**操作系统实验一实验报告**

**银行柜员服务问题**

学号：2019011008

班级：无92

姓名：刘雪枫

**目 录**

[一、 实验目的 1](#_Toc105618017)

[二、 实验平台 1](#_Toc105618018)

[三、 实验原理 1](#_Toc105618019)

[四、 算法设计思路 1](#_Toc105618020)

[五、 代码实现 2](#_Toc105618021)

[（一） 封装信号量操作 2](#_Toc105618022)

[（二） 封装线程操作 3](#_Toc105618023)

[（三） 计时器 3](#_Toc105618024)

[（四） 算法实现 4](#_Toc105618025)

[（五） 结果测试 6](#_Toc105618026)

[六、 运行测例 8](#_Toc105618027)

[七、 目录结构 11](#_Toc105618028)

[八、 思考题 12](#_Toc105618029)

# 实验目的

1. 通过对进程间通信同步/互斥问题的编程实现，加深理解信号量和P、V操作的原理；
2. 对Windows或Linux涉及的几种互斥、同步机制有更进一步的了解；
3. 熟悉Windows或Linux中定义的与互斥、同步有关的函数。

# 实验平台

本实验在x86\_64平台的Ubuntu 20.04 LTS操作系统上进行，编程语言采用C++11和Python 3.8.10，构建工具采用GNU Make 4.2.1。

# 实验原理

实验采用信号量（semaphore）的P、V操作，实现不同线程间的同步和资源访问的互斥。其中，用于实现互斥的二元信号量采用互斥量（mutex）。

C++11提供了一系列的与线程同步和互斥相关的标准库，本质上是pthread的封装。例如，std::thread用于创建线程，std::mutex用于实现互斥量，std::lock\_guard和std::unique\_lock是对各种锁的RAII封装。实验中将使用它们进行同步与互斥操作。

# 算法设计思路

实验采用同步算法的顾客（customer）线程和柜员（teller）线程的伪代码如下：

1. queue q;
2. mutex q\_mtx;
3. semaphore customer\_sem = 0;
5. **struct** customer
6. {
7. **int** idx;
8. semaphore sem = 0;
9. };
11. **void** customer\_action(customer self)
12. {
13. enter\_bank();
14. lock(q\_mtx);
15. q.push(self);
16. V(customer\_sem);
17. unlock(q\_mtx);
19. P(self.sem);
20. leave\_bank();
21. }
23. **void** teller\_action(teller self)
24. {
25. customer\* cust;
26. **while** (**true**) {
27. P(customer\_sem);
28. lock(q\_mtx);
29. cust = q.pop();
30. unlock(q\_mtx);
31. serve(cust);
32. V(cust->sem);
33. }
34. }

# 代码实现

## 封装信号量操作

首先，由于C++11没有提供信号量的封装，因此此处使用POSIX的semaphore.h中的信号量，将其封装为C++类semaphore，代码位于semaphore.hpp中。该类提供的接口如下：

1. #include <semaphore.h>
3. **class** semaphore
4. {
5. **public**:
6. semaphore(unsigned **int** init\_val);
8. semaphore(**const** semaphore&) = **delete**;
9. semaphore& operator=(**const** semaphore&) = **delete**;
11. **void** wait();
12. **void** post();
13. **int** get\_value() **const**;
14. ~semaphore() noexcept;
16. **private**:
17. ::sem\_t \_sem;
18. };

## 封装线程操作

C++11已经提供了std::thread类，用于管理线程，但其不是RAII的，管理不当会抛出异常。因此将其封装为RAII的线程类raii\_thread，在析构函数中可以自动执行join操作，即类似于C++20的std::jthread。代码实现在raii\_thread.hpp中。该线程类的接口与std::thread基本相同，提供线程的join、detach操作。

## 计时器

先在timer.hpp中编写timer类，用于获取当前时间。时间获取的值为自公元1970年元旦到当前的时间的值。本次实验的获取时间的单位选择毫秒，使用200毫秒作为一个时间单位。即若一个顾客的服务时间为n（n为整数），则实验中的服务时间为200 \* n毫秒。这个值可以在bank.hpp的bank<Judger>类中的time\_zoom变量设置。本实验中该值为200：

1. **using** timer\_t                  = timer<std::chrono::milliseconds>;
2. **static** constexpr **int** time\_zoom = 200;

## 算法实现

在bank.hpp中编写bank类实现前面伪代码所描述的算法。在与伪代码不同的是，顾客线程中，先休眠指定的时间，用于模拟顾客在休眠过后的时刻才进入银行。而代码中的Judger用来封装进入银行、离开银行、开始服务、结束服务的各种操作，可以用于在控制台显示log信息。这些操作的互斥由Judger内部保证。则顾客线程函数（\_customer\_action）和柜员线程函数（\_teller\_action）如下：

1. **void**
2. \_customer\_action(customer& self\_customer)
3. {
4. timer\_t::sleep(self\_customer.get\_enter\_time() \* time\_zoom);
6. // Enter bank
8. Judger::enter\_bank(self\_customer.get\_idx(), self\_customer.get\_enter\_time());
9. {
10. std::unique\_lock<std::mutex> lock(\_custumer\_queue\_mtx);
11. \_customer\_queue.emplace(&self\_customer);
12. \_customer\_sem.post(); // V
13. }
15. // Wait for server end
17. self\_customer.wait\_sem();
19. // Leave bank
21. Judger::leave\_bank(self\_customer.get\_idx(), \_get\_current\_time());
22. }
23. **void**
24. \_teller\_action(**int** teller\_idx)
25. {
26. **while** (**true**) {
28. // Wait for customer
30. \_customer\_sem.wait();
32. // Serve customer
34. customer\* servee = nullptr;
35. {
36. std::unique\_lock<std::mutex> lock(\_custumer\_queue\_mtx);
37. servee = \_customer\_queue.front();
38. assert(servee != nullptr);
39. \_customer\_queue.pop();
40. }
41. Judger::start\_serve(teller\_idx, servee->get\_idx(), \_get\_current\_time());
43. timer\_t::sleep(servee->get\_serv\_time() \* time\_zoom);
45. Judger::end\_serve(teller\_idx, servee->get\_idx(), \_get\_current\_time());
46. servee->post\_sem();
47. }
48. }

该算法在bank类的构造函数bank::bank中运行：

1. std::vector<raii\_thread> customer\_thrds;
2. customer\_thrds.reserve(customer\_thrds.size());
4. **for** (**int** i = 0; i < nteller; ++i) {
5. raii\_thread(&bank::\_teller\_action, **this**, i).detach();
6. }
8. **for** (**int** i = 0; i < (**int**)customers.size(); ++i) {
9. customer\_thrds.emplace\_back(&bank::\_customer\_action, **this**, std::ref(\*customers[i]));
10. }
12. **for** (auto& thrd : customer\_thrds) {
13. thrd.join();
14. }

## 结果测试

最后编写judger，对各种行为进行输出，并对算法的执行结果进行评判。其中，进入银行、离开银行、开始服务、结束服务的四个操作分别为：

1. **void**
2. enter\_bank(**int** customer\_idx, **int** time)
3. {
4. std::unique\_lock<std::mutex> lock(\_mtx);
5. std::printf("[Customer: %d] entered the bank at %d.\n", customer\_idx, time);
6. \_event\_list.emplace\_back(\_event\_type\_t::enter\_bank, customer\_idx, time);
7. }
9. **void**
10. leave\_bank(**int** customer\_idx, **int** time)
11. {
12. std::unique\_lock<std::mutex> lock(\_mtx);
13. std::printf("[Customer: %d] left the bank at: %d.\n", customer\_idx, time);
14. \_event\_list.emplace\_back(\_event\_type\_t::leave\_bank, customer\_idx, time);
15. }
17. **void**
18. start\_serve(**int** teller\_idx, **int** customer\_idx, **int** time)
19. {
20. std::unique\_lock<std::mutex> lock(\_mtx);
21. std::printf("[Teller: %d] started to serve [customer: %d] at time: %d\n", teller\_idx, customer\_idx, time);
22. \_event\_list.emplace\_back(\_event\_type\_t::start\_serve, teller\_idx, customer\_idx, time);
23. }
25. **void**
26. end\_serve(**int** teller\_idx, **int** customer\_idx, **int** time)
27. {
28. std::unique\_lock<std::mutex> lock(\_mtx);
29. std::printf("[Teller: %d] end serving [customer: %d] at time: %d\n", teller\_idx, customer\_idx, time);
30. \_event\_list.emplace\_back(\_event\_type\_t::end\_serve, teller\_idx, customer\_idx, time);
31. }

可以看到，四个操作在内部使用mutex进行互斥，并输出相应信息，再对操作进行记录。

最后，编写judge函数，对结果进行检查。检查包括但不限于是否有顾客被服务两次、是否有两个柜员服务了同一个顾客、是否有一个柜员同时服务两个顾客，等等。该judge函数在bank的构造函数中算法执行结束后调用检查：

1. bank(**int** nteller, **const** std::vector<std::unique\_ptr<customer>>& customers)
2. : Judger(nteller, customers), \_customer\_sem(0u)
3. {
4. std::vector<raii\_thread> customer\_thrds;
5. customer\_thrds.reserve(customer\_thrds.size());
7. **for** (**int** i = 0; i < nteller; ++i) {
8. raii\_thread(&bank::\_teller\_action, **this**, i).detach();
9. }
11. **for** (**int** i = 0; i < (**int**)customers.size(); ++i) {
12. customer\_thrds.emplace\_back(&bank::\_customer\_action, **this**, std::ref(\*customers[i]));
13. }
15. **for** (auto& thrd : customer\_thrds) {
16. thrd.join();
17. }
19. **if** (Judger::judge()) {
20. std::cout << "Test passed!" << std::endl;
21. } **else** {
22. std::cout << "Test failed!" << std::endl;
23. }
24. }

该judger代码的完整实现位于main.cpp中。

# 运行测例

第一个测例采用实验报告的默认测例：

1 1 10

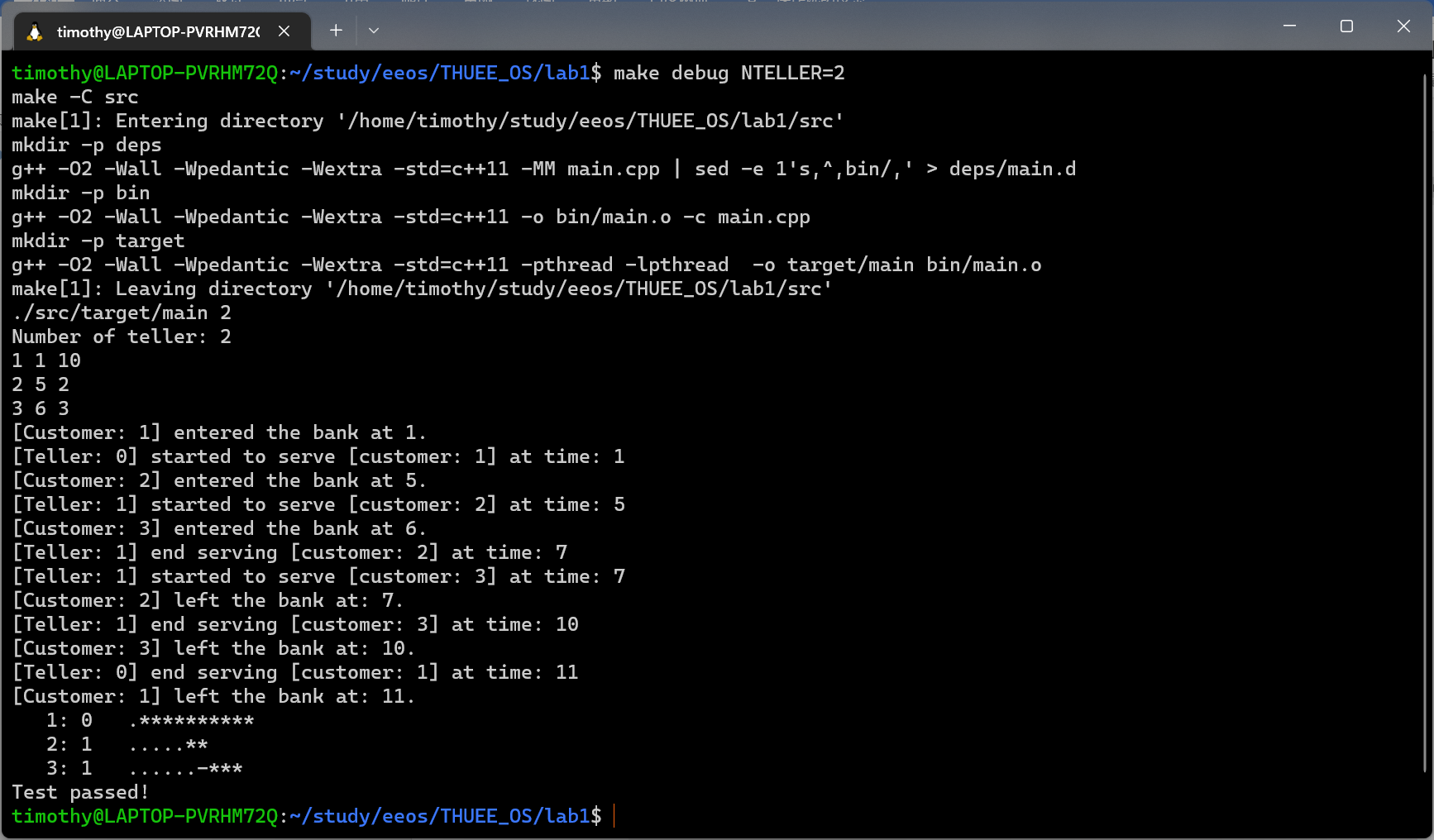
2 5 2

3 6 3

指定银行柜员数为2，运行命令：

$ make debug

然后输入该测例（Ctrl+d表示输入结束），查看输出结果：



其中，顾客号为1~3，柜员号为0~1。通过观察运行时的信息可以看到，柜员0在时刻1~10服务顾客1，柜员1在时刻5~6服务顾客2，柜员1在时刻7~9服务顾客3。整个过程符合预期。

之后的图形输出是每个顾客的可视化展示。最开始为“顾客号: 柜员号”，之后，’.’表示还未进入银行，’-’表示进入银行但尚未开始服务，’\*’表示正在接受服务。

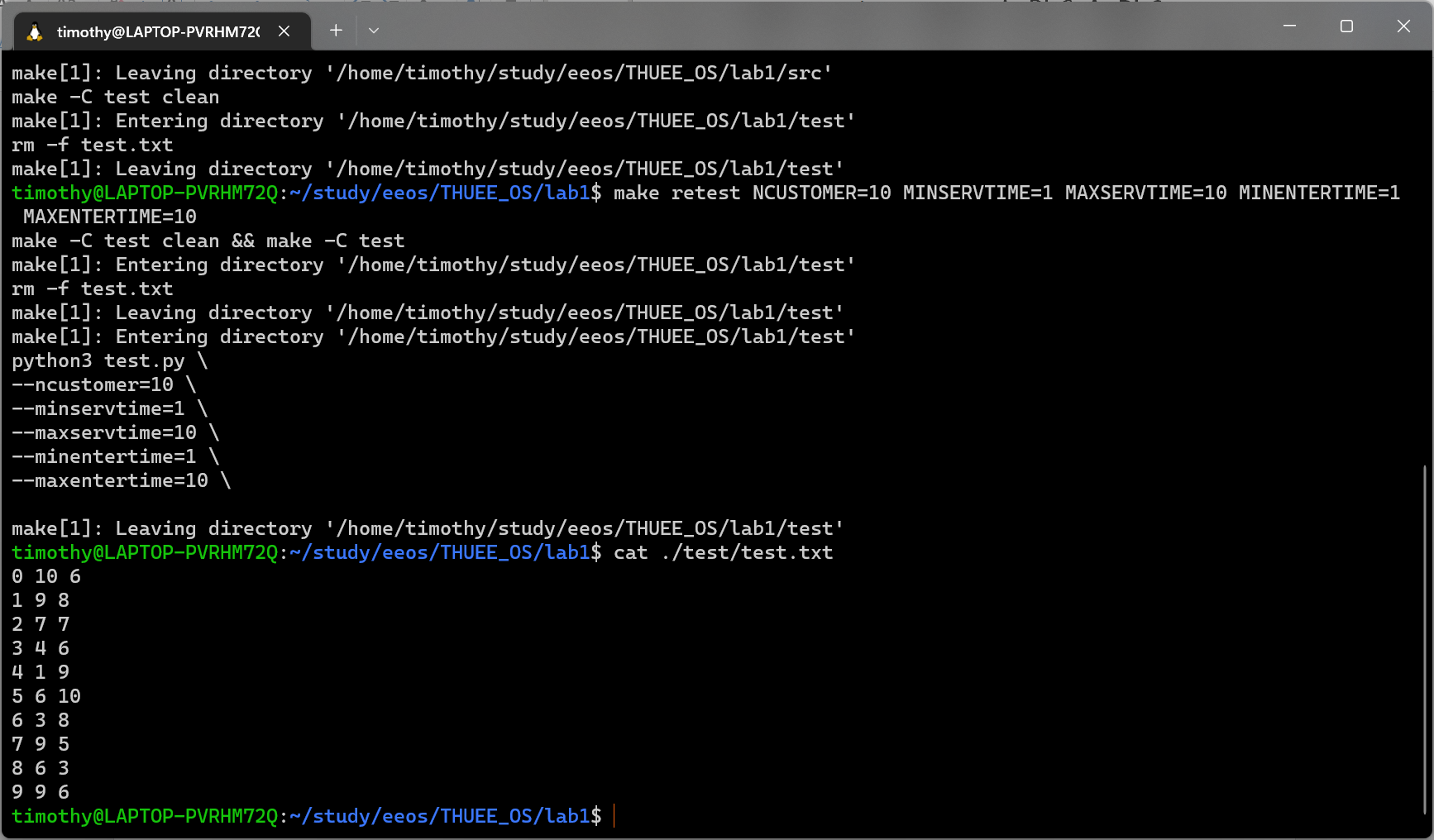
最后输出Test passed表示没有出现任何冲突。

然后运行其他的测试用例。本实验通过编写Python代码随机生成任意的测例，代码位于test/test.py中。运行：

$ make retest NCUSTOMER=10 MINSERVTIME=1 MAXSERVTIME=10

MINENTERTIME=1 MAXENTERTIME=10

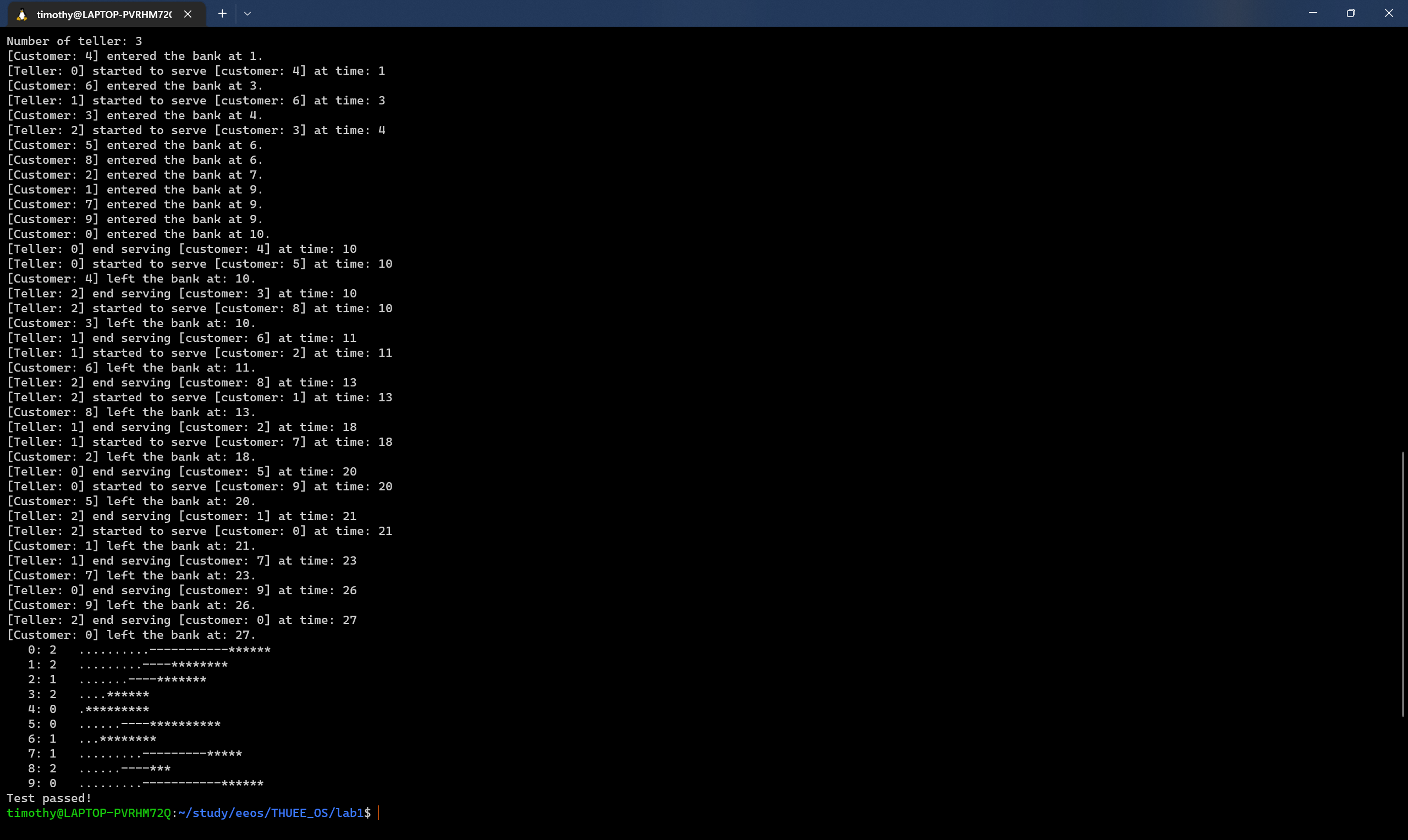
生成一个具有10个顾客，每个顾客服务时间为1~10、进入时间为1~10的测例。执行cat ./test/test.txt可以看到测例：



然后运行：

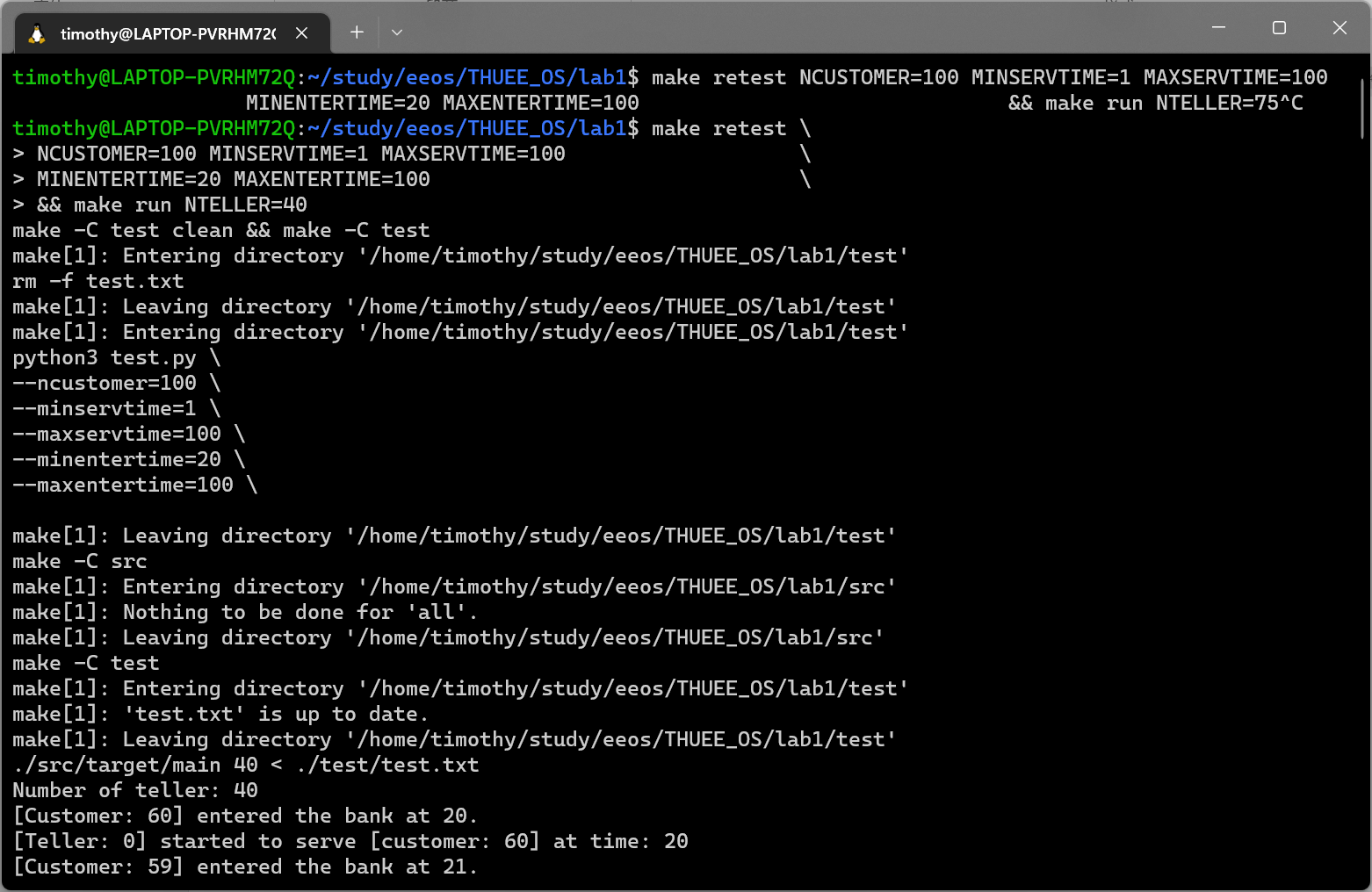
$ make run NTELLER=3

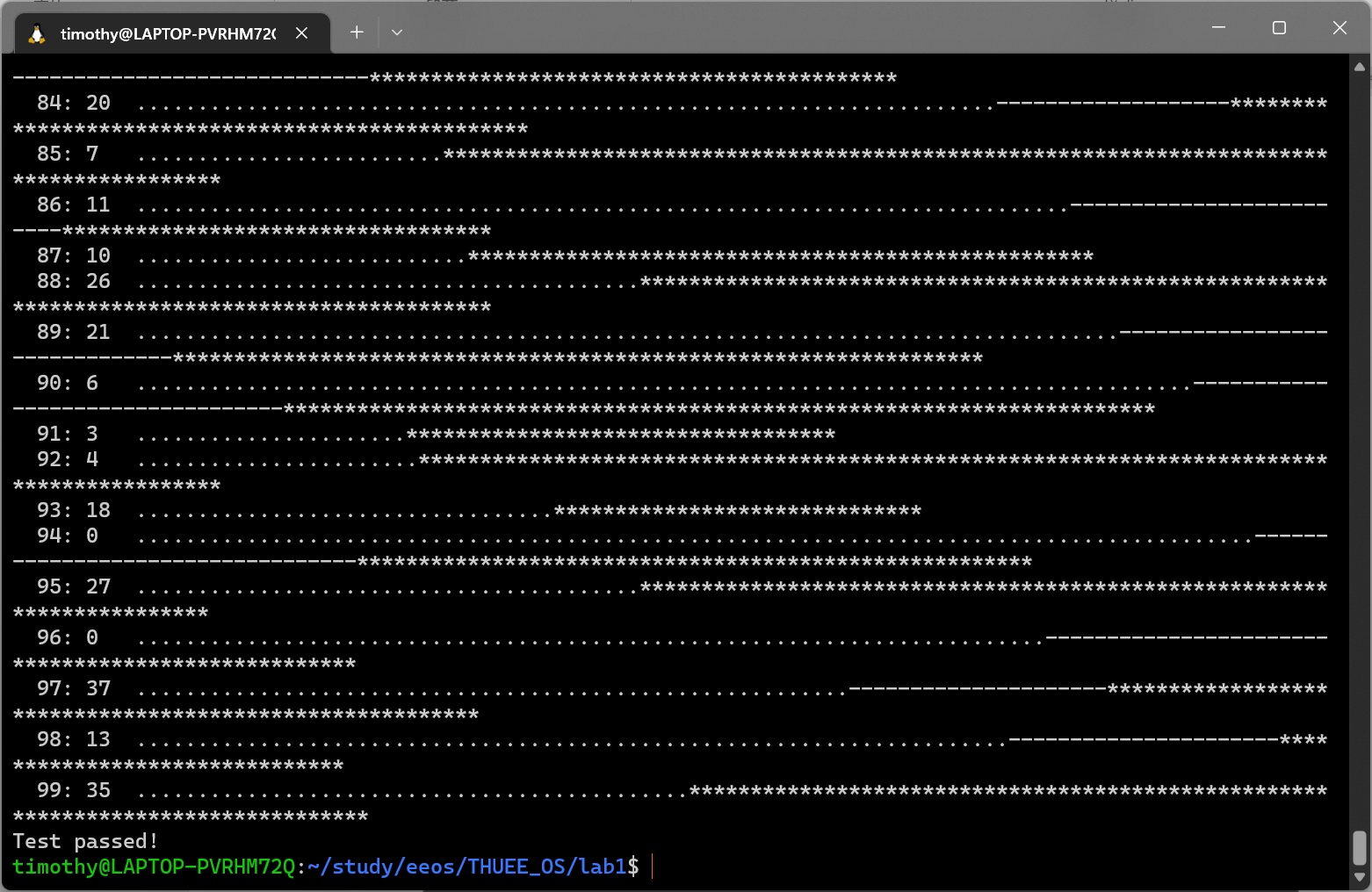
指定有3个银行柜员，即可运行该测例，结果如下：



通过观察输出，可以看到成功通过该测例。

继续增大测试数据量，增加到数百个顾客和数十个柜员，可以看到，仍然可以通过测试：





经过大量的数据测试，本算法均通过。

值得一提的是，在测试过程中发现一个问题。起初，bank.hpp中的time\_zoom的值取100而非200，本代码在Ubuntu 20.04 LTS上是运行完美的，但是后来我使用了Cygwin64运行便会偶尔出现实际服务时间略大于或略小于期望服务时间，或是顾客离开银行的时间滞后于结束服务的时间的情况。经过我的排查，发现并不是算法出现的问题，而是因为睡眠时间和计时器所获取的时间相差较大造成的。对此，我有一些推测。一来可能是由于Windows系统的线程调度问题，其时间片较长；二来可能是由于Cygwin提供的接口，或是Windows系统底层的API，其获取时间不够准确，或睡眠与指定的时间差别较大；第三可能是由于Cygwin兼容层内部的偏差；第四也可能是Cygwin兼容层内部一些函数的时间耗时较长，或是Windows本身进行内核态与用户态的切换耗时较长，导致一些操作不能在50毫秒内很快完成，等等。具体是何种原因还需要进一步探索和查阅资料。对此，我把time\_zoom改成了200，来一定程度上弥补这种偏差，在Cygwin64上也能成功运行。

# 目录结构

本实验的代码的目录结构如下：

.

├── Makefile

├── src

│   ├── Makefile

│   ├── bank.hpp

│   ├── customer.hpp

│   ├── main.cpp

│   ├── raii\_thread.hpp

│   ├── semaphore.hpp

│   └── timer.hpp

└── test

├── Makefile

└── test.py

# 思考题

1. 柜员人数和顾客人数对结果分别有什么影响？

答：

* 1. 固定柜员人数，探究顾客人数的影响。

固定柜员人数为5，服务时间和进入时间均为1~20的随机数。调整顾客人数，经过多次测试取平均值并把结果保留到整数，绘制时间与顾客人数的表格和曲线图：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 顾客人数 | 3 | 5 | 10 | 20 | 50 | 100 | 200 |
| 所需时间 | 28 | 30 | 36 | 54 | 113 | 217 | 430 |

可以看到，当顾客数较少时，所用时间随顾客数的增长较为缓慢，但增长速度在加快；而当顾客数较多（比柜员人数远远多）时，所用时间随顾客人数呈现线性增长趋势。

究其原因，当顾客数很少时，当空闲柜员数不少于同一时刻需要服务的顾客时，顾客都可以立即得到服务，即他们之间可以并行，因此在顾客数很少时，顾客数对总服务时间不会有太大影响。但是当顾客数很大时，柜员人数不足，考虑极限情况，柜员几乎不停息地工作，这时柜员已经接近饱和，总时间取决于服务时间总量，因此所用时间随顾客线性增加，且斜率为每个顾客的平均服务时间除以顾客人数，即：。从图像可以看到，斜率确实是在2.2附近。

* 1. 固定顾客人数，探究柜员人数的影响。

固定顾客人数为50，服务时间和进入时间均为1~20的随机数。调整柜员人数，绘制时间与柜员人数的表格和曲线图：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 柜员人数 | 1 | 3 | 5 | 10 | 20 | 30 | 40 | 50 | 60 | 70 |
| 所需时间 | 541 | 185 | 114 | 65 | 44 | 40 | 39 | 38 | 38 | 38 |



由上图可以看到，随着柜员人数的增加，总时间起初迅速下降，之后下降的速度减缓，最后逐渐过渡到平稳。

究其原因，当柜员人数较少的时候，每个时刻几乎都有顾客在等待，可以看做是工作饱和，所有柜员几乎都在不停地工作，此时柜员人数增加几倍则总时间大约就缩短为原来的几分之一，此时两者近似成反比例关系。随着柜员人数增加，工作便不是饱和的，此时便会有更多的柜员有更多的休息时间，因此增加柜员对时间的缩短效果逐渐降低。当柜员数量足够多，以至于达到柜员数不少于同时需要服务的顾客数的时候，任何顾客都可以再刚刚进入银行时得到服务，因此此时再增加柜员数，便不会缩短总服务时间。因此，曲线呈现最后的平直阶段。

* 1. 当顾客人数或柜员人数非常多时，实验发现，结果就会出现一定的错误。在进行之前的测试时，我曾把柜员数开至数万量级（线程数更多会造成资源不足而无法开辟线程），此时结果并不符合预期。究其原因，当线程数很多的时候，操作系统调度线程的时间会很长，以至于无法很快让能够运行的线程很快地开始运行，因此总服务时间会上升，并且响应会存在较大的延迟。

1. 实现互斥的方法有哪些？各自有什么特点？效率如何？

答：

* 1. 禁用中断

进入临界区之前，关闭硬件中断，防止在执行期间被调度。特点是只适合单核处理器，不适合多核处理器。而且，由于禁用和开启中断属于特权指令，因此用户无法直接管理中断，需要由操作系统提供接口。而操作系统把控制中断的权力交给用户，也是非常不安全的。用户利用此功能编写的恶意程序，或是用户写出的程序出错，都会造成严重的后果。禁用中断的效率是相对比较高的。

* 1. 忙等待

一些算法例如严格轮换法、Petersen算法、硬件指令法，都使用了忙等待的算法，不断地去判断某个或某些变量的值。这种算法在用户态也实现相对简单，而且不需要陷入操作系统内核，避免了用户态和内核态切换的开销。但是一来忙等待占用CPU资源，在使用效率上较低，二来还可能存在优先级反转等问题。

* 1. 信号量

信号量采用P、V两个原语来实现同步和互斥，且阻塞不会占用CPU，是一种较常用的方法，CPU使用效率较高。但是有一些信号量实现有时会陷入到内核态，进行用户态和内核态的转换，会带来一定的开销。

* 1. 互斥量

如果只有信号量是二元的，且只有同步的P、V操作（即P和V在一个线程内成对出现，且先P后V），则信号量可以简化为互斥量。互斥量在使用上相对简单，并且相对于信号量来说性能更好、效率更高，但是也和信号量存在相同的问题。

* 1. 管程

管程是编程语言提供的一种特性，只允许一个线程在管程函数内，互斥由编译器来保证，给程序员的编程带来了一些便利。但是有些语言不支持管程，且一些管程实现可能无法在中途阻塞。

* 1. 条件变量

条件变量也是一种实现同步和互斥的工具，与互斥量配合使用。其优点是相比于信号量来说更简单，易于使用。但是条件变量的解锁和睡眠、唤醒和加锁并不是原子的，并且条件变量存在虚假唤醒等问题需要额外考虑。

* 1. 无锁的原子操作

一些原子操作可以不依靠锁实现，而是使用一些特殊的支持原子操作的硬件指令。这类指令可以保证一个操作是原子的，且不存在多核CPU上cache不一致的问题。这种方法在用户态使用不需要陷入到内核，执行效率很高，性能比使用锁要好。但是适用范围比使用锁要窄。

* 1. RCU（读-复制-更新）

一种依靠原子操作的无锁算法，用于大型数据结构的更新。此算法依赖于原子操作，不需要在更新期间锁住数据结构，也不需要内核态和用户态的转换，效率很高。但适用范围也是有限的。

* 1. 利用进程间通信的其他方式也可以实现互斥。例如，使用共享内存、使用网络套接字、使用消息队列，等等。这些都需要操作系统来参与，因此开销较大，执行效率不高。