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. 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。

二、 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。

三、 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。

四、 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;
```

```
4.
5.  struct customer
6.  {
7.      int idx;
8.      semaphore sem = 0;
9.  };
10.
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);
18.
19.     P(self.sem);
20.     leave_bank();
21. }
22.
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. }
```

五、 代码实现

（一）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：

```
#include <semaphore.h>

class semaphore
{
public:
    semaphore(unsigned int init_val);

    semaphore(const semaphore&) = delete;
    semaphore& operator=(const semaphore&) = delete;

    void wait();
    void post();
    int get_value() const;
    ~semaphore() noexcept;

private:
    ::sem_t _sem;
};
```

（二） 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。

（三） 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：

```
using timer_t                   = timer<std::chrono::milliseconds>;
```

```
2.  static constexpr int time_zoom = 200;
```

（四） 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);
5.
6.      // Enter bank
7.
8.      Judger::enter_bank(self_customer.get_idx(), self_customer.get_ent
    er_time());
9.      {
10.         std::unique_lock<std::mutex> lock(_custumer_queue_mtx);
11.         _customer_queue.emplace(&self_customer);
12.         _customer_sem.post(); // V
13.     }
14.
15.     // Wait for server end
16.
17.     self_customer.wait_sem();
18.
19.     // Leave bank
20.
21.     Judger::leave_bank(self_customer.get_idx(), _get_current_time());

22. }
```

```
1.  void
2.  _teller_action(int teller_idx)
3.  {
```

```
    while (true) {

        // Wait for customer

        _customer_sem.wait();

        // Serve customer

        customer* servee = nullptr;
        {
            std::unique_lock<std::mutex> lock(_custumer_queue_mtx);
            servee = _customer_queue.front();
            assert(servee != nullptr);
            _customer_queue.pop();
        }
        Judger::start_serve(teller_idx, servee->get_idx(), _get_current_time());

        timer_t::sleep(servee->get_serv_time() * time_zoom);

        Judger::end_serve(teller_idx, servee->get_idx(), _get_current_time());
        servee->post_sem();
    }
}
```

该算法在 bank 类的构造函数 bank::bank 中运行：

```
std::vector<raii_thread> customer_thrds;
customer_thrds.reserve(customer_thrds.size());

for (int i = 0; i < nteller; ++i) {
    raii_thread(&bank::_teller_action, this, i).detach();
}

for (int i = 0; i < (int)customers.size(); ++i) {
    customer_thrds.emplace_back(&bank::_customer_action, this, std::ref(*customers[i]));
}

for (auto& thrd : customer_thrds) {
    thrd.join();
```

```
14. }
```

（五） 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.  }
8.
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_id
    x, time);
14.     _event_list.emplace_back(_event_type_t::leave_bank, customer_idx,
     time);
15. }
16.
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. }
24.
25. void
26. end_serve(int teller_idx, int customer_idx, int time)
27. {
```

```
    std::unique_lock<std::mutex> lock(_mtx);
    std::printf("[Teller: %d] end serving [customer: %d] at time: %d\n", teller_idx, customer_idx, time);
    _event_list.emplace_back(_event_type_t::end_serve, teller_idx, customer_idx, time);
}
```

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：

```
bank(int nteller, const std::vector<std::unique_ptr<customer>>& customers)
    : Judger(nteller, customers), _customer_sem(0u)
{
    std::vector<raii_thread> customer_thrds;
    customer_thrds.reserve(customer_thrds.size());

    for (int i = 0; i < nteller; ++i) {
        raii_thread(&bank::_teller_action, this, i).detach();
    }

    for (int i = 0; i < (int)customers.size(); ++i) {
        customer_thrds.emplace_back(&bank::_customer_action, this, std::ref(*customers[i]));
    }

    for (auto& thrd : customer_thrds) {
        thrd.join();
    }

    if (Judger::judge()) {
        std::cout << "Test passed!" << std::endl;
    } else {
        std::cout << "Test failed!" << std::endl;
    }
}
```

该 judger 代码的完整实现位于 main.cpp 中。

六、 运行测例

第一个测例采用实验报告的默认测例：

1 1 10

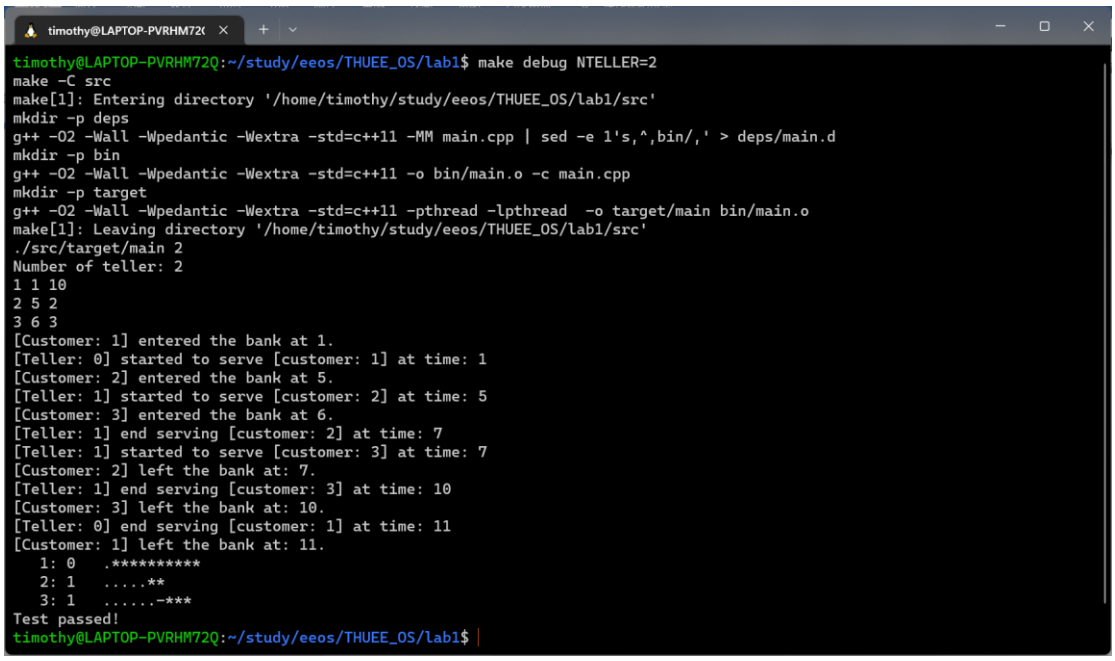
2 5 2

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。

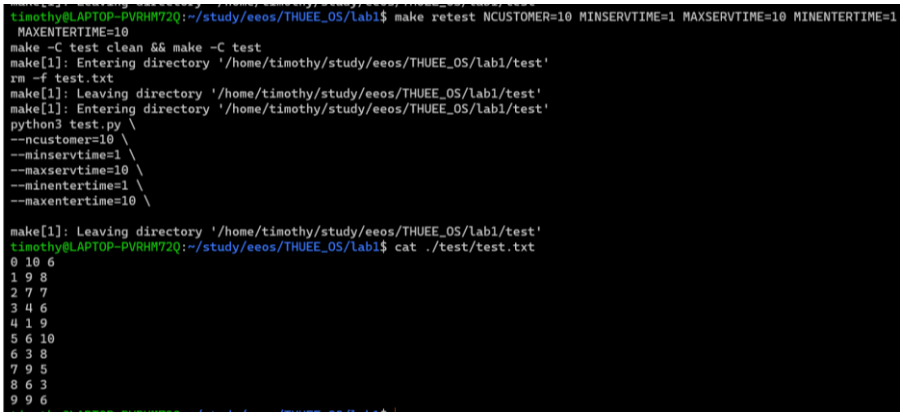
之后的图形输出是每个顾客的可视化展示。最开始为“顾客号: 柜员号”，之后，’.’表示还未进入银行，’-’表示进入银行但尚未开始服务，’*’表示正在接受服务。

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
```

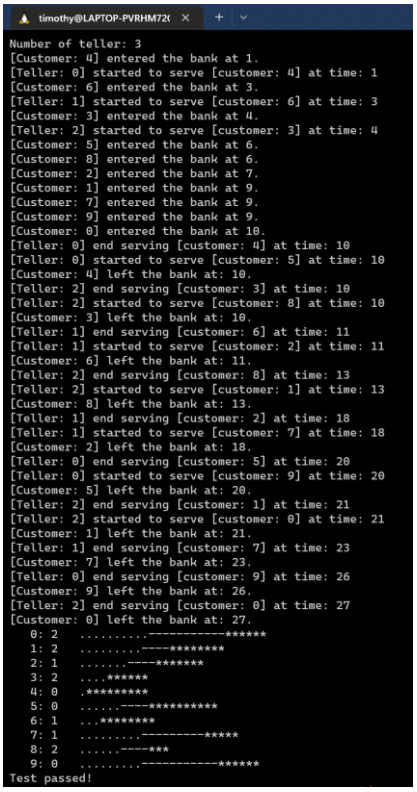
生成一个具有 10 个顾客，每个顾客服务时间为 1~10、进入时间为 1~10 的测例。执行 cat ./test/test.txt 可以看到测例：



然后运行：

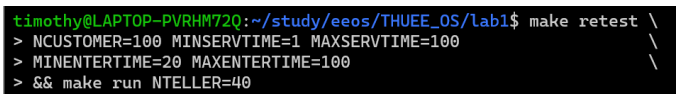
```
$ make run NTELLER=3
```

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。

七、 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
```

八、 思考题

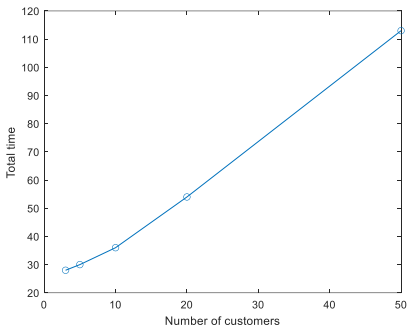
1. 柜员人数和顾客人数对结果分别有什么影响？

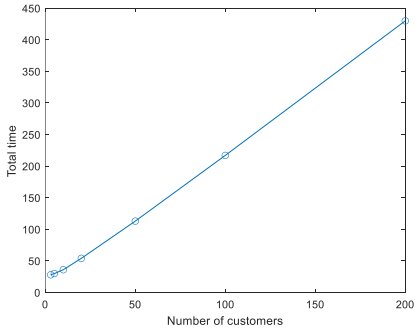
答：

a) 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。

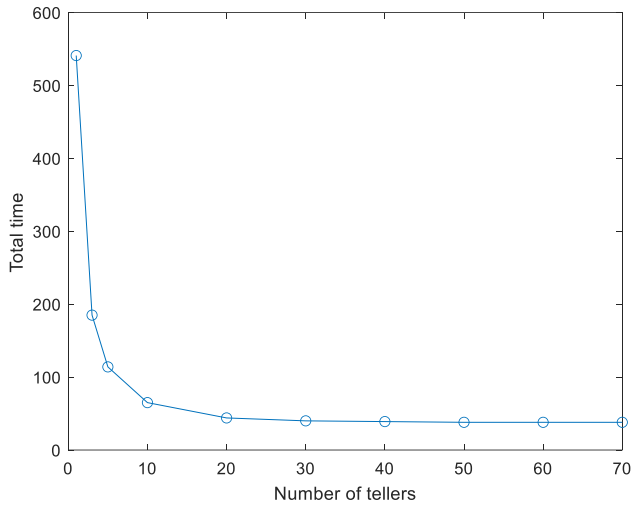
究其原因，当顾客数很少时，当空闲柜员数不少于同一时刻需要服务的顾客时，顾客都可以立即得到服务，即他们之间可以并行，因此在顾客数很少时，顾客数对总服务时间不会有太大影响。但是当顾客数很大时，柜员人数不足，考虑极限情况，柜员几乎不停息地工作，这时柜

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：$\frac{1+20}{2} \div 5 = 2.2$。从图像可以看到，斜率确实是在 2.2 附近。

b) 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。

c) 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。

2. 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。

(2) 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。

(3) 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。

(4) 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。

(5) 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。

(6) 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。

(7) 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。

(8) 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。

(9) 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。