### 一、操作系统概述

操作系统是掌控计算机上所有事情的软件系统。

#### 基本特征

1、并发：并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。并行需要硬件支持，如多流水线或者多处理器。操作系统通过引入进程和线程，使得程序能够并发运行。

2. 共享：共享是指系统中的资源可以被多个并发进程共同使用。有两种共享方式：互斥共享和同时共享。互斥共享的资源称为临界资源，例如打印机等，在同一时间只允许一个进程访问，需要用同步机制来实现对临界资源的访问。

3. 虚拟：虚拟技术把一个物理实体转换为多个逻辑实体。主要有两种虚拟技术：时分复用技术和空分复用技术。多个进程能在同一个处理器上并发执行使用了时分复用技术，让每个进程轮流占有处理器，每次只执行一小个时间片并快速切换。虚拟内存使用了空分复用技术，它将物理内存抽象为地址空间，每个进程都有各自的地址空间。地址空间和物理内存使用页进行交换，地址空间的页并不需要全部在物理内存中，当使用到一个没有在物理内存的页时，执行页面置换算法，将该页置换到内存中。

4. 异步：异步指进程不是一次性执行完毕，而是走走停停，以不可知的速度向前推进。只要环境和自己不变，结果和时间都不变。

#### 基本功能

1. 进程管理

进程控制、进程同步、进程通信、死锁处理、处理机调度等。

2. 内存管理

内存分配、地址映射、内存保护与共享、虚拟内存等。

3. 文件管理

文件存储空间的管理、目录管理、文件读写管理和保护等。

4. 设备管理

完成用户的 I/O 请求，方便用户使用各种设备，并提高设备的利用率。

主要包括缓冲管理、设备分配、设备处理、虛拟设备等。

#### 大内核和微内核

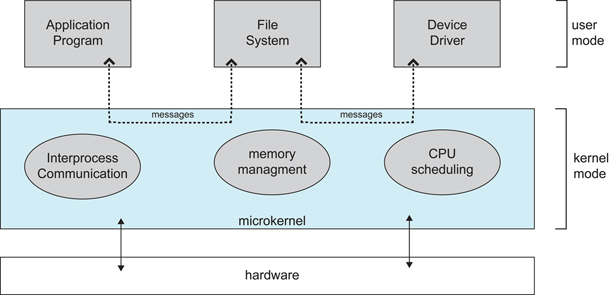
1. 大内核

大内核是将操作系统功能作为一个紧密结合的整体放到内核。由于各模块共享信息，因此有很高的性能。

2. 微内核

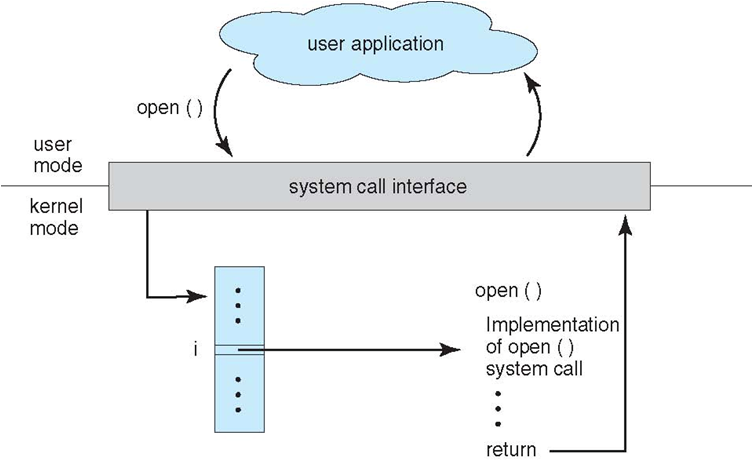
由于操作系统不断复杂，因此将一部分操作系统功能移出内核，从而降低内核的复杂性。移出的部分根据分层的原则划分成若干服务，相互独立。

在微内核结构下，操作系统被划分成小的、定义良好的模块，只有微内核这一个模块运行在内核态，其余模块运行在用户态。因为需要频繁地在用户态和核心态之间进行切换，所以会有一定的性能损失。

[](https://github.com/CyC2018/CS-Notes/blob/master/pics/2_14_microkernelArchitecture.jpg)

#### 系统调用

如果一个进程在用户态需要使用内核态的功能，就进行系统调用从而陷入内核，由操作系统代为完成。

[](https://github.com/CyC2018/CS-Notes/blob/master/pics/tGPV0.png)

### 二、进程管理

#### 在进程两种关系

**1.同步**：多个进程按一定顺序执行；

互斥：多个进程在同一时刻只有一个进程能进入临界区。

**2.信号量机制进程同步**的一个工具，使用它能够合理地分配CPU资源，管理进程

信号量（Semaphore）是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

down : 如果信号量大于 0 ，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；

up ：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

0 表示临界区已经加锁，1 表示临界区解锁。

typedef int semaphore;

semaphore mutex = 1;

void P1() {

down(&mutex);

// 临界区

up(&mutex);

}

void P2() {

down(&mutex);申请使用了资源，减少

// 临界区

up(&mutex);释放资源，增加

}

**3.使用信号量实现生产者-消费者问题**

问题描述：使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入物品；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走物品。

**死锁：**不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行 down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

#define N 100

typedef int semaphore; semaphore mutex = 1;

semaphore empty = N;

semaphore full = 0;

void producer() {

while(TRUE) {

int item = produce\_item();

down(&empty);

down(&mutex);

insert\_item(item);

up(&mutex);

up(&full);

}

}

void consumer() {

while(TRUE) {

down(&full);

down(&mutex);

int item = remove\_item();

consume\_item(item);

up(&mutex);

up(&empty);

}

}

#### 进程调度算法

不同环境的调度算法目标不同。处理机的调度实际上就是用不同的算法来将我们的作业合理分配，提高CPU的利用率。达到公平性、平衡性。

**1. 批处理系统**

批处理系统没有太多的用户操作，在该系统中，调度算法目标是保证吞吐量和周转时间（从提交到终止的时间）。

1.1 先来先服务 first-come first-serverd（FCFS）按照请求的顺序进行调度。

有利于长作业，但不利于短作业，按照作业提交或进程变为就绪状态的先后次序,分派CPU；

当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU(非抢占方式)。

1.2 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。

长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

提高系统的吞吐量。

1.3 最短剩余时间优先 shortest remaining time next（SRTN）

按估计剩余时间最短的顺序进行调度。多用于剥夺式的调度中。

也就是短作业优先算法的升级版

**2. 交互式系统**

交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。

2.1 时间片轮转

将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

如果时间片太小，进程切换得太频繁，在进程切换上就会花过多时间。

如果时间片过长，那么实时性就不能得到保证。

2.2 优先级调度

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。

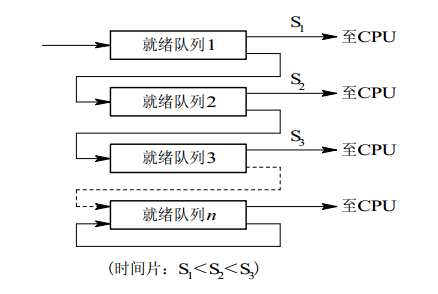
2.3 多级反馈队列（不太懂）

设置多个就绪队列，分别赋予不同的优先级，如逐级降低，队列1的优先级最高。

每个队列时间片大小都不同，规定优先级越低则时间片越长，如逐级加倍，例如 1,2,4,8,..。进程在第一个队列没执行完，就会被移到下一个队列。这种方式下，之前的进程只需要交换 7 次。

每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。



**3. 实时系统**

实时系统要求一个请求在一个确定时间内得到响应

分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

### 三、死锁

#### 死锁的检测

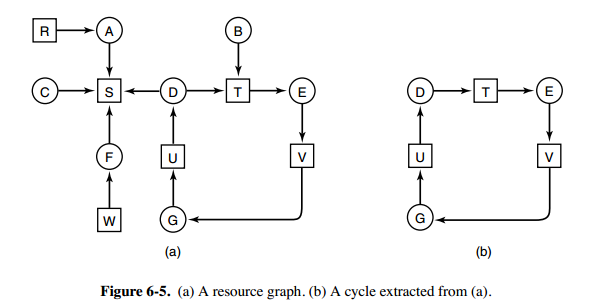
死锁检测算法是当进程进行资源请求时检查并发进程组是否构成资源的请求和占用环路。如果不存在这一环路，则系统中一定没有死锁。

因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任务措施的方案会获得更高的性能。

当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

**1. 每种类型一个资源的死锁检测**

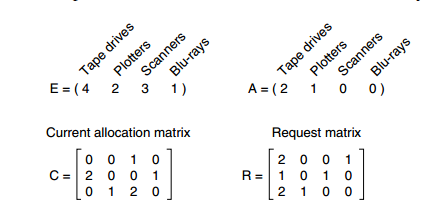
[](https://github.com/CyC2018/CS-Notes/blob/master/pics/b1fa0453-a4b0-4eae-a352-48acca8fff74.png)

上图为资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程，进程指向资源表示进程请求获取该资源。

满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是通过检测有向图是否存在环来实现也就是检测到死锁的发生。

**2. 每种类型多个资源的死锁检测**

[](https://github.com/CyC2018/CS-Notes/blob/master/pics/e1eda3d5-5ec8-4708-8e25-1a04c5e11f48.png)

上图中，有三个进程四个资源，每个数据代表的含义如下：

E 向量：资源总量

A 向量：资源剩余量

C 矩阵：每个进程所拥有的资源数量，每一行都代表一个进程拥有资源的数量

R 矩阵：每个进程请求的资源数量

进程 P1 和 P2 所请求的资源都得不到满足，只有进程 P3 可以，让 P3 执行，之后释放 P3 拥有的资源，此时 A = (2 2 2 0)。P2可以执行，执行后释放 P2 拥有的资源，A = (4 2 2 1) 。P1 也可以执行。所有进程都可以顺利执行，没有死锁。

算法总结如下：

系统是安全的，是指系统中的所有进程能够按照某一种次序分配资源，并且依次地运行完毕，这种进程序列{ P1 ，P2 …Pn}就是安全序列。

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死锁进程。

1. 寻找一个没有标记的进程 Pi，它所请求的资源小于等于 A。
2. 如果找到了这样一个进程，那么将 C 矩阵的第 i 行向量加到 A 中，标记该进程，并转回 1。
3. 如果没有这样一个进程，算法终止。

#### 死锁避免

对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

**1. 安全状态**

**2.** **多个资源的银行家算法**

检查一个状态是否安全的算法如下：

**设Requesti是进程Pi的请求向量，如果Request i[j]=K，表示进程Pi需要K个Rj类型的资源。当Pi发出资源请求后，系统按下述步骤进行检查：**

**(1) 如果Request i[j]≤Need[i, j]，便转向步骤(2)； 否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。**

**2) 如果Request i[j]≤Available[j]，便转向步骤(3)； 否则，表示尚无足够资源，Pi须等待。**

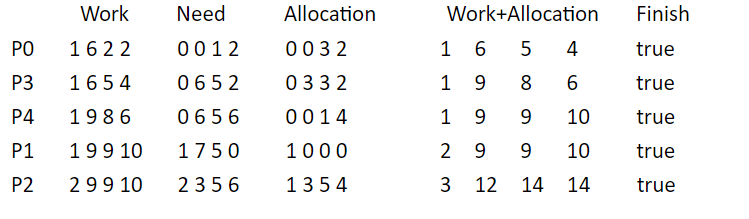
**(3) 系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值：**

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

**Allocation[i, j] = Allocation[i, j] + Request i[j];**

**Need[i, j] = Need[i, j] - Request i[j];**

**(4) 系统执行安全性算法，检查此次资源分配后系统是否处于安全状态。若安全，才正式将资源分配给进程Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待**



①Request2(1,2,2,2)<=Need2(2,3,5,6)

②Request2(1,2,2,2)<=Available(1,6,2,2)

③系统先假定可为P2分配资源，并修改Available，Allocation2和Need2向量：

Available=(0,4,0,0)

Allocation2=(2,5,7,6)

Need2=(1,1,3,4)

此时再进行安全性检查，发现 Available=(0,4,0,0) 不能满足任何一个进程，所以判定系统进入不安全状态，即不能分配给P2相应的Request(1,2,2,2)。

### 四、设备管理

#### 设备管理概述

计算机系统的一个重要组成部分是I/O系统，在该系统中包括用于实现信息输入、输出和存储功能的设备和相应的设备控制器，在有些大型机中，还有I/O通道或I/O处理机。

计算机的I/O系统是主机和外设之间的数据传送系统

与计算机相连接的外部设备：

（1）字符设备（character device），又叫做人机交互设备。用户通过这些设备实现与计算机系统的通信。它们大多是以字符为单位发送和接受数据的，数据通信的速度比较慢。例如，键盘和显示器为一体的字符终端、打印机、扫描仪、包括鼠标等，还有早期的卡片和纸带输入和输出机。含有显卡的图形显示器的速度相对较快，可以用来进行图像处理中的复杂图形的显示。

（2）块设备（block device），又叫外部存储器，用户通过这些设备实现程序和数据的长期保存。与字符设备相比，它们是以块为单位进行传输的，如磁盘、磁带和光盘等。块的常见尺寸为512~32768B之间。

设备管理的对象是I/O设备，设备控制器和I/O通道。

设备管理的基本任务是完成用户提出的I/O请求，提高I/O速度，改善I/O设备的利用率。

设备管理的功能包括缓冲区管理、设备分配、设备处理、虚拟设备以及实现设备独立性等

#### 设备控制器

设备控制器是计算机中的一个实体，其主要职责是控制一个或多个I/O设备，以实现I/O设备和计算机之间的数据交换。它是CPU与I/O设备之间的接口，它接收从CPU发来的命令，并去控制I/O设备工作，以使处理机从繁杂的设备控制事务中解脱出来。

设备控制器是一个可编址的设备，若控制可连接多个设备时，则应含有多个设备地址，并使每一个设备地址对应一个设备。

设备控制器的复杂性因不同设备而异，相差甚大，于是可把设备控制器分成两类：

1. 一类是用于控制字符设备的控制器，
2. 另一类是用于控制块设备的控制器。

设备控制器的基本功能：

(1)接收和识别命令

(2)数据交换

(3)标识和报告设备的状态

(4)地址识别

(5)数据缓冲

(6)差错控制

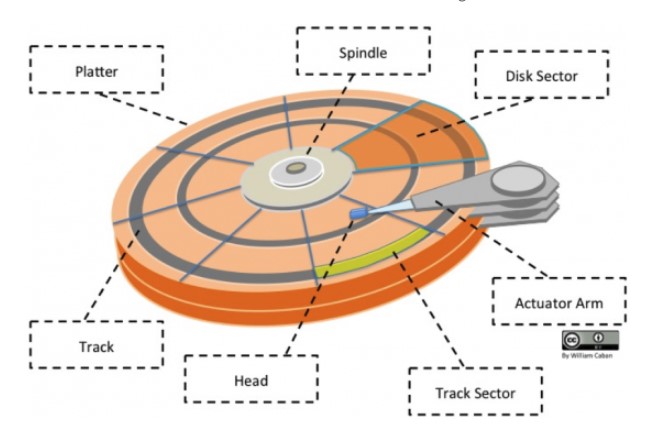
设备控制器的组成：

(1)设备控制器与处理机的接口

(2)设备控制器与设备的接口

(3)I/O逻辑

#### 磁盘结构

[](https://github.com/CyC2018/CS-Notes/blob/master/pics/014fbc4d-d873-4a12-b160-867ddaed9807.jpg)

#### 磁盘调度算法

读写一个磁盘块的时间的影响因素有：

旋转时间（主轴转动盘面，使得磁头移动到适当的扇区上）

寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）

实际的数据传输时间

其中，寻道时间最长，因此磁盘调度的主要目标是使磁盘的平均寻道时间最短。

1. 先来先服务

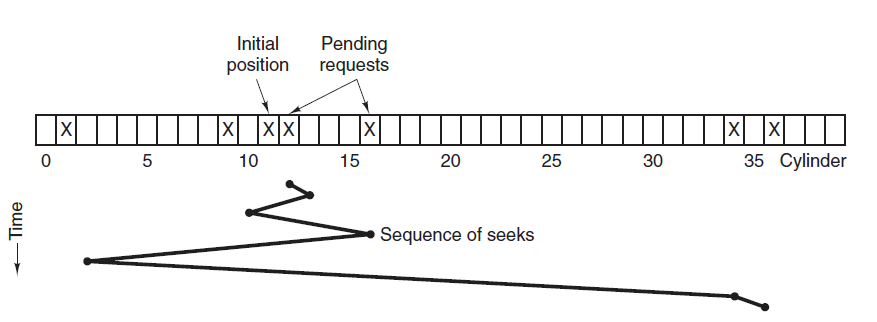
按照磁盘请求的顺序进行调度。

优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。

2. 最短寻道时间优先

**优先调度与当前磁头所在磁道距离最近的磁道。**

虽然平均寻道时间比较低，但是不够公平。如果新到达的磁道请求总是比一个在等待的磁道请求近，那么在等待的磁道请求会一直等待下去，也就是出现饥饿现象。具体来说，两端的磁道请求更容易出现饥饿现象。

[](https://github.com/CyC2018/CS-Notes/blob/master/pics/4e2485e4-34bd-4967-9f02-0c093b797aaa.png)

队列：(1) 2 4 0 5 1

寻道总长：(2-1)+(4-2)+(4-0)+(5-0)+(5-1)=16 16/5=3.2

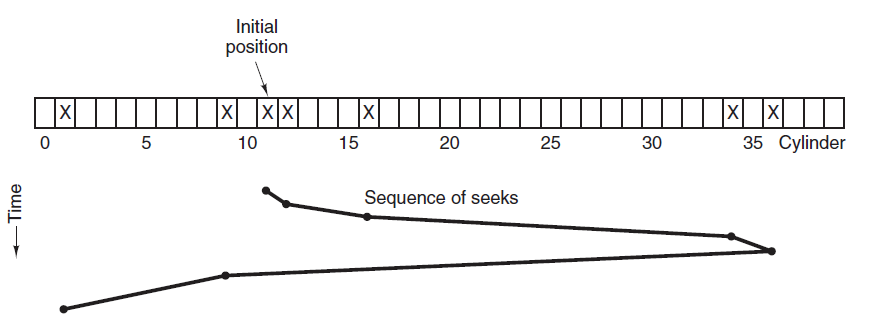
3. 电梯算法

SCAN

电梯总是**保持一个方向运行，直到该方向没有请求为止，然后改变运行方向。**

电梯算法（扫描算法）和电梯的运行过程类似，总是按一个方向来进行磁盘调度，直到该方向上没有未完成的磁盘请求，然后改变方向。

因为考虑了移动方向，因此所有的磁盘请求都会被满足，解决了 SSTF 的饥饿问题。

[](https://github.com/CyC2018/CS-Notes/blob/master/pics/271ce08f-c124-475f-b490-be44fedc6d2e.png)

#### 设备独立性

**绝对号(或物理设备名):**

为了便于对这些外设进行管理，系统对每台进入计算机系统中的设备都给定一个对应的编号，作为调用时识别和区分设备用。这种编号无任何重复，一般被称为设备的绝对号(或物理设备名)。

**相对号(或称逻辑设备名):**

为了方便用户，也为了提高外设利用率，在计算机中规定用户申请外设时，只需要向系统说明所需用的某类设备，至于真正在实际中使用哪台设备，由系统根据这类设备的应用情况作出分配。即使用户所需多台同样的设备，系统也允许用户按自己的使用要求提出编号，这种由用户申请设备时所用的编号称为相对号(或称逻辑设备名)、

有了设备的绝对号和相对号后，用户编制程序使用的设备与实际使用的设备无关，这就是设备的独立性。

)设备独立性优点1) 设备分配时的灵活性。2)易于实现I/O重定向。

#### 设备控制表

系统中的每台设备都有一张设备控制表(DCT)。在DCT中充分体现出了设备的类型、标识符、状态(忙/闲)、重复执行次数、设备号以及与该设备相连的设备控制器的地址和等待使用该设备的进程队列。

系统为每个控制器配置一张用于记录本控制器情况的表，称为控制器控制表(COCT)。

系统为每个通道都配置一张通道控制表，通道控制表(CHCT)。

系统设备表(System Device Table, SDT)表中，每个接入系统中的外围设备都占有一个表目项，记录了该设备的名称、标识及设备控制表DTC的入口地址等相关的信息。SDT表在整个系统中只有一张，全面反映了系统中的外设资源的类型、数量、占用情况等，

#### 直接存储器访问DMA控制方式

**DMA控制器(DMAC**)是一种在系统内部转移数据的独特外设

可以将其视为一种能够通过一组专用总线将内存和外部存储器与每个具有DMA能力的外设连接起来的控制器

它之所以属于外设，是因为它是在处理器的编程控制下来执行传输的。

值得注意的是，通常只有数据流量较大(KBps或者更高)的外设才需要支持DMA能力，这些应用方面典型的例子包括视频、音频和网络接口

DMA方式，在一定程度上提升了IO速度，把CPU从低效地IO操作中解脱出来，提高了CPU的利用率，从而提高了整个系统的效率。

特点：

1)数据传送的单位：数据块

2)数据传输的方向：设备(控制器)←→内存

2 DMA控制器的组成 ：DMA控制器由命令/状态寄存器CR、内存地址寄存器MAR、数据寄存器DR和数据计数器DC组成。

原理：

在实现DMA传输时，是由DMA控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即DMA传输前，CPU要把总线控制权交给DMA控制器，而在结束DMA传输后，DMA控制器应立即把总线控制权再交回给CPU。一个完整的DMA传输过程必须经过DMA请求、DMA响应、DMA传输、DMA结束4个步骤。

DMA技术的出现，使得外围设备可以通过DMA控制器直接访问内存，与此同时，CPU可以继续执行程序。

那么DMA控制器与CPU怎样分时使用内存呢?通常采用以下三种方法：

(1)停止CPU访内存外围设备要求传送一批数据时，由DMA控制器发一个停止信号给CPU，要求CPU放弃对地址总线、数据总线和有关控制总线的使用权。

(2)周期挪用：当I/O设备没有DMA请求时，CPU按程序要求访问内存；一旦I/O设备有DMA请求，则由I/O设备挪用一个或几个内存周期。

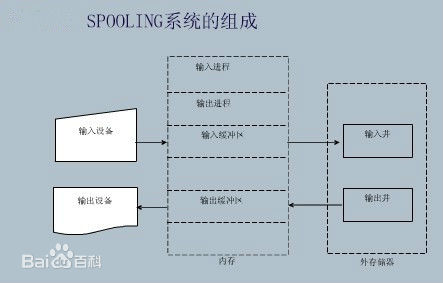
(3)DMA与CPU交替访问内存：如果CPU的工作周期比内存存取周期长很多，此时采用交替访内的方法可以使DMA传送和CPU同时发挥最高的效率。

#### SPOOLing技术

输入井和输出井。这是在磁盘上开辟的两个大存储空间。输入井是模拟脱机输入时的磁盘设备，用于暂存I/Q设备输入的数据；输出井是模拟脱机输出时的磁盘，用于暂存用户程序的输出数据。

内存中形成缓冲区，在高级设备形成输出井和输入井，传递的时候，从低速设备传入缓冲区，再传到高速设备的输入井，再从高速设备的输出井，传到缓冲区，再传到低速设备。

将一台独享打印机改造为可供多个用户共享的打印机，是应用SPOOLing技术的典型实例。具体做法是：系统对于用户的打印输出，但并不真正把打印机分配给该用户进程，而是先在输出井中申请一个空闲盘块区，并将要打印的数据送入其中；然后为用户申请并填写请求打印表，将该表挂到请求打印队列上。若打印机空闲，输出程序从请求打印队首取表，将要打印的数据从输出井传送到内存缓冲区，再进行打印，直到打印队列为空。



若系统的某台台行式打印机采用了虚拟设备技术，那么若有进程要求对它打印输出时,SPOOLing系统并不是将这台打印机直接分配给进程，而是在共享设备（磁盘）上的输出,SPOOLing存储区中为其分配一块存储空间，进程的输出数据以文件形式表示的。各进程的数据输出文件形成了一个输出队列，由输出POOLing系统控制这台打印机进程，依次将队列中的输出文件实际打印输出。在SPOOLing系统中，实际上并没有为任何进程分配，而只是在输入井和输出井中，为进程分配一存储区和建立一章I/O请求表。这样，便把独占设备改造为共享设备。

将一台物理的IO设备虚拟为多台的逻辑IO设置，这样多个用户就可以共享一台物理IO设置了。

虚拟设备技术假脱机操作(Simultaneous Peripheral Operations On-Line，SPOOLing)，利用专门的外围控制机，将低速I/O设备上的数据传送到高速磁盘上；或者相反。

利用假脱机技术，可把独享设备转变成具有共享特征的虚拟设备，从而提高设备利用率。

由于我们的多道程序技术引入后，我们的利用专门的外围控制机就可以使用程序来进行模拟了。

1)高速虚拟I/O操作：

应用程序的虚拟I/O比实际I/O速度提高，缩短应用程序的执行时间(尽快完成计算，并释放占用的计算机资源)。

另一方面，程序的虚拟I/O操作时间和实际I/O操作时间分离开来。

2)实现对独享设备的共享：

由SPOOLing程序提供虚拟设备，可以对独享设备依次共享使用。

### 五、存储器管理

#### 存储器的基础知识

首先，一般的存储器我们就会认为它包含着三部分：

寄存器：速度最快，但是造价高

主存储器：速度次之，被通俗称为内存

外存：速度最慢，用于存储文件数据，因为上边两种一旦断电，数据就会丢失。这个用来做持久化存储的。

因此，我们的存储器往往是使用三层结构的。

#### 程序的装入和链接

在操作系统的角度而言，我们面对存储器就是面对程序的装入和连接。用户程序中使用的地址称为相对地址或逻辑地址或虚拟地址。

一般地，用户程序向要在系统上运行，就要经历下面几个步骤：

编译：对用户源程序进行遍历，形成若干个目标模块

链接：将目标模块以及他们所需要的库函数链接在一起，形成完整的模块。

装入：将模块装入内存

1.绝对装入方式(Absolute Loading Mode)：

在早期，当程序装入内存时，指令存储在内存中的物理地址与其逻辑地址完全相同.

可重定位装入方式（避免地址叠加）

2.采用可重定位的装入方式(Relocation loading Mode)时，如果能够将不同的的程序装入到地址范围不同的物理内存空间，避免分配给各个程序的内存空间叠加(相交或“撞车”)，就能实现将多个程序安全装入内存，使它们共享内存空间，从而支持多任务系统

如果使用可重定位装入方式，就必须要解决：逻辑地址对物理地址之间的转换

3.静态装入：在程序运行前一次性装入

静态链接是由链接器在链接时将库的内容加入到可执行程序中的做法。

链接器是一个独立程序，将一个或多个库或目标文件(先前由编译器或汇编器生成)链接到一块生成可执行程序。

4、动态装入：在程序运行时，分批装入到内存中

随着内存技术的发展，为了让更多的程序投入有限的内存并发运行，操作系统只将运行程序所必须的模块装入内存，其他诸如帮助系统等不常用的程序模块不装入内存，进而只装入能够使程序运行起来的那部分模块，更加节省了空间。

5.静态链接

静态链接是由链接器在链接时将库的内容加入到可执行程序中的做法。

链接器是一个独立程序，将一个或多个库或目标文件(先前由编译器或汇编器生成)链接到一块生成可执行程序。

6.动态链接

运行时动态链接则是把当前模块调用的模块推迟到调用的时候再载入。

#### 连续分配存储管理方式

操作系统将内存分为系统区和用户区两部分，比例为1:3。

系统区仅提供给OS使用，通常是放在内存的低址部分，用户区指系统区以外的全部内存空间，提供给用户使用，通常在高地址部分。

内存分配方式指的是对用户区的分配方式。原因是用户提交的任务只能在用户区运行。

**固定分区方式**

为了支持多道程序系统和分时系统，支持多个程序并发执行，引入了分区式存储管理。

分区式存储管理是把内存分为一些大小相等或不等的分区，操作系统占用其中一个分区，其余的分区由应用程序使用，每个应用程序占用一个或几个分区。

系统对内存的管理和控制通过数据结构—分区说明表进行。

分区说明表说明各分区号、分区大小、起始地址和是否是空闲区(分区状态)。

内存的分配释放、存储保护以及地址变换等都通过分区说明表进行。

**动态分区**

如果内存分区的划分不是预先划定，而是根据所要运行的程序的大小分配内存，在内存分配阶段就不会出现内存碎片。动态分区技术应运而生。

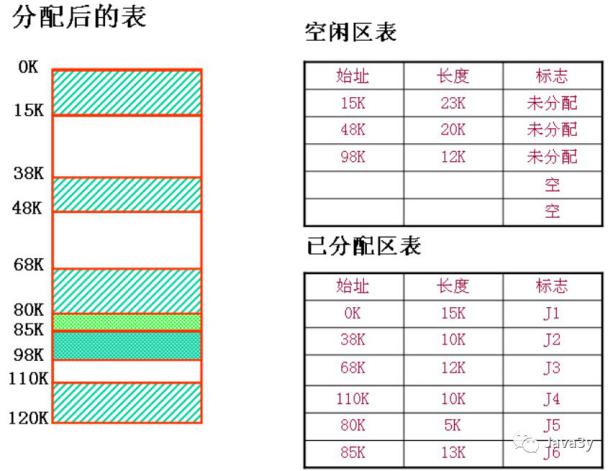
1 基本问题：

1)动态分区的基本思想：在作业执行前不直接建立分区，分区的建立是在作业的处理过程中进行的。且其大小可随作业或进程对内存的要求而改变。

2)动态创建分区：在装入程序时按其初始要求分配，或在其执行过程中通过系统调用进行分配或改变分区大小，按需分配。

3)采用的数据结构：内存分配表，由两个表格组成。一个是已分配区表，另一张是空闲区表.

动态分区就有两张表来进行说明了。



动态分区分配内存时从可用表或自由链中寻找空闲区的常用方法

1)首次适应算法(First Ft Algorithm, FFA)首次适应法要求可用表或自由链按起始地址递增的次序排列

2)最佳适应算法(Best Fit Algorithm, BFA)要求按空闲区大小从小到大的次序组成空闲区可用表或自由链。

3)最坏适应算法(Worst Fit Algorithm, WFA)要求空闲区按其大小递减的顺序组成空闲区可用表或自由链。

**碎片问题**

4)碎片问题

经过一段时间的分配回收后，内存中存在很多很小的空闲块。它们每一个都很小，不足以满足分配要求； 但其总和满足分配要求。这些空闲块被称为碎片。

5)碎片问题的解决

通过在内存移动程序的技术将所有小的空闲区域合并为大的空闲区域，这种技术称为紧凑技术，也称为紧缩技术、紧致技术、浮动技术、搬家技术。

#### 可重定位分区分配

将碎片合成一大块---可重定位的动态分区

动态分区很完美地在内存初次分配阶段解决了内存空间浪费的问题。但是，内存是需要重复利用的。随着任务的不断运行完毕，内存空间会被回收；同时，操作系统又会不断接收新的任务，内存空间会被分配。

于是，当运行一个新任务时，只能从回收回来的分区上分配内存，一方面动态分区技术在一定程度上退化为固定分区，另一方面分配后余下的碎片(小块的内存空间)就被保留了下来，较小的内存空间被分配出去的可能性很小，造成了浪费(碎片）。

针对动态分区中碎片之和大于任务所需空间的情况，考虑对内存空间采用紧凑技术进行整理，将**已进入内存的任务所占有的内存空间尽量搬到较低的地址，**相对的，空闲碎片的会被换到了高地址空间。

当所有进入内存的任务都被搬到较低的地址后，空闲碎片都被移动到了内存空间的高地址空间。

于是，所有的碎片被整合成了一个大块，从而可以装载任务。这就是可重定位的动态分区。

将碎片合成一大块---可重定位的动态分区

实现

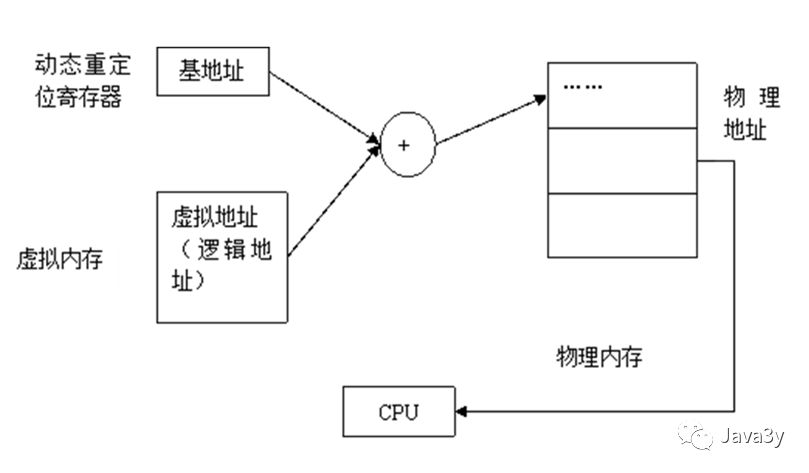
1）动态重定位依靠硬件地址变换机构完成。

2）地址重定位机构需要基地址寄存器(BR)和程序虚地址寄存器(VR)。

指令或数据的虚地址(VA)，也称为逻辑地址(LA)。内存地址MA(MA)，也称物理地址(PA)

**实现逻辑地址到物理地址的转换，可以通过公式MA＝(BR)十(VR)完成。**

图5-10所示为指令或数据的逻辑地址到物理地址的转换过程



优点

1. 可以对内存进行非连续分配。2)动态重定位提供了实现虚拟存储器的基础。3)有利于程序段的共享。

4 动态重定位分区的分配算法

1)主干是动态分区的分配算法。2)在动态分区的基础上增加了紧凑技术。3)内存分配算法

#### 基本分页存储管理

上面的存储都是连续的内存分配技术，我们的分页存储是离散的。与其花费巨大的代价搬家，不如离散地存储在这些碎片中

* 一种离散存储的方法是面向系统的，将内存用户空间划分为大小相等(2nB，如4KB)的物理块；
* 另一种方法是面向用户的，将内存空间划分为物理段，32位系统中，以高16位表示段号，低16位表示段内地址。

这就是汇编语言中定义一个段时大小不可以超过64KB的根本原因。因为段内寻址的时候，只有2个字节（16个二进制位）用来表示地址偏移，2^16=64KB。

离散存储思想产生的原因：

(1)紧凑技术的弊端

尽管可重定位分区方式下的紧凑技术使得碎片得以利用，提高了内存空间的利用率。但这是以牺牲时间进行搬家而获得的，代价是很高的。

(2)求变创新

从内存连续分配的思想上突破、创新，提出离散的内存分配方式，从而使存在大量碎片和无足够大的连续空间供分配这一矛盾得以缓解。

离散存储的基本概念

将一个进程直接分散地装入到许多不相邻的分区中，而无须“紧凑”的分配方式，称为离散分配。

离散分配的种类包括分页存储管理、分段存储管理和段页式存储管理。

不具备页面对换功能的分页存储管理方式称为基本分页存储管理方式。

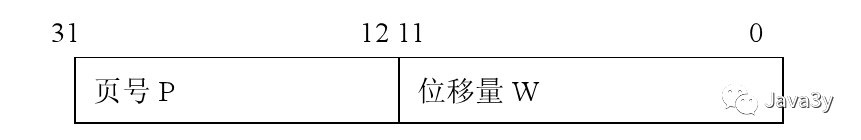
#### 分页存储管理

分页存储管理的基本方法：页面和物理块 页面 页面大小。

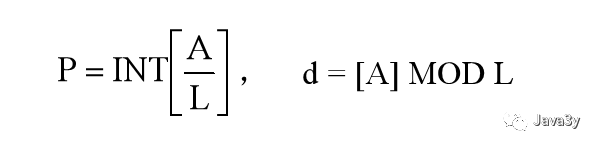
将用户作业的地址空间分成若干个大小相同的区域，称为页面或页，并为每个页从“0”开始编号；

相应地，主存空间也分成与页大小相同的若干个存储块，或称为物理块或页框(frame)，并且采用同样的方式为它们进行编号，从0开始：0块，1块，…，n-1块

分页地址中的地址结构如下：

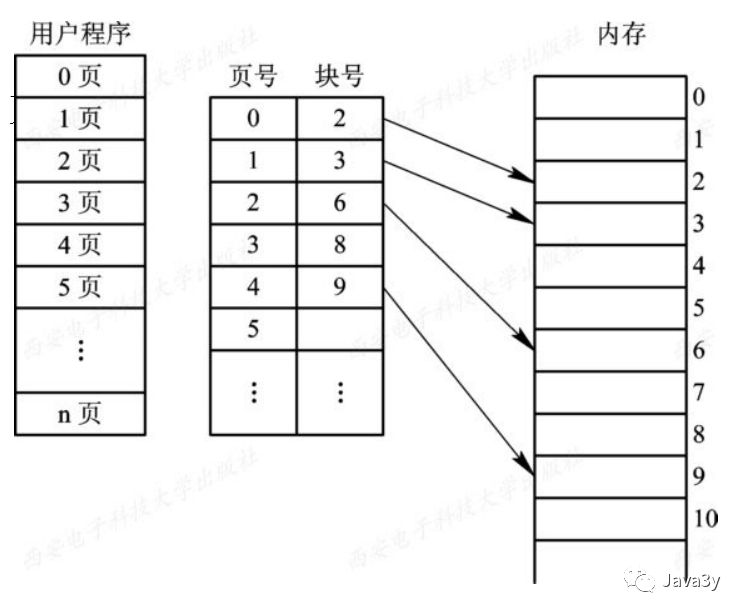


对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：



**页表**

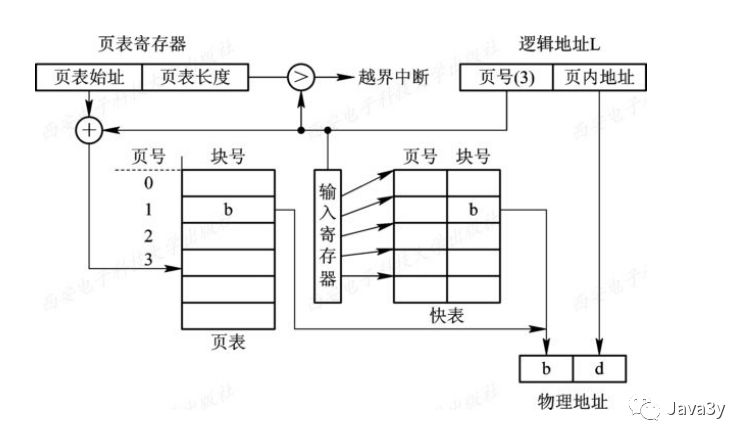
在分页系统中，允许将进程的各个页离散地存储在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称页表



进程在运行期间，需要对程序和数据的地址进行变换，即将用户地址空间中的逻辑地址变换为内存空间中的物理地址，由于它执行的频率非常高，每条指令的地址都需要进行变换，因此需要采用硬件来实现。页表功能是由一组专门的寄存器来实现的。一个页表项用一个寄存器。

由于页表是存放在内存中的，这使CPU在每存取一个数据时，都要两次访问内存。第一次是访问内存中的页表，从中找到指定页的物理块号，再将块号与页内偏移量W拼接，以形成物理地址。第二次访问内存时，才是从第一次所得地址中获得所需数据(或向此地址中写入数据)。因此，采用这种方式将使计算机的处理速度降低近1/2。可见，以此高昂代价来换取存储器空间利用率的提高，是得不偿失的。

因此，我们采用：具有快表的地址变换机



#### 分段存储管理

用户把自己的作业按照逻辑关系划分为若干个段，每个段都从0开始编址，并有自己的名字和长度。因此，程序员们都迫切地需要访问的逻辑地址是由段名(段号)和段内偏移量(段内地址)决定的，这不仅可以方便程序员编程，也可使程序非常直观，更具可读性。

在实现对程序和数据的共享时，是以信息的逻辑单位为基础的。

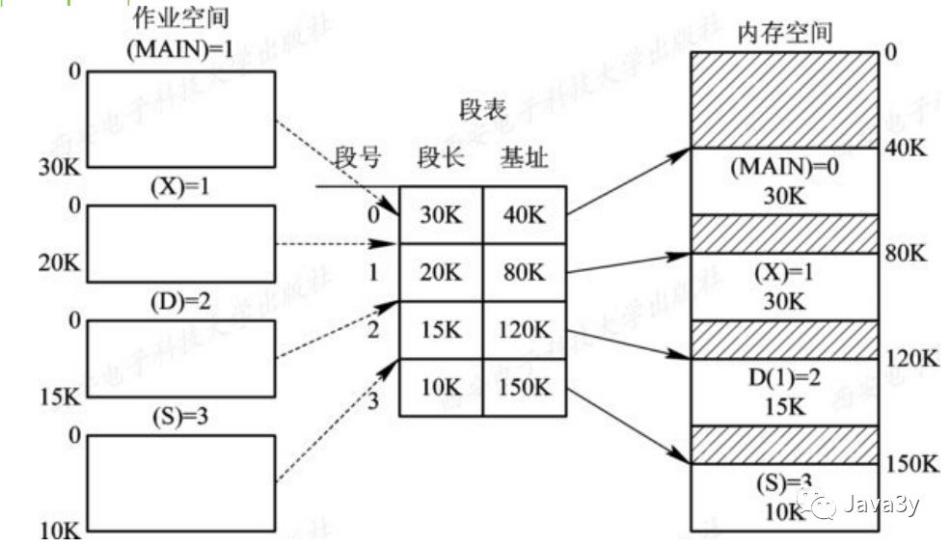
分页系统中的“页”只是存放信息的物理单位(块)，并无完整的逻辑意义，这样，一个可被共享的过程往往可能需要占用数十个页面，这为实现共享增加了困难。

信息保护同样是以信息的逻辑单位为基础的，而且经常是以一个过程、函数或文件为基本单位进行保护的。

在实际应用中，往往存在着一些段，尤其是数据段，在它们的使用过程中，由于数据量的不断增加，而使数据段动态增长，相应地它所需要的存储空间也会动态增加。然而，对于数据段究竟会增长到多大，事先又很难确切地知道。对此，很难采取预先多分配的方法进行解决。

**分段系统的基本原理**

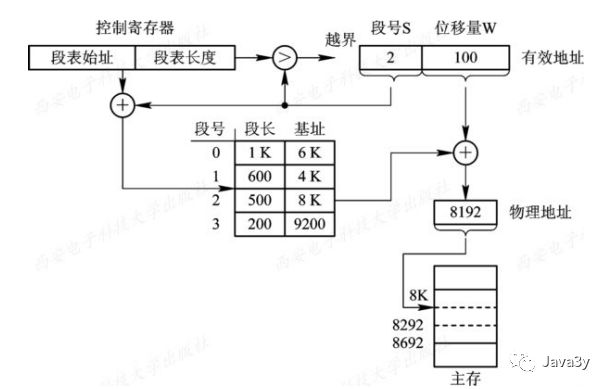
在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段定义了一组逻辑信息。例如，有主程序段MAIN、子程序段X、数据段D及堆栈段S等。



**段表**

在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在分段式存储管理系统中，则是为每个分段分配一个连续的分区。进程中的各个段，可以离散地装入内存中不同的分区中。为保证程序能正常运行，就必须能从物理内存中找出每个逻辑段所对应的位置。这就需要段表了

为了实现进程从逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表始址和段表长度TL。在进行地址变换时，系统将逻辑地址中的段号与段表长度TL进行比较。若S>TL，表示段号太大，是访问越界，于是产生越界中断信号。若未越界，则根据段表的始址和该段的段号，计算出该段对应段表项的位置，从中读出该段在内存的起始地址。然后，再检查段内地址d是否超过该段的段长SL。若超过，即d>SL，同样发出越界中断信号。若未越界，则将该段的基址d与段内地址相加，即可得到要访问的内存物理地址。图示出了分段系统的地址变换过程。



**分页和分段的总结**

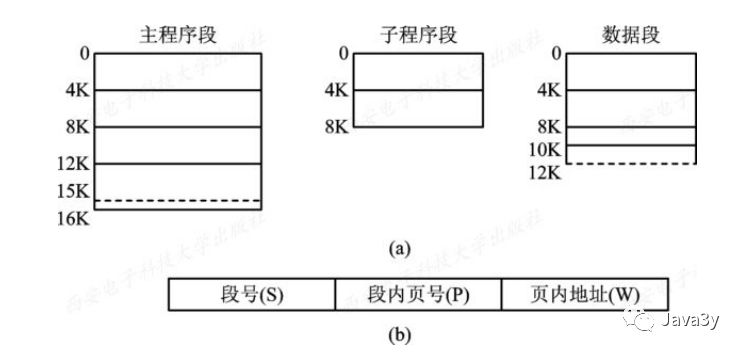
（1）页是信息的物理单位，分页是为了实现离散的分配方式，以消减主存“碎片”，提高主存的利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，它包含一组意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。

（2）页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而一个系统只能有一种大小的页面。段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

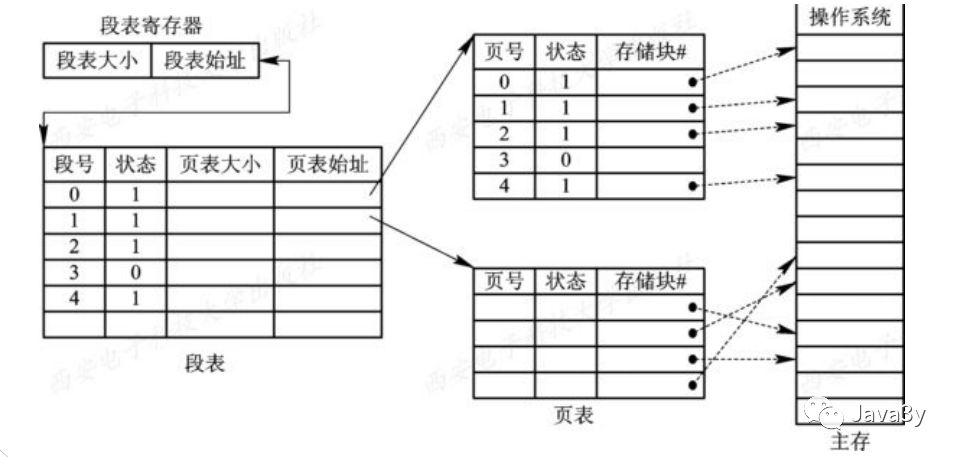
（3）分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需要利用一个记忆符，即可表示一个地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

#### 段页式存储管理

段页式系统的基本原理是分段和分页原理的结合，即先将用户程序分成若干个段，再把每个段分成若干个页，并为每一个段赋予一个段名。图(a)示出了一个作业地址空间的结构。该作业有三个段：主程序段、子程序段和数据段；页面大小为 4 KB。在段页式系统中，其地址结构由段号、段内页号及页内地址三部分所组成，如图(b)所示。

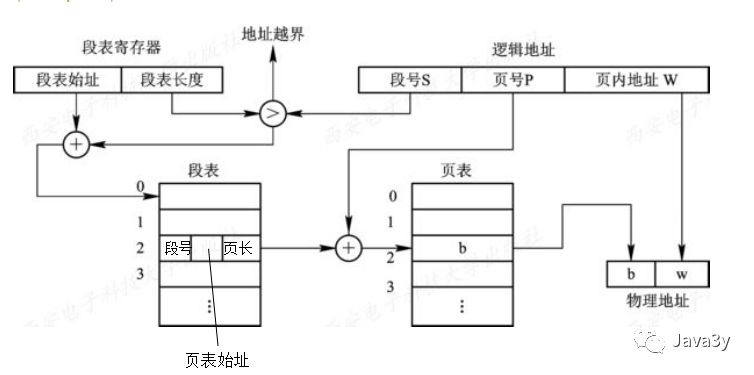


在段页式系统中，为了实现从逻辑地址到物理地址的变换，系统中需要同时配置段表和页表。段表的内容与分段系统略有不同，它不再是内存始址和段长，而是页表始址和页表长度。图示出了利用段表和页表进行从用户地址空间到物理(内存)空间的映射。



**地址变换过程：**

　在段页式系统中，为了便于实现地址变换，须配置一个段表寄存器，其中存放段表始址和段长TL。进行地址变换时，首先利用段号S，将它与段长TL进行比较。若S < TL，表示未越界，于是利用段表始址和段号来求出该段所对应的段表项在段表中的位置，从中得到该段的页表始址，并利用逻辑地址中的段内页号P来获得对应页的页表项位置，从中读出该页所在的物理块号b，再利用块号b和页内地址来构成物理地址。图示出了段页式系统中的地址变换机构。



#### 虚拟存储器

要求将一个作业全部装入内存后方能运行，于是，出现了下面这样两种情况：

　(1) 有的作业很大，其所要求的内存空间超过了内存总容量，作业不能全部被装入内存，致使该作业无法运行；

　(2) 有大量作业要求运行，但由于内存容量不足以容纳所有这些作业，只能将少数作业装入内存让它们先运行，而将其它大量的作业留在外存上等待

局部性原理：

程序在执行时将呈现出局部性规律，即在一较短的时间内，程序的执行仅局限于某个部分，相应地，它所访问的存储空间也局限于某个区域。时间局部性和空间局部性

**虚拟存储器的定义**

　当用户看到自己的程序能在系统中正常运行时，他会认为，该系统所具有的内存容量一定比自己的程序大，或者说，用户所感觉到的内存容量会比实际内存容量大得多。但用户所看到的大容量只是一种错觉，是虚的，故人们把这样的存储器称为虚拟存储器。

**实现方法**

主要的硬件支持有：

(1) 请求分页的页表机制。

(2) 缺页中断机构。

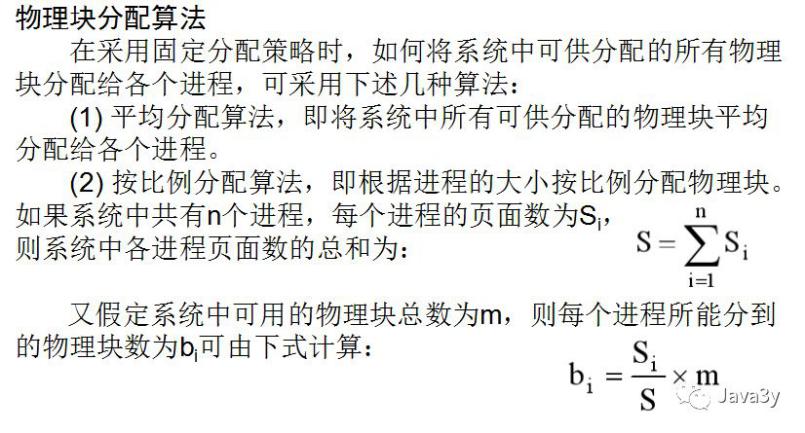
(3) 地址变换机构。

为了实现请求分页，系统必须提供一定的硬件支持。计算机系统除了要求一定容量的内存和外存外，还需要有请求页表机制、缺页中断机构以及地址变换机构。

**在请求分页系统**中需要的主要数据结构是请求页表，其基本作用仍然是将用户地址空间中的逻辑地址映射为内存空间中的物理地址。为了满足页面换进换出的需要，在请求页表中又增加了四个字段。这样，在请求分页系统中的每个页表应含以下诸项

C:\Users\夏\AppData\Local\Temp\WeChat Files\9710a7e9b8130254f96c10bce9fc3aeb.png

一个显而易见的事实是，随着为每个进程所分配的物理块的减少，将使进程在执行中的缺页率上升，从而会降低进程的执行速度。最小物理块数是指能保证进程正常运行所需的最小物理块数，当系统为进程分配的物理块数少于此值时，进程将无法运行。



考虑优先权的分配算法。在实际应用中，为了照顾到重要的、紧迫的作业能尽快地完成，应为它分配较多的内存空间。通常采取的方法是把内存中可供分配的所有物理块分成两部分：一部分按比例地分配给各进程；另一部分则根据各进程的优先权进行分配，为高优先进程适当地增加其相应份额。在有的系统中，如重要的实时控制系统，则可能是完全按优先权为各进程分配其物理块的。

**页面调入策略**

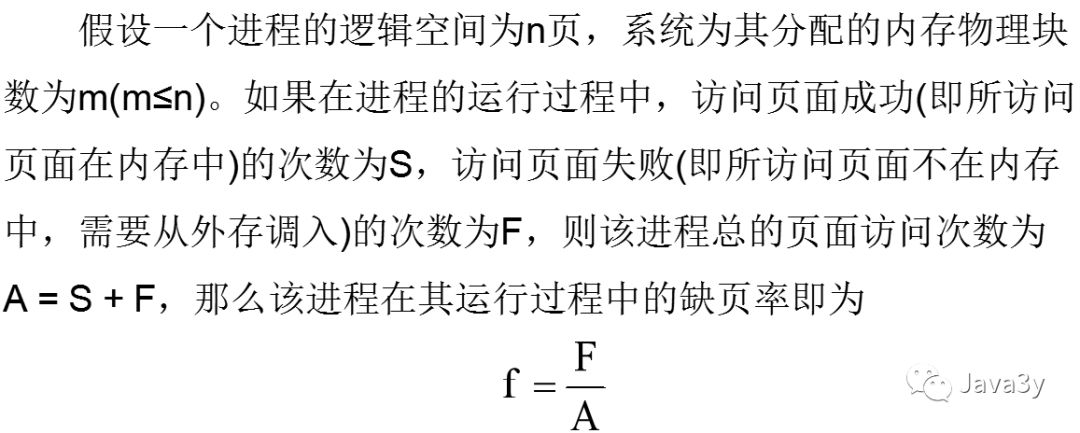
为使进程能够正常运行，必须事先将要执行的那部分程序和数据所在的页面调入内存。现在的问题是：

(1) 系统应在何时调入所需页面：预调页策略；请求调页策略。

(2) 系统应从何处调入这些页面：系统拥有足够的对换区空间，这时可以全部从对换区调入所需页面，以提高调页速度系统缺少足够的对换区空间，这时凡是不会被修改的文件，都直接从文件区调入；而当换出这些页面时，由于它们未被修改，则不必再将它们重写到磁盘(换出)，以后再调入时，仍从文件区直接调入。但对于那些可能被修改的部分，在将它们换出时便须调到对换区，以后需要时再从对换区调入。

(3) 是如何进行调入的：每当程序所要访问的页面未在内存时(存在位为“0”)，便向CPU发出一缺页中断，中断处理程序首先保留CPU环境，分析中断原因后转入缺页中断处理程序

**缺页率**

****

事实上，在缺页中断处理时，当由于空间不足，需要置换部分页面到外存时，选择被置换页面还需要考虑到置换的代价，如页面是否被修改过。没有修改过的页面可以直接放弃，而修改过的页面则必须进行保存，所以处理这两种情况时的时间也是不同的。假设被置换的页面被修改的概率是β，其缺页中断处理时间为ta，被置换页面没有被修改的缺页中断时间为tb，那么，缺页中断处理时间的计算公式为 t=β×ta+(1—β)×tb

#### 页面置换算法

在进程运行过程中，若其所要访问的页面不在内存，而需把它们调入内存，但内存已无空闲空间时，为了保证该进程能正常运行，系统必须从内存中调出一页程序或数据送到磁盘的对换区中。但应将哪个页面调出，须根据一定的算法来确定。通常，把选择换出页面的算法称为页面置换算法(Page-Replacement Algorithms)。置换算法的好坏将直接影响到系统的性能。

**最佳(Optimal)置换算法**

最佳置换算法是由Belady于1966年提出的一种理论上的算法。其所选择的被淘汰页面将是以后永不使用的，或许是在最长(未来)时间内不再被访问的页面。采用最佳置换算法通常可保证获得最低的缺页率。但由于人们目前还无法预知，一个进程在内存的若干个页面中，哪一个页面是**未来最长时间内不再被访问的**，因而该算法是无法实现的，但可以利用该算法去评价其它算法。

**该算法是无法实现的，但是会利用该算法去评价其它算法。**

**先进先出(FIFO)页面置换算法**

　FIFO算法是最早出现的置换算法。该算法总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面予以淘汰。该算法实现简单，只需把一个进程已调入内存的页面按先后次序链接成一个队列，并设置一个指针，称为替换指针，使它总是指向最老的页面。但该算法与进程实际运行的规律不相适应，因为在进程中，有些页面经常被访问，比如，含有全局变量、常用函数、例程等的页面，FIFO算法并不能保证这些页面不被淘汰。

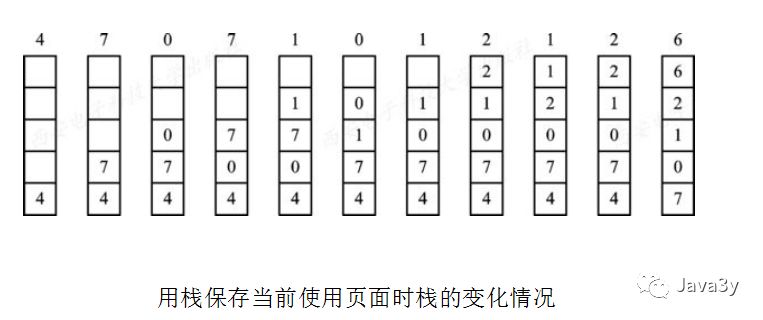
淘汰最先进入内存的页面，但可能那些页面也会被再次使用。所以这个算法也不太行。

**LRU(Least Recently Used)最近最久未使用置换算法**

FIFO置换算法的性能之所以较差，是因为它所依据的条件是各个页面调入内存的时间，而页面调入的先后并不能反映页面的使用情况。最近最久未使用(LRU)的页面置换算法是根据页面调入内存后的使用情况做出决策的。

需要的硬件支持：

可利用一个特殊的栈保存当前使用的各个页面的页面号。每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。因此，栈顶始终是最新被访问页面的编号，而栈底则是最近最久未使用页面的页面号。假定现有一进程，它分有五个物理块，所访问的页面的页面号序列为：4，7，0，7，1，0，1，2，1，2，6



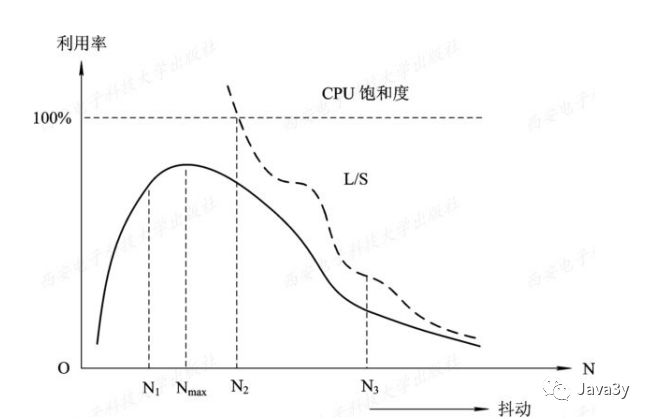
**继续换入的都是缺页，开始也算**

**抖动**

“抖动”与工作集 ：

　　由于请求分页式虚拟存储器系统的性能优越，在正常运行情况下，它能有效地减少内存碎片，提高处理机的利用率和吞吐量，故是目前最常用的一种系统。但如果在系统中运行的进程太多，进程在运行中会频繁地发生缺页情况，这又会对系统的性能产生很大的影响，故还须对请求分页系统的性能做简单的分析。

由于虚拟存储器系统能从逻辑上扩大内存，这时，只需装入一个进程的部分程序和数据便可开始运行，故人们希望在系统中能运行更多的进程，即增加多道程序度，以提高处理机的利用率。\*\*但处理机的实际利用率却如图中的实线所示。



产生“抖动”的原因：

　　发生“抖动”的根本原因是，同时在系统中**运行的进程太多，由此分配给每一个进程的物理块太少，不能满足进程正常运行的基本要求**，致使每个进程在运行时，频繁地出现缺页，必须请求系统将所缺之页调入内存。这会使得在系统中排队等待页面调进/调出的进程数目增加。显然，对磁盘的有效访问时间也随之急剧增加**，造成每个进程的大部分时间都用于页面的换进/换出，**而几乎不能再去做任何有效的工作，从而导致发生处理机的利用率急剧下降并趋于0的情况。我们称此时的进程是处于“抖动”状态。