操作系统实验 进程间同步/互斥 银行柜员服务问题

姓名：陈彦旭

班级：无24

1 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。

2 问题描述

银行有 n 个柜员负责为顾客服务，顾客进入银行先取一个号码，然后等着叫号。当某个柜员空闲下来，就叫下一个号。编程实现该问题，用 P、V 操作实现柜员和顾客的同步。

3 实现要求

1. 某个号码只能由一名顾客取得；
2. 不能有多于一个柜员叫同一个号；
3. 有顾客的时候，柜员才叫号；
4. 无柜员空闲的时候，顾客需要等待；
5. 无顾客的时候，柜员需要等待。

4 设计思路

> OS: Ubuntu 24.04.2 LTS x86_64.
>
> Language: C++ (-std=C++20).

首先，让我们想象银行中有一个处理取号、叫号等操作的“机器人”，它在顾客和柜员之间充当沟通的媒介，顾客取号在机器人上操作，然后机器人通知柜员有新顾客到来，柜员叫号通过机器人通知顾客。因此我们重点关注机器人如何工作。

对于顾客线程，大致流程为：

1. 顾客到达后，先取号码，将自己的号码加入到等待队列中。这里不同的顾客对“当前取号号码”和“等待队列”应设置为互斥访问，因为不能有2个顾客同时取同一个号码，入队操作也是互斥的，不能有两个顾客同时入队。
2. 机器人通知柜员有新顾客到来，必要的时候可以叫号。
3. 顾客等待机器人叫号，当自己处于等待队列第一名且有空闲柜员的时候，会被相应柜员叫号，开始办理业务，结束后退出。

对于柜员线程，大致流程为：

1. 最开始没有顾客，处于空闲状态。
2. 等待顾客到来。
3. 有顾客到来时叫号，从等待队列中取出第一个顾客。这里不同的柜员对“等待队列”应设置为互斥访问，因为不能有两个柜员同时从队列中取出一个顾客。
4. 通知机器人可以办理这名顾客。
5. 服务当前顾客。服务完成后，循环执行步骤2。

设置以下主要变量：

1. 当前取号号码 `getting_number` ，有新顾客到来取号后递增。需要加锁。
2. 保护取号号码 `getting_number` 的锁 `getting_number_mutex` 防止多个顾客同时取号。
3. 等待队列 `waiting_queue` ，存放已经取号但未被叫号、正等待空闲柜员的顾客。需要加锁。
4. 保护等待队列 `waiting_queue` 的锁 `waiting_queue_mutex` ，防止不同顾客同时入队，也防止顾客取号和柜员叫号同时进行。
5. 代表“有顾客等待被服务”的信号量 `customer_ready` 。值大于0时，代表有正在等待的顾客，若有柜员空闲可开始办理；值小于0时，代表没有顾客，有多个柜员正在等待。柜员线程执行 P 操作代表机器人尝试为柜员叫号，尝试将信号量减1，如果值小于等于0则柜员线程进入阻塞等待；顾客线程执行 V 操作代表有顾客到来，机器人会为顾客取号，将信号量加1。
6. 代表“有柜员空闲”的信号量 `teller_ready` ，值大于0时，代表有柜员空闲，顾客到来时可直接开始办理；值小于0时，代表没有柜员空闲，有多个顾客正在等待。顾客线程执行 P 操作代表机器人尝试为顾客安排柜员服务，尝试将信号量减1，如果值小于等于0则顾客线程进入阻塞等待；柜员线程执行 V 操作代表有柜员空闲，机器人为柜员叫号，将信号量加1。

本程序中时间的单位均为**毫秒(ms)**，由于线程运行指令的速度极快，所以程序中顾客线程和柜员线程中会在不同状态下，对前一条指令或后一条指令进行时间点捕获，存在一些误差，但是在毫秒量级上对精度几乎没有影响。

C++ 的标准输出流 `std::cout` 并不是线程安全的。顾客线程和柜员线程在运行过程中都需要输出一些信息便于调试，但是短时间内不同线程都进行输出时，可能会导致输出内容交错，因此我们考虑对 `std::cout` 也加上互斥锁 `cout_mutex` ，利用模板参数包 `Args &&...args` 和完美转发 `std::forward` ，封装为线程安全的输出函数 `safe_print()` ，确保线程在输出信息的时候不串扰。该函数定义在命名空间 `utils` 中。

程序中需要保证所有顾客线程同时启动，这样不同顾客线程的时间戳才是同步的。如果通过一般的方法，遍历包含顾客指针的容器 `tellers` 然后依次 `teller->start()` （这句话就已经立即启动了线程），不同顾客线程的真实启动时间存在微小的时间差，也即不同顾客线程视角下银行开门的时间不同。因此我们使用 C++20 中的 `std::barrier` ，设置一个全局变量屏障 `customers_barrier` ，类型为独享智能指针 `std::unique_ptr<>` ，所有顾客线程访问的是同一个屏障、到达同一个屏障，设置屏障的数量为“总顾客数+1”。顾客线程开始后先到达屏障处集合等待。在启动完所有顾客线程之后，所有顾客线程已经到达屏障，主线程再最后到达屏障处，此时总数量达到“总顾客数+1”，屏障解开，所有顾客线程同时放行，此时的时间点即为“银行开门时间”，保证了时间参考点的统一。

柜员线程需要比顾客线程更早启动，提前等待因为没有顾客的时候它们本身就处于等待阻塞状态。银行开门之前柜员应该就准备好工作，这也符合实际情况。而且柜员线程也不需要保证同时启动，不需要有统一的参考时间，因为它们可以通过拿到顾客指针，访问顾客线程中的成员变量，得到银行开门、开始服务、结束服务等时间点的信息。

顾客到达银行后，需要“取号+入队”，需要“锁号码号码+等待队列”。柜员空闲后，需要“取队首顾客号码+叫号”，需要锁“等待队列+叫号号码”。这里有两种常见的方法：

（1）设置三个互斥锁，分别保护“取号号码”、“等待队列”和“叫号号码”。此方法的缺点在于，将顾客的“取号+入队”这两个操作分离、柜员的“取队首顾客号码+叫号”这两个操作分离，但现实中顾客在机器上取号之后会立即进入队列，这两个操作的间隙不可能会有其他顾客插队，不会出现有顾客取号早但是入队晚的现象，也就是入队顺序和到达顺序是完全相同的。柜员访问到队首顾客后会立即叫这位顾客的号码，不会被其他柜员抢先。此方法中顾客“取号+入队”需要先后获取两个互斥锁，在两个锁的间隙可能会被与它几乎同时到达取号的另一位顾客线程抢先，柜员线程也会出现类似现象。

```
// arrive and get a number
{
    std::unique_lock<std::mutex> lock(globals::getting_number_mutex);
    // ...
}

// join the waiting queue
{
    std::unique_lock<std::mutex> lock(globals::waiting_queue_mutex);
    // ...
}
```

（2）设置两个互斥锁，分别保护“取号号码+等待队列”和“等待队列+叫号号码”。此方法可以避免出现第一种方法的插队现象，但是顾客线程和柜员线程任意一个在进行操作时都会锁住“等待队列”，顾客取号的时候柜员无法叫号，柜员叫号的时候顾客无法取号，也即顾客取号和柜员叫号不能并发，会相互阻塞。这也不符合实际情况。

针对这个问题，我们的解决方法是：
（1）对于顾客线程，采用第一种方法：“取号+入队”分离，分别用两个锁保护，插队现象出现概率极小，但仍无法完全避免，这也是本程序的一个局限性。
（2）对于柜员线程，只保留用一把保护“等待队列”的锁（这个锁和顾客线程使用的锁应为同一个全局变量）。叫号方式不采用一般的“全局叫号号码，顾客循环监听叫号号码是否为自己所取号码”的方式（类似于传统的广播叫号），因为这样每一位等待的顾客都需要轮流获取访问“叫号号码”的锁，效率较低。本程序使用 C++20 中 `std::promise, std::future` 的特性,。每个顾客对象内部都持有一个 `std::promise<int> called_promise_` 与 `std::future<int> called_future_` 成员，顾客线程在 `called_future_.get()` 处阻塞，等待某个柜员叫号。柜员线程从等待队列中取出一个顾客对象指针后，通过提供的 `notify_called()` 接口调用 `called_promise_.set_value(...)`，将自己的姓名发送给顾客，顾客线程立即从 `called_future_.get()` 返回，并且得知是哪个柜员叫自己。这种方式使得每对顾客和柜员之间可以精确配对、单点通信，类似于清华大学玉树园餐厅的取餐方式：顾客买餐后得到一个叫号器，菜品准备好后工作人员发送信号到对应叫号器，顾客看到叫号器闪烁后才取餐。

还有一些值得考虑的问题：

（1）如何确定所有顾客服务完成？

即使所有柜员都空闲、等待队列中也没有顾客，也即银行中一个顾客都没有，也无法确定后面是否会到达新的顾客，不能作为结束的条件。所以我们采用“上帝视角”的方法，在事先知道将要到达的“顾客总人数”的情况下（通过读取数据文件可以得到），设置一个全局计数器 `served_customers_number` 代表“已服务顾客人数”，并使用 `std::atomic<int>` 类型，保证“计数器”的读取和修改都是原子性的，无锁也能安全操作（相比于使用互斥锁更简

单，因为这里只需要读写一次“计数器”即可，而互斥锁更适合锁住多个操作的场景）。每当服务完一位顾客，对应的柜员线程将“计数器”递增。每一个柜员线程都循环检查当前是否有“已服务顾客人数<顾客总人数”，作为循环条件，否则柜员线程可以安全退出。

```
// void Teller::serve()
while (globals::served_customers_number.load() < globals::customers_number)
{
    // wait for a customer in the queue
    globals::customer_ready->acquire();

    // ...

    // add a served customer
    globals::served_customers_number.fetch_add(1);
}
```

（2）所有顾客服务完成后，如何通知所有柜员线程退出下班？

如果仅仅将上一问中的“已服务顾客人数<顾客总人数”作为循环条件，还存在一个问题：最后一个顾客服务完成后，是服务他的柜员线程最后一次递增了“计数器”，并在下一次循环开始时判断“已服务顾客人数<顾客总人数”为 false，因此能够退出。但是其他柜员线程刚才开始这一轮循环时，最后一名顾客尚未服务完，所以判断“已服务顾客人数<顾客总人数”为 true，但此时银行中已经没有顾客了，顾客资源信号量 `customer_ready` 的计数值为0，因此其他柜员线程都阻塞在循环刚开始尝试获取 `customer_ready` 的地方，无法退出。

针对这个问题，我们注意到顾客和柜员线程的重要区别：顾客是寻求服务的一方，柜员是提供服务的一方。如果顾客线程没有获取到柜员信号量，它必须阻塞，但是一定会等到获取成功的时候，因为柜员不可能在没有服务完所有顾客之前下班。但是在没有顾客的情况下，如果让柜员线程会一直阻塞，它无法知道将来是否还有顾客到来。只能通过上帝视角，在所有顾客服务完成之后通知柜员线程，而柜员线程不能在尝试获取顾客信号量的地方阻塞太长时间，需要抽空查看银行的通知，才能知道自己是否可以下班，因此采用信号量 `std::counting_semaphore` 的 `try_acquire_for()` 方法，而非简单的 `acquire()` 。在尝试获取信号量时，至多阻塞一段时长，如果没有获取到就直接进入下一轮循环，下一轮循环的开始依然尝试获取信号量，这样能够让柜员线程定期检查没有接收到“下班通知”，也既能实现与一直阻塞相同的效果。

基于上述讨论和分析，顾客线程的完整流程为：

1. 在屏障处 `customers_barrier` ，所有顾客线程统一起跑，并记录当前的出发时间点 `start_time_point_` 。
2. 休眠一段时间，模拟到达银行，记录当前到达银行的时间点 `arrive_time_point_` 。
3. 尝试获取取号的互斥锁 `getting_number_mutex` ，否则等待。根据 `getting_number` 取号，并递增 `getting_number++` ，释放互斥锁。
4. 尝试获取等待队列的互斥锁 `waiting_queue_mutex` ，否则等待。将自己的指针加入到等待队列 `waiting_queue` 中。释放互斥锁。
5. 对信号量 `customer_ready` 执行 V 操作，唤醒一个柜员线程。
6. 根据自身的成员变量 `called_future_.get()` ，等待有一位空闲柜员通知到自己，并得到将要服务自己的柜员编号 `served_by_` 。
7. 在得知自己被一位柜员叫号之后，对信号量 `teller_ready` 执行 P 操作，尝试获取一位空闲柜员为自己服务，否则阻塞。

8. 记录当前开始服务的时间点 `serve_time_point_` 。开始办理服务，线程休眠一定时间。
9. 办理结束，记录当前结束服务的时间点 `leave_time_point_` ，线程退出。

柜员线程的完整流程为：

1. 对信号量 `customer_ready` 执行 P 操作，尝试获取一位等待的顾客，否则阻塞一小段时间。如果获取失败，进入下一轮循环。
2. 尝试获取等待队列的互斥锁 `waiting_queue_mutex` ，否则等待。从等待队列 `waiting_queue` 中取出一位顾客指针，赋给自己当前所服务顾客指针 `serving_for` 。释放互斥锁。
3. 通过当前所服务的顾客指针调用叫号接口 `serving_for->notify_called()`，通知该顾客他正在叫号。
4. 对信号量 `teller_ready` 执行 V 操作，唤醒一个顾客线程。
5. 开始办理服务，线程休眠一定时间。
6. 办理完成后，将记录服务顾客总数的全局计数器加1 `served_customers_number.fetch_add(1)` ，并记录下此次服务流程的信息，加入到记录向量 `service_records_` 中，将当前所服务的顾客指针 `serving_for` 置空。
7. 循环执行步骤1。循环条件为“已服务顾客人数<顾客总人数”，否则才能退出。

主线程的运行过程：

1. 从 JSON 文件读取配置。
2. 从测试样例文件读取顾客信息，保存在顾客信息结构体中。
3. 初始化部分全局变量。
4. 根据柜员总人数，创建柜员对象，运行柜员线程。
5. 根据顾客信息结构体，创建顾客对象，运行顾客线程。
6. 主线程到达屏障，统一放行所有顾客线程。
7. 等待所有顾客线程和柜员线程结束。
8. 将顾客线程和柜员线程的信息输出到日志文件中。

5 文件结构说明

项目文件结构为：

```
.
├── build_and_run.sh          # 编译源代码、运行可执行文件的脚本文件
├── CMakeLists.txt            # CMake 构建文件，包含编译和链接的配置
├── config
|   └── config.json           # 有关读写文件路径的配置文件
├── data
|   ├── example_1.txt         # 简单测试样例
|   ├── example_2.txt         # 较复杂测试样例
|   ├── output_log2.txt       # 较复杂测试样例的输出信息
|   ├── customer_log2.txt     # 较复杂测试样例的顾客线程日志
|   └── teller_log2.txt       # 较复杂测试样例的柜员线程日志
├── include                   # 头文件目录
|   ├── customer.hpp
```

```
|   ├── globals.hpp
|   ├── json.hpp
|   ├── teller.hpp
|   └── utils.hpp
├── report
|   ├── report.md              # 实验报告
|   └── report.pdf             # 实验报告
├── run.sh                     # 运行可执行程序的脚本文件
├── scripts
|   ├── generate_data.py       # 生成随机测试样例的脚本
|   └── run.py                 # 编译运行脚本，作用与 run.sh 相同
└── src                        # 源代码目录
    ├── customer.cpp
    ├── main.cpp
    ├── teller.cpp
    └── utils.cpp
```

关于头文件和源文件的详细说明：

1. `customer.hpp, customer.cpp`：
    1. 声明和定义简单的顾客信息结构体 `struct CustomerInfo`，记录顾客姓名、到达时间、服务时间。
    2. 声明和定义顾客类 `class Customer`，主要实现顾客线程的创建和运行中的同步与互斥。
2. `teller.hpp, teller.cpp`：
    1. 声明和定义服务记录结构体 `struct ServiceRecord`，记录柜员服务的顾客姓名、号码、开门时间点、服务开始时间点、服务结束时间点。
    2. 声明和定义柜员类 `class Teller`，主要实现柜员线程的创建和运行中的同步与互斥。
3. `globals.hpp`：声明所有全局变量，全部封装在 `globals` 命名空间中。其中重要的信号量、互斥锁、等待队列需要被不同线程访问和修改。
4. `utils.hpp, utils.cpp`：声明和定义一些工具类函数，全部封装在 `utils` 命名空间中。
    1. `safe_print()`：线程安全的输出函数。
    2. `load_config_from_json()` 从 JSON 文件中读取配置。
    3. `parse_customer_info()`：从文件中解析顾客信息。
    4. `output_customer_thread_info()`：输出顾客线程的运行信息。
    5. `output_teller_thread_info()`：输出柜员线程的运行信息。
5. `json.hpp`：使用 `nlohmann/json` 的 JSON 解析库。项目地址https://github.com/nlohmann/json。
6. `main.cpp`：实现主线程的主文件。

6 样例测试

对于实验指导书中提供的简单测试样例，在文件 `data/example_1.txt` 中：

```
1 1 10
2 5 2
3 6 3
```

设置2位柜员，运行时输出信息为：

```
Total 3 customers will arrive.
Total 2 tellers will serve.
All tellers are ready.
All customers are ready.
[Customer 1] arrived, getting number 1.
[Teller 1] serving [Customer 1], number 1, for 10 ms.
[Customer 1] being served by [Teller 1] for 10 ms.
[Customer 2] arrived, getting number 2.
[Teller 2] serving [Customer 2], number 2, for 2 ms.
[Customer 2] being served by [Teller 2] for 2 ms.
[Customer 3] arrived, getting number 3.
[Teller 2] finished serving [Customer 2].
[Customer 2] finished.
[Teller 2] serving [Customer 3], number 3, for 3 ms.
[Customer 3] being served by [Teller 2] for 3 ms.
[Teller 2] finished serving [Customer 3].
[Customer 3] finished.
[Teller 1] finished serving [Customer 1].
[Customer 1] finished.
Total 3 / 3 customers have been served.
```

顾客线程运行日志：

```
customer  number  arrive at  serve at  leave at  wait time  serve time  teller
-------------------------------------------------------------------------------
1         1       1          1         11        0          10          1
2         2       5          5         7         0          2           2
3         3       6          7         10        1          3           2
```

柜员线程运行日志：

```
customer  number  begin at  end at  serve time
-----------------------------------------------
teller 1
1         1       1         11      10

teller 2
2         2       5         7       2
3         3       7         10      3
```

结果符合预期，程序运行正确。

使用 `scripts/generate_data.py` 可以生成随机测试样例。大幅增加顾客人数、适当增加柜员人数，调整到达时间和服务时间的随机数取值范围（间隔太小会导致顾客等待时间快速增长，银行拥挤。间隔太大会导致顾客到达即服务，没有等待），模拟高并发场景。直观上来讲，当“平均服务时间”明显大于“最晚到达时间/顾客总人数”（也即平均到达时间间隔），会出现明显的等待。

设置柜员人数为5，顾客人数为100，到达时间分布范围为(1,1000)，服务时间分布范围为(50,150)。由此生成较复杂的测试样例 `data/example_2.txt` ，运行输出结果见 `output_log.txt` ，顾客线程运行日志见 `data/customer_log.txt` ，柜员线程运行日志见 `data/teller_log.txt` 。可见后面的顾客等待时间很长，达到700以上，大概相当于队列前面平均有7位顾客。柜员人数对运行结果影响很大，当柜员人数从5增加到8时，大部分顾客的等待时间都为0，到达就能服务，最后的几位顾客在队列前面大约只有2名顾客。

7 思考题

（1）柜员人数和顾客人数对结果分别有什么影响?

答：柜员可以并发的服务多个顾客，柜员人数越多，顾客等待的时间越短，一名柜员单位时间内服务的顾客数量减少，最终顾客平均等待时间会接近于0，也就是到达时就有柜员空闲，可以立即服务。但是柜员数量增大到一定程度时，会出现部分柜员长期空闲，银行内顾客数量少于柜员数量，总是有空闲柜员不断阻塞等待，造成柜员资源浪费。从之前的测试中也可以看出，刚开始柜员数量较少时增加柜员数量，例如从2增加到5，可以显著增加服务效率、降低顾客等待时间，从5增加到8时，此时大部分顾客等待时间已经为0，此后再增加柜员数量，柜员就会过饱和，能够完全覆盖所有到达的顾客，不会对服务总时间造成影响了。

顾客人数较少时，柜员可以满足所有顾客需求，顾客几乎不需要排队。当顾客数量增多时，队列中的顾客数量增多，后到的顾客等待时间会更长，平均等待时间增大。尤其是顾客的平均服务时间大于顾客到达的间隔的时候，前面的顾客还没有服务完，后面就有顾客不断到达，等待队列不断增长，等待时间随着顾客的顺序而迅速增大。而当顾客的平均服务时间等于或者小于顾客到达的间隔，即使顾客人数很多，但是相邻两个顾客到达的间隔内，平均会有一个柜员服务完成，会从队列中取出两外一个人，因此等待队列中顾客数量会趋于稳定或处于较少状态，时间足够长后，即使总顾客人数较多，平均等待时间也会趋于稳定，不会迅速增长。

（2）实现互斥的方法有哪些？各自有什么特点？效率如何?

答：

信号量法。通过整型计数器控制对共享资源的访问，P 操作用于申请资源，V 操作用于释放资源，且 PV 操作本身是原子性的。可以有效避免忙等待，适用于多进程多线程场景。

锁变量。只通过设置一个锁变量，简单直观，但也有可能产生忙等待，容易因竞争条件或操作顺序错误导致锁变量失效。

轮转法。设置整型变量 turn，记录当前轮到哪一个进程进入临界区，但是效率较低，仅适用于两个进程的互斥，不适用于进程数量多或执行时间不均的场景。而且要求进程严格按照顺序轮流进入临界区，可能违反“空闲让进”条件——当无进程在临界区时，任何有权使用临界区的进程可以进入。且程序代码不对称，可能产生忙等待。

Peterson 算法。除了设置整型变量 turn，还设置数组 interested 表示进程是否想要进入临界区。既能保证互斥访问，又解决了轮转法的缺点，但是可能产生忙等待，也仅适用于两个进程的互斥。

禁止中断，是一种硬件方法。进程进入临界区前执行“关中断”指令，离开临界区后执行“开中断”指令，只有在发生时钟中断或其他中断时才会进程切换，优点是实现简单，缺点是把禁止中断的权限交给了用户，破坏了系统响应中断的能力，导致系统可靠性较差，仅使用与单处理器，不适用于多处理器。

硬件指令方法。利用处理机提供的专门的硬件指令，对一个字的内容进行检测和修改，读写操作由一条指令完成，保证读操作与写操作不被打断。其优点是实现较为简单，适用于任意数目的进程，支持进程中存在多个临界区（只需为每个临界区设立一个布尔变量）。缺点是会产生忙等待，浪费 CPU 时间。

8 实验感想

我在编写程序之前，一直将各种变量、信号量想象成为分属顾客和柜员的，但后来我发现，将这些所有变量想象成为银行里集中处理所有数据的“号码机器人”，这样只需要考虑号码机器和顾客之间的通信、号码机器与柜员之间的通信，将顾客与柜员之间的关系解耦，这样更容易设计程序的整体框架和通信机制。实验的核心在于信号量和互斥锁的设计，以及在何时获取和释放信号量、有由谁来操作、操作的顺序、互斥变量的访问，弄清楚这些问题就完成了程序的主要功能。另外，我在程序中使用到 C++20 的一些新特性，比如 `std::barrier` 和 `std::promise, std::future`，它们与程序中的通信机制符合较好，因此也为编写代码带来了较大便利，同时也促使我思考到了一些更加细节的问题，例如“所有顾客服务完成后，如何通知所有柜员线程退出下班？”，在 debug 过程中我也成功找到了合适的函数解决这个问题。本次实验增进了我对进程间同步与互斥的理解和掌握，以及信号量、互斥锁等机制的正确使用。