山东大学 计算机科学与技术 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201900161140 | 姓名：张文浩 | | 班级： 201900161140 |
| 实验题目：进程互斥实验 | | | |
| 实验学时：2 | | 实验日期： 5.10 | |
| 实验目的：  进一步研究和实践操作系统中关于并发进程同步与互斥操作的一些经典问题 的解法，加深对于非对称性互斥问题有关概念的理解。观察和体验非对称性互 斥问题的并发控制方法。进一步了解 Linux 系统中 IPC 进程同步工具的用法，训 练解决对该类问题的实际编程、调试和分析问题的能力。 | | | |
| 硬件环境：  **自带的笔记本，win10** | | | |
| 软件环境：vmware的ubuntu | | | |
| 实验步骤与内容：   1. 根据实验指导书完成实例实验。 2. 运行实例实验：    1. 启动control。先启动两个reader，再启动一个writer。根据下图可以看出，在启动reader后，reader一直在缓冲区中读取AAAAAAAA。     在启动writer后，writer每隔8秒写一次缓冲区，将缓冲区内的字母加一，两个reader分别按照10秒和1秒的间隔在缓冲区中度数据，根据control窗口的结果可以看出，实现了互斥操作。且在reader和writer同时提出请求时，先相应reader的请求，符合实验规定。     * 1. 再次运行实例实验，这次启动两个reader和两个writer，并按照不同的顺序执行，根据下图实验结果可以看出，也满足了操作之间互斥的要求。     题目：更改上面程序产生饥饿现象： 在control文件中修改，再收到写请求后不阻塞等待读请求完成，而是直接开启下一次循环，也就是说如果一直有读请求在读，写请求就永远不会得到相应。    运行结果如下：    可以看出reader速度设置为1，writer速度设置为5，因为一直在读，所以写请求一直得不到相应，产生饥饿现象。   1. 独立实验   理发店问题:假设理发店的理发室中有 3 个理发椅子和 3 个理发师,有一个可容  纳 4 个顾客坐等理发的沙发。此外还有一间等候室,可容纳 13 位顾客等候进入理发  室。顾客如果发现理发店中顾客已满(超过 20 人)  ,就不进入理发店。  在理发店内,理发师一旦有空就为坐在沙发上等待时间最长的顾客理发,同时空  出的沙发让在等候室中等待时间最长的的顾客就坐。顾客理完发后,可向任何一位  理发师付款。但理发店只有一本现金登记册,在任一时刻只能记录一个顾客的付款。  理发师在没有顾客的时候就坐在理发椅子上睡眠。理发师的时间就用在理发、收款、  睡眠上。  思路：  采用一个关于账本的信号量s\_account，实现账本的互斥。  采用一个顾客的信号量s\_customer，实现顾客到来的同步。  还需要一个沙发消息队列和等候室的消息队列，模拟顾客进入沙发和等候室等待的过程。  在理发师barber进程中：  先以阻塞的方式尝试在沙发队列中接受顾客，如果沙发上有顾客，就进行理发，并唤醒顾客进程，让下一个顾客进入沙发。  如果沙发上没有顾客，理发师就睡觉。  在顾客customer进程中：  先查看沙发上（即沙发消息队列中）顾客数量，如果沙发上的顾客数量小于4（即沙发没坐满），就让等候室的一位顾客加入到沙发中，如果等候室没顾客，就让一位新顾客加入到沙发中。如果等候室人满了，就down顾客的同步变量，等待barber进程理完发利用up唤醒。  实验结果如下：  在本次运行中，先启动barber，因为一开始没有顾客，三个理发师都在睡觉。然后启动customer，可以看到一直顺利进行，一直到第14个顾客来的时候沙发坐满了，就进入等候室等待，随后又进入沙发。因为本次运行中理发的速度和顾客到来的速度相差不大，所以没有出现等候室慢的情况。    再次运行独立实验，这次运行先启动customer，让顾客填等候室然后阻塞。可以看到在第18个顾客想要进入时，因为等候室慢无法进入。    随后启动barber进程，可以看到随着理发的开始，等候室的顾客开始逐渐进入沙发，而customer进程被barber进程唤醒，使得新的顾客开始进入等候室。 | | | |
| 结论分析与体会：  通过本次实验，我进一步研究和实践了操作系统中关于并发进程同步与互斥操作的一些经典问题的解法，加深了对于非对称性互斥问题有关概念的理解。观察和体验了非对称性互斥问题的并发控制方法。进一步了解了 Linux 系统中 IPC 进程同步工具的用法，训练了解决对该类问题的实际编程、调试和分析问题的能力。  (1)非对称性互斥是指并发进程有且仅有一个可以执行某一段代码（称为临界区），具体是那个进程进入临界区是有临界区之前的代码（称为进入区）决定的，当进入临界区的调度机制是非对称，即某些进程具有优先进入的权利称为非对称的，例如此时有一个理发师从沙发叫走了一个顾客，需要一个顾客从等候室进入沙发室，此时恰巧新来了以顾客，此时顾客进程和理发师进程都想要进入临界区（修改或查询等候室顾客数目），但按照正常人的思维，肯定是理发师先访问临界区，修改等候室顾客数目，然后是新来顾客访问临界区，查询是否可以进入理发店。  （2）进程饥饿指当等待时间给进程推进和响应带来明显影响称为进程饥饿。 产生饥饿的主要原因是：在一个动态系统中，对于每类系统资源，操作系统需要确定一个分配策略，当多个进程同时申请某类资源时，由分配策略确定资源分配给进程的次序。有时资源分配策略可能是不公平的，即不能保证等待时间上界的存在。在这种情况下，即使系统没有发生死锁，某些进程也可能会长时间等待．当等待时间给进程推进和响应带来明显影响时，称发生了进程饥饿，当饥饿到一定程度的进程所赋予的任务即使完成也不再具有实际意义时称该进程被饿死。  （3）本实验的饥饿现象表现在当顾客数目较少的情况下某一理发师进程可能一直被阻塞，即其余理发师持续服务。   （4）解决进程饥饿现象的唯一方法是修改进入临界区进程选择机制，打破不公平调度机制，使得想进入临界区的进程可以在有限等待中进入临界区。  （5）linux系统中进程可以通过共享内存和消息传递机制通信，通过消息传递队列，具体实现是发送进程发送信息给内核，然后内核实现了转发功能，接受进程从内核那里接收到发送进程所发送的信息。适合传递少量信息，不适合传递大量信息，大量信息适合用共享内存机制传递。 | | | |

附源代码

**Ipc.h:**

/\*

\* Filename : ipc.h

\* copyright : (C) 2006 by zhonghonglie

\* Function : 声明 IPC 机制的函数原型和全局变量

\*/

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <sys/msg.h>

#include <unistd.h>

#include <time.h>

#define BUFSZ 256

#define MAXVAL 100

#define STRSIZ 8

#define WRITERQUEST 1 //写请求标识

#define READERQUEST 2 //读请求标识

#define FINISHED 3 //读写完成标识

#define SOFA 4 //沙发的数量

#define WAIT 5 //等候室

/\*信号灯控制用的共同体\*/

typedef union semuns {

int val;

} Sem\_uns;

/\* 消息结构体\*/

typedef struct msgbuf {

long mtype;

int mid;

} Msg\_buf;

key\_t buff\_key;

int buff\_num;

char \*buff\_ptr;

int shm\_flg;

int quest\_flg;

key\_t quest\_key;

int quest\_id;

int respond\_flg;

key\_t respond\_key;

int respond\_id;

int get\_ipc\_id(char \*proc\_file,key\_t key);

char \*set\_shm(key\_t shm\_key,int shm\_num,int shm\_flag);

int set\_msq(key\_t msq\_key,int msq\_flag);

int set\_sem(key\_t sem\_key,int sem\_val,int sem\_flag);

int down(int sem\_id);

int up(int sem\_id);

int sem\_flg;

key\_t s\_account\_key;

int s\_account\_val;

int s\_account\_sem;

key\_t s\_customer\_key;

int s\_customer\_val;

int s\_customer\_sem;

int q\_flg;

key\_t q\_sofa\_key;

int q\_sofa\_id;

key\_t q\_wait\_key;

int q\_wait\_id;

**ipc.c:**

#include "ipc.h"

/\*

\* get\_ipc\_id() 从/proc/sysvipc/文件系统中获取 IPC 的 id 号

\* pfile: 对应/proc/sysvipc/目录中的 IPC 文件分别为

\* msg-消息队列,sem-信号量,shm-共享内存

\* key: 对应要获取的 IPC 的 id 号的键值

\*/

int get\_ipc\_id(char \*proc\_file,key\_t key)

{

FILE \*pf;

int i,j;

char line[BUFSZ],colum[BUFSZ];

if((pf = fopen(proc\_file,"r")) == NULL){

perror("Proc file not open");

exit(EXIT\_FAILURE);

}

fgets(line, BUFSZ,pf);

while(!feof(pf)){

i = j = 0;

fgets(line, BUFSZ,pf);

while(line[i] == ' ') i++;

while(line[i] !=' ') colum[j++] = line[i++];

colum[j] = '\0';

if(atoi(colum) != key) continue;

j=0;

while(line[i] == ' ') i++;

while(line[i] !=' ') colum[j++] = line[i++];

colum[j] = '\0';

i = atoi(colum);

fclose(pf);

return i;

}

fclose(pf);

return -1;

}

/\*

\* 信号灯上的 down/up 操作

\* semid:信号灯数组标识符

\* semnum:信号灯数组下标

\* buf:操作信号灯的结构

\*/

int down(int sem\_id)

{

struct sembuf buf;

buf.sem\_op = -1;

buf.sem\_num = 0;

buf.sem\_flg = SEM\_UNDO;

if((semop(sem\_id,&buf,1)) <0) {

perror("down error ");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

int up(int sem\_id)

{

struct sembuf buf;

buf.sem\_op = 1;

buf.sem\_num = 0;

buf.sem\_flg = SEM\_UNDO;

if((semop(sem\_id,&buf,1)) <0) {

perror("up error ");

exit(EXIT\_FAILURE);

}

return EXIT\_SUCCESS;

}

/\*

\* set\_sem 函数建立一个具有 n 个信号灯的信号量

\* 如果建立成功，返回 一个信号灯数组的标识符 sem\_id

\* 输入参数：

\* sem\_key 信号灯数组的键值

\* sem\_val 信号灯数组中信号灯的个数

\* sem\_flag 信号等数组的存取权限

\*/

int set\_sem(key\_t sem\_key,int sem\_val,int sem\_flg)

{

int sem\_id;

Sem\_uns sem\_arg;

//测试由 sem\_key 标识的信号灯数组是否已经建立

if((sem\_id = get\_ipc\_id("/proc/sysvipc/sem",sem\_key)) < 0 )

{

//semget 新建一个信号灯,其标号返回到 sem\_id

if((sem\_id = semget(sem\_key,1,sem\_flg)) < 0)

{

perror("semaphore create error");

exit(EXIT\_FAILURE);

}

//设置信号灯的初值

sem\_arg.val = sem\_val;

if(semctl(sem\_id,0,SETVAL,sem\_arg) <0)

{

perror("semaphore set error");

exit(EXIT\_FAILURE);

}

}

return sem\_id;

}

/\*

\* set\_shm 函数建立一个具有 n 个字节 的共享内存区

\* 如果建立成功，返回 一个指向该内存区首地址的指针 shm\_buf

\* 输入参数：

\* shm\_key 共享内存的键值

\* shm\_val 共享内存字节的长度

\* shm\_flag 共享内存的存取权限

\*/

char \* set\_shm(key\_t shm\_key,int shm\_num,int shm\_flg)

{

int i,shm\_id;

char \* shm\_buf;

//测试由 shm\_key 标识的共享内存区是否已经建立

if((shm\_id = get\_ipc\_id("/proc/sysvipc/shm",shm\_key)) < 0 )

{

//shmget 新建 一个长度为 shm\_num 字节的共享内存,其标号返回到 shm\_id

if((shm\_id = shmget(shm\_key,shm\_num,shm\_flg)) <0)

{

perror("shareMemory set error");

exit(EXIT\_FAILURE);

}

//shmat 将由 shm\_id 标识的共享内存附加给指针 shm\_buf

if((shm\_buf = (char \*)shmat(shm\_id,0,0)) < (char \*)0)

{

perror("get shareMemory error");

exit(EXIT\_FAILURE);

}

for(i=0; i<shm\_num; i++) shm\_buf[i] = 0; //初始为 0

}

//shm\_key 标识的共享内存区已经建立,将由 shm\_id 标识的共享内存附加给指

//针 shm\_buf

if((shm\_buf = (char \*)shmat(shm\_id,0,0)) < (char \*)0)

{

perror("get shareMemory error");

exit(EXIT\_FAILURE);

}

return shm\_buf;

}

/\*

\* set\_msq 函数建立一个消息队列

\* 如果建立成功，返回 一个消息队列的标识符 msq\_id

\* 输入参数：

\* msq\_key 消息队列的键值

\* msq\_flag 消息队列的存取权限

\*/

int set\_msq(key\_t msq\_key,int msq\_flg)

{

int msq\_id;

//测试由 msq\_key 标识的消息队列是否已经建立

if((msq\_id = get\_ipc\_id("/proc/sysvipc/msg",msq\_key)) < 0 )

{

//msgget 新建一个消息队列,其标号返回到 msq\_id

if((msq\_id = msgget(msq\_key,msq\_flg)) < 0)

{

perror("messageQueue set error");

exit(EXIT\_FAILURE);

}

}

return msq\_id;

}

**barber.c：**

#include "ipc.h"

int main (int argc, char \*argv[])

{

int rate;

//可在在命令行第一参数指定一个理发速度

if(argv[1] != NULL) rate = atoi(argv[1]);

else rate = 5; //不指定为 5 秒

struct msqid\_ds msg\_sofa\_info;

Msg\_buf msg\_arg;

sem\_flg = IPC\_CREAT | 0644;

//建立一个互斥帐本信号量

s\_account\_key=100;

s\_account\_val=1;

s\_account\_sem=set\_sem(s\_account\_key,s\_account\_val,sem\_flg);

//建立一个同步顾客信号量

s\_customer\_key=200;

s\_customer\_val=0;

s\_customer\_sem=set\_sem(s\_customer\_key,s\_customer\_val,sem\_flg);

//建立沙发消息队列

q\_flg=IPC\_CREAT | 0644;

q\_sofa\_key=300;

q\_sofa\_id=set\_msq(q\_sofa\_key,q\_flg);

//建立等候室消息队列

q\_flg=IPC\_CREAT | 0644;

q\_wait\_key=400;

q\_wait\_id=set\_msq(q\_wait\_key,q\_flg);

//建立 3 个理发师进程;

int pid[3];

int i;

for(i=0; i<3; i++)

{

pid[i]=fork();

if(pid[i]==0)

{

while(1)

{

//查看沙发上顾客的人数

msgctl(q\_sofa\_id, IPC\_STAT, &msg\_sofa\_info);

//如果沙发上没人理发师就睡觉

if(msg\_sofa\_info.msg\_qnum==0)

printf("%d 号理发师睡眠\n",getpid());

//以阻塞方式从沙发队列接收一条消息

msgrcv(q\_sofa\_id,&msg\_arg,sizeof(msg\_arg),SOFA,0);

//唤醒顾客进程(让下一顾客坐入沙发)

up(s\_customer\_sem);

printf("%d 号理发师开始给 %d 号顾客理发\n",getpid(),msg\_arg.mid);

sleep(rate);

printf("%d 号理发师给 %d 号顾客理发完成\n",getpid(),msg\_arg.mid);

//互斥使用记账本

down(s\_account\_sem);

printf("%d 号理发师和 %d 号顾客使用记账本\n",getpid(),msg\_arg.mid);

up(s\_account\_sem);

}

}

}

return EXIT\_SUCCESS;

}

**Customer.c:**

#include "ipc.h"

int main(int argc, char \*argv[])

{

int rate;

//可在在命令行第一参数指定一个顾客到达速度

if(argv[1] != NULL) rate = atoi(argv[1]);

else rate = 1; //不指定为 1 秒

srand ((unsigned) time(NULL));

Msg\_buf msg\_arg;

struct msqid\_ds msg\_sofa\_info; //沙发上的顾客数量

struct msqid\_ds msg\_wait\_info; //等候室中的顾客数量

sem\_flg = IPC\_CREAT | 0644;

//建立一个互斥帐本信号量

s\_account\_key = 100;

s\_account\_val = 1; //初始值为1，说明开始的时候记账本空着，可以被使用

s\_account\_sem = set\_sem(s\_account\_key, s\_account\_val, sem\_flg);

//建立一个同步顾客信号量

s\_customer\_key = 200;

s\_customer\_val = 0; //初始值为0，说明一开始没有顾客

s\_customer\_sem = set\_sem(s\_customer\_key, s\_customer\_val, sem\_flg);

//建立一个沙发消息队列

q\_flg = IPC\_CREAT | 0644;

q\_sofa\_key = 300;

q\_sofa\_id = set\_msq(q\_sofa\_key, q\_flg);

//建立一个等候室消息队列

q\_wait\_key = 400;

q\_wait\_id = set\_msq(q\_wait\_key, q\_flg);

int customerNumber = 1;

while (1){

msgctl(q\_sofa\_id, IPC\_STAT, &msg\_sofa\_info);

//沙发上的顾客数量小于4个，说明没坐满，等待顾客进入的消息

if (msg\_sofa\_info.msg\_qnum < 4)

{

quest\_flg = IPC\_NOWAIT; //以非阻塞方式接收消息

//以非阻塞的方式在等候室队接收一条消息（有顾客/没顾客）

if (msgrcv(q\_wait\_id, &msg\_arg,sizeof(msg\_arg), WAIT, quest\_flg)>= 0)

{

msg\_arg.mtype = SOFA;

printf("%d号新顾客坐入沙发\n", msg\_arg.mid);

msgsnd(q\_sofa\_id, &msg\_arg, sizeof(msg\_arg), IPC\_NOWAIT);

}

//等候室没顾客，新来的顾客直接进入沙发队列

else

{

msg\_arg.mtype = SOFA;

msg\_arg.mid = customerNumber;

customerNumber++;

printf("%d号新顾客坐入沙发\n", msg\_arg.mid);

msgsnd(q\_sofa\_id, &msg\_arg, sizeof(msg\_arg),IPC\_NOWAIT);

}

}

//沙发坐满了

else

{

//读取等候室顾客的数量

msgctl(q\_wait\_id, IPC\_STAT, &msg\_wait\_info);

//等候室没坐满，新顾客可以进入等候室

if (msg\_wait\_info.msg\_qnum < 13)

{

msg\_arg.mtype = WAIT;

msg\_arg.mid = customerNumber;

printf("沙发座满,%d号新顾客进入等候室\n",customerNumber);

customerNumber++;

msgsnd(q\_wait\_id, &msg\_arg, sizeof(msg\_arg), IPC\_NOWAIT);

}

else

{

printf("等候室满,%d号新顾客没有进入理发店\n", customerNumber);

down(s\_customer\_sem);

}

}

//用进程休眠一个随机时间模拟顾客到达的时间间隔。

sleep(rate);

}

return EXIT\_SUCCESS;

}