# 操作系统复习整理

## 操作系统的结构

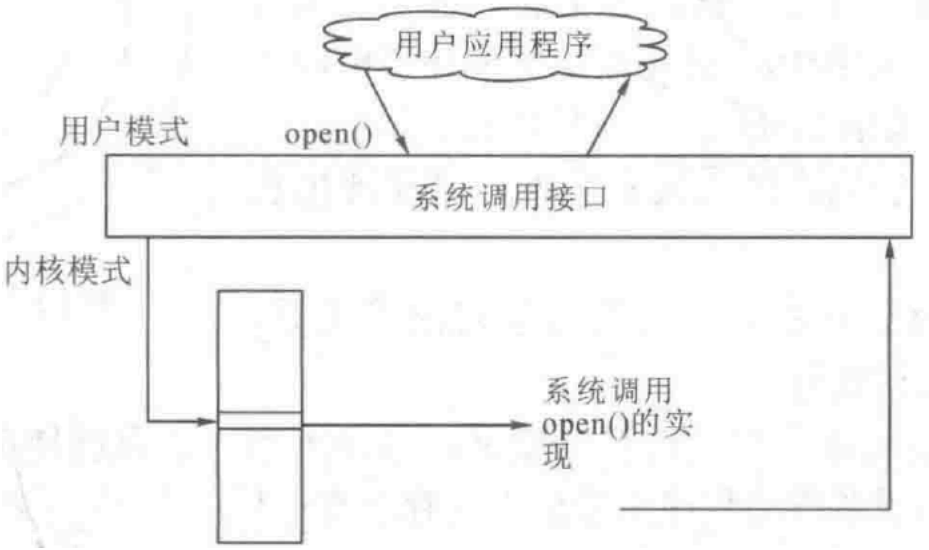
### 操作系统接口

- 命令行界面(Command-Line Interface, CLI)
- 图形用户界面(Graphic-User Interface, GUI)

### 系统调用

内核提供一系列具备预定功能的多内核函数，通过一组称为**系统调用**(System call)的接口呈现给用户。系统调用把应用程序的请求传给内核，调用相应的内核函数完成所需的处理，并将处理结果返回给应用程序。

系统调用的工作过程：运行时支持系统（由编译器直接提供的函数库）提供了系统调用接口 (System-call Interface)，以链接到操作系统的系统调用。系统调用接口截取API 函数的调用，并调用操作系统中的所需系统调用。通常，每个系统调用都有一个相关数字，而系统调用接口会根据这些数字来建立一个索引列表。系统调用接口就可调用操作系统内核中的所需系统调用，并返回系统调用状态与任何返回值。

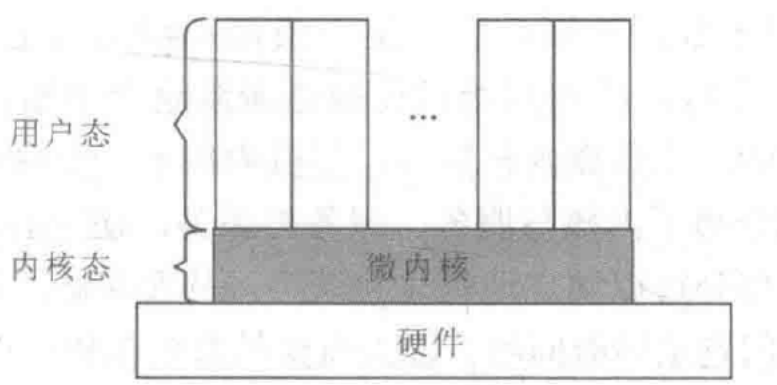



向系统传递参数：

- 通过寄存器传参，适用于参数较少时
- 参数比较多的时候，存在内存中，内存地址通过寄存器传递
- 通过程序压入堆栈，并提供操作系统弹出

### 操作系统结构

- 整体式结构
- 层次式结构
- 微内核结构
  - 大内核系统将操作系统的主要功能模块都作为一个紧密联系的整体运行在内核态，从而为应用提供高性能的系统服务。因为各管理模块之间共享信息，能有效利用相互之间的有效特性，所以具有无可比拟的性能优势。
  - 微内核结构将内核中最基本的功能保留在内核，而将那些不需要在内核态执行的功能移到用户态执行，从而降低了内核的设计复杂性。而那些移出内核的操作系统代码根据分层的原则被划分成若干服务程序，它们的执行相互独立，交互则都借助于微内核进行通信。

- 

    - 优点：解决操作系统的内核代码难以维护的问题，降低了内核的设计复杂性，实现高可靠性。模块中的错误虽然会使模块崩溃，但不会让整个系统死机。应用于实时、工业、航空、军事
    - 缺点：性能问题，频繁在核心态和用户态之间切换，造成操作系统的执行开销偏大
    - **低级存储管理**：把每个虚拟页映射到一个物理页框，存储管理的大部分功能，包括保护一个进程的地址空间免于另一个进程的干涉
    - **进程间通信**(IPC)：进程之间或线程之间进行通信的基本形式是消息，基于进程间相关联的端口
    - **I/O**
    - **中断管理**：识别但不处理
- 模块化结构
    - 利用面向对象编程技术来生成模块化的内核
    - Linux
    - 将内核镜像的尺寸保持在最小，并具有最大的灵活性；便于检验新的内核代码，而不需重新编译内核并重新引导。

## 虚拟机

每一台虚拟机都是在Hypervisor的基础上建立起来的。Hypervisor是一种运行在物理服务器和操作系统之间的中间软件层，可允许多个操作系统和应用共享一套基础物理硬件，因此也可以看作是虚拟环境中的“元”操作系统：它可以协调访问服务器上的所有物理设备和虚拟机，也叫作虚拟机监视器(Virtual Machine Monitor，VMM)。Hypervisor是所有虚拟化技术的核心。非中断地支持多工作负载迁移的能力是Hypervisor的基本功能。当服务器启动并执行Hypervisor时，它会给每一台虚拟机分配适量的内存、CPU、网络和磁盘，并加载所有虚拟机的客户操作系统。

# 进程与线程

## 进程基础

### 程序的顺序执行和并发执行

- **顺序执行**：系统所有资源被这个程序独占，具有顺序性、封闭性、可再现性的特点
- **并发执行**：共享和竞争资源

### 进程的定义与特征

**进程**是并发执行的程序在一个数据集上的执行过程，是系统分配资源和调度的基本单位。

- 动态性。进程是程序在并发系统内的一次执行，一个进程有一个产生到消失的生命周期。
- 并发性。正是为了描述程序在并发系统内执行的动态特性才引入了进程，没有并发就没有进程。
- 独立性。每个进程的实体是一个能独立运行、独立获得资源、独立调度的基本单位。
- 异步性。进程按各自独立的、不可预知的速度向前推进，并按异步方式运行。

进程与程序的**区别**在于：

- 进程是一个动态的实体，有生命期，由创建而产生，由调度而执行，由撤销而消亡：而程序是一个静态的实体，只是一组指令的有序集合。如果把程序比作一个剧本，那么进程就是这个剧本的一次演出。
- 进程具有并发性，各进程的执行是独立的，执行速度是异步的；而程序是不能并发执行的。
- 进程具有独立性，是一个能独立运行的基本单位。没有建立进程的程序是不能作为独立的单位参与运行的。

**进程控制块PCB**记录了操作系统所需的、用于描述并控制进程运行所需的所有信息，包括进程的**描述信息、控制信息和资源信息。**操作系统通过进程控制块来感知进程的存在，掌握其所处的状态以达到管理和控制进程的目的。
**进程标识符**用于唯一地识别一个进程。

进程是由一段*可执行的程序*、*数据*（程序数据、用户栈以及可修改的程序）、*系统栈*（用于保存参数、过程调用地址以及系统调用地址）以及*进程控制块PCB*组成的集合，该集合也可称为**进程映像**。


图 3-1　进程控制块

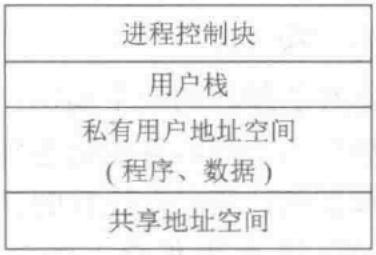
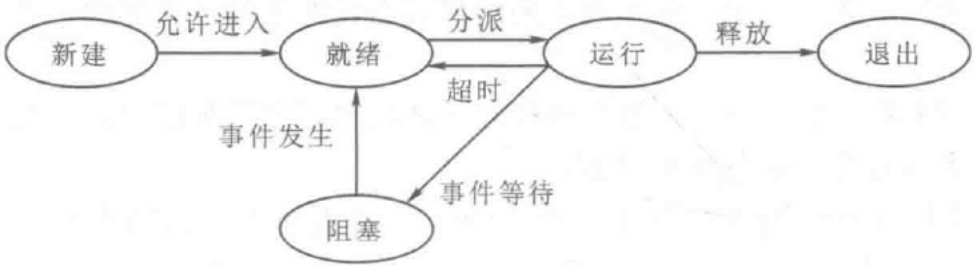



图 3-2　进程映像

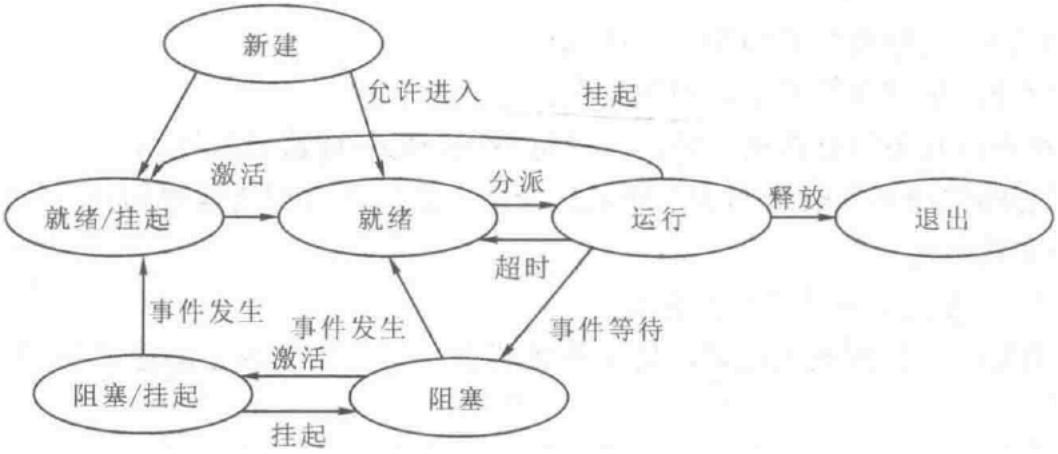
通过对进程控制块PCB的存取，操作系统为进程分配资源，进行调度。当进程被调度执行时，系统按PCB中程序计数器指出的地址执行程序；当进程被中断时，操作系统会将程序计数器和处理器寄存器（上下文数据）保存于PCB中，并修改进程状态（阻塞或就绪）；通过修改PCB中的状态为运行态，并将PCB中的程序计数器和进程上下文数据加载到处理器寄存器中，使进程恢复运行；当进程执行结束后，通过释放PCB来释放进程占有的各种资源。

## 进程的状态

- **初始态**：进程刚被创建时，处于初始状态，系统已经为它创建了PCB，但还未加载到内存中。
- **终止态**：进程执行结束或被取消，但其PCB等信息还未释放。
- **运行态**：进程占有处理器，正在执行。
- **阻塞态**：进程等待某个时间的发生。
- **就绪态**：具备执行的所有条件，等待系统分配处理器后执行。进程在它的生命期里的状态会不断发生变化。




当内存中的进程都在等待I/O，同时内存容量又不足以容纳更多新进程时，处理器处于空闲状态。解决该问题的办法是交换，即将内存中某个进程的一部分或全部换出到磁盘，使它处于“**挂起**”状态，然后接受一个新进程的请求，将其装入内存运行。



# 进程控制

- **系统态/控制态/内核态**：具有对处理器以及所有指令、寄存器和内存的控制能力
- **用户态**：只能执行规定的指令，访问指定的寄存器和存储区

系统进程运行在系统态，用户进程运行在用户态。

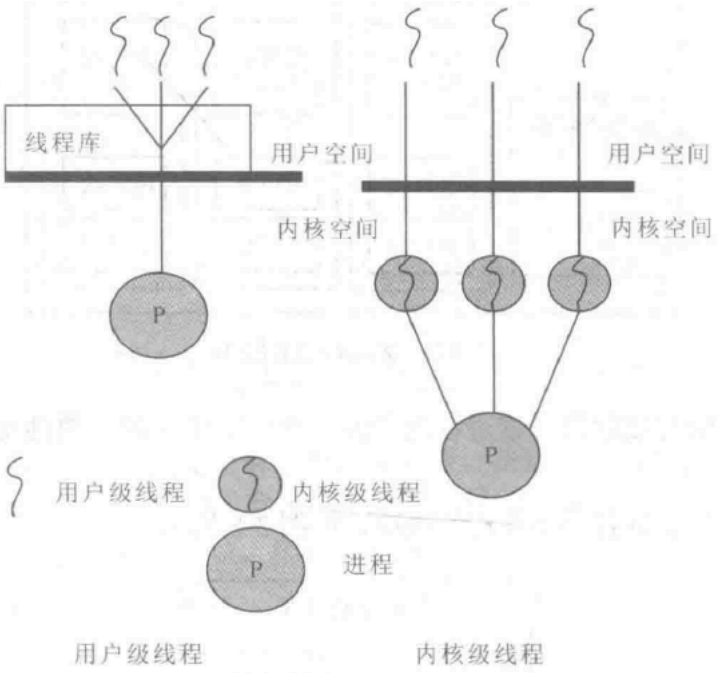
## 进程的创建与终止

- **进程创建**
  i. 给新进程分配一个唯一的进程标识符(**pid**)，并申请一个空白的PCB（PCB是有限的）。若PCB申请失败，则创建失败。
  ii. 给进程分配空间分配包括程序、数据、用户栈等。
  iii. 将新进程插入就绪队列和进程隶属关系族群中。
  iv. 创建或扩充其他数据结构，如为进程创建记账文件等。
- **进程撤销**
  i. 从PCB队列中检索到被终止进程的PCB，并修改其状态。
  ii. 若该进程有子孙进程，则应将其所有子孙进程终止。
  iii. 释放该进程的所有资源。
  iv. 将该进程的PCB从系统PCB队列中删除。
- **进程阻塞**
  i. 停止运行，修改PCB中状态为“阻塞”。
  ii. 将进程插入阻塞队列。如果系统设置了多个不同原因的阻塞队列，则将进程插入到具有相同原因的阻塞队列中。
  iii. 系统进行调度并完成进程切换。保留被阻塞进程的处理器状态于PCB中，转操作系统调度程序，选择一个就绪进程将处理器分配给它，并按新进程PCB中的处理器状态设置CPU环境，完成进程切换。
- **进程唤醒**
  i. 把被阻塞的进程从等待该事件的阻塞队列中移出，唤醒原语将进程从外存调入内存。
  ii. 将其PCB中的现行状态由阻塞改为就绪，然后再将该PCB插入到就绪队列中。
  iii. 假如采用的是抢占调度策略，则每当有新进程进入就绪队列时，检查是否要进行重新调度，即比较被唤醒进程与当前进程的优先级，决定处理机的归属。

# 线程

线程又被称为轻型进程，在引入了线程的系统中，线程是一个可以独立行和调基本单位，但不再是拥有资源的独立单位。它只拥有少量运行中必不可少的资源（如程序计数器、一组寄存器和栈），它可与同属一个进程的线程*共享*该进程拥有的所有资源。

- 进程和线程的区别和联系
  - 在引入了线程的系统中，线程是作为调度和分派的基本单位，而进程是作为资源分配的基本单位。同一进程中的线程切换不会引起进程切换。
  - 在引入了线程的系统中，不仅进程之间可以并发执行，而且一个进程的多个线程之间也可以并发执行，因此可以更有效地使用系统资源，具有更好的并发性。
  - 进程是拥有资源的独立单位，而线程只拥有少量的资源，但它可以访问其隶属进程的资源。因此，线程创建、终止和切换时的系统开销比进程小。
  - 由于同一进程的线程共享内存和文件，因此在同一个地址空间里，它们之间的通信无需调用内核，从而提高了通信的效率。
- 用户级线程与内核级线程
  - **内核级线程**：线程管理的所有工作都是由内核完成的。用户通过操作系统给应用程序提供的应用程序编程接口API来进行进程管理。
    - 可以将同一进程的多个线程调度到不同的处理器上并行执行。如果一个进程中的一个线程被阻塞，内核可以调度同一进程中的多个线程运行。
    - 内核程序本身也可以使用多线程，从而可以提高操作系统内核程序执行的效率。
    - 当把控制从一个线程转到同一进程的另一个线程时，需要内核参与，处理器状态首先从用户态转到内核态，系统调用结束后，再切换回用户态。
  - **用户级线程**：线程管理的所有工作都是由应用程序完成的，内核意识不到线程的存在，内核以进程为单位进行调度。操作系统提供给用户一个线程库对线程进行操作。
    - 优点
      - 由于所有线程管理数据都在进程的用户地址空间中，所以线程切换不需要内核参与，节省了模式切换的开销。
      - 线程的调度算法和调度过程都由用户自行选择，与操作系统内核无关。因为操作系统调度的单位是进程，在进程内部，用户可根据需要设置线程调度算法。
      - 用户级线程可以在任何操作系统中运行，不需对底层内核进行修改。因为线程库是一组应用程序级别函数。
    - 缺点
      - 当一个用户级线程因调用一个系统调用而阻塞时，其所属的进程中的所有线程都会被阻塞。
      - 一个多线程应用程序无法使用多处理器技术，因为内核只为进程分配一个处理器，所以一个进程中只有一个线程能够执行，一个进程中的多个线程无法并行执行。




# 进程同步

## 进程的互斥

**临界资源**：在某段时间内只允许一个进程使用的资源
**临界区**：使用临界资源的部分程序
访问临界资源的进程可描述为

```
do {
    进入区
        临界区
    退出区
        剩余区
} while(TRUE)
```

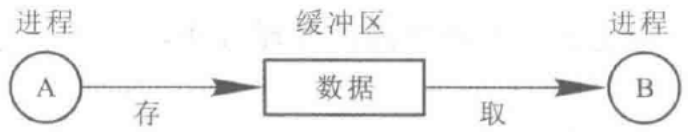
- 系统需满足：
  - 互斥：一次最多一个进程能够进入临界区，当有进程在临界区执行时，其他进程若想要进入临界区，则需要等待。
  - 有限等待：不能让一个进程无限制地在临界区内执行，即任意进入临界区的进程必须在有限时间内退出临界区。
  - 空闲让进：如果某进程退出临界区，而有其他进程正在等待进入临界区时，应当让这个程序进入。
- 使用**硬件**实现互斥
  - 中断禁用
  - 专用机器指令
- 使用**信号量**实现互斥
  - 基本原理：两个或多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号。
  - 当信号量s.value的值为非负整数时，表示某类可用资源的数目；当信号量s.value的值为负整数时，则s.value的绝对值等于由于请求该资源而被阻塞的进程的总数。

```
struct semaphore {
    int value;
    struct QUEUETYPE *queue;
}

void wait(semaphore s)
{
    s.value = s.value - 1;
    if (s.value < 0)
        block(s.queue); // 将进程阻塞，并将其投入等待队列s.queue
}

void signal(semaphore s)
{
    s.value = s.value + 1;
    if (s.value <= 0)
        wakeupWait(s.queue); // 唤醒被阻塞进程，并将其从等待队列s.queue取出，投入就绪队列
}
```

## 进程的同步

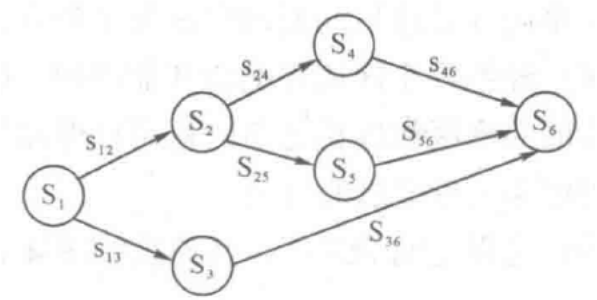



```
Semaphore Bufempty, Buffull;
Bufempty.value = n; Buffull.value = 0;

void A()
{
    wait(Bufempty); //按照FIFO方式选择一个空闲缓冲区
    save(data);
    signal(Buffull);
}

void B()
{
    wait(Buffull); //按照FIFO方式选择一个装满数据的缓冲区
    retrieve(data);
    signal(Bufempty);
}

void main()
{
    parbegin(A, B);
}
```



```
Semaphore S_12, S_13, S_24, S_25, S_36, S_46, S_56;
S_12 = S_13 = S_24 = S_25 = S_36 = S_46 = S_56 = 0;

void S_1(){ S_1 execute; signal(S_12); signal(S_13); }
void S_2(){ wait(S_12); S_2 execute; signal(S_24); signal(S_25); }
void S_3(){ wait(S_13); S_3 execute; signal(S_36);}
void S_4(){ wait(S_24); S_4 execute; signal(S_46); }
void S_5(){ wait(S_25); S_5 execute; signal(S_56); }
void S_6(){ wait(S_36); wait(S_46); wait(S_56); S_6 execute; }
```

## 进程之间的通信

- 共享内存
  - 注意多个进程之间对一个给定存储区访问的互斥，用户程序自己解决（信号量）
- 管道通信
  - 互斥。当一个进程正在对管道进行读或写操作时，另一个进程必须等待。
  - 同步。管道的大小是有限的，所以当管道满时，写进程必须等待，直到读进程把它唤醒为止。同理，当管道没有数据时，读进程也必须等待，直到写进程将数据写入管道后，读进程才被唤醒。
  - 确认对方是否存在。只有确认对方存在时，才能进行通信。
- 消息传递通信
  - 消息队列

## 管程

把分散在各进程中的临界区集中起来进行管理，防止进程有意或无意的违法同步操作。
一个管程定义了一个数据结构和在该数据结构上能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。
管程在结构上由以下三部分组成：

- 管程所管理的共享数据结构。这些数据结构是对应临界资源的抽象。
- 建立在该数据结构上的一组操作。
- 对上述数据结构进行初始化的语句。

# 死锁

## 死锁产生的必要条件

- **互斥条件**。互斥条件指进程的共享资源必须保持使用的互斥性，即任何一个时刻只能分配给一个进程使用。互斥条件是形成死锁最根本的原因，因为如果资源不要求排它性地使用，那么一定不会造成请求资源而无法满足的局面。
- **占有且等待条件**。一个进程占有了某些资源之后又要申请新的资源而得不到满足时，处于等待资源的状态，且不释放已经占用的资源。
- **不可剥夺条件**。任何进程不能抢夺另一个进程所占用的资源，即已经被占用的资源只能由占用进程自己来释放。
- **环路条件**。存在一组进程 P1，P2，⋯，Pn，其中每个进程分别等待另一个进程所占用的资源，形成环路等待条件。

## 死锁的处理方法

### 死锁的预防

- 破坏互斥
- 禁止占用且等待
  - 进程开始前申请全过程需要的全部资源
  - 进程开始前仅获取初期需要的资源，运行过程中释放分配到的、已经使用完毕的资源，再请求新的资源
- 废除“不可抢占”条件
- 破坏循环等待条件

### 死锁的避免

死锁的避免采用的是资源分配拒绝策略，又称为银行家算法。在该方法中，允许进程动态地申请资源，但是系统在进行资源分配之前，先计算资源分配的安全性，如果此次分配不会导致系统进入不安全状态，便将资源分配给进程，否则不予以分配，进程只能等待。
安全状态就是指至少存在一个安全序列<P1, P2, ... , Pn> ，按照这个序列为进程分配所需的资源，直到满足最大需求，使得每个进程都可以顺利完成。若系统不存在这样一个安全序列，则称为系统处于不安全状态。

虽然并非所有不安全状态都是死锁状态，但是系统进入不安全状态后，便可能进入死锁状态：反之，只要系统处于安全状态，系统便可以避免死锁。因此，避免死锁的实质在于：**如何使系统不进入不安全状态。**

### 死锁的检测

- 单体资源
  - 构建资源分配图，不存在环则不存在死锁
- 多体资源
  - 银行家算法

### 死锁的解除

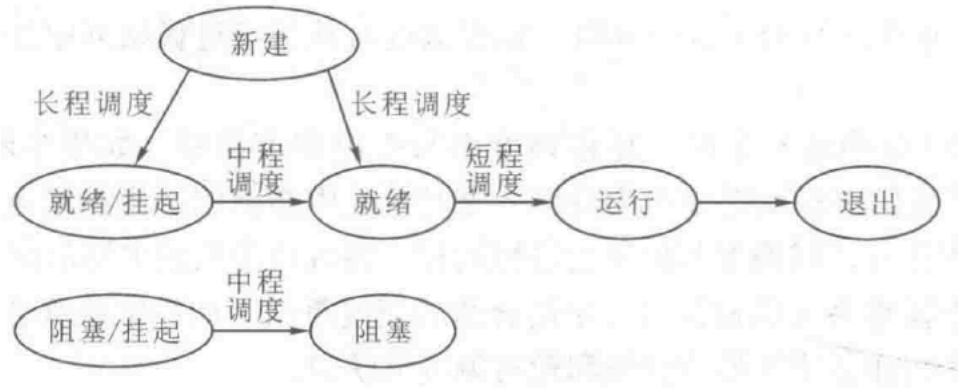
- 撤销所有的死锁进程
- 把每个死锁的进程恢复到前面定义的某个检查点，并重新运行这些进程
- 有选择地撤销死锁过程，直到不存在死锁
- 剥夺资源，直到不存在死锁

# 处理器调度

## 调度策略

- 非抢占式调度策略：一旦进程处于运行状态，它不断运行，直到运行结束或运行至阻塞时进行调度。即当运行进程主动释放CPU时，才执行调度程序。
- 抢占式调度策略：当前正在运行的程序可能被系统中断，转为就绪态。抢占的发生可能是在一个高优先级进程到达时，或在一个中断发生后一个阻塞的进程变为就绪时，或者基于周期性的时间中断（时间片）时。

## 分级调度




- 长程调度（作业调度）
  - 限制系统的并发度
  - 先来先服务FCFS、最短作业优先SJF、最高响应比优先HRRN
- 中程调度
  - 提高内存利用率和系统吞吐量
- 短程调度（进程调度）

## 调度算法

### 先来先服务(First Come First Served, FCFS)调度算法

它按照请求CPU的顺序分配CPU。新就绪的进程依次排在就绪队列尾部，当CPU空闲时，调度排在就绪队列头部的进程。

### 优先级调度算法

动态优先级：与进程占有CPU时间的长短成反比，与在就绪队列中等待的时间成正比

### 最短作业优先(Shortest Job First, SJF)调度算法

难点在于如何直到下次CPU执行的长度，常用于长程调度（作业调度），不常用于进程调度
基于抢占式的SJF：最短剩余时间优先调度算法，通过比较当前就绪队列中进程的剩余时间和新到的进程所需的时间进行调度

如果进程按给定时间到达就绪队列，使用最短剩余时间优先调度算法调度这组进程，则调度顺序如图 6-10 所示。

| 进程 | 到达时间 | 执行时间 |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

图 6-9　一组进程 4

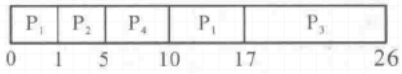



图 6-10　甘特图 5

最初的时候，因为只有进程 $P_1$，所以 $P_1$ 开始执行。

进程 $P_2$ 在时间 1 到达，系统激发调度程序，此时进程 $P_1$ 剩余时间 (7 ms) 大于进程 $P_2$ 需要的时间 (4 ms)，因此进程 $P_1$ 被抢占，进程 $P_2$ 被调度。

进程 $P_3$ 在时间 2 到达，调度进程又开始执行，此时 $P_2$ 进程的剩余时间为 3 ms，新进程 $P_3$ 的执行时间大于 3 ms，故 $P_2$ 继续执行。

进程 $P_4$ 在时间 3 到达，系统再一次激发调度程序，此时 $P_2$ 进程的剩余时间为 2 ms，新进程 $P_4$ 的执行时间大于 2 ms，故 $P_2$ 仍然占用处理器继续执行，直到执行结束。

然后，调度程序在就绪队列中的剩余进程中，按照最短作业优先的原则调度进程执行。

对于这个例子，进程 P1 的等待时间为 (10 − 1)，进程 $P_2$ 的等待时间为 0，$P_3$ 的等待时间为 (17 − 2)，P4 的等待时间为 (5 − 3)。故这组进程的平均等待时间为

$$\frac{((10-1)+0+(17-2)+(5-3))}{4} = 6.5 \text{ ms}$$

如果使用非抢占 SJF 调度算法，那么平均等待时间为 7.75 ms。

### 最高响应比优先(Highest Response Ratio Next, HRRN)调度算法

```
R = (W + S) / S
```

W为等待时间，S为预计的执行时间

如果进程按给定的信息，使用最高响应比优先调度算法调度这组进程，结果如图 6-12 所示。

| 进程 | 到达时间 | 执行时间 |
|---|---|---|
| $P_1$ | 0 | 10 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 2 |
| $P_4$ | 3 | 1 |
| $P_5$ | 4 | 5 |

图 6-11　一组进程 5

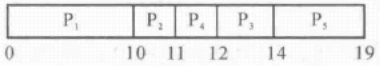



图 6-12　甘特图 6

0 时刻时 $P_1$ 运行，10 时刻时 $P_1$ 运行完，此时 $P_2 \sim P_5$ 的响应比分别为

$P_2$: $\frac{(1+9)}{1}=10$;

$P_3$: $\frac{(2+8)}{2}=5$;

$P_4$: $\frac{(1+7)}{1}=8$;

$P_5$: $\frac{(5+6)}{5}=2.2$，因此进程 $P_2$ 获得处理器运行，直到 11 时刻结束。

$P_2$ 运行完，此时 $P_3 \sim P_5$ 的响应比分别为

$P_3$: $\frac{(2+9)}{2}=5.5$;

$P_4$: $\frac{(1+8)}{1}=9$;

$P_5$: $\frac{(5+7)}{5}=2.4$，因此进程 $P_4$ 获得处理器执行，直到 12 时刻结束。

$P_4$ 运行完，此时 $P_3$、$P_5$ 的响应比分别为

$P_3$: $\frac{(2+10)}{2}=6$;

$\frac{(5+8)}{5}=2.6$，因此执行 $P_3$，最后执行 $P_5$。

进程的平均等待时间为 $\frac{(0+(10-1)+(11-2)+(12-3)+(14-4))}{5}=\frac{37}{5}=7.4\text{ms}$

### 轮转(RR)调度算法

调度程序按就绪队列中进程的顺序依次调度进程运行，每个进程每次运行一个时间片，时间片结束时，正运行的进程对 CPU的拥有权被剥夺，状态由运行转为就绪，重新排在就绪队列末尾等、下一次调度，CPU被分配给下一个就绪进程。
通常应用于分时系统

# 内存管理

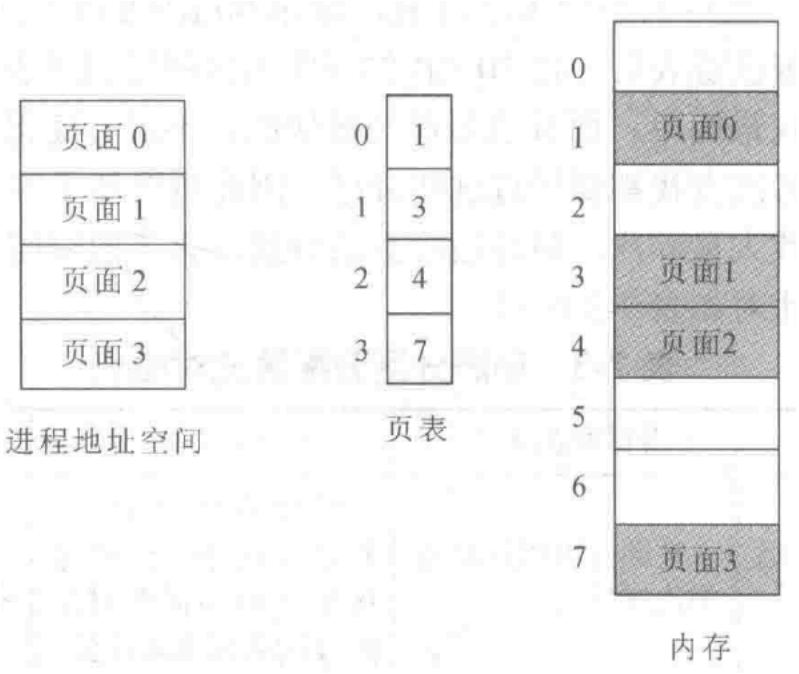
## 内存空间连续分配方案

- 单一连续分配
- 固定分区分配
  - 大小固定
  - 大小不等
- 动态分区分配
  - 首次适应(First Fit)
    - 空闲分区以地址递增的顺序链接，系统分配内存时按顺序查找，找到第一个能满足要求的空闲分区
  - 最佳适应(Best Fit)
    - 空闲分区按容量大小递增形成分区链
  - 最坏适应(Worst Fit)
    - 空闲分区按容量大小递减形成分区链

| 算法 | 算法思想 | 分区排列顺序 | 优点 | |
|---|---|---|---|---|
| 首次适应算法 | 从头到尾找适合的分区 | 空闲分区以地址递增次序排列 | 综合看性能最好。算法开销小，<br>回收分区后一般不需要对空闲分区队列重新排序 | |
| 最佳适应算法 | 优先使用更小的分区，<br>以保留更多大分区 | 空闲分区以容量大小递增次序排列 | 会有更多的大分区被保留下来，<br>能满足大进程需求 | 会产<br>回收分 |
| 最坏适应算法 | 优先使用更大的分区，<br>以防止产生太小的不可用的碎片 | 空闲分区以容量大小递减次序排列 | 算法查找效率高，<br>不用每次都从低地址的小分区开始检索 | 回收分区 |

## 分页存储管理

分页存储管理将进程的逻辑地址空间分成若干个大小相等的页面（虚页），相应地，也把内存的物理地址空间分成若干个页帧（页框），页帧大小与虚页的大小相等。

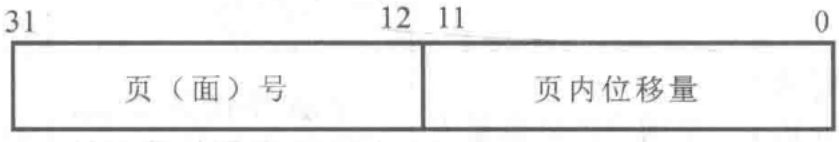



每个进程有一张页表，在进程地址空间内所有页依次在页表中有一页表项，记录了相应页在内存中对应的页帧号。页表的作用是**实现从页号到页帧号的地址映射**。

假设某系统物理内存大小为4GB，页面大小为4KB ，则每个页表项至少应该为多少字节？
由于4GB = 2^32 B，4KB = 2^12 B，因而4GB的内存总共会被分为 2^32 / 2^12 = 2^20个内存块，内存页帧号的范围应该是0 ~ 2^20 - 1，
因此至少要20个二进制位才能表示这么多的内存页帧号，故每个页表项至少要3个字节才够（每个字节8个二进制位）。
各页表项会按顺序存放在内存中，如果该页表在内存中存放的起始地址为X，则M号页面对应的页表项是存放在内存地址，为X + 3 * M。

### 地址变换

- 基本页式地址变换机构
  - 在系统中只设置一个页表寄存器PTR(Page-Table Register)，其中存放**页表在内存的始址**和**页表的长度**。程序未执行时，页表的始址和页表长度存放在本进程的PCB中。当调度程序调度到该进程时，才将这两个数据装入页表寄存器中。
  - 虚地址结构



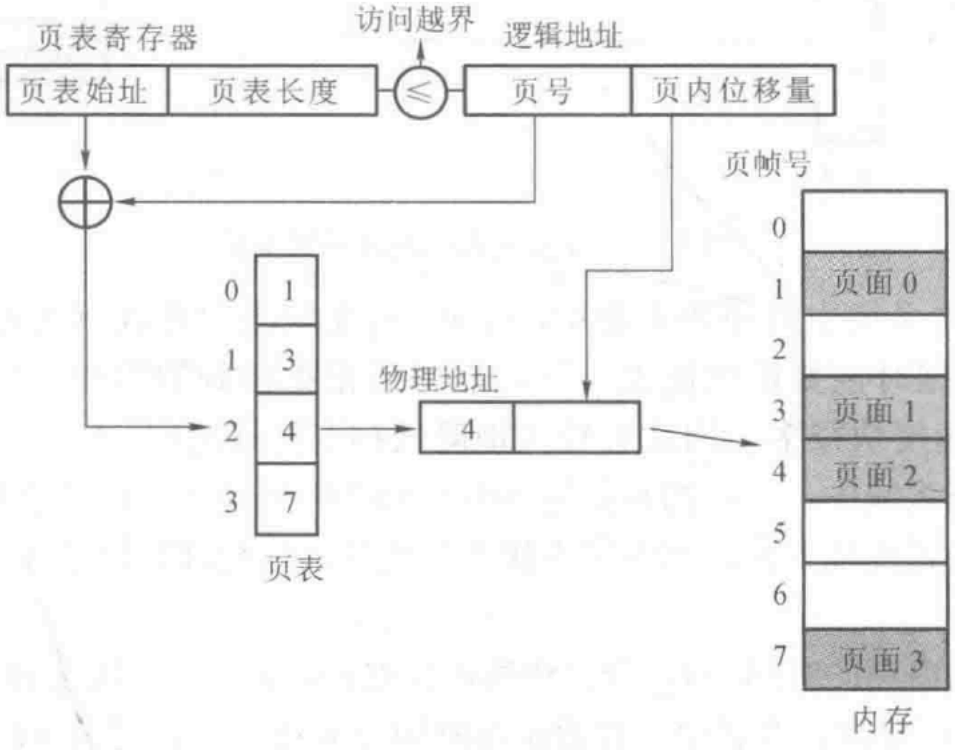



图 7-18　基本页式地址变换结构

- 具有快表的地址变换机构
  - 为了提高地址变换速度，可在地址变换机构中增设一个专用且快速的硬件缓冲器（又称为“联想寄存器”或“快表”），在IBM系统中又取名为TLB，用来存放最近访问的那些页表项。
  - 若在快表中未找到对应的页表项，则还需再访问内存中的页表，同时需要将此页表项写入到快表中（修改快表）。如果联想寄存器已满，则需要淘汰一个表项并进行置换。
  - 假定访问一次内存的时间为t，查找快表所需要的时间为λ，快表的命中率为a，内存的有效访问时间为 EAT，则：
    - 基本页式地址变换机构： `EAT = 2t`
    - 具有快表的地址变换机构: `EAT = (λ + t)a + (λ + 2t)(1 - a) = 2t + λ - at`

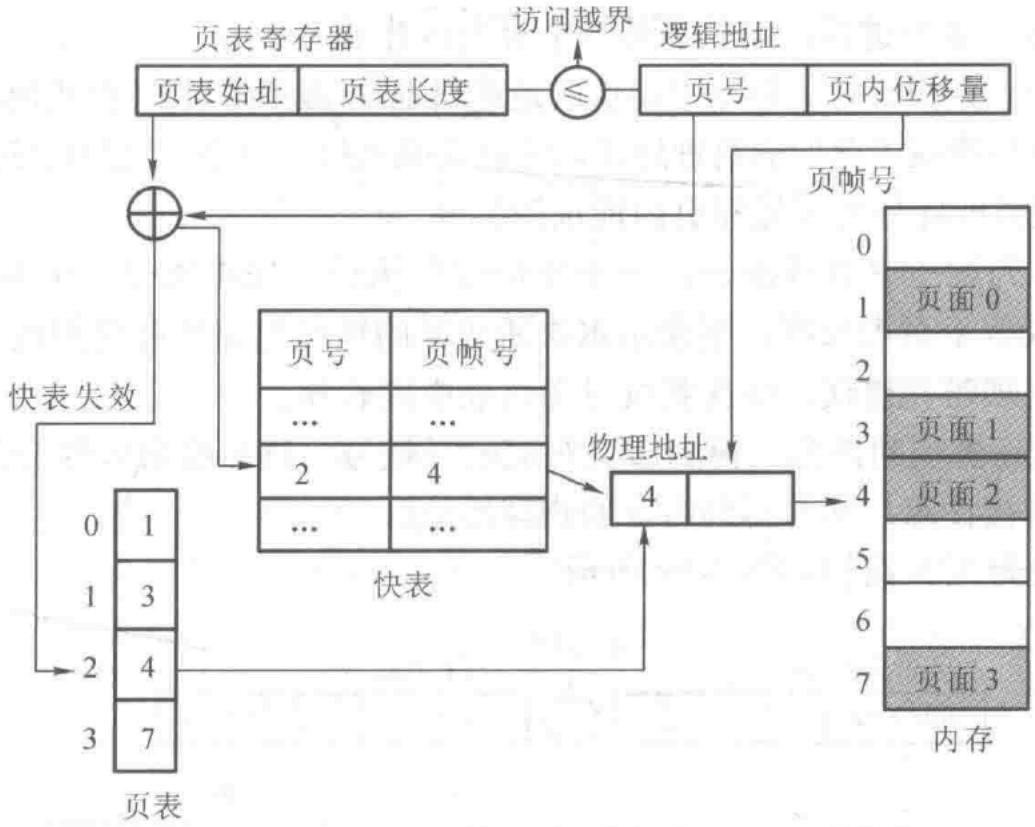



图 7-19　具有快表的地址变换结构

- 两级页表


图 7-20　两级页表的逻辑地址

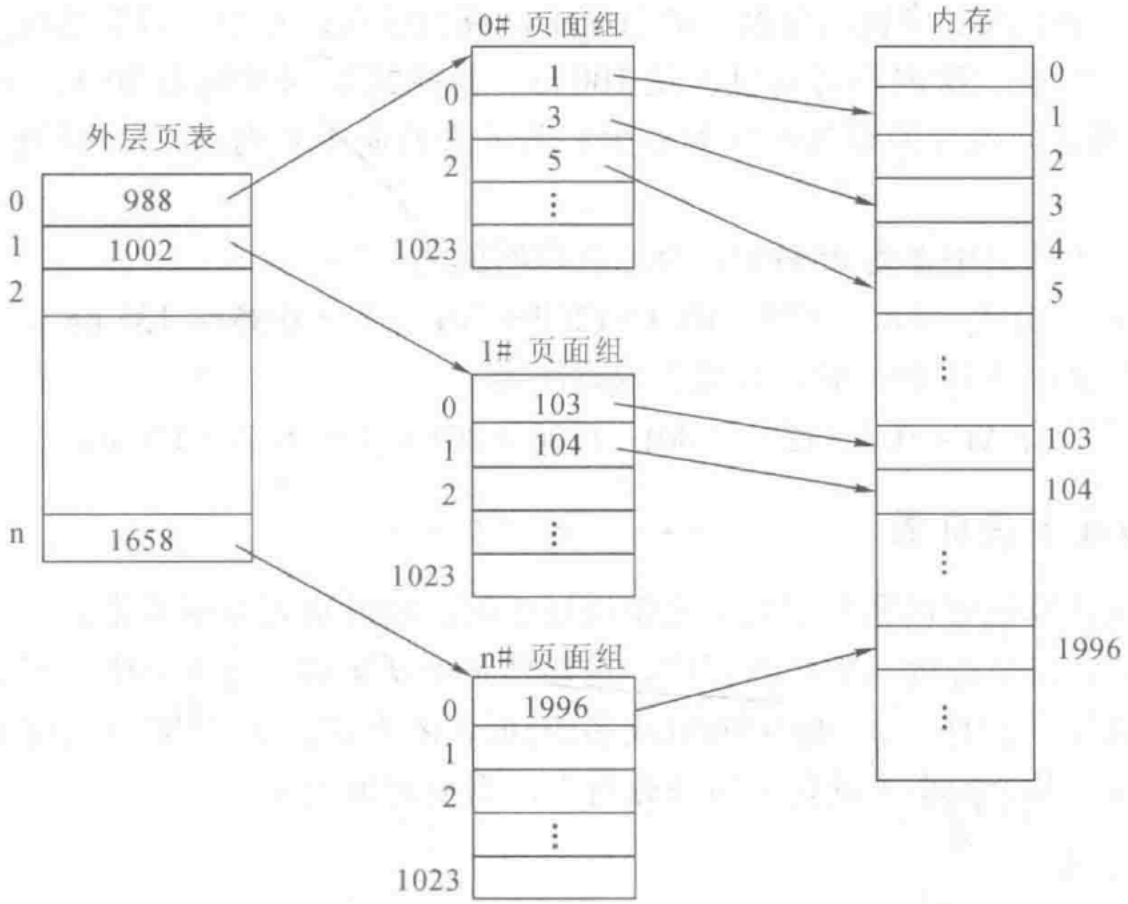



图 7-21　两级页表结构

## 段式存储管理

地址空间由若干个逻辑分段组成，每一个分段是一组逻辑意义完整的信息集合，并且有自己名字（段名），每一段都是以0开始的连续的一维地址空间：整个进程的地址空间是二维的，由**段号**与**段内位移量**（段内地址）组成。
每个进程有一张段表，每个段在表中有一个表项，包括段号、段长和内存起始地址。段表的主要功能是**实现逻辑段到内存空间之间的映射。**


图 7-22　地址空间

| | 分页 | 分段 |
|---|---|---|
| 管理思想 | 页是信息的物理单位，是为了管理主存的方便而划分的：<br>页的大小固定不变，实现了程序的非连续存放 | 段是信息的逻辑单位，是根据用户的需要划分的；<br>段的大小是不固定的，是山其完成的功能决定。因此，<br>段对用户是可见的 |
| 虚地址 | 页式向用户提供的是一维地址空间，<br>页号和页内偏移是机器硬件的功能 | 段式向用户提供的是二维地址空间 |
| 共享与存储访问控制 | 可以实现页面共享，但使用受到诸多限制，访问控制困难 | 便于共享逻辑完整的信息，易于实现存取访问权限控制 |
| 动态链接 | 不支持动态链接 | 支持动态链接 |

## 虚拟存储技术

### 请求分页存储管理

- 请求分页的页表机制

| 页号 | 页帧号 | 状态位 | 访问位 | 修改位 | 辅存地址 |
|---|---|---|---|---|---|

图 7-27　扩展后的页表

- 页帧号：指出该页在主存中的占用的页帧号。
- 状态位：指出该页是否已经调入主存（通常1表示已在内存，0表示未载入内存）。
- 访问位：记录该页是否被访问（通常1表示已访问，0表示未访问）。
- 修改位：表示该页调入主存后是否被修改（通常1表示发生修改，0表示未发生修改）。判断该页是否回写的依据。
- 辅存地址：指示该页在磁盘上的地址。

- 缺页中断机构
  - 在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否在内存中。每当所要访问的页面不在内存时，会发生缺页错误，因此产生一个**缺页中断**，此时操作系统会根据页表中的辅存地址在外存中找到所缺的一页，将其调入内存。
  - 与一般中断的区别
    - 在指令执行期间产生和处理缺页中断信号。
    - 一条指令在执行期间，可能产生多次缺页中断。
    - 缺页中断返回时，执行产生中断的那一条指令：而一般的中断返回时，执行下一条指令。
- 地址变换机构

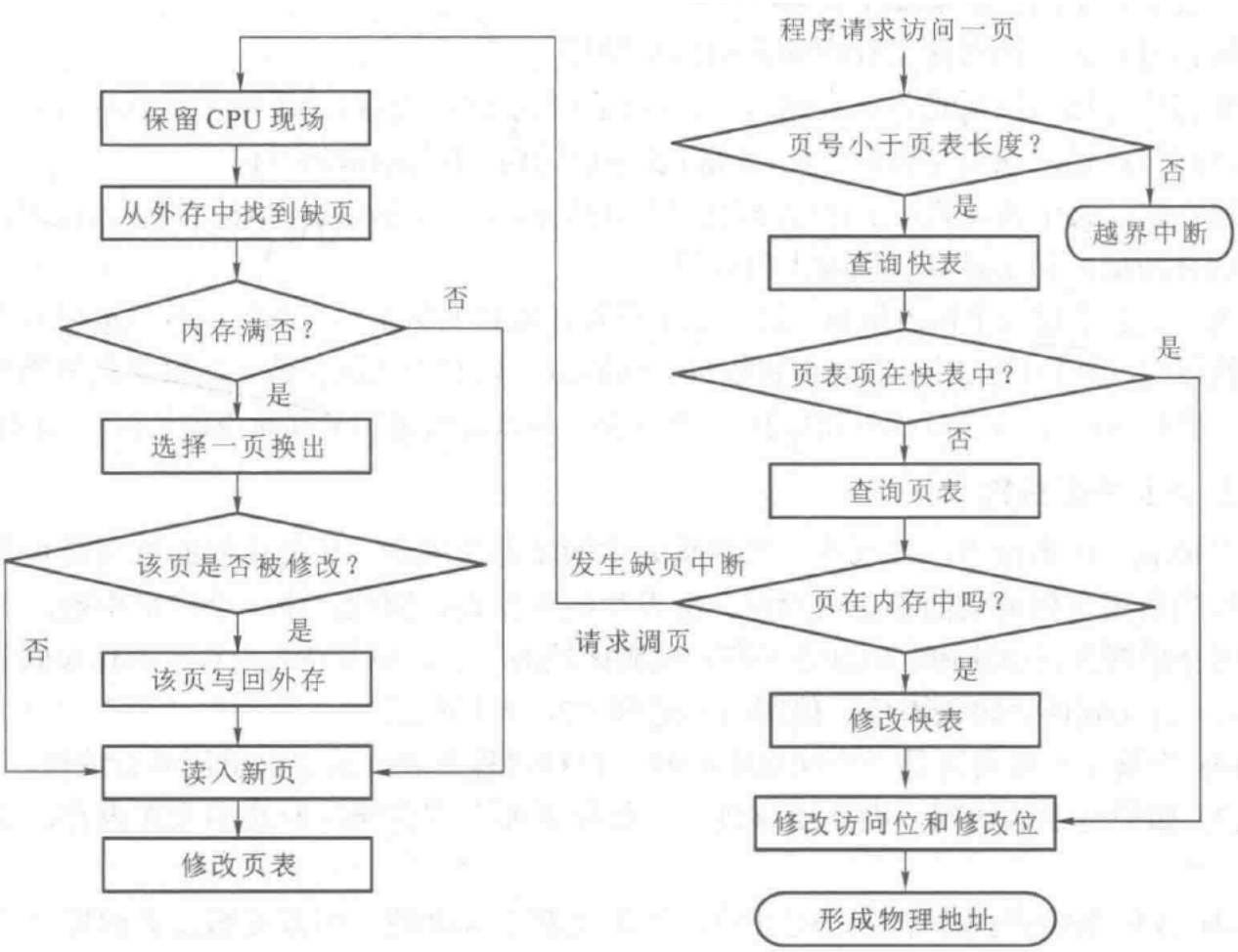



图 7-28　地址变换与缺页处理过程

## 页面置换算法

#### 先进先出(FIFO)页面置换算法

总是选择在主存中停留时间最长的页面进行置换。

#### 最优(OPT)页面置换算法

选择的被淘汰页面是以后永不使用的或在最长时间内不再被访问的。

#### 最久未使用(LRU)页面置换算法

选择在最近一段时间里最久没有使用过的页面。

- 使用计数器
- 使用堆栈，最近使用的页面从堆栈中移除并放到顶部

## 页面缓冲算法

- 空闲页面链表：用于分配给频繁发生缺页的进程，以降低该进程的缺页率。
- 修改页面链表：为了减少已修改页面换出的次数，减少磁盘I/O次数

## 页帧分配算法

- 平均分配法
- 按比例分配法：按进程地址空间所占的页面比例分配

## 页帧分配策略

局部分配策略（同一进程内）/全局分配策略（可运行进程间）

#### 系统抖动

随着进程的增加，CPU的利用率也会增加，但是如果同一时间进程过多，每个进程占用的帧就相应变少，就可能出现进程执行时需要经常性地发生缺页中断、CPU利用率又降低的现象；而这时，操作系统还以为是进程数量太少导致的，还继续加入进程，导致每个进程占用的帧更少、CPU利用率更低的恶性循环，这种现象称为**系统抖动**。

- 原因
  - 分配的页帧数量太小
  - 置换算法选择不当
- 解决方法
  - **工作集策略**：研究一个进程实际使用多少帧。操作系统监视每个进程的工作集，并为它分配大于其工作集的帧数。如果还有足够的额外帧，那么可启动另一进程。如果所有进程工作集大小的总和增加，以致超过内存可用帧的总数，就可能会导致系统抖动，则操作系统会选择一个进程来挂起。该进程的页面被写出（交换），并且其页帧可分配给其他进程。挂起的进程以后可以重启。
  - **缺页率策略**：检测系统中缺页错误的情况。设置所需缺页率的上下限。如果实际缺页率超过上限，则可为进程再分配更多的页帧；如果实际缺页率低于下限，则可从进程中移走页帧。

# 文件系统

## 文件系统层次结构


- **用户接口**：文件系统为用户提供与文件相关的各种系统调用函数，如新建、打开、读写、关闭、删除文件，建立、删除目录等。此层由若干程序模块组成，每一模块对应一条系统调用，用户发出系统调用时，控制即转入相应的陷阱机构处理执行并返回结果。
- **文件目录结构**：文件目录结构的主要功能是管理文件目录，其任务有管理活跃文件目录表、管理读写状态信息表、管理用户进程的打开文件表、管理与组织在存储设备上的文件目录结构、调用下一级存取控制模块。
- **存取控制模块**：文件保护的功能主要由该层软件实现。例如，它把用户的访问要求与文件控制块FCB中指示的访问控制权限进行比较，以确认访问的合法性。
- **逻辑文件系统与文件信息缓冲区**：逻辑文件系统与文件信息缓冲区的主要功能是根据文件的逻辑结构将用户要读写的逻辑记录转换成文件逻辑结构内的逻辑块号。
- **物理文件系统**：物理文件系统的主要功能是把逻辑记录所在的逻辑块号转换成实际的物理地址。
- **辅助分配模块**：辅助分配模块的主要功能是管理辅存空间，即负责分配辅存空闲空间和回收辅存空间。
- **设备管理模块**：设备管理模块的主要功能是分配设备、分配读写用缓冲区、磁盘调度、启动设备、处理设备中断、释放设备读写缓冲区、释放设备等。

## 文件的逻辑结构

- 无结构的流式文件：对文件内的信息不再划分单位，依次的一串字符流构成的文件
- 有结构的记录式文件：用户把文件内的信息按逻辑上独立的含义划分信息单位，每个单位成为一个逻辑记录
  - **堆结构文件**：按生成的先后顺序排列。有利于记录追加，较好地用于穷举查找，不利于按关键字查找。


  - **顺序结构文件**：记录在文件中按照某个规则排序，通常按照某个关键字排序。
  - **散列文件结构**：使用散列函数将记录的关键字值经过计算转化为记录的逻辑号（记录的编号）。

## 文件的读写方式

- **顺序读写**：按文件的逻辑地址顺序读取，主要用于磁带文件，也适用于磁盘上的顺序存储的文件

- **直接读写（随机读写）**：文件指针可以定位到指定逻辑块，无需一块一块地移动指针
- **索引读写**：建立索引表，包含键值与逻辑号。通过搜索索引找到键值对应的逻辑块号，再读出需要的记录信息

# 文件的物理结构与组织

- **连续文件（顺序文件）**：将一个文件中逻辑上连续的信息存放到存储介质依次相邻的物理块上便形成顺序结构
- **链接文件**：一个文件的信息存放在若干不连续的物理块中，各块之间通过指针链接
  - 隐式链接文件
  - 显示链接文件：把用于链接文件各物理块的指针，显式地存放在内存的文件访问表 (FAT)。文件的控制块FCB中存放每一条链的链首指针所对应的盘块号：FAT表项中存入链接指针，即下一个块号。
- **索引文件**：系统为每个文件建立一个索引表，表中每一栏目指出文件信息所在的逻辑块号和与之对应的物理块号。索引表的物理地址则由文件说明信息项（文件目录）给出。

# 目录管理

- **文件控制块**(File Control Block, FCB)：操作系统为管理文件而设置的一组具有固定格式的数据结构，存放了为管理文件所需的所有属性信息（文件属性或元数据）。文件控制块一般应包括文件标志和控制信息、逻辑结构信息、物理结构信息、使用信息、管理信息等。文件控制块的作用就是操作系统和要处理的文件之间相联系的一条纽带，操作系统要依靠FCB中的数据完成对文件的读或写操作。
- **文件目录**：管理文件名和文件物理位置之间的映射关系
- **目录结构**
  - 单级目录结构
    - 优点：简单；能实现按名存取
    - 缺点：查找费时；重名问题；不利于文件共享
  - 两级目录结构：在第一级目录（主文件目录）下为每用户单独建立一个用户文件目录
    - 优点：有利于文件的管理、共享和保护：适用于多用户系统，不同的用户可以命名相同文件名的文件，一定程度上解决了命名冲突的问题
    - 缺点：没有从根本上解决文件数目过大时，同一个用户文件目录下的命名冲突的问题与文件搜索时间较长的问题
  - 多级目录结构（树型目录结构）：每个文件系统有一个根目录，在根目录中可以包含若干子目录和文件，在子目录中不但可以包含文件，而且还可以包含下一级子目录。
    - 优点：用户可以将不同类型和不同功能的文件分类储存，既方便文件管理和查找，还允许不同文件目录中的文件具有相同的文件名
    - Windows、Unix、Linux、DOS

# 空闲空间的管理

- 位示图
- 空闲块列表
- 空闲链表法
  - 空闲块链

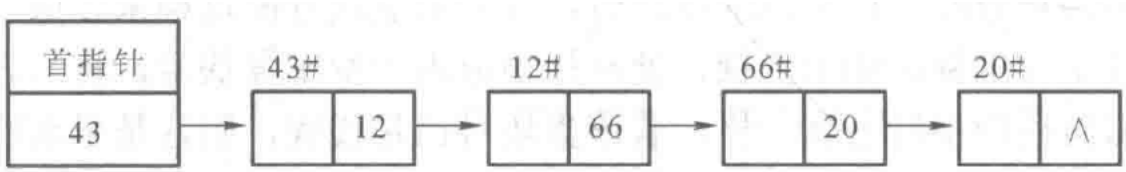


图 8-14　空闲块链

  - 成组链接法(Unix/Linux)：系统中，将空闲块分成若干组，每100 个空闲块为一组，每组的第一个空闲块登记了下一组空闲块的**空闲块总数**和**物理盘块号**。如果一组的第一个空闲块号等于0，则意味着该组是最后一组

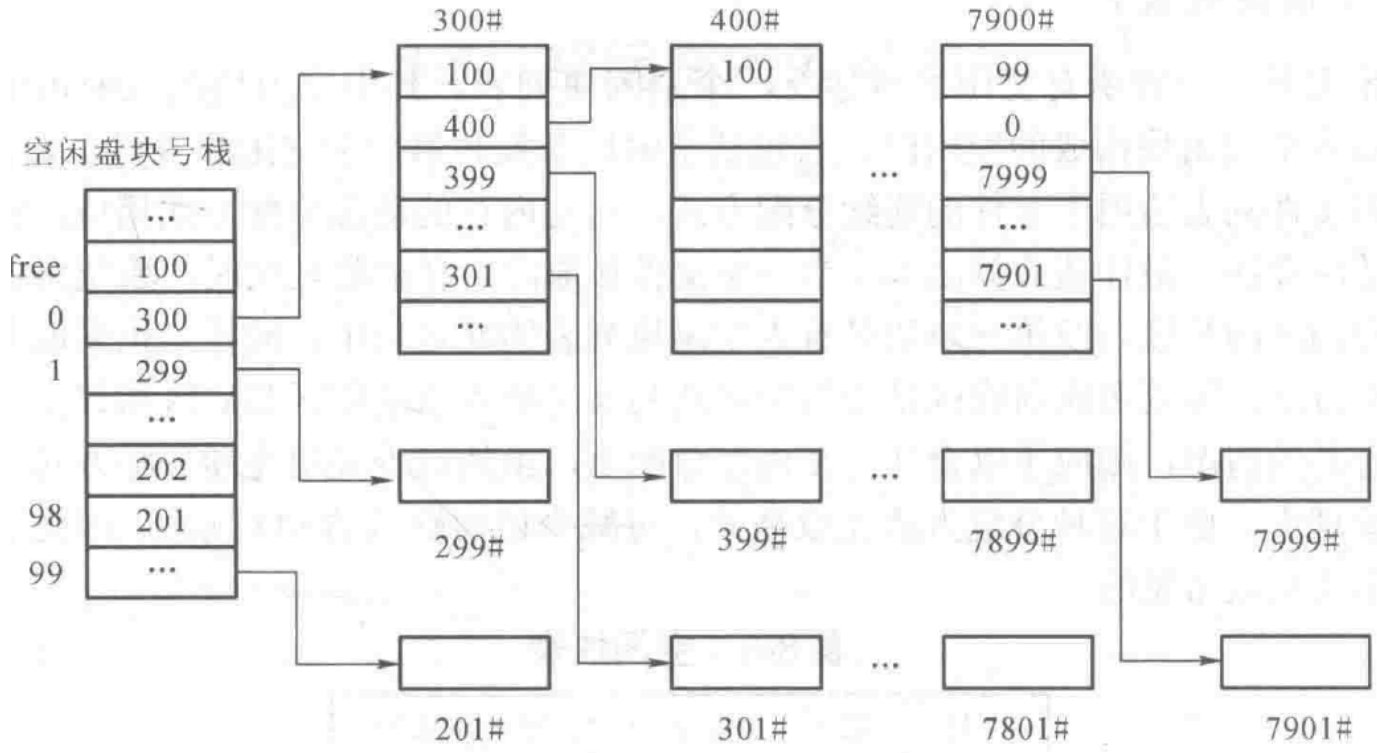



图 8-15　成组链接法

# I/O系统

- I/O端口：I/O控制器中CPU能访问的各类寄存器
  - 独立编址：专用的I/O端口编址，需要有专用的I/O指令
  - 统一编址：内存映射，占用内存，执行时间增加
- 设备控制器的功能
  - 数据缓冲
  - 差错控制
  - 数据交换
  - 标识和报告设备的状态
  - 接收和识别命令
  - 地址识别

## 设备数据传输控制方法

- 轮询方式：CPU不断循环测试I/O设备的状态端口
- 中断控制方式
- DMA(Direct Memory Access)方式：CPU向DMA控制器下达指令，让其处理数据的传送，完毕后再把信息反馈给CPU
- 通道方式

# 缓冲技术

- 缓解设备之间速度差异的矛盾
- 缓解设备之间传输数据大小不一致的矛盾
- 支持应用程序I/O的语义复制

## 缓冲区的分类

- 硬件缓冲：为了协调系统和硬件之间的数据传输速度的差异
  - Cache(高速缓冲寄存器)、硬盘缓存区
- 内存缓冲：内存中规划处一个存放系统输入或输出数据的缓冲区
  - Linux的页缓存

- 软件缓冲

## 缓冲技术的分类

- 单缓冲：一般情况下缓冲区与物理块大小一样，缓冲区移到用户空间后立即请求另一块。适用于数据生产者设备与消费者设备之间的速率相差较大的情况
- 双缓冲与多缓冲：进程读取缓冲区数据的同时另一缓冲区也在存放数据，适用于数据生产者设备与消费者设备之间的速率相差不大的情况
- 缓冲池：系统维护着inq(输入缓冲队列)、outq(输出缓冲队列)、emq(空缓冲队列)和hin、sin、hout、sout四个工作缓冲区

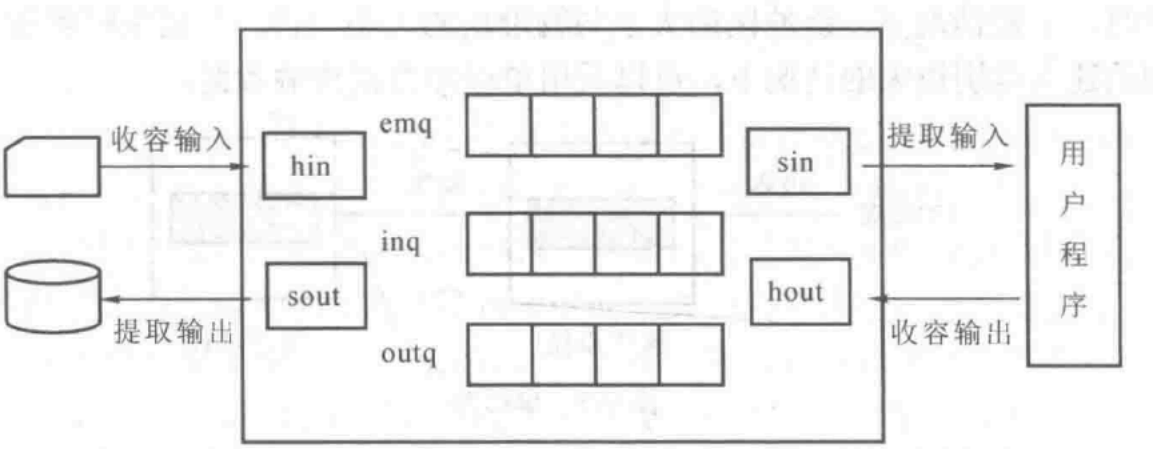


图 9-9　缓冲池

## 虚拟设备

通过虚拟技术将一台独占设备虚拟成多台逻辑设备，供多个用户进程同时使用。

- 设备分配
  - 静态分配（独占设备）
  - 动态分配（共享设备）
  - 虚拟分配（虚拟设备）

# 大容量存储器

```
磁盘的大小 = 磁头数 × 柱面数 × 扇区数 × 每个扇区的大小
磁头数 = 盘面数 × 2
柱面数表示磁盘每面盘片有几条磁道
扇区数表示每条磁道上有几个扇区，每个扇区的大小一般是512B

磁盘读写时间 = 寻道时间 + 旋转延迟 + 数据传输
```

## 磁盘调度

### 先来先服务(First Come First Served, FCFS)调度算法

根据进程请求访问磁盘的先后顺序进行调度

### 最短寻道时间优先(Shortest Seek Time First, SSTF)调度算法

处理距离当前磁头位置的最短寻道时间的请求，可能导致饥饿

### 扫描(SCAN)调度算法

从一端到另一端后反转

### 循环扫描(Circle SCAN, C-SCAN)调度算法

从一端到另一端后，立即返回到开头并且不处理回程的请求

### LOOK调度算法

在SCAN算法基础上只需要移动到最远的请求，不用到末端

# EulerOS操作系统

- 特点
  - 全面支持鲲鹏处理器
  - 极致性能
  - 高可靠/高保障
  - 高安全
  - 支持容器
- 华为 STaaS(Storage as a Service, STaas)解决方案提供自动化的存储服务发放和智能数据管理，支持关键业务云化，助力客户数据中心云化转型。

## 系统功能特性

- 系统管理
  - 系统守护进程systemd
  - 文件系统
  - glibc库
  - RAS技术（可靠性、有效性、适合性）
  - 负载均衡技术
  - 内核热补丁
  - TSX（事务性同步扩展）特性
  - 备电关核
- 网络
  - 网络链路故障检测
  - bbr算法
  - ipvlan接口
  - IPv6
  - 监听队列哈希桶元素
- 内存管理
  - 内存大页vma管理特性
  - per_cpu变量内存
  - Page Cache管理
  - 内存预留
- 处理器调度
  - 高性能自旋锁（MCS锁）
  - 用户态进程禁抢占
  - 高性能定时器
  - 中断均衡irqbalance
- 调测运维
  - 监控报警
  - Kbox
  - Kdump
  - 日志