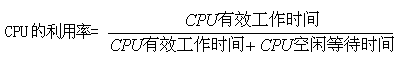
# 操作系统期末总复习

第十章：处理机调度

处理机调度性能评价指标：

1. 处理机利用率：



1. 作业周转时间：批处理用户从作业提交给系统开始，到作业完成为止的时间间隔称作业周转时间：

**带权周转时间**是作业的周转时间***Ti***与系统为它提供服务的时间***TSi***之比，方便比较不同规模作业获得服务质量，**平均带权周转时间：**

是评价系统服务质量的重要指标

1. 系统吞吐量：吞吐量是指在单位时间内系统所完成的作业数。对于很多批处理系统来说，吞吐量越高，经济效益越好。
2. 响应时间：交互式进程从提交一个请求(命令)到接收到响应之间的时间间隔称响应时间，响应时间越短，用户体验性越好。

**适合批处理系统的**调度算法：

1. 先来先服务调度算法（**FCFS，first come first served**）

算法思想：调度最先进入就绪队列的作业或进程，一个进程一旦分得处理机，便执行下去，直到该进程完成或阻塞时，才释放处理机。

* FCFS优点：简单
* FCFS不足：有时短作业所需等待时间过长，对短作业不公平。

1. 短作业优先算法（**SJF，short job first**）

算法思想：该算法从就绪队列或后备队列中**选出所需CPU时间最短的进程或作业**，调度运行或为之分配处理机。

* 优点：
  + **SJF的平均作业周转时间比FCFS要小，调度性能比FCFS好。**
  + **具有最小的平均周转时间（如何证明？）**
* 不足：
  + **实现SJF调度算法需要知道作业所需运行时间，实现上有困难**
  + **SJF算法对长作业不理,甚至导致饥饿问题（长期得不到服务）**

1. 响应比最高者的优先算法（**HRRF**）

算法原理：作业选择同时考虑等待时间和作业大小两个因素，每次分配处理机时，根据响应比选择作业，将CPU分配给响应比最高的作业，

响应比＝1+等待时间/估计运行时间

算法特点(根据响应比计算公式）：

* 短作业估计运行时间短，容易得到较高响应比
* 作业等待时间足够长后，也将获得足够高的响应比，不会发生饥饿现象
* 既照顾短作业又不使长作业的等待时间过长，增强了公平性，改进了调度性能。

**评价：**

* **因HRRF是SJF与FCFS的折衷，**调度性能（平均周转时间）也介入二者之间
* 克服了SJF、FCFS两种算法对某些作业不利的问题

1. 基于优先级调度算法（）

算法思想：

这种算法是根据确定的优先级来选取作业，每次总是选择优先数高的作业

规定用户作业优先数的方法：

（a）由用户规定优先数（外部优先数）

用户提交作业时，根据急迫程度规定适当的优先级，如果涉及收费，优先级越高，收费将越高

（b）由系统计算优先数(内部优先数)

有很多方法，考虑作业运行时间、用户给出的优先数、等待时间、作业类型等，SJF和HRRF也是一种特殊的优先级调度算法；

1. 最短剩余时间优先（**SRTF，Shortest Remaining Time First**）

算法思想：把SJF算法改为抢占模式，又称抢占式短作业优先算法，一个新作业进入就绪状态，如果新作业需要的CPU时间比当前正在执行的作业剩余下来还需的CPU时间短，SRTF强行赶走当前正在执行作业。

**适合分时系统的调度算法：进程调度算法**

1. 时间片轮转调度算法（**Round Robin**）

**算法思想：**

OS为每个进程分配一个时间片（time slice， 通常为10-100ms之间），每个进程在所分配到的时间内拥有CPU使用权。当某个正在执行的进程其所分配的时间片用完时，OS强行从该进程夺取CPU使用权，并将该进程重新放到就绪队列末尾。

1. 优先权调度算法（**HPF Highest-Priority-First**）

算法思想：为每个进程设置一个由数字表示的优先级，作为进程竞争CPU的能力，系统总是选择优先级高进程投入运行。有静态优先级和动态优先级两种确定方法

（1）**静态优先级**

**优先权在创建进程时确定**，且在进程的整个运行期间保持不变，根据**效益原则**和**资源平衡**(鼓励使用相对空闲的外设资源，不鼓励频繁使用CPU资源)两个原则进行优先级计算：

* + 进程类型：系统进程高，用户进程低
  + 资源需求：所需CPU时间多，优先级低，所需I/O时间多，优先级高----**哪个原则？**
  + 用户要求，紧急程度高、付费多，优先级高---**哪个原则？**

（2）**动态优先级**

动态优先级在进程存在过程中不断发生变化，根据**资源平衡、公平原则**计算优先级：

* + - 进程的等待时间：等待时间长，优先级高----**哪个原则？**
    - 进程的运行时间：运行时间长，优先级低 ----**哪个原则？**
    - 进程使用资源的类型等
* 动态优先级方法资源利用率高，公平性好，但开销较大，实现较为复杂

抢占式与非抢占式式算法

1. **非抢占式优先级算法**

算法思想：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，该进程便一直执行下去，直到完成或因发生某事件而放弃处理机时，系统方可重新分配处理机。

1. **抢占式优先级算法**

算法思想：系统把处理机分配给就绪队列中优先权最高的进程，使之执行。但在其执行期间，只要出现了另一个优先权更高的进程，进程调度程序就立即停止当前进程的执行，重新将处理机分配给新到的优先权最高的进程。

多级反馈队列调度算法MFQ

算法思想：

* 设置**多个就绪队列，并为每个队列赋予不同的优先级**。队列1的优先级最高，其余队列逐个降低
* 各队列中进程时间片的大小各不相同，**优先级越高，时间片就越短**
* **新进程进入总是插入队列1的末尾，按FCFS等待调度**。若一个时间片未完成，超时后转入队列2的末尾，按FCFS再次等待调度，如此下去，**如果进入队列*n*则按RR算法调度执行**。
* **仅当队列1为空时，才调度队列2中的进程运行**。若队列I中的进程正执行，此时有新进程进入优先级较高的队列中，**则新进程将抢占运行，原进程转移至下一队列**

第十一章 死锁

预防死锁

1. 破坏“请求和保持”条件

静态资源分配法：**所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源，若能满足则全部分配，进程往下执行，否则一个也不分配，进程等待。**

**优点：简单、易行且安全。**

**缺点：**

**①资源利用率低**

**②对于很多应用，一次提出所有资源申请困难**

1. 破坏“不可抢占”条件

请求**未果则释放已有资源 ：要求已经保持了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。从而破坏了“不可抢占”条件。**

**缺点：**

* **程序复杂，代价高。**
* **因为反复地申请和释放资源，致使进程的执行被无限地推迟。**

1. 破坏“循环等待”条件

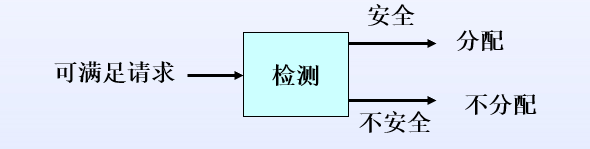
有序资源分配法：这种方法中规定，系统将所有的资源按其类型进行线性排队，并赋予不同的序号。所有进程对资源的请求必须严格**按资源序号递增的次序**提出，这样，在所形成的资源分配图中，不可能再出现环路，因而摒弃了“循环等待”条件

1. 破坏互斥条件

将互斥资源改成共享资源

避免死锁

* 原因：预防死锁的几种策略，会严重地损害了系统性能。因此应放宽对资源使用的限制，来提高系统性能，同时能够避免死锁的发送。
* **死锁避免定义**:在系统运行过程中，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，进程等待。



银行家算法：

当一个进程提出资源请求时，银行家算法要做的工作其要点是：

①判断有无实施资源分配的可能。如果系统有能力，则实施预分配。

②预分配。

③判断分配后系统是否安全，若安全，则真正实施分配。

要求：

* 进程：事先申明所需资源最大量（并不分配）
* 系统：对每个可满足的资源申请命令进行安全性检查。

***资源分配的安全性***是指要保证至少有一个进程能够运行到结束，并且通过回收该进程所占用的资源再分配能依次使其他进程运行结束，然后继续回收资源、继续分配等，直到全部进程运行结束。如果计算出的资源分配是不安全的，系统将拒绝分配。

**（1）数据结构：**

1. int **Available[1..m]; //系统可用资源**
   * **Available[i]=k表示系统中现有Ri类资源k个。**
2. int Max **[1..n][1..m]; //进程最大需求**
   * **Need[ i,j ]=k表示进程Pi最多需要资源类Rj中k个资源实例。**
3. **int Allocation[1..n] [1..m]; //当前分配**
   * **Allocation[i,j]=k表示进程Pi当前已分得k个Rj类资源。**
4. **int Need[1..n] [1..m]; //尚需资源**
   * **Need[i,j]=k表示进程Pi还需要分得k个Rj类资源才能完成其任务**
5. int **Request[1..n] [1..m]; //当前请求**

**Request[i,j]=k表示进程Pi申请Rj类资源中k个资源实例。**

临时变量：

**int Work[1..m];**

**boolean Finish:[1..n];**

**（2）银行家算法---资源分配算法**

假设某一时刻，进程Pi提出了资源请求Request[j] ，银行家算法的操作过程可用以下各步表示：

(1) 分配能力检查：如果Request［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［j］≤Available［j］，便转向步骤(3)；否则， 表示尚无足够资源，Pi须等待。

(3)预分配：系统对进程Pi实施资源的预分配

* + Available［j］=Available［j］-Requesti［j］;
  + Allocation［i,j］=Allocation［i,j］+Requesti［j］;
  + Need［i,j］=Need［i,j］-Requesti［j］;

(4)安全性检测：通过调用安全性算法判断此次分配是否要真正实施。调用安全性算法，根据返回值判断此次分配的真正实施是否安全。如果安全，则真正实施分配；如果不安全，则取消预分配,让Pi等待。

**（3）安全性检测算法**

(1)设置两个向量：

① 工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时，Work∶=Available;

② Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish［i］∶=false; 当有足够资源分配给进程时，再令Finish［i］∶=true。

(2)从进程集合中找到一个能满足下述条件的进程：

① Finish［i］=false;

② Need［i,j］≤Work［j］； 若找到， 执行步骤(3)， 否则，执行步骤(4)。

(3)当进程Pi获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］∶=Work［i］+Allocation［i,j］;

Finish［i］∶=true;

go to step 2;

(4)如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。

产生死锁的必要条件

**⑴互斥条件（ mutual exclusion ）：**一个资源在同一时刻只能分配给一个进程。任何时刻一个资源仅为一个进程独占，若另一进程请求一个已被占用的资源时，必须等到占用者释放资源。

**⑵请求和保持条件(hold-while-applying) ：**进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有。

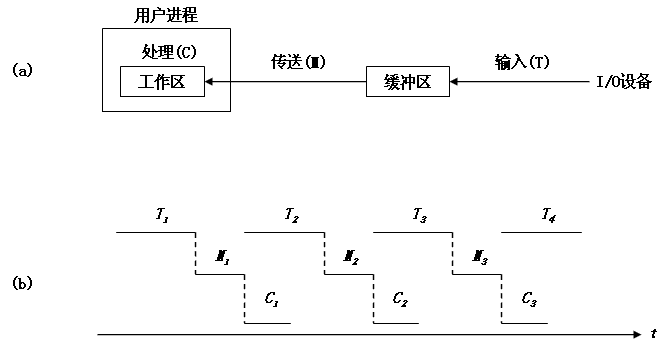
**⑶不可抢占条件(non preemption)**：进程已获得的资源，在未使用完之前不能被抢占，只能在进程使用完时由自己释放。

**⑷循环等待条件(circular wait)：叫环路等待条件，**存在一个循环等待链，其中，每一个进程分别等待它前一个进程所持有的资源，造成永远等待

第十三章 I/O系统

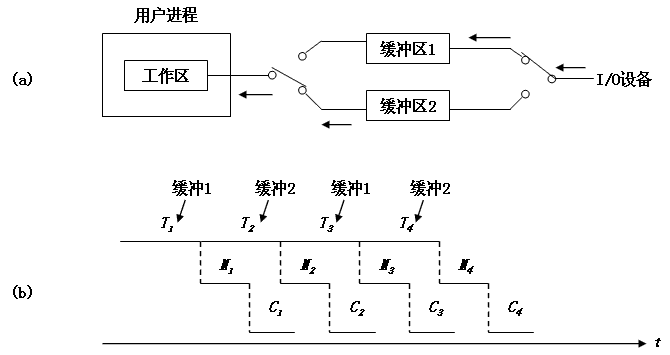
单缓冲区

* 在单缓冲情况下，每当用户进程发出一I/O请求时，操作系统便在主存中为之分配一缓冲区，如图所示。系统对每一块数据的处理时间表示为Max(C,T)+M。



双缓冲区

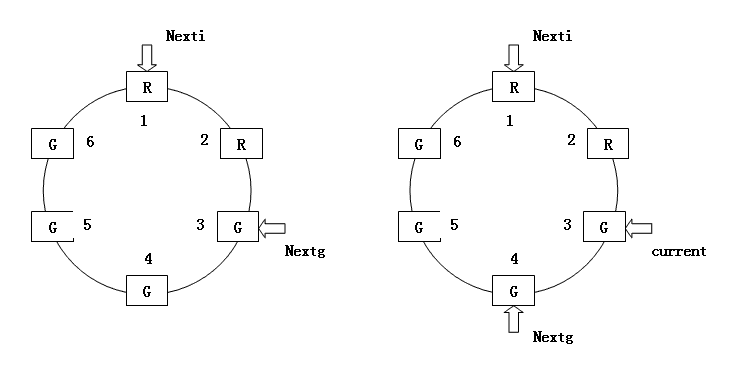
也称为缓冲对换(Buffer Swapping)。在设备输入时，先将数据送入第一缓冲区，装满后便转向第二缓冲区。此时操作系统可以从第一缓冲区中移出数据，并送入用户进程(见图)。接着由CPU对数据进行计算。在双缓冲时，系统处理一块数据的时间可以粗略地认为是Max(C，M，T) 。



环形缓冲区

**(1) 多个缓冲区：在环形缓冲中包括多个缓冲区，其每个缓冲区的大小相同。作为输入的多缓冲区可分为三种类型: 用于装输入数据的空缓冲区R、已装满数据的缓冲区G以及计算进程正在使用的现行工作缓冲区C，如图7-23所示。**

**(2) 多个指针：用于指示计算进程下一个可用缓冲区G的指针Nextg、 指示输入进程下次可用的空缓冲区R的指针Nexti，用于指示计算进程正在使用的缓冲区C的指针Current。**



I/O控制方式

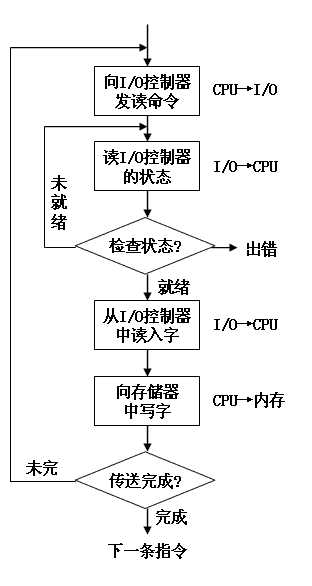
1. 轮询控制方式

①在处理机向控制器发出一条I/O指令，启动输入设备输入数据时，要同时把状态寄存器中的忙/闲标志busy置为1，

②然后便不断地循环测试busy（称为轮询）。当busy=1时，表示输入机尚未输完一个字(符)，处理机应继续对该标志进行测试，直至busy=0，表明输入机已将输入数据送入控制器的数据寄存器中。

③于是处理机将数据寄存器中的数据取出，送入内存指定单元中，这样便完成了一个字(符)的I/O。接着再去启动读下一个数据，并置busy=1。

目前轮询控制方式主要用于在单片机环境输入输出。



1. 中断控制方式

**I/O控制的实质是如何保证发送下次I/O命令(包括读写I/O设备)前，前一条命令已经完成。一种简单的中断控制过程如下：**

**①当某进程要从I/O设备输入数据时，便向相应设备的命令寄存器，发出一条I/O命令，**

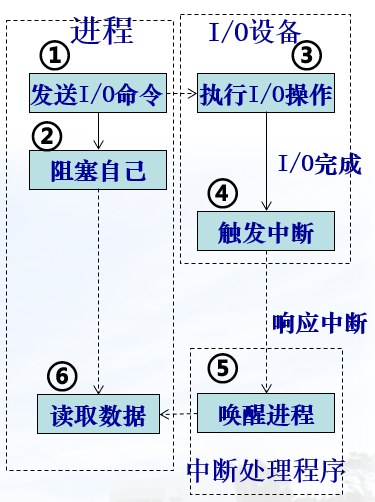
**②然后进程阻塞自己，CPU分配给其它进程。**

**③ I/O设备按照该命令的要求，去执行输入/输出操作。此时，CPU与I/O设备并行操作。**

**④一旦输入操作完成，数据进入设备数据寄存器，I/O设备便向CPU发送一中断信号。**

**⑤CPU响应中断，执行相应设备中断处理程序，唤醒等待进程。**

1. **被唤醒的进程从I/O设备数据寄存器读取输入数据。**



1. **直接存储器访问(DMA)方式**

**1）直接存储器访问方式的引入**

**· 采用中断驱动I/O方式时的CPU，是以字（节）为单位进行干预的。如果将这种方式用于块设备的I/O，显然是极其低效的。**

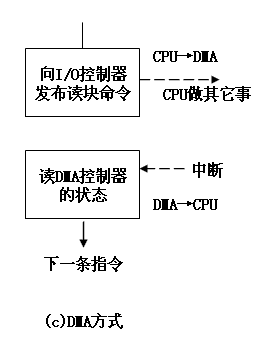
**·为了进一步减少CPU对I/O的干预，而引入了直接存储器访问方式。DMA方式的特点：**

**①数据传输的基本单位是数据块；**

**②所传送的数据，是从设备直接送入内存的，或者相反；**

**③仅在传送一个或多个数据块的开始和结束时，才需CPU干预，整块数据的传送是在控制器的控制下完成的。**

**DMA方式较之中断驱动方式，又进一步提高了CPU与I/O设备的并行操作程度。**



**（2）DMA控制器的组成**

**①主机（存储器）与DMA控制器的接口；**

**②DMA控制器与块设备的接口；**

**③I/O控制逻辑。**

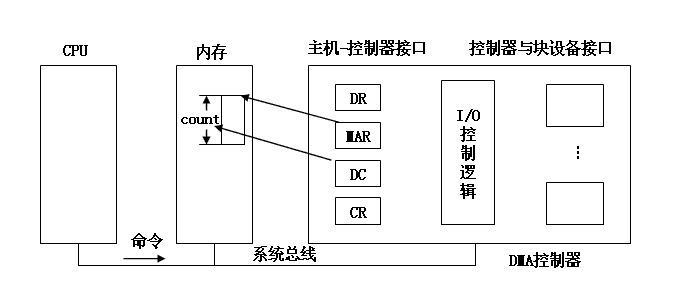
**为了实现在主机与控制器之间，成块数据的直接交换，必须在DMA控制器中，设置如下四类寄存器：**

**①命令/状态寄存器CR；**

**②内存地址寄存器MAR；**

**③数据寄存器DR；**

**④数据计数器DC。**

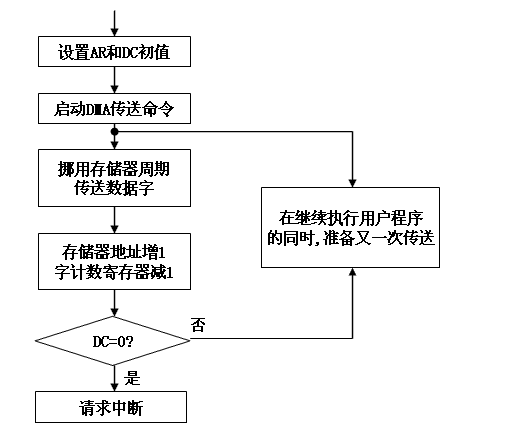


**（3）DMA工作过程**

**①当CPU要从磁盘读入一数据块时，便向磁盘控制器发送一条读命令。该命令被送入命令寄存器CR中。同时，将本次要读入数据在内存的起始目标地址，送入内存地址寄存器MAR中。将要读数据的字(节)数，送入数据计数器DC中。将磁盘中的源地址，直接送至DMA控制器的I/O控制逻辑上。**

**②启动DMA控制器进行数据传送。   
③当DMA控制器已从磁盘中读入一个字(节)的数据，并送入数据寄存器DR后，再挪用一个存储器周期（？），将该字（节）传送到MAR所指示的内存单元中。**

**④然后便对MAR内容加1，将DC内容减1，若减1后DC内容不为0，表示传送未完，便继续传送下一个字(节)；否则，由DMA控制器发出中断请求。**



1. **通道控制方式（channel）**

**通道是一种特殊功能的处理器，有自己的指令和程序，可以实现对外围设备的统一管理和外围设备与内存之间的数据传送**

**引入通道的目的——进一步减少数据输入输出对整个系统运行效率的影响**

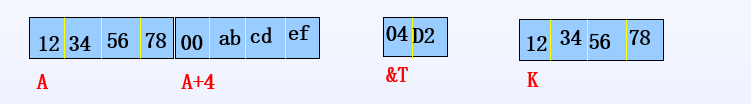
**与DMA方式相比，通道方式增加了CPU与通道操作的并行能力，增加了通道之间以及同一通道内各设备间的并行操作能力，为用户提供了灵活增加外设的可能性**

**其他特性，请参考教材**

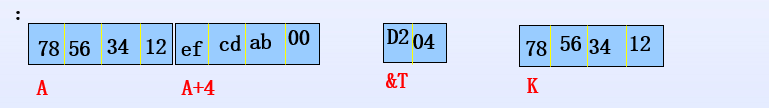
第八章 网络编程

* **套接字：含有进程接收信息的完整地址（Socket地址：IP地址、端口号）**
* 字节序：

大端模式：高位在低地址字节，一般网络序



小端模式：**低位在低地址字节，一般主机序**



* 网络通信API函数

(一）客户端

（1）创建套接字

int socket(int domain, int type , int protocol);

示例：client\_sock = socket(AF\_INET , SOCK\_STREAM, 0);

（2） **connect 函数**

int connect (int client\_sock , struct sockaddr \*serv\_addr , int addrlen);

（3）包装函数open\_client\_sock

（二）服务器端

int bind(int serv\_sock, struct sockaddr \*my\_addr , int addrlen);

int listen(int serv\_sock, int backlog);

open\_listen\_sock包装函数：

int accept(int listen\_sock, struct sockaddr \*addr , int \*addrlen);

（三）数据收发

ssize\_t send(int sock, const void \*buff, size\_t nbytes, int flags);

返回: 若成功则为实际发送字节数，若出错则为SOCKET\_ERROR.

ssize\_t recv(int sock, void \*buff, size\_t nbytes, int flags);

返回: 若成功返回实际接收字节数，若出错则为SOCKET\_ERROR，如果recv函数在等待协议接收数据时网络中断了，则返回0

第十二章 存储管理

* **多级存储体系的原因：以较低的成本获得容量大、速度快的存储资源**
* 逻辑地址与物理地址

用户能看到的程序代码、变量、堆栈的地址为逻辑地址（又称程序地址，虚拟地址VA），按用户方便使用的方式安排，比如：从0开始编码、一个程序段、数据段的逻辑地址是连续的

它们所在实际内存区域实际地址为物理地址PA，代码、变量、堆栈的物理地址是用户看不到的

一般来说，虚拟地址≠物理地址

将逻辑地址与物理地址分开，有很多好处，如：实现进程隔离，提高内容利用率、实现进程和内核保护

* 连续分配储存管理方式

基本思想：为每一个用户程序分配一个地址连续的内存块，程序中代码或数据的逻辑地址相邻，对应存储单元物理地址也相邻。有四类分配方法：

* + **单一连续分配**
  + **固定分区分配**
  + **动态分区分配**
  + **动态可重定位分配**

1. **单一连续区存储管理**

**管理方法：内存分为两个连续区域：系统区，用户区。系统区为操作系统，用户区装载一道应用程序。适合单道程序系统，如CP/M、MS-DOS、RTII等**

* **不足：应用领域受限**

1. **固定分区管理**

**内存分配方法：把内存划分为若干个固定大小的连续分区**。**一旦划分好，在系统运行期间不再重新划分**

* **好处：支持多道程序设计**
* **不足：若分区比作业大，超出部分（分区内碎片）被浪费掉，存储器利用率是其最大问题；分区数固定，限制活跃进程数目**

1. **动态分区分配**

**内存分配方法：系统存储器不预先划分分区，在每次作业装入时，根据作业大小要求，从可用存储空间中切一块大小与所要求相近的存储区分配给作业。**

* **优点：解决了固定分区的内碎片问题**
* **不足：**
  + **存在外碎片（小得难以分配出去的分区）问题，存储器利用率仍然很低。（性能）**
  + **难以实现存储共享、存储扩充（功能）**
* **内存分配（也适合固定分区）**
* 利用某种算法（**为何需要算法**），从空闲分区表（链）中找到所需大小分区，将其分配给请求作业（或进程）使用。分区选择一般有三种方法：
* **最先匹配法(first-fit)(首次适配算法)：空闲区按地址从小到大排列，拿在空闲区表中第一个符合要求的分区，从中切出要求大小的内存块分配出去**
  + **好处：较大的空闲分区可以被保留在内存高端。**
  + **不足：但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。**
* **最佳匹配法(best-fit)：空闲区按其大小从小到大排列，找到其大小与要求相差最小的空闲分区（即第一个满足要求的分区进行分配）**
  + **较大的空闲分区可以被保留**
  + **但会形成较多无法利用的较小分区(外部碎片)**
* **最差匹配法(worst-fit)：空闲区按其大小从大到小排列，找到最大的空闲分区（即空闲区表中第一个分区），从中切出要求大小的内存块**
  + **基本不留下小空闲分区，但较大的空闲分区不被保留。**

**地址转换（地址重定位、地址映射）**

MMU设置一个**重定位寄存器（或称基址寄存器），**用来存放程序(数据)在内存中的起始地址，指令执行时，处理机给出相对地址，MMU按以下方法计算绝对地址是：**绝对地址=相对地址+重定位寄存器**

动态分区的改进

1. **动态可重定位分区分配（内存紧凑）**

**连续分配方式的一个重要特点是，一个系统或用户程序必须被装入一片连续的内存空间中。当一台计算机运行了一段时间后，它的内存空间将会被分割成许多小的分区（若不能使用，就成为碎片），而缺乏大的空闲空间。**

**内存紧凑方法将内存中的所有作业进行移动，使它们全都相邻接。这样，即可把原来分散的多个小分区拼接成一个大分区，这时就可把一个作业装入该区。**

**这种方法虽然能较好地应对动态分区内存管理的碎片问题，但也有知名的不足：**

**内存紧凑开销大，耗时**

**内存紧凑期间，用户作业（或进程）都需暂停执行**

1. **多重分区存储管理**

**基本思想：给一个作业分配多个内存分区，每个分区可装入作业的一个或多个段，使一些尺寸较小的分区也能得到利用，提高内存利用率。**

**优点：**

**可以实现分区共享**

**诸进程可以共享数据分区。**

**要求：**

**系统需设置多对基址－界限寄存器，实现难度大。**

**不足**

**受硬件实现的制约，分配给作业的分区数不能很多，该方法仍然没有从根本上解决动态分区的碎片问题**

**改进：用段表替换多对基址-界限基础（段式存储管理，后面讲）**

（四）**分页存储管理方式**

**只要系统可用物理内存块之和满足作业内存容量要求，就可为作业分配内存，任何一个空闲的内存块都是可以利用的，因此，分页存储管理圆满地解决了可变分区的碎片问题。**

1. **分页模式下逻辑地址运算**

**相对地址为：**L**A**=**P×L+d**

* **页号P=LA / L**

页内偏移量**d=LA-P\*L=LA mod L**

1. 分页环境下物理地址计算

**物理地址为：PA**=**Q×L+d**

* **若物理地址为PA，块长为L，则块号Q=PA / L**

块内地址**d=PA-Q\*L=PA mod L**

* 分页系统中，地址转换（地址重定位）的流程是：

**①从逻辑地址LA中提取页号P；**

**②用页号P查页表，得到物理块号Q；**

**③用物理块号Q和页内地址d，计算物理地址PA。**

地址变换机构

1. 地址变换过程

**①从逻辑地址LA中提取页号P、页内地址d**

**②若页号>页表长度，则越界异常**

**③页号+页表地址得到页表项地址，**

**④查页表，得物理块块号Q,即PT[P]**

1. **将物理块号Q与页内地址拼接，的物理地址PA**
2. 地址变换机构的组成

**①内存页表；**

**②页表地址寄存器(如CR3);**

**③地址转换电路；**

**④越界中断检测与中断处理**

1. 软硬件分工

* 软件(OS)：
* 创建页表(装载程序时)；
* 越界异常处理
* 硬件(MMU)：
* 地址变换；
* 异常监测

1. 地址转换效率计算方法

记快表查找时间为λ，实际物理内存访问所需时间为t,内存有效访问时间为EAT，则有效范围时间EAT计算方法如下：

被访问页在内存，对应页表项在在快表：

**EAT= λ+t**

被访问页在内存，对应页表项不在快表：

快表、主存页表同时查找：**EAT= t +t,** （一般不作此假定 ）

先查快表，再查页表：EAT= **λ+t +t**

快表命中率为a,先查快表，再查页表，不考虑快表更新：

***EAT= a×* *λ*+(1-*a*)(*λ*+*t*)+*t* =**a( λ+t)+(1-a)( λ+2t)

页式储存管理内存共享方法

只要让两个进程各自页表中的有关表项指向共享的数据页框，**但对具有连续虚拟地址的分页系统，内存共享仍然不方便**：很难恰好使特定页面全部单元正好都是共享数据。

（四）分段存储管理方式

采用分区和分页存储管理，存在一些问题：

* **信息共享问题。没有在逻辑上划分程序，有时在一个页面中会包含毫不相关的两段代码，使得信息共享不方便。**
* **动态链接问题。一个大的程序可能包含数百、数千个程序模块构成，很多用户希望在程序执行过程中装入、动态连接所需模块，降低对内存需求，提高装载效率，而分区管理和页式管理只能采用静态链接。**

1. **基本思想：**

**把程序按内容或过程(函数)关系分成段，每段有自己的名字，每个段一个地址空间**

* + **在装入内存后，每个段占用一片连续的存储区域，而各段之间可以不连续，可按段进行存储共享和保护。**
  + **分段存储方式给用户提供了更方便灵活的程序设计环境。**
  + **段式管理可将那些经常访问的段驻留内存，而把那些在将来一段时间内不被访问的段放入外存，待需要时自动调入的方法实现二维虚拟存储器，较好地支持虚拟存储器**

1. **内存分配**

**在分段存储管理中内存以段为单位进行分配， 给每个段分配一块连续的内存分区。各分区的大小由对应段的大小决定。**

**这些类似于动态分区分配方式，其区别： 在分段存储管理系统中， 一个作业或进程可以有多个段， 这些段可以离散地放入内存的不同的分区中。**

（五）段页式存储管理方式

1 **段页式存储管理方式的引入**

**★分页系统是以页面作为内存分配的基本单位，能有效地提高内存利用率，但信息共享等不方便；**

**★分段系统是以段作为内存分配的基本单位，它能够更好地满足用户多方面的需要（信息共享、动态链接等），但采用分区方式管理物理内存，仍然存在碎片问题；**

**★段页式存储管理方式是对两种存储管理方式的“各取所长”，既具有分段系统的便于实现、分段可共享、易于保护、可动态链接等一系列优点，又能像分页系统那样，很好地解决内存的外部碎片问题。**

**2 基本思想**

**地址空间划分：作业的地址空间仍按其逻辑结构分段。每个段又被进一步分成若干大小相同的页面。内存空间则分成与页面大小相等的物理块。**

**作业装入：在作业装入内存时，一个页面装入内存一个块中。各个段的页面装入内存时可以占用互不连续的内存块**

**采用虚拟存储技术：作业的页面不必一次全部装入内存，而采用请求页式管理的方法，当要访问的页面不在内存时，就产生缺页中断**

请求分页存储管理

**实现思想：**

* 应用程序在运行之前，仅**须将那些当前要运行的少数页面或段，先装入内存便可运行**，其余部分暂留在外存（虚拟存储器）上。
* 程序在运行时，如果它所要访问的页已调入内存，便可继续执行；但**如果程序所要访问的页尚未调入内存（称为缺页），便发出缺页中断请求，由OS利用指定页面调入内存**。
* 若此时**内存已满，无法再装入新页，OS应将内存中暂时不用的页调至外存（虚拟存储器）上**，腾出页帧后，再所需页面调入内存（**页面置换**），使程序得以继续执行

页面置换算法

1. **最佳置换算法(OPT)**

**算法思想：总是选择以后永不使用的，或许是在最长(未来)时间内不再被访问的页面，进行替换。**

**采用最佳置换算法，通常可保证获得最低的缺页率。但这种算法仅仅是理论上，应用中无法实现，因为无从预测进程未来的页面走向。**

1. **先进先出页面置换算法（FIFO）**

算法思想：当需要调入新的页面，但无可用空闲物理块时，总是**选择在内存中驻留时间最长的页并予以淘汰，用腾出的物理块来载入新的页面**。

**但采用先进先出算法，有一个异常现象（Belady异常）：有时分配的物理块愈多， 缺页率反而升高**

先进先出算法具有不稳定特征，一般不用作页面置换算法，但因为它简单，被用于高速缓存替换算法。

1. **LRU置换算法（最久最近未使用算法）**

**算法思想：根据程序局部性原理，认为程序最近一段时间内经常访问的地址，也会是不久将来经常访问的地址，认为最后一次访问时间距离当前时间最久的页面，是不久将来访问可能性比较小的页面。因此选择最后一次访问时间距离当前时间最长的一页并淘汰之，即淘汰没有使用的时间最长的页。**

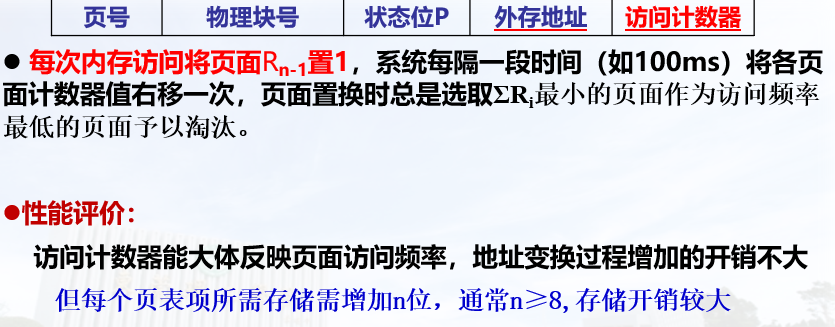
**很多研究成果表明，采用LRU算法缺页率比FIFO要少，能带来更高的系统性能，被广泛看好**

**LRU算法需要MMU在每次访存操作后，自动记录页面访问时间，或页面访问顺序，以便置换算法找到最近最久未使用页面。一般有寄存器、和堆栈两种方法。**

1. **最少使用LFU置换算法（计数器法）**

**算法思想：选择在最近时期使用最少的页面作为淘汰页**

**实现方法：为内存中的每个页面设置一个访问计数器**R=R**n-1**R**n-2**R**n-3** … R**2**R**1**R**0 ，记录该页面被访问的频率。页表结构为：**



1. Clock置换算法

该置换算法仅为每个页面增加一个访问位，将内存中所有页面通过链接指针链接成一个循环队列



算法思想：每次页面置换时，仅检查访问位A，若A为0，就直接替换之。若A为1，则将其置0，再检查队列中下一个页面。若队列中所有页面的访问位A全为1，则置0后，返回对首继续检查。结果，上一轮检查以来，A没有被置1的第一个页面，将被选做淘汰页面。

* 性能评价：

粗略地实现了LRU算法功能

页表开销仅增加1 bit，地址转换过程仅增加一次访问位置位操作

1. 页面缓冲算法PBA

**设计思想(以VAX/VMS操作系统为例)：**

**在系统设置两个物理块链表：**

* 1. **空闲页面链表是一个空闲物理块链表，用于分配给频繁发生缺页的进程，当有一个未被修改的页要换出时，并不立即将新页面读入该物理块，而将该物理块挂在空闲链表的末尾；**
  2. **修改页面链表由已修改的页面所形成的链表，当一个已修改页面被选做淘汰页时，若暂不把已修改换出页面写回磁盘，将它们链到一个修改页面链末尾，当被换出页面数目达到一定值时，再由OS将它们一起写回到磁盘上。**

**每次需读入新的页面时，若新的页面在两个物理块链表之一中，则直接将该物理块取出直接使用，因为其中保存了该页面淘汰前的内容；否则，从某个链表摘取第一个物理块，将心页面内容读入其中。**

**性能分析：**

**①PBA算法显著地降低了页面换进、换出的频率，显著减少使磁盘I/O的次数，提高页面置换算法效率；**