返回-1
- 说明
  - **IPC_CREAT**一般由服务器程序创建消息队列时使用
  - 如果是客户程序，必须打开现有的消息队列，不使用**IPC_CREAT**

消息队列的基本操作—msgrcv()

- 功能
  - 消息队列中读取一个消息
- 函数原型
  - ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t size, long type, int flag);
- 参数说明
  - msgid: 消息队列描述字
  - msgp: /消息存储位置
  - size: 消息内容的长度
  - type: 请求读取的消息类型
  - flag: 规定队列无消息时内核应做的操作
    - ✓ IPC_NOWAIT：如果没有满足条件的消息，立即返回，此时，errno=ENOMSG
    - ✓ IPC_EXCEPT：type>0配合使用，返回队列中第一个类型不为type的消息
    - ✓ IPC_NOERROR：如果队列中满足条件的消息内容大于所请求的size字节，则把该消息截断，截断部分将丢失

消息队列的基本操作—msgrcv()的工作流程

- 检查消息队列描述符和许可权
  - 若合法，继续执行
  - 否则返回
- 根据type的不同分成三种情况处理
  - type=0：接收该队列的第一个消息，并将它返回给调用者
  - type>0：接收类型type的第一个消息
  - type<0：接收小于等于type绝对值的最低类型的第一个消息
- 当所返回消息大小等于或小于用户的请求时，内核便将消息正文拷贝到用户区，并从消息队列中删除此消息，然后唤醒睡眠的发送进程
- 如果消息长度比用户要求的大时，则返回出错

消息队列的基本操作—msgsnd()

- 功能
  - 向消息队列发送一个消息
- 函数原型
  - int msgsnd(int msqid, struct msgbuf *msgp, size_t size, int flag);
- 参数说明
  - 与msgrcv()类似
- 说明
  - msgflg有意义的标志为IPC_NOWAIT，指明在消息队列没有足够空间容纳要发送的消息时，msgsnd是否等待
- 内核须对msgsnd( )函数完成的工作
  - 检查消息队列描述符、许可权及消息长度
    - ✓ 若合法，继续执行
    - ✓ 否则，返回
  - 内核为消息分配消息数据区，将消息正文拷贝到消息数据区
  - 分配消息首部，并将它链入消息队列的末尾
  - 修改消息队列头中的数据，如队列中的消息数、字节总数等
  - 唤醒等待消息的进程

消息队列的基本操作—msgctl()

- 功能
  - 修改消息队列状态信息，如查询消息队列描述符、修改它的许可权及删除该队列等。
- 函数原型
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
- 参数说明
  - cmd：是规定的命令
    - ✓ IPC_STAT：查询有关消息队列情况的命令
    - ✓ IPC_SET：按buf指向的结构中的值，设置与此队列相关的结构中的下列4个字段：msg_perm.uid、msg_perm.gid、msg_perm、mode和msg_qbytes。此命令只能由下列两种进程执行

      1）有效用户ID等于msg_perm.cuid或msg_perm.uid

      2）具有超级用户特权的进程
    - ✓ IPC_RMID：从系统中删除该消息队列以及仍在该队列上的所有数据。这种删除立即生效。可使用该此命令进程同IPC_SET。

消息队列使用示例

❖ **msgtest.c**

```
struct msgsbuf{
   int mtype;
   char mtext[1];
}msg_sbuf;
struct msgmbuf {
   int mtype;
   char mtext[10];
}msg_rbuf;

main( ){
   int gflags,sflags,rflags;
   key_t key;
   int msgid;
   int reval;

   struct msqid_ds msg_ginfo,msg_sinfo;
```

消息队列使用示例

❖ **msgtest.c**

```
/*创建一个消息队列,输出消息队列缺省属性*/
char* msgpath="/UNIX/msgqueue";
key=ftok(msgpath,'a');
gflags=IPC_CREAT|IPC_EXCL;
msgid=msgget(key,gflags|00666);
if(msgid==-1) {
   printf("msg create error\n");
   return;
}
msg_stat(msgid,msg_ginfo);
sflags=IPC_NOWAIT;
msg_sbuf.mtype=10;
msg_sbuf.mtext[0]='a';
```

消息队列使用示例

❖ msgtest.c

```
/*发送一个消息,输出消息队列属性*/
reval=msgsnd(msgid,&msg_sbuf,sizeof(msg_sbuf.mtext),sflags);
if(reval==-1) {
   printf("message send error\n");
}
msg_stat(msgid,msg_ginfo);
rflags=IPC_NOWAIT|MSG_NOERROR;
reval=msgrcv(msgid,&msg_rbuf,4,10,rflags);
if(reval==-1)
   printf("read msg error\n");
else
   printf("read from msg queue %d bytes\n",reval);
```

消息队列使用示例

❖ **msgtest.c**

```
/*从消息队列中读出消息后，输出消息队列属性*/
msg_stat(msgid,msg_ginfo);
msg_sinfo.msg_perm.uid=8;
msg_sinfo.msg_perm.gid=8;
msg_sinfo.msg_qbytes=16388;
/*此处验证超级用户可以更改消息队列的缺省msg_qbytes*/
/*注意这里设置的值大于缺省值*/
reval=msgctl(msgid,IPC_SET,&msg_sinfo);
if(reval==-1) {
   printf("msg set info error\n");
   return;
}
msg_stat(msgid,msg_ginfo);
/*验证设置消息队列属性*/
reval=msgctl(msgid,IPC_RMID,NULL);/*删除消息队列*/
if(reval==-1) {
   printf("unlink msg queue error\n");
   return;
}
```

消息队列使用示例

❖msgtest.c

```
void msg_stat(int msgid,struct msqid_ds msg_info) {
  int reval;
  sleep(1);/*只是为了后面输出时间的方便*/
  reval=msgctl(msgid,IPC_STAT,&msg_info);
  if(reval==-1) {
    printf("get msg info error\n");
    return;
  }
  printf("\n");
  printf("current number of bytes on queue is %d\n",msg_info.msg_cbytes);
  printf("number of messages in queue is %d\n",msg_info.msg_qnum);
  printf("max number of bytes on queue is %d\n",msg_info.msg_qbytes);
  /*每个消息队列的容量（字节数）都有限制MSGMNB，值的大小因系统而异*/
  /*在创建新的消息队列时，msg_qbytes的缺省值就是MSGMNB*/
  printf("pid of last msgsnd is %d\n",msg_info.msg_lspid);
  printf("pid of last msgrcv is %d\n",msg_info.msg_lrpid);
  printf("last msgsnd time is %s", ctime(&(msg_info.msg_stime)));
  printf("last msgrcv time is %s", ctime(&(msg_info.msg_rtime)));
  printf("last change time is %s", ctime(&(msg_info.msg_ctime)));
  printf("msg uid is %d\n",msg_info.msg_perm.uid);
  printf("msg gid is %d\n",msg_info.msg_perm.gid);
}
```

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

信号量与信号量集

❖信号量

- 信号量是具有整数值的对象，表示可用资源的数量
  - ✓申请资源时减1
  - ✓资源不足时进程可以睡眠等待、也可以立即返回
- 进程间可以利用信号量实现同步和互斥
- 信号量类型
  - ✓二值信号量：信号量的值只能取0或1
  - ✓计算信号量：信号量的值可以取任意非负整型值

❖信号量集

- 信号量的集合
- 用于多种共享资源进程 间的同步

struct ipc_ids结构定义

❖任务：初始化全局变量**semid_ds**

```
struct ipc_ids {
 int size;      /* entries数 组 的 大 小 */
 int in_use;    /* entries数 组 已 使 用 的 元 素 个 数 */
 int max_id;
 unsigned short seq;
 unsigned short seq_max;
 truct semaphore sem;  /*控 制 对 ipc_ids结 构 的 访 问 */
 spinlock_t ary;/*自 旋 锁 控 制 对 数 组 entries的 访 问 */
 struct ipc_id* entries;
};
```

信号量与信号量集结构定义

- 定义位置
  - include/linux/sem.h
- 信号量结构定义

```
struct sem {
    int semval;/*信 号 量 的 当 前 值 */
    int sempid;/*最 近 对 信 号 量 操 作 进 程 的 pid */
};
```

- 信号量集合结构定义

```
struct sem_array {
  struct kern_ipc_perm  sem_perm; /*IPC许 可 结 构 */
  time_t sem_otime; /*最 近 一 次 操 作 时 间 */
  time_t sem_ctime; /*最 近 一 次 的 改 变 时 间 */
  struct sem     *sem_base; /*指 向 第 一 个 信 号 量 */
  struct sem_queue *sem_pending;/*挂 起 操 作 队 列 */
  struct sem_queue  **sem_pending_last; /*上 一 次 挂 起 的 操 作 */
  struct sem_undo      *undo;/*集 合 上 的 undo请 求 */
  unsigned long   sem_nsems;/*信 号 量 的 个 数 */
};
```

sem_queue结构定义

- 功能
  - 描述每个睡眠进程
  - 每个进程的task_struct都维护一个sem_queue链表
    - ✓ struct sem_queue *semsleeping;

```
struct sem_queue {
   struct sem_queue *next;  /*队 列 中 的 下 一 个 元 素 */
   struct sem_queue **prev; /*队 列 中 的 前 一 个 元 素 */
   struct task_struct *sleeper;/*睡 眠 进 程 的 描 述 符 */
   struct sem_undo * undo;
   int    pid;      /*睡 眠 进 程 的 pid */
   int    status;    /*操 作 的 完 成 状 态 */
   struct sem_array *sma; /*所 属 的 信 号 量 集 合 */
   struct sembuf *sops; /*挂 起 的 操 作 集 合 */
   int    nsops;    /*挂 起 的 操 作 个 数 */
   ......
};
```

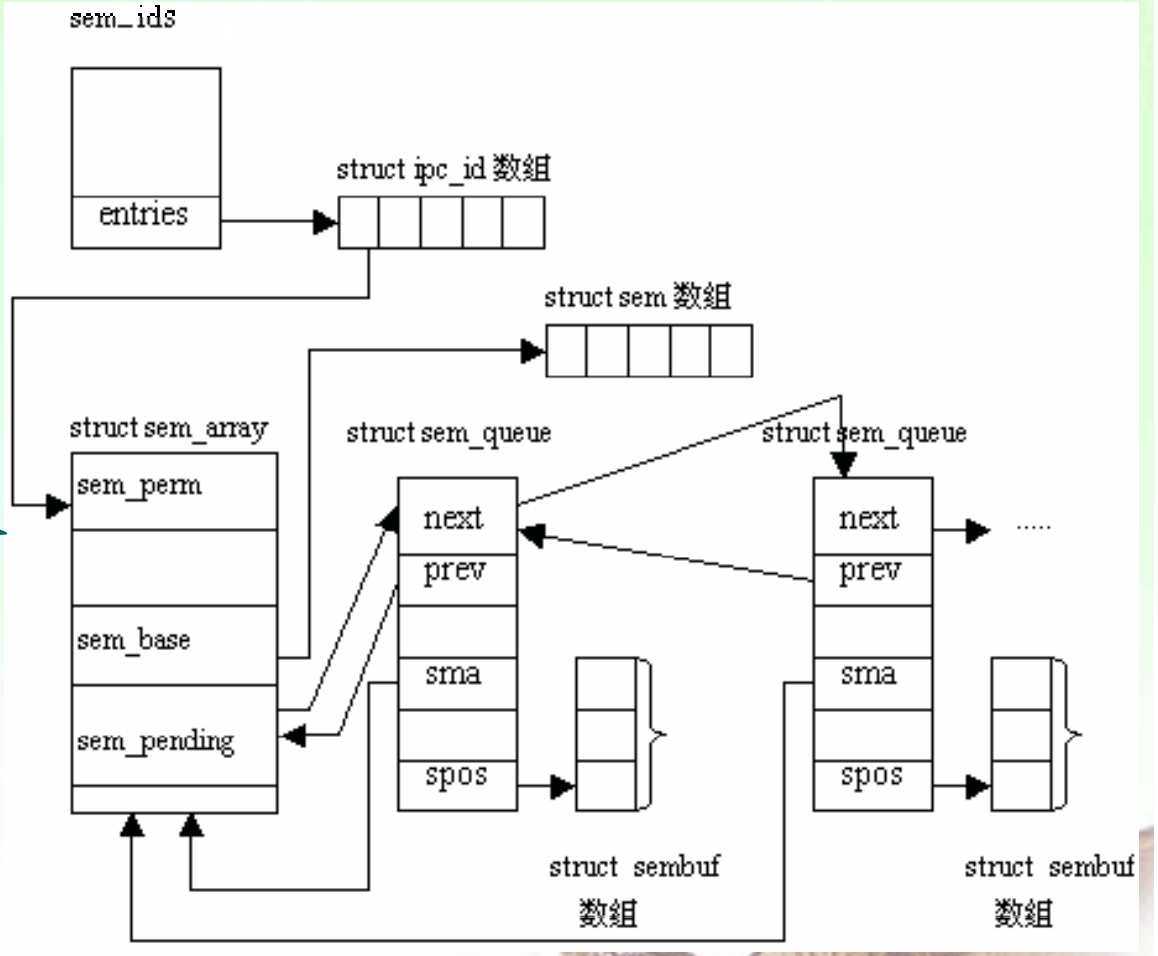
sem_undo结构定义

- 功能
  - 每个任务的task_struct都有一个undo请求链表
    - ✓ struct sem_undo *semundo;
    - ✓ 表示进程占用，资源未还
  - 当进程退出时，它们被自动执行

```
struct sem_undo {
    /*该进程的下一个条目，链入task_struct中的队列*/
    struct sem_undo *   proc_next;
    /*信号集的下一个条目，链入sem_array中的队列*/
    struct sem_undo *   id_next;
    int             semid;      /*信号量集ID*/
    /*每信号量一个调整数组*/
    short *         semadj;
};
```

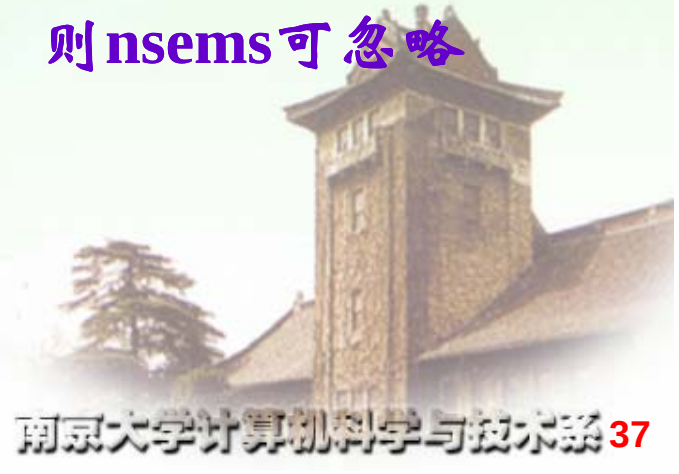
信号量各数据结构之间的关系

- **semid_ds**
  - 数据类型
    - ✓ Struct ipc_ids
  - 内核全局数据结构
  - 重要成员
    - ✓ entries
- **struct sem_array**
  - 表示每个信号量集合
  - 重要成员
    - ✓ sem_base
- **struct sem_queue**
  - 描述每个睡眠进程



信号量的基本操作—semget()

- 功能
  - 创建一个新信号量或打开一个存在的信号量
- 函数原型
  - **int semget(key_t key, int nsems, int flag)**
- 参数说明
  - **nsems**
    - ✓ 打开/创建的信号量集包含的信号量数
    - ✓ 最大值定义：**#define SEMMSL 250[Linux/sem.h]**
    - ✓ 若显式打开一个现有信号量集合，则nsems可忽略
- 返回值
  - 调用成功返回信号量描述字
  - 失败时返回-1



sys_semget系统调用

```
asmlinkage long sys_semget (key_t key, int nsems, int semflg){
    int id, err = -EINVAL;
    struct sem_array *sma;
    /*判 断 信 号 量 是 否 越 界 */
    if (nsems < 0 || nsems > sc_semmsl)
        return -EINVAL;
    down(&sem_ids.sem);
    if (key == IPC_PRIVATE) {/*创 建 新 的 信 号 量 集 */
        err = newary(key, nsems, semflg);
    } else if ((id = ipc_findkey(&sem_ids, key)) == -1) {
        if (!(semflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = newary(key, nsems, semflg);
    } else if (semflg & IPC_CREAT && semflg & IPC_EXCL) {
        err = -EEXIST;
    } else { /*信 号 量 存 在 */
        /*通 过 ID在 sem_id中 找 到 信 号 量 集 */
        sma = sem_lock(id);
        if(sma==NULL)
            BUG();
        if (nsems > sma->sem_nsems)
            err = -EINVAL;
        else if (ipcperms(&sma->sem_perm, semflg))
            err = -EACCES;
        else /*由 ipc_id生 成 信 号 集 的 id*/
            err = sem_buildid(id, sma->sem_perm.seq);
        sem_unlock(id);
    }
    up(&sem_ids.sem);
    return err;
} ? end sys_semget ?
```

newary系统调用

- 功能
  - 分配新的信号量集的大小
    - ✓ **ipc_alloc()**
  - 将信号量集中的成员**sem_perm**加到全局结构**sem_ids**中
    - ✓ **ipc_addid()**

信号量的基本操作—semop()

- 功能
  - 增加/减小信号量值，相应于对共享资源的释放和占有
- 函数原型
  - int semop(int semid, struct sembuf *sops, unsigned nsops);
- 参数说明
  - semid
    - ✓ 信号量集ID
  - sops
    - ✓ 指向类型为sembuf的一个数组
  - nsops
    - ✓ sops指向的数组大小
- 返回值
  - 调用成功返回0
  - 失败返回返回-1

sembuf结构

- 定义位置
  - /inclide/linux/sem.h

```
struct sembuf {
  ushort  sem_num; /*在 数 组 中 信 号 量 的 索 引 值 */
  short   sem_op;  /*信 号 量 操 作 值 (正 数 、 负 数 或 0)*/
  short   sem_flg; /*操 作 标 志 , 为 IPC_NOWAIT或 SEM_UNDO*/
};
```

  - sem_num
    - ✓ 指明对信号量集的哪一个信号操作
  - sem_op作说明
    - ✓ >0：释放资源
    - ✓ =0：wait-for-zero
    - ✓ <0：申请资源
  - sem_flg说明
    - ✓ IPC_NOWAIT：当期望的操作无法完成时，直接返回
    - ✓ SEM_UNDO：自动释放标记

信号量的基本操作—semctl()

- 功能
  - 对信号量进行控制，可用于删除一个信号量
- 函数原型
  - **int semctl(int semid, int semnum, int cmd, union semun arg);**
- 参数说明
  - **cmd**
    - ✓指定具体的操作类型
  - **arg**
    - ✓设置或返回信号量信息
- 返回值
  - 调用成功返回0
  - 失败时返回-1

信号量的基本操作—semctl()

- cmd命令类型
  - IPC_STAT
    - ✓ 将信号量集的信息复制到arg，此时第二个参数无用
  - IPC_SET
    - ✓ 与上一个操作相反
  - IPC_RMID
    - ✓ 删除信号量集，不使用第四个参数
  - GETALL/SETALL
    - ✓ 获取/设置所有信号量的值
  - GETVAL/SETVAL
    - ✓ 获得/设置指定信号量的值
  - GETNCNT
    - ✓ 返回等待semnum所代表信号量的值增加的进程数
  - GETPID
    - ✓ 返回最后一个对semnum所代表信号量执行semop操作的进程ID
  - GETZCNT
    - ✓ 返回等待semnum所代表信号量的值变成0的进程数

```
union semun {
    int val;                /* value for SETVAL */
    struct semid_ds *buf;       /* buffer for IPC_STAT & IPC_SET */
    unsigned short *array;      /* array for GETALL & SETALL */
    struct seminfo *__buf;      /* buffer for IPC_INFO */
    void *__pad;
};
```

信号量应用示例

❖ semtest.c

```
#include <linux/sem.h>
#include <stdio.h>
#include <errno.h>
#define SEM_PATH "/unix/my_sem"
#define max_tries 3

int semid;
main(){
  int flag1,flag2,key,i,init_ok,tmperrno;
  struct semid_ds sem_info;
  struct seminfo sem_info2;
  union semun arg;       //union semun:  请参考附录2
  struct sembuf askfor_res, free_res;
  flag1=IPC_CREAT|IPC_EXCL|00666;
  flag2=IPC_CREAT|00666;
  key=ftok(SEM_PATH,'a');
  //error handling for ftok here;
  init_ok=0;
  semid=semget(key,1,flag1);//create a semaphore set that only includes one semphore.
```

信号量应用示例

❖ semtest.c

```
if(semid<0) {
  tmperrno=errno;
  perror("semget");
  if(tmperrno==EEXIST){
    semid=semget(key,1,flag2);
    //flag2 只包含了IPC_CREAT标志,
    //参数nsems(这里为1)必须与原来的信号灯数目一致
    arg.buf=&sem_info;
    for(i=0; i<max_tries; i++) {
      if(semctl(semid, 0, IPC_STAT, arg)==-1){
        perror("semctl error");
        i=max_tries;
      } else{
        if(arg.buf->sem_otime!=0){
          i=max_tries;
          init_ok=1;
        }else
          sleep(1);
      }
    }
    if(!init_ok){
      arg.val=1;
      if(semctl(semid,0,SETVAL,arg)==-1)
        perror("semctl setval error");
    }
  } else {
    perror("semget error, process exit");
    exit(-1);
  }
```

信号量应用示例

❖semtest.c

```
} else{ //semid>=0; do some initializing
  arg.val=1;
  if(semctl(semid,0,SETVAL,arg)==-1)
    perror("semctl setval error");
}
//get some information about the semaphore and the limit of semaphore in redhat8.0
arg.buf=&sem_info;
if(semctl(semid, 0, IPC_STAT, arg)==-1)
  perror("semctl IPC_STAT");
printf("owner's uid is %d\n",    arg.buf->sem_perm.uid);
printf("owner's gid is %d\n",    arg.buf->sem_perm.gid);
printf("creater's uid is %d\n",    arg.buf->sem_perm.cuid);
printf("creater's gid is %d\n",    arg.buf->sem_perm.cgid);
arg.__buf=&sem_info2;
if(semctl(semid,0,IPC_INFO,arg)==-1)
  perror("semctl IPC_INFO");
printf("the number of entries in semaphore map is %d \n", arg.__buf->semmap);
printf("max number of semaphore identifiers is %d \n", arg.__buf->semmni);
printf("mas number of semaphores in system is %d \n", arg.__buf->semmns);
printf("the number of undo structures system wide is %d \n", arg.__buf->semmnu);
printf("max number of semaphores per semid is %d \n", arg.__buf->semmsl);
printf("max number of ops per semop call is %d \n", arg.__buf->semopm);
printf("max number of undo entries per process is %d \n", arg.__buf->semume);
printf("the sizeof of struct sem_undo is %d \n", arg.__buf->semusz);
printf("the maximum semaphore value is %d \n", arg.__buf->semvmx);
```

信号量应用示例

❖semtest.c

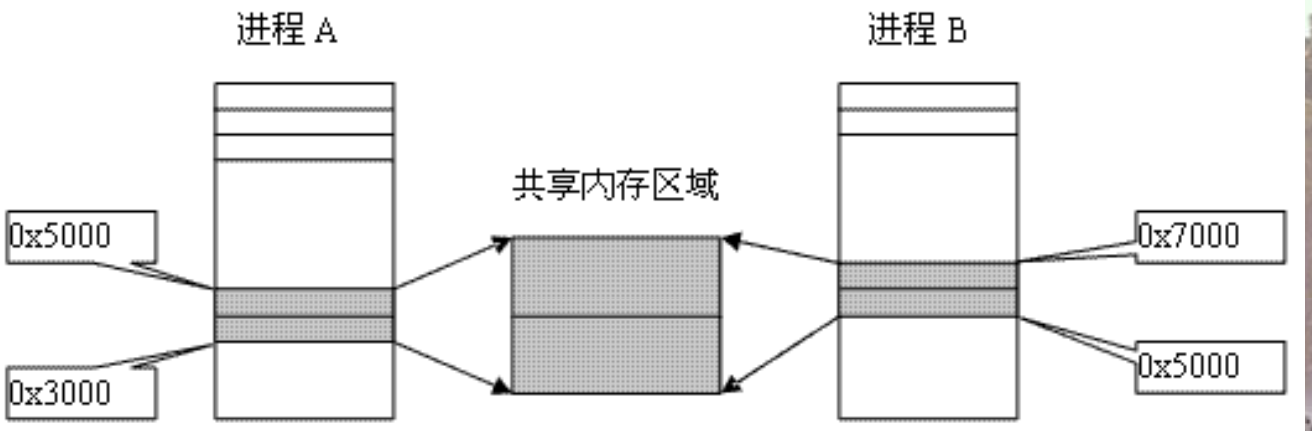
```
  //now ask for available resource:
  askfor_res.sem_num=0;
  askfor_res.sem_op=-1;
  askfor_res.sem_flg=SEM_UNDO;
  if(semop(semid,&askfor_res,1)==-1)//ask for resource
    perror("semop error");
  sleep(3); //do some handling on the sharing resource here, just sleep on it 3 seconds
  printf("now free the resource\n");
  //now free resource
  free_res.sem_num=0;
  free_res.sem_op=1;
  free_res.sem_flg=SEM_UNDO;
  if(semop(semid,&free_res,1)==-1)//free the resource.
    if(errno==EIDRM)
       printf("the semaphore set was removed\n");
  //you can comment out the codes below to compile a different version:
  if(semctl(semid, 0, IPC_RMID)==-1)
    perror("semctl IPC_RMID");
  else
    printf("remove sem ok\n");
}
```

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

共享内存概述

- 基本思想
  - 多个进程共享的一块内存区域
    - ✓不同进程可把共享内存映射到自己的一块地址空间
    - ✓不同进程进行映射的地址空间不一定相同
    - ✓共享内存区的进程对该区域的操作是互见的
- 优点
  - 数据交换效率高，无须用户态切、核心态切换开销

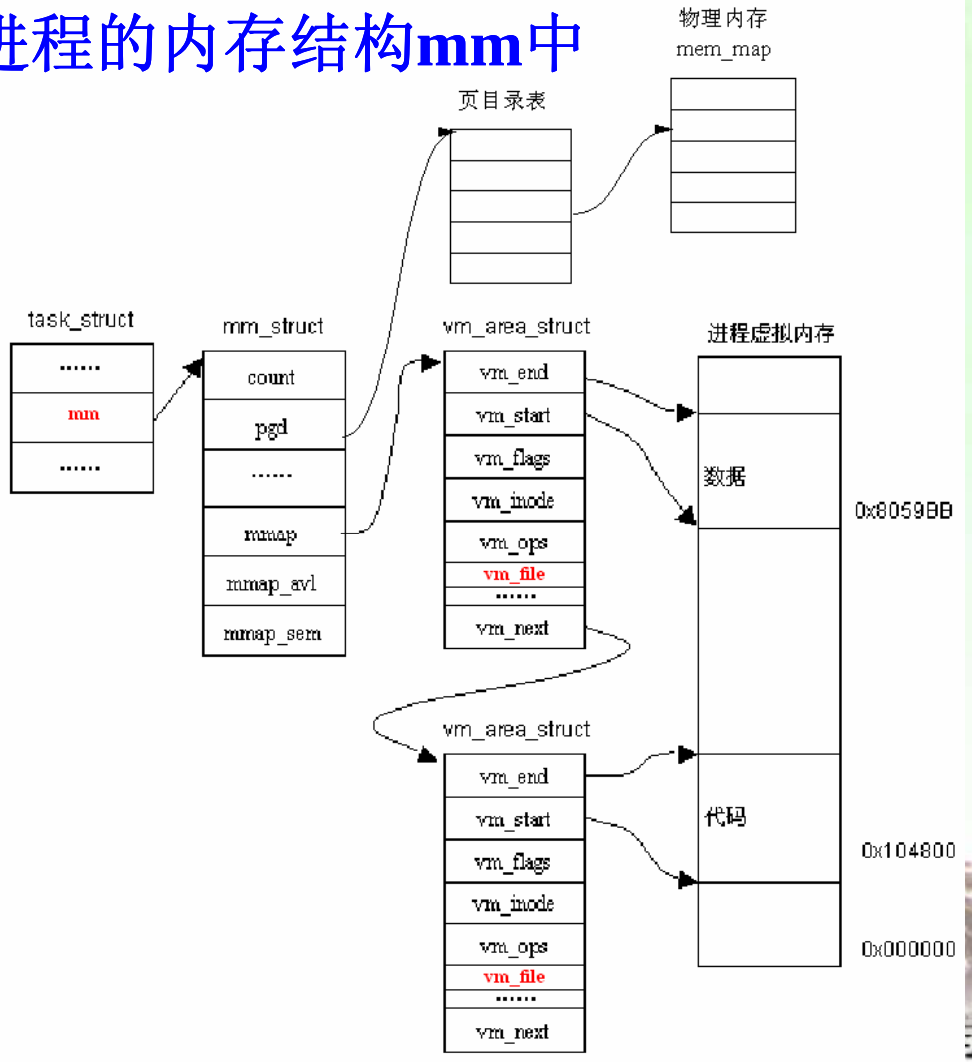


共享内存实现途径

- **mmap()系统调用**
  - 将普通文件在不同的进程中打开并映射到内存
  - 不同进程通过访问映射来达到共享内存目的
- **POSIX共享内存机制（Linux 2.6未实现）**
- **System V共享内存**
  - 在内存文件系统**tmpfs**中建立文件
  - 文件映射到不同进程空间

进程与共享内存的关联

- 创建vm_area_struct[inlcude/linux/mm_types.h]
- 链接到进程的内存结构mm中

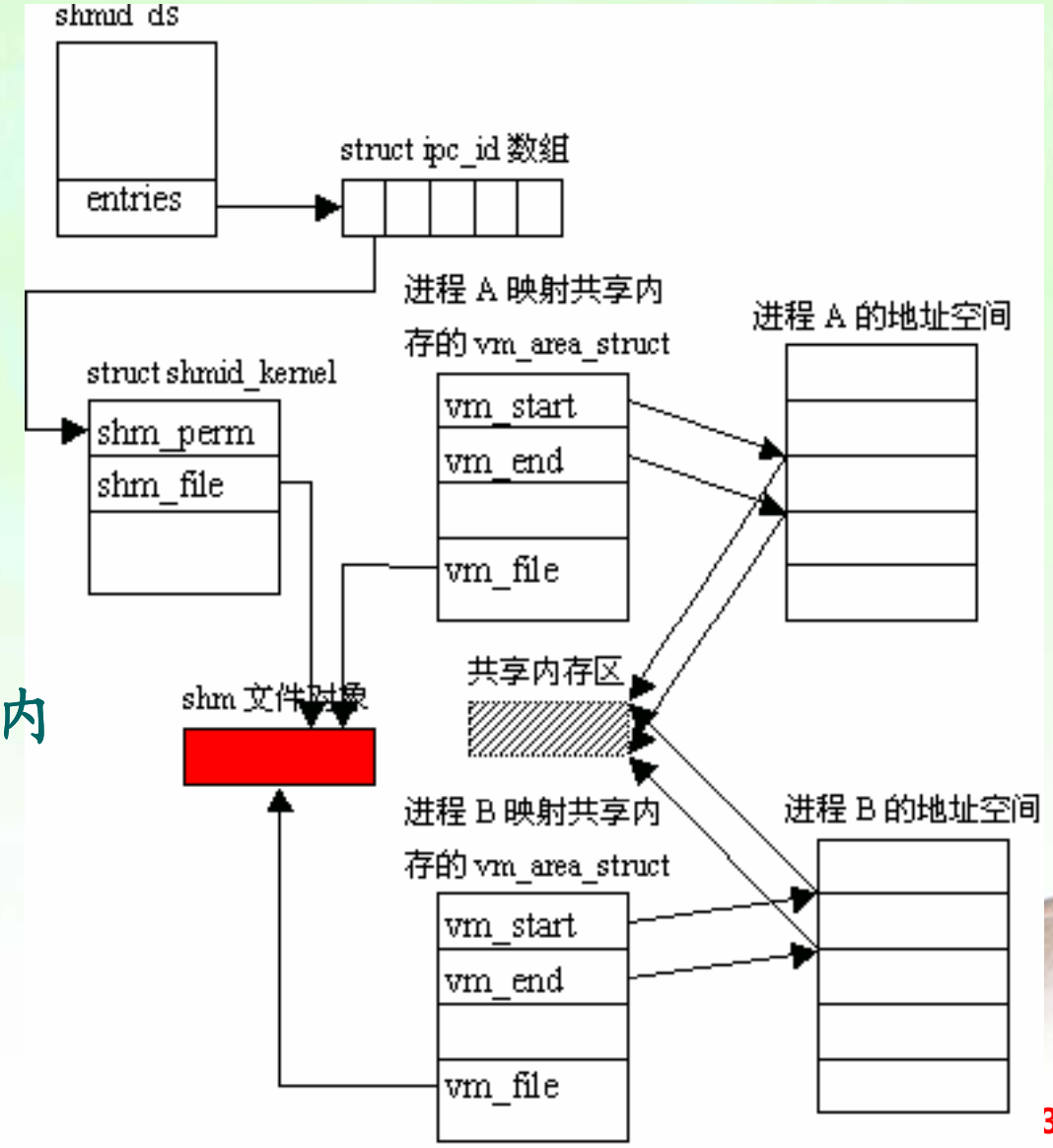


System V IPC共享内存的实现

- 通过映射到tmpfs中的shm文件对象实现共享主存
  - 每个共享主存区对应tmpfs中的一个文件（通过shmid_kernel关联）
- 创建方法
  - 基本函数
    - ✓ shmget()
  - 创建过程
    - ✓ 从主存申请共享主存管理结构，初始化相应shmid_kernel结构
    - ✓ 在tmpfs中创建并打开一个同名文件
    - ✓ 在主存中建立与该文件对应的dentry及inode结构
    - ✓ 返回相应标识符
  - 说明
    - ✓ 新建共享主存区不属于任何进程，各进程可通过函数shmget映射到自己的虚拟空间
    - ✓ 每个共享主存区都有一个内核控制结构struct shmid_kernel

共享内存相关数据结构

- shm_ids
  - 数据类型
    - ✓ struct ipc_ids
  - 内核全局数据结构
  - 重要成员
    - ✓ entries
- shmid_kernel
  - 每个共享内存区的内核控制结构
  - 重要成员
    - ✓ shm_file



ipc_ids相关结构定义

```
struct ipc_ids {
 int size;      /* entries数 组 的 大 小 */
 int in_use;    /* entries数 组 已 使 用 的 元 素 个 数 */
 int max_id;
 unsigned short seq;
 unsigned short seq_max;
 truct semaphore sem;  /*控 制 对 ipc_ids结 构 的 访 问 */
 spinlock_t ary;/*自 旋 锁 控 制 对 数 组 entries的 访 问 */
 struct ipc_id* entries;
};

struct ipc_id {
    struct kern_ipc_perm* p;
};
```

```
struct kern_ipc_perm {
__kernel_key_t key;     /*IPC键 */
__kernel_uid_t uid;     /*资 源 所 有 者 的 UID*/
__kernel_gid_t gid;     /*资 源 所 有 者 的 GID*/
__kernel_uid_t cuid;    /*创 建 这 个 资 源 的 进 程 UID */
__kernel_gid_t cgid;    /*创 建 这 个 资 源 的 进 程 GID*/
__kernel_mode_t mode; /*文 件 系 统 类 型 的 权 限 */
unsigned short seq;     /*位 置 使 用 序 号 */
};
```

shmid_kernel结构定义

- 内核私有结构，存储被映射文件地址
  - 每个共享主存区对象对应特殊存储块shm中的一个文件
  - 一般情况下，不支持read( )、write( )等函数访问shm文件

```
struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm    shm_perm;
    struct file *           shm_file;
    unsigned long           shm_nattch; //已 建 立 映 射 的 数 据
    unsigned long           shm_segsz;//共 享 内 存 区 大 小
    time_t                  shm_atim;
    time_t                  shm_dtim;
    time_t                  shm_ctim;
    pid_t                   shm_cprid;
    pid_t                   shm_lprid;
    struct user_struct      *mlock_user;
};
```

共享主存基本操作—shmget()

- 功能
  - 创建/获得一个共享主存区
- 函数原型
  - **int shmget(key_t key, int size, int flag);**
- 参数说明
  - **key:** 共享主存区键值
  - **size:** 大小（以字节计）
    - ✓ 创建新段时必须指定其size
    - ✓ 访问现存段时size=0
- 返回值
  - 成功返回0
  - 失败则返回-1
- 说明
  - 创建共享内存区时，内核中将创建shmid_kernel实例

sys_shmget系统调用

- 功能
  - shmget()对应的系统调用，创建/获得共享内存区
- 关键函数
  - newseg()
    - ✓创建新共享内存
  - shmlock()
    - ✓获取现存共享内存

sys_shmget系统调用实现代码

```
asmlinkage long sys_shmget (key_t key, size_t size, int shmflg){
    struct shmid_kernel *shp;
    int err, id = 0;

    down(&shm_ids.sem); //加 锁
    if (key == IPC_PRIVATE) {//创 建 共 享 内 存 区
        err = newseg(key, shmflg, size);
    } else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
        if (!(shmflg & IPC_CREAT))
            err = -ENOENT;
        else
            err = newseg(key, shmflg, size);//创 建 共 享 内 存 区
    } else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
        err = -EEXIST;
    } else { //获 取 共 享 内 存 区
        shp = shm_lock(id); //找 到 对 应 shmid_kernel结 构
        if(shp==NULL)
            BUG();
        if (shp->shm_segsz < size)//检 查 共 享 内 存 大 小
            err = -EINVAL;
        else if (ipcperms(&shp->shm_perm, shmflg))//检 查 访 问 权 限
            err = -EACCES;
        else
            err = shm_buildid(id, shp->shm_perm.seq);
        shm_unlock(id);
    }
    up(&shm_ids.sem);
    return err;
}? end sys_shmget ?
```

shm_lock系统调用

- 功能
  - 获取现有共享内存区

```
#define shm_lock(id)     ((struct shmid_kernel*)ipc_lock(&shm_ids,id))
```

```
extern inline struct kern_ipc_perm* ipc_lock(struct ipc_ids* ids, int id){
    struct kern_ipc_perm* out;
    int lid = id % SEQ_MULTIPLIER;
    if(lid >= ids->size)
        return NULL;

    spin_lock(&ids->ary);
    out = ids->entries[lid].p;
    if(out==NULL)
        spin_unlock(&ids->ary);
    return out;
}
```

共享主存基本操作—shmat()

- 功能
  - 将共享主存区连接到进程虚拟地址空间
- 函数原型
  - void *shmat(int shmid, char *addr, int flag);
- 参数说明
  - addr
    - ✓ Shmaddr=NULL：由系统选择地址，这是推荐方法
    - ✓ Shmaddr!=NULL：返回地址取决于flag是否指定SHM_RND
      1）未指定SHM_RND，附接到shmaddr指定的地址
      2）指定SHM_RND，进位到最低可用SHMLBA地址
  - Flag
    - ✓ SHM_RND
      SHM_RDONLY：以只读的方式映射到进程的地址空间
      SHM_REMAP：替代掉与指定参数重叠的现存共享区段的映射
- 返回值
  - 成功时为返回映射区起始地址
  - 出错时为-1

共享主存基本操作—shmdt()

- 功能
  - 把一个共享主存区从指定进程的虚地址空间断开
- 函数原型
  - **int shmdt(char *addr);**
- 参数说明
  - **addr**
    - ✓ 需断开连接的虚地址，即由**shmat( )**返回的虚地址。
- 返回值
  - 调用成功返回**0**
  - 失败时返回**-1**

共享主存基本操作—shmctl()

- 功能
  - 控制共享主存区属性，读取或修改其状态信息
- 函数原型
  - int shmctl(int shmid, int cmd, struct shmid_ds *buf);
- 参数说明
  - buf
    - ✓ 用户缓冲区地址
  - cmd
    - ✓ IPC_STAT：读取shmid_ds结构，并存放在由buf指向的结构中
    - ✓ IPC_SET：按buf的值设置与此段相关结构中的下列3个字段：shm_perm.uid、shm_perm.gid以及shm_perm.mode。
    - ✓ IPC_RMID：从系统中删除该共享存储段
    - ✓ SHM_LOCK：锁住共享存储段
    - ✓ SHM_UNLOCK：解锁共享存储段
- 返回值
  - 调用成功返回0
  - 失败返回-1

共享主存示例

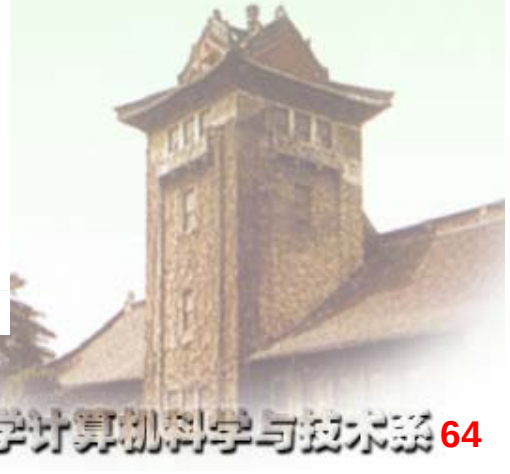
❖ **shmwritetest.c**

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct{
   char name[4];
   int age;
}people;
void main(int argc,char ** argv){
   int shm_id,i;
   key_t key;
   char temp;
   people *p_map;
   char* name = ".";
   key=ftok(name,0);
   printf("key is : %d\n",key);
   if(key==-1)
      perror("ftok error");
   shm_id=shmget(key,4096,IPC_CREAT|0600);
   printf("hello!");
```

共享主存示例

❖ **shmwritetest.c**

```
/*   if(shm_id==-1)
   {
     perror("shmget error");
     return;
   }*/
   printf("hello,shmat success! ");
   shmat(shm_id,NULL,0);
   printf("hello,shmat success! ");
   p_map=(people *)shmat(shm_id,NULL,0);
   temp='a';
   printf("hello,shmat success! ");
   for(i=0; i<10; i++) {
     temp+=1;
     memcpy((*(p_map+i)).name,&temp,1);
     (*(p_map+i)).age=20+i;
   }
   if(shmdt(p_map)==-1)
     perror("dtach error");
}
```



共享主存示例

❖ shmreadtest.c

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct{
  char name[4];
  int age;
}people;
void main(int argc,char ** argv){
int shm_id,i;
key_t key;
people*p_map;
char*name=".";
key=ftok(name,0);
if(key==-1)
  perror("ftok error");
shm_id=shmget(key,4096,IPC_CREAT);
if(shm_id==-1){
perror("shmget error");
return;
}
p_map=(people *)shmat(shm_id,NULL,0);
for(i=0;i<10; i++){
printf("name:%s\n",(*(p_map+i)).name);
printf("age %d\n",(*(p_map+i)).age);
}
if(shmdt(p_map)==-1)
  perror("detach error");
}
```

System V IPC进程通信综合示例

- 基于信号量、共享内存、信号在父子进程之间实现数据同步访问
  - 在系统中创建两个共享内存区
  - 父进程通过键盘输入读取两次数据，分别写入两个共享队列
  - 子进程从共享内存读取数据，并输出到屏幕
  - 同步关系
    - ✓ 只有父进程在向第一个共享内存中写入数据后，子进程才能开始输出数据
  - 程序终止
    - ✓ 父进程接收到ctrl+c时，回收IPC资源，终止程序

System V IPC进程通信综合示例

```
#define SHMKEY1 (key_t)0x10
#define SHMKEY2 (key_t)0x15
#define SEMKEY  (key_t)0x20
#define SIZ 5*BUFSIZ
#define IFLAGS (IPC_CREAT|IPC_EXCL)
#define ERR ((struct databuf*)-1)
struct sembuf p1 = {0, -1, 0};
struct sembuf v1 = {0, 1, 0};
struct sembuf p2 = {1, -1, 0};
struct sembuf v2 = {1, 1, 0};
static int shmid1, shmid2, semid;
struct databuf{
   int d_nread;
   char d_buf[SIZ];
};
void fatal(char *mes){
   perror(mes);
   exit (1);
}
```

System V IPC进程通信综合示例

```
void getshm (struct databuf **p1, struct databuf **p2){//创建共享内存
  if ((shmid1=shmget(SHMKEY1, sizeof(struct databuf), 0600|IFLAGS)) < 0){
       fatal ("shmget");
  }

  if ((shmid2=shmget(SHMKEY2, sizeof(struct databuf), 0600|IFLAGS)) < 0){
       fatal ("shmget");
  }
  //映射
  if ((*p1=(struct databuf*)(shmat(shmid1, 0, 0))) == ERR){
       fatal("shmat");
  }

  if ((*p2=(struct databuf*)(shmat(shmid2, 0, 0))) == ERR){
       fatal("shmat");
  }
}
```

System V IPC进程通信综合示例

```
int getsem(){
   int semid;
   if ((semid=semget(SEMKEY, 2, 0600|IFLAGS)) < 0){
        fatal("segmet");
   }
   //初始化信号量,假定第一个信号灯没有资源
   if (semctl(semid, 0, SETVAL, 0) < 0){
        fatal ("semctl");
   }
   //第二个有信息,如果两个信号都没有资源,将会出现死锁
   if (semctl(semid, 1, SETVAL, 1) < 0){
        fatal ("semctl");
   }
   return (semid);
}
```

System V IPC进程通信综合示例

```
void myremove(){
   if (shmctl(shmid1, IPC_RMID, NULL) < 0){
        fatal("shmctl");
   }
   if (shmctl(shmid2, IPC_RMID, NULL) < 0){
        fatal("shmctl");
   }
   if (semctl(semid, 0, IPC_RMID, NULL) < 0)
        fatal("semctl");
   exit(0);
}
```

System V IPC进程通信综合示例

```
void reader(int semid, struct databuf *buf1, struct databuf *buf2){
  for(;;){
    //从键盘读入数据,放在buf1->d_buf,
    buf1->d_nread = read(0, buf1->d_buf, SIZ);
    //v操作,释放第一个信号灯资源
    semop(semid, &v1, 1);
    //p操作,申请第二个信号灯资源,若没有资源,则阻塞进程直到有资源.
    semop(semid, &p2, 1);
     buf2->d_nread = read(0, buf2->d_buf, SIZ);
     semop(semid, &v2, 1);
     semop(semid, &p1, 1);
  }
}
```

System V IPC进程通信综合示例

```
void writer(int semid, struct databuf *buf1, struct databuf *buf2){
  for(;;) {
    //p操作,等待第一个信号的资源
    semop(semid, &p1, 1);
    //1代表输出到屏幕
    write(1, buf1->d_buf, buf1->d_nread);
    semop(semid, &v1, 1);
    semop(semid, &p2, 1);
    write(1, buf2->d_buf, buf2->d_nread);
    semop(semid, &v2, 1);
  }
}
```

System V IPC进程通信综合示例

```
int main() {
  int semid, pid;
  struct databuf *buf1, *buf2;
  semid = getsem();
  getshm (&buf1, &buf2);

  switch(pid=fork()){
    case -1:
      fatal("fork");
      break;
    case 0:
      writer(semid, buf1, buf2);
      break;
    default:
      //将ctrl+c与myremove函数关联
      //用于回收系统资源（共享内存区,信号量）
      signal(SIGINT, myremove);
      reader(semid, buf1, buf2);
      break;
  }
  exit(0);
}
```

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

消息队列实现进程间通信

- 实验说明
  - 利用消息队列解决客户及服务进程之间的通信问题
- 解决方案
  - 服务进程首先需要创建一个消息队列，随后客户进程可以打开消息队列，从而实现消息的发送和接收
  - 涉及到的函数包括**msgget(), msgsnd(), msgrev()及msgctl()**

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

信号量实现进程同步

❖实验说明

- 利用信号量解决生产者-消费者问题

❖解决方案

- 生产者－消费者问题是典型的进程同步问题，其本质是如何控制并发进程对有界共享区的访问
- 生产者进程生产产品，然后将产品放置在一个空缓冲区中供消费者进程消费
- 消费者进程从缓冲区中获得产品，然后释放缓冲区
- 当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个空缓冲区
- 当消费者进程消费产品时，如果没有满的缓冲区，那么消费者进程将被阻塞，直到新的产品被生产出来

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

基于信号量采用多线程技术实现进程同步

❖实验说明

- 利用信号量解决理发师问题，可采用pthread线程库的sem_wait( )及sem_post( )等函数实现

❖解决方案

- 理发师问题是经典的进程同步问题，共需要3个信号量和一个控制变量来协调理发师、理发椅及顾客间的活动
- 信号量customers(不包括正在理发的顾客)，用来记录等候理发的顾客数，并用作阻塞理发师进程，初值为0;
- 信号量barbers，用来记录正在等候顾客的理发师数，并用作阻塞顾客进程，初值为0;
- 信号量mutex用于互斥，初值为1
- 控制变量waiting用来记录等候理发的顾客数，实际上是customers的一份拷贝，初值均为0
- 之所以使用waiting是因为无法读取信号量的当前值

主要内容

- 背景知识
  - System V的进程间通信机制
  - 消息队列
  - 信号量
  - 共享主存
- 实验内容
  - 消息队列实现进程间通信
  - 信号量实现进程同步
  - 基于信号量采用多线程技术实现进程同步
  - 共享主存实现进程间通信

共享主存实现进程间通信

❖实验说明

- 利用共享主存解决读者-写者问题
- 要求由写者创建一个共享主存，并向其中写入数据，读者进程随后从该共享区中访问数据

❖解决方案

- 为基于共享主存解决读者-写者问题，需要由写进程首先创建一个共享主存，并将该共享主存区分别映射到读者进程和写者进程的虚拟空间
- 随后，写进程开始向共享主存写数据，读进程从共享主存区获取数据

第4章 System V IPC进程通信