进程管理和进程调度

进程管理

程序的并发性：逻辑上相互独立的程序，宏观上执行时间重叠，微观上相互竞争资源

进程是并发执行的程序在资源分配和资源管理时的基本单位，可以视为程序运行的一个实例

程序和进程的对比：1、程序是静态的、进程是动态的 2、进程需要分配系统资源、程序直选哟存储空间 3、进程具有生命周期、程序是永久的· 4、进程具有并发的概念、程序没有并发的概念 5、一个程序可能属于一个或者多个进程，只要对应的数据集不同即可

PCB：进程控制块，包括：A描述信息、B控制信息、C资源管理信息、D保护现场信息 进程=程序+数据+PCB

A：pid、uid、家族信息、程序存放地址； B：进程当前状态、进程优先级、程序开始地址； C、存储器、文件系统信息、I/O设备信息

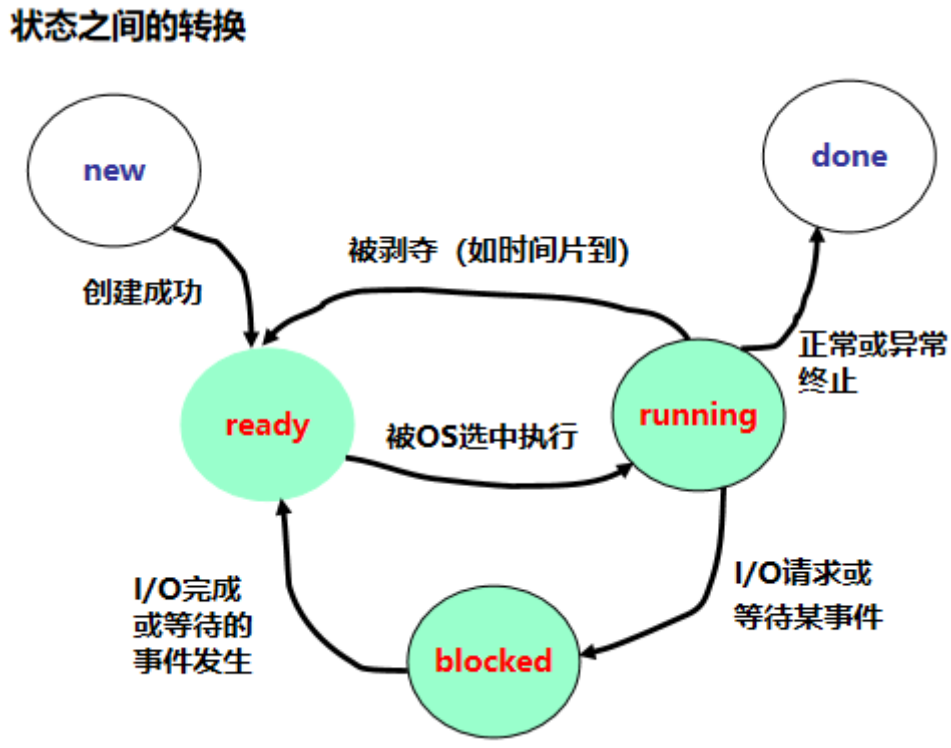
PCB存放在内存特定位置，构建PCB表，表的大小反映了最多存在的进程；相同状态的PCB同属于一个链表。

进程具有五种状态：新建、就绪、运行、阻塞、结束

创建PCB但没有分配资源，处于新建状态->分配资源之后进入就绪状态->经进程调度后获得处理器，进入运行状态->由于等待事件进入阻塞状态->等待事件结束，进入就绪状态->进程内容执行完毕，进入结束状态

进程的创建、撤销、状态转换均通过原语来实现。原语：在内核态执行的具有特定功能的程序段。（两种：1、机器指令级，执行过程中不允许中断 2、功能级，不能并发执行或者保证并发执行的顺序性）六种原语：创建原语、撤销~、阻塞~、唤醒~、挂起~、激活~。另外还有改变进程优先级的原语。原语具有原子性。

新建状态：分配资源但没有进入就绪队列；结束状态：不在运行，但暂时留在系统中

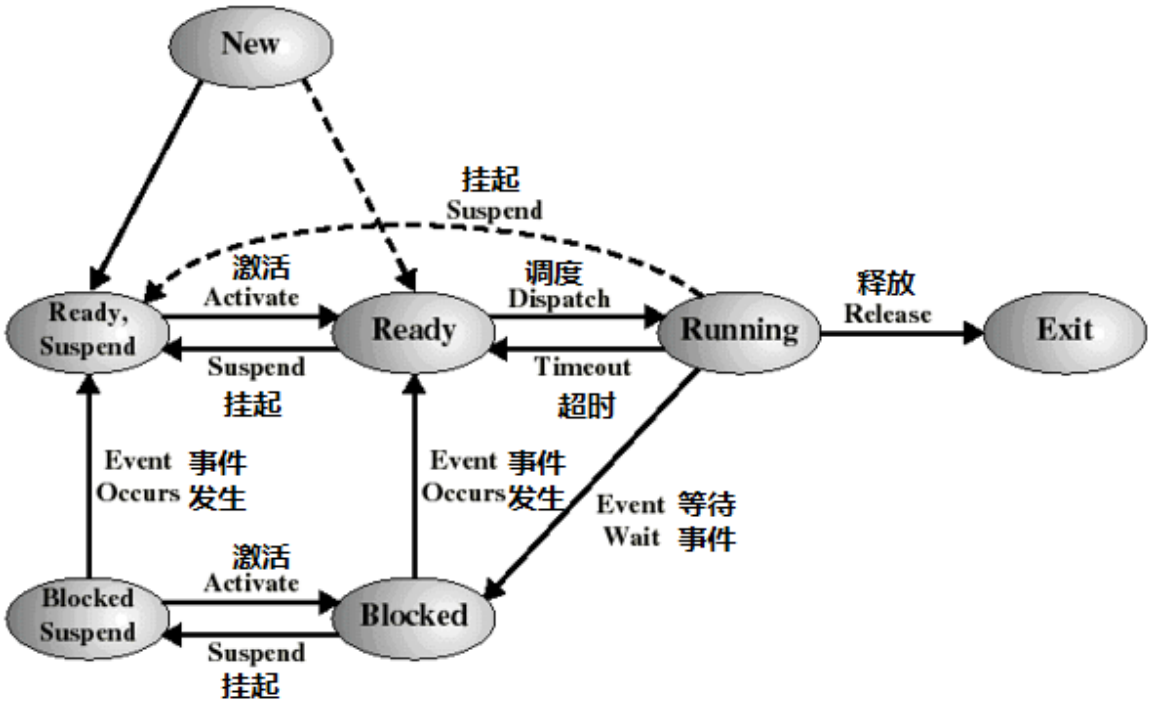


另外还有一种七状态模型：增加了挂起（由内存调入外村）功能。具体来讲，就绪状态分为活动就绪+静止就绪；阻塞状态分为活动阻塞+静止阻塞

- ✓ 阻塞挂起(Blocked, suspend)：进程在外存并等待某事件的出现
- ✓ 就绪挂起(Ready, suspend)：进程在外存，但只要进入内存，即可运行

阻塞→阻塞挂起：没有进程处于就绪状态或就绪进程要求更多内存资源时，可能发生这种转换，以提交新进程或运行就绪进程
就绪→就绪挂起：当有高优先级阻塞（系统认为会很快就绪的）进程和低优先级就绪进程时，系统可能会选择挂起低优先级就绪进程
运行→就绪挂起：对抢占式系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态

就绪挂起→就绪：没有就绪进程或挂起就绪进程优先级高于就绪进程时，可能发生这种转换
阻塞挂起→阻塞：当一个进程释放足够内存时，系统可能会把一个高优先级阻塞挂起（系统认为会很快出现所等待的事件）进程从外存转到内存



进程同步

同步：严格的时序关系，条件准备好才开始执行，否则阻塞；互斥：不同时访问临界资源

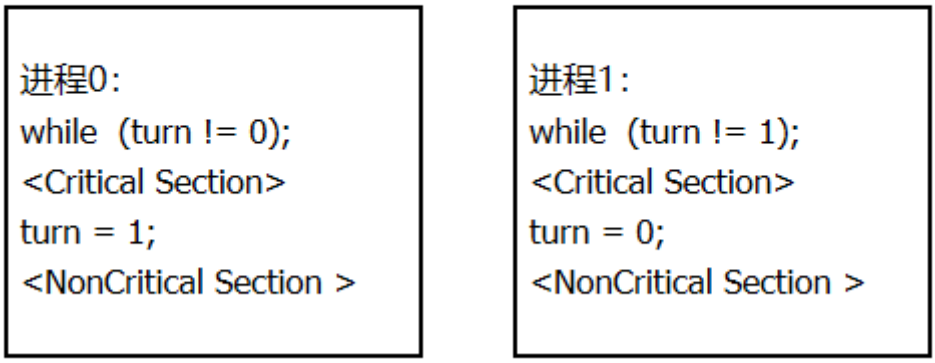
在进程互斥中，一组并发进程需要满足：1、某时刻最多一个进程处于临界区中 2、进程不在临界区中时，不能阻止其他进程进入临界区 3、当一个进程申请进入临界区后，在有限的时间内你能够成功进入临界区

实现互斥的方案：1、锁变量lock，进入临界区前根据变量lock判断是否有其它进程位于临界区中，但是必须保证操作的不可分离性，否则还是可能出现多个进程同时进入临界区的情况。并且也要注意其中的忙等待问题。

```
while  (lock) ;
lock = 1;
<Critical Section>
lock = 0;
<NonCritical Section >
```

2、严格轮转法，共享变量指示可以进入临界区的进程，但是也会出现忙等待现象。另外也会出现多个进程同时进入临界区的现象。

以2个进程为例。turn = { 0: 允许进程0进入临界区，初始值
1: 允许进程1进入临界区



3、peterson方法，进入或者离开临界区前调用其他函数来判断能否进入或者离开。

```
enter_region(process); //process是 进入/离开临界区的进程号
<Critical Region>
leave_region(process);
<Noncritical Region>
```

```
#define  FALSE  0
#define  TRUE   1
#define   N         2 // 进程的个数
int  turn;          // 轮到谁?
int  interested[N];     // 兴趣数组，所有元素初始值均为FALSE
void  enter_region (int  process)  // process为进程号 0 或 1
{
    int  other;  // 另外一个进程的进程号
    other  =  1  -  process;
    interested[process]  =  TRUE; // 表明本进程希望进入临界区
    turn  =  process;             // 设置标志位
    while ( turn == process && interested[other] == TRUE);
}
void  leave_region (int  process)
{
    interested[process]  =  FALSE; // 本进程将离开临界区
                                   }
```

4、关中断，进入临界区前关中断、离开临界区后关中断，相当于禁用了进程切换的功能，但是这个权限太高了，交给用户进程会带来安全性问题；并且对于多处理器系统来说无效。

5、机器指令的方式，本质上跟peterson方法差不多，但是实现方法是通过机器指令来实现的，通过设置lock锁变量，enter/leave函数修改lock变量。会带来两个问题：cpu利用率低+优先级反转（死锁）

信号量=整型变量+队列，对其操作只能通过PV原语进行，P-，若s<0，则阻塞当前进程进入队列；V+，若s<=0，则唤醒队列中的进程。

经典的进程同步问题：

（一）生产者消费者，使用三个信号量：mutex用于解决互斥；empty表示缓冲区为空的个数；full表示缓冲区为满的个数。初始化：mutex=1，full=0，empty=N（缓冲区大小）

```
void producer()//需要特别注意的一个问题是V的次序可变，但P的次序应该尽可能使mutex也就是互斥操作靠近临界区，防止引起死锁问题
{
```

```
    produce_a();//生产一个产品
    P(empty);//占用一个缓冲区
    P(mutex);//进入缓冲区
    load();//放置产品
    V(mutex);//离开缓冲区
    V(full);//增加一个产品
}
void consumer()
{
    P(full);//消费一个产品
    P(mutex);//进入临界区
    consume_a();//消费
    V(mutex);//离开临界区
    V(empty);//空位置+1
}
```

（二）读者写者问题，多个reader，多个writer，多个reader可以同时读，但任何一个writer不能与其他进程同时进入缓冲区。mutex=1，countmutex=1,count=0

```
void writer()
{
    P(mutex);
    write();
    V(mutex);
}
void reader()
{
    P(countmutex);
    if(count==0)//通过count来间接与writer达成互斥
    {
        P(mutex);
    }
    count++;
    V(countmutex);
    read();
    P(countmutex);
    count--;
    if(count==0)
    {
        V(mutex);
    }
    V(countmutex);
}
```

（三）哲学家进餐，需要注意防止出现一种情况是：所有人都拿起了左手边叉子，等待右手边的叉子，进入死锁，比如下面这种情况：

```
mutex[]={1,1,1,1,1};//互斥五个叉子
void eater(int i)
{
    P(mutex[i]);
    P(mutex[i+1 %N]);
    eat();
    V(mutex[i]);
    V(mutex[i+1 %N]);
}
```

（2）一次只允许1个哲学家吃饭，只需要一个信号量用于互斥所有哲学家，吃饭之前执行P，吃完后执行V。

（3）引入一个信号量，用于互斥取左右叉子的操作，也就是说，要取叉子，应当左右手同时取出。

（4）如果不做限制，最多同时4个人吃饭，因此引入一个信号量num=4；在原来的pv操作外添加针对num的PV操作，也就是说最多允许四个人取叉子吃饭。

（5）人为规定取叉子的顺序（先小后大）；或者人为规定哲学家取叉子的顺序：奇数先左后右，偶数相反。

```
#define TRUE  1
#define  N  5                        //哲学家数
#define LEFT  (i-1+N) % N   //哲学家i的左邻居号
#define RIGHT  (i + 1) % N //哲学家i的右邻居号
#define THINKING  0          //哲学家正在思考
#define HUNGRY  1            //哲学家想取得叉子
#define EATING   2            //哲学家正在吃饭
int state[] = {THINKING,THINKING,THINKING,THINKING,THINKING}; //哲学家状态
Semaphore_t  mutex = 1, //临界区互斥
                      s[] = {0, 0, 0, 0, 0}; //表示哲学家是否具备得到叉子吃饭的条件
void philosopher (int i) //i是哲学家编号：0~N-1
{
    while (TRUE)  {
        think();
        take_forks(i); //取左右2把叉子
        eat();
        put_forks(i); //放左右2把叉子
    }
}
void take_forks(int i)
{
    P(mutex); //进入临界区
    state[i] = HUNGRY;
    test(i); //看是否能进餐
    V(mutex); //离开临界区
    P(s[i]); //取得叉子进餐
}
void put_forks(int i)
{
    P(mutex); //进入临界区
    state[i] = THINKING;
    test(LEFT); //唤醒满足条件的左邻居进餐
    test(RIGHT); //唤醒满足条件的右邻居进餐
    V(mutex); //离开临界区
}
void test(int i)
{
    if (state[i] == HUNGRY && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
       state[i] = EATING;
       V(s[i]);
    }
}
```

（四）理发师理发

A、第一类问题，不限制椅子的个数，需要两个信号量：barber=1，customer=0

```
void cuter()
{
    P(customer);
    cut();
    V(barber);
}
void custom()
{
    P(barber);
    V(customer);
    becut();
}
```

B、第二类问题，限制椅子的个数为3，不包括理发师的椅子，增加一个变量num记录正在等待的人数，num_mutex=1;

```
void cuter()
{
    P(customer);
    P(num_mutex);
    if(num==0)
    {
        V(customer)
    }
    num--;
    V(num_mutex);
    cut();
    V(barber);
}
void custom()
{
    P(num_mutex);
    num++;
    if(num>4)
    {
        num--;
        V(num_mutex);
        return;
    }
    V(customer);
    V(num_mutex);
    P(barber);
    becut();
}
```

进程间通信

高级通信（大量数据）；低级通信（控制信息）。比如信号量是属于低级通信原语。

高级通信有4种实现方式：1、主从 2、会话 3、共享存储区 4、消息传递

1、主进程可以使用从进程资源、从进程受到主进程控制、关系固定 2、有点类似于CS模式 3、同一个物理块映射到多个进程的内存空间 4、（缓冲队列/邮箱）可以是一种非实时通信，这是他的一个优势，具体实现的时候我觉得可以当作生产者消费者问题来处理；一个实例是 管 道 ， 但是只能单向通信，要双向通信需要两条管道。

线程

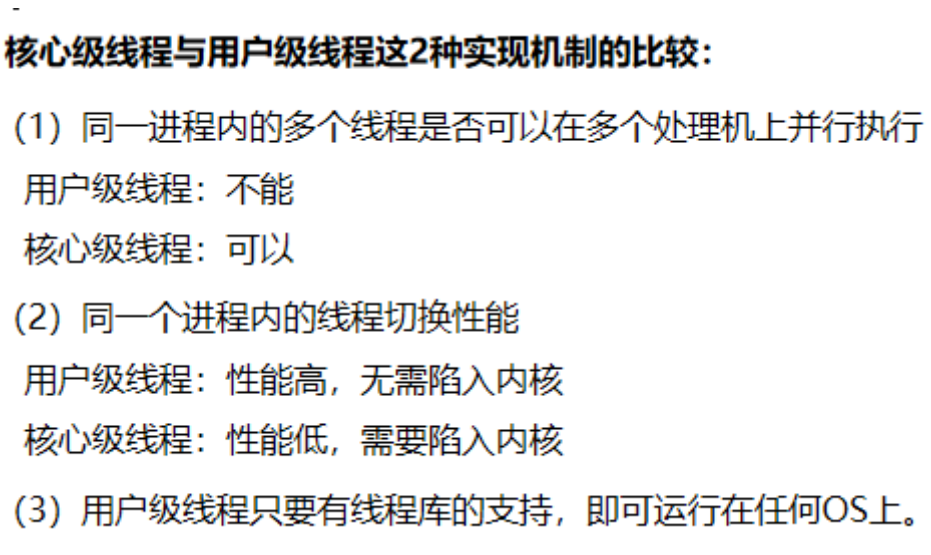
线程引入的原因来自于进程实现多任务的弊端：1、上下文切换系统开销大 2、进程之间共享变量实现复杂

在一个进程中增加多个线程（可以并发执行、共享地址空间），每个线程拥有各自的堆栈，其余数据共享进程的数据，那么相对而言就饿能够有效解决进程实现多任务的弊端。描述线程的数据结构与PCB类似，是TCB，保存堆栈和状态信息。

线程两种：（用户线程、系统线程，分别对应用户态和内核态，也就是用户进程和系统进程）

1、用户级进程：用户程序管理线程（通过系统提供的线程库实现），操作系统管理进程，进程是调度的基本单位，也是资源分配的实体。发生线程调度时不涉及处理器的变化，只改变线程的上下文

2、内核级线程：操作系统管理线程，进程是资源分配的实体，调度的基本单位是线程，



进程调度

从就绪队列中选择一个进程分配CPU：1、借助PCB中进程的状态 2、选择合适的进程 3、进程的上下文切换

衡量进程调度算法的指标：周转时间、平均周转时间、带权周转时间、平均带权周转时间（周转时间=等待+执行）目标：CPU利用率高、吞吐量大、系统开销小、周转时间短、公平性、调度算法简单

进程调度依靠中断来实现，时机：（抢占式、非抢占式）时间片用尽、被高优先级进程抢占、进程执行完毕、I/O请求、等待事件发生、自我阻塞（PV）、

进程调度算法：A、FCFS先来先服务。按照先后顺序获得CPU，非抢占式。优点：实现简单，相对公平，呢能够让先就绪的进程鲜活的CPU ；缺点：1、适合CPU繁忙型，不适合I/O繁忙型，对某些短进程可能不公平，可能会出现短进程等待时间过长的现象。2、平均等待时间波动较大 3、有利于长进程不利于短进程

B、在就绪队列中挑选预计执行时间最短的进程占用CPU，因此需要用历史执行时间来估计预期执行时间。优点：CPU吞吐量大，最优平均周转时间 缺点：短作业有利，长作业不利，极端情况下出现长作业得不到CPU的情况

C、最高相应比优先：R=（等待事件+预计执行时间）/预计执行时间，相当于综合上面这两种方案，既考虑了等待时间，有考虑了执行时间，兼顾了CPU吞吐量和CPU利用率。但是缺点是每次都需要重新计算R，带来较大的系统开销。

D、时间片轮转：进程先后成队列，以此获得时间片，是一种抢占式，所以不适用非抢占式的资源。队首分配时间片，用完后回到队尾。因此涉及到时间片大小选取的问题，过大或者过小都有问题，通常：系统要求的响应时间/当前就绪队列中的进程数=时间片大小。不适合I/O密集型

优化为虚拟时间片轮转，建立两个就绪队列，优先级高低之分。依据时间片是否用完被中断来将进程放置在不同队列中，先处理高优先级队列。

E、基于优先级判断（静态/动态）

F、多级反馈队列。建立多个队列，具有不同的优先级，优先处理高优先级队列。时间片用尽->下一个低优先级的队列中；被抢占->原队列队尾；最后一个队列是时间片轮转。同时考虑优先级和公平性。

死锁：一组并发进程，每个进程都占有一部分资源，但仍需要一部分资源才能向前推进，但他们需要的资源被该组中其他进程占有，导致该组各个进程都在等待其他进程释放资源，无法向前推进，即对资源的永久占有和永久保持。出现的原因就是：所需要的资源被其他进程所占有/系统提供的资源有限

死锁四个必要条件：1、互斥；2、部分分配；3、不可剥夺；4、环路

对于死锁，对待的态度是：检测+恢复/预防/视而不见

检测：A、每种资源只有一个，绘制资源分配图，是否构成环路 B、每种资源有多个，基于矩阵的检测算法。

```
设有M种资源，N个进程
1）数据结构
E[M]：总资源数；E[i]：资源i的个数
A[M]：当前可用资源数；A[i]：资源i的可用数
C[N][M]：当前分配矩阵；C[i][j]：进程i对资源j的占有数
         第i行是进程i当前占有的资源数
R[N][M]：申请矩阵；R[i][j]：进程i对资源j的申请数
         第i行是进程i申请的资源数
F[N]：进程标记；F[i]取0/1，为1表示进程i能够执行
已分配资源数 + 可用资源数 = 总资源数
① 置F[0]~F[N]为0;
② 寻找一个满足下列条件的进程i：
    F[i] == 0 且
    R[i] <= A，即R[i][j] <= a[j], 0<=j<=M  //进程i申请的资源可满足
③ 若找不到这样的进程，则算法终止;
④  A = A + C[i];
      F[i] = 1;
   转②继续;
算法结束后，所有F[i]==0的进程(i)都会死锁。
```

从死锁中恢复：1、剥夺某个进程（消耗处理机时间最少、产出最少、估计剩余时间最长）占有的资源；2、回退到某个检查点；3、关闭进程

避免：（银行家算法）基于是否存在安全序列来判断，存在安全序列->不会进入死锁；不存在安全序列->**可能**进入死锁；安全序列不唯一。

> **银行家算法中所用的主要数据结构：(1)可用资源向量Available 记录资源的当前可用数量。如果Available[j]==k，表示现有Rj类资源k个。 (2) 最大需求矩阵Max 每个进程对各类资源的最大需求量 (3)分配矩阵Allocation 已分配给每个进程的各类资源的数量 (4)需求矩阵Need 进程对各类资源尚需要的数量 Need = Max - Allocation (5)请求向量Request 记录某个进程当前对各类资源的申请量**

当Pi发出资源请求后，系统按下列步骤进行检查：
(1)如果Request > Need[i]，则出错；
(2)如果Request > Available，则Pi阻塞；
(3)假设把要求的资源分配给进程Pi，则有

```
Available = Available - Request;
Allocation[i] = Allocation[i] + Request;
Need[i] = Need[i] - Request;
```

检查此次资源分配后，系统是否处于安全状态。若安全，正式将资源分配给进程Pi，以完成本次分配；否则，恢复原来的资源分配状态，让进程Pi等待。

检查安全状态所使用的方法与之前的检测方法一致。但是这种算法开销大、且需要提前知道对资源的最大需求。