处理机管理

调度的层次

作业调度（高级调度）、内存调度（中级调度）、进程调度（低级调度）

进程状态切换依赖于调度。调度是为了合理的处理计算机软/硬件资源。

-	高级调度	中级调度	低级调度
功能	按照某种机制，将外存上后备队列中的作业调入内存，为其创建进程、分配资源、插入就绪队列。	内存空间紧张时，将暂时不具备运行条件的进程挂起（调至外存），释放内存资源。内存满足挂起状态的某进程需求时，由中级调度算法决定把外存上的具备运行条件的挂起进程重新调入内存。	将处理机分配给就绪队列中的某进程。是最基本的调度、操作系统核新部分，批处理OS、分时OS、实时OS等的必备功能。
作用	外存→内存（面向作业）	外存→内存（面相进程）	内存→CPU
频率	最低	中等	最高
影响	无→创建态→就绪态	挂起态→就绪态 （阻塞挂起→阻塞态）	就绪态→运行态

调度方式

非抢占式优先调度算法:
系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程一直执行到完成。

抢占式优先调度算法:
执行期间只要出现另一优先权更高的进程，进程调度程序就立即停止当前进程（原有限权最高的进程），重新分配处理机给新的优先权更高的进程。

调度时机

- 不能进行进程调度、切换的情况
  - 在处理终端过程中
  - 进程在操作系统内核程序临界区中
  - 其它需要完全屏蔽中断的原子操作的过程中
- 应该调度
  - 发生引起调度条件且当前进程无法继续运行
  - 中断处理、自陷处理结束

调度的实现方式

1. 具有高、低两级调度的调度队列模型：按照等待事件的不同，可以放入不同的队列中（不同队列可能有着不同的调度算法）。
2. 具有三级调度时的队列模型：

调度的准则

调度的目的是为了资源利用率高，尽可能保证所有进程的权利是一致的。要求CPU利用率高。

周转时间：作业从提交到系统至完成的时间间隔，周转时间T = 完成时间TF - 提交时间TB

带权周转时间T：Tweight，作业的周转时间T与系统为它提供服务时间Ts之比，W = T/Ts

平均周转时间：所有作业周转时间的平均值，T=(1/n)[iΣ(i=1)Ti]

平均带全周转时间：所有作业周转时间的平均值，W=(1/n)[iΣ(i=1)Ti/Tsi]

调度的算法

调度算法	算法思想	调度类型	优先级
先来先服务	从后背作业队列汇总萱蓦若干最先进入	作业调度&进程调度	等待时间
短作业优先	以运行/计算时间为优先级	作业调度&进程调度	运行时间
优先级调度	以优先数为优先级	作业调度&进程调度	优先数
高响应比优先	响应比=等待事件+运行时间/运行时间	作业调度	响应比
时间片轮转	所有进程按照轮询模式运行	进程调度	轮询模式
多级反馈队列	设置优先级不同的队列里，队列内部按照FCFS规则	进程调度	队列位置

先来先服务FCFS

以等待时间为优先级的调度算法。从后备作业队列中选择若干年最先进入该队列的作业调入内存，为其分配资源、创建进程、放入就绪队列。
利于长作业，不利于短作业；利于CPU繁忙的作业，不利于IO繁忙的作业；可以用于进度、作业调整。

短作业优先SJF

以运行/计算时间为优先级，优先将运行计算时间短的若干作业调入内存。可用于进程、作业调度

利于短作业；比FCFS改善平均周转时间和平均带全周转时间，缩短作业的等待事件，提高系统吞吐量。
但必须预知作业运行时间，对长作业非常不利（长作业的周转时间会明显增长）；人机无法实现交互；完全未考虑作业紧迫度，不能保证紧迫性作业能得到及时处理。

优先级调度算法PSA

依赖外部赋予的优先级（一般假设优先数越大，优先级越小）。可由于进程、作业调度。
优先级在进程创建时被生成，按创建后优先级能否改变分为静态优先级、动态优先级；按更高优先级能否抢占正在执行的进程分为菲薄多事、剥夺式。
优先级设置：系统进程>用户进程，交互型进程>非交互型进程，IO型进程>计算型进程。

高响应比优先调度算法

每次选择作业投入运行时，先计算此时后备作业队列中每个作业的响应比RP然后选择其数值最大的作业投入运行。只用于作业调度。

优先权Rp = 1 + 等待时间/服务时间 = 响应(周转)时间/服务时间。

若作业等待时间相同，则要求服务的时间越短，其优先权越高，此时类似SJF算法利于短作业；
若要求服务时间相同，作业优先权取决于等待时间，此时类似FCFS算法利于长作业；
对于长时间的优先级，可以为随等待事件的增加而提高，当等待时间足够长，也可以获得处理机。

时间片轮转

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，若进程未在该时间片内执行完则会剥夺处理机，将进程放到就绪队列队尾。分时操作系统常用，可以及时响应用户请求，保证所有进程在面对CPU时的优先级一样。只用于进程调度。

多级反馈队列

动态调整进程优先级和时间片大小。只适用于进程调度。

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。新锦成到达时先进入第1级队列，按FCFS原则排队等待被分配时间片，若用完时间片后进程还未结束则进入下一级队列队尾，若此时已经在最下级队列对位，则重新放回最下级队列队尾。只有第k级队列为空，才会为k+1级对头的进程分配时间片，被抢占的处理机进程重新放回原队列队尾。

进程的上下文切换

CPU寄存器和程序计数器就是CPU的上下文，都是CPU在运行任何任务前必须的依赖环境。
进程具有共享性、并发性。进程的状态切换依赖于调度。调度需要进程控制块PCB。

进程的上下文切换就是把前一个任务的CPU上下文保存，然后加载新任务的上下文到这些寄存器和程序计数器，最后挑战到程序计数器所指的新位置，运行新任务。

调度实现方式：具有高、低两级调度的调度队列模型；具有三级调度的调度队列模型。

系统调用过程：传递系统调用参数 -> 执行陷入指令（用户态） -> 执行相应的内请求核程序处理系统调用（核心态） -> 返回应用程序

进程由内核管理和调度，进程的切换只发生在内核态。进程的上下文包括虚拟内存、栈、全局变量（用户空间资源），内核对战、寄存器状态（内核空间状态）。
进程的上下文切换比系统调用多了：保存内核态资源前，先把进程的用户态资源保存；加载下一进程内核态后，还需刷新进程的虚拟内存和用户栈。

进程上下文切换的场景：

- 进程时间片耗尽，就会被挂起并切换到其它等待CPU的进程运行
- 系统资源不足，进程要等待资源满足后才能运行
- 进程通过睡眠函数sleep主动挂起
- 有优先级更高的进程运行
- 硬件中断

死锁的概念和性质

计算机有可重用性资源（计算机外设），消耗性资源（数据、消息），可抢占性资源（不引起死锁，CPU、内存），不可抢占行资源（光驱、打印机）。多道程序系统可借助多个进程的并发执行改善系统资源利用率，提

高系统吞吐量，但可能导致死锁。

死锁是当多个进程在程序运行该过程中因争夺资源而造成一种僵局。此状态OS若无外力作用酱无法向前推进。

死锁的原因有：系统资源的竞争（不可剥夺资源），程序推进顺序非法（请求和释放资源的顺序不当，信号量的使用不当），死锁产生的必要条件（互斥条件、不剥夺条件、请求并保持条件、循环等待条件）

死锁产生的必要条件

- 互斥使用：在一段时间内某资源只能由一个进程占用。其它进程请求只能等待直到该资源进程使用完毕将其释放
- 请求保持：进程已占用至少一个资源，有申请被其他进程占用不释放的资源，则会请求进程阻塞，但不释放资源
- 不可抢占：进程已获资源在使用完之前，不能被其他进程抢占，只能在使用完后自己释放
- 循环等待：发生死锁时，必须存在一个进程-资源循环链

死锁的处理策略

静态策略：预防死锁；
动态策略：避免死锁，检测死锁，解除死锁。

安全状态：系统能按照某种进程顺序（安全序列）为每个进程分配其所需资源，直到满足每个进程对资源的最大需求，使每个进程都可以顺利完成（如果系统无法找到这样的安全序列，则称系统处于不安全状态）。

死锁预防

通过实现采取某种限制措施，破坏死锁产生的必要条件，达到预防死锁的目的。

1. 条件一：互斥使用：由设备的固有条件决定，不仅不能改变，还能加以改变。
2. 条件二：请求保持条件：可破坏
    1. 协议1：所有进程开始运行前，必须一次性申请其在整个运行过程中所需的全部资源。运行期间不再申请新资源；等待期间不占有任何资源，破坏“保持”条件；协议简单安全，但会造成资源浪费，经常发生金城街现象
    2. 协议2：进程获得运行初期所需资源后便开始运行。资源利用率高，能有效减少饥饿现象的概率
3. 条件三：不可抢占：可破坏
    - 已经保持了某些不可被强占资源的进程，提出新的资源请求而不能得到满足时，其申请到的资源可以被剥夺，以后需要重新申请。实现复杂，代价大
4. 条件四：循环等待：可破坏
    - 对系统所有资源类型进行现行排序，并赋予不同的序号；进程必须按照序号递增顺序申请资源。对资源安排很重要，系统资源利用率和吞吐率比其它策略有明显改善；资源序号相对稳定性导致新设备的增加受限，由于作业使用资源的顺序和系统资源序号顺序未必相同，可能导致资源浪费。

死锁避免

资源动态分配过程中用某种方式防止系统进入*不安全状态*，避免发生死锁。
避免死锁比预防死锁是假的限制条件弱，较完善的系统常用死锁避免处理死锁，允许进程动态地申请资源，但OS分配资源前要计算资源分配的安全性。

银行家算法

引入：
可利用资源向量Available数组，每一个元素代表一类可以用的资源数目；
最大需求矩阵Max（如果Max[i,j]=K，表示进程i需要Rj类资源的最大数目为K）；
分配矩阵Allocation（如果Allocation[i,j]=K，表示进程i当前已分到的Rj类资源数目为K）；
需求矩阵Need（如果Need[i,j]=K，表示进程i还需Rj类资源K个）。

设Requesti是进程Pi的请求向量，如果Requesti[j]=K，表示进程需要K个Rj类型资源，发出资源请求后将按如下步骤检查：

1. if Requesti[j] ≤ Need[i,j]，转到步骤2；否则出错
2. if Requesti[j] ≤ Available[j]，跳转3；否则Pi需要等待
3. OS试探着把资源分配给进程Pi，并修改下列数值：
    - `Available[j] := Available[j] - Requesti[j];`
    - `Allocation[i,j] := Allocation[i,j] + Requesti[j];`
    - `Need[i,j] := Need[i,j] - Requesti[j]`

随后系统执行安全性算法检查此次资源分配后系统是否出于安全状态，以判断是否需要正式分配给Pi。安全性检查算法步骤如下：

1. 设置两个向量：
    1. 工作向量Work：系统可供给进程继续运行所需的各类资源数目，在执行安全算法开始时，Work := Available
    2. Finish：表示系统是否有足够资源分配给进程，使之运行完成。开始时Finish[i]:=false；有足够资源时Finish[i]:=true
2. 找到一个能满足下述条件的进程（若找到则执行步骤3；否则执行4）
    1. Finish[i]=false;
    2. Need[i,j]≤Work[j]
3. 进程Pi获取资源并顺利执行完成后，释放资源，执行：
    - `Work[j] := Work[i] + Allocation[i,j];`
    - `Finish[i] := true;`
    - *`go to step 2;`*
4. 如果所有进程Finish[i]=true都满足，表示系统处于安全状态；否则处于不安全状态

死锁检测

通过系统设置的检测机构来及时检测死锁的发生，精确确定与死锁有关的进程和资源，清除已发生死锁的进程。

引入资源分配图描述系统死锁情况，G=(N,E)，G为图，N表示G的顶点集合，E表示G的边集合。

- 检测定理：利用简化资源分配图检测当前系统是否处于死锁状态
    - 在资源分配图中找出一个既不阻塞也不独立的进程节点Pi
    - 顺利时，Pi可获得所需资源继续运行，直到完成时释放其占有的全部资源，相当于消去Pi请求边和分配边，使之成为孤立节点

死锁解除

检测到系统已发生死锁时，需将进程从死锁状态中解脱。
它和死锁检测相配套。通过回收资源并再分配给处于阻塞状态的进程，是指转为就绪状态继续运行。

解除方法有：

- 抢占资源：强行剥夺部分死锁状态进程所占资源，并分配给其它死锁状态的进程使之能够运行，从而解除死锁
- 撤销/挂起/终止进程：
  - 终止所有死锁进程（简单但代价大）
  - 逐个终止进程：一次选择付出代价最小的进程并终止，直到有足够资源打破循环等待，并将系统从死锁状态解脱
- 付出代价最小的死锁接触算法：经解除进程的代价按升序排列，每次终止队首进程，直到死锁解除

补充知识点

- 进程调度算法采用固定时间片轮转调度算法时，时间片过大会使得时间片轮转发算法转化为FCFS算法
- FCFS算法有利于CPU繁忙型作业，不利于IO繁忙型作业
- 所有进程同时到达时，进程平均周转时间最短的事短进程优先算法。短进程算法是性能最好的算法
- 综合考虑进程等待时间和执行时间的是高响应比算法。响应比 = （等待时间+服务时间)/服务时间
- 满足段任务有限且不会发生饥饿现象的调度算法是高响应比优先算法

-	**先来先服务**	**高响应比优先**	**时间片轮转**	**非抢占式短任务优先**
优先短任务?	×	√	×	√
会发生饥饿?	×	×	×	√

- 计算算法性能的表格：

任务	**提交时间**	**运行时间**	**开始运行**	**结束运行**	**等待时间+服务时间**	**响应比**

- 有n太互斥使用的同类设备，三个并发进程需要3、4、5太设备，可确保不发生死锁的设备数为2+3+4+1 = 10
- 死锁预防会限制用户申请资源的顺序，死锁避免不会