并发性：同步和互斥

并发

- 单处理器，多道程序
  - 交替执行（伪并行）
- 多处理器，多道程序
  - 并行执行
- 并发导致的问题
  - 死锁：多个进程因相互等待所占用的资源而不能继续执行
  - 活锁：多个进程间为了响应其他进程中的变化而继续改变自己的状态，但不做有用的工作
  - 饥饿：可运行的进程尽管可以继续执行，但被调度器无限忽视而不能被调度执行
  - *竞争：多个线程或进程在读写同一个共享数据时，结果依赖于他们的相对执行时间*
    - 通过引入互斥来解决

进程同步和互斥

- 临界资源：一次只能被一个进程使用的资源
  - 被一个进程使用的意思是无论这个进程处于ready，blocked 还是running都不能被其他进程使用
- 临界区：一段代码用于访问临界资源，不能同时有两个以上的进程同时使用
- **互斥：多个进程不能同时访问临界资源的现象**
- **同步：多个进程之间为完成共同任务基于某个条件来协调执行先后关系而产生的协作制约关系**
  - **本质上来说，同步也是一种互斥，但是同步是更为严格的一种互斥，进程之间有顺序关系**

互斥的实现方法

- 原子操作：不能被打断的操作
- 中断禁用（硬件方式）
  - 进入临界区前禁用中断，离开临界区使能中断
  - 缺点
    - 无法适用于多处理器
    - 其他线程可能处于饥饿状态，处理器的效率降低
- 专用机器指令（软件方式）
- **信号量**
- 管程

信号量

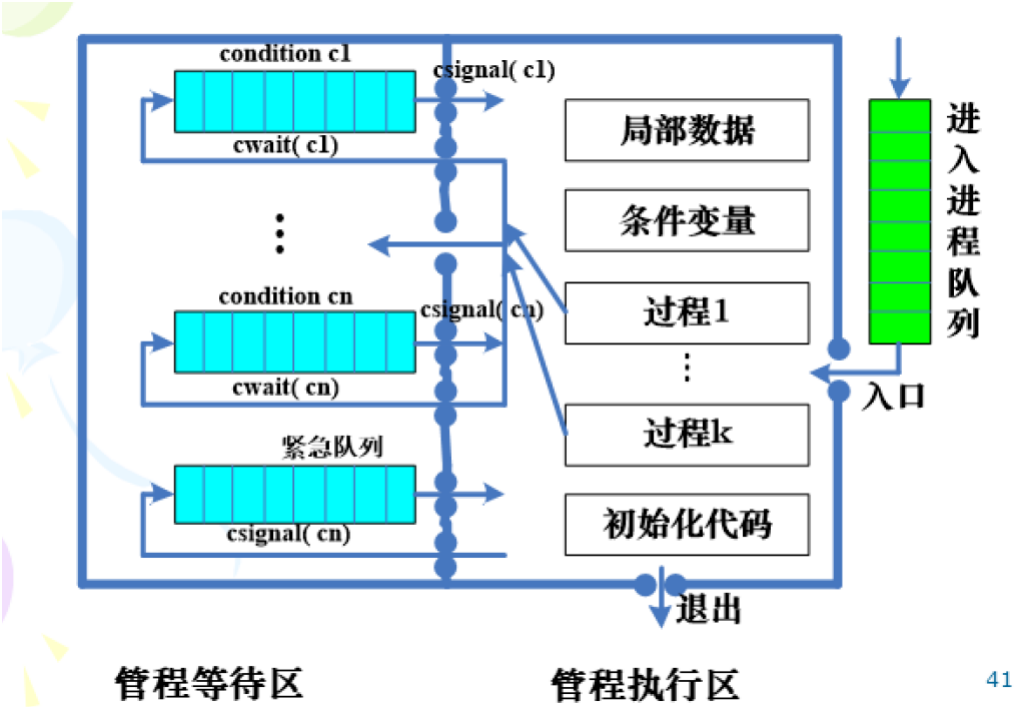
- 整型变量
- wait操作使信号量-1，如果信号量<0，执行wait操作的进程blocked
- signal操作使信号量+1，如果信号量<=0，被wait操作阻塞的进程唤醒
- **具体来说，进程进入临界区时，执行wait操作（加标签），出临界区时，执行signal操作（去标签）**
- ```
  struct semaphore{
      int count;
      queueType queue;
  }
  void semWait(semaphore s){
      s.count --;
      if (s.count<0){
          place this process in s.queue;
          block this process;
      }
  }
  void semSignal(semaphore s){
      s.count++;
      if (s.count<=0){
          remove a process P from s.queue;
          place process P on ready list;
      }
  }
  ```

互斥信号量

- 取值为0 和 1
- **Wait操作检查信号量的值**
  - 为0，执行wait操作的进程blocked。
  - 为1，将互斥信号量的值置为0，执行进程后续的操作
- **Signal操作检查阻塞进程队列**
  - 如果非空，被wait操作阻塞的进程唤醒
  - 如果为空，将互斥信号量的值置为1
- **也就是说，互斥信号量为1代表当前没有进程在使用临界资源，为0代表有一个进程在使用临界资源，但是不知道是否有进程在处于阻塞状态**
- ```
  struct binary_semaphore{
      enum {1,0} value;
      queueType queue;
  };
  void semWaitB(binary_semaphore s){
      if (s.value == 1)
          s.value = 0;
      else{
          place this process in s.queue;
  ```

```
        block this process;
    }
}
void semSignalB(binary_semaphore s){
    if (s.queue.is_empty()==True){
        s.value = 1;
    }else{
        remove a process P from s.queue;
        place process P on ready list;
    }
}
```

管程

- 一种程序设计模式/封装的数据结构，可以实现进程的互斥与同步
- 管程的互斥机制
  - 任意时刻管程中只能有一个活跃进程
- 管程中的同步机制
  - 条件变量 condition
    - `cwait(c)`：将调用wait操作的进程**阻塞**在该条件变量对应的队列里
    - `csignal(c)`：从该条件变量对应的队列里**释放一个阻塞进程**
  - 

    - 紧急队列：执行`csignal(cn)`操作后，从`condiction cn`队列中取出一个阻塞进程转为就绪态存放于紧急队列中
      - 之后根据调度算法决定紧急队列中的进程使用管程还是正在使用管程的进程继续使用管程
      - 紧急队列的优先级更高
    - **只有进入进程队列和紧急队列中的进程处于就绪态**
- Hoare管程
  - 执行`Csignal(cn)`操作后，释放一个阻塞进程，进行进程切换，停掉现在正在使用管程的进程，刚释放的进程使用管程
- Mesa管程

- `Cnotify(cn)` **释放一个阻塞进程**，转为就绪态，存放于紧急队列中，当正在使用管程的进程执行完后，进行进程切换
- 定时执行 `Cnotify(cn)`，防止出现饥饿问题
- `Cbroadcast(cn)` 将 `cn` 阻塞队列中的**所有进程释放**，转为就绪状态，存于紧急队列中，之后选择哪个进程使用管程由调度算法决定

进程间通信的方法/消息传递

- send
  - 阻塞式发送（blocking send）. 发送方进程会被一直阻塞， 直到消息被接受方进程收到。
  - 非阻塞式发送（nonblocking send）。 发送方进程调用 send() 后， 立即就可以其他操作。
- receive
  - 阻塞式接收（blocking receive） 接收方调用 receive() 后一直阻塞， 直到消息到达可用。
  - 非阻塞式接受（nonblocking receive） 接收方调用 receive() 函数后， 要么得到一个有效的结果， 要么得到一个空值， 即不会被阻塞。
- **在设计进程通信时需要关注的几个问题：**
  - **同步（采用哪种方式同步）**
    - 阻塞发送，阻塞接收
    - 非阻塞发送，阻塞接收
    - 非阻塞发送，非阻塞接收
  - **寻址（采用哪种方式寻址）**
    - 直接寻址：进程《---》进程
    - 间接寻址：进程《---- mailbox --》进程
  - **消息格式（采用哪种消息格式）**
    - 固定格式
    - 变长格式
  - **排队原则**
    - 先进先出
    - 优先级

同步互斥的典型问题

生产者/消费者问题

- 一个或多个生产者用于产生数据，存于缓冲区中。一个消费者从缓冲区中取数据
- ```
  /*无限缓冲区*/
  semaphore n = 0; // 标识当前数据个数
  semaphore s = 1; //信号量，用于互斥操作
  void producer(){
      while(True){
          produce();//produce不需要使用临界资源
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);//n++
      }
  }
  ```
  ```

```
void consumer(){
    while(True){
        semWait(n);//如果n<=0,阻塞当前消费者
        semWait(s);//根据互斥条件阻塞该进程
        take();
        semSignal(s);
        consume();
    }
}
void main(){
    prebegim(produce,consumer)
}
```

- ```
  /*有限缓冲区*/
  semaphore n = 0; // 标识当前数据个数
  semaphore s = 1; //信号量，用于互斥操作
  semaphore e = sizeofbuffer;//标识缓冲区剩余空间大小
  void producer(){
      while(True){
          produce();//produce不需要使用临界资源
          semWait(e);//e--,如果e<0（缓冲区中没有空间），阻塞该进程
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);//n++
      }
  }
  void consumer(){
      while(True){
          semWait(n);//n--,如果n<=0,阻塞当前消费者
          semWait(s);//根据互斥条件阻塞该进程
          take();
          semSignal(s);
          semSignal(e);//e++
          consume();
      }
  }
  void main(){
      prebegim(produce,consumer)
  }
  ```

读者/写者问题

- 多个进程共享数据区
  - 任意多的读进程可以同时读数据
  - 一次只有一个写进程可以写数据
  - 如果写进程正在写数据，禁止读进程读数据
- ```
  /*读进程优先*/
  int readcount = 0;
  semaphore x =1, wsem = 1;  //wsem用于写间互斥与读写互斥，X 用于readcount 互斥
  void reader(){
      while(True){
          semWait(x);
  ```

```
        readcount ++;           //readcount为临界资源，需要互斥
        if (readcount == 1){    //只要有一个读者正在读数据时，就禁止写者写数据
            semWait(wsem);
        }
        semSignal(x);
        readunit();
        semWait(x);
        readcount --;
        if (readcount == 0){
            semSignal(wsem);
        }
        semSignal(x);
    }
}
void writer(){
    while(True){
        semWait(wsem);
        writeunit();
        semSignal(wsem)
    }
}
```