



# 华中科技大学

## 操作系统原理实验报告

姓 名： 胡澳

学 院： 计算机科学与技术学院

专 业： 计算机科学与技术

班 级： CS1706 班

学 号： U201714761

指导教师： 石柯

分数	
教师签名	

2019 年 12 月 25 日

# 目 录

<b>1</b>	<b>实验一 进程控制</b>	<b>1</b>
1.1	实验目的	1
1.2	实验内容	1
1.3	实验设计	1
1.3.1	开发环境	1
1.3.2	实验设计	2
1.4	实验调试	3
1.4.1	实验步骤	3
<b>2</b>	<b>实验二 线程同步与通信</b>	<b>5</b>
2.1	实验目的	5
2.2	实验内容	5
2.3	实验设计	5
2.3.1	开发环境	5
2.3.2	实验设计	6
2.4	实验调试	9
2.4.1	实验步骤	9
2.4.2	实验调试	10
2.4.3	实验心得	11
<b>3</b>	<b>实验三 共享内存与进程同步</b>	<b>12</b>
3.1	实验目的	12
3.2	实验内容	12
3.3	实验设计	12
3.3.1	开发环境	12
3.3.2	实验设计	12
3.4	实验调试	14
3.4.1	实验步骤	14
3.4.2	实验调试	16
3.4.3	实验心得	16
<b>4</b>	<b>实验四 文件系统</b>	<b>18</b>
	<b>附录 实验代码</b>	<b>19</b>
	实验一	19
	实验二	21
	实验三	26

# 1 实验一 进程控制

## 1.1 实验目的

1. 加深对进程的理解，进一步认识并发执行的实质。
2. 分析进程争用资源现象，学习解决进程互斥的方法。
3. 掌握 Linux 进程基本控制。
4. 掌握 Linux 系统中的软中断和管道通信。

## 1.2 实验内容

编写程序，演示多进程并发执行和进程软中断、管道通信。

父进程使用系统调用 `pipe()` 建立一个管道,然后使用系统调用 `fork()` 创建两个子进程，子进程 1 和子进程 2；

子进程 1 每隔 1 秒通过管道向子进程 2 发送数据：

I send you x times. (x 初值为 1，每次发送后做加一操作)

子进程 2 从管道读出信息，并显示在屏幕上。

父进程用系统调用 `signal()` 捕捉来自键盘的中断信号（即按 `Ctrl+C` 键）；当捕捉到中断信号后，父进程用系统调用 `Kill()` 向两个子进程发出信号，子进程捕捉到信号后分别输出下列信息后终止：

Child Process 1 is Killed by Parent!

Child Process 2 is Killed by Parent!

父进程等待两个子进程终止后，释放管道并输出如下的信息后终止

Parent Process is Killed!

## 1.3 实验设计

### 1.3.1 开发环境

操作系统：Arch Linux

Linux 内核：x86\_64 Linux 5.4.2-arch1-1

编译器：gcc (GCC) 9.2.0

### 1.3.2 实验设计

主程序的流程图如图 1-1 所示。父进程首先创建两个子进程用于通信的管道 `pipefd`，然后依次创建两个子进程 `p1` 和 `p2`，然后设置中断信号 `SIGINT`，将该信号与信号处理函数 `sig_func` 关联起来。子进程 `p1` 被创建后，首先设置忽略中断信号 `SIGINT`，并将 `SIGUSR1` 信号与 `sig_func` 函数相关联，然后 `p1` 进入一个无限的循环，每次循环中，`p1` 向通信管道 `pipefd` 中写入一个字符串，然后将字符串中的一个数字加一，接下来 `p1` 进程休眠 1s，然后开始下一次循环。子进程 `p2` 被创建后，设置忽略 `SIGINT` 信号，将 `SIGUSR2` 信号与 `sig_func` 函数关联。与 `p1` 进程相对于的，`p2` 也进入一个死循环，每次循环中，`p2` 从管道中读取一个字符串，并将其输出，在休眠 1s 后进入下一次循环。

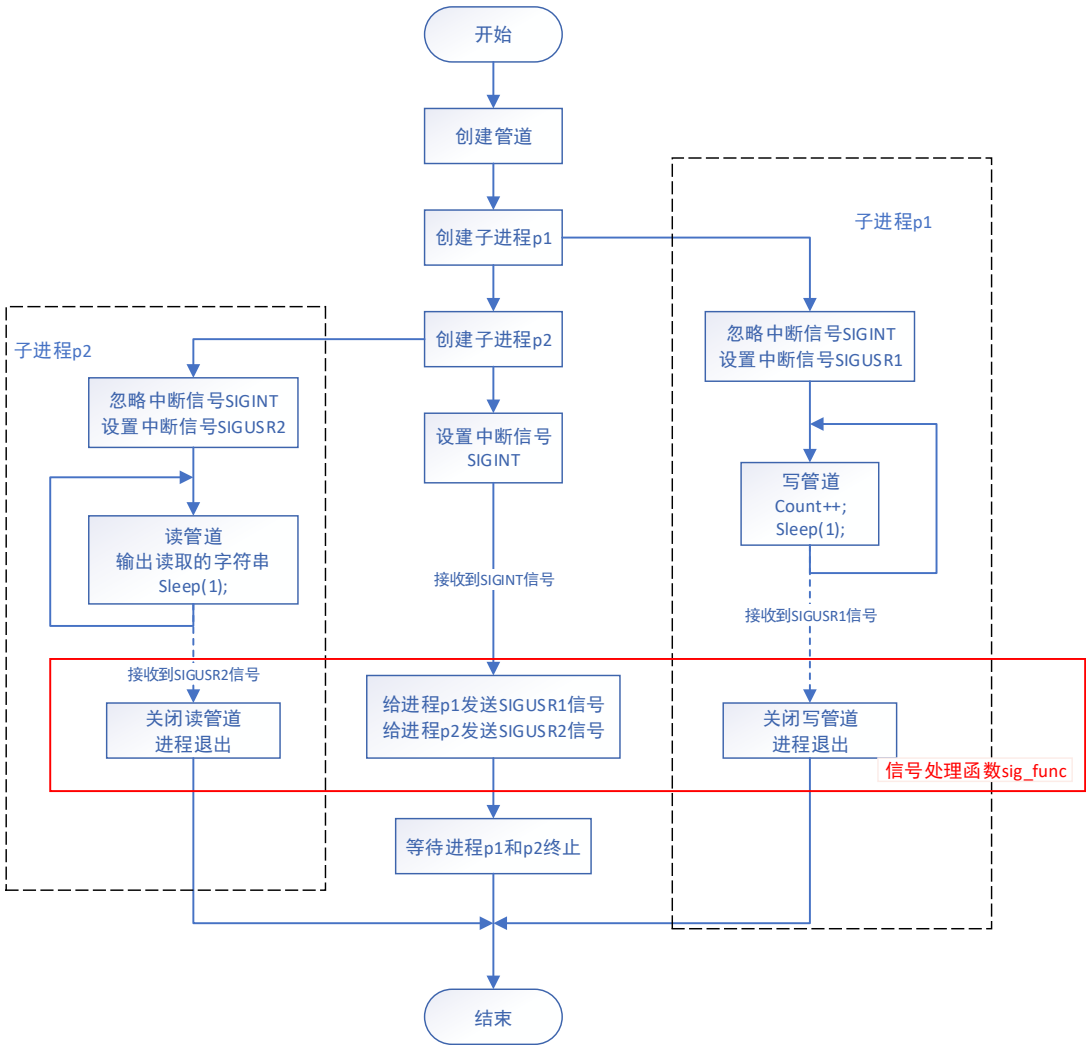


图 1-1 进程控制流程图

当在终端中输入 `Ctrl+C` 后，操作系统将给父进程发送一个 `SIGINT` 信号，父进程捕获到该信号后，将调用 `sig_func` 函数处理该信号。在 `sig_func` 函数中，父进程将向子进程 `p1` 发送 `SIGUSR1` 信号，向子进程 `p2` 发送 `SIGUSR2` 信号，然

后父进程退出信号处理函数，并等待 p1 和 p2 结束。于此同时，操作系统也会向子进程 p1 和 p2 发送 SIGINT 信号，但是由于两个子进程设置了对该信号的忽略，因此他们不会对该信号做出响应。

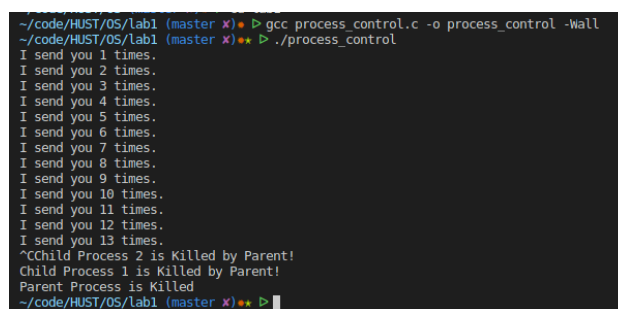
当 p1 收到来自父进程的 SIGUSR1 信号，p1 调用 sig\_func 处理该信号，对于该信号，p1 将关闭管道 pipefd 的写端口，然后退出进程。类似的，p2 收到 SIGUSR2 信号后将关闭管道的读端口并退出进程。当 p1 和 p2 均处理完信号并退出后，父进程将完成对两个子进程的等待，并在输出相应提示信息后退出。

另外，程序会处理创建管道、创建子进程时可能出现的错误。若程序创建管道或创建子进程 p1 时出错，则父进程将输出相应的错误信息后直接退出，若程序在创建子进程 p2 时出错，此时需要对已经创建成功的管道以及子进程 p1 进行相应的处理，对于子进程 p1，父进程向其发送 SIGUSR1 信号，使 p1 调用信号处理函数从而终止 p1 进程，注意到此时 p1 进程已经关闭了管道的写端口，因此父进程还需要关闭管道的读端口，然后父进程输出错误信息并退出。

## 1.4 实验调试

### 1.4.1 实验步骤

完成源代码后，使用 gcc 编译，编译命令行为 gcc process\_control.c -o process\_control -Wall 得到可执行文件 process\_control，执行该文件，运行一段时间后输入 Ctrl+C，运行结果如图 1-2 所示。



```
~/code/HUST/OS/lab1 (master X) ➤ gcc process_control.c -o process_control -Wall
~/code/HUST/OS/lab1 (master X) ➤ ./process_control
I send you 1 times.
I send you 2 times.
I send you 3 times.
I send you 4 times.
I send you 5 times.
I send you 6 times.
I send you 7 times.
I send you 8 times.
I send you 9 times.
I send you 10 times.
I send you 11 times.
I send you 12 times.
I send you 13 times.
^CChild Process 2 is Killed by Parent!
Child Process 1 is Killed by Parent!
Parent Process is Killed
~/code/HUST/OS/lab1 (master X) ➤
```

图 1-2 进程控制运行截图

由运行过程可知，每过 1s 时间程序将输出一行 “I send you x times.”，结合源代码可知，输出是由子进程 p2 完成的，而输出内容中数字 x 的递增是由子进程 p1 完成的，由此可知，程序正确完成了子进程 p1 和 p2 之间通过管道的通信。当输入 Ctrl+C 后，程序依次输出了子进程 p2、p1 终止时的输出信息，在此之后，父进程输出了其退出的信息，该现象表明程序正确的处理了操作系统发送给父进程的 SIGINT 信号，同时，子进程也正确忽略了操作系统的 SIGINT 信号并正确响应了父进程发送给他们的信号。

通过上述运行结果可知，程序正确展示了多进程并发执行、进程软中断的执行以及进程间通过管道通信的过程。

## 2 实验二 线程同步与通信

### 2.1 实验目的

1. 掌握 Linux 下线程的概念。
2. 了解 Linux 线程同步与通信的主要机制。
3. 通过信号灯操作实现线程间的同步与互斥。

### 2.2 实验内容

#### 1) 线程同步

设计并实现一个计算线程与一个 I/O 线程共享缓冲区的同步与通信，程序要求：

- 两个线程,共享公共变量 a;
- 线程 1 负责计算 (1 到 100 的累加, 每次加一个数);
- 线程 2 负责打印 (输出累加的中间结果);
- 主进程等待子线程退出。

#### 2) 线程互斥

编程模拟实现飞机售票：

- 创建多个售票线程；
- 已售票使用公用全局变量；
- 创建互斥信号灯；
- 对售票线程临界区施加 P、V 操作；
- 主进程等待子线程退出，各线程在票卖完时退出。

### 2.3 实验设计

#### 2.3.1 开发环境

操作系统：Arch Linux

Linux 内核：x86\_64 Linux 5.4.2-arch1-1

编译器：g++ (GCC) 9.2.0

## 2.3.2 实验设计

### 1. 实验内容 1 线程同步实验设计

本实验中两线程间的同步关系如图 2-1 所示。为实现预期的功能，需要在图中 0 和 1 出分别设置一个信号灯。不妨假设计算线程先计算，然后打印线程输出计算结果。则信号灯 1 的初值应赋为 1，信号灯 0 的初值应赋为 0。计算线程需要获取到信号灯 1 后才能开始执行，其执行完成后，释放信号灯 0，相应的，打印线程获取到信号灯 0 后才能够执行，执行完成后，释放信号灯 1。这样即可实现两个线程的同步操作。

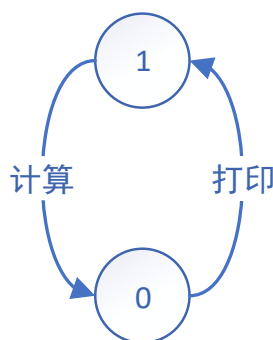


图 2-1 计算/打印线程同步关系

线程同步实验的程序流程图如图 2-2 所示。主线程创建一个包含两个信号灯的信号灯集 `semid` 并根据上述讨论结果给两个信号灯赋初值。接下来，主线程创建两个子线程，分别用于执行计算和输出操作。线程创建完成后，主线程即可等待两个子线程执行结束。待两个子线程均执行完成后，主线程退出。

对于计算子线程，该线程为一个无限循环，在循环体中，首先需要获取信号灯 1，即对信号灯 1 进行 P 操作，若操作失败，则该线程进入等待队列，直到输出线程输出当前 `count` 值后对信号灯 1 进行 V 操作后，该线程被调入就绪队列，进而在获得执行权限后进行 P 操作成功时的操作。否则，则表明此时的 `count` 值已经由输出线程输出，计算线程首先判断 `count` 值是否大于 100，若已经大于则表示线程执行完毕，线程对信号灯 0 调用 V 操作，并退出该线程，若 `count` 不大于 100，则程序应继续执行，计算线程将 `count` 的值加一，并输出相应的提示信息，然后对信号灯 0 调用 V 操作，表明此时的 `count` 值需要被输出线程输出。完成上述操作后，线程进入下一次循环。

同样的，对于输出子线程，其同样为一个无限循环，在循环体中，首先需要对信号灯 0 进行 P 操作，若操作成功，则表明当前的 `count` 值还未被输出，此时，线程首先判断该 `count` 值是否大于 100，若大于 100 则不用输出，线程对信号灯 1 进行 V 操作后退出。若 `count` 不大于 100，线程输出包含该 `count` 值的信息，



然后在信号灯 1 进行 V 操作，表示当前 count 值已被输出过后，进行下一次循环。

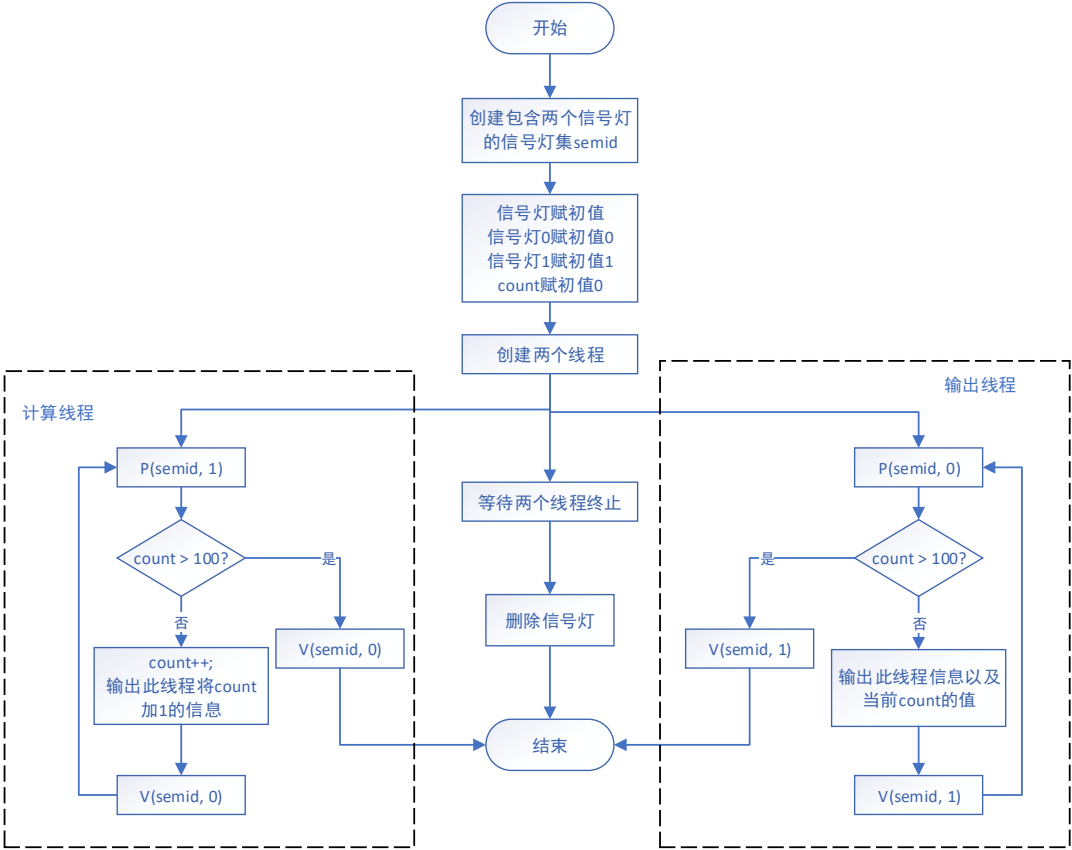


图 2-2 线程同步程序流程图

## 2. 实验内容 2 线程互斥实验设计

本实验中，将有多个线程需要访问同一个变量，此时需要对该临界变量进行互斥访问以保证每个线程可以获得正确的数据，同时也能够保证该变量能够被正确的修改。由于是对一个变量的互斥访问，因此本实验中只需要使用一个信号灯来控制该变量在同一时刻只被一个线程访问即可。

线程互斥程序的主线程流程图如图 2-3 所示，售票线程流程图如图 2-4 所示。在主线程中，程序首先创建一个只包含一个信号灯的信号灯集，并给该信号灯赋初值为 1。然后通过循环创建多个售票线程，待子线程创建结束后，主线程开始逐个的等待每个线程结束并输出每个线程的结束信息，待所有售票线程全部结束后，主线程删除信号灯集，然后输出程序结束信息并退出。

售票线程是一个无限循环，在循环体中，首先线程将对信号灯调用 P 操作以获取对互斥变量 sale 的访问权限，若线程未能获取到该权限，则线程将进入等待队列，直到当前获得 sale 访问权限的线程调用 V 操作释放该权限后，从等待队列中逐个的赋予访问权限。若线程成功通过 P 操作获取到对 sale 的访问权限，则线程首先判断当前已卖出票数 sale 是否小于总票数 TICKET\_NUM，若小于，

则当前有票可买，该线程卖出一张票，将已卖出票数 `sale` 加一并输出卖票信息，然后调用 `V` 操作释放对 `sale` 的访问权限。若此时票已卖完，则线程输出票卖完的信息，调用 `V` 操作释放权限后退出。

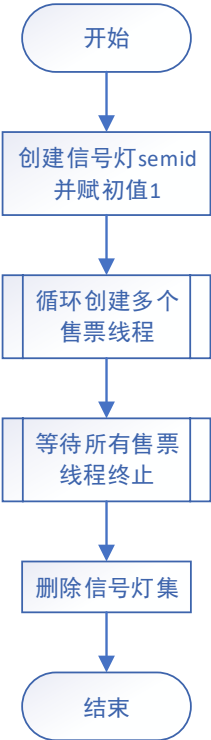


图 2-3 线程互斥-主线程流程图

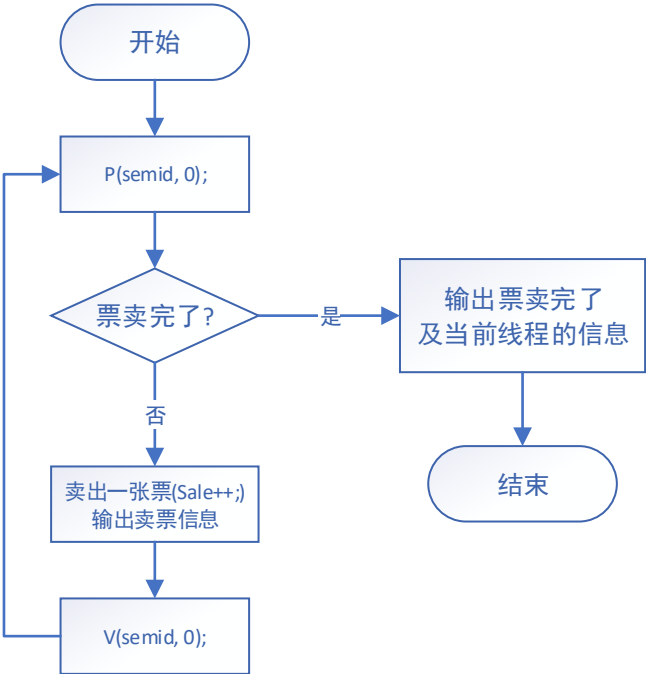


图 2-4 线程互斥卖票线程流程图

在此程序中，主线程中使用循环结构创建大量线程，在创建线程时需要判断线程创建是否成功，否则在后续的线程操作中会出现不可预期的异常情况发生。当主线程检测到某个线程创建失败时，将输出相应的错误信息，并向所有已经创

建成功的线程发送 `cancel` 信号，将已创建的线程全部终止，然后等到所有子线程终止完成后，主线程退出。

## 2.4 实验调试

### 2.4.1 实验步骤

#### 1. 实验内容 1 线程同步

完成源代码后，使用 `g++` 编译，编译命令行为 `g++ thread_syn.cpp -Wall -o thread_syn -lpthread`，编译后得到可执行文件 `thread_syn`，执行该文件，输出如图 2-5 所示。有运行结果可知，计算线程和输出线程交替输出，计算线程每将 `count` 值加 1 后，输出线程将把更新后的值输出，并且只有当输出线程输出了当前的 `count` 值后，计算线程才会对其做下一次加一操作。即程序正确的完成了计算和输出线程之间的同步操作。

```
~/code/HUST/OS/Lab2 (master x) ➤ g++ thread_syn.cpp -Wall -o thread_syn -lpthread
~/code/HUST/OS/Lab2 (master x) ➤ ./thread_syn
calculate thread add count from 0 to 1
print by thread 2
1
calculate thread add count from 1 to 2
print by thread 2
2
calculate thread add count from 2 to 3
print by thread 2
3
calculate thread add count from 3 to 4
print by thread 2
4
calculate thread add count from 4 to 5
print by thread 2
5
calculate thread add count from 5 to 6
print by thread 2
6
calculate thread add count from 6 to 7
print by thread 2
7
calculate thread add count from 7 to 8
print by thread 2
8
calculate thread add count from 8 to 9
print by thread 2
9
calculate thread add count from 9 to 10
print by thread 2
10
calculate thread add count from 10 to 11
print by thread 2
11
calculate thread add count from 11 to 12
print by thread 2
12
calculate thread add count from 12 to 13
```

图 2-5 线程同步运行结果

#### 2. 实验内容 2 线程互斥

完成源代码后，使用 `g++` 编译，编译命令行为 `g++ thread_mutex.cpp -Wall -o thread_mutex -lpthread`，编译后得到可执行文件 `thread_mutex`，执行该文件的输出如图 2-6 所示。

```
~/code/HUST/OS/Lab2 (master x) ➤ ./thread_mutex
thread 0 create successful threadID is 140612551640832
thread 1 create successful threadID is 140612543248128
thread 2 create successful threadID is 140612534855424
sale thread 0 sales ticket 1
sale thread 0 sales ticket 2
sale thread 0 sales ticket 3
sale thread 0 sales ticket 4
sale thread 1 sales ticket 5
sale thread 2 sales ticket 6
sale thread 0 sales ticket 7
sale thread 1 sales ticket 8
sale thread 2 sales ticket 9
sale thread 0 sales ticket 10
sale thread 1 sales ticket 11
sale thread 0 sales ticket 12
sale thread 2 sales ticket 13
sale thread 1 sales ticket 14
sale thread 0 sales ticket 15
sale thread 2 sales ticket 16
sale thread 1 sales ticket 17
sale thread 0 sales ticket 18
sale thread 2 sales ticket 19
sale thread 1 sales ticket 20
thread 0 no ticket, thread exit
thread 2 no ticket, thread exit
thread 1 no ticket, thread exit
wait thread 0 successful
wait thread 1 successful
wait thread 2 successful
finished
```

图 2-6 线程互斥运行结果

由运行结果可知，程序成功创建出 3 个线程，然后 3 个线程开始以随机的顺序交替卖票，当票全部卖完后，三个线程依次退出，随后，主线程输出程序运行完成的信息并退出。

## 2.4.2 实验调试

在实验内容 2 线程互斥中，可以通过限制程序运行的地址空间来模拟当系统资源不足导致线程创建失败的情况。在 Linux 下可以使用 `ulimit` 命令查看和修改系统对当前 shell 下运行程序的限制。根据 `ulimit` 的输出可以得知，默认情况下，每个线程需要的栈空间为 8192KB，而系统对其虚拟内存空间是不做限制的。我们使用 `ulimit -v 24576` 将程序的虚拟内存空间限制为  $3 * 8192 = 24576$ KB，即单个进程只能够同时运行 3 个线程，否则将出现内存不足的问题。我们在内存空间受限的情况下运行程序，由于程序中将创建 3 个售票线程，加上主线程，在该进程中一共将有 4 个线程，然后由于内存空间的限制，第三个售票线程将无法创建成功，但在程序实际运行情况中，并未出现 `pthread_create` 函数创建线程时返回失败，相反，在程序执行到主线程调用 `pthread_join` 函数等待子线程终止时发生段错误。该运行结果如图 2-7 所示。

```
~/code/HUST/OS/lab2 (master x) ➤ ./thread_mutex
sale thread 0 sales ticket 1
sale thread 1 sales ticket 2
sale thread 0 sales ticket 3
sale thread 1 sales ticket 4
sale thread 0 sales ticket 5
sale thread 1 sales ticket 6
sale thread 0 sales ticket 7
sale thread 1 sales ticket 8
sale thread 0 sales ticket 9
sale thread 1 sales ticket 10
sale thread 0 sales ticket 11
sale thread 1 sales ticket 12
sale thread 0 sales ticket 13
sale thread 1 sales ticket 14
sale thread 0 sales ticket 15
sale thread 1 sales ticket 16
sale thread 0 sales ticket 17
sale thread 1 sales ticket 18
sale thread 0 sales ticket 19
sale thread 1 sales ticket 20
thread 0 no ticket, thread exit
thread 1 no ticket, thread exit
wait thread 0 successful
wait thread 1 successful
[1] 2574 segmentation fault (core dumped) ./thread_mutex
~/code/HUST/OS/lab2 (master x) ➤
```

图 2-7 限制内存空间后创建线程失败导致段错误

通过调试，我最终发现，在 `pthread_create` 创建线程时，虽然其返回值为 0，表明线程创建成功，但是实际上其获得的线程 ID 为 0，即表明线程创建失败。将获得的线程 ID 为 0 一起作为判断条件来处理线程创建失败的情况，即可正确处理上述一场情况。修改后程序的运行结果如图 2-8 所示。

```

~/code/HUST/OS/lab2 (master x) ★ ▶ ulimit -v 24576;semctl(semid, 0, SETVAL, 1)
~/code/HUST/OS/lab2 (master x) ★ ▶ ./thread_mutex semctl(semid, 0, IPC_RMID, 1)
thread 0 create successful threadID is 139699110622976:if("init sem 0 fa
thread 1 create successful threadID is 139699102230272\n 0;
thread 2 create failed                                96      }
thread 0 is canceled                                  97      // 给信号灯赋初值
thread 1 is canceled                                  98
~/code/HUST/OS/lab2 (master x) ★ ▶ 99      for (int i = 0; i < THREE

```

图 2-8 正确处理线程创建出错

### 2.4.3 实验心得

在本次实验中，我对 Linux 中的线程、线程间的关系、同步/互斥关系以及信号灯的使用有了基本的了解。同时，在复现线程创建失败的情况以及对该情况做相应处理的过程中，我对 Linux 的资源管理机制有了些许的了解，另外，我也认识到，在资源受限的情况下的多线程编程中，应该使用线程池的处理方式而不是直接开更多的线程。

## 3 实验三 共享内存与进程同步

### 3.1 实验目的

- 1、掌握 Linux 下共享内存的概念与使用方法；
- 2、掌握环形缓冲的结构与使用方法；
- 3、掌握 Linux 下进程同步与通信的主要机制。

### 3.2 实验内容

1. 利用多个共享内存（有限空间）构成的环形缓冲，将源文件复制到目标文件，实现两个进程的誊抄。
2. 对下列参数设置不同的取值，统计程序并发执行的个体和总体执行时间，分析不同设置对缓冲效果和进程并发执行的性能影响，并分析其原因：
  - (1) 信号灯的设置；
  - (2) 缓冲区的个数；
  - (3) 进程执行的相对速度。

### 3.3 实验设计

#### 3.3.1 开发环境

操作系统：Arch Linux

Linux 内核：x86\_64 Linux 5.4.2-arch1-1

编译器：g++ (GCC) 9.2.0

其他工具：dd(coreutils) 8.31

hdparm v9.58

md5sum(GNU coreutils) 8.31

#### 3.3.2 实验设计

1. 实验内容 1：利用环形缓冲区实现两个进程的誊抄。

写缓冲区进程和读缓冲区进程之间的同步关系如图 3-1 所示。为实现两个进程之间的正确誊抄，需要对缓冲区设置两个信号灯，分别用于确定缓冲区需要读和可写的状态。由于初始状态下，所有的缓冲区均可写且不可读，因此信号灯 0 的初值应置为 0，对于信号灯 1，有两种赋初值的方式，可以将信号灯 1 的初值赋为 1，表示将所有的缓冲区视为一个整体，每次要操作其中的某一个缓冲区时，其余的缓冲区均不可操作，另一种方式为将信号灯 1 的初值赋为缓冲区数量，此时各缓冲区中间可以并行操作。两个进程通过对这两个信号灯的 PV 操作，即可实现对缓冲区的同步访问。

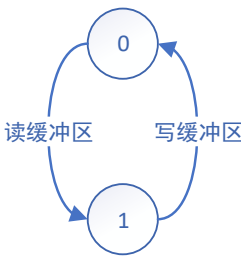


图 3-1 读、写缓冲区进程同步关系

两进程誊抄的流程图如图 3-2 所示。主进程首先向操作系统申请共享内存空间、创建信号灯，并根据上面的分析为信号灯赋初值。接下来，主进程创建一个读缓冲区进程和一个写缓冲区进程，创建完成后，主进程等待两个子进程终止，当两个子进程均结束后，释放共享内存空间并删除信号灯，然后主进程结束。

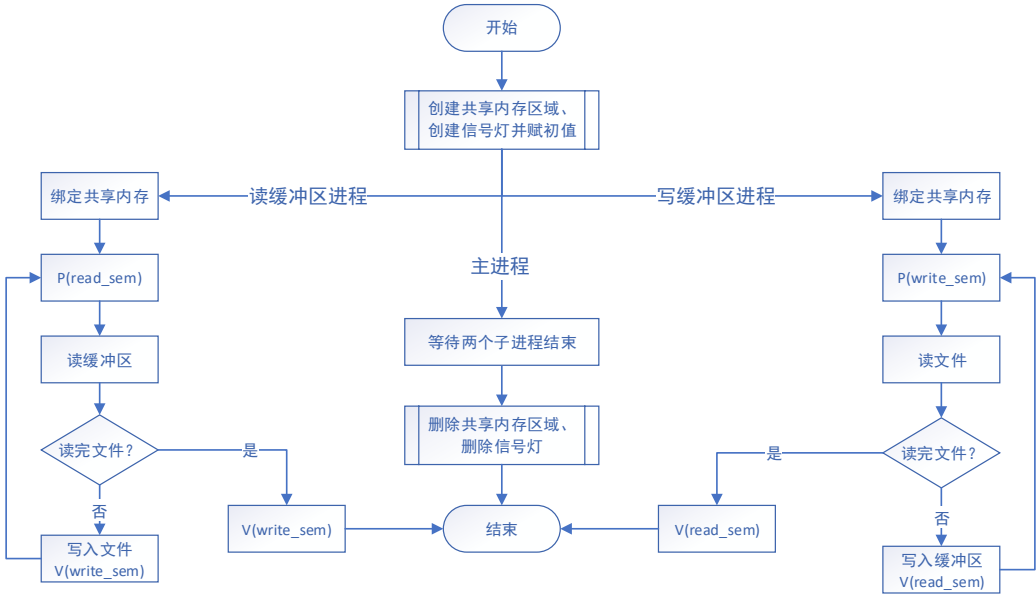


图 3-2 进程誊抄流程图

两个子进程被创建后，首先创建相应的指针并将其绑定到主进程申请的共享内存空间。然后对于写缓冲区进程，不断的读取文件并将其写入到缓冲区中，待文件读取完成后，向缓冲区中写入相应的标志，写缓冲区进程结束；对于读缓冲

区进程，不断的从缓冲区中读取数据，并将其写入到相应的文件中，当该进程从缓冲区中读取到写进程写入的文件读取完成的标志时，读缓冲区进程结束。

同时，当两个子进程在操作缓冲区时，需要使用上面分析过的信号灯操作，对缓冲区实现同步访问，从而确保誊抄结果的正确性。

## 2. 实验内容 2：分析不同参数的设置对缓冲效果以及进程并发效果的影响。

从信号灯的设置、缓冲区个数以及进程执行的相对速度三个方面对誊抄总时间的影响来确定其对缓冲效果和进程并发效果的影响。

信号灯有两种设置方式，分别为将所有缓冲区视为一个整体或者将其视为多个独立的缓冲区。若将所有缓冲区视为一个整体，则将图 3-1 中的信号灯 1 的初值赋为 1，此时，在同一时刻，两个进程中只有一个进程能够访问缓冲区，两个进程实际上并未并行的使用缓冲区。若将每个缓冲区视为单独的个体，则将图 3-1 中的信号灯 1 的初值赋为缓冲区的个数，此时，在一定的条件下，两个进程可以并行的使用缓冲区，且能够通过信号灯的同步操作实现对缓冲区的同步访问。

在程序中设置不同的缓冲区的个数，对同一个文件进行誊抄，记录誊抄花费的时间，比较这些时间即可判断不同的缓冲区个数对缓冲效果和进程并发效果的影响。对于进程执行的相对速度，可以通过比较将文件从高速设备誊抄到低速设备和从低速设备誊抄到高速设备两种不同的方式誊抄所花费的时间来分析其对缓冲效果的影响。

## 3.4 实验调试

### 3.4.1 实验步骤

#### 1. 利用环形缓冲区实现进程间的誊抄。

完成源代码后，使用 g++ 编译，编译命令行为 `g++ process_share_memory.cpp -o process_share_memory`，得到可执行文件 `process_share_memory`。

使用 linux 的 `dd` 命令以及随机伪设备 `/dev/urandom` 生成一个随机的输入文件，生成文件的命令行为 `dd if=/dev/urandom of=./input.txt bs=16M count=20`，该命令将生成一个文件大小为 320MiB 的由随机数据组成的文件 `input.txt`，命令执行结果如图 3-3 所示。

```
~/code/HUST/OS/lab3 (master x) ➤ dd if=/dev/urandom of=input.txt bs=16M count=20
记录了 20+0 的读入
记录了 20+0 的写出
335544320 bytes (336 MB, 320 MiB) copied, 2.05526 s, 163 MB/s
```

图 3-3 dd 生成随机输入文件



另外，为了避免由于硬盘缓存对实验结果的影响，使用 `hdparm` 关闭硬盘缓存。查看硬盘缓存状态的命令行为 `hdparm -W /dev/sdb`，关闭硬盘缓存的命令行为 `hdparm -W 0 /dev/sdb`。命令执行结果如图 3-4 所示。

运行编译得到的可执行文件 `process_share_memory`，运行结果如图 3-5 所示。运行后得到输出文件 `output.txt`。使用 `md5sum` 命令计算输入文件 `input.txt` 和输出文件 `output.txt` 的 md5 值，计算结果如图 3-6 所示，由计算结果可知，输入输出文件的 md5 值相等，即表明两文件完全相同，即誊抄功能正确实现。

```
~/code/HUST/OS/lab3 (master x) ➤ sudo hdparm -W /dev/sdb
/dev/sdb:
write-caching = 1 (on)
~/code/HUST/OS/lab3 (master x) ➤ sudo hdparm -W 0 /dev/sdb
/dev/sdb:
setting drive write-caching to 0 (off)
write-caching = 0 (off)
```

图 3-4 `hdparm` 关闭磁盘缓存

```
~/code/HUST/OS/lab3 (master x) ➤ ./process_share_memory
readbuf::process started
writebuf::process started
writebuf::write file data finished, process exit
writebuf::CPU time is 328.046000 ms, run time is 902 ms
readbuf::read file data finished, process exit
readbuf::CPU time is 1029.860000 ms, run time is 3587 ms
main::process readbuf exit
main::process writebuf exit
main::process main exit
main::CPU time is 0.696000 ms, run time is 3588 ms
```

图 3-5 程序运行结果

```
~/code/HUST/OS/lab3 (master x) ➤ md5sum input.txt
9d4ad2bb5f032d071ee66e409cfad578 input.txt
~/code/HUST/OS/lab3 (master x) ➤ md5sum output.txt
9d4ad2bb5f032d071ee66e409cfad578 output.txt
```

图 3-6 输入/输出文件 md5 值计算

2. 不同参数对缓冲效果以及进程同步效果的影响。

首先考虑信号灯的设置方式对缓冲效果的影响，将缓冲区数量设置为 30，分别按照两种方式设置信号灯，将信号灯初值分别置为 1 和 30，记录誊抄程序执行时间，当信号灯初值取 1 时，程序执行时间为 3653ms，当信号灯初值置为 30 时，程序执行时间为 3435ms。由上述结果可以看到，对每个缓冲区单独考虑的缓冲效果要好于将所有的缓冲区视为一个整体。事实上，将所有的缓冲区视为一个整体时，读写缓冲区进程实际上时串行执行的，设置的缓冲区并没有起到预期的作用。

接下来我们考虑缓冲区数量对缓冲效果的影响。将信号灯初值赋为缓冲区的个数，然后逐渐增加缓冲区数量，统计程序执行的时间，统计结果如表 3-1 所示。

表 3-1 缓冲区数量与运行时间关系

缓冲区数量	1	3	5	10	20	30
运行时间	3694	3583	3412	3401	3433	3410

由上述数据可以看出，当缓冲区数量由 1 开始逐渐增大时，缓冲效果会变好，但是当缓冲区数量变得更大后，缓冲区的数量对缓冲效果将没有明显的影响。由于上述测试数据是将一个文件从硬盘誊抄到同一块硬盘，二者速度相当，因此少数几个缓冲区即可实现读写缓冲区进程并行执行，当缓冲区更多时，并不会继续提高誊抄速度。

最后我们测试进程相对速度对缓冲效果的影响。我们将主存中分割 2G 的空间作为内存文件系统使用，操作的命令行为 `sudo mount -t ramfs -o size=2G ramfs /mnt/ramfs`。将内存作为快速设备，将硬盘作为慢速设备，分别使用进程誊抄程序将文件从硬盘誊抄到内存、从内存誊抄到硬盘，记录进程占用的 CPU 时间如表 3-2 所示。由该数据可以看出，当誊抄的两个设备速度相差较大时，缓冲区可以起到一定的缓冲加速的作用，但是当拷贝的数据量较大时，快速设备对缓冲区的操作很快就可以完成，而慢速设备需要经过较长时间才能够完成操作，最终二者仍然会表现为交替执行的状态。

表 3-2 快速设备与慢速设备之间的相互誊抄

	硬盘 -> 内存	内存 -> 硬盘
写缓冲区进程	225	307
读缓冲区进程	308	925

### 3.4.2 实验调试

在实验中，当程序向操作系统申请共享内存空间失败后，程序异常退出，然后再次执行程序时，将会一块共享内存都无法申请到。通过查阅相关的资料，我发现程序在结束后，与使用 `malloc` 函数或 C++ 中的 `new` 动态申请内存空间不同，使用 `shmget` 获取到的共享内存空间并不会被自动释放，而是需要在程序中手动释放申请的共享内存。因此，当程序正常运行时，在主进程等待两个子进程结束后，需要调用 `shmctl` 中的相应功能释放共享内存，同时，若程序执行过程中出现任何的可捕获的问题需要提前终止，也需要对共享内存进行释放操作。

### 3.4.3 实验心得

本次实验中，我完成了两个进程之间通过使用共享内存缓冲区的方式的实现进程间的誊抄。通过功能的实现，我了解了 `linux` 中共享内存的基本使用方法，对进程同步的概念和实现也有了进一步的认识。

通过设置不同的参数，我直观的看到了对缓冲区不同的操作方式、不同的缓冲区数量对誊抄效果的影响，在这个过程中，我看到了多次执行同一个程序也会

出现较大的执行时间上的差异,这一现象也让我认识到了操作系统不确定性的特点。

## 4 实验四 文件系统

略

# 附录 实验代码

## 实验一

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

__pid_t p1;
__pid_t p2;
int pipefd[2];

void sig_func(int sig_no) {
    if (sig_no == SIGINT) {
        if (kill(p1, SIGUSR1) == 1) {
            printf("In parent process, send SIGUSR1 to p1 failed\n");
        }
        if (kill(p2, SIGUSR2) == 1) {
            printf("In parent process, send SIGUSR2 to p2 failed\n");
        }
    } else if (sig_no == SIGUSR1) {
        close(pipefd[1]);
        printf("Child Process 1 is Killed by Parent!\n");
        exit(0);
    } else if (sig_no == SIGUSR2) {
        close(pipefd[0]);
        printf("Child Process 2 is Killed by Parent!\n");
        exit(0);
    } else {
        printf("unknown signal\n");
    }
}

int main() {
    if (pipe(pipefd) == -1) {
        printf("pipe create error\n");
        return 0;
    }
```

```

p1 = fork();
if (p1 == -1) {
    printf("fork p1 error\n");
    return 0;
}
if (p1 != 0) {
    // main
    p2 = fork();
    if (p2 == -1) {
        printf("fork p2 error\n");
        kill(p1, SIGUSR1);
        waitpid(p1, NULL, 0);
        return 0;
    }
    if (p2 != 0) {
        // main
        if (signal(SIGINT, sig_func) == SIG_ERR) {
            printf("In parent process, ignore SIGINT failed\n");
        }
    } else {
        // process 2
        if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
            printf("In child 2 process, ignore SIGINT failed\n");
        }
        if (signal(SIGUSR2, sig_func) == SIG_ERR) {
            printf("In child 2 process, set SIGUSR2 failed\n");
        }
        char str[30];
        while (1) {
            read(pipefd[0], str, 30);
            printf("%s", str);
        }
    }
} else {
    // process 1
    if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
        printf("In child 1 process, ignore SIGINT failed\n");
    }
    if (signal(SIGUSR1, sig_func) == SIG_ERR) {
        printf("In child 1 process, set SIGUSR1 failed\n");
    }
    char str[30];
    int times = 1;

```

```

        while (1) {
            sprintf(str, "I send you %d times.\n", times++);
            write(pipefd[1], str, 30);
            sleep(1);
        }
    }

    waitpid(p1, NULL, 0);
    waitpid(p2, NULL, 0);
    printf("Parent Process is Killed\n");
    exit(0);
}

```

## 实验二

// 实验内容 1 线程同步

```

#include <pthread.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

```

```

union semun {
    int val; // SETVAL 使用的值
    struct semid_ds *buf; // IPC_STAT、IPC_SET 使用缓存区
    unsigned short *array; // GETALL、SETALL 使用的数组
    struct seminfo *__buf; // IPC_INFO(Linux 特有) 使用缓存区
};

```

// P 操作

```

void P(int semid, int index) {
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = -1;
    sem.sem_flg = 0;
    semop(semid, &sem, 1);
    return;
}

```

// V 操作

```

void V(int semid, int index) {
    struct sembuf sem;
    sem.sem_num = index;

```

```

sem.sem_op = 1;
sem.sem_flg = 0;
semop(semid, &sem, 1);
return;
}

int count;
int semid; // 信号灯 ID

void *calculate_func(void *arg) {
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    // 收到 cancel 信号时运行到终止点再终止
    while (1) {
        pthread_testcancel(); // 设置终止点
        P(semid, 1);
        if (count > 100) {
            V(semid, 0);
            return NULL;
        }
        printf("calculate thread add count from %d to %d\n", count, count + 1);
        count++;
        V(semid, 0);
    }

    return NULL;
}

void *print_func(void *arg) {
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    // 收到 cancel 信号时运行到终止点再终止
    while (1) {
        pthread_testcancel(); // 设置终止点
        P(semid, 0);
        if (count > 100) {
            V(semid, 1);
            return NULL;
        }
        printf("print by thread 2\t\t%d\n", count);
        V(semid, 1);
    }

    return NULL;
}

```



```

int main() {
    // 创建信号灯
    semid = semget(IPC_PRIVATE, 2, IPC_CREAT | 0666);
    if (semid == -1) {
        printf("semget failed\n");
        return 0;
    }

    semun arg;
    arg.val = 0;
    if (semctl(semid, 0, SETVAL, arg) == -1) {
        printf("init sem 0 failed\n");
        semctl(semid, 0, IPC_RMID);
        return 0;
    }
    arg.val = 1;
    if (semctl(semid, 1, SETVAL, arg) == -1) {
        printf("init sem 1 failed\n");
        semctl(semid, 0, IPC_RMID);
        return 0;
    }
    // 给信号灯赋初值

    count = 0;

    pthread_t calculate, print;
    if (pthread_create(&calculate, NULL, calculate_func, NULL) == -1) {
        printf("create calculate thread failed\n");
        return 0;
    }

    if (pthread_create(&print, NULL, print_func, NULL) == -1) {
        printf("create print thread failed\n");
        pthread_cancel(calculate);
        pthread_join(calculate, NULL);
        // 终止计算线程并等待其结束
        return 0;
    }

    // 计算和输出线程创建成功

    pthread_join(calculate, NULL);
    pthread_join(print, NULL);
}

```

```

    semctl(semid, 0, IPC_RMID);

    printf("\nfinished\n");

    return 0;
}

// 实验内容 2 线程互斥
#include <pthread.h>
#include <stdio.h>
#include <sys/sem.h>
#include <unistd.h>

#define THREAD_NUM 3
#define TICKET_NUM 20

int sale;
int semid;
pthread_t thread_id[THREAD_NUM];

union semun {
    int val; // SETVAL 使用的值
    struct semid_ds *buf; // IPC_STAT、IPC_SET 使用缓存区
    unsigned short *array; // GETALL、SETALL 使用的数组
    struct seminfo *__buf; // IPC_INFO(Linux 特有) 使用缓存区
};

// P 操作
void P(int semid, int index) {
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = -1;
    sem.sem_flg = 0;
    semop(semid, &sem, 1);
    return;
}

// V 操作
void V(int semid, int index) {
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = 1;
    sem.sem_flg = 0;

```

```

    semop(semid, &sem, 1);
    return;
}

void *saleTicket(void *arg) {
    // printf("thread %ld func started\n", (pthread_t *)arg - thread_id);
    // pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);

    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    // 收到 cancel 信号时运行到终止点再终止

    usleep(100); // 睡 100ms

    while (1) {
        pthread_testcancel(); // 设置终止点
        P(semid, 0);
        if (sale >= TICKET_NUM) {
            printf("thread \t %ld\t\t no ticket, thread exit\n",
                (pthread_t *)arg - thread_id);
            V(semid, 0);
            return NULL;
        }
        sale++;
        printf("sale thread \t %ld\t\t sales ticket \t%d\n",
            (pthread_t *)arg - thread_id, sale);
        V(semid, 0);
    }
}

int main() {
    sale = 0; // 开始时，已售票数量为 0

    // 创建信号灯
    semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);
    if (semid == -1) {
        printf("semget failed\n");
        return 0;
    }

    semun arg;
    arg.val = 1;
    if (semctl(semid, 0, SETVAL, arg) == -1) {
        semctl(semid, 0, IPC_RMID);
        printf("init sem 0 failed\n");
    }
}

```

```

        return 0;
    }
    // 给信号灯赋初值

    for (int i = 0; i < THREAD_NUM; i++) {
        // printf("it's time to create thread %d\n", i);
        if (pthread_create(thread_id + i, NULL, saleTicket,
                           (void *)&(thread_id[i])) == -1 ||
            thread_id[i] == 0) {
            printf("thread %d create failed\n", i);
            for (int j = 0; j < i; j++) {
                pthread_cancel(thread_id[j]);
            }
            for (int j = 0; j < i; j++) {
                pthread_join(thread_id[j], NULL);
                printf("thread %d is canceled\n", j);
            }
            // 终止所有线程 并等待线程终止完成
            return 0;
        }
        printf("thread %d create successful threadID is %ld\n", i,
              thread_id[i]);
    }

    // 所有售票线程创建完成

    for (int i = 0; i < THREAD_NUM; i++) {
        pthread_join(thread_id[i], NULL);
        printf("wait thread %d successful\n", i);
    }

    semctl(semid, 0, IPC_RMID); // 删除信号灯集

    printf("finished\n");

    return 0;
}

```

## 实验三

```

#include <stdio>
#include <iostream>
#include <stdlib.h>

```

```

#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/timeb.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

#define SHM_LEN 10000
#define SHM_NUM 5
#define INIT_SEM SHM_NUM
#define INPUT_FILE ("input.txt")
#define OUTPUT_FILE ("output.txt")
// #define INPUT_FILE ("/mnt/ramfs/output.txt")

struct buffer {
    int len;
    char b[SHM_LEN];
};

union semun {
    int val; // SETVAL 使用的值
    struct semid_ds *buf; // IPC_STAT、IPC_SET 使用缓存区
    unsigned short *array; // GETALL、SETALL 使用的数组
    struct seminfo *__buf; // IPC_INFO(Linux 特有) 使用缓存区
};

// P 操作
void P(int semid, int index) {
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = -1;
    sem.sem_flg = 0;
    semop(semid, &sem, 1);
    return;
}

// V 操作
void V(int semid, int index) {
    struct sembuf sem;
    sem.sem_num = index;
    sem.sem_op = 1;
    sem.sem_flg = 0;
    semop(semid, &sem, 1);
    return;
}

```

```

}

int getBufLen(char *);

int main(void) {
    // 创建共享内存
    int shmid[SHM_NUM];
    shmid_ds buf;

    for (int i = 0; i < SHM_NUM; i++) {
        shmid[i] = shmget(IPC_PRIVATE, sizeof(struct buffer), IPC_CREAT | 0666);
        if (shmid[i] == -1) {
            printf("create share memory %d failed\n", i);
            for (int j = 0; j < i; j++) {
                shmctl(shmid[j], IPC_RMID, &buf);
            }

            return 0;
        }
    }

    // 创建信号灯
    int read_sem = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);
    if (read_sem == -1) { // 信号灯创建失败
        printf("create read sem failed\n");
        return 0;
    }

    int write_sem = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);
    if (write_sem == -1) {
        printf("create write sem failed\n");
        return 0;
    }

    // 信号灯赋初值
    semun read_arg, write_arg;
    read_arg.val = 0;
    write_arg.val = INIT_SEM;
    if (semctl(read_sem, 0, SETVAL, read_arg) == -1 ||
        semctl(write_sem, 0, SETVAL, write_arg) == -1) {
        // 信号灯初始化失败
        semctl(read_sem, 0, IPC_RMID);
        semctl(write_sem, 0, IPC_RMID);
        printf("init sem failed\n");
        return 0;
    }
}

```

```

}

__pid_t readbuf, writebuf;
clock_t begin_clock, end_clock;
begin_clock = clock();
timeb begin, end;
ftime(&begin);
readbuf = fork();
if (readbuf == -1) {
    printf("main::create process readbuf failed\n");
    return 0;
}
if (readbuf != 0) {
    // main
    // printf("main::create readbuf process success\n");
    writebuf = fork();
    if (writebuf == -1) {
        printf("main::create process writebuf failed\n");
        kill(readbuf, SIGINT);
        return 0;
    }
    if (writebuf != 0) {
        // main
        // printf("main::create writebuf process success\n");
        waitpid(readbuf, NULL, 0);
        printf("main::process readbuf exit\n");
        waitpid(writebuf, NULL, 0);
        printf("main::process writebuf exit\n");
        // 等两个进程结束
        // 删除共享内存空间
        for (int i = 0; i < SHM_NUM; i++) {
            shmctl(shmid[i], IPC_RMID, &buf);
        }
        // 删除信号灯
        semctl(read_semid, 0, IPC_RMID);
        semctl(write_semid, 0, IPC_RMID);
        printf("main::process main exit\n");

        ftime(&end);
        end_clock = clock();
        printf("main::CPU time is %lf ms, run time is %lld ms\n",
            1000.0 * (end_clock - begin_clock) / CLOCKS_PER_SEC,
            (end.time - begin.time) * 1000 + end.millitm -
            begin.millitm);
    }
}

```

```

        return 0;
    } else {
        // write process
        printf("writebuf::process started\n");
        begin_clock = clock();

        // char *writeBuffer[SHM_NUM];
        struct buffer *writeBuffer[SHM_NUM];
        // detach 共享内存
        for (int i = 0; i < SHM_NUM; i++) {
            writeBuffer[i] =
                (struct buffer *)shmat(shmid[i], NULL, IPC_CREAT | 0666);
        }

        FILE *fp = fopen(INPUT_FILE, "r");
        if (fp == NULL) {
            printf("In writebuf process, open file error\n");
            kill(readbuf, SIGINT);
            return 0;
        }

        int count = 0;
        int writeSize = 0;
        while (1) {
            P(write_semid, 0);
            writeBuffer[count]->len =
                fread(writeBuffer[count]->b, sizeof(char), SHM_LEN, fp);
            if (writeBuffer[count]->len == 0) {
                printf("writebuf::write file data finished, "
                    "process exit\n");
                V(read_semid, 0);
                fclose(fp);
                ftime(&end);
                end_clock = clock();
                printf(
                    "writebuf::CPU time is %lf ms, run time is %lld ms\n",
                    1000.0 * (end_clock - begin_clock) / CLOCKS_PER_SEC,
                    (end.time - begin.time) * 1000 + end.millitm -
                    begin.millitm);
                exit(0);
            }

            // printf("writebuf::%s\n", writeBuffer[count]);
            V(read_semid, 0);
        }
    }
}

```



```

        count = (count + 1) % SHM_NUM;
    }
}
} else {
    // read process
    printf("readbuf::process started\n");
    begin_clock = clock();
    struct buffer *readBuffer[SHM_NUM];
    // char *readBuffer[SHM_NUM];
    // attach 共享内存
    for (int i = 0; i < SHM_NUM; i++) {
        readBuffer[i] =
            (struct buffer *)shmat(shmid[i], NULL, IPC_CREAT | 0666);
    }

    FILE *fp = fopen(OUTPUT_FILE, "w");
    if (fp == NULL) {
        printf("In readbuf process, open file error\n");
        kill(writebuf, SIGINT);
        return 0;
    }

    int count = 0;
    while (1) {
        P(read_semid, 0);
        if (readBuffer[count]->len == 0) {
            printf("readbuf::read file data finished, process exit\n");
            V(write_semid, 0);
            fclose(fp);
            ftime(&end);
            end_clock = clock();
            printf("readbuf::CPU time is %lf ms, run time is %lld ms\n",
                1000.0 * (end_clock - begin_clock) / CLOCKS_PER_SEC,
                (end.time - begin.time) * 1000 + end.millitm -
                begin.millitm);
            exit(0);
        }

        fwrite(readBuffer[count]->b, sizeof(char), readBuffer[count]->len, fp);
        // printf("readbuf::%s\n", readBuffer[count]);
        V(write_semid, 0);
        count = (count + 1) % SHM_NUM;
    }
}
}

```

```
    return 0;  
}
```