操作系统

1. 说出五种进程调度算法，Windows 和 Linux 采用的是哪一种

> **1.先来先服务 first-come first-serverd（FCFS）**
>
> 按照请求的顺序进行调度。非抢占式，开销小，无饥饿问题，响应时间不确定（可能很慢）；
>
> 对短进程不利，对IO密集型进程不利。
>
> **2.最短作业优先 shortest job first（SJF）**
>
> 按估计运行时间最短的顺序进行调度。非抢占式，吞吐量高，开销可能较大，可能导致饥饿问题；
>
> 对短进程提供好的响应时间，对长进程不利
>
> **3.优先级调度算法**
>
> 为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。
>
> 4.**时间片轮转**
>
> 将所有就绪进程按 FCFS 的原则排成一个队列，用完时间片的进程排到队列最后。抢占式（时间片用完时），开销小，无饥饿问题，为短进程提供好的响应时间；

> 若时间片小，进程切换频繁，吞吐量低；若时间片太长，实时性得不到保证。
>
> 5.**最高响应比优先**
>
> 响应比 = 1+ 等待时间/处理时间。同时考虑了等待时间的长短和估计需要的执行时间长短，很好的平衡了长短进程。非抢占，吞吐量高，开销可能较大，提供好的响应时间，无饥饿问题。
>
> **6.多级反馈队列调度算法**
>
> 设置多个就绪队列1、2、3...，优先级递减，时间片递增。只有等到优先级更高的队列为空时才会调度当前队列中的进程。如果进程用完了当前队列的时间片还未执行完，则会被移到下一队列。
>
> 抢占式（时间片用完时），开销可能较大，对IO型进程有利，可能会出现饥饿问题。

Windows采用的是优先级调度算法

Windows处理器调度机制

一. 调度优先级

1. Windows处理器调度力度为**线程**，Windows为每一个线程分配调度优先级
2. 调度器根据优先级采用**抢占式调度策略**，让具有最高优先级的线程首先执行
3. Windows内核使用**32个优先级**分别来表示线程要求执行的紧迫性，用0~31的数字表示
4. 根据优先级的功能不同，可以分为3组：16个实时优先级别（16~31），15个可变优先级（1~15），1个系统优先级（0），为内存页清零线程保留
5. 当创建进程时，可以赋予一下优先级别：实时，高，高于一般，一般，低于一般和空闲
6. 当创建线程时，在进程的优先级别进一步赋予一下优先级别：尽量实时，最高，高于一般，一般，低于一般，最低和空闲
7. 应用优先级和系统的优先级别对应关系

36	15	12	10	8	6
35	14	11	9	7	5
24	13	10	8	6	4
23	12	9	7	5	3
22	11	8	6	4	2
实时	高	高于一般	一般	低于一般	空闲

8. 处理器调度时参考两个优先级设置：**一个是从当前线程所在的进程的基准优先级，另外一个是线程的优先级**
9. 一般来说，应该线程运行在可变优先级别（1~15）的范围内，如果需要进入实时优先级别（16~31）范围来运行，必须取得更高的调度优先级特权

Linux2.6采用的是：

对于一般进程，优先级调度

对于实时进程，先来先服务和时间片轮转

2.6调度规则

调度器早期2.6 把linux内核空间的进程优先级化为0-139之间，优先级数值越小优先级越高。0-99优先级的进程调度使用的是RT策略，100-139优先级的进程调度是非RT策略的。

0-99的RT策略分为两种：

- SCHED_FIFO ： 不同优先级按照优先级高先跑到睡眠，优先级低的再跑，同等优先级先进先出。
- SCHED_RR：不同优先级按照优先级高先跑到睡眠，优先级低的再跑，同等优先级轮转。

当全部的0-99的进程均跑到睡眠状态，操作系统就会调度到优先级在100-139的普通进程进行跑。

普通进程是按照nice进行调度的，nice值越高优先级最低。nice值计算 = 优先级 - 120 。比如优先级为100的进程，其nice值为-20，而优先级为139的进程其nice值为+19；

对于普通进程，不会因为你优先值高而可以一直独占cpu资源，而是会进行轮转。那么优先级对于普通进程有啥作用呢？主要有两点作用：

- 1、轮转时候获取更多时间片，
- 2、在刚刚醒来的时刻可以抢占优先级低的进程。**nice值越高优先级最低。**

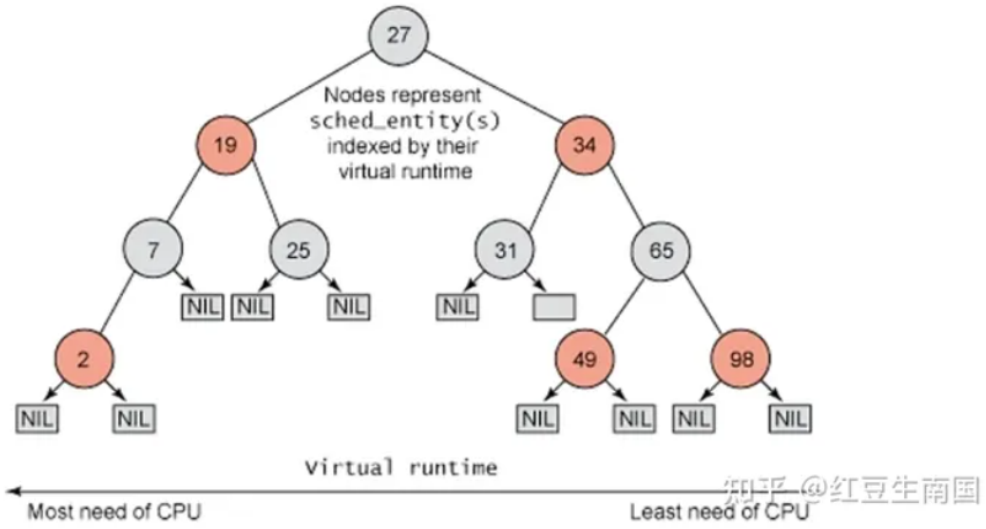
当我们在linux中给进程设置一个静态nice值时候，但是我们知道linux要优先调度IO型进程，我们可以在启动进程时候设置nice值，在2.6版本中linux会在进程运行时候根据进程的睡眠/计算等情况，来对进程进行奖励和惩罚。linux会探测你是喜欢等待还是喜欢计算，越睡优先级越高。（希望IO类型进程被优先调度到）

后期，Linux对普通进程提出了CFS算法，也就是完全公平调度算法

2、CFS公平调度算法

linux针对普通进程提出了cfs算法。完全公平调度。要照顾到普通进程的完全公平。用到的数据结构是红黑树：红黑树，左边节点小于右边节点的值。

- 运行到目前为止vruntime最小的进程
- 同时考虑了CPU/I0和nice



图中的2、7、19等数字代表 进程运行到目前为止的运行时间
$vruntime = phyruntime/weight * 1024$，phyruntime为物理运行时间，weigth跟nice值有关（有个映射表 nice值越低weight越高）。**每次 linux调度的时候 会调度到目前为止vruntime最小的进程。**

这个算法完全照顾了nice值低的/io消耗型进程等；

- 照顾了io消耗型进程：因为io消耗性喜欢谁，所以phyruntime值小，那么整体vruntime值小就容易比调度到；
- 照顾了nice值低的进程：nice值低，权重越高，那么vruntime就越低，就约容易被调度到；

2. 简要介绍自己的 OS 实验课做了什么

实验内容

本书设计的操作系统实验分为 6 个实验（Lab1 ~ Lab6），目标是在一学期内自主开发一个小型操作系统。各个实验的相互关系如图 1 所示，具体实验内容如下。

1. **内核、启动和 `printf`：**通过 PC 启动的实验，掌握硬件的启动过程，理解链接地址、加载地址和重定位的概念，学习如何编写裸机代码。
2. **内存管理：**理解虚拟内存和物理内存的管理，实现操作系统对虚拟内存空间的管理。
3. **进程与异常：**通过设置进程控制块和编写进程创建、进程中止和进程调度程序，实现进程管理；编写通用中断分派程序和时钟中断例程，实现中断管理。
4. **系统调用与 `fork`：**掌握系统调用的实现方法，理解系统调用的处理流程，实现本实验所需的系统调用。
5. **文件系统：**通过实现一个简单的、基于磁盘的、微内核方式的文件系统，掌握文件系统的实现方法和层次结构。
6. **管道与 shell：**实现具有管道，重定向功能的命令解释程序 shell，能够执行一些简单的命令。最后将 6 部分链接起来，使之成为一个能够运行的操作系统。

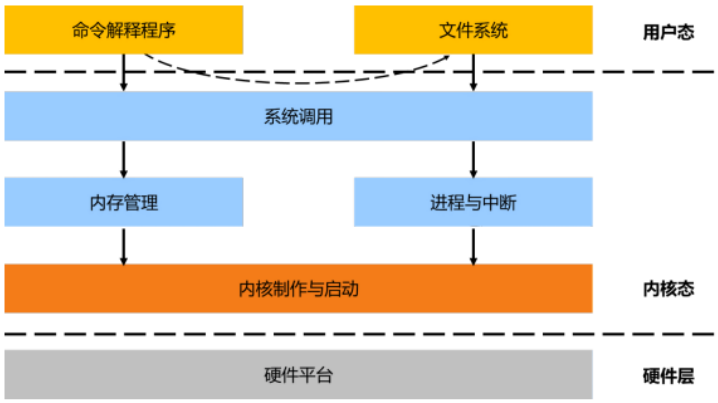


图 1: 实验内容的关系

另外，考虑有些学生对 Linux 系统、GCC 编译器、Makefile 和 git 等工具不熟悉，专门设置了一个 Lab0，主要介绍 Linux、Makefile、git、vi 和仿真器的使用以及基本的 shell 编程等，为后续实验的顺利实施打好基础。

在OS实验课上，我们实现了一个可以在 MIPS 平台上运行的小型操作系统，包括操作系统启动、物理内存管理、虚拟内存管理、进程管理、中断处理、系统调用、文件系统、Shell 等操作系统的主要功能。

3. **进程有哪些状态，它们之间如何转换**
   - 运行（running）态：进程占有处理器**正在运行的状态。**进程已获得CPU，其程序正在执行。在单处理机系统中，只有一个进程处于执行状态； 在多处理机系统中，则有多个进程处于执行状态。

- 就绪（ready）态：进程具备运行条件，等待系统分配处理器以便运行的状态。**当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行**，进程这时的状态称为就绪状态。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列。
- 等待（wait）态：又称阻塞态或睡眠态，指进程不具备运行条件，正在等待某个事件完成的状态。也称为等待或睡眠状态，一个进程正在等待某一事件发生（例如请求I/O而等待I/O完成等）而暂时停止运行，这时即使把处理机分配给进程也无法运行，故称该进程处于阻塞状态。

五态模型在三态模型的基础上增加了新建态（new）和终止态（exit）：

- 新建态：对应于进程被创建时的状态，尚未进入就绪队列。创建一个进程需要通过两个步骤：1.为新进程分配所需要的资源和建立必要的管理信息。2.设置该进程为就绪态，并等待被调度执行。
- 终止态：指进程完成任务到达正常结束点，或出现无法克服的错误而异常终止，或被操作系统及有终止权的进程所终止时所处的状态。处于终止态的进程不再被调度执行，下一步将被系统撤销，最终从系统中消失。终止一个进程需要两个步骤：1.先对操作系统或相关的进程进行善后处理（如抽取信息）。2.然后回收占用的资源并被系统删除。

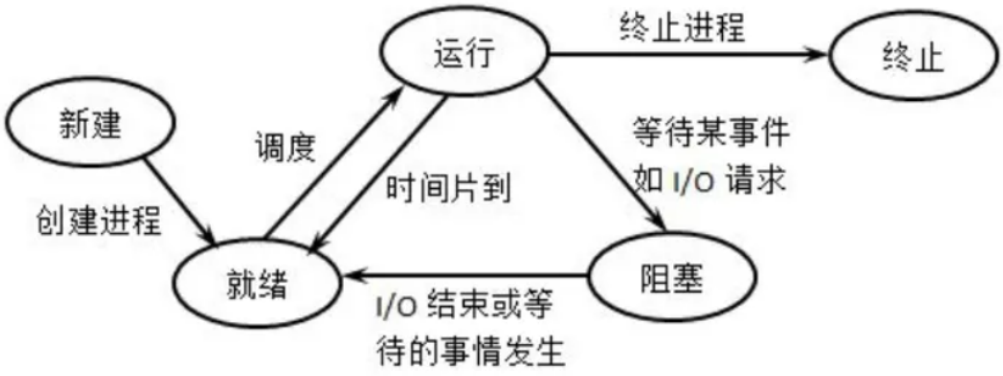


图 2. 进程的五态模型

引起进程状态转换的具体原因如下：

- 运行态→阻塞态：等待使用资源；如等待外设传输；等待人工干预。
- 阻塞态→就绪态：资源得到满足；如外设传输结束；人工干预完成。
- 运行态→就绪态：运行时间片到；出现有更高优先权进程抢占CPU。
- 就绪态→运行态：CPU 空闲时选择一个就绪进程。

4. **时间片到了进程还没执行完，变成什么状态**

   就绪态。

5. **进程同步的意义**

   我们把异步环境下的一组并发进程因直接制约而互相发送消息、进行互相合作、互相等待，使得各进程按一定的速度执行的过程称为进程间的同步。

   进程同步的主要任务是使**并发执行的诸进程**之间能有效地**共享资源**和**相互合作**，使执行的结果具有**可再现性**。

   进程同步的主要意义包括以下几点：

   1. **数据一致性**：在一组并发进程**访问共享资源**（如内存、文件、数据库等）时，需要确保数据的一致性，避免出现竞争条件和数据损坏。进程同步可以确保多个进程按照一定的顺序来访问共享资源，从而避免数据不一致的问题。
   2. **程序正确性**：进程同步有助于确保程序的正确性，尤其是在多线程和多进程环境下。
   3. **资源分配和调度**：进程同步也可以用于资源的合理分配和调度，以确保每个进程都有机会访问所需的资源，从而提高系统的效率和性能。

   同步机制规则：

   空闲让进：当无进程处于临界区时，表明临界资源处于空闲状态，应允许一个请求进入临界区的进程立即进入自己的临界区，以有效地利用临界资源。（多中选一）
   忙则等待：当已有进程进入临界区时，表明临界资源正在被访问，因而其它试图进入临界区的进程必须等待，以保证对临界资源的互斥访问。（互斥访问）
   有限等待：　对要求访问临界资源的进程，应保证在有限时间内能进入自己的临界区，避免陷入"死等"状态。　(避免死等)
   让权等待：　当进程不能进入自己的临界区时，应立即**释放处理机**，避免进程陷入"忙等"状态。　(避免忙等)

6. **Windows / Linux 下进程间通信的手段**

进程间通信：管道，共享内存，消息队列，信号，套接字(Sockets)

Windows:

1. **命名管道（Named Pipes）**：命名管道是一种在不同进程之间进行通信的方法，其中一个进程将数据写入管道，而另一个进程则可以从管道中读取数据。
2. **邮件槽（Mailslots）**：邮件槽是一种用于进程之间传递消息的通信机制，适用于在本地或网络上进行进程通信。
3. **共享内存（Shared Memory）**：Windows允许进程创建共享内存区域，多个进程可以将数据写入和读取自这些共享内存区域。
4. **Windows消息队列（Message Queues）**：进程可以使用Windows消息队列来发送和接收消息，通常用于GUI应用程序之间的通信。
5. **套接字（Sockets）**：虽然套接字通常与网络通信相关，但它们也可以用于本地进程之间的通信，例如使用本地回环地址进行通信。
6. **Windows进程间通信（Interprocess Communication，IPC）API**：Windows提供了一系列IPC API，如事件、信号量、互斥体等，用于进程之间的同步和通信。

Linux:

1. **管道（Pipes）**：Linux管道是一种进程间通信的简单方式，用于在两个相关进程之间传递数据。
2. **命名管道（Named Pipes）**：类似于Windows的命名管道，允许不相关的进程之间进行通信。
3. **共享内存（Shared Memory）**：Linux允许进程创建共享内存段，多个进程可以将数据写入和读取自这些共享内存段。
4. **消息队列（Message Queues）**：Linux提供了消息队列，允许进程通过队列来发送和接收消息。
5. **信号（Signals）**：信号是Linux中一种轻量级的进程间通信方式，进程可以发送信号给其他进程来通知它们发生了某些事件。
6. **套接字（Sockets）**：套接字不仅用于网络通信，还可以用于本地进程之间的通信，例如使用本地套接字地址。
7. **System V IPC**：Linux还提供了System V IPC机制，包括信号量、共享内存和消息队列，用于更复杂的进程通信需求。
8. **DBus（Desktop Bus）**：DBus是一种用于Linux桌面应用程序之间通信的高级IPC机制，通常用于跨进程的桌面集成。

附注：

Linux线程间通信：互斥体，信号量，条件变量

Windows线程间通信：临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphore）、事件（Event）

7. **DMA 的全称是什么，解决了什么问题**

   DMA全程是“Direct Memory Access”的缩写，中文意为“直接内存访问”。DMA是一种计算机系统中的技术和硬件机制，用于解决主要的问题是**减轻CPU对数据传输的负担**以及**提高数据传输的效率。**

   DMA解决了以下两个主要问题：

   1. **减轻CPU负担**：在计算机系统中，数据传输通常需要从外部设备（如硬盘、网络适配器、显卡等）到内存，或者从内存到外部设备。在没有DMA的情况下，CPU通常需要直接参与这些数据传输，即从设备读取数据或将数据写入设备，这会占用CPU的大量时间和资源。DMA**允许外部设备通过主板上的DMA控制器直接访问系统内存，而不需要CPU的干预。**这意味着CPU可以继续执行其他任务，而不会被数据传输操作所阻塞，从而提高了系统的整体性能和响应速度。
   2. **提高数据传输效率**：DMA通过直接内存访问，能够更快速和高效地执行数据传输操作。与CPU相比，**DMA控制器通常更快速，可以在不引起CPU负载的情况下执行数据传输。**这对于需要高吞吐量的任务（如音视频处理、数据备份、网络通信等）尤为重要。

   DMA解决了CPU参与数据传输时的性能瓶颈问题，提高了系统的整体性能和效率。它允许外部设备直接与系统内存交互，而无需CPU的干预，减轻了CPU的负担，并提高了数据传输的速度和效率。

8. **死锁的条件、解决办法、预防办法**

   死锁的条件：

   1. **互斥条件**
      临界资源是独占资源，进程应互斥且排他的使用这些资源。

2. **占有和等待条件**
   进程在请求资源得不到满足而等待时，不释放已占有资源。
3. **不剥夺条件**
   资源不能被强制性地从一个进程或线程中剥夺，只能由占有资源的进程自愿释放。
4. **循环等待条件**
   多个进程之间形成一个资源等待的环路。例如，进程A等待进程B占有的资源，进程B等待进程C占有的资源，而进程C又等待进程A占有的资源。这种循环等待条件导致了死锁。

解决办法：

- **死锁预防**

死锁预防的策略是：破坏死锁形成的条件之一

1. **破坏互斥条件**

使资源同时访问而非互斥使用，就没有进程会阻塞在资源上，从而不发生死锁。

> 只读数据文件、磁盘等软硬件资源均可采用这种办法管理；
> 但是许多资源是独占性资源，如可写文件、键盘等只能互斥的占有；
> 所以这种做法在许多场合是不适用的。

2. **破坏占有和等待条件**

采用**静态分配**的方式，静态分配的方式是指进程必须在执行之前就申请需要的全部资源，且直至所要的资源全部得到满足后才开始执行。

> 实现简单，但是严重的减低了资源利用率。
> 因为在每个进程占有的资源中，有些资源在运行后期使用，有些资源在例外情况下才被使用，可能会造成进程占有一些几乎用不到的资源，而使其他想使用这些资源的进程等待。

3. **破坏不剥夺条件**

剥夺调度能够防止死锁，但是只适用于**内存和处理器资源**。

方法一：占有资源的进程若要申请新资源，必须**主动释放已占有资源**，若需要此资源，应该向系统重新申请。

方法二：资源分配管理程序为进程分配新资源时，若有则分配；否则将**剥夺此进程已占有的全部资源，并让进程进入等待资源状态**，资源充足后再唤醒它重新申请所有所需资源。

4. **破坏循环等待条件**

给系统的**所有资源编号，规定进程请求所需资源的顺序必须按照资源的编号依次进行。**

> 采用层次分配策略，将系统中所有的资源排列到不同层次中

一个进程得到某层的一个资源后，只能申请较高一层的资源

当进程释放某层的一个资源时，必须先释放所占有的较高层的资源

当进程获得某层的一个资源时，如果想申请同层的另一个资源，必须先释放此层中已占有的资源

- **死锁避免**

  安全状态定义：至少存在一个资源分配序列不会导致死锁，即所有进程都能够顺利运行到结束，不安全状态指非安全的一个状态。

  银行家算法（Banker's Algorithm）：银行家算法是一种静态死锁避免算法，最初由Edsger Dijkstra提出。它**基于资源的最大需求**、**已分配资源**和**可用资源**来决定**是否分配资源给进程。只有在分配资源不会导致系统进入不安全状态时，才会允许资源的分配。**这个算法适用于有限数量的资源和进程。

- **死锁检测和恢复**

  1. 死锁检测

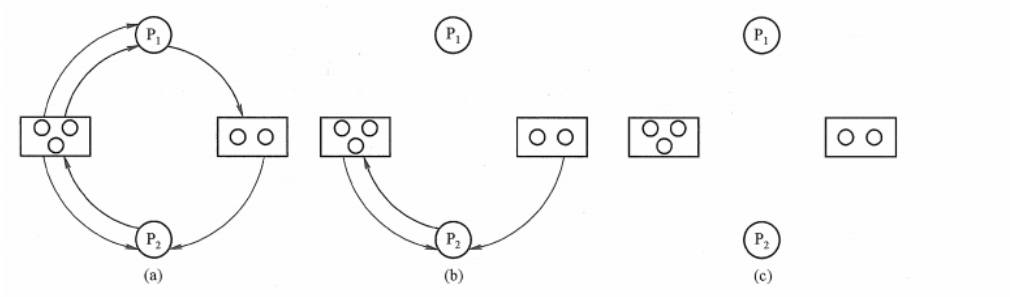
     资源分配图算法用于动态地检测和避免死锁。

     资源分配图中，圆圈代表一个进程，方框代表一类资源，方框内的圆圈代表该类资源的一个单位的资源。它使用资源分配图来表示系统中资源和进程之间的关系。进程请求资源时，会在图中添加一条请求边；释放资源时，会删除相应的边。

     简化资源分配图来判断是否存在死锁，也称**死锁定理，具体步骤如下：**

1）在资源分配图中，找出既不阻塞又不孤立的进程Pi（既找出一条有向边与它相连，且该有向边对应的资源的申请量小于等于系统中已有的空闲资源数量），消去它所有的请求边和分配边，使之成为孤立的节点。

2）进程Pi所释放的资源，可以唤醒某些因等待这些资源而阻塞的进程，原来的阻塞进程可能变为非阻塞进程。



大家一起看一下图二，按照死锁定理中，第一步找出的进程为P1，因为它申请的资源还有，故等P1进程完了后，将P1持有的进程资源释放之后，将P1的申请边和持有边给删除，就如(b)图，此时，P2申请的资源可以得到满足，所以，P2可以正常的执行，所以P2的相关边也可以删除，最后，**整个图中没有有向边的存在，那么就说明没有死锁**，相反，如果说资源分配图，不可以进行简化，那么就说明系统出现了死锁。

2. 死锁解除

1）资源剥夺法。将一些死锁进程暂时挂起来，并且抢占它的资源，并将这些资源分配给其他的死锁进程 ，要注意的是应该防止被挂起的进程长时间得不到资源而处于资源匮乏的状态

2）撤销进程法。强制撤销部分甚至全部死锁并剥夺这些进程的资源。撤销的原则可以按照进程优先级和撤销进程的代价高低进行。

3）进程回退法，让一或多个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而非被剥夺。这个方法要求系统保持进程的历史信息，并设置还原点。

9. **介绍银行家算法，实际场景下是否使用**

银行家算法（Banker's Algorithm）是一种用于死锁避免的静态算法，最初由计算机科学家Edsger Dijkstra于1965年提出。该算法得名于一个类比，将计算机系统比作银行，进程请求资源类比为顾客请求贷款。银行家算法的主要目标是确保系统不会进入可能导致死锁的状态。

银行家算法基于以下三个关键概念：

1. **可用资源向量（Available）**：表示当前系统中每种资源的可用数量。在算法执行过程中，该向量会动态更新，以反映资源的实际可用情况。
2. **最大需求矩阵（Need）**：对于每个进程，表示它对每种资源的最大需求量。这个矩阵在系统启动时就已经确定。
3. **已分配矩阵（Allocation）**：对于每个进程，表示它已分配的资源数量。这个矩阵表示当前进程已经获得的资源。

银行家算法的基本思想是，**只有当分配资源不会导致系统进入不安全状态时，才允许分配资源给进程。**安全状态是指系统中有足够的资源可以满足所有进程的最大需求，而不会发生死锁。银行家算法按照以下步骤进行资源分配决策：

1. 当进程请求资源时，系统首先检查是否有足够的可用资源满足该请求。如果有足够资源，则分配资源给进程，同时减少可用资源和已分配资源。
2. 然后，系统模拟释放资源后的情况，检查是否仍然存在一个安全序列，即可以满足所有进程的最大需求，而不会发生死锁。如果存在安全序列，就允许分配资源；否则，拒绝分配资源。
3. 如果拒绝分配资源，进程必须等待，直到有足够的资源可用。一旦满足条件，分配资源并继续执行。

银行家算法的优点是可以有效地避免死锁，只有在安全的情况下才会分配资源，从而保护系统的稳定性。然而，银行家算法也有一些限制和缺点：

- 需要**提前知道每个进程的最大需求**，这在某些情况下可能难以确定。
- 算法本身需要消耗额外的计算资源来**模拟资源分配**，可能**降低系统性能。**
- 算法假设**资源请求是静态的，不会发生变化。**在动态环境中，算法可能会变得复杂。

由于目前难以明确每个进程的最大需求，并且资源请求可能是动态的，进程的数量也是动态的，加之该算法可能降低系统性能，所以银行家算法的实用性并不高。在上世纪的大型机中进行应用还是可能的。

10. **Cache 的替换策略**

（内存换页的策略也是一样的）

Cache的容量是有限的，当Cache的空间都被占满后，如果再次发生缓存失效，就必须选择一个缓存块来替换掉。常用的替换策略有以下几种：

1. **最近最少使用（Least Recently Used，LRU）**：LRU替换策略选择**最近最久未被使用的缓存块**进行替换。它基于数据的使用历史，认为最久未被使用的数据最有可能在未来也不会被使用。实现LRU替换可能需要额外的硬件支持，例如使用双向链表或栈来维护数据的使用顺序。
2. **先进先出（FIFO）**：FIFO替换策略选择最早进入缓存的数据块进行替换。这是一种简单的替换策略，但它不一定能够反映出数据的实际使用情况，因为最早进入缓存的数据可能在未来仍然频繁使用。
3. **二次机会（Second-Chance）**：二次机会替换策略是一种改进的FIFO策略，它会检查最早进入缓存的数据块，但如果该数据块最近被访问过，则会推迟替换，给它第二次机会。
4. **最不常用（Least Frequently Used，LFU）**：LFU替换策略选择**最不常被使用的缓存块进行替换**。它通过维护每个数据块的使用计数来决定哪个数据块最不常被使用。然而，LFU也可能受到计数不准确的影响。
5. **最不常用最近使用（Least Recently Used Recently Used，LRU-K）**：LRU-K替换策略结合了LRU和LFU的思想。它维护每个数据块的使用历史，但只考虑最近的K次访问。这可以在一定程度上减小LRU的开销，同时保持一定的替换效果。
6. **随机替换（Random Replacement）**：随机替换策略是一种基于随机选择要替换的数据块的方法。它不考虑数据的使用模式，因此可能不太有效。

（LRU:未被使用，或者是最后一次使用时间最早；

LFU:一段时间内使用次数最少

二者的时间段也不一定一致）

LRU是最近最少使用页面置换算法(Least Recently Used),也就是首先淘汰最长时间未被使用的页面!

LFU是最近最不常用页面置换算法(Least Frequently Used),也就是淘汰一定时期内被访问次数最少的页!

比如,第二种方法的时期T为10分钟,如果每分钟进行一次调页,主存块为3,若所需页面走向为2 1 2 1 2 3 4

注意,当调页面4时会发生缺页中断

若按LRU算法,应换页面1(1页面最久未被使用) 但按LFU算法应换页面3(十分钟内,页面3只使用了一次)

可见LRU关键是看页面最后一次被使用到发生调度的时间长短,

而LFU关键是看一定时间段内页面被使用的频率!

11. 线程和进程的联系与区别，为什么要引入线程

	进程	线程
定义	资源（CPU时间、内存等）分配的最小单位	程序执行的最小单位
区别	1.独立的地址空间，每创建一个进程，就会给改进程分配一个4G的虚拟内存空间; 2.进程是建立在虚拟内存的基础之上的。	1.没有单独的地址空间（同一进程内的线程共享进程的地址空间）； 2.主要是为了将进程的资源申请和调度属性分开。
联系	1.一个进程有几个线程组成; 2.线程与同属一个进程的其他的线程共享进程所拥有的全部资源。	
OS而言	1.uCOS只有线程的概念，uCOS的整个程序可以理解为一个进程，而其中的任务就可以理解为一个个线程，有自己的堆栈和局部变量，但没有单独的地址空间; 2.对于windows，linux等，其有进程和线程。	
线程的优势	1.和进程相比，它是一种非常"节俭"的多任务操作方式。对进程而言，创建一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间; 2.与进程相比，线程间的通信机制更方便。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便; 3.提高应用程序响应。使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程; 4.使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上; 5.改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。	

为什么引入线程：（可以参考上免线程的优势）

引入线程目的是**减少并发执行时的时空开销。**因为进程的创建、撤销、切换较费时空，它既是调度单位，又是资源拥有者。线程的引入进一步提高了**程序并发执行的程度**，从而进一步**提高了资源的利用率和系统的吞吐量。**

线程是**系统独立调度和分派的基本单位**，基本上不拥有系统资源，只需要少量的资源(指令指针IP，寄存器，栈)，但可以**共享其所属进程所拥有的全部资源**，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有：

- 自己的调用栈（call stack）
- 自己的**寄存器**环境（register context）
- 自己的**线程本地存储**（thread-local storage）。

12. **内存泄漏的原因**

内存泄漏是指在计算机程序中，分配的内存空间在不再需要时未能正确释放或回收，从而导致程序占用越来越多的内存资源，最终可能导致程序性能下降甚至崩溃。内存泄漏的原因可以有多种，以下是一些常见的原因：

1. **未释放动态分配的内存**：当程序动态分配内存（例如使用`malloc`、`new`等操作符）来存储数据，但没有在不再需要时使用`free`、`delete`等来释放这些内存时，就会发生内存泄漏。
2. **引用计数问题**：引用计数是一种内存管理技术，用于跟踪对象的引用数量。如果引用计数不正确地增加或减少，可能导致对象的内存无法正确释放。
3. **循环引用**：当两个或多个对象相互引用，而且它们之间的引用没有被适当地打破，就会发生循环引用。这会阻止垃圾回收器识别和释放不再使用的对象。
4. **未关闭资源**：在使用文件、网络连接、数据库连接等资源时，如果未正确关闭或释放这些资源，就会导致资源泄漏，最终可能导致内存泄漏。
5. **缓存问题**：在某些情况下，程序可能会缓存数据以提高性能，但如果没有适时地清理或更新缓存，就会导致内存泄漏。
6. **事件监听器未移除**：在事件驱动的编程中，如果注册了事件监听器但未在不再需要时移除它们，可能会导致对象无法被垃圾回收，从而造成内存泄漏。

7. **指针丢失**：当程序中的指针丢失（例如，指针被赋予了错误的值或被覆盖），就无法再访问到分配的内存，导致内存泄漏。

13. **页式存储管理的优劣**

    - **页式管理**

      基本原理是将各进程的虚拟空间划分为若干个长度相等的页。把内存空间按页的大小划分为片或者页面，然后把页式虚拟地址与内存地址建立一一对应的页表，并用相应的硬件地址转换机构来解决离散地址变换问题。页式管理采用请求调页和预调页技术来实现内外存存储器的统一管理。

      优点：**没有外碎片**，**每个内碎片不超过页的大小**。

      缺点：程序全部装入内存，要求有相应的硬件支持，如地址变换机构缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。增加了机器成本和系统开销。

    - **段式管理**

      基本思想是把程序按内容或过程函数关系分成段，每段有自己的名字。一个用户作业或者进程所包含的段对应一个二维线性虚拟空间，也就是一个二维虚拟存储器。段式管理程序以段为单位分配内存，然后通过地址映射机构把段式虚拟地址转换为实际内存物理地址。

      优点：可以分别编写和编译，可以针对不同类型的段采取不同的保护，可以按段为单位来进行共享，包括通过动态链接进行代码共享。

      缺点：会产生碎片。

    - **段页式管理**

      系统必须为每个作业或者进程建立一张段表以管理内存分配与释放、缺段处理等。另外由于一个段又被划分为若干个页，每个段必须建立一张页表以把段中的虚页变换为内存中的实际页面。显然与页式管理时相同，页表也要有相应的实现缺页中断处理和页面保护等功能的表项。

      段页式管理是段式管理和页式管理相结合而成，具有两者的优点。

由于管理软件的增加，复杂性和开销也增加。另外需要的硬件以及占用的内存也有所增加，使得执行速度下降。

	页式存储管理	段式存储管理
目的	实现非连续分配，解决碎片问题	更好地满足用户需要
信息单位	页（物理单位）	段（逻辑单位）
大小	固定（由系统定）	不定（由用户程序定）
内存分配单位	页	段
作业地址空间	一维	二维
优点	有效解决了碎片问题（没有外碎片，每个内碎片不超过页大小）；有效提高内存的利用率；程序不必连续存放。	更好地实现数据共享与保护；段长可动态增长；便于动态链接

14. **分页会产生内碎片还是外碎片**

    内碎片，每个内碎片的大小不超过页的大小

15. **使用时间片轮转算法调度进程，如果开了很多进程，会不会影响用户的响应时间，如何解**

    决

    会影响用户的响应时间，因为时间片轮转法是没有优先级的，根据FCFS原则进行服务，当进程过多时，进程的响应时间就会变长。解决方法可以是：

> **1.先来先服务 first-come first-serverd（FCFS）**
>
> 按照请求的顺序进行调度。非抢占式，开销小，无饥饿问题，响应时间不确定（可能很慢）；
>
> 对短进程不利，对IO密集型进程不利。
>
> **2.最短作业优先 shortest job first（SJF）**
>
> 按估计运行时间最短的顺序进行调度。非抢占式，吞吐量高，开销可能较大，可能导致饥饿问题；
>
> 对短进程提供好的响应时间，对长进程不利
>
> **3.优先级调度算法**
>
> 为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。
>
> 4.**时间片轮转**
>
> 将所有就绪进程按 FCFS 的原则排成一个队列，用完时间片的进程排到队列最后。抢占式（时间片用完时），开销小，无饥饿问题，为短进程提供好的响应时间；

> 若时间片小，进程切换频繁，吞吐量低；若时间片太长，实时性得不到保证。
>
> 5.**最高响应比优先**
>
> 响应比 = 1+ 等待时间/处理时间。同时考虑了等待时间的长短和估计需要的执行时间长短，很好的平衡了长短进程。非抢占，吞吐量高，开销可能较大，提供好的响应时间，无饥饿问题。
>
> **6.多级反馈队列调度算法**
>
> 设置多个就绪队列1、2、3...，优先级递减，时间片递增。只有等到优先级更高的队列为空时才会调度当前队列中的进程。如果进程用完了当前队列的时间片还未执行完，则会被移到下一队列。
>
> 抢占式（时间片用完时），开销可能较大，对IO型进程有利，可能会出现饥饿问题。

---

1. 改用最高响应比优先调度算法，同时考虑等待时间的长短以及需要的执行时间的长短，满足用户的响应时间要求。
2. 改用多级反馈队列调度算法，引入优先级，可以将用户进程的优先级调高，从而提高实时性以及响应比。

16. **如何在自己的OS 和 CPU 上进行高级语言编程**

1. 学习了解CPU架构以及支持的汇编语言，选择一款汇编语言，例如MIPS；

   （如果需要，则CPU也可以自己开发，比如我们计组开发的MIPS架构的五级流水线CPU）

2. 编写操作系统内核，实现启动，内存管理，进程管理，文件系统等；

3. 编写高级语言编译器，使得可以将选定的高级语言代码编译成目标CPU架构的汇编代码；
4. 使用高级语言代码编写源程序，例如C,C++等；
5. 编写链接器和加载器，将汇编代码或可执行文件加载到内存中运行；
6. 使用调试工具，例如GDB来识别和解决问题。

1. 学习计算机体系结构和汇编语言：

- 学习计算机体系结构的基本原理，包括CPU、内存、寄存器、指令集等。
- 掌握汇编语言编程，了解如何编写和调试汇编程序，以及如何与硬件进行交互。

2. 开发自己的操作系统：

- 学习操作系统的基本概念，包括进程管理、内存管理、文件系统等。
- 开发或选择一个适合您的平台的简单操作系统内核。常见的选择包括Linux、FreeRTOS、BareMetal等。
- 编写自己的操作系统内核代码，包括启动引导程序、中断处理、任务调度等。

3. 编写高级语言编译器：

- 了解高级语言编译器的工作原理，包括词法分析、语法分析、语义分析和代码生成等阶段。
- 编写高级语言编译器，使得可以将高级语言代码编译成目标CPU架构的汇编代码。

4. 编写高级语言代码：

- 选择一种高级编程语言，例如C、C++、Rust等，编写应用程序代码。
- 使用编译器将高级语言代码编译成目标CPU架构的汇编代码或二进制可执行文件。

5. 链接和加载：

- 编写自己的链接器和加载器，将**汇编代码或可执行文件加载到内存中**以运行。

6. 测试和调试：

- 进行严格的测试和调试，包括单元测试、集成测试和系统测试。
- 使用调试工具，如GDB（GNU调试器）等，来识别和解决问题。

7. 硬件支持：

- 确保您的CPU和硬件支持必要的功能，如内存管理单元（MMU）、中断控制器等。
- 开发或选择适当的设备驱动程序，以支持外部硬件设备。