2.6 进程调度

- 在计算机系统中，进程总数一般均多于CPU数，必然会出现竞争CPU的情况
- 进程调度就是按一定策略、动态地把CPU分配给处于就绪队列中的某一进程执行
- 两种基本的进程调度方式，抢占方式和非抢占方式，也称剥夺式和非剥夺式调度

2.6 进程调度

- 剥夺原则有：优先权原则、短进程优先原则、时间片原则——就绪队列中一旦出现符合上述原则的进程，系统便立即剥夺当前运行进程的CPU使用权，进行进程切换。
- 非剥夺式调度中，一旦CPU分配给了某进程，即使就绪队列中出现了优先级比它高的进程，系统也不能抢占运行进程的CPU使用权，而必须等待该进程主动让出。

2.6 进程调度

- 可能引发进程调度的时机：
  - 正在运行的进程运行完毕；
  - 运行中的进程要求I/O操作；
  - 执行某种原语操作(如P操作)导致进程阻塞；
  - 比正在运行的进程优先级更高的进程进入就绪队列；
  - 分配给运行进程的时间片已经用完

2.6 进程调度

- 进程调度使用频率高，其性能优劣直接影响操作系统的性能。
- 根据不同的系统设计目标，可有多种进程调度策略：例如系统开销较少的静态优先数法、适合于分时系统的时间片轮转法以及动态优先数反馈法等。
- 评价调度算法的好坏，用得较多的是批处理系统中的周转时间、平均周转时间和带权周转时间及分时系统中的响应时间。

2.6 进程调度

- 周转时间是指从作业提交给系统开始，到作业完成为止的间隔时间；
- 平均周转时间是指各作业周转时间的平均值；
- 带权周转时间是指周转时间与系统为它提供服务的时间之比；
- 响应时间是指从键盘命令进入(按下回车键为准)到开始在终端上显示结果的时间间隔。
- 对调度算法，以上时间越短越好。除此之外，系统吞吐量、CPU利用率及各类资源的平衡利用情况也是评价调度算法的标准。

2.6.1 进程调度模型

- **高级调度**(High-Level Scheduling)，又称作业调度，在分时和实时系统中不需要。其主要功能是根据一定的算法，从后备作业中选出若干个作业，分配必要的资源，如内存、外设等，为它建立相应的用户作业进程和为其服务的系统进程（如输入、输出进程），将其程序和数据调入内存，等待进程调度程序对其执行调度，并在作业完成后作善后处理工作

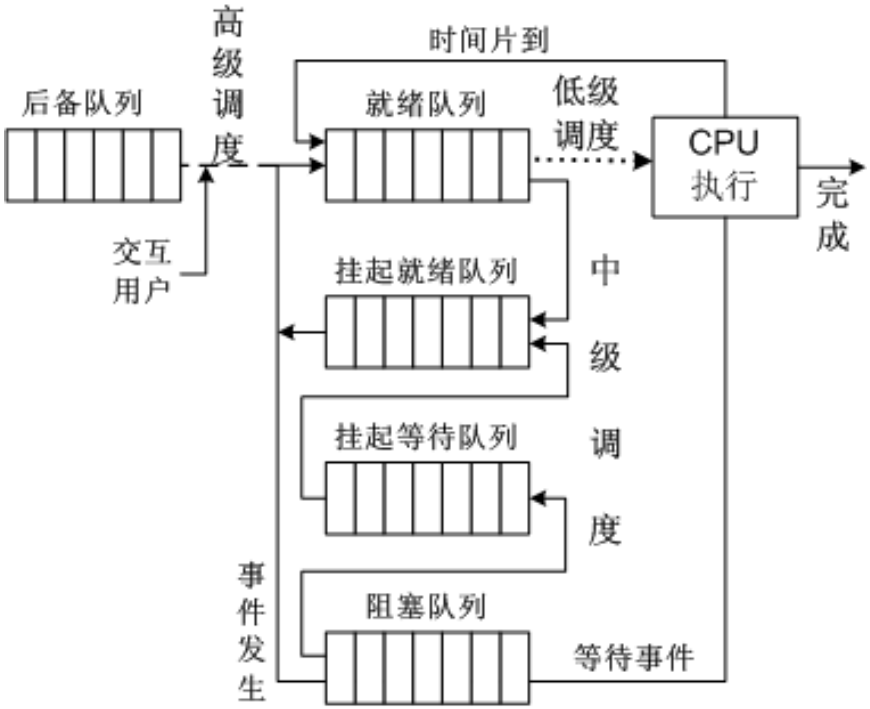
2.6.1 进程调度模型

- **中级调度**(Intermediate-Level Scheduling)，又称为平衡调度，在采用虚拟存储技术的系统中引入，以提高系统吞吐量。其功能是在内存使用情况紧张时，将一些暂时不能运行的讲程从内存对换到外存上等待。当以后内存有足够的空闲空间时，再将合适的进程重新换入内存，等待进程调度。

2.6.1 进程调度模型

- **低级调度**：又称**进程调度**，主要功能是根据一定的算法将CPU分派给就绪队列中的一个进程。进程调度的运行频率很高，在分时系统中往往几十毫秒就要运行一次。
- 进程调度是操作系统中最基本的一种调度。一般操作系统中都必须有进程调度，而且它的优劣直接影响整个系统性能。
- 不同操作系统的调度模型各有不同，一级调度系统仅设有低级调度，二级调度系统拥有高级调度和低级调度，还有些系统则为三级调度。

2.6.1 进程调度模型


高级调度
时间片到
后备队列
就绪队列
低级调度
CPU执行
完成
交互用户
挂起就绪队列
中级调度
挂起等待队列
事件发生
阻塞队列
等待事件

2.6.2 调度算法选择/评价准则

- 调度算法也称为调度策略，用于控制对CPU的分配，是CPU调度中非常重要的环节。
- 评价调度算法的相关标准如下：
  - 处理器利用率
  - 响应时间
  - 周转时间、带权周转时间
  - 平均作业周转时间、平均作业带权周转时间
  - 系统吞吐量
  - 公平性

2.6.2 调度算法选择/评价准则

（1）处理器利用率——CPU有效工作时间与CPU总的运行时间之比，即：

CPU利用率 =

CPU有效工作时间 / CPU总的运行时间

CPU总的运行时间=CPU有效工作时间+CPU空闲时间

对于轻负载系统，CPU的利用率大约为40%；

对于重负载系统，CPU的利用率可达90%。

2.6.2 调度算法选择/评价准则

（2）响应时间（Response Time）——交互环境下用户从键盘提交请求开始，到系统首次产生响应为止的时间，或者是屏幕上显示出结果为止的时间。

是分时系统和实时系统衡量调度性能的一个重要指标。

响应时间=从终端键盘输入的请求信息传送到系统的时间 + 系统对用户请求的处理时间

+ 生成的响应信息回送到终端显示器的时间

2.6.2 调度算法选择/评价准则

（3）周转时间——用户作业提交给操作系统开始到作业完成为止的时间，是批处理系统衡量调度性能的一个重要指标。

周转时间Ti = 作业在后备队列中的等待调度时间

+进程在就绪队列上等待调度的时间

+进程在CPU上的运行时间

+进程等待I/O或其它事件发生的时间

或简化为：$T_i = T_f - T_s$

即：周转时间 = 完成时刻 - 提交时刻

2.6.2 调度算法选择/评价准则

（4）**带权周转时间**：W_i = 作业的周转时间T_i / 系统为作业提供的服务时间T_{si}，显然带权周转时间总大于1

（5）**平均作业周转时间**：$T = (\Sigma T_i) / n$

（6）**平均作业带权周转时间**：$W=(\Sigma W_i) / n$

（7）**系统吞吐量**（Throughput）：单位时间内完成的作业/进程数目

（8）**公平性**：确保每个用户和进程获得合理的CPU份额或其它资源份额，不出现饥饿情况

2.6.2 调度算法选择/评价准则

- 用户希望CPU利用率和系统吞吐量越大越好，响应时间和周转时间越小越好。
- 对于实时系统，作业调度的关键在于能否满足作业的实时要求，对周转时间等指标并不特别着重。

2.6.3 调度算法

1．先来先服务（First-Come First-Served，FCFS）：按进程就绪的先后顺序调度，到达得越早，就越先执行，获得CPU的进程，未遇到其它情况时，将一直运行下去

- 是一种非抢占式调度算法，简单易实现
- 只考虑作业/进程等待时间，没有考虑执行时间长短、运行特性和对资源的要求
- 既适用于进程调度也适用于作业调度

2.6.3 调度算法

【例2-1】系统中现有5个作业A、B、C、D、E同时提交（到达顺序也为ABCDE），其预计运行时间分别10、1、2、1、5个时间单位，如表2.1所示，计算FCFS调度下作业的平均周转时间和平均带权周转时间。

作业ID	预计需运行时间
A	10
B	1
C	2
D	1
E	5

2.6.3 调度算法

- 设作业到达时刻为0，系统运行情况见下表：

作业ID	运行时间	等待时间	开始时间	完成时间	周转时间	带权周转时间
A	10	0	0	10	10	1
B	1	10	10	11	11	11
C	2	11	11	13	13	6.5
D	1	13	13	14	14	14
E	5	14	14	19	19	3.8
平均周转时间 平均带权周转时间		T =（10+11+13+14+19）/ 5 = 13.4 W =（1+11+6.5+14+3.8）/ 5 = 7.26				

2.6.3 调度算法

- FCFS调度算法特性：
  - 对长作业非常有利，对短作业不利
  - 对CPU繁忙型作业有利，对I/O繁忙型作业不利，因为进程I/O阻塞状态结束后，需要再次排队等待被分配CPU，因此I/O型作业的周转时间和等待时间都非常长
  - 非抢占式算法，对响应时间要求高的进程不利
  - 平均作业周转时间与作业提交顺序有关

2.6.3 调度算法

2．短作业优先（Shortest-Job-First，SJF）----以进入系统的作业所要求的CPU服务时间为标准，总选取估计所需CPU时间最短的作业优先投入运行

- **SJF调度算法特性：**
  - **算法易于实现，是对FCFS 算法的改进，其目标是减少平均周转时间**
  - **非抢占式算法，对响应时间要求高的进程不利**
  - **既适用于进程调度也适用于作业调度，但在实际应用中较少用于进程调度**

2.6.3 调度算法

【例2-2】同例2-1，采用SJF算法调度作业

运行次序	运行时间	等待时间	开始时间	完成时间	周转时间	带权周转时间
B	1	0	0	1	1	1
D	1	1	1	2	2	2
C	2	2	2	4	4	2
E	5	4	4	9	9	1.8
A	10	9	9	19	19	1.9
平均周转时间 平均带权周转时间		T =（1+2+4+9+19）/ 5 = 7 W =（1+2+2+1.8+1.9）/ 5 = 1.74				

2.6.3 调度算法

中国矿业大学

➢ SJF调度算法特性：

- 对长作业不利，如果系统不断接收短作业，可能会出现饥饿现象
- SJF的平均作业周转时间比FCFS要小，故它的调度性能比FCFS好
- 实现SJF调度算法需要知道作业所需运行时间，而要精确知道一个作业的运行时间是办不到的

2.6.3 调度算法

3．最短剩余时间优先（Shortest Remaining Time First，SRTF）——若一就绪状态的新作业所需的CPU时间比当前正在执行的作业剩余任务所需CPU时间还短，SRTF将打断正在执行作业，将执行权分配给新作业

- **SRTF将SJF算法改为抢占式，只要有新作业进入就绪队列，就可能会引发进程切换。**

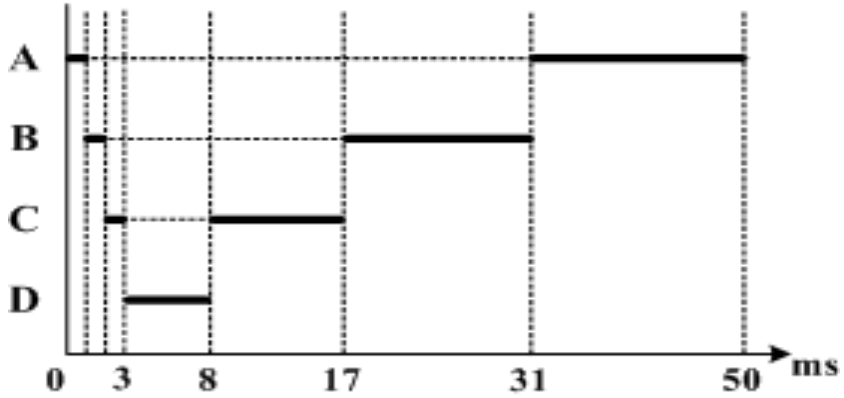
2.6.3 调度算法

- SRTF调度算法特性：
  - 长进程仍有可能出现饥饿现象
  - 必须计算运行、剩余时间，系统开销增大
  - 抢占式调度，系统性能比SJF要好
  - 既适用于进程调度也适用于作业调度

2.6.3 调度算法

【例2-3】作业A、B、C、D需要运行的时间分别为20ms、15ms、10ms、5ms。A作业在0ms到达，B作业在1ms到达，C作业在2ms到达，D作业在3ms到达，计算SRTF调度下作业的平均周转时间和平均带权周转时间 。

2.6.3　调度算法



- A、B、C、D 周转时间分别为50ms、30ms、15ms、5ms
- 平均周转时间为（50＋30＋15＋5）／ 4＝25.00 ms
- A、B、C、D 带权周转时间为分别为2.5、2、1.5、1
- 平均带权周转时间为（2.5＋2＋1.5＋1）／ 4 ＝ 1.75

2.6.3 调度算法

4. **高响应比优先**（Highest Response Ratio First，**HRRF**）—是FCFS与SJF两种算法的折衷，既考虑作业等待时间，又考虑作业的运行时间，既照顾短作业又不使长作业等待过久，改善了调度性能，**仍属于非抢占式算法**

- 响应比为作业的响应时间与作业所需运行时间之比，简化为：
- 响应比＝1+(已等待的时间/估计运行时间)

2.6.3 调度算法

中国矿业大学

HRRF算法特性：

- 短作业容易得到较高响应比，长作业在等待了足够长的时间后，也将获得足够高的响应比，因此**不会发生饥饿现象**
- 需要经常计算作业的响应比，导致额外的开销
- **HRRF**算法的平均周转时间和平均带权周转时间都介于**FCFS**与**SJF**算法之间，比SJF算法差，比FCFS算法优，在现实中其可以实现，结果也比较可靠

2.6.3 调度算法

中国矿业大学

HRRF算法特性：

- **既适用于进程调度也适用于作业调度**
- **如果在算法中引入抢占调度，则算法过程会更复杂，因为所有作业的响应比是动态变化的，抢占时间的计算需要解多个方程得到**

2.6.3 调度算法

【例2-4】系统中现有3个作业A、B、C先后提交（到达），其参数如表所示，计算HRRF调度下作业的平均周转时间和平均带权周转时间

作业ID	提交时间	预计需运行时间
A	00:00	2:00
B	00:10	1:00
C	00:25	0:25

2.6.3 调度算法

作业ID	提交时间	运行时间	开始时间	完成时间	周转时间	带权周转时间
A	00:00	2:00	00:00	02:00	2	1
B	00:10	1:00	02:25	03:25	3.25	3.25
C	00:25	0:25	02:00	02:25	2	4.8
平均周转时间 平均带权周转时间		T =（2+3.25+2）/ 3 = 2.4 W =（1+3.25+4.8）/ 3 = 3				

2.6.3 调度算法

5．**优先权调度算法**（Highest-Priority-First，HPF）——根据作业/进程的优先权进行进程调度，每次总是选取优先权高的作业/进程调度，也称优先级调度算法

- 一般是抢占式调度
- 既适用于进程调度也适用于作业调度

2.6.3 调度算法

- 优先权通常用一个整数表示，也叫优先数
  - Windows系统中有0~31共32个优先级，31最高
  - Unix系统中，使用数值-20~+19来表示优先级，-20优先级最高
- 优先权可由系统或用户给定
  - 系统会按照进程的执行时间长短以及对资源的要求而给定进程的优先权
  - 用户在进程送入系统时，为自己的进程指定一个优先数，优先数反映了用户对进程执行的急切程度

2.6.3 调度算法

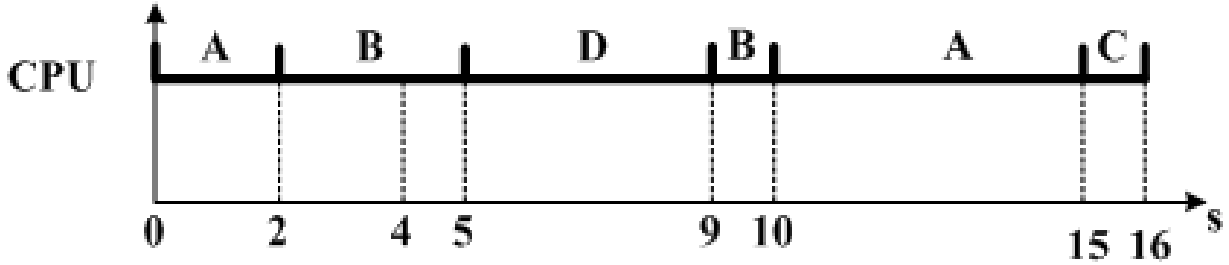
优先权存在两种形式：

- **静态优先权**指进程的优先权在进程进入系统时给定后不再改变，这可能会导致饥饿现象
- **动态优先权**指进程的优先权在进程进入系统时给定，随着进程的运行和等待时间的变化而发生变化，该情况与响应比相似

2.6.3 调度算法

【例2-5】系统的进程调度采用抢占式优先权调度算法，优先数越小优先权越高，其参数如表所示，求平均周转时间和平均等待时间

作业ID	提交时间	预计需运行时间	优先数
A	0	7	3
B	2	4	2
C	4	1	4
D	5	4	1



作业ID	提交（到达）时间	运行结束时间
A	0	15
B	2	10
C	4	16
D	5	9

平均周转时间T =（15+8+12+4）/ 4 = 9.75

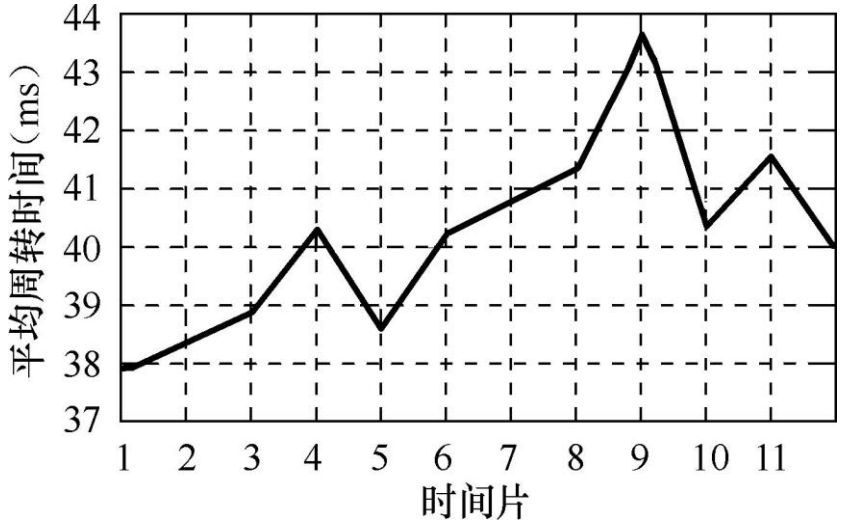
平均等待时间T_w =（8+4+11+0）/ 4 = 5.75

2.6.3 调度算法

6. **时间片轮转调度算法**（Round-Ribon，RR）：调度程序把CPU分配给进程使用一个规定的时段，称为一个时间片（如100ms），就绪队列中的进程轮流获得CPU的一个时间片，当一个时间片结束时，系统剥夺该进程执行权，等候下一轮调度

- **属于抢占式调度，仅适用于进程调度**
- 时间片的长短，影响进程的进度
  - 需要从进程数、切换开销、系统效率和响应时间等方面综合考虑，确定时间片大小

- 系统的花销主要体现在进程切换上
- 当时间片大到每个进程足以完成时，时间片调度算法便退化为FCFS算法



时间片和平均周转时间关系

2.6.3 调度算法

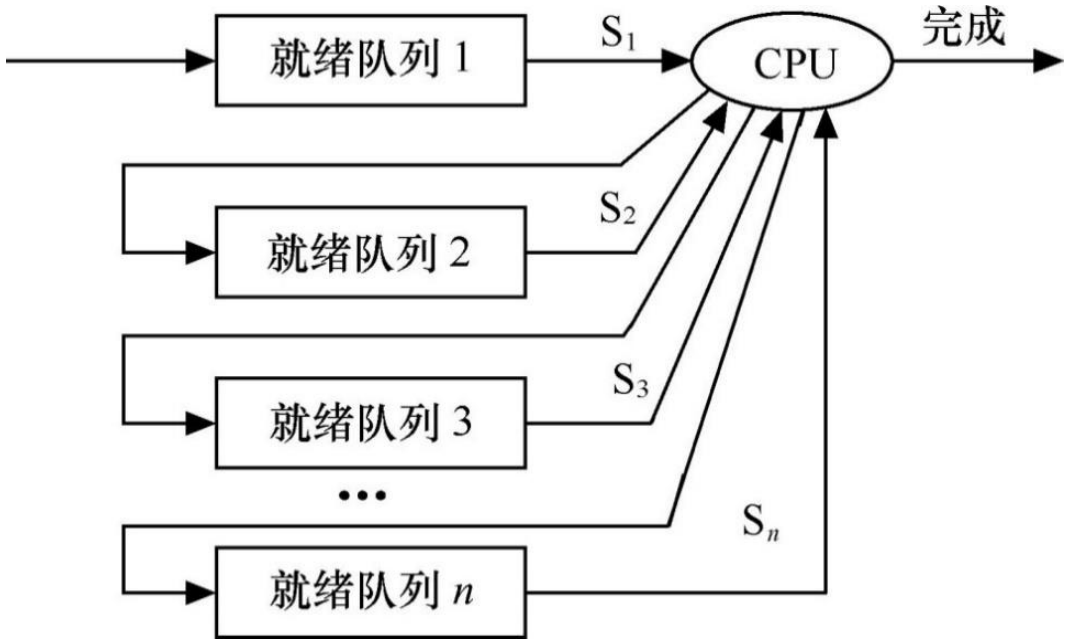
7. 多级反馈队列（Multilevel-Feed-Queue，MFQ）—又称反馈循环队列，是一种基于时间片的进程多级队列调度算法

- 系统设置多个就绪队列，最高级就绪队列的优先级最高，随着就绪队列级别的降低优先级依次下降
- 较高级就绪队列的进程获得较短的时间片

2.6.3 调度算法

- 新建进程首先进入最高优先级队列，随着得到CPU执行的次数增加，该进程逐渐进入较低级就绪队列
- 不需事先知道各进程所需运行时间，因而可行性较高，同时综合考虑了进程的时间和优先权因素，既照顾了短进程，又照顾了长进程，是一种综合调度算法，被广泛应用于各种操作系统中

2.6.3 调度算法



多级反馈队列调度算法

2.6.3 调度算法

- 多级反馈队列调度算法在许多操作系统中得到了应用，如**UNIX**和**Linux**
  - 在BSD UNIX操作系统中，进程的就绪队列有32个，按照0-31号的顺序，优先级逐渐降低，其中0-7号就绪队列用于系统进程，8-31号就绪队列用于用户进程
  - 系统进程的优先级高于用户进程，在每个就绪队列内部，采用时间片轮转调度，时间片是变化的，最大时间片不能超过100ms

课堂练习： 下表给出进程I，2，3的提交时间和运行时间。请采用最短剩余时间优先（抢占式）和高响应比优先（非抢占式），求各进程的平均周转时间。

进程号	提交时间	运行时间
1 2 3	0.0 0.4 1.0	8.0 4.0 1.0

最短剩余时间优先：平均周转时间 （1+5+13）/3 =6.33
高响应比优先：　　平均周转时间 （8+8+12.6）/3 =9.53

进程号	提交时间	运行时间	结束时间	周转时间
1 2 3	0.0 0.4 1.0	8.0 4.0 1.0	13 5.4 2	13 5 1

进程号	提交时间	运行时间	结束时间	周转时间
1 2 3	0.0 0.4 1.0	8.0 4.0 1.0	8 13 9	8 12.6 8

2.6.4 多CPU系统中的调度

- 多处理器系统的作用是利用系统内的多个CPU来并行执行用户进程，以提高系统的吞吐量或用来进行冗余操作以提高系统的可靠性。
- 系统的多个处理器在物理上处于同一机壳中，有一个单一的系统物理地址空间，多个处理器共享系统内存、外设等资源。

- 多CPU系统的类型主要有两种：
  - 主-从模式：只有一个主处理器，运行操作系统，管理整个系统资源，并负责为各从处理器分配任务，从处理器有多个，执行预先规定的任务及由主处理器分配的任务。这种类型的系统无法做到负载平衡，可靠性不高，很少使用。
  - 对称处理器模式SMP：所有处理器都是相同的、平等的，共享一个操作系统，每个处理器都可以运行操作系统代码，管理系统资源，是目前比较常见的多CPU系统类型。

2.6.4 多CPU系统中的调度

- 多处理器系统中，比较有代表性的进（线）程调度方式有以下几种方式：
  - 1）自调度
  - 2）组调度/群调度（Gang Scheduling）
  - 3）专用处理器分配
  - 4）动态调度

2.6.4 多CPU系统中的调度

（1）自调度：由单处理器环境下的调度方式演变而来，在系统中设置有一个公共的进程或线程的就绪队列，所有的处理器在空闲时都可自己到该队列中取得一进程（线程）来运行。

优点：不会出现处理器空闲的情况，任何处理器都可利用OS的调度例程去选择一线程执行，其组织方式和调度算法可沿用单处理器所用的算法。

2.6.4 多CPU系统中的调度

缺点：系统中只能设置一个就绪线程队列，处理器必须互斥地访问该队列，很容易形成系统瓶颈。当线程阻塞后又重新就绪时，可能要更换处理器，因而使高速缓存的使用效率很低。同步协作型的线程很难同时获得处理器而同时运行，会使某些线程频繁切换。

2.6.4 多CPU系统中的调度

（2）组调度/群调度（Gang Scheduling），是将相关的一组线程组织为一个调度群，分配到一组处理器上去执行。相关线程或进程能够并行，有效减少阻塞，减少切换，降低调度频率，改善了系统性能。系统调度时采用面向进程或面向线程的方式分配处理器。

2.6.4 多CPU系统中的调度

（3）专用处理器分配，在一个进程执行期间，专门为该进程分配一组处理器，对应每一个线程一个，这组处理器供该进程专用直至结束。

实际上是组调度/群调度的极端形式，追求高度并行，避免低级调度，但会造成处理器的严重浪费。

2.6.4 多CPU系统中的调度

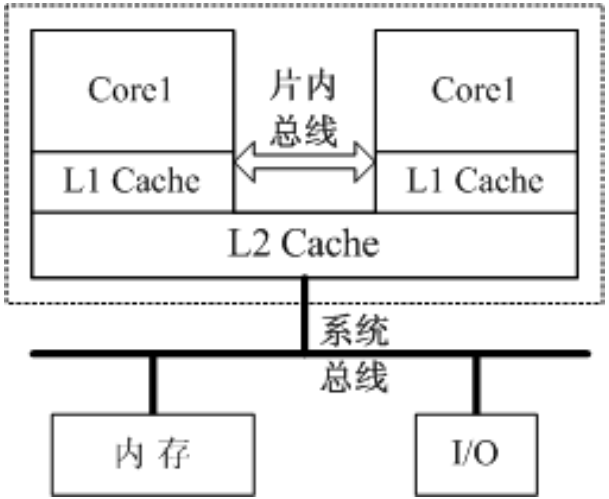
（4）动态调度，由操作系统和进程共同进行调度，操作系统负责在进程间分配处理器，进程内的线程调度可由进程负责。

2.6.5 多核CPU中的调度

- 多核处理器是指在一枚处理器中集成两个或多个计算引擎（内核/core），称为CMP（Chip multiprocessors）结构，其思想是将大规模并行处理器中的SMP（对称多处理器）集成到同一芯片内，并用高速片内总线互联。
- 操作系统会将每个内核作为分立的逻辑处理器，各内核并行调度执行不同的进程/线程。通过在多个内核之间划分任务，多核处理器可在特定的时钟周期内执行更多任务。

2.6.5 多核CPU中的调度

- 多核处理器通常采用共享二级Cache的结构，即每个处理器核心拥有私有的一级Cache，且所有处理器核心共享二级Cache。



2.6.5 多核CPU中的调度

- 对于多核CPU，优化操作系统任务调度算法是保证效率的关键
  - 全局队列调度——（多数系统）
  - 局部队列调度

2.6.5 多核CPU中的调度

- **全局队列调度：**指操作系统维护一个全局的任务等待队列，当系统中有一个CPU核心空闲时，操作系统就从全局任务等待队列中选取就绪任务开始在此核心上执行。
- 优点是CPU核心利用率较高。

2.6.5 多核CPU中的调度

- **局部队列调度：**操作系统为每个CPU内核维护一个局部的任务等待队列，当系统中有一个CPU内核空闲时，便从该核心的任务等待队列中选取恰当的任务执行。
- 优点：任务基本上无需在多个CPU核心间切换，有利于提高CPU核心局部Cache命中率。
- 目前多数多核CPU操作系统采用的是基于全局队列的任务调度算法。

2.6.5 多核CPU中的调度

- **中断：**多核的各处理器之间需要通过中断方式进行通信，所以多个处理器之间的本地中断控制器和负责仲裁各核之间中断分配的全局中断控制器也需要封装在芯片内部。
- 另外，多核CPU是一个**多任务系统**。由于不同任务会竞争共享资源，因此需要系统提供同步与互斥机制。而传统的用于单核的解决机制并不能满足多核，需要利用硬件提供的“读－修改－写”的原子操作或其他同步互斥机制来保证。

2.6.5 多核CPU中的调度

- 除了上述情况以外，多核CPU与多CPU系统中的调度算法基本一致，毕竟每个处理器核心实质上都是一个相对简单的单线程微处理器或多线程微处理器。

2.7 死锁

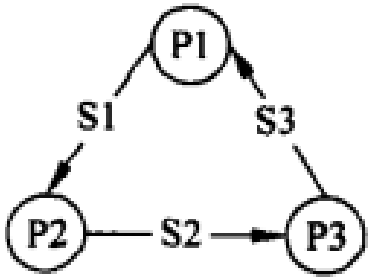
- 一个进程集合中的每个进程都在等待只能由该集合中的其它一个进程才能引发的事件，则称一组进程或系统此时发生了死锁
  - 例如，有一跨河的独木桥，当两人相对而行至桥中间而互不相让，就会出现谁也不能过河的死锁局面
- 死锁一旦发生，会使整个系统瘫痪而无法工作

2.7.1 死锁的产生原因

- 死锁产生的原因主要有两个：
  - 并发进程对临界资源的竞争
    - 例如哲学家进餐问题
  - 并发进程推进顺序不当
    - 程序执行顺序问题

2.7.1 死锁的产生原因

- 系统中有三个进程P1、P2和P3，相互之间需要传递消息S1、S2、S3，即：进程P1发送消息S1给进程P2，进程P2发送消息S2给进程P3，进程P3发送消息S3给进程P1



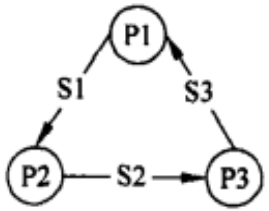
2.7.1 死锁的产生原因

- 如果每个进程都是先发送消息、再接收消息成功后才能向前推进，可能有如下情况：

  P1：send（S1），receive（S3）；

  P2：send（S2），receive（S1）；

  P3：send（S3），receive（S2）；



- 则三个进程都能够顺利发送并收到所需信息，继续运行下去

2.7.1 死锁的产生原因

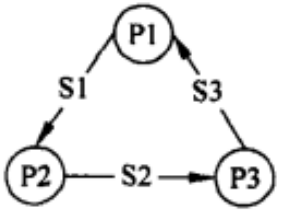
- **如果进程间推进顺序如下：**

  P1：receive（S3），send（S1）；

  P2：receive（S1），send（S2）；

  P3：receive（S2），send（S3）；

- **三个进程永远都不能接收到所需要的信息，不能继续运行，发生了死锁**



2.7.2 死锁产生的必要条件

1971年，Coffman等人总结出死锁发生的四个**必要条件**：

（1）**互斥条件**（Mutual exclusion）—资源的使用是互斥的

- 如果资源已经被一个进程占有，则再请求该资源的进程只能等待，直到占有资源的进程用完后归还

2.7.2 死锁产生的必要条件

1971年，Coffman等人总结出死锁发生的四个**必要条件**：

（2）**占有并等待条件**（Hold and wait）——已经得到某些资源的进程申请新资源时，若请求的资源不能得到，则已得到的资源也不会释放

2.7.2 死锁产生的必要条件

1971年，Coffman等人总结出死锁发生的四个**必要条件**：

（3）**不剥夺条件**（No pre-emption）——当某进程得到资源后，只能由其自身主动释放，系统或其它进程不能剥夺该进程已经获得的资源

2.7.2 死锁产生的必要条件

（4）**环路等待条件**（Circular wait）——系统中若干进程间形成等待环路，每个进程都在等待相邻进程正占用的资源，形成永远等待

- 这四个条件是死锁发生的必要条件，而非充分条件，而且第四个条件并不能独立存在。其中任意一条不满足，都不会发生死锁。因此只要破坏上述几个条件之一，即可防止死锁。

2.7.2 死锁产生的必要条件

- 只要破坏上述几个条件之一，即可防止死锁

（1）破坏第1个条件，使资源可同时访问而不是互斥使用，可行性较差

  - 可重入程序，只读数据文件，磁盘等可以采用这种方法，但许多资源如打印机，可写文件等由于其特殊性质决定只能互斥地使用；因此，对大多数资源来讲，破坏互斥条件不现实

2.7.2 死锁产生的必要条件

- 破坏第2个条件，进程必须获得所需的所有资源才能运行——采用静态分配策略，但是严重降低资源利用率
  - 静态分配策略：一个进程所需要的全部资源必须在该进程执行前申请并得到后，进程才能执行；如果该进程不能得到全部所需资源，则系统不对该进程进行资源分配，进程必须等待
  - 会严重降低资源利用率，因为在每个进程所有占有的资源中，有些资源在运行后期才用甚至在例外情况下才用

2.7.2 死锁产生的必要条件

- 破坏第3个条件，采用剥夺式调度方法，只适用于CPU和内存分配
  - 实现起来复杂，且要付出很大的代价，而且反复申请和释放资源，延长进程周转时间，增加系统开销，同时降低了系统的吞吐量

2.7.2 死锁产生的必要条件

- 破坏条件4，采用层次分配策略
  - 资源被分成多个层次，当进程得到某一层的一个资源后，它只能再申请较高层次的资源，要释放某层的一个资源时，必须先释放占有的较高层次的资源
  - 层次分配的变种，按序分配策略：把系统的所有资源排序，例如，系统若共有n个进程，共有m个资源（R_1，R_2，……，R_m），规定进程不得在占用资源R_j后再申请R_i（i<j）

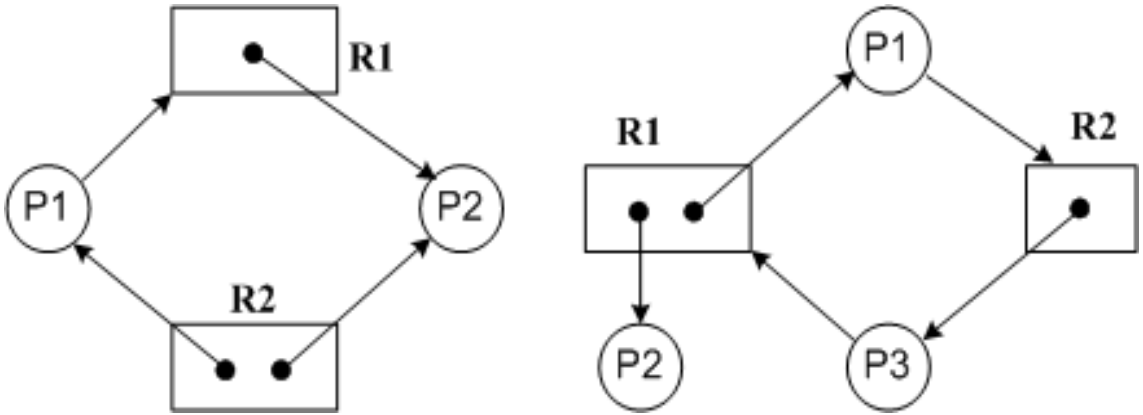
2.7.2 死锁产生的必要条件

- 破坏条件4，采用层次分配策略
  - （设i<j），假设两个进程A和B死锁，原因是A获得R_i并请求R_j，而B获得R_j并请求R_i，则在按序分配策略下，这种情况是不可能的
- 缺点：低效，会限制新设备类型的增加，使进程执行速度变慢，并可能在没有必要的情况下拒绝访问某资源

2.7.2 死锁产生的必要条件

- 进程死锁问题可用**系统资源分配图描述**：
  - **圆圈表示进程，资源类用方框表示**，框中的圆点代表单个该类资源，使用有向边连接进程和资源。
  - **申请边从进程指向资源类方框，表示进程正在等待资源；分配边从单个资源圆点指向进程，表示进程已经获得资源。**

2.7.2 死锁产生的必要条件



系统资源分配图

2.7.2 死锁产生的必要条件

- **根据进程-资源分配图得出如下结论：**

  （1）如果进程-资源分配图中无环路，则此时系统没有发生死锁

  （2）如果进程-资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充要条件，环路中的进程便为死锁进程

  （3）如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件

2.7.3 死锁的避免

- 死锁避免算法是通过资源分配算法分析系统是否存在一个并发进程的状态序列，在确定不会产生进程循环等待的情况下，才将资源真正分配给进程，以保证并发进程不会产生死锁
- 如果进程的资源请求方案会导致死锁，系统拒绝执行；如果一个资源的分配会导致死锁，系统拒绝分配

2.7.3　死锁的避免

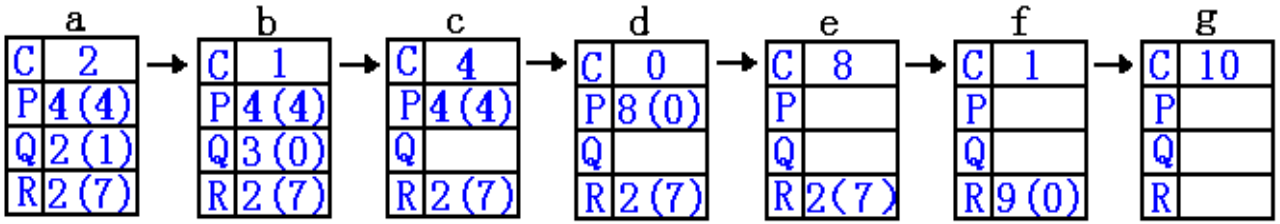
- Dijkstra在1965年提出了避免死锁的**银行家调度算法**，该算法是以银行系统所采用的借贷策略（尽可能放贷、尽快回收资金）为基础而建立的算法模型

2.7.3　死锁的避免

- 例：假定银行有可供借用资金10法郎，现有三个用户P、Q、R来银行办理借款业务，他们最大借款总额分别为8法郎，3法郎和9法郎；
- 又假定在某时刻状态为：P已借4法郎尚需4法郎；Q已借2法郎尚需1法郎；R已借2法郎尚需7法郎，目前银行余款为2法郎，请问银行家该如何借出余款才能避免破产？

2.7.3 死锁的避免

按下图可以让所有顾客完成借款任务



在上图中，由状态a转入状态b，若先将1法郎借给R用户，使R变为3(6)状态，系统便会进入死锁。

银行家算法就是根据上例得出的。

2.7.3 死锁的避免

- 在此模型中，进程相当于贷款客户，系统资源相当于资金，调度程序相当于银行家
- 系统对进程提出以下约束条件：

（1）每个进程必须事先声明其资源需求

（2）每个进程每次提出部分资源申请并获得分配

（3）进程获得所需资源，执行完毕后，必须及时将所占资源归还系统

2.7.3 死锁的避免

- 在进程保证遵守上述约束的基础上，系统将保证：

1. 如果一个进程所请求最大资源数不超过系统所有的资源总数，则系统一定分配资源给进程
2. 如果系统在收到进程的请求时因资源不足而使进程等待，系统会保证在有限的时间内使进程获得资源

2.7.3　死锁的避免

银行家算法的思路：

（1）在某一时刻，各进程已获得所需的部分资源；有一进程提出新的资源请求，系统将剩余资源试探性地分配给该进程

（2）如果此时剩余资源能够满足余下的某些进程的需求，则将剩余资源分配给能充分满足的、资源需求缺口最大的进程，运行结束后释放的资源再并入系统的剩余资源集合

2.7.3　死锁的避免

银行家算法的思路：

（3）反复执行第2步，直到所有的进程都能够获得所需而运行结束，说明第1步的进程请求是可行的，系统处于安全状态，相应的进程执行序列称为系统的**安全序列**

（4）如果试分配后，依照第2步方法所有的进程都试探过而不能顺利运行结束，则不能满足该进程的资源请求，即不存在安全序列，则系统是**不安全的**

银行家算法所需的数据结构：

- 假设系统中有n个进程，m类资源；
- 系统当前资源剩余量向量

Available[m] = { R_1，R_2，……，R_m }；

- n个进程对m类资源的需求声明矩阵

Claim[n][m]

Claim[i][j]的值表示进程i对j类资源的总需求量

银行家算法所需的数据结构：

- 假设系统中有n个进程，m类资源；
- n个进程已获得的各类资源数量矩阵

  Possession[n][m]，

  Possession[i][j]的值k表示进程i已获得k个j类资源；
- n个进程的各类资源需求缺口矩阵

  Shortage[n][m]=Claim–Possession，

  Shortage[i][j]的值s表示进程i还需要（缺）s个j类资源；

银行家算法所需的数据结构：

- 某进程i在某时刻发出的资源请求向量
  Request[m]，取值随具体情况而定
- 前4个数据结构及其取值确定了系统在某一时刻的状态，如果算法尝试资源分配方案能够使得所有进程安全运行完毕，则说明：

  该状态安全，资源分配方案可行！！！

银行家算法的算法细化说明：

1）判断请求向量Request的有效性——超过相应进程总需求量则报错，超过系统目前剩余量则阻塞；

2）就系统资源剩余量对Request进行试分配：

Available[*] = Available[*] - Request[*];

Possession[i][*] = Possession[i][*] + Request[*];

Shortage[i][*] = Shortage[i][*] - Request[*];

3）执行安全性测试算法，若安全则确认试分配方案，否则进程i阻塞；

执行安全性测试算法细化说明：

① 定义工作向量Rest[*] = Available[*]，进程集合 Running{*}，布尔量possible= true；

② 从Running集合中找出P_k，满足条件 Shortage[k][*] < Rest[*]；

③ 找到合格的进程P_k，则释放其占用资源（Rest[*] = Rest[*] + Possession[k][*]），将其从Running集合中去掉，重复步骤②；

执行安全性测试算法细化说明：

④ 找不到合格的进程P_k，possible为false，退出安全性测试算法；

⑤ 最终检查Running集合，为空则返回安全，非空则不安全。

银行家算法示例：

- 假设系统中有5个进程{ P_0，P_1，P_2，P_3，P_4}，3类系统资源{ A，B，C}，各拥有资源数{10，5，7}，T_0时刻系统状态如表2.8 所示。

表2.8 T_0时刻系统状态

进程ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	3，3，2
P_1	3，2，2	2，0，0	1，2，2	
P_2	9，0，2	3，0，2	6，0，0	
P_3	2，2，2	2，1，1	0，1，1	
P_4	4，3，3	0，0，2	4，3，1	

银行家算法示例：

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	3，3，2
P_1	3，2，2	2，0，0	1，2，2	
P_2	9，0，2	3，0，2	6，0，0	
P_3	2，2，2	2，1，1	0，1，1	
P_4	4，3，3	0，0，2	4，3，1	

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	5，3，2
P_1				
P_2	9，0，2	3，0，2	6，0，0	
P_3	2，2，2	2，1，1	0，1，1	
P_4	4，3，3	0，0，2	4，3，1	

银行家算法示例：

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	5，3，2
P_1				
P_2	9，0，2	3，0，2	6，0，0	
P_3	2，2，2	2，1，1	0，1，1	
P_4	4，3，3	0，0，2	4，3，1	

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	7，4，3
P_1				
P_2	9，0，2	3，0，2	6，0，0	
P_3				
P_4	4，3，3	0，0，2	4，3，1	

银行家算法示例：

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	7，4，3
P_1				
P_2	9，0，2	3，0，2	6，0，0	
P_3				
P_4	4，3，3	0，0，2	4，3，1	

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	7，4，5
P_1				
P_2	9，0，2	3，0，2	6，0，0	
P_3				
P_4				

银行家算法示例：

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	7，4，5
P_1				
P_2	9，0，2	3，0，2	6，0，0	
P_3				
P_4				

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	10，4，7
P_1				
P_2				
P_3				
P_4				

银行家算法示例：

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0	7，5，3	0，1，0	7，4，3	10，4，7
P_1				
P_2				
P_3				
P_4				

进程 ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P_0				10，5，7
P_1				
P_2				
P_3				
P_4				

银行家算法示例：

- 可以断言目前系统处于安全状态，因为存在一个执行序列{P_1，P_3，P_4，P_2，P_0}，能满足安全性条件（结果不唯一）
- 现有进程P_1发出资源请求Request = {1，0，2}。系统为了确定能否满足该请求，采用银行家算法中的安全性测试算法，首先检查Request的有效性：
- Request（1，0，2）≤Shortage[1]（1，2，2）；
- Request（1，0，2）≤Available（3，3，2）；

银行家算法示例：

- 通过有效性检查后，系统按P1的请求进行试分配，此时（T1时刻）系统状态如表2.9 所示

表2.9 T1时刻系统状态

进程ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P0	7，5，3	0，1，0	7，4，3	2，3，0
P1	3，2，2	3，0，2	0，2，0	
P2	9，0，2	3，0，2	6，0，0	
P3	2，2，2	2，1，1	0，1，1	
P4	4，3，3	0，0，2	4，3，1	

银行家算法示例：

- 经分析，仍存在一个执行序列{P1，P3，P4，P0，P2}，能满足安全性条件，因此刚才的试分配方案是可行的
- 现又有进程P4发出资源请求向量（3，3，0），因系统剩余资源向量Available（2，3，0）小于该请求向量，所以无法通过有效性检查，P4进程阻塞

银行家算法示例：

- 然后P0发出资源请求向量（0，2，0），虽可通过有效性检查，但试分配后，系统的剩余资源不能满足任何进程的需求缺口，因而无法找到一个执行序列，将导致系统进入不安全状态，所以不能按 P0 的请求进行资源分配

银行家算法示例：

- P0：（0，2，0），

进程ID	Claim	Possession	Shortage	Available
	A，B，C	A，B，C	A，B，C	A，B，C
P0	7，5，3	0，3，0	7，2，3	2，1，0
P1	3，2，2	3，0，2	0，2，0	
P2	9，0，2	3，0，2	6，0，0	
P3	2，2，2	2，1，1	0，1，1	
P4	4，3，3	0，0，2	4，3，1	

- 银行家算法是一个很经典的死锁避免算法，理论性很强，看起来似乎很完美，但其实现要求进程不相关，并且事先要知道进程总数和各进程所需资源情况，所以可行性并不高

2.7.4 检测与解除

- 在许多系统中并未刻意去预防和避免死锁，对资源分配不施加任何限制
- 让系统定时运行一个死锁检测程序，判断系统内是否有死锁发生
- 如果发生了死锁，再采取措施解除死锁
- 死锁的检测算法可以采用基于死锁定理的检测，也可以采用类似于银行家算法中的安全性测试算法

2.7.4 检测与解除

- 在系统中，需要决定死锁检测的频率，如果检测太频繁，会花大量的时间检测死锁，浪费CPU的处理时间；如果检测的频率太低，死锁进程和系统资源被锁定的时间过长，资源浪费大
- 通常的方法是在CPU的使用率下降到一定的阈值时实施检测。当死锁发生次数多，死锁进程数增加到一定程度时，CPU的处理任务减少，CPU空闲时间增多

2.7.4 检测与解除

系统检测到死锁后，常用的死锁解除方法有：

- 重启：重新启动死锁进程，甚至重启操作系统
- 撤销：撤销死锁进程，回收资源，优先选择占用资源最多或者撤销代价最小的，撤销一个不行就继续选择撤销进程，直至解除死锁
- 剥夺：剥夺死锁进程资源再分配，选择原则同上
- 回滚：根据系统保存的检查点，使进程或系统回退到死锁前的状态

2.7.4 检测与解除

- 虽然大多数操作系统都潜在地受到死锁的威胁，但是，从解决死锁的角度出发，需要了解死锁发生的频率、系统受死锁影响的程度
- 如果死锁发生的频率高，对系统的影响很大，甚至导致系统崩溃，则应该采取有力的措施去预防死锁、避免死锁，不惜代价检测死锁并解除死锁
- 但如果死锁几个月才发生一次，则不会选择牺牲系统性能和可用性去检测死锁、解除死锁

2.8 线程的基本概念

- 线程：80年代中期出现，比进程更小的运行基本单位，能够提高系统内程序并发执行的速度，减少系统开销，从而可进一步提高系统的吞吐量。
- 近几年，线程概念得到了广泛应用，不仅在新推出的操作系统中，大都已引入了线程概念，而且在新推出的数据库管理系统和其它应用软件中，也都采用多线程技术来改善系统的性能。

2.8.1 线程的引入

- 操作系统中引入**进程**的目的，是为了使多个程序并发执行，以改善资源利用率及提高系统的吞吐量。
- 操作系统中再引入**线程**则是为了减少程序并发执行时所付出的时空开销，使操作系统并发粒度更小、并发性更好。

2.8.1 线程的引入

- 进程是操作系统资源分配和调度执行的基本单位，一个能独立运行的实体。
- 系统创建进程时，应为之分配其所必需的、除CPU以外的所有资源，如内存、I/O设备以及建立相应的PCB。
- 系统撤消进程时，必须先对这些资源进行回收操作，然后再撤消PCB。
- 进程切换时，要保留当前进程的CPU环境和设置新进程的CPU环境，为此须花费许多CPU时间。

2.8.1 线程的引入

- 总之，由于进程是一个资源拥有者，因而在进程的创建、撤消和切换中，系统必须为之付出较大的时空开销。也正因为如此，在系统中所设置的进程数目不宜过多，进程切换的频率也不宜太高，但这也就限制了并发程度的进一步提高。

中国矿业大学

2.8.1 线程的引入

- 为了使多个程序更好地并发执行，同时又尽量减少系统的开销，要把CPU调度和资源分配针对不同的活动实体进行，以使之轻装运行，而对拥有资源的基本单位，又不频繁地对之进行切换。
- 在这种思想的指导下，产生了线程的概念。

2.8.1 线程的引入

- 线程是操作系统进程中能够独立执行的实体（控制流），是处理器调度和分派的基本单位。
- 线程是进程的组成部分，线程只拥有一些在运行中必不可少的资源（如程序计数器、一组寄存器和栈），与同属一个进程的其它线程共享进程所拥有的全部资源。
- 一个线程可以创建和撤消另一个线程；同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中也呈现出间断性。

2.8.1 线程的引入

- 线程有**就绪、阻塞和执行**三种基本状态，有的系统中线程还有终止状态。
- 挂起状态对线程是没有意义的，如果进程挂起后被对换出主存，则其所有线程因共享了进程的地址空间，也必须全部对换出去。
- 线程也有一一对应的描述和控制数据结构TCB（Thread Control Block）。不同进程间的线程互不可见，同一进程内的线程间通信主要基于全局变量。

2.8.2 线程与进程的区别与联系

- 线程又称为轻型进程(Light-Weight Process)或进程元；而传统的进程称为重型进程(Heavy-Weight Process)，它相当于只有一个线程的任务。
- 在引入了线程的操作系统中，通常一个进程都有若干个线程，至少需要有一个主线程。
- 在调度、并发性、系统开销、拥有资源等方面，线程与进程有所区别。

中国矿业大学

2.8.2 线程与进程的区别与联系

- 在传统的操作系统中，拥有资源的基本单位和独立调度的基本单位都是进程。在引入线程的操作系统中，则把线程作为调度和分配的基本单位，而把进程作为拥有资源的基本单位，使传统进程的两个属性分开，线程便能轻装运行，从而可显著地提高系统的并发程度。
- 在同一进程中，线程的切换不会引起进程的切换，在由一个进程中的线程切换到另一个进程中的线程时，将会引起进程的切换。

2.8.2 线程与进程的区别与联系

- 不论是传统的操作系统，还是支持线程的操作系统，进程都是拥有资源的独立单位，拥有自己的资源。
- 一般地说，线程自己不拥有系统资源(只有一些必不可少的资源)，但可以访问、共享其隶属进程的资源——进程的代码段、数据段以及打开的文件、I/O设备等。

2.8.2 线程与进程的区别与联系

- 在创建或撤消进程时，操作系统所付出的开销显著大于创建或撤消线程时的开销。
- 在进行进程切换时，涉及到当前进程整个CPU环境的保存以及新被调度运行的进程的CPU环境的设置。
- 而线程切换只需保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。可见，进程切换的开销也远大于线程切换的开销。

2.8.2 线程与进程的区别与联系

- 由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现，也变得比较容易。在有的系统中，线程的切换、同步和通信都无需操作系统内核的干预。

2.8.3 线程的三种实现方式

① 在操作系统内核实现的内核级线程（Kernel Level Thread，KLT），如Windows，OS/2等。

- 线程管理的全部工作由操作系统内核在内核空间实现。系统为应用开发使用内核级线程提供了一组API函数，通过调用API函数，即可实现线程的创建和控制管理。
- 内核建立和维护进程的进程控制块（PCB），线程控制块（TCB），内核的调度是在线程的基础上进行的。

2.8.3 线程的三种实现方式

- 内核级线程实现的优点：在多处理器上，内核能够同时调度同一进程的多个线程并行执行；如果进程中的一个线程阻塞，内核能够调度同一进程的其它线程占有处理器；内核级线程的数据结构和堆栈均较小，内核级线程的切换快，提高了处理器的效率；内核级线程自身可以用多线程技术实现，提高了系统的并行性和执行速度。

2.8.3 线程的三种实现方式

- 内核级线程实现的缺点：应用程序的线程运行在用户空间，而线程调度和管理在内核空间，即使是同一进程运行，当对线程的控制需要从一个线程传送到另一个线程时，也要经过用户态到核心态，再从内核态到用户态的模式切换，系统开销较大。

2.8.3 线程的三种实现方式

② 在用户空间实现的用户级线程（User Level Thread，ULT），如Java的线程库等。

- 线程管理的全部工作由应用程序在用户空间实现，内核不知道线程的存在。用户级线程由用户空间运行的用户级线程库实现，应用开发通过线程库进行程序设计，用户级线程库是线程运行的支撑环境

2.8.3 线程的三种实现方式

- 优点：线程切换不需要内核特权方式，应用开发可自主选择调度算法，能够运行在任何操作系统上。
- 缺点：如果一个线程被阻塞，则该线程所在进程的所有线程都将被阻塞，不能选择该进程的其他线程运行，浪费处理器资源；系统在进程调度时分配给进程的处理器时间需要由所有的线程分享，如果系统有多个处理器，也不能实现在一个进程中的线程并行执行。

2.8.3 线程的三种实现方式

③ 同时支持两种线程的混合式线程实现，如Solaris系统。

- 混合式线程既可以具备内核级线程实现的优点，又可以具备用户级线程实现的优点。

小 结

- 本章引入进程、线程、并发、调度、同步、死锁等概念。
- 介绍了进程状态与转换，进程调度算法、进程同步机制与通信机制、银行家算法等经典理论，最后介绍了线程管理。
- 本章是操作系统课程的核心和难点所在。