《Linux 内核分析课程项目》

专业名称 人工智能

姓　　名 郑凯航

学　　号 37220222203885

任课教师 陈毅东

目录

部分实现概括

1. 完成了项目要求，程序可以正常运行
2. 合理利用 c++的面向对象，构建多个抽象类，代码高内聚，低耦合
3. 合理利用了管道，信号量集，共享内存等 IPC 机制实现了进程同步
4. 利用大模型和提示词技术，构建复杂测试样例
5. 实现了多种调度算法
6. 使用常用的指标，在测试样例上进行复杂度比较分析并进行改进
7. 构建了完善的日志系统，用于调试和可视化输出
8. 对简单样例的输出进行合理的分析
9. 对实验细节部分进行一定修改，使其更符合真实情况
10. 代码结构完整，注释清晰
11. 代码实现过程可见 Elysiaaaaaaaa/linux_hw: linux 大作业
12. 代码本地运行可见 B 站，UID：474086052，6.12 截止后定时发送

实验目的

项目要求使用 Linux 高级 IPC 机制（包括信号量和共享内存）来模拟一个双向隧道内的交通流量控制系统，实现进程同步。

要求如下：

出于安全原因，隧道内一次不能超过 N 辆车，并且同一时间隧道内只能有单

向车流。

如果某个方向的车流正在通过，另一方向的车流必须等待当前方向的车辆全部离开后才能进入。

入口处的交通灯控制车辆的进入，入口和出口处的车辆检测器则可以检测交通流量。

当隧道已满或者对向车辆未完全驶出时，到达的汽车将不允许进入隧道，直到满足车辆进入隧道的条件。

隧道内的每辆车都可以访问和修改一个用以模拟隧道邮箱系统的共享内存段（可以看成是一个数组，访问操作操作包括：r 和 w），这样，隧道内的车辆就在进隧道后保持其手机通讯（隧道将阻塞手机信号）。

隧道外的汽车则不需要访问该共享内存段。

每辆车都由一个 Linux 进程来模拟。

当汽车在隧道内，它们的操作是允许并发的，并且必须以 Linux 高级 IPC 机制进行同步。

如果一个给定邮箱尚未因写操作而锁定，则对它的读操作可以并行，但对同一邮箱的写操作则同一时刻只允许一个。

实现环境

实现过程中使用到了如下环境配置

CMake: 3.10

c++: 11

wsl2

Ubuntu20.02

基于 linux IPC 机制完成，实现过程中使用到的 c/c++库如下：

输入输出相关

`<iostream>` → C++ 标准输入输出流（cin, cout）

`<fstream>` → 文件输入输出流（ifstream）

`<sstream>` → 字符串流（stringstream）

`<cstdio>` / `<stdio.h>` → C 标准输入输出

容器与算法

`<vector>` → 动态数组容器

`<map>` → 容器

`<algorithm>` → 常用算法

时间与日期

`<chrono>` → 高精度时间、计时器

`<ctime>` → C 风格时间处理

多线程同步

`<mutex>` → C++ 互斥锁

`<pthread.h>` → POSIX 线程库

`<semaphore.h>` → POSIX 信号量

系统与进程管理

`<unistd.h>` → 系统调用（sleep, fork）

`<sys/types.h>` → 系统数据类型（pid_t）

`<sys/sem.h>` → 信号量控制

`<sys/shm.h>` → 共享内存控制

`<sys/wait.h>` → 进程等待与回收

其他

`<random>` → 随机数生成器

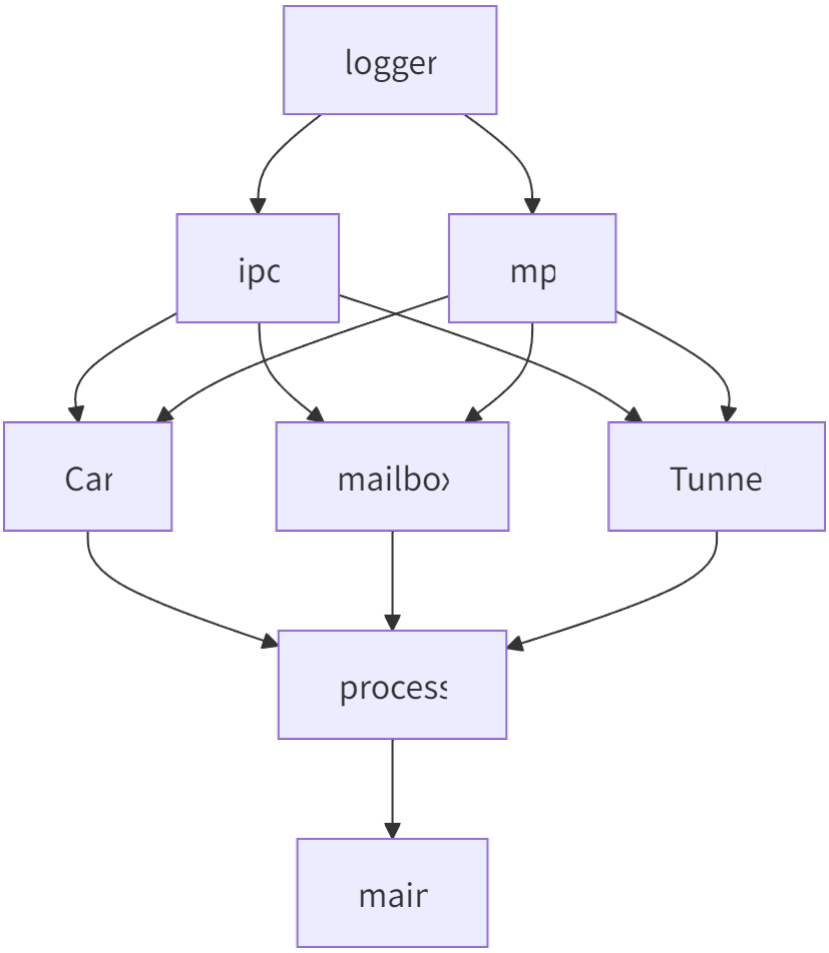
`<cstdlib>` / `<stdlib.h>` → C 标准库

`<cstring>` / `<string.h>` → C 字符串处理

`<string>` → C++ 字符串类

实现方法

本章中我将基于自底向上的逻辑介绍程序中各个功能的完整实现



实现过程中 logger 用于打印日志，其他的类中都使用其进行日志打印

ipc 和 mp 文件主要实现了相关基本函数的封装，加入了错误检测，当相关函数分配错误时，会打印日志并释放资源，中断程序

Car,mailbox,Tunnel 是程序中三个重要的类别，其中 Car 用来模拟车辆，记录车辆具体的操作以及其他关键属性。Tunnel 用来保存隧道相关属性，信号量以及共享内存区。mailbox 用来保存邮箱访问相关属性，信号量以及共享内存区，并实现邮箱访问互斥。

process 用来综合上述实现的三个类相关功能，模拟车辆进入隧道和访问邮箱的相关操作

基础功能封装

logger

`Logger` 类是一个简单的日志记录器，用于程序运行期间输出带时间戳和日志级别的信息。

主要功能如下：

13. **时间戳生成**：
    基于 `std::chrono::high_resolution_clock` 提供相对 `baseTime` 的毫秒级时间戳。
14. **日志级别管理**：
    支持 `INFO`、`WARN`、`ERROR` 三级日志级别，并能将其转化为字符串输出。
15. **统一输出接口**：
    `log()` 函数统一处理日志信息的拼接与输出到标准输出流（`std::cout`）。

`logger` 提供了三种级别的日志类型，分别为 `info`, `warn`, `error`，用来打印不同级别的日志

`setBaseTime()`用于设置基础时间 `baseTime`，其中模拟过程开始时为 0，以毫秒为单位

`log()`函数用来提供信息可视化，默认输出到终端，输出后进行缓冲区刷新

`getTimestamp()`用于计算当前时间和基础时间时间的时间间隔，以毫秒为单位

ipc.h

封装了 IPC 函数，主要用于简化对 Linux 下 System V 信号量和共享内存的管理，包括 **信号量（semaphore）** 和 共享内存（shared memory）

`sem_get()` 用于创建或获取一个 **信号量集**，并选择是否初始化以及初始化的值。

`sem_get_val()` 用于获取指定信号量的当前值。

`Wait()` 用于执行 P 操作（等待操作）。

`Signal()` 用于执行 V 操作（释放操作）。

`sem_del()` 用于删除信号量集。

`shm_init()` 用于获取（或创建）共享内存段。

`shm_conn()` 用于将共享内存映射到当前进程。

`shm_disconn()` 用于断开共享内存连接。

`shm_del()` 用于删除共享内存段。

这个模块主要是对 Linux IPC 的 **信号量** 和 **共享内存** 接口的封装，增加了错误日志记录（依赖 `Logger`）和程序异常退出保护

mp.h

这个模块主要对 Linux 的 **进程管理** 系统调用进行了简单封装，包括 `fork()` 和 `wait()`，并集成了日志记录。

`Fork()` 是对 Linux 系统调用 `fork()` 的封装，用于创建子进程。如果 `fork()` 失败（返回值 `<0`），使用 `Logger` 输出错误日志，并终止程序。

`Wait()` 是对 Linux 系统调用 `wait()` 的封装，用于等待子进程结束。

信号量与资源管理模块

Car.h

面向对象、用于模拟隧道中汽车调度与通信（IPC）的 C++ 模块，并用枚举 `Direction`、`State` 表示方向与状态

在初始化过程中直接使用输入流对 input 文件进行解析，提取相关信息，为了避免部分操作由于顺序问题（前一个操作的开始时间大于后一个操作开始时间）导致部分操作无法运行，在读取完后基于操作开始时间进行排序，尽可能的执行所有操作

此外还实现了相关答应函数，可以将车辆基本信息都打印出来

```
----------------------
Car ID: 1
Direction: Eastbound
tunnel_travel_time: 224 ms
Operations:
  Read operation: Time: 10, Mailbox: 5, Length: 20
  Read operation: Time: 10, Mailbox: 5, Length: 20
  Write operation: Data: hello, Time: 20, Mailbox: 5, Length: 5
  Write operation: Data: 2021nian, Time: 20, Mailbox: 3, Length: 8
  Write operation: Data: world, Time: 30, Mailbox: 5, Length: 5
----------------------
----------------------
Car ID: 2
Direction: Eastbound
tunnel_travel_time: 212 ms
Operations:
  Read operation: Time: 20, Mailbox: 5, Length: 10
  Write operation: Data: xinxixueyuan, Time: 20, Mailbox: 1, Length: 12
  Write operation: Data: xiamendaxue, Time: 40, Mailbox: 8, Length: 11
  Read operation: Time: 100, Mailbox: 7, Length: 30
----------------------
----------------------
```

具体细节如下

成员变量

`int car_id`：存储每辆车的唯一标识符。

`Direction direction_`：存储汽车行驶的方向（东行或西行）。

`start_time`：存储汽车开始穿越隧道的时间点。高精度的计时

`std::chrono::milliseconds cost_time`：存储预计穿越隧道的时间（单位：

毫秒）。

`State state`：表示汽车当前的状态，分为 `WAITING`、`INNER` 和 `OUT`。表示车辆正在等待进入隧道，已进入隧道，已经穿越隧道

`model_str`：存储汽车的手机内存

`operations`：存储一汽车操作（如读/写邮箱的操作）

`vector<int> m`：每个邮箱的读指针数组。

`adjusted_travel_time`：实际隧道穿越时间。在基本穿越时间的 30%范围内浮动

函数接口

`Car(int id, Direction direction, int fluctuation, int travel_time)` 构造函数，根据输入初始化 `Car` 对象

`int getCarId() const` 返回车辆 ID。

`std::string getDirectionStr() const` 返回车辆行驶方向。

`void add_operation(bool isWrite, int time, int mailbox, const std::string& data, int length)` 添加一次邮箱操作（读/写）。

`void show() const` 展示车辆全部的 operation

Tunnel

`Tunnel` 类是一个用于模拟双向隧道通行控制的类，提供隧道访问相关信号量主要功能如下：

- 控制隧道内 **车辆数量**，避免超载。
- 使用**信号量**和**共享内存**机制实现隧道方向和隧道内车辆数量的多进程间同步。 确保隧道内的车辆 **方向一致**，防止对向车辆冲突。

- 使用互斥锁来设置临界区，避免多个进程对，为了保证多个进程之间的互斥锁同步，tunnel 中仅保存信号量 ID，随后通过之前定义的 sem_get 获取或创建一个跨进程可见的信号量。来实现同步

成员变量

```
int tunnel_number_of_cars;  // 信号量：最大隧道车数
int total_number_of_cars_tunnel; // 信号量：隧道中总车数
int car_count_;              // 当前隧道内的车辆数量
Direction current_direction_; // 当前隧道的方向（东或西）
int mutex_;                  // 信号量：互斥锁
int block_;                  // 信号量：阻塞对向车辆
key_t car_count_key;
int mutex_;    // 用来保护内部状态
int block_;    // 用来阻塞不符合方向的车
```

函数接口

`Tunnel::Tunnel`

通过预定义的值生成多个共享资源的唯一 key， 并用这些 key 创建信号量

`show()`

用来打印当前隧道状态（方向、车辆数）。

mailbox

实现了一个 `mailbox` 类，模拟了具有读写操作的多邮箱系统，使用共享内存和信号量来同步多个进程对邮箱的访问。

为每个邮箱分配共享内存区域，并通过信号量控制对这些邮箱的访问，确保多个进程可以安全地进行读写操作。

成员变量

```
int semid;
int shmid;
int total_number_of_mailboxes;
int memory_segment_size;
```

```
char* shared_memory;
int* reader_counts; // 每个邮箱的读者计数
int reader_count_semid; // 用于保护读者计数的信号量
```

int semid;

与共享内存中邮箱相关的信号量 ID。用于同步对邮箱数据的访问，确保在同一时刻只有一个进程可以对邮箱执行写操作。

`int shmid;`

保存共享内存的 ID。共享内存用于在不同进程之间共享数据，`shmid` 是通过 `shm_init()` 创建共享内存时返回的标识符。该内存段包含多个邮箱的数据，并允许多个进程对邮箱进行读写操作。

`int total_number_of_mailboxes;`

记录了邮箱的总数。

`int memory_segment_size;`

指定每个邮箱所使用的内存段的大小。

`char* shared_memory;`

该变量是指向共享内存区域的指针。通过这个指针，程序能够直接访问共享内存。每个邮箱的数据存储在 `shared_memory` 中的不同位置。

`int* reader_counts;`

该变量是一个指针，指向一个整数数组，用于跟踪每个邮箱的读者数量。每当有进程读取邮箱时，相应的读者计数会增加。通过这个计数，可以判断是否有读者正在访问某个邮箱，并对并发操作进行控制。

`int reader_count_semid;`

保存了用于保护读者计数的信号量 ID。每当修改 `reader_counts` 数组时，都需要获得这个信号量，以确保多个进程访问该数组时不会发生数据竞争。读者计数的信号量在 `readMailbox` 方法中使用，以确保对读者计数的修改是线程安全的。

函数接口

mailbox::mailbox

当 mailbox 对象被创建时，首先根据传入的路径名 (pathname) 和项目 ID (proj_id)，通过 ftok 生成唯一的键值 (key_t)，分别为邮箱信号量、读者计数信号量以及共享内存生成不同的键。接着，使用 sem_get 函数为邮箱分配一个包含 num_mailboxes 个信号量的集合，并初始化为 1，确保邮箱操作的互斥访问；同样地，为保护读者计数，还单独创建一组信号量。

随后，程序通过 shm_init 创建一块共享内存，大小为所有邮箱数据区 (num_mailboxes * mem_size) 加上用于存储每个邮箱读者计数的额外空间 (num_mailboxes * sizeof(int))。创建完成后，使用 shm_conn 连接到这块共享内存，并将 shared_memory 指针指向它。最后，将指向读者计数区的 reader_counts 指针定位到共享内存中邮箱数据区之后的空间，并使用 memset 将所有读者计数初始化为 0。

mailbox::~mailbox()

当 mailbox 对象被销毁时，首先通过 shmdt 函数将当前进程与共享内存分离，若分离失败，会输出错误信息。接着，使用 semctl 函数对邮箱信号量 (semid) 和读者计数信号量 (reader_count_semid) 进行 IPC_RMID 操作，将它们从系统中移除，以释放系统资源。如果这些操作失败，同样会通过 perror 输出相应的错误信息。最后，使用 shmctl 对共享内存执行 IPC_RMID 操作，将整块共享内存从系统中删除，确保不会产生内存泄漏。

```
void mailbox::readMailbox(int mailbox_index, std::string& result,
int op_time, const
std::chrono::time_point<std::chrono::high_resolution_clock>&
start_time)
```

负责从指定邮箱读取数据

在读取邮箱数据时，首先通过 `Wait(reader_count_semid, mailbox_index)` 获取保护读者计数的信号量，以确保对读者计数的修改是安全的。接着，将对应邮箱的读者计数加一，如果当前是第一个读者，还需要额外获取该邮箱的访问信号量 `semid`，以避免写入操作的干扰。随后，计算从操作开始到当前所经过的时间，并根据需要调整操作时间，之后从共享内存中读取邮箱的数据内容。数据读取完成后，需要再次获取保护读者计数的信号量，将读者计数减一，如果当前是最后一个读者，则释放邮箱的访问信号量 `semid`，保证其他写入操作可以正常进行。

`void writeMailbox`

该函数负责向指定邮箱写入数据:

在进行邮箱写入操作时，首先通过 `Wait(semid, mailbox_index)` 获取对应邮箱的信号量，以确保在写入过程中不会有其他进程对该邮箱进行读写操作。随后，计算当前已消耗的操作时间，若尚未达到设定的操作时长，则使用 `usleep` 等待剩余时间，然后通过 `memcpy` 将要写入的数据拷贝到共享内存中。如果写入的数据长度超过邮箱容量，会对数据进行截断，并在末尾添加终止符 `'\0'` 以标识结束。接着，将该邮箱的读指针重置为 0，以便后续读取从开头开始。最后，写入操作完成后释放 `semid` 信号量，允许其他进程访问该邮箱。

`void show()`

该函数用于展示系统中所有邮箱的当前内容和读者计数。它通过遍历每个邮箱的索引，从 0 到 `total_number_of_mailboxes - 1`，依次获取各个邮箱的数据。为了获取每个邮箱的内容，它调用 `readMailbox` 函数，并传入一个虚拟的起始时间 `dummyStartTime`，且操作时间参数设为 0，以确保只是读取数据而不

产生额外延迟。读取完数据后，函数会拼接一条包含邮箱索引、邮箱中数据内容以及当前读者计数的消息，并通过 Logger::log 函数以信息级别（INFO）记录或输出这条日志，从而达到监控和展示系统当前状态的目的。

高级调度与控制模块

process

process 类是负责总体控制，它结合了隧道调度（Tunnel）、邮箱通信（mailbox）、汽车信息管理（cars），并利用 start_time 记录和追踪整个进程的时间维度信息，模拟车辆进出隧道

成员变量

```
Tunnel* tunnel;
mailbox* mail_box;
std::vector<Car> cars;
std::chrono::time_point<std::chrono::high_resolution_clock> start_ti
me; // 添加起始时间成员变量
```

Tunnel* tunnel

指向 Tunnel 类的指针，包含隧道控制相关的信号量。

用于管理隧道中汽车的调度和容量控制，比如判断是否允许车辆进入、离开隧道等。

mailbox* mail_box

指向 mailbox 类的指针。

负责车辆进程访问邮箱（共享内存、信号量）相关的内容

std::vector cars

Car 类型的动态数组。

保存当前系统中所有汽车的实例信息，包括它们的属性（如方向、速度、编号等），用于调度和管理。

`start_time`
高精度时间点,记录系统启动的起始时间，用于后续计算程序运行时间、汽车操作持续时间等。

函数接口

`process::process`
初始化 `process` 类的核心组件，包括隧道（`Tunnel`）、邮箱（`mailbox`）和车辆数组

首先程序通过 `ftok` 函数基于给定的 `pathname` 和 `proj_id` 计算出隧道共享内存的键值（`key_t`），接着使用 `shmget` 申请或获取一个共享内存段（`shmid_tunnel`）。成功获取共享内存后，程序调用 `shmat` 将这块共享内存挂载到当前进程的地址空间，并检查挂载是否成功。随后在共享内存上直接构造 Tunnel 对象，实现了多个进程共享同一个 Tunnel 实例的目的。
类似的，程序用相同的方式为 mailbox 模块分配和挂载共享内存：先通过 ftok 生成 mailbox 的共享内存键，再通过 shmget 获取内存段，通过 shmat 挂载到进程地址空间，最后使用 placement new 在共享内存中构造 mailbox 实例，初始化时传入邮箱数、内存大小、proj_id 和 pathname。
最后，构造函数为车辆数组 `cars` 预留空间，以便后续存储每个车辆对象。
`process::~process()`
因为 `Tunnel` 是通过 **placement new** 在共享内存中构造的，编译器不会自动调用析构，需要手动管理。因此显式调用 `Tunnel` 对象的析构函数来销毁

tunnel 中的内容，随后，它调用 shmdt 将 tunnel 占用的共享内存段从当前进程中分离（detach），并检查操作是否成功，若失败则输出错误信息。

接着用同样的方式处理 mailbox 对象：先显式调用其析构函数，再调用 shmdt 解除共享内存映射，并进行错误检查。

process::enter(Car *car)

函数的主要功能是控制车辆进入隧道的过程，并根据隧道的当前状态做出相应的处理。有两种处理模式，第一种使用红绿灯，其信号周期切换，另一种在一个方向车辆不为空时隧道方向始终不变

函数通过信号量 Wait(tunnel->mutex_, 0) 获取对隧道的访问权限，确保操作的线程在修改隧道状态时不会被其他线程干扰。然后，它进入一个 while 循环，尝试根据隧道的状态决定车辆是否可以进入隧道。

- **同向且空间充足**：如果隧道已经有车辆，但车辆的行驶方向与当前隧道的方向一致，并且隧道的容量未满，车辆也可以进入，系统更新车辆状态和时间戳。
- **方向不同**：如果隧道当前的行驶方向与车辆的方向相反，车辆需要等待。系统记录该车辆因方向不一致而无法进入隧道的日志。
- **隧道已满**：如果隧道已经达到最大容量，车辆同样需要等待，并且系统记录该车辆因隧道已满而无法进入的日志。

  每次判断完后，函数会释放信号量 Signal(tunnel->mutex_, 0)，允许其他线程或进程进行操作。如果车辆无法进入隧道，它会通过 Wait(tunnel->block_, 0) 进入阻塞状态，等待隧道条件改变后再次尝

试进入。成功进入隧道后，最后一次释放信号量，确保退出时不会影响其他进程。

`process::leave(Car *car)`
处理车辆离开隧道的过程，并对隧道状态进行适当更新。

16. **加锁操作**：函数首先通过 `Wait(tunnel->mutex_, 0)` 获取对隧道的访问权限，确保在更新隧道状态时不会有其他线程或进程干扰。
17. **车辆离开隧道**：接下来，车辆离开隧道时，更新隧道内的车辆数量 `tunnel->car_count_`，减少 1，并将车辆的状态 `car->state` 设置为“离开隧道”（`State::OUT`）。
18. **唤醒等待车辆**：如果隧道内的车辆数少于最大容量且有车辆在等待进入隧道，则调用 `Signal(tunnel->block_, 0)` 唤醒一个在相同方向上等待的车辆，让它进入隧道。
19. **解锁操作**：完成状态更新后，释放隧道的访问权限，通过 `Signal(tunnel->mutex_, 0)` 解锁，允许其他进程或线程进行操作。

`leave` 负责车辆能够离开隧道并更新隧道内的车辆计数，同时，如果隧道有空余空间且有车辆在等待，函数会唤醒相应的等待车辆。

`process::isGreenLight`
判断隧道是否为绿灯，决定车辆是否能继续行驶。

`process::switchDirection`

根据当前隧道的方向，将其切换为相反方向（东向 -> 西向 或 西向 -> 东向）。在更新隧道的方向后，通过信号量通知等待的车辆。

process::main_process

模拟多个车辆在隧道中行驶和操作邮箱系统的过程，并通过子进程实现每辆车的独立操作。通过多进程和管道等通信机制，模拟车辆进入、在隧道中通信、离开隧道的全过程。

20. **记录起始时间和日志初始化**：函数首先记录当前时间作为起始时间，并初始化日志系统，记录“PROCESS BEGAN”日志。
21. **创建管道**：使用 `pipe(pipefd)` 创建管道，准备进行父子进程间的通信。若创建失败，则终止程序。
22. **创建子进程**：通过 `Fork()` 创建多个子进程，每个子进程对应一个车辆。父进程会继续执行循环，直到所有子进程启动完毕。
23. **子进程逻辑**：

- **进入隧道**：每辆车会通过 `enter(&cars[i])` 函数进入隧道。
- **车辆操作**：车辆在隧道中会进行一系列的读写操作，操作数据来自于预设的 `cars[i].operations`。每次操作时，车辆会记录当前时间与预期时间的差值，若超过预定时间则视为超时。
- **执行读写操作**：对于写操作，车辆将数据写入邮箱系统；对于读操作，车辆会从邮箱读取数据并更新自己的 `model_str` 字符串。
- **模拟休眠**：在车辆操作完成后，根据预计的结束时间计算休眠时长，确保车辆在正确的时间点离开隧道。
- **离开隧道**：车辆通过 `leave(&cars[i])` 函数离开隧道，退出隧道前还会通过管道将数据传递给父进程。

24. **父进程逻辑**：

- **关闭写端**：父进程关闭管道的写端，准备从管道读取数据。
- **等待子进程**：父进程等待所有子进程执行完毕，确保所有车辆的操作都完成。
- **读取管道数据**：父进程从管道中读取每辆车的最终数据，并将这些数据更新到每辆车的 `model_str` 字段中。

main

多进程模拟系统的主函数，模拟了多个汽车进入隧道后进行邮箱操作的场景。整个流程涉及汽车的初始化、隧道的管理、邮箱的读写操作以及进程间的通信。

1. 读取配置文件

程序通过 `txt_reader` 类读取传入的配置文件（通过命令行参数传递）。`txt_reader` 类的主要职责是从文件中解析并提取相关的配置数据。

2. 初始化 `process` 对象

`process` 类负责真题模拟。包括隧道的初始化、管理汽车的进入与离开以及邮箱操作的处理。通过将邮箱数量和内存大小等参数传递给 `process` 类的构造函数，程序初始化了相关的邮箱资源。`process` 对象还包含了一个 `cars` 容器，用来存储所有的汽车对象。

3. 汽车信息初始化

循环遍历每辆车，通过 `reader.input_car()` 方法获取汽车的详细信息，并将这些信息传递给 `process` 类进行初始化。每辆车的初始化过程中，程序调用

`p.init_car(reader)` 来设置汽车的相关属性，并将汽车对象添加到 `process` 类的 `cars` 容器中。初始化完成后，程序展示了所有汽车的相关信息，确保每辆车的状态正确。

4. 主进程执行

调用 `p.main_process()` 启动模拟的主流程。在这个函数中，程序为每辆车创建了一个子进程，这些子进程模拟了汽车进入隧道、进行邮箱操作并最终离开隧道的过程。使用了多进程设计，每辆车的操作并行执行

5. 子进程行为

每个子进程代表一辆汽车，它执行以下步骤：

- 汽车首先进入隧道，并记录进入时间。
- 进入隧道后，汽车会执行邮箱操作，包括读取和写入邮箱。操作过程中，程序会检查是否超时，如果超时，则记录警告信息。
- 完成操作后，汽车离开隧道，并将其最终状态通过管道发送给父进程。此时，子进程退出，避免继续执行父进程的代码。

6. 父进程行为

父进程负责等待所有子进程完成操作。在所有子进程结束后，父进程关闭管道的写端，确保数据不会丢失。接着，父进程从管道读取每辆车的操作结果，并将其展示出来，最终输出整个模拟过程的结果。

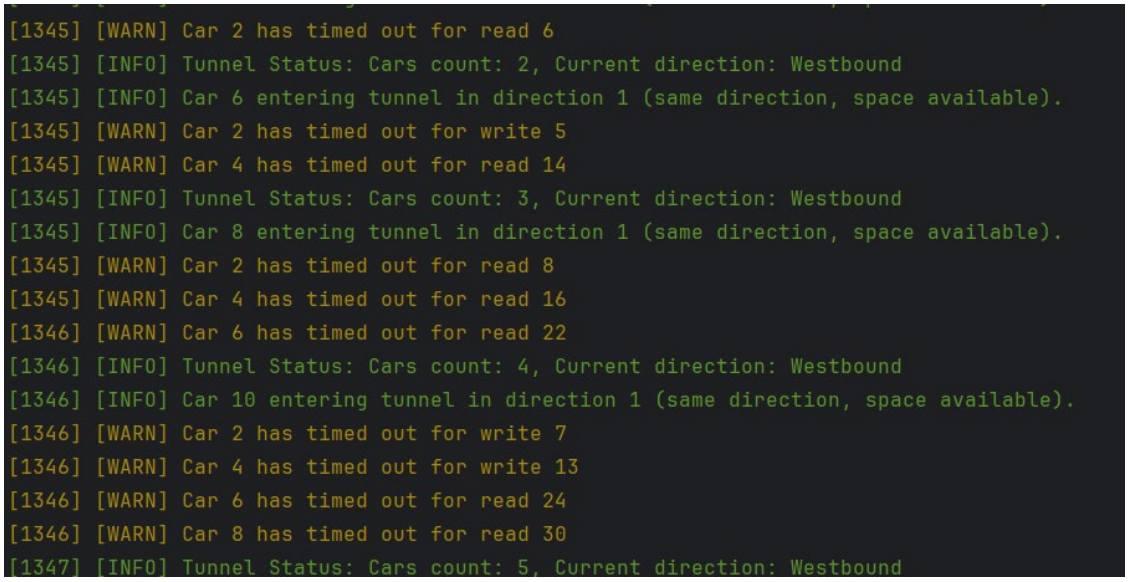
7. 程序退出

程序在所有子进程完成任务后正常退出。此时，所有汽车的操作日志已被记录，并且邮箱系统的状态已被展示。

细节更改

指令超时

由于输入中制定了具体指令完成时间，但是在实际调度中往往很难预测车辆何时到达隧道，因此在程序运行时，若检测到当前指令无法完成，则直接跳过这条指令，开始执行下一条



进入时间

为使得模拟符合真实情况，设置了函数，限制汽车进入隧道的时间最小为 `cars[i].car_id%10 * 100`，从而有效提升了实验结果中的各项指标

```
    if(ex_input) {
//      休眠一段时间，直到car_id % 10 * 100才进入隧道
        auto car_current_time : time_point<...>  = std::chrono::high_resolution_clock::now();
        auto car_elapsed_time : long long  = std::chrono::duration_cast<std::chrono::milliseconds>(
                d: car_current_time - start_time).count();
        if (cars[i].car_id%10 * 100 - car_elapsed_time > 0) {
            usleep((cars[i].car_id%10 * 100 - car_elapsed_time) * 1000);
        }
    }
```

读取算法

原本的要求为接收到一条命令时，占有对应的邮箱直到到达指定时间，经过试验后发现指令完成率非常低，因此修改逻辑为休眠，直到到达指定时间的前 5 毫秒才占有对应邮箱，经过实验发现，效率稍有提升

```
for (const auto& op : Operation const &  : cars[i].operations) {
    t++;
    auto current_time : time_point<...>  = std::chrono::high_resolution_clock::now();
    auto elapsed_time : long long  = std::chrono::duration_cast<std::chrono::milliseconds>(
            d: current_time - start_time).count();
    if (elapsed_time > op.time) {
        if(op.isWrite){
            Logger::log( level: LogLevel::WARN,
                         message: "Car " + std::to_string( val: cars[i].car_id) + " has timed out for writ
        }else{
            Logger::log( level: LogLevel::WARN,
                         message: "Car " + std::to_string( val: cars[i].car_id) + " has timed out for read
        }
    } else {
        auto remaining_time : long long  = op.time - elapsed_time;
        if (remaining_time > 10) {
```

红绿灯算法

```
auto last_switch_time : time_point<...>  = start_time;
while (true) {
    auto current_time : time_point<...>  = std::chrono::high_resolution_clock::now();
    auto elapsed_time : long long  = std::chrono::duration_cast<std::chrono::milliseconds>( d: curre
    if (int(elapsed_time/this->switch_time)%2&&use_rg) {
        switchDirection();
        last_switch_time = current_time;
    }

    // 检查是否所有子进程都已退出
    int status;
    pid_t pid = waitpid(-1, &status, WNOHANG);
    if (pid == -1) {
        // 没有更多子进程
        break;
    }
}
```

引入了红绿灯算法，相较于两个方向相互抢占，使用红绿灯进行调度可能会提升效率，但是结果发现再加入后效果略有降低，具体分析见下文

IPC 机制使用

IPC 资源标识

在相关文件下定义了如下参数偏移量，以确保生成的标识符唯一

```
#define PROJ_MUTEX_KEY_OFFSET 0
#define PROJ_BLOCK_KEY_OFFSET 1
#define PROJ_CARCOUNT_KEY_OFFSET 2
#define PROJ_MAXCARS_KEY_OFFSET 3
#define PROJ_TOTALCARS_KEY_OFFSET 4
#define PROJ_SEMKEY_KEY_OFFSET 5
#define PROJ_READER_KEY_OFFSET 6
#define PROJ_MEMORY_KEY_OFFSET 7
#define PROJ_SHM_MAILBOX_OFFSET 8
#define PROJ_DIRECTION_TUNNEL_OFFSET 9
#define PROJ_ZERO_CAR_OFFSET 10
#define PROJ_SHM_TUNNEL_OFFSET 11
```

mailbox

邮箱访问控制

使用了基于读者优先的策略

每个邮箱（mailbox）分配一个信号量 `semid[i]`。

当进行写入或第一个读者进入时，必须 **Wait(semid, i)**，锁住邮箱。

写操作完成或最后一个读者退出时，**Signal(semid, i)**，释放邮箱。

从而确保写操作时，没有读者或其他写者。多个读者可以并发读，但一旦有写入者，必须独占。

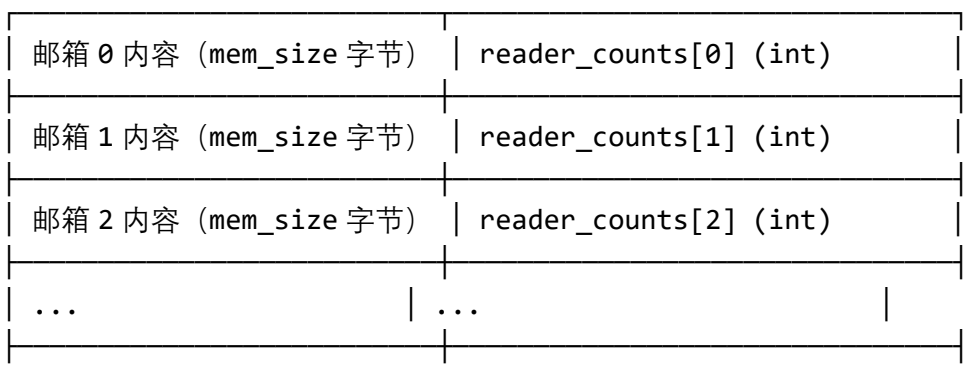
读者计数保护

防止两个或多个读者进程同时修改 `reader_counts[i]`，导致加 1 时丢失或读者数量判断错误，进而错误加锁或解锁邮箱。

邮箱共享内存

为了同步不同进程中的邮箱内容，使用了共享内存来作为邮箱内容的存储和交换数据的手段。

内存结构：



```
| 邮箱 N-1 内容（mem_size 字节） | reader_counts[N-1] (int)      |
└──────────────────────────────┴──────────────────────────────┘
```

前半段：`num_mailboxes * mem_size` 字节，存放每个邮箱的实际数据。

后半段：`num_mailboxes * sizeof(int)` 字节，存放 `reader_counts`，记录：

- 当前邮箱的活跃读者数量；
- 读到的位置偏移（通过 `mailbox_index + total_number_of_mailboxes` 存偏移量）。

邮箱访问流程

25. **读者进入**

- 锁 `reader_count_semid[i]`，`reader_counts[i]++`。
- 如果第一个读者 → 锁 `semid[i]`（阻止写入）。
- 解锁 `reader_count_semid[i]`。

2. **读写邮箱**

- 基于共享内存中的数据结构，以字符形式读取数据

3. **读者退出**

- 锁 `reader_count_semid[i]`，`reader_counts[i]--`。
- 如果最后一个读者 → 解锁 `semid[i]`。
- 解锁 `reader_count_semid[i]`。

4. **写者写入**

- 直接锁 `semid[i]`（保证没有读者和写者）。
- 写入后，解锁 `semid[i]`。

保护对象	信号量名	功能描述
邮箱访问互斥	`semid`	保障同一时间只有写者或多个读者在操作邮箱数据。
读者计数保护	`reader_count_semid`	防止多线程同时修改 `reader_counts[i]` 导致计数混乱。

Tunnel

Tunnel 本身类似一个信号量集合，内部定义了有关隧道汽车数量，隧道方向，隧道是否为空，互斥访问隧道等相关的信号量，其具体使用在 process 类中。

信号量	用途
`mutex_`	隧道访问互斥
`block_`	分别限制两个方向隧道车辆容量，阻塞/唤醒等待进入的车辆
`direction_changed_`	分别控制两个方向是否可以通行
`zero_car_`	分别监测两个方向上隧道清空状态，与红绿灯方向切换有关

process

隧道状态同步

车数、方向

需要多进程共享的隧道状态有：

- **tunnel->car_count_** 当前隧道里的车辆数量
- **tunnel->current_direction_** 当前允许通过的方向（东向 Eastbound / 西向 Westbound）

因为多个进程（对应每辆车）要同时读写这两个状态，因此使用了共享内存+信号量机制，将整个 Tunnel 放在共享内存中，从而避免了进行繁琐的同步

同时为了避免竞争条件，需要保证对共享变量的互斥访问，这一部分使用了信号量进行保护

隧道状态	共享方式	同步方式
车数 car_count_	共享内存（shmget + shmat）	信号量 mutex_ + block_
方向 current_direction_	共享内存（shmget + shmat）	信号量 direction_changed_ + zero_car_

共享内存部分

```
key_t shm_key_tunnel = ftok(pathname, proj_id + PROJ_SHM_TUNNEL_OFFSET);
int shmid_tunnel = shmget(shm_key_tunnel, sizeof(Tunnel), IPC_CREAT | 0666);
void *shmaddr_tunnel = shmat(shmid_tunnel, nullptr, 0);
tunnel = new (shmaddr_tunnel) Tunnel(proj_id, pathname);
```

shmget 分配一个共享内存段，用于存放 Tunnel 结构体。 shmat：将共享内存段挂载到进程地址空间。随后使用 placement new 直接在共享内存上构造 Tunnel 对象，保证多进程访问的是同一块物理内存。这样，不管哪个进程访问

`tunnel->car_count_` 或 `tunnel->current_direction_`，用的都是同一份内存。

信号量同步部分

因为有多个进程要同时读写共享内存，因此需要使用信号量进行同步，避免车辆数和方向出现竞争问题

代码里通过 `Wait()` 和 `Signal()` 实现互斥：

```
Wait(tunnel->mutex_, 0);   // 上锁
// 修改 car_count_ 和 current_direction_
Signal(tunnel->mutex_, 0); // 解锁
```

mutex_ 用于保证对 car*count* 和 current*direction* 的互斥访问

block_ 用于挂起等待的车辆，当有位置空出或方向切换时唤醒

direction_changed_ 用于控制红绿灯模式下的方向切换

zero_car_ 用于监测隧道清空，用于方向切换时等待隧道清零

无红绿灯模式下的同步方式

在没有红绿灯调度的情况下，车量基于间接通信的方式来通过隧道。需要满足如下要求：

26. 保证同一时间隧道内只有一种方向的车
27. 不能发生两方向车同时进入隧道
28. 支持多辆同方向的车同时通过，但是不能超过隧道容量

    主干代码：

```
//        无红绿灯
        Wait(tunnel->mutex_, 0); // 获取信号量
        while (true) {
            if (tunnel->car_count_ == 0) {
                // 隧道没有车，设置方向并进入
                if(tunnel->current_direction_ != car->direction_){
                    switchDirection();
                }
                tunnel->car_count_ += 1;
                if(car->direction_ == Direction::Eastbound){
                    Wait(tunnel->block_,0);
                }else{
                    Wait(tunnel->block_,1);
                }
                car->start_time = std::chrono::high_resolution_clock::
now();
                car->state = State::INNER;
                break;
            } else if (tunnel->current_direction_ == car->direction_ &
&
                       tunnel->car_count_ < maximum_number_of_cars_in_
tunnel) {
                // 同一方向且未达到最大容量，进入
                (tunnel->car_count_)++;
                if(car->direction_ == Direction::Eastbound){
                    Wait(tunnel->block_,0);
                }else{
                    Wait(tunnel->block_,1);
                }
                break;
            } else if (tunnel->current_direction_ != car->direction_)
{
                // 方向不同，等待
                Signal(tunnel->mutex_, 0); // 释放信号量
                if(car->direction_ == Direction::Eastbound){
                    Wait(tunnel->direction_changed_,0);
                    Signal(tunnel->direction_changed_,0);
                }else{
                    Wait(tunnel->direction_changed_,1);
                    Signal(tunnel->direction_changed_,1);
                }
                Wait(tunnel->mutex_, 0); // 再次获取信号量
            } else {
                // 隧道已满，等待
```

```
                Signal(tunnel->mutex_, 0); // 释放信号量
                if(car->direction_ == Direction::Eastbound){
                    Wait(tunnel->block_,0);
                    Signal(tunnel->block_,0);
                }else{
                    Wait(tunnel->block_,1);
                    Signal(tunnel->block_,1);
                }
                Wait(tunnel->mutex_, 0); // 再次获取信号量
            }
        }
        Signal(tunnel->mutex_, 0); // 释放信号量
```

信号量的使用主要集中在 enter 方法中，在不使用红绿灯时，以 tunnel->direction_changed_, tunnel->mutex_, tunnel->block_ 进行同步

tunnel->mutex_用来保证同一时刻仅有一辆车驶入/使出，保护车数 car_count_
和方向 current_direction_ ，避免竞争

当遇到隧道容量已满时，必须阻止车辆进入，此时使用到了 tunnel->block_进行阻塞，为了适配红绿灯，此处使用了两个信号量组成的信号量组，分别代表两个方向

为了避免进程在阻塞期间继续占有 tunnel->mutex_，所以需要对 tunnel->mutex_进行释放，等到 tunnel->block_被激活后再次占有 tunnel->mutex_

tunnel->direction_changed_ 用来限制车辆方向必须与隧道方向一致，同样为了适配红绿灯而采用了两个方向

为了避免一个进程在一段时间内通过多个信号量从而在错误时间进入隧道，enter 代码主题设置在一个循环中，只有当某一时刻满足进入条件，车辆才能进入

有红绿灯模式下的同步方式

红绿灯（traffic light）本质上是**全局调度器**，通过集中控制简化了进程之间的协作。

同步需求变成：

29. 红灯对应方向的车必须等待，直到信号灯变换且没有对向车行驶
30. 绿灯方向可以自由通过，若在隧道中信号灯改变，依旧正常运行直到出隧道

    在这种情况下，需要额外的信号量来检测 1. 隧道方向是否切换以及 2. 如果信号量切换，隧道中反方向的车是否已经走完

    为此，引入了 direction_changed_和 **zero_car_**

    direction_changed_是一个信号量组，0 号标识东方向通行，1 则标识西向通行，在任何时刻必须一个是 1 一个是 0

    zero_car_ 用于监测隧道清空，隧道清零时被激活

```
//    有红绿灯
    Wait(tunnel->mutex_, 0); // 获取信号量
    while (true) {
        if (isGreenLight(car->direction_) && tunnel->car_count_ < max
imum_number_of_cars_in_tunnel) {
            // 同一方向且未达到最大容量，车辆可以进入
            if(tunnel->car_count_==0){
                if(car->direction_==Direction::Eastbound)
                    Signal(tunnel->zero_car_, 0);
                else
                    Signal(tunnel->zero_car_, 1);
            }
            (tunnel->car_count_)++;
            if(car->direction_ == Direction::Eastbound){
                Wait(tunnel->block_,0);
            }else{
                Wait(tunnel->block_,1);
```

```
            }
            break;
        } else if (tunnel->current_direction_ != car->direction_) {
//            方向不同
            Signal(tunnel->mutex_, 0); // 释放信号量
            if(car->direction_ == Direction::Eastbound) {
                Wait(tunnel->direction_changed_, 0); // 等待方向变化的
信号量
                Signal(tunnel->direction_changed_, 0); // 等待方向变化
的信号量
                Wait(tunnel->zero_car_, 1); // 等待对方向没车
                Signal(tunnel->zero_car_, 1);
            }
            else{
                Wait(tunnel->direction_changed_, 1); // 等待方向变化的
信号量
                Signal(tunnel->direction_changed_, 1); // 等待方向变化
的信号量
                Wait(tunnel->zero_car_, 0); // 等待对方向没车
                Signal(tunnel->zero_car_, 0);
            }
            Wait(tunnel->mutex_, 0); // 再次获取信号量
        } else {
//            车容量
            Signal(tunnel->mutex_, 0); // 释放信号量
            if(car->direction_ == Direction::Eastbound){
                Wait(tunnel->block_,0);
                Signal(tunnel->block_,0);
            }else{
                Wait(tunnel->block_,1);
                Signal(tunnel->block_,1);
            }
            Wait(tunnel->mutex_, 0); // 再次获取信号量
        }
    }
    Signal(tunnel->mutex_, 0); // 释放信号量
```

首先介绍 direction_changed_使用

31. 初始化

在 process 构造函数中，direction_changed_的值被设置为[1,0]，表示当前方向为东，

32. 方向切换

在 switchDirection 函数中，将 direction_changed_两个方向的值同时取反

```
void process::switchDirection() {
    if(tunnel->current_direction_ == Direction::Eastbound){
        tunnel->current_direction_ = Direction::Westbound;
        Signal(tunnel->direction_changed_,1);
        Wait(tunnel->direction_changed_,0);
    }else{
        tunnel->current_direction_ = Direction::Eastbound;
        Signal(tunnel->direction_changed_,0);
        Wait(tunnel->direction_changed_,1);
    }
}
```

4. enter

direction_changed_ 仅在当前车辆方向和隧道方向不同时起作用，通过连续的 wait 和 Singal，进程可以在对应方向的信号量设置为 1 时通过，且使得信号量的值不变，其他同方向的车也可继续通过

```
Wait(tunnel->direction_changed_, 0); // 等待方向变化的信号量
Signal(tunnel->direction_changed_, 0); // 等待方向变化的信号量
```

zero_car_ 使用

zero_car_ 用于监测隧道对应方向是否清空，用于方向切换时等待隧道清零，避免隧道中同时出现对向的车，当隧道为空时值为 1，否则为 0

33. **zero_car_** 初始设为[1,1]

34. 当车辆进入时若隧道中没有其他车辆，作为当前隧道中唯一的车辆，将对应方向 **zero_car_** 设为 0

35. 当车辆离开时若隧道中没有其他车辆，作为当前隧道中最后的车辆，将 **zero_car_** 对应分量设为 1

36. 红绿灯切换时，车辆进程首先要验证

```
Wait(tunnel->direction_changed_, 0); // 等待方向变化的信号量
Signal(tunnel->direction_changed_, 0); // 等待方向变化的信号量
Wait(tunnel->zero_car_, 0); // 等待隧道空的信号量
Signal(tunnel->zero_car_, 0); // 等待隧道空的信号量
```

测试及分析

以如下输入为例

```
2
1
200
5
30
car 1 0
w 'start_op' 20 1
r 10 15 1
w 'mid_op' 30 3
r 15 35 1
w 'end_op' 25 5
r 12 18 5
end
car 2 1
r 15 25 2
w 'tunnel_start' 22 4
r 13 17 4
w 'tunnel_end' 28 5
r 16 22 5
end
```

数据读取部分

前五行分别为

汽车总数 隧道最大汽车容量 穿过隧道的时间 邮箱数量 每个邮箱的最大字数

首先是程序对于输入的解析，前五个参数正确被解析

随后是两辆车的信息，ID 分别为 1，2，方向部分 0 代表 East，1 代表 West

之后是通过时间，可以看到 192,238 均在合理范围内

随后是两辆车的具体读写过程，可以看到具体条目与输入一致，顺序则按照时间排序，京可能贪心的满足所有需求

```
elysia@DESKTOP-R0QJMOS:/mnt/d/Code/c++/linux_hw$ ./linux_hw ./input/test.txt ./linux_hw ./input/test.txt
total_number_of_cars_tunnel: 2
maximum_number_of_cars_in_tunnel: 2
tunnel_travel_time: 200
total_number_of_mailboxes: 5
memory_segment_size: 30
----------------------

----------------------
Car ID: 1
Direction: Eastbound
tunnel_travel_time: 238 ms
Operations:
  Read operation: Time: 15, Mailbox: 1, Length: 10
  Read operation: Time: 18, Mailbox: 5, Length: 12
  Write operation: Data: start_op, Time: 20, Mailbox: 1, Length: 8
  Write operation: Data: end_op, Time: 25, Mailbox: 5, Length: 6
  Write operation: Data: mid_op, Time: 30, Mailbox: 3, Length: 6
  Read operation: Time: 35, Mailbox: 1, Length: 15
----------------------
----------------------
Car ID: 2
Direction: Westbound
tunnel_travel_time: 192 ms
Operations:
  Read operation: Time: 17, Mailbox: 4, Length: 13
  Write operation: Data: tunnel_start, Time: 22, Mailbox: 4, Length: 12
  Read operation: Time: 22, Mailbox: 5, Length: 16
  Read operation: Time: 25, Mailbox: 2, Length: 15
  Write operation: Data: tunnel_end, Time: 28, Mailbox: 5, Length: 10
----------------------
[0] [INFO] PROCESS BEGAN
```

模拟部分

红绿灯调度

```
[0] [INFO] PROCESS BEGAN
[0] [INFO] Tunnel direction switched to Eastbound
[1] [INFO] Car 1 entering tunnel in direction 0 (same direction, spac
e available).
[1] [INFO] Tunnel Status: Cars count: 1, Current direction: Eastbound
[1] [WARN] Car 2 waiting due to opposite direction (direction 1), tun
nel occupied by direction 0.
[6] [INFO] Car 1 lock 1
[15] [INFO] Car 1 Reader: .
[15] [INFO] Car 1 lock 5
[18] [INFO] Car 1 Reader: .
[18] [INFO] Car 1 lock 1
[20] [INFO] Car 1 Reader: .
[21] [INFO] Car 1 lock 5
[25] [INFO] Car 1 Reader: .
[25] [INFO] Car 1 lock 3
[31] [INFO] Car 1 Reader: .
[31] [INFO] Car 1 lock 1
[35] [INFO] Car 1 Reader: start_op.
[200] [INFO] Tunnel direction switched to Westbound
[239] [INFO] Car 1 Leave.
[239] [INFO] Car 2 entering tunnel in direction 1 (same direction, sp
ace available).
[239] [INFO] Tunnel Status: Cars count: 1, Current direction: Westbou
nd
[239] [WARN] Car 2 has timed out for read 4
[240] [WARN] Car 2 has timed out for write 4
[240] [WARN] Car 2 has timed out for read 5
[240] [WARN] Car 2 has timed out for read 2
[240] [WARN] Car 2 has timed out for write 5
[400] [INFO] Tunnel direction switched to Eastbound
[430] [INFO] Car 2 Leave.
[431] [INFO] Car 1 Start : 0 Should do: 6 Had do: 6 Reader: start_op.
[431] [INFO] Car 2 Start : 238 Should do: 5 Had do: 0 Reader: .
[432] [INFO] Mailbox: 1 Info: Data: "start_op", Reader Count: 0
[432] [INFO] Mailbox: 2 Info: Data: "", Reader Count: 0
[432] [INFO] Mailbox: 3 Info: Data: "mid_op", Reader Count: 0
[432] [INFO] Mailbox: 4 Info: Data: "", Reader Count: 0
[432] [INFO] Mailbox: 5 Info: Data: "end_op", Reader Count: 0
[432] [INFO] Global Finish Rate: 0.545455
```

```
[433] [INFO] Mean Start Time: 119.000000
[433] [INFO] PROCESS FINISH
```

37. **01 时刻**
    系统启动，日志记录“PROCESS BEGAN”；
    隧道初始方向设置为东方
    车辆 1 进入隧道，方向为 0（代表东行），且此时隧道为空，记录“Car 1 entering tunnel in direction 0 (empty tunnel)”；
    隧道状态更新为有 1 辆车，当前方向为东行，“Tunnel Status: Cars count: 1, Current direction: Eastbound”；
    车辆 2 因车辆 1 在隧道内且方向相反（方向 1，代表西行）而等待进入，记录“Car 2 waiting due to opposite direction (direction 1), tunnel occupied by direction 0”。

38. **6 - 35 时刻**
    车辆 1 对多个邮箱进行操作，依次记录“Car 1 lock 1”“Car 1 Reader:. ”“Car 1 lock 5”等，涉及对邮箱 1、5、3 等的锁定和读取操作。
    车辆 1 在隧道内根据任务需求对不同邮箱执行读写操作，这些操作按时间顺序进行，每次操作都先锁定邮箱，操作完成后释放锁，保证对共享邮箱的操作满足读写互斥和同步要求。

39. **200 时刻**
    由于红绿灯设置，此时隧道内部方向反转，但是由于 1 车还在隧道内部，因此 2 车必须等待

40. 239 时刻
    车辆 1 离开隧道，记录“Car 1 Leave”；
    此时隧道方向合适，且没有反方向的车，因此车辆 2 进入隧道，方向为 1（西行），此时隧道为空，记录“Car 2 entering tunnel in direction 1 (empty tunnel)”，隧道状态更新为有 1 辆车；
    由于车辆 2 对多个邮箱的读写操作出现超时，记录“Car 2 has timed out for read 4”“Car 2 has timed out for write 4”等多条超时信息。

41. 400 时刻
    预定时间到达，隧道方向再次反转

42. 430 时刻
    车辆 2 离开隧道，记录“Car 2 Leave”；

43. 431 时刻及以后
    父进程展示隧道中相关信息，并进行性能分析

车辆邮箱方面，Car1 邮箱读取了 1 号邮箱中自己写入的 start_op，而 car2 由于没有执行任何指令，邮箱为空

隧道邮箱中，1 号邮箱，3 号邮箱和 5 号邮箱分别记录了 car1 写入的信息

性能分析方面，Car1 完成了全部 6 条指令，而 car2 由于等待红绿灯，所有指令都没有完成，最后的全部指令完成率是 54.54%，平均启动时间是

非红绿灯调度

```
[0] [INFO] PROCESS BEGAN
[0] [INFO] Tunnel direction switched to Eastbound
[1] [INFO] Car 1 entering tunnel in direction 0 (empty tunnel).
[1] [INFO] Tunnel Status: Cars count: 1, Current direction: Eastbound
[1] [WARN] Car 2 waiting due to opposite direction (direction 1), tun
nel occupied by direction 0.
[5] [INFO] Car 1 lock 1
[15] [INFO] Car 1 Reader: .
[16] [INFO] Car 1 lock 5
[18] [INFO] Car 1 Reader: .
[18] [INFO] Car 1 lock 1
[20] [INFO] Car 1 Reader: .
[20] [INFO] Car 1 lock 5
[25] [INFO] Car 1 Reader: .
[25] [INFO] Car 1 lock 3
[30] [INFO] Car 1 Reader: .
[30] [INFO] Car 1 lock 1
[35] [INFO] Car 1 Reader: start_op.
[239] [INFO] Tunnel direction switched to Westbound
[239] [INFO] Car 1 Leave.
[239] [INFO] Car 2 entering tunnel in direction 1 (empty tunnel).
[239] [INFO] Tunnel Status: Cars count: 1, Current direction: Westbou
nd
[240] [WARN] Car 2 has timed out for read 4
[240] [WARN] Car 2 has timed out for write 4
[240] [WARN] Car 2 has timed out for read 5
[240] [WARN] Car 2 has timed out for read 2
[241] [WARN] Car 2 has timed out for write 5
[431] [INFO] Tunnel direction switched to Eastbound
[431] [INFO] Car 2 Leave.
[432] [INFO] Car 1 Start : 1 Should do: 6 Had do: 6 Reader: start_op.
[432] [INFO] Car 2 Start : 239 Should do: 5 Had do: 0 Reader: .
[432] [INFO] Mailbox: 1 Info: Data: "start_op", Reader Count: 0
[433] [INFO] Mailbox: 2 Info: Data: "", Reader Count: 0
[433] [INFO] Mailbox: 3 Info: Data: "mid_op", Reader Count: 0
[433] [INFO] Mailbox: 4 Info: Data: "", Reader Count: 0
[433] [INFO] Mailbox: 5 Info: Data: "end_op", Reader Count: 0
```

```
[433] [INFO] Global Finish Rate: 0.545455
[433] [INFO] Mean Start Time: 120.000000
[434] [INFO] PROCESS FINISH
```

在这个两辆车例子下两种情况类似，都是在 1 号车出隧道后 2 号车立刻进入，

但是在复杂情况下程序会有较大不同

44. **0 时刻**
    系统启动，日志记录“PROCESS BEGAN”；
    车辆 1 进入隧道，方向被设置为 0（代表东行），且此时隧道为空，记录“Car 1 entering tunnel in direction 0 (empty tunnel)”；
    隧道状态更新为有 1 辆车，“Tunnel Status: Cars count: 1, Current direction: Eastbound”；
    车辆 2 因车辆 1 在隧道内且方向相反（方向 1，代表西行）而等待进入，记录“Car 2 waiting due to opposite direction (direction 1), tunnel occupied by direction 0”。

45. **5 - 35 时刻**
    车辆 1 对多个邮箱进行操作，依次记录“Car 1 lock 1”“Car 1 Reader:. ”“Car 1 lock 5”等，涉及对邮箱 1、5、3 等的锁定和读取操作。
    车辆 1 在隧道内根据任务需求对不同邮箱执行读写操作，这些操作按时间顺序进行，每次操作都先锁定邮箱，操作完成后释放锁，保证对共享邮箱的操作满足读写互斥和同步要求。

46. 239 时刻
    车辆 1 离开隧道，记录“Car 1 Leave”；
    此时隧道为空，因此车辆 2 进入隧道，设置方向为 1（西行），记录“Car 2 entering tunnel in direction 1 (empty tunnel)”，隧道状态更新为有 1 辆车；
    由于车辆 2 对多个邮箱的读写操作出现超时，记录“Car 2 has timed out for read 4”“Car 2 has timed out for write 4”等多条超时信息。

47. 431 时刻
    车辆 2 离开隧道，记录“Car 2 Leave”；

48. 431 时刻之后
    父进程展示隧道中相关信息，并进行性能分析

车辆邮箱方面，Car1 邮箱读取了 1 号邮箱中自己写入的 start_op，而 car2 由于

没有执行任何指令，邮箱为空

隧道邮箱中，1 号邮箱，3 号邮箱和 5 号邮箱分别记录了 car1 写入的信息，

性能分析方面，Car1 完成了全部 6 条指令，而 car2 由于等待红绿灯，所有指令都没有完成，最后的全部指令完成率是 54.54%，平局启动时间和红绿灯一致

复杂样例下的效率分析

其他参数设置

分为使用红绿灯组和不使用红绿灯组，红绿灯的切换时间固定为车辆穿行时间的 2 倍

样例说明

基于 GPT 构建了如下输入样例

49. 简单

    汽车总数 < 5

    隧道容量 < 5

    穿过隧道的时间 < 200

    邮箱数量 < 5

    每个邮箱的最大字数 < 30

50. 中等

    汽车总数 < 40

    隧道容量 < 8

    穿过隧道的时间 < 500

    邮箱数量 < 35

    每个邮箱的最大字数 < 40

51. 复杂

    汽车总数 < 70

    隧道容量 < 20

    穿过隧道的时间 < 600

    邮箱数量 < 40

    每个邮箱的最大字数 < 50

测试结果

	无	红	绿	灯	有	红	绿	灯
	s1	s2	s3	mean	s1	s2	s3	mean
完成率	66.7%	50.0%	50.0%	55.6%	66.7%	50.0%	50.0%	55.6%
总耗时	325	390	260	325	325	395	259	326
启动时间	61	108	71	80	60	90	71	74

	无	红	绿	灯	有	红	绿	灯
	m1	m2	m3	mean	m1	m2	m3	mean
完成率	27.9%	22.4%	18.9%	23.1%	22.9%	16.4%	20.5%	19.9%
总耗时	2791	3471	3230	3164	3270	4883	3840	3998
启动时间	989	1297	1200	1162	1165	1972	1378	1505

	无	红	绿	灯	有	红	绿	灯
	h1	h2	mean		h1	h2	mean	

	无 红 绿 灯			有 红 绿 灯		
完成率	24.3%	21.3%	22.8%	17.1%	20.0%	18.6%
总耗时	3532	20225	11879	4883	21038	12961
启动时间	1335	11365	6350	1972	11617	6795

分析

52. 完成率

    完成率计算为汽车完成的读写指令数量汽车总的读写指令数量可以看出在数据量较小时 s 两者的指令完成率都较高，当数据量增大时，两种方法的准确率都有所下降

53. 总耗时

    总耗时为从程序开始到程序完全终止时消耗的时间，以毫秒为单位

    随着数据规模的增加，红绿灯表现略差于不适用红绿灯

54. 启动时间

    启动时间为从汽车开始到汽车到达终点时消耗的时间的平均值，以毫秒为单位

    启动时间

    和总耗时的趋势相似，随着数据规模增大，红绿灯表现略差于不适用红绿灯

可以看到，随着数据规模增大，红绿灯表现在三个指标上均略差于不适用红绿灯，经过分析有如下几点原因

55. 数据生成

所用数据均由 GPT 生成的数据，以 m3 为例，两种算法总耗时均在 3000~4000 范围内，但是生成的数据中各个车辆的读写时间则主要集中 0~1000，因此只有很小一部分车辆可以成功在规定时间上读写，其他的车辆由于操作超时，会导致对应的操作直接被跳过

```
r 580 70 18
w 'change29' 870 19
r 1160 73 20
end
car 30 1
w 'info30' 300 21
r 600 72 22
w 'change30' 900 23
r 1200 75 24
end
car 31 0
```

56. 数据格式

所给输入中的读写命令均指定了具体的时间，而由于进程调度的不确定，导致输入数据所给的时间区间很难正确的落在车辆真实处于隧道中的时间，尽管我将读写逻辑转换为在预设读写前 5 毫秒锁定邮箱，完成率变化任然不大，

57. 车辆起始时间

由于输入中没有限定起始时间，因此在 0 秒时，其实所有的车辆就都已经等待进入队列，此时 红绿灯算法会导致在一段时间内即使隧道方向改变，但是由于隧道内部仍然有反方向的车，使得隧道方向的车辆无法进

入，这一情况持续到隧道方向再次改变，相较于不适用红绿灯，使用红绿灯的算法在这一段时间内浪费了部分隧道的带宽

58. 红绿灯算法调度开销

为了周期的控制隧道方向转变，红绿灯算法需要使用更多的信号量对车辆进行限制，这一部分也消耗了计算资源

修改逻辑后测试结果

修改汽车进入时间为 `cars[i].car_id%10 * 500`，以下是对比结果

由于 hard 中的数据对处理时间进行过优化，因此此处只对比 small 和 middle

	无					有				
	红			灯	对比	红			灯	对比
	s1	s2	s3	mean	mean	s1	s2	s3	mean	mean
完成率	100.0%	100.0%	100.0%	100.0%	55.6%	100.0%	100.0%	100.0%	100.0%	55.6%
总耗时	1626	1174	1117	1305	325	1627	1253	1626	1502	326
启动时间	0	0	0	0	80	49	59	50	52	74

	无红绿灯对比					有红绿灯对比				
	m1	m2	m3	mean	mean	m1	m2	m3	mean	mean
完成率	100.0%	100.0%	100.0%	100.0%	23.1%	100.0%	100.0%	100.0%	100.0%	19.9%
总耗时	6478	6589	6413	6439	3164	6393	5373	7083	6283	3998
启动时间	795	877	651	774	1162	1268	10	960	746	1505

关于完成率，可以看到，在经过优化汽车进入时间后，两种情况下汽车的读写操作均能全部完成，这一结果与每辆汽车读写操作数量有关（最多 4 条），但也可以证明在真实情况下，无论是否使用红绿灯，两种调度算法都能处理完所有汽车任务。

关于总耗时，总耗时增加是因为引入了延迟调度，由于汽车要等待到指定时间才能进入，相比于之前的直接进入，总耗时均有增加，属于正常现象

关于启动时间，可以看到在 simple 数据，不使用红绿灯情况下，由于两个车间接到达，相差 500ms，大于一辆车的通行时间，因此几乎无等待，启动时间均为 0

即使在更加复杂的数据中，相较于之前的输入，经过修改后的数据格式也可以显著的降低等待时间，提高整体调度效率。

横向对比发现，引入红绿灯会增加总耗时，这是因为增加了等待调度的延迟，但这种延迟带来了调度秩序的提升。

有红绿灯在 simple 场景（small 数据集）里作用不明显，而随着数据复杂度增加，能减少复杂场景（middle 数据集）的启动等待时间，在极端情况下甚至可以将启动时间降低一个数量级

程序框架说明

总框架

```
.
├── CMakeLists.txt      Cmake 文件
├── README.md           说明文件
├── build               编译目录
├── include             头文件
├── input               输入样例
├── linux_hw            可执行文件
└── src                 cpp 文件
```

include/src 文件说明

include

```
.
├── Car.h               模拟车辆进程
├── Tunnel.h            模拟隧道，提供隧道访问相关信号量
├── ipc.h               常用 ipc 函数封装
├── logger.h            日志格式化打印
├── mailbox.h           模拟邮箱，提供邮箱访问相关信号量
```

```
├── mp.h                提供进程控制相关函数
├── process.h           模拟车辆进出隧道，用于总的调度
└── txt_reader.h        读取输入
```

```
src

.
├── Car.cpp
├── Tunnel.cpp
├── ipc.cpp
├── logger.cpp
├── mailbox.cpp
├── main.cpp            程序入口
├── mp.cpp
└── process.cpp
```

input 文件说明

```
input

.
├── hard            复杂测试数据
│   ├── h1.txt
│   ├── h2.txt
│   └── h3.txt
├── middle          中等测试数据
│   ├── m1.txt
│   ├── m2.txt
│   └── m3.txt
├── simple          简单测试数据
│   ├── s1.txt
│   ├── s2.txt
│   └── s3.txt
└── test.txt
```

实验总结

实验围绕多车辆在隧道内的调度与邮箱读写操作展开，通过设计有红绿灯与无红绿灯两种调度策略，使用了信号量，共享内存等信息来对不同进程中的各个数据进行同步，系统性地完成了调度算法，并对资源效率与性能进行了简单的分析。

代码实现过程中我综合使用了管道，信号量，共享内存，互斥锁等 IPC 组件，实现了进程间调度算法，并基于实验任务说明对具体细节做了一定调整，经过测试可以提升一些系统的性能

在实验过程中，我对于之前学过的操作系统相关调度算法有了更加清楚的认知，同时实验中还结合本课程中介绍的管道等组件，而不仅仅局限于信号量和共享内存，编码的过程加深了我对于进程调度的理解，也锻炼了我的编码能力

实验结果表明，系统在数据解析、任务初始化、邮箱互斥控制及基本调度逻辑方面表现稳定，能够准确地调度多车场景下的复杂任务。在有红绿灯调度条件下，系统实现了方向公平性，但由于严格的时间片分配，导致部分车辆出现长时间等待甚至任务超时，整体任务完成率不高，平均启动时间相较于无红绿灯算法较长，性能瓶颈较为明显。相比之下，无红绿灯调度展现出更高的系统吞吐率和灵活性，更加适合实验任务中给定条件，但在真实情况下（车辆按序到达，而不是直接在隧道前排长队），我认为红绿灯调度可以有效减少每个进程的等待时间

实验体会

附上实验过程中我遇到的一些有关 IPC 机制的问题及解决方案

进程数据同步问题

在实现过程中，我遇到了许多这类问题

- 多个进程中车辆同时进入，在子进程中打印隧道状态，显示隧道始终为空

- 在子进程，车辆读取邮箱内容且能够正常输出，但是在父进程的最终无法正产生输出
- 信箱在子进程中被正确赋值，但是在最后主进程中的显示过程中无内容输出

  根本原因是子进程中对于父进程中创建元素的修改不会直接影响父进程中的信息

  我使用了以下方法进行解决
- 共享内存
- 管道

  两种方法均很好的进行了父子进程的同步

信号量同步问题

在构建 process 代码，尤其是在编写 enter 函数过程中，遇到了许多信号量同步问题，以下是几个典型的例子

1. 某一 Car 在一段时间后通过了设置的多个信号量限制，成功进入错误的隧道中

   为了修复这一 BUG，修改了 enter 函数主体，将代码整体放在一个循环中，只有当某一时刻 Car 进程完全符合对应条件才能进入隧道，否则就继续阻塞，这种方法虽然在部分情况下会导致消耗部分时间，但是可以避免程序运行错误
2. 两种调度算法适配问题

   在无红绿灯情况下，隧道方向只有在完全为空时才可能变换，因此不

需要考虑隧道内部是否剩余车，但是使用红绿灯后进入隧道时要考虑当前隧道中反方向的车是否走干净，为此不得不引入两个信号量分别表示隧道中对应方向是否有车

3. 由于 block 信号量设置错误，导致在很长时间内隧道只能行驶一辆车，最后将所有信号量的值打印出来成功找到问题，是 block 初始化错误
4. 在实现较为复杂的 process 类过程中，由于给每个函数内部都加上了使用 mutex 构建的信号量来进行同步，导致函数之间不能相互调用否则在调用函数的内部会阻塞在 Wait 操作上。为此将函数功能做了更细的拆分，仅在必要的函数中使用了信号量进行同步，而仅被类函数调用的函数则不设同步，而是设为 private 避免被外界调用造成同步错误
信号量同步是我在代码构件中遇到的最大问题

测试结果无法复现

测试时候发现若同时运行多个执行程序，最后的结果无法复现

创建信号量使用的标识均由同一个函数 `ftok(pathname, proj_id + PROJ_ZERO_CAR_OFFSET)`和相同参数组成。推测是由于在同一个环境下，导致同时运行的不同程序之间共享了部分信号量和共享内存，由于程序内部信号量逻辑正确实现，导致两个程序以某种方式恰好能正常运行导致

在部分情况下，两个程序会同时卡死，这可能是因为同一个信号量在两个进程中同时被 Wait