

# Operating System - P25

管程 (Java) Java 是面向对象的语言，支持用户级线程，允许将方法（过程）划分为类，是一种真正支持管程的编程语言。synchronized 关键字加入方法声明时，Java 保证在某个线程执行该方法时，不允许其他线程执行该方法。

```
public class ProducerConsumer { 外部类 (outer class) 创建并启动两个线程 producer/consumer
    static final int N = 100;
    static producer p = new producer();
    static consumer c = new consumer();
    static our-monitor mon = new our-monitor(); 管程 our-monitor，带有两个同步 (synchronized) 线程
    public static void main (String args[]) {
        p.start();
        c.start();
    }
}
```

Static class Producer extends Thread { }

```
public void run() { 线程代码
    int item;
    while (true) {
        item = produce-item();
        mon.insert(item);
    }
}
```

private int produce-item() { ... }

生产者 producer 代码

Static class consumer extends Thread { }

```
public void run() { 线程代码
    int item;
    while (true) {
        item = mon.remove();
        consume-item(item);
    }
}
```

消费者 consumer 代码

```
private void consume-item(int item) { ... }
```

消费者 consumer 代码

```
private void go-to-sleep() { try { wait(); } catch (InterruptedException exc) { ... } }
```

类 our-monitor 包含缓冲区，管理变量，两个同步方法

```
private int count = 0, lo = 0, hi = 0;
```

```
public synchronized void insert(int val) {
```

```
    if (count == N) go-to-sleep();
    buffer[hi] = val;
    hi = (hi + 1) % N;
    count = count + 1;
    if (count == 1) notify();
}
```

```
public synchronized void remove() {
```

```
    if (count == 0) go-to-sleep();
    val = buffer[lo];
    lo = (lo + 1) % N;
    count = count - 1;
    if (count == N - 1) notify();
    return val;
}
```

```
private void go-to-sleep() { try { wait(); } catch (InterruptedException exc) { ... } }
```

类 our-monitor 包含缓冲区，管理变量，两个同步方法

```
hi / N-1
```

可以将

看作一个环形

结构

hi / N-1

# Operating System - P26

注意管程是一个编程语言概念，要求编译器必须要识别管程并用某种方式对互斥作出安排。

管程和信号量无法解决分布式系统上拥有私有内存的CPU以局域网相连的情形。

消息传递 (message passing) 方法是在进程间通信中使用两条原语，类似于信号量，是系统调用而非语言成分。

send(destination, &message); 向一个给定的目标发送一条消息。

receive(source, &message); 从一个给定的源(或任意)接收一条消息。

如果没有可用消息，接收者可能被阻塞直到有消息，也可能带着错误码返回。

确认 (acknowledgement)，为了防止消息丢失，接收者在接收到信息时立即回送一条特殊信息。如果发送方在一段时间内未收到回送，则重发消息。

当消息正确接收，而确认丢失时，通过在原始消息嵌入一个连续序号来区分新消息和老消息。

注意：在 send 和 receive 调用中指定的进程必须是没有二义性的。

身份认证 (authentication) 用于客户端确认通信对象是真正的文件服务器而非冒充者。

```
#define N 100 // 假设所有消息有相同大小，在尚未接收到派出消息时，由OS自动缓冲，类似于共享内存缓冲区
void consumer(void) {
    int item, i; message m; // 向生产者发送N条空消息
    for(i=0; i<N; i++) send(producer, &m); // (N个空缓冲槽)
    while(TRUE) { // 接收包含数据的消息
        receive(producer, &m);
        item = extract_item(&m); // 提取数据
        send(producer, &m); // 发送空消息
        consumer_item(item); // 处理数据项
    }
}

void producer(void) {
    int item, i; message m;
    while(TRUE) { // 产生放入缓冲区的数据
        item = producer_item();
        build_message(&m, item); // 生成将发送的消息
        send(consumer, &m); // 发送数据给消费者
    }
}
```

消息地址 方案一：为每个进程分配一个唯一的地址，让消息按进程的地址编号。

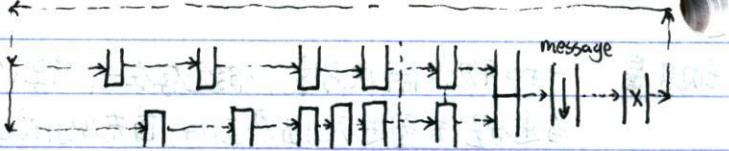
信箱 (mailbox)，用来对一定数量的消息进行缓冲，而 send 和 receive 调用的是信箱地址而非进程地址。

会合 (rendezvous)：彻底取消缓冲，执行 send / receive 将被阻塞，直到有一个 receive / send 发生。

消息传递接口 (Message-Passing Interface, MPI) 广泛应用于科学计算的消息传递系统。

## Operating System - P27

考虑会话的模式，可看作两条相互 send  
对齐的时间轴 receive



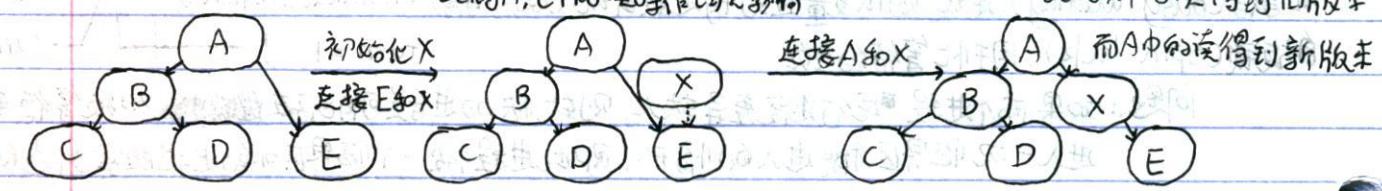
屏障 (barrier) 用实现在线程组中，除非所有进程都就绪进入下一阶段，否则任何进程都不能进入下一阶段。如同一个进程到达屏障时即被阻拦，直到所有进程到达屏障，并一起解除阻拦。典型问题如物理或工程中的驰豫问题，即不断迭代计算矩阵。

避免锁      最快的锁是根本没有锁

读-复制-更新(Read-Copy-Update, RCU), 将更新过程中的移除和再分配过程分离, 一集多用

确保每个读操作要么读取旧的数据版本，要么读取新的版本，而非两者组合

此时 A, E 中的读操作均无影响 此时 X 中的读得到旧版本



使A连接到C  
分离A和B  
移除B和D

**宽限期** (grace period), 读者通过读端临界区访问数据, 临界区中不允许阻塞或休眠, 则期间每个进程至少有一次在简单的方案是等到所有线程执行完一次上下文切换, 则宽限期结束并回收

调度程序 (Scheduler), 在OS中当有多个进程或线程竞争CPU时,选择下一个运行的进程或线程  
使用调度算法 (Scheduling algorithm), 目标是在CPU是稀缺资源的情况下, 提高性能

CPU利用率 由于进程切换的代价较高,所以调度必须考虑

用户态必须切换到内核态，保存当前进程的状态和存储寄存器值以便重新装载

在许多系统中，内存映像（页表内的内存访问位）也必须保存

选定新进程后，将其内存映像重新装入MMU，用同样的方法处理其他进程。

另外，切换会使内存高速缓存失效，强迫高速缓存从内存中动态重装两次（进入退出内核）

# Operating System - P28

## 进程行为

几乎所有进程的(磁盘或网络)I/O请求和计算都是交替突发的

典型观点，认为使用了CPU即计算，而当等待外部设备完成工作而被阻塞时，才是I/O活动

## 计算密集型

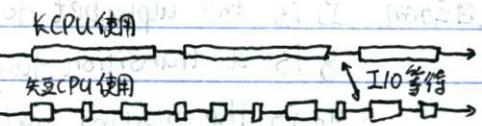
(compute-bound) 较长时间CPU集中使用，较少频度I/O等待

## I/O密集型

(I/O-bound) 较短时间CPU集中使用，频繁I/O等待

注意I/O型并不是指I/O请求时间长，而是指I/O请求的频率高

随着CPU越来越快，进程更倾向于I/O密集型，主要因为CPU改进速度快于硬盘



## 何时调度

创建一个新进程后，决定运行父进程或运行子进程

进程退出时，从就绪的进程中选择一个，如果没有就绪，通常运行系统提供的空闲进程

进程阻塞时(如I/O或信号量)，从就绪的进程中选择一个，阻塞原因也可能成为选择因素

在发生I/O中断时，某些等待该I/O的进程即就绪，决定运行新就绪进程，当前进程或其他进程

## 非抢占式算法

挑选一个进程，运行直至被阻塞(阻塞在I/O或等待另一进程)，或者直到该进程自动释放CPU

## 抢占式算法

挑选一个进程，运行某个固定时段的最大值(基于硬件时钟提供的中断)，时段结束后自己

## 分类

批处理：非抢占式算法，或长时间周期的抢占式算法，因为无需快速响应短请求

交互式：抢占是必须的，避免进程霸占CPU或程序错误导致无限期排斥其他进程

实时：抢占不必要，因为进程了解可能长时间得不到运行，所以通常快速完成工作并阻塞

## 目标

公平：给每个进程公平的CPU份额，或者说相似的进程应得到相似的服务

## 通用

策略强制执行：调度程序必须保证规定的策略被执行

平衡：保持系统的所有部分尽可能忙碌，调度程序通过仔细检查进程，可以保持系统运行得更好

吞吐量(throughput)指系统每小时完成的作业数量。注意吞吐量最大化的算法不一定有最小周转时间

## 批处理

周转时间(turnaround time)指从提交作业时刻到完成时刻的统计平均时间，规则通常是小就是好的

CPU利用率，常用于对批处理系统的度量，但并不是一个好的参数，因为未必需要保持CPU始终忙碌

## 交互式

响应时间，从发出命令到得到响应的时间，能有多让所有的交互式请求首先运行的原则是好服务

## 系统

均衡：满足用户的期望，因为用户对做一件事情需要多长时间总有一种固有的(通常不正确的)看法

满足截止时间，避免数据丢失，最主要的要求是满足所有的(或大多数)截止时间要求

## 实时系统

由于实时系统的特征是或多或少必须满足截止时间

可预测性：在多媒体系统中避免品质降低，要求调度程度必须是高度可预测和有规律的

# Operating

## System - P29

批处理系统：

先来先服务 (first-come first-served)，非抢占式算法允许进程按照请求CPU的顺序使用CPU。  
优点是易于理解且便于在程序中运用，以一个单链表 (single linked list) 记录所有就绪进程。  
选取一个进程运行即从队列头部移除一个进程即可。  
添加新的作业或阻塞进程就将新附加在队列的末尾即可。  
缺点是按顺序执行且非抢占式难以达到平衡，即系统所有部分都保持忙碌。  
如一个运行 1s 的计算密集型进程和一个需要 1000s 读磁盘操作的 I/O 密集型进程。  
非抢占式导致每次 I/O 进程都要等待计算进程完成，则需花费 1000s 才能完成。  
方案是改为抢占式算法；若每 10ms 抢占计算进程，则 I/O 进程可在 1s 完成。

最短作业优先 (shortest job first)，适用于运行时间可以预知的非抢占式批处理调度算法。

优点是对于周转时间而言，可以证明最短作业优先是最优的。

缺点是只有在所有的作业都可同时运行的情况下（或到达时间一样），最短作业优先才是最优的。

如运行时间 2, 4, 1, 1，平均则

到达时间 0, 0, 3, 3.3



平均 4.45

最短剩余时间优先 (shortest remaining time next)，最短作业优先的抢占式版本，总是选择剩余运行时间最短的进程。

注意：运行时间必须提前掌握，新作业到达时，其整体时间同当前进程的剩余时间比较。

如果新进程需要时间更长，则挂起当前进程并运行新进程。

缺点是只有短作业获得良好的服务，即长时间进程面临周转时间过长的问题。

交互式系统：

轮转调度 (round robin) 最古老，最简单，最公平且使用最广的算法。（或进程阻塞或切换）

每个进程分配一个时间片 (quantum)，允许进程在时间段中运行，时间片结束则 CPU 切换。

优点是很容易实现，调度程序仅需维护一张可运行进程列表，进程用完时间后移到队列末尾。

如

注意时间片的长度相对于操作系统进行进程切换 (process switch) 或上下文切换 (context switch) 的时间。

如果时间片过短，会导致进程频繁切换，降低 CPU 效率。

如管理任务 (保存和装入寄存器值及内存映像，更新表格与列表，清除和重新调入内存高速缓存)

管理任务 1ms，时间片 4ms，则 CPU 要浪费 20% 的时间。

如果时间片过长，引起短的交互式请求的响应时间过长。

如短时间内到达的多个请求中，运行时间较短的进程排在就绪队列末尾。

如 50 个请求，时间片 500ms 的时间片，而同时到达 50 个请求，则队列末尾的进程响应时间会长达数秒。

# Operating System - P30

优先级调度 基本思想是每个进程被赋予一个优先级，并允许优先级最高的可运行进程先运行

与之相比，轮转调度做了一个隐含的假设，即所有的进程同等重要

注意：调度程序可能在每个时钟中断降低当前进程的优先级，以防止高优先级进程无休止运行  
优先级可以静态赋予。另外UNIX系统中存在命令nice，允许用户自愿地降低自己进程的优先级  
优先级也可以动态赋予，如I/O密集型的进程，其大部分时间等待I/O结束。

则当其需要CPU时可立即分配，以便启动下一个I/O进程

方案是将优先级设为1/4，于该进程在上一个时间片中所占部分。

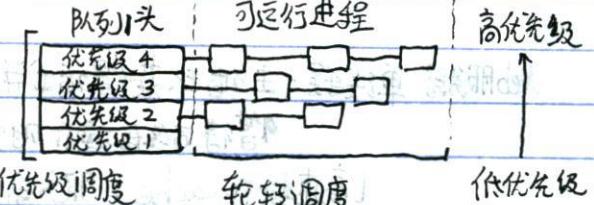
如50ms的时间片，占用31ms的进程优先级为50，占用25ms的进程优先级为2

注意在这个方案中，I/O进程由于等待I/O完成而阻塞，所以会始终占有高优先级直至就绪  
或者可以将进程分为若干类。

在各类间采用优先级，在类内部采用轮转调度

只要上一优先级的类有就绪进程，则不考虑下一优先级

注意：如果不调整优先级，则低优先级会产生饥饿。



优先级调度

轮转调度

低优先级

= 多级队列 (Compatible Time Sharing System, CTSS). MIT开发的最早使用优先级调度的兼容分时系统

问题是IBM 7094内存只能放进一个进程，导致进程切换速度太慢

方案是为CPU密集型进程设置较长时间片以获得比频繁分配短时间片更高效率

问题是长时间片会影响进程的响应时间

方案是设立优先级类，最高优先级1个时间片，次高2个时间片，用完时间片的进程移往下一优先类

如需100个时间片的进程共需装入7次： $1+2+4+8+16+32+37$ ，相比轮转需装入100次

最短进程优先 交互式进程通常遵循：等待命令，执行命令，等待命令，执行命令，如此循环

由于最短作业优先常常伴随着最短响应时间，考虑如何从可运行进程中找到最短的

方案是根据进程过去的行为进行推断，用已有估计和最近运行的加权值来改进估计时间

如 $aT_0 + (1-a)T_1$ ，通过选择 $a$ 决定是快速忘记旧的运行时间，还是长时间记住

$$T_0 \rightarrow \frac{1}{2}T_0 + \frac{1}{2}T_1 \rightarrow \frac{1}{4}T_0 + \frac{1}{4}T_1 + \frac{1}{2}T_2 \rightarrow \frac{1}{8}T_0 + \frac{1}{8}T_1 + \frac{1}{4}T_2 + \frac{1}{2}T_3$$

老化 (aging) 通过当前测量值和先前估计值进行加权平均而得到下一个估计值的技术

适用于预测值必须基于先前值的情况，即  $f_{n+1} = F(f_n, t_{n+1})$

保证调度 先向用户作出响石角的性能保证，然后实现保证，如n个用户保证各获得1/n

方案是系统必须跟踪各个进程自创建以来已使用的CPU时间，

并计算出真正使用CPU时间和应获得时间之比，

调度程序总是转向比率最低的进程，直到该进程比率超过其前一个竞争者

# Operating System - P31

彩票调度 (lottery scheduling), 相比保证调度，既可给出类似预测结果而又有简单的实现方法的算法

基本思想是为进程提供各种系统资源 (如CPU时间) 的彩票

当需要调度决策时，随机抽出一张彩票，拥有该彩票的进程获得资源

“所有进程是平等的，但是某些进程更平等一些” (George Orwell)

拥有子份额彩票的进程 在较长的运行中大约得到子份额的系统资源

规则是清楚，也容易量化优先级的分类

另外彩票调度是反应迅速的，因为新进程获得彩票份额后，下一次调度即有等同机会获得资源

进程之间如果有协作，可以交换所拥有的彩票。

如客户端向服务器发送信息并阻塞后，可将彩票交给服务器，以增加服务器份额

注意彩票调度适用于需要对时间量化分配且份额可随时间变动的问题

如对帧数 10, 20, 25 的视频流进程分配份额 10, 20, 25 的彩票

公平分享调度 如果调度算法只关注进程自身而不关注所有者，则在轮转或相同优先级调度下，拥有进程多的用户占用更多

方案是在调度处理之前考虑谁占有进程这一因素

实际上是对系统资源分配的策略从进程级上移到用户级

如用户 A 运行  $A_1, \dots, A_{10}$  共 10 个进程 轮转 ( $A:B=10:1$ ) :  $A_1 | A_2 | \dots | A_{10} | B_1 | A_1 | \dots$

而用户 B 运行  $B_1, \dots, B_6$  共 6 个进程 保证 ( $A:B=1:1$ ) :  $A_1 | B_1 | \dots | B_6 | A_1 | B_1 | \dots$

则考虑一般轮转和不同比例的保证 保证 ( $A:B=2:1$ ) :  $A_1 | A_2 | \dots | A_7 | A_8 | B_1 | \dots$

实时系统 是时间起主导的系统，注意正确但迟到的应答往往比没有应答更糟糕

典型如外部物理设备向计算机发送请求，而计算机必须在一个确定的时间范围内做出恰当反应

硬实时 (hard real time) 指系统必须满足绝对的截止时间

软实时 (soft real time) 指虽然不希望系统偶尔错过时间，但可以容忍

实时性是通过把程序划分为一行为可预测和提前掌握的一组进程实现的

进程一般寿命较短，会以极快速度运行完成，如进入：等待中断、阻塞、睡眠等状态

在检测到一个外部信号时，调度程序按照满足所有截止时间的要求调度进程

按响应方式分类为周期性 (以规则的时间间隔发生) 或非周期性 (发生时间不可预知)

可调度的 指实际上能调度被实现的，如对 m 个周期事件，事件 i 以周期  $P_i$  发生，并需要 CPU 时间  $C_i$  处理

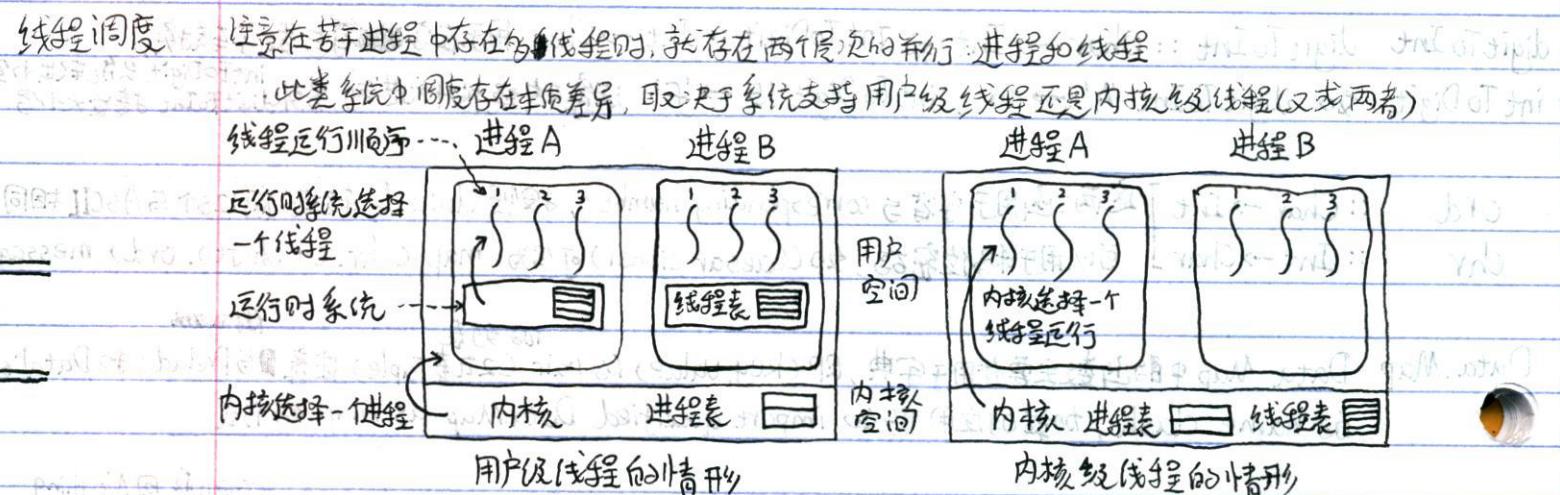
则 可调度  $\leftrightarrow \sum C_i / P_i \leq 1$  注意隐含假设是进程切换开销忽略，另外非周期事件可用统计频率

调度算法可以是静态的 (系统开始前作出决策) 或动态的 (运行过程中进行决策)

注意静态算法必须提前掌握所有信息，如所需完成的工作和必须满足的截止时间

# Operating System - P32

**策略与机制** 当一个进程有许多子进程在其控制下运行，主进程可能掌握子进程的优先级顺序。而如果调度算法无法从用户进程接收调度决策信息，则很少能做出最优选择。方法是调度机制(scheduling mechanism)与调度策略(scheduling policy)分离的原则。调度机制位于内核，调度策略则由用户进程决定。如内核使用优先级调度算法，然后提供用户进程一条设置/改变优先级的系统调用或使用彩票调度算法，然后提供用户主进程一条转移子进程彩票所有权的系统调用。



**注意：** 用户级线程可能调度为  $A_1, A_2, B_1, B_2, B_3, A_2, A_3, \dots$ ，  
内核级线程可能调度为  $A_1, B_1, A_1, B_2, A_2, B_3, A_1, A_3, \dots$ ，且互相不可能出现对方的调度模式。

注意其差异在于时间片的分配，用户级线程系统分配予进程，而内核级线程系统分配予线程。  
另外时间片用完时，用户级线程由内核选择下一个进程，必定产生上下文切换。  
而在线程运行完成时，用户级线程由运行时系统在进程内选择运行一个可用线程。

注意在用户级线程中，可用的调度算法与内核可用的进程调度算法是一样的。  
其中轮转调度和优先级调度更为常用。  
但要注意，运行时系统缺乏时钟中断来避免线程运行时间过长。  
但由于线程间为协作关系而非进程间的竞争关系，所以影响不大。

**差别之一在于性能：** 用户级线程的线程通常仅需少量的机器指令，而内核级线程需要完整的上下文切换，修改内存映像，重新装入高速缓存。  
但内核级可以选择优先运行同一进程内的就绪线程。  
**差别之二在于定制：** 用户级线程可以使用专为程序定制的线程调度程序。

一般而言，定制的线程调度程序比内核更好地满足程序需求。