一、操作系统引论

- 目前存在着多种类型的 OS，不同类型的 OS，其目标各有所侧重。通常在计算机硬件上配置的 OS，其目标有以下几点：

  1. 方便性
  2. 有效性
  3. 可扩充性
  4. 开放性

- 作为软件接口，给用户提供三种方式：

  (1) 命令方式。这是指由 OS 提供了一组联机命令(语言)， 用户可通过键盘输入有关命令，来直接操纵计算机系统。

  (2) 系统调用方式。OS 提供了一组系统调用，用户可在自己的应用程序中通过相应的系统调用，来操纵计算机。

  (3) 图形、窗口方式。用户通过屏幕上的窗口和图标来操纵计算机系统和运行自己的程序。

- 在一个计算机系统中，通常都含有各种各样的硬件和软件资源。归纳起来可将资源分为四类：处理器、存储器、 I/O 设备以及信息(数据和程序)。相应地，OS 的主要功能也正是针对这四类资源进行有效的管理，即：（四类资源管理者）

  处理机管理， 用于分配和控制处理机；

  存储器管理，主要负责内存的分配与回收；（最重要是内存管理）

  I/O 设备管理，负责 I/O 设备的分配与操纵；

  文件管理，负责文件的存取、共享和保护。

  可见，OS 确是计算机系统资源的管理者。事实上，当今世界上广为流行的一个关于 OS 作用的观点，正是把 OS 作为计算机系统的资源管理者。

- OS 用作扩充机器

  对于一台完全无软件的计算机系统(即**裸机**)，即使其功能再强，也必定是难于使用的。如果我们在裸机上覆盖上一层 I/O 设备管理软件，用户便可利用它所提供的 I/O 命令，来进行数据输入和打印输出。此时用户所看到的机器， 将是一台比

裸机功能更强、使用更方便的机器。

通常把覆盖了软件的机器称为**扩充机器或虚机器（虚拟机）**。如果我们又在第一层软件上再覆盖上一层文件管理软件，则用户可利用该软件提供的文件存取命令，来进行文件的存取。此时，用户所看到的是台功能更强的虚机器。如果我们又在文件管理软件上再覆盖一层面向用户的窗口软件，则用户便可在窗口环境下方便地使用计算机，形成一台功能更强的虚机器。

- 操作系统的发展过程

无操作系统的计算机系统　第一台计算机诞生(1945 年)到 50 年代中期的计算机，属于第一代，这时还未出现 OS。

人工操作方式有以下两方面的缺点：

(1) 用户独占全机。 (2) CPU 等待人工操作。

单道批处理系统（50 年代）

过程：一批作业以脱机方式输入到磁带上，在监督程序的控制下连续处理。

该系统的主要特征如下：

(1) 自动性 无人工干预。（缺少人机交互的特性，但相比之前较好）

(2) 顺序性 按进入内存的先后执行。（很难实现程序间的合作性 共享）

(3) 单道性 内存中只保存一道作业。（资源利用率低、吞吐量少）

注：作业一般仅用于批处理操作系统中。整个过程从输入计算机外存开始到计算机输出结果为止这一任务过程成为作业。

摩尔定律：每隔 18 个月，硬件数量增加一倍。

多道批处理系统（60 年代）

多道：内存中同时存放多个相互独立的程序。

多道技术是**共享**的基础。

(1) 多道性 内存中多道程序可并发执行。

(2) 无序性 完成时间和进入内存先后无关。

(3) 调度性 作业从提交（送到系统的外存）到完成经过两次调度。

1）作业调度 外存→内存（选多个）

2）进程调度 分配处理机（选 1 个）

多道批处理系统的**优缺点 ：**

优点：

(1) 资源利用率高。

(2) 系统吞吐量大。

吞吐量：单位时间内完成的总工作量

吞吐量大的原因：（1）资源忙。

（2）完成或运行不下去时才切换。

缺点：

(3) 平均周转时间长。

作业周转时间：从作业进入系统开始到完成并退出系统经历的时间。

(4) 无交互能力。

修改和调试极不方便。

分时系统（60 年代）

定义：一台主机上连接了多个终端，同时允许多个用户通过自己的终端，以交互方

式使用计算机，共享主机的资源。

产生动力：

(1) 人—机交互。（边运行边调试）

(2) 共享主机。 （设备昂贵）

(3) 便于用户上机。（在终端上输入和控制）

特征：

(1) 多路性 即同时性，宏观上同时微观上轮流。

(2) 独立性 每个用户感觉独占主机。

(3) 及时性 较短时间响应（2-3 秒）。

(4) 交互性。

实时系统（60 年代）

所谓“实时”，是表示“及时”，而实时系统(Real-Time System)是指系统能及时(或即时)响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有实时任务协调一致地运行。

应用需求：

（1）实时控制：工业生产、武器控制飞机的自动驾驶

计算机操作系统的三种基本类型：多道批处理系统、分时系统、实时系统。

（2）实时信息处理：订票系统（加锁操作解决数据冗余）

- 操作系统的基本特征

并发性（最重要特征）

并行性和并发性是既相似又有区别的两个概念，**并行性**是指两个或多个事件在同一时刻发生；而**并发性**是指两个或多个事件在同一时间间隔内发生。在多道程序环境下，并发性是指在一段时间内，宏观上有多个程序在同时运行，但在单处理机系统（宏观并发微观串行）中，每一时刻却仅能有一道程序执行，故微观上这些程序只能是分时地交替执行，倘若在计算机系统中有多个处理机，则这些可以并发执行的程序便可被分配到多个处理机上（多处理机系统：可能有并行。），实现并行执行，即利用每个处理机来处理一个可并发执行的程序，这样，多个程序便可同时执行。

共享性

两种资源共享方式：

（1）互斥共享方式 对临界资源的访问

临界资源：在一段时间内只允许一个进程访问的资源称为临界资源或独占资源。如打印机、磁带机等硬件、栈、变量和表格等

（2）同时访问方式

多个进程同时访问的资源，如：磁盘、重写码写的文件。

并发和**共享**是操作系统的两个**最基本的特征**

虚拟性

虚拟：是指通过某种技术把一个物理实体变为若干个逻辑上的对应物。如：虚拟处理机、虚拟内存、 虚拟外部设备和虚拟信道等。

虚拟是通过分时来实现的。

在虚拟处理机技术中，是通过多道程序设计技术，让多道程序并发执行的方法，来分时使用一台处理机的。此时， 虽然只有一台处理机，但它能同时为多个用户服务，使每个终端用户都认为是有一个 CPU 在专门为他服务。亦即， 利用多道程序设计技术，把一台物理上的 CPU 虚拟为多台逻辑上的 CPU，也称为**虚拟处理机**，

我们把用户所感觉到的 CPU 称为虚拟处理器。

异步性

进程以人们不可预知的速度向前推进。

- 操作系统的五大功能：
  - 资源管理
    1. 处理机管理（硬件）
    2. 存储器管理（硬件）
    3. 设备管理（硬件）
    4. 文件管理（软件）
  - 5. 用户接口
    - 命令接口（用户和计算机系统之间）
      - 联机用户接口（有交互）
      - 脱机用户接口（无交互）
    - 程序接口
    - 图形接口

联机用户接口

适用：几乎所有计算机的操作系统中。

组成：**命令+终端处理程序+命令解释程序**

过程：用户在终端或控制台上每键入一条命令后，系统便立即转入命令解释程序，对该命令加以解释并执行该命令。在完成指定功能后，控制又返回到终端或控制台上，等待用户键入下一条命令。这样，用户可通过先后键入不同命令的方式，来实现对作业的控制，直至作业完成。

用户在键盘上输入**命令；**

终端处理程序接收命令并显示在屏幕上；

命令解释程序解释并执行该**命令**。（操作系统的最高层）

脱机用户接口

适用：批处理系统。又称批处理用户接口。（由预输入过程）

组成：**JCL+作业说明+命令解释程序**

JCL：作业控制语言 Job Control Language

过程：用户把对作业的控制用 JCL 写在**作业说明书**上，**命令解释程序按照作业说明书**解释并执行。

程序接口

目的：为用户程序访问系统资源而设置。（是用户程序取得操作系统服务的惟一途径）

组成：一组系统调用

系统调用：一个系统调用是一个能完成特定功能的子程序。

- 操作系统的结构设计

  第一代：无结构操作系统

  第二代：模块化 OS 结构

  第三代：分层式 OS 结构

  第四代：微内核 OS 结构 （20 世纪 90 年代）

  我们把第一代至第三代的 OS 结构， 称为传统的 OS 结构，而把微内核的 OS 结构称为现代 OS 结构。

二、进程管理

- **进程**的基本概念

  程序在并发环境中的执行过程

资源分配和独立运行的基本单位

进程定义

进程是进程实体的运行过程，是系统进行资源分配和调度的基本单位。

- 程序顺序执行时的特征

(1) 顺序性：处理机的操作严格按照程序所规定的顺序执行，即每一个操作必须在下一个操作执行前结束。

(2) 封闭性： 程序在封闭环境下执行，结果不受外界环境影响。

(3) 可再现性：只要环境和初始条件相同，程序重复执行时总得到相同结果。

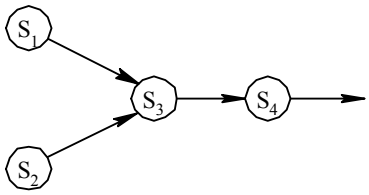
- 前趋图

前趋图(Precedence Graph)是一个有向无循环图，记为 DAG(Directed Acyclic Graph)，用于描述进程之间执行的前后关系。图中的每个结点可用于描述一个程序段或进程，乃至一条语句；结点间的有向边则用于表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)“→”。

→={(Pi, Pj)|Pi must complete before Pj may start}，如果(Pi, Pj)∈→,可写成 Pi→Pj，称 Pi 是 Pj 的直接前趋，而称 Pj 是 Pi 的直接后继。在前趋图中，把没有前趋的结点称为初始结点(Initial Node)，把没有后继的结点称为终止结点(Final Node)。

每个结点还具有一个重量(Weight)，用于表示该结点所含有的程序量或结点的执行时间。

- 程序的并发执行



程序并发执行的特征

1) 间断性 共享、合作、制约导致：执行——暂停——执行

2) 失去封闭性 资源状态由多程序改变

3) 不可再现性 相同环境和初始条件，重复执行结果不同。

- 进程的特征

1) 结构特征

PCB 进程控制块(头脑,一定处于内存当中)→动态特征的集中反映 Process Control Block

程序段 → 描述要完成的功能

数据段 → 操作对象及工作区

程序段和数据段可能存储与外存中，运行过程中从外存动态调入

2) 动态性

进程最基本的特征是**动态性**。

进程的生命周期：进程由创建而产生，由调度而执行，有撤销而消亡的过程。

3) 并发性：多个进程同在内存中，且能在一段时间同时运行。

4) 独立性：进程是一个能独立运行、独立分配资源、独立接收调度的基本单位。

5) 异步性：进程按各自独立的、不可预知的速度向前推进。

- 进程和程序的关系

（1）进程是一个动态概念，程序是一个静态概念。

（2）进程具有并行特征，程序没有。

（3）进程是竞争资源的基本单位。

（4）一个程序对应多个进程，一个进程为多个程序服务。

- 进程的三种基本状态

  1) 就绪(Ready)状态　进程已经分配了除处理机以外的所有必要资源，只要获得处理机就能够执行的状态。

  这样的进程可能有多个，通常排成一个队列，称就绪队列。

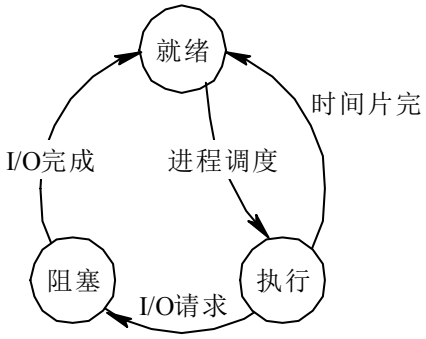
  2) 执行状态　已经获得 CPU，正在运行。

  在单处理机系统只有一个进程处于执行状态。多处理机系统则有多个处于执行状态。

  3) 阻塞状态　正在执行的进程由于发生某事件而暂时无法继续执行，放弃处理机而进入的状态，又称等待状态。

  引起阻塞的事件：请求 I/O，申请缓存。

  进程的基本状态转换：（四种转换）



- 挂起状态

  引入挂起状态的原因：

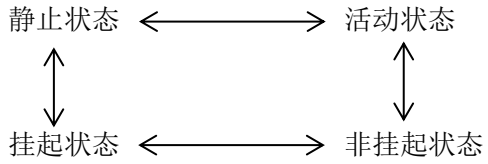
  (1) 终端用户的请求。

  (2) 父进程请求。

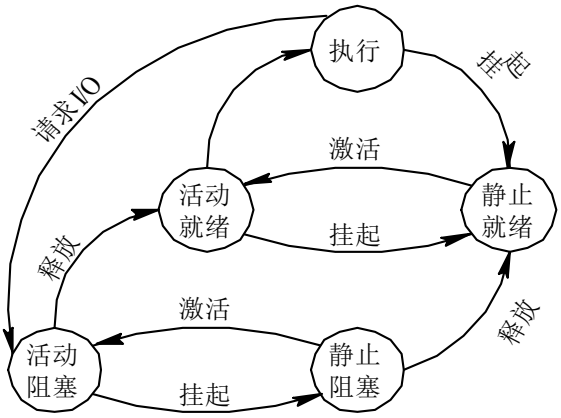
  (3) 负荷调节的需要。　（实时操作系统）

  (4) 操作系统的需要。

挂起引起的状态转换：



有挂起状态的进程状态图



注：活动就绪在内存，静止就绪外存

- 进程控制块

PCB 是 OS 中最重要的**记录型**结构。

OS 是根据 PCB 来对并发执行的进程进行控制和管理的。

PCB 是进程存在的唯一标志。

PCB 常驻内存。

OS 专门开辟 PCB 区将所有 PCB 组织成若干链表或队列。

进程控制块中的信息

1) 进程标识符

(1) 内部标识符 进程唯一的数字编号，给 OS 使用

(2) 外部标识符 由字母、数字组成，给用户使用

2) 处理机状态

处理机中主要的寄存器：

① 通用寄存器 8~32 个，用于暂存信息

② 指令计数器 要访问的下一条指令地址

③ 程序状态字 PSW 条件码、执行方式（程序在系统态/用户态执行）、中断屏蔽标志

④ 用户栈指针 用户进程拥有的系统栈，存放过程和系统调用参数及调用地址

3) 进程调度信息 在多个就绪状态进程中选择进程占领处理机

进程状态

进程优先级 整数 越大，优先级越高

与调度算法有关的信息

事件 如：阻塞原因

4) 进程控制信息

程序和数据地址 程序调用时使用

进程同步和通信机制

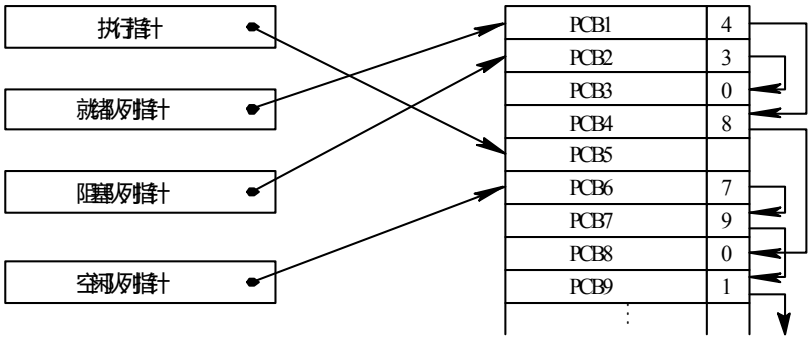
资源清单：除 CPU 之外所需资源与已经分配资源清单 （进程执行过程中动态分配资源）

链接地址：给出了本进程(PCB)所在队列中的下一个进程的 PCB 的首地址

进程控制块的组织方式

1）链接方式

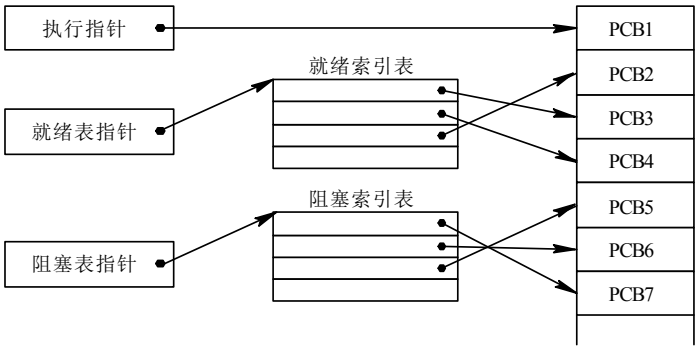
把统一状态的 PCB，用其中的链接字链接成一个队列 如就绪队列、阻塞队列（根据不同阻塞原因）、空白队列



4 指 4 号 PCB 处于队列末尾，用 0 占位

2）索引方式

建立就绪索引表、阻塞索引表等。把索引表在内存的首地址放在内存的的专用单元中。



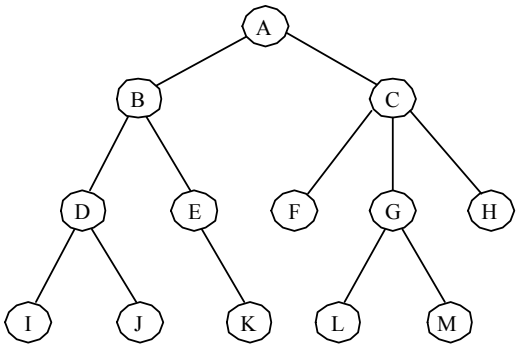
- 进 程 控 制

进程管理中最基本功能是进程控制。

进程控制任务：进程的创建、终止、进程状态的转变等

进程控制一般由 **OS 内核**来实现

进程图 （层次结构 树状 进程的家族关系 自子进程可以继承父进程的资源）



引起创建进程的事件

(1) 用户登录。

(2) 作业调度。 } 由系统内核创建

(3) 提供服务。

(4) 应用请求。→由用户自己创建

进程的创建 (背)

原语 CREAT（）按下列步骤创建一个新进程：

(1)申请空白 PCB。

(2) 为新进程分配资源。

(3) 初始化进程控制块。

(4) 将新进程插入就绪队列，如果进程就绪队列能够接纳新进程， 便将新进程插入就绪队列。

PCB 的初始化

1）初始化标识信息

2）初始化处理机状态信息

3）初始化处理及控制信息

引起进程终止的事件

正常结束 当程序运行到 Holt 指令时，将产生一个中断，去通知 OS 本进程已经完成。

异常结束

① 越界错误。这是指程序所访问的存储区，已越出该进程的区域；

② 保护错。进程试图去访问一个不允许访问的资源或文件，或者以不适当的方式进行访问，例如，进程试图去写一个只读文件；

③ 非法指令。程序试图去执行一条不存在的指令。出现该错误的原因，可能是程序错误地转移到数据区，把数据当成了指令；

④ 特权指令错。用户进程试图去执行一条只允许 OS 执行的指令；

⑤ 运行超时。进程的执行时间超过了指定的最大值；

⑥ 等待超时。进程等待某事件的时间， 超过了规定的最大值；

⑦ 算术运算错。进程试图去执行一个被禁止的运算，例如，被 0 除；

⑧ I/O 故障。这是指在 I/O 过程中发生了错误等。

外界干预

① 操作员或操作系统干预。 由于某种原因，例如，发生了死锁， 由操作员或操作系统终止该进程；

② 父进程请求。 由于父进程具有终止自己的任何子孙进程的权利， 因而当父进程提出请求时，系统将终止该进程；

③ 父进程终止。 当父进程终止时，OS 也将他的所有子孙进程终止。

进程的终止过程

(1) 根据被终止进程的标识符，从 PCB 集合中检索出该进程的 PCB，从中读出该进程的状态。

(2) 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真，用于指示该进程被终止后应重新进行调度。

(3) 若该进程还有子孙进程，还应将其所有子孙进程予以终止，以防他们成为不可控的进程。

(4) 将被终止进程所拥有的全部资源，或者归还给其父进程， 或者归还给系统。

(5) 将被终止进程(它的 PCB)从所在队列(或链表)中移出， 等待其他程序来搜集信息。

- 进程的阻塞和唤醒

引起进程阻塞和唤醒的事件（了解）

1）请求系统服务

2) 启动某种操作

3) 新数据尚未到达

4) 无新工作可做

进程阻塞过程

正在执行的进程，当发现上述某事件时，由于无法继续执行，于是进程便**通过调用阻塞原语 block 把自己阻塞**。可见，进程的阻塞是进程自身的一种主动行为。进入 block 过程后，由于此时该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞，并将 PCB 插入阻塞队列。如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞(等待)队列。 最后，转调度程序进行重新调度，将处理机分配给另一就绪进程，并进行切换，亦即，保留被阻塞进程的处理机状态(在 PCB 中)，再按新进程的 PCB 中的处理机状态设置 CPU 的环境。

入口→保留当前进程的 CPU 现场→置该进程状态→进入等待队列→转进程调度

进程唤醒过程（其他合作进程唤醒）

由唤醒原语 WAKEUP 完成

入口→从等待队列中摘除被唤醒进程→置该进程为就绪状态→进入就绪队列→转进程调度或返回

进程的挂起

挂起原语：suspend( ) 将指定进程或处于阻塞状态的进程挂起

进程原语的执行过程：

首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态的进程，则将之改为静止阻塞。，若正在执行，则转向调度程序重新调度。

- 两种形式的制约关系

(1) 间接相互制约关系。 进程间由于共享某种系统资源，而形成的相互制约。

(2) 直接相互制约关系。 进程间由于合作而形成的相互制约

进程的两大关系

互斥 间接相互制约关系 是并发执行的多个进程由于竞争同一资源而产生的相互排斥的关系

同步 直接相互制约关系 是进程间共同完成一项任务时直接发生相互作用的关系

同步进程间具有合作关系

在执行时间上必须按一定顺序协调进行

临界资源 一次仅允许一个进程使用的共享资源 如打印机、磁带机、表格

临界区 在每个进程中访问临界资源的那段程序 进程必须互斥进入临界区

访问临界资源的循环进程描述如下：

```
repeat
        entry section        →   检查临界资源是否能访问
    critical section;
        exit section
    remainder section;
until false;
```

→ 将临界区标志设为未访问

同步机制应遵循的规则：

(1) 空闲让进。
(2) 忙则等待。
(3) 有限等待。
(4) 让权等待。

- 信号量机制

信号量是一种数据结构
信号量的值表示相应资源的使用情况
信号量的值仅由 P、V 操作来改变

整型信号量

整型数　　用 S 表示 ＜=0 不可用

P 操作（wait）原语

```
wait(S):
    while S≤0 do no-op   //循环检测 S 的状态
          S：=S-1;
```

V 操作（signal）原语

```
signal(S):          S：=S+1;
```

wait(S)和 signal(S)是两个标准的原子操作(Atomic Operation) 。
缺点：只要信号量 S<=0 就不断测试，不满足让权等待。

记录型信号量

记录型结构，包含两个数据项：

```
type semaphore=record
          value:integer;
          L:list of process;
          end
```

相应地，wait(S)和 signal(S)操作可描述为：

```
procedure wait(S)   //申请一个单位的资源
      var S: semaphore;
      begin
        S.value：      =   S.value-1;
        if S.value＜0 then block(S,L)
      end
  procedure signal(S)   //释放一个单位的资源
      var S: semaphore;
      begin
       S.value：      =   S.value+1;
```

if S.value≤0 then wakeup(S,L);
end

在记录型信号量机制中，S.value 的**初值**表示系统中**某类资源的数目**， 因而又称为**资源信号量**，对它的每次 wait 操作，意味着进程请求一个单位的该类资源，因此描述为 S.value：=S.value-1； 当 S.value＜0 时，表示该类资源已分配完毕，因此进程应调用 block 原语，进行自我阻塞，放弃处理机，并插入到信号量链表 S.L 中。可见，该机制遵循了“让权等待”准则。 此时 **S.value 的绝对值表示在该信号量链表中已阻塞进程的数目**。 对信号量的每次 signal 操作，表示执行进程释放一个单位资源，故 S.value： =S.value+1 操作表示资源数目加 1。 若加 1 后仍是 S.value≤0，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用 wakeup 原语，将 S.L 链表中的第一个等待进程唤醒。如果 **S.value 的初值为 1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。**

S.value>=0 是代表系统中可用资源的数目

S.value <0 其绝对值代表等待使用资源的进程的个数

AND 型信号量

在两个进程中都要包含两个对 Dmutex 和 Emutex 的操作， 即

process A: process B:

wait(Dmutex); wait(Emutex);

wait(Emutex); wait(Dmutex);

若进程 A 和 B 按下述次序交替执行 wait 操作：

process A: wait(Dmutex); 于是 Dmutex=0

process B: wait(Emutex); 于是 Emutex=0

process A: wait(Emutex); 于是 Emutex=-1 A 阻塞

process B: wait(Dmutex); 于是 Dmutex=-1 B 阻塞

解决上述产生的死锁状态

基本思想是：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。

在 wait 操作中，增加了一个“AND”条件，故称为 AND 同步，或称为同时 wait 操作，即

Swait(Simultaneous wait)定义如下：

Swait(S1, S2, …, Sn)

if S1≥1 and … and Sn≥1 then

for i： = 1 to n do

Si： = Si-1;

endfor

else

place the process in the waiting queue associated with the first Si found with Si＜1, and set the program count of this process to the beginning of Swait operation

endif

Ssignal(S1, S2, …, Sn)

for i： = 1 to n do

Si=Si+1;

Remove all the process waiting in the queue associated with Si into the ready queue.

endfor;

信号量集

一般“信号量集”的几种特殊情况：

(1) Swait(S, d, d)。 此时在信号量集中只有一个信号量 S， 但允许它每次申请 d 个资源，当现有资源数少于 d 时，不予分配。

(2) Swait(S, 1, 1)。 此时的信号量集已蜕化为一般的记录型信号量(S＞1 时)或互斥信号量(S=1 时)。

(3) Swait(S, 1, 0)。这是一种很特殊且很有用的信号量操作。当 S≥1 时，允许多个进程进入某特定区；当 S 变为 0 后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

- 利用信号量实现进程互斥

利用信号量实现进程互斥的进程可描述如下：

```
Var mutex:semaphore：  =  1;  //互斥的初值是 1
    begin
    parbegin       //部分
     process 1: begin
                  repeat
                      wait(mutex);   //申请资源
                      critical section   //临界区
                      signal(mutex);   //退出区
                      remainder section   //对临界区的使用都这样写 剩余区
                  until false;
  end
      process 2: begin
                    repeat
                        wait(mutex);
                        critical section
                        signal(mutex);
                        remainder section
                   until false;
                   end
    parend
```

在实现互斥时应注意：

wait(mutex)和 signal(mutex)必须成对的出现

缺 wait(mutex)将会引起系统混乱，不能保证对临界资源的**互斥**访问

缺 signal(mutex)将会使该临界资源永久不被释放

- 经典进程的同步问题

生产者—消费者问题

有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有 n 个缓冲区的缓冲池，生产者进程将它所生产的产品放入一个缓冲区中； 消费者进程可从一个缓冲区中取走产品去消费。

尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即（制约关系）不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

例如：在输入时，输入进程是生产者，计算进程是消费者；

而在输出时，则计算进程是生产者，而打印进程是消费者。

利用记录型信号量解决生产者—消费者问题

假定在生产者和消费者之间的公用缓冲池中，具有 n 个缓冲区，这时可利用互斥信号量 mutex 实现诸进程对缓冲池的互斥使用；利用信号量 empty 和 full 分别表示缓冲池中空缓冲区和满缓冲区的数量。又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。对生产者—消费者问题可描述如下：

```
Var mutex, empty, full:semaphore：  =  1,n,0 ;
    buffer:array［0, …, n-1］ of item;  //记录下一个数据放到哪个位置
    in, out: integer：  =  0, 0;  //放入数据的地址、取出数据的地址
    begin
     parbegin
      proceducer:begin
            repeat
              …
            producer an item nextp;  //循环生产数据
              …
            wait(empty);
            wait(mutex);
            buffer(in)：  =  nextp;
            in：  =  (in+1) mod n;  //l 临界区
            signal(mutex);  //释放缓冲池
            signal(full);
            until false;
          end

      consumer:begin
            repeat
              wait(full);
              wait(mutex);
              nextc：  =  buffer(out);
              out：  =  (out+1) mod n;
              signal(mutex);
              signal(empty);
              consumer the item in nextc;
            until false;
          end
    parend
  end
```

注意：

首先，在每个程序中用于实现互斥的 wait(mutex)和 signal(mutex)必须成对地出现； 其次，对资源信号量 empty 和 full 的 wait 和 signal 操作，同样需要成对地出现，但它们分别处于**不同的程序中**。例如，wait(empty)在计算进程中，而 signal(empty)则在打印进程中，计算进程若因执行 wait(empty)而阻塞， 则以后将由打印进程将它唤醒；最后，在每个程序中的多个 wait 操作顺序不能颠倒。应先执行对资源信号量的 wait 操作，然后再执行对互斥信号量的 wait 操作，否则可能引起进程死锁。

哲学家进餐问题

分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

Var chopstick: array［0, …, 4］ of semaphore;

所有信号量均被初始化为 1， （记录型信号量）第 i 位哲学家的活动可描述为：

```
repeat
  wait(chopstick［i］);
  wait(chopstick［(i+1) mod 5］);   //筷子从 0-4
      …
eat;
                              …
signal(chopstick［i］);
signal(chopstick［(i+1) mod 5］);
                          …
think;
until false;
```

假如五个哲学家同时饥饿而各自拿起左边的筷子，会使五个信号量均为 0，当他们再拿起右边筷子时，都将无限期等待。

可采取以下几种解决方法：

(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。按此规定，将是 1、 2 号哲学家竞争 1 号筷子；3、4 号哲学家竞争 3 号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

利用 AND 信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的 AND 同步问题，故用 AND 信号量机制可获得最简洁的解法。

```
Var chopsiick array ［0, …, 4］ of semaphore：      =  (1,1,1,1,1);
    processi
        repeat
        think;
        Sswait(chopstick［(i+1) mod 5］, chopstick ［i］);
        eat;
```

```
        Ssignat(chopstick ［(i+1) mod 5］, chopstick ［i］);
      until false;
```

读者-写者问题

一个数据文件或记录可被多个进程共享。其中有些进程要求读，而另一些要求些或更改只要求读的进程称为“Reader 进程”，其他进程称为“Writer 进程”

允许多个 Reader 进程同时读一个共享对象，不允许一个 Writer 进程和其他 Reader 进程或 Writer 进程同时访问共享对象。

利用记录型信号量解决读者-写者问题

为实现 Reader 与 Writer 进程间在读或写时的互斥而设置了一个互斥信号量 Wmutex。另外，再设置一个整型变量 Readcount 表示正在读的进程数目。由于只要有一个 Reader 进程在读，便不允许 Writer 进程去写。因此，仅当 Readcount=0，表示尚无 Reader 进程在读时，Reader 进程才需要执行 Wait(Wmutex)操作。若 wait(Wmutex)操作成功，Reader 进程便可去读，相应地，做 Readcount+1 操作。同理，仅当 Reader 进程在执行了 Readcount 减 1 操作后其值为 0 时，才须执行 signal(Wmutex)操作，以便让 Writer 进程写。又因为 Readcount 是一个可被多个 Reader 进程访问的临界资源，因此，应该为它设置一个互斥信号量 rmutex。

读者-写者问题可描述如下：

```
Var rmutex, wmutex:semaphore：        =  1,1;
    Readcount:integer：        =  0;
    begin
    parbegin
     Reader:begin
         repeat
          wait(rmutex);
          if readcount=0 then wait(wmutex);
           Readcount：        =  Readcount+1;
          signal(rmutex);
            …
          perform read operation;
            …
  wait(rmutex);
          readcount：        =  readcount-1;
          if readcount=0 then signal(wmutex);
          signal(rmutex);
         until false;
       end
    writer:begin
         repeat
          wait(wmutex);
          perform write operation;
          signal(wmutex);
         until false;
```

```
        end
    parend
end
```

- 进程通信

进城通信是指进程之间的信息交换。

交换的信息量 一个状态或数值

上千个字节

进程通信的类型

1）低级通信：进程的互斥和同步

2）高级通信：指用户可直接利用 OS 提供的一组**通信命令**，高效的传送大量数据的一种通信方式。对用户透明。

高级通信的分类：共享存储器系统、消息传递系统、管道通信

共享存储器系统

(1) 基于共享数据结构的通信方式。进程之间通过某种数据结构，如缓冲池进行通信属于低级通信方式

(2) 基于共享存储区的通信方式。 为了传送大量的数据，在存储器中划出一块共享存储区，进程可通过对共享存储区进行读或写来实现通信，属于高级通信方式。

消息传递系统

信息交换的单位是消息或报文，分两种：

1. 直接通信方式：发送进程直接把消息发给目标进程
2. 间接通信方式：进程之间的通信需要通过某种中间实体，该进程用来暂存发送进程发送给目标进程的消息；接收进程中则从该实体中取出对方发送给自己的消息。

这种中间实体称为信箱。

消息在信箱中可以安全保存，只允许核准目标用户随时读取，故可实现非实时通讯。

计算机网络中将消息称为报文。

直接通信方式

发送进程和接收进程都以显式方式分别提供对方的标识符。

系统提供两条通信原语：

Send(Receiver,message);

Receive(Sender,message);

例如，原语 Send(P2, m1)表示将消息 m1 发送给接收进程 P2; 而原语 Receive(P1，m1)则表示接收由 P1 发来的消息 m1。

解决生产者-消费者问题（直接通信方式）

```
repeat
    ...
  produce an item in nextp;
    ...
  send(consumer, nextp);
until false;
repeat
```

```
    receive(producer, nextc);
      ...
    consume the item in nextc;
  until false;
```

间接通信方式

信箱的创建和撤消　进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享)；对于共享信箱，　还应给出共享者的名字。当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

消息的发送和接收　当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信。

Send(mailbox, message); 将一个消息发送到指定信箱；

Receive(mailbox, message); 从指定信箱中接收一个消息；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。

1) 私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用**单向通信链路**的信箱来实现。　当拥有该信箱的**进程结束时，信箱也随之消失**。

2）公用信箱

它由**操作系统**创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用**双向通信链路**的信箱来实现。通常，公用信箱在系统运行期间**始终存在**。

3) 共享信箱

它由**某进程**创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。

在利用信箱通信时，在发送进程和接收进程之间，存在以下四种关系：

(1) **一对一关系**。这时可为发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。

(2) **多对一关系**。允许提供服务的进程与多个用户进程之间进行交互，也称为客户/服务器交互(client/server interaction)。

(3) **一对多关系**。允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式，向接收者(多个)发送消息。

(4) **多对多关系**。允许建立一个公用信箱，让多个进程都能向信箱中投递消息；也可从信箱中取走属于自己的消息。

管道(Pipe)通信（建立在文件系统上）

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名 pipe 文件。向管道(共享文件)提供输入的发送进程(即写进程)，　以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于 UNIX 系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：① 互斥，即当

一个进程正在对 pipe 执行读/写操作时，其它(另一)进程必须等待。② 同步，指当写(输入)进程把一定数量(如 4 KB)的数据写入 pipe，便去睡眠等待， 直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。③ 确定对方是否存在，只有确定了对方已存在时，才能进行通信。

- 线　程（用于多 CPU 系统和网络操作系统）

  进程：使多个程序能并发执行，以提高资源利用率和系统吞吐量。

  引入线程：是为了减少程序在并发执行时所付出的时空开销，使 OS 具有更好的并发性

  引入线程的目的：进程是**可拥有资源的独立单位**和可**独立调度和分派的基本单位**。

  创建、撤销和切换中系统必须为之付出较大的时空开销。故进程，其数目不宜过多进程切换的频率也不宜过高

  进程不应同时作为拥有资源的单位和可独立调度和分派的基本单位，应该轻装上阵。

  线程的属性

  (1) 轻型实体。

  (2) 独立调度和分派的基本单位。

  (3) 可并发执行。

  (4) 共享进程资源。

三、处理机调度与死锁

- 一个批处理型作业从进入系统并驻留在外存的后备队列开始，直至作业运行完毕，可能要经历的三级调度：

  高级调度(High Scheduling)，又称作业调度、长程调度、接纳调度。

  作用：把外存上处于后备队列中的作业调入内存，并为它们创建进程、分配资源、排在就绪队列上，准备执行。

  分时系统、实时系统，通常不需要作业调度。

  低级调度(Low Level Scheduling) ，也称为进程调度、短程调度

  作用：决定就绪队列中的哪个进程应获得处理机，然后由分派程序把处理及分配给该进程的具体操作。

  在 OS 中都必须配置。

  1) 非抢占方式(Non-preemptive Mode)　一旦把处理机分配给某进程后，便让该进程一直执行，直至该**进程完成或阻塞**时，才在把处理机分配给其他进程。

  在采用非抢占调度方式时，可能引起进程调度的因素可归结为这样几个：

  ① 正在执行的进程执行完毕， 或因发生某事件而不能再继续执行；

  ② 执行中的进程因提出 I/O 请求而暂停执行；

  ③ 在进程通信或同步过程中执行了某种原语操作，如 P 操作(wait 操作)、Block 原语、Wakeup 原语等。

  这种调度方式的<u>优点</u>是实现简单、系统开销小，适用于大多数的批处理系统环境。但它难以满足紧急任务的要求——立即执行，因而可能造成难以预料的后果。显然，在要求比较严格的实时系统中，不宜采用这种调度方式。

  2) 抢占方式(Preemptive Mode) 允许暂停正在执行的进程，将处理机分派给另一进程。

  (1) 优先权原则。

(2) 短作业(进程)优先原则。
(3) 时间片原则。

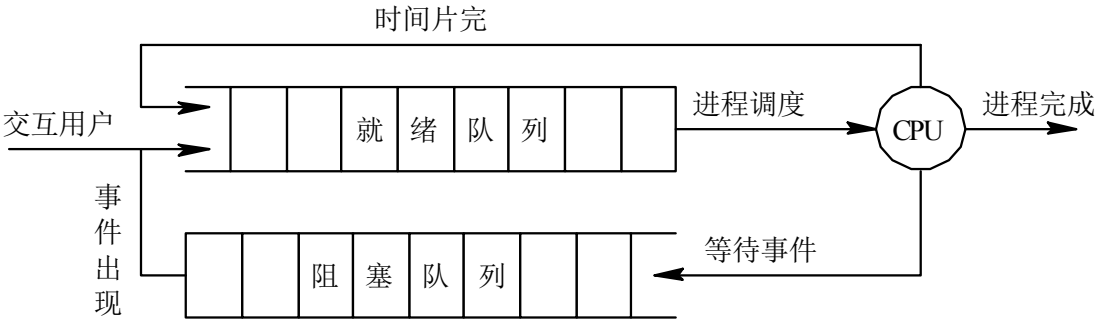
中级调度(Intermediate-Level Scheduling)
又称中程调度(Medium-Term Scheduling)
目的：是为了提高内存利用率和系统吞吐量。
作用：使那些暂时不能运行的进程不再占用宝贵的内存资源，而将它们调至外存上去等待，把此时的进程状态称为就绪驻外存状态或挂起状态。当这些进程重又具备运行条件、且内存又稍有空闲时，由中级调度来决定把外存上的哪些又具备运行条件的就绪进程，重新调入内存，并修改其状态为就绪状态，挂在就绪队列上等待进程调度。
又称对换。

- 调度队列模型

1. 仅有进程调度的调度队列模型

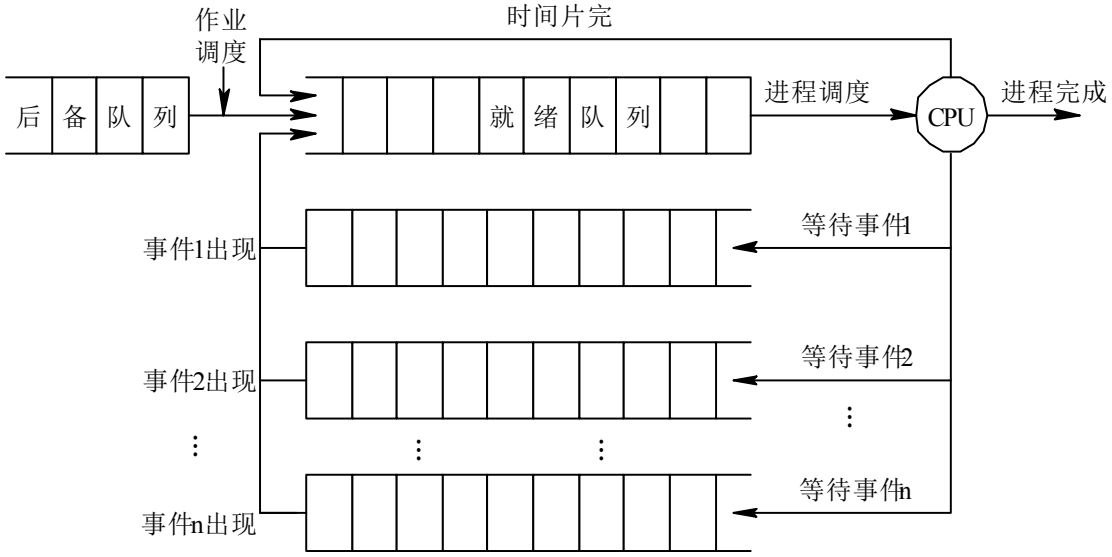
通常，把就绪进程组织成 FIFO 队列，每当创建新进程时排在就绪队列的末尾，按时间片轮转方式运行。

进程在执行时，出现三种情况：
1）任务在时间片内完成，进程便在释放处理机后进入完成状态。
2）任务在时间片内未完成，OS 便将该任务再放入就绪队列的末尾。
3）在执行期间，进程因为某事件而被阻塞后，将 OS 放入阻塞队列。



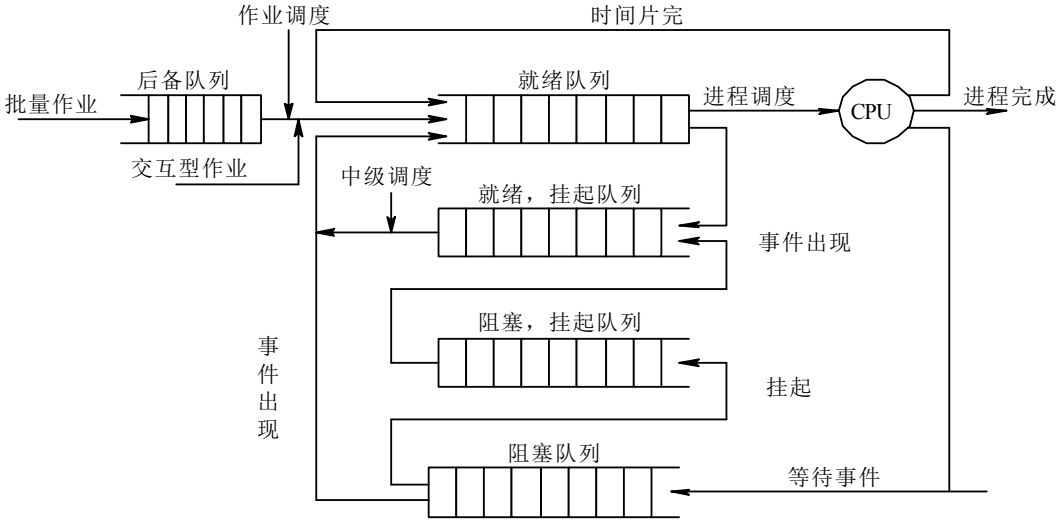
2. 具有高级和低级调度的调度队列模型

与前一模型的差别：
（1）就绪队列的形式。批处理系统中最常用的是优先权队列。也可采用无序链表方式。
（2）设置多个阻塞队列。



3. 同时具有三级调度的调度队列模型

调出时，可使进程状态由内存就绪转变为外存就绪，由内存阻塞转变为外存阻塞；在中级调度室外存就绪转变为内存就绪。



- 选择调度方式和调度算法的若干准则

  1. 面向用户的准则

  (1) **周转时间**短。

  (2) **响应时间**快。

  (3) 截止时间的保证。

  (4) 优先权准则。

  2. 面向系统的准则

  (1) 系统吞吐量高。

  (2) 处理机利用率好。

(3) 各类资源的平衡利用。

周转时间：从作业被提交给系统开始，到作业完成为止的这段时间间隔称为作业周转时间。包括四部分时间：

在外存后备队列上等待调度的时间

进程在就绪队列上等待调度的时间

进程在 CPU 上执行的时间

进程等待 I/O 才做完成的时间

平均周转时间：

$$T=\frac{1}{n}\left[\sum_{i=1}^{n}T_i\right]$$

带权周转时间：$W=T/T_S$

T：作业的周期时间

T_s：系统为它提供服务的时间（真正运行时间）。

平均带权周转时间：

$$W=\frac{1}{n}\left[\sum_{i=1}^{n}\frac{T_i}{T_{Si}}\right]$$

例：有如下三道作业。系统为它们服务的顺序是：1/2/3.求平均周转时间和带权周转时间。

作业	提交时间/时	运行时间/h
1	10.00	2
2	10.10	1
3	10.25	0.25

解：

作业	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
1	10.00	2	10.00	12.00	2	2/2
2	10.10	1	12.00	13.00	2.90	2.9/1
3	10.25	0.25	13.00	13.25	3	3/0.25

平均周转时间：T=(2+2.9+3)/3=2.63h

平均带权周转时间：W=(2+2.9+12)/3=5.3h

响应时间：是从用户通过键盘提交一个请求开始直至系统首次产生响应为止的时间间隔。它包括三部分时间：

从键盘输入的请求信息传送到处理机的时间

处理机对请求信息进行处理的时间

将响应信息回送到终端显示器的时间

是**分时系统的**重要原则

截止时间是指某任务必须开始执行的最迟时间，或必须完成的最迟时间。

对于严格的**实时系统**，其调度方式和调度算法必须能保证这一点。

吞吐量指单位时间内系统所完成的作业数。

评价**批处理**系统性能的重要指标。

与作业的**平均长度**有关。对于大型作业，一般吞吐量约为每小时一道作业，对于中小型作业，其吞吐量可达到数十道作业 。

- 调度算法：根据系统的资源分配策略所规定的资源分配算法。

不同的系统和系统目标，通常采用不同的调度算法

1. 先来先服务调度算法

作业调度中每次从后备作业队列中，选择一个或多个最先进入该队列的作业调入内存，为它们分配资源、创建进程，然后放入就绪队列。

进程调度时每次从就绪队列中，选择一个最先进入该队列的进程分配处理机使之运行。直到完成或阻塞后，才放弃处理机。

是一种最简单的调度算法，既可用于作业调度也可用于进程调度。

FCFS（first come first serve）算法

有利于长作业（进程），而不利于短作业（进程）。

有利于 CPU 繁忙型作业，而不利于 I/O 繁忙型作业。（用带权周转时间评价）

进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99

CPU 繁忙型，带权周转时间接近于 1，I/O 繁忙型作业带权周转时间远远大于 1

2. 短作业(进程)优先调度算法

短作业(进程)优先调度算法 SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。

短作业优先(SJF)的调度算法，是从**后备队列**中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。

而短进程优先(SPF)调度算法，则是从**就绪队列**中选出一估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时，再重新调度。

调度算法 \ 作业情况	进程名	A	B	C	D	E	平　均
	到达时间	0	1	2	3	4	
	服务时间	4	3	5	2	4	
FCFS (a)	完成时间	4	7	12	14	18	
	周转时间	4	6	10	11	14	9
	带权周转时间	1	2	2	5.5	3.5	2.8
SJF (b)	完成时间	4	9	18	6	13	
	周转时间	4	8	16	3	9	8
	带权周转时间	1	2.67	3.1	1.5	2.25	2.1

SJ(P)F 调度算法也存在不容忽视的缺点：

(1) 该算法对长作业不利，如作业 C 的周转时间由 10 增至 16，其带权周转时间由 2 增至 3.1。更严重的是，如果有一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序

总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度。

(2) 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业(进程)会被及时处理。

(3) 由于作业(进程)的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

3. 高优先权优先调度算法

1. 优先权调度算法的类型

1）非抢占式优先权算法

在这种方式下，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，直至完成； 或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于**批处理系统**中；也可用于某些**对实时性要求不严的实时系统**中。

2）抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程 i 时，就将其优先权 Pi 与正在执行的进程 j 的优先权 Pj 进行比较。如果 Pi≤Pj，原进程 Pj 便继续执行；但如果是 Pi＞Pj, 则立即停止 Pj 的执行，做进程切换，使 i 进程投入执行。显然，这种抢占式的优先权调度算法，能更好地满足紧迫作业的要求，故而常用于**要求比较严格的实时系统**中， 以及**对性能要求较高的批处理和分时系统**中。

2. 优先权的类型

1) 静态优先权 在创建进程时确定的，且在进程的整个运行期间保持不变。一般地，优先权是利用某一范围内的一个整数来表示的，例如，0~7 或 0~255 中的某一整数， 又把该整数称为优先数。只是具体用法各异：有的系统用“0”表示最高优先权，当数值愈大时，其优先权愈低；而有的系统恰恰相反。

优缺点：简单、开销小；不精确，仅在要求不高的系统中使用。

确定进程优先权的依据有如下三个方面：

(1) 进程类型。

(2) 进程对资源的需求。

(3) 用户要求。

2) 动态优先权 在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。

3. 高响应比优先调度算法

引入动态优先权，并使作业优先级随着等待时间的增加而以速率 a 提高。

优先权的变化规律可描述为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$$

由于等待时间与服务时间之和，就是系统对该作业的响应时间，故该优先权又相当于**响应比 R_P**。据此，又可表示为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$$

分析：

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高， 从而也可获得处理机。

优点：兼顾长短作业

缺点：由于做相应比计算，增加了系统开销。

4.时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则，排成一个队列，每次调度时，把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程，在一给定的时间内，均能获得一时间片的处理机执行时间。（分时系统）

5. 多级反馈队列调度算法

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。 第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，**在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小**。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第 i+1 个队列的时间片要比第 i 个队列的时间片长一倍。

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列中便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行； 仅当第 1~(i-1) 队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 1~(i-1)中的任何一个队列)，则此时新进程将**抢占**正在运行进程的处理机，即由调度程序把正在运行的进程放回到第 i 队列的末尾，把处理机分配给新到的高优先权进程。

多级反馈队列调度算法的性能

(1) 终端型作业用户。 作业较小，只要使作业（进程）在第一队列所规定的时间片内完成，便可令用户满意。

(2) 短批处理作业用户。其周转时间短。

(3) 长批处理作业用户。不必担心作业长期得不到处理。

- 实时调度的基本条件

  1. 提供必要的信息

  (1) 就绪时间。

  (2) 开始截止时间和完成截止时间。

(3) 处理时间。

(4) 资源要求。

(5) 优先级。

2. 系统处理能力强

在实时系统中，通常都有着多个实时任务。若处理机的处理能力不够强，则有可能因处理机忙不过来而使某些实时任务不能得到及时处理， 从而导致发生难以预料的后果。假定系统中有 m 个周期性的硬实时任务，它们的处理时间可表示为 Ci，周期时间表示为 Pi，则在单处理机情况下，必须满足下面的限制条件：

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

系统才是可调度的。假如系统中有 6 个硬实时任务，它们的周期时间都是 50 ms，而每次的处理时间为 10 ms，则不难算出，此时是不能满足上式的，因而系统是不可调度的。

解决的方法是提高系统的处理能力，其途径有二：其一仍是采用单处理机系统，但须增强其处理能力， 以显著地减少对每一个任务的处理时间；其二是采用多处理机系统。假定系统中的处理机数为 N，则应将上述的限制条件改为：

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq N$$

3. 采用抢占式调度机制

4. 具有快速切换机制

该机制应具有如下两方面的能力：

(1) 对外部中断的快速响应能力。为使在紧迫的外部事件请求中断时系统能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短， 以免耽误时机(其它紧迫任务)。

(2) 快速的任务分派能力。在完成任务调度后，便应进行任务切换。为了提高分派程序进行任务切换时的速度， 应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销。

- 实时调度算法的分类

按任务性质：硬实时调度算法、软实时调度算法

按调度方式：非抢占式调度算法、抢占式调度算法

按调度时间：静态调度算法、动态调度算法

1. 非抢占式调度算法 算法简单、用于小型实时系统或要求不严的实时控制系统。

(1) **非抢占式轮转调度算法。** 用于工业群控系统，一台计算机控制若干个相同的（或类似的）对象，为每个被控对象建立一个实时任务，并将它们拍成一个轮转队列。调度程序每次选择队列中的第一个任务运行。完成后，便把他们挂在轮转队列的末尾，等待下一次调度运行，这次调度程序在选择下一个（队首）任务执行。可获得数秒至数十秒响应时间。

(2) **非抢占式优先调度算法。**针对有一定要求的系统，为实时性要求高的任务赋予较高的优先级。优先安排在就绪队列队首，待当前任务结束后，被调度执行。响应时间为数秒或数百毫秒

2. 抢占式调度算法 应用于响应时间在数十毫秒以下的系统

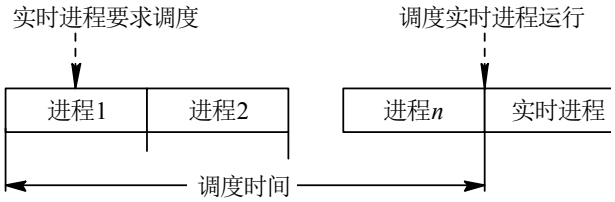
根据抢占发生时间不同分类：

(1) 基于时钟中断的抢占式优先权调度算法

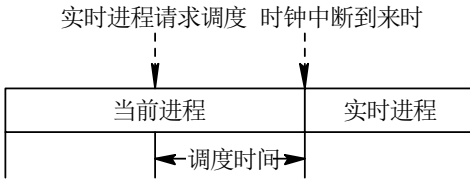
高优先级的实时任务到达后不立即抢占，等到时钟中断到来时再重新分配处理机。

(2) 立即抢占的优先权调度算法。

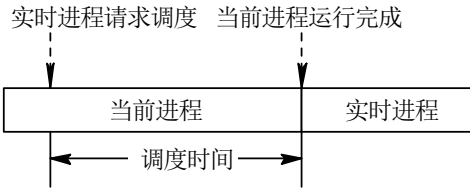
高优先级的实时任务到达后，只要当前任务未处于临界区就立即把处理机分配给他。



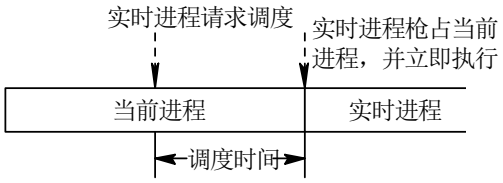
(*a*) 非抢占轮转调度



(*c*) 基于时钟中断抢占的优先权抢占调度



(*b*) 非抢占优先权调度



(*d*) 立即抢占的优先权调度

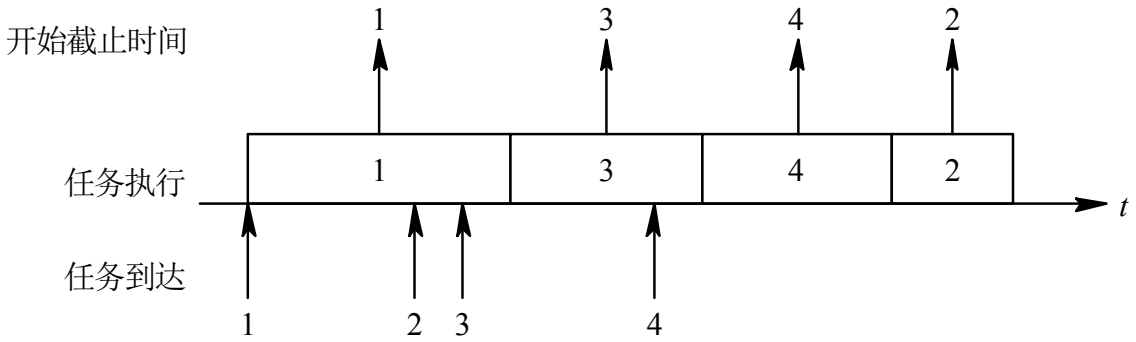
常用的几种实时调度算法

最早截止时间优先即 EDF(Earliest Deadline First)算法

根据任务的**开始截止时间**确定优先级，截止时间越早优先级越高

系统中保持一个实时任务**优先级就绪队列**，调度程序选择**对首任务**分配处理机

可采取抢占式和非抢占式调度



最低松弛度优先即 LLF(Least Laxity First)算法

根据任务紧急(或松弛)的程度，来确定任务的优先级。任务的紧急程度愈高，为该任务所赋予的优先级就愈高， 以使之优先执行。

系统中保持一个实时任务**优先级就绪队列**，调度程序选择**对首任务**分配处理机

可采取抢占式和非抢占式调度

- 多处理机系统（MPS）中的调度

提高系统性能的主要途径有两条：

1）提高元器件的运行速度，特别是处理机芯片的速度

2）改进计算机系统的体系结构，特别是在系统中引入多个处理器或多台计算机

功能较强的主机系统和服务器都采用多处理器系统

处理器数目可为两个至数千。

多处理器系统的类型

1. 从多处理机之间耦合的紧密程度上，可分为

(1) 紧密耦合(Tightly Coupted)MPS。

通过**高速总线或高速交叉开关**，来实现多个**处理器**之间的互连

地址总线、控制总线（单向）、数据总线

它们共享主存储器系统和 I/O 设备，并要求将主存储器划分为若干个能独立访问的存储器模块，以便多个处理机能同时对主存进行访问。系统中的所有资源和进程，都由操作系统实施统一的控制和管理。

(2) 松散（弛）耦合(Loosely Coupled)MPS。

通过**通道**或**通信线路**，来实现多台计算机之间的互连。每台计算机都有自己的存储器和 I/O 设备，并配置了 OS 来管理**本地资源和在本地运行的进程**。因此，每一台计算机都能**独立地工作**， 必要时可通过通信线路与其它计算机交换信息，以及协调它们之间的工作。

2. 根据系统中处理器的相同与否可分为：

对称 MPS 对称多处理器系统 SMPS(Symmetric MultiProcessor System)。在系统中所包含的各处理器单元，在功能和结构上都是相同的， 当前的绝大多数 MPS 都属于 SMP 系统。例如，IBM 公司的 SR/6000 Model F50, 便是利用 4 片 Power PC 处理器构成的

非对称 MPS 非对称多处理器系统。在系统中有多种类型的处理单元， 它们的功能和结构各不相同，其中只有一个主处理器，有多个从处理器。

- 进程分配方式（针对多处理机系统）

多处理器系统中进程的调度与系统结构有关。

同构系统中进程可以分配到任一处理器上。

非对称系统中进程只能分配到某一合适运行的处理器上。

对称多处理器系统中的进程分配方式

在 SMP 系统中，所有的处理器都是相同的，因而可把所有的处理器作为一个处理器池(Processor pool)，由调度程序或基于处理器的请求， 将任何一个进程分配给池中的任何一个处理器去处理。在进行进程分配时，可采用以下两种方式之一。

1) 静态分配(Static Assigenment)方式

特点：进程被固定分配到一个处理器上；与单机进程调度方式相同。

优点：开销小

缺点：各处理机忙闲不均匀

2) 动态分配(Dynamic Assgement)方式

设置一个公共就绪队列，进程可被分配到任一处理器上

优点：消除了忙闲不均的现象

非对称 MPS 中的进程分配方式

对于非对称 MPS， 其 OS 大多采用主—从(Master-Slave)式 OS， 即 OS 的核心部分

驻留在一台主机上(Master)， 而从机(Slave)上只是用户程序， 进程调度只由主机执行。每当从机空闲时， 便向主机发送一索求进程的信号， 然后， 便等待主机为它分配进程。 在主机中保持有一个就绪队列， 只要就绪队列不空， 主机便从其队首摘下一进程分配给请求的从机。从机接收到分配的进程后便运行该进程， 该进程结束后从机又向主机发出请求。

优点：系统处理简单，因为进程分配由主机独自处理，使进程间的同步问题得以简化。

- 进程(线程)调度方式

1. 自调度(Self-Scheduling)方式

1) 自调度机制

在多处理器系统中，自调度方式是最简单的一种调度方式。它是直接由单处理机环境下的调度方式演变而来的。 **在系统中设置有一个公共的进程或线程就绪队列， 所有的处理器在空闲时，都可自己到该队列中取得一进程(或线程)来运行。**在自调度方式中，可采用在单处理机环境下所用的调度算法，如先来先服务(FCFS)调度算法、最高优先权优先(FPF)调度算法和抢占式最高优先权优先调度算法等。

2) 自调度方式的优点

自调度方式的主要优点表现为：首先，系统中的公共就绪队列可按照单处理机系统中所采用的各种方式加以组织； 其调度算法也可沿用单处理机系统所用的算法，亦即，很**容易将单处理机环境下的调度机制移植到多处理机系统中**， 故它仍然是当前多处理机系统中较常用的调度方式。其次，**处理机的利用率高，只要系统中有任务，或者说只要公共就绪队列不空，就不会出现处理机空闲的情况，也不会发生处理器忙闲不均的现象**，因而有利于提高处理器的利用率。

2) 自调度方式的缺点

(1) 瓶颈问题。

(2) 低效性。

(3) 线程切换频繁。

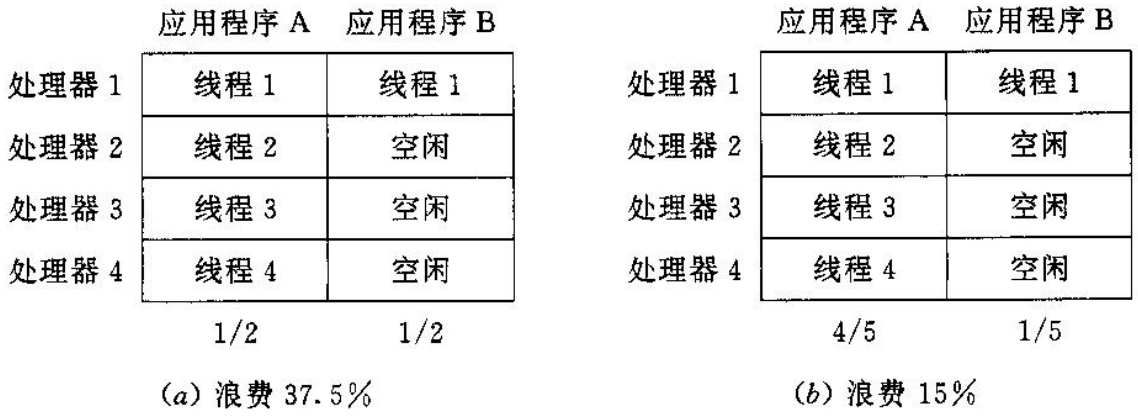
2. 成组调度(Gang Scheduling)方式

是指将一个进程中的一组线程，分配一组处理器去执行。

在成组调度时，如何为应用程序分配处理器时间，

1) 面向所有应用程序平均分配处理器时间

2) 面向所有线程平均分配处理器时间



优点：减少线程切换，优于自调度

3. 专用处理器分配(Dedicated Processor Assigement)方式

指在一个应用程序执行期间，专门为该应用程序分配一组处理器，每一个线程一个处理器，供应用程序专用直至该应用程序完成。

缺点：造成单个处理机的浪费

引入原因：多处理机系统中单个处理机的利用率不很重要

在一个应用程序的运行中完全避免了进程或线程的切换，大大加速运行。

- **死锁**

多个进程在运行过程中因竞争资源而造成的一种僵局。

各并发进程彼此等待对方拥有的资源，且在得到对方资源前不释放自己的资源。

产生死锁的原因

(1) 竞争资源。

(2) 进程间推进顺序非法。

1. 竞争资源引起进程死锁

1) 可剥夺和非剥夺性资源

2) 竞争非剥夺性资源

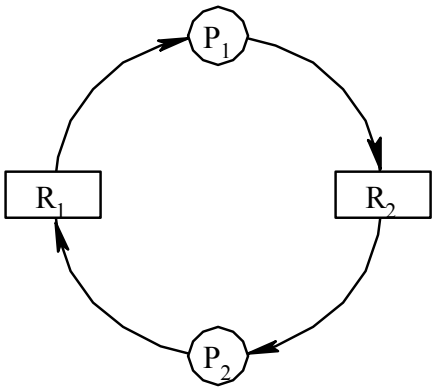
3) 非剥夺资源的数量不能满足进程运行的需要，会使进程在运行过程中，因争夺这些资源而陷入僵局。

3) 竞争临时性资源

资源分成两类：

可剥夺性资源：指进程在获得这类资源后，该资源可以被其它进程或系统剥夺。如：CPU、主存

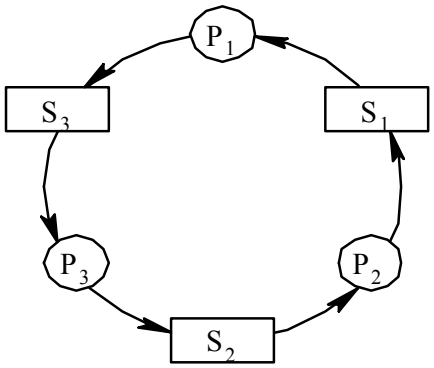
不可剥夺资源：当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放，如磁带机、打印机。



I/O 设备共享时的死锁情况

竞争临时性资源

临时性资源：由一个进程产生由另一个进程使用暂短时间后便无用的资源。



进程之间通信时的死锁

产生死锁的必要条件：

（1）互斥条件

（2）请求和保持条件

（3）不可剥夺条件

（4）环路等待条件

指在发生死锁时，必然存在一个进程——资源环形链，即进程集合{P0、P1、P2.....Pn}中的 P0 正在等待 P1 占有的资源；P1 正在等待一个 P2 占有的资源，……，Pn 正在等待已被 P0 占用的资源。

处理死锁的基本方法

(1) 预防死锁。用严格的条件限制

设置某些限制条件，破坏四个条件中的一个或几个条件。 简单、较易实现

(2) 避免死锁。 不严格

事先防范策略

在资源的动态分配过程中，用某种方法去防止系统进入不安全状态

可获得较高的资源利用率及系统吞吐量

实现上有一定的难度。

(3) 检测死锁。

(4) 解除死锁。 撤销或挂起一些进程

预防死锁

1. 摒弃“请求和保持”条件 开始运行前一次性申请全部所需资源

优点：简单、易实现、安全。

缺点：资源被严重浪费，恶化了系统利用率；是进程延迟运行

2. 摒弃“不剥夺”条件 再申请时放弃已有，等需要再申。

实现复杂，代价大

可能反复申请和释放，而使进程的执行无限的延迟、延长了近程的周转时间增加了系统开销、降低系统吞吐量。

3. 摒弃“环路等待”条件 系统将所有资源按类型排队，并赋予不同的序号。所有进程请求资源严格按照资源序号递增的次序提出，防止出现环路。

缺点：

（1）序号必须相对稳定，限制了新设备类型的增加。

（2）作业（进程）使用资源顺序与系统规定的顺序不同而造成资源的浪费

（3）限制了用户编程。

- 系统安全状态

指系统能按某种进程顺序(P1, P2, …， Pn)(称〈P1, P2, …, Pn〉序列为安全序列)，来为每个进程 Pi 分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。如果系统无法找到这样一个安全序列，则称系统处于不安全状态。

系统进入不安全状态后可能进入死锁

只要系统出于安全状态，系统便可避免进入死锁状态。

- 利用银行家算法避免死锁

银行家算法中的数据结构

(1) 可利用资源向量 Available。这是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 Available［j］=K，则表示系统中现有 Rj 类资源 K 个。

(2) 最大需求矩阵 Max。这是一个 n×m 的矩阵，它定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。如果 Max［i,j］=K，则表示进程 i 需要 Rj 类资源的最大数目为 K 。

(3) 分配矩阵 Allocation。这也是一个 n×m 的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 Allocation［i,j］=K，则表示进程 i 当前已分得 Rj 类资源的数目为 K。

(4) 需求矩阵 Need。这也是一个 n×m 的矩阵，用以表示每一个进程尚需的各类资源数。如果 Need［i,j］=K，则表示进程 i 还需要 Rj 类资源 K 个，方能完成其任务。

Need［i,j］=Max［i,j］-Allocation［i,j］

银行家算法

设 Requesti 是进程 Pi 的请求向量，如果 Requesti［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 Requesti［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 Requesti［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］-Requesti［j］;

Allocation［i,j］：=Allocation［i,j］+Requesti［j］;

Need［i,j］：=Need［i,j］-Requesti［j］;

(3) 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。

3.安全性算法

(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。

假定系统中有五个进程｛P0, P1, P2, P3, P4｝和三类资源｛A, B, C｝，各种资源的数量分别为 10、5、7，在 T0 时刻的资源分配情况如图 3-15 所示。

进程 \ 资源情况	Max A B C	Allocation A B C	Need A B C	Available A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2
				(2 3 0)
P_1	3 2 2	2 0 0	1 2 2	
		(3 0 2)	(0 2 0)	
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	

(1) T0 时刻的安全性：

进程 \ 资源情况	Work A B C	Need A B C	Allocation A B C	Work＋Allocation A B C	Finish
P_1	3 3 2	1 2 2	2 0 0	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_2	7 4 5	6 0 0	3 0 2	10 4 7	true
P_0	10 4 7	7 4 3	0 1 0	10 5 7	true

(2) P1 请求资源：P1 发出请求向量 Request1(1，0，2)，系统按银行家算法进行检查：

① Request1(1, 0, 2)≤Need1(1, 2, 2)

② Request1(1, 0, 2)≤Available1(3, 3, 2)

③ 系统先假定可为 P1 分配资源，并修改 Available, Allocation1 和 Need1 向量，由此形成的资源变化情况如图 3-15 中的圆括号所示。

④ 再利用安全性算法检查此时系统是否安全。

进程 \ 资源情况	Work A B C	Need A B C	Allocation A B C	Work＋Allocation A B C	Finish
P_1	2 3 0	0 2 0	3 0 2	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_0	7 4 5	7 4 3	0 1 0	7 5 5	true
P_2	7 5 5	6 0 0	3 0 2	10 5 7	true

(3) P4 请求资源：P4 发出请求向量 Request4(3，3，0)，系统按银行家算法进行检查：

① Request4(3, 3, 0)≤Need4(4, 3, 1);

② Request4(3, 3, 0) > Available(2, 3, 0)，让 P4 等待。

(4) P0 请求资源：P0 发出请求向量 Requst0(0，2，0)，系统按银行家算法进行检查：

① Request0(0, 2, 0)≤Need0(7, 4, 3);

② Request0(0, 2, 0)≤Available(2, 3, 0);

③ 系统暂时先假定可为 P0 分配资源，并修改有关数据，如图 3-18 所示。

进程 \ 资源情况	Allocation A B C	Need A B C	Available A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- 死锁的检测与解除

死锁的检测

1. 资源分配图(Resource Allocation Graph)

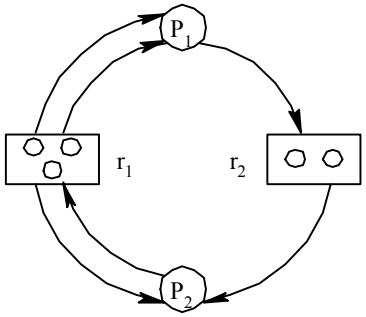
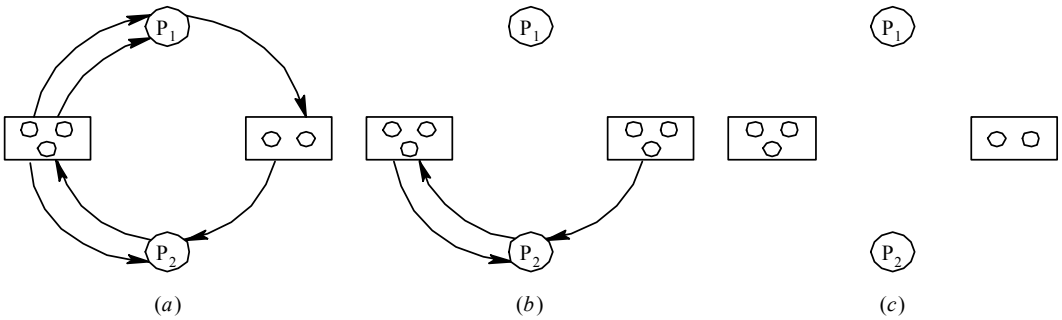


图 3-19 每类资源有多个时的情况

(2) 凡属于 E 中的一个边 e∈E，都连接着 P 中的一个结点和 R 中的一个结点，e={pi, rj}是资源请求边，由进程 pi 指向资源 rj， 它表示进程 pi 请求一个单位的 rj 资源。e={rj, pi}是资源分配边，由资源 rj 指向进程 pi, 它表示把一个单位的资源 rj 分配给进程 pi。

2. 死锁定理



(a) (b) (c)

3. 死锁检测中的数据结构

(1) 可利用资源向量 Available，它表示了 m 类资源中每一类资源的可用数目。

(2) 把不占用资源的进程(向量 Allocation：=0)记入 L 表中， 即 Li∪L。

(3) 从进程集合中找到一个 Requesti≤Work 的进程，做如下处理：① 将其资源分配图简化，释放出资源，增加工作向量 Work：=Work+Allocationi。 ② 将它记入 L 表中。

(4) 若不能把所有进程都记入 L 表中， 便表明系统状态 S 的资源分配图是不可完全简化的。 因此，该系统状态将发生死锁。

```
Work ：=Available;
    L：  =  {Li|Allocationi=0∩Requesti=0}
  for all Li 赁    L do
     begin
          for all Requesti≤Work do
              begin
                  Work：  =Work+Allocationi;
                 Li∪L;
              end
         end
     deadlock：  =    (L={p1, p2, …, pn});
```

死锁的解除

(1) 剥夺资源。

(2) 撤消进程。

为把系统从死锁状态中解脱出来，所花费的代价可表示为：

R(S)min=min{Cui}+min{Cuj}+min{Cuk}+…

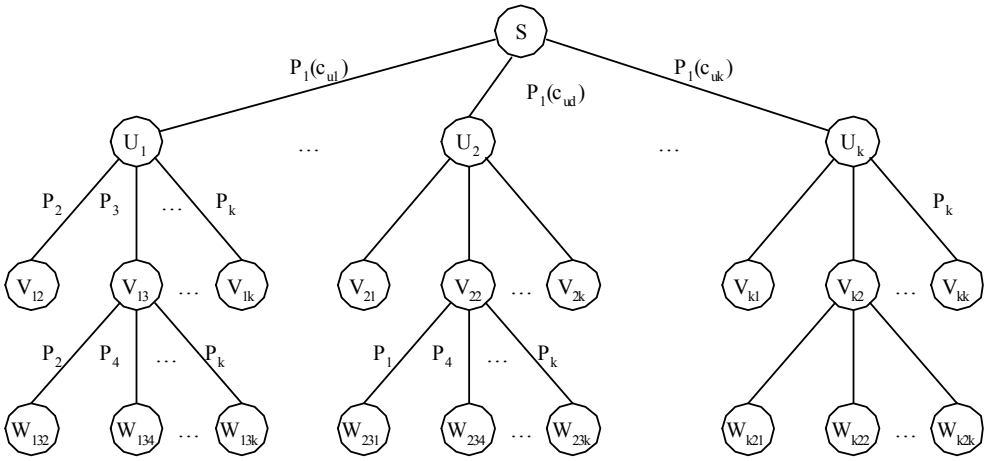


图 3-21 付出代价最小的死锁解除方法

四、存储器管理

4.1 程序的装入和链接

4.2 连续分配方式

4.3 基本分页存储管理方式

4.4 基本分段存储管理方式

4.5 虚拟存储器的基本概念

4.6 请求分页存储管理方式

4.7 页面置换算法

4.8　请求分段存储管理方式

存储器管理的功能

（1）内存分配——为每个进程分配一定的内存空间

（2）地址映射——把程序中所用的相对地址转换成内存中的物理地址

（3）内存保护——检查地址的合法性，防止越界访问

（4）内存扩充——解决“求大于供”的问题，采用虚拟存储技术

程序的装入和链接

从用户的源程序进入系统到相应的程序在机器上运行，所经历的主要处理阶段有：编译阶段、连接阶段、装入阶段和运行阶段。

编译程序：将用户源代码编译成若干个目标模块。

链接程序：将一组目标模块及它们所需要的库函数链接在一起，形成一个完整的装入模块。

装入程序：将装入模块装入内存。

内存空间（或物理空间、绝对空间）由内存一系列存储单元所限定的地址范围。

逻辑地址空间（或地址空间）由程序中逻辑地址组成的地址范围。

相对地址(或逻辑地址)用户程序经编译之后的每个目标模块都以 0 为基地址顺序编址，这种地址称为相对地址。

绝对地址（或物理地址）内存中各物理存储单元的地址是从统一的基地址顺序编址，这种地址称为绝对地址。

- 程序的装入

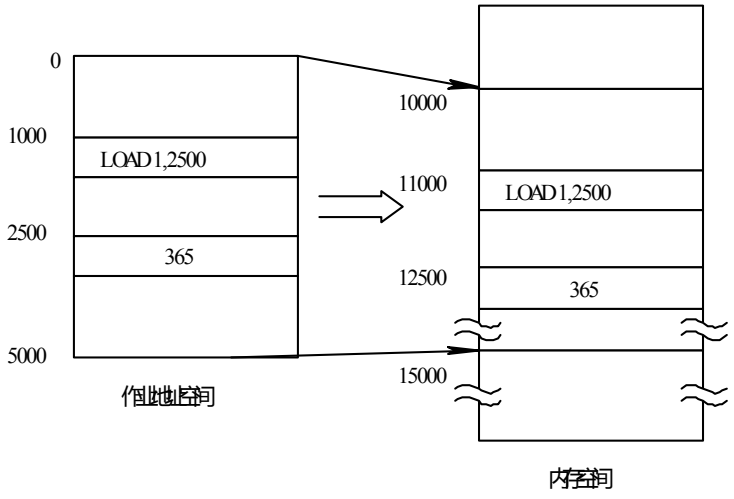
1. 绝对装入方式(Absolute Loading Mode)

程序中所使用的绝对地址，既可在编译或汇编时给出， 也可由程序员直接赋予。 但在由程序员直接给出绝对地址（逻辑地址与绝对地址相同）时， 不仅要求程序员熟悉内存的使用情况，而且一旦程序或数据被修改后，可能要改变程序中的所有地址。因此，通常是宁可在程序中采用符号地址，然后在编译或汇编时，再将这些符号地址转换为绝对地址。

2. 可重定位装入方式(Relocation Loading Mode)

目标模块从 0 编址，其它地址相对于起始地址计算

重定位：装入时对目标程序中指令和数据的修改过程。



作业装入内存时的情况

3. 动态运行时装入方式(Denamle Run-time Loading)

动态运行时的装入程序，在把装入模块装入内存后，并不立即把装入模块中的相对地址

转换为绝对地址，而是把这种地址转换推迟到程序真正要执行时才进行。因此， 装入内存后的所有地址都仍是相对地址。允许程序在内存中移动。

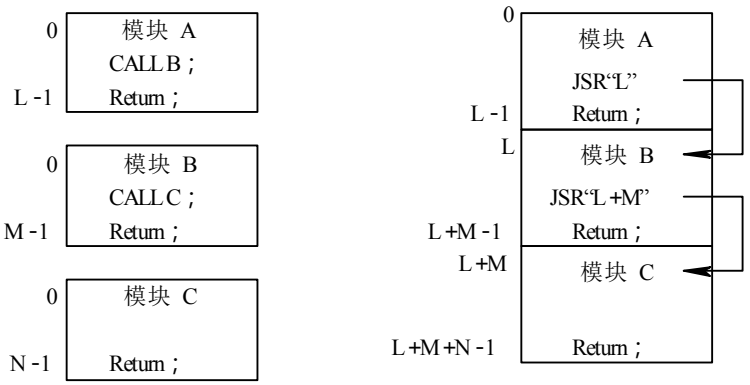
- 程序的链接

  1. 静态链接方式(Static Linking) 执行前将目标模块和他们的库函数，连接成一个完整的装配模块。

  两个问题：

  对相对地址修改

  变换外部调用标号



(*a*) 目标模块　　(*b*) 装入模块

  2. 装入时动态链接(Load time Dynamic Linking)

  将某些目标模块的链接，推迟到执行时才进行。即在执行过程中若发现一个被调用模块尚未装入内存时，由操作系统去找该模块，将它装入内存，并把它连接到调用者模块上。

  装入时动态链接方式有以下优点：

  (1) 便于修改和更新。

  (2) 便于实现对目标模块的共享。

- 连续分配方式

  为一个用户程序分配一个连续的内存空间：

  分为：单一连续分配、固定分区分配、动态分区分配、动态（可）重定位分区分配

  单一连续分配 内存分系统区和用户区，系统区供 OS 使用，通常是放在内存的低址部分；用户区是指除系统区以外的全部内存空间， 提供给用户使用。在内存中仅驻留一道程序，整个用户区为一用户独占。这种分配方式仅能用于单用户、单任务 OS 中。

  固定分区分配 最简单的多道程序的存储管理方式，将内存分为几个固定大小的区域，每个区域装入一道作业。

  划分分区的方法

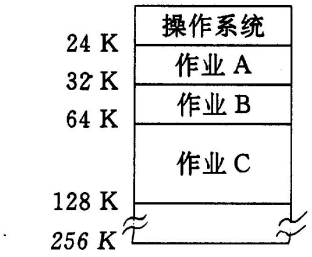
  (1) 分区大小相等， 即使所有的内存分区大小相等。 缺乏灵活性。用于一台计算机控制多个相同对象。

  (2) 分区大小不等。 可根据程序大小为它分配适当分区。

  内存分配 将分区按大小进行分配，建立分区使用表，表项包含分区的起始地址、大小、状态。

分区号	大小(K)	起址(K)	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	已分配

(a) 分区说明表



(b) 存储空间分配情况

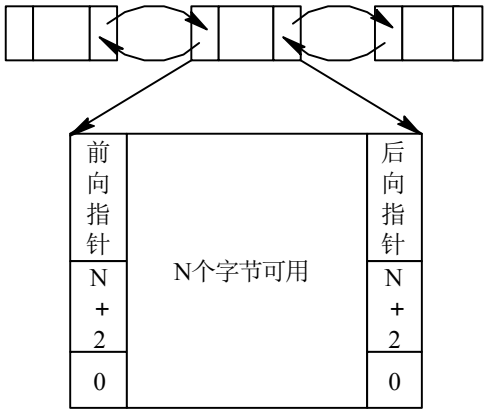
固定分区使用表

动态分区分配

数据结构

(1) 空闲分区表。每个分区占一个表目，包含分区序号、分区地址、分区大小。 与分区说明表类似。

(2) 空闲分区链。在每一个分区的起始部分设置用于控制分区的信息、向前指针，在分区尾部设置一个后向指针，形成双向链表。



分区分配算法

(1) 首次适应算法 FF。

空闲分区链以**地址递增**的次序链接。

分配时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止；再按作业的大小，从该分区中划出一块内存空间分配给请求者，余下的空闲分区仍留在空闲连中。若从链首直至链尾都不能找到一个满足要求的分区，则失败返回。

(2) 循环首次适应算法，该算法是由首次适应算法演变而成的。每一次从上次找到的下一个分区开始查找。指导找到一个能满足要求的分区。

(3) 最佳适应算法。将所有的空闲分区按其**容量从小到大**的顺序形成一个空闲分区链，第一次找到的空闲区必然是最佳的。

分配和回收操作

分区分配操作

1) 分配内存

设请求的分区大小：u.size

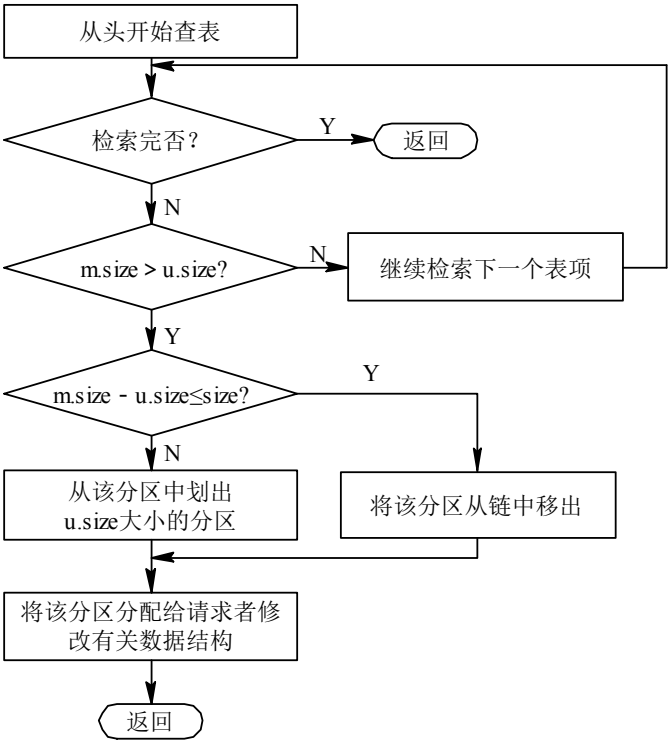
设空闲分区大小：m.size

不可再切割的剩余分区大小：size

如果 m.size-u.size<=size 将整个分区分配给请求者，否则剩余部分留在空闲分区链（表）

中。

将分区的首地址返回给调用者。



2) 回收内存

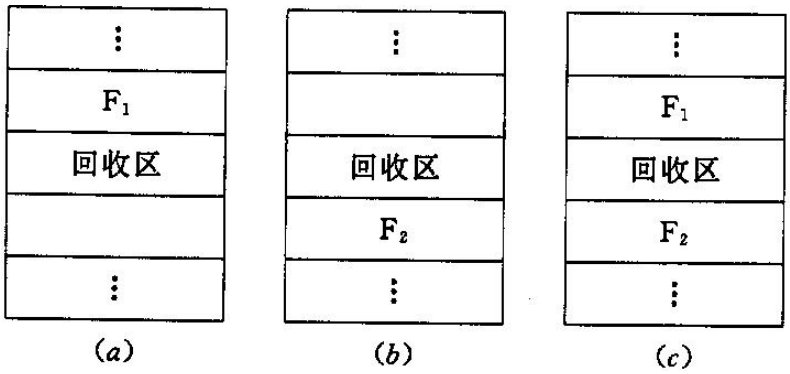
当进程释放内存时，系统根据回收区的首值，从空闲区链（表）中找到相应的插入点，回收区可能出现四种情况：

（1）与插入点的前一个空闲区 F1 相邻接。 并到 F1

（2）与插入点的后一个空闲分区 F2 相邻接。 F2 并到回收区

（3）同时与插入点的前、后两个分区邻接。 三区合并

（4）既不与 F1 邻接，又不与 F2 邻接。 单独建立空闲区

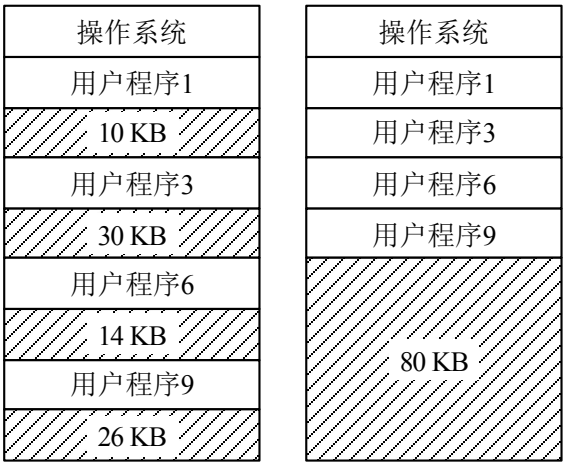


动态（可）重定位分区分配

1. 动态重定位的引入

碎片：内存中不能被利用的小分区称为“零头”或“碎片”。

分散的小分区拼接成一个大分区的方法，称为“拼接”或“紧凑”。



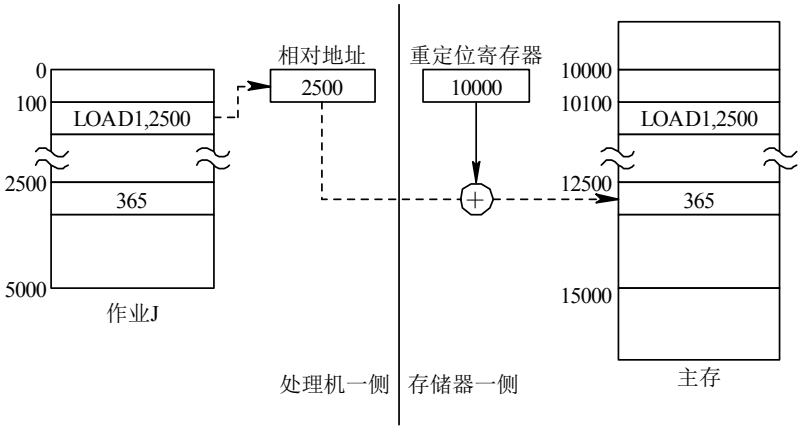
(*a*) 紧凑前　　(*b*) 紧凑后

紧凑的示意

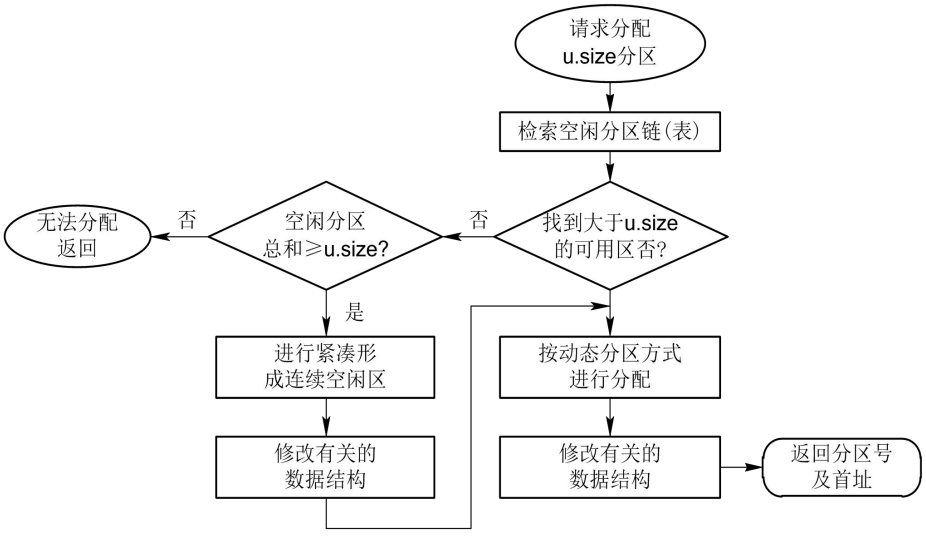
2. 动态重定位的实现

地址变换的过程是在程序执行期间，随着对每条指令或数据的访问自动进行的，故称为动态重定位。

增设硬件机构：重定位寄存器。



动态重定位分区分配算法



对换(Swapping) 即中级调度

把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据，调入内存。对换是提高内存利用率的有效措施。

整体对换：以进程为单位，又称进程对换。

部分对换：

（1）页面对换，以“页”为单位

（2）分段对换，以“段”为单位

进程对换实现功能：

对换空间的管理

为了能对对换区中的空闲盘块进行管理，在系统中应配置相应的数据结构，以记录外存的使用情况。其形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表中的每个表目中应包含两项， 即对换区的首址及其大小，它们的单位是盘块号和盘块数。

外存包括：

文件区 为提高空间利用率采用**离散分配**方式

对换区 为提高对换速度采用**连续分配**方式

进程的换出

每当一进程由于创建子进程而需要更多的内存空间，但又无足够的内存空间等情况发生时，系统应将某进程换出。 其过程是：系统首先选择处于阻塞状态且优先级最低的进程作为换出进程，然后启动盘块，将该进程的程序和数据传送到磁盘的对换区上。若传送过程未出现错误，便可回收该进程所占用的内存空间，并对该进程的进程控制块做相应的修改。

进程的换入

系统应定时地查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将其中换出时间(换出到磁盘上)最久的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。

- 离散分配方式

基本思想：将一个进程分散的装入不相邻的分区中

离散分配的基本单位是页，则称为分页存储管理方式；如果离散分配的基本单位是段，则称为分段存储管理方式。

基本分页存储管理方式

不具备页面对换功能

又称纯分页存储管理方式

不支持虚拟存储器功能

分页存储管理

将一个进程的逻辑地址空间分成若干个大小相等的片，称为页面或页，并为各页编号，从 0 开始，如第 0 页、第一页等。

把内存空间分成与页面大小相等的若干个存储块，称为块或者页框，也加以编号，如 0#、1#块等。

以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。

页内碎片：由于进程的最后一页经常不满一块而形成不可利用的碎片，称之为“页内碎片”。

页面大小

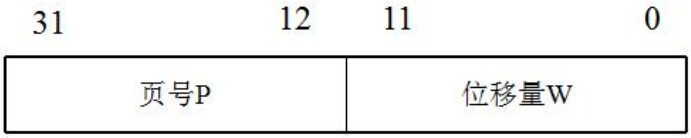
通常 512B-8KB

页面太小：页内碎片小，提高内存利用率，但页表过长，占内存；降低对换效率

页面太大：提高了对换速度，但页内碎片大，降低了内存利用率

地址结构

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：

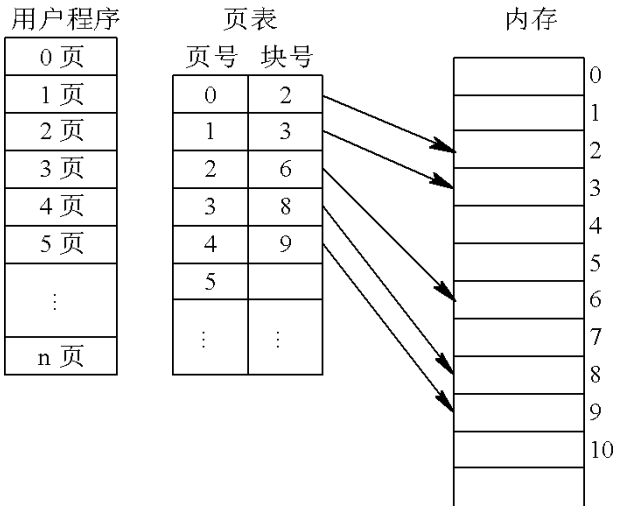
$$P = INT\left[\frac{A}{L}\right]$$

$$d = [A]MODL$$

3. 页表

页面系统为了能在内存中找到每个页面对应的物理块而为进程建立的一张页面映像表，简称**页表**。

页表作用：实现从页号到物理块号的地址映射



表象中常设有存取控制字段

一位：允许读/写

只读

两位：允许读/写

只读

只执行

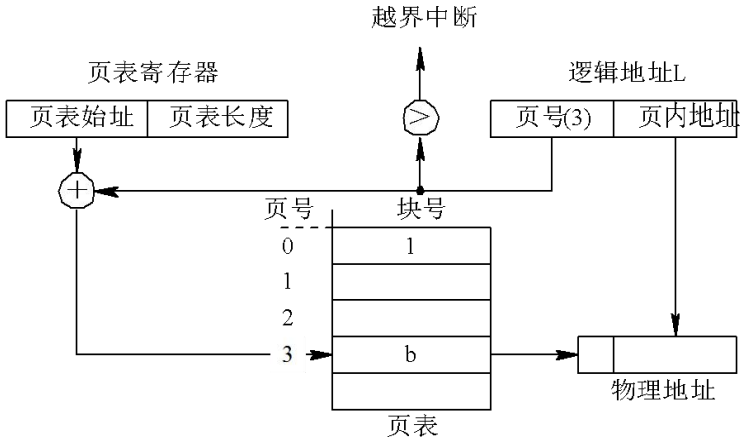
- 地址变换机构

基本任务：实现从逻辑地址到物理地址的转换。实际上是将逻辑地址中的页号，转换为内存中的物理块号。

地址变换任务是借助于页表来完成的

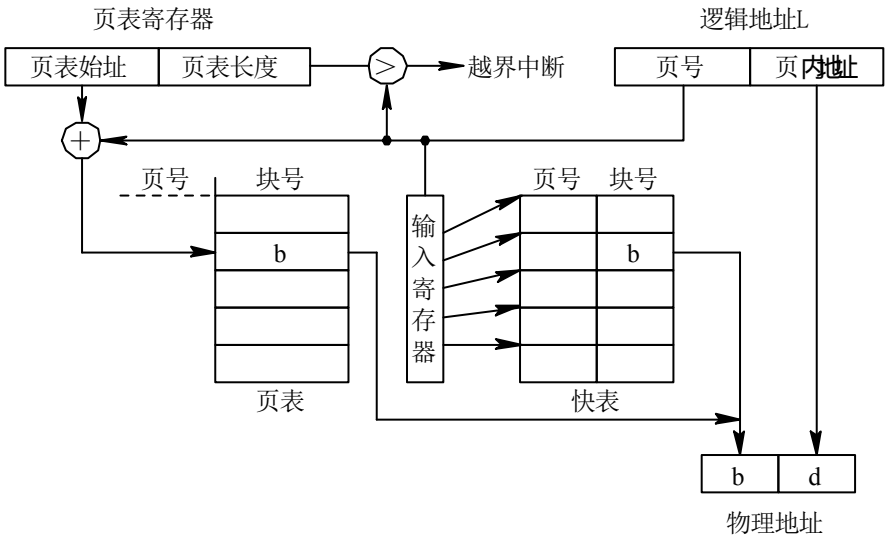
页表由一组专门的寄存器来实现，一个页表项用一个寄存器。

页表大多数驻留在内存中。
系统中只设置一个页表寄存器 PTR，存放页表在内存的始值和页表的长度。
进程未执行时，页表的始值和页表长度存放在本进程的 PCB 中。当调度到进程时装入页表寄存器中。



分页系统的地址变换机构

具有快表的地址变换机构
快表引入原因
CPU 存取一个数据时要两次访问内存，第一次访问页表，找到指定页的物理块号，再将块号与页内偏移量 W 拼接成物理地址
第二次访问内存时**从所得地址中获得所需数据（或向此地址中写入数据）**
为提高地址变换速度：
增设了一个具有并行查询能力的高速缓冲寄存器，称为“**联想寄存器**”或“**快表**”，用于存放当前访问的**页表项**。



快表通常只放 16-512 个页表项，大型作业表只能将其一部分页表项放入其中，从块表能找到所需页表项的命中率可达 90%

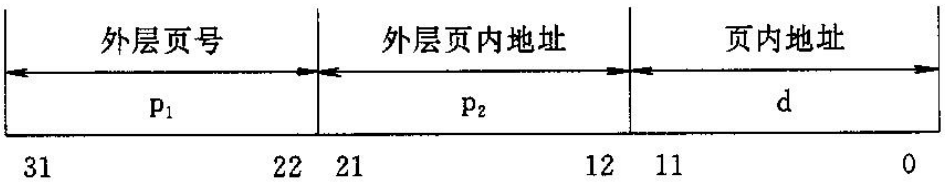
- 两级和多级页表

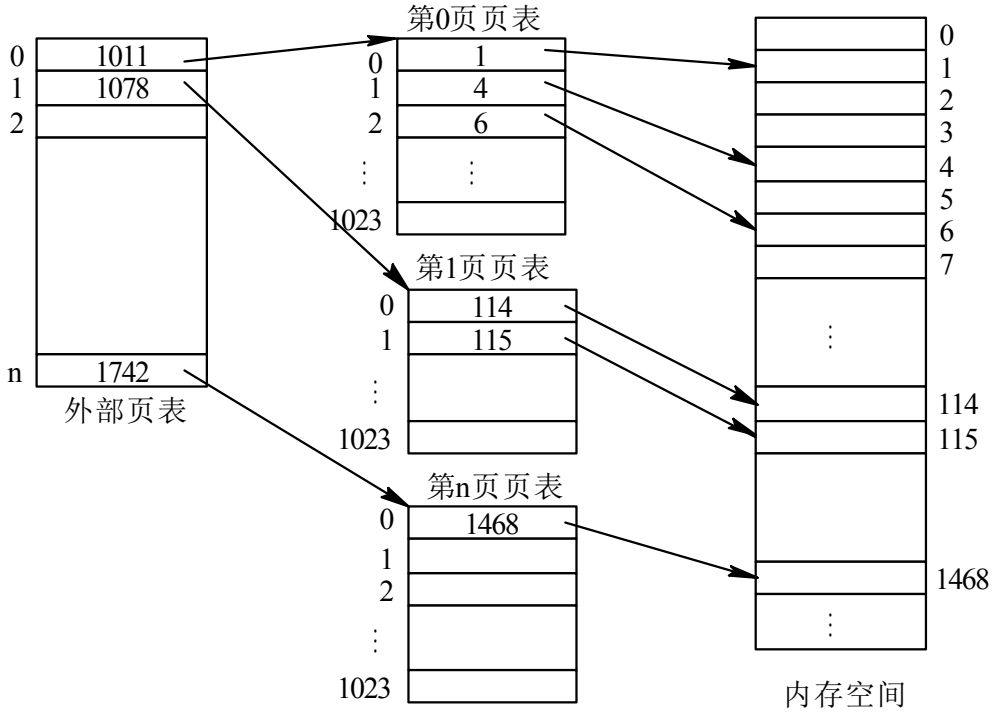
现代的大多数计算机系统，都支持非常大的逻辑地址空间(2^{32}~2^{64})。在这样的环境

下，页表就变得非常大，要占用相当大的内存空间。例如，对于一个具有 32 位逻辑地址空间的分页系统，规定页面大小为 4 KB 即 212 B，则在每个进程页表中的页表项可达 1 兆个之多。又因为每个页表项占用一个字节， 故每个进程仅仅其页表就要占用 4 KB 的内存空间，而且还要求是连续的。 可以采用这样两个方法来解决这一问题：① 采用离散分配方式来解决难以找到一块连续的大内存空间的问题：② 只将当前需要的部分页表项调入内存， 其余的页表项仍驻留在磁盘上，需要时再调入。

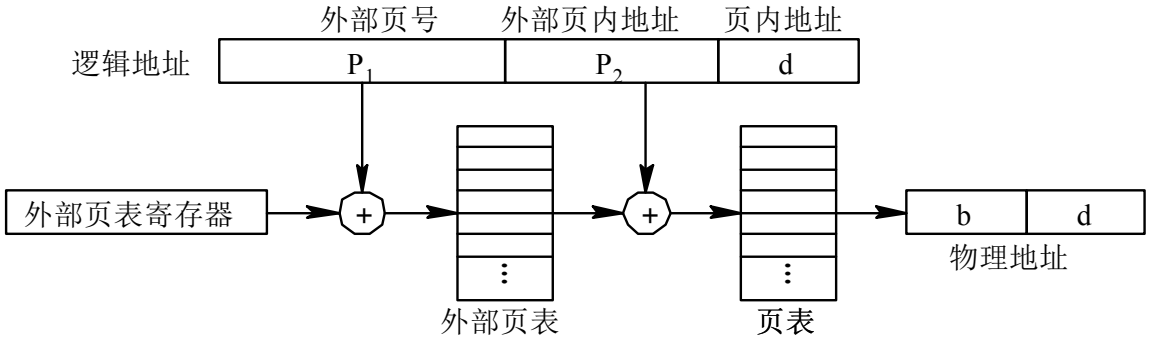
两级页表：为离散分布的页表再建立一张页表，称为外层页表，在每个页表项中记录了页表页面的物理块号。

逻辑地址结构可描述如下：





两级页表结构



具有两级页表的地址变换机构

- 基本分段存储管理方式

1. 引入分段存储管理方式， 主要是为了满足用户和程序员的下述一系列需要：
   1) 方便编程
   2) 信息共享
   3) 信息保护
   4) 动态增长
   5) 动态链接

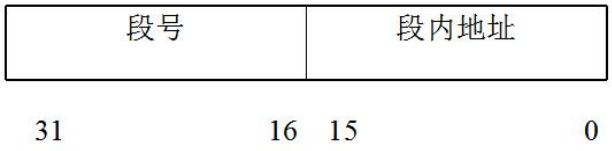
2.分段系统的基本原理

分段：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，每个段定义一组逻辑信息。

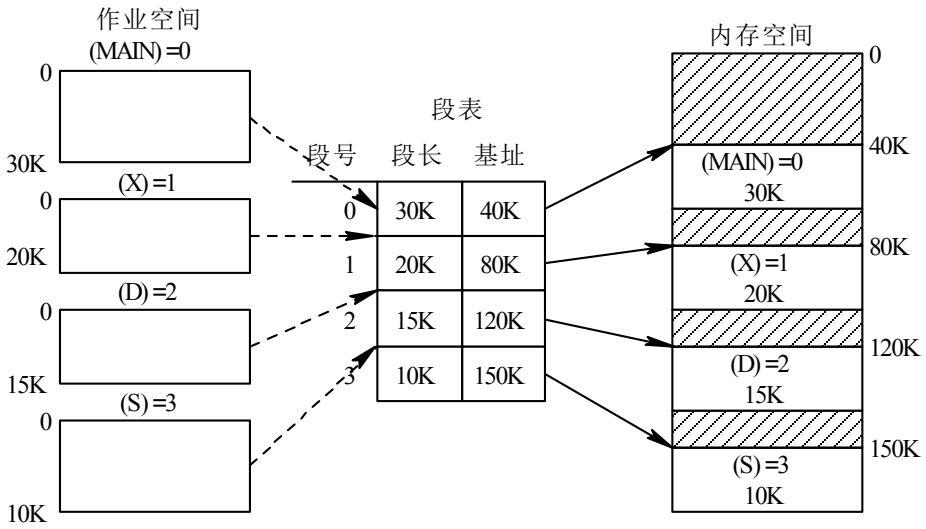
每个段从 0 开始编址。采用连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而段长不等。

整个作业的地址空间分成多个段，是二维的。

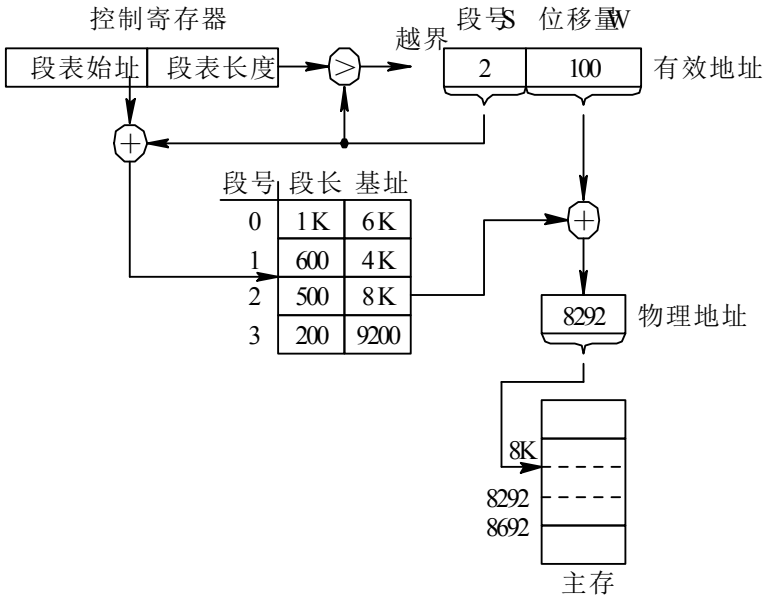
分段地址中的地址具有如下结构：



段表：系统为每一个进程建立一张段映射表，简称段表。用于实现从逻辑段到物理内存的映射。



利用段表实现地址映射



分段系统的地址变换过程

分页和分段的主要区别

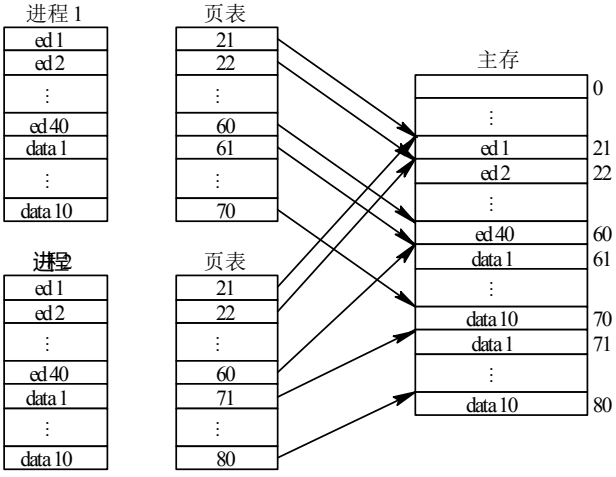
(1) 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头，提高内存的利用率。或者说， 分页仅仅是由于系统管理的需要而不是用户的需要。段则是信息的逻辑单位，它含有一组其意义相对完整的信息。 分段的目的是为了能更好地满足用户的需要。

(2) 页的大小固定且由系统决定，由系统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，因而在系统中只能有一种大小的页面；而段的长度却不固定，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，程序员只需利用一个记忆符，即可表示一个地址； 而分段的作业地址空间则是二维的，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。

3.信息共享

可重入代码：又称纯代码：一种允许多个进程同时访问但不允许任何进程对它进行修改的代码。



4.段页式存储管理方式

虚拟存储器的实现方法

分页请求系统

（1）硬件支持。

① **请求分页的页表机制**，它是在纯分页的页表机制上增加若干项而形成的，作为请求分页的数据结构；② **缺页中断机构**，即每当用户程序要访问的页面尚未调入内存时 便产生一缺页中断，以请求 OS 将所缺的页调入内存；③ **地址变换机构**， 它同样是在纯分页地址变换机构的基础上发展形成的.

(2) 实现请求分页的软件。

虚拟存储器的特征

1.多次性

2.对换性

3. 虚拟性

- 请求分页存储管理方式

请求分页中的硬件支持

内存分配策略和分配算法

调页策略

请求分页中的硬件支持

1. 页表机制

基本作用是将逻辑地址变换为物理地址，在页表中再增加若干项，供换进换出。

页号	物理块号	状态位 P	访问字段 A	修改位 M	外存地址

2. 缺页中断机构

请求分页系统中每当所要访问的页面不在内存时，便要产生缺页中断、请求将所缺之页调入内存。

与一般中断的区别：

（1）在指令执行期间产生和处理中断信号。

（2）一条指令在执行期间可能产生多次缺页中断。

内存分配的三个问题：

最小物理块数

能保证进程正常运行所需的最小物理块数，与计算机的硬件结构有关。取决于**指令的格式、功能**和**寻址方式。**

物理块的分配策略

1）固定分配局部置换

为每一个进程分配一个固定页数的内存空间，在整个运行过程中都不改变。

如果缺页，则只能从该进程的页面中选择一页换出，再调入一页。

困难：应为每个进程分配多少个页的内存难以确定，若太少会**频繁地出现缺页中断**降低吞吐量；太多，又使**内存中进程数减少**，进而可能造成 CPU 或其他资源空闲，而且进程对换会花分费更多的时间。

2）可变分配全局置换

先为每个进程分配一定数目的物理块。OS 保持一个空闲物理块队列。

缺页时，系统从空闲物理块队列中，取出一个物理块分配给该进程。并将预调入的缺页装入其中。

当空闲物理块队列为空时从内存中选择一页调出。

3）可变分配局部置换

基于进程的类型或程序员的要求，为每个进程分配一定数目的物理块。

缺页时从该进程中的页面中选择一页换出。

如果进程频繁地发生缺页中断，则在为该进程分配附加的物理块。

若一个进程的缺页率特别低，则可适当减少该进程的物理块。

物理快的分配算法：

平均分配算法

按比例分配算法 进程大小

考虑优先权分配算法

调页策略

1）预调页策略 主要应用于首次调入

2）请求调页策略

从何处调入页面 （对换区都在外存）

在请求分页系统中的外存分为两部分：用于存放文件的文件区和用于存放对换页面的对换区。通常，由于对换区是采用连续分配方式，而事件是采用离散分配方式，故对换区的磁盘 I/O 速度比文件区的高。这样，每当发生缺页请求时，系统应从何处将缺页调入内存，可分成如下三种情况：

(1) 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。为此，在进程运行前， 便须将与该进程有关的文件，从文件区拷贝到对换区。

(2) 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，由于它们未被修改而不必再将它们换出，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) UNIX 方式。由于与进程有关的文件都放在文件区，故凡是未运行过的页面，都应从文件区调入。而对于曾经运行过但又被换出的页面，由于是被放在对换区，因此在下次调入时，应从对换区调入。由于 UNIX 系统允许页面共享，因此， 某进程所请求的页面有可能已被其它进程调入内存，此时也就无须再从对换区调入。

页面调入过程

每当程序所要访问的页面未在内存时，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。该程序通过查找页表，得到该页在外存的物理块后， 如果此时内存能容纳新页，则启动磁盘 I/O 将所缺之页调入内存，然后修改页表。如果内存已满，则须先按照某种置换算法从内存中选出一页准备换出；如果该页未被修改过，可不必将该页写回磁盘；但如果此页已被修改， 则必须将它写回磁盘，然后再把所缺的页调入内存， 并修改页表中的相应表项，置其存在位为“1”，并将此页表项写入快表中。在缺页调入内存后，利用修改后的页表， 去形成所要访问数据的物理地址，再去访问内存数据。

- 页面置换算法 （选择换出页面的算法）

抖动：刚被换出的页很快又被访问，需重新调入，又需再选一页调出，如此频繁地更换页面的现象称为抖动。

抖动导致进程在运行中，把大部分时间花费在页面置换上。

1. 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。

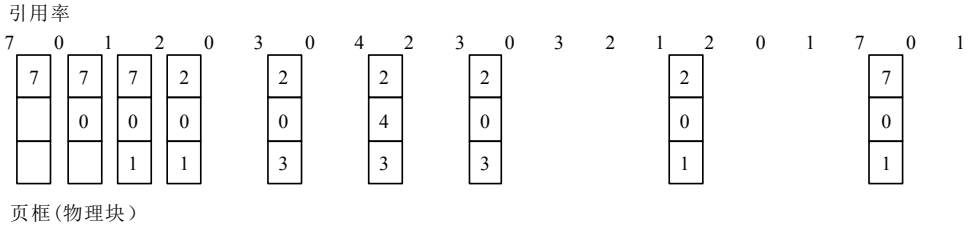
其所选择的被淘汰页面，将是以后永不使用的， 或许是在最长(未来)时间内不再被访问的页面。采用最佳置换算法，通常可保证获得最低的缺页率。 但无法估计那个页面是长时间不用的，故该算法无法实现。

可利用该算法评价其他算法。

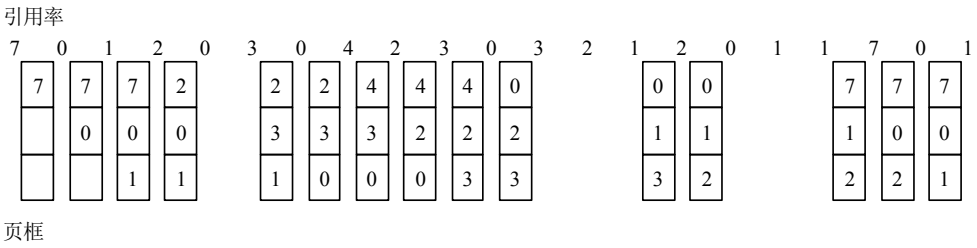
假定系统为某进程分配了三个物理块， 并考虑有以下的页面号引用串：

7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1，7，0，1

进程运行时， 先将 7，0，1 三个页面装入内存。 以后， 当进程要访问页面 2 时， 将会产生缺页中断。此时 OS 根据最佳置换算法， 将选择页面 7 予以淘汰。

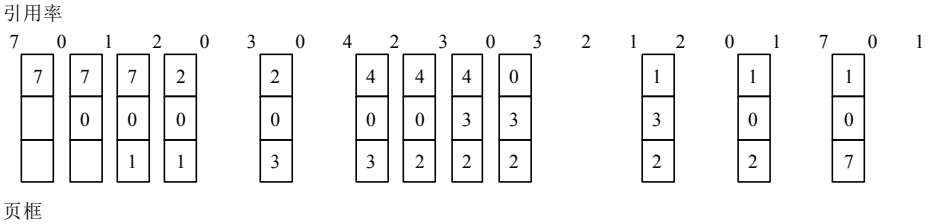


2. 先进先出(FIFO)页面置换算法



3.最近最久未使用(LRU)置换算法

根据页面调入内存后的使用情况 最近的过去近似最近的将来



4.Clock 置换算法

1）简单的 Clock 置换算法

每页设置一位访问位。再将内存中的所有页面都通过链接指针链成一个循环队列。

当某也被访问时，其访问位置 1.淘汰时检查其访问位，如果是 0 则换出；若是 1 则重新将它复 0.再按 FIFO 算法检查下个页面。到队列中的最后一个页面时，若其访问位仍为 1，则再返回到队首再去检查第一个页面。

又称为最近未用算法。

2）改进型 Clock 置换算法

由访问位 A 和修改位 M 可以组合成下面四种类型的页面：

1 类(A=0, M=0): 表示该页最近既未被访问， 又未被修改， 是最佳淘汰页。

2 类(A=0, M=1)： 表示该页最近未被访问， 但已被修改， 并不是很好的淘汰页。

3 类(A=1, M=0)： 最近已被访问， 但未被修改， 该页有可能再被访问。

4 类(A=1, M=1): 最近已被访问且被修改， 该页可能再被访问。

其执行过程可分成以下三步：

(1) 从指针所指示的当前位置开始， 扫描循环队列， 寻找 A=0 且 M=0 的第一类页面， 将所遇到的第一个页面作为所选中的淘汰页。 在第一次扫描期间不改变访问位 A。

(2) 如果第一步失败，即查找一周后未遇到第一类页面， 则开始第二轮扫描，寻找 A=0 且 M=1 的第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将所有扫描过的页面的访问位都置 0。

(3) 如果第二步也失败，亦即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复 0。 然后重复第一步，如果仍失败，必要时再重复第二步，此时就一定能找到被淘汰的页。

5.其它置换算法

1）最少使用(LFU： Least Frequently Used)置换算法

2）页面缓冲算法(PBA： Page Buffering Algorithm)

采用可变分配和局部置换方式

如果页面未被修改，就把它直接放入空闲链表；否则，放入以修改页面的链表中。

页面在内存并不做物理移动而只是将页表中的表项移到链表之中

- 请求分段存储管理方式

请求分段中的硬件支持

1. 段表机制

段名	段长	段的基址	存取方式	访问字段 A	修改位 M	存在位 P	增补位	外存始址

在段表项中， 除了段名(号)、 段长、 段在内存中的起始地址外， 还增加了以下诸项：

(1) 存取方式。 表示本段的属性：只读、只执行、允许读/写

(2) 访问字段 A。访问频率

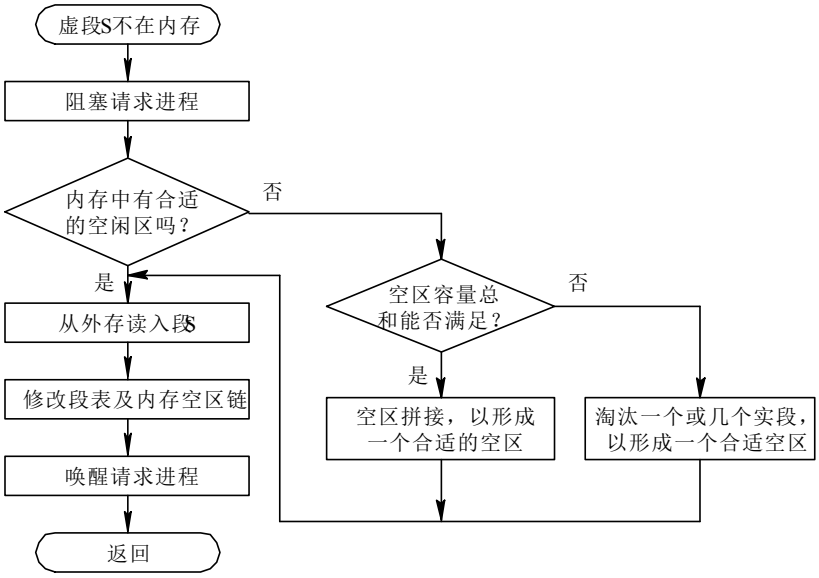
(3) 修改位 M。用于表示该也在进入内存后是否被修改过，供置换页面时参考。

(4) 存在位 P。指示本段是否已调入内存，供程序访问时参考

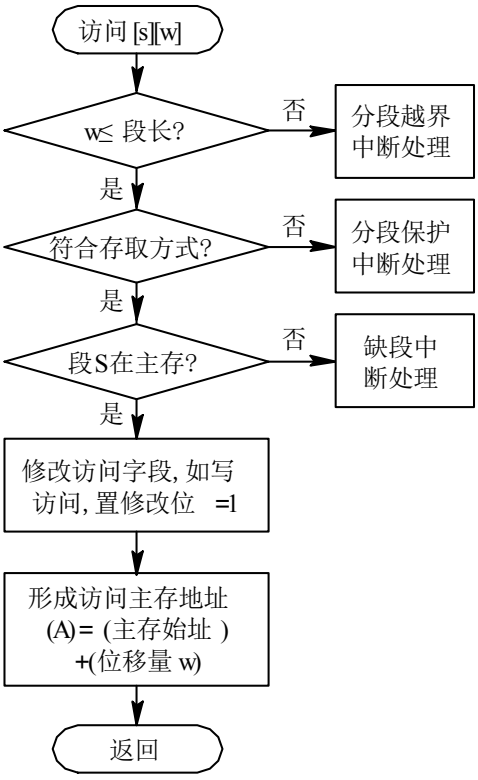
(5) 增补位。 特有的字段，用于本段在运行过程中，是否做过动态增长

(6) 外存始址。 本段在外存中的起始地址，即起始盘块号

2. 缺段中断机构

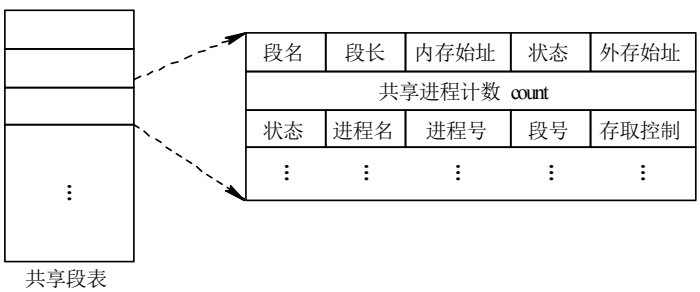


3. 地址变换机构 请求分段系统的地址变换过程



- 分段的共享与保护

1. 共享段表



（1）共享进程计数 COUNT
（2）存取控制字段。对于一个共享段，应给不同的进程以下不同的存取权限。
（3）段号。对于一个共享段，不同的进程可以各用不同的段号去共享该段。

2. 共享段的分配与回收

1) 共享段的分配

在为共享段分配内存时，对第一个请求使用该共享段的进程，由系统为该共享段分配一物理区，再把共享段调入该区，同时将该区的始址填入请求进程的段表的相应项中，还须在共享段表中增加一表项，填写有关数据，把 count 置为 1；之后，当又有其它进程需要调用该共享段时，由于该共享段已被调入内存，故此时无须再为该段分配内存，而只需在调用进程的段表中，增加一表项，填写该共享段的物理地址；在共享段的段表中，填上调用进程的进程名、存取控制等，再执行 count：=count+1 操作，以表明有两个进程共享该段。

2) 共享段的回收

当共享此段的某进程不再需要该段时，应将该段释放， 包括撤在该进程段表中共享段所对应的表项，以及执行 count：= count-1 操作。若结果为 0，则须由系统回收该共享段的物理内存，以及取消在共享段表中该段所对应的表项， 表明此时已没有进程使用该段；否则(减 1 结果不为 0)， 则只是取消调用者进程在共享段表中的有关记录。

3. 分段保护

1) 越界检查 将逻辑地址空间的段号与段表长度进行比较，如果段号大于等于段表长度，将发出地址越界中断信号。

检查段内地址是否大于等于段长。

2) 存取控制检查

(1)只读

(2) 只执行

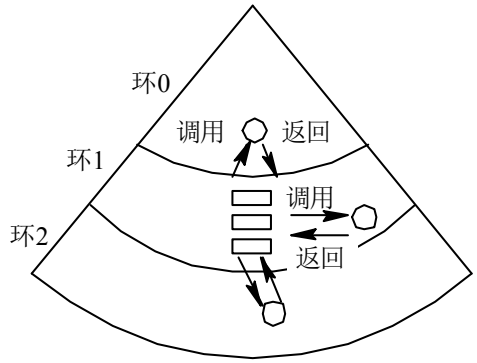
(3) 读/写

3) 环保护机构 低编号的环具有高优先权

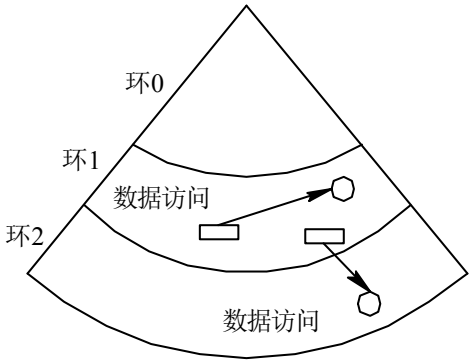
程序的访问和调用遵循以下规则：

一个程序可以访问驻留在相同环或较低特权环中的数据。

一个程序可以调用驻留在相同环或较高特权环中的服务



(*a*) 程序间的控制传输



(*b*) 数据访问

五、设备管理

设备管理的主要功能：

缓冲区管理

设备分配

设备处理
虚拟设备
实现设备独立性

5.1 I/O 系 统

实现信息输入、输出和存储的系统。包括：
I/O 设备
总线
设备控制器
I/O 通道
I/O 处理机

1. I/O 设备的类型

1) 按传输速率分类

按传输速度的高低，可将 I/O 设备分为三类。

第一类是**低速设备**，这是指其传输速率仅为每**秒钟几个字节至数百个字节**的一类设备。属于低速设备的典型设备有键盘、 鼠标器、语音的输入和输出等设备。

第二类是**中速设备**，这是指其传输速率在**每秒钟数千个字节至数万个字节**的一类设备。典型的中速设备有行式打印机、激光打印机等。

第三类是**高速设备**， 这是指其传输速率在**数百千个字节至数十兆字节**的一类设备。 典型的高速设备有磁带机、 磁盘机、 光盘机等。

2) 按信息交换的单位分类

可将 I/O 设备分成两类。

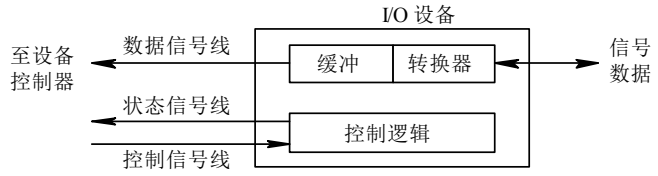
第一类是**块设备(Block Device)**，这类设备**用于存储信息**。 由于信息的存取总是以数据块为单位， 故而得名。 它属于**有结构设备**。典型的块设备是磁盘，每个盘块的大小为 512 B~4 KB。磁盘设备的基本特征是其传输速率较高，通常每秒钟为几兆位；另一特征是**可寻址**，即对它可随机地读/写任一块；此外，磁盘设备的 I/O 常采用 DMA（直接存取）方式。

第二类是**字符设备(Character Device)**，用于**数据的输入和输出**。 其基本单位是字符， 故称为字符设备。基本特征是**传输效率低**和**不可寻址。**输入输出时常采用中断驱动方式。

3) 按设备的共享属性分类

这种分类方式可将 I/O 设备分为如下三类：
(1) 独占设备。 在一段时间内只允许一个用户（进程）访问设备。如打印机
(2) 共享设备。 在一段时间只允许多个用户（进程）访问设备。如：磁盘
(3) 虚拟设备。

2. 设备与控制器之间的接口



数据信号。对输入是由设备发送给设备控制器的；对输出是由设备控制器所接受的比特流。
控制信号：是设备控制器发送给设备的、用于规定设备执行、读或写操作的信号。
状态信号：用于指示设备当前的状态

5.1.2 设备控制器

是 CPU 和 I/O 设备的接口

分成两大类：

用于控制字符设备的控制器

用于控制块设备的控制器

微型机和小型机中的控制器常做成印制电路卡形式，称接口卡。

1. 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

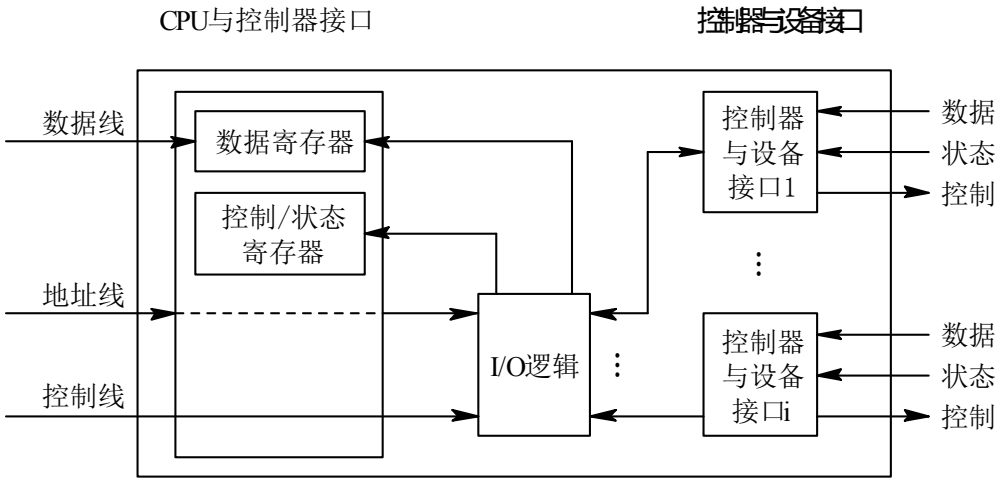
6) 差错控制

2. 设备控制器的组成

1)设备控制器与处理机接口

2)设备控制器与设备接口

3)I/O 逻辑



与处理机接口：

三类信号线：数据线、地址线、控制线

两类寄存器：数据寄存器、控制/状态寄存器

与设备接口：

一个接口连接一台设备

每个接口中含有数据、地址、控制信号

控制器的 I/O 逻辑根据处理机发的地址信号选择设备接口

I/O 逻辑

通过一组控制线与处理机交互

处理及利用它向控制器发送 I/O 命令

I/O 逻辑对接收到的命令进行译码

5.1.3 I/O 通道

1. I/O 通道(I/O Channel)设备的引入

实际上，I/O 通道是一种特殊的处理机。它具有执行 I/O 指令的能力，并通过执行通道(I/O)程序来控制 I/O 操作。但 I/O 通道又与一般的处理机不同，主要表现在以下两个方面：

一是其指令类型单一，这是由于通道硬件比较简单， 其所能执行的命令，主要局限于与 I/O 操作有关的指令； 再就是通道没有自己的内存，通道所执行的通道程序是放在主机的内存中的， 换言之，是通道与 CPU 共享内存。

2. 通道类型

1) 字节多路通道(Byte Multiplexor Channel)

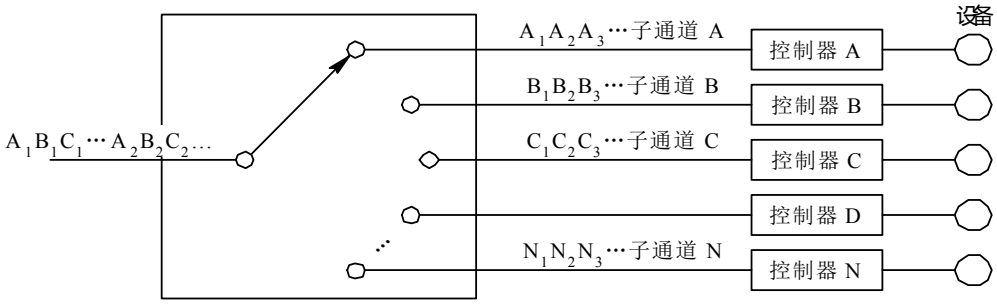
含有非分配型子通道，其数量从几十到几百，每台子通道连接一台 I/O 设备。

子通道按时间片轮转方式共享主通道

第一个子通道控制其 I/O 设备完成一个字节的交换后，便立即腾出字节多路通道（主通道），让给第二个子通道使用，依次类推，所有通道轮转一周后返回。

只要扫描每个子通道的速度足够快，而连接到子通道上的设备速率较小时，不丢数据。

连接低速或中速设备时，便不丢失信息。



2) 数组选择通道(Block Selector Channel)

字节多路通道不适于连接高速设备，这推动了按数组方式进行数据传送的数组选择通道的形成。这种通道虽然可以连接多台高速设备，但由于它只含有一个分配型子通道，在一段时间内只能执行一道通道程序， 控制一台设备进行数据传送， 致使当某台设备占用了该通道后，便一直由它独占， 即使是它无数据传送，通道被闲置， 也不允许其它设备使用该通道， 直至该设备传送完毕释放该通道。可见，这种通道的利用率很低。

3) 数组多路通道(Block Multiplexor Channel)

数组选择通道虽有很高的传输速率，但它却每次只允许一个设备传输数据。数组多路通道是将数组选择通道传输速率高和字节多路通道能使各子通道(设备)分时并行操作的优点相结合而形成的一种新通道。它含有多个非分配型子通道， 因而这种通道既具有很高的数据传输速率，又能获得令人满意的通道利用率。也正因此，才使该通道能被广泛地用于连接多台高、中速的外围设备，其数据传送是按数组方式进行的。

3. “瓶颈”问题

I/O 性能经常成为系统瓶颈

（1）CPU 性能不等于系统性能，响应时间也是一个重要因素

（2）CPU 性能越高，与 I/O 差距越大

弥补：更多的进程

（3）进程切换多，系统开销大

通道价格昂贵，使机器中的通道数量必然减少，这往往使它成了 I/O 的瓶颈。解决瓶颈问题的最有效方法，便是增加设备到主机间的通路而不增加通道。

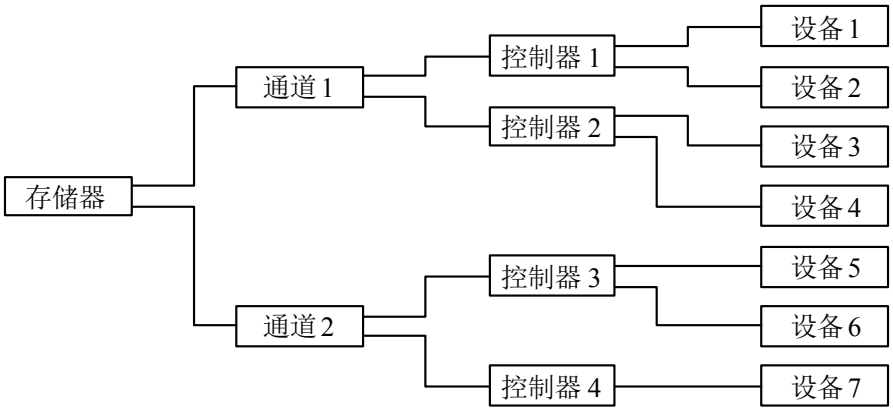


图 5-4 单通路 I/O 系统

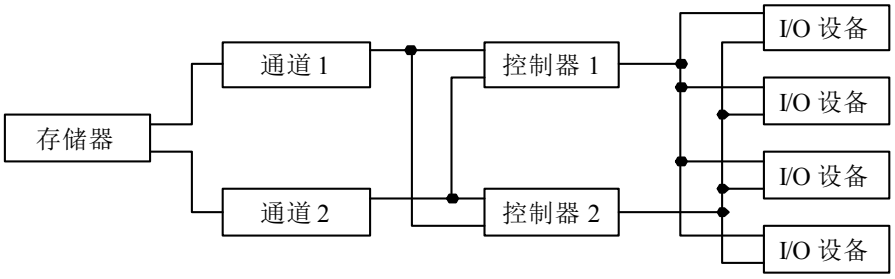


图 5-5 多通路 I/O 系统

5.1.4 总线系统

总线：将计算机系统中的各个子系统（CPU、内存、外设等）相互连接，且连接是共享的。

优点：低成本（一线多用）

灵活性（易于增加设备，便于两个系统间共享外设）

缺点：本身形成了通讯瓶颈，限制 I/O 吞吐量。

总线分类：

数据总线、地址总线、控制总线

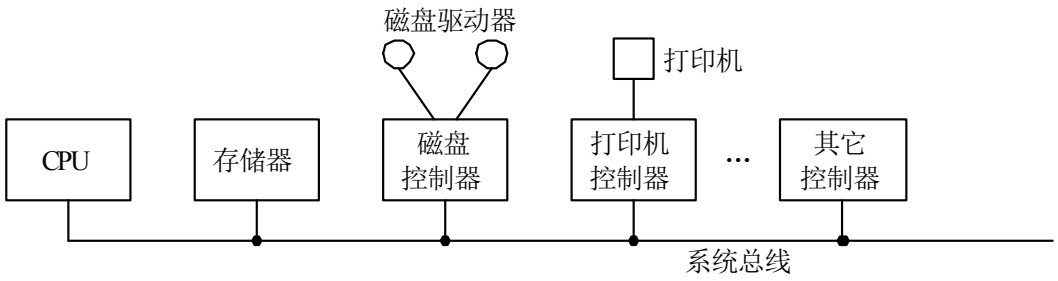


图 5-6 总线型 I/O 系统结构

1. ISA 和 EISA 总线

1) ISA(Industry Standard Architecture)总线

这是为了 1984 年推出的 80286 型微机而设计的总线结构。 其总线的带宽为 8 位，最高传输速率为 2 Mb/s。之后不久又推出了 16 位的(EISA)总线，其最高传输速率为 8 Mb/s，

后又升至 16 Mb/s， 能连接 12 台设备。

2) EISA(Extended ISA)总线

到 80 年代末期，ISA 总线已难于满足带宽和传输速率的要求， 于是人们又开发出扩展 ISA(EISA)总线，其带宽为 32 位，总线的传输速率高达 32 Mb/s，同样可以连接 12 台外部设备。

2. 局部总线(Local Bus)

1) VESA(Video Electronic Standard Association)总线

带宽为 32 位

最高传输速率为 132MB/S

VESA 总线用于 486 微机中

缺点：

连接的设备数仅为 2~4 台

在控制器中无缓冲，难适应处理器速度的提高，不支持 PENTIUM 微机。

2) PCI(Peripheral Component Interface)总线

支持 64 位系统

CPU 和外设间插入一复杂的管理层，用于协调数据传输和提供一致数据接口。

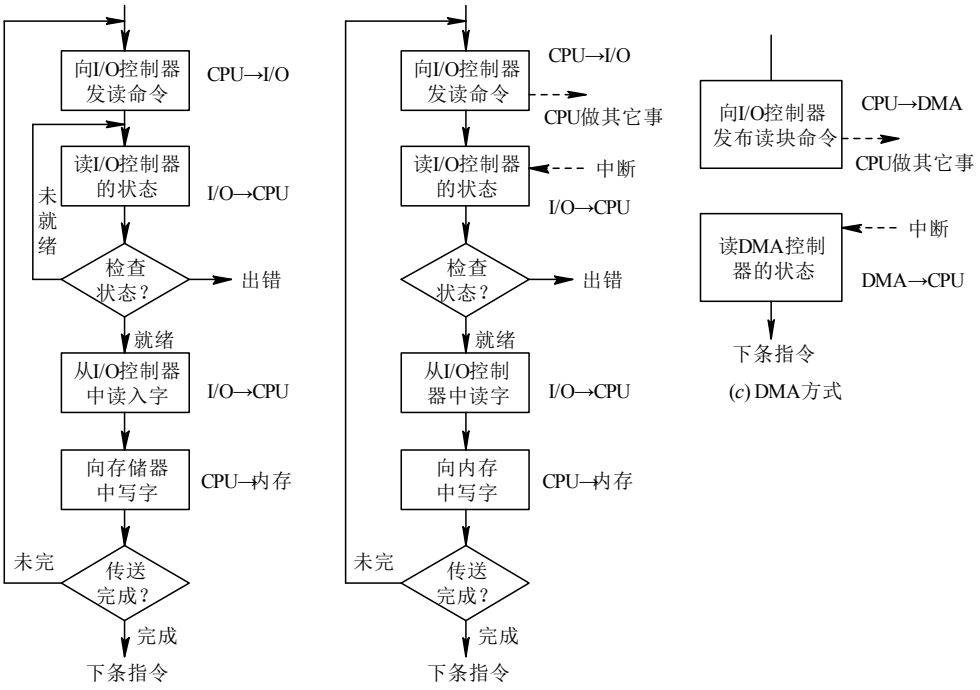
管理层中有数据缓冲，将线路的驱动能力放大，使 PCI 最多能支持 10 种外设，并使高时钟频率的 CPU 能很好地运行。

即可连接 ISA、EISA 等传统型总线，又可支持 PENTIUM 的 64 位系统

5.2 I/O 控制方式

5.2.1 程序 I/O 方式

在程序 I/O 方式中，由于 CPU 的高速性和 I/O 设备的低速性， 致使 CPU 的绝大部分时间都处于等待 I/O 设备完成数据 I/O 的循环测试中， 造成对 CPU 的极大浪费。在该方式中，CPU 之所以要不断地测试 I/O 设备的状态，就是因为在 CPU 中无中断机构， 使 I/O 设备无法向 CPU 报告它已完成了一个字符的输入操作。



(*a*) 程序I/O方式　　(*b*) 中断驱动方式　　(*c*) DMA方式

5.2.2 中断驱动 I/O 控制方式

在 I/O 设备输入每个数据的过程中，由于无须 CPU 干预，因而可使 CPU 与 I/O 设备并行工作。仅当输完一个数据时，才需 CPU 花费极短的时间去做些中断处理。可见，这样可使 CPU 和 I/O 设备都处于忙碌状态，从而提高了整个系统的资源利用率及吞吐量。例如，从终端输入一个字符的时间约为 100 ms， 而将字符送入终端缓冲区的时间小于 0.1 ms。 若采用程序 I/O 方式，CPU 约有 99.9 ms 的时间处于忙—等待中。 采用中断驱动方式后，CPU 可利用这 99.9 ms 的时间去做其它事情，而仅用 0.1 ms 的时间来处理由控制器发来的中断请求。 可见，中断驱动方式可以成百倍地提高 CPU 的利用率。

5.2.3 直接存储器访问 DMA I/O 控制方式

1. DMA(Direct Memory Access)控制方式的引入

该方式的特点是：① 数据传输的基本单位是数据块，即在 CPU 与 I/O 设备之间，每次传送至少一个数据块；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。可见，DMA 方式较之中断驱动方式，又是成百倍地减少了 CPU 对 I/O 的干预，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

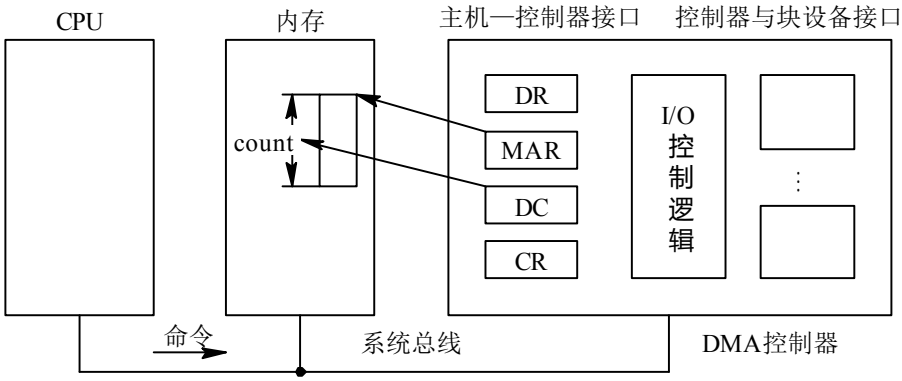


图 5-8 DMA 控制器的组成

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：

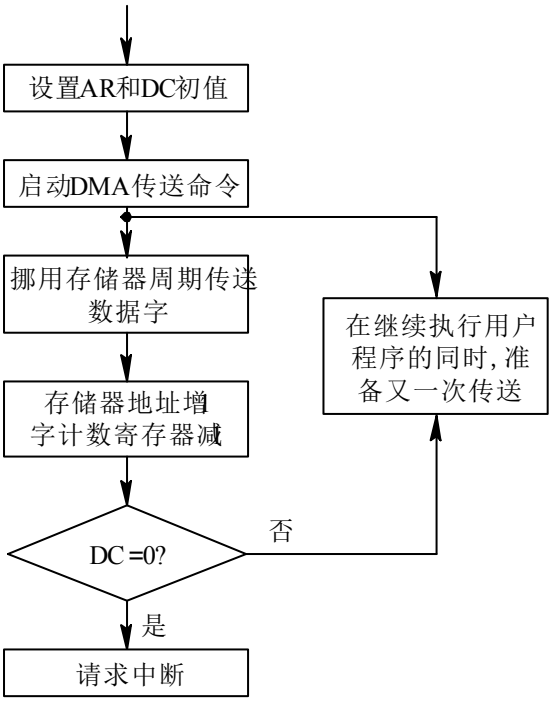
(1) 命令/状态寄存器 CR。用于接收从 CPU 发来的 I/O 命令或有关控制信息， 或设备的状态。

(2) 内存地址寄存器 MAR。在输入时，它存放把数据从设备传送到内存的起始目标地址；在输出时，它存放由内存到设备的内存源地址。

(3) 数据寄存器 DR。用于暂存从设备到内存，或从内存到设备的数据。

(4) 数据计数器 DC。 存放本次 CPU 要读或写的字(节)数。

3. DMA 工作过程



5.2.4　I/O 通道控制方式

1. I/O 通道控制方式的引入

I/O 通道方式是 DMA 方式的发展，它可进一步减少 CPU 的干预，即把对一个数据块的读(或写)为单位的干预，减少为对一组数据块的读(或写)及有关的控制和管理为单位的干预。同时，又可实现 CPU、通道和 I/O 设备三者的并行操作，从而更有效地提高整个系统的资源利用率。例如，当 CPU 要完成一组相关的读(或写)操作及有关控制时，只需向 I/O 通道发送一条 I/O 指令，以给出其所要执行的通道程序的首址和要访问的 I/O 设备，通道接到该指令后，通过执行通道程序便可完成 CPU 指定的 I/O 任务。

2. 通道程序

(1) 操作码。

(2) 内存地址。

(3) 计数。

(4) 通道程序结束位 P。

(5) 记录结束标志 R。

操作	P	R	计数	内存地址
WRITE	0	0	80	813
WRITE	0	0	140	1034
WRITE	0	1	60	5830
WRITE	0	1	300	2000
WRITE	0	0	250	1850
WRITE	1	1	250	720

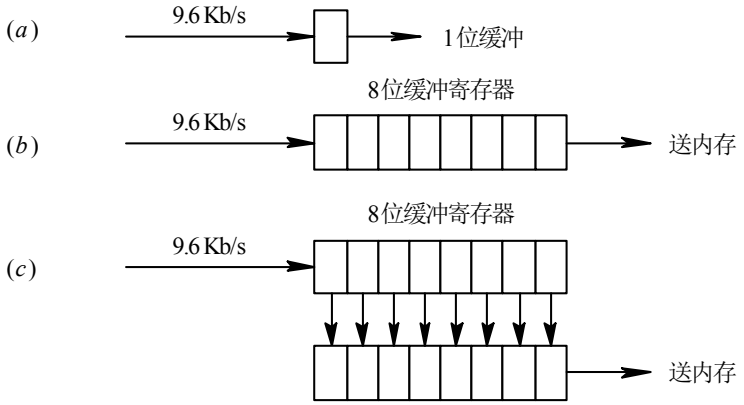
5.3 缓 冲 管 理

5.3.1 缓冲的引入

(1) 缓和 CPU 与 I/O 设备间速度不匹配的矛盾。

(2) 减少对 CPU 的中断频率， 放宽对 CPU 中断响应时间的限制。

(3) 提高 CPU 和 I/O 设备之间的并行性。



5.3.2 单缓冲和双缓冲

1. 单缓冲(Single Buffer) Max(C,T)+M

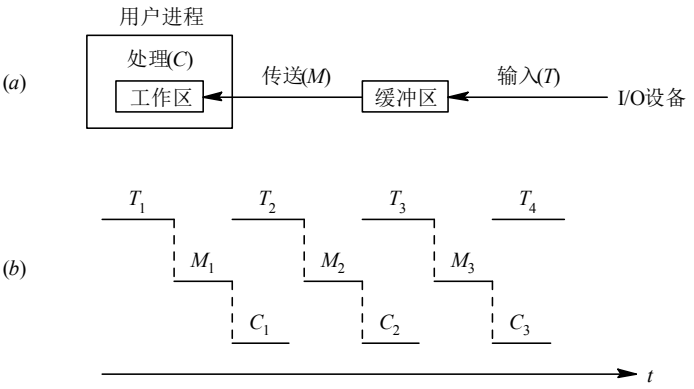


图 5-11 单缓冲工作示意图

2. 双缓冲(Double Buffer) Max(C,T)

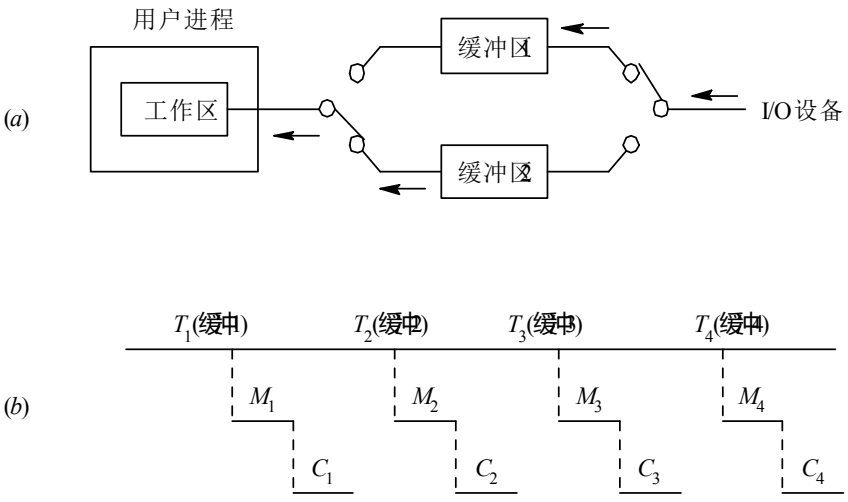
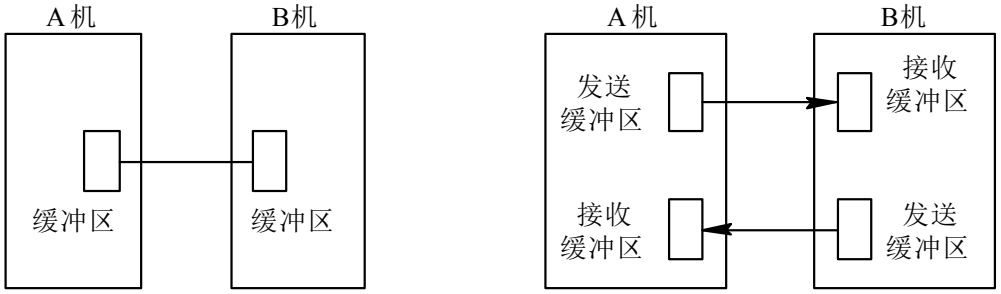


图 5-12 双缓冲工作示意图

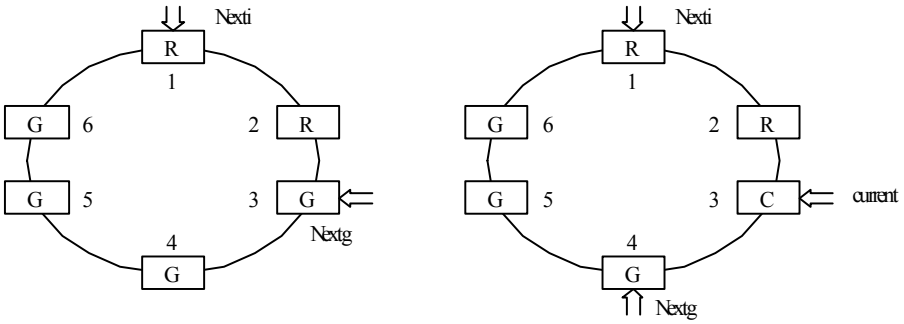


(*a*) 单缓冲　　(*b*) 双缓冲

图 5-13 双机通信时缓冲区的设置

5.3.3 循环缓冲

1. 循环缓冲的组成　多个缓冲区，多个指针



2. 循环缓冲区的使用

(1) Getbuf 过程。

(2) Releasebuf 过程。