Linux 内核分析
期末实验报告

学 院 系 别 信 息 学 院

专 业 名 称 人 工 智 能

学 生 姓 名 王 子 涵

学 生 学 号 22920192204084

任 课 教 师 陈 毅 东

2022 年 06 月 10 日

Linux内核分析

一、实验目的

实现一个 Linux 下的多进程并发程序以模拟某个宾馆的预约系统运作情况。

二、实验要求

- 每个客人用一个进程表示，该进程可模拟客人的所有操作：预约、取消预约、查询预约情况。
- 两个或多个客人可能同时执行同样的交易，因此必要时应该按原子方式（互斥地）实现上述这些操作。
- 如果客人企图预约被占用的房间，则程序应拒绝该请求并打印“房间被占用”信息；如果客人想预约的是不存在的房间，则系统也应打印错误信息。
- 客人的预约按天计，预约类型包括如下三类：
  a. 预约某个特定的房间
  b. 预约多个连续房间（房号连续）
  c. 预约多个房间，但不指定房间号
- 请注意，处理针对多个房间的预约请求时应该采用原子方式，即：要么都预约成功，要么都预约失败。
- 判断房间号连续时，宾馆最大房间号的下一个连续房间号是 1。
- 上述第三种预约类型虽不限房号，但所有分配房间必须在时间上能完全满足需求。
- 为正确实现上述并发操作，须使用 Linux 信号量对宾馆房间相关信息的存取进行控制；宾馆的所有房间相关信息则应使用共享存储区来存放。
- 合法的交易请求有：
  - **reserve 房间号 年 月 日 预约天数 预约姓名**
  - **cancel 房间号 年 月 日 预约天数 预约姓名**
  - **reserveblock 房间数 第一个房间号 年 月 日 预约天数 预约姓名**
  - **cancelblock 房间数 第一个房间号 年 月 日 预约天数 预约姓名**
  - **reserveany 房间数 年 月 日 预约天数 预约姓名**
  - **cancelany 房间数 年 月 日 预约天数 预约姓名**
  - **check 预约姓名**
- 所有预约都是针对 2022 年或 2023 年的。
- 只有预约过的房间才能取消预约，允许部分取消！
- 宾馆的合法房间号是可能不连续的，分楼层编号或保留设备损坏房间等都可能导致此情况。
- 预约姓名不能包含空格。
- 为了模拟交易延时，在输入文件中，每个客人的请求序列前都会包含三行信息分别用来表示客人可执行的三类操作（预约、取消、查询）的执行秒数（见下面的输入文件模板）。实现时可通过在相应操作临界区中调用 sleep 加以实现。
- 程序的输入是一个文本文件，模板如下：

```
n // 房间总数（不超过 128）
room_number // 合法房间号
room_number // 合法房间号
……
room_number // 合法房间号
m // 客人总数（不超过 255）
customer // 标志串：开始一个客人请求序列
reserve reserve_time // 秒数
cancel cancel_time // 秒数
check check_time // 秒数
…… // 合法的请求序列
end. // 标志串：结束一个客人请求序列
……
customer // 标志串：开始一个客人请求序列
reserve reserve_time // 秒数
cancel cancel_time // 秒数
check check_time // 秒数
…… // 合法的请求序列
end. // 标志串：结束一个客人请求序列
```

- 程序输出：针对输入的各交易的执行情况以及最后的宾馆房间分配信息。

三、代码结构及项目框架

代码结构：

项目框架：

整个宾馆预约系统分为五个程序文件（hotel.h, hotel.c, my_ipc.h, my_ipc.c, main.c）、一份配置说明、一份测试文件和Makefile文件。

程序文件：

- hotel.h

定义日期、房间、人员、人员操作、宾馆等类。

- hotel.c

定义一些具体的实现方法。

- my_ipc.h

定义信号灯机制和共享内存来实现多进程通信。

- my_ipc.c

定义具体实现方法。

- main.c

模拟整个预约操作以及读取测试文件。

四、实现方法

hotel.h

该文件定义了一些最基本的类：

```
> class date//日期类…

> struct op//客户操作结构体…

> struct customer//客户类…

> struct reserved_info//房间预订信息结构体…

  void copy_res(struct reserved_info &a,struct reserved_info b); //复制订房信息


> class room//房间类…

> class hotel//宾馆类…

  bool check_data(int month,int day);// 检查日期的合法性

  void read_hotel_msg(FILE *fp,class hotel *myhotel);//从文件读取hotel信息

  void read_cust_msg(FILE *fp,int &cust_num, vector<struct customer> &cus_vec);//从文件读取customer信息
```

1. class date

定义类成员：年、月、日：

```
public:
    int year;
    int month;
    int day;

    date(){}

    date(int y,int m,int d)
    {
        year=y;
        month=m;
        day=d;
    }
```

void add:

- 函数功能：计算当前日期加上n天后的日期。
- 具体实现：按照月份不同，分别进行计算。

```
if(month==2){
    if(day>28){
        day=day-28;
        month++;
    }
    else
        break;
}
else if(month==4||month==6||month==9||month==11){
    if(day>30){
        day=day-30;
        month++;
    }
    else
        break;
}
else if(month==12){
    if(day>30){
        day=day-30;
        year++;
        month=1;
    }
}
```

```
else{
    if(day>31){
        day=day-31;
        month++;
    }
    else
        break;
}
```

注意：12月要单独考虑，因为涉及到年份。

int compareTo:

- 函数功能：进行日期比较。
- 具体实现：依次比较年月日即可。

```
int compareTo(class date D)//日期比较
{
    if(year>D.year)
        return 1;
    else if(year<D.year)
        return -1;
    else{
        if(month>D.month)
            return 1;
        else if(month<D.month)
            return -1;
        else
        {
            if(day>D.day)
                return 1;
            else if(day<D.day)
                return -1;
            else
                return 0;
        }
    }
}
```

2. struct op

定义客户的预约操作：

```
struct op//客户操作结构体
{
    char op_type[20];  //操作类型
    int op_time;
    int room_NO; //房间号或者第一个房间号
    int room_num;//房间数目
    class date start_date;//预约起始日期
    int days;     //天数
};
```

3. struct customer

定义客户名字、操作数量以及具体操作的集合：

```
struct customer//客户类
{
    char name[50];
    int op_num;
    struct op cus_op[MAX_OP];//操作集合
};
```

4. struct reserved_info

定义基本预约信息：起止时间、预约天数以及姓名。

```
struct reserved_info//房间预订信息结构体
{
    class date start_date ;         //预约的起始时间
    class date end_date ;           //预约的结束时间
    int days ;      //预约天数
    char  name[10] ;        //预约者 姓名
};
```

5. class room

定义房间基本信息以及一些操作：

```
class room//房间类
{
public:
    int room_NO ;   //房间号
    int res_num;//预约信息数量
    struct reserved_info res_info[3]; //房间被预约信息数组
    room() ;
    bool can_reserved(class date start_date,int days,int &loca); //检查房间在请求时间内是否被预订
    bool addreserve(class date start_date,int days,char name[50],int loca);//新增预订信息
    bool delreserve(int loca); //删除预订信息
    bool search(class date start_date,int days,char name[50],int &loca);//查找订房信息
    bool search_by_name(char name[50]);//按名字查询订房信息并显示
} ;
```

6. class hotel

定义房间数量，以及打印信息：

```
class hotel//宾馆类
{
public:
    class room rooms[MAX_ROOM] ;
    int roomnum ;
    hotel() ;
    bool init_hotel(vector<int>&vec) ;//初始化宾馆
    int location_room ( int room_NO ) ; //依据房间号找出房间在数组中的位置，没有返回-1
    bool do_request (char name[50], struct op cusop ) ;//宾馆处理客户请求
    void print();//显示宾馆房间的分配信息
} ;
```

hotel.c

1. bool check_data

- 函数功能：检查日期的合法性；
- 具体实现：首先检查年份，要求2022或2023；接着检查月份，对应的月份天数不同；

```
bool check_data(int year,int month,int day){
    if(year!=2022 && year!=2023)
        return false;
    else if ( (month >= 1) && (month <= 12) ){
        if ( month == 2 ){
            if ( day >= 1 && day <= 28 )
                return true ;
        }
        else if ( month==1||month==3||month==5||month==7||month==8||month==10||month==12){
            if ( day >=1 && day <= 31 )
                return true ;
        }
        else{
            if ( day >= 1 && day <= 30 )
                return true ;
        }
    }
    return false ;
}
```

注意：2022与2023均是平年！

2. bool room::can_reserved

- 函数功能：检查某个房间是否可以被预定；
- 具体实现：

调用add函数来计算该预约的结束日期；之后判断两个条件：该房间的预约信息为0||本次预约结束时间早于其他预约的开始时间，满足其一即可代表该房间可以被预约，并记预约序号为0表示该预约排在首位；若均不满足，则检查本次预约排在哪两次预约之间，并记录预约序号；

若上述条件还不满足，那么表示本次预约与其他预约产生冲突，记录序号为-1并返回false；

```
bool room::can_reserved(class date start_date,int days,int &loca){
    int i,j;
    class date end_date(start_date.year,start_date.month,start_date.day);
    end_date.add(days);
    i=0;j=1;
    if(res_num==0 || res_info[0].start_date.compareTo(end_date)>0){
        loca=0;
        //return true;
    }
    else{
        while(j!=res_num && res_info[i].end_date.compareTo(start_date) < 0){
            if(res_info[j].start_date.compareTo(end_date)>0 ){
                loca=j;
                break;
            }
            else{
                i++;
                j++;
            }
        }
        if(j==res_num && res_info[i].end_date.compareTo(start_date) < 0){
            loca=j;
            //return true;
        }
        else
            loca=-1;

    }
    if(loca>=0)
        return true;
    else
        return false;
}
```

3. bool room::addreserve

- 函数功能：在某个房间的预约信息组中新增一条预约；
- 具体实现：首先调用add函数计算本次预约的结束时间，接着根据can_reserved函数记录的序号，按序插入数组即可；

```
bool room::addreserve(class date start_date,int days,char name[50],int loca){
    class date end_date(start_date.year,start_date.month,start_date.day);
    end_date.add(days);
    struct reserved_info resinf;
    resinf.days=days;
    resinf.end_date=end_date;
    resinf.start_date=start_date;
    strcpy(resinf.name,name);
    for(int i=res_num;i>=loca+1;i--){
        copy_res(res_info[i],res_info[i-1]);
    }
    copy_res(res_info[loca],resinf);
    res_num++;
    return true;
}
```

4. bool room::delreserve

- 函数功能：删除某个预约信息；
- 具体实现：由于用数组存放预约信息，因此从被删除的那条开始逐条覆盖即可；

```
bool room::delreserve(int loca){
    for(int i=loca;i<res_num;i++){
        copy_res(res_info[i],res_info[i+1]);
    }
    res_num--;
    return true;
}
```

5. bool room::search

- 函数功能：查找某个预约信息；
- 具体实现：依次判断预约的起始日期年月日，以及预约天数，完全相同则返回那条索引；

```
bool room::search(class date start_date,int days,char name[50],int &loca){
    int i=0;
    while(i<res_num){
        if(strcmp(res_info[i].name,name)==0 &&
            res_info[i].start_date.year==start_date.year &&
            res_info[i].start_date.month==start_date.month &&
            res_info[i].start_date.day==start_date.day &&
            res_info[i].days==days)
        {
            loca=i;
            return true;
        }
        i++;
    }
    return false;
}
```

6. bool room::search_by_name

- 函数功能：根据名字进行查询；
- 具体实现：比较名字即可，存在符合要求的预约信息就打印出来；

```
bool room::search_by_name(char name[50]){
    int i=0;
    bool flag=false;
    while(i<res_num){
        if(strcmp(res_info[i].name,name)==0){
            flag=true;
            printf("%s check result: reserve NO.%d from %d.%d.%d for %ddays \n",name,room_NO,
                res_info[i].start_date.year,res_info[i].start_date.month,res_info[i].start_date.day,
                res_info[i].days);
        }
        i++;
    }
    return flag;
}
```

7. bool hotel::init_hotel

- 函数功能：初始化宾馆，定义房间号和房间数量；
- 具体实现：

```
bool hotel::init_hotel(vector<int>&vec){
    roomnum = vec.size() ;
    for ( int i = 0 ; i < vec.size() ; i++ ){
        rooms[i].room_NO = vec[i];
    }
    return true ;
}
```

8. int hotel::location_room

- 函数功能：找出某房间号的相对位置；
- 具体实现：依次判断即可；

```
int hotel::location_room ( int room_NO ){
    for ( int i = 0 ; i <roomnum ; i++ ){
        if ( rooms[i].room_NO == room_NO )
            return i ;
    }
    return -1 ;
}
```

9. bool hotel::do_request

- 函数功能：处理预约请求；
- 具体实现：

这个是最重要的函数！！用来处理各个请求：

a. **reserve预约**

首先检查日期是否合法；

接着检查房间号是否合法；

最后调用can_reserved函数检查该预约是否与其他的冲突，若不冲突，则添加预约信息；

b. **cancel取消预约**

首先检查日期是否合法；

接着检查房间号是否合法；

最后检查该房间是否有对应的预约信息，有则删除；

c. **reserveblock预约多个连续房间**

首先检查日期是否合法；

接着检查房间号是否合法；

然后判断这几个连续房间是否都满足预约要求，可以的话就添加预约信息；

判断房间连续这里有几个注意的地方！！！

i. 首先检查输入序列是否连续。e.g.宾馆初始化时房间号203,204,205,207,208……这种就不符合连续；

ii. 初始化宾馆时，注意楼层问题。e.g.101,102,103,104,201,202……顾客要预约五个连续房间，那么这种序列就不满足需要；

因此，在判断连续房间时，要注意房间号的数字问题；

d. **cancelblock取消多个连续房间的预约**

首先检查日期是否合法；

接着检查房间号是否合法；

最后检查这些房间是否有对应的预约信息，有则删除；

e. **reserveany预约多个房间，非连续**

首先检查日期是否合法；

接着检查房间号是否合法；

最后检查按照房间序列，依次调用can_reserved函数，若可以预约，则添加预约信息，直到房间数量满足为止；

f. **cancelany取消多个房间的预约**

首先检查日期是否合法；

接着检查房间号是否合法；

最后检查这些房间是否有对应的预约信息，有则删除；

g. **check检查预约信息**

调用search_by_name函数，查询某个客人的预约信息；

10. void read_hotel_msg

- 函数功能：读取输入文件中的房间信息；
- 具体实现：利用fscanf读取文件流，按照之前约定的测试文件格式来读取对应位置；

```
void read_hotel_msg(FILE *fp,class hotel *myhotel)
{
    fscanf(fp,"%d",&(myhotel->roomnum));
    for(int i=0;i<myhotel->roomnum;i++)
    {
        fscanf(fp,"%d",&(myhotel->rooms[i].room_NO));
        myhotel->rooms[i].res_num=0;
    }
}
```

11. void read_cust_msg

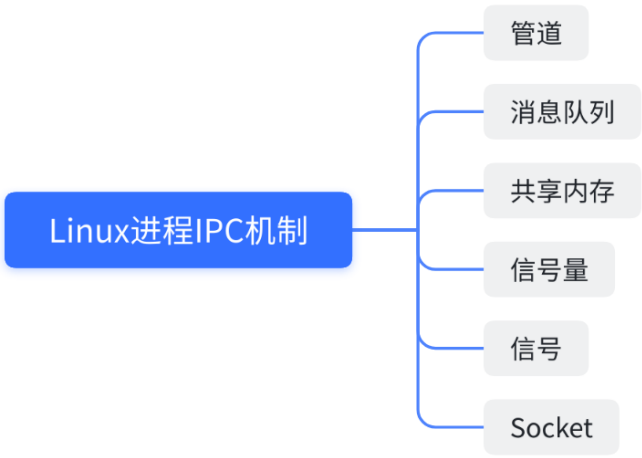
- 函数功能：读取输入文件中的顾客信息；
- 具体实现：利用fscanf读取文件流，按照之前约定的测试文件格式来读取对应位置；接着利用strcmp函数来匹配对应的预约请求；

```
void read_cust_msg(FILE *fp,int &cust_num, vector<struct customer> &cus_vec)
{
    fscanf(fp,"%d",&cust_num);//顾客数量
    char buf[30];
    struct customer cust;
    for(int i=1;i<=cust_num;i++)
    {
        cust.op_num=0;
        fscanf(fp,"%s",buf);//读取customer标识符
        fscanf(fp,"%s",buf);
        while(strstr(buf,"end") == NULL)
        {
            strcpy(cust.cus_op[cust.op_num].op_type,buf);
            if(strcmp(buf,"reserve")==0 || strcmp(buf,"cancel")==0)
            {
                fscanf(fp,"%d %d %d %d %d %s %d",&(cust.cus_op[cust.op_num].room_NO),&(cust.cus_op[cust.op_nu
                    &(cust.cus_op[cust.op_num].start_date.day),&(cust.cus_op[cust.op_num].days),cust.name,&(cu
            }
            else if(strcmp(buf,"reserveblock")==0 || strcmp(buf,"cancelblock")==0)
            {
                fscanf(fp,"%d %d %d %d %d %d %s %d",&cust.cus_op[cust.op_num].room_num,&cust.cus_op[cust.op_nu
                    &cust.cus_op[cust.op_num].start_date.day,&cust.cus_op[cust.op_num].days,cust.name,&cust.cu
            }
            else if(strcmp(buf,"reserveany")==0 || strcmp(buf,"cancelany")==0)
            {
                fscanf(fp,"%d %d %d %d %d %s %d",&cust.cus_op[cust.op_num].room_num,&cust.cus_op[cust.op_num]
                    &cust.cus_op[cust.op_num].start_date.day,&cust.cus_op[cust.op_num].days,cust.name,&cust.cu
            }
            else if(strcmp(buf,"check")==0)
            {
                fscanf(fp,"%s %d",cust.name,&cust.cus_op[cust.op_num].op_time);
            }
            else
            {
                cust.cus_op[cust.op_num].op_time=100;
                char c;
                fscanf(fp,"%c",&c);
                while(c!='\n')
                {
                    fscanf(fp,"%c",&c);
                }
            }

            cust.op_num++;
            fscanf(fp,"%s",buf);
        }
        cus_vec.push_back(cust);
    }
}
```

my_ipc.h

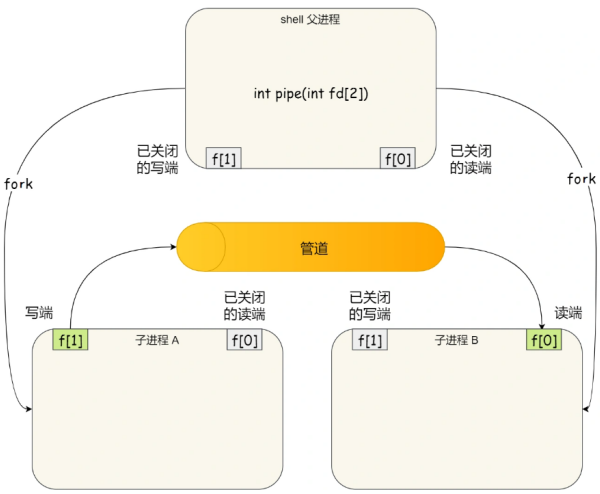
首先我们要了解linux进程IPC机制是怎么实现的；



如图，我们依次分析一下这些IPC机制：

1. 管道

所谓的管道，就是内核里面的一串缓存。从管道的一段写入的数据，实际上是缓存在内核中的，另一端读取，也就是从内核中读取这段数据。另外，管道传输的数据是无格式的流且大小受限。



对于匿名管道，它的通信范围是存在父子关系的进程。因为管道没有实体，也就是没有管道文件，只能通过 fork 来复制父进程 fd 文件描述符，来达到通信的目的。

对于命名管道，它可以在不相关的进程间也能相互通信。因为命令管道，提前创建了一个类型为管道的设备文件，在进程里只要使用这个设备文件，就可以相互通信。

不管是匿名管道还是命名管道，进程写入的数据都是缓存在内核中，另一个进程读取数据时候自然也是从内核中获取，同时通信数据都遵循**先进先出**原则，不支持 lseek 之类的文件定位操作。

2. 消息队列

管道的通信效率很低，不适合进程间频繁地交换数据，因此，产生了消息队列这一通信模式；

消息队列是保存在内核中的消息链表，在发送数据时，会分成一个一个独立的数据单元，也就是消息体（数据块），消息体是用户自定义的数据类型，消息的发送方和接收方要约定好消息体的数据类型，所以每个消息体都是固定大小的存储块，不像管道是无格式的字节流数据。如果进程从消息队列中读取了消息体，内核就会把这个消息体删除。

但消息队列也存在不足的点：**一是通信不及时，二是附件也有大小限制；**

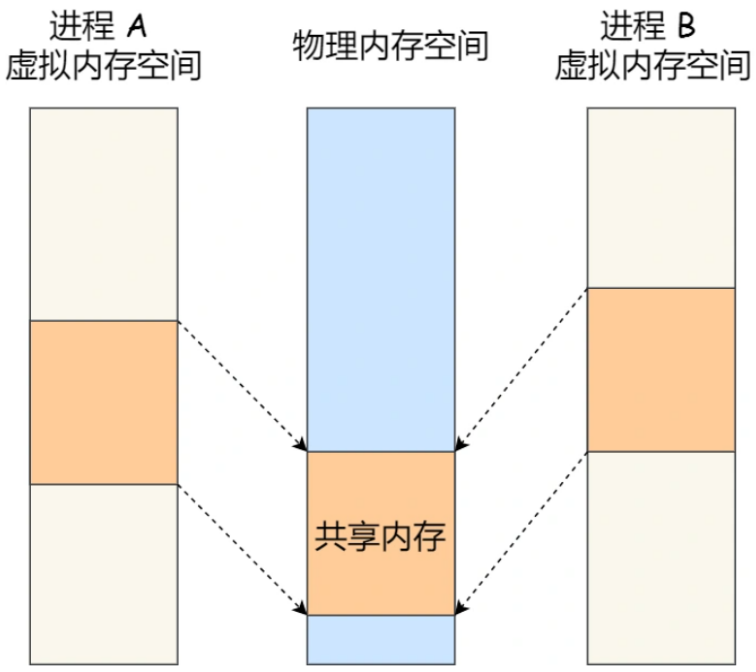
消息队列不适合比较大数据的传输，因为在内核中每个消息体都有一个最大长度的限制，同时所有队列所包含的全部消息体的总长度也是有上限。

消息队列通信过程中，存在用户态与内核态之间的数据拷贝开销，因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，同理另一进程读取内核中的消息数据时，会发生从内核态拷贝数据到用户态的过程。

3. 共享内存

消息队列的读取和写入的过程，都会有发生用户态与内核态之间的消息拷贝过程。而共享内存的方式，就很好的解决了这一问题。

共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中。这样这个进程写入的东西，另外一个进程马上就能看到了，都不需要拷贝来拷贝去，传来传去，大大提高了进程间通信的速度。



共享内存的实现机制

而本次实验，我们就用到了共享内存。

4. 信号量

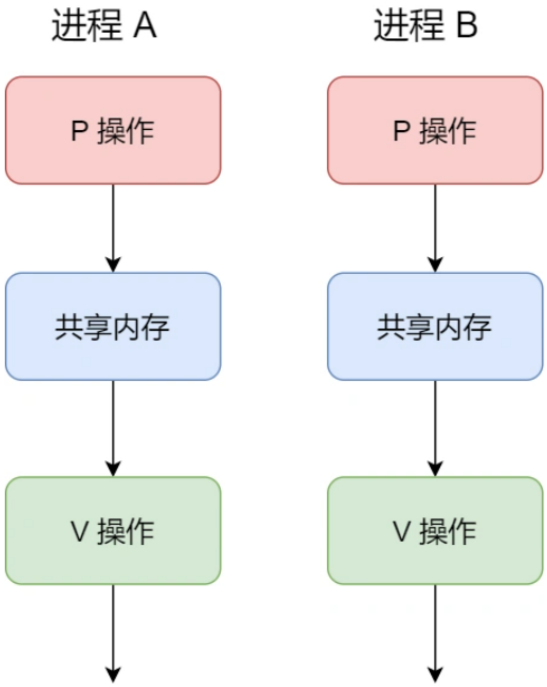
如果多个进程同时修改同一个共享内存，很有可能就冲突了。为了防止多进程竞争共享资源，而造成的数据错乱，所以需要保护机制，使得共享的资源，在任意时刻只能被一个进程访问。正好，信号量就实现了这一保护机制。

信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据。

信号量表示资源的数量，控制信号量的方式有两种原子操作：

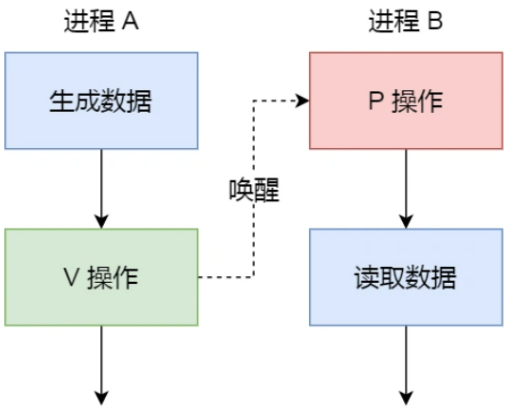
- 一个是 P 操作，这个操作会把信号量减去 1，相减后如果信号量 < 0，则表明资源已被占用，进程需阻塞等待；相减后如果信号量 >= 0，则表明还有资源可使用，进程可正常继续执行。
- 另一个是 V 操作，这个操作会把信号量加上 1，相加后如果信号量 <= 0，则表明当前有阻塞中的进程，于是会将该进程唤醒运行；相加后如果信号量 > 0，则表明当前没有阻塞中的进程；

我们首先看一下信号量实现进程间互斥：



- 进程 A 在访问共享内存前，先执行了 P 操作，由于信号量的初始值为 1，故在进程 A 执行 P 操作后信号量变为 0，表示共享资源可用，于是进程 A 就可以访问共享内存。
- 若此时，进程 B 也想访问共享内存，执行了 P 操作，结果信号量变为了 -1，这就意味着临界资源已被占用，因此进程 B 被阻塞。
- 直到进程 A 访问完共享内存，才会执行 V 操作，使得信号量恢复为 0，接着就会唤醒阻塞中的线程 B，使得进程 B 可以访问共享内存，最后完成共享内存的访问后，执行 V 操作，使信号量恢复到初始值 1。

再来看一下信号量实现进程间同步：



- 如果进程 B 比进程 A 先执行了，那么执行到 P 操作时，由于信号量初始值为 0，故信号量会变为 -1，表示进程 A 还没生产数据，于是进程 B 就阻塞等待；

- 接着，当进程 A 生产完数据后，执行了 V 操作，就会使得信号量变为 0，于是就会唤醒阻塞在 P 操作的进程 B；
- 最后，进程 B 被唤醒后，意味着进程 A 已经生产了数据，于是进程 B 就可以正常读取数据了。

本次实验，我们也用到了信号量；

5. **信号**

上面说的进程间通信，都是常规状态下的工作模式。**对于异常情况下的工作模式，就需要用「信号」的方式来通知进程。**

信号是进程间通信机制中唯一的异步通信机制，因为可以在任何时候发送信号给某一进程，一旦有信号产生，我们就有下面这几种，用户进程对信号的处理方式。

- 执行默认操作。Linux 对每种信号都规定了默认操作，例如，上面列表中的 SIGTERM 信号，就是终止进程的意思。
- 捕捉信号。我们可以为信号定义一个信号处理函数。当信号发生时，我们就执行相应的信号处理函数。
- 忽略信号。当我们不希望处理某些信号的时候，就可以忽略该信号，不做任何处理。有两个信号是应用进程无法捕捉和忽略的，即 `SIGKILL` 和 `SEGSTOP`，它们用于在任何时候中断或结束某一进程。

6. **Socket套接字**

前面提到的管道、消息队列、共享内存、信号量和信号都是在同一台主机上进行进程间通信，那要想跨网络与不同主机上的进程之间通信，就需要 Socket 通信了。

这个我在计算机网络课程实验中使用的已经非常多了，在此不做赘述；

接下来看一下my_ipc.h的实现：

首先定义一个联合体，用于之后的一些命令：

```
union semun
{   int val;                    //仅用于SETVAL命令
    struct semid_ds * buf;  //用于IPC_SET等命令
    ushort * array;             //用于SETALL等命令
};
```

定义信号量的一些相关函数；

定义共享内存的一些相关函数；

通过这些来实现linux进程IPC机制；

具体实现见my_ipc.c；

my_ipc.c

定义上述函数的具体实现；

1. int new_sem(char * path ,int id ,int sem_num ,int val)

- 函数功能：创建新信号量，并且初始化；
- 函数参数：

  path,id:用于生成IPC键；

  sem_num:指明信号灯集合中包含的信号灯数；

  val: 信号灯初值；
- 返回值：信号灯标识符；
- 具体实现：

  利用ftok函数将path和id转换成为一个key_t值：

```
ipc_key = ftok(path,id);
if( ipc_key == -1)
{
    perror("ftok error");
    exit(0);
}
```

  之后调用semget函数创建信号灯集合：

```
sem_id = semget(ipc_key,sem_num,IPC_CREAT|0666);
if(sem_id ==-1)
{
    perror("semget_IPC_CREAT error");
    exit(0);
}
```

  再使用semctl对信号灯集合进行控制：

```
for(i = 0 ; i < sem_num ; i++)
{
    if(semctl(sem_id,i,SETVAL,options) == -1)
    {
        perror("semctl_setval error");
        exit(0);
    }
}
return sem_id;
```

2. void down(int sem_id,int sem_NO)

- 函数功能：获取信号量，成功获取后信号量计数值减1，获取不成功进程进入睡眠状态；
- 函数参数：

  sem_id:信号灯标识符；

  sem_NO:信号灯号；
- 具体实现：调用semop函数，实现进程的wait操作；

```
void down(int sem_id,int sem_NO)
{
   struct sembuf op;
   op.sem_num = sem_NO;
   op.sem_op = -1;
   op.sem_flg = 0;
   if(semop(sem_id, &op, 1) == -1)
   {
     perror("semop_down error");
     exit(0);
   }
}
```

3. void up(int sem_id,int sem_NO)

- 函数功能：释放信号量，信号量计数值加1，此时在此信号量上等待的进程可以获得它；
- 函数参数：

  sem_id:信号灯标识符；

  sem_NO:信号灯号；
- 具体实现：同理，调用semop函数，实现进程的signal操作；

```
void up(int sem_id,int sem_NO)
{
    struct sembuf op;
   op.sem_num = sem_NO;
   op.sem_op = 1;
   op.sem_flg = 0;
   if(semop(sem_id, &op, 1) == -1)
   {
     perror("semop_up error");
     exit(0);
   }
}
```

4. void clear_sem(int sem_id)

- 函数功能：清除信号量；
- 函数参数：

  sem_id:信号灯标识符；
- 具体实现：调用semctl实现IPC_RMID，删除信号灯集合；

```
void clear_sem(int sem_id)
{

    if(semctl(sem_id,0,IPC_RMID) == -1)
        perror("semctl_IPC_RMID error");

}
```

5. int new_shm(char *path,int id,size_t shmsize)

- 函数功能：创建共享内存区；
- 函数参数：

  path,id:用于生成IPC键；

  shmsize：共享内存区的大小，以字节为单位；
- 返回值：内存区标识符；
- 具体实现：

  与new_sem函数的实现方式类似；

  ftok得到key_t值：

```
ipc_key = ftok(path,id);
if( ipc_key == -1)
{
    perror("ftok error");
    exit(0);
}
```

shmget创建共享内存区：

```
shm_id = shmget(ipc_key,shmsize,IPC_CREAT|0666);
if(shm_id == -1)
{
    perror("shmget_IPC_CREAT error");
    exit(0);
}
```

my_shmat链接共享内存区，my_shmdt切断共享内存区：

```
p=(char *)my_shmat(shm_id);
p[0]='\0';
my_shmdt((const void *)p);

return shm_id;
```

6. void * my_shmat(int shm_id)

- 函数功能：将一个打开的共享内存区附接到调用进程的地址空间；
- 函数参数：

  shm_id:内存区标识符；
- 返回值：该内存映射区起始地址；
- 具体实现：调用shmat函数链接共享内存区；

```
void* my_shmat(int shm_id)
{
    void *p;
    char *q;
    p=shmat(shm_id,NULL,0);
    q=(char *)p;
    if( ((long)q) == -1 )
    {
        perror("shmat error");
        exit(0);
    }
    return p;
}
```

7. void my_shmdt(const void *addr)

- 函数功能：切断共享内存区与本进程地址空间的联系；
- 函数参数：

  addr:内存映射区起始地址；
- 具体实现：调用shmdt函数切断链接；

```
void my_shmdt(const void *addr)
{
    if(shmdt(addr) == -1)
    {
        perror("shmdt error");
        exit(0);
    }
}
```

8. void clear_shm(int shm_id)

- 函数功能：清除共享内存；
- 函数参数：

  shmid:内存区标识符；
- 具体实现：调用shmctl函数清除共享内存区；

```
void clear_shm(int shm_id)
{
    if(shmctl(shm_id,IPC_RMID,NULL) == -1)
    {
        perror("shmctl_IPC_RMID error");
        exit(0);
    }
}
```

main.c

创建共享内存区和信号灯，模拟宾馆预约过程，读取文件，调用相关函数完成相应的功能，最后输出相关信息即可；

```
sem_hotel=new_sem("./" ,770 ,1 ,1);
shm_hotel=new_shm("./",111, sizeof(class hotel));
```

创建信号量和共享内存区

```
FILE *fp;
if((fp=fopen(argv[1],"r"))==NULL)
{
    printf("Can not open file [%s]\n",argv[1]);
    exit(0);
}

myhotel=(class hotel *)my_shmat( shm_hotel ); //将hotel信息映射到共享内存空间
read_hotel_msg(fp,myhotel);

my_shmdt((const void *)myhotel);

read_cust_msg(fp,cus_num, cus_vec);
```

读取测试文件内容

接着要模拟预约过程，这一步需要用到fork操作；

有几个顾客，就要fork派生出几个子进程；

子进程来各自执行wait和signal操作，并利用信号量来保证每个子进程操作的原子性；

最后父进程等待所有子进程执行完毕，释放信号量和共享内存区：

```
for(j=1; j<=cus_num; j++)  //等待所有子进程结束;
    wait(NULL);

myhotel=(class hotel *)my_shmat( shm_hotel );
myhotel->print();
my_shmdt((const void *)myhotel);
//删除信号量和共享内存 ;
clear_sem(sem_hotel);
clear_shm(shm_hotel);
```

makefile

借鉴上学期操作系统实验的经验，本次实验我同样编写了一个makefile文件用来统一编译所有的代码文件，并在最后删掉中间产生的临时文件；

```
wzh: main.o my_ipc.o hotel.o
    g++ -o wzh main.o my_ipc.o hotel.o

main.o: main.c my_ipc.h hotel.h
    g++ -c main.c my_ipc.h hotel.h

my_ipc.o: my_ipc.c my_ipc.h
    g++ -c my_ipc.c my_ipc.h

hotel.o: hotel.c hotel.h
    g++ -c hotel.c hotel.h

.PHONY : clean
clean :
    -rm wzh main.o my_ipc.o hotel.o my_ipc.h.gch hotel.h.gch
```

五、运行方法

输入make命令进行编译；

接着输入./wzh test.txt；

（test.txt表示测试文件）

六、测试

测试文件：

```
9
101
102
103
104
201
202
203
301
302
4
customer
reserve 101 2022 12 31 2 Tom 100
reserve 102 2022 12 31 2 Tom 200
check Tom 90
cancel 101 2022 12 31 2 Tom 150
end.

customer
reserveblock 2 102 2022 12 29 5 Jack 120
reserveany 2 2022 5 1 3 Jack 200
check Jack 100
cancelany 1 2022 5 1 3 Jack 130
check Jack 100
end.

customer
check Jony 100
end.

customer
reserveblock 3 201 2022 6 3 4 Mark 400
reserve 101 2022 5 3 4 Mark 150
check Mark 100
end.
```

测试文件的设计主要考虑到以下几点：

1. 跨年预约的日期显示问题；
2. 预约冲突的处理；
3. 连续预约房间不够；
4. 预约多个房间的失败情况；

基于此，我设计了一份自以为比较有代表性的测试文件。

```
wzh@wzh-virtual-machine:~/Desktop/test/wzh$ ./wzh 1.txt
Begin------------------------------
Tom reserve NO.101 from 2022.12.31 for 2days **Success
Jack reserveblock 2rooms(NO.102) from 2022.12.29 for 5days **Success
Tom reserve NO.102 from 2022.12.31 for 2days **Fail:room has reserved
Mark reserveblock 3rooms(NO.201) from 2022.6.3 for 4days **Success
Jony check result: no reserve information
Jack reserveany 2rooms(NO.102) from 2022.5.1 for 3days **Success
Tom check result: reserve NO.101 from 2022.12.31 for 2days
Mark reserve NO.101 from 2022.5.3 for 4days **Fail:room has reserved
Jack check result: reserve NO.101 from 2022.5.1 for 3days
Jack check result: reserve NO.102 from 2022.5.1 for 3days
Jack check result: reserve NO.102 from 2022.12.29 for 5days
Jack check result: reserve NO.103 from 2022.12.29 for 5days
Tom cancel NO.101 from 2022.12.31 for 2days **Success
Mark check result: reserve NO.201 from 2022.6.3 for 4days
Mark check result: reserve NO.202 from 2022.6.3 for 4days
Mark check result: reserve NO.203 from 2022.6.3 for 4days
Jack cancelany 1rooms(NO.101) from 2022.5.1 for 3days **Success
Jack check result: reserve NO.102 from 2022.5.1 for 3days
Jack check result: reserve NO.102 from 2022.12.29 for 5days
Jack check result: reserve NO.103 from 2022.12.29 for 5days
```

```
-----------------hotel information---------------
room NO.101:---
free
room NO.102:---
name:Jack  start_date:2022.5.1  end_date:2022.5.3  days:3
name:Jack  start_date:2022.12.29  end_date:2023.1.3  days:5
room NO.103:---
name:Jack  start_date:2022.12.29  end_date:2023.1.3  days:5
room NO.104:---
free
room NO.201:---
name:Mark  start_date:2022.6.3  end_date:2022.6.6  days:4
room NO.202:---
name:Mark  start_date:2022.6.3  end_date:2022.6.6  days:4
room NO.203:---
name:Mark  start_date:2022.6.3  end_date:2022.6.6  days:4
room NO.301:---
free
room NO.302:---
free
End------------------------------
```

Tom预约101号，2022.12.31-2022.1.1，成功！

Jack预约102号开始的连续两间房（102,103），2022.12.29-2022.1.2，成功！

注意，Tom的第二条预约信息由于耗时较长，所以排在Jack第一条预约的后面；

Tom预约102号，2022.12.31-2022.1.1，由于此时102号在2022.12.29-2023.1.2期间处于被预约状态，所以本次预约失败！

Tom取消101号的预约，成功！

……

Mark连续预约3间房，201,202,203,预约日期2022.6.3-2022.6.6，成功！

……

综上，经过仔细比对，结果正确！！

七、实验总结

本次实验复杂度很高，也很难；

首先就是要模拟一个宾馆预约系统，这种模拟系统的项目一直是我的弱项，有点写不出这种代码工程量特别大的项目；

另外就是要正确使用linux进程IPC机制，将多进程融入这个庞大的宾馆预约系统中，使其具有可行性，这也产生了一些困难；

实验中遇到的主要问题如下：

a. 整体无思路

这里我是在github上搜了很多关于linux下进程IPC机制的应用，其中有一个关于模拟银行存款的项目对我帮助最大；

整体框架采用的就是这个项目https://github.com/ftylove/IPC；

b. 共享内存不会实现

看linux源码，上网查找各种资料自学；

c. linux系统的不熟悉

硬着头皮写，写着写着就熟练了；

d. makefile文件的编写

这个是寻求室友的帮助的，看他直接一步就编译好了，我不会写，请教的他；

e. 测试文件的选取

多写，多想，最后选了一个最合适的比较有代表性的；

f. 共享内存溢出

这个没有解决，我的问题，只能减少房间数和操作数；

g. 没有做出更好的优化

数组可以换成树结构，来实现快速插入和删除，同时又不至于让查询变得太慢；

查找时也可以采用二分法等快速查询方法，不用遍历；

大学这三年来也写过不少大作业了，代码能力确实得到了锻炼；

写完初版代码后，运行的过程中又产生了无数的错误，当时真的急得焦头烂额以至于忘记记录那些代码错误了；

最后对于各个功能的一步步调试也是非常的占用时间；

不管怎么说，结果是好的，陈老师上课讲的很清晰，对我的帮助是非常大的；

成功完成了本次试验，我为自己鼓掌，也很感谢老师的精彩授课，受益匪浅，获益良多！！！！