Linux Final Project

一、实验目的

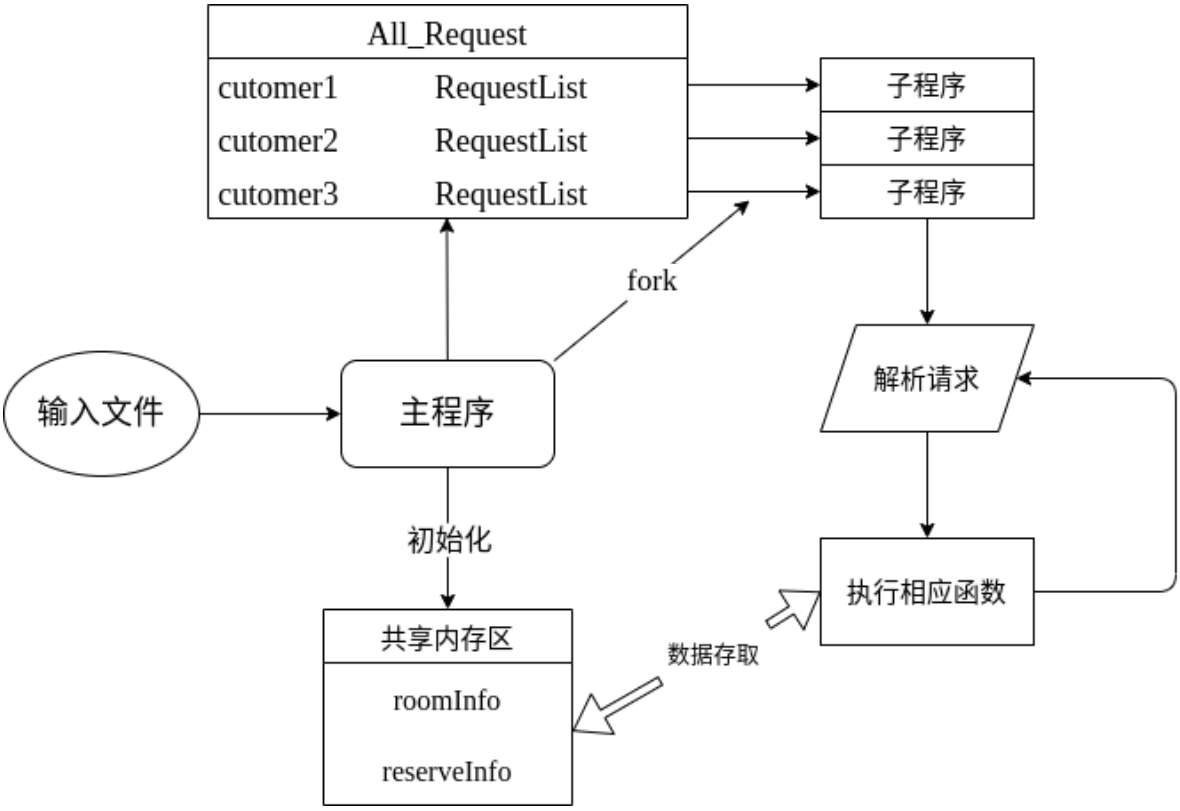
实现一个Linux下的多进程并发程序以模拟某个宾馆的预约系统运作情况

二、实验方法

学习了《UNIX网络编程卷2：进程间通信》消息传递，同步、共享内存区三部分，消息传递最后是没用到。

c语言编写，主要依靠linux下的多进程编程，使用共享内存区进行子进程之间的协同工作，在修改共享内存区时，借助信号量来实现互斥。

三、程序框架说明



四、代码运行测试

Quick Start

- ./bin目录下有编译完成可以直接运行的二进制文件main

```
./main [-f <filepath>]
```

- 从头开始

```
mkdir build
cd build
cmake ..
make
cd ../bin
./main [-f <filepath>]
```

测试文件结果及说明

默认路径下的测试文件为./resource/in_little.txt

结果

pdf显示效果不好，可以到https://github.com/iesimple/linux_Final_Project/blob/master/Linux%20Final%20Project.md

提前说明： 形如 "reserveany room_num=2 room_id=0 2022-5-1 reserve_days=1 name=customer2" 只是把请求打印，方便解释；除此之外，不同类型的请求某些值可能是没有的，比如check只有name，没有的值统一为0，输出的时候输出全部，因此可以看到挺多0；每一行最后的#后面是说明文字，原始输出是没有的。

```
reserve       room_num=1   room_id=11  2023- 8-10 reserve_days=2
 name=customer0#1.2预约了两间房，23-5-1 2天
reserveany    room_num=2   room_id=0   2022- 5-1  reserve_days=1
 name=customer2#2.0预约了11号房，23-8-10 2天
reserveblock  room_num=2   room_id=7   2023- 7-1  reserve_days=2
 name=customer1#3.1预约了7、8，23-7-1 2天
check         room_num=0   room_id=0      0- 0-0  reserve_days=0
 name=customer1#4.见#3
--------------------------
| room_id |     date     |
|     7   |  2023  7  1  |
|     7   |  2023  7  2  |
|     8   |  2023  7  1  |
|     8   |  2023  7  2  |
--------------------------
reserveblock  room_num=2   room_id=10  2023- 6-13 reserve_days=2
 name=customer0#5.2预约了7、8，22-11-29 4天
reserveblock  room_num=2   room_id=7   2022-11-29 reserve_days=4
 name=customer2#6.0预约了10、11，23-6-13 2天
cancel        room_num=1   room_id=7   2022-11-29 reserve_days=2
 name=customer2#7.2预约了7，22-11-29 两天
check         room_num=0   room_id=0      0- 0-0  reserve_days=0
 name=customer0#8.见#2、#6
--------------------------
| room_id |     date     |
|    10   |  2023  6 13  |
|    10   |  2023  6 14  |
|    11   |  2023  6 13  |
|    11   |  2023  6 14  |
|    11   |  2023  8 10  |
|    11   |  2023  8 11  |
--------------------------
reserve       room_num=1   room_id=11  2023- 8-10 reserve_days=2
 name=customer1#9.冲突了，可以结合上面的输出来看
Your request is not available!
```

```
cancelblock   room_num=1   room_id=7   2023- 7-1  reserve_days=2
 name=customer1#10.1取消了7，23-7-1 2天
cancel        room_num=1   room_id=7   2022-11-29 reserve_days=1
 name=customer2#11.2取消了7，22-11-29 1天
Your request is not available!
check         room_num=0   room_id=0      0- 0-0  reserve_days=0
 name=customer1#12.本来是#4的输出，#10取消部分
--------------------------
| room_id |     date     |
|     8   |  2023  7  1  |
|     8   |  2023  7  2  |
--------------------------
check         room_num=0   room_id=0      0- 0-0  reserve_days=0
 name=customer2#13.见#1、#5、#7、#11
--------------------------
| room_id |     date     |
|     1   |  2022  5  1  |
|     2   |  2022  5  1  |
|     7   |  2022 12  1  |
|     7   |  2022 12  2  |
|     8   |  2022 11 29  |
|     8   |  2022 11 30  |
|     8   |  2022 12  1  |
|     8   |  2022 12  2  |
--------------------------
cancel        room_num=1   room_id=7   2022-12-1  reserve_days=1
 name=customer2#14.2取消了7，22-12-1 1天
check         room_num=0   room_id=0      0- 0-0  reserve_days=0
 name=customer2#15.见#14
--------------------------
| room_id |     date     |
|     1   |  2022  5  1  |
|     2   |  2022  5  1  |
|     7   |  2022 12  2  |
|     8   |  2022 11 29  |
|     8   |  2022 11 30  |
|     8   |  2022 12  1  |
|     8   |  2022 12  2  |
--------------------------
---------------Hotel stat-----------------                    #最后宾馆的预约情况
------------------------------------------                    #因为这里人为设计了
每个人最后都是check请求
|     date     | customer_name | room_id |                    #因此可以和上面每个
人最后的输出比较来看
|  2022  5  1  |   customer2   |     1   |                    #也就是上面#8 #12
#15这三部分
|  2022  5  1  |   customer2   |     2   |                    #比较之后是一样的，
也就是程序逻辑上应该是没什么问题的
|  2022 12  2  |   customer2   |     7   |
|  2022 11 29  |   customer2   |     8   |
|  2022 11 30  |   customer2   |     8   |
|  2022 12  1  |   customer2   |     8   |
|  2022 12  2  |   customer2   |     8   |
|  2023  7  1  |   customer1   |     8   |
|  2023  7  2  |   customer1   |     8   |
|  2023  6 13  |   customer0   |    10   |
|  2023  6 14  |   customer0   |    10   |
|  2023  6 13  |   customer0   |    11   |
```

```
|  2023  6 14  |   customer0   |    11   |
|  2023  8 10  |   customer0   |    11   |
|  2023  8 11  |   customer0   |    11   |
----------------------------------------
```

测试文件自动生成脚本说明

```
./srcipt/requestGenerate.py
```

1. 参照项目说明文档要求编写，保证生成的每个请求是符合文档中的合法请求的。
2. 输入房间数量、顾客数量完成输入文件的生成。
3. 正常房间号是从1开始每次加1，但是因为有房间号不连续的问题，所以这里以0.1的概率使下一个房间号比前一个再多1。
4. 客人的名字这里统一使用 "customer<序号>" 的形式。
5. 一个客人请求中的名字，都会是发起请求的那个人的名字，也就是发起请求的人不能给其他人预约或者取消。
6. 一个客人请求数在 [1,10]。
7. 一个客人请求的序列中，前两个一定是 "reserve.*" 这三种，这是为了减少无意义的请求。
8. 一个客人请求的序列中，如果后面出现了 "cancel.*" 这三种，那这个请求中的房价号和时间会尽量参照之前的请求，也就是说，希望取消的预约一般是存在的，这同样也是为了减少无意义的请求。
9. 请求中最后一个值，也就是用户操作时间，这里设置为[1,5]，但是在客人数量大于等于10之后，会统一设置为0，这是考虑到测试时候的时间，而且就我的理解来看，操作时间不会影响程序本身的逻辑以及正确与否，只是会影响用户之间请求执行的先后，进而对结果产生影响（比如本来是客人A先预约了1号房，但是有操作时间，导致变成了客人B先预约了1号房），仅此而已。

五、实验体会

linux下的多进程编程还是很有意思的，可以更好地了解操作系统，更改地明白进程是如何运作的。除此之外，我也在实际实践的时候发现了进程与线程的差异（一开始用的线程，结果发现全局变量怎么是共享的，那我还要共享内存区做什么），之前一直理解的有欠缺，这次算是弄清楚了。

六、问题及解决

1. 在使用有名信号量时，最好在sem_open前提前sem_unlink，这是因为在调试的时候，如果某次程序中间出错，没有执行到最后的sem_unlink，文件系统中会存在该name的文件，同时又很不巧地是，在sem_open的oflags参数设置为O_CREAT | O_EXCL，导致打不开已经创建的信号量，在没有设置出错处理函数的时候，很难发现问题所在。
2. UNIX网络编程卷2中提到的，某些需要name的函数，可能需要px_ipx_name进行name的变换，以适应不同系统，在虚拟机里貌似不需要管，直接使用一个字符串即可，而且使用路径还会报如下错误：

   ```
   Invalid argument
   errno = 22
   ```

3. strcpy不要写成strcmp，不然怎么复制的过去!!!
4. getline()注意有没有读取到之前的换行符
5. strsep()注意windows下的换行符和linux下的换行符是不一样的!!!

   所以最好用strsep(str, " \r\n")
6. C里面 '' 和 "" 差别很大，前者是字符后者是字符串，在传递参数的时候""是指针，而''却会被认为是个整数!!!

7. 裂开，搞错了线程和进程

> 在使用全局变量的时候发现，主进程和他创建的所有不同子线程都是可以访问创建的全局变量的，那么问题来了，既然都可以访问了，我要共享内存区做什么...google...回想以前学的操作系统，一个进程创建的所有线程是共享进程中的资源的，因而线程间通过全局变量就可以达到共同操作同一块数据的作用，只有进程之间才是需要共享内存区来达到协同工作的目的。

8. 指针和共享内存区

> 千万要注意的是，进程本身是不共享内存的，因而普通指针指向的空间只有一个进程可以使用，这也是共享内存区为什么要存在的理由，当然比如文件指针也可以达到进程间协同工作的目的，但是这一类的操作往往需要经过内核，而共享内存区一个至关重要的作用就是让不同的进程可以不经过内核就实现数据共享。

不理解的问题

1. 父进程fork()出一个子进程之后，子进程是复制了父进程的所有数据？

```
id = 9863 ptr = 0x559fb6a2d4e0
id = 9865 ptr = 0x559fb6a2d4e0
id = 9868 ptr = 0x559fb6a2d4e0
id = 9866 ptr = 0x559fb6a2d4e0
id = 9869 ptr = 0x559fb6a2d4e0
id = 9867 ptr = 0x559fb6a2d4e0
id = 9870 ptr = 0x559fb6a2d4e0
id = 9871 ptr = 0x559fb6a2d4e0
id = 9872 ptr = 0x559fb6a2d4e0
id = 9864 ptr = 0x559fb6a2d4e0
parent ptr = 0x559fb6a2d4e0
```

上面是一个父进程parent创建了很多子进程，ptr是一个指针，指向一个结构体变量，按照我的理解，对于每个子进程来说，那个变量应该也被复制了一份，那么在子进程中变量地址的应该改变了吧，为什么还可以用同样的地址来访问呢？（这里做过测试，在所有子进程都通过这个指针都可以读取相同的数据，但是在子进程中修改数据之后进程之间数据就不一样了）
这里我也想到有一种可能性，是地址空间在编址的时候，每个有亲缘关系的进程都一样，也就是访问地址的“编号”一样，但是实际的空间不一样，这样就可以解释了。