```
    ...;
}
```

2.3.7 经典同步问题

在多道程序环境下，进程同步问题非常重要。有很多经典的进程同步问题，其中最具代表性的是“生产者 – 消费者问题”、“哲学家进餐问题”和“读者 – 写者问题”。

1. 生产者 – 消费者问题

生产者 – 消费者问题是对合作进程中内部关系的抽象化。例如，当输入进程和计算进程相互合作时，输入进程属于生产者（输入要计算的变量值），计算进程可以归为消费者；对于计算和打印进程，计算进程可以归为生产者，而打印进程则属于消费者。因此，生产者 – 消费者问题是一种最具实用性和代表性的同步问题。

生产者 – 消费者问题可以描述为一组生产者和一群消费者一起工作，通过一个大小为 n 的有限缓冲区进行生产和消费。一个缓冲区可以容纳 n 个产品，其中生产者负责投放产品，消费者负责消费产品。

由于生产者和消费者共享的缓冲区的大小为 n，缓冲区最多可以容纳 n 个产品。因此，生产者和消费者进程在操作过程中受到以下两个约束：

(1) 如果缓冲区已满，生产者不能继续生产，否则会使产品溢出。

(2) 如果缓冲区为空，消费者不能继续消费，否则会导致从空缓冲区拿取产品。

图 2.21　生产者 – 消费者问题的同步关系

如图2.21所示，为避免产品溢出或从空缓冲区拿取产品，应满足以下同步规则：

(1) 如果缓冲区已满，生产者生产结束后应当被阻塞，等待消费者取出产品、缓冲区有空位后将其唤醒。

(2) 如果缓冲区空，那么消费者试图拿取产品时也必须被阻塞，等待生产者把产品放进去后将其唤醒。

为了实现上述进程的同步关系，可以设置两个信号量：empty 表示当前空缓冲区的数量，初始值为 n；full 表示当前非空缓冲区的数量，初始值为 0。此外，缓冲区是临界资源，需设置互斥信号量 mutex，实现生产者和消费者进程对缓冲区的互斥访问。

生产者 – 消费者问题的程序实现如下：

```
semaphore mutex = 1, empty = n, full = 0;
void producer() {
   while(TRUE) {
      生产一个新的产品;
      P(empty);
      P(mutex);
      将产品放入空闲缓冲区;
      V(mutex);
      V(full);
```

```
    }
}
void consumer() {
    while(TRUE) {
        P(full);
        P(mutex);
        从缓冲区取出一个产品;
        V(mutex);
        V(empty);
        消费该产品;
    }
}
```

信号量 mutex 保证同一时刻只能有一个进程对缓冲区进行操作（放入或取走产品）。需要注意的是 P 操作的顺序很重要，P(empty/full) 必须在 P(mutex) 之前，不能颠倒。否则假设消费者首先访问临界区，会被阻塞，但此时生产者希望往缓冲区放东西时，无法进入临界区，就形成了死锁。V 操作的顺序则允许颠倒。

2. 哲学家进餐问题

哲学家进餐问题是进程同步的一个典型问题。问题描述：如图2.22所示，有五个哲学家（图中表示为圆圈），他们共用一张圆桌，围坐在五把椅子上，桌上有五个碗和五只筷子（筷子表示为线条），他们的行为模式是思考和进餐交替进行。通常情况下，哲学家会思考。当哲学家饥饿的时候，他会尝试拿取离自己最近的左右两根筷子，只有当他手上有两根筷子的时候，才能吃饭。饭后，他放下筷子，继续思考。

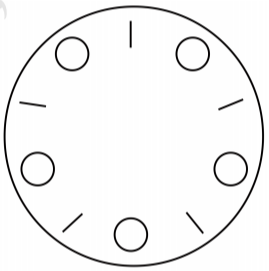

图 2.22 哲学家进餐示意图

在哲学家进餐问题中，临界资源是筷子。值得注意的是，该问题和生产者 – 消费者问题的区别在于由于位置的不同，每根筷子都不相同。必须为每根筷子分别定义一个初始值为 1 的互斥信号量 chopstick[i]，其代码如下所示：

```
semaphore chopstick[5] = {1, 1, 1, 1, 1};
```

其中第 i 位哲学家的活动可描述如下：

```
while(TRUE) {
    P(chopstick[i]);
    P(chopstick[(i+1) % 5]);
    |$\cdots$|
    吃饭;
    |$\cdots$|
    V(chopstick[i]);
    V(chopstick[(i+1) % 5]);
    |$\cdots$|
    思考;
    |$\cdots$|
}
```

上述方法保证了筷子的互斥使用，但仍可能导致死锁发生。如果五个哲学家同时拿起他们左边的筷子，就会出现循环等待的情况，导致死锁发生。常见的几种解决方案如下：

(1) 最多允许四名哲学家同时进食（破坏了死锁产生的“循环等待”条件）。

(2) 只有在哲学家的左右两边都有筷子的情况下才被允许就餐（破坏了死锁产生的“请求和保持”条件）。

(3) 让编号为奇数的哲学家先拿左筷子，再拿右筷子，编号为偶数的哲学家先拿右筷子，再拿左筷子（破坏了死锁产生的“循环等待”条件）。

【提示】死锁产生的四个必要条件将在下一节详细阐述。

3. 读者 – 写者问题

读者 – 写者问题是对数据对象（数据文件或记录）的访问模型，问题描述如下：有一个共享文件 F，允许几个进程（读者）同时读取 F 中包含的信息，但在任何时刻只允许一个进程（写者）写入或修改 F 中的信息。当一个进程正在读取，不允许其他进程写入（但可以读取）；当一个进程正在写入，其他进程的任何读写操作均不被允许。

【提示】若生产者作为一个写进程，消费者作为一个读进程，生产者 – 消费者问题能不能看作是一个特殊的读者 – 写者问题？答案是否定的。生产者与写进程的区别在于，它还会同时读取队列指针，确定在哪里写，以及确定缓冲区是否已满；消费者也不仅仅是一个读进程，其同样涉及队列指针和缓冲区操作。

读者 – 写者问题有三种典型的算法，即读者优先算法、写者优先算法和读写公平算法，下面分别进行讨论。

① 读者优先

在该算法中，读进程具有优先权，其访问临界区时，只要有一个读进程在读，读进程就能继续获得对临界区的控制权，但可能会导致写进程饥饿。读者优先的算法描述如下：

(1) 当一个读进程正在读取时，写进程必须等待。

(2) 在写进程等待时，随后的读进程仍可以进入临界区读取。

(3) 只有在没有读进程在读取的情况下，写进程才可以写入。

读者优先的程序实现如下：

```
semaphore rmutex = 1; //定义readcount的互斥信号量，初始值为1
semaphore wmutex = 1; //定义用于写的互斥信号量，初始值为1
int readcount = 0; //用于记录当前读者数量，初始值为0
void reader() {  //读进程
   do {
      P(rmutex); //（readcount操作互斥）对rmutex执行P操作，防止他人访问
      if (readcount == 0)
         P(wmutex); //（同步）如果自己是第一个读者，禁止写者写
      readcount++; //读者数量加1
      V(rmutex); //（readcount操作互斥）对rmutex执行V操作，允许他人访问
      进行读操作;
      P(rmutex); //（readcount操作互斥）对rmutex执行P操作，防止他人访问
      readcount--; //读者数量减1
      if (readcount == 0)
         V(wmutex); //（同步）如果自己是最后一个读者，允许写者写
      V(rmutex); //（readcount操作互斥）对rmutex执行V操作，允许他人访问
    } while(TRUE);
}
void writer() { // 写进程
   do {
      P(wmutex); //  （写互斥）对wmutex执行P操作，防止他人访问文件
      进行写操作;
      V(wmutex); //  （写互斥）对wmutex执行V操作，允许他人访问文件
   } while(TRUE);
}
```

首先，定义一个初始值为 1 的互斥信号量 wmutex，第一个进入的读者和所有进入的写者在对

共享文件 F 进行操作之前，必须先执行 P(wmutex) 操作，在结束对 F 的操作后，必须执行 V(wmutex) 操作。

此外，定义一个整数型变量 readcount，用于表示当前正在读取 F 的进程数量。读进程在读取 F 之前，必须执行 readcount 加 1 的操作；而在读取 F 之后，必须执行 readcount 减 1 的操作。readcount 变量是一种临界资源，因此定义一个 rmutex 信号量，实现所有的读者进程对 readcount 变量的互斥访问。

② 写者优先

在该算法中，写进程具有优先权，如果有写进程希望访问临界区，就会禁止新的读进程进入临界区，解决了写进程的饥饿问题。写者优先的算法描述如下：

(1) 当一个读进程在读取时，写进程不能进入临界区。

(2) 当一个写进程表示希望进行写入时，随后的读进程不得进入临界区读取。

(3) 只有当所有的写进程都离开临界区后，读进程才可以进入临界区读取。

写者优先的程序实现如下：

```
semaphore rmutex = 1; //定义readcount的互斥信号量，初始值为1
semaphore wmutex = 1; //定义writecount的互斥信号量，初始值为1
semaphore read_write = 1; //用于在读取时检测是否有写者的互斥信号量，初始值为1
semaphore write_read = 1; //用于在写入时防止他人进入的互斥信号量，初始值为1
int readcount = 0; //用于记录当前读者数量，初始值为0
int writecount = 0; //用于记录当前写者数量，初始值为0
void reader(){ //读进程
   do{
      P(read_write); //(读写互斥)在读取时检测是否有写者，如有写者则等待
      P(rmutex); //(readcount操作互斥)对rmutex执行P操作，防止其他读者访问
      readcount++; //读者数量加1
      if(readcount == 1)
         P(write_read); //(读写互斥)如果自己是第一个读者，禁止写者写
      V(rmutex); //(readcount操作互斥)对rmutex执行V操作，允许其他读者访问
      V(read_write); //(读写互斥)允许其他读者或写者访问文件
      进行读操作;
      P(rmutex); //(readcount操作互斥)对rmutex执行P操作，防止其他读者访问
      if(readcount == 1)
         V(write_read); //(读写互斥)如果自己是最后一个读者，允许他人读写
      readcount--; //读者数量减1
      V(rmutex); //(readcount操作互斥)对rmutex执行V操作，允许其他读者访问
   } while(TRUE);
}
void writer(){ //写进程
   do{
      P(wmutex); //(写互斥)对wmutex执行P操作，防止其他写者访问writecount
      writecount++; //写者数量加1
      if(writecount == 1)
         P(read_write); //(读写互斥)如果自己是第一个写者，禁止读者读
      V(wmutex); //(写互斥)对wmutex执行V操作，允许其他写者访问writecount
      P(write_read); //(读写互斥)若没有读者或写者在访问文件，则可写入
      进行写操作;
      V(write_read); //(读写互斥)写操作完成，允许其他读者或写者访问文件
      P(wmutex); //(写互斥)对wmutex执行P操作，防止其他写者访问writecount
      if(writecount == 1)
         V(read_write); //(读写互斥)如果自己是最后一个写者，允许读者读
      writecount--; //写者数量减1
      V(wmutex); //(写互斥)对wmutex执行V操作，允许其他写者访问writecount
```

```
    } while(TRUE);
}
```

首先定义两个信号量 read_write 和 write_read，初始值均为 1，用于互斥地读和写。定义两个整型变量 readcount 和 writecount，表示当前读取 F 和写入 F 的进程数量。在读者读取 F 之前，readcount 加 1，在完成读取 F 之后，readcount 减 1。同理，在写者写入 F 之前，writecount 加 1，在完成写入 F 之后，writecount 减 1。变量 readcount 和 writecount 是临界资源，因此定义信号量 rmutex 和 wmutex，分别实现读者进程对 readcount 变量的互斥访问和写者进程对 writecount 变量的互斥访问。

③ 读写公平

读写公平算法不偏袒读者或写者中的任何一方，仅根据读写请求的到达顺序来赋予读写的权利。读写公平的算法描述如下：

(1) 当没有读进程或写进程时，允许读进程或写进程进入。

(2) 当一个读进程试图读取时，若有写者正在等待写操作或进行写操作时，应等待其完成写操作后，才可以开始读操作。

(3) 同理，当一个写进程试图写入时，若有读者正在等待或进行读操作时，也需要进行等待。

读写公平的程序实现如下：

```
semaphore rmutex = 1; //定义readcount的互斥信号量，初始值为1
semaphore wmutex = 1; //定义用于写的互斥信号量，初始值为1
semaphore read_write = 1; //用于在读取时检测是否有写者的互斥信号量，初始值为1
int readcount = 0; //用于记录当前读者数量，初始值为0
void reader() { //读进程
   do {
      P(read_write); //（读写互斥）在读取时检测是否有写者，如有写者则等待
      P(rmutex); //（readcount操作互斥）对rmutex执行P操作，防止他人访问
      if (readcount == 0)
         P(wmutex); //（同步）如果自己是第一个读者，禁止写者写
      readcount++; //读者数量加1
      V(rmutex); //（readcount操作互斥）对rmutex执行V操作，允许他人访问
      V(read_write); //（读写互斥）允许其他读者或写者访问文件
      进行读操作;
      P(rmutex); //（readcount操作互斥）对rmutex执行P操作，防止他人访问
      readcount--; //读者数量减1
      if (readcount == 0)
         V(wmutex); //（同步）如果自己是最后一个读者，允许写者写
      V(rmutex); //（readcount操作互斥）对rmutex执行V操作，允许他人访问
   } while(TRUE);
}
void writer(){ // 写进程
   do {
      P(read_write); //（读写互斥）检测是否有读者或写者，如有则等待
      P(wmutex); //（写互斥）对wmutex执行P操作，防止他人访问文件
      进行写操作;
      V(wmutex); //（写互斥）对wmutex执行V操作，允许他人访问文件
      V(read_write); //（读写互斥）允许其他读者或写者访问文件
   } while(TRUE);
}
```