一、实验目的

1. 加深对进程同步于通信操作的直观认识；
2. 掌握 Linux 操作系统的进程、线程机制和编程接口；
3. 掌握 Linux 操作系统的进程和线程间的同步和通信机制；
4. 掌握经典同步问题的编程方法；

二、实验环境

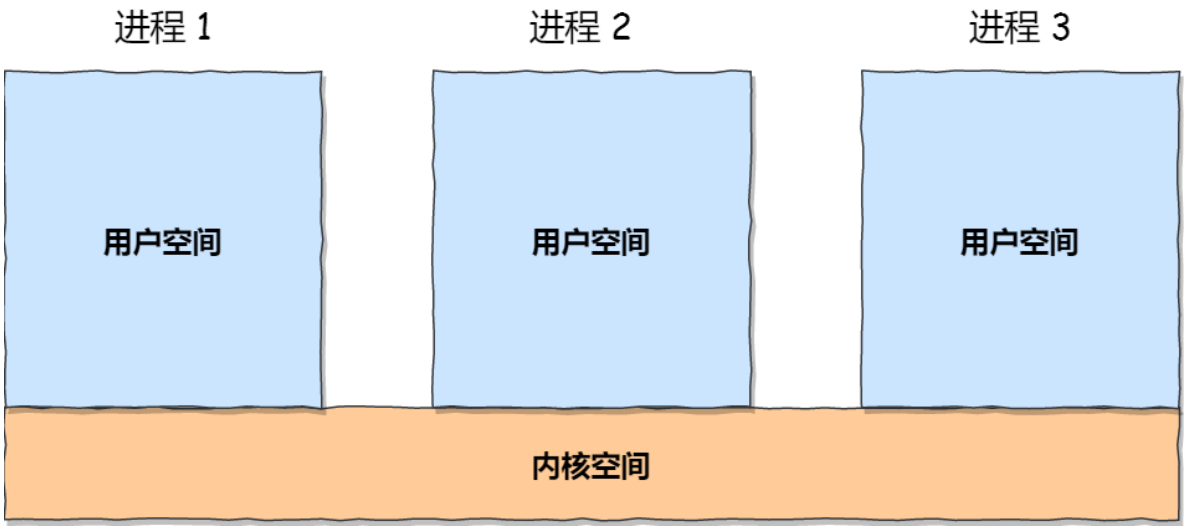
硬件：桌面 PC
软件：Linux 或其他操作系统

三、实验内容

1. 可以使用 Linux 虚拟机或其它 Unix 类操作系统；
2. 学习该操作系统提供的进程、线程创建的函数使用方法；
3. 学习该操作系统提供的共享内存、管道、消息队列的通信机制；
4. 利用该操作系统提供的进程间同步的信号量，线程间同步的互斥量使用方法

四、实验步骤与结果

1. 每个进程的用户地址空间都是独立的，一般而言是不能互相访问的，所以进程之间的通信必须通过内核。对应的示意图如下图所示



其中 Linux 内核提供了不少进程通信的机制，其中包括有管道、消息队列、共享内存，接下来笔者将会介绍这三种通信方式。

管道

其中很常见的例子就是显示有关 mysql 的进程，使用的指令为 ps -ef | grep mysql

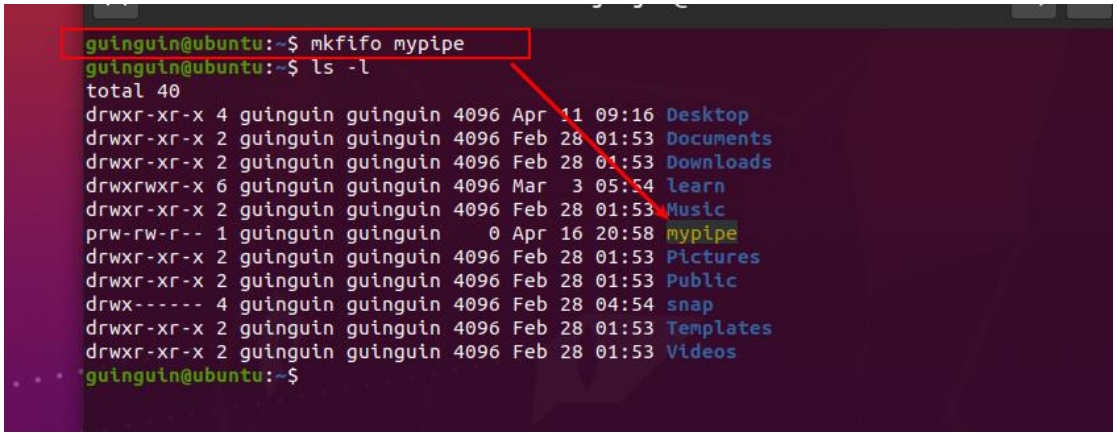
```
$ ps auxf | grep mysql
```

上面命令中的 '|' 就是一个管道，它的功能将前一个命令的 ps auxf 的输出，作为后一

个命令 grep mysql 的输入。同时在这里又可以看出，管道传输时单向的，如果想相互通信，我们需要创建两个管道才行。

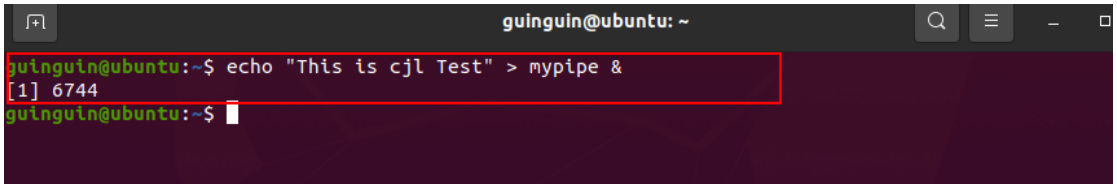
同时具体来讲，上述我们得知上面这种管道是没有名字的，所以该管道表示是一种匿名管道，用完了就直接销毁。

另一种管道就是命名管道，也被叫做 FIFO，因为数据是先进先出的传输方式。在使用命名管道前，先需要通过 mkfifo 命令来创建，并指定对应的管道的名字：

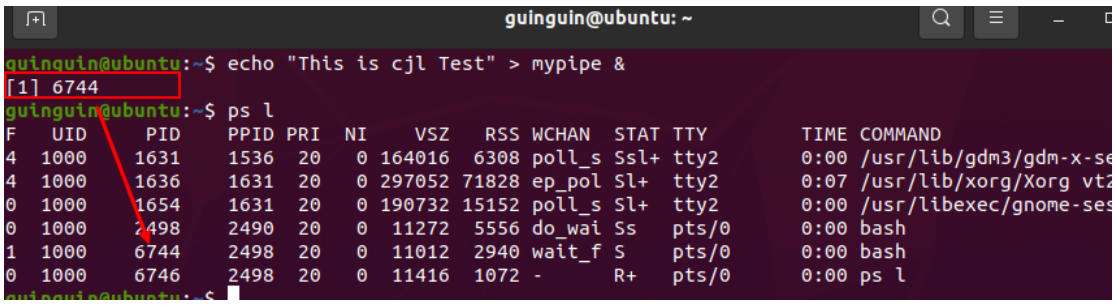


如上图所示，使用对应的指令创建了一个命名管道，并用 ls -l 指令进行查看，可以看到这个文件的类型就是 p，也就是 pipe 的首字母。

接下来，我们尝试向管道中写入一些数据来进行测试。如下图所示

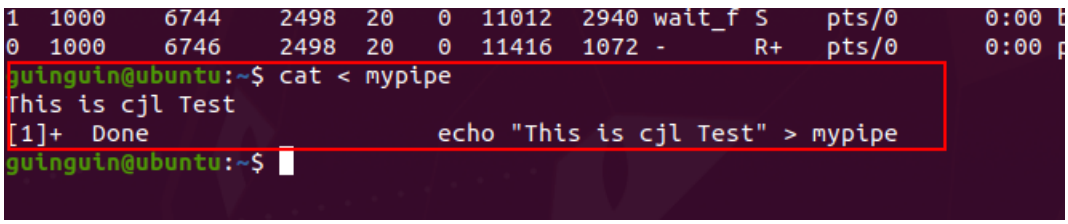


我们将 echo 指令放入后台运行，此时发现 echo 指令对应的进程并没有结束，而是在后台。



此时我们通过 ps l 指令正在处于 s 态，等待另一个进程进行数据的读取。

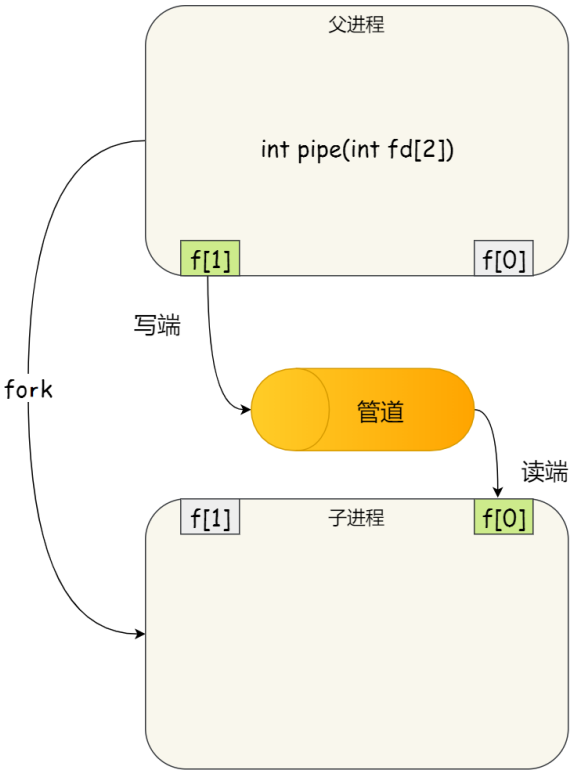
于是，我们执行另外一个命令来读取管道里的数据



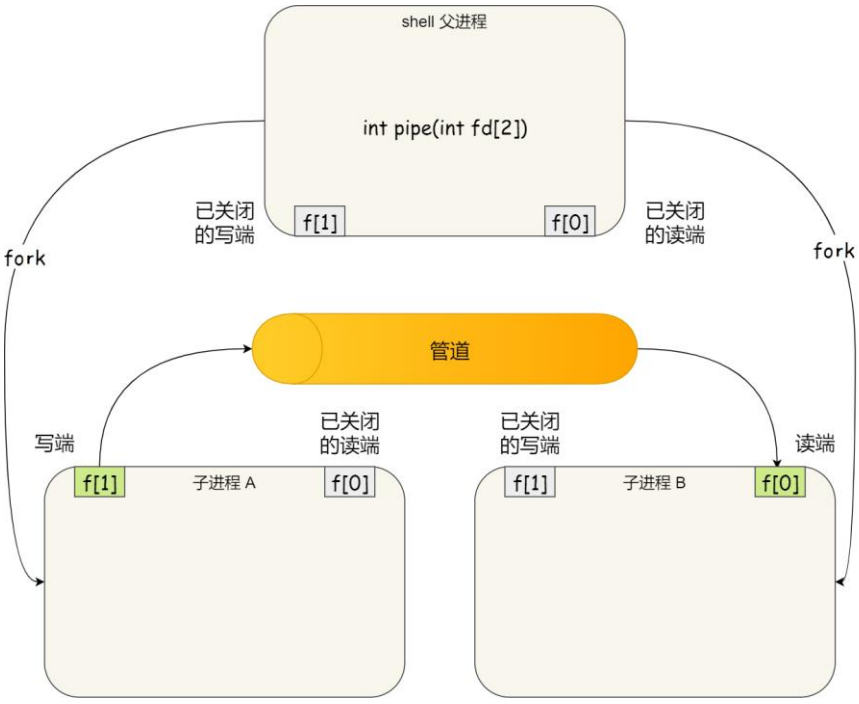
可以发现，成功获取到了写入管道的数据，并且对应的 echo 的进程成功结束。至此管

道数据的写入和读取流程就完成了，这也是两个进程进行管道通信的基础。通过分析我们可以看出，管道这种通信效率很低，不适合进程频繁地交换数据。当然，它的好处就在于实现非常简单，同时使用者也非常容易知道管道里的数据已经被另外一个进程获取了。

同时匿名管道创建背后原理使用了一个 pipe 的系统调用，在这里就不多做赘述，有兴趣的读者可以在网上的资料中参考一下。



父子进程匿名管道通信模型。



双子进程管道通信模型

另外，对于命名通道，它可以在不相关的进程中进行通信。因为命令管道，提前创建了一个类型为管道的设备文件，在进程里只要使用这个设备文件，不同进程之间就可以进行相互通信了。

最后，不管是匿名管道和命名管道，进程的写入的数据都是缓存在内核中，另一个进程读取数据的时候自然也是从内核中进行获取，同时通信的数据必须满足 FIFO(先进先出)的原则，同时说明了不支持 lseek 之类的文件定位的操作。

消息队列

前面说到管道的通信方式是效率低的，因此管道不适合进程间频繁地交换数据。

对于这个问题，消息队列的通信模式就可以解决。比如，A 进程要给 B 进程发送消息，A 进程把数据放在对应的消息队列后就可以正常返回了，B 进程需要的时候再去读取数据就可以了。同理，B 进程要给 A 进程发送消息也是如此。

再来，消息队列是保存在内核中的消息链表，在发送数据时，会分成一个一个独立的数据单元，也就是消息体（数据块），消息体是用户自定义的数据类型，消息的发送方和接收方要约定好消息体的数据类型，所以每个消息体都是固定大小的存储块，不像管道是无格式的字节流数据。如果进程从消息队列中读取了消息体，内核就会把这个消息体删除。

消息队列生命周期随内核，如果没有释放消息队列或者没有关闭操作系统，消息队列会一直存在，而前面提到的匿名管道的生命周期，是随进程的创建而建立，随进程的结束而销毁。

消息这种模型，两个进程之间的通信就像平时发邮件一样，你来一封，我回一封，可以频繁沟通了。

但邮件的通信方式存在不足的地方有两点，一是通信不及时，二是附件也有大小限制，这同样也是消息队列通信不足的点。

消息队列不适合比较大数据的传输，因为在内核中每个消息体都有一个最大长度的限制，同时所有队列所包含的全部消息体的总长度也是有上限。在 Linux 内核中，会有两个宏定义 MSGMAX 和 MSGMNB，它们以字节为单位，分别定义了一条消息的最大长度和一个队列的最大长度。

消息队列通信过程中，存在用户态与内核态之间的数据拷贝开销，因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，同理另一进程读取内核中的消息数据时，会发生从内核态拷贝数据到用户态的过程。

共享内存

消息队列的读取和写入的过程，都会有发生用户态与内核态之间的消息拷贝过程。那共享内存的方式，就很好的解决了这一问题。

现代操作系统，对于内存管理，采用的是虚拟内存技术，也就是每个进程都有自己独立的虚拟内存空间，不同进程的虚拟内存映射到不同的物理内存中。所以，即使进程 A 和进程 B 的虚拟地址是一样的，其实访问的是不同的物理内存地址，对于数据的增删查改互不影响。

共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中。这样这个进程写入的东西，另外一个进程马上就能看到了，都不需要拷贝来拷贝去，传来传去，大大提高了进程间通信的速度。

所以共享内存的本质就是通过虚拟地址空间同一映射到同一块物理内存中，这块区域就作为多个进程的共享空间。同时需要注意到多个进程共享使用内存空间就会产生一个进程的同步问题。为了防止多进程竞争共享资源造成错误的数据，一般再用信号量来保证任意时刻

只能被一个进程所访问。对应的信号量的内容在这里就不赘述了，不是第一部分的叙述重点。

总结：

Linux 内核提供了不少进程间通信的方式，其中最简单的方式就是管道，管道分为「匿名管道」和「命名管道」。

匿名管道顾名思义，它没有名字标识，匿名管道是特殊文件只存在于内存，没有存在于文件系统中，shell 命令中的「|」竖线就是匿名管道，通信的数据是无格式的流并且大小受限，通信的方式是单向的，数据只能在一个方向上流动，如果要双向通信，需要创建两个管道，再来匿名管道是只能用于存在父子关系的进程间通信，匿名管道的生命周期随着进程创建而建立，随着进程终止而消失。

命名管道突破了匿名管道只能在亲缘关系进程间的通信限制，因为使用命名管道的前提，需要在文件系统创建一个类型为 p 的设备文件，那么毫无关系的进程就可以通过这个设备文件进行通信。另外，不管是匿名管道还是命名管道，进程写入的数据都是缓存在内核中，另一个进程读取数据时候自然也是从内核中获取，同时通信数据都遵循先进先出原则，不支持 lseek 之类的文件定位操作。

消息队列克服了管道通信的数据是无格式的字节流的问题，消息队列实际上是保存在内核的「消息链表」，消息队列的消息体是可以用户自定义的数据类型，发送数据时，会被分成一个一个独立的消息体，当然接收数据时，也要与发送方发送的消息体的数据类型保持一致，这样才能保证读取的数据是正确的。消息队列通信的速度不是最及时的，毕竟每次数据的写入和读取都需要经过用户态与内核态之间的拷贝过程。

共享内存可以解决消息队列通信中用户态与内核态之间数据拷贝过程带来的开销，它直接分配一个共享空间，每个进程都可以直接访问，就像访问进程自己的空间一样快捷方便，不需要陷入内核态或者系统调用，大大提高了通信的速度，享有最快的进程间通信方式之名。但是便捷高效的共享内存通信，带来新的问题，多进程竞争同个共享资源会造成数据的错乱。

那么，就需要**信号量**来保护共享资源，以确保任何时刻只能有一个进程访问共享资源，这种方式就是互斥访问。信号量不仅可以实现访问的互斥性，还可以实现进程间的同步，信号量其实是一个计数器，表示的是资源个数，其值可以通过两个原子操作来控制，分别是 P 操作和 V 操作。

2. 编写程序实现生产者和消费者问题

查看题目，我们设计对应四个信号量来控制缓冲区的并发读写。

对应的设计如下

公共信号量 w_mutex 控制同一时间只有一人操作缓冲区

公共信号量 empty 表示缓冲区是否有空位，1 则有空位

公共信号量 full 表示缓冲区是否有数据，1 则有数据

消费者私有信号量 r_mutex 保证同一时间只有一个消费者在读数据

对应的生产者写数据的时候，具体的分为 5 个步骤：

① 等待 empty，确保缓冲区中有空位

② 等待 w_mutex，互斥访问缓冲区

③ 写操作

④ 释放 w_mutex，让出访问权

⑤ 释放 full，确保缓冲区中有空位

注意到互斥资源信号量 w_mutex 一定要放在最内层，防止死锁。

对于一个消费者读取数据，具体步骤如下。注意到有多个消费者，所以需要多加一个私有的互斥信号量 r_mutex 来保证只有一个消费者。

首先来看对应的生产者的程序 producer.c

```c
int main() {
    // share memory
    key_t key = (key_t)114514;
    int shmid = shmget(key, 2048, 0666 | IPC_CREAT);    // create memory return id 创建内存空间 返回内存id
    printf("share memory id: %d\n", shmid);
    printf("--------------------\n");
    void* buffer = shmat(shmid, 0, 0);  // get addr by memory id

    // create public mutex
    // 以下创建对应的公共信号量
    // w_mutex=1, empty=1, full=0
    sem_t* w_mutex = sem_open("w_mutex", O_CREAT | O_RDWR, 0666, 1);
    sem_t* empty = sem_open("empty", O_CREAT | O_RDWR, 0666, 1);
    sem_t* full = sem_open("full", O_CREAT | O_RDWR, 0666, 0);

    char str[2048];
```

开始先创建对应的 buffer 内存空间，供生产者和消费者互斥访问。在程序中用 shmget 创建内存空间然后返回对应的 memory id，之后再用 memory id 传入到 shmat 函数得到分配得到的 buffer 的地址。

然后创建对应的 public mutx，包括 w_mutex，empty，full 三个全局信号量，并且为其设置合理的初始值。

注意本次实验中设置的对应的 buffer 的大小为 2048 字节

```c
    // 启动一个生产者进程
    while(1) {
        printf("producer> ");
        scanf("%s", str);

        sem_wait(empty);    // wait empty
        sem_wait(w_mutex);  // wait w_mutex

        strcpy(buffer, str);//进行一个写(生产操作)

        sem_post(w_mutex);  // signal w_mutex
        sem_post(full);     // signal full
    }

    return 0;
```

初始化好共享区域 buffer 和信号量后，然后再启动生产者的核心功能。这里 while 表示生产者不断探测 buffer 区域尝试写入数据。根据上文的相关解释，先 wait empty 看缓冲区是否为空，然后再 wait w_mutex 看是否可以互斥访问。上述两个信号量通过后则开始进行写/生产操作，然后再进行一个释放 post 操作，顺序和 wait 恰好相反。

其中消费者进程的代码也类似，不过该进程需要启动两个消费者进程进行并发读取缓冲区，所以在这里使用 pthread_create 启动两个并行的消费者进程。

同时还需要注意到需要增加一个 r_mutex 信号量来进行两个消费者进程的并行互斥访问。

```c
    // create read mutex
    // 新增的一个信号量 a signal mutex add
    r_mutex = sem_open("r_mutex", O_CREAT | O_RDWR, 0666, 1);
```

```c
int main() {
    // share memory 注意到key必须和生产者的一致 属于同一块共享内存
    key_t key = (key_t)114514;
    int shmid = shmget(key, 2048, 0666 | IPC_CREAT);    // create
    buffer = shmat(shmid, 0, 0);    // get addr

    // get public mutex
    w_mutex = sem_open("w_mutex", O_CREAT | O_RDWR);
    empty = sem_open("empty", O_CREAT | O_RDWR);
    full = sem_open("full", O_CREAT | O_RDWR);

    // create read mutex
    // 新增的一个信号量 a signal mutex add
    r_mutex = sem_open("r_mutex", O_CREAT | O_RDWR, 0666, 1);
    pthread_t t1, t2;
    // 启动两个消费者进程
    pthread_create(&t1, NULL, (void*)&myRead, NULL);
    pthread_create(&t2, NULL, (void*)&myRead, NULL);
```

对应的 myRead 消费者核心逻辑如下图所示

```c
void myRead() {
    long int tid = (long int)syscall(__NR_gettid);
    while(1) {
        sem_wait(full); // wait full
        sem_wait(r_mutex);  // wait r_mutex
        sem_wait(w_mutex);  // wait w_mutex

        char str[2048];
        // 提取数据操作 消费者core code
        strcpy(str, buffer);
        printf("[tid: %lu]: %s\n", tid, str);
        memset(buffer, 0, 2048);    // clear

        sem_post(w_mutex);  // signal w_mutex
        sem_post(r_mutex);  // signal r_mutex
        sem_post(empty);    // signal empty
    }
}
```

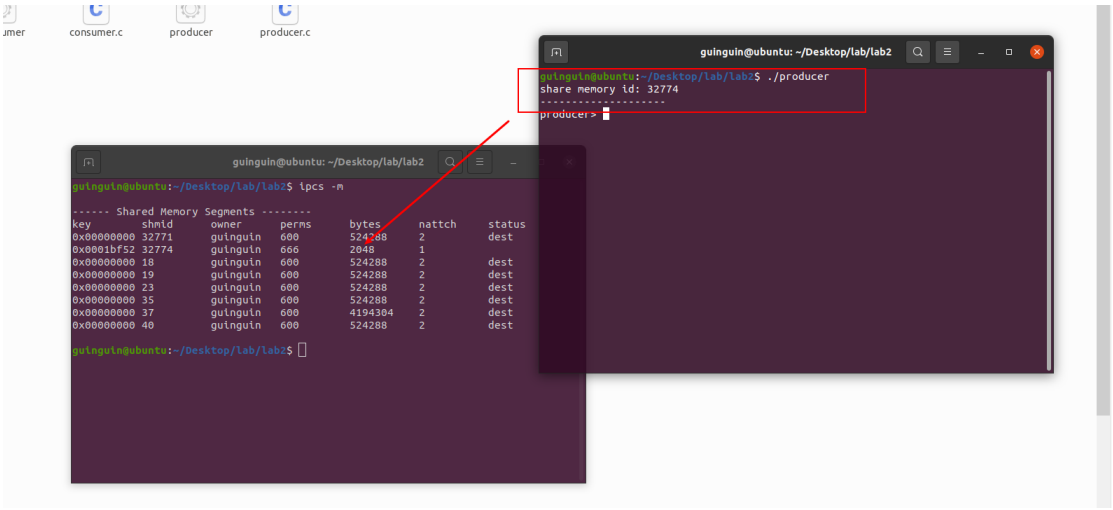
和生产者类似，要注意到每个信号量 wait 和 post 的顺序，防止出现死锁的现象。对应假如的 r_mutex 在 w_mutex 之前，full 之后，其他的顺序和消费者的顺序是一致的。接下来我们对该消费者和生产者程序进行编译运行。

注意到因为<pthread.h>只是声明，所以 gcc 编译的时候需要外部链接库，不然就会出现下列的报错信息。

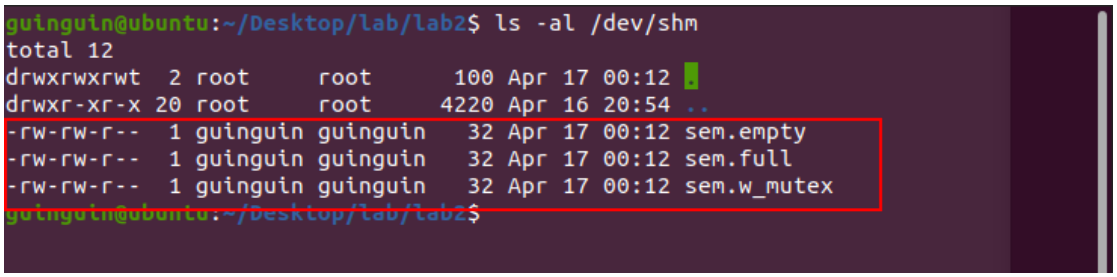
```
guinguin@ubuntu:~/Desktop/lab/lab2$ gcc producer.c -o producer
/usr/bin/ld: /tmp/ccs44O2q.o: in function `main':
producer.c:(.text+0xa4): undefined reference to `sem_open'
/usr/bin/ld: producer.c:(.text+0xcb): undefined reference to `sem_open'
/usr/bin/ld: producer.c:(.text+0xf2): undefined reference to `sem_open'
/usr/bin/ld: producer.c:(.text+0x134): undefined reference to `sem_wait'
/usr/bin/ld: producer.c:(.text+0x143): undefined reference to `sem_wait'
/usr/bin/ld: producer.c:(.text+0x16b): undefined reference to `sem_post'
/usr/bin/ld: producer.c:(.text+0x17a): undefined reference to `sem_post'
collect2: error: ld returned 1 exit status
```

加上-lpthread 外部链接后就可以编译成功了。

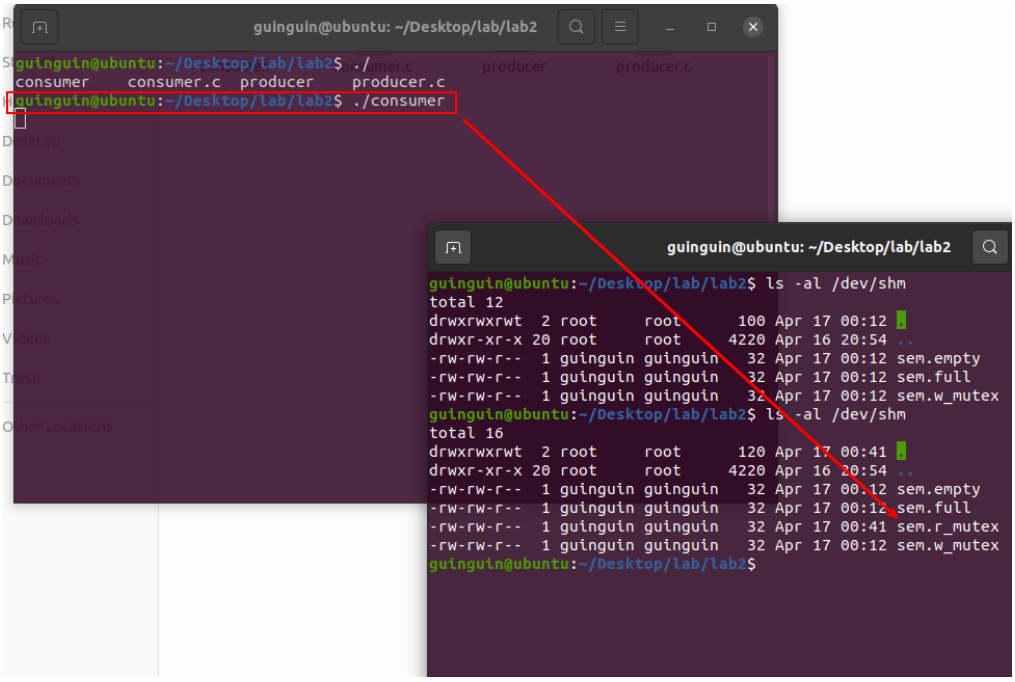
```
guinguin@ubuntu:~/Desktop/lab/lab2$ gcc producer.c -o producer -lpthread
guinguin@ubuntu:~/Desktop/lab/lab2$ gcc consumer.c -o consumer -lpthread
```



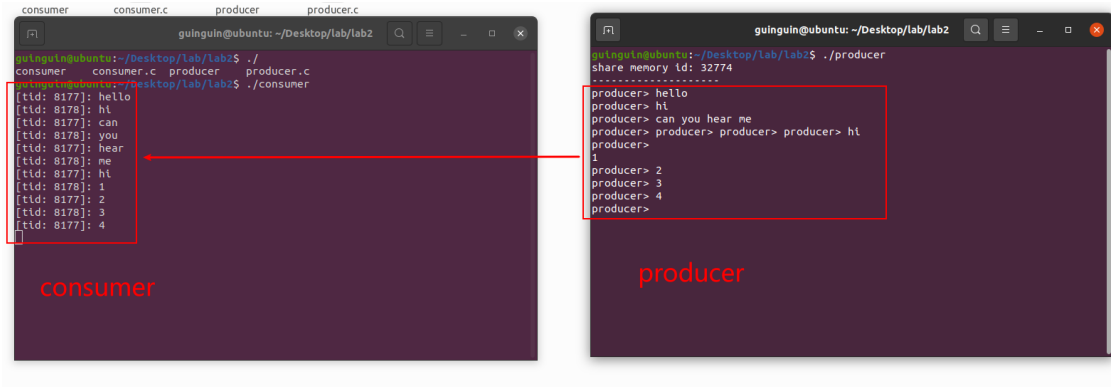
我们运行对应的生产者进程(produce)，然后再键入 ipcs-m 指令查看共享内存块，发现一块大小为 2048 的共享内存，对应的 key 也是在系统调用 shmget 时输入的 key，所以判定这一块区域就是我们生产者进程开辟出来的共享内存区域。

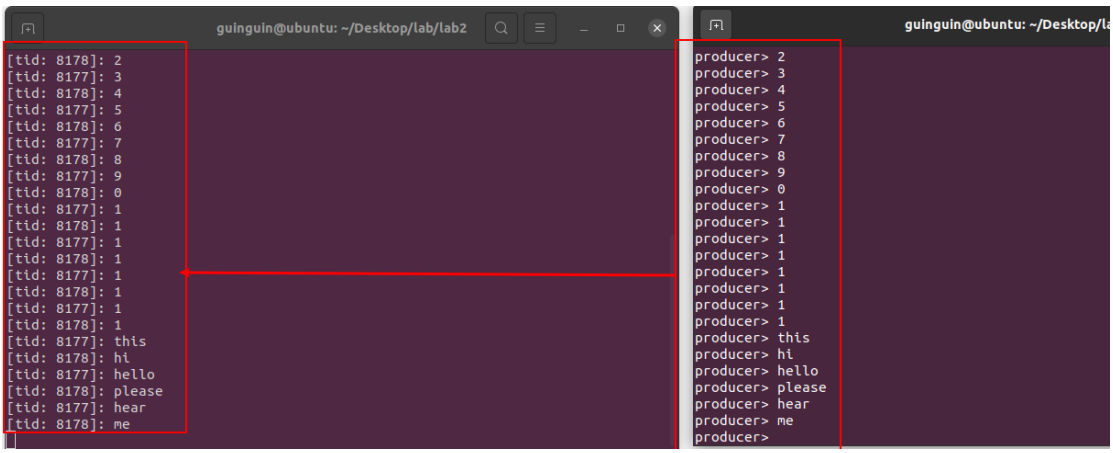


同时我们输入 ls -al /dev/shm，查看对应的目录文件来检查创建出来的信号量。可以看到启动生产者进程后产生了三个信号量，分别是 sem.empty，sem.full，sem.w_mutex。然后我们启动对应的消费者进程，然后再查看对应的信号量。



可以看到启动消费者进程后，信号量中增加了一个 sem.r_mutex 信号量。





然后我们再生产者进程中输入数据，会发现消费者进程会轮流读取共享内存中的数据。（消费者中的 tid 不停进行切换）。

两个消费者线程同时请求读缓冲区，在 8177 号线程霸占缓冲区的时候，因为信号量的机制，8178 号线程被挂载到信号量的等待队列。一旦 8177 号线程释放对应的信号量，8178 号线程马上就可以被调出来。

而因为是死循环 while1 不断请求读缓冲区，8177 号线程马上转而又被调进等待队列，等 8178 运行完后进行释放，马上又可以上处理机进行运行，同时这个时候 8178 又被调进了等待队列中，所以出现两个消费者线程交替读取缓冲区的情况。

我们此时尝试提前中止一下消费者，那么此时生产者会进入一种陷阱状态。

消费者终止后第一次生产没有问题，但是此时缓冲区已经被装满，而且因为消费者进程被中止了，无法进行缓冲区读取释放 empty 信号量，所以生产者再次进行生产的时候就会卡在 wait(empty)中无法推进，此时就会进入陷阱状态等待 empty 信号量的释放。

可以看到 produce>没有被打印，所以此时生产者进程被阻塞。

此时我们强制中止生产者进程

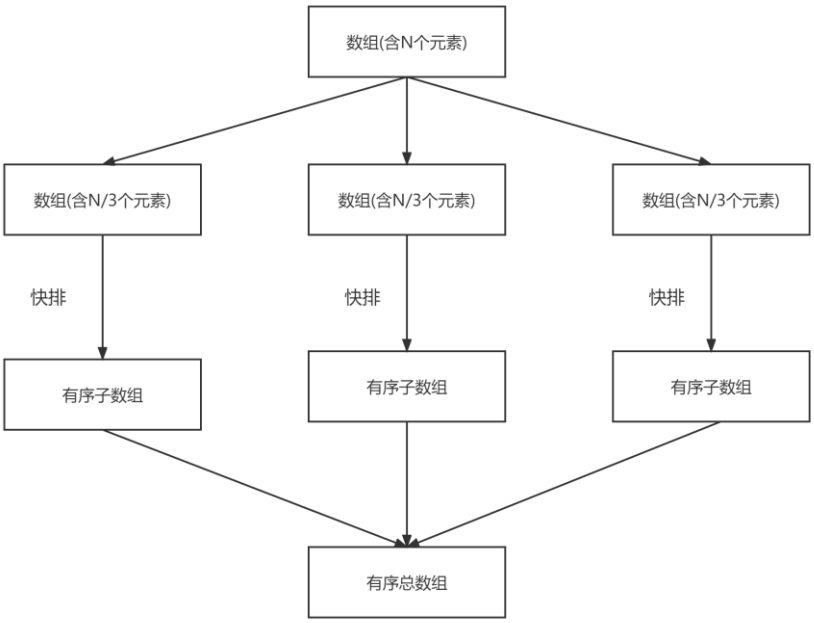
此时我们就完成了单生产者，多消费者模型的编写

在 linux 环境实现了经典的进程同步模型。

最后我们进行第三部分。

实现一个排序算法，算法输入为长度 600 万的浮点数（通过随机数生成），改为多线程形式（注意不同线程同步），比较不同线程数量（线程数量 1、2、4、6、8、10、16）条件下运行时间的差异，并结合课程理论进行分析。

对应的多线程的程序的设计思路也是非常简单，对应主要流程图如下图所示，下图为三线程排序作为例子。



对应的设计思路如上图所示

其中将数组均分为三份，然后对三个区间的数组使用多线程并行快排。其中对于这个例子而言，第一条快排分支对应线程 1，第二条快排分支对应线程 2，第三条快排分支对应线程 3。

这里例子是对应的线程数量为 3，我们也可以修改线程数量进行多次测试。

对应的具体代码实现如下

```
// 线程的数量
const int thread_num = 16;
// 数据量
const int data_num = 6000000;
// 容器的存储量 和数据量 保持一致
const int Maxn = data_num;
int a[data_num];
```

预先准备好各种数据，包括数据量，线程数量，排序容器等等，为后续编写程序打好准备。

```
// 多线程归并合并好的区间 归并的核心函数 二路归并
void merge(int left1, int right1, int left2, int right2) {
    int n = right2 - left1 + 1;
    int* data = new int[n];
    for(int i = 0; i < n; i++) data[i] = 0;
    int l1 = left1, l2 = left2, i = 0;
    while(l1 <= right1 && l2 <= right2) {
        if(a[l1] < a[l2]) data[i++] = a[l1++];
        else data[i++] = a[l2++];
    }
    while(l1 <= right1) data[i++] = a[l1++];
    while(l2 <= right2) data[i++] = a[l2++];
    int id = 0;
    for(int i = left1; i <= right2; i++) a[i] = data[id++];  // 将data中合并好的数据进行一个迁移
    delete[] data;
}
```

以上就是二路归并的代码，将对应的两块有序区域合并为一块大的有序区域，具体代码和归并排序中的 merge 一致，在这里不多做解释。

```
// 快排core代码 快速排序
void sort(int left, int right) {
    if(left >= right) return;
    int l = left, r = right;
    while(l < r) {
        while(l < r && a[l] <= a[r]) l++;
        if(l != r) swap(a[l], a[r]);
        while(l < r && a[l] <= a[r]) r--;
        if(l != r) swap(a[l], a[r]);
    }
    sort(left, l - 1);
    sort(l + 1, right);
    return;
}
```

接下来就是快排的核心代码，在这里使用双指针并交换的方式进行排序。每一轮下来 l 和 r 都停在 pivot 标兵处，最后递归处理每一个子串就可以了。这里给每一个线程提供了对应的快排基础。

```
void* __sort(void* segment) {
    struct node* Segment;
    Segment = (struct node*)segment;
    cout << "Thread " << Segment->id << ":Sort begin!" << endl;
    sort(Segment->l, Segment->r);
    cout << "Thread " << Segment->id << ":Sort End!" << endl;
    void* tmp;
    return tmp;
}
```

这个就是每一个线程对应的执行函数。每一个函数里面会传入一个 segment 来表示数组均分后的一段，然后线程调用的函数通过对这一段的左端点和右端点进行一个快速排序即可。

```
struct node {
    int l, r;
    int id;
};
```

其中 node 就是一个结构体，包含左右端点的大小即可对应的 id。

```
int main() {
    // 随机播种种子
    srand((unsigned)time(NULL));
    // 输入数据量为600w的数据
    for(int i = 0; i < data_num; i++) a[i] = rand();
    //cout << "init array" << endl;
    //for(int i = 0; i < data_num; i++) cout << a[i] << " ";
    cout << endl;
    node* s = new node[thread_num];
    // 将区间分为thread_num段 记录每一段的id 左端点 右端点
    // 一共需要分为 thread_num 个段
    // 每一端 左端点 l -> i * (data_num / thread_num)   r-> (i + 1) * (data_num / thread_num) - 1
    for(int i = 0; i < thread_num; i++) {
        s[i].id = i;
        s[i].l = i * (data_num / thread_num);
        s[i].r = (i + 1) * (data_num / thread_num) - 1;
    }
```

上图就是主函数，首先先随机生成对应数据大小的数组容器，然后对应进行平均分段，存储到 s 里面。

```
// 线程容器
pthread_t t[thread_num];

// 开始计时
time_t begin = clock();

// 启动对应的线程 最后一个传入的void* -> args
for(int i = 0; i < thread_num; i++) pthread_create(&t[i], NULL, __sort, (void*)&s[i]);
// 用来等待thread_num个子线程的结束 处理好所有十块区间的各自的快速排序
// 才可以回到对应的主线程进行一个合并的操作
for(int i = 0; i < thread_num; i++) pthread_join(t[i], NULL);
// 最后将排序好的thread_num块区域进行合并 得到最后的结果
int left = 0, right = data_num / thread_num - 1;
for(int i = 1; i < thread_num; i++) {
    merge(left, right, s[i].l, s[i].r);
    right += (data_num / thread_num);
}

// 结束计时
time_t end = clock();

// 打印时间
cout << (double)(end - begin) / CLOCKS_PER_SEC;
```

然后生命线程容器，对于每一个线程调用__sort 函数，并传入对应的分段，同时还要注意为了保证归并之前所有的并行快排都已经排序完成，需要对每一个线程做一个 join 同步的函数，保证在归并前数组局部有效。最后再通过 merge 归并所有的有序子数组得到一个大数组然后停止计时并打印。

具体的源代码我也已经以附件的方式提交，老师可以具体参考与查看。

测试结果

单线程

```
Thread 0:Sort begin!
Thread 0:Sort End!
0.568
```

双线程

```
Thread 0:Sort begin!
Thread 1:Sort begin!
Thread 1:Sort End!
Thread 0:Sort End!
0.27
```

四线程

```
Thread 0:Sort begin!
Thread 3:Sort begin!
Thread 1:Sort begin!
Thread 2:Sort begin!
Thread 0:Sort End!
Thread 1:Sort End!
Thread 3:Sort End!
Thread 2:Sort End!
0.156
```

六进程


Thread 0:Sort begin!
Thread 3:Sort begin!
Thread 1:Sort begin!
Thread 4:Sort begin!
Thread 5:Sort begin!
Thread 2:Sort begin!
Thread 4:Sort End!
Thread 2:Sort End!
Thread 0:Sort End!
Thread 5:Sort End!
Thread 3:Sort End!
Thread 1:Sort End!
0.126

八线程

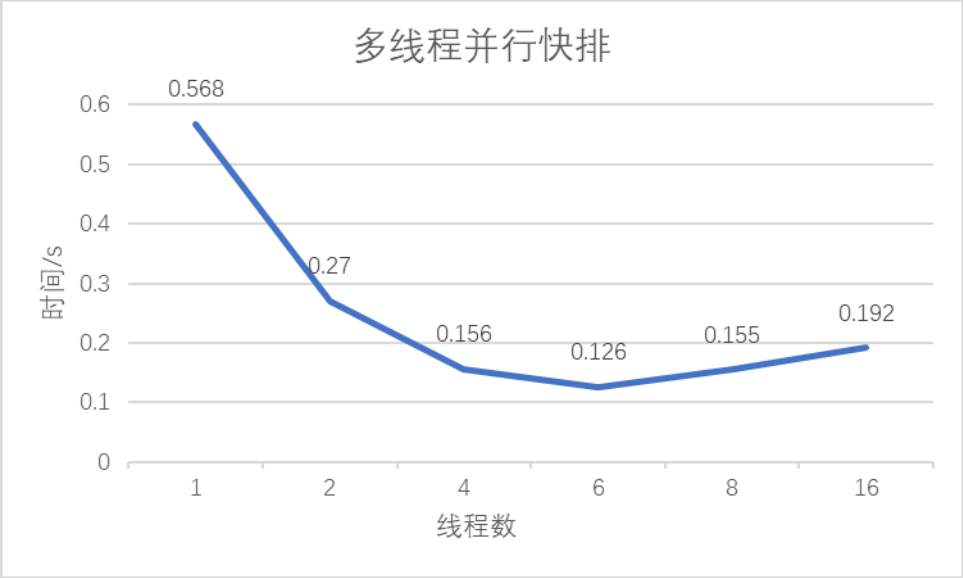

Thread 0:Sort begin!
Thread 4:Sort begin!
Thread 3:Sort begin!
Thread 2:Sort begin!
Thread 1:Sort begin!
Thread 5:Sort begin!
Thread 7:Sort begin!
Thread 6:Sort begin!
Thread 0:Sort End!
Thread Thread 3:Sort End!
1:Sort End!
Thread Thread 4:Sort End!
2:Sort End!
Thread 5:Sort End!
Thread 6:Sort End!
Thread 7:Sort End!
0.155

十六线程


Thread 0:Sort begin!
Thread 8:Sort begin!
Thread 4:Sort begin!
Thread 9:Sort begin!
Thread 2:Sort begin!
Thread 14:Sort begin!
Thread 1:Sort begin!
Thread 12:Sort begin!
Thread 7:Sort begin!
Thread 10:Sort begin!
Thread 6:Sort begin!
Thread 15:Sort begin!
Thread 5Thread 13:Sort begin!
Thread 9Thread 4:Sort End!
Thread 14:Sort End!
Thread 12Thread 3:Sort begin!
Thread 0:Sort End!
Thread 15:Sort End!
:Sort End!
Thread 1:Sort End!
Thread 11:Sort begin!
:Sort End!
:Sort begin!
Thread 2:Sort End!
Thread 6:Sort End!
Thread 10:Sort End!
Thread 8:Sort End!
Thread 7:Sort End!
Thread 13:Sort End!
Thread 3:Sort End!
Thread 11:Sort End!
Thread 5:Sort End!
0.192

对应的打印信息和时间如上图所示，同时把对应的时间做成和线程数做成可视化折线图，结果如下图所示。

	A	B	C	D	E	F	G
1	线程数	1	2	4	6	8	16
2	时间/s	0.568	0.27	0.156	0.126	0.155	0.192



可以看到，线程数为 6 的时候花费的时间是最少的，当开辟的线程数小于 6 或者大于 6 的时候，时间都会变长，效率都会变低。由进程的相关知识进行分析，当线程数小的时候，排序的并行度较低，不能很好的发挥多线程的有点，所以效率会不够好。当线程数太大的时候，线程切换以及线程同步的开销远大于并行度提升带来的好处，此时排序的效率又会大大减小。故当线程数达到一个适中的值的时候，并行度和线程切换和同步带来的开销达到平衡和最优的时候，此时的效率最高。

对于本次实验而言，对于 600w 数据的排序，其中线程数为 6 的时候效率是最高的。

五、心得体会

1 通过本次实验，我对进程的通信，包括管道，消息队列，共享内存，信号量等概念有了更加深刻和形象的理解

2 通过本次实验，自主完成一个生产者，两个消费者线程通信和同步的模型的编写，通过自己设定信号量同步线程，自己开辟共享内存进行通信，对线程同步的底层有了更加深刻的认识

3 通过对多线程并行排序的编写，更加熟悉通过 Liunx 系统调用来完成多线程编程，对多线程环境下的排序算法有了更加深刻的认识和理解

4 对 Linux 系统的操作更加熟练，对各种 Linux 指令更加熟练运用

5 感受到了多线程的魅力和强大，不同于普通单线程的编程环境，我们需要考虑的因素更多，包括一些同步、互斥访问的问题

6 线程并不是越多越好，并行度越高的程序带来的后果就是线程切换和同步的开销更大，反而程序的效率还会降低