4.1 处理器调度原理和设计

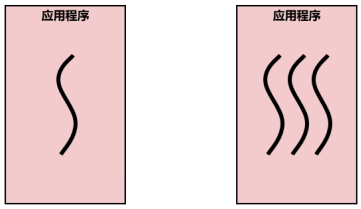
一.指令流与执行

1.1应用程序与指令

指令流

一个应用程序内部可以由一个或多个**逻辑上互相独立执行的指令序列**组成。这种独立执行的指令序列叫做**指令流**。CPU靠执行指令流来完成应用程序的功能。

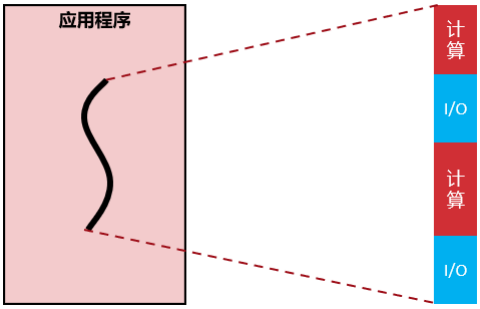
问题 为何在一个应用程序里面需要设置多个指令流?



计算期与I/O期

一个**指令流**往往由**两部分组成**：一部分负责**计算**，另一部分负责**I/O**。两个部分交相点缀，组成一整条指令流。指令流是程序逻辑的组成单位。

在计算机指令流中，**计算期**通常是指处理器执行指令所需的时间，而**I/O期**则是指处理器从输入输出设备读取或写入数据所需的时间



问题 某个指令流可以**没有计算期**（Burst）吗？可以**没有I/O期**吗?

为什么？在一些特殊情况下，可能会不存在其中之一或两者都不存在

指令流中可能不存在计算期的情况通常发生在一些特殊的指令或者操作上，例如NOP指令（No Operation），该指令本身不进行任何计算操作，只是为了消耗一些处理器时间，或者是一些无条件跳转指令，该指令不需要进行计算操作，只需要跳转到目标地址即可。

指令流中可能不存在I/O期的情况通常发生在没有任何输入输出操作的情况下，例如纯计算型任务或者是一些计算机内部的操作，例如缓存操作、分支预测等等，这些操作都不需要从输入输出设备读取或写入数据，因此不需要进行I/O操作。

指令流的分工

多个指令流可以分工合作，完成程序的功能。

按性质分工

不同的指令流处理不同性质的工作，如一些指令流主要负责I/O， 另一些指令流主要负责计算等等。

按对象分工

不同的指令流处理不同部分的工作，如每个指令流负责处理一 部分数据或一个服务对象。

1.2顺序与并发执行

指令流间的执行顺序——顺序执行

一个指令流执行完成后，再去执行另一个指令流。

优势 安排工作简单，一件事情做完了再去做另一件事情。

劣势 如果一件事情没做完之前，需要暂时放下去做另一件事情的话， 不可能办得到。

问题 什么场合可能出现一件事情没做完会被打断?

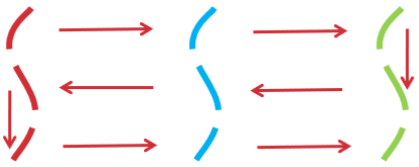
需要打断工作的常见情况

1. **优先级** 某些任务就是比其它任务更加紧急。如果另一个紧急的任务需 要应用程序立马去处理，当前的任务就必须被打断。
2. **进行I/O** 相对于CPU，I/O设备的速度是很慢的。一旦开始I/O，当前的指 令流就不占用CPU了。即便让它继续运行，它能做的也只是反复 查询I/O设备的状态寄存器来判断I/O是否完成。
3. **资源超限** 这个指令流占用了太多CPU时间了，可能达到了某个限额，再放任它继续执行可能造成公平问题，尤其是跨应用程序时：某个 应用程序的指令流将一直占用CPU，其它任何应用程序的指令流 都必须等它结束执行才能得到CPU。
4. **恶意程序** 某个指令流的唯一工作就是空转（这可能是因为它本身是恶意 程序，或被恶意程序入侵，或出BUG），燃烧CPU时间，并且永 不退出，要等到它退出除非机器烧毁或者停电。
5. **相互通信** 某些指令流可能依赖其它指令流传递给它的数据才能工作。如 果它无法从另一个指令流那里得到数据，就要一直等待下去， 直到其它指令流返还数据。这在顺序执行模型下是不可能的。

改进的顺序执行

合作执行

将每个指令流**打断成多份**，**每一份之内都顺序执行**，但背靠背 执行的两份**不一定来自同一个指令流。**在每份指令流的末尾， 都**通知操作系统主动放弃CPU**，CPU将转去**执行下一份指令流。**



合作执行的**特点**:

（1）**交替执行** 指令流在自己执行的途中就可以出让CPU的控制权。

（2）**自愿放弃** 指令流在不需要CPU的时候可以自愿放弃CPU。自愿放弃CPU以 及执行完毕是交还CPU控制权的唯二方法。

（3）**顺序确定** 如果希望，每个指令流都可以在放弃CPU时指明自己希望哪个指令流继承CPU的使用权。当然，如果不指定，那么随机选择一个指令流来运行。

问题 合作执行解决了上述常见问题的哪些?

（1）**优先级** 解决了一部分，现在紧急的指令流可以在份与份之间插入了。 但是没办法在份与份之间插入。

（2）**进行I/O** 仅考虑CPU效率，完全解决，指令流只要进入I/O就放弃CPU。

（3）**资源超限** 应用程序的某一段一直不放弃CPU的话，还是拿它没办法。

（4）**恶意程序** 同上，挖矿程序肯定不会放弃CPU，不然它还挖什么矿。

（5）**互相通信** 完全解决，指令流只要在等待其它指令流的回复就放弃CPU。

顺序执行和合作执行的问题

（1）**缺乏强制性** 无法强制剥夺指令流的CPU控制权。如果指令流是恶意的，或者 自私的，就没办法对它加以约束了。因此，我们需要一种能够 强制剥夺指令流对CPU控制权的方法。

（2）**并发执行** 将合作执行的条件放宽一点，允许一个指令流在任何时候被打 断，并且新的指令流插入进来。又叫**抢占式执行**。每个指令流 都在自己的虚拟CPU上执行，而且虚拟CPU的先后没法预测。

（3）**细粒度** 每个指令流的每条指令之间都可能被打断。

（4）**不可预测性** 在任何一个指令流被打断后，该指令流无法预测下一个接替它 执行的是哪个指令流。

1.3指令流的描述

指令流的描述

在并发执行中，指令流可能**随时被打断**。被打断的指令流的状态信息不能丢失。

这些**状态信息**包括两部分，

一部分是**指令流自己的上下文**，另一部分则是**该指令流对CPU的占用状态**。

上下文 指令流的上下文包括什么？为什么?

提示：考虑原理类似的中断上下文。

寄存器组 一个指令流的上下文就是足够使其恢复被打断时的状态的内容。 这一般**包括了其寄存器组和执行栈**。在切换指令流时，一个非常方便的做法就是把寄存器组保存在执行栈上，合适时再恢复。

状态

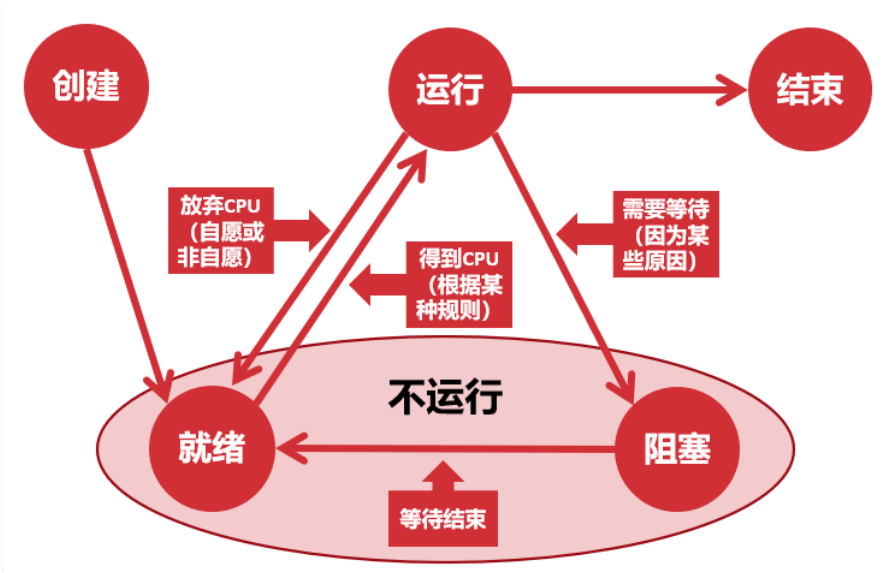
一个指令流可以按照它**对CPU的占用状态**分成几类?

（1）**运行** 当前指令流正在运行。

（2）**不运行** 当前指令流不在运行。再细分，不运行可能是因为：

<1>**就绪** 当前指令流可以运行了，但CPU因为某些原因还没轮到 它，轮到它就可以立马运行。

<2> **阻塞** 当前指令流在等待，就算有空闲CPU也没法运行



二.线程与处理器

2.1CPU时间和线程

处理器时间的分配

指令流与时间

操作系统需要给指令流分配CPU时间，然后让指令流拿着这个 CPU时间配额去运行。但是，指令流是用户程序的逻辑组成部分， 操作系统并不知道用户程序里面有几条指令流。

线程

操作系统提供给应用程序的一种对CPU时间的抽象机制。它是CPU**时间分配**的基本对象。应用程序通过**将自己的指令流与线程对应起来，使指令流获得CPU时间分配。**操作系统通过运行线程， 来运行依附在这个线程上的指令流。

也可以说，**指令流通过依附于线程，获得了在CPU上运行的权利。**

应用程序视角 在应用程序看来，自己的逻辑组织是**一系列并发执行的指令流。**

操作系统视角 在操作系统看来，应用程序的运行组织是**一系列被分配了CPU时 间的线程。**

问题 相比于指令流，线程的描述要包括什么数据呢?

2.2线程的描述

线程的描述

时间预算

线程是时间分配的基本对象，那线程必然有**一个参数描述它被分配了多少间。**这个数值称为时间预算，可以是一个有限的数值，也可以是一个无限的数值。

操作系统在运行线程时会时刻关注线程的时间预算是否耗尽； 如果耗尽，操作系统就切换到其它有时间预算的线程去执行。

优先级

系统中有多个线程同时具备非零的时间预算，如何决定运行哪 一个？因此，线程必须具备一个参数来描述其优先度。当系统遇到多个可以运行的线程时，系统可以决定运行其中优先级最高的那个。

优先级一般用一个数值来代表，在绝大多数系统中，**数字越小， 优先级越高。**

CPU抢占关系实际上是一个偏序集（越紧急的东西，不一定总是越重要；即便是紧急的东西，也不代表它可以不受限制地发生） ，优先级实际上是这个偏序集的一种简化全序描述。不过，这种描述在很多时候（桌面计算等场合）都够用了。

问题 这些数据放在用户程序里面还是操作系统里面?

线程的上下文

上下文 线程为什么也要有**上下文**?

考虑线程上的执行流因为主动等待 需要操作系统介入的I/O完成或者意外地被外设中断打断而暂停运行的场合。

内核阻塞

操作系统并不知道指令流的存在。因此，在遇到线程上的指令流陷入内核阻塞的时候，内核只能**暂停执行当前这个线程**，切换到别的线程去执行了。更麻烦的是，**线程什么时候陷入内核， 依附在线程上的指令流是不知道的**，也即可能发生抢占。

一旦一个线程阻塞在内核，对它上面依附的所有指令流来说， 时间就都凝固了。因此，这些指令流都停止运行。

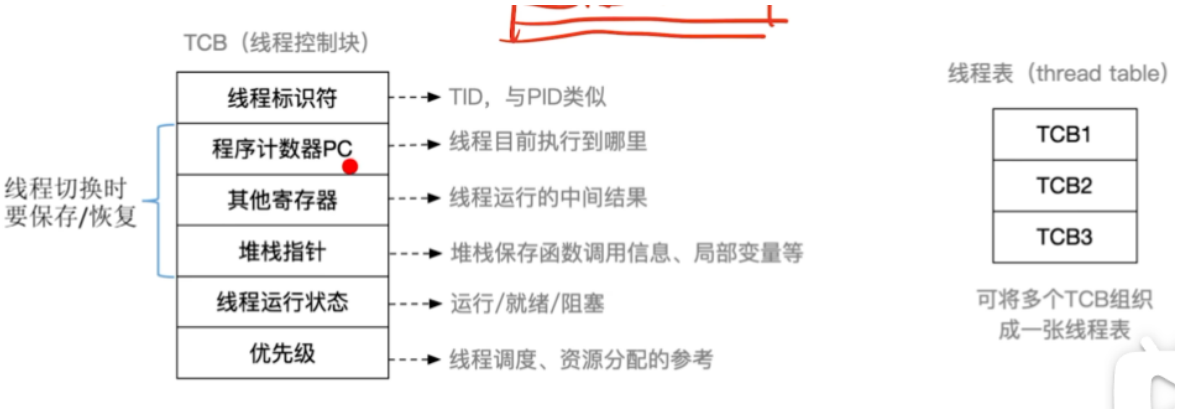
线程上下文 和指令流上下文一样，线程的上下文也是其寄存器组。

线程的状态 和指令流的状态是类似的，包括**运行、就绪和阻塞**三个状态。

问题 线程的上下文和状态放在用户程序里面还是操作系统里面?

线程控制块(TCB)

线程控制块 操作系统用以描述和管理线程的内核对象，一般至少包含线程的**时间预算、优先级、运行状态及上下文**，有时还会包含一些 **身份信息**（如线程名、线程号）或**统计信息**（如总计CPU时间） 等。它在数据结构上一般是C语言的一个结构体。



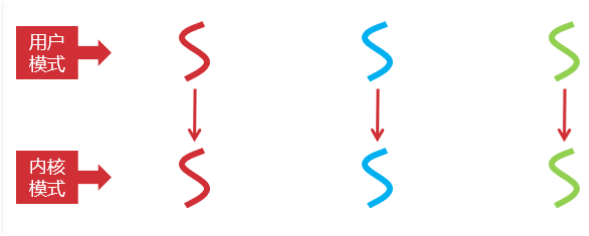
在那些有内核模式的处理器中，**线程控制块位于内核空间，只有操作系统可以更改，应用程序无法更改。**

线程号	0x1294
当前剩余时间预算	134
优先级	5
寄存器组	AX = 0X1234 BX = 0x5678 CX = 0xABCD .....
总消耗时间预算数	1247814

线程状态与指令流状态

线程的状态 指令流和线程都有就绪、运行、阻塞这三个状态。那么，这三 个状态和指令流的三个状态有什么区别和联系呢?

（1）指令流与线程：一对一

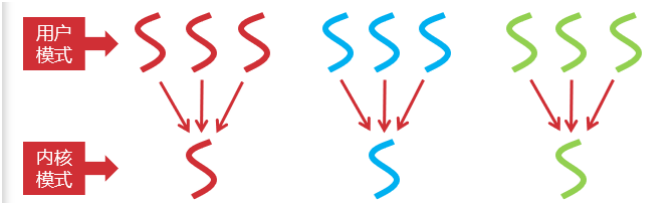


优点 简单。最简单，最好实现，最常见，常见到足以让人把指令流 和线程的概念混淆起来。

缺点 每个指令流都成了**单独分配时间**的对象，很多时候不需要这样。 这样会增加操作系统的负担，因为操作系统需要**为每个线程创建一些单独的管理数据**，而且每次**切换当前CPU上的指令流**都需要通知操作系统。

对应关系 此时指令流的状态和线程的状态是一对一对应的。**线程处于什么状态，指令流就处于什么状态。**

（2）指令流与线程：多对一



优点 高效。同一个线程中的多个指令流可以借由附着在同一个线程上共享一份执行时间，它们在内核中也被当作一个对象来处理， 其TCB仅仅创建一份。对于每一个线程上附着的多个就绪指令流， 应用程序负责决定哪个指令流得到线程从而运行，并切换到它。

通常而言，只有紧密协作的指令流才会被放在同一个线程上， 而且操作系统并不需要知道指令流的存在，因此仍然可以在那 些一对一的操作系统中实现。

对应关系

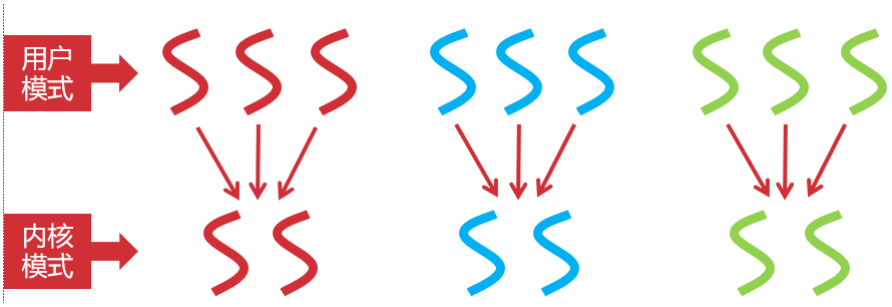
线程处于运行状态，说明其上的指令流中有一个在运行。

线程处于就绪状态，说明其上的指令流中至少存在一个就绪的。

线程处于阻塞状态，说明其上的指令流中至少一个发起了需要操作系统介入的阻塞态。

其它指令流 状态无法预测，因为只有应用程序才知道它。操作系统是不知 道它们的，因为操作系统感知的只是作为时间分配对象的线程 的状态。线程阻塞了，意味着内核只知道这个时间分配对象上 附着的某个指令流在请求内核完成一个一时半会无法完成的功 能，因此内核能做的就是暂停整个时间分配对象的执行。

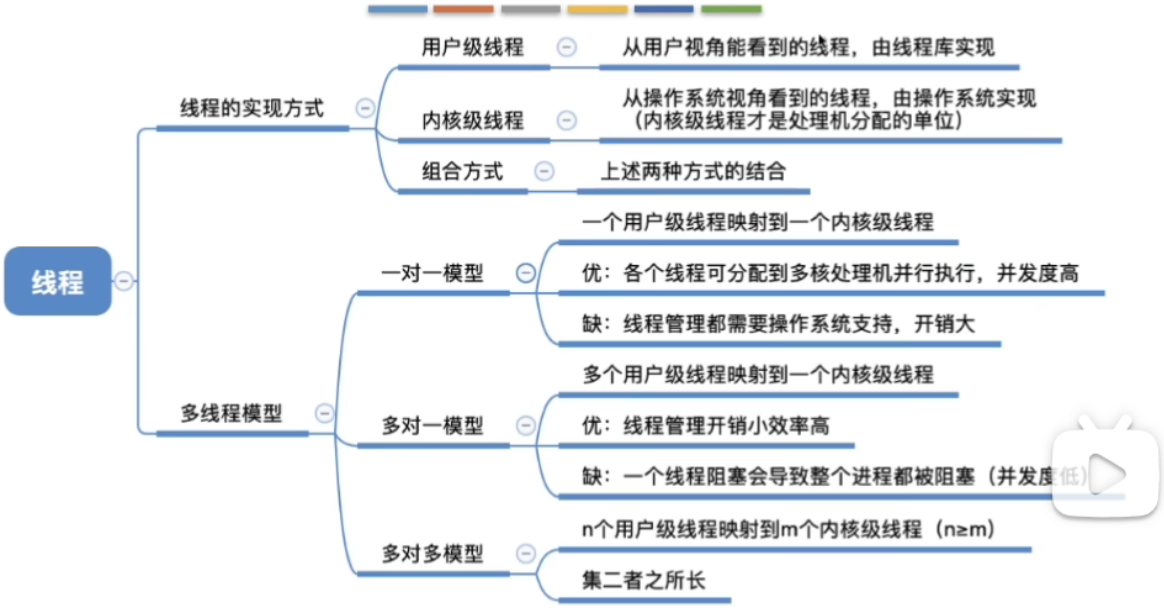
指令流与线程：多对多



原因 一旦指令**流阻塞在内核**（阻塞在应用程序中则无此问题，因为应用程序 可以自主选择一个新的指令流来使用这个线程），就会阻塞整个线程。 而如果线程上还有能运行的指令流，这些指令流也停止运行。我们自然 不希望这样；然而，每个线程仅提供一个TCB，当前阻塞在内核的指令流 的上下文已经占用这个TCB里面的寄存器组了。

优点 那么，我们可以考虑给这些指令流提供多于一个上下文，也就是让多个指令流对应多个线程，**任何一个不阻塞的线程都可以运行任何一个不阻塞的指令流，指令流可以在线程之间迁移。**

缺点 臭名昭著地**难以正确实现。**内部细节很多很复杂，这就是灵活的代价。



2.3纤程与协程

纤程与协程：一些额外概念

协程(强调合作） 合作执行的一组指令流。不仅强调它们**不是时间分配的独立对象**（区别 于线程），而且强调**只有其中某个指令流主动放弃CPU时**，其它指令流 才可得到CPU进行运行，并且**放弃CPU的那个指令流还倾向于指定谁来接替它的执行。**

实现 一般地，协程被实现为**在用户模式下的一系列数据结构**，这些 数据结构中**会保存协程的寄存器上下文。**在常见的实现中**协程对线程都是多对一结构**，因此在用户模式就可以完成互相切换， 无需通知内核，切换的效率远高于线程。

特别地，对于**无栈协程**（也即那些不使用栈，或者在放弃CPU时能保证栈内不存在有效数据的情况下），它们可以用C语言的 #define配合switch-case实现。

纤程（强调轻量） 合作执行的一组指令流。相比于协程，放弃CPU的那个指令流不倾 向于 直接指定接替执行者，而**倾向于唤起一个在用户空间的调度器**，由它来决 定下一个执行的指令流是谁。纤程之间不一定有紧密的合作关系，仅仅 是强调它们比线程要轻量，也即多个纤程共享一个线程。

比较 同多于异，几乎是同义词，都是指令流，只不过侧重点不同。

三.处理器调度算法

3.1调度算法原理

优先级和时间片

问题 现在有一系列线程在系统中运行。操作系统中的调度器负责决定每次运行哪个线程，运行多久，但又如何决定做这个决定所需的每个线程的优先级和时间片？

解决方案一

事先指定。对于某些系统，我们可以**根据某线程负责运行的指令流的性质**，将某些线程的优先级设置得高一些、时间片设置 的多一些，以体现资源分配的倾向性。整个系统只要按照这种设置来运行就足够完成其功能了。

优点 对任务知根知底，且任务固定时，这就是最好选择。

缺点 这要求操作系统**知晓整个计算机体系内的应用程序的性质**，基本上要求应用程序都是同一个团队开发的。更精确地讲，**需要额外信息来进行手动干预。**

用途 对于那些**功能简单**，但要求**绝对可靠**的系统，这是一个非常好的办法。它保证了某些指令流具有**绝对的资源优势**，确保那些指令流承担的工作总能按时正确完成。

例子 刹车；火箭；飞机；导弹

固定优先级调度算法

固定优先级 Fixed-Priority，FP

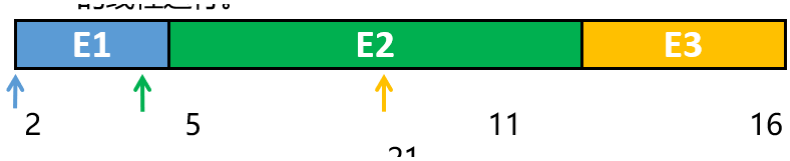
所有线程按照事先给定的优先级排序运行。时间片无限长。

线程	到达时间	运行时间	优先级
E1	2	4	1
E2	5	10	3
E3	11	5	2

在一般的操作系统中，优先级数字越小，优先级越高

非抢占式 调度仅在线程结束时发生，此时从队列里拿出一个优先级最高 的线程运行。(只允许进程主动放弃处理器)

实现简单，系统开销小，但是无法及时处理紧急任务，适合早期的批处理系统。



抢占式 调度在线程结束和线程就绪时都发生，此时从队列里拿出一个 优先级最高的线程运行。这等于说，如果有一个新的高优先级线程加入进来，它会取代当前的任务，立即获得CPU并运行。

（如果一个更重要或更紧急的进程需要适用处理机，则立即暂停正在执行的进程，将处理机分配给更重要紧迫的那个进程）

可优先处理更紧急的进程，也可以实现让进程按照时间片轮流执行的功能，适合分时系统，实时操作系统。

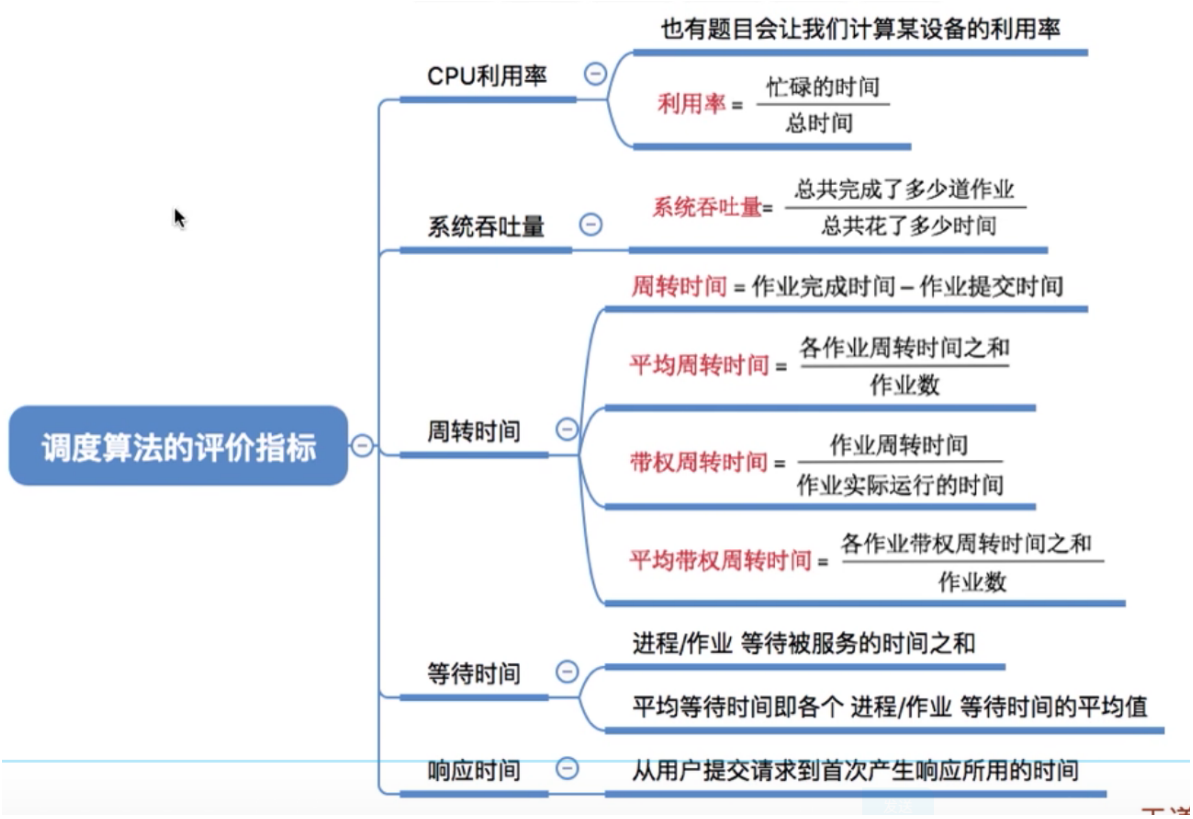
解决方案二

从应用程序的行为来推测其性质，并且按照某种策略给它分配优先级和时间片。首先，我们需要一些领导原则，决定我们的 策略的最终目的；第二，这种我们需要一些程序行为的测度， 作为刻画程序行为的手段。

问题 领导原则包括哪些?

提示：考虑多道程序设计的初衷。

调度算法的评价指标



1. **CPU利用率** 前面提过，一个很大的推动力就是保证处理器不会闲着。闲着 就是浪费资源，因为CPU要折旧，而且即便不使用也会耗电。因 此，我们需要保证CPU的利用**效率**。

系统的响应 多个程序可能同时竞争CPU。它们都宣称自己最需要CPU来做计 算，而且确实也都会提交一时半会算不完的任务。因此，我们 不能让某个应用程序霸占CPU，而是需要保证多个程序都能分到 一部分CPU来运行自己，保证CPU的利用**公正**。

问题 保证**效率**简单，还是保证**公正**简单?

保证CPU效率 非常容易，只要确保每次都选择CPU一直在跑某个就绪的线程就 可以了。只要CPU一直在做计算，CPU的效率肯定就是100%。

公平的常见测度

2. **吞吐率** 单位时间内执行完的线程的个数

如果在某段时间内，执行完成的任务越多，说明CPU分配越普惠， 就越公平。

假设在时间T内，有N个线程完成执行，就说它的吞吐率为N/T。

3. **平均等待时间** 线程从就绪态到运行态平均等待的时间

如果任何一个就绪的线程都越能尽快得到CPU，说明CPU分配的 歧视性成分越低，就越公平。

假设一共有N个线程，它们从就绪到得到CPU之前必须等W1，...， Wn时间，则平均等待时间为(W1+W2+...+Wn)/N。

4. **平均周转时间** 线程从就绪态到阻塞或停止态平均花费的时间

如果任何一个就绪的线程都越能尽快完成其执行，说明CPU分配 的歧视性成分越低、包容性成分越高，就越公平。

假设一共有N个线程，它们从就绪到得到停止运作分别经过 T1， ... ，Tn时间，则平均等待时间为(T1+T2+...+Tn)/N。

若设线程的运行时间为Ri，则Ti=Wi+Ri。

问题 平均周转时间测度与平均等待时间测度下，公平的具体含义发 生了什么变化？为什么？

正义的常见测度

5. **响应时间**

从用户提交第一个请求到系统做出第一个反馈的时间。

用户必然希望系统越快做出响应越好，不要拖拖拉拉。**如果响应时间越短，说明执行顺序的安排越符合用户的意图，就越正义。**

问题 平均周转时间和响应时间相比有何区别？为何说前者体现了公平，后者体现的则是正义？

平均周转时间关心一切线程的运行时间，而响应时间仅仅关心 那个用户刚刚提交的线程产生的第一个反馈。前者关心所有线 程，后者只关心某个线程，仅仅是因为那个线程是用户刚刚提 交的，它的响应结果可能是用户正在等待的。

6. **综合指标** 采用**平均带权周转时间**，其计算式为(T1/C1+T2/C2+...+Tn/Cn)/N。 T/C又叫**响应比**，T/C=1+(W/C)。

响应比 = (等待时间 + 服务时间) / 服务时间 = 1 + (等待时间 / 服务时间)

T表示每个作业的带权周转时间，C表示每个作业的CPU时间或执行时间，

观察 对于同样的W，C越小，W/C越大；对于同样的C，W越大，W/C 也越大。一个越短任务的等待时间越长，对这个指标越不利。

公平体现在该式包含了对所有任务的考虑。

正义体现在该式对越短任务的越长等待越不容忍

公平的极端：先来先服务

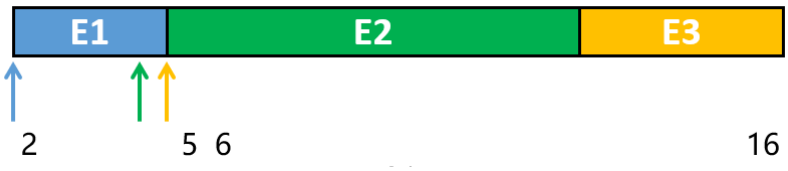
先来先服务 First-Come First-Served，FCFS

所有任务按照其提交时间排序，提交时间越早的任务优先级越 高，越优先得到CPU。调度仅在任务结束时发生。

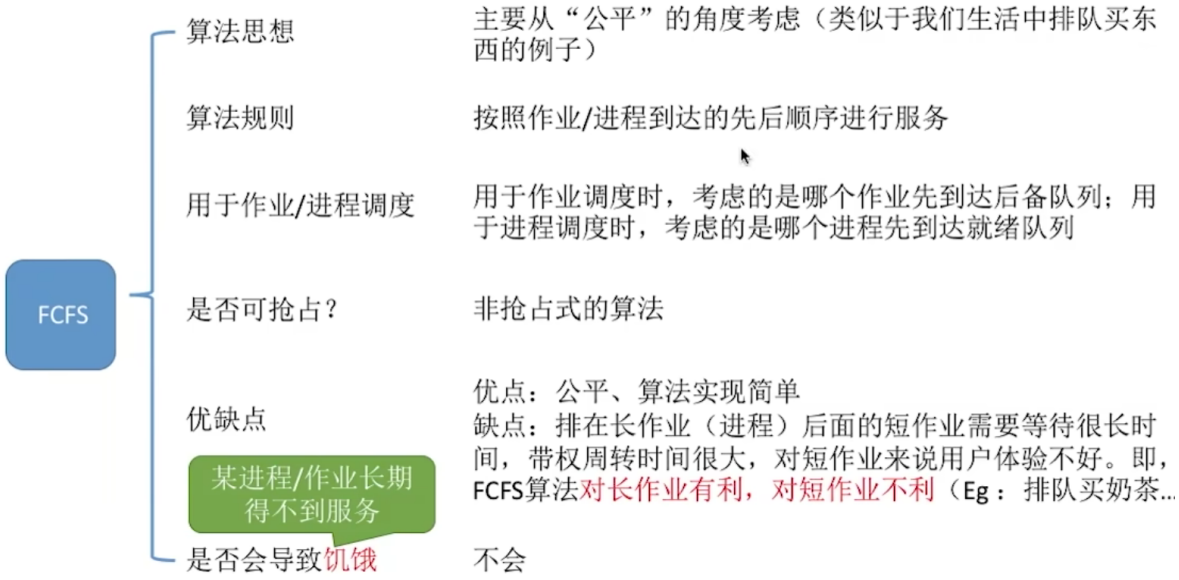
特点 公平简单有效，对于任务短小且简单的场合这就足够了。

例 三个线程E1、E2、E3，参数如下所示，提交给操作系统。

线程	到达时间	运行时间
E1	2	4
E2	5	10
E3	6	5



问题 E1~E3的等待时间和周转时间各是多少？整个体系的平均时间呢？



正义的极端：短作业优先

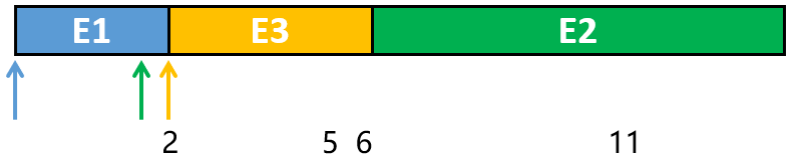
短作业优先 Shortest-(Remaining)Job First，SJF

所有任务按照其运行时间排序，运行时间越短的任务优先级越高，越优先得到CPU。调度可在任务结束时和任务提交时发生。

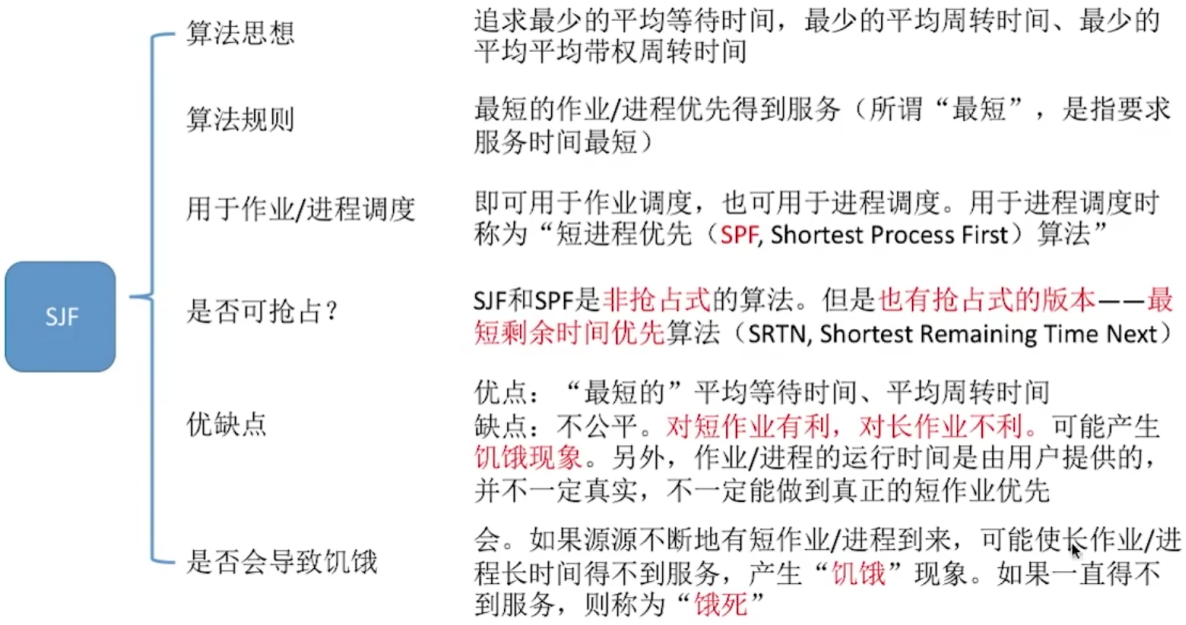
特点 响应性好，适合简单的交互式系统。

例 三个线程E1、E2、E3，参数如下所示，提交给操作系统。

线程	到达时间	运行时间
E1	2	4
E2	5	10
E3	6	5



问题 看上去SJF就是比FCFS好，短作业很快就能得到响应。果真如此?



调度算法的在线性

在线算法 On-Line Algorithm

算法必须在启动后逐个接受输入并即时给出输出。输入并非一 次交给算法的，算法无法预测之后会遇到什么输入。

问题 如果在E3执行完成前，用户向系统中提交E4、E5、E6，这些任务 所需要的执行时间都比E3短，那么意味着它们会插队到E3之前。

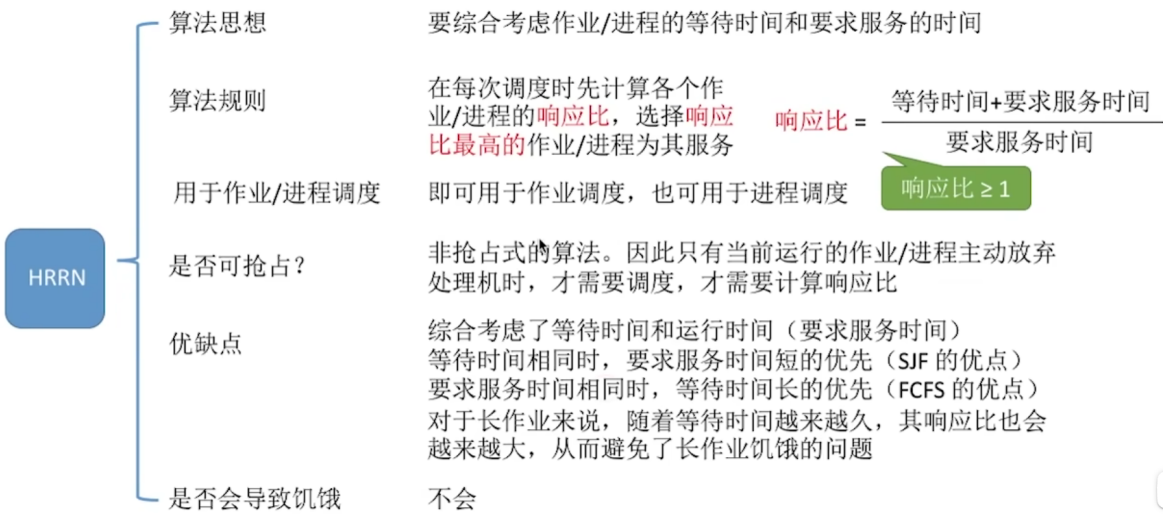
如果用户一直提交短作业，那意味着E2将永远得不到执行了。 这和FCFS的情况不同，对于FCFS，后到的作业一定后运行。

饥饿现象 依照某种资源分配策略，**某些请求无限增加时，另一些请求将永远得不到分配。**在CPU调度的问题上，它体现为某些线程将永 远不能获得CPU。

响应比高优先 Highest Response Ratio Next，HRRN

所有任务按照其响应比排序，**响应比越高的任务优先级越高**， 越优先得到CPU。调度仅在任务结束时发生。是FCFS和SJF的折中， 具备两方优点但又不极端。

通常而言，调度仅在任务结束时发生。



交互系统的调度

问题 前面提到的各个算法中，要么使用基于固定优先级的抢占（FP或 者SJF），要么就干脆排排坐吃果果。前者在复杂的场景会导致 饥饿，从而可能导致系统卡死，后者则无法在一个任务开始执 行后将其中途打断。那么，能否有一种算法，不产生饥饿，又 能打断长任务以获得交互能力呢?

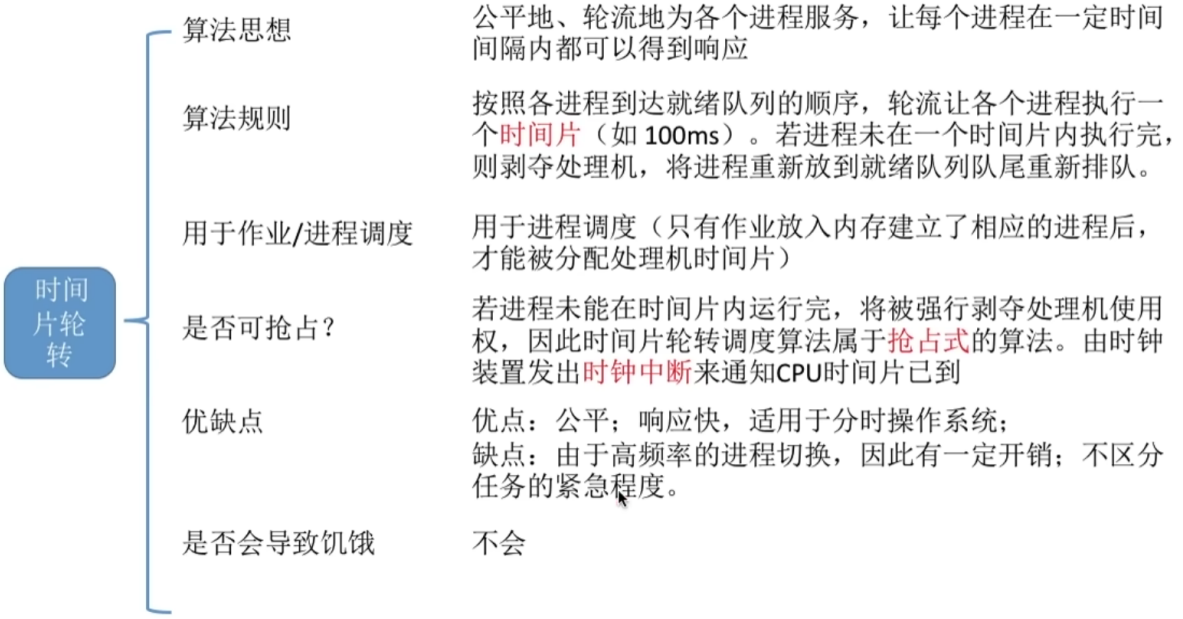
时间片轮转法 Round-Robin，RR（动态优先级）

将**每个线程的时间预算切成规模较小的时间片**，每次只运行一 片时间，然后再运行下一个任务。可以看作是**FCFS的一种改进**： 任务先来先执行，但执行时间到就换成下一个任务，等到所有 任务都轮到一遍了，再回到第一个任务执行。

RR是抢占式的，因为它会**打断超时线程**的执行。

特点 交互性好，所有线程都获得了执行的机会，无论长短；可以保 证在某个时间周期之内，所有的线程都获得至少一次执行机会。

问题 时间片的大小怎么决定？长了会怎么样？短了呢？每次每个线程分配到的时间片都必须一样吗?



时间片轮转法的改进

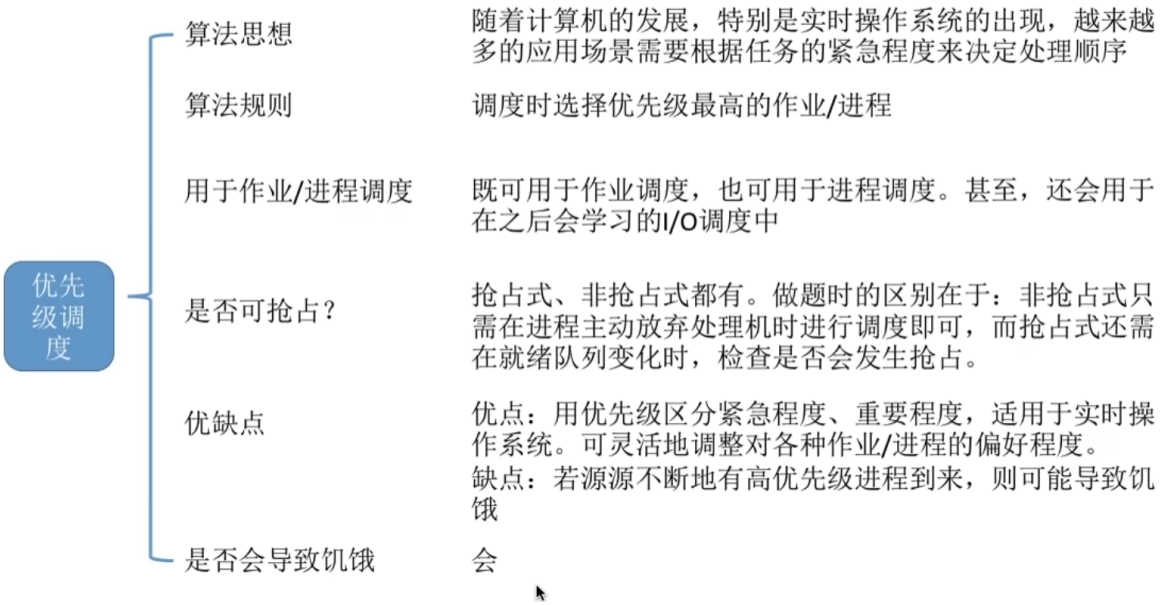
问题 时间片轮转法无视了进程的固定优先级。如何把固定优先级加 回系统，使其能够做到保证平均CPU获得的基础上，满足某些特 别任务的高优先级需求?

固定优先级时间片轮转法 Fixed-Priority Round-Robin，FPRR

将时间片轮转法进行改进，同一个优先级的任务采取时间片轮转法，不同优先级的任务之间则采取严格的抢占式固定优先级调度。

线程	到达时间	运行时间	优先级
E1	0	4	1
E2	0	10	2
E3	0	5	2

特点 非常适合某些小型实时系统。即便在大型系统中，有时候也能 取得不错的性能。最重要的是，在所有效果不错的调度器里面， 它能做到O(1)查询。（调度器的性能为何重要？）



复杂系统的调度：按线程行为分类

问题 单一的调度算法各有各的缺点。怎么将这些调度算法组合起来， 形成一个综合的算法?

多级队列 将线程按类型分成不同的队列，每个队列采用适合自身的调度算法，队列之间又有队列间的调度算法。

问题 同一个线程可能承载不同的指令流；同一个指令流的行为在不 同的执行阶段也可能发生改变。

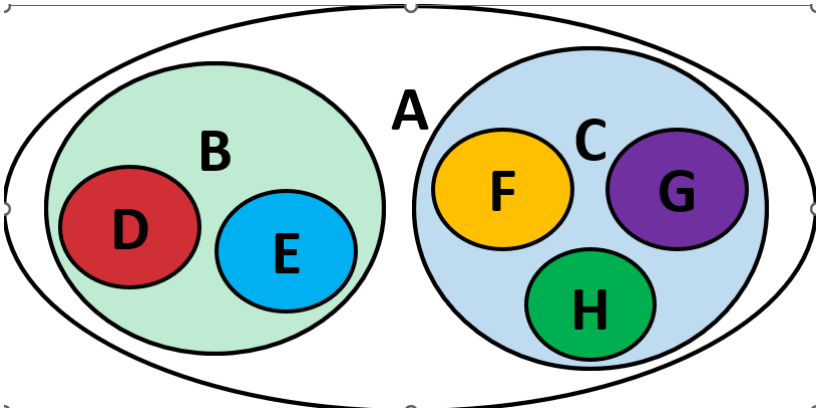
多级反馈队列 在多级队列的基础上，实时动态检测每个线程的行为，并在原 有队列不合适时为其更换到合适的队列。

层次化调度算法：按线程所在的子系统分类

问题 除了将线程按照自身类别分类，还可以按照什么方法分类?

层次化调度 将一个系统分成多个子系统，每个子系统内部有自己的子调度 器并采用适合自身的调度算法，一个父调度器则负责调度多个 子系统。子系统间的时间干扰可以被父调度器加以限制，非常 适合混合关键度（Mixed-Criticality）系统。

关于层次化调度，我们在实时系统和混合关键度系统的章节再加以涉及。



广义的调度算法

长期调度 决定选中哪些**可执行文件**，并将它们装入虚拟内存。

主要是围绕着**内核对象的创建和虚拟内存的映射。**

调度的是抽象层次的任务集合，一般对应着一项有实际意义的 工作。

长期调度一般是用户编写的各类定时执行脚本（Cron Job）或者 用户模式实用程序（Sun Grid Engine）完成的，因为要装入什么 东西只有用户自己知道。

中期调度 决定哪些准备好的工作需要实际装入物理内存，哪些装在物理 内存里的工作暂时不执行，因而需要换出来。

主要是围绕者**页面文件**的扇入和扇出。

调度的一般是空间保护域，也即进程。

（短期）调度 决定哪个装入物理内存的应用程序实例（进程）的**线程**可以得 到CPU来执行。通常指的调度就是短期调度。关于中期调度和长 期调度，我们在讲存储器管理时再来涉及。

3.2多处理器与并发

并发性 多个指令流依附于多个位于不同物理处理器上线程，做到了多 个指令流的真正同时执行。并发是并行的一种特殊场合，它只 有在多核处理器上才有可能实现。

并发与并行 并发是并行的一种具体实现，在并发环境中，不仅并行程序的 各个指令流的指令执行的先后顺序无法预测，而且这些指令流 实现了真正的同时、一齐执行。在单核处理器的并发环境中， 无法实现并行，因为只有一个CPU，不可能同时执行多道程序， 仅仅是交替执行多道程序让它们看上去在同时运行。

并发和并行**不是一对矛盾概念或者反对概念**，而是**上级和下级**、 **抽象和具体**、**包含和被包含**的关系。如果有人考你并发和并行 有什么区别，那大概是想考查"**是否在多个CPU上同时执行**"这个 考点。

最终问题 多处理器调度可能遇到哪些问题?

4.2第二节 处理器调度机制和实现

一.线程的实现

1.1系统线程总览

系统中的线程总览：按优先级划分

按照操作系统中的**线程的优先级**，大致可以将它们分为如下五类。

1. 输入输出线程 I/O线程必须与硬件打交道。这意味着，它们要对硬件的变动做出即时的响应，因而一般位于极高优先级段，且是实时的。

例子 鼠标键盘驱动、网卡驱动、显卡驱动。

速率限制 限制驱动程序占用CPU的时间或频次，以防驱动程序占用大量CPU。部分外设会大量生成中断，这将导致系统频繁发生线程 调度和切换（调度和切换都不是免费的），降低CPU的利用率。

又称为**节流（Throttle）**。后面我们还要回顾这个概念

2. 系统服务线程 一系列为应用程序提供必要服务的线程。这些线程承担着重要 的职责，因而一般位于较高优先级段。

例子 网络协议栈、文件系统、用户登录服务、远程桌面服务等

3. 前台程序线程 当前用户正在交互的应用程序中的线程。它们可能正在等待用 户的输入，或者正需要把输出反馈给用户，因此总是希望它们 能较为优先的得到CPU。但考虑到它们不如系统服务和驱动程序那样紧急，因此在系统中处于中间优先级位置。

例子 文字处理软件、视频游戏、浏览器、即时通信软件等。

4. 后台服务线程 提供一些时间不敏感的背景服务的线程。这些线程一般不直接 和系统中时间敏感的部分产生交集，但对长期运行的系统的稳 定往往十分重要。如果CPU上当前没有什么重要的任务，它们就运行。它们处于较低的优先级。

例子 磁盘碎片整理、文件索引服务、垃圾文件清理、杀毒软件等。

5. CPU空闲线程 由操作系统提供的、当CPU确实无事可做时运行的线程。它通常 只做一件事，就是反复将当前处理器置于睡眠态以节约功耗， 它处于最低优先级。

备注 在多CPU系统中，每个CPU上都有一个空闲线程。当该CPU无线 程可调度时，就调度它。

系统中的线程总览：按来源划分

按照操作系统中的线程的来源，大致可以将它们分为如下三类。

1. 驱动程序线程 驱动程序由设备提供商编写。如果需要安装新的硬件，那么必 须安装适合它的驱动程序。驱动程序线程的运行交给操作系统 管理，用户不能直接管理。

2. 系统内核线程 完成操作系统功能所必备的线程，它们运行由操作系统开发商 提供的功能。这些线程对用户而言是透明的。

3. 应用程序线程 一般的应用程序的线程，它们运行由应用程序开发商提供的应 用程序中包含的指令流。

备注 这个分类不是绝对的。事实上，操作系统一般会自带一些驱动 程序，而驱动程序的发行包中也可能含有应用程序组件。

系统中的线程总览：按特权划分

按照操作系统中的线程的特权，大致可以将它们分为如下两类。

1. 内核线程 运行在内核空间的线程；它们运行时，CPU处于内核模式。它 们可以直接访问内核的一切资源。正因如此，它们均由操作系 统内核启动和管理，甚至可以视作是内核的一部分。

只有可抢占内核有内核线程。后面详讲。

2. 用户线程 运行在用户空间的线程；它们运行时，CPU处于用户模式。它们 进入内核的唯一方法是系统调用。常说的线程就是指用户线程。

概念辨析 “内核线程（Kernel Thread）/用户线程（User Thread）”不要 和“内核级线程（Kernel-Level Thread）/用户级线程（User-Level Thread）”搞混了，前者是指线程运行时的CPU模式，而后者则 是指操作系统是否知道它们的存在。内核线程和用户线程都是 内核级线程，而所谓的“用户级线程”则是指协程和纤程。

这两个叫法之所以叫成这样是历史原因。不必深究。

系统中的线程总结：

线程的优先级	一般情况下的特权模式	一般情况下的提供方	常见调度策略
输入输出线程	内核模式	驱动程序	EDF，FPRR
系统服务线程	内核模式	操作系统	FPRR，FCFS
前台程序线程	用户模式	应用程序	CFS，常规RR
后台服务线程	用户模式	操作系统、应用程序	FCFS
CPU空闲线程	内核模式	操作系统	-

1.2线程的实现

线程控制块TCB

线程号	0x1294
当前剩余时间预算	134
优先级	5
寄存器组	AX = 0X1234 BX = 0x5678 CX = 0xABCD .....
总消耗时间预算数	1247814

1.3上下文切换

线程的上下文

为什么线程也有上下文?

考虑线程是的**执行流因为主动等待需要操作系统介入的I/O完成或者意外的被外设中断打断**而暂停运行的场合。

内核阻塞

操作系统不知道指令流的存在。因此在遇到线程上的指令流陷入内核阻塞的时候，内核只能暂停执行当前这个线程，切花u你到别的线程去执行了，更麻烦的是线程什么时候陷入内核，依附在线程上的指令流是不知道的，也即可能发生抢占。

线程的上下文

和指令流上下文一样，线程的上下文也是其寄存器组

线程的状态

和指令流的状态乐死，包括运行，就绪和阻塞三个状态

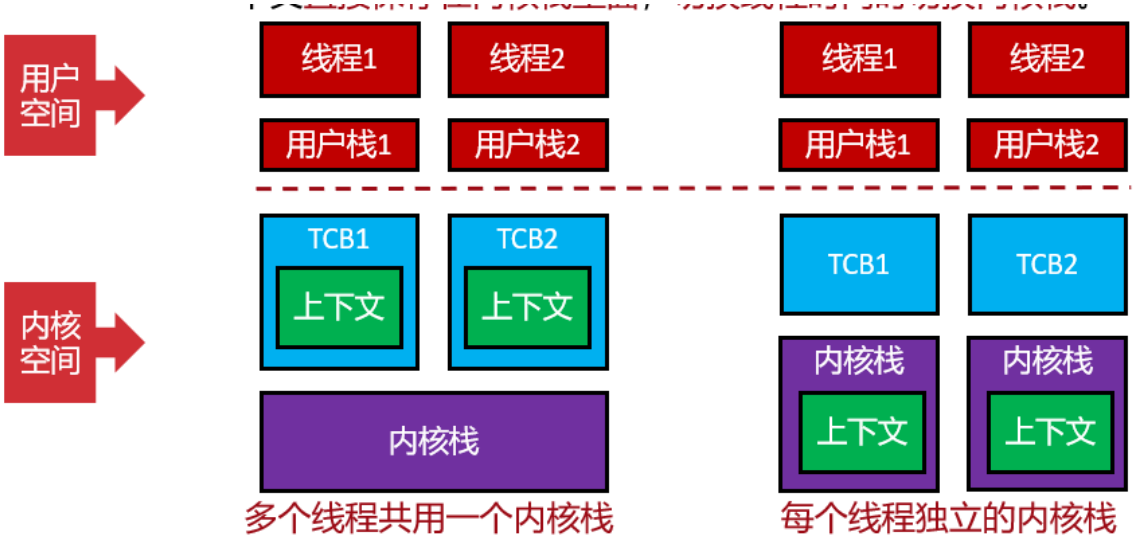
线程与内核栈

多个线程共用一个内核栈线程切换过程中，多一个栈上寄存器与TCB的数据交换过程，

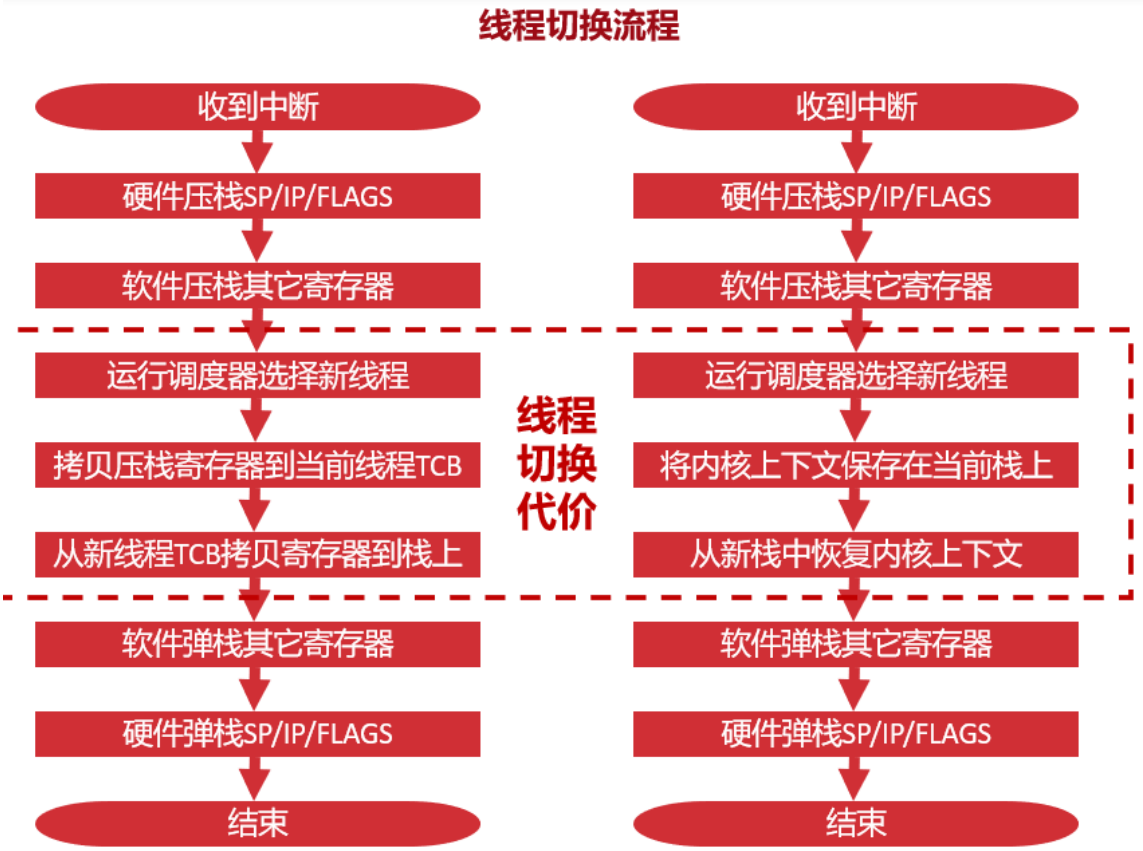
每个线程独立的内核栈线程切换过程中，需要切换内核上下文和内核栈，

两个方法的区别在哪里？

提示：宏内核的复杂性；共用一个线程栈，则内核上下文也只 有一个，因此内核执行完某一个线程前不能响应其它事务。



线程切换流程



可抢占内核

系统调用等操作系统事务的执行可以被中途打断，转去执行其他更加紧急的事务，并在合适的时候恢复系统调用的执行。这对缩短系统的最坏响应时间有益，他要求在打断系统执行时保存内核的上下文，因此每个线程都需要一个内核栈。

辨析：可抢占内核和抢占式调度

前者指的是内核的执行可以被打断，后者指的是应用程序的执行可以被打断，一个支持应用程序抢占式调度的内核，自身完全可以是不可抢占的。

二.调度算法实现

2.1Linux

Linux调度器的历史

（1）单队列调度器　整个系统只有一个运行队列，每次调度时遍历队列

（2）多队列调度器，区分实时线程和非实时线程

（3）O（n）调度器，基于时间片轮转的调度器

（4）O（1）调度器，多链表位图调度器，无需遍历队列

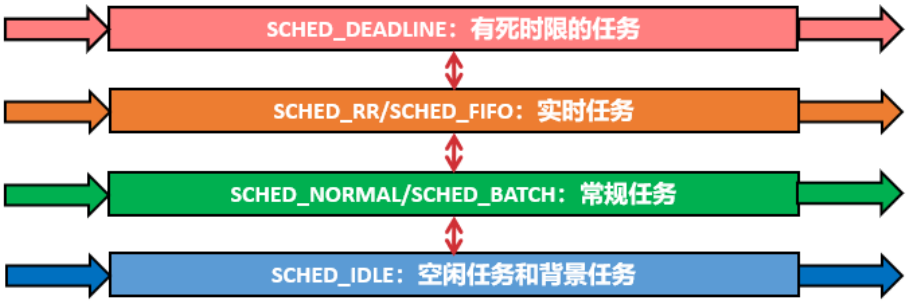
（5）CFS调度器　更公平的调度器，但是数据结构和算法设计稍稍复杂，其时间按发展的实际上是O(logn)

（6）截止期调度器　固定截至期限的任务

（7）现在　能量感知调度，算力感知调度

操作系统的线程调度：Linux

调度队列类别：Linux具备如下几个队列，其队列的**优先级从高到低**分别为：



SCHED_DEADLINE使用最早截止时间（Earliest Deadline First，EDF） 调度策略执行那些最紧急的任务。

SCHED_RR和**SCHED_FIFO**分别使用固定优先级时间片轮转法（FPRR）和先到先服务法（FCFS）调度那些实时任务。

SCHED_NORMAL、**SCHED_BATCH**和**SCHED_IDLE**使用完全公平调度（**Completely Fair Scheduler，CFS**）策略调度那些常规任务。

本课程**重点讲CFS。**EDF和FPRR的实现留到后面实时系统再讲

CFS调度器的设计思想——保证每个线程对CPU有相同，公平的占有能力

虚拟CPU切分

每个线程都有一个虚拟CPU，它**完全占有一个CPU**。CFS的核心思想就是**将物理CPU切割为这些虚拟CPU，并且保证这些虚拟CPU 以最精确的粒度齐头并进**，好像一个主频为kN的CPU被切分为k个主频为N的CPU，并且每个CPU分别运行一个线程那样。

理想运行时间

基于虚拟CPU切分的思想，我们为每个线程准备一个理想运行时 间（Ri）变量。随着实际时间（wall clock time）的增长，Ri也自 动按比例增长。假设系统中有10个任务，且在现实中已经过去 10秒，则每个任务应得的Ri都各增加1秒。

虚拟运行时间

同时，我们为每个线程维护一个虚拟运行时间（Rv）变量。 该变量记载线程实际上得到了多少CPU。

运行时间差值

定义运行时间差值Rd=Ri-Rv。

调度决定

基于Rd，在每次调度时都调度**Rd最大**的那个线程并给予它一定的 CPU运行期。这样就保证了当前最饥饿的那个线程总是被先调度。

算法优化

查找最小Rv

如果每个Ri都是按比例增长的，那么找最大的Rd就是等价于找最小的Rv。因此，我们只要每次调度选择那个Rv最小的线程，他肯定是Rd最大的最饥饿线程，

Rv的数据结构

我们需要查询Rv最小的那个线程

线程就绪，阻塞或更新Rv时，我们需要添加或去除线程

问题：什么数据结构来驹泽Rv以及线程的TCB

使用红黑树

原因：红黑树是一种自平衡二叉查找树，从根结点触发到任何叶子结点的路上的长度最多不超过2倍，最短的情况，路径上均是黑色节点，最长的情况是黑红交替。

红黑树的规则：

根节点和叶子节点都是黑色

每个红色节点的两个子节点都是黑色

任意节点到叶子节点路径都包含相同数目的黑色节点。

红黑树在内核的其他功能如文件系统和驱动程序中使用也十分普遍，因此CFS调度器并未增加新的数据结构，只是复用了在内核中久经验证的同一套代码，不增加编程和调试工作量

同时保证了调度器的平均性能和最坏性能

CFS的额外优化

按比例增长的Rv

事实上，并非每个Rv都是同步增长的，Linux调度器会给每个线程不同的Rv比例，对于那些被偏爱的线程，其Rv值在记载时会**乘上一个比较小的系数**，这样它的运行时间就被往少了算。分配时间时就会更加倾向他。

进程组

按照某些规则将一些进程分成一组，其中的线程使用的**总时间被限制**，这样可以限制某些用户或程序占用CPU时间，放置集体作弊

占用限额

允许对某些线程设备CPU占用率的上下限。

CPU亲和性

在多CPU中，某些线程的数据在某些CPU核的缓存上已经缓存好了，如果要调度他到其他核，就要重新预热缓存，会浪费时间。

2.2windows

调度队列类别 Windows具备如下几个队列，其队列的优先度从高到低分别为：

Windows的优先级数字越大，优先级越高。实时线程的优先级为31-16，它们采用固定优先级法调度，常规应用程序的优先级可变，在15-1。

2.3MacOS

注意这里：MacOS和iOS特别强调交互性，专门给那些前台和交互系统开了窗口。苹果系统对流畅度的追求可见一斑。苹果系统的优先级也是数值越大越高。

三.系统线程接口

3.1线程的基本操作

1. 创建create

   创建一个线程，包括它的线程控制块等内核数据结构，操作还会返回线程的句柄Handle，这一般时它的线程号（Thread ID）

2. 出让Yield

   通知操作系统当前线程不需要更多CPU时间，自愿放弃CPU

3. 终止 Terminate/Exit

   线程自愿停止自己的执行并退出，可以返回一个返回值

4. 同步Join/Wait

   使一个线程等待另一个线程终止，并回去它的返回值，选哟给出被等待线程的TID。很多时候，这个操作也会导致线程销毁，因为终止的线程除了交还它的返回值以外没有任何意义。同步操作等待的对象必须是由发起者创建的，且对象不能脱离（Detach）发起者

5. 销毁 Destory/Delete

   销毁一个同步完成的线程。

6. 设置优先级

   几乎所有操作系统都允许用户修改线程的优先级。好处是显而易见的： 用户可以将希望得到较好响应性的应用程序的线程设置得较高。

7. 设置调度策略

   部分操作系统允许用户给线程指定甚至亲自提供一个调度策略。有时候 用户比操作系统更清楚什么策略适合自己的线程。

8. 设置处理器亲和性

   通知操作系统当前线程只能在哪（些）CPU上调度。这对于提高数据的 局部性很有益处，能增进CPU的效率。在那些异构架构（如Intel大小核） 上，还可以指定哪些线程使用大核、哪些线程使用小核，从而最大化体 系性能。

9. 设置线程栈大小

   在创建线程时通知操作系统给它分配更大的用户模式运行栈。某些线程 中可能使用递归算法，这使得它们对栈的消耗量偏高。

3.2线程接口实例——pthreads运行时库

一个POSIX标准的运行时库，实现了线程的所有常用功能。它并非操作系统的原生接口，而是调用操作系统的接口来完成自己的功能。当然，也可以自行调用操作系统的原生接口来操作线程，但不如使用pthreads库来得好移植。

pthreads接口	Linux原生接口	Windows原生接口	功能
pthread_create	fork/clone	CreateThread	创建线程
pthread_yield	sched_yield	SwitchToThread	出让处理器
pthread_join	waitpid/wait4	WaitForSingleObject	同步线程
pthread_cancel	tkill	TerminateThread	删除线程
pthread_exit	exit	ExitThread	退出当前线程
pthread_self	gettid	GetThreadId	查询自己的句柄
pthread_ setschedparam	sched_setparam	SetThreadPriority	设置调度器，包括策略和优先级
pthread _setaffinity_np	sched_setaffinity	SetThreadAffinityMask	设置处理器亲和性
pthread _attr_setstacksize	setrlimit	CreateThread中携带	设置线程栈大小

语言运行时环境

1. Runnable Java
2. Thread C# PHP
3. Worker Javascript
4. threading Python
5. Goroutine GO