

Lecture 3: 知识 - 处理器调度

调度的基本概念

什么是调度

概念：什么是调度

- 概念：
 - 当有一堆任务要处理，但由于资源有限，这些事情没法同时处理；
 - 这就需要确定某种规则来决定处理这些任务的顺序，这就是**调用**研究的问题；
- 概念：处理机调度
 - 就是从就绪队列中按照一定的算法选择一个进程并将处理机分配给它运行，以实现进程的并发执行；
- 补充：调度的对象
 - 对于支持线程的操作系统，操作系统实际调度的是内核级线程而非进程；
 - 不过，术语进程调度（`process scheduling`）或线程调度（`thread scheduling`）常常交替使用；
- 图示：进程的 `CPU` 状态状态

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
139297	jyy	20	0	2356	576	512	R	23.5	0.0	0:02.27	a.out
139314	jyy	20	0	2356	576	512	R	23.5	0.0	0:01.80	a.out
139322	jyy	20	0	2356	576	512	R	23.5	0.0	0:01.65	a.out
139347	jyy	20	0	2356	576	512	R	23.5	0.0	0:01.41	a.out
139306	jyy	20	0	2356	576	512	R	17.6	0.0	0:01.95	a.out
139328	jyy	20	0	2356	576	512	R	17.6	0.0	0:01.57	a.out
139334	jyy	20	0	2356	512	448	R	17.6	0.0	0:01.53	a.out
139342	jyy	20	0	2356	576	512	R	17.6	0.0	0:01.45	a.out
1	root	20	0	167504	11564	8372	S	0.0	0.1	0:02.32	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp

三个层次的调度

概念：什么是作业

- 作业 = 一个具体的任务；
- 用户向系统提交一个作业 = 用户让操作系统启动一个程序（来处理一个具体的任务）；

概念：高级调度（作业调度）

- 概念：
 - 按一定的原则从**外存上处于后备队列的作业**中挑选一个（或多个）**作业**，给它们分配内存等必要资源，并建立相应的进程（建立 `PCB`），以使它（们）获得竞争处理机的权利；
 - 一定发生，但是频率低；
- 过程：

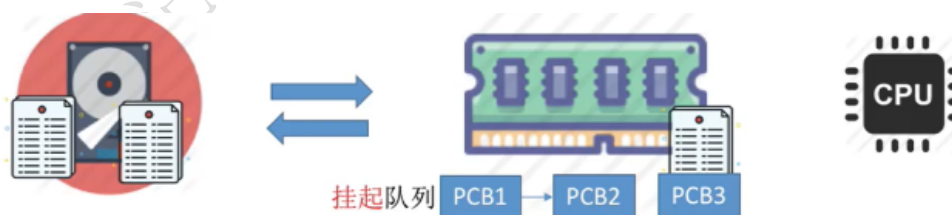
- 外存 -> 内存;
- 面向作业, 使得进程完成从 创建态 -> 就绪态;
- 解释:
 - 高级调度是辅存 (外存) 与内存之间的调度。
 - 每个作业只调入一次, 调出一次;
 - 作业调入时会**建立相应的 PCB, 作业调出时才撤销 PCB**。
 - 高级调度主要指**调入**的问题, 因为只有调入的时机需要操作系统来确定, 但调出的时机必然是作业运行结束才调出;
- 图示:



- 后备队列

概念: 中级调度 (内存调度)

- 概念:
 - 1. 决定将**哪个处于挂起状态的进程**重新调入**内存**;
 - 2. 将内存中长期处于阻塞状态或者不运行的进程从内存挂起到外存, **节省内存空间**;
 - 不一定发生;
- 过程:
 - 外存 -> 内存;
 - 面向进程, 中级调度实际上是存储器管理中的对换功能;
- 特点:
 - 一个进程可能会被多次调出、调入内存, 因此中级调度发生的频率要比高级调度更高;
- 图示:



引入了虚拟存储技术之后, 可将暂时不能运行的进程调至外存等待。等它重新具备了运行条件且内存又稍有空闲时, 再重新调入内存。

这么做的目的是为了**提高内存利用率和系统吞吐量**。

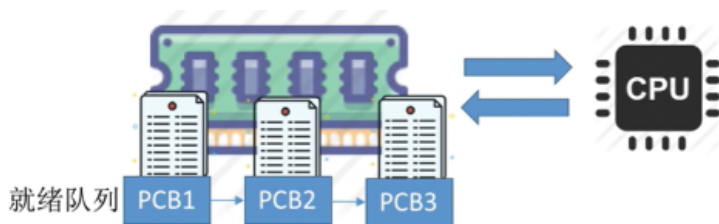
暂时调到外存等待的进程状态为**挂起状态**。值得注意的是, **PCB**并不会一起调到外存, 而是**会常驻内存**。PCB中会记录进程数据在外存中的存放位置, 进程状态等信息, 操作系统通过内存中的PCB来保持对各个进程的监控、管理。被挂起的进程PCB会被放到的**挂起队列**中。

-

概念: 低级调度 (进程调度)

- 概念:
 - 其主要任务是按照某种方法和策略从**就绪队列中选取一个进程**, 将处理机分配给它;
 - 进程调度是操作系统中最基本的一种调度, 在一般的操作系统中都必须配置进程调度;
 - 一定发生, 频率高;

- 特点：
 - 进程调度的频率很高，一般几十毫秒一次；
- 过程：
 - 内存 -> CPU；
 - 面向进程；
- 图示：



分析：三层调度的联系与对比

- 图示：

	要做什么	调度发生在..	发生频率	对进程状态的影响
高级调度 (作业调度)	按照某种规则，从后备队列中选择合适的作业将其调入内存，并为其创建进程	外存→内存 (面向作业)	最低	无→创建态→就绪态
中级调度 (内存调度)	按照某种规则，从挂起队列中选择合适的进程将其数据调回内存	外存→内存 (面向进程)	中等	挂起态→就绪态 (阻塞挂起→阻塞态)
低级调度 (进程调度)	按照某种规则，从就绪队列中选择一个进程为其分配处理器	内存→CPU	最高	就绪态→运行态

七状态模型

概念：挂起

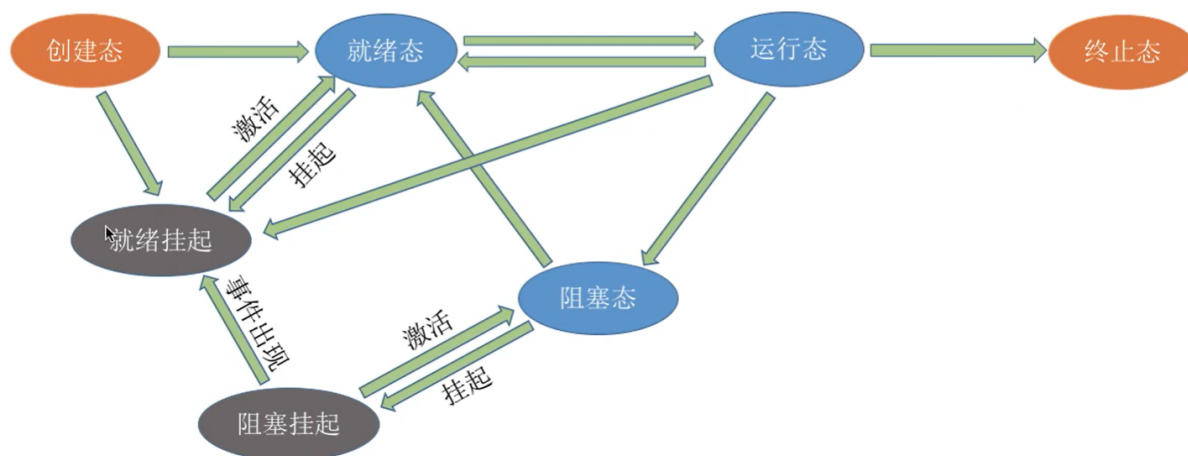
- 概念：
 - 挂起状： **suspend**
 - 暂时调到外存等待的进程状态为挂起状态；
- 两种状态：
 - 就绪挂起状态
 - 阻塞挂起状态
- **挂起** 和 **阻塞** 的区别：
 - 1. 两种状态都是暂时不能获得 **CPU** 的服务；
 - 2. 挂起态是将进程映像调到**外存**；
 - 3. 阻塞态下进程映像**在内存中**；

概念：七状态模型

- 解释：
 - 就是在原本的五状态模型上增加了挂起这一状态，此状态又分为就绪时的挂起与阻塞时的

挂起，进而变成七种状态；

- 图示：



调度的目标

调度的算法性能指标

概念： CPU利用率

- 概念：指 CPU 忙碌的时间占总时间的比例；
- 公式：

$$CPU\text{利用率} = \frac{CPU\text{有效工作时间}}{CPU\text{有效工作时间} + CPU\text{空闲等待时间}}$$

- 注意：
 - (1) 可以使用甘特图来进行计算；
 - (2) 计算作业完成时间时，要注意 CPU 与设备、设备与设备之间是可以并行的；
- 补充：对于一个实际系统，它的范围应从 40%（轻负荷系统）到 90%（重负荷系统）；

概念： 系统吞吐量

- 概念：单位时间内完成作业的数量；
- 公式：

$$\text{系统吞吐量} = \frac{\text{总共完成了多少道作业}}{\text{总共花了多少时间}}$$

概念： 周转时间

- 概念：指从作业提交给系统开始，到作业完成为止花了多少时间。
- 组成：周转时间为所有时间段之和；
 - 1. 等待进入内存：作业在外存后备队列上等待作业调度（高级调度）的时间；
 - 2. 就绪队列中等待：进程在就绪队列上等待进程调度（低级调度）的时间；
 - 3. 进程在 CPU 上执行的时间
 - 4. 进程等待 I/O 操作完成的时间
- 注意：后三项在一个作业的整个处理过程中，可能发生多次；

- 公式： 周转时间 = 作业完成时间 - 作业提交时间

概念：几种周转时间

- 公式：

$$\text{平均周转时间} = \frac{\text{各作业周转时间之和}}{\text{作业数}}$$

- 公式：

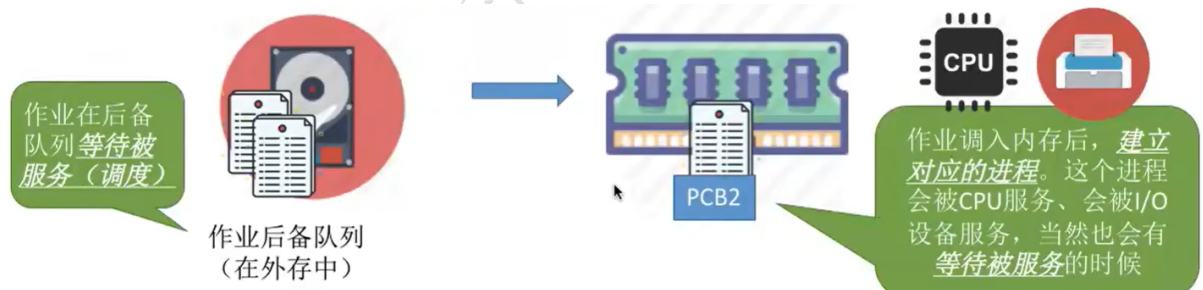
$$\text{带权周转时间} = \frac{\text{作业周转时间}}{\text{作业实际运行的时间}} = \frac{\text{作业完成时间}-\text{作业提交时间}}{\text{作业实际运行的时间}}$$

- 公式：

$$\text{平均带权周转时间} = \frac{\text{各作业带权周转时间之和}}{\text{作业数}}$$

概念：等待时间

- 概念：
 - 指进程/作业处于 **等待 CPU 的时间之和**，其不包含运行的时间。等待时间越长，用户满意度越低；
 - CPU 调度算法实际上并不影响作业执行或 I/O 操作的时间，只影响作业在就绪队列中等待所花的时间。因此，衡量一个调度算法的优劣，常常只需简单地考察等待时间；
- 图示：



对于进程来说，等待时间就是指进程建立后等待被服务的时间之和，在等待I/O完成的期间其实进程也是在被服务的，所以不计入等待时间。

对于作业来说，不仅要考虑建立进程后的等待时间，还要加上作业在外存后备队列中等待的时间。

一个作业总共需要被CPU服务多久，被I/O设备服务多久一般是确定不变的，因此调度算法其实只会

- 影响作业/进程的等待时间。当然，与前面指标类似，也有“**平均等待时间**”来评价整体性能。

概念：响应时间

- 概念：指从用户提交请求到首次产生响应所用的时间；

概念：响应比

- 公式：

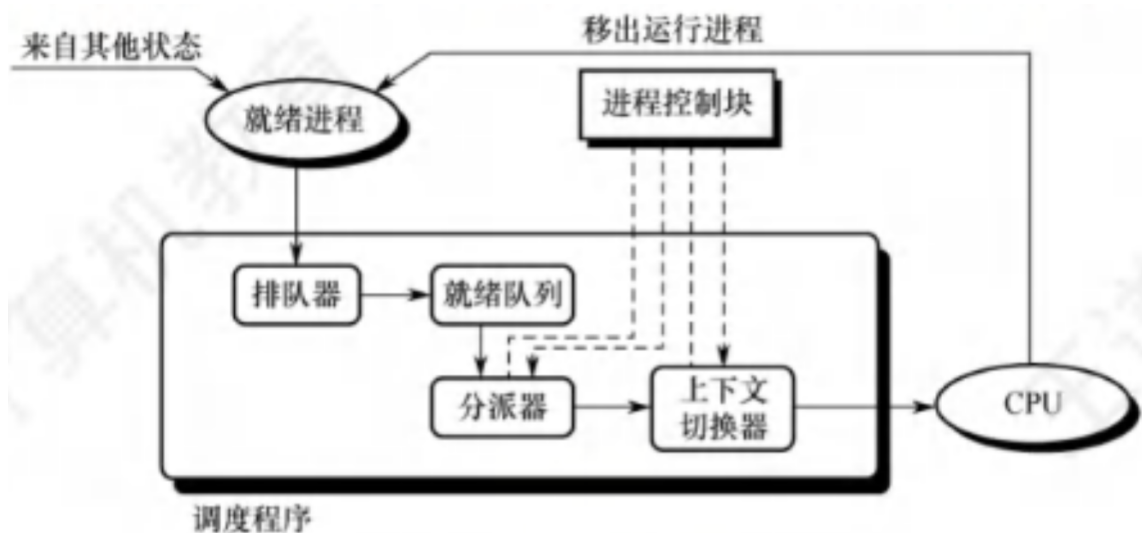
$$\text{响应比} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$$

调度的实现

调度程序

概念：调度程序

- 概念：
 - 用于调度和分派 CPU 的组件称为调度程序，它通常由三部分组成；
 - 调度程序是一个模块，用来将 CPU 控制交给由短期调度程序选择的进程；
- 组成：
 - (1) 排队器：
 - 将系统中的所有就绪进程按照一定的策略排成一个或多个队列，以便于调度程序选择；
 - 每当有一个进程转变为就绪态时，排队器便将它插入相应的就绪队列；
 - (2) 分派器：
 - 依据调度程序所选的进程，将其从就绪队列中取出，将 CPU 分配给新进程；
 - (3) 上下文切换器：
 - 在对 CPU 进行切换时，会发生两对上下文的切换操作：
 - 第一对：
 - 将当前进程的上下文保存到其 PCB 中，再装入分派程序的上下文，以便分派程序运行；
 - 第二：
 - 移出分派程序的上下文，将新选进程的 CPU 现场信息装入 CPU 的各个相应寄存器；
 - 补充：定时器
 - 真题：2018年
 - 定时器用于产生时钟中断，当时间片使用完后，需要产生时钟中断；
 - 时钟中断需要更新以下内容：
 - (1) 内核中时钟变量的值；
 - (2) 当前进程占用 CPU 的时间；
 - (3) 当前进程在时间片内剩余的执行时间；
 - 图示：
 - 调度程序结构



闲逛进程

概念： 闲逛进程

- 概念：
 - 调度程序永远的备胎，没有其他就绪进程时，运行闲逛进程（idle）；
- 特点：
 - 1. 优先级最低；
 - 2. 可以是 0 地址指令，占一个完整的指令周期（指令周期末尾例行检查中断）；
 - 3. 能耗低；

面试题： 什么是僵尸进程

- 僵尸进程是已完成且处于终止状态，但在进程表中却仍然存在的进程；
- 僵尸进程一般发生在有父子关系的进程中，一个子进程的进程描述符 PCB 在子进程退出时不会释放，只有当父进程通过 `wait()` 或 `waitpid()` 获取了子进程信息后才会释放；
- 如果子进程退出，而父进程并没有调用 `wait()` 或 `waitpid()`，那么子进程的进程描述符仍然保存在系统中；

面试题： 什么是孤儿进程

- 一个父进程退出，而它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程；
- 孤儿进程将被 `init` 进程（进程 ID 为 1 的进程）所收养，并由 `init` 进程对它们完成状态收集工作；
- 因为孤儿进程会被 `init` 进程收养，所以孤儿进程不会对系统造成危害。

调度的时机

概念： 可以进行调度的情况

- 真题：2021年
- (1) 创建新进程：
 - 创建新进程后，由于父进程和子进程都处于就绪态，因此需要决定是运行父进程还是运行子进程，调度程序可以合法地决定其中一个进程先运行；

- (2) 进程正常或异常结束：
 - 进程正常结束后或者异常终止后，必须从就绪队列中选择某个进程运行；
 - 若没有就绪进程，则通常运行一个系统提供的**闲逛进程**；
- (3) 申请 IO 时：
 - 当进程被自己的 I/O 请求、信号量操作或其他原因而被阻塞时，必须调度其他进程运行；
- (4) IO 完成时：
 - 当其他的 IO 设备完成后，发出 I/O 中断，原先等待 IO 的进程从阻塞态变为就绪态，此时需要决定是让新的就绪进程投入运行，还是让中断发生时运行的进程继续执行；
- (5) 更高优先级任务：
 - 当有更紧急的任务 (如更高优先级的进程进入就绪队列) 需要处理时，被强行剥夺 CPU (被动放弃)；
- (6) 中断处理结束；
- (7) 进程的时间片用完；
 - 当前进程的时间片用完时，被强行剥夺 CPU (被动放弃)

概念：不能进行进程切换的情况

- 三种情况：
 - (1) 处理中断时：
 - 在**处理中断**的过程中。中断处理过程复杂，与硬件密切相关，很难做到在中断处理过程中进行进程切换；
 - (2) 内核临界区：
 - 进程在操作系统**内核程序临界区**中；
 - (3) 原语：
 - 在原子操作过程中 (**原语**) ；
 - 原子操作不可中断，要一气呵成 (如修改 PCB 中进程状态标志，并把 PCB 放到相应队列) ；

• **真题：**2012年

• 题目：

【2012 统考真题】若某单处理器多进程系统中有多就就绪态进程，则下列关于处理机调度的叙述中，错误的是 ()。

- A. 在进程结束时能进行处理机调度
- B. 创建新进程后能进行处理机调度
- C. 在进程处于临界区时不能进行处理机调度
- D. 在系统调用完成并返回用户态时能进行处理机调度

• 解析

- A 和 B 显然可以进行调度；
- D 选项：一般来说，处理机调度是在内核态完成的。当系统调用执行结束返回用户态时，处理机会继续运行先前被调度的进程，直到该进程重新陷入内核态或者被其他更高优先级的进程抢占 CPU。这个过程是由处理机硬件支持的，即硬件会在每个时钟周期结束后检查是否需要进行调度；

- C 选项：若 C 选项进程所处的临界区不是操作系统**内核程序临界区**，那么 C 可以进行处理机调度，因此 C 错误；

概念：临界资源

- 一个时间段内只允许一个进程使用的资源；
- 各进程需要互斥地访问临界资源；

概念：临界区

- 概念：
 - 访问临界资源的那段代码；
- 分类：
 - 1. 普通临界区；
 - 2. 内核临界区；
 - 内核程序临界区：一般是用来访问某种内核数据结构的，比如进程的就绪队列（由各就绪进程的 PCB 组成）
- 注意：
 - 1. 内核临界区内访问的资源如果长时间得不到释放，影响其他工作；
 - 2. 因此处于**临界区的进程可以被中断或者抢占进而被调度**。比如请求一个打印机资源时，可以让它放弃 CPU，而让其他进程运行；

调度的方式

概念：非剥夺调度方式

- 概念：
 - 又称**非抢占**方式：即只允许**进程主动**放弃处理机。
 - 在运行过程中即便有更紧迫的任务到达，当前进程依然会继续使用处理机，直到该进程终止或主动要求进入阻塞态；
- 特点：
 - 实现简单开销小，但是无法及时处理紧急任务，适合于早期的批处理系统
- 举例：
 - 非抢占式调度在有些硬件平台上是唯一的方法，因为它不需要特殊硬件（如定时器）来支持抢占调度；

概念：剥夺调度方式

- 概念：
 - 又称**抢占**方式；
 - 当一个进程正在处理机上执行时，如果有一个更重要或更紧迫的进程需要使用处理机，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；
- 特点：
 - 优点：
 - 可以优先处理更紧急的进程，也可实现让各进程按时间片轮流执行的功能（通过时钟中断）；

- 缺点：
 - 抢占的功能会影响操作系统的内核设计；
 - 在处理系统调用时，内核可能为进程而忙于某个活动，这些活动可能涉及改变重要的内核数据；
- 举例：
 - Windows 95 引入抢占调度，所有之后的 Windows 操作系统都使用了抢占调度；
 - Macintosh 操作系统 MacOSX 采用抢占调度，而之前的 Macintosh 操作系统采用协作调度；

用户级线程调度与内核级线程调度

概念：用户级线程调度与内核级线程调度

- 概念：用户级线程调度
 - 由于内核并不知道线程的存在，所以内核还是和以前一样，选择一个进程，并给予时间控制；
 - 由**进程中的调度程序**决定哪个线程运行。
- 概念：内核级线程调度
 - 内核选择一个特定线程运行，通常**不用考虑该线程属于哪个进程**；
 - 对被选择的线程赋予一个时间片，如果超过了时间片，就会强制挂起该线程；
- 区别：
 - 用户级线程的线程切换在同一进程中进行，仅需少量的机器指令；
 - 内核级线程的线程切换需要完整的上下文切换、修改内存映像、使高速缓存失效，这就导致了若干数量级的延迟；

典型调度算法

面试题：介绍常见的进程调度算法

- 主要分为两大类：
- 第一类：非交互式
 - 第一种是 FCFS 先来先服务算法，先请求 CPU 的进程首先分配到 CPU，按照作业/进程到达的先后顺序进行服务。其优点是公平，并且算法实现简单，不会导致饥饿。但问题是用户体验不好：排在长作业（进程）后面的短作业需要很长时间，带权周转时间很大，对短作业来说用户体验不好；
 - 第二种是短作业优先算法，**最短**的作业/进程优先得到服务（所谓**最短**，是指要求服务时间最短）。当 CPU 变为空闲时，它会被赋给具有最短 CPU 执行的进程，如果两个进程具有同样长度的 CPU 执行，那么可以由 FCFS 来处理。其优点是大部分情况下是**最短**的平均等待时间、平均周转时间，但缺点是可能会导致饥饿，并且作业/进程的运行时间是由用户提供的，并不一定真实（用户故意提交大量短作业），不一定能做到真正的短作业优先；
 - 第三种是高响应比优先算法，此算法综合考虑作业/进程的**等待时间和要求服务的时间**，在每次调度时先计算各个作业/进程的响应比，选择响应比最高的作业/进程为其服务。
- 第二类：交互式

- 第一种是时间片轮转算法，公平、轮流地为各个进程服务，让每个进程在**一定时间间隔内**都可以得到响应，使得每个用户可以和系统交互，相比于非交互式算法，改善了人机环境。优点是公平，并且响应快，适用于分时操作系统。缺点是开销大，由于高频率的进程切换，因此有一定开销，并且不区分进程的紧急程度；
- 第二种是优先级调度算法，每个作业/进程有各自的优先级，调度时选择优先级最高的作业/进程。缺点是若源源不断地有高优先级进程到来，则可能导致饥饿（无穷阻塞）；
- 第三种是多级队列调度算法，将**就绪队列分成多个单独队列**，系统中按**进程类型**设置多个队列，进程创建成功后插入某个队列。每个队列有自己的调度算法，比如：前台队列（交互式）可以采用 **RR** 算法调度，而后台队列（批处理）可以采用 **FCFS** 算法调度。队列之间应有调度，通常采用固定优先级抢占调度；

典型调度算法：非交互式

先到先得：FCFS 算法

概念：先来先服务 - **FCFS** (First Come First Serve)

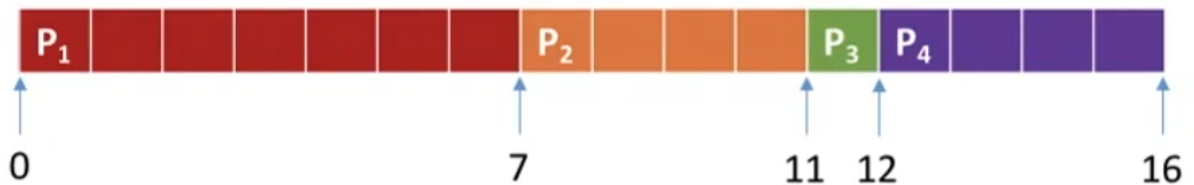
- 思想
 - 主要从**公平**的角度考虑（类型于我们生活中排队买东西）
- 规则
 - 先请求 CPU 的进程首先分配到 CPU，按照作业/进程到达的先后顺序进行服务；
- 应用：
 - 用于**作业调度**时，考虑的是哪个作业先到达后备队列；
 - 用于**进程调度**时，考虑的是哪个进程先到达就绪队列；
- 方式：
 - 一般是非抢占式的；
- 特点：
 - 优点：
 - 1. 公平；
 - 2. 算法实现简单；
 - 缺点：
 - 1. 用户体验不好：排在长作业（进程）后面的短作业需要很长时间，带权周转时间很大，对短作业来说用户体验不好；
 - 2. 对长作业有利，对短作业不利；
 - 3. FCFS 策略的平均等待时间通常不是最小；
 - 不会导致饥饿；

举例：FCFS 算法举例

- 进程图：

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

- 调度图（甘特图）：



- 性能计算：

周转时间 = 完成时间 - 到达时间

$P1=7-0=7$; $P2=11-2=9$; $P3=12-4=8$; $P4=16-5=11$

- 带权周转时间 = 周转时间 / 运行时间

$P1=7/7=1$; $P2=9/4=2.25$; $P3=8/1=8$; $P4=11/4=2.75$

- 注意：

- 本例中的进程都是纯计算型的进程，一个进程到达后要么在等待，要么在运行；
- 如果是又有计算、又有 IO 操作的进程，其等待时间就是 周转时间 - 运行时间 - IO 操作的时间；

短作业：SJF 算法

概念：短作业优先 - SJF (Shortest Job First)

- 别名：

- 另外一种名字是**最短下次 CPU 执行** (shortest-next-CPU-burst) 算法；

- 思想

- 追求最少的**平均等待时间**，最少的平均周转时间，最少的平均带权周转时间

- 规则：

- **最短**的作业/进程优先得到服务（所谓**最短**，是指要求服务时间最短）
- 当 CPU 变为空闲时，它会被赋给具有最短 CPU 执行的进程；
- 如果两个进程具有同样长度的 CPU 执行，那么可以由 FCFS 来处理；

- 应用：

- 即可用于作业调度，也可用于进程调度；
- 用于进程调度时称为 **短进程优先** (SPF, Shortest Process First) 算法。

- 抢占：

- SJF 和 SPF 是非抢占式的算法；
- 有抢占式的版本：最短剩余时间优先算法 (SRTN, Shortest Remaining Time Next)

- 特点：

- 优点：
 - 1. 大部分情况下是**最短**的平均等待时间、平均周转时间；
 - 2. SJF 调度经常用于**批处理下的长期调度**，因为在这种系统下，用户有意精确估计进程时间；
- 缺点：
 - 1. 不公平：对短作业有利，对长作业不利；
 - 2. 可能产生饥饿现象；
 - 3. 作业/进程的运行时间是由用户提供的，并不一定真实（用户故意提交大量短作业），不一定能做到真正的短作业优先；
 - 4. 不能在短期 CPU 调度级别上加以实现：难以知道下次 CPU 执行的长度；
- 饥饿：
 - 会
 - 如果源源不断地有短作业/进程到来，可能使长作业/进程长时间得不到服务，产生饥饿现象；
 - 如果有进程**一直得不到服务，则称为饿死**；
- 结论：
 - 在所有进程都**几乎同时到达时**，采用 SJF 调度算法的平均等待时间、平均周转时间最少；
 - SJF 依然可以获得较少的平均等待时间、平均周转时间；

举例：短作业/进程优先调度算法

- 进程图：

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

- 调度图：



- 性能计算：

周转时间 = 完成时间 - 到达时间

$$P1=7-0=7; P3=8-4=4; P2=12-2=10; P4=16-5=11$$

带权周转时间 = 周转时间/运行时间

$$P1=7/7=1; P3=4/1=4; P2=10/4=2.5; P4=11/4=2.75$$

- 等待时间 = 周转时间 - 运行时间

$$P1=7-7=0; P3=4-1=3; P2=10-4=6; P4=11-4=7$$

最短剩余时间：SRTN 算法

概念：最短剩余时间优先算法 - SRTN 算法

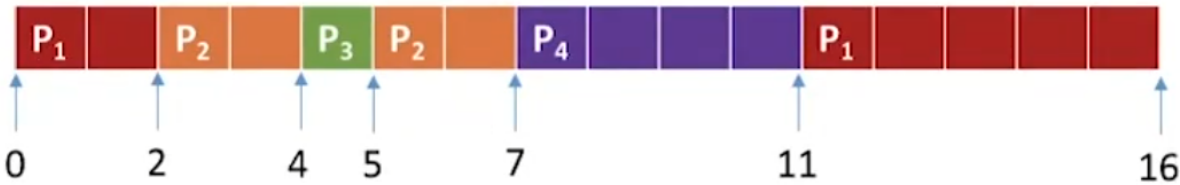
- 概念：
 - 每当有进程加入就绪队列改变时就需要调度，如果新到达的进程剩余时间比当前运行的进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。
 - 另外，当一个进程完成时也需要调度；
- 时机：
 - 1. 每当有进程加入就绪队列改变时就需要调度；
 - 2. 如果新到达的进程剩余时间比当前运行的进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列；
 - 3. 当一个进程完成时也需要调度；
- 抢占：
 - 是**抢占式**的算法；
- 特点：
 - 平均周转时间最短；

举例：最短剩余时间优先算法

进程图：

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

调度图：



运行：

- 0时刻（P1到达）：P₁（7）
- 2时刻（P2到达）：P₁（5）、P₂（4）
- 4时刻（P3到达）：P₁（5）、P₂（2）、P₃（1）
- 5时刻（P3完成且P4刚好到达）：P₁（5）、P₂（2）、P₄（4）
- 7时刻（P2完成）：P₁（5）、P₄（4）

高响应比：HRRN 算法

概念：高响应比优先 - HRRN (Highest Response Ratio Next)

- 思想：
 - 要综合考虑作业/进程等待时间和要求服务的时间
- 规则：
 - 在每次调度时先计算各个作业/进程的响应比，选择响应比最高的作业/进程为其服务
- 应用：
 - 即可用于作业调度，也可用于进程调度
- 抢占：
 - 非抢占式**的算法；
 - 因此只有当前运行的作业/进程主动放弃 CPU 时，才需要调度，才需要计算响应比
- 特点：
 - 综合考虑了等待时间和运行时间（要求服务时间）
 - 公式包含了等待时间和要求的服务时间： $\text{响应比} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$
 - 等待时间相同时，要求服务时间短的优先（SJF 的优点）
 - 公式中，要求服务时间变大时，结果越趋向于 1；
 - 要求服务时间特别短，则此时其响应比非常大，优先被调度；
 - $\text{响应比} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$
 - 要求服务时间相同时，等待时间长的优先（FCFS 的优点）
 - 由公式可知， $\text{响应比} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$ 中服务时间相同时、分子中等待时间大的、响应比更大；
 - 对于长作业来说，随着等待时间越来越久，其响应比也会越来越大，从而避免了长作业饥饿的问题；
- 饥饿：
 - 不会
 - 真题：**2011年
 - 非抢占式 SJF 和 HRRN 算法哪一个不会导致饥饿？原因是什么？
 - HRRN 一定不会导致饥饿；
 - SJF 考虑了任务的长短，但短进程优先有导致饥饿现象的风险，倘若就绪队列中一直有短任务进程进入，则长任务进程会一直等待，因此有饥饿的风险；
 - HRRN 中，短任务计算优先级时因为分母更小，会比长任务优先，但长进程只要等待的时间足够长，一定会比短任务优先级高，故不会发生饥饿

举例：HRRN 算法

- 规则总结：
 - 只有当前运行的进程主动放弃 CPU 时（正常/异常完成，或主动阻塞），才需要进行调度；
 - 调度时计算所有就绪进程的响应比，选响应比最高的进程上处理机；
- 进程图：

进程	到达时间	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

• 运行过程：

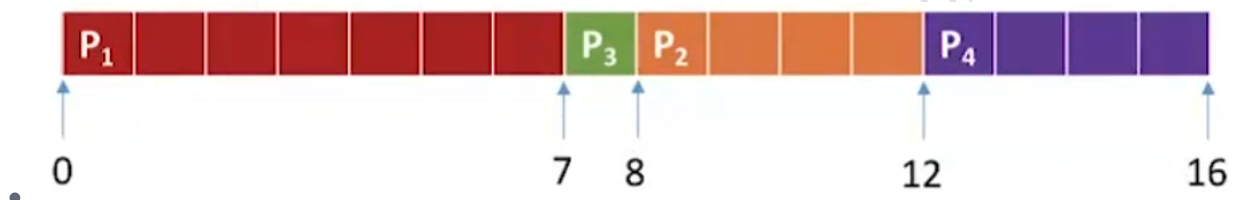
0时刻：只有 P_1 到达就绪队列， P_1 上处理机

7时刻（ P_1 主动放弃CPU）：就绪队列中有 P_2 (响应比 $= (5+4)/4=2.25$)、 P_3 ($((3+1)/1=4)$)、 P_4 ($((2+4)/4=1.5)$)，

8时刻（ P_3 完成）： P_2 (2.5)、 P_4 (1.75)

- 12时刻（ P_2 完成）：就绪队列中只剩下 P_4

• 调度图：



非交互式算法比较

概念：三种非交互式算法比较

• 图示：

算法	思想&规则	可抢占?	优点	缺点	考虑到等待时间&运行时间?	会导致饥饿?
FCFS	自己回忆	非抢占式	公平；实现简单	对短作业不利	等待时间 $\vee$ 运行时间 $\times$	不会
SJF/S PF	自己回忆	默认为非抢占式，也有SJF的抢占式版本最短剩余时间优先算法（SRTN）	“最短的”平均等待/周转时间；	对长作业不利，可能导致饥饿；难以做到真正的短作业优先	等待时间 $\times$ 运行时间 $\vee$	会
HRRN	自己回忆	非抢占式	上述两种算法的权衡折中，综合考虑的等待时间和运行时间		等待时间 $\vee$ 运行时间 $\vee$	不会

• 结论：

- 这三种算法交互性糟糕，不区分紧急程度，一般只适用于早起的批处理系统；

概念：计算规则总结

- **FCFS**：先到先进行；
- **SJF**：服务时间最短的先进行；
- **SRTN**：最短剩余时间的进程抢占进行；
- **HRRN**：最高响应比的进行；
 - 响应比 = $\frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$

典型调度算法：交互式

RR 算法

概念：时间片轮转 - RR (Round-Robin)

- 思想：
 - 公平、轮流地为各个进程服务，让每个进程在**一定时间间隔内**都可以得到响应；
 - 使得每个用户可以和系统交互，相比于非交互式算法，改善了人机环境；
- 规则：
 - 按照各进程到达就绪队列的顺序，**轮流让各个进程执行一个时间片**（如 100 ms）；
 - 若进程未在一个时间片内执行完，则剥夺处理机，将进程重新放到**就绪队列尾**重新排队；
 - 真题：2020年
 - RR 算法不涉及到进程阻塞队列，会涉及到：（1）进程控制块（2）时钟中断处理程序（3）进程就绪队列；
 - 就绪队列作为循环队列，CPU 调度程序循环整个就绪队列，为每个进程分配不超过一个时间片的 CPU；
 - RR 算法的性能很大程度取决于时间片的大小；
- 应用：
 - 主要用于**进程调度**（只有作业放入内存建立了相应的进程后，才能被分配处理机片）；
- 抢占：
 - 是；
 - 若进程未能在时间片内运行完，将被强行剥夺处理机使用权，因此时间片轮转调度算法属于抢占式的算法；
 - 由时钟装置发出**时钟中断**来通知 CPU 时间片已到；
- 特点：
 - 优点：
 - 1. 公平；
 - 2. 响应快：适用于分时操作系统
 - 缺点：
 - 1. 开销大：由于高频率的进程切换，因此有一定开销；
 - 2. 不区分任务的紧急程度；
- 饥饿：
 - 不会
- 注意：
 - RR 可以满足用户交互的交互系统；
 - 但实时系统还要求对**事件的处理**具有优先级，此时 RR 不能应对，需要使用带有优先级的调度算法；
 - 真题：2009年
 - 时间片轮转调度算法将处理机时间划分成固定长度的时间片，考虑了进程的执行时间而未考虑等待时间；

概念：关于时间片的大小

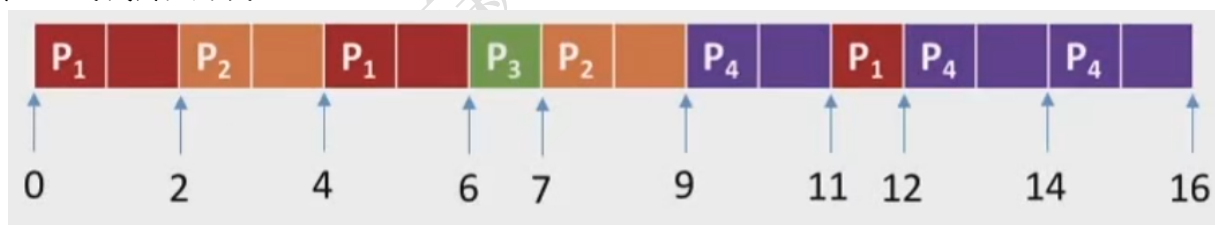
- (1) 时间片过大
 - 时间片太大，使得每个进程都可以在一个时间片内就完成，则时间片轮转调度算法退化为先来先服务调度算法，并且会增大进程响应时间；
 - 因此时间片不能太大；
- (2) 时间片过小
 - 进程调度、切换是有时间代价的（保存、恢复运行环境），因此如果时间片太小，会导致进程切换过于频繁，系统会花大量的时间来处理进程切换，从而导致实际用于进程执行的时间比例减少；
 - 可见时间片也不能太小（一般来说，设计时间片时要让切换进程的开销占比不超过1%）；

举例：RR 算法

- 进程图：

进程	到达时间	运行时间
P1	0	5
P2	2	4
P3	4	1
P4	5	6

- 调度图：时间片大小为 2



- 运行过程：
 - 0 时刻（P1 (5)）：0 时刻只有 P1 到达就绪队列，让 P1 上处理机运行一个时间片
 - 2 时刻（P2 (4) → P1 (3)）：2 时刻 P2 到达就绪队列，P1 运行完一个时间片，被剥夺处理机，重新放到队尾
 - 4 时刻（P1 (3) → P3 (1) → P2 (2)）：4 时刻，P3 到达，先插到就绪队尾，紧接着，P2 下处理机也插到队尾
 - 6 时刻（P3 (1) → P2 (2) → P4 (6) → P1 (1)）：6 时刻，P1 时间片用完，下处理机，重新放回就绪队尾，发生调度
 - 7 时刻（P2 (2) → P4 (6) P1 (1)）：虽然 P3 的时间片没用完，但是由于 P3 只需运行 1 个单位的时间，运行完了会主动放弃处理机，因此也会发生调度。队头进程 P2 上处理机。

优先级调度算法

概念：优先级调度 - Priority-Scheduling

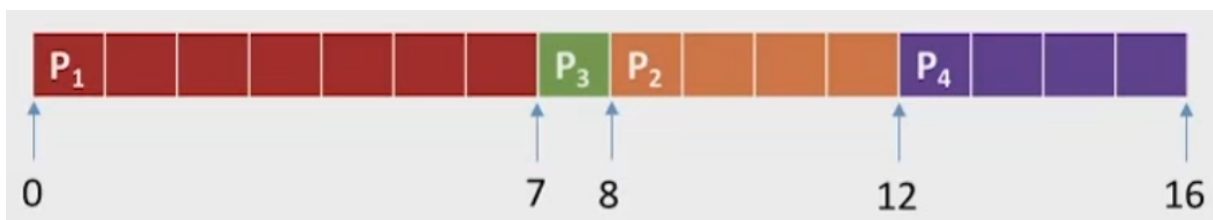
- 思想：
 - 需要根据任务的紧急程度来决定处理顺序
- 规则：
 - 每个作业/进程有各自的优先级，调度时选择优先级最高的作业/进程
- 应用：
 - 既可用于作业调度，也可用于进程调度
 - 甚至用于 I/O 调度
- 抢占：
 - 抢占式、非抢占式都有；
 - 区别：
 - 非抢占式：只需在**进程主动**方式处理机时进行调度即可
 - 抢占式：需在**就绪队列变化**时，检查是否会发生抢占；
- 特点：
 - 优点：
 - 用优先级区分紧急程度，重要程度，适用于实时操作系统。
 - 可灵活地调整对各种作业/进程的偏好程度。
 - 缺点：
 - 若源源不断地有高优先级进程到来，则可能导致饥饿（无穷阻塞）
 - 据说，在 1973 年关闭 MIT 的 IBM 7094 时，发现有一个低优先级进程早在 1967 年就已提交，但是一直未能运行
 - 解决方法 -> 老化：逐渐增加在系统中等待很长时间的进程的优先级；
- 饥饿：
 - 会

举例：优先级调度 - 非抢占式

- 进程图：
 - 优先数越高，优先级越高

进程	到达时间	运行时间	优先数
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2

- 调度图：



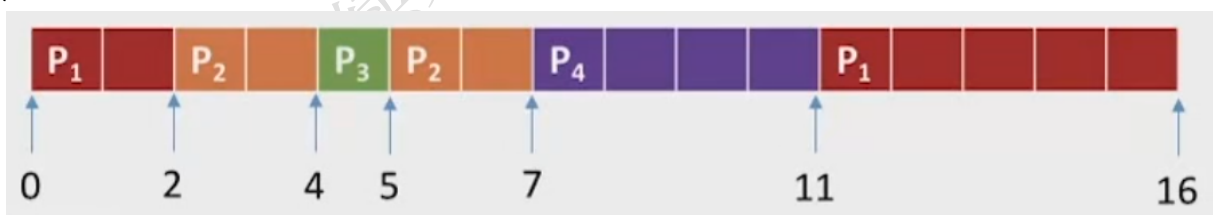
- 运行过程：
 - 0 时刻（P 1）：只有 P 1 到达，P 1 上处理机；
 - 7 时刻（P 2、P 3、P 4）：P 1 运行完成主动放弃处理机，其余进程都已到达，P 3 优先级最高，P 3 上处理机；
 - 8 时刻（P 2、P 4）：P 3 完成，P 2、P 4 优先级相同，由于 P 2 先到达，因此 P 2 优先上处理机；
 - 12 时刻（P 4）：P 2 完成，就绪队列只剩 P 4，P 4 上处理机；

举例：优先级调度 - 抢占式

- 进程图：

进程	到达时间	运行时间	优先数
P1	0	7	1
P2	2	4	2
P3	4	1	3
P4	5	4	2

- 调度图：



多级队列调度算法

概念：多级队列调度算法

- 概念：
 - 将**就绪队列分成多个单独队列**，系统中按**进程类型**设置多个队列，进程创建成功后插入某个队列；
 - 每个队列有自己的调度算法，比如：前台队列（交互式）可以采用 **RR** 算法调度，而后台队列（批处理）可以采用 **FCFS** 算法调度；
 - 队列之间应有调度，通常采用固定优先级抢占调度；
- 特点：
 - 队列之间可采取**固定优先级**或**时间片**划分；
 - 固定优先级：高优先级空时低优先级进程才能被调度；

- 时间片划分：如三个队列分配时间 50%、40%、10%；
- 各队列可采用不同的调度策略，如：系统进程队列采用优先级调度、交互式队列采用 RR、批处理队列采用 FCFS；
- 图示：三类进程的优先级
 - 系统进程：优先级最高；
 - 交互式进程：需要及时交互，所以设定为交互式进程；
 - 批处理进程；

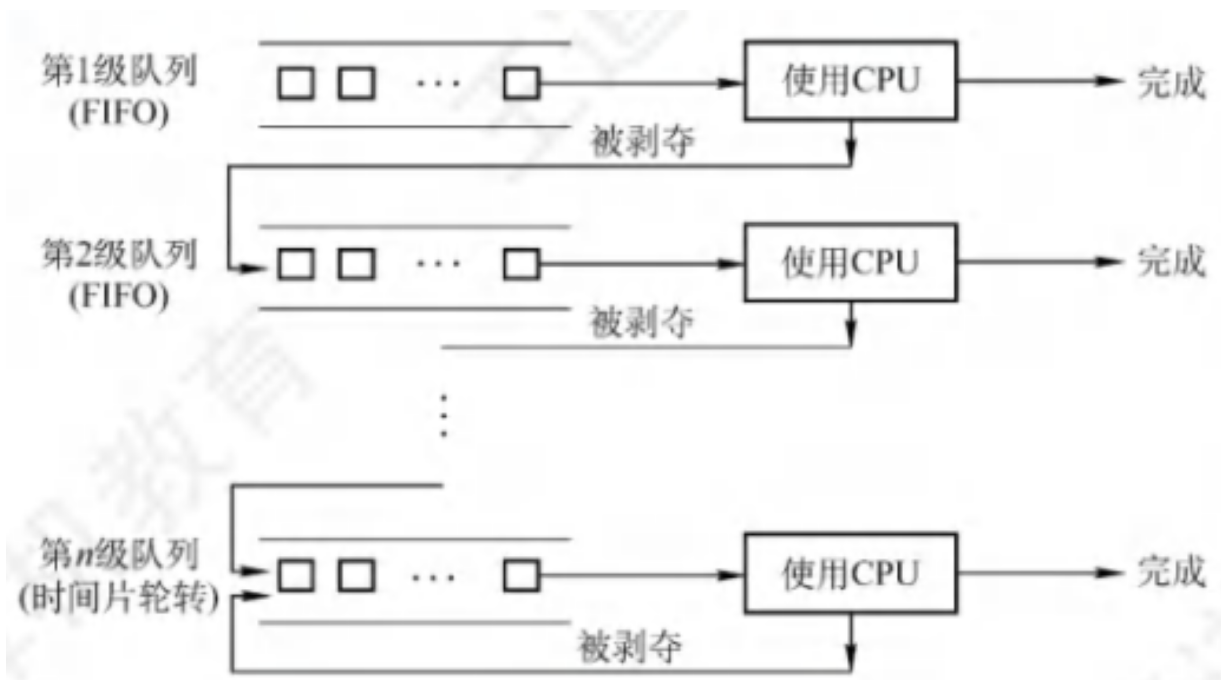


多级反馈队列调度算法

概念：多级反馈队列调度算法

- 思想：
 - 对其他调度算法的折中权衡（尤其是对多级队列调度算法的改进）；
 - 允许进程在不同优先级队列之间迁移，类似 IO 密集型进程以及交互进程可以更好的在 high 优先级队列上被处理；
 - 或者在 **较低优先级** 队列中等待过长的进程会被 **移到更高优先级** 队列；
- 规则：
 1. 设置多级就绪队列，各级队列优先级从高到低，时间片从小到大。
 2. 每个队列都采用 **FCFS** 算法：
 - 新进程进入内存后，新进程到达时先进入 **第 1 级** 队列，按 **FCFS** 原则排队等待被分配时间片；
 - 当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统；
 - 若它在一个时间片结束时尚未完成，调度程序将其转入第 2 级队列的末尾等待调度；
 - 若它在第 2 级队列中运行一个时间片后仍未完成，再将它放入第 3 级队列，以此类推；
 - 当进程最后被降到第 **n** 级队列后，在第 **n** 级队列中便采用时间片轮转方式运行；
 3. 按队列优先级调度：
 - 仅当第 **1** 级队列为空时，才调度第 2 级队列中的进程运行；
 - 仅当第 **1~i-1** 级队列均为空时，才会调度第 **i** 级队列中的进程运行；

- 若 CPU 正在执行第 i 级队列中的某个进程时，又有新进程进入任何一个优先级较高的队列，此时须立即将正在运行的进程放回到第 i 级队列的末尾，而将 CPU 分配给新到的高优先级进程；
- 4. 赋予各个队列的进程运行时间片的大小各不相同：在优先级越高的队列中，每个进程的时间片就越小；
 - 例如，第 $i+1$ 级队列的时间片要比第 i 级队列的时间片长 1 倍；
- 举例：
 - 每个进程在进入就绪队列后，就被添加到队列 0 内；
 - 队列 0 内的每个进程都有 8 ms 的时间片，如果一个进程不能在这一时间片内完成，那么它就被移到队列 1 的尾部；
 - 如果队列 0 为空，队列 1 头部的进程会得到一个 16 ms 的时间片；
 - 如果它不能完成，那么将被抢占，并添加到队列 2；
 - 只有当队列 0 和 1 为空时，队列 2 内的进程才可根据 FCFS 来运行；
- 抢占：
 - 抢占式的算法；
 - 在 k 级队列的进程运行过程中，若更上级的队列（ $1 \sim k-1$ 级）中进入了一个新进程，则由于新进程处于优先级更高的队列中，因此新进程会抢占处理机，原来运行的进程放回 k 级队列队尾。
- 特点：
 - 1. 对各类型进程相对公平（FCFS 的优点）；
 - 2. 每个新到达的进程都可以很快就得到响应（RR 的优点）；
 - 3. 短进程只用较少的时间就可完成（SPF 的优点）；
 - 4. 不必实现估计进程的运行时间（避免用户作假）；
 - 5. 可灵活地调整对各类进程的偏好程度，比如 CPU 密集型进程，I/O 密集型进程；
- 饥饿：
 - 会
 - 如果一直有短进程进来，则可能会导致原本的长进程一直得不到时间片；
- 扩展：
 - 可以将因 I/O 而阻塞的进程重新放回原队列，这样 I/O 型进程就可以保持高优先级；
- 图示：



分析：二级反馈队列调度算法

真题：2019年

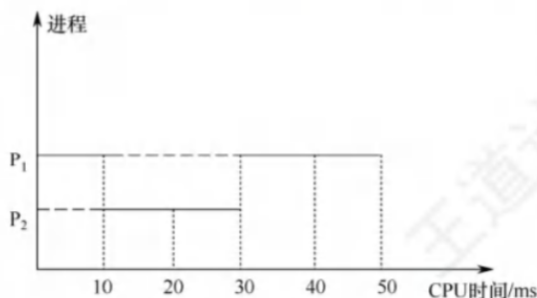
【2019 统考真题】系统采用二级反馈队列调度算法进行进程调度。就绪队列 Q_1 采用时间片轮转调度算法，时间片为 10ms；就绪队列 Q_2 采用短进程优先调度算法；系统优先调度 Q_1 队列中的进程，当 Q_1 为空时系统才会调度 Q_2 中的进程；新创建的进程首先进入 Q_1 ； Q_1 中的进程执行一个时间片后，若未结束，则转入 Q_2 。若当前 Q_1 、 Q_2 为空，系统依次创建进程 P_1 、 P_2 后即开始进程调度， P_1 、 P_2 需要的 CPU 时间分别为 30ms 和 20ms，则进程 P_1 、 P_2 在系统中的平均等待时间为（ ）。

- A. 25ms B. 20ms C. 15ms D. 10ms

解答：

- 画出甘特图，分析 CPU 执行时间轴；

进程 P_1 、 P_2 依次创建后进入队列 Q_1 ，根据时间片调度算法的规则，进程 P_1 、 P_2 将依次被分配 10ms 的 CPU 时间，两个进程分别执行完一个时间片后都会被转入队列 Q_2 ，就绪队列 Q_2 采用短进程优先调度算法，此时 P_1 还需要 20ms 的 CPU 时间， P_2 还需要 10ms 的 CPU 时间，所以 P_2 会被优先调度执行，10ms 后进程 P_2 执行完成，之后 P_1 再调度执行，再过 20ms 后 P_1 也执行完成。运行图表述如下。



进程 P_1 、 P_2 的等待时间分别为图中的虚横线部分，平均等待时间 = (P_1 的等待时间 + P_2 的

- 等待时间)/2 = (20 + 10)/2 = 15，因此答案选 C。

举例：多级反馈队列调度算法

规则总结：

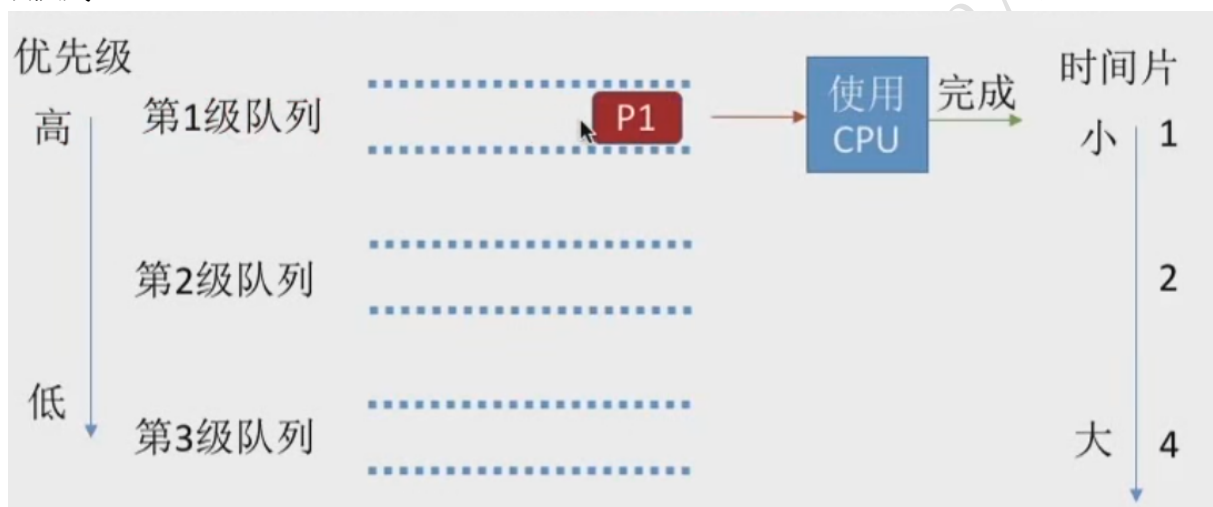
- 设置多级就绪队列，各级队列优先级从高到低，时间片从小到大；
- 新进程到达时先进入第 1 级队列，按 FCFS 原则排队等待被分配时间片；

- 若用完时间片进程还未结束，则进程进入下一级队列队尾。如果此时已经在最下级的队列，则重新放回最下级队列队尾；
- 只有**第 k 级队列为空时，才会为 k+1 级队头的进程分配时间片**；

• 进程图：

进程	到达时间	运行时间
P1	0	8
P2	1	4
P3	5	1

• 优先级队列：



交互式算法比较

概念：三类交互式算法比较

• 图示：

算法	思想&规则	可抢占?	优点	缺点	会导致饥饿?	补充
时间片轮转		抢占式	公平，适用于分时系统	频繁切换有开销，不区分优先级	不会	时间片太大或太小有何影响?
优先级调度		有抢占式的，也有非抢占式的。注意做题时的区别	区分优先级，适用于实时系统	可能导致饥饿	会	动态/静态优先级。各类型进程如何设置优先级? 如何调整优先级?
多级反馈队列	较复杂，注意理解	抢占式	平衡优秀 666	一般不说它有缺点，不过可能导致饥饿	会	

• 总结：

- 比起早期的批处理操作系统来说，由于计算机造价大幅降低，因此之后出现的交互式操作系统（包括分时操作系统、实时操作系统等）更注重系统的响应时间、公平性、平衡性等指标；
- 而这几种算法恰好也能较好地满足交互式系统的需求。因此这三种算法适合用于交互式系统。（比如 UNIX 使用的就是多级反馈队列调度算法）；

上下文及其切换机制

概念：上下文切换

- 概念：

- 切换 CPU 到另一个进程需要保存当前进程状态、并恢复另一个进程的状态，这个任务称为上下文切换；
- 进程上下文采用进程 PCB 表示，包括 CPU 寄存器的值、进程状态和内存管理信息等；
- 当进行上下文切换时，内核将旧进程状态保存在其 PCB 中，然后加载经调度而要执行的新进程的上下文；
- 在切换过程中，进程的运行环境产生实质性的变化；

- 实现：

- (1) 挂起 + 保存 PCB：

- 挂起一个进程，将 CPU 上下文保存到 PCB，包括程序计数器和其他寄存器；
- 真题：2012年
- 中断、上下文切换和子程序调用都需要压栈以保存现场，它们都需要保存程序计数器、通用寄存器的数值，但中断和上下文切换需要保存 PSW 程序状态寄存器，子程序调用时不需要保存 PSW；

- (2) 移入队列：

- 将进程的 PCB 移入相应的队列，如就绪、在某事件阻塞等队列；

- (3) 选择：

- 选择另一个进程执行，并更新其 PCB；

- (4) 载入上下文：

- 恢复新进程的 CPU 上下文；

- (5) 更改 PC：

- 跳转到新进程 PCB 中的程序计数器所指向的位置执行；

- 补充：xv6 下的进程上下文存储

- 在 CPU 的某个临时寄存器中存储了一个地址，此地址指向的内容把寄存器状态加载然后保存，接着进入到用户态的 OS 代码中；（进入到 usertrap() 当中）；
- 把进程的 trapframe 分配到固定地址，将 trapframe 保存在 sscratch 上，之后所有的进程的状态机都被存储到了内存当中，包括内存、寄存器等等。保存完毕后切换到内核线程执行；
- 打印了某进程的 trappoline 指针解引用后的内容：

```
(gdb) p/x *p
```

```
$12 = {lock = {locked = 0x0, name = 0x8000b1e0, cpu = 0x0}, state = 0x4,
      chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1, parent = 0x0,
      kstack = 0x3fffffd000, sz = 0x3000, pagetable = 0x87f6f000,
      trapframe = 0x87f77000, context = {ra = 0x8000332a, sp = 0x3fffffda60,
      s0 = 0x3fffffda90, s1 = 0x800146f8, s2 = 0x505050505050505,
      s3 = 0x505050505050505, s4 = 0x505050505050505, s5 = 0x505050505050505,
```

- 实现：

```

trap:
    cli
    subq $48, %rsp
    pushq %r15
    pushq %r14
    pushq %r13
    pushq %r12
    pushq %r11
    pushq %r10
    pushq %r9
    pushq %r8
    pushq %rdi
    pushq %rsi
    pushq %rbp
    pushq %rdx
    pushq %rcx
    pushq %rbx
    pushq %rax
    pushq $0 // cr3, saved in __am_irq_handle

    movq %rsp, %rdi
    . call __am_irq_handle

```

分析：模式切换与上下文切换

- 模式切换与上下文切换是不同的。模式切换时，CPU 逻辑上可能还在执行同一进程，用户进程最开始都运行在用户态，若进程因中断或异常进入核心态运行，执行完后又回到用户态刚被中断的进程运行；
- 用户态和内核态之间的切换称为模式切换，而不是上下文切换，因为没有改变当前的进程；
- 上下文切换只能发生在内核态，它是多任务操作系统中的一个必需的特性；

面试题：什么是进程上下文切换？

- (1) 挂起 + 保存 PCB：
 - 挂起一个进程，将 CPU 上下文保存到 PCB，包括程序计数器和其他寄存器；
 - 真题：2012年
 - 中断、上下文切换和子程序调用都需要压栈以保存现场，它们都需要保存程序计数器、通用寄存器的数值，但中断和上下文切换需要保存 PSW 程序状态寄存器，子程序调用时不需要保存 PSW；
- (2) 移入队列：
 - 将进程的 PCB 移入相应的队列，如就绪、在某事件阻塞等队列；
- (3) 选择：
 - 选择另一个进程执行，并更新其 PCB；
- (4) 载入上下文：
 - 恢复新进程的 CPU 上下文；

- (5) 更改 `PC` :
 - 跳转到新进程 `PCB` 中的程序计数器所指向的位置执行;

补充: 上下文切换时间的优化

- 在上下文切换时, 需要执行大量 `load` 和 `store` 指令, 以保存寄存器的内容, 因此会花费较多时间;
- 现在已有硬件实现的方法来减少上下文切换时间, 通常采用两组寄存器, 其中一组供内核使用, 一组供用户使用;
- 这样, 上下文切换时, 只需改变指针, 让其指向当前寄存器组即可;

多处理器调度

基本概念

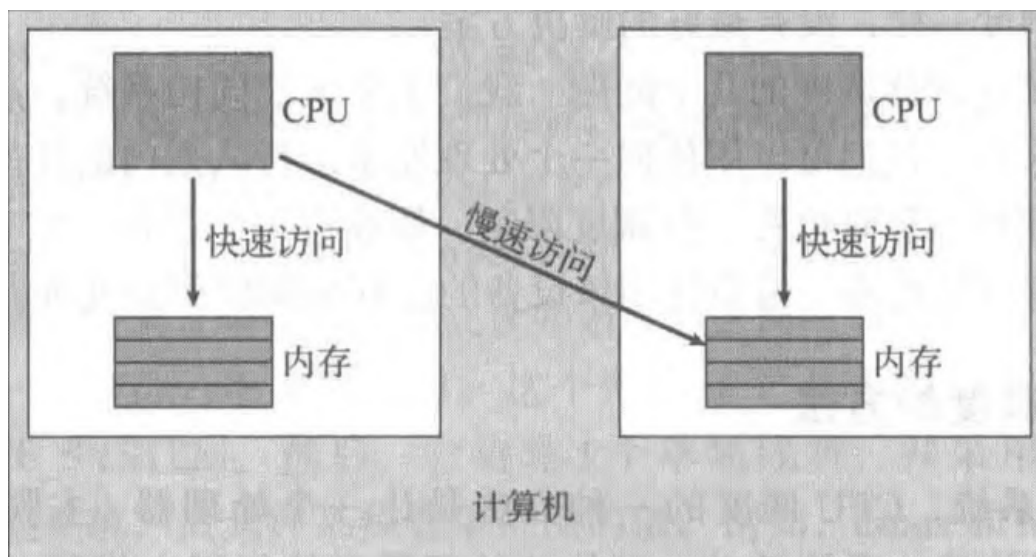
概念: 两种处理方式

- 非对称多处理:
 - 概念:
 - 让一个处理器 (主服务器)处理所有调度决定、IO 处理以及其他系统活动, 其他的处理器只执行用户代码;
 - 优点:
 - 简单, 因为只有一个处理器访问系统数据结构, 减少了数据共享的需要;
- 对称多处理:
 - 概念:
 - 1. 每个处理器自我调度;
 - 2. 所有进程处于一个共同的就绪队列中, 或每个处理器都有它自己的私有就绪进程队列;
 - 3. 每个处理器的调度程序都检查共同就绪队列, 以便选择执行一个进程;
 - 注意:
 - 需要对多个处理器试图访问和更新共同的数据结构进行仔细检查;

处理器亲和性

概念: 处理器亲和性

- 引入:
 - 当进行进程调度时, 如果进程转移到其他处理器进行运行, 第一个处理器缓存的内容就被设为无效, 第二个处理器缓存应重新填充;
 - 缓存的无效或重新填充的代价高, 所以大多数多处理器调度系统试图避免将进程从一个处理器移到另一个处理器, 而是试图让 **一个进程运行在同一个处理器上**;
- 概念:
 - 一个进程对它运行的处理器具有亲和性, 试图让一个进程运行在同一个处理器上;
- 图示:



概念：软亲和性和硬亲和性

- 软亲和性：
 - 当一个操作系统试图保持进程运行在同一处理器上时 (但不保证它会这么做), 这种情况称为软亲和性;
- 硬亲和性：
 - 允许某个进程运行在某个处理器子集上 (通过提供系统调用以便支持硬亲和性) ;

负载均衡

概念：负载均衡与多处理器

- 概念：
 - 1. 通过负载均衡充分利用多处理器的优点, 防止出现一个或多个处理器会空闲、其他处理器会处于高负载状态;
 - 2. 负载均衡设法将负载平均分配到 SMP 系统的所有处理器;
- 两种方式：
 - 推迁移: 一个特定的任务周期性地检查每个处理器的负载, 如果发现不平衡, 那么通过将进程从超载处理器推到空闲或不太忙的处理器;
 - 拉迁移: 空闲处理器从一个忙的处理器上拉一个等待任务时, 发生拉迁移;

概念：负载均衡和处理器亲和性

- 保持一个进程运行在同一处理器上的好处是进程可以利用它在该处理器缓存内的数据;
- 无论是从一个处理器向另一处理器推或拉进程, 都会失去这个好处;