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:201800130005	姓名：张畅	班级：2018 级计科 3 班
实验题目：进程的互斥实验		
实验学时：2	实验日期：2020.12.2	

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。

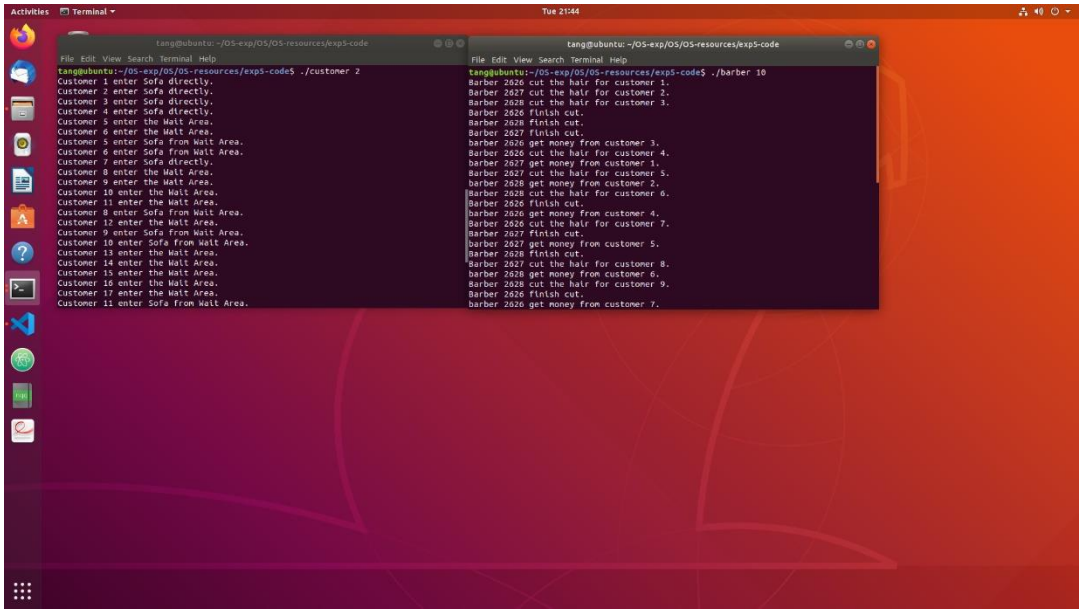
实验过程中遇到和解决的问题：

（记录实验过程中遇到的问题，以及解决过程和实验结果。可以适当配以关键代码辅助说明，但不要大段贴代码。）

在实验过程中，对题意中的结账这一概念的理解存在一定的偏差，这种偏差会导致运行速度的降低，在实验的过程中与助教交流了对结账过程的理解，完善了这一过程的程序，使得整个系统效率更高。

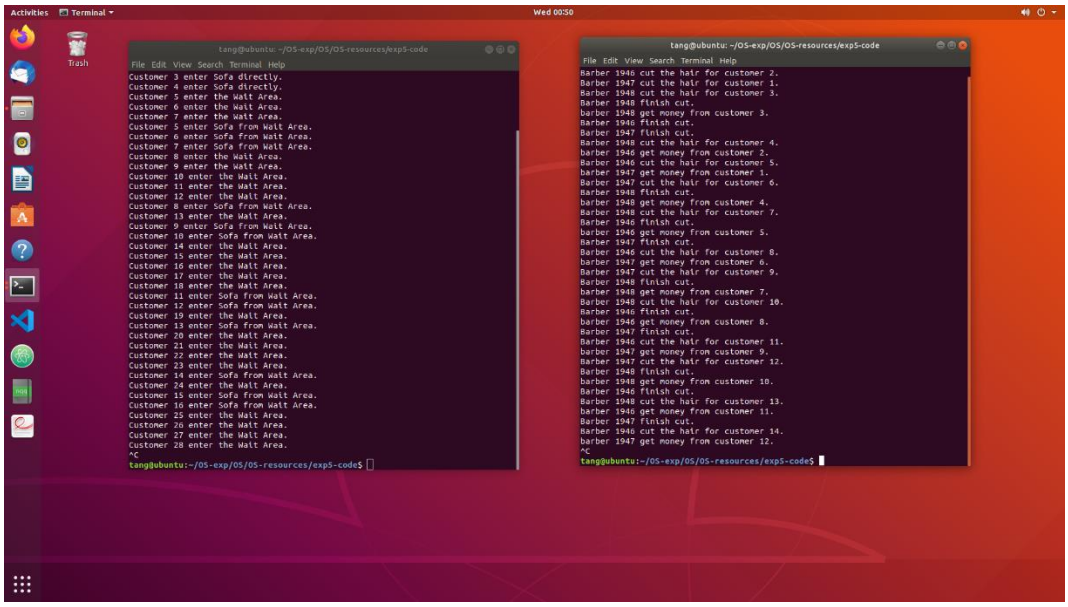
开始对系统的构思是每次剪完发之后，该理发师就直接为该顾客结账，但是一旦有顾客在结账，就会导致无意义的忙等效率低下。

修改之后，加入了一个结账的消息队列，这样在剪发结束后，理发师就可以去剪其他顾客的头发。但是经过测试这种设计仍然会导致忙等。



经过思考和与助教的交流，发现这种忙等的情况来源于如果三个顾客同时付款，由于系统优先考虑付款，就会到账两个理发师在等待而不是剪发。修改了理发

师的考虑顺序，改为在账本被占用时，不去等待而是剪发，经过测试，这种系统拥有较高的效率，比较完善：



实验步骤与内容：

1、熟悉基本的同步互斥调用方法

在上一个同步调用的实验中，已经基本熟悉了解决同步互斥问题的三种数据结构的：共享内存、信号灯和消息队列。在本实验中比较重要的是信号灯与消息队列的使用，对上述两种知识的使用进行了复习。

2、调试并且测试样例程序

为了更加全面、清晰地了解样例程序的逻辑结构，同时熟悉互斥问题的使用，对实例程序进行了分析。

算法中有一控制进程，带有 3 个不同类型的消息信箱，它们分别是：读请求信箱、写请求信箱和操作完成信箱。

读者：在访问邻接资源之前需要先向控制进程发送读请求信息

写者：在访问邻接资源之前需要先向控制进程发送写请求信息

两者都只有得到允许的消息之后才可以进入临界区域进行读写，读写完成之后还需要想控制进程发送操作完成消息。

控制进程使用一个变量 count 控制读写者互斥的访问临界资源并允许写者优先。count 的初值：需要一个比最大读者数还要大的数，本例取值为 100。当 count 大于 0 时说明没有新的读写请求，控制进程接收读写者新的请求：

如果收到写者完成消息，count 的值加 100；如果收到读者完成消息，对 count 的值加 1。如果收到写者请求消息， count 的值减 100；如果收到读者请求消息，对 count 的值减 1。

状态说明：

当 count 等于 0 时说明写者正在写，控制进程等待写者完成后再次令 count

的值等于 100。当 count 小于 100 时说明读者正在读，控制进程等待读者完成后对 count 的值加 1。

状态转换：

+ count>0 ---- 读状态或者空闲状态
  + 读申请：可以直接读
  + 写申请：等待所有读进程 FINISHED，直到 count=100，可以写
  + 完成：收到 FINISHED，只能是读结束，count++
+ count=0 ---- 写状态
  + 读申请：只能读，收到 FINISHED，只能是写结束，count=100，可以读
  + **写申请：不能写**，不用考虑
  + 完成：只能是写，不用考虑

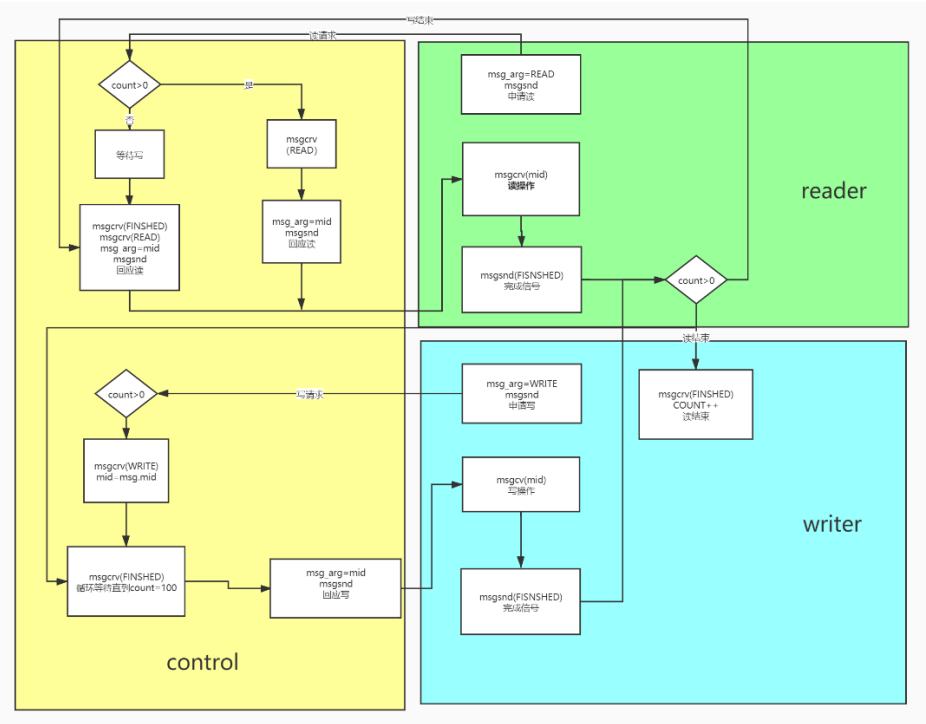
实现的流程：

实现这个过程的核心变量时消息队列和 msg_arg,他的两个变量

mtype：消息的类型(读请求，写请求，操作完成和进程编号)，其余三个较为容易理解，进程编号用于在进行申请的允许时使用，使用申请的进程 pid 作为消息类型。

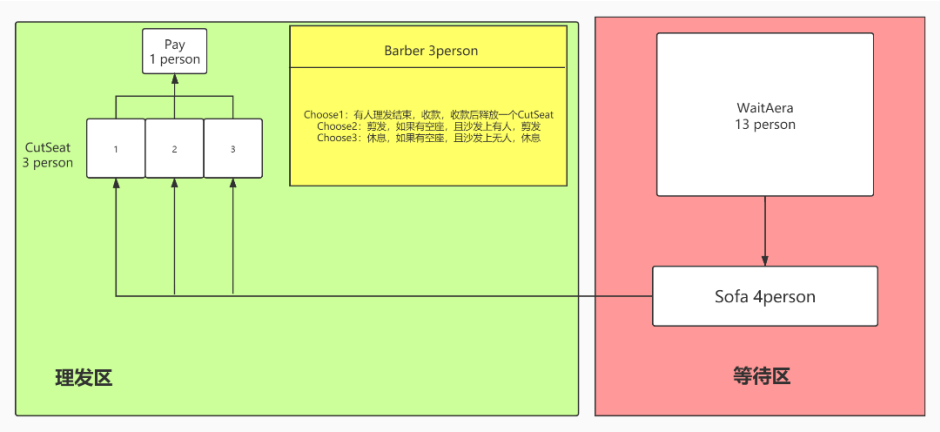
mid：申请进程的编号(reader 或者 writer)

使用直观的逻辑图描述示例程序的设计：



3、设计实验逻辑结构

本实验的执行过程比较复杂，将整体的状态逻辑和转换流程使用图示表述较为清晰，经过设计大致如下：



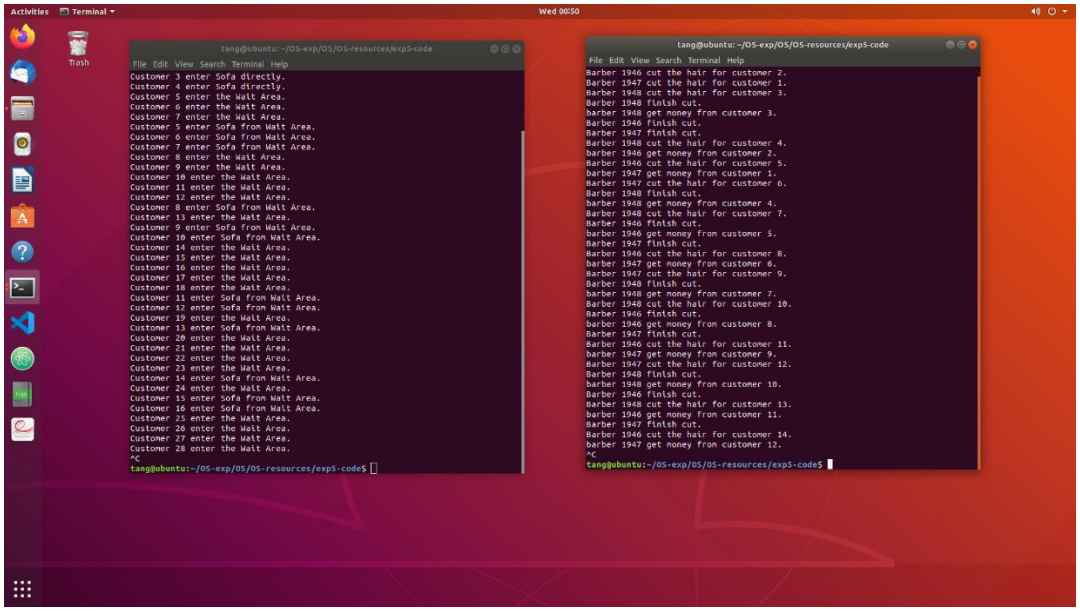
基本将程序的执行分为两个部分：等待区负责等待区域和沙发队列的维护，理发区则负责从沙发队列中拉出一个顾客进行理发并负责收款的维护。经过与助教的讨论，采取了一种效率最高的方式：

等待区域动态维护沙发区，若等待区域有人且沙发不满，则将队列前面的人放入沙发。当有人进入，如果沙发区不满则直接进入沙发，否则放入等待区域中。按照这种方式就可以保证沙发队列的正常工作。

理发区域采用将理发和付款分开处理的方式，从而保证系统的运行效率最高。在理发师处于空闲状态时，优先考虑是否需要结账，如果不需要或者账本正在被使用则从沙发队列中选取一个顾客进行剪发，这种系统可以保障所有的理发师能够最大限度地工作而不忙等，从而有效地提高系统的运行效率。

4、编写代码并且测试

清楚了上述的代码逻辑，利用信号量和消息队列，实现了代码并进行了测试，测试结果：最大化利用了理发师的时间，尽可能保证系统不阻塞。



实验总结：
经过实验四进程的同步实验和本实验进程的互斥实验，基本掌握了在操作系统调用层面，解决一些并发问题之间的同步与互斥问题的基本方法，利用实践较好地验证了理论课学习的同步互斥问题的基础知识，并且对利用系统调用解决同步互斥问题有了比较深刻的理解。
在调试示例实验的过程中，发现示例实验虽然能够避免读进程被饿死，但是接连不断的读进程可能也会造成写进程的饿死。基于这种情况和同学进行了讨论，并向助教提出了一些针对性的修改意见。通过对示例程序的精细调试，对同步互斥问题有了更加深刻的理解，并且具备了一定的编程解决同步互斥问题的能力。
在完成实验的过程中，由于对题意理解存在一定的偏差，设计编写出的系统存在比较大忙等开销，性能较差。与助教交流之后，修改了设计中的一些细节，保证了系统能够较为高效地工作。同时也意识到同步互斥问题在实现需求的同时，保证程序运行的高效也是重要的一个环节，对同步互斥问题的设计有了全新的认识。

附录：程序源代码
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进程睡眠秒数，以调解进程执行速度

    if (argv[1] != NULL) rate = atoi(argv[1]);
    else rate = 3;

    Msg_buf msg_arg;
    struct msqid_ds SofaMsg_buf;//用于返回当前的沙发消息队列情况

    //顾客同步信号量
    sem_flg = IPC_CREAT | 0644;
    CustomerSem_key = 101;
    CustomerSem_val = 0;
    CustomerSem_sem = set_sem(CustomerSem_key, CustomerSem_val, sem_flg);

    //账本互斥信号量
    sem_flg = IPC_CREAT | 0644;
    AccountSem_key = 201;
    AccountSem_val = 1;
    AccountSem_sem = set_sem(AccountSem_key, AccountSem_val, sem_flg);

    //沙发消息队列
```

```
    Queue_flag = IPC_CREAT | 0644;
    SofaQueue_key = 301;
    SofaQueue_id = set_msq(SofaQueue_key, Queue_flag);

    //等候区消息队列
    Queue_flag = IPC_CREAT | 0644;
    WaitQueue_key = 401;
    WaitQueue_id = set_msq(WaitQueue_key, Queue_flag);

    //账本消息队列
    Queue_flag = IPC_CREAT | 0644;
    AccountQueue_key = 501;
    AccountQueue_id = set_msq(AccountQueue_key, Queue_flag);

    //三个理发师进程：本进程，两个子进程
    pid_t pid1,pid2;
    pid1 = fork();

    if (pid1 == 0)//理发师 1
    {
        while (1)
        {
            //IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲 buf
            msgctl(SofaQueue_id, IPC_STAT, &SofaMsg_buf);

            if (SofaMsg_buf.msg_qnum == 0)//保持睡觉
                printf("Barber %d is Sleeping.\n", getpid());

            if(msgrcv(AccountQueue_id,&msg_arg,sizeof(msg_arg),PAID,IPC_NOWAIT)>0)
            {
                down(AccountSem_sem);
                printf("barber %d get money from customer %d.\n", getpid(),
msg_arg.mid);
                sleep(1);
                up(AccountSem_sem);
            }

            msgrcv(SofaQueue_id, &msg_arg, sizeof(msg_arg), SOFA, 0);//从沙发拉一个
剪发

            up(CustomerSem_sem);//唤醒顾客进程(让下一顾客坐入沙发)

            //剪头发的过程
```

```
                printf("Barber %d cut the hair for customer %d.\n", getpid(), msg_arg.mid);

                sleep(rate);

                msg_arg.mtype=PAID;
                msgsnd(AccountQueue_id,&msg_arg,sizeof(msg_arg),IPC_NOWAIT)
        }
    }
    else
    {
        pid2 = fork();
        if (pid2 == 0)//理发师 2
        {
            while (1)
            {
                //IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲
buf
                msgctl(SofaQueue_id, IPC_STAT, &SofaMsg_buf);

                if (SofaMsg_buf.msg_qnum == 0)//保持睡觉
                    printf("Barber %d is Sleeping.\n", getpid());

                if(msgrcv(AccountQueue_id,&msg_arg,sizeof(msg_arg),PAID,IPC_NOWAIT)>0)
                {
                    down(AccountSem_sem);
                    printf("barber %d get money from customer %d.\n", getpid(),
msg_arg.mid);
                    sleep(1);
                    up(AccountSem_sem);
                }

                msgrcv(SofaQueue_id, &msg_arg, sizeof(msg_arg), SOFA, 0);//从沙发拉一个
剪发

                up(CustomerSem_sem);//唤醒顾客进程(让下一顾客坐入沙发)

                //剪头发的过程
                printf("Barber %d cut the hair for customer %d.\n", getpid(), msg_arg.mid);

                sleep(rate);

                msg_arg.mtype=PAID;
                msgsnd(AccountQueue_id,&msg_arg,sizeof(msg_arg),IPC_NOWAIT)
```

```
            }
        }
        else//理发师 3
        {
            while (1)
            {
                //IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲
buf
                msgctl(SofaQueue_id, IPC_STAT, &SofaMsg_buf);

                if (SofaMsg_buf.msg_qnum == 0)//保持睡觉
                    printf("Barber %d is Sleeping.\n", getpid());

                if(msgrcv(AccountQueue_id,&msg_arg,sizeof(msg_arg),PAID,IPC_NOWAIT)>0)
                {
                    down(AccountSem_sem);
                    printf("barber %d get money from customer %d.\n", getpid(),
msg_arg.mid);
                    sleep(1);
                    up(AccountSem_sem);
                }

                msgrcv(SofaQueue_id, &msg_arg, sizeof(msg_arg), SOFA, 0);//从沙发拉一个
剪发

                up(CustomerSem_sem);//唤醒顾客进程(让下一顾客坐入沙发)

                //剪头发的过程
                printf("Barber %d cut the hair for customer %d.\n", getpid(), msg_arg.mid);

                sleep(rate);

                msg_arg.mtype=PAID;
                msgsnd(AccountQueue_id,&msg_arg,sizeof(msg_arg),IPC_NOWAIT)
            }
        }
    }
    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进程睡眠秒数，以调解进程执行速度
    if (argv[1] != NULL) rate = atoi(argv[1]);
    else rate = 3;

    Msg_buf msg_arg;
    struct msqid_ds SofaMsg_buf;//用于返回当前的沙发消息队列情况
    struct msqid_ds WaitMsg_buf;//用于返回当前的等待区消息队列情况

    //顾客同步信号量
    sem_flg = IPC_CREAT | 0644;
    CustomerSem_key = 101;
    CustomerSem_val = 0;
    CustomerSem_sem = set_sem(CustomerSem_key, CustomerSem_val, sem_flg);

    //账本互斥信号量
    sem_flg = IPC_CREAT | 0644;
    AccountSem_key = 201;
    AccountSem_val = 1;
    AccountSem_sem = set_sem(AccountSem_key, AccountSem_val, sem_flg);

    //沙发消息队列
    Queue_flag = IPC_CREAT | 0644;
    SofaQueue_key = 301;
    SofaQueue_id = set_msq(SofaQueue_key, Queue_flag);

    //等候区消息队列
    Queue_flag = IPC_CREAT | 0644;
    WaitQueue_key = 401;
    WaitQueue_id = set_msq(WaitQueue_key, Queue_flag);

    //账本消息队列
    Queue_flag = IPC_CREAT | 0644;
    AccountQueue_key = 501;
    AccountQueue_id = set_msq(AccountQueue_key, Queue_flag);

    int CustomerNum = 1;
    while (1)
    {
        //IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲 buf
        msgctl(SofaQueue_id, IPC_STAT, &SofaMsg_buf);
```

```
        if (SofaMsg_buf.msg_qnum < 4)//沙发没满，可能需要更新
        {
            Request_flag = IPC_NOWAIT;
            //非阻塞方式接收消息
            if (msgrcv(WaitQueue_id, &msg_arg,sizeof(msg_arg), WAIT, Request_flag)>= 0)//沙
发已经满：从等待区最前面拿一个人放入沙发
            {
                msg_arg.mtype = SOFA;
                printf("Customer %d enter Sofa from Wait Area.\n", msg_arg.mid);
                msgsnd(SofaQueue_id, &msg_arg, sizeof(msg_arg), IPC_NOWAIT);
            }
            else//沙发没满，顾客直接进入沙发
            {
                msg_arg.mtype = SOFA;
                msg_arg.mid = CustomerNum++;
                printf("Customer %d enter Sofa directly.\n", msg_arg.mid);
                msgsnd(SofaQueue_id, &msg_arg, sizeof(msg_arg),IPC_NOWAIT);
                sleep(rate);
            }
        }
        else
        {

            //IPC_STAT 为非破坏性读，从队列中读出一个 msgid_ds 结构填充缓冲 buf

            msgctl(WaitQueue_id, IPC_STAT, &WaitMsg_buf);


            if (WaitMsg_buf.msg_qnum < 13)//等待区没满，顾客直接进入等待区
            {
                printf("Customer %d enter the Wait Area.\n",CustomerNum);
                msg_arg.mtype = WAIT;
                msg_arg.mid = CustomerNum++;
                msgsnd(WaitQueue_id, &msg_arg, sizeof(msg_arg), IPC_NOWAIT);
                sleep(rate);
            }
            else//等待区满了,阻塞
            {
                printf("HairShop is full,Customer %d need go away...\n", CustomerNum);
                down(CustomerSem_sem);//使用信号量实现阻塞
                sleep(rate);
            }
        }
    }
    return EXIT_SUCCESS;
```

```
}
```