

Copyright@潘禧辰

contact: [panxichen.pxc@gmail.com](mailto:panxichen.pxc@gmail.com)

资料来源:

- 吴晨涛老师 [wuct@cs.sjtu.edu.cn](mailto:wuct@cs.sjtu.edu.cn)
  - 简书[操作系统](#)专栏
  - repo [SJTU-course-notes](#) by [Galaxies99](#)
- 

## Ch 01 Introduction

架构  
操作系统能做什么  
定义  
计算机系统组成  
中断  
启动流程  
存储结构  
计算机系统结构  
操作系统两个特征  
双模式运行  
进程管理器 chapter3-8  
内存管理器 chapter 9-1  
存储管理器 chapter 12-15  
缓存  
数据搬运流程

## Ch 02 操作系统结构

操作系统服务  
系统调用 system calls  
操作系统设计  
    Arduino  
    FreeBSD  
    UNIX  
    LINUX  
    设计理念  
        分层设计理念  
        微内核  
            结构  
            模块化编程 可以加载的内核模块 LKM loadable kernel modules  
        系统服务  
            链接和加载 该部分来自  
            混合驱动  
            例子

## Ch 03 进程 process

进程概念  
进程在内存中存储  
进程状态  
进程控制块 PCB  
Linux进程表示  
进程的调度  
    调度队列  
    调度程序  
        进程切换 (上下文切换)  
        移动操作系统的多任务  
进程的运行

- 进程创建
  - 父子进程
  - 进程终止
- IPC 进程间通信
  - 共享内存
  - 消息传递
    - 直连 direct communication
    - 间接通讯 indirect communication
  - 同步 synchronous
- 网络进程通信
  - 套接字
  - 远程过程调用 (RPC)
  - 管道
    - 普通管道
    - 命名管道 (FIFO)

## Ch 04 线程threads和并发concurrency

- 线程
  - 多线程服务器
  - 多线程好处
  - 并行的不同类型
  - 线程模型
    - 多对一模型
    - 一对一模型
    - 多对多模型
  - 线程库
    - Pthreads
    - 线程池
    - fork-join
    - 信号处理
    - 线程的取消
    - 调度程序激活

## Ch 05 CPU Scheduling CPU 调度

- 基本概念
  - Dispatcher
  - 衡量指标
- 进程调度算法
  - 先到先服务 (First-Come First-Served, FCFS)
  - 最短作业优先 (Shortest-Job-First, SJF)
    - 例子
      - 若考虑Arrival time
  - 抢占式
  - 优先级调度 (priority-scheduling)
    - 例子
  - 时间片轮转调度 (Round-Robin, RR)
    - 例子
  - 带轮询的优先级调度
  - 多级队列调度
- 线程调度算法
  - 竞争范围
  - Pthreads调度
- 多处理调度
  - 负载均衡 Load balancing
  - Real-Time CPU Scheduling 实时CPU调度
- 不同系统调度算法
  - Linux
  - Windows
    - 优先级
  - 利特尔公式

## Ch 06 synchronization tools

背景

生产者消费者问题

竞态

同步

临界区问题

PeterSon算法

硬件同步

互斥锁

信号量

死锁和饥饿

生产者消费者问题

读者-作者问题

哲学家就餐问题

优先级反转

管程 Monitor

条件变量

## Ch 08 死锁 Deadlocks

资源分配图算法

例子

结论

死锁预防

死锁避免 Avoidance

资源分配图的作用

银行家算法

例子

终止进程来解决死锁

抢占进程来解决死锁

进程同步例题

## Ch 09 主存 main memory

基本概念

地址绑定

逻辑地址空间和物理地址空间

动态加载和动态链接

内存分配

分区

动态存储分配问题

碎片

分段

基本概念

分段的硬件实现

分页

基本概念

分页的硬件支持

例题

分页的效率讨论

分页的安全讨论

共享页

分层分页

哈希页表

倒置页表

Inter 32位与 64 位体系

IA-32架构

IA-32分段

IA-32 分页

X86-64

ARM 架构

虚拟内存

- 基本概念
- 请求调页 demand paging
- 交换空间
- 写时复制
- 页面置换 page replacement
  - 基本概念
- 页面置换
  - FIFO 页面置换算法
  - 最优页面置换 (OPT)
  - LRU页面置换算法 last recently used
    - LRU的具体实现
  - 页面缓冲池
- 页框分配
  - 固定分配
  - 全局分配与局部分配
  - Buddy System 伙伴系统
  - Slab Allocator 块分配器
  - trashing 颠簸

## Ch 11 大容量存储

### 磁盘

- 结构
- 衡量指标
- 控制
- SSD
  - 闪存颗粒
    - SLC
    - MLC
  - SSD与HDD
  - 磁带
- 磁盘连接
- 存储区域网络
- 磁盘调度
  - FCFS(先来先服务)
  - 最短寻道时间优先 (SSTF)
  - SCAN调度(扫描)
  - C-SCAN(循环扫描)
  - LOOK 调度
- 磁盘管理
  - 磁盘格式化
  - 引导块
  - 坏块
- RAID结构
  - RAID
  - RAID 级别

## Ch 12 I/O System

- 概述
- I/O硬件
- 轮询
- 中断
- 直接内存访问 (DMA)
- 应用程序I/O接口
  - 块与字符设备
  - 网络设备
  - 时钟与计时器
  - 非阻塞I/O与异步I/O
- 内核I/O子系统
  - I/O调度
  - 缓冲

缓存  
假脱机和设备预留  
内核数据结构  
I/O请求转换硬件操作

#### Ch 13-15 文件系统

基本概念  
文件  
文件系统的结构  
文件系统内部文件结构  
访问方法  
    顺序访问  
    直接访问：  
        文件系统的安装  
        文件共享  
文件系统结构  
文件系统实现  
    概述  
    分区与安装  
    虚拟文件系统  
    目录的实现  
    分配方法  
    空闲空间管理

#### Ch 18 虚拟机 virtual machines

组成  
虚拟机层次

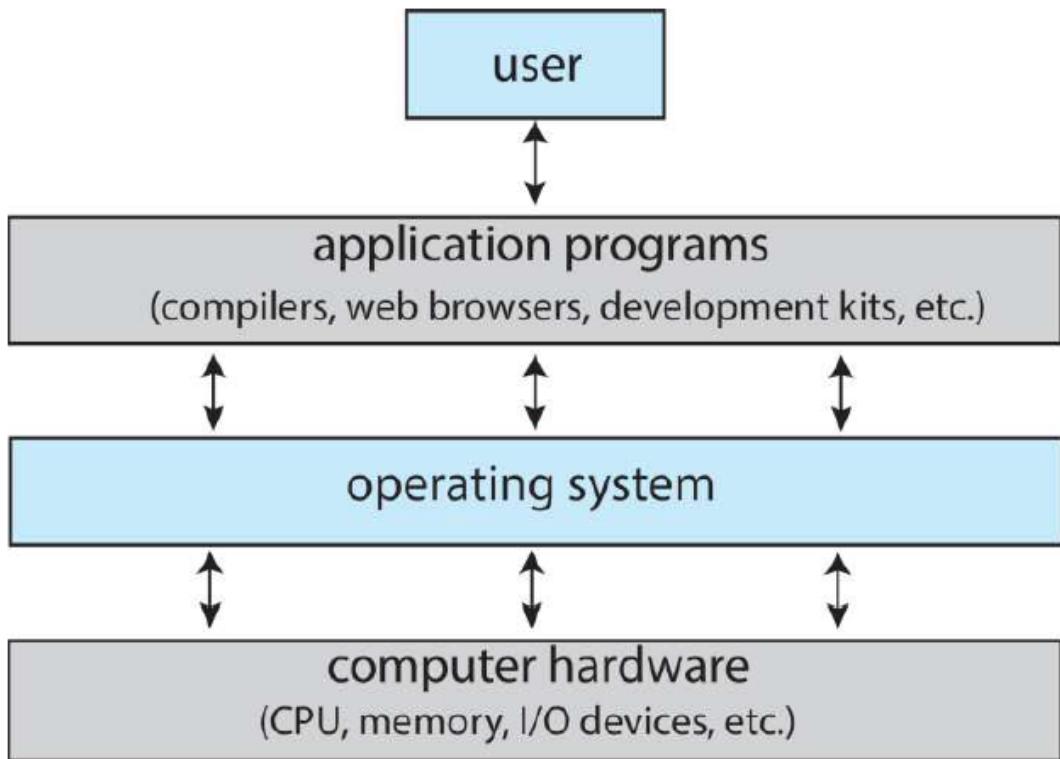
# Ch 01 Introduction

---

## 架构

---

- 硬件
- 操作系统
- 应用程序
- 用户



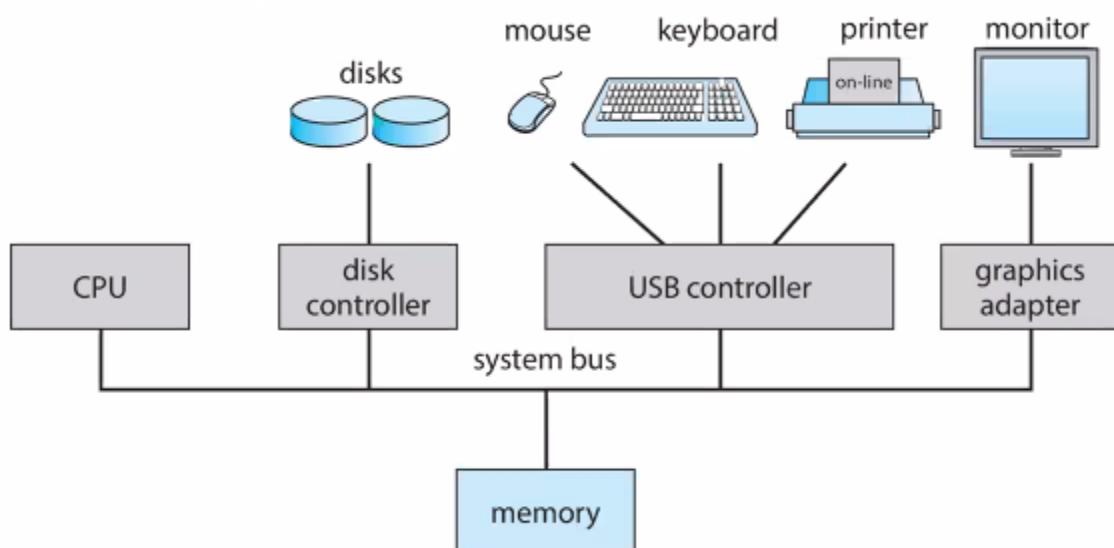
## 操作系统能做什么

- 帮助用户使用
- 资源分配器
- 软硬件的控制程序
- 管理用户程序的执行
- 硬件和控制程序之间的媒介

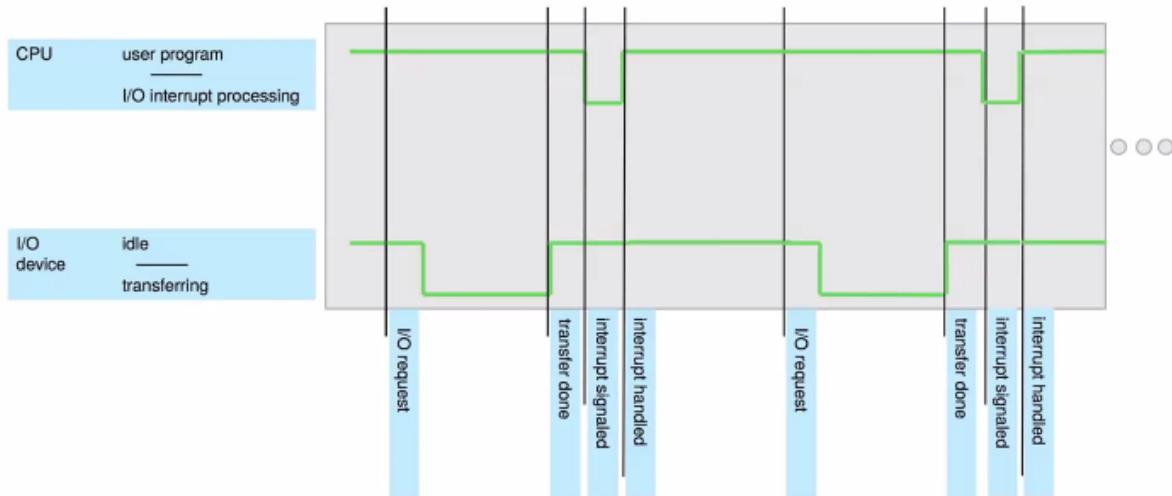
## 定义

- 内核：在计算机上始终运行的一个程序
- 系统程序：与操作系统一同安装，但不是内核的一部分
- 应用程序：与操作系统无关的程序

## 计算机系统组成



# 中断

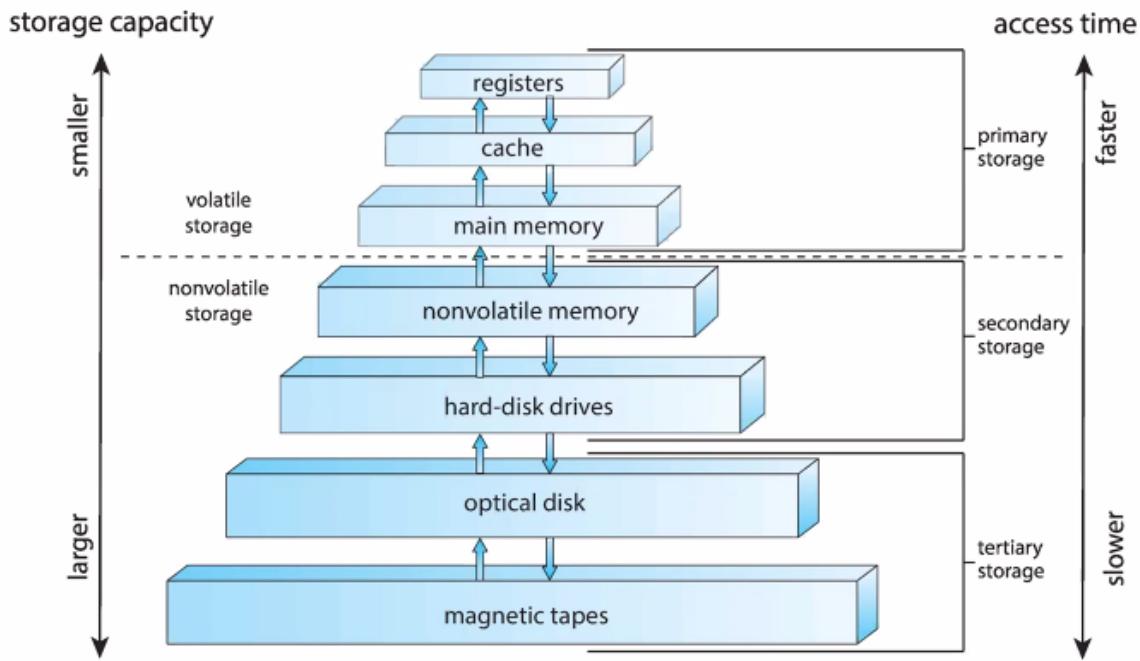


- Kernel **interrupt driven** (hardware and software)
  - Hardware interrupt by one of the devices
  - Software interrupt (**exception** or **trap**):
    - ▶ Software error (e.g., division by zero)
    - ▶ Request for operating system service – **system call**
    - ▶ Other process problems include infinite loop, processes modifying each other or the operating system
- 硬件中断
- 软件中断
  - 异常
  - 陷阱trap
  - 系统调用
  - 软件错误
  - 无限循环

# 启动流程

bootstrap program引导程序在上电或者重新引导时加载，通常存储在ROM或EPROM中，称为固件，可以初始化系统并加载操作系统内核

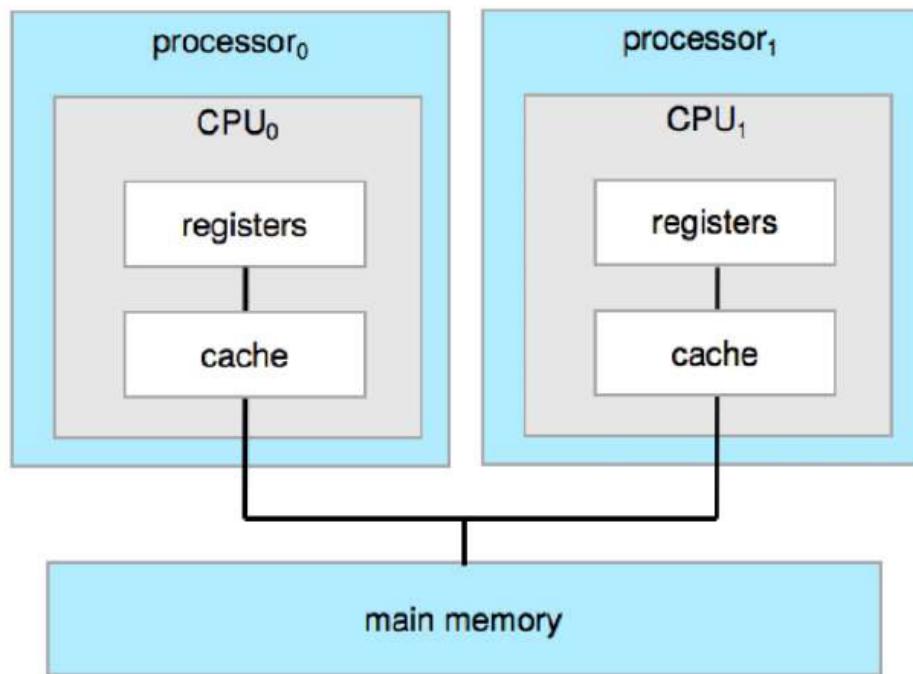
# 存储结构



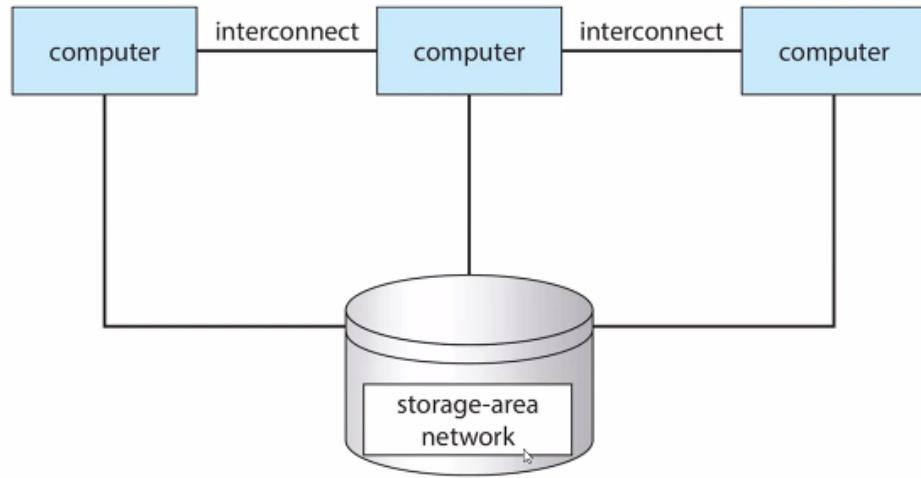
- 磁记录
- 光记录
- 电记录

## 计算机系统结构

- 单处理器
- 多处理器
  - 非对称式
    - 每个处理器被分配单独任务
  - 对称式
    - 每个处理器共同处理相同任务



- 集群系统
  - SAN存储局域网

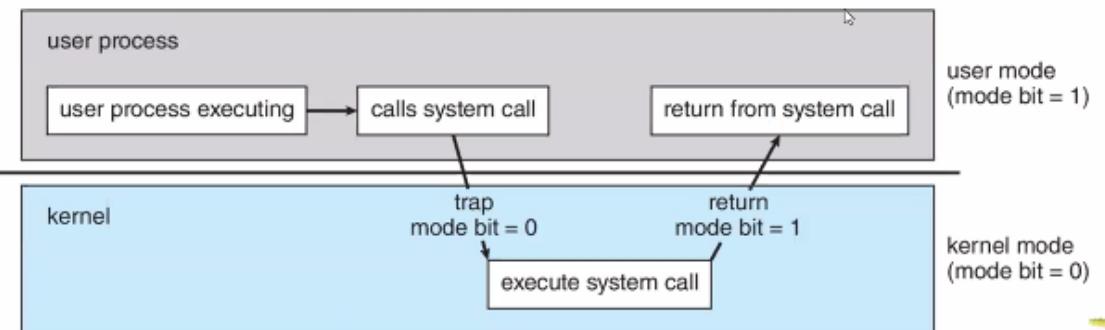


## 操作系统两个特征

- 多程序并行 (batch system)
  - 通过调度使CPU一直有任务执行
  - 提高并行执行效率
- 多任务并行 (分时系统)
  - 将资源分时，任务轮转执行
  - 提高用户体验

## 双模式运行

- 用户态
- 内核态
- 二者并非固定不变，部分用户态应用程序可切换至内核态



- 内核态一些指令是受保护的，只可在内核态执行

## 进程管理器 chapter3-8

- 暂停
- 中止
- 同步
- 通讯

## 内存管理器 chapter 9-1

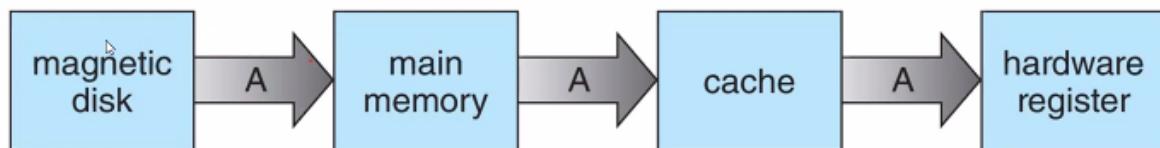
# 存储管理器 chapter 12-15

## 缓存

- 高速存储设备为低速存储设备进行缓存

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

## 数据搬运流程

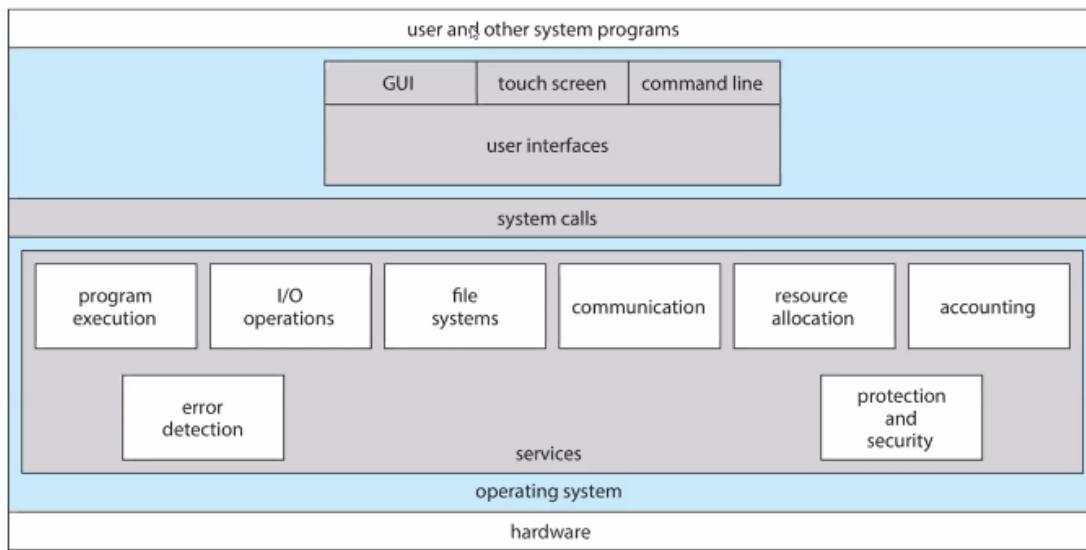


## Ch 02 操作系统结构

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Operating-System Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

## 操作系统服务

- UI
  - 命令行
  - 图形化
  - 触摸
  - 批处理
- 程序执行
- I/O操作
- 文件系统操作
- communication
- 错误检测
- 资源分配
- 日志
- 安全和防护

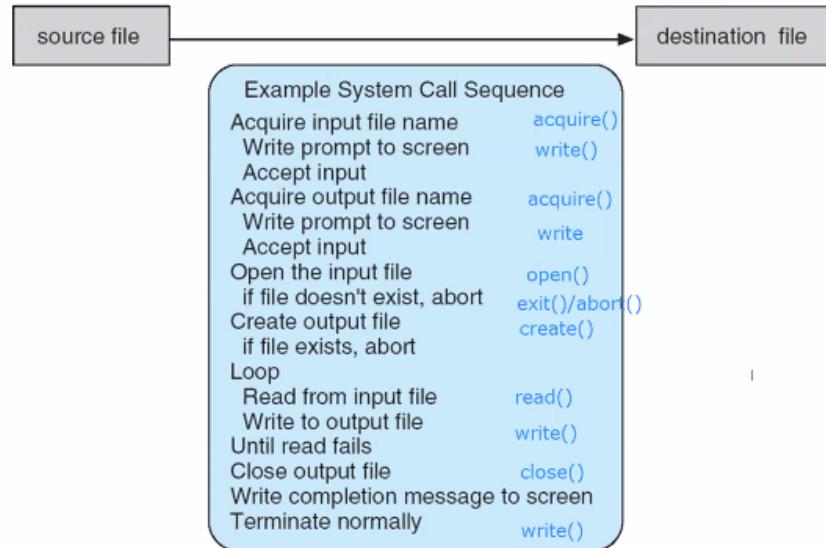


## 系统调用 system calls

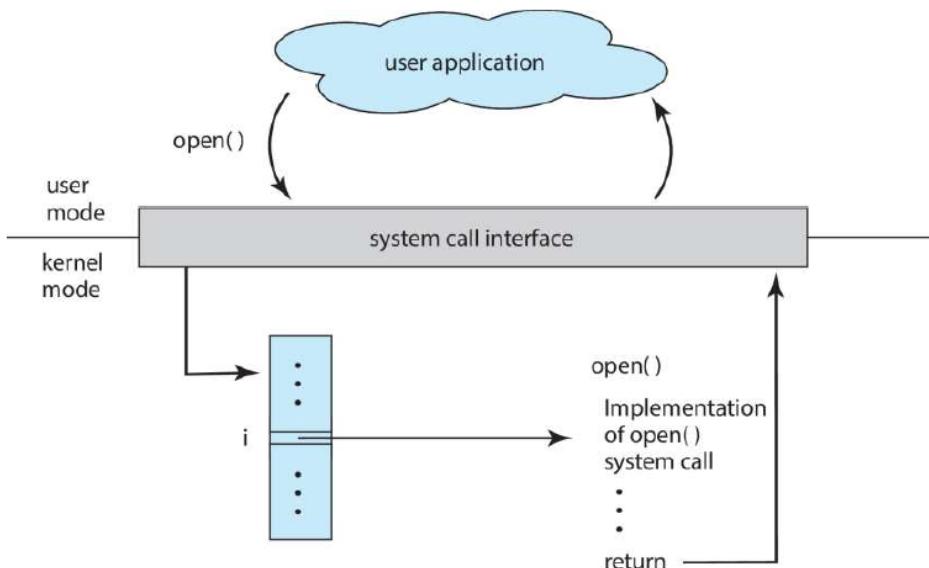
- 与API区别
  - system calls是系统层，api是编程语言层
  - system calls一般是C/C++，api可以是任何语言
  - system calls内核态，api用户态
  - system calls简单，api复杂

■ System call sequence to copy the contents of one file to another file

API: `copy()`



## API – System Call – OS Relationship

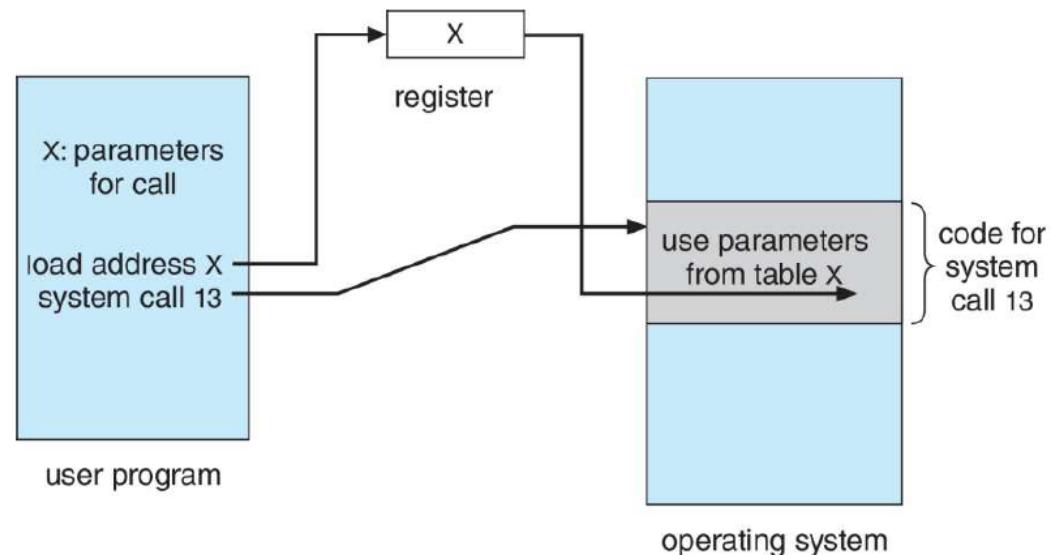


2.19

- 系统调用传参



## Parameter Passing via Table



2.21

- 系统调用种类

- 进程控制

- Process control

- create process, terminate process
      - end, abort
      - load, execute
      - get process attributes, set process attributes
      - wait for time
      - wait event, signal event
      - allocate and free memory
      - Dump memory if error
      - **Debugger** for determining **bugs, single step** execution
      - **Locks** for managing access to shared data between processes

- 文件管理



## ■ File management

- create file, delete file
- open, close file
- read, write, reposition
- get and set file attributes

◦ 设备管理

## ■ Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

◦ 信息维护

## ■ Information maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

◦ 通讯

## ■ Communications

- create, delete communication connection
- send, receive messages if **message passing model** to **host name** or **process name**
  - ▶ From **client** to **server**
- **Shared-memory model** create and gain access to memory regions
- transfer status information
- attach and detach remote devices

◦ 保护

## ■ Protection

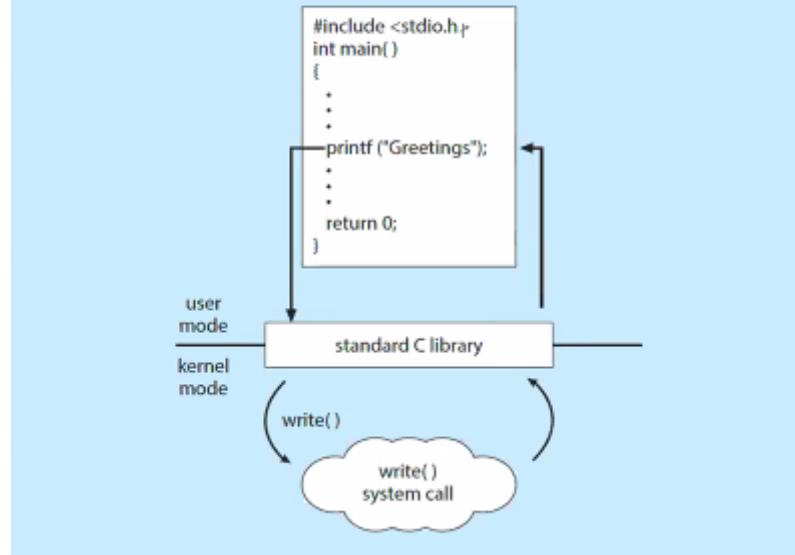
- Control access to resources
- Get and set permissions
- Allow and deny user access

- Unix 系统调用例子

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS		
The following illustrates various equivalent system calls for Windows and UNIX operating systems.		
	Windows	Unix
Process control	CreateProcess() 创建进程 ExitProcess() 退出进程 WaitForSingleObject() 等待进程	fork() exit() wait()
File management	CreateFile() 创建文件 ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() 数据流控制 ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

## THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



c++ lib `printf()` -> system call `write()` -> linux `printk()`

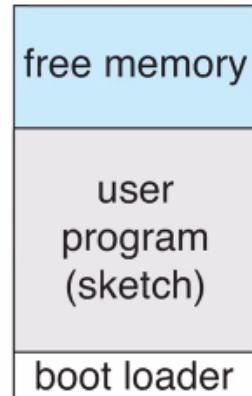
## 操作系统设计

### Arduino

- Single-tasking
- No operating system
- Programs (sketch) loaded via USB into flash memory
- Single memory space
- Boot loader loads program
- Program exit -> shell reloaded



(a)



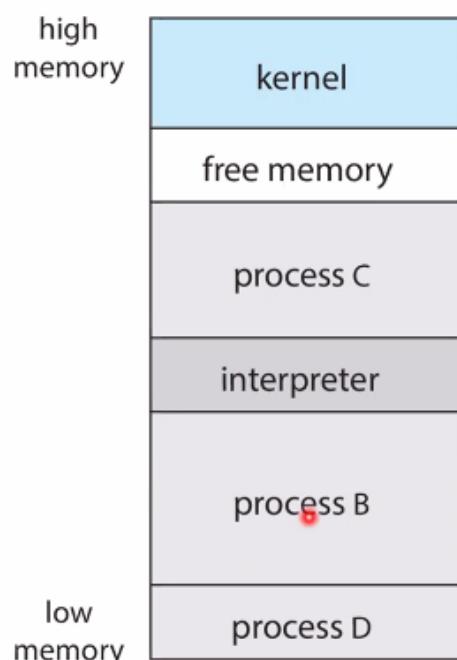
(b)

At system startup

running a program

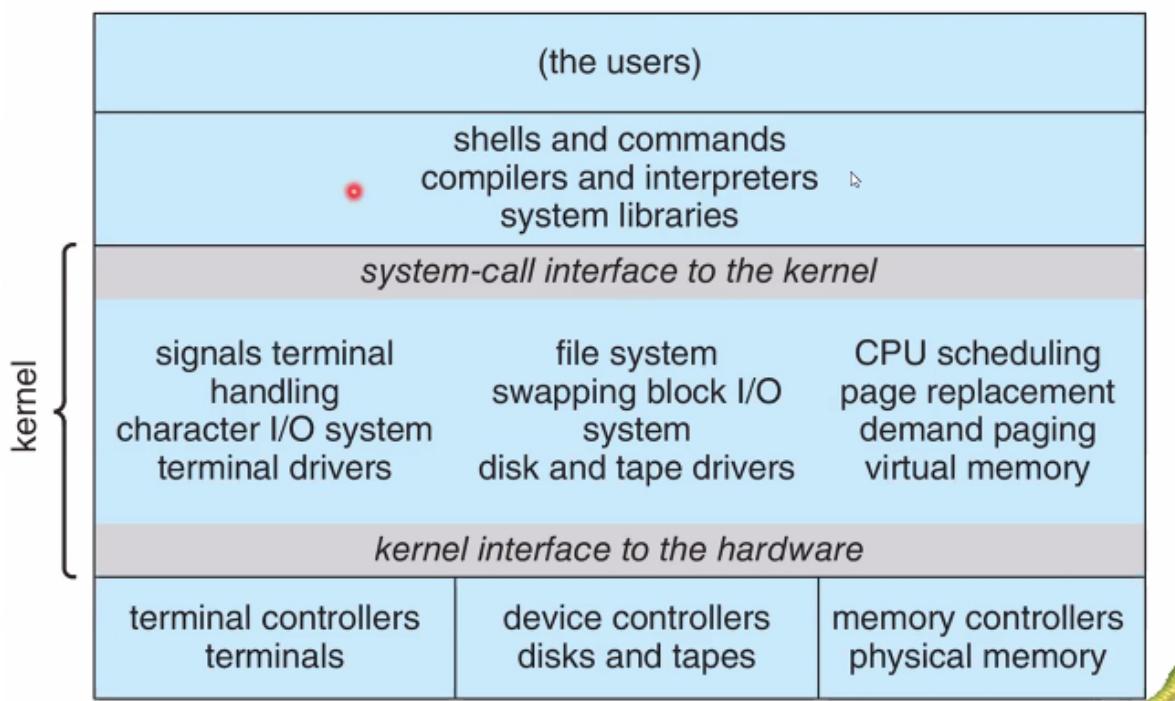
### FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



## UNIX

Beyond simple but not fully layered



Layer 0: hardware

Layer 1: device drivers (kernel interface to hardware)

Layer 2: OS kernel

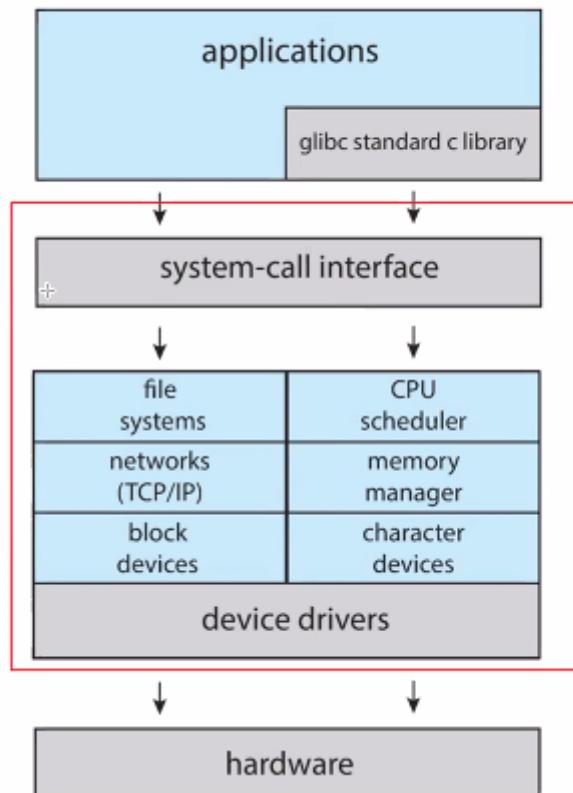
Layer 3: system call interface

Layer 4: user interface

Layer 5: users

## LINUX

## Monolithic plus modular design

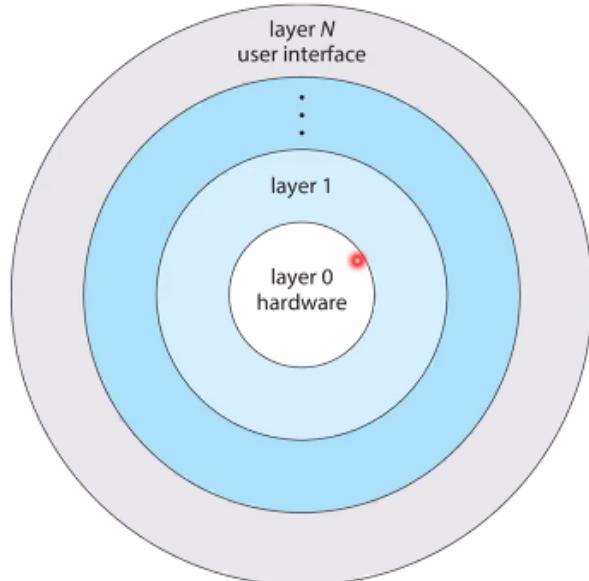


红圈部位为system

## 设计理念

### 分层设计理念

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



分成若干层次

优点：

- 易于调试和修改：每一层被“限定”了相应的功能，调试时不用考虑其他层次出错的可能性，更改仅影响系统的有限部分，而不是影响操作系统的所有部分。
- 便于进行设计和编写：较低层为较高层隐藏了一定的数据结构、操作和硬件。较高层利用较低层所提供的功能来实现，不需要知道如何实现这些操作。

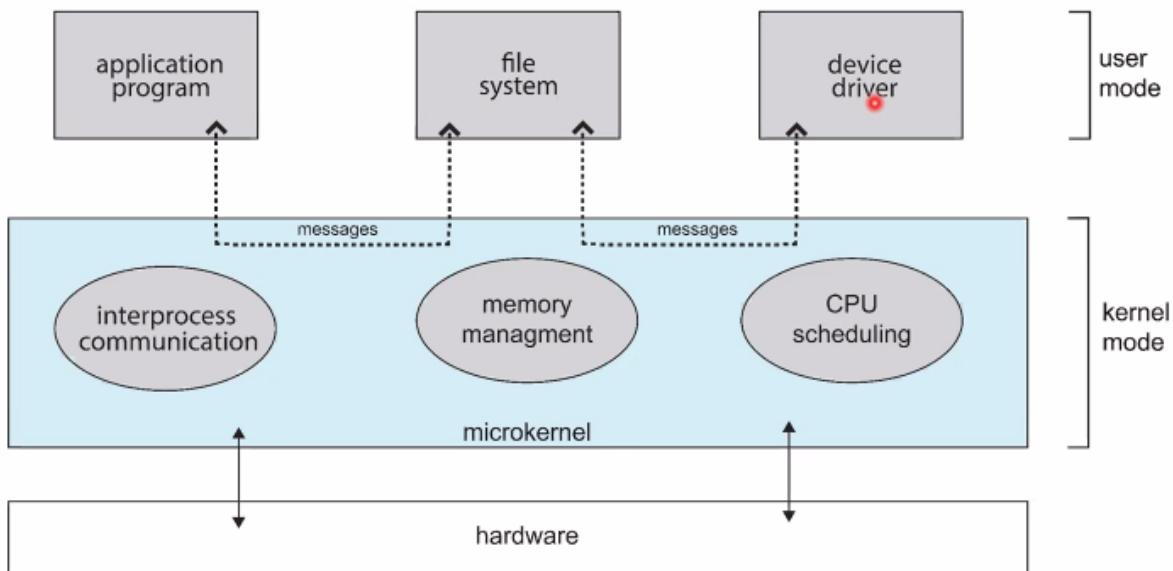
缺点：

- 效率较低，浪费资源，层间的数据传递需要额外的开销

## 微内核

- 内核模块很小，核心内核部分不多
- 内核代码都是必不可少的，便于从微内核进行扩展，容易添加新命令
- 未集成体系结构，松耦合，便于去适应新的体系结构
- 更加可靠和安全
- 用户空间与内核空间通信会有额外性能开销

### 结构



## 模块化编程 可以加载的内核模块 LKM loadable kernel modules

- 面向对象方法
- 每个模块方便删和增

## 系统服务

### 链接和加载 该部分来自

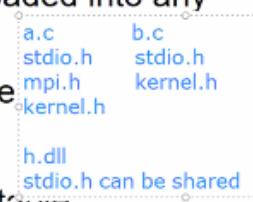
链接 (Linkers)：程序在运行的时候，可能调用了其他文件（如头文件），需要将当前程序与其他文件进行链接。

- 静态链接：一个程序运行的时候，直接建立链接；优点：稳定；缺点：如果有多个程序执行，链接相同文件，文件会被加载多次，导致内存浪费。
- 动态链接：为了解决文件被加载多次，需要写一个DLL文件，声明哪些程序可以共享一个头文件。优点：不需要重复加载某些头文件，内存空间节省，效率更高；缺点：需要写很多DLL文件。

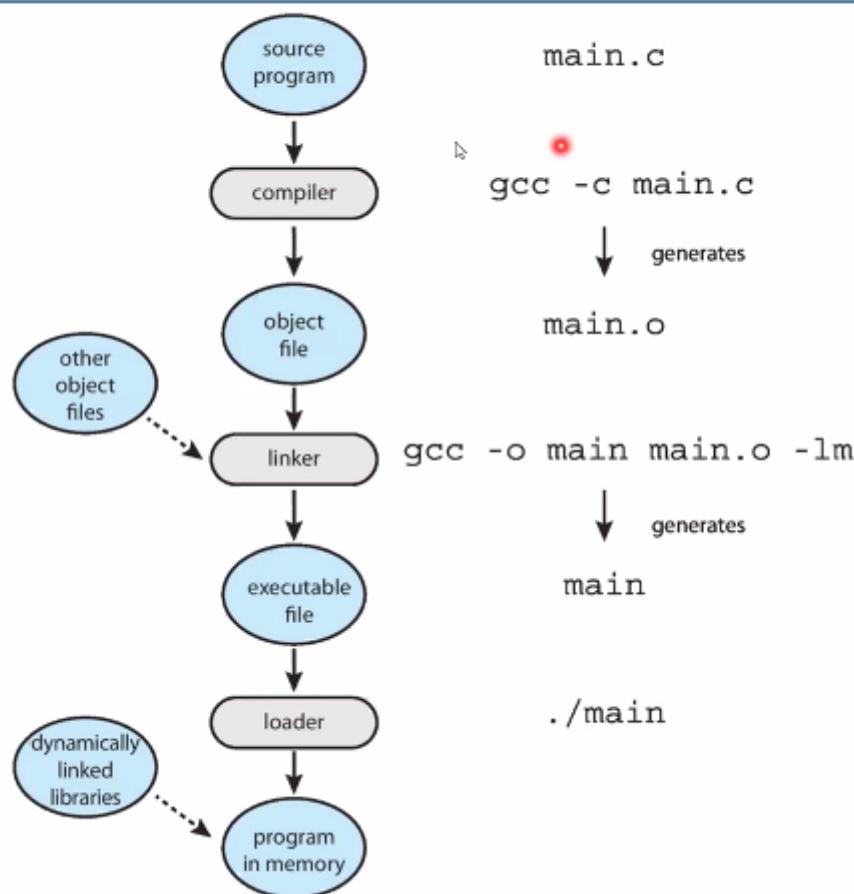
加载 (Loaders)：程序在运行的时候，可能调用了其他文件（如头文件），需要将其他文件的内容加载到内存里以供使用。

- 静态加载：一个程序运行的时候，将这个程序全部加载进内存。
- 动态加载：一个程序运行的时候，只将当前需要用到的内容加载进内存，内存的内容会不断替换，以保证空余内存。

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable file**
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them



## The Role of the Linker and Loader

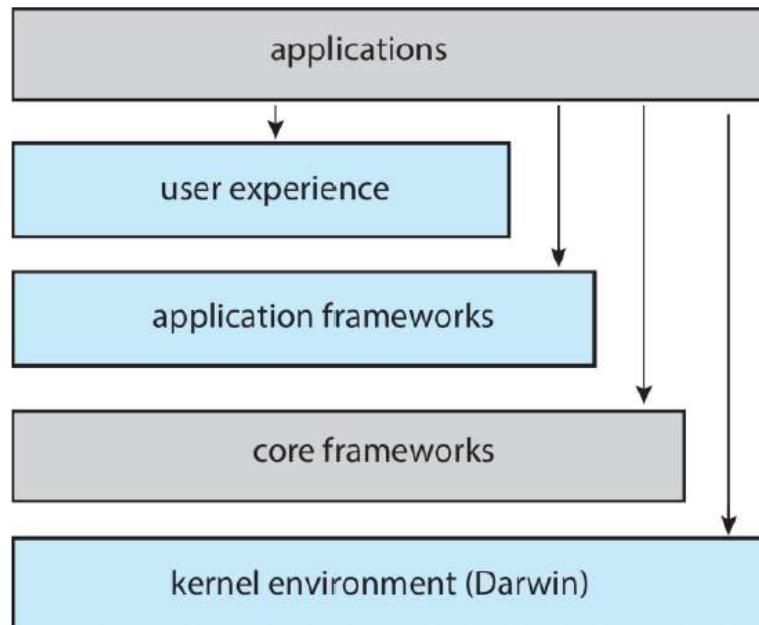


混合驱动

例子



## macOS and iOS Structure



2.49



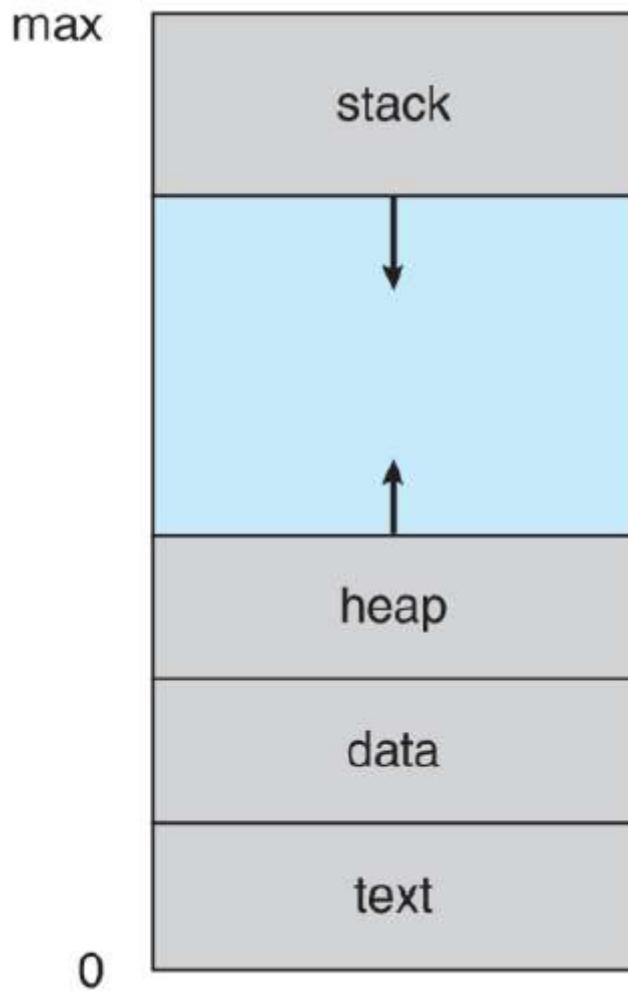
## Ch 03 进程 process

### 进程概念

- 操作系统执行作为进程运行的各种程序。
- 流程-正在执行的程序；流程执行必须按顺序进行
  - Multiple parts
    - The program code, also called **text section**
    - Current activity including **program counter**, processor registers
    - **Stack** containing temporary data
      - ▶ Function parameters, return addresses, local variables
    - **Data section** containing global variables
    - **Heap** containing memory dynamically allocated during run time



### 进程在内存中存储

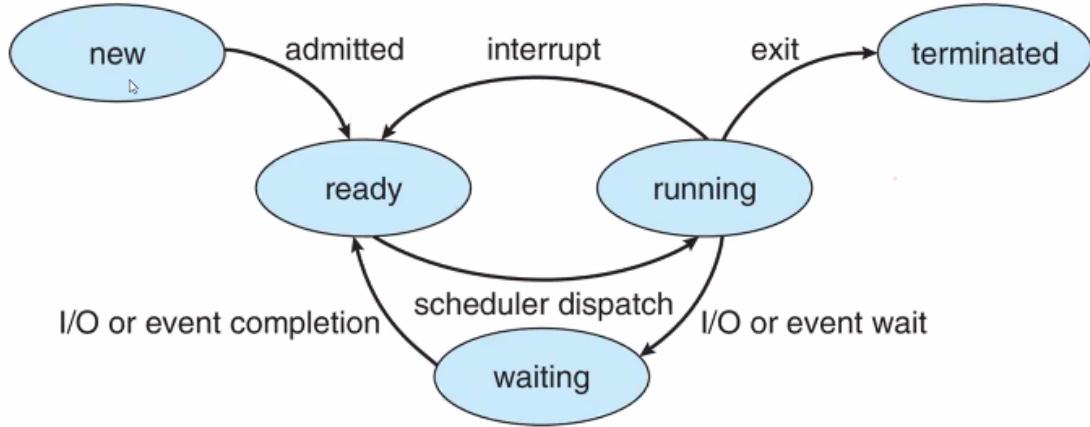


- **文本段(text section)或代码段(code section)**
- **程序计数器**的值和**处理器寄存器**的内容
- **栈**: 包括临时数据, 比如函数参数, 返回地址和局部变量
- **数据段**: 全局变量, 常量, 静态变量
- **堆**: 动态内存分配

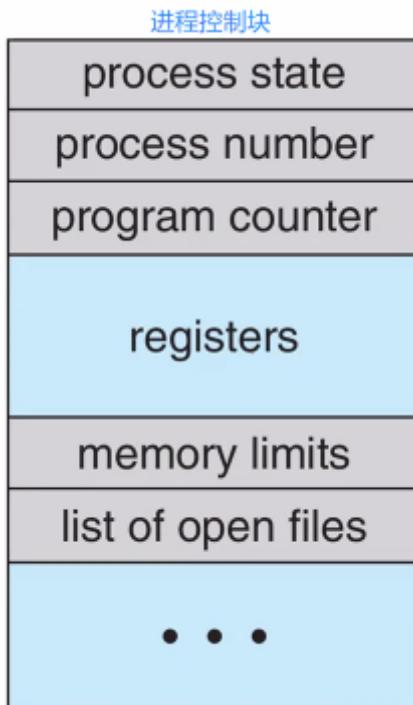
## 进程状态

---

- **新建 (new)** : 进程正在创建。
- **运行 (running)** : 指令正在运行。
- **等待 (waiting)** : 进程等待某个事件发生 (如I/O完成或者某个信号)。
- **就绪 (ready)** : 进程等待分配处理器。
- **终止 (terminated)** : 进程已经完成执行。



## 进程控制块 PCB



PCB中包含了很多的特定的进程相关信息：

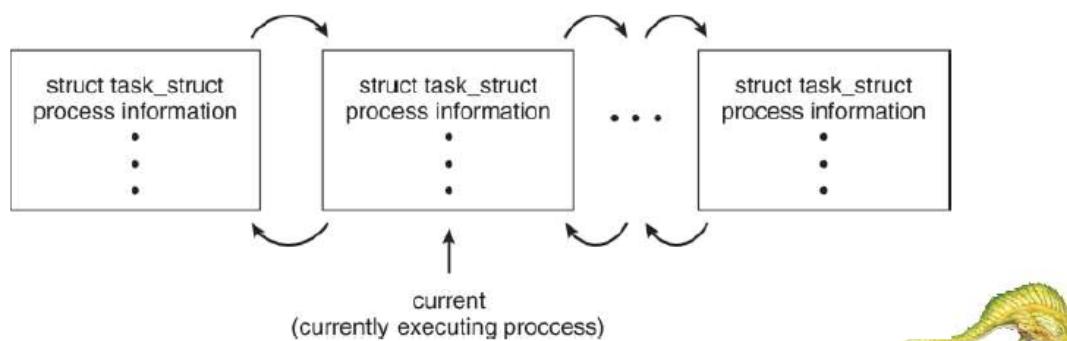
- **进程状态(state):** 状态可以包括，新建，就绪，运行，等待和停止。
- **进程序号(Process Number):** pid
- **程序计数器(PC):** 计数器表示进程将要执行的下一个指令的地址。CPU到进程和线程抽象的重要CPU物理寄存器。
- **CPU寄存器(CPU register):** 根据计算机体系结构的不同，寄存器的类型和数量也会不同。包括累加器，索引寄存器，堆栈指针，通用寄存器和其他条件码信息寄存器。在发生中断时，这些状态信息和程序计数器一起保存到PCB中，一边进行上下文的切换。
- **CPU调度信息(CPU scheduling information):** 这类信息包括进程的优先级和调度队列的指针和其他调度参数。
- **内存管理信息(memory-management information):** 根据操作系统使用的内存系统，这类信息可以包括基地址和界限寄存器的值，页表和段表。
- **记录信息(accounting information):** 这类信息包括CPU时间，实际使用时间时间期限，作业和进程数量等。
- **I/O状态信息(I/O status information):** 这类信息包括分配给进程的I/O设备列表，打开的文件列表。

# Linux进程表示

linux 操作系统的进程控制块是采用C语言的结构体 task\_struct 来表示，它位于内核的源代码目录内的头文件<linux/sched.h>。这个结构体包含用于表示进程的所有必要信息，包括进程状态，调度和内存管理信息，打开的文件列表，指向父进程的指针以及子进程，兄弟进程列表的指针。

Represented by the C structure `task_struct`

```
pid t_pid;                      /* process identifier */  
long state;                     /* state of the process */  
unsigned int time_slice /* scheduling information */  
struct task_struct *parent; /* this process's parent */  
struct list_head children; /* this process's children */  
struct files_struct *files; /* list of open files */  
struct mm_struct *mm; /* address space of this process */
```

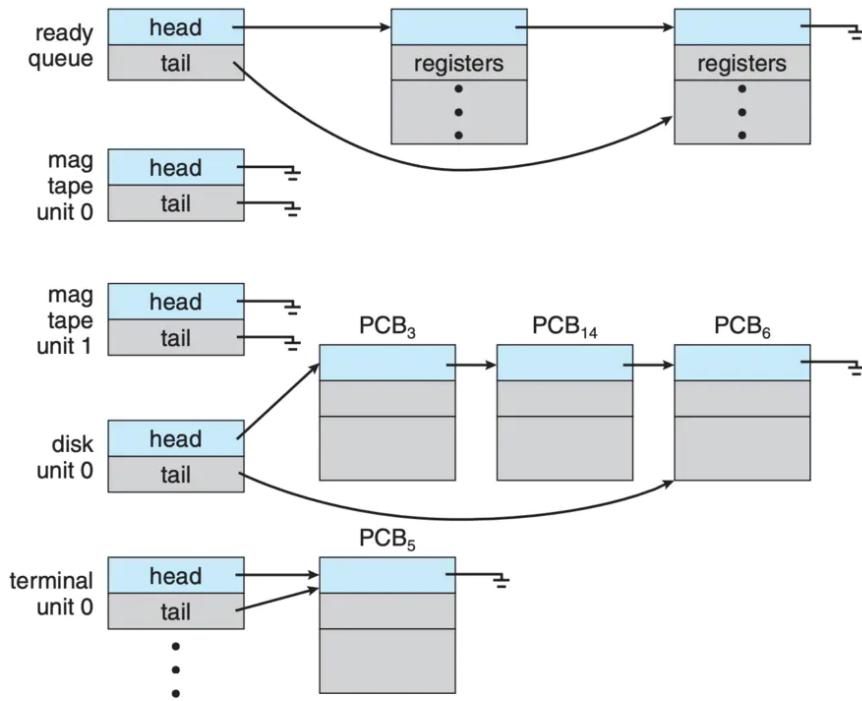


## 进程的调度

### 调度队列

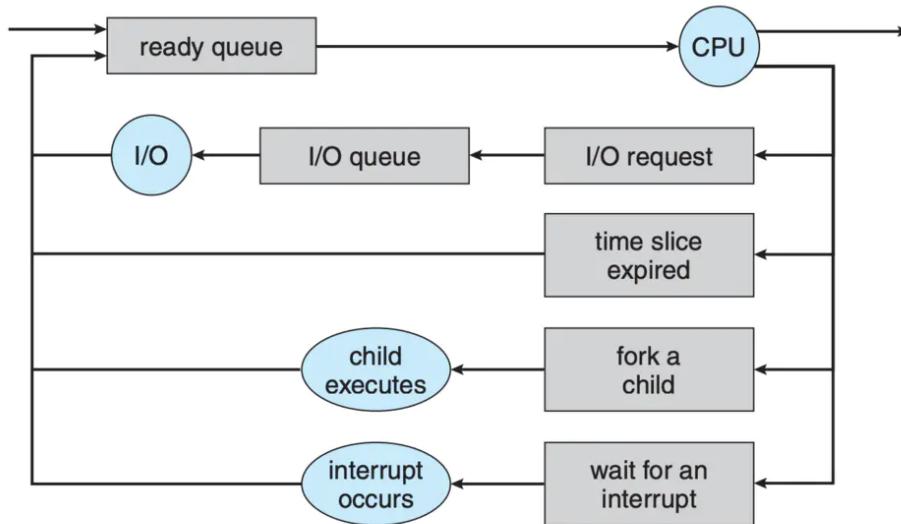
进程在进入系统时，会被加入**作业队列**(job queue)，这个队列包括系统内所有的进程。除此之外，系统还有一个**就绪队列**。它保存了驻留在内存中的，就绪的，等待运行的进程，一般是一个**链表结构**来实现的。其头节点有两个指针，用于指向链表的第一个和最后一个PCB块；每个PCB有个指针指向队列里的下一个PCB。

系统还有其他队列。当一个进程被分配了CPU以后，它执行了一段时间，最终退出，或被中断，或等待特定事件发生(I/O事件完成)。假设进程向一个共享设备发出请求，由于具有许多进程，磁盘可能忙于其他进程的I/O进程，因此该进程可能需要等待磁盘。等待I/O设备的进程列表，也被放到了一个队列中，这个队列就是**设备队列**。每个设备都有属于自己的设备队列。



**Figure 3.5** The ready queue and various I/O device queues.

CPU的调度



**Figure 3.6** Queueing-diagram representation of process scheduling.

1. 进程进入系统后，加入到就绪队列中，这时进程处于就绪状态。
2. 进程被调入到CPU中后，就进入到了运行状态。
3. 运行时需要访问I/O设备，进程被放入设备队列中，进入挂起状态(或者等待状态)，直到I/O事件就绪后，在进入到就绪对列中。

## 调度程序

在上面的进程的整个生命周期中，进程会在各种调度队列中迁移。

操作系统为了调度必须按照一定的方式从这些队列中选择进程。进程选择通过**调度程序（调度器）**进行调度。

通常，对于有大量的需要执行进程的系统中。可能进程的任务不能全部加载到内存中，这个时候会有部分的进程保存在磁盘中。这部分磁盘的数据，他作为一个虚拟的内存池被称为**虚拟内存**。

调度程序可以分为**长期调度程序**和**短期调度程序**。

- **长期调度程序**会把在磁盘上的进程加载到内存，把最近不会被执行的进程从内存中挪出。
- **短期调度程序(CPU调度程序)**从就绪队列中选择进程分配CPU。

为什么会有两个调度程序？

这两种调度程序的主要区别是执行效率。短期调度程序必须经常为CPU选择新的进程。长期调度程序执行并不频繁，但是它控制着内存中进程的数量，也就是并发进程的程度。

长期调度程序会进行认真的选择。一般根据进程的类型分为：**I/O密集型进程**和**CPU密集型进程**。

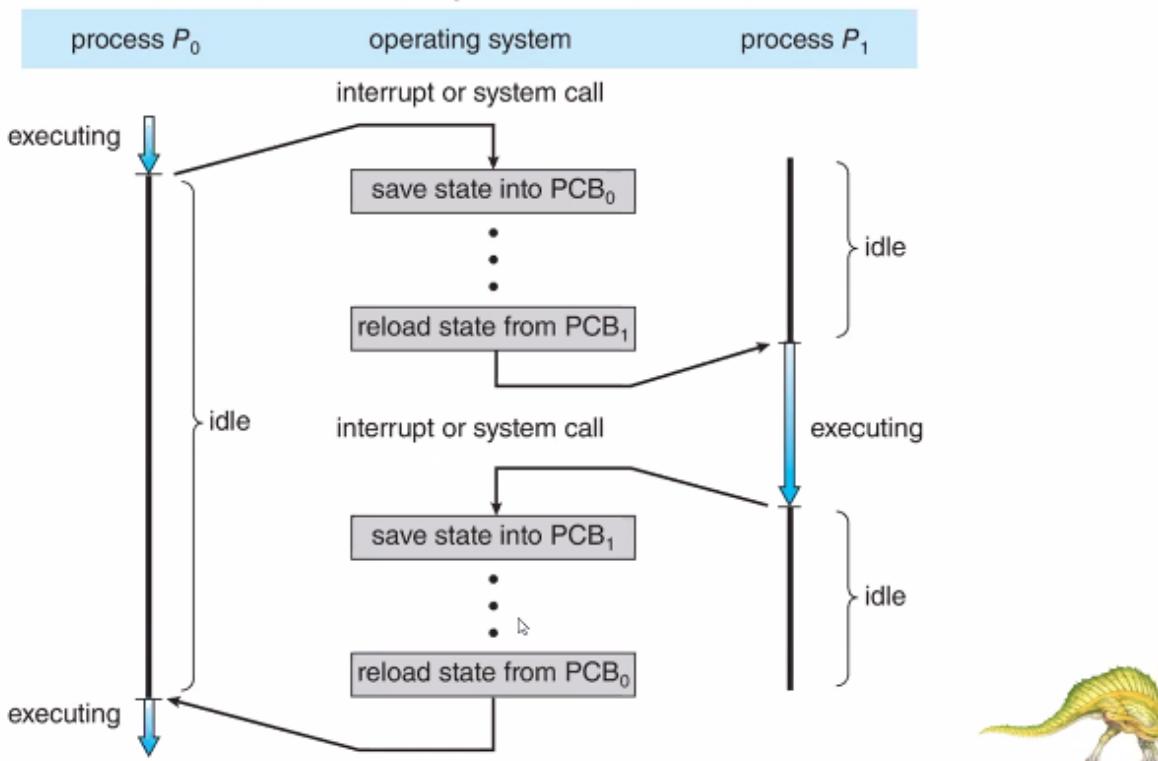
如果所有进程都是I/O密集型进程，那么I/O设备队列一直都是满的，而就绪队列是空的，短期调度程序就一直处于空闲。

当所有进程都是CPU密集型进程，那么I/O设备队列都是空的，I/O设备就没有得到使用。所以长期调度程序是保证系统使用CPU和I/O设备平衡的。

## 进程切换（上下文切换）

# CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.



- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

## 移动操作系统的多任务

由于移动设备的限制，早期版本的ios系统是不支持多任务的，但是在ios4 后，苹果系统编程的API中提供了多任务支持，允许前台程序（占用屏幕的程序）和后台程序（位于内存中，但不占用屏幕）。但是也是有限制：

- 运行单个，长度有限的任务(例如:通过网络完成下载内存)
- 接受不了事件通知(如 新的电子邮件消息)
- 可长时间运行的后台程序(如 音频播放器)

由于电池寿命和内存的影响，Apple也可能会限制多任务。

Android系统对后台运行的运行没有这种限制，如果一个应用程序必须后台处理，那么这个应用必须使用**服务**，也就是后台进行运行的，必须是独立的组件。以音频流程序为例：如果程序后台运行，那么服务会为后台应用继续发送音频文件到音频设备驱动程序。服务没有用户界面，并且占用内存少，因此为移动环境提供更高效的多任务支持。

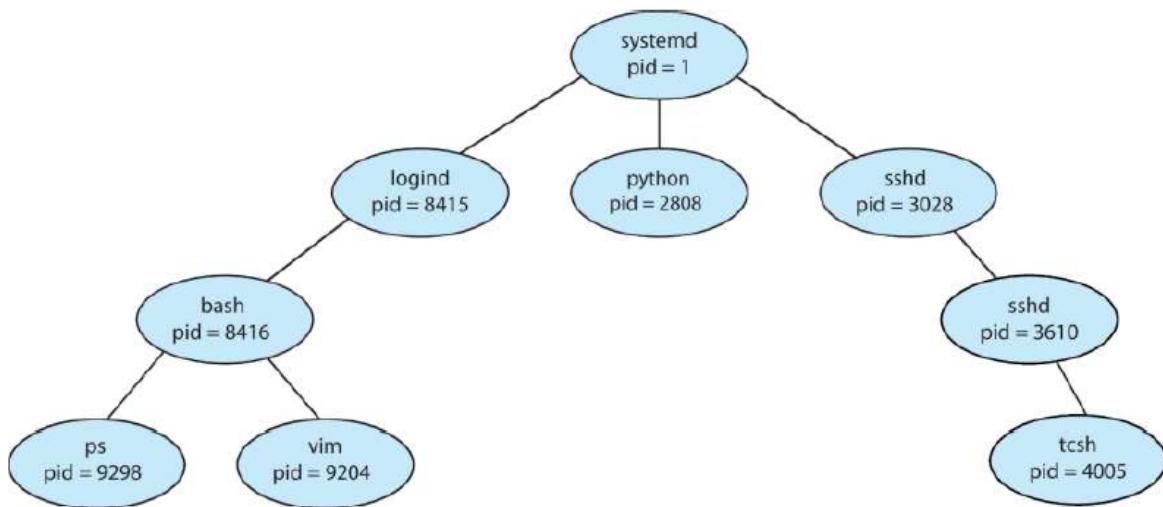
## 进程的运行

---

### 进程创建

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

进程在执行的过程中可以创建多个新的进程。创建进程成为父进程，而新的进程成为子进程。每个子进程又可以创建其他进程，从而形成进程树。下图为一个Linux系统的典型的进程树。



## 父子进程

一般来说，当一个进程创建子进程时，该子进程会需要一定的资源来完成任务。子进程可以从操作系统那里直接获取资源，也可以只从父进程里获得。父进程可能在子进程之间分配资源或者共享资源（主要是打开的I/O设备，也有可能是内存）。

在进程创建新进程时，可能有两种情况：

- 父进程与子进程并发进行。
- 父进程等待，直到某个子进程执行完成。

新进程的地址空间也有两种可能：

- 子进程是父进程的复制品。

- 子进程加载另外一个程序。

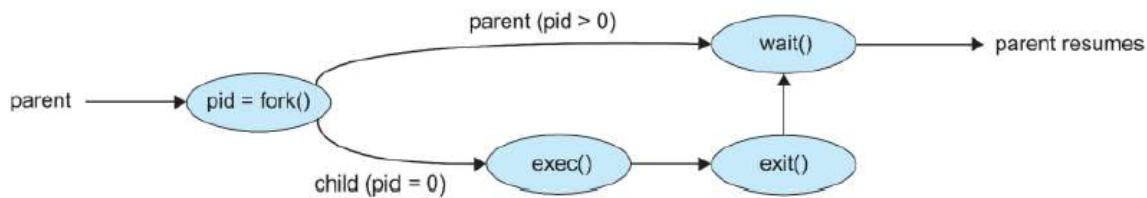
这里看下Unix和Linux的具体实现

每个进程都有一个唯一的整型进程标识符来标识。进程是通过`fork()`系统调用来创建的。每个进程的地  
址空间复制了原来的父进程的地址空间。这种机制允许父子进程之间可以轻松的通信。

这两个进程都继续执行`fork()`之后的指令。但是对于子进程，系统调用`fork()`的返回值是0，对于父进程，  
返回的是子进程的进程标识符。根据返回返回值的不同对父子进程进行不同的操作。

通常`fork()`系统调用之后，子进程一般会使用`exec()`系统调用，以新的程序来进行替换新进程的内存空  
间，系统调用`exec()`加载二进制文件到内存中（破坏包含系统调用`exec()`的原来进程内存内容），并开始  
执行新的进程。

父进程能够创建更多的子进程，或者如果在子进程运行时没有什么可做，那么他就会使用系统调用  
`wait()`把自己移出就绪队列，等待子进程完成，这里是任意一个子进程。



- 父进程等待子进程结束

## 进程终止

当进程完成执行最后语句并通过系统调用`exit()`请求操作系统删除自身时，进程终止。这时，进程可以返  
回状态值到父进程（如果父进程调用了`wait()`）。所有进程资源，如物理和虚拟内存，文件表，I/O缓冲区等，都由操作系  
统释放。

在其他情况下，进程也有可能出现进程终止。进程通过适当的系统调用，可以终止另一个进程。通常，  
只有终止进程的父进程才能执行这一系统调用，否则用户可以任意终止彼此的进程。

父进程终止子进程的原因有很多，如：

- 子进程使用了超过他分配的资源
- 分配给子进程的任务，不再需要
- 父进程正在退出，而且有些系统不支持无父进程的子进程继续运行。

第三种的情况，是在某些系统中存在，那么，父进程创建的所有子进程都应该终止，比如关机，这种  
株连九族的现象叫做**级联终止**，一般由操作系统来启动。

在Unix和Linux系统中，父进程可以通过系统调用`wait()`等待子进程的终止，系统调用`wait()`可以通过参数，让父进程获取子进程的终止状态；`wait()`也返回终止子进程的标识符，让父进程知道哪个子进程被  
终止了。

当一个进程终止时，操作系统会释放其资源，不过，它位于进程表中的条目还是在的，直到父进程调用  
`wait()`；这是因为进程表包含了进程的退出状态。

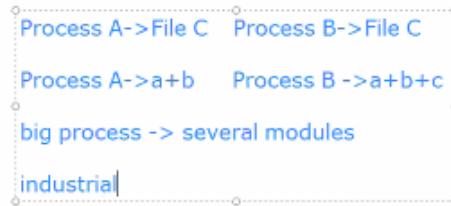
当进程已经终止，父进程没有`wait()`的进程，称为**僵尸进程**。

所有进程在中的时候都会进入到这个状态，但一般僵尸进程只是很短暂的存在，一旦父进程调用了  
`wait()`，僵尸进程的进程描述符和它的进程表中的条目都会被释放。

如果父进程没有执行`wait()`就终止了，那么它的子进程就变成了**孤儿进程**，在Unix和Linux系统，他们的  
父进程就会变成`pid = 1`的`init`进程。`init`会定期进行`wait()`操作。以便收集孤儿进程的状态，并  
释放孤儿进程标识符和进程表条目。

## IPC 进程间通信

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience



- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**

- 信息分享
- 计算加速
- 模块化
- 便捷

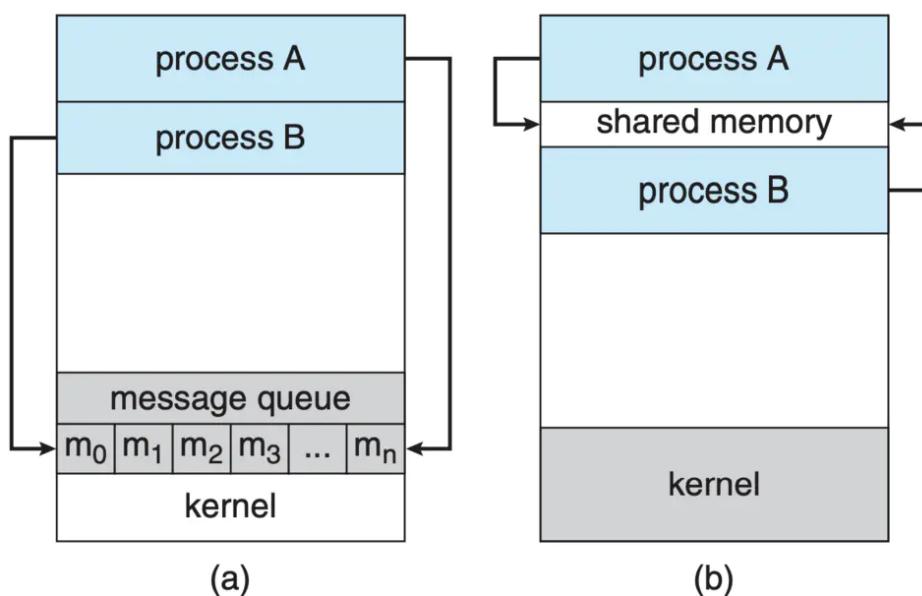
操作系统的并发执行的进程可以是独立的也可以是协作的。一个进程的运行不会影响其他进程的运行，那么这个进程时**独立的**。显然，不与任何其他进程共享数据的进程是独立的。相反的，进程之间就是**相互协作的**。协作的进程之间会有共享的数据。

协作进程需要进行进程间通信 (InterProcess Communication, IPC) 机制，以允许进程相互交换数据与信息。

就目前大多数系统来说，进程间的通信一般由两种基本模型：**共享内存** 和 **消息传递**。

消息传递在交换少量的数据量来说很有用，而且在分布式系统中，消息传递是比共享内存更容易实现。如业界的kafka。

共享内存的优点是快于消息队列，这是因为消息传递经常使用系统调用，而共享内存是在内存里映射一个共享内存区域，访问他就和内存的速度一样，不需要内核进行干预。



**Figure 3.12** Communications models. (a) Message passing. (b) Shared memory.

## 共享内存

采用共享内存进行通信，需要通信的进程公共的共享内存区域。通常，一个进程创建一个共享内存区域，这块内存会驻留在创建共享内存段的进程的地址空间里，其他希望使用共享内存的进程需要将这块地址附加到自己的地址空间里。

## 消息传递

消息传递提供一种机制，以便允许进程不必共享地址空间来实现通信和同步。对分布式系统特别有用。假如两个进程，P和Q需要进程通信，他们需要互相发送和接受消息：他们必须有**通信链路**。该通信链路有多种实现方法。但这里不关心物理实现。

一般消息传递工具，至少提供两种操作：

```
send (messge)  
receive(message)
```

### ■ Implementation of communication link

- Physical:
  - ▶ Shared memory
  - ▶ Hardware bus
  - ▶ Network
- Logical:
  - ▶ Direct or indirect
  - ▶ Synchronous or asynchronous
  - ▶ Automatic or explicit buffering



### 直连 direct communication

- 链路被自动建立
- 两个通讯进程间有且只有一个链路
- 链路可能是单向的，但更多的是双向的
- 实时通讯

### 间接通讯 indirect communication

- 消息通过邮箱 (buffer)
  - 每个邮箱有独一无二的id
  - 只有进程间共享一个邮箱时，才可以进行通讯
  - 邮箱可由多个进程共享
  - 每两个进程间可能有多个共享的邮箱（类似群聊）
  - 可能是单向的或是双向的
- 操作
  - 建立新的邮箱
  - 发送和接受邮件
  - 删除邮箱
- 问题

## Mailbox sharing

- $P_1, P_2$ , and  $P_3$  share mailbox A
- $P_1$ , sends;  $P_2$  and  $P_3$  receive
- Who gets the message?

## 同步 synchronous

进程间通信可以通过调用原语send() 和receive()来进行。实现这些原语有很多种设计方案。  
消息在传递的时候可以是阻塞的(*blocking*)，也可以是非阻塞的(*nonblocking*)，也称为同步的(*synchronous*)和异步的(*asynchronous*)。

- 阻塞发送：发送进程阻塞，直到消息由接收进程或者邮箱所接收。
- 非阻塞发送：发送进程发送消息，不用等待接收进程回复，继续自己的操作。
- 阻塞接收：接收进程阻塞，直到有消息可用。
- 非阻塞接收：接收进程收到一个有效消息或者空消息。

## 网络进程通信

了解了使用共享内存和消息传递进行通信。但是现在使用最多的还是基于客户端/服务器通信的方式。  
这里介绍网络通信最常用的两种策略和历史的IPC机制管道：套接字(socket)，远程程序调用 (RPC)  
和管道。

### 套接字

套接字为通信的端点。通过网络通信的每对进程需要使用一对套接字，每个进程表示一个端点。  
每个套接字由一个IP地址和一个端口组成。通常，套接字采用客户端——服务器的架构。服务器通过监听指定端口，来等待客户请求。服务器收到请求后，接收来到套接字的连接，从而完成套接字的连接。

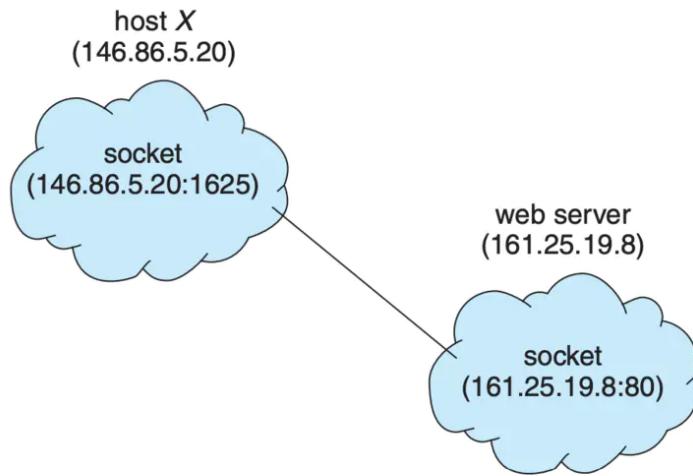


Figure 3.20 Communication using sockets.

### 远程过程调用 (RPC)

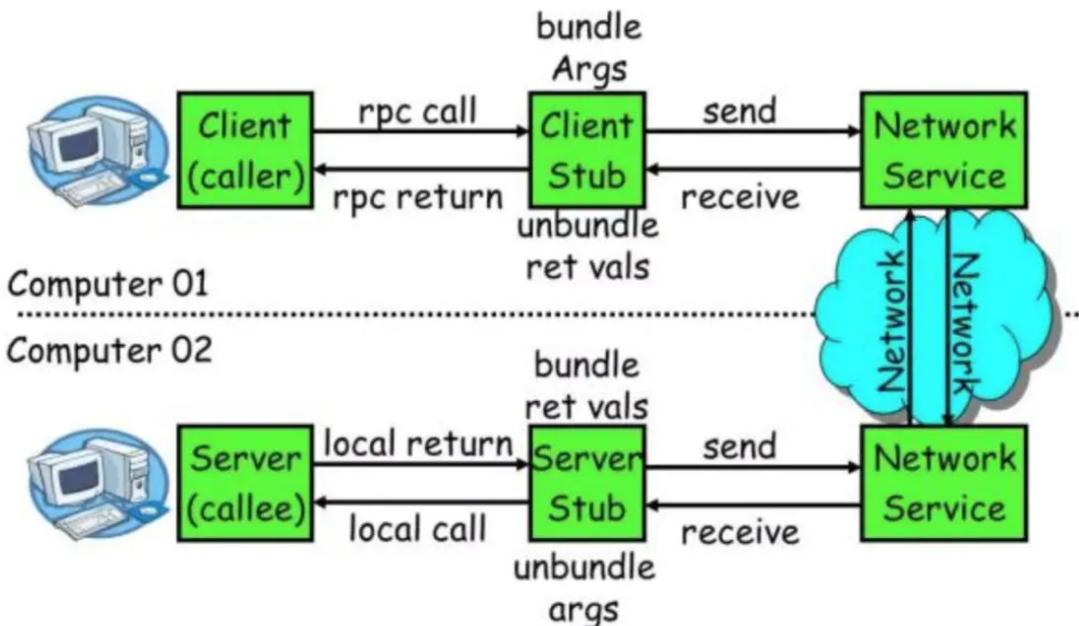
RPC 与IPC相对应，是目前业界许多互联网公司常用的通信方式。

和IPC不同的是，RPC通信交换的消息具有明确的结构，而不仅仅是数据包。消息传递到RPC服务，RPC服务监听远程系统的端口号；消息包含用于指定，执行的函数的一个标识符和传给函数的参数。函数按照要求来进行执行，而所有的结构会通过另一种消息传递给请求方。

过程：

RPC语义上允许客户端调用位于远程主机的过程，就如同调用本地的过程一样。它一般通过客户端提供的**存根**(stub)，当用户调用远程过程时，RPC系统调用适当的存根，并且传递远程过程参数，都有存根定位服务器的端口，并且封装参数并且对rpc的内部结构进行序列化并进行打包。然后向服务器发送一个消息。

服务器根据类似于存根接收到这个消息，并且调用定义好的Rpc协议格式，对封装的结构解包并进行反序列化，得到客户端的消息。



## 管道

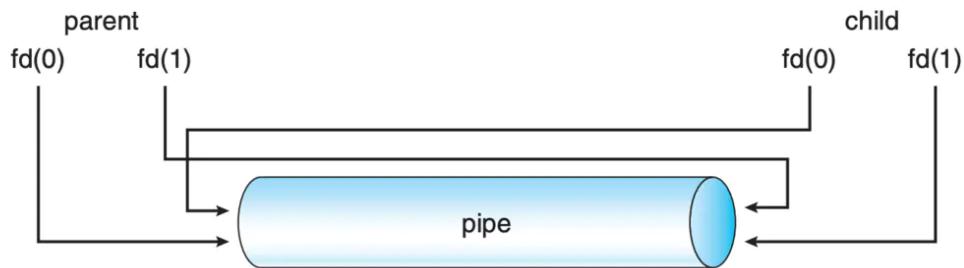
管道允许两个进程进行通信，早期是Unix系统最早使用的一种IPC通信机制。现在基本不怎么用了。这里做最基本的管道设计考虑。

- 管道允许单向通信还是双向通信。
- 如果允许双向通信，他是半双工的还是全双工的。
- 通信进程之间是否应该有一定的关系（父子进程）
- 管道通信是否只能在单机进行，是否可以在网络上用？

根据管道的类型分为：普通管道和命名管道

## 普通管道

- 普通管道是单向的，要双向通信，就要两个管道。
- 普通管道通信是在父子进程间进行的。所以肯定不能进行网络通信。



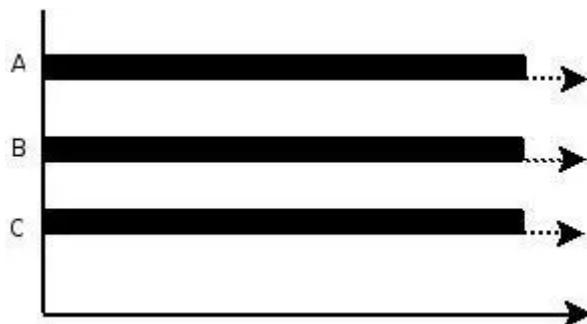
**Figure 3.24** File descriptors for an ordinary pipe.

### 命名管道 (FIFO)

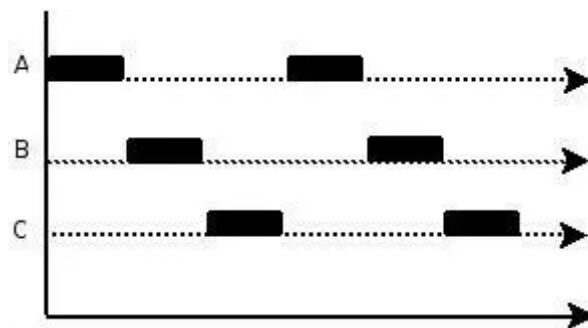
- 专线、大流量
- 命名管道可以是双向的。但是具体在实现的时候一般都是半双工的，如果要实现双向通信，需要两个FIFO。
- 不需要是父子关系。
- 只能在一台机器上使用，网络通信还是选择套接字。

## Ch 04 线程threads和并发concurrency

**并行(parallel):** 指在同一时刻，有多条指令在多个处理器上同时执行。所以无论从微观还是从宏观来看，二者都是一起执行的。



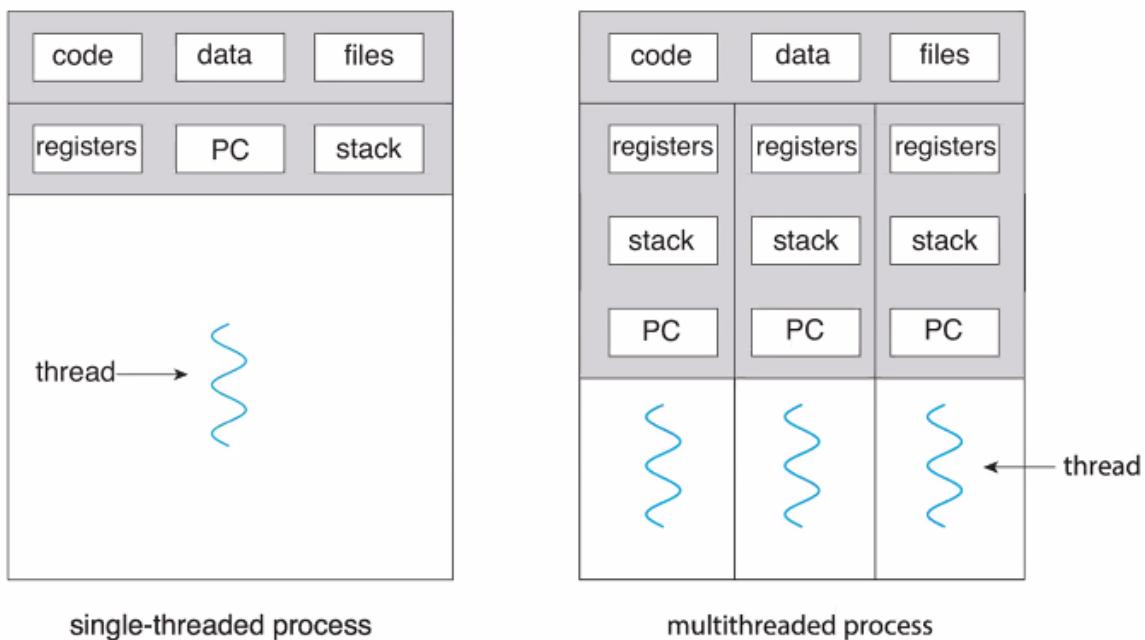
**并发(concurrency):** 指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



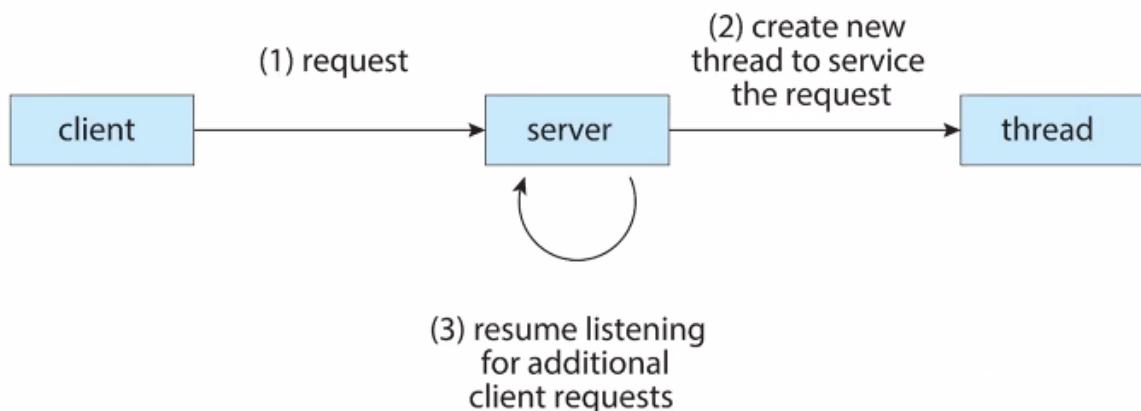
### 线程

每个线程是CPU使用的基本单元；他包括线程ID,程序计数器，寄存器和堆栈。它与统一进程的其他线程共享代码段，数据段和其他操作系统资源，如打开文件和信号。

每个传统的进程只有一个控制线程。如果一个进程拥有多个控制线程，那么它能同时执行多个任务。下图说明了传统单线程和多线程进程的差异。



## 多线程服务器

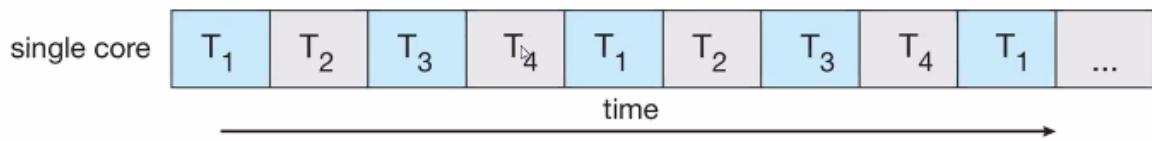


## 多线程好处

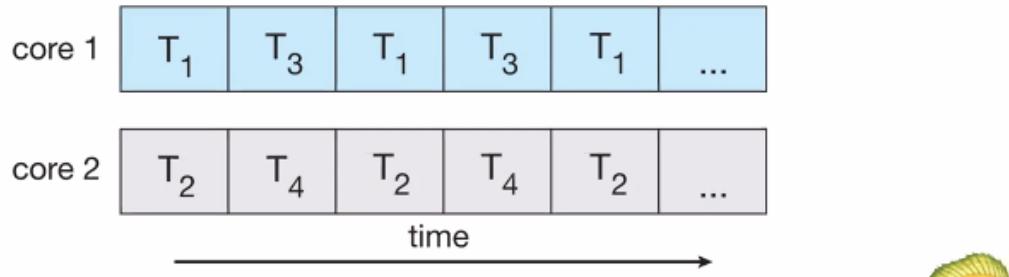
- 响应时间更短
- 线程间更容易实现共享
- 线程需要的资源少于进程 更加经济
- 更容易扩展到多核

多核处理器提供了更好的多线程环境

## ■ Concurrent execution on single-core system:



## ■ Parallelism on a multi-core system:



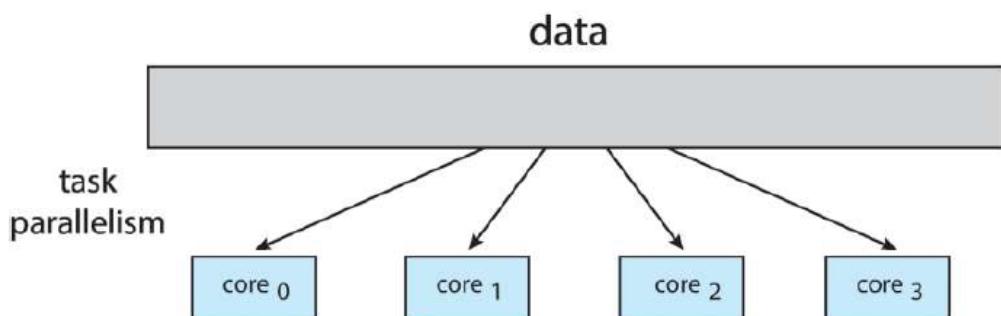
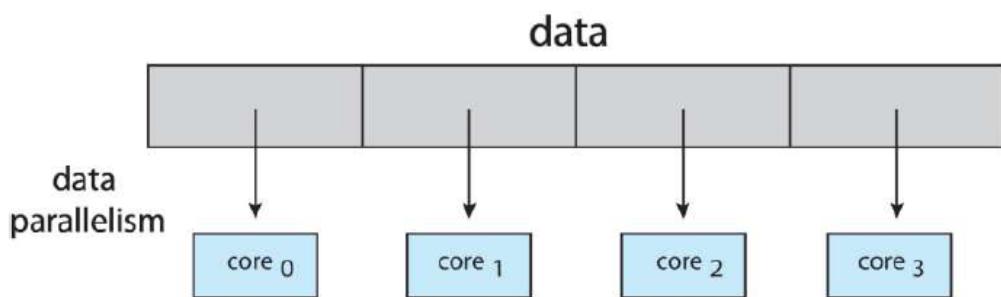
## 并行的不同类型

- 数据并行

将相同数据的子集分布在多个内核上，每个内核上执行相同的操作

- 任务并行

在各个线程之间分配线程核心，每个线程执行独立的操作



## 线程模型

在线程运行的过程中，一般分为用户线程和内核线程。

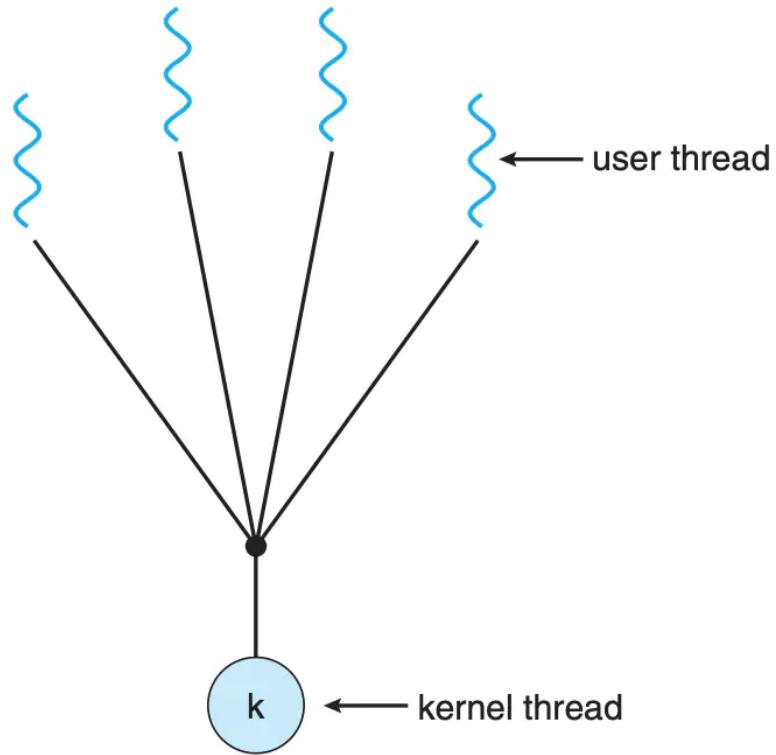
用户线程位于内核之上，它的管理无需内核支持。

内核线程由操作系统来直接支持和管理。几乎所有的现代操作系统都是支持内核线程的。

在实现的方式上，用户线程和内核线程存在某种关系。一般分为三种：一对一模型，多对一模型，多对多模型。

## 多对一模型

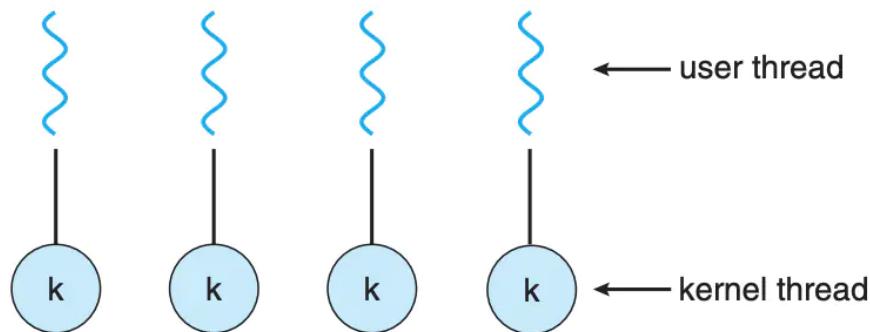
多对一模型映射多个用户线程到一个内核线程。线程管理是由用户控件的线程库来完成的，因此效率更高。但是，一个线程执行阻塞系统调用，那么整个进程将会被阻塞。而且，在任意时间只能有一个线程可以访问内核。万一内核线程也在同步等待其他事件的发生，那么此时，进程的其他线程也就得不到执行了。



**Figure 4.5** Many-to-one model.

### 一对一模型

一对一模型：每个用户线程映射到一个线程。这个模型，在一个线程执行系统调用的时候，能够允许另一个线程继续执行。它也允许多个线程并行运行在多核处理器上。这个模型的缺点就是：创建一个用户线程就要创建一个内核线程，内核线程的创建开销影响程序的性能。所以在这个模型中，系统对支持的线程的数量是有限制的。

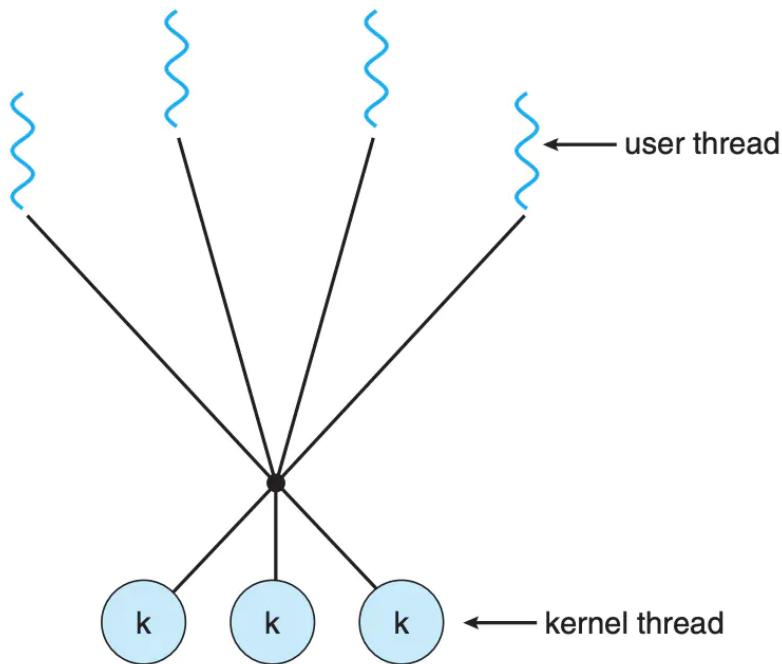


**Figure 4.6** One-to-one model.

## 多对多模型

多对多模型多路复用多个用户线程到同样数量或者数量更小的内核线程上。这种模型结合了前两种的方式。没有了前两种的缺点。

开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器上进行并发执行。而且当一个用户执行系统调用的时候，内核可以调度另外的内核线程来执行其他线程的操作。



**Figure 4.7** Many-to-many model.

## 线程库

线程库为程序员提供了创建和管理线程的API。

在展开对线程的API的整理之前，首先了解两个概念：**异步线程**和**同步线程**。

**异步线程**：一旦父线程创建了一个子线程，父线程继续自身的执行，不用管子线程何时终止，这个父子线程会并发执行，独立运行。由于各自独立运行，他们通常会有很少的数据共享。

**同步线程**：如果父线程创建一个子线程或者多个子线程后，那么在恢复执行之前，它需要等待所有子线程的终止，就出现了同步线程。由父线程创建的子线程并发执行，但是父线程没有办法继续工作。一旦有线程终止，它就会与父线程连接。所有子线程完成后，父线程才继续工作，这种方式叫做**分叉-连接策略**。

## Pthreads

Pthreads库是Posix 标准定义的线程创建与同步API。它是线程的行为规范，而不是实现。大多数的操作系统都实现了这个线程规范。

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int sum; // 这个数据是被线程共享的数据
void *runner(void* param) {
```

```

printf("son thread is here.\n");
int i = 0;
int upper = atoi((char*)param);
sum = 0;
for (i = 0; i <= upper; ++i) {
    sum += i;
}
pthread_exit(0);
}

int main(int argc, char* argv[]) {
pthread_t tid;
pthread_attr_t attr; // 设置线程属性

if (argc != 2) {
    fprintf(stderr, "usage: a.out <integer value> \n");
    return -1;
}
if (atoi(argv[1]) < 0) {
    fprintf(stderr, "%d must be >=0\n", atoi(argv[1]));
    return -1;
}

// 获取默认的线程属性
pthread_attr_init(&attr);

// 创建线程
pthread_create(&tid, &attr, runner, argv[1]);

// 等待线程退出
pthread_join(tid, NULL);
printf(" parent thread is here.\n");

printf(" sum = %d\n", sum);
return 0;
}

```

根据上面的代码来大概讲解下创建线程的API。

pthread\_t tid; 声明了线程的线程标识符。每个线程都有一个唯一的线程标识符。  
 每个线程都有一组属性，包括堆栈大小和调度信息。pthread\_attr\_t attr; 表示线程属性。  
 通过调用pthread\_attr\_init(&attr) 可以设置这些属性。由于没有任何属性，这里为默认属性。  
 pthread\_create(); 可以创建一个单独的线程。除了传递线程的标识符和属性外，还要传递一个函数名，  
 这里为runner。以便线程从哪个函数开始执行，最后一个需要传递命令行参数。  
 这里程序里有两个线程。一个为main的主线程，另外一个是线程从runner()开始执行。根据上面的分叉-连接策略：主线程通过pthread\_join()来等待runner()完成。runner() 线程在pthread\_exit()之后就会终止。

## 线程池

为了说明线程池，先来看一个多线程的web服务器。

在web服务中，每当服务器收到一个请求时，他就会创建一个单独的线程来处理请求。虽然创建一个线程的代价比创建一个进程的代价要小，但是仍然存在问题。

- 线程在创建需要的时间是多少，处理完这个请求后，线程还是会被注销。
- 如果允许所有并发请求都通过新线程来处理，那么系统的资源和内存需要受到限制，因为不可能无限制的创建新的线程。

解决方法就是线程池。

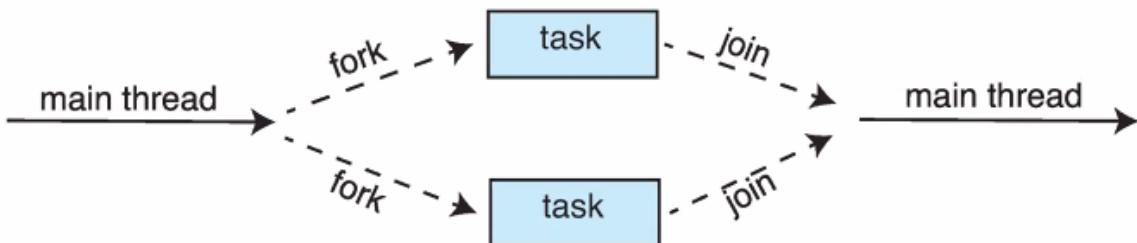
**线程池**的主要思想就是：在进程开始的时候就创建一定量的线程。并加载到线程池中等待工作。当服务器受到请求时，它会唤醒池内的一个线程去处理请求。一旦线程完成了服务，它在回到池中等待再次被唤醒。如果池内没有可用线程，服务器会等待，知道有新的空线程为止。

## fork-join



## Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.



## 信号处理

**Unix信号**用于通知某个进程特定的事件已经发生。比如，I/O就绪，或者用户键盘输入了ctrl+c等。  
这里在了解下**同步信号**和**异步信号**。

和以前的概念一样，同步表示一直在等待信号的处理。异步表示发完信号就可以执行其他操作。一般进程接到信号后，可以有两种处理方法：

- 缺省的信号处理程序。（操作系统定义的默认处理）
- 用户自定义的信号处理程序。（比如异常捕捉等）

这里就会产生一个问题。

1. 信号发给所有的使用的线程。（不同进程的线程）
2. 信号发给某个进程内所有线程。
3. 信号发给某个进程内的某些线程。
4. 规定一个特性线程只处理，接收外部的所有信号。

Unix 传递信号的系统调用为：

`kill(pid_t pid, int signal);`

发送给某个进程的 signal 信号。

`pthread_kill(pthread_t tid, int signal);`

发给指定的线程 signal 信号。

## 线程的取消

线程撤销是在线程完成之前终止线程。

例子：用户按下网页浏览器上的按钮，以停止进一步的加载网页。通常加载网页可能是多个线程，每个图像可能都是一个单独的线程在加载，当用户点下停止按钮，所有的网页加载线程都要被撤销。

需要撤销的线程称为**目标线程**。目标线程的终止一般分为

- **异步撤销**: 一个线程立即终止目标线程。
- **延迟撤销**: 目标线程需要检查它自身是否可以终止, 比如它有依赖其他线程的终止, 自己才能终止。

所以问题来了:

当系统为进程分配资源的时候, 分配到了已撤销的线程怎么办? 比如 线程退出了, 它的子线程和他在并行执行, 但是资源分配到了父线程。还有就是已经撤销的线程正在更新和其他线程共享的数据, 撤销会有问题。

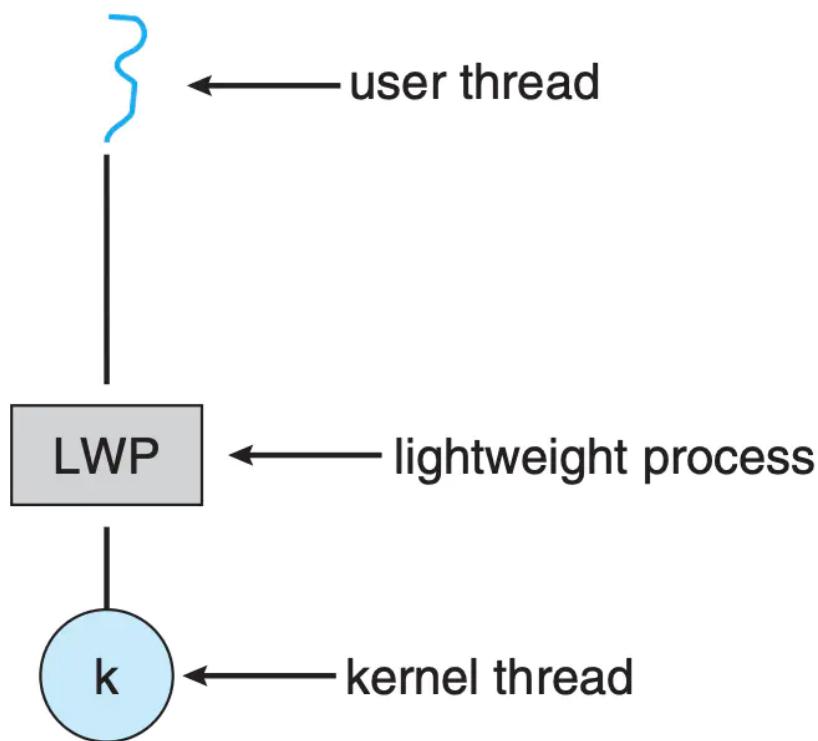
操作系统收回撤销线程的系统资源, 不会收回所有的资源。所以在最后线程撤销的时候, 最后有可能资源泄露。

`pthread_cancel()` 可以发起线程额撤销。这个是Pthread线程库提供的API。但是它并不能解决上面的问题。

## 调度程序激活

多线程编程最后的一个问题就是线程库与内核之间的通信。

许多系统在实现多对多的模型是, 都在用户和内核线程之间加了一个中间数据结构轻量级进程 LWP(LightWeight process)。



**Figure 4.13 Lightweight process (LWP).**

用户线程库与内核之间的一种通信方案称为**调度器激活**。

内核提供一组虚拟处理器LWP给应用程序, 而应用程序可以调度用户线程到任意一个虚拟处理器上。此外, 内核应将有关特事件通知应用程序。这个步骤 称为**回调**。它由线程库通过**回调处理程序**来处理。

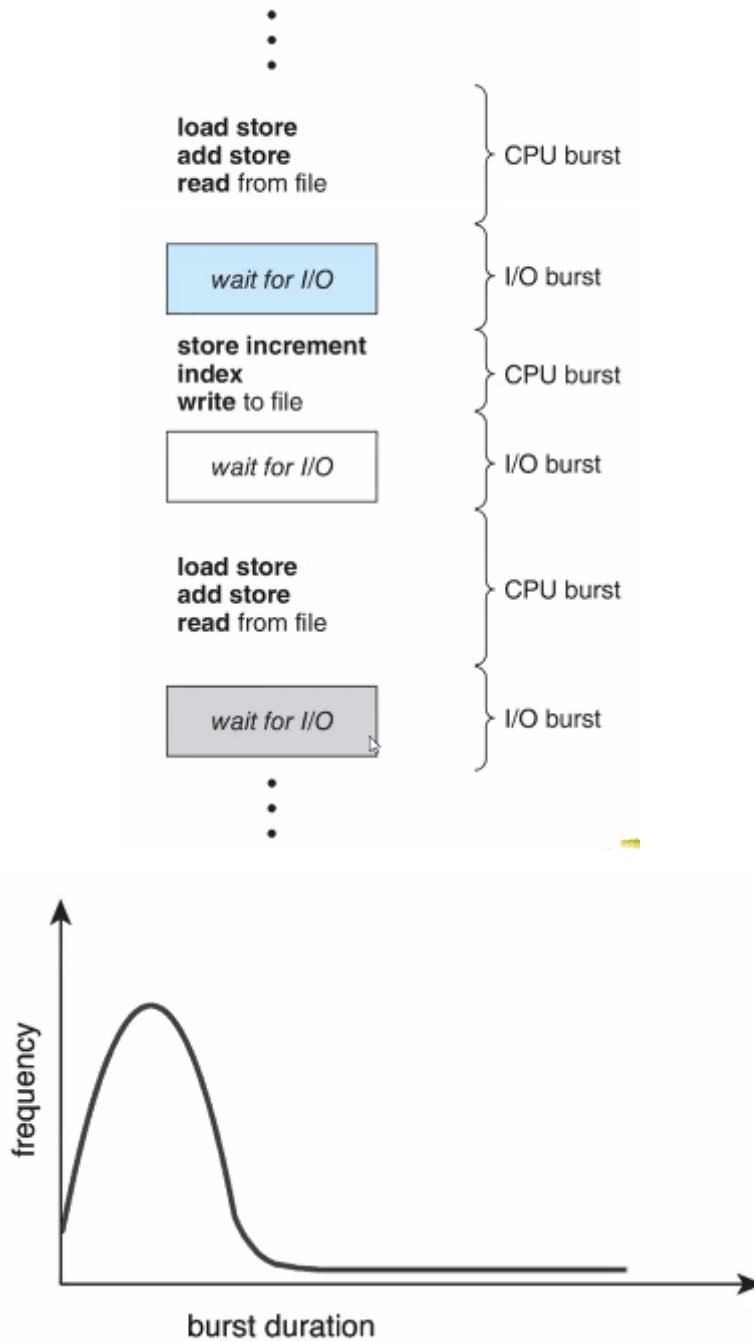
当一个应用程序的线程要阻塞时, 它会触发一个回调的事件。内核向应用程序发出一个回调, 通知它有一个线程将会阻塞并标识特定线程。然后分配一个新的虚拟处理器给应用线程。应用程序在这个新的虚拟处理器上进行回调处理程序, 保存它的阻塞状态, 并释放阻塞线程运行的虚拟处理器。接着, 回调处理函数调度一个新的线程到虚拟处理器上。当阻塞线程等待的事件发生时, 内核向线程库在发出另一个

回调，通知线程库，之前等待的事件发生了。该回调也需要虚拟处理器，内核可能分配一个新的虚拟处理器，或者抢占一个用户线程再次执行会回调处理程序。

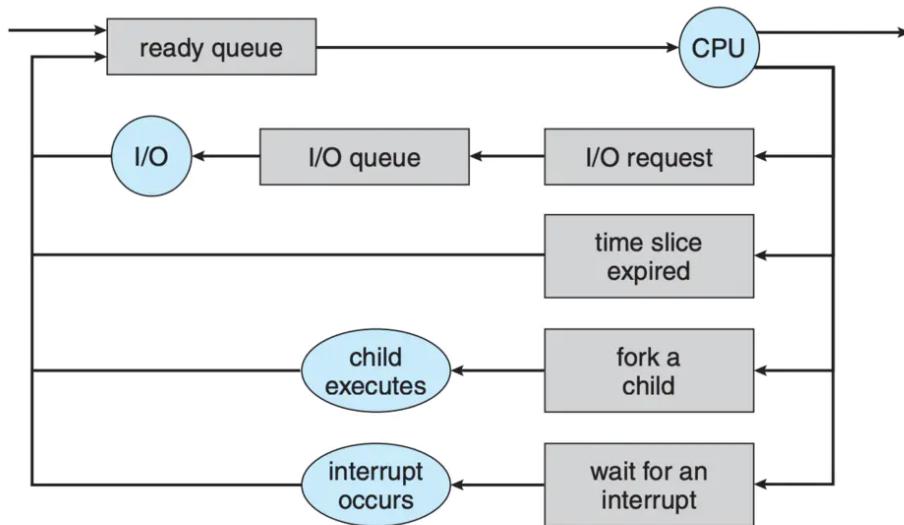
# Ch 05 CPU Scheduling CPU 调度

## 基本概念

CPU burst和I/O burst交替运行



根据之前进程一章中的介绍，进程的执行一般会分为两个执行周期：**CPU执行**和**I/O等待**。

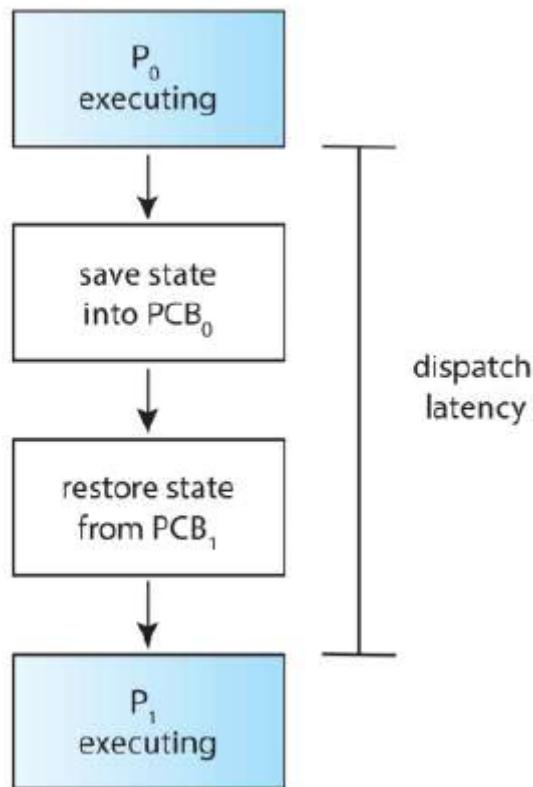


**Figure 3.6** Queueing-diagram representation of process scheduling.

**抢占调度：**在进程执行的过程中，CPU被操作系统中断而执行其他进程的调度，称为抢占调度。

**非抢占调度：**一个进程分配到CPU后，该进程会一直占用CPU，直到它终止或者切换到等待状态。

## Dispatcher



- 分派器模块将CPU的控制权交给了短期调度程序选择的进程。
  - 切换上下文
  - 切换到用户态
  - 跳到用户程序中适当的位置以重新启动该程序
- dispatch latency：调度程序停止一个进程并启动另一个进程所花费的时间

## 衡量指标

- CPU利用率：保持CPU尽可能繁忙
  - 吞吐量：按时间单位完成执行的进程数
  - 周转时间：作业完成时间-作业提交时间
  - 等待时间：进程在就绪队列中等待的时间
  - 响应时间：从提交请求到生成第一个响应所花费的时间，而不是输出（用于分时环境）
- 前两个依赖于硬件环境，通常用后三个衡量调度算法优劣

## 进程调度算法

### 先到先服务 (First-Come First-Served, FCFS)

最简单的调度算法：可以采用FIFO队列（First In First Out）实现。

FCFS调度是非抢占的。

甘特图



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$



$$\text{Avg. Turnaround Time} = (24+27+30)/3=27$$

$$\text{Avg. Response Time: } 17$$

P1\P2\P3先后同时到达

### 最短作业优先 (Shortest-Job-First, SJF)

当CPU变为空闲时，它会执行占用CPU最短的进程。

可以证明SJF的平均等待时间最短

SJF 调度是非抢占的。

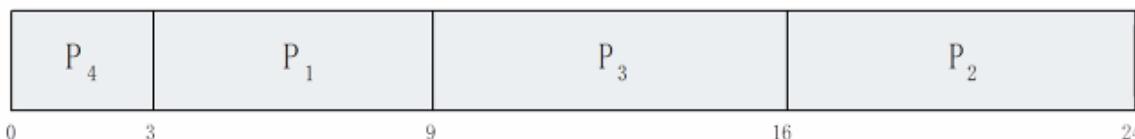


- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$

例子

Process	Arrival Time	Burst Time
$P_1$	0.0	6
$P_2$	2.0	8
$P_3$	4.0	7
$P_4$	5.0	3

### ■ SJF scheduling chart



■ Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$



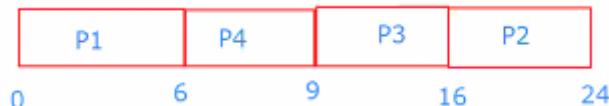
Avg. Turnaround Time =  $(9+24+16+3)/4=13$

Avg. Response Time = 7

甘特图未考虑Arrival time

若考虑Arrival time

$P_1 \rightarrow P_4 \rightarrow P_3 \rightarrow P_2$



平均等待时间 =  $(0 + 1 + 5 + 14)/4 = 5$

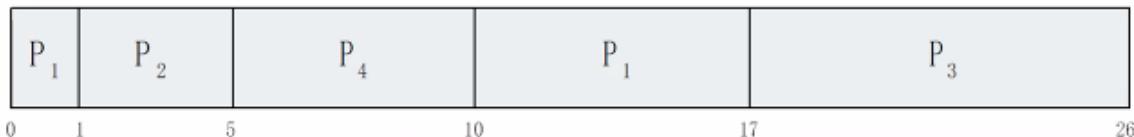
平均周转时间 =  $(6 + 4 + 12 + 22)/4 = 11$

平均响应时间 = 5

**最短剩余时间优先算法SRTF 抢占式**

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

### ■ Preemptive SJF Gantt Chart



■ Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5 \text{ msec}$

平均周转时间=  $(17 + 4 + 24 + 7)/4 = 13$

平均响应时间=  $(0 + 0 + 15 + 2)/4 = 4.25$

## 优先级调度 (priority-scheduling)

- 每个进程都关联一个优先级数字（整数）
- 将CPU分配给优先级最高的进程（UNIX中最小整数=最高优先级，Win中相反）
  - 抢占/非抢占
- SJF是优先级调度，其中优先级的所预测的逆下一个CPU突发时间
- 问题是可能出现饿死：低优先级进程可能永远不会执行
- 解决方案是老化：随着时间的推移，优先级越高处理

### 例子

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

### ■ Priority scheduling Gantt Chart



■ Average waiting time = 8.2 msec

$$\text{平均周转时间} = (1 + 6 + 16 + 18 + 19)/5 = 12$$

$$\text{平均响应时间} = (0 + 1 + 6 + 16 + 18)/5 = 8.2$$

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>	<u>Arrival Time</u>
$P_1$	10	3	0
$P_2$	1	1	1
$P_3$	2	4	2
$P_4$	1	5	3
$P_5$	5	2	4

抢占式的

$$\text{平均等待时间} = (6 + 0 + 14 + 15 + 0)/5 = 7$$

$$\text{平均周转时间} = (16 + 1 + 16 + 16 + 5)/5 = 10.8$$

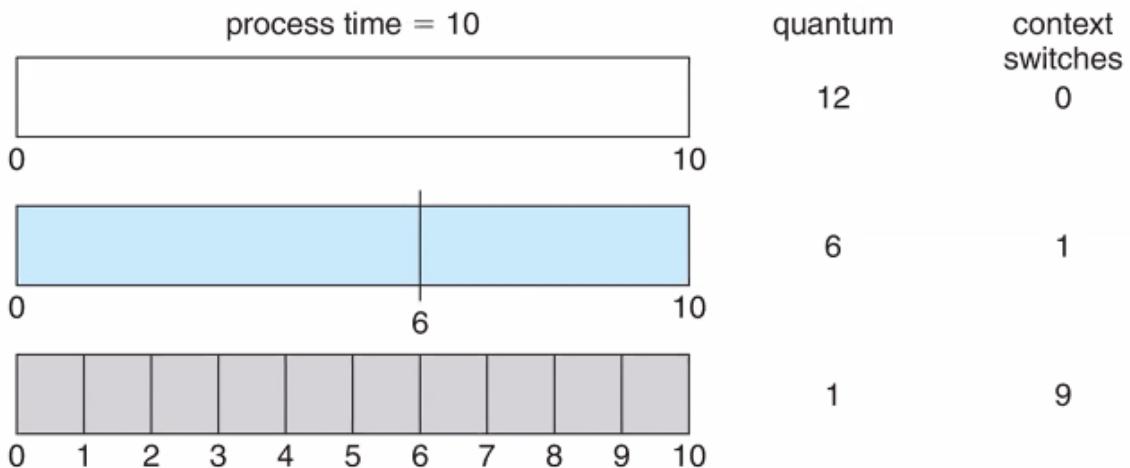
$$\text{平均响应时间} = (0 + 0 + 14 + 15 + 0)/5 = 5.8$$

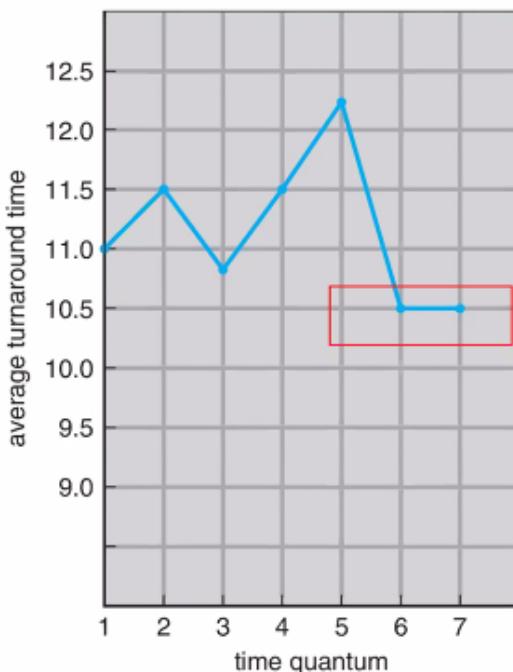
## 时间片轮转调度 (Round-Robin, RR)

- 每个进程占用一小部分CPU时间 (时间片 q)，通常为10到100毫秒。在这段时间之后已用完，该进程将被抢占并添加到准备队列的末尾。
- 如果就绪队列中有n个进程，并且quantum为q，则每个进程一次最多以q个时间单位的块获取CPU时间的 $1/n$ 。没有进程等待超过 $(n-1) \cdot q$ 个时间单位。
- 时器中断每个quantum以安排下一个过程

表现

- $q$ 大  $\Rightarrow$  FIFO
- $q$ 小  $\Rightarrow$  就上下文切换而言， $q$ 相对较大，否则切换开销太高





process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

80% of CPU bursts should be shorter than  $q$

重点关注长任务

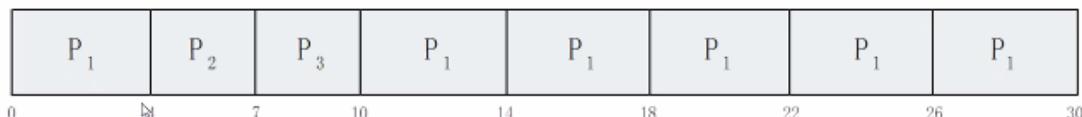
### 例子



## Example of RR with Time Quantum = 4

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
- $q$  usually 10ms to 100ms, context switch < 10 usec



$$\text{平均等待时间} = (6 + 4 + 7)/3 = 5.67$$

$$\text{平均周转时间} = (30 + 7 + 10)/3 = 15.67$$

$$\text{平均响应时间} = (0 + 4 + 7)/3 = 3.67$$

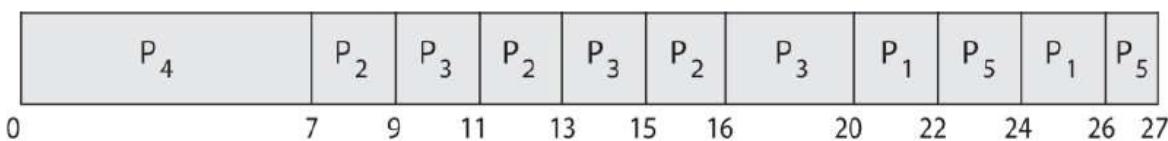
通常，平均周转率高于SJF，但能够更好响应

- $q$  与上下文切换时间相比应该更大
- $q$  通常为10ms到100ms，上下文切换<10微秒

## 带轮询的优先级调度

Process	Burst Time	Priority
$P_1$	4	3
$P_2$	5	2
$P_3$	8	2
$P_4$	7	1
$P_5$	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with 2ms time quantum



$$\text{平均等待时间} = (0 + 11 + 12 + 22 + 24)/5 = 13.8$$

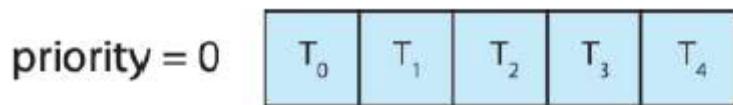
$$\text{平均周转时间} = (7 + 16 + 20 + 26 + 27)/5 = 19.2$$

$$\text{平均响应时间} = (0 + 7 + 9 + 20 + 22)/5 = 11.6$$

## 多级队列调度

将进程分为不同的级别，比如，前台进程和后台进程。这两种类型的进程的响应时间要求不同，进而调度算法也不同。

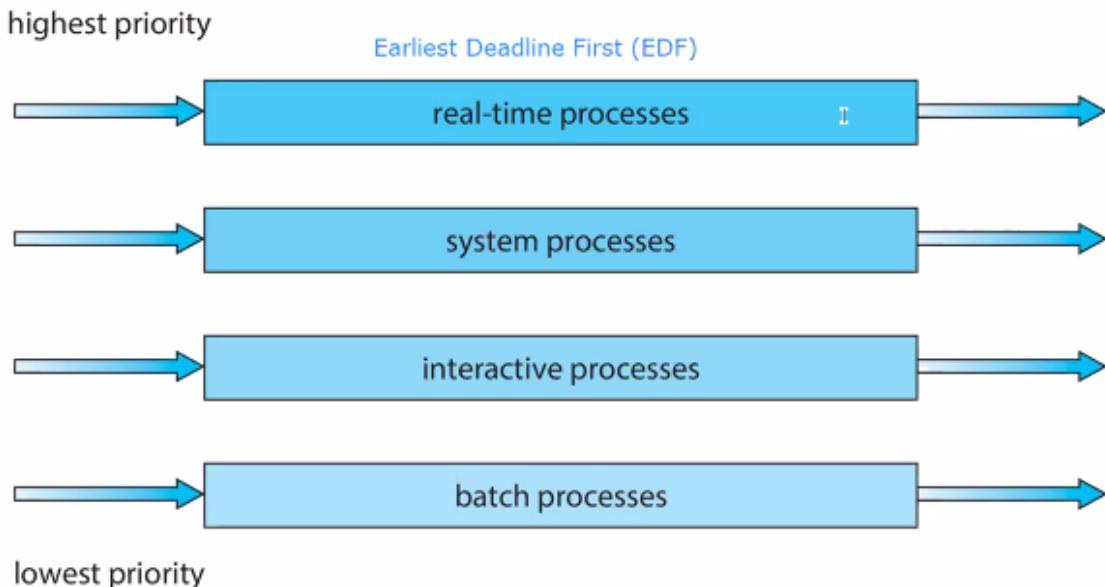
**多级队列调度**是将就绪队列分为多个单独的队列。根据进程属性。然后处于不同队列的调度算法不同。



●  
●  
●



## Prioritization based upon process type



## 线程调度算法

### 竞争范围

由线程库调度的线程，他只会和它所在进程之间的线程进行竞争，这叫做**进程竞争范围-PCS**  
由内核进行调度的线程，它会和系统中所有线程进行竞争，这种叫做**系统竞争范围-SCS**

### Pthreads调度

Pthreads 库的API 实现了两种竞争范围的接口：

- PTHREAD\_SCOPE\_PROCESS: 采用进程竞争范围。
- PTHREAD\_SCOPE\_SYSTEM: 采用系统竞争范围。

## 多处理调度

对于多处理系统，CPU 调度的一种方法是让一个处理器 处理所有的调度决定、I/O 处理以及其他活动，其他的处理器只执行用户代码。这种称谓**非对称多处理**。这种调度比较简单，因为不存在多个CPU共享数据。

还有一种方法是**对称多处理 (SMP)**。即每个处理器自我调度，所有进程都可能处于一个共同的就绪队列，或者每个处理器都有自己私有的就绪队列。这时就需要保证两个处理器不会同时选择同一个进程。

处理器亲和性：当一个进程运行在一个特定的处理器上时，它会进行缓存数据。但是当一个进程在下次执行的时候，切换到了其他的处理器时，这个缓存就无效了。所以大多数SMP系统都会保证进程从一个处理器移到另外一个处理器。这就是处理器的亲和性。

## 负载均衡 Load balancing

SMP系统上，重要的就是保持所有的处理器的负载平衡。否则就会出现，一个或者多个处理器空闲，而其他处理器处于高负载状态。

负责均衡通常有两种办法：**推迁移(push migration)**和**拉迁移(pull migration)**。推迁移指的是一个特定的任务周期性的检查每个处理器的负载，如果发现不平衡，那么通过将进程从超负载处理器推送到空闲的处理器上。对应的，空闲的处理器从忙的处理器上拉取一个任务，就是拉迁移。Linux调度程序实现了这两种技术。

## Real-Time CPU Scheduling 实时CPU调度

- 软实时系统-关键实时任务具有最高优先级，但不能保证何时安排任务
- 硬实时系统-必须在截止日期之前完成任务

## 不同系统调度算法

### Linux

- 优先级优先算法
- 两个优先级范围：分时和实时
- 实时范围从0到99， nice value从100到140
- 映射到全局优先级，数值越小表示优先级越高
- 优先级越高q越大
- 只要时间片中有剩余的时间，即可运行任务（活动）
- 如果没有剩余时间（过期），则在所有其他任务使用其片之前不可运行
- 每个CPU运行队列数据结构中跟踪的所有可运行任务
  - 两个优先级阵列（活动，过期）
  - 任务按优先级索引
  - 当不再活动时，交换阵列
- 运行良好，但交互过程的响应时间较差

### Windows

#### 优先级

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

## 利特尔公式

- $n$ =平均队列长度
- $W$ =排队平均等待时间
- $\lambda$ =排队平均到达率
- Little's定律-在稳定状态下，离开队列的进程必须等于到达的进程，因此： $n=\lambda \times W$

- 对任何调度算法和到达分布有效
- 例如，如果平均每秒有7个进程到达，并且通常有14个进程在队列中，则每个进程的平均等待时间 =2秒

# Ch 06 synchronization tools

---

## 背景

- 进程可以同时执行
- 随时可能会中断，部分完成执行
- 并发访问共享数据可能会导致数据不一致
- 保持数据一致性需要机制来确保合作流程的有序执行
- 问题说明：

假设我们想提供一个解决消费者生产者问题的解决方案，以解决所有缓冲区的问题。为此，我们可以使用一个整数计数器来跟踪完整缓冲区的数量。最初，counter设置为0。生产者在生成新缓冲区后将其递增，消费方在使用缓冲区后将其递减。

## 生产者消费者问题

### 竞态

- counter++ could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- counter-- could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = counter	{register1 = 5}
S1: producer execute register1 = register1 + 1	{register1 = 6}
S2: consumer execute register2 = counter	{register2 = 5}
S3: consumer execute register2 = register2 - 1	{register2 = 4}
S4: producer execute counter = register1	{counter = 6 }
S5: consumer execute counter = register2	{counter = 4}

正常counter=5

## 同步

在之前的学习中，了解了进程能与系统内其他执行进程相互影响。协作进程之间或能直接共享逻辑地址空间（代码和数据），或能通过文件或者消息来共享数据。前一种是通过线程来实现的。

这里以进程之间最常见的同步问题，**生产者-消费者**（也叫**有界缓冲区**）问题来开始讨论吧。

问题描述：假设有一个固定大小的缓冲区，生产者生产数据写入，消费者取出数据进行消费。当缓冲区满的时候，生产者必须等待；当缓冲区为空时，消费者必须等待。

对于这个问题，我们进行深入分析，首先有一个必须共享的缓冲区。所以先进行下面的定义。

```
typedef struct {
    ...
} Item;
#define BUFFER_SIZE 10
Item buffer[BUFFER_SIZE];
int count = 0;      // 缓冲区中的数据量
int in = 0;         // 用来标识生产者存放下一个数据位置
int out = 0;        // 用来标识消费者取出下一个数据位置
```

我们假设缓冲区的数据为BUFFER\_SIZE，则我们可以得到这样的伪代码：

```
// producer

while (true) {
    A = producer();

    // 缓冲区满，等待
    while (BUFFER_SIZE == count) ;

    buffer[in] = A;
    in = (in + 1) % BUFFER_SIZE;
    count++;

}

// consumer

while (true) {
    // 缓冲区空，等待
    while (0 == count) ;

    B = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
}
```

虽然以上代码在各自的程序中各自正确，但是在并发执行时，可能会存在问题，因为count这个数据是两个程序共享的。

"count++" 可以按照机器语言分解：

```
register1 = count;
register1 = register + 1
count = register;
```

相应的"count--" 可以分解为：

```
register2 = count ;
register2 = register2 - 1;
count = register2;
```

在CPU交替执行的时候，会产生三种结果。

$T_0:$	<i>producer</i>	execute	$register_1 = counter$	{ $register_1 = 5$ }
$T_1:$	<i>producer</i>	execute	$register_1 = register_1 + 1$	{ $register_1 = 6$ }
$T_2:$	<i>consumer</i>	execute	$register_2 = counter$	{ $register_2 = 5$ }
$T_3:$	<i>consumer</i>	execute	$register_2 = register_2 - 1$	{ $register_2 = 4$ }
$T_4:$	<i>producer</i>	execute	$counter = register_1$	{ $counter = 6$ }
$T_5:$	<i>consumer</i>	execute	$counter = register_2$	{ $counter = 4$ }

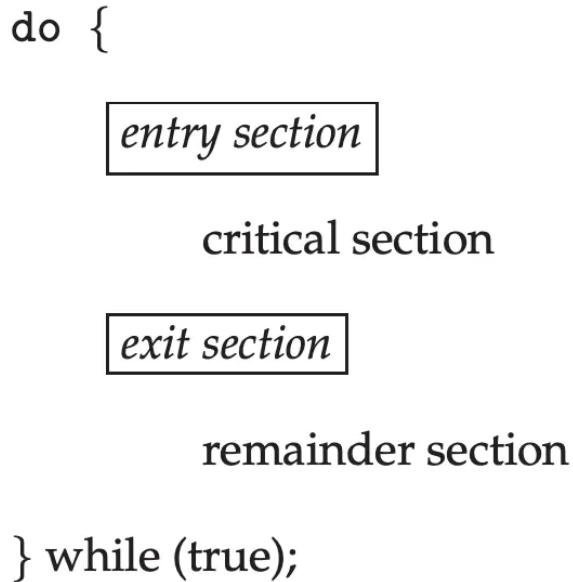
这样的并发执行，我们得到count = 4; 如果T4, T5交换顺序，那么将得到6；

像这种，多个进程并发访问和操作同一数据并且执行结果与特定顺序有关，称为**竞争条件**。

## 临界区问题

**临界区**：假设某个系统有n个进程{P0, P1, P2 .... Pn-1} 每个进程有一段代码，进程在执行的过程中，可能修改公共变量，更新一个表，写一个文件。这段代码叫做临界区。

当有进程在临界区执行的时候，其他进程不允许在它们的临界区内执行。一般的进程，我们会根据他们的结构分为**进入区 (entry section)**，**退出区(exit section)**，**临界区(ctritical section)** 和 **剩余区 (remainder section)**。如图所示：



**Figure 5.1** General structure of a typical process  $P_i$ .

entry section和exit section要保证临界区只有该进程在执行。可以使用blocking

临界区问题的解决方案需要满足三个要求：

1. **互斥 mutual exclusion**: 如果进程 $P_i$ 在其临界区内执行，那么其他进程不能再其临界区执行。
2. **进展 progress**: 如果没有进程在其临界区，并且有进程需要进入临界区，那么只有那些不在剩余区内执行的进程可以参选，以便确定谁能下次进入临界区，而且这种选择不能无限推迟。
3. **有限等待 bounded waiting**: 从一个进程作出进入临界区的请求直到这个请求允许为止，其他进程进入临界区的次数具有上限。

临界区的抢占式和非抢占式两种方法取决于内核是抢占式还是非抢占式

- 抢占式-在内核模式下运行时可以抢占进程
- 非抢占式-运行直到退出内核模式，阻止或自愿产生CPU
- 在内核模式下不会存在竞态

## PeterSon算法

针对临界区问题，PeterSon给出了自己的算法，只要适用于**两个**线程交错执行临界区和剩余区。

假设有两个线程， $P_0$ 和 $P_1$ ，这里只有两个线程，所以这里用 $i$ ,  $j$ 来表示，如果线程 $P_i$ 表示一个线程，那么 $P_j$ 表示另一个线程，两个线程共享数据

```

int turn;
bool flags[2];

```

变量turn表示哪个线程可以进入到临界区，如果 $turn == 0$ ;表示进程 $P_0$ 可以进入缓冲区。

数组flags用于指示进程是否已准备好进入临界区。 $flag[i] = true$ 表示进程 $P_i$ 准备就绪。判断一个线程能否进入缓冲区需要其他线程不在临界区。比如 $P_i$ 线程需要判断， $P_j$ 不在临界区内。他需要两个条件来判断 $flags[j]$  和 $turn == j$ ; 所以可以得到这样的伪码结构：

```

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = false;

    remainder section

} while (true);

```

**Figure 5.2** The structure of process  $P_i$  in Peterson's solution.

$P_j$ 的代码

```

flag[j]=true;
turn=i;|
while(flag[i]&&turn == i);

```

```

flag[j]=false;

```

为了进入临界区，进程 $P_i$ 首先设置 $flags[i] = true;$ 表示第*i*个线程准备进入临界区。

$turn == j;$ 这个是在执行过程中，根据操作系统调度的顺序，来选择的线程的一个值。因为假设 $P_0$ 和 $P_1$ 两个线程同时执行到了这里，并且都对 $flags$ 值进行了设置，那么根据操作系统的调度，有可能两个同时被执行了，虽然这里会有不同的结果，但是最后的 $turn$ 只能有一个，要么为0，要么为1。

$while (flags[j] && turn == j);$ 这个判断主要是看另外一个线程有没有在临界区，如果在临界区，那么等待它退出；(上一步的j不能被替换，因为他会使这个循环直接退出)。两个线程的while代码是互斥的。

$flags[i] = false;$ 当退出临界区的时候，设置这个值，会使另外一个忙等待的线程跳出循环，进入到临界区。

## 硬件同步

对于单处理器环境，临界区问题可简单的加以解决：在修改共享变量的时候只要禁止中断。这样就能保证当前指令正确的执行，且不会被抢占。

然而，在多处理器环境，多处理器的中断禁止会很耗时，因为消息要传递到所有的处理器。消息传递会延迟进入临界区，并且降低多核的效率。另外，如果系统时钟是终端来更新的，那么问题就更复杂了。

因此，许多的现代系统提供特殊的硬件指令，用于检测和修改字的内容，或者用于原子的交换两个字（作为不可中断的指令）。它们通过这些特殊的指令，相对简单的解决临界区问题。

这里仅以`test_and_set()`指令为例来说明：

```

boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}

```

**Figure 5.3** The definition of the `test_and_set()` instruction.

`test_and_set()`的指令是原子的。因此，不论CPU如何并发的执行，在指令上，`test_and_set`是不能被打断的。因此可以把临界区问题这样实现

```

do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);

```

**Figure 5.4** Mutual-exclusion implementation with `test_and_set()`.

## 互斥锁

上面的方法是基于硬件实现的，在临界区问题上，应用程序设计人员能用到最简单的工具就是**互斥锁**（mutex lock）。

利用互斥锁的临界区问题的结构为：

```

do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);

```

**Figure 5.8** Solution to the critical-section problem using mutex locks.

这里可以看到，一个进入临界区时应该得到锁；在退出时释放锁。

在这个结构中，我们给出`acquire()`和`release()`的定义：

```

// available 表示锁是否可用
acquire() {
    // 如果临界区不可用，进行忙等待。
    while (!available) ;
    available = false;
}

release() {
    available = true;
}

```

这里实现的一个缺点是：他需要忙等待。一般在实现时，互斥锁都是采用硬件来实现的。

这里介绍的这个方法处于忙等待。一般把在访问临界区时，不断地要通过忙等待判断锁是否可用的互斥锁称为 **自旋锁** (spin lock)

自旋锁与互斥锁有点类似，但是自旋锁不会引起调用者阻塞，如果自旋锁已经被别的执行单元保持，调用者会一直循环检查该自旋锁的保持者是否已经释放了锁，所以才叫自旋。自旋锁是忙等待，互斥锁是睡眠。

自旋锁的优点：自旋锁一般持有的代码段耗时比较短，当进程获取到自旋锁后，不用进行上下文的切换。

自旋锁一般用于多核系统中，自旋锁在一个处理器上“旋转”，其他线程在其他核上执行命令。

## 信号量

信号量是线程同步中另一种方法。

通常信号量是一个整数变量。信号量一般由**两个操作**`wait()` 和`signal()`，`wait()`操作称为**P操作**(荷兰语: proberen, 测试)，`signal()`称为**V操作**(荷兰语: verhogen, 增加)。

信号量一般根据使用情况分为**计数信号量**和**二进制信号量**。

二进制信号量可以当做互斥锁来用。

计数信号量可以用于控制访问具有多个实例的某种资源。

P操作：当一个线程执行操作`wait()`并且发现信号量值不为正的时候，他必须等待，这里的等待不是忙等待，而是阻塞自己，把自己放入到和信号量有关的队列中。

V操作：当阻塞的线程等到其他线程执行了`signal()`操作后，线程会被`wakeup()`，将它从I/O队列加入到就绪队列中。

忙等待的信号量的实现是：

```

wait(s) {
    while (s <= 0)
        ;    // busy wait
    s--;
}

signal(s) {
    s++;
}

```

阻塞的信号量的实现为：

```

typedef struct {
    int value;
    struct process * list; // 进程的队列
} semaphore;

```

```

wait (semaphore& s) {
    s.value -- ;
    // 如果value 的值小于它们的使用量，把进程加入到队列中进行挂起
    if (s.value < 0) {
        /* add this process to s.list */
        block();
    }
}

signal(semaphore & s) {
    s.value++;
    // 如果s->value的绝对值 表示挂起的队列的长度，如果挂起队列有值，就唤醒一个进程
    if (s->value <= 0) {
        remove a process from s.list;
        wakeup();
    }
}

```

## 死锁和饥饿

虽然上面的的进程队列，可以避免忙等待的情况，但是具有等待队列的信号量实现可能导致：两个进程或者多个进程，无限等待一个事件，而该事件只有由等待的进程之一来产生，就会出现**死锁**。

看下死锁的情况：

假设有一个系统，它有两个进程，P0和P1，每个访问共享信号的S和Q，这两个的初始值都为1。

$P_0$	$P_1$
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
.	.
.	.
.	.
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

这里就会看到， $P_0$ 进程在等待 $P_1$ 进程的S,Q,  $P_1$ 进程在等待进程的 $P_0$ 的Q,S；由于执行顺序的不确定，就会导致死锁。

除此之外，如果进程**无限等待**信号量而得不到执行，那么我们称这类问题为**饥饿**。即信号量有关的信号有关的队列按照后进先出的顺序来增加和删除，第一个信号量将一直得不到响应。

## 生产者消费者问题

生产者与消费者是互相合作的关系，我们说，为完成某种任务而建立的多个进程，这些进程因为要在某些位置上协调他们的工作次序而等待。

比如：A进程要工作必须等待B进程的一个结果，如果仅仅是A进程单方面的需要B进程的一个结果，那这张制约关系就是单方向的（此处的单方向和下面双方向是我个人的理解而用的词汇），如果同时B进程的工作也需要A进程工作的结果，那么这就是双方向的互相制约了。

而生产者-消费者问题里的同步关系我认为是双方向的，原因如下：**生产者要生产的前提是缓冲区没满，而缓冲区没满是消费者运行后的结果，同样消费者要运行的前提是缓冲区不空，而缓冲区不空是生产者不断生成的结果。所以，按本人的理解就是双方向制约的关系。**

也许有人好奇为什么要搞这么细，原因很简答，在指定同步关系的信号量的时候一个制约就是一个信号量，本题的同步关系需要两个信号量。一个是消费者通知生产者是否可以生产的“有空位”信号量——empty，一个是生产者通知消费者要消费的“有信息”信号量——full

```
mutex=1
```

```
full=0
```

```
empty=n
```

```
wait(s) {  
    while (s <= 0)  
        ; // busy wait  
    s--;  
}  
signal(s) {  
    s++;  
}
```

生产者

```
while (true) {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
}
```

消费者

```
while (true) {  
    wait(full);  
    wait(mutex);  
    ...  
    /* remove an item from buffer to  
    next_consumed */  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    /* consume the item in next_consumed */  
    ...  
}
```

## 读者-作者问题

问题描述：

假设一个数据库为多个并发线程所共享。有的线程只读，有的线程可以读写。因此要求：①允许多个读者可以同时对文件执行读操作；②只允许一个写者往文件中写信息；③读写互斥

```
rw_mutex=1
```

```
mutex=1
```

```
read_count=0
```

设置信号量read\_count为计数器，用来记录当前读者数量，初值为0；  
设置mutex为互斥信号量，用于保护更新read\_count变量时的互斥；  
设置互斥信号量rw\_mutex用于保证读者和写者的互斥访问

```
wait(s) {
    while (s <= 0)
        ;      // busy wait
    s--;
}
signal(s) {
    s++;
}
```

作者

```
while (true) {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
}
```

读者

```
while (true){
    wait(mutex);
    read_count++;
    if (read_count == 1) //若不为1则说明有其他读者，此时可读
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
}
```

**哲学家就餐问题**



问题描述：

由Dijkstra提出并解决的哲学家就餐问题是典型的同步问题。该问题描述的是五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替的进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。

用信号量解法：

每只筷子都用一个信号量来表示，一个哲学家通过执行wait()试图获取左边和右边的筷子，它会通过signal()释放筷子。因此：

```
semaphore  chopstick[5] = {1,1,1,1,1}

// 第i个哲学家的伪码表示为:
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    // eating
    signal(chopstick[(i+1)%5]);
    signal(chopstick[i]);
}
```

虽然这一解决方案保证两个邻居不能同时进食，但是它可能导致死锁，因此还是应被拒绝的。假若所有5个哲学家同时饥饿并拿起左边的筷子。所有筷子的信号量现在均为0。当每个哲学家试图拿右边的筷子时，他会被永远推迟。

死锁问题有多种可能的补救措施：

- 允许最多4个哲学家同时坐在桌子上。
- 只有一个哲学家的两根筷子都可用时，他才能拿起它们（他必须在临界区内拿起两根筷子）。
- 使用非对称解决方案。即单号的哲学家先拿起左边的筷子，接着右边的筷子；而双号的哲学家先拿起右边的筷子，接着左边的筷子。

```
monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING } state [5] ;
    condition self [5]; //每位哲学家条件变量

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
```

```

        self[i].wait;
    }

void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
}

void test (int i) {
    if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) && (state[(i + 1)
% 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}

```

哲学家i的代码

```

DiningPhilosophers.pickup(i);
/** EAT **/
DiningPhilosophers.putdown(i);

```

不会死锁，可能饥饿

## 优先级反转

假设有三个进程，L，M和H，他们的优先级顺序为L< M < H。假定进程H需要资源R，而R被进程L访问。通常进程H将等待L用完资源R，但是在假设M在这个过程中进入了可运行状态，从而抢占了L，从而导致了进程优先级低的M影响了H的执行，这种问题就是**优先级翻转**。

在解决这类问题的时候，采用了**优先级继承协议**。

比如：上面的例子中，优先级继承协议将允许有关进程L临时继承进程的H的优先级，从而防止M抢占执行。当进程L用完资源R后，它将放弃继承的进程H的优先级，从而采用原来的优先级。

## 管程 Monitor

利用信号量可以同一种方便有效的进程同步机制，但是他们产生的错误也是难以检测到的。

为了解决这类问题，操作系统定义了管程的概念。

管程其实是一种语法定义的数据和函数的集合。就类似于一个函数，但是它可以保证在管程的这段结构代码里，每次只有一个进程在管程内处于活动状态。

---

```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        . . .
    }

    function P2 ( . . . ) {
        . . .
    }

    .

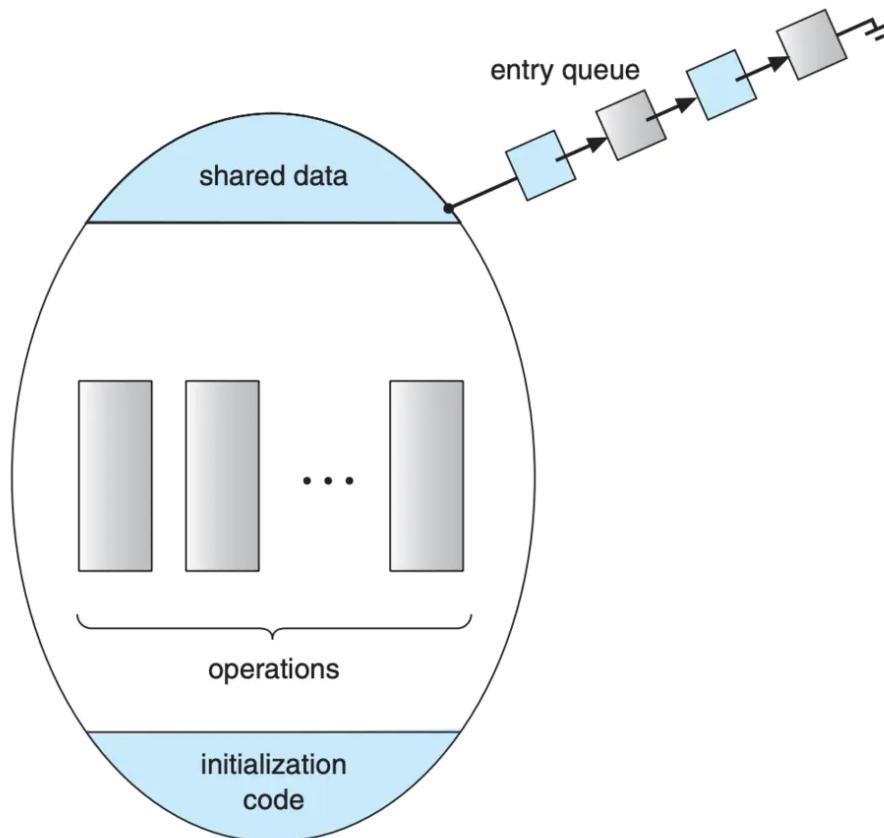
    .

    function Pn ( . . . ) {
        . . .
    }

    initialization_code ( . . . ) {
        . . .
    }
}
```

**Figure 5.15** Syntax of a monitor.

而且管程在进行同步时，是下面的结构，一次只有一个管程在代码段中执行，其他都在入口队列中排队。



**Figure 5.16** Schematic view of a monitor.

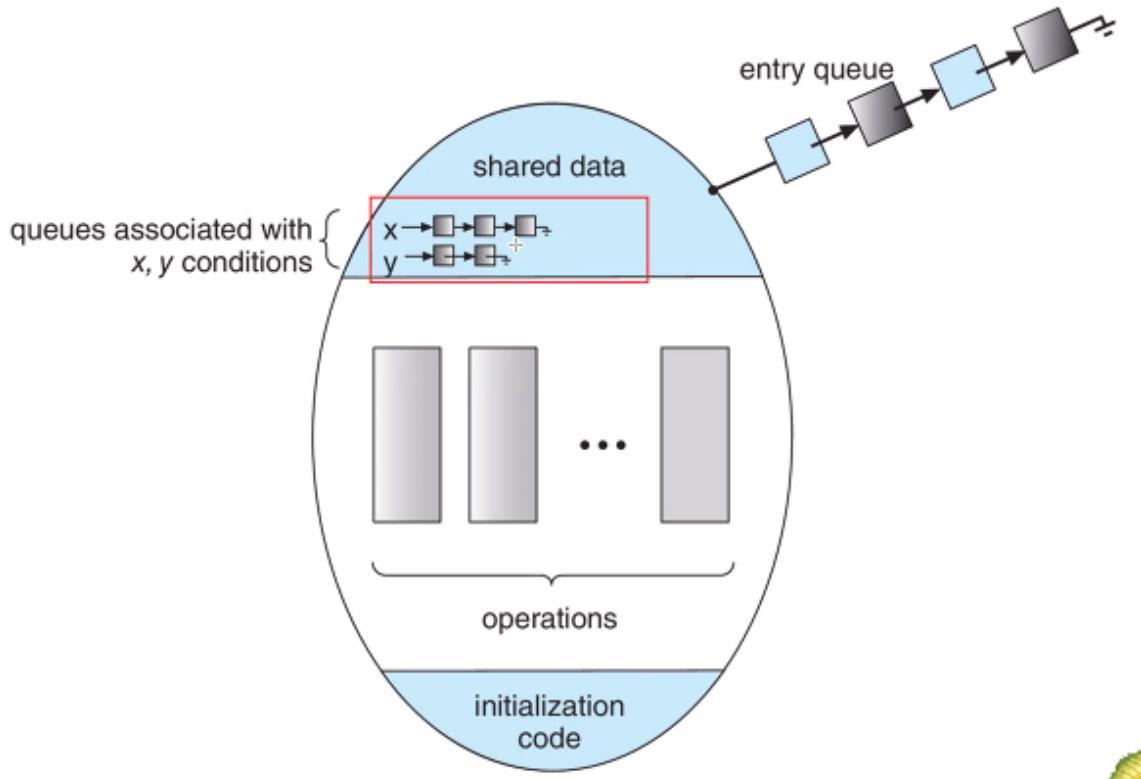
进入管程时的互斥是由编译器负责，但通常是用一个互斥量的方式来进行的。虽然管程提供了一种实现互斥的方法，但是，当进程无法继续运行的是被阻塞。比如：在生产者-消费者中，我们怎么让生产者在缓冲区满的时候阻塞呢？

解决的办法就是，一个新的同步机制**条件变量**；

条件变量一般只有两个操作：wait()和signal();

当一个管程无法继续运行时，它会在某个条件变量上执行wait();该操作使得调用线程自身阻塞直到其他管程执行signal(), 并且还将另外一个以前在等待管程之外的线程调入管程。

### 条件变量



一般条件变量和互斥量是一起使用的。

这种模式用于让一个线程锁住一个互斥量，然后当它不能获得某个结果的时候，挂起并等待一个条件变量。

比如生产者消费者中，假设生产者缓冲区满了，需要阻塞，那么可以使用条件变量将其阻塞。

### ■ Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0;
```

### ■ Each function **F** will be replaced by

```
wait(mutex);
...
body of F;
...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

### ■ Mutual exclusion within a monitor is ensured

next是指针

next\_count指队列中剩余元素

- For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation **x.wait()** can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```



条件变量的wait操作一般干了三件事：

- 1、给互斥锁解锁
- 2、把调用线程投入睡眠，直到另外某个线程就本条件调用signal。
- 3、然后，在返回前重新给互斥锁上锁（没有获得锁时一直阻塞在这里）

## Ch 08 死锁 Deadlocks

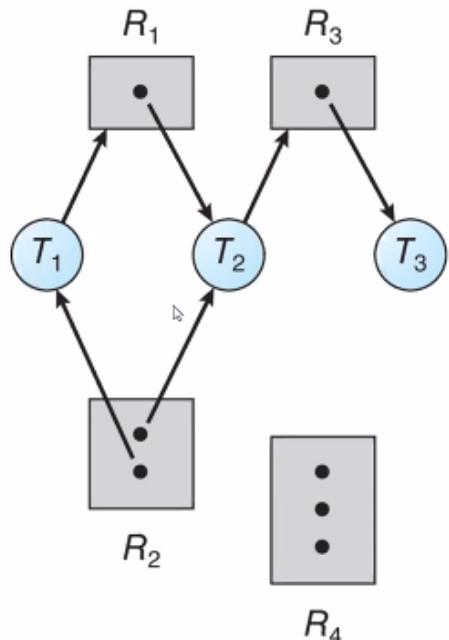
如果在一个系统中一下四个同时满足，那么就有可能引起死锁：

- 互斥。必须至少有一个资源只能允许一个线程使用。
- 占用并等待。一个线程应该占有一个资源并且的等待其他资源。
- 非抢占。资源不能被抢占。
- 循环等待。有一组等待进程(P0,P1,...,Pn), P0等待的资源P1占用, P1的资源P2占用..., Pn等待的资源P0占用, 形成一个死循环。

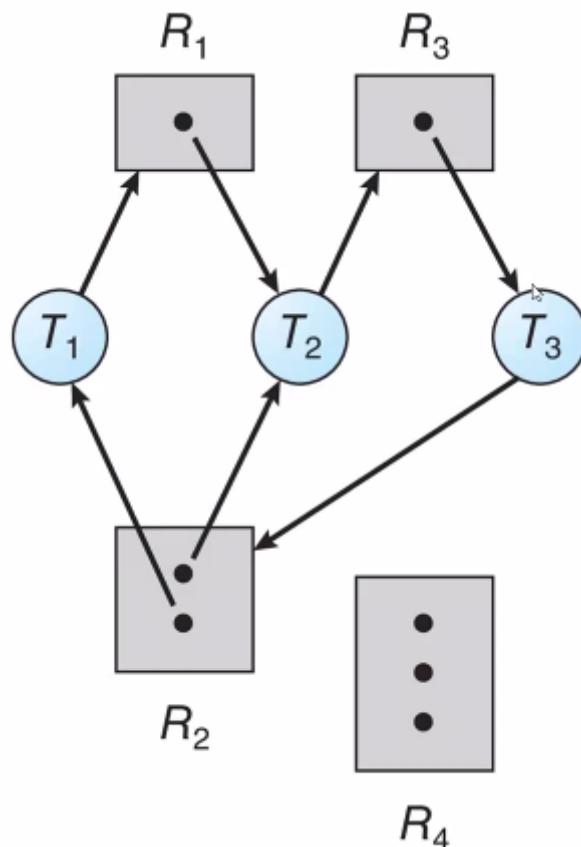
## 资源分配图算法

### 例子

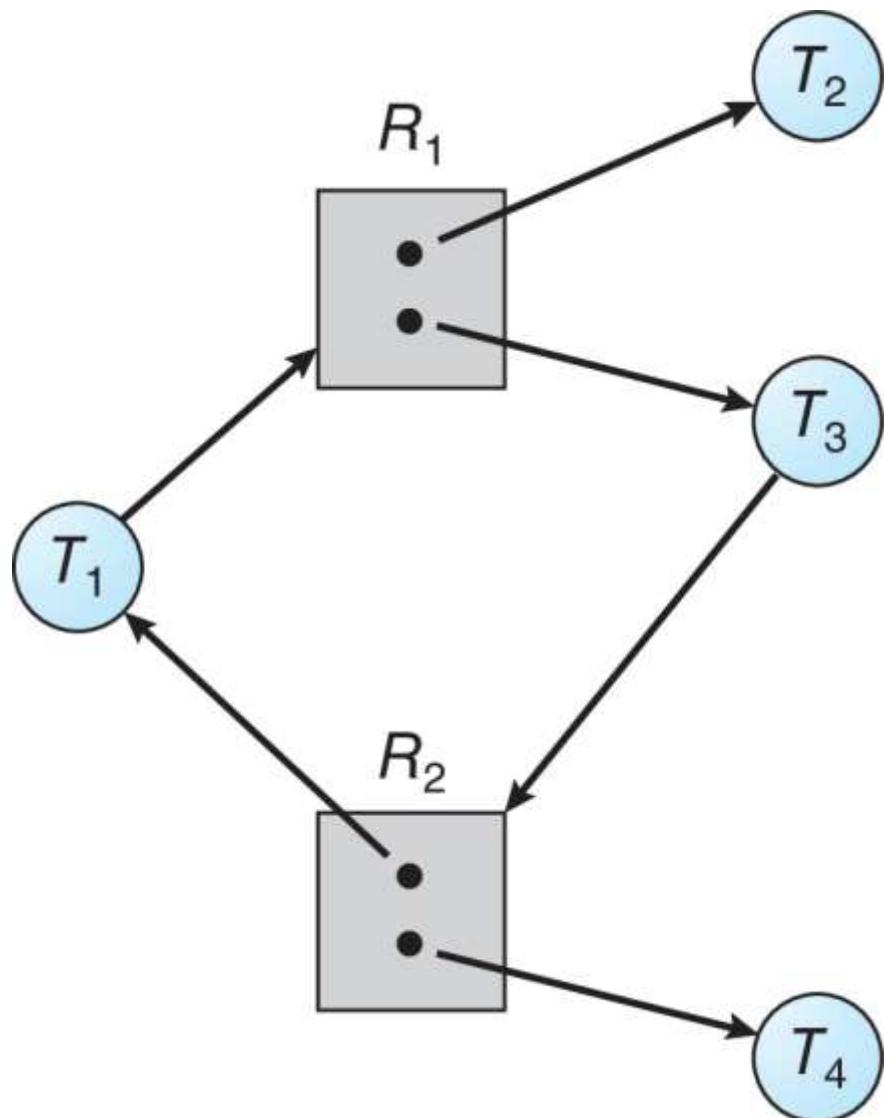
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



不会死锁，因为T3可以执行完毕释放掉，之后依次解锁T2和T1



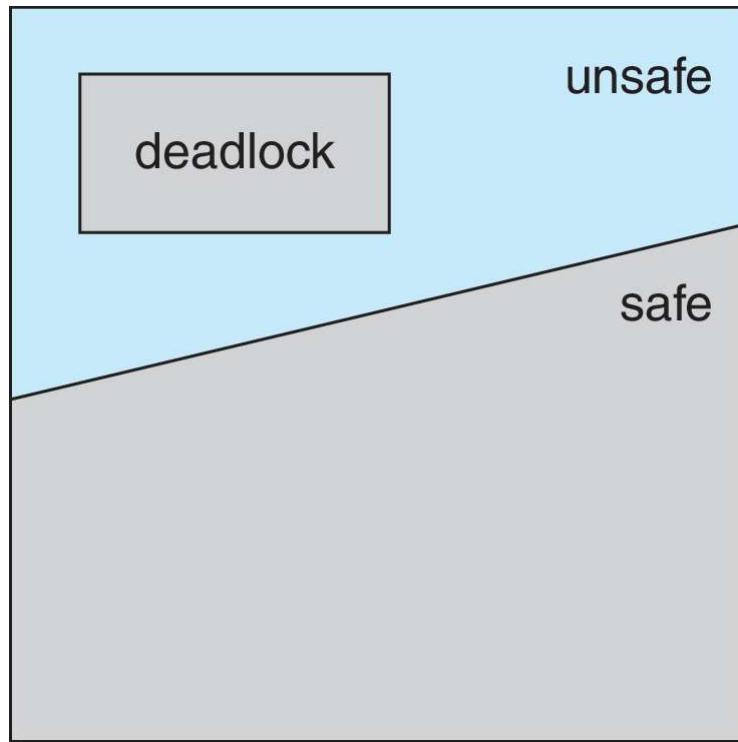
会死锁，形成有向全连接图



虽然有有向全连接图，但不会死锁。因为T2或T4释放掉R1或R2后锁可以解开。

## 结论

- 如果无有向全连接图则无死锁
- 如果存在有向全连接图
  - 如果每种资源类型仅一个实例，则一定死锁
  - 如果每种资源类型有多个实例，则可能出现死锁
- 如果系统处于安全状态，则无死锁
- 如果系统处于不安全状态，则可能出现死锁
  - $RAG \rightarrow \text{no cycle} \rightarrow \text{no deadlock}$   
 $\rightarrow \text{safe state}$
  - $RAG \rightarrow \text{cycle} \rightarrow \text{possibly deadlock} \rightarrow \text{unsafe state}$
- avoidance->确保系统永远不会进入不安全状态。



## 死锁预防

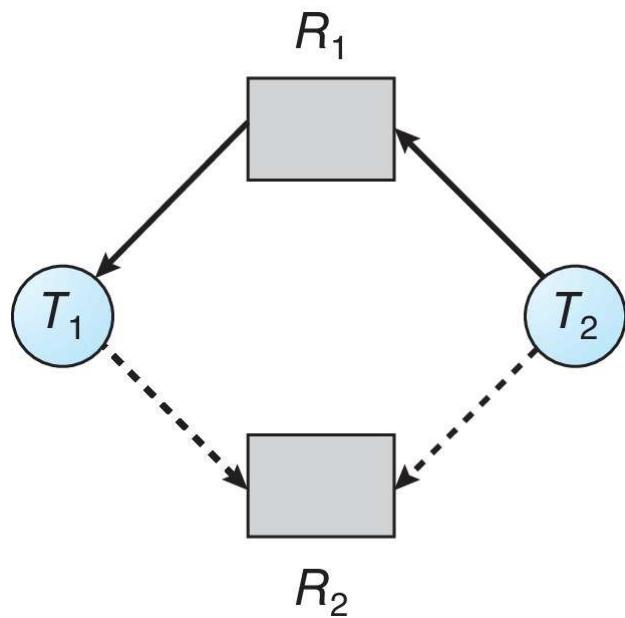
使死锁的四个必要条件之一无效即可

- 互斥：可共享资源（例如，只读文件）可以不互斥；不可共享的资源无法实现，不考虑
- 持有和等待：必须保证进程在请求资源时不会持有任何其他资源
  - 要求进程在开始执行之前请求并分配其所有资源，或者仅在进程未分配任何资源时才允许进程请求资源
  - 资源利用率低；可能饿死
  - 不考虑
- 非抢占
  - 如果一个拥有某些资源的进程请求另一个不能立即分配给它的资源，那么当前持有的所有资源将被释放
  - 将抢占资源添加到进程正在等待的资源列表中
  - 仅当进程可以重新获得其旧资源以及所请求的新资源时，它才会重新启动
- 循环等待：强制所有资源类型进行总排序，并要求每个进程按递增的顺序请求资源
  - 使循环等待条件无效最常见。只需为每个资源（即互斥锁）分配一个唯一的编号。进程必须按顺序获取资源。
  - 使用链表实现

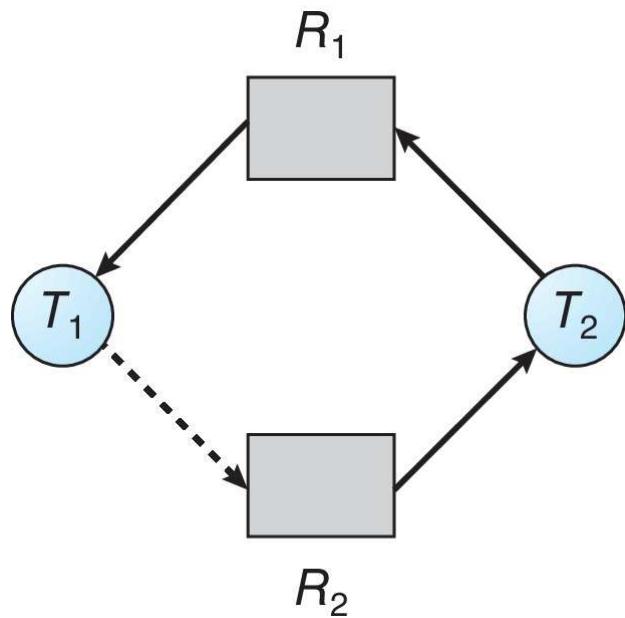
## 死锁避免 Avoidance

要求系统具有一些其他可用的先验信息

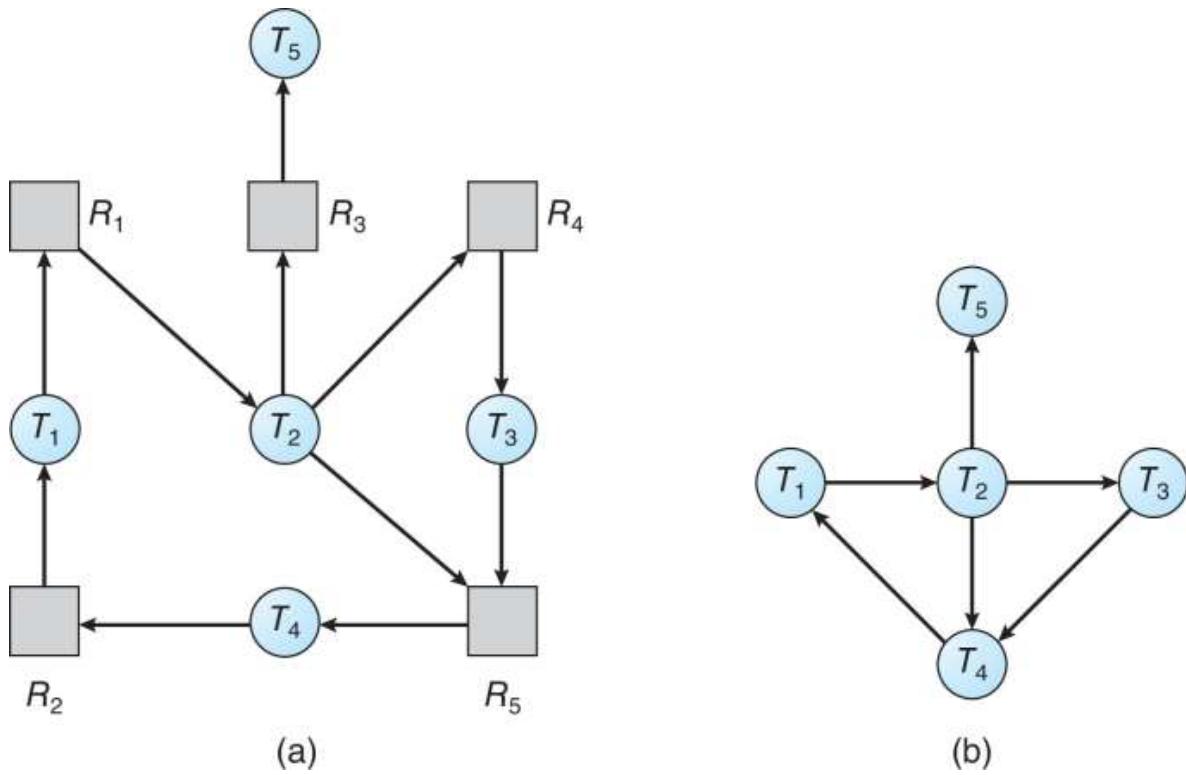
- 最简单，最有用的模型要求每个进程声明它可能需要的每种类型的最大资源数量
- 算法会动态检查资源分配状态，以确保永远不会出现循环等待条件
- 资源分配状态由可用和已分配资源的数量以及进程的最大需求定义
- 并行变串行



不会死锁



可能死锁



找到中心节点 $T_2$

先并行执行 $T_1$   $T_3$   $T_4$   $T_5$

再执行 $T_2$

## 资源分配图的作用

- 判断系统是否safe/unsafe
- avoidance中要申请资源前用虚线进行看是否会发生死锁，死锁避免
- wait for graph中牺牲中心节点

## 银行家算法

资源分为四种类型

- 可用资源：当前可用的空闲资源数
- 最大资源：每个进程请求的最大资源
- 已分配资源：已经分配给某进程的资源
- 需要资源：某进程可能需要的更多资源，最大资源-已分配资源

## 例子

- 5 processes  $P_0$  through  $P_4$ ;
- 3 resource types:  
 $A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>			
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	Need
$P_0$	0	1	0	7	5	3	3	3	2	$P_0\ 743$
$P_1$	2	0	0	3	2	2	$P_1\ 122$			
$P_2$	3	0	2	9	0	2	$P_2\ 600$			
$P_3$	2	1	1	2	2	2	$P_3\ 011$			
$P_4$	0	0	2	4	3	3	$P_4\ 431$			

$P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_0$

- $P_1$  执行之后 532
- $P_3$  执行之后 743
- $P_4$  执行之后 745
- $P_2$  执行之后 10 4 7
- 该路径可行

## 终止进程来解决死锁

- 中止所有陷入僵局的进程
- 一次中止一个过程，直到消除死锁周期
- 我们应该选择哪个顺序来终止进程？
  - 程序的优先级
  - 计算了多长时间，还要多少时间完
  - 进程使用的资源
  - 进程完成需要的资源
  - 需要终止多少个进程
  - 过程是交互式还是批处理？

## 抢占进程来解决死锁

选择最小牺牲的进程

回滚到之前的安全状态

有的进程一直被牺牲，所以需要提高优先级

## 进程同步例题

1. Assume a museum can contain 5,000 people, and it has one entrance and one exit. Only one people can pass each entrance or exit simultaneously. The description of visit process is as below,[↓](#)

Visit Process P[↓](#)

```
{↓  
...↓
```

Enter the museum;[↓](#)

```
...↓  
Visit the museum;↓
```

```
...↓  
Exit;↓  
...↓  
}↓
```



Add necessary semaphores and wait/signal operations to provide synchronization for different processes. Then write the pseudo code and illustrate the meaning and initialization values of various semaphores. (4')[↓](#)

[↓](#)

```
enter = 1;  
exit = 1;  
empty = 5000;  
Visit Process P  
{  
...  
wait(empty);  
wait(enter);  
Enter the museum;  
signal(enter);  
...  
visit the museum;  
...  
wait(exit);  
Exit;  
signal(exit);  
signal(empty);  
...  
}
```

题中描述只有进出博物馆两个过程是互斥的，所以对这两个过程加锁，此外博物馆一共可容纳5000人，故需要对总人数作出限制。

## Ch 09 主存 main memory

### 基本概念

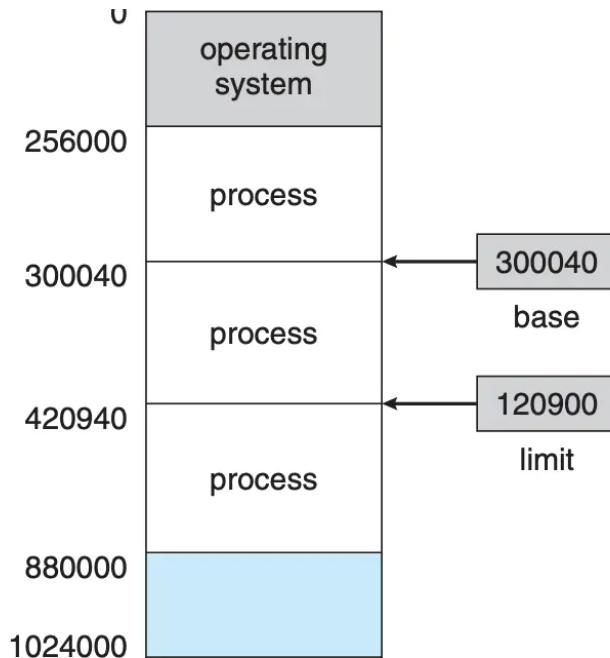
内存是现在计算机运行的核心。CPU可以直接访问的通用存储只有**内存**和处理器的**内置寄存器**。机器指令可以用内存地址作为参数，但是磁盘地址不可以。

在访问速度上，寄存器的内容一般都可以在一个时钟周期解释并执行完。但是对于内存，可能需要多个时钟周期。所以为了访问速度上能更加快速，一般会有**高速缓存**。

在系统安全上，首先，用户的进程不能影响操作系统的执行；在多用户系统上，还应该保护不同的进程之间不能互相影响。所以一般会有专门的硬件方式来进行保护。

为了进程之间不会相互影响，首先，我们需要确保每个进程都有一个单独的内存空间。单独的进程内存空间可以保护进程不互相影响。

为了分开内存空间，我们需要能够确定一个进程可以访问的合法地址的范围；**并且确保该进程只能访问这些合法地址**。一般通过两个寄存器来实现，**基址base address寄存器和变址limit address寄存器**。如下图，如果基址是300040，而变址寄存器为120900，那么程序可以合法访问到的地址为300040 ~ 420939的所包含的地址。



**Figure 8.1 A base and a limit register define a logical address space.**

### 地址绑定

将逻辑地址与物理地址通过映射关联起来

通常，程序以二进制可执行文件的形式存储在磁盘上。为了执行，程序被调入内存并放入进程空间内。

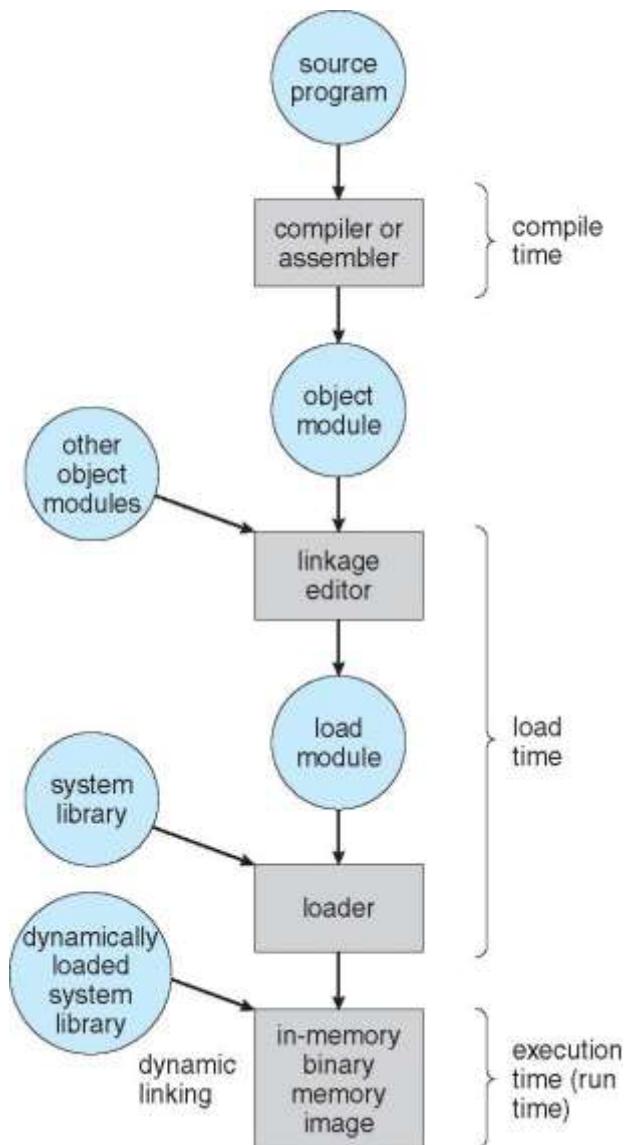
根据所使用的内存管理方案，进程在执行时，可以在磁盘和内存之间移动。在磁盘上等待调入内存以便执行的进程形成输入队列（input queue）。

通常的步骤是从输入队列中选取一个进程并装入内存。进程在执行时，会访问内存中的指令和数据。最后，进程终止，其地址空间将被释放。

许多系统允许用户进程放在物理地址的任意位置。这种组合方式会影响用户程序能够使用的地址空间。在绝大多数情况下，用户程序在执行前，会经过好几个步骤，在这些步骤中，地址可能有不同的表示形式，源程序中的地址通常是用符号（如count）来表示，编译器通常将这些符号地址绑定（bind）在可重定位的地址（如：从本模块开始的第14字节）。链接程序或加载程序再将这些可重定位的地址绑定成绝对地址（如74014）。每次绑定都是从一个地址空间到另一地址空间的映射。

通常，将指令与数据绑定到内存地址有以下几种情况：

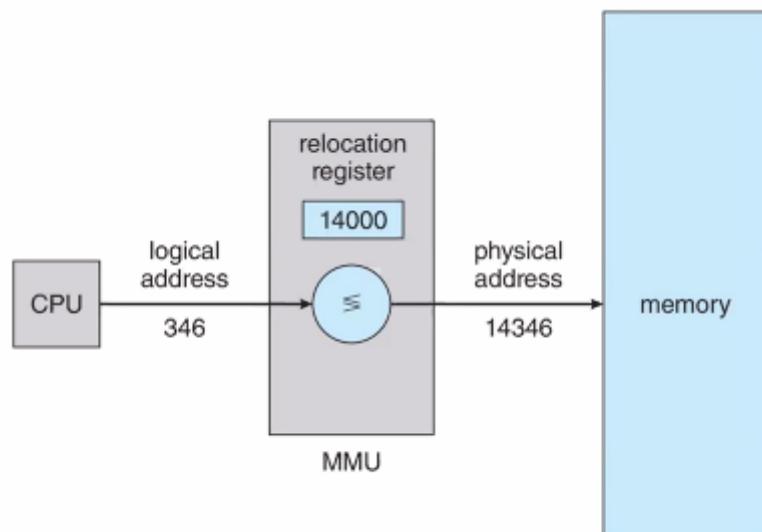
- 编译时（compile time）：如果编译时就知道进程将在内存中的驻留地址，那么就可以生成绝对代码（absolute code）。如果将来开始地址发生变化，那么就必须重新编译代码。
- 加载时（load time）：当编译时不知道进程将驻留在内存的什么地方，那么编译器就必须生成可重定位代码（reloadable code）。绑定会延迟到加载时才进行。如果开始地址发生变化。只需要重新加载用户代码已引入改变值。
- 执行时（execution time）：如果进程在执行时可以从一个内存段移到另一个内存段，那么绑定必须延迟到执行时才发生。绝大多数通用计算机操作系统采用这种方法。



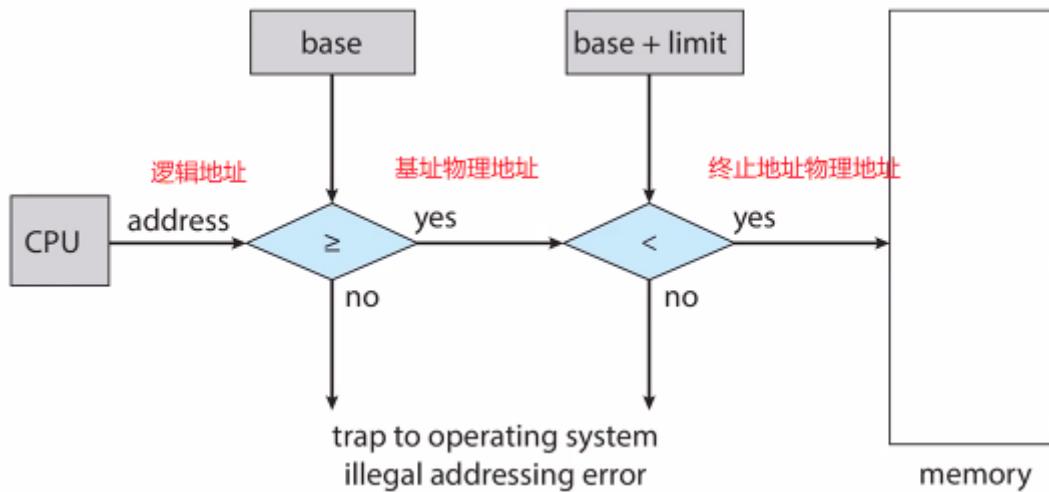
### 逻辑地址空间和物理地址空间

在程序执行的时候，从程序中加载的地址，称为逻辑地址。它是一个照片，我可以用手指出哪里是这个人的眼睛，耳朵，鼻子，但是这只是在照片的位置。这个人本人的肉体才是真正存在的东西。物理地址就好比这个人的肉体，是一个实际存在的东西。

加载时的地址绑定方法生成一个相同的逻辑地址和物理地址。然而，执行时，逻辑地址和物理内存上的地址不一定是一样的。后面内存分配会讲到。



逻辑地址经过重定向定位到物理地址



为何使用两套地址空间?

- CPU register 空间有限, 不能存储太大的地址
- 每台机器硬件配置不同, 物理地址大小不同
- 用物理地址编程会给程序员开发带来较大难度。

### 动态加载和动态链接

动态加载指的是一个进程只有在被调用时才被加载, 而不是一次性把所有的进程全部放进内存中。动态链接指的是不同的文件可能共享同一个模块, 可以仅加载一次, 在其他文件用到时, 采用动态链接, 就不需要重新加载。经常被称为共享库 (shared library)。

动态链接库的好处是, 一个库的更新, 所有使用它的程序都会自动使用新的版本。

## 内存分配

我们通常需要把多个进程同时放在内存中, 因此我们需要进程内存分配。一般内存分为两个区域: 一个用于驻留在操作系统中, 另一个用于用户进程。操作系统可以放在高内存位置, 也可以放在低内存位置, 影响这一决定的通常为中断向量的位置。一般PC是放在低内存位置的。

### 分区

内存分配最简单的分配方式就是将内存分为多个**固定大小的分区**。每个分区可以包含一个进程, 因此, 多道程序设计的受限于分区数。使用这种多分区方法, 那么当一个分区空闲时, 可以从进程的输入队列中选择一个进程, 加载到空闲分区。虽然这种方案在现在的操作系统中已经不用, 但是它的思想为后面很多内存分配提供了思想。

除此之外, 还有一个**可变分区**方案, 它的主要思想是:

操作系统有一个表, 用于记录哪些内存可用, 哪些内存已经使用。

开始, 所有的内存都有可用于用户进程, 因此可以看做是一个很大的内存。随着进程进入系统, 操作系统根据所有进程的内存需求和现有的内存情况, 决定哪些系统可分配内存。

在任何时候, 都有一个可用块大小的列表和一个输入队列。操作系统根据调度算法来对输入队列进行排序。内存不断地分配给进程, 直到下一个进程的内存需求不能满足为止, 这时没有足够大的可用块来加载进程。或者继续往下扫描, 输入队列, 看看有没有其他内存需求比较小的进程可以被满足。

### 动态存储分配问题

通常, 按上面的分配算法, 可用的内存块分散在内存里的不同大小的**孔** (**这里既是一个一个小的内存块**) 的集合。当进程需要内存时, 系统为该进程查找足够大的空, 如果孔太大, 那么就分为两块: 一块分配给内存, 另一块还给孔集合。当进程终止时, 他将释放内存, 该内存将还给孔集合。

如果新孔与其他孔相邻, 那么将这个孔合并成一个大孔。这是系统继续检测, 有没有符合要求的处于等

待的进程。

这种分配方法被称为**动态存储分配问题**。这种方法在我们程序设计中有很多的例子，比如（C++STL的内存分配，memory cache）。

这种方法有许多变种，其中根据从一组可用的孔中选择一个空闲孔的最为常用的方法包括：**最先适应，最优适应，最差适应**。

- **最先适应**：分配首个足够大的孔。查找从头开始，也可以从上次首次适应结束的时候开始，寻找一个足够大的空闲孔，分配个进程。
- **最优适应**：分配足够小的孔，只要可以满足进程的内存需求。需要查找整个列表，所以可能要求，列表按大小进程排序，以便更快速的找到最小满足的孔。
- **最差适应**：分配最大的孔。这个和最优相对，也要查找整个列表。

经过模拟显示：首次适应和最优适应的执行效果和空间利用方面都要优于最差适应。

## 碎片

**外部碎片**：根据上面的三种算法，随着进程的加载到内存和从内存退出，空闲内存空间被分为小的片段。当总的可用内存之和可以满足请求但并不连续时，他不能分配给进程。这些不连续的内存块就成了碎片了。

**内部碎片**：假设有一个1800字节的碎片，并采取上面的算法进行分配，进程需要1798的内存。那么分配完了以后，就剩下两个字节，这两个字节也需要维护，但是维护的代价比他本身的价值要大的多，所以在分配的时候，把这两个字节也分配给进程，那么这两个字节就在进程内部形成了碎片。

在清理内存碎片的时候，我们有两种解决方案：

- 把所有的进程移到内存的一端，把孔移到另一端，但是这种代价太高。
- 允许进程的逻辑地址空间不是连续的；从而让代码和数据分离，进程在使用的时候，当物理内存可用，就允许进程分配内存，然后根据不同的地址寄存器和变址寄存器来控制和保护进程的内存。也就是**分段和分页技术**。

## 分段

### 基本概念

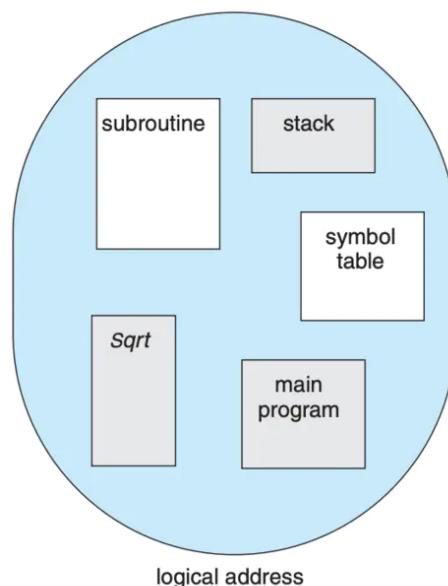


Figure 8.7 Programmer's view of a program.

在编写代码的时候，程序员认为它是由主程序加上一组方法，过程或者函数的集合。他还包括这种数据结构：对象，数组，堆栈，变量。每个模块或者数据元素通过名字来引用。而不关心具体的内存位置。

**分段**就是支持这种用户视图的内存管理方法。逻辑地址空间是由一组段构成。每个段都有名称和长度，比如，代码段，数据段，堆栈，堆等等。

地址指定了段名称和段内偏移。所以段一般是由两个量来进行表示：

<段号, 偏移>

通常，在编译用户程序的时候，编译器会自动生成：

- 代码
- 全局变量
- 堆
- 栈
- 程序库

### 分段的硬件实现

虽然用户可以实现通过**二维地址**来引用程序内的对象，但是实际物理内存还是一维的字节数组。因此我们需要我们需要定义一个映射方法，把二维地址转换为一维地址。

这个地址是通过**段表**来实现的。段表的每个条目都有一个**段基地址**和**段界限**。段基地址包含该段在内存中的开始物理地址，而界限地址指定该段的长度。

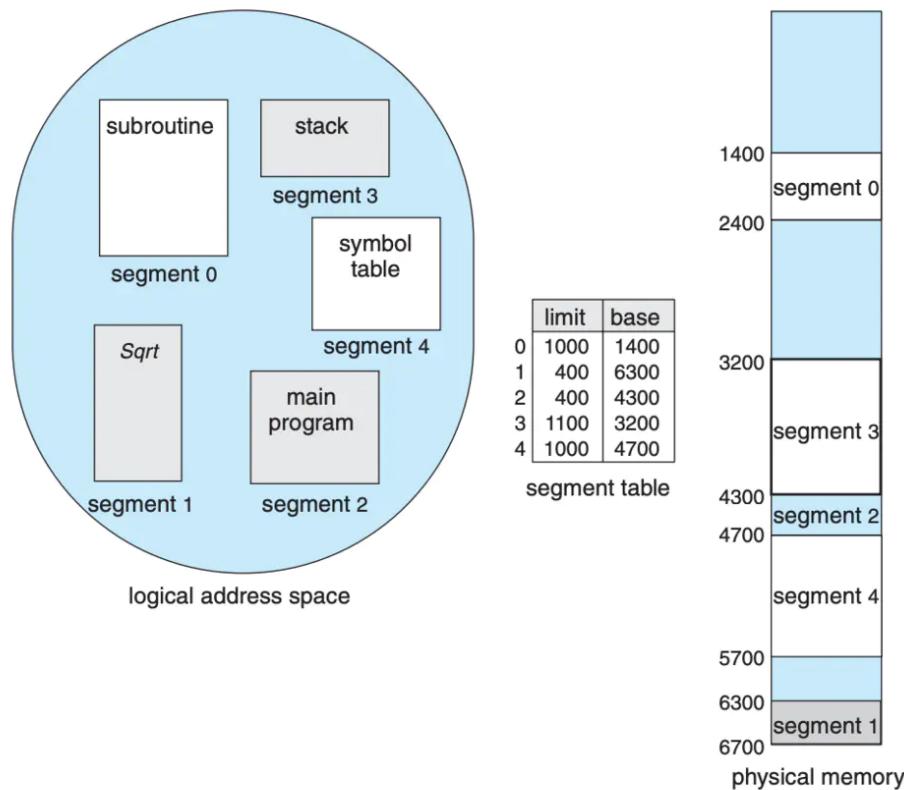
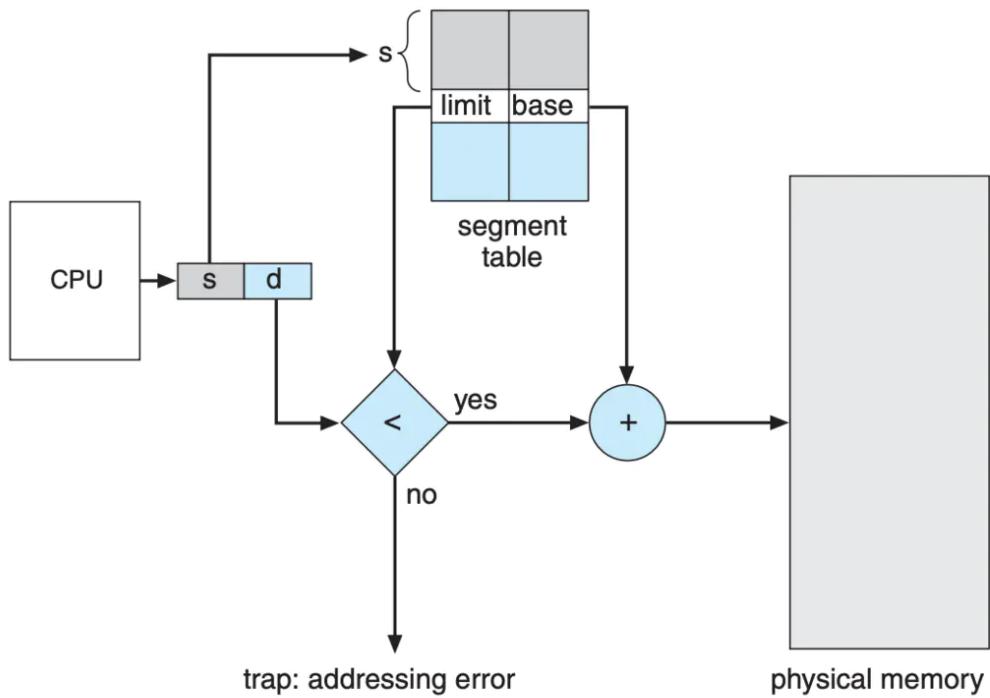


Figure 8.9 Example of segmentation.

从图中，我们可以看到段表和每个段的分布情况。

再看看分段的硬件实现，每个逻辑地址都有两个部分来实现，**段号s**和**偏移d**。当我们从CPU的指令中得到一个地址时，它有段号和偏移d，段号首先去段表中进行索引，获取到段基(base)地址，然后加上偏移d,然后和界限地址进行比较，得到最终的物理内存地址。**(这里段表上两个元素实际上是地址寄存器和界限寄存器组成的结构体数组)**。



**Figure 8.8** Segmentation hardware.

## 分页

分段允许进程的物理地址空间可以分为多个段，从而地址可以是非连续的。然而分段还是避免不了有外部碎片的情况，因为在给每个段分配内存的时候，各个段的大小是不一样的。从而让整块物理内存还是会有缝隙。分页避免了这种情况。

### 基本概念

实现分页的最基本方法涉及到将物理内存分为固定大小的块，称为帧或者页帧(frame，从英文名可以看出来这是个框架，大小应当是2的指数倍，512bytes-16Mbytes，一般为4KB)；而将逻辑内存也分为同一大小的块，叫做页或者页面（page，从英文名可以看出来这是一张纸）。

当需要执行一个进程时，它的页从文件系统或者磁盘中加载到内存的可用页帧中。磁盘也被划分为固定大小的块，它与单个内存帧（frame）或者分为多个内存帧的大小一样。

### 分页的硬件支持

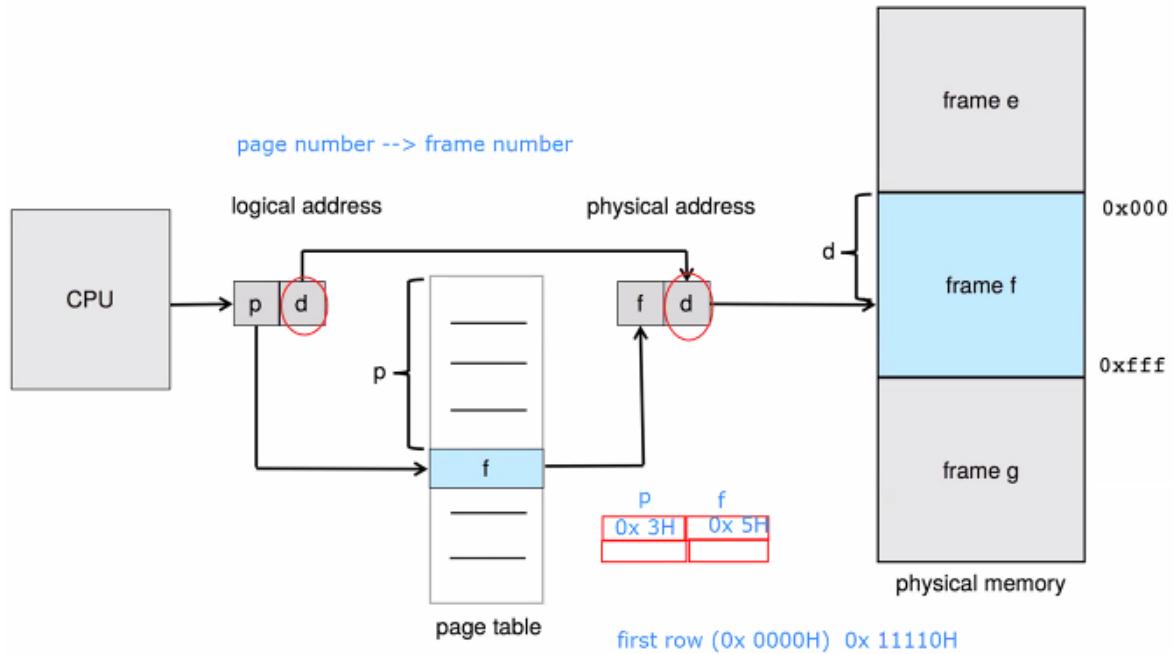
分页的硬件支持和分段类似，由CPU解码后的指令地址分为两个部分：页码（page number）和页偏移（page offset）。

<i>p</i>	<i>d</i>
1101	1110 0001 0010
DE12H	

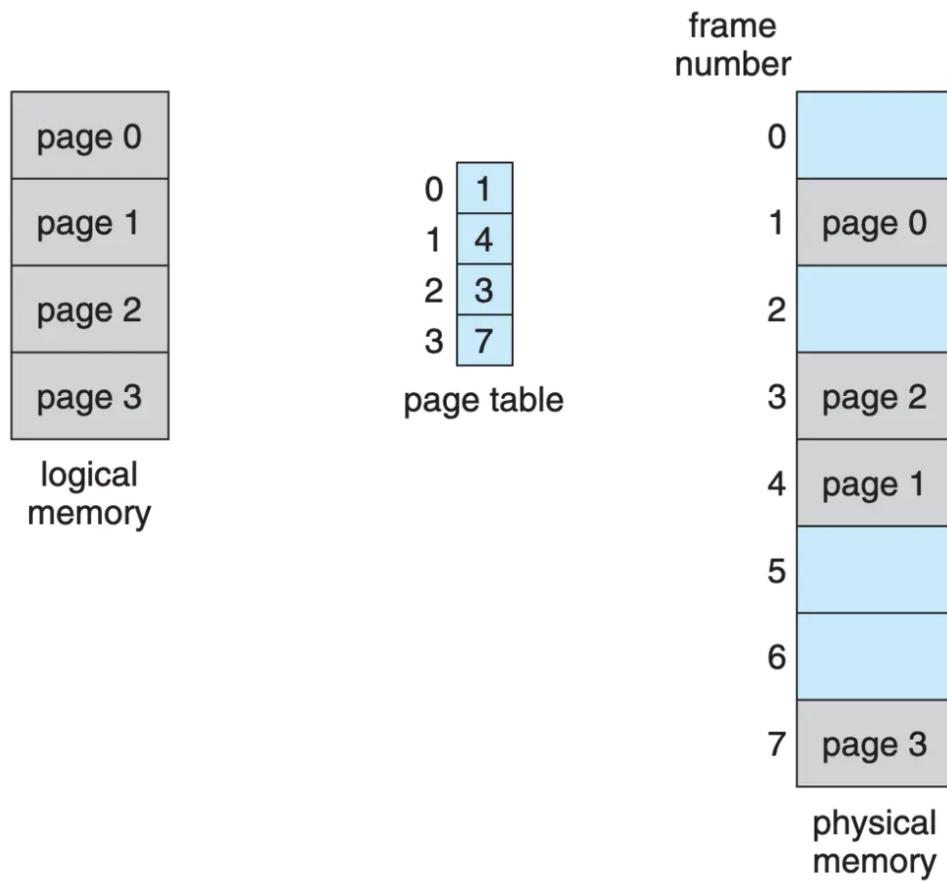
0x de12H

p 6bits	d 10bits (page size 1KB)
0011 0111	0x 37H
0010 0001 0010	0x 212H

页码作为页的索引。页表包含每页所有物理内存的基地址。这个基地址与页偏移的组合形成了物理内存地址。



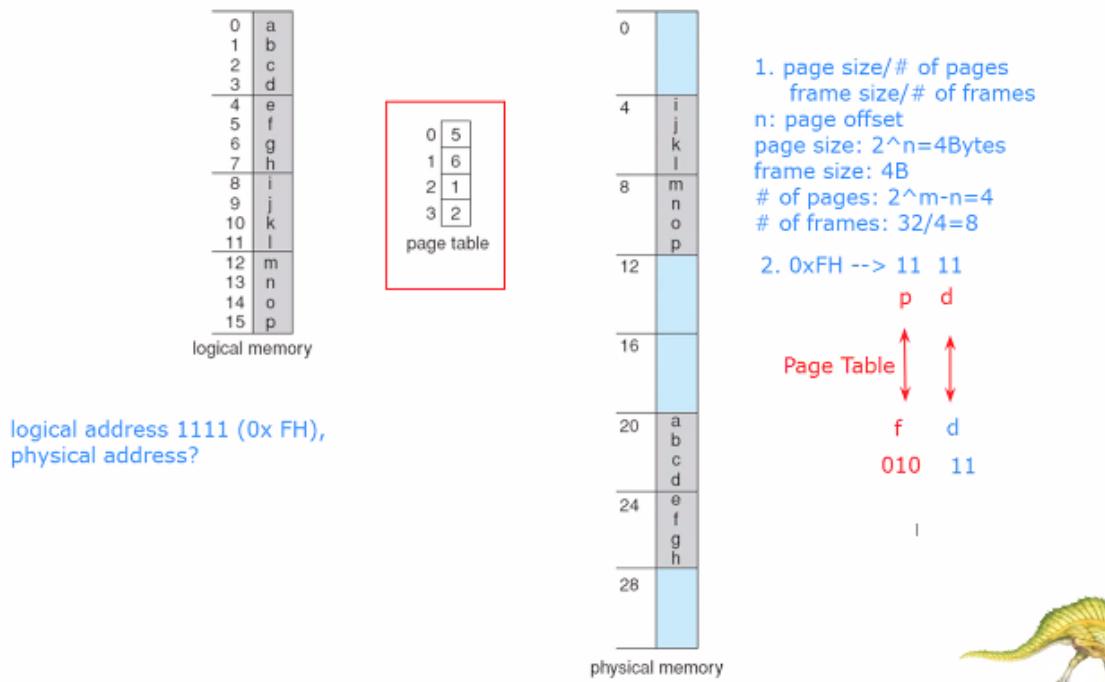
再看看页表的结构(注意这里的地址用的是帧码，也就是frame的页号，而不是字节)。



## 例题

共4位， offset2位

■ Logical address:  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



逻辑地址15的p对应物理地址11的p

我们通过上面的学习，应该可以看到，分页本身是一种动态的重定向。每个逻辑地址由分页硬件绑定某一个物理地址。采用分页类似于一个基址寄存器，每个基址代表一个一个内存帧。

当系统进程需要执行时，它将检验该进程的大小，进程的每页都需要一帧。因此一个进程需要n页，那么内存中至少应该有n帧。如果有，那么可分配给新进程。进程的第一页装入一个已分配的帧，帧码会在装入后写到进程的页表中，下一页分配给另一帧，然后帧码也会写到进程的页表中。



Figure 8.13 Free frames (a) before allocation and (b) after allocation.

有了分页，程序员就可以将内存看做一整块来处理。内存的分配和管理交给操作系统。他可以认为他的程序在逻辑上是连续的，但是实际上，物理内存中他们是分布在不同的帧上的。在操作系统管理内存的分配时：它需要知道哪些帧已经分配，哪些帧空着，总共有多少帧。这些都在内核的叫做帧表（frame table）的数据结构中。在帧表中每个条目对应一个帧，以表示该帧是空闲还是已占用；如果占用，是

被那个进程占用。

## 分页的效率讨论

- 页表具体的结构是怎样的？

每个操作系统都有自己保存页表的方法。有的系统为每个进程分配一个页表，然后通过指针保存在PCB中。当启动一个进程的时候，他应该首先加载一个用户寄存器，并通过保存的用户页表来定义正确的硬件页表值。

- 页表的硬件实现时怎样的？

页表的硬件实现有很多种，但是最简单的方法是通过一组专用的寄存器来实现。这些寄存器应用高速逻辑电路来构成，以高效的进行分页地址的转换。由于每次访问内存都要经过分页映射，因此效率是一个重要的考虑因素。CPU分派器在加载其他寄存器时，当然也需要加载这些寄存器，当然，这些页表寄存器的加载指令也是特权的。

- 用寄存器来实现页表的问题？

如果页表比较小的时候，用页表寄存器时效率是很高的。但是现在的计算机基本都允许页表很大，对这些机器，采用寄存器就不行了。因此，页表需要放在内存中，并将**页表基址寄存器**

**(PTBR)** 指向页表，改变页表只需要操作这一个寄存器就可以了。用**页表长度寄存器 (PTLR)** 存储页表长度

- 采用页表基址寄存器的效率？

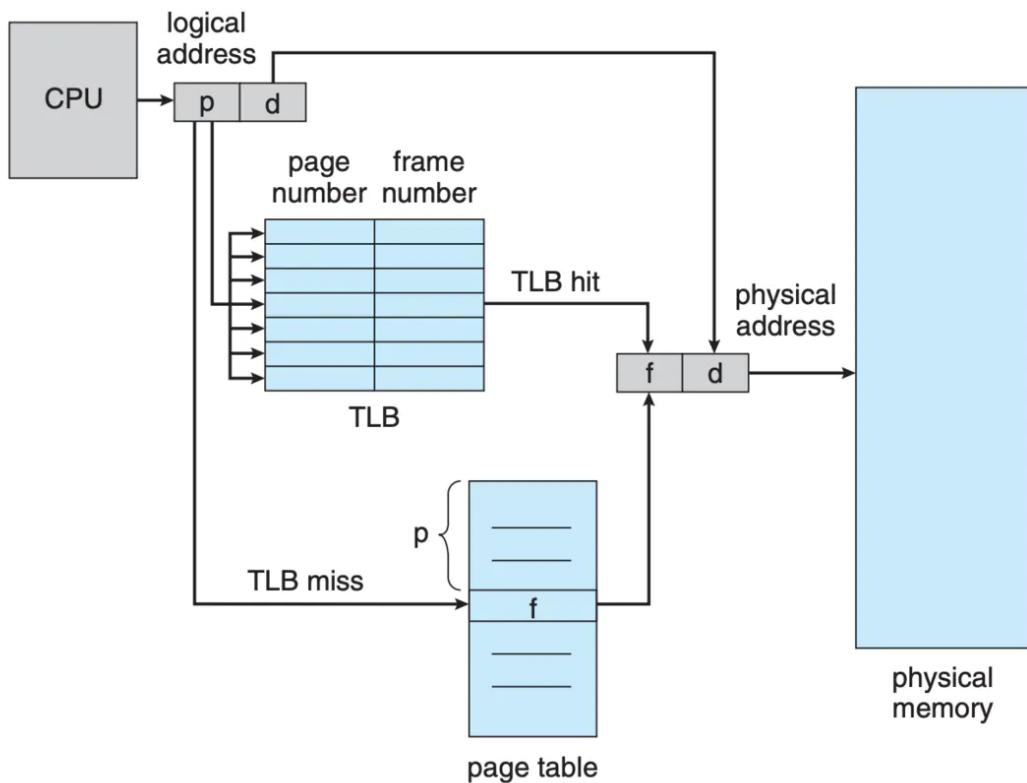
采用这种方法的问题是访问用户内存位置的所需时间，如果需要访问位置 $i$ ，那么应该首先利用PTBR的值，再加上 $i$ 的页码，作为偏移，来查找页表。这一任务需要内存访问。根据所得的帧码，再加上页偏移，就得到了真正的物理地址。接着就可以访问内存内的所需位置。采用这种方案，访问一个字节需要两次内存访问（一次是页表，一次是字节）。这样内存的访问效率就减半了。

因为效率问题，我们的解决方案是采用专用的，小的，查找快速的高速硬件缓冲，它称为**转换表缓冲区 (TLB)**。它是一个关键的高速内存。

-TLB的工作原理

TLB条目由两部分组成：键和值。当关联内存根据给定值查找时，它会同时与所有的键进行比较。那么就得到相应的值。搜索的特别快。现代TLB的查找硬件是指令流的一部分，基本不会考虑性能代价。

TLB只包含了少量的页表条目。当CPU产生一个逻辑地址后，它的页码就送到TLB，如果他能找到这个页码，它的帧码也就立即可用，可用于访问内存。如果页码不在TLB中，那么就需要访问页表（访问内存）。取决于CPU，这可能由硬件自动处理或者通过系统的中断来处理。当得到帧码后，就可以用它来访问内存，同时还会把页码和页帧加到TLB中。



**Figure 8.14** Paging hardware with TLB.

有的TLB在每个TLB条目中还保存地址空间标识符 (ASIDs)，他唯一标识每个进程，并为进程提供地址空间的保护。

TLB是一个硬件功能，我们不需要关心，但是了解他的功能和特性，有利于我们在开发时，进行系统的优化。

EAT 有效访问时间

**Hit ratio – percentage of times that a page number is found in the TLB**

An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.

Suppose that 10 nanoseconds to access memory.

- If we find the desired page in TLB then a mapped-memory access take 10 ns
- Otherwise we need two memory access so it is 20 ns

$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

## 分页的安全讨论

分页环境下的内存保护是通过与每个帧关联的保护位来实现的，通常这些保护位会保存在页表中。

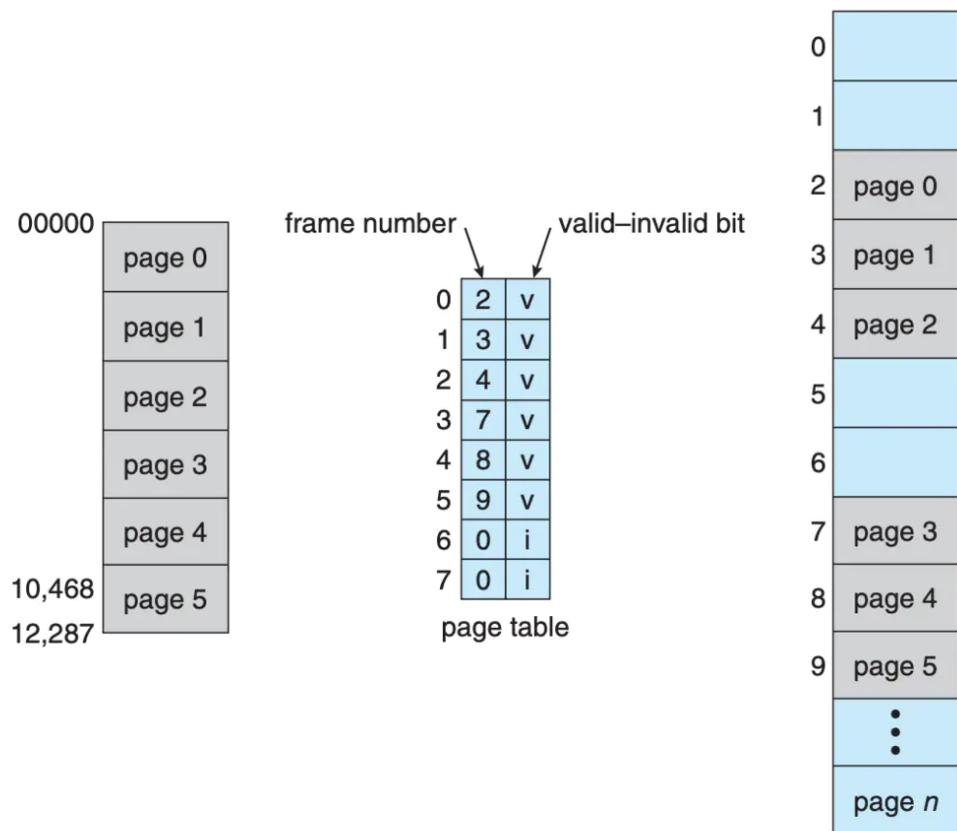
用一个位可以定义一个页是可读可写或只可读。每次内存引用都要通过页表，来查找正确的帧码。在计算物理地址的同时，还可以通过检查保护位来保护系统对内存的正确操作。

还有一个位通常与页表中的每一条项目相关联：**有效-无效位**。

当该位是有效位时，该值表示相关的页在进程的逻辑空间内，因此它是合法的页。

当该位为无效位时，表示相关的页不在进程的逻辑地址空间内。

通过有效-无效位，非法地址会被捕捉，然后操作对该地址进行允许和不允许对某页的访问。



**Figure 8.15** Valid (v) or invalid (i) bit in a page table.

对于上图：14位地址空间 (016383) 的系统，假设有一个程序，它的有效地址空间是010468。如果页的大小是2k,那么他会有5个页表。然而，如果试图产生页表 6和页表7，那么操作系统就会根据有效-无效位的捕捉到非法操作。

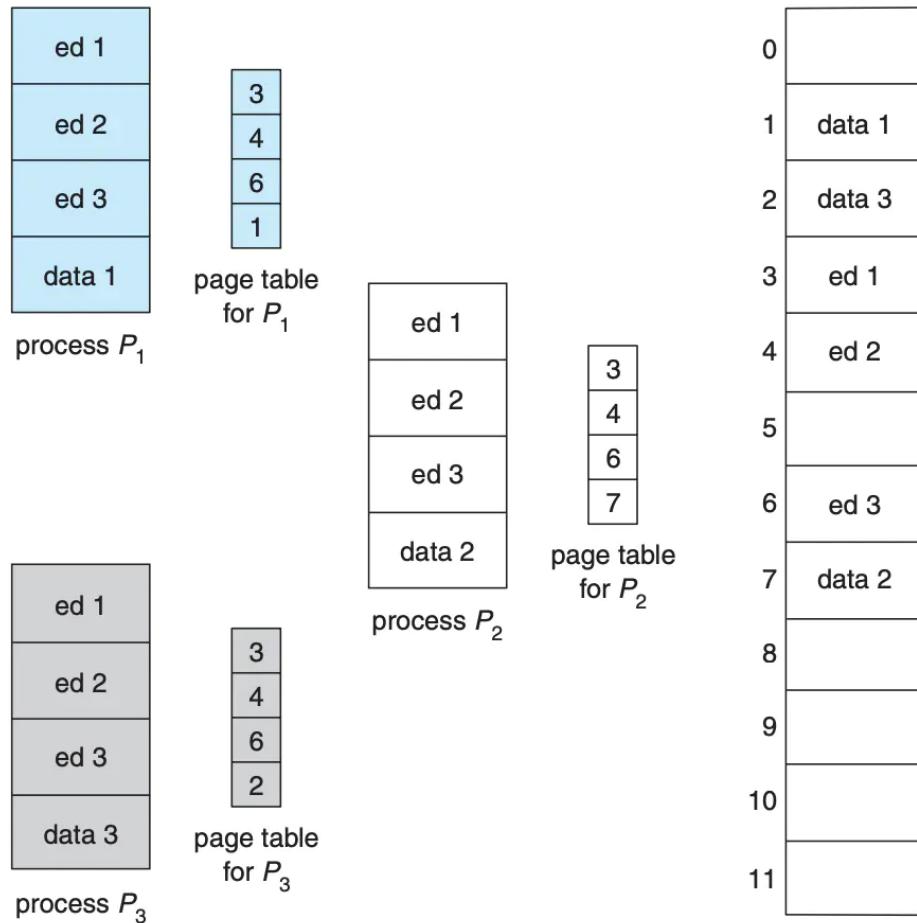
## 共享页

分页的内存机制，还有一个优先就是可以共享公共代码。

对于一个分时系统，假设有一个支持40个用户的系统，每个用户都指向一个文本编辑器，每个文本编辑器包括150k的代码和50k的数据空间，那么就需要8000kb的内存来支持40个用户。

但是，如果代码是**可重入代码**。则可以进行共享。

**可重入代码**是不能自我修改的代码，他在执行期间不会改变。因此两个或者多个进程可以同时执行相同的代码。每个进程都有自己的寄存器和数据存储，以保证数据的正确性和安全。



**Figure 8.16** Sharing of code in a paging environment.

所以在物理内存中只需要保存一个编辑器的副本，每个用户的页表映射到编辑器的同一个物理副本，但是数据页映射到不同的帧。因此支持40个用户，只需要一个编辑器副本，和40个50k的数据空间，所以只需要2150k,而不是8000k。

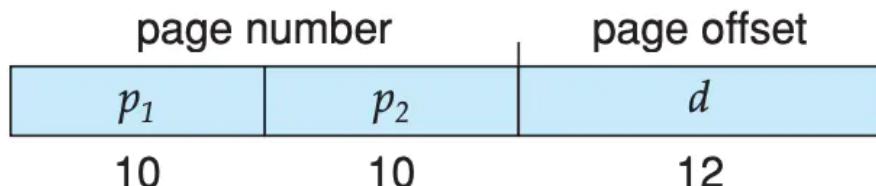
## 分层分页

大多数现代计算机系统支持大逻辑地址空间 ( $2^{32} \sim 2^{64}$ )。这种情况下，页表本身可以非常大。例如：假如具有32位逻辑地址空间的一个计算机系统。如果系统的页大小为4KB( $2^{12}$ )。那么页表可以多达100万的条目 ( $2^{32} / 2^{12}$ )。假设某个项目有4字节。那么每个进程需要4MB的地址物理地址来存储页表本身。显然，我们并不想在内存中连续分配这么多页表。

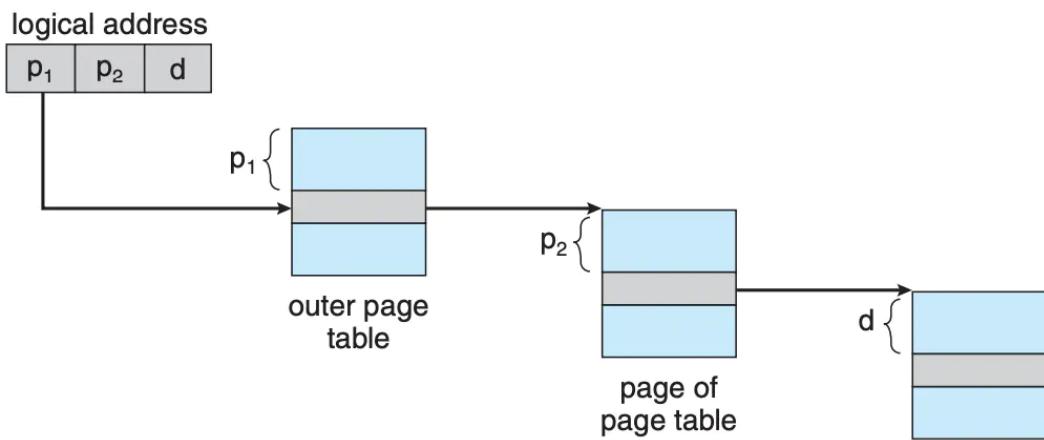
这个问题的一个简单的解决方法就是讲页表划分为更小的块。完成这种划分方法有很多种。

最简单的方法就是使用两层分页算法，就是将页表再分页，例如，再次假设一个系统，具有32位逻辑地址空间和4K大小的页。一个逻辑地址被分为20位的页码和12位的页偏移。

因此要对20位的页表进行再分页，所以该页码可以分10位的页码和10位的偏移。这样一个逻辑地址就会分为如下表示。



其中 $p_1$ 表示的用来访问外部页表的索引，而 $p_2$ 是内部页表的页偏移。采用这种结构的地址转换方法。由于地址转换有外向内，所以这种也称为**向前映射页表**。



**Figure 8.18** Address translation for a two-level 32-bit paging architecture.

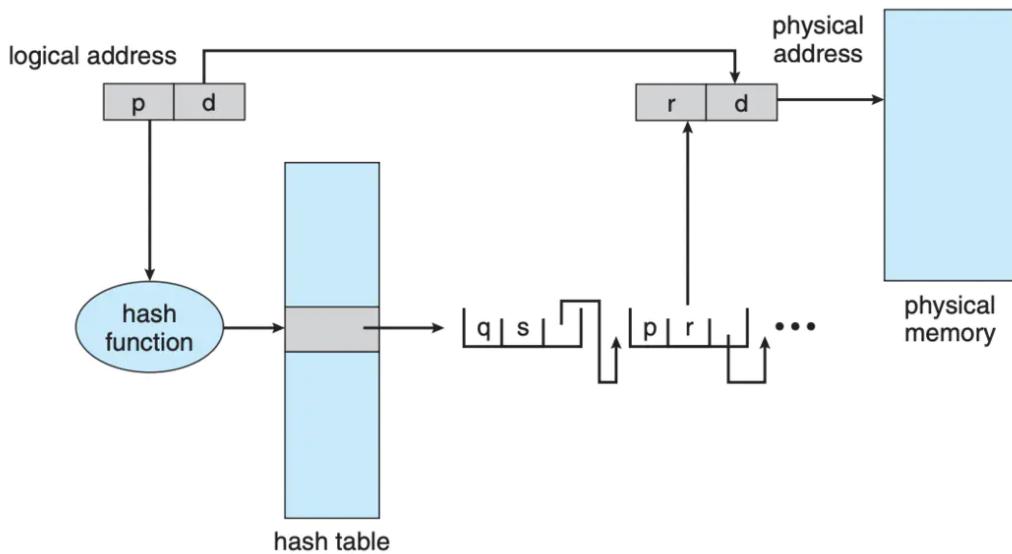
在这种分页结构的方案中，假设，系统是64位系统，那么当它的地址空间就有 $2^{64}$ ，当再以4KB作为地址的话，那么页表就会 $2^{52}$ 个条目，那么就把页表进行细分，从而形成三级分层分页，四级分层分页等等。

为了装换每个逻辑地址，74位的系统需要7个级别的分页，如此多的内存访问时不可取的，从而分层分页在64位的系统并不是最优的。

## 哈希页表

处理大于32位的地址空间的常用方法是**哈希页表**，采用虚拟页码作为哈希表值。哈希页表的每一个条目都包括一个链表，该链表的元素哈希到同一位置（这表示它们有了哈希冲突）。每个元素由三个字段组成：虚拟页码，映射的帧码，指向链表内下一个元素的指针。

该算法的工作如下：虚拟地址的虚拟页码哈希到哈希表。用虚拟页码与链表内的第一个元素的第一个字段相比较。如果匹配，那么相应的帧码（第二个字段）就用来形成物理地址。如果不匹配，那么与链表内的后续节点的第一个字段进行比较。以查找匹配的页码。该方案如图：



**Figure 8.19** Hashed page table.

这里书上提到的虚拟页码可以只看作是页码。（之所以叫虚拟页码，是因为根据虚拟内存的概念，逻辑地址空间可以比物理地址大，所以多出来的部分被称为虚拟的，具体介绍会在下一章提到）。

已提出用于64位地址空间的这个方案的一个变体。

此变体采用 **聚簇页表** 类似于哈希页表。不过哈希表内的每个条目引用多个页而不是单个页。单个页表的条目可以映射到多个物理帧。聚簇页表对于 **稀疏** 地址空间特别有用。这里引用的是不连续的并且散布在整个地址空间。

## 倒置页表

通常，每个进程都有一个关联的页表。该进程所使用的每个页都在页表中有一项（或者每个虚拟页都有一项）。这种表示方法比较自然，因为进程是通过虚拟地址来引用页的。然后是操作系统将这些地址转换为物理内存地址。

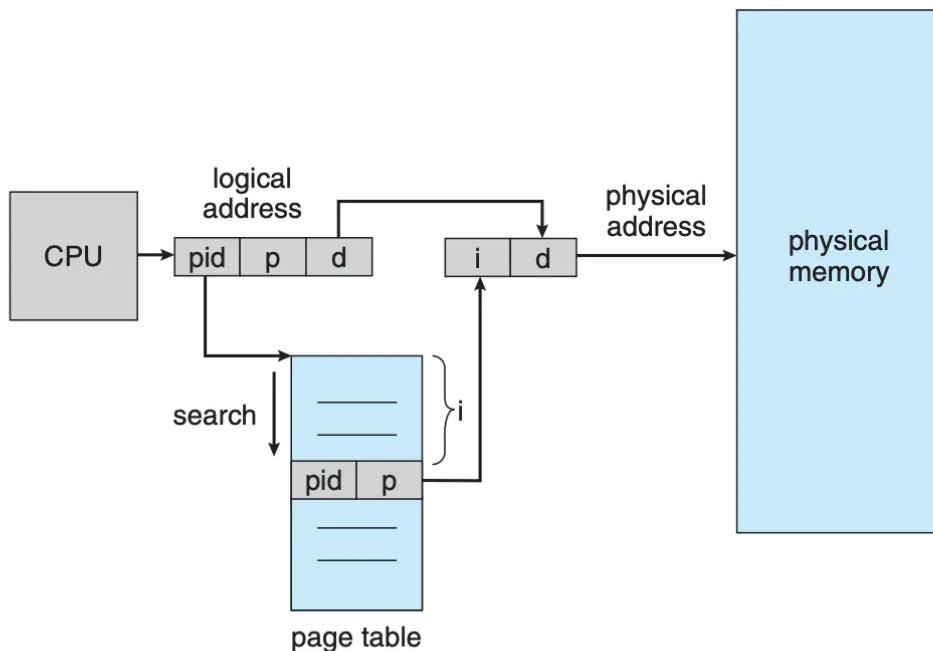
由于页表是按照虚拟地址排序的，操作系统可计算所对应条目在页表的位置，可以直接使用该值。这种方法缺点就是：当每个页表包含百万级的数目时。会有性能问题，而且需要大量的内存来保存页表信息。

解决的方法处理上面的两种方法外，还有一种就是**倒置页表**。

这里先介绍一个IBM RT 的倒置页表的表示方法：

<pid, 页码, 偏移>

对于每个真正的内存页或者帧，倒置页表只有一个条目。每个条目包含**保存在真正内存位置上的页的虚拟地址**，以及**拥有该页的进程信息**。具体的过程如图：



**Figure 8.20** Inverted page table.

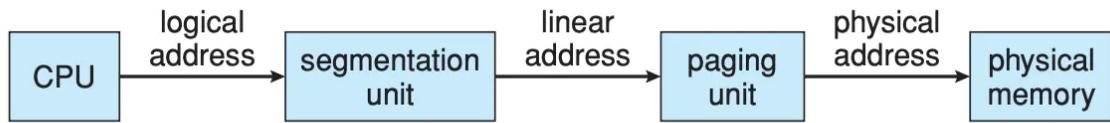
这里的进程的信息就是以前提到的 空间地址标识符 (ASID)。主要是由于一个倒置页表通常包含了多个不同的映射物理内存的地址空间。具体进程的每个逻辑页可映射相应的物理帧。

采用倒置页表的系统在实现共享内存的时候会有问题，因为共享内存的实现为：将多个地址空间映射到同一个物理地址。这种方法，不能用于倒置页表，因为每个物理页只有一个虚拟的页条目，一个物理页不能有多个共享的虚拟地址。

## Intel 32位与 64 位体系

IA-32架构

IA-32 系统的内存管理可以分为分段和分页两个部分，工作如下：CPU 生成逻辑地址，并交给分段单元，分段单元为每个逻辑地址生成一个线性地址。然后线性地址交给分页单元，以生成内存的物理地址。



**Figure 8.21** Logical to physical address translation in IA-32.

### IA-32分段

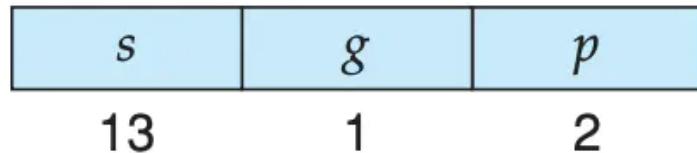
IA-32 架构允许一个段的大小最多可以达到4G，每个进程最多有16K个段。进程的逻辑地址空间分为两部分。

第一部分最多由8K段组成，这部分是单个进程私有；

第二部分也是最多由8K段组成，这部分是所有进程共享。

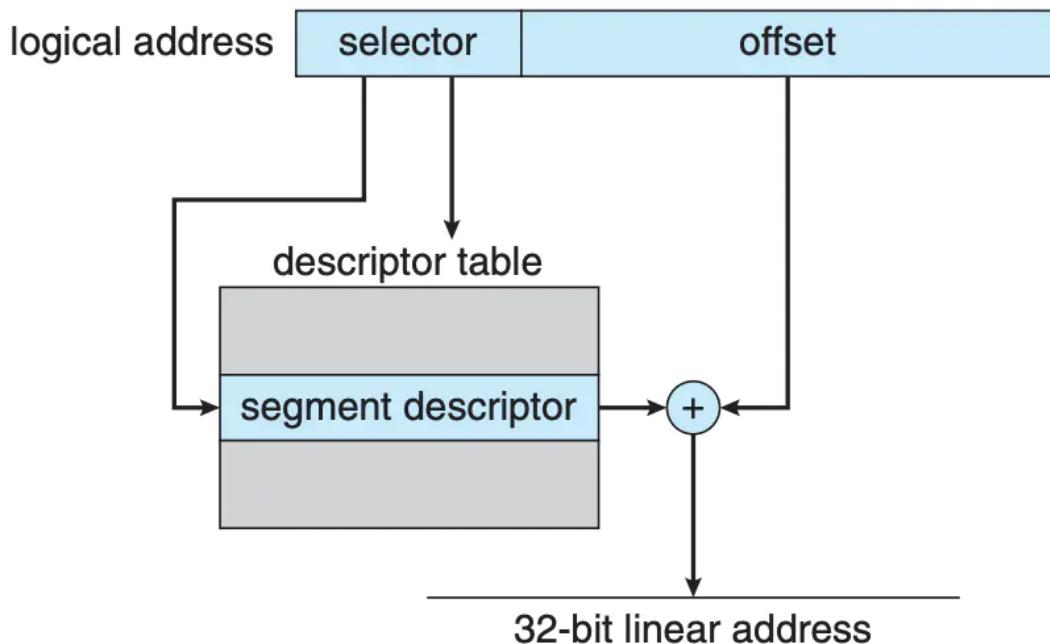
第一部分保存在**局部描述符表 (LTD)** 中，第二部分保存在**全局描述符表(GDT)**中，他们的每个条目都是8个节，包括一个段的详细信息。比如段基地址和段界限。

逻辑地址一般为二元数组（选择器，偏移），选择器是一个16位的数：



其中*s*表示段号，*g*表示实在LTD中还是在GDT中，*p*表示保护信息。

段的寻址过程为：



**Figure 8.22** IA-32 segmentation.

## IA-32 分页

IA-32架构的页可分为4K，或者4M。采用4K的页，IA-32采用二级分页方法。其中的32位的寻址和表示请参照二级分页算法。

为了提高物理内存的使用率，IA-32 的页表可以被交换存在磁盘。因此，页目录的条目通过一个**有效位**，以表示该条目所指的页表实在内存还是在磁盘上。如果页表再磁盘上，则操作系统可通过其他31位来表示页表的磁盘位置。之后根据需要调入内存。

随着软件开发人员的逐步发现，32位架构的4GB内存限制，Intel通过**页地址扩展**，以便允许访问大于4GB的物理地址空间。

引入页地址扩展，主要是将两级的分页方案扩展到了三级方案，后者的最后两位用于指向页目录指针表。

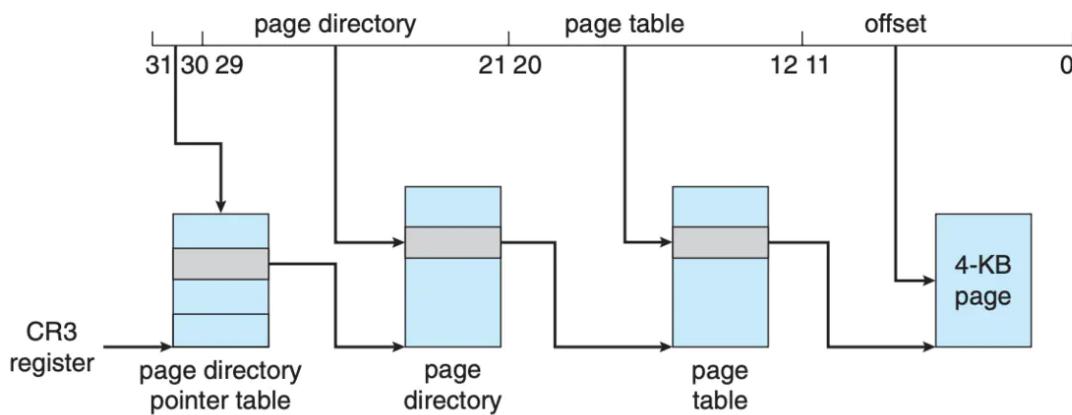


Figure 8.24 Page address extensions.

页地址扩展使得地址空间从32位增加到了36位。Linux和Mac OS X都支持了这项技术。

## X86-64

X86-64 支持更大的逻辑和物理地址空间。支持64位的地址空间意味着可寻址的内存达到惊人的 $2^{64}$ 字节。64位系统有能力访问那么多的内存，但是实际上，目前设计的地址远没有那么多。

目前提供的x86-64 架构的机器最多采用四级分页，支持48位的虚拟地址。它的页面大小可以4KB，2MB，或者1G。

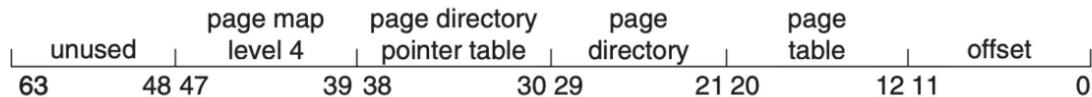


Figure 8.25 x86-64 linear address.

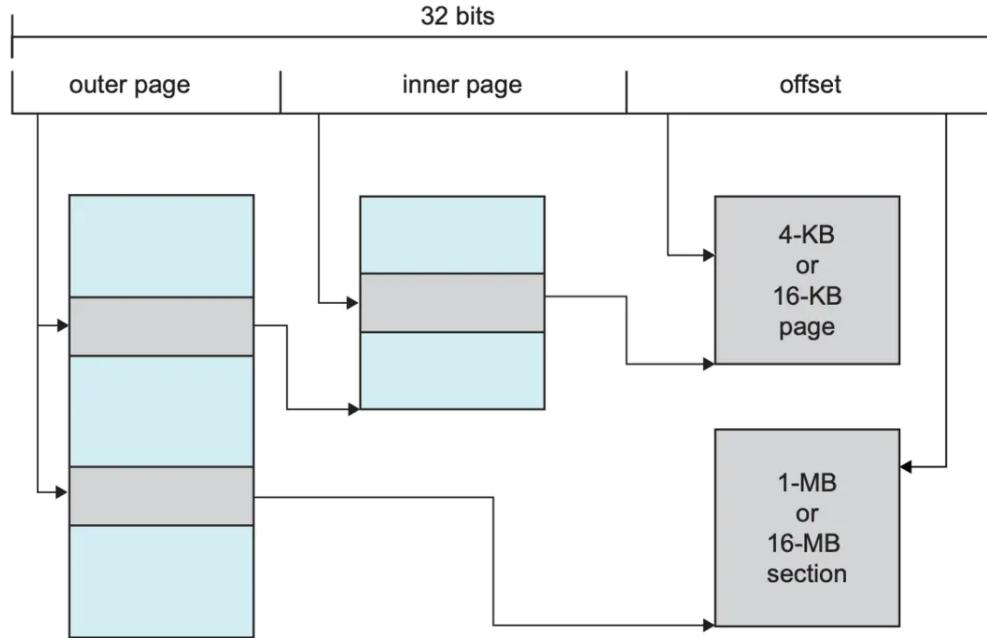
## ARM 架构

虽然Intel的芯片占了大部分的市场，但是移动设备的架构一直采用的是32位ARM的架构。现在的iPhone 和iPad 都或得了ARM的授权。Android的智能手机也都是ARM的处理器。

ARM支持的页面大小：

- 4KB 或者16KB。

- 1MB 或者16MB的页（称为段）。
- 系统使用的分页取决于所引用的是页还是段。  
一级分页用于1MB和16MB的段。二级分页用于4KB和16KB的页。  
ARM的MMU的地址转换如图：



**Figure 8.26** Logical address translation in ARM.

ARM架构还支持两级TLB（高速缓存）。在外部，有两个微TLB：一个用于数据，另一个用于指令。微TLB也支持（ASID）进程地址空间标识符。在内部有一个主TLB。地址转换从微TLB级开始。如果没有找到，那么再检查主TLB。如果还没找到，再通过页表进行硬件查找。

## 虚拟内存

经过以前的学习，了解到了内存的管理策略。不管是分页，还是分段，或者是页表的管理策略。这些都有一个共同的目标：将多个进程都保存在内存中，这一操作，都是为了保证在进程启动时，进程处于内存中。

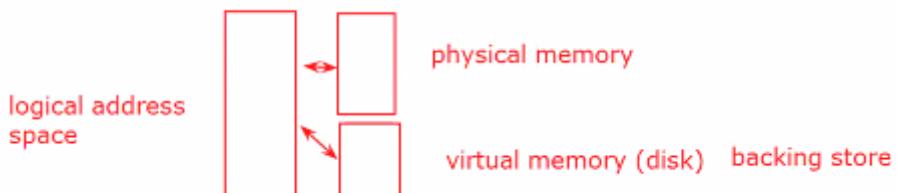
但是虚拟内存技术允许执行进程不必完全处于内存。这就使得程序可以完全大于物理内存，从而在开发的过程中，程序员不用担心内存的限制。

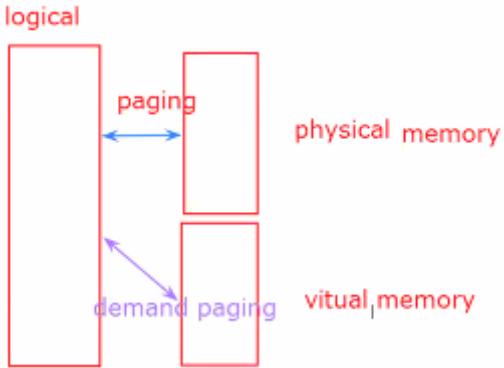
### 基本概念

**虚拟内存：**虚拟内存可以是一种逻辑上扩充物理内存的技术。它会把进程的逻辑地址空间进行扩展。

（书中说的虚拟内存，可以看做是假象的逻辑内存，虽然在平常使用top命令看到了虚拟内存表示一些具体的物理实体，但是而书上一直用虚拟内存这个词个人认为不是很好）。

**虚拟地址空间：**其实这里的虚拟地址空间感觉还是可以用逻辑地址空间的概念来代替。它表示进程在内存中的逻辑地址范围。

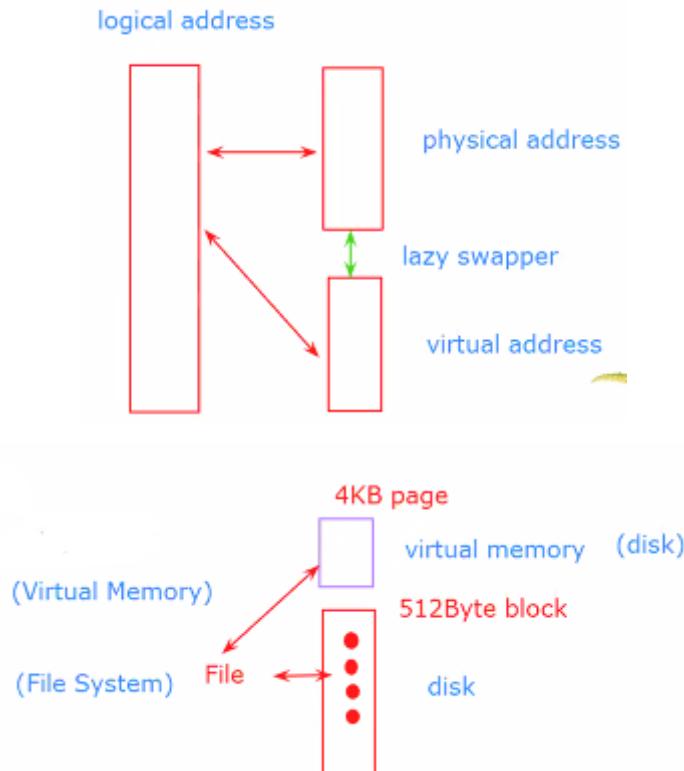




## 请求调页 demand paging

我们在把进程加载到内存的过程中，并不是把进程的所有执行代码都加载到内存，而是在需要执行的时候才加载进内存，这种技术叫**请求调页**，常常用于虚拟内存系统。

对于请求页的逻辑内存，页面只有在程序执行期间被请求时才被加载。因此，从未访问的那些页从不加载到物理内存中。而这种交换的方式也被称为**惰性交换**。减少需要的memory和I/O，提高响应速度和用户数。

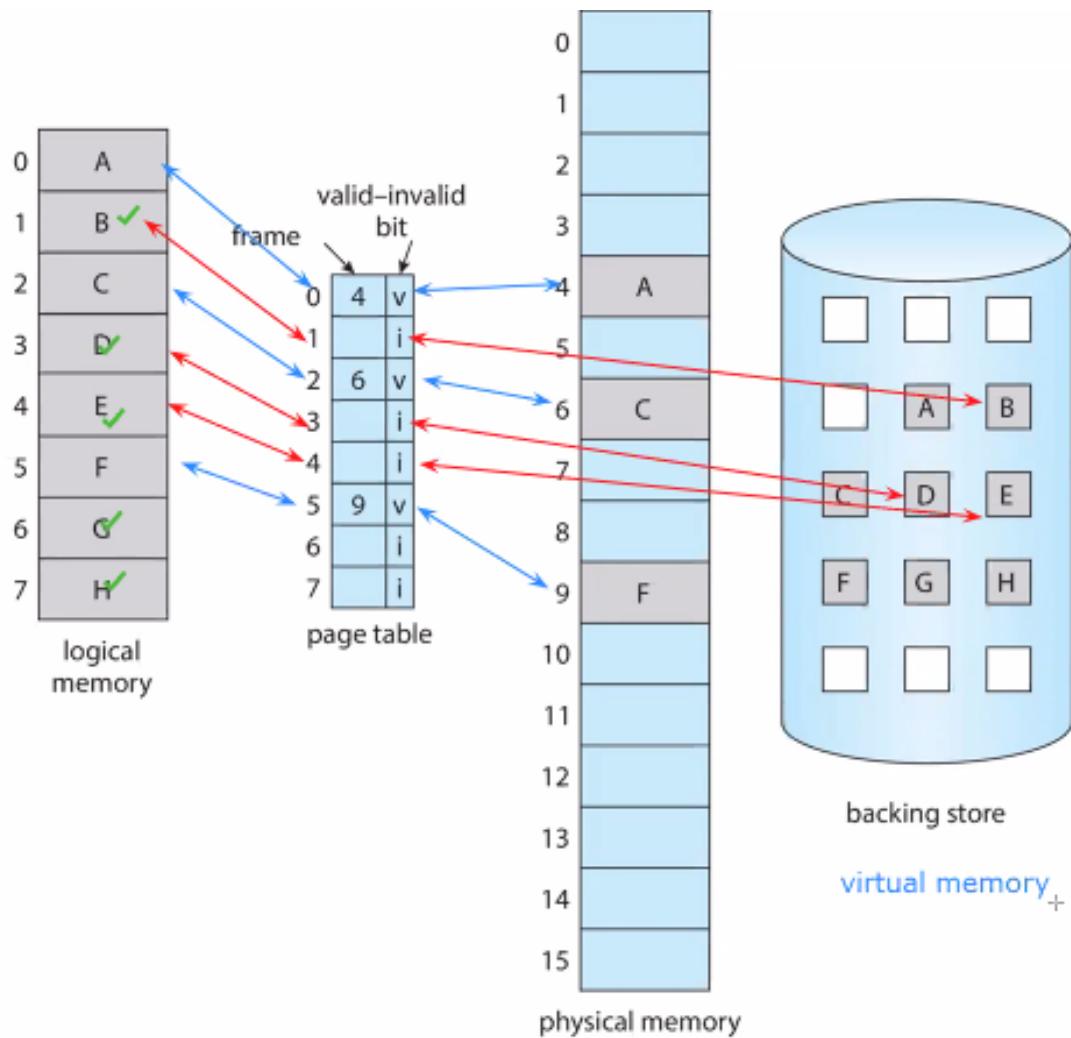


**基本执行过程：**当换入进程时，调页程序会猜测在该进程被再次换出之前，会用到哪些页。调页程序不是调入整个进程，而是把哪些要使用的页调入内存。从而减少交换时间，而且还能避免物理内存空间的浪费。

**具体实现：**在使用这种方案的时候，需要一定的硬件支持。以区分内存的页面和磁盘的页面。通常在页面上会有一个保护位，它的最后一位就是用于提供这个功能的。这个位是**有效-无效位**。

当这个位被置为有效位时，相关的页面是合法的，并且在内存中。

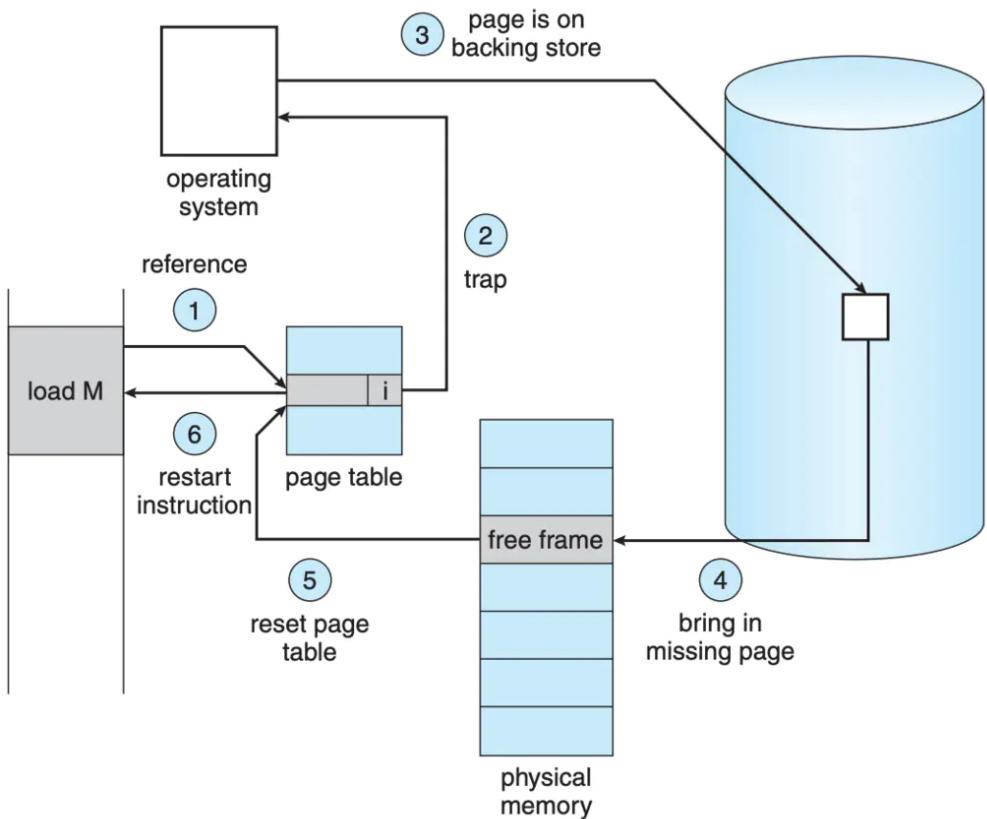
当这个位被置为无效位时，页面无效或者是有效但是在磁盘上。



当进程在执行过程中的时候，如果访问内存驻留的页面时，会一切顺利。但是当访问到尚未调入的页面的时候，标记为无效的页面会产生**缺页错误**。分页硬件在通过页表转换地址时，会发现无效位被设置，从而陷入操作系统。

一般页错误的处理很简单：

1. 检查这个进程的内部表，以确认该引用是有效的还是无效的内存访问。
2. 如果引用无效，那么终止进程。如果引用有效但是没有调入内存，那么现在就调入。
3. 找到一个空闲帧。
4. 调度一个磁盘操作，以将所需页面读到刚分配的帧。
5. 磁盘读取完成，修改进程内部表和页表。以指示该页现在处于内存中。
6. 重新启动被陷入中断的指令。该进程能访问所需要的页面，就好像原来他就在内存中一样。



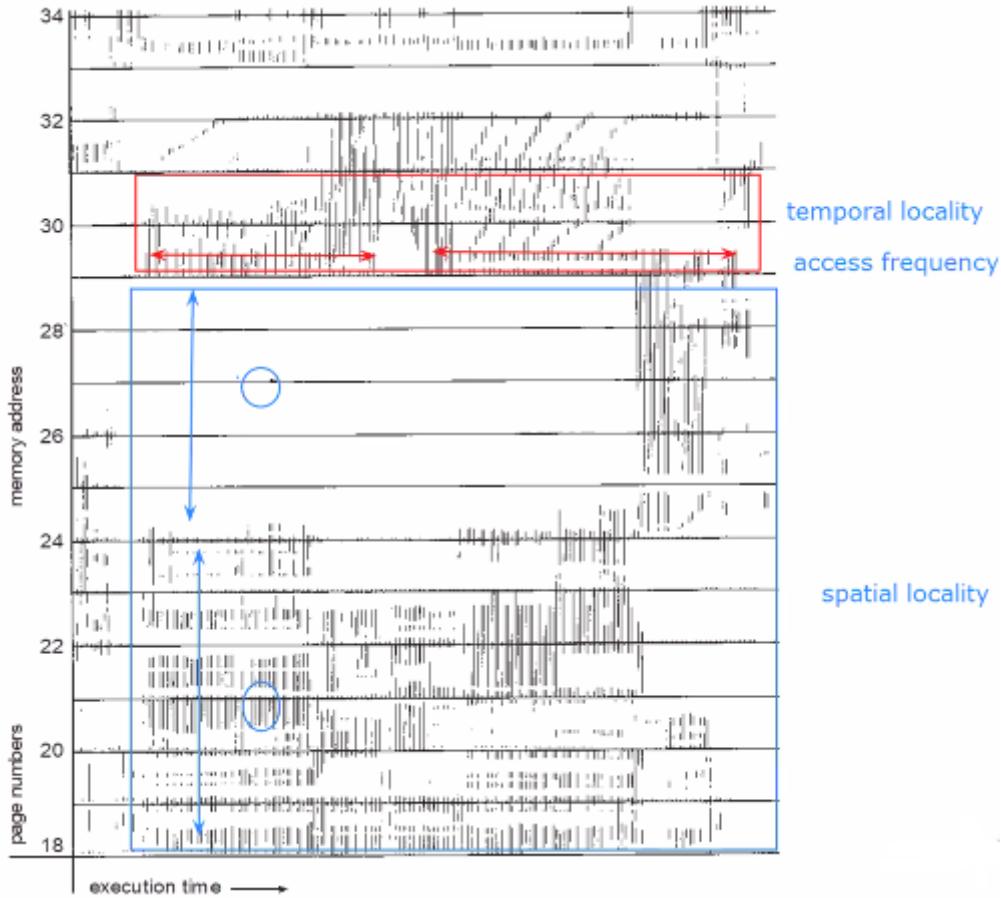
**Figure 9.6** Steps in handling a page fault.

时间局部性

空间局部性

访问频率

1. 频繁访问少量数据
2. 频繁访问临近的数据



## 交换空间

在进程执行过程中，需要一个辅助设备，用于保存不在内存中的那么页面。这种外存通常为高速硬盘，称为**交换设备**，用于交换的这部分磁盘叫做**交换空间**。

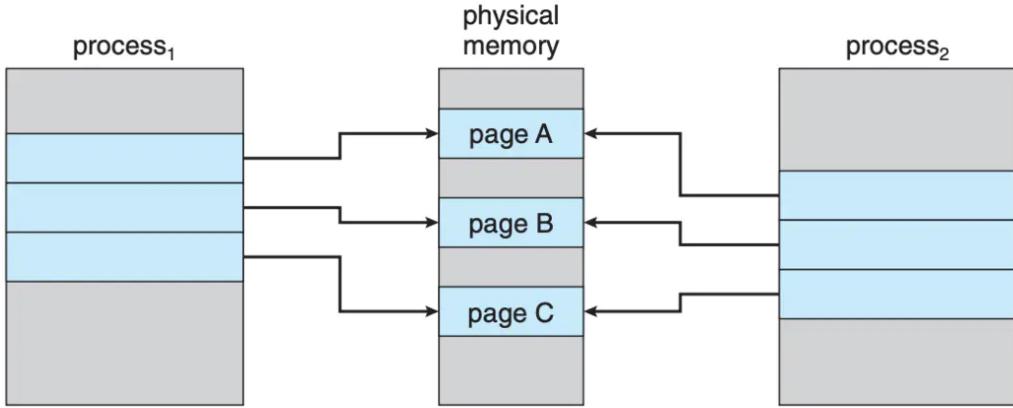
请求调页的中使用的交换空间的磁盘I/O 通常要快于文件系统的。交换空间的文件系统更快，因为它是按更大的块来分配的。并且不采用文件查找和间接分配方法。

因此，系统可以在进程启动时，将整个文件映像复制到交换空间，然后从交换空间执行请求调页，从而获取到好的分页吞吐量。

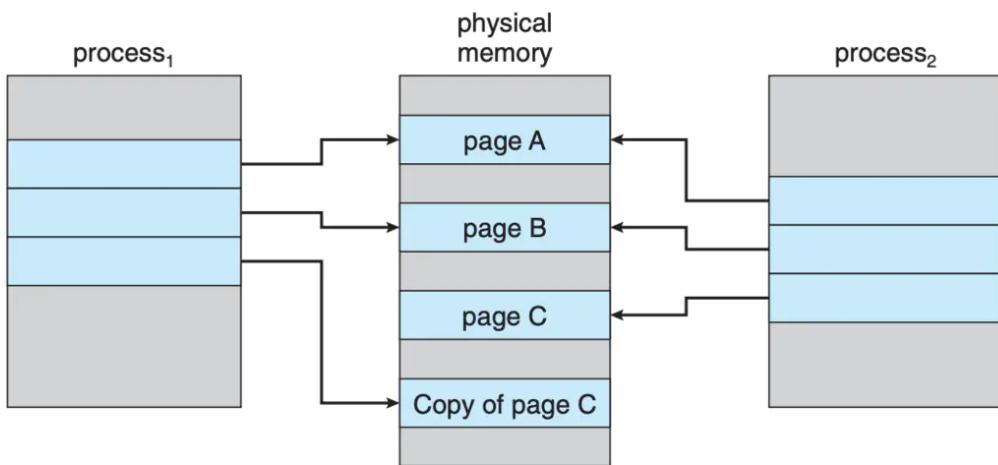
另一个选择是开始时在文件系统进行请求调页，但是在置换页面时将换出的页面写入交换空间。这保证了后续的调页都是从交换空间完成的。

## 写时复制

在父进程调用fork()创建子进程时后，我们曾说子进程应该创建一个父进程地址空间的副本，复制属于父进程的页面。然而考虑到了许多子进程在创建之后立马执行了exec()函数替换了，所以复制父进程的页面可能没有必要。因此我们采用了**写时复制技术**。他通过允许父进程和子进程最初共享相同的页面来进行工作。这些页面是共享页面，被标记为写时复制，这意味着任何进程写入共享页面，那么就创建一个共享页面的副本。



**Figure 9.7** Before process 1 modifies page C.



**Figure 9.8** After process 1 modifies page C.

可以看到使用写时复制技术，仅复制任何一个进程修改的页面，所有未修改的页面可以由父进程和子进程共享。

还要注意，只有可以修改的页面才需要标记写时复制。不能修改的页面可以父子共享（比如代码段）。

在实现上，许多操作系统都为这类请求提供了一个空闲的**页面池**（和个人认为和内存池差不多）。当进程的堆栈或者堆要进行扩展时，或者有写时复制的请求时，通常分配这里的空闲页面。操作系统分配这些页面通常采用**按需填0**的技术。以清理原来的内存。

Linux 提供了 fork() 的变种，vfork()。vfork()的操作不同于写时复制的fork()。它会将父进程挂起，子进程使用父进程的地址空间。

由于vfork()不会发生写时复制，所以他的修改的页面在父进程中是可以看到的。当子进程在创建后立即会被exec()的情况下，可以使用vfork()。因为他们有复制页面，可以高效的启动新进程。

## 页面置换 page replacement

在内存分配页面的时候，不仅用于保存程序页面，用于I/O的缓冲也需要消耗大量的内存，这种使用会增加内存置换算法的压力。确定多少内存给用户程序，多少内存给I/O缓冲去是个很棘手的问题，有的系统给I/O缓冲分配了固定百分比的内存，有些系统则允许用户进程和I/O子系统进行竞争所有内存。

### 基本概念

当用户进程在执行时，可能发生缺页错误。操作系统确定所需要页面的磁盘位置，但是却发现内存上没有空闲的帧。所有内存都在使用。这时，操作系统不能终止一个进程来释放，因此这样太不友好了。此时，它会选择交换出一个进程，以释放它在所有帧并降低多道程序。这种技术就是**页面置换**。

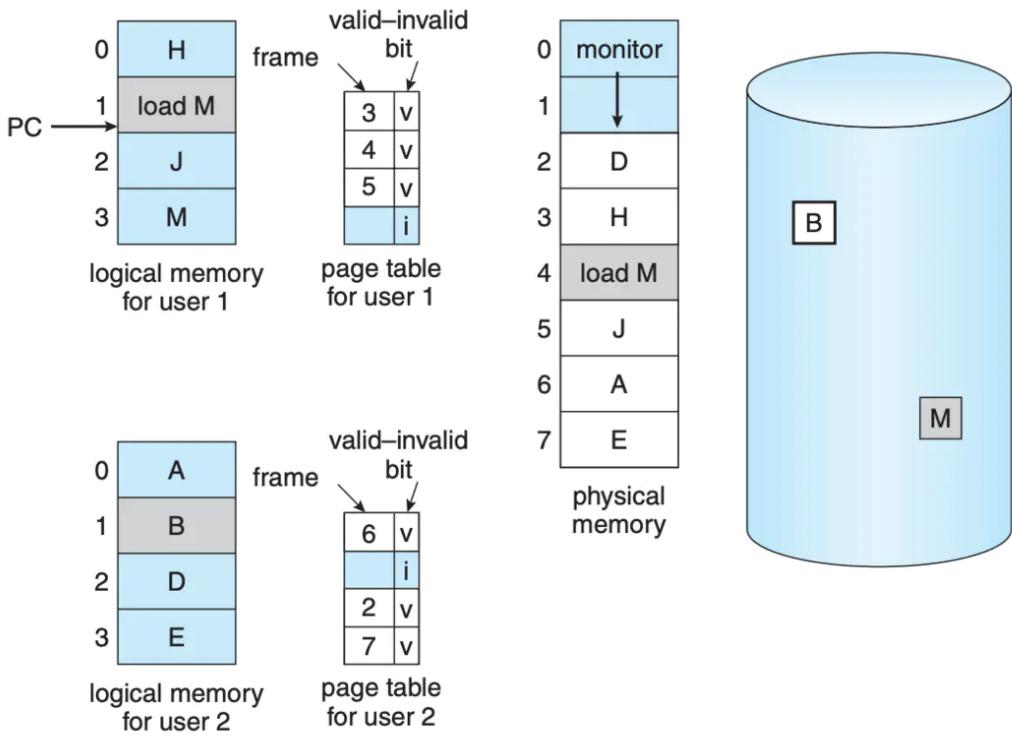


Figure 9.9 Need for page replacement.

## 页面置换

在没有空闲帧的情况下，那么就要查找当前不在使用的一个帧，并释放它，它就是哪个**牺牲帧**，他被换出到交换空间，并修改它的页表。

可以看到当没有空闲帧的情况下，需要两个页面传输（一个传入，一个传出）。这种情况世界加倍了缺页错误处理时间，并增加了有效的访问时间。

为了减少置换的时间，系统提供了一个**修改位**。

在这种方案中，每个页面或者帧都有一个修改位，两者的关联采用硬件。每当一个页面内的任何字节被写入时，它的页面修改位会由硬件来设置，以表示该页面被修改过。

当要选择一个页面置换时，它会先看这个页面或者帧有没有被修改过，如果没有修改过就不用将它换出，直接进行换入，将它的空间覆盖。

为了实现请求调页，那么就要涉及两个重要问题，**帧分配算法**和**页面置换算法**。

## FIFO 页面置换算法

在置换算法中，FIFO 是最简单的算法。

FIFO 页面置换算法为每个页面记录了调用的时间。当必须置换页面时，选择最旧的页面，而不需要记录调入页面的确切时间。可以创建一个FIFO队列来管理所有的内存页面。置换的每次都是队首。

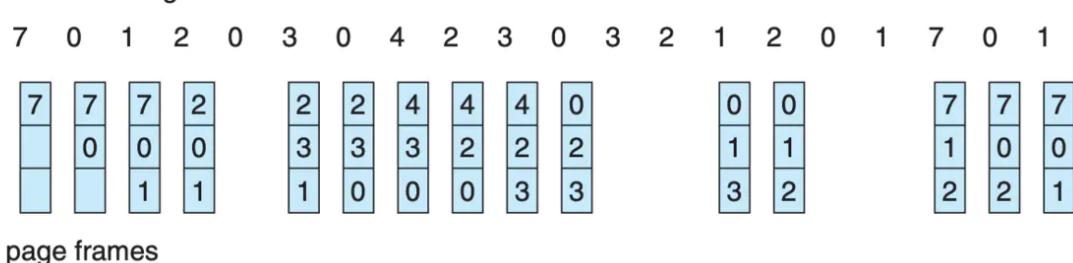
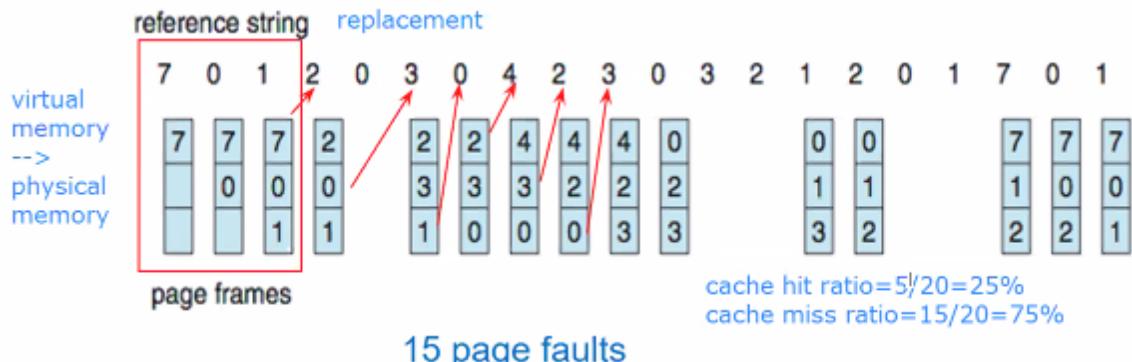


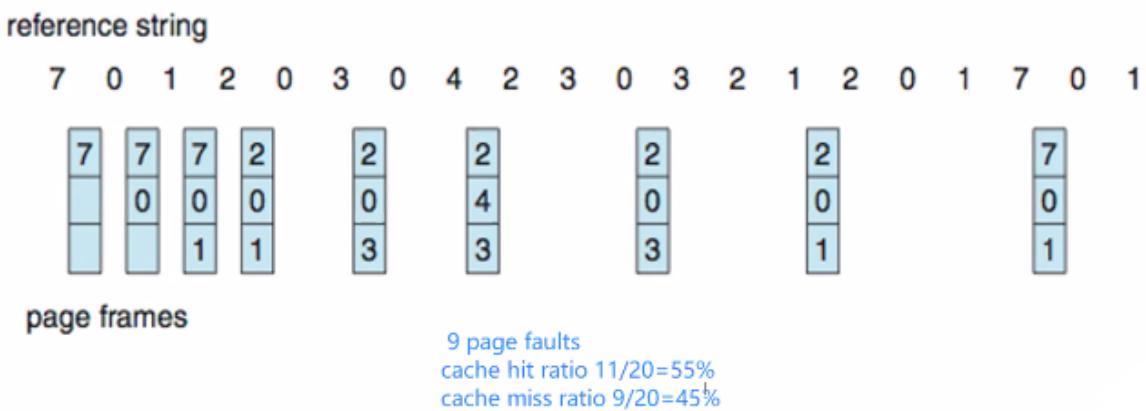
Figure 9.12 FIFO page-replacement algorithm.



FIFO页面置换算法易于理解和编程，但是它的性能并不是总是十分理想的。

## 最优页面置换 (OPT)

最优页面置换：指的是置换最长时间不会使用的页面。



**注意看：**第五个和第6个数之间的，0, 3, 0, 4。这里0刚一执行就被换出去了，这个是和下面LRU算法的区别。

这种方法会产生最低的可能缺页错误率。然而，最优置换算法难以实现，因为需要应用直到引用串的未来推算。

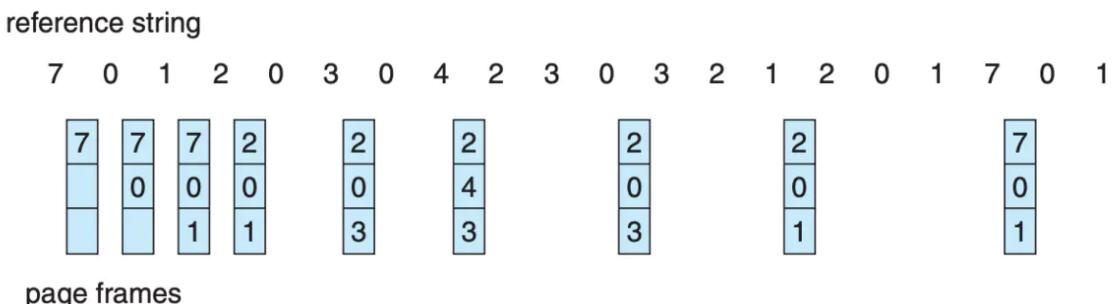
## LRU页面置换算法 last recently used

如果最优算法不可行，那么最优算法的近似或许可以。FIFO和OPT算法的关键区别在于，一个是页面调入内存的时间，一个是页面将来可能使用的时间。两者都不是最可控的。那么我们使用了最近的过去作为将来的近似，那么可以置换最长时间没有使用的页。

**最近最少使用(LRU)的算法：**LRU 置换将每个页面与它的上次使用的时间关联起来。

**注意看两个概念：**OPT的算法是最长时间不会使用，而LRU是最长时间没有使用。细细的思考，就会发现一个是未来时间，一个是过去时间。

如果还没看懂，就来看看两个图的比较。



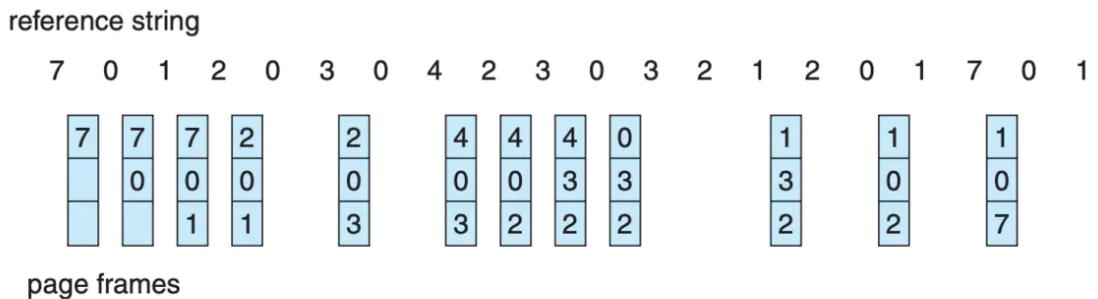


Figure 9.15 LRU page-replacement algorithm.

看出来区别了吗？OPT算法的在第五次置换的时候把刚执行过的0页面换出去了。这个它虽然名字是最优置换，但是不一定就是最优的解法了。

LRU算法通常被人们认为是不错的策略，也是被很多系统用到的页面置换算法。

### LRU的具体实现

LRU算法可能需要硬件辅助；它的问题是：确定由上次使用时间定义的帧的顺序：

- 方案一：计数器

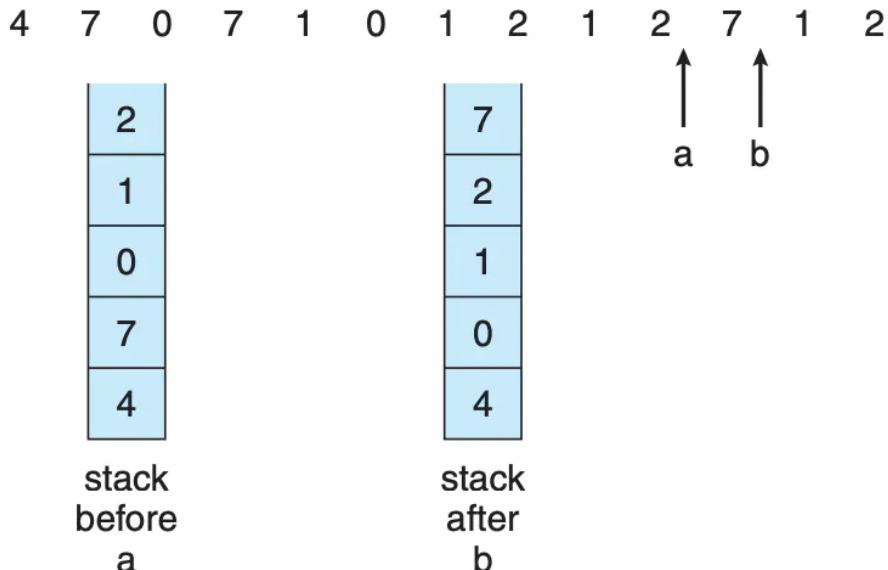
在简单的情况下，为每个页表条目关联一个使用的时间域，并为CPU添加一个逻辑时钟。每次内存引用都会有递增时钟。每当进行页面引用的时候，时钟寄存器（硬件支持）的内容会复制到相应页表条目的时间域。然后在需要置换的时候，扫描整个页表，查找LRU页面。

- 方案二：堆栈

每当页面被引用时，它就从堆栈中移除并放在顶部。这样最近使用的页面总会在堆栈的顶部，最近最少使用的页面总会在堆栈的底部。

因为会有频繁的删除和插入操作，所以最好使用双向链表来实现。

### reference string



页面置换算法还有很多其他的变种。比如：最不经常使用（LFU last frequently used），LRFU，ARC，最经常使用（MFU），但是这些算法的实现是昂贵的。而且还不是最优的。

### 页面缓冲池

除了特定的页面置换算法外，还经常有其他的措施。比如在系统中，会保留一个空闲帧缓冲池。当出现缺页终端时，它会首先在缓冲池中读取空闲帧。这允许进程快速启动。

# 页框分配

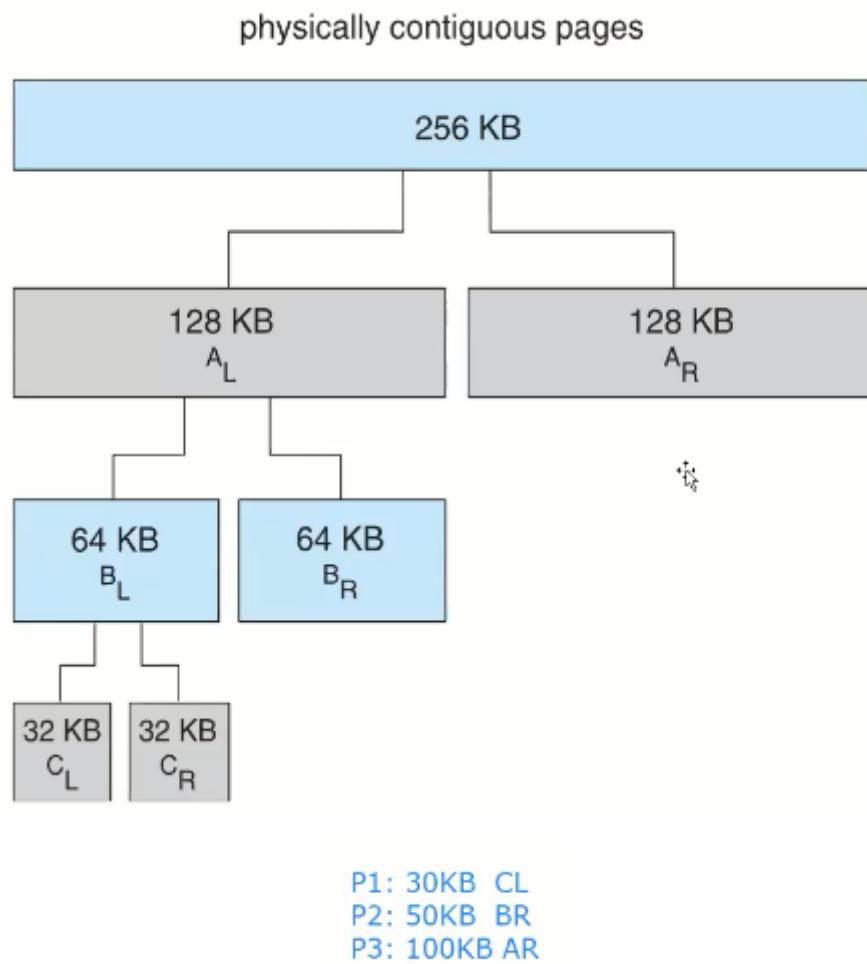
## 固定分配

- 等分 100个页框5个进程  $100/5=20$
- 按照进程大小分配

## 全局分配与局部分配

依据所有进程的执行情况与单个进程执行情况

## Buddy System 伙伴系统



分配2的倍数个页

可以获得大块的连续内存空间

避免外在碎片

容易回收

## Slab Allocator 块分配器

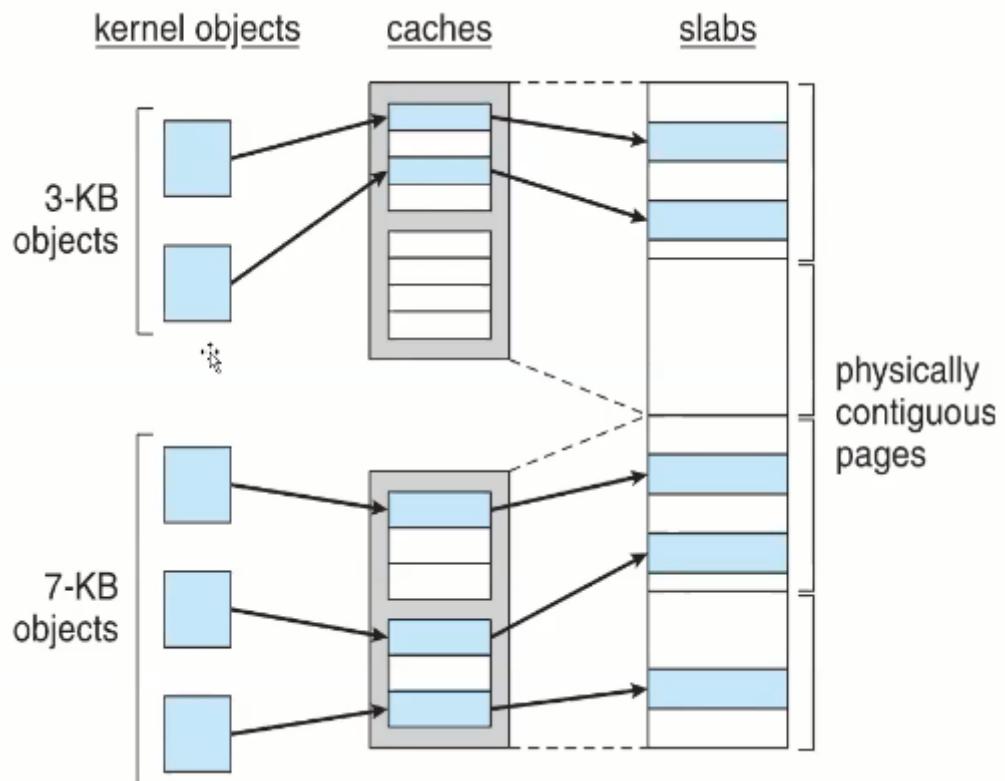
针对需要少量页的数据

可以减少内在碎片

实现方法：

在cache中离散

在slabs整合为连续页



## trashing 颠簸

**Thrashing.** A process is busy swapping pages in and out

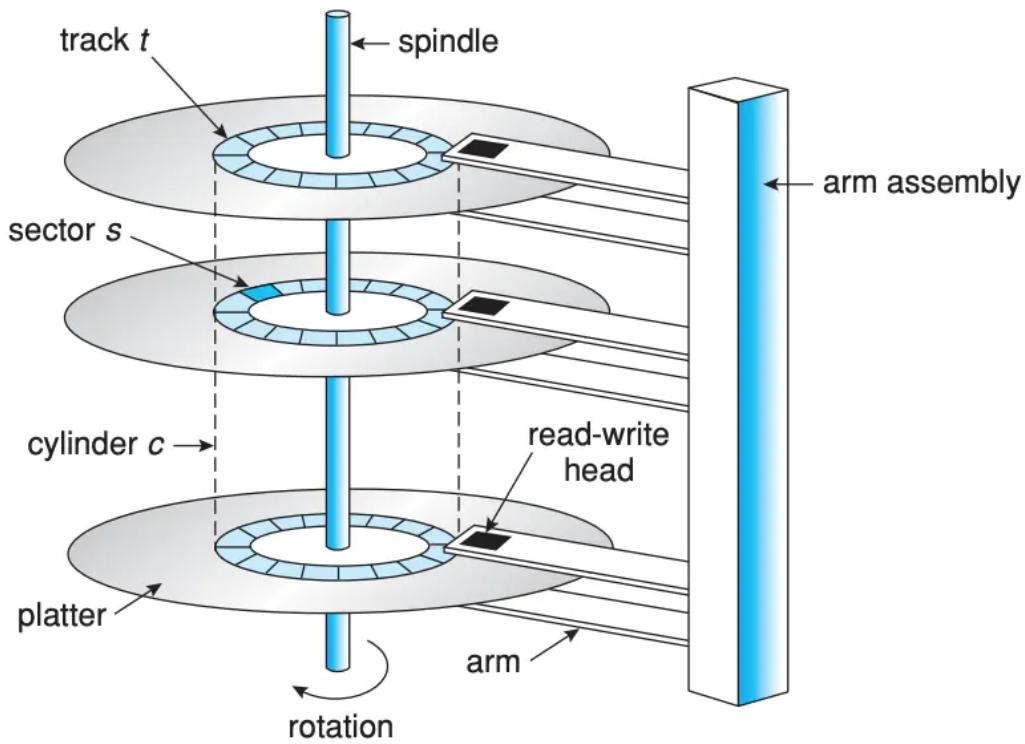
分配不到足够的连续内存空间

# Ch 11 大容量存储

## 磁盘

现代计算机系统最常用的外部大量存储就是磁盘或者硬盘。

### 结构

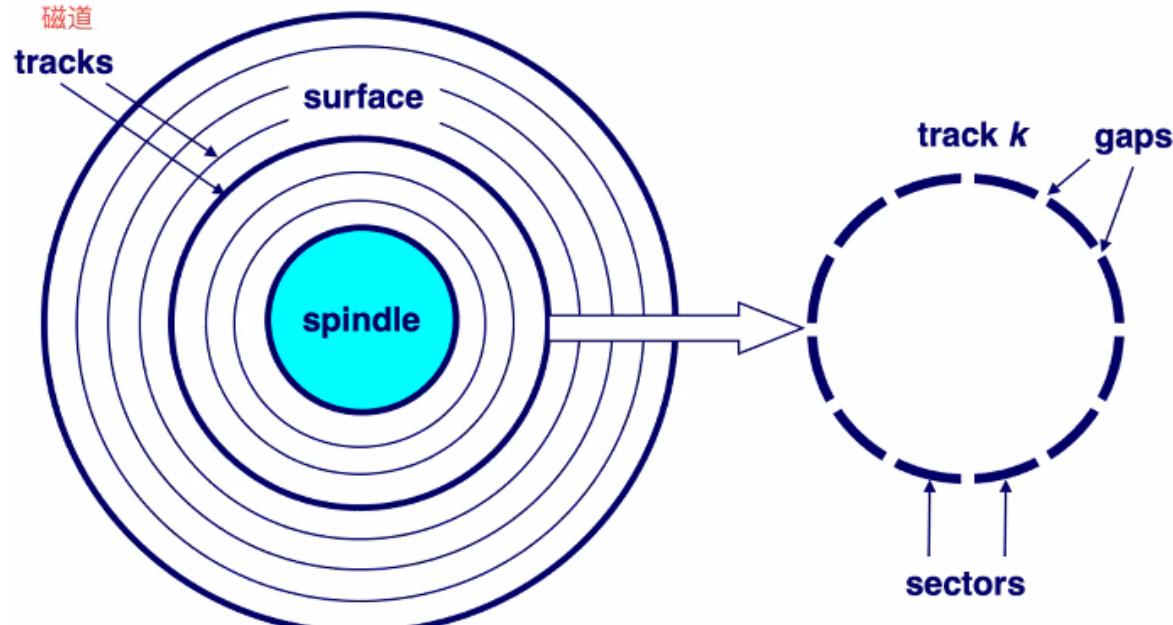


**Figure 10.1** Moving-head disk mechanism.

磁盘的大致结构为图上所示。他包括一系列的盘片（platter），在盘片的两面都涂着磁质材料。通过在盘片上进行磁性记录可以保存信息。

读写磁头（read-write head）在一个盘片“飞行”从而可以读写数据。磁头是悬挂在磁臂（arm）上的，磁臂将所有磁头作为一个整体和进行一起移动。

每一个磁盘的盘片（platter）上逻辑的可以分为磁道（track）。磁道又可以分为扇区（sector）。不同盘片的磁道形成了柱面（cylinder）。每个磁盘驱动器有数千个同心柱面，每个柱面又有数百个扇区。

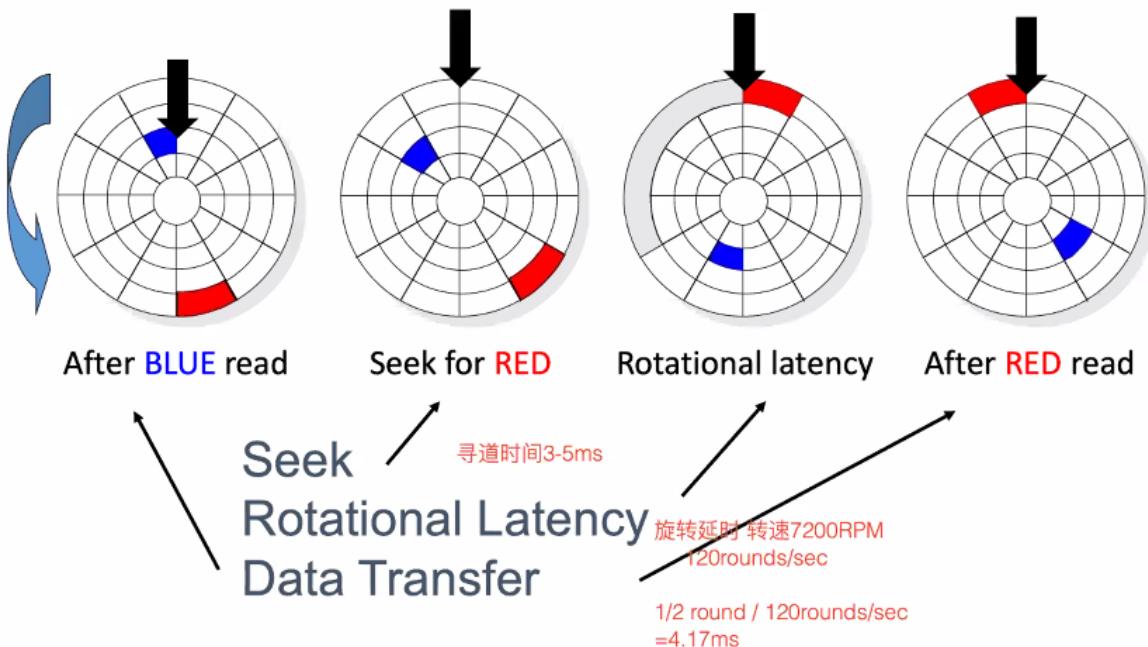


## 衡量指标

一般判断一个磁盘的好坏是通过**传输速率**和**定位时间**。

**传输速率**: 在驱动器和计算机之间数据传输的速率。

**定位时间**又分为**寻道时间** (移动磁臂到所有的柱面所需时间) 和 **旋转延时** (旋转磁臂到所要的扇区所需要的时间)。



data transfer大概100-150MB/s

传输小数据，微秒级别，data transfer忽略不计。

传输大数据，如10GB需要data transfer100s，寻道时间和旋转延时忽略不计

对于100MB连续数据：1s

对于100MB 1KB离散数据： $100000 * 5 = 500$ s

## 控制

磁盘驱动器通过I/O总线的一组电缆连到计算机。在使用中有多种可用的总线，这就对应了多种 I/O 接口。其中包括 **硬盘接口技术 (ATA)**，**串行接口 (SATA)**，外部串行 (eSATA),通用串口总线 (USB)。

数据传输的总线由**控制器**来进行。在计算机的那一头的叫做**主机控制器**，在I/O驱动器的那一头为**磁盘控制器**。

I/O操作在执行的时候，计算机会下达一个命令到主机控制器，接着主机控制器通过消息将该命令下达给磁盘控制器，磁盘控制器开始操作磁盘驱动器工作，磁头和磁臂开始工作。**磁盘控制器通常具有内置缓存，他会把数据传输到缓存上，而到主机的传输，则由内置缓存和主机控制器进行。**

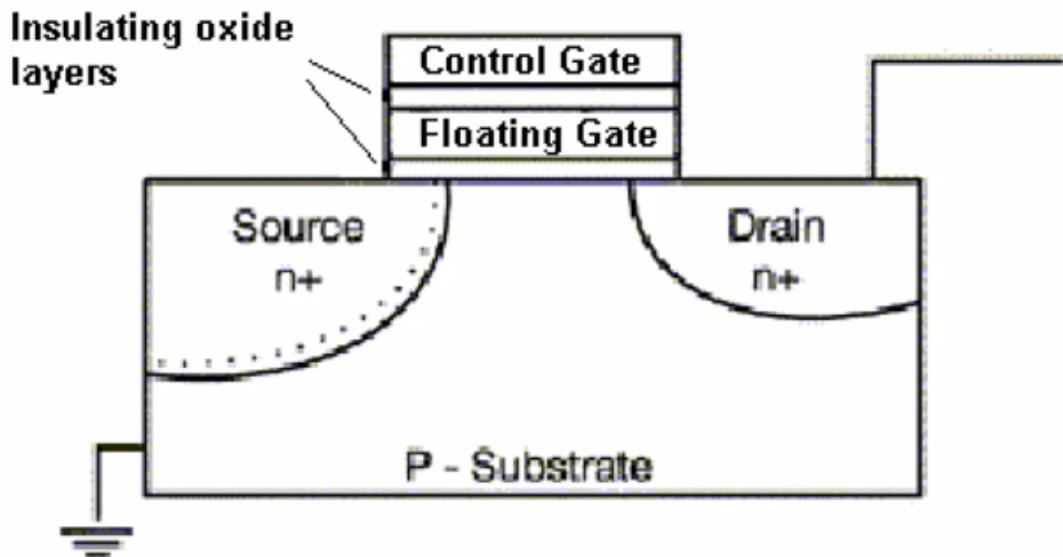
## SSD

SSD 具有与传统硬盘相同的特性，容量大，但是它会更可靠，没有移动部件；而且更快，没有寻道时间和延迟。

- 性能好
- 功耗低
- 体积小

## 闪存颗粒

## Flash Memory Cell



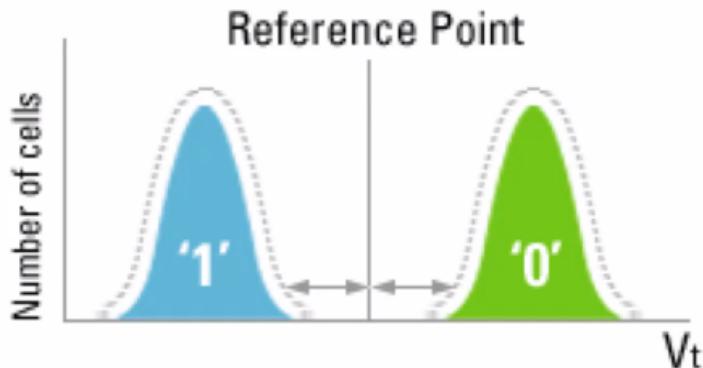
cell电子多时为导体

电子少时为绝缘体

### SLC

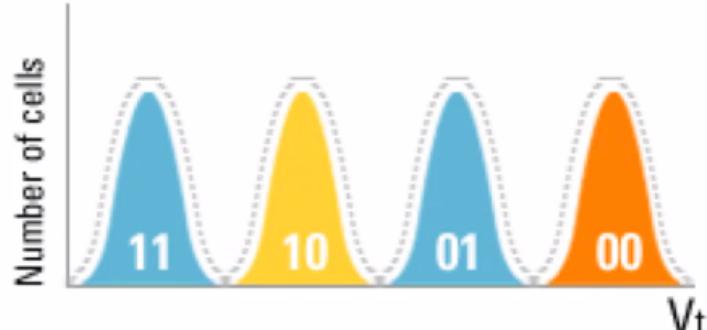
性能很好，寿命长，价格昂贵

**SLC**  
One bit per cell

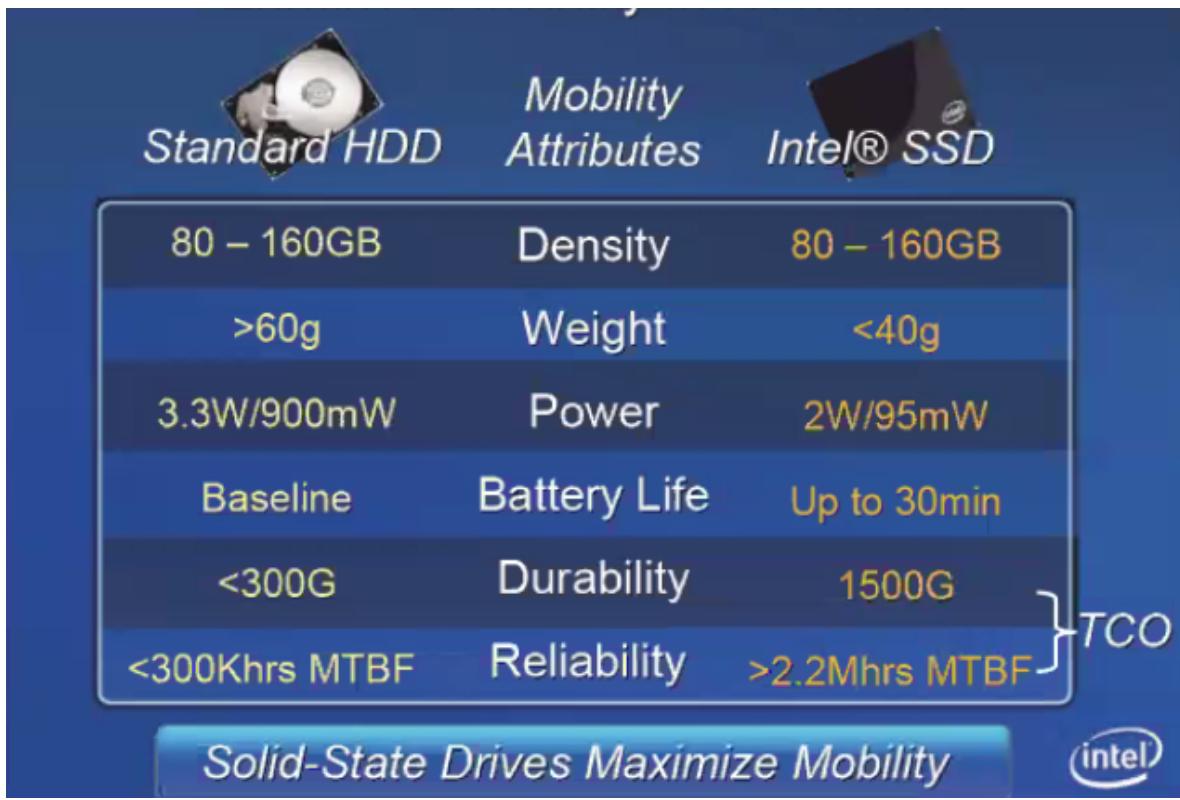


### MLC

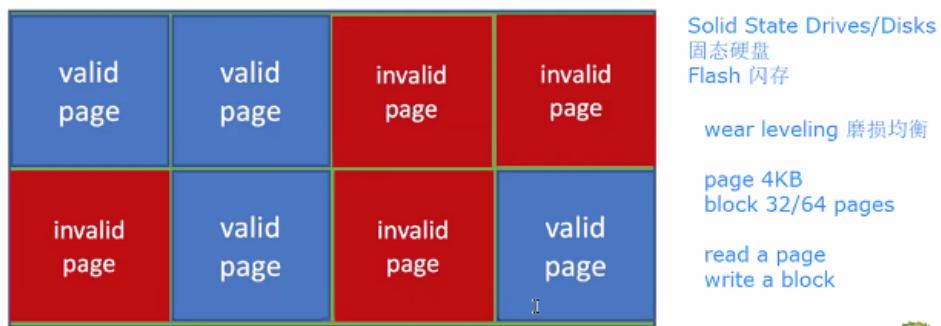
**MLC**  
Two bits per cell



## SSD与HDD



- With no overwrite, pages end up with mix of valid and invalid data
- To track which logical blocks are valid, controller maintains **flash translation layer (FTL)** table
- Also implements **garbage collection** to free invalid page space
- Allocates **overprovisioning** to provide working space for GC
- Each cell has lifespan, so **wear leveling** needed to write equally to all cells



NAND block with valid and invalid pages



## 磁带

- 保存时间长 30y

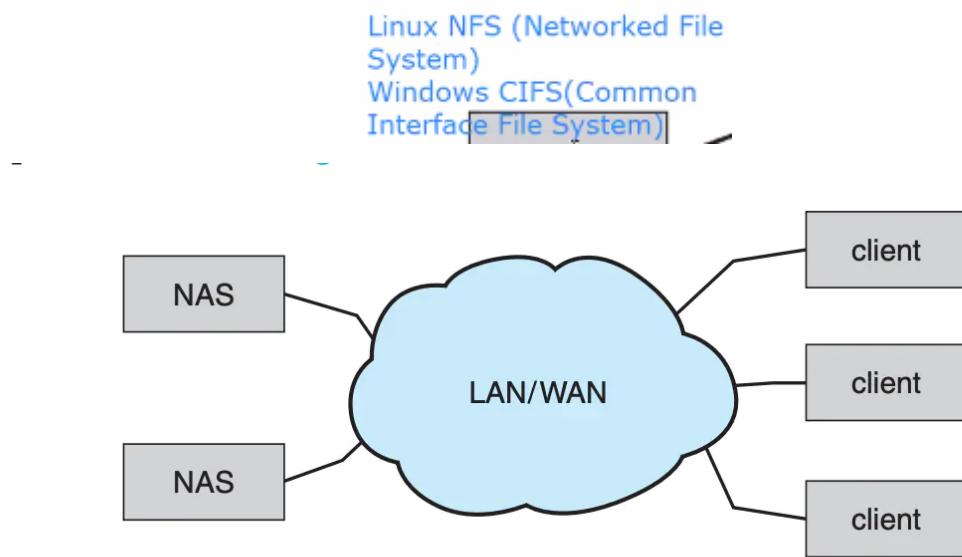
## 磁盘连接

计算机访问磁盘有两种方式。一种方式是通过I/O端口，称为主机连接存储，另外一种是分布式系统的远程主机（称为网络连接存储）。

主机连接存储：主机连接通过本地的端口来访问存储。这些端口使用多种技术。典型的台式机采用I/O总线，一般SATA的接口更为常用。

网络连接存储：网络连接存储（NAS）设备是一种专用的存储结构，可以通过数据网络来进行远程访问。客户机通过远程过程调用（RPC）来访问连接存储。RPC通过TCP或者UDP协议进行访问。

协议：

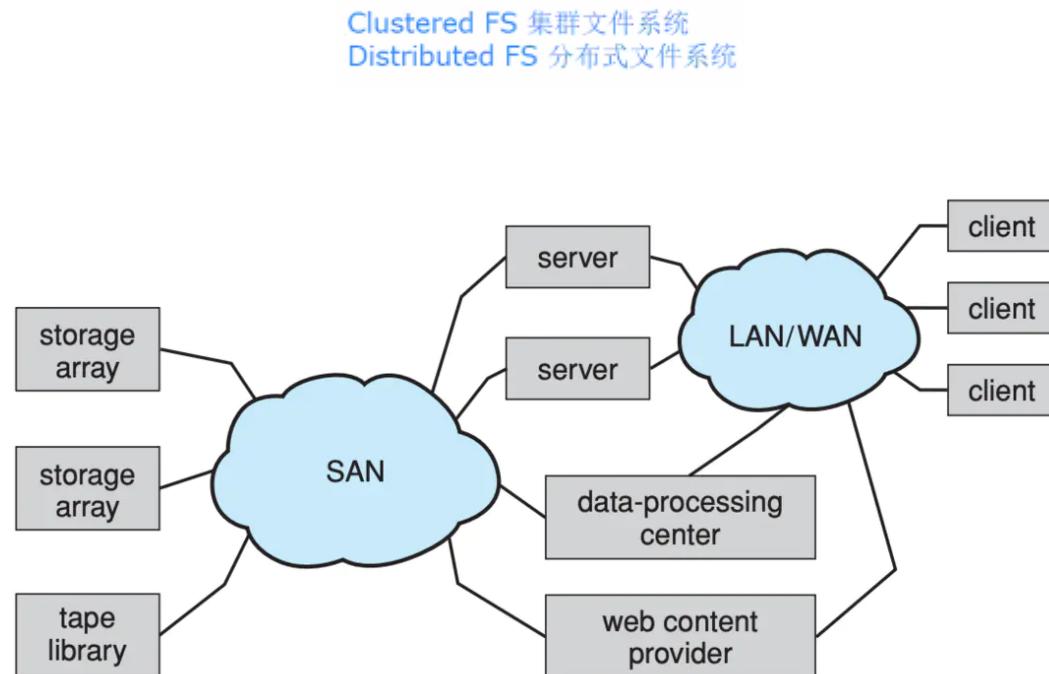


**Figure 10.2** Network-attached storage.

## 存储区域网络

网络连接存储有一个缺点：存储I/O操作消耗太多的网络带宽。从而增加网络的延迟。这在大型C/S的模式下非常严重。

存储区域网络(SAN)为专用网络，它是在网络协议上使用了一种新的存储协议。



**Figure 10.3** Storage-area network.

它的优势是在于灵活。多个主机和多个存储阵列（storage array）之间可以连接到同一个SAN上，存储可以动态分配到主机。也就是说当主机的磁盘不够时，他可以通过配置SAN来进行动态的磁盘分配。

## 磁盘调度

每当进程需要进程磁盘访问的时候，他就向操作系统发出一个系统调用。每个请求需要一些信息：

- 这个操作是输入还是输出
- 传输的磁盘地址是什么
- 传输的内存是什么
- 传输的扇区数是多少

如果磁盘驱动器和控制器空闲，则立即执行请求，否则需要将这个请求添加到磁盘驱动器的请求队列中。

对于处于请求队列的所有I/O请求，操作系统在进行选取的时候，应该需要一种调度算法来保证I/O设备的效率最大化。那么如何来评估这个效率呢

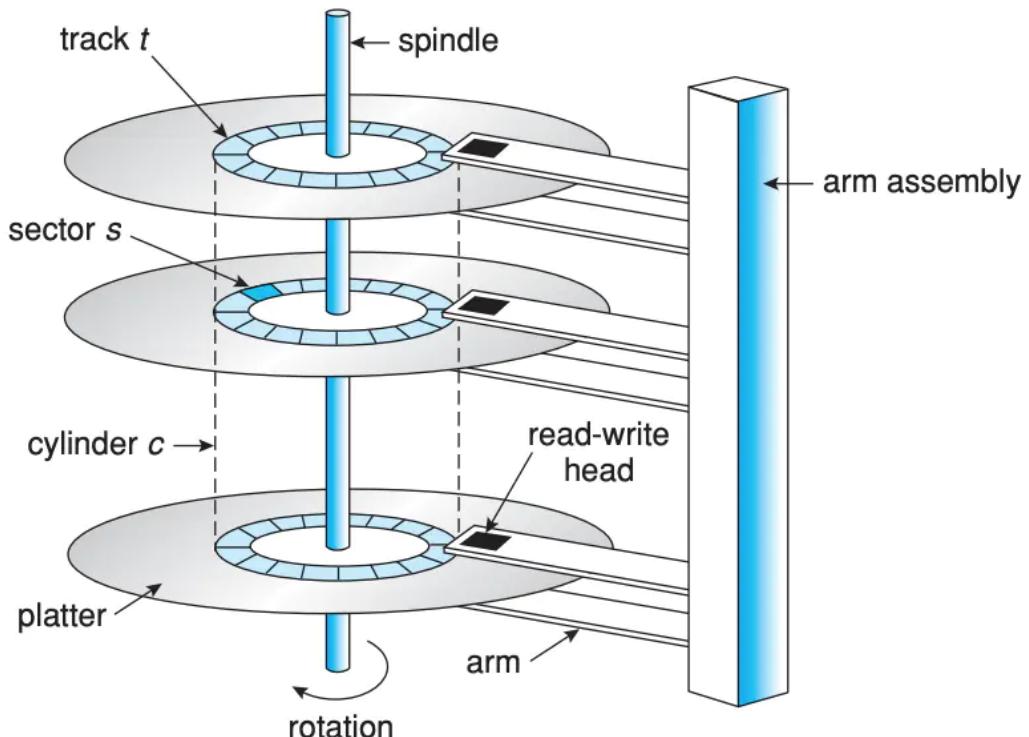
对于磁盘驱动器（就是开始那张图的全部零件加起来的装置）是根据两个量来进行评估的。**访问速度**和**磁盘带宽**。

根据上面的图再来看看下两个概念：

**寻道时间**：移动磁臂到所有的柱面所需时间。

**旋转延时**：旋转磁臂到所要的扇区所需要的时间。

**磁盘带宽**：传输自己的总数除以所需要的总时间。

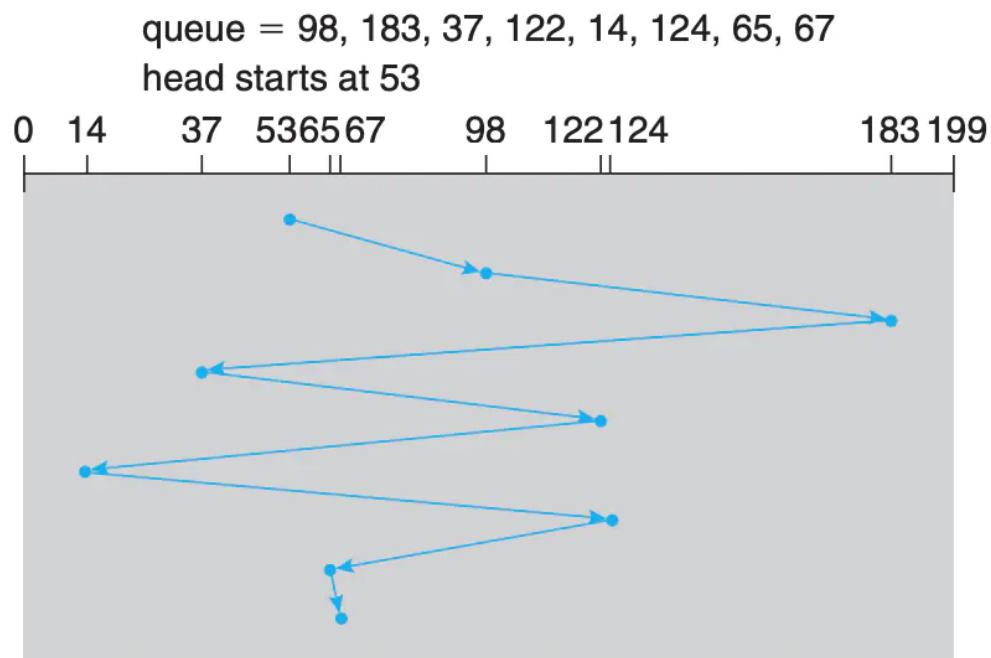


**Figure 10.1** Moving-head disk mechanism.

在考虑三个因素的情况下，下面有不同的调度算法。

### FCFS(先来先服务)

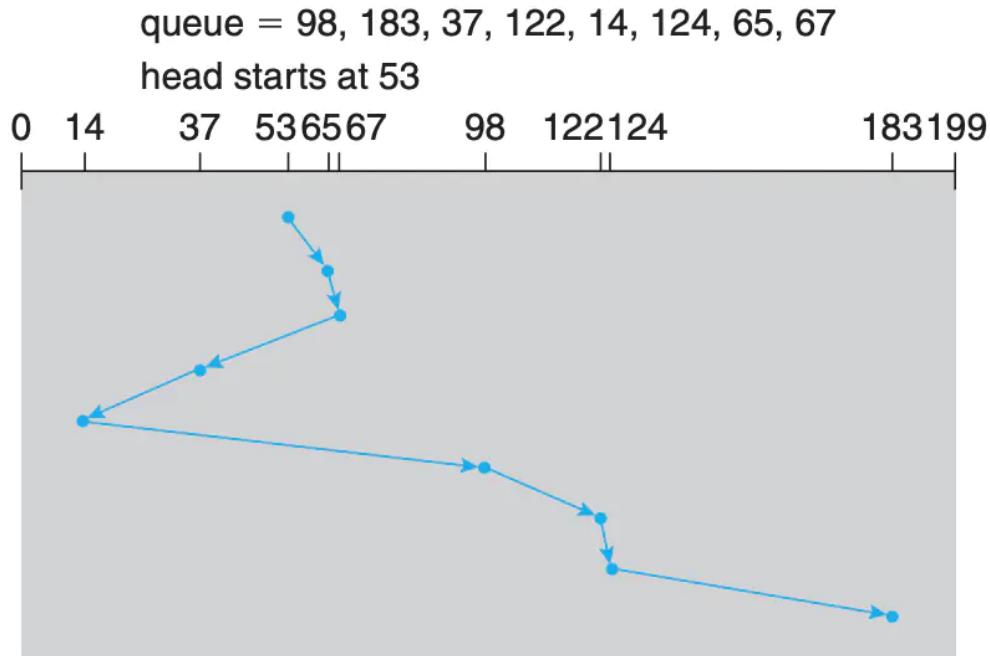
和进程的CPU调度和内存调度一样，先到的先服务。实现起来比较简单，直接通过一个FIFO的队列实现即可。



**Figure 10.4** FCFS disk scheduling.

### 最短寻道时间优先 (SSTF)

顾名思义：当在移动磁头到别处处理请求之前，处理靠近当前磁头位置最近的请求。  
和CPU调度的最短作业优先 (SJF) 调度一样，它会导致磁盘请求的饥饿。



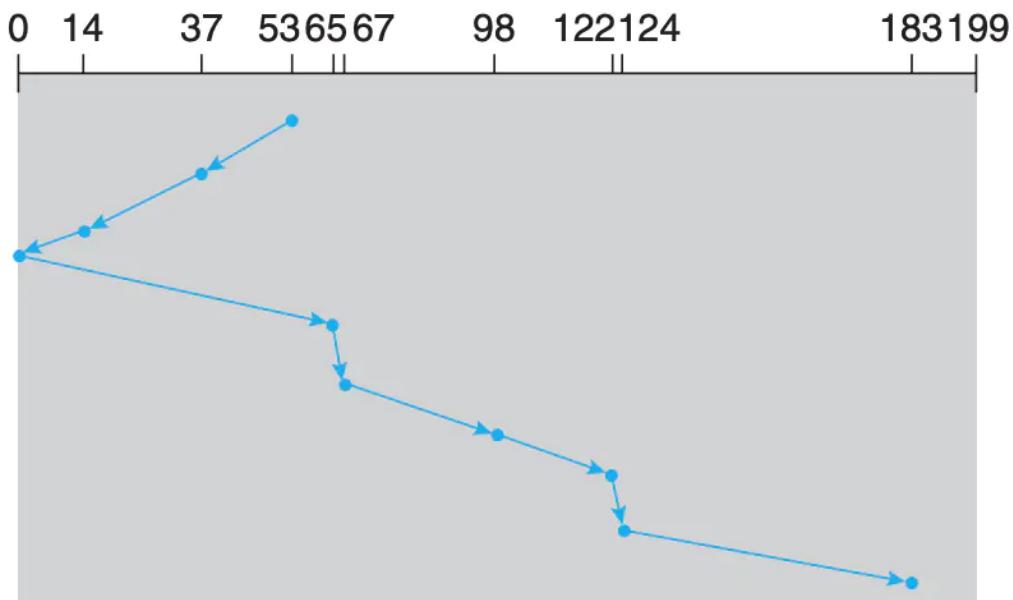
**Figure 10.5** SSTF disk scheduling.

### SCAN调度(扫描)

这种调度方法也叫电梯调度，这个比较好理解，磁臂从磁盘的一端开始，向另一端移动，在移过每个柱面时，处理请求。当到达磁盘的另一端时，磁头反方向移动，继续处理。

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



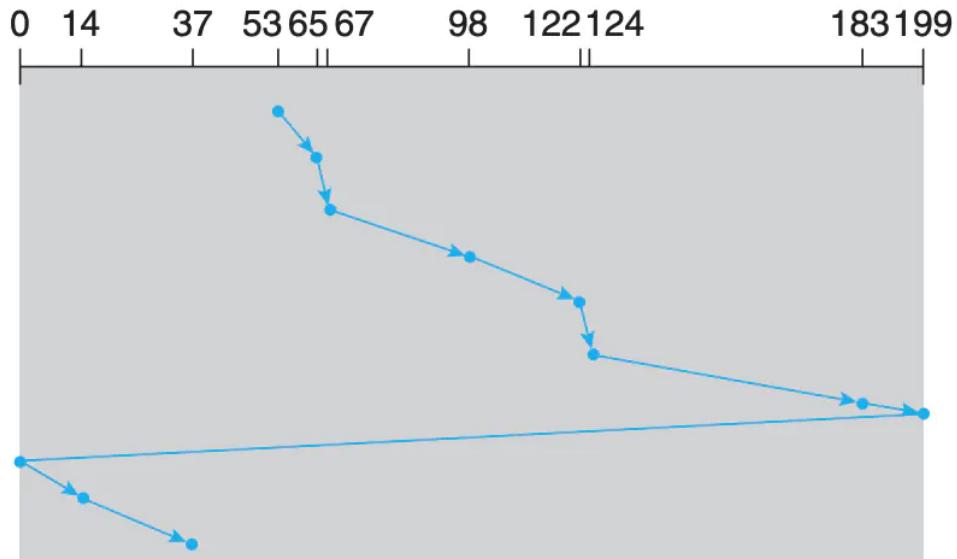
**Figure 10.6** SCAN disk scheduling.

### C-SCAN(循环扫描)

这个是电梯算法的一个变种，它会把向一个方向进行扫描，直到磁盘的另一端结束，然后立马回到磁盘的另外一端，再次进行扫描。

queue = 98, 183, 37, 122, 14, 124, 65, 67

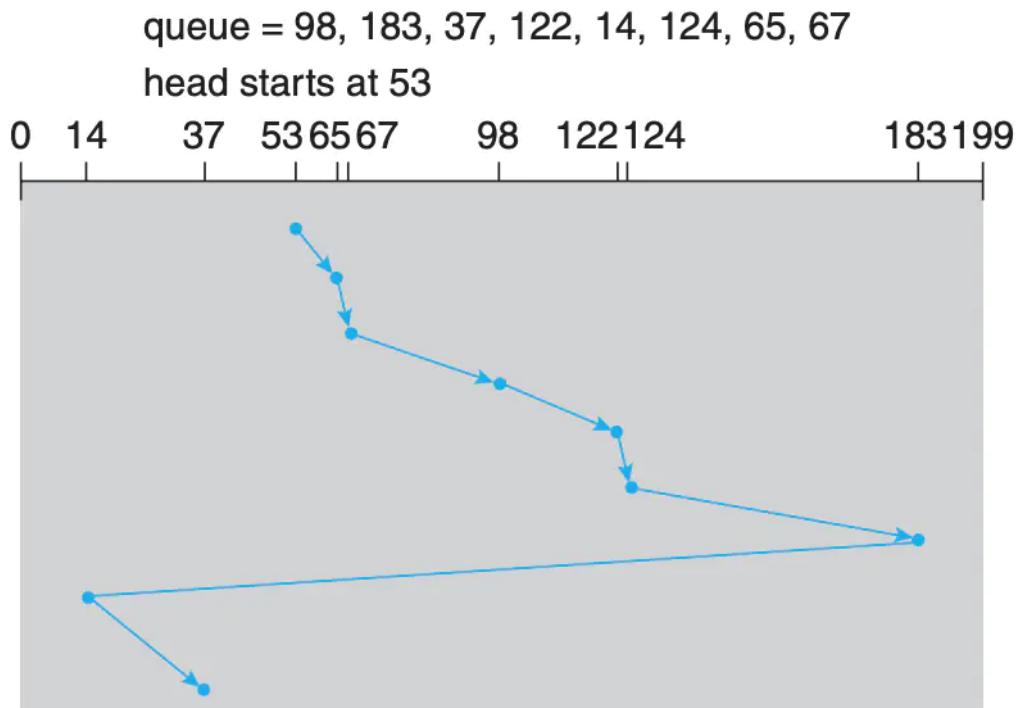
head starts at 53



**Figure 10.7** C-SCAN disk scheduling.

### LOOK 调度

它是C-SCAN的变种，它在扫描的时候并不是扫描整个磁盘，而是在队列中的最小和最大之间。



**Figure 10.8** C-LOOK disk scheduling.

如何选择调度算法？

SSTF是最常见的，但是对于磁盘负荷较大的系统，SCAN和C-SCAN会更好，因为他不会产生饥饿。当然对于任何调度算法，性能还在余于请求的数量和类型。除此之外，文件的分配方式也会影响磁盘的服务请求。后面来讲文件系统结构和方式。

对于没有移动磁头的SSD来说，它通常可以用FCFS的策略。

## 磁盘管理

### 磁盘格式化

一个新的磁盘是一个空白盘：它只是一个磁性记录材料的盘子。在磁盘上可以存储数据之前，他必须分为扇区，以便磁盘控制器能够读写。这个过程叫做**低级格式化**或者**物理格式化**。

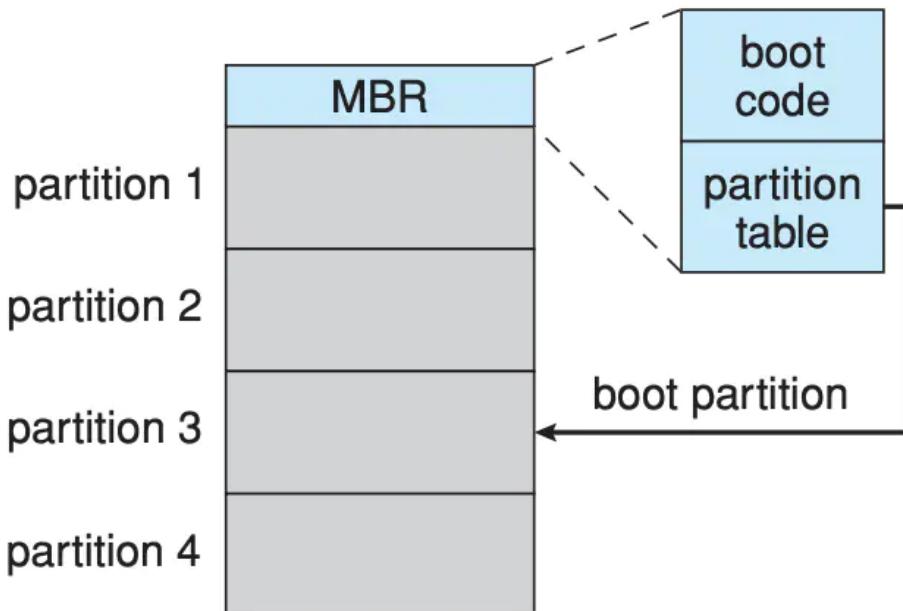
低级格式化为每个扇区使用特殊的数据结构，填充磁盘。每个扇区的数据结构通常包括**头部**，**数据区域**和**尾部**。头部和尾部包含了一些磁盘控制器的使用信息。如扇区的纠错代码（Error-Correcting, ECC）。当控制器通过正常I/O写入一个扇区的数据时，ECC采用根据区域所有字节而计算的新值来进行更新。在读取一个扇区时，ECC会重新计算。并且与原来的值进行比较。如果存储的和计算的值不一样，表示扇区已经损坏，并且扇区可能已经损坏。

大多数磁盘在制作的时候已经进行了低级格式化。这种格式化也方便磁盘制造商进行测试磁盘，并且初始化逻辑块号到无损磁盘扇区的映射。

### 引导块

为了开始运行计算机，如打开电源或者重启时，它必须有一个**初始化的程序**来运行。它初始化系统的所有部分，从CPU寄存器到设备控制器和内存，接着启动操作系统。

对于大多数计算机，自举程序处于**只读存储器（ROM）**。它是只读的数据，不会被计算机病毒影响，所以位于ROM的自举程序一般放在固定位置，它的功能就是在通电或者复位的时候，从磁盘上调入完整的**引导程序**。这个引导程序存储在磁盘上，通常称为启动块。



**Figure 10.9 Booting from disk in Windows.**

windows允许将磁盘分为多个分区，有一个分区是**引导分区（boot partition）**，其中包含了操作系统和设备程序。一般的windows系统将引导代码存放在磁盘的第一个扇区，它为**主引导记录（MBR）**。windows的启动过程，通电后，首先加载并允许在系统ROM中的代码，这个代码会指示系统从MBR上加载引导代码。除了引导代码，MBR还包括一个分区表和一个表示（标示那个分区为引导系统）。

## 坏块

因为磁盘具有移动部件并且容错差（磁头在磁盘表面上方），容易出现故障。有时候，这种故障时彻底的，需要更换磁盘，并且从备份介质上将其内容恢复到新盘。更为常见的是，一个或者多个扇区坏掉，而且大多数磁盘出厂时就有坏块。

复杂的磁盘在恢复坏块时，它的控制器维护磁盘内的坏块列表。这个列表在**出厂低级格式化**时初始化，并且在磁盘使用寿命内更新。低级格式化将一些块放在一边作为备用，操作系统看不到这些块。控制器可以采用备用块来逻辑地代替坏块。这种方案叫做**扇区备用**。

一般的扇区处理：

- 操作系统尝试读取逻辑块87
- 控制器计算ECC,发现87为坏块。他向操作系统汇报这一情况。
- 当操作系统下次启动时可以运行特殊命令，请求控制器转换成备用块替代坏块
- 之后的访问87，就使用的是备用块。

它的替代方案是：**扇区滑动**。假定17号扇区变坏，并且第一个可用备用逻辑扇区在202之后，那么他就会进行滑动，类似于在有序数组的插入一个中间值，所有的扇区都往后移动一个扇区。直到扇区18写入扇区19。然后以后的17就映射到现在的18。

## RAID结构

**磁盘冗余阵列（Redundant Array of Independent Disk, RAID）** 是由多块硬盘通过RAID控制器控制管理组成的一个更大容量的逻辑盘，在操作系统中识别为一个盘符。

## RAID

什么是RAID?

RAID :磁盘阵列

1988年由加利福尼亚大学伯克利分校提出来的。

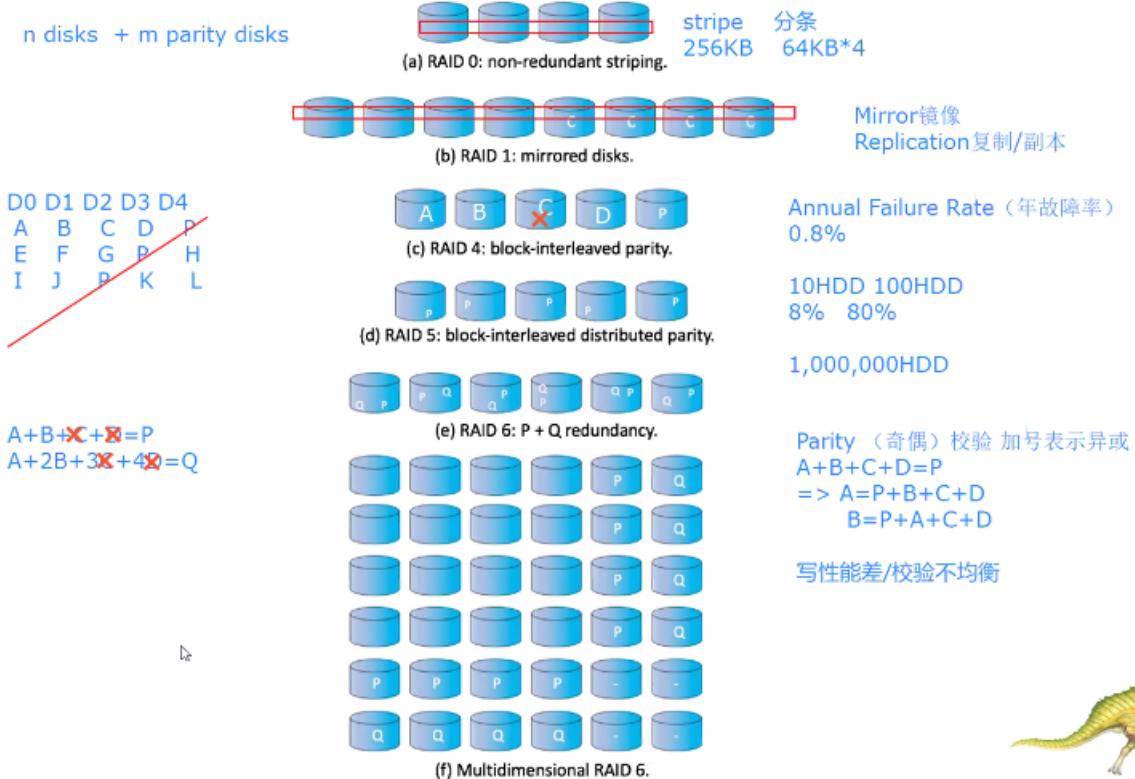
多个磁盘合成一个“阵列”来提高更好的性能，冗余，或都提供。

RAID 可以提高磁盘I/O能力，也可以提高磁盘的耐用性。

RAID的实现方式：

- 可以外接：通过扩展卡提供适配性
- 内置式RAID:主板集成RIAD控制器，安装方便，目前主流的服务器都是内置式的。
- 软件实现：通过os系统实现。

## RAID 级别

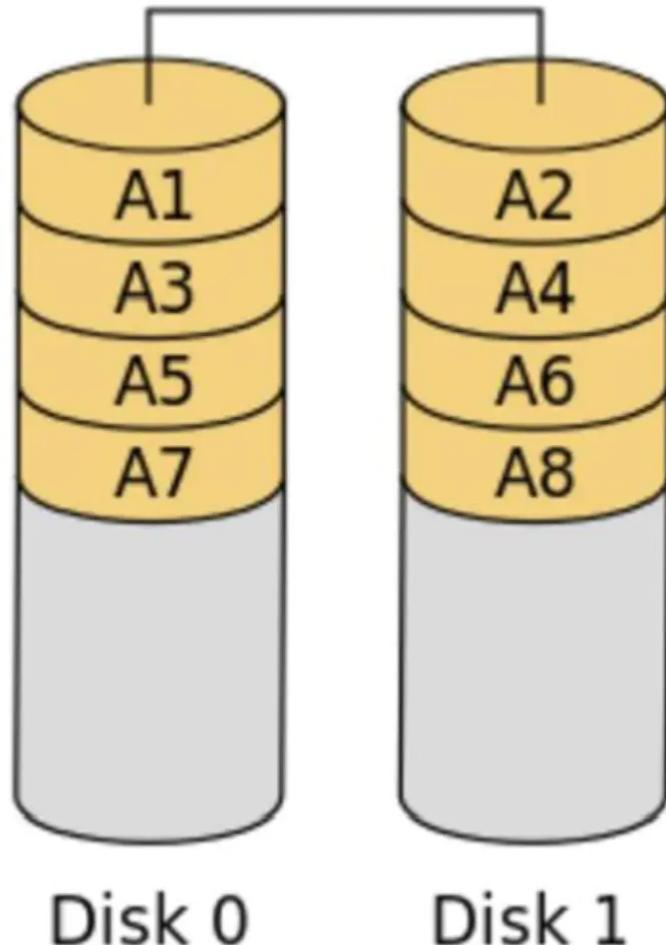


RAID-0 : RAID-0 是将多块硬盘捆绑成为一个大容量的逻辑磁盘，可以同时从多块硬盘读取数据，也可以同时往多块硬盘写入数据。磁盘I/O 是单块硬盘的多倍。在所有RAID模式下，同等硬盘下，RAID-0是最快的。但是RAID0 没有数据冗余能力，一个磁盘损坏，所有数据都丢失。

特点：

- 读写都得到提升
- 可用空间大 ( $n \times \min$ )
- 没有容错能力，只要有一块硬盘损坏，数据就丢失了。
- 需要2个或以上组成。

# RAID 0



## RAID-1：

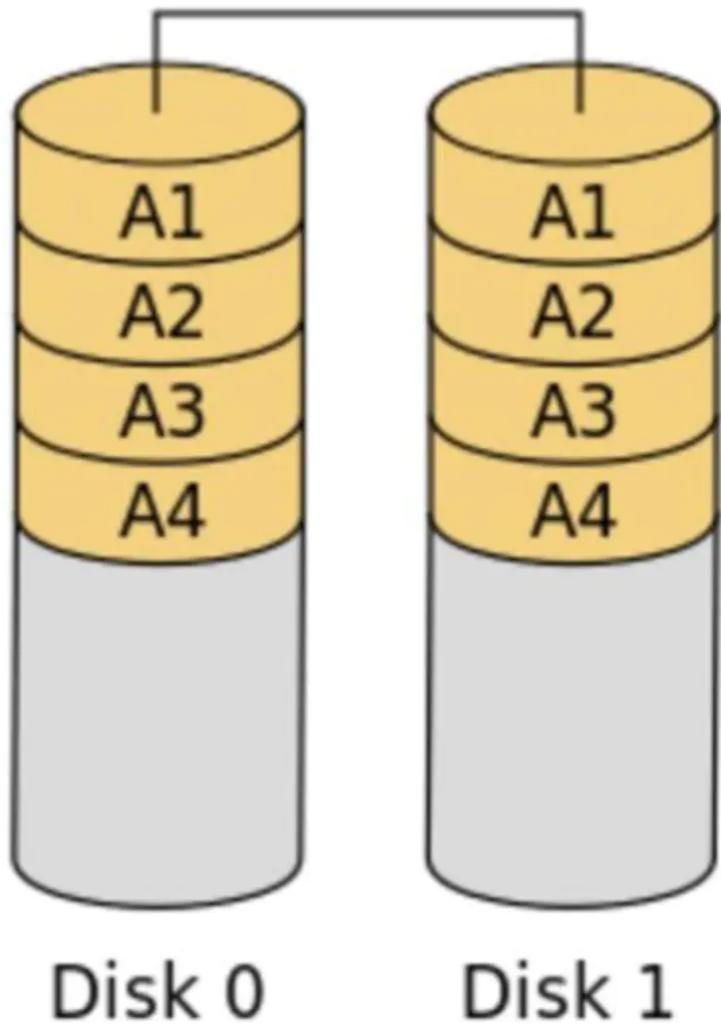
RAID-1也被称为镜像，仅用于两块硬盘的情况下，同样的数据在两块硬盘上分别存储一份，两块硬盘中的数据完全相同。即使有一块硬盘出现问题，也不会影响数据安全与中断系统运行。

RAID-1 主要用于数据安全性比较高的环境，比如，数据库，电脑的系统盘等。RAID1不会提高性能。

特点：

- 读性能提升，但写性能下降
- 可用空间只有百分之50，相当于有一个备份盘。
- 有冗余能力
- 至少要2个或以上的硬盘

# RAID 1



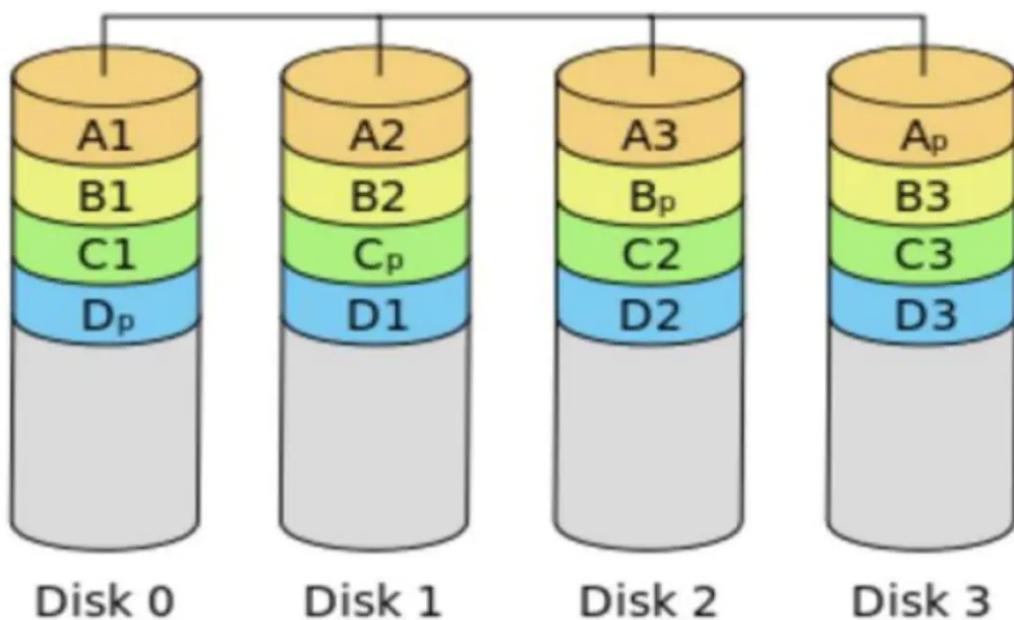
RAID-5：

RAID 5至少需要三个硬盘，RAID5 不是对存储的数据备份，而是把数据和相对应的奇偶校验信息存储到组成RAID5 的各个磁盘上，并且就检验码信息和相对应的数据分别存储在不同的磁盘上。当RAID5 的任何一个磁盘数据反生损坏，可以利用剩下的数据和相应的奇偶校验码信息去恢复被损坏的数据。

特点：

- 读写性能提升
- 可用空间是 $n-1$
- 有容错能力，但最多1块硬盘损坏。
- 需要3块或以上的磁盘组成。

## RAID 5



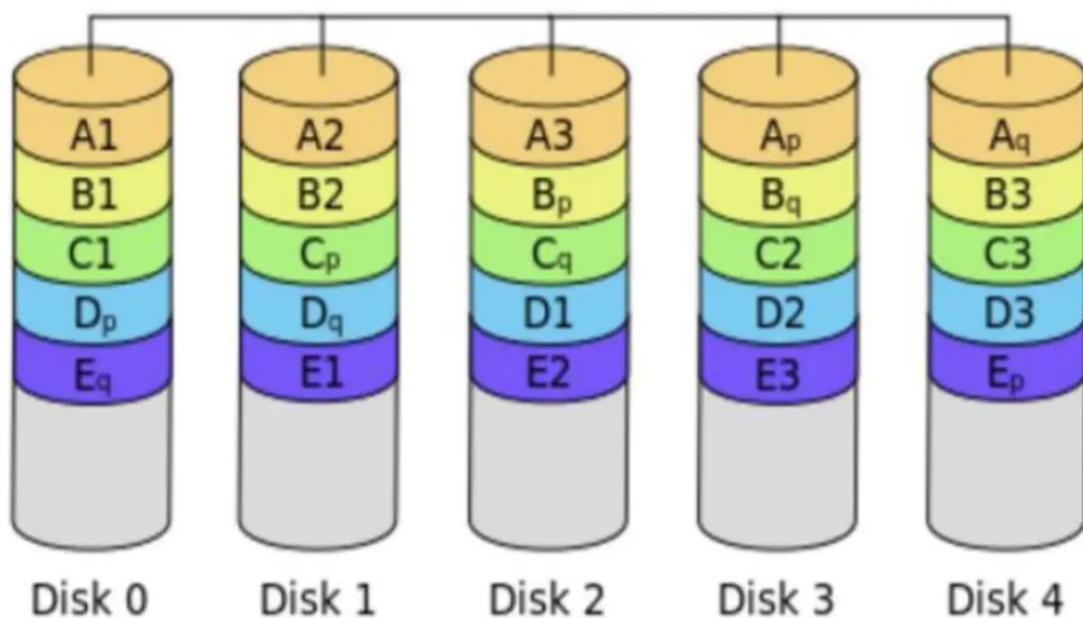
RAID-6 :

RAID6 至少需要4个硬盘，与RAID5相似，RAID6也是不对存储数据进行备份，而是把数据和相对应的奇偶校验信息存储在RAID6的各个磁盘，与RAID5不同的是，RAID6有两个校验盘，即使同时有两块硬盘出问题，它也可以利用剩下的数据和相应的奇偶信息恢复损坏的磁盘。

特点：

- 读写性能提升
- 可用空间n-2
- 有容错能力，允许最多2块硬盘损坏
- 需要4块或以上组成

## RAID 6



磁盘阵列比较表

R A I D 模 式	硬 盘 数 量	磁 盘 冗 余	可 用 容 量	读 取 性	写 入 能	优势	劣势	适合的应用
JB O D	随 意	无	全 部 容 量	1	1	每块硬盘在操作系统中显示为一个独立的盘符，各个硬盘之间相互独立。	只是单盘的速度	数据离线备份
0 ≥ 2		无	全 部 容 量	n	n	磁盘空间利用率100%，在硬盘数量相同时，速度是各种RAID模式中最快的。	一块硬盘出现故障，所有数据都会丢失。采用RAID0模式，数据务必要做好备份，有备份就很安全。	当硬盘数量在2-6块，而且希望传输速度快时适用
1	2	1	1	1	1	安全性好	速度没有提升	电脑系统盘，2块盘时要求数据安全
5 ≥ 3	1	n - 1	n - 1	n - 1	n - 1	既提供磁盘冗余，也提高了传输性能。	初始化与RAID重建需要的时间比较长	3-8盘位存储系统的不错选择。
6 ≥ 4	2	n - 2	n - 2	n - 2	n - 2	既提供磁盘冗余，也提高了传输性能。安全性比RAID5更高。	初始化与RAID重建需要的时间比较长。磁盘利用率比RAID5低。	6-12盘位存储系统的不错选择。
10	4	2	2	2	2	安全性高，速度是单盘的2倍	一般仅用于4块硬盘的情况下，磁盘利用率50%	4盘位存储系统
5 ≥ 0 6	2	n - 2	n - 2	n - 2	n - 2	安全性高，随着硬盘数量的增加，可以提供很好的传输性能。硬盘数量越多，速度越快。	当硬盘数量超过12块时，是一种不错的选择。超过18块时，可以做多组RAID5	一般用于10-16盘位的存储系统

## Ch 12 I/O System

计算机的两个主要工作是I/O和处理，在很多情况下，主要工作是I/O,而处理只是附带的。例如：当浏览器网页和编辑文本时，直接兴趣是读取或者输入信息，而非计算。

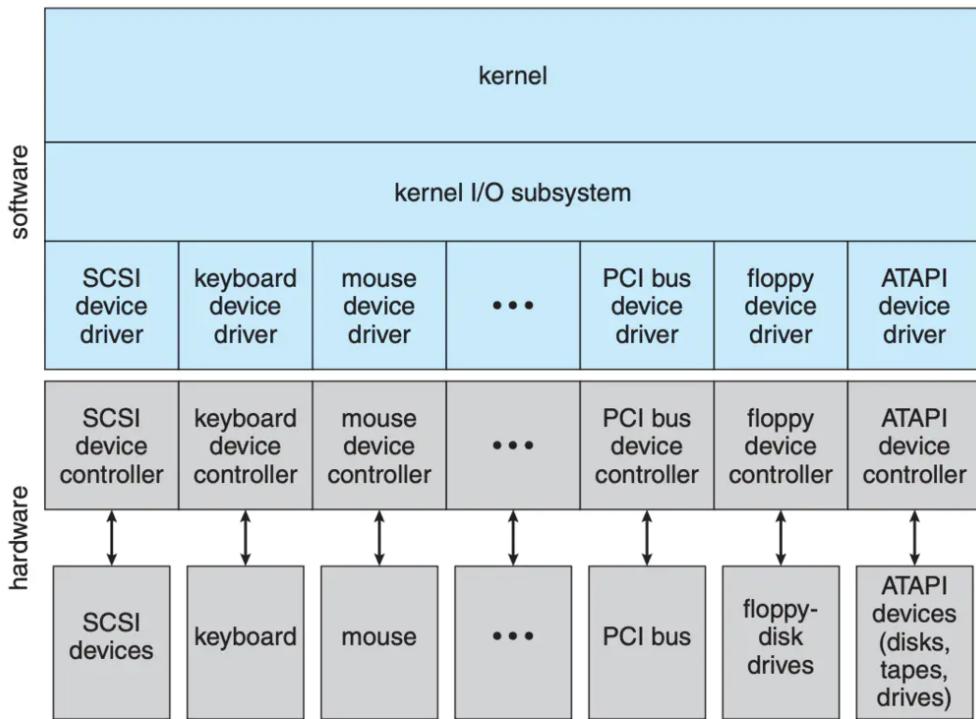
计算机的操作系统I/O 的功能是，管理和控制I/O 操作和I/O设备。

### 概述

计算机设备的控制是操作系统的设计人员的主要关注之一。因为I/O设备的功能与速度差异很大。所以需要采用不同的方法来控制设备，这些方法构成了内核的I/O子系统。

I/O设备技术呈现两个冲突趋势。一方面。软件和硬件的接口标准化日益增长。这个趋势有助于将改进和升级设备，从而可以集成到计算机中。另一方面，I/O设备的种类也日益增多。为了封装各种设备的细节与特点，操作系统内核采用设备驱动程序。

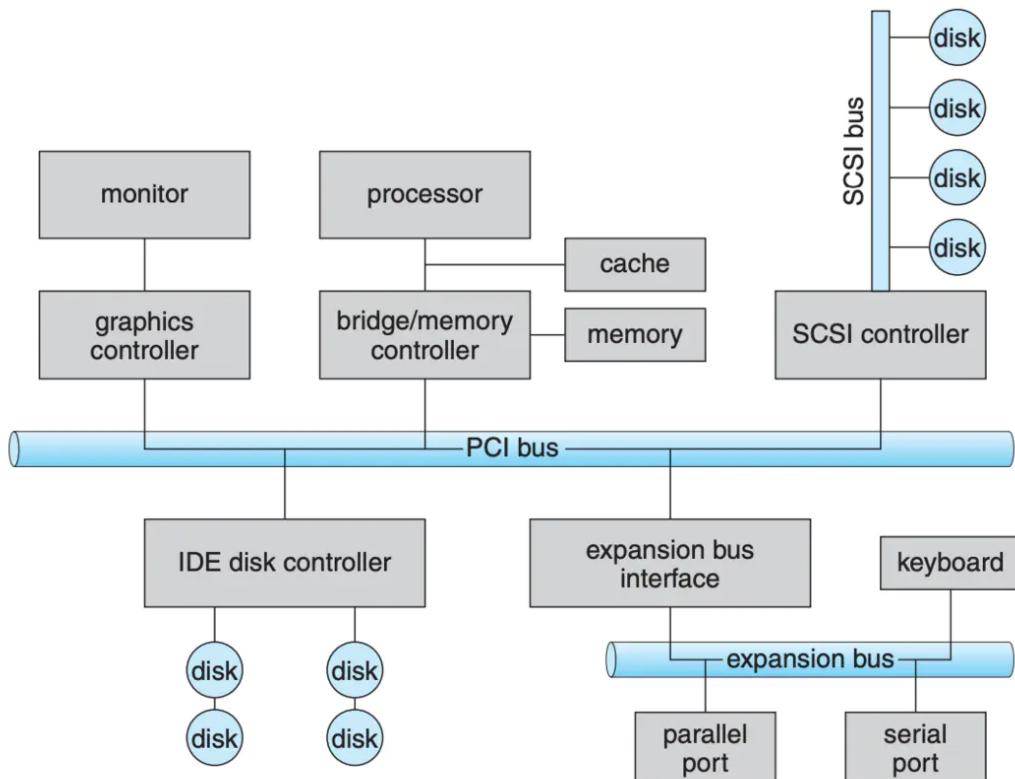
**设备驱动程序**为I/O子系统提供了统一的设备访问接口。



**Figure 13.6** A kernel I/O structure.

## I/O硬件

设备与计算机系统的通信，可以通过电缆或者空气来进行发送信息。设备与计算机的通信通过一个连接点或者端口，比如串行端口。连接这些端口电缆就是总线。



**Figure 13.1** A typical PC bus structure.

**总线**在计算机体系中应用广泛。上图是一个典型的PC总线结构，图中，PCI总线将内存和各个设备进行连接。其他的设备都需要连接到PCI总线上。扩展总线连接相对较慢的设备。如键盘和串口USB端口。4个磁盘通过小型计算机系统接口(SCSI)总线连接到SCSI控制器。

**控制器**是可以操作端口和总线或者设备的一组电子器件。串行端口控制器是计算机内的单个芯片，用于控制串口线路信号。SCSI控制器一般是比较复杂，遵循SCSI协议。一般为单独的电路板。他包括，处理器，微代码和一些专业内存。

处理器如何对控制器发出命令和数据以便完成I/O传输？

控制器是具有多个寄存器的，用于数据和控制信号。处理器通过读写这些寄存器的位模式与控制器通信。这种通信的一种方式是，通过使用特殊的I/O指令 针对I/O端口 地址传输一个字节或字。

I/O指令出发总线线路，选择适当的设备，并将位移入或者移出设备寄存器。

或者设备控制器支持内存映射I/O。在这种情况下，设备控制寄存器被映射到处理器的地址空间，处理器执行I/O 请求是通过标准数据传输指令读写映射到物理内存的设备控制器。

I/O端口通常由四个寄存器组成，即状态，控制，数据输入，数据输出寄存器。

数据输入寄存器：被主机读出以便获取数据。

数据输出寄存器：被主机写入以发送数据。

状态寄存器：包含一些主机可以读取的位，例如当前命令是否完成，数据输入寄存器是否有数据可读，是否出现设备故障等。

控制寄存器：主机写入，以便启动命令或者改变设备模式。

## 轮询

主机与控制器之间交互的完整协议可以很复杂，但基本的握手概念是比较简单的。握手概念可以通过例子来解释。

假设采用2个位协调控制器和主机之间的生产者和消费者关系，控制器通过状态寄存器的忙来显示状态。控制器工作忙时就置忙，而可以接收下个命令时就清忙位。主机通过命令 寄存器的 命令就绪位来表示意愿。当主机有命令需要控制器执行时，命令就绪位被置为1。

当主机需要通过端口来输出数据时，主机与控制器之间的握手的协调如下：

1. 主机重复读取忙位，直到该位清零。
2. 主机设置命令寄存的写位，并写出一个字节到数据输出寄存器。
3. 主机设置命令就绪位。
4. 当主控器注意到命令就绪位被设置，则设置忙位。
5. 控制器读取命令寄存器，并看到写命令。它从数据输出寄存器中读取一个字节，并向设备执行I/O 操作。
6. 控制器清除命令就绪位，清除状态寄存器的故障位表示设备I/O成功，清除忙位表示完成。

对于每个字节重复这个循环。在步骤1 中，主机处于忙等待，或者轮询。

## 中断

基本的中断机制的工作原理是：CPU 硬件有一条线，称为 中断请求线（IRL）。CPU 在执行完每条指令的后，都会检查是否有中断请求，当CPU检测到控制器已在IRL上发出了一个信号时，CPU执行状态保留并且跳到内存固定位置的中断处理程序中。中断处理程序确定中断原因，并执行必要处理，执行状态恢复。并且返回中断指令以便CPU 回到中断前的位置继续执行。

设备控制器通过中断请求线发送信号引起中断，CPU捕获中断，并分派到中断处理程序，中断处理程序通过处理设备来清除中断。

## 直接内存访问（DMA）

对于执行大量传输的设备，例如磁盘驱动器，如果通过昂贵的CPU来按字节的方式发送数据到控制寄存器，则会增加CPU的负担。所以为了避免这种情况，将这一部分任务交给一个专用的处理器（DMA控制器）。

在启动DMA 传输时，主机将DMA 命令块写到内存。该块包含传输来源地址，目标地址和传输的字节数。CPU将这个命令块写入DMA控制器后，继续工作。DMA 再继续直接操作内存总线，将地址放到总线，在没有主CPU的帮助下，进行传输。

DMA控制器与设备控制器之间的握手，通过一组对称的 DMA 请求和DMA确认来完成的。当有数据需要传输时，设备控制器发送信号到 DMA 请求线路。这个信号使得DMA控制器占用内存总线，发送所需地址到内存地址总线，并发送信号到DMA确认线路。当设备控制器收到DMA确认信号时，他就传输数据到内存，并且清除DMA 请求信号。

当DMA 控制器占用内存总线时，CPU被暂时阻止访问总线，但是仍然可以返回主缓存。虽然这种周期窃取会减慢CPU的计算，但是将数据传输给工作交给DMA控制器通常能够改进总的系统性能。

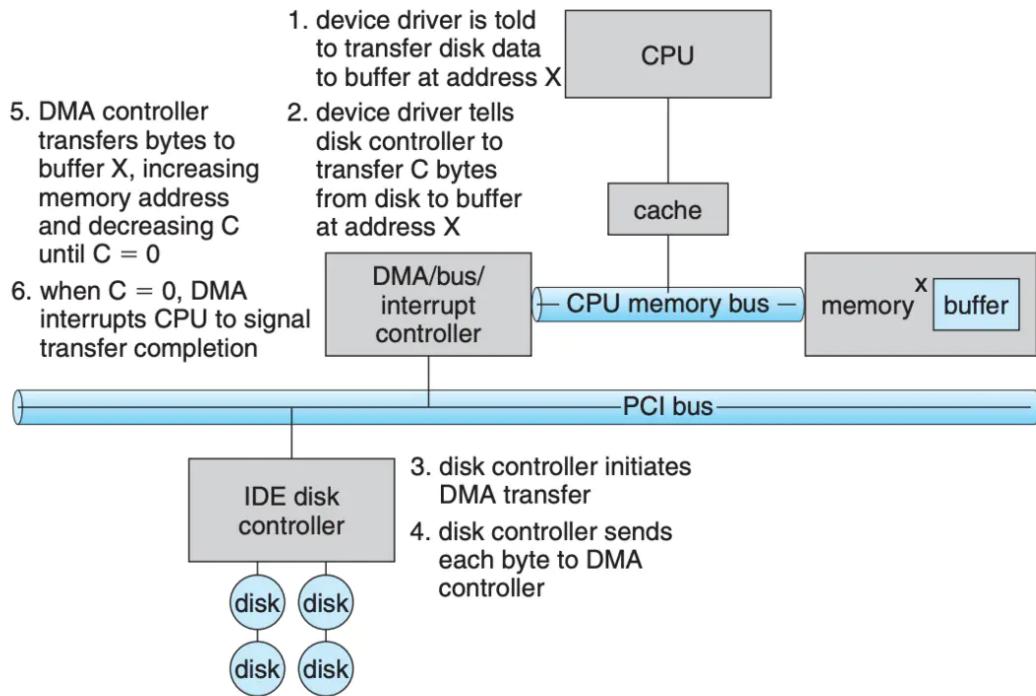
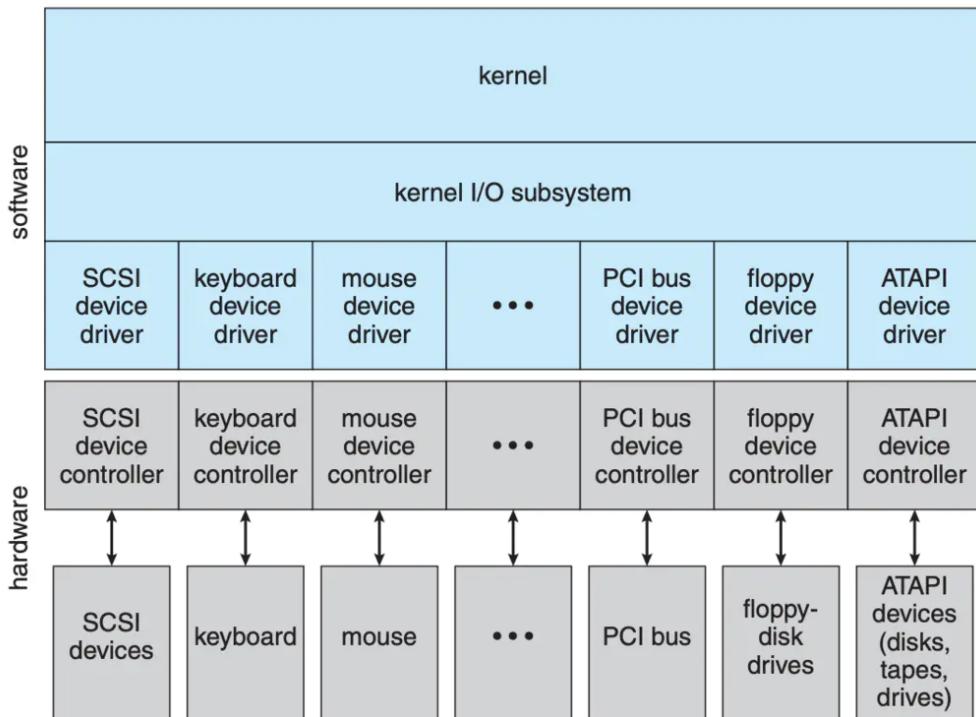


Figure 13.5 Steps in a DMA transfer.

## 应用程序I/O接口



**Figure 13.6** A kernel I/O structure.

在操作系统对I/O设备封装的时候，因为各个设备的不同，所以会形成上面的那个I/O结构。接下面主要说明一些不同的设备的处理方法。

## 块与字符设备

**块设备接口**为磁盘驱动器和其他基于块设备的访问。它提供了read(), write(), seek()等接口屏蔽了底层设备的差异。

键盘是**字符流设备的**。通常使用get() 和put() 接口来进行封装。其他设备还有键盘，鼠标，modem。

## 网络设备

因为网络I/O的性能和寻址的特点不同于磁盘I/O，大多数操作系统提供的网络I/O接口不同于 read()-write()-seek()操作。许多系统都使用socket接口。

为了支持实现服务器，套接字还支持了select()函数。调用select可以得知，那个套接字已经有消息接收和处理。

## 时钟与计时器

大多数计算机都有硬件时钟来定时器。以便实现三种功能

1. 获取当前时间
2. 获取经过时间
3. 设置定时器，以便在T时出发操作X

测量经过的时间和触发操作的硬件称为**可编程间隔定时器**。他可以设置一段时间，然后触发中断；调度程序采用这种机制产生中断，以便抢占时间片用完的进程。

I/O系统使用这种机制，定期刷新脏的缓存到磁盘，网络子系统中，定时取消由于网络拥塞或者故障而产生的太慢的操作。

## 非阻塞I/O与异步I/O

系统调用接口的另一方面设计选择阻塞I/O 和非阻塞I/O。当应用程序执行阻塞系统调用时，应用程序的执行就被挂起。应用程序会从操作系统的运行队列移到等待队列。等待系统调用完成后，再回到运行队列。

有些用户级进程需要使用非阻塞I/O。比如，一个用户接口，用来接收键盘和鼠标的输入，同时处理数据并显示到屏幕。

针对非阻塞系统，系统一般提供了异步系统调用。异步调用立即返回，无需等待I/O完成。应用程序继续执行代码，等将来I/O 完成了，通过设置应用程序地址空间的某个变量，或通过触发信号或软件中断，或者执行回调函数，来通知应用程序。

## 内核I/O子系统

内核提供与I/O相关的许多服务。包括调度，缓冲，假脱机，设备预留及错误处理。

### I/O调度

调度一组I/O请求意味着，确定好顺序，来执行它们。应用程序执行系统调用的顺序很少是最佳的。

操作系统开发人员通过为每个设备维护一个请求等待队列，来实现队列。当应用程序发出阻塞I/O的系统调用时，该请求被添加到相应的设备的队列。I/O调度程序重新安排队列顺序。以便提高总的效果和应用程序平均的响应时间。之前就有提到过磁盘的调度算法。

当内核支持异步I/O时，它必须能够同时跟踪许多I/O请求。为此，操作系统可能会将等待附加到设备状态表中。内核管理此表，其中每个条目对应每个I/O 设备。每个表条目表明设备的类型，地址和状态。如果设备忙于一个请求，则请求的类型和其他参数都被保存在该设备的表条目中。

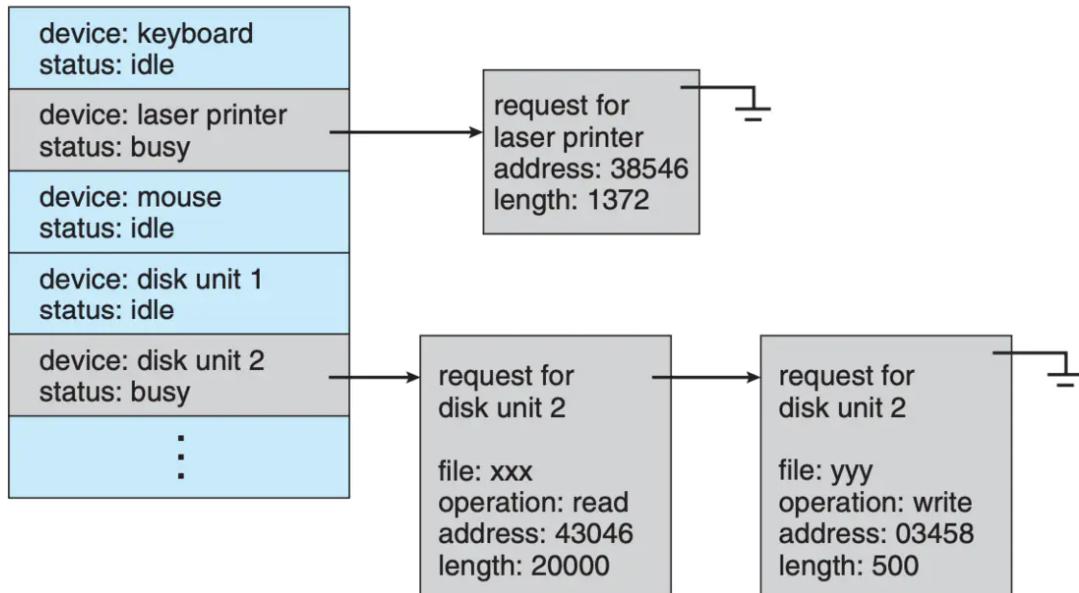


Figure 13.9 Device-status table.

### 缓冲

调度I/O操作是I/O子系统提高计算机效率的一种方法。另外一种方法是通过缓冲，缓存和假脱机，使用内存或磁盘的存储空间。

**缓冲区**是一块内存区域，用于保存在两个设备之间或者设备和应用程序之间传输的数据。采用缓冲有三个理由：

1. 处理数据流的生产者和消费者之间的速度不匹配。例子：通过调制解调器接收一个文件，并保存在硬盘。调制解调器的速度比硬盘慢1000倍。这样创建一个缓冲区在内存中，以便累积从调制解调器出接受的字节。当整个缓冲区填满时，就可以通过一次磁盘操作写入磁盘。

2. 调节传输代价不一致的设备。这种不一致在计算机网络中特别常见，缓冲区大量用于消息的分段和重组。在发送端，一个大的消息分成若干个小的网络分组。这些网络分组通过网络传输，而接收端将它们放在重组缓冲区内，以便完成的源数据进行重组映射。
3. 支持应用程序I/O的复制语义。假设应用程序有一个数据缓冲区，它希望写到磁盘。它调用系统调用write，提供缓冲区的指针和表示所写字节数量的整数。在系统调用返回后，如果应用程序更改缓冲区的内容，那么会发生什么？采用复制语义，写到磁盘的数据版本保证是系统调用时的版本，而与应用程序缓冲区的任何后续操作无关。  
系统调用write 返回到应用程序之前，复制应用程序缓冲区到系统内核缓冲区。磁盘写入通过内核缓冲区来执行。以便应用程序缓冲区的后续更改没有影响。

## 缓存

缓冲和缓存的区别是：缓冲可以保存数据项的唯一现有版本。而缓存只是提供了一个位于其他地方的数据项的更快存储版本。

缓存和缓冲的功能不同，但是有时一个内存区域可以用于两个目的。

如为了保留复制语义和有效地调度磁盘I/O。操作系统采用内存中缓冲区来保存磁盘数据。这些缓冲区也用作缓存，以便提高文件的I/O效率。这些文件可被多个程序共享，或者更快速的写入和重读。

当内核收到文件I/O请求时，内核首先访问缓冲区缓存。以便查看文件区域是否已经在内存中可用。

## 假脱机和设备预留

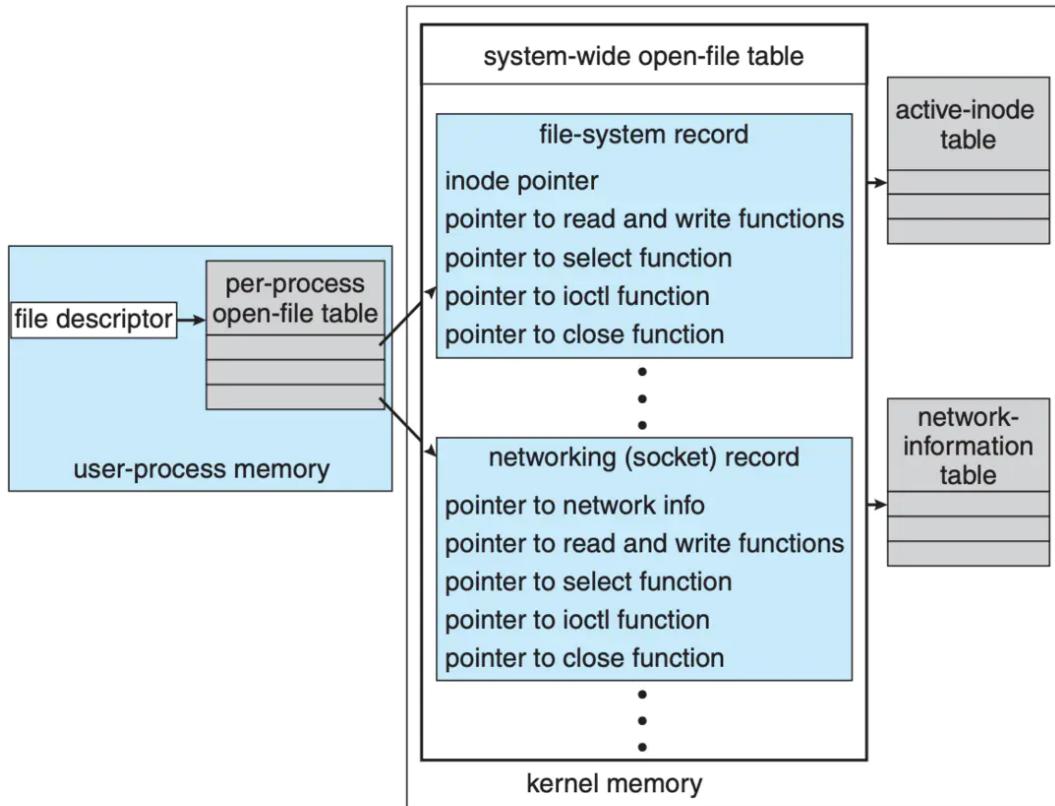
假脱机是保存设备输出的缓冲区，这些设备，如打印机，不能接受交叉的数据流。虽然打印机只能一次打印一个任务，但是多个应用程序可能希望并发打印输出，而不能让他们的输出混合在一起。操作系统拦截所有的打印数据。来解决这个问题。

应用程序的输出先是假脱机到一个单独的磁盘文件。当应用程序完成打印后，假脱机系统排序相应的假脱机文件。以便输出到打印机。

## 内核数据结构

内核需要保存I/O组件的使用状态信息。它通过各种内核数据结构来完成。内核使用许多类似的结构来跟踪网络连接，字符设备和其他的I/O操作。

Unix提供了各种实体的文件系统访问，如用户文件，原始设备和进程的地址空间。虽然这些实体都支持read操作，但是语义不同。当读取用户文件时，内核首先需要检查缓冲区缓存，然后决定是否执行磁盘I/O。当读取原始磁盘时，内核需要确保，请求大小是磁盘扇区大小的倍数而且与扇区边界对齐。当读取进程映像时，内核秩序从内存读取数据。



**Figure 13.12** UNIX I/O kernel structure.

## I/O请求转换硬件操作

操作系统如何将应用程序请求连到网络线路或者特定的磁盘扇区。

从磁盘文件读文件。应用程序通过文件名称引用数据。对于磁盘，文件系统通过文件目录对文件名进行映射，从而得到文件的空间分配。

但是如何建立文件名称到磁盘控制器的连接？

MS-DOS 文件名称的第一部分，在冒号之前表示特定硬件设备的字符串。C：是主硬盘的每个文件名称的第一部分。C: 通过设备表映射到特定的端口地址。由于冒号分隔符，设备名称空间不同于文件系统的名称空间。

Unix 通过常规文件系统的名称空间来表示设备名称。与具有冒号分隔符的MS-DOS系统不用， Unix没有路径 没有明确的设备部分。为了解析路径， Unix 检查安装表内的名称，以查找最长的匹配前缀；安装表的相应条目给出了设备名称。

# Ch 13-15 文件系统

## 基本概念

### 文件

文件是操作系统对存储设备的物理属性加以抽象定义的逻辑存储单位。**是数据的逻辑视图**

文件包括文件的属性和操作：

属性包括：名称，标识符，类型，位置，尺寸，权限，创建和修改时间，文件的所有者和使用者等等。

文件的操作包括：创建文件，读文件，写文件，重新定位文件的读写位置，删除文件和截断文件。

### 进行文件操作的实质？

上面的文件操作涉及搜索目录。为了方便文件操作的搜索更简单，许多系统在首次使用文件之间进行系统调用open()。操作系统在**内核上**维护一个**打开文件表**用于维护所有打开文件的信息。当进行访问文件时，直接使用这个表的索引指定文件。

当多个进程一起访问同一个文件时，操作系统采用两级文件表，进程维护一个文件表和操作系统维护一个进程表。单个进程的文件表执行整个系统的打开文件表。一旦有进程打开了一个文件表，系统表就会包含该文件的条目，当一个进程执行open()，进程表和系统表都增加一个条目（加入系统表中没有这个文件时）。当系统表中有这个条目时，它会进行将文件打开计数加1。每次close() 将打开计数减1。当为0时，可以删除。

### 读取文件时会有读者-作者问题，怎么解决？

文件系统一般会提供共享锁和独占锁。

mandatory共享锁类似于读者优先的读写者问题。

advisory独占锁类似于写者优先的读写者问题。

## 文件系统的结构

文件类型也可用于指示文件的内部结构。源文件和目标文件都具有一定的结构，以便匹配读取他们程序的格式。

### 文件系统内部文件结构

磁盘系统通常具有明确的块大小，这是扇区大小决定的。所有的磁盘I/O按块为单位执行，而所有的块的大小相同。一般为512字节。

要对一个文件的位置的偏移进行定位。

## 访问方法

我们在访问文件信息的时候，一般根据文件的结构来进行访问。

### 顺序访问

最简单的访问方式是顺序访问，文件信息按顺序加以处理，比如编辑器。

有时需要在中间加入或者访问中间的，需要将当前的文件指针前移或者后移。

### 直接访问：

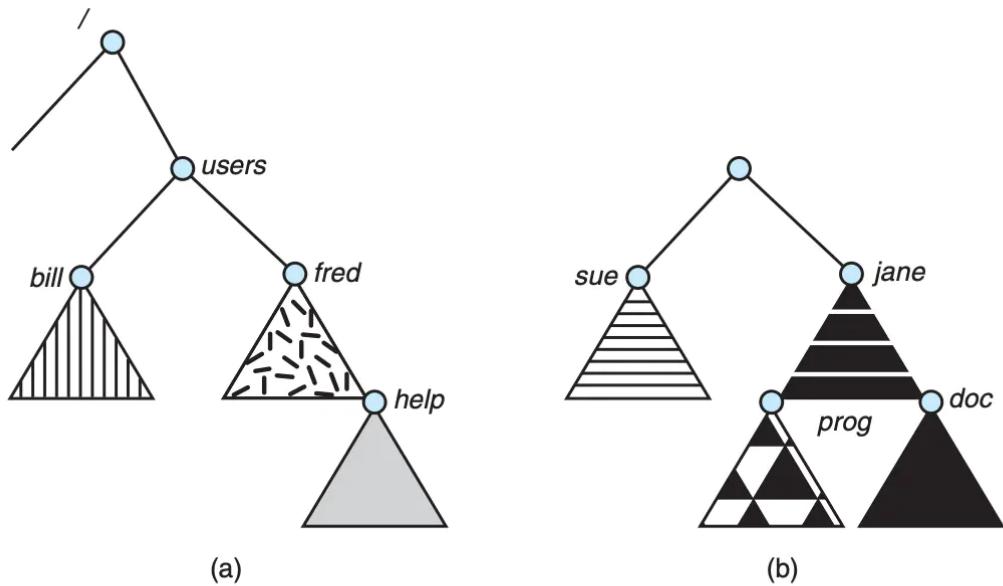
一般的文件由**固定长度的逻辑记录**组成。以按任意顺序的快速读取和写入记录。数据库一般就是这样的模式。

作为一个例子：对于一个航班订票系统，将特定航班的所有信息存储在由航班编号标识的块中。我们访问的时候直接用航班编号定位到数据块。而不是后移文件指针进行访问。

## 文件系统的安装

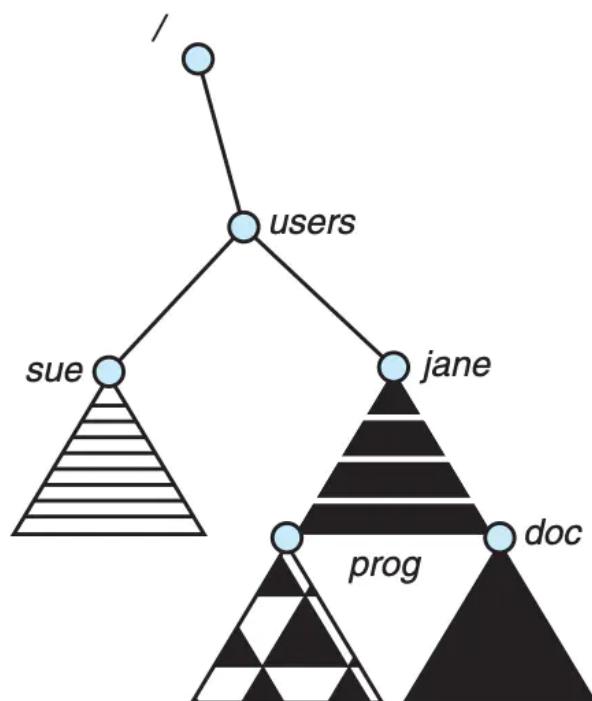
正如文件在使用前必须要打开一样，在使用文件系统之前必须先安装（mount）。一个新的文件系统，要被操作系统识别，必须先进行安装。

为了说明文件系统的安装，如下图，其中三角形表示所感兴趣的目录子树，左边a图是一个现有的文件系统，b图是一个未安装的位于/device/dsk上的文件系统。这时，只有现有文件系统上的文件可以访问。



**Figure 11.14** File system. (a) Existing system. (b) Unmounted volume.

将/device/dsk上的卷安装到/users后的文件系统后，就可以进行访问了。



**Figure 11.15** Mount point.

当操作系统支持多个用户时，多个用户的文件之间应该支持共享，和文件保护。这就是文件的权限问题。

一般用户都采用文件的所有者，所属组的概念。

### 远程文件系统

随着网络技术的发展，远程计算机之间进行通信很常见，所以在不同的计算机之间进行文件共享称为可能。

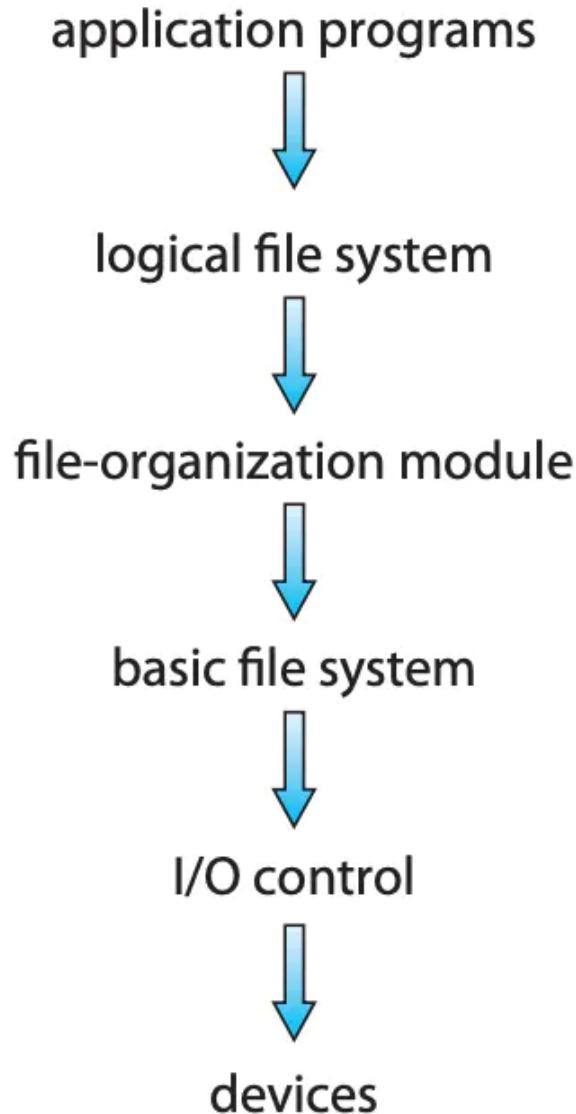
一般文件有三种方法：

- 通过ftp等程序在机器之间手动传输文件。
- 通过**分布式文件系统（DFS）**，远程目录在本地可以直接访问。
- www 万维网。

除此之外，云计算也慢慢变为一种方式。

## 文件系统结构

文件系统本身通过由许多不同层组成，每层的设计利用更低层的功能，创造新的功能。



**Figure 12.1** Layered file system.

I/O控制层(I/O control): 包括设备驱动和中断处理程序，以在主内存和磁盘系统之间进行信息交流。

基本文件系统(basic file system): 本层需要发送通用命令给设备驱动层，并且还要管理内存缓冲区和保存各种文件系统，目录和数据块的缓存。

文件组织模块 (file-organization module) : 知道文件及其逻辑块以及物理块。由于知道所用的文件系统的类型和文件位置，文件组织可以将逻辑块地址转换成物理地址以供基本文件系统传输。

逻辑文件系统 (logical file system) : 管理元数据信息。元数据包括文件系统的所有结构，而不包括实际数据，逻辑文件系统管理目录结构，以便根据给定的文件名称为文件组织模块提供所需的信息。



## 文件系统实现

### 概述

稳健性系统的实现需要采用多个磁盘和内存的结构。虽然这些结构因操作系统和文件系统而异，但是还是有些通用原则的。

在磁盘上，文件系统包括以下信息：

- **引导控制块**: 指示如何启动存储在哪里的操作系统，如果磁盘不包含操作系统，那么这部分为空，一般为启动分区的第一个扇区。Unix上称为**引导块**，Windows系统称为**分区引导扇区**
- **卷控制块**: 包括卷(分区)的详细信息，如分区的数量和块的大小，空闲块的数量和指针，空闲的文件控制块(FCB)和FCB指针。Unix称为**超级块**，Windows称为主控文件表。
- 文件系统的目录结构和用于组织文件。Unix中包含文件名和相关的inode的号码。
- 每个文件的FCB包括该文件的许多信息。它有一个唯一的标识号。以便与目录条目相关联。

在内存中的信息：

- 内存的**安装表**: 包含每个安装卷的信息。
- 内存中的目录结构的缓存含有最近访问的目录的信息。
- 整个系统的打开文件表。
- 每个进程的打开文件表。
- 缓冲区保存的文件系统的块。

superblock 卷 NTFS/EXT  
dentry 目录项 directory  
inode 文件属性 metadata  
file 文件

### **文件的创建过程：**

为了创建新的文件，应用程序调用逻辑文件系统。逻辑文件系统直到目录结构的格式。它会分配一个FCB。然后将相应的目录读到内存，然后使用新的文件名和FCB进行更新，将它写会磁盘。

### **文件的读写过程：**

- 打开文件的实际操作：

一旦文件被创建，他就可以进行I/O操作。不过他首先应该被打开，系统调用open()将文件名传递到逻辑文件系统。系统首先搜索整个系统的打开文件表，以便确定这个文件是否被其他进程使用。如果是，则在进程的打开文件表中创建一个条目，并让他指向现有的系统的打开文件表，进程的打开文件表的计数加1。

如果没有，则根据给定的文件名来搜索目录结构。（部分的目录结构通常缓存在内存中，以加快目录操作。）找到文件后，它的FCB会被复制到内存的整个系统的开发文件表中。该表不但存储FCB，还跟踪打开文件进程的数量。接下来，在整个进程的打开文件表中，会创建一个条目，指向整个系统打开文件条目的一个指针，以及其他域。这些域可能包含文件系统的当前位置的指针和打开文件的访问模式。随后，进程的打开文件表的计数加1。

- open()文件最终返回一个文件指针。以后所有的操作都用这个指针，UNIX中称为**文件描述符**，Windows称为**文件句柄**。
- 进程关闭文件时，它的单个进程的条目会被删除，并且系统的打开文件表中这个文件的打开计数会被减1。当文件计数为0时，他把文件的元数据写入此案。然后系统的打开文件表中删除此文件的条目。

## **分区与安装**

磁盘布局可以有多种，具体取决于操作系统。一个磁盘可以分成多个区，或者一个卷跨越多个磁盘的多个分区。后者就是RAID的形式。

分区可以使原始的，什么文件系统都没有，什么都没有的时候就是**原始磁盘**。UNIX的交换分区就是一个原始磁盘，有些数据库也使用原始磁盘，并且格式数据，以满足他们的要求。

**引导的信息**可以存储在各自分区中，而且可以有自己的格式，不一定和要加载的文件系统的保存结构一样。因此在这个阶段，还没有加载操作系统，更不会有文件系统。因此引导信息通常是一系列的块，可作为映像加载到内存。映像一般的执行从预先定义的位置执行。这个引导的信息里面应该包含应该以怎样的结构来解析接下来的文件系统信息，从而找到内核，并进行执行。

许多系统可以有多个引导程序，因为可以安装双系统。

**根分区**，包括操作系统内核其他文件系统，在启动时安装，其他卷可以在引导时自动安装以后安装。

## **虚拟文件系统**

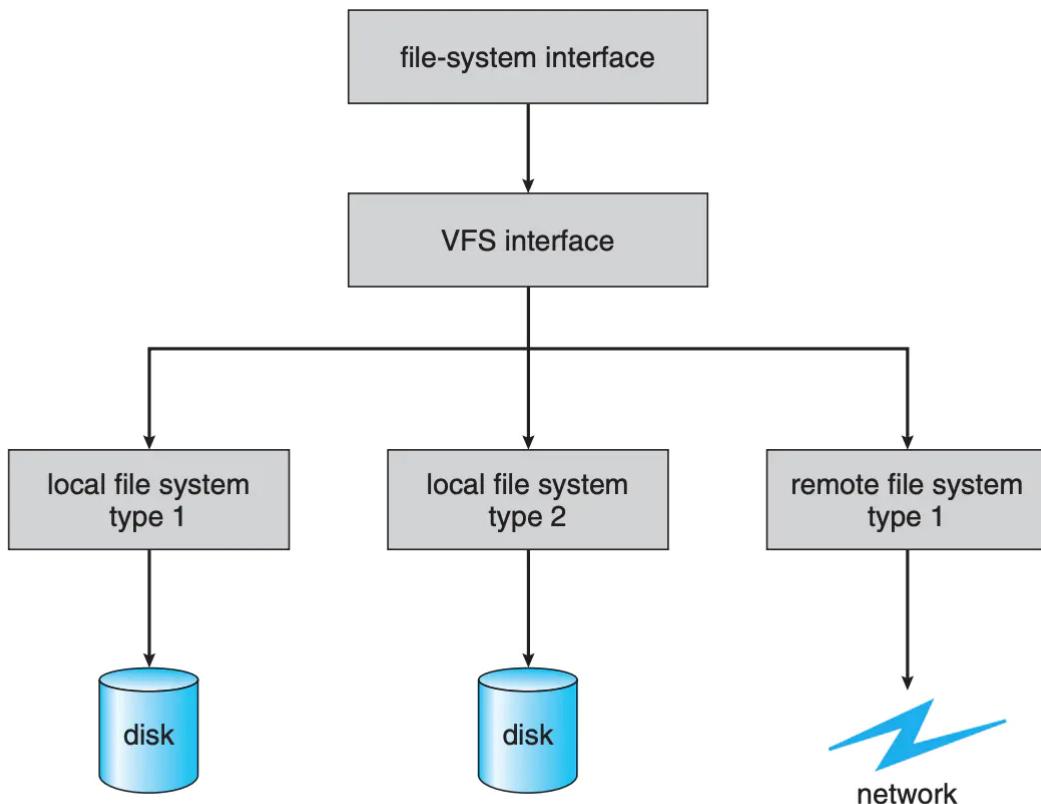
现代操作系统必须同时支持多中类型的文件系统，但是操作系统如何才能实现将多个类型的文件系统集成到目录结构中呢？用户如何在多个文件系统中无缝的迁移？

在大多数的操作系统中，都支持了在不同文件类型可通过相同的类型来实现。它也包括网络文件系统类型。

数据结构和程序用于隔离基本系统调用的功能和实现细节。因此，文件系统的实现由三个主要层组成。

- 第一次为文件系统接口层。基于系统调用open, read, write 和close。
- 虚拟文件系统层（VFS）。他提供两个功能：

1. 通过定义一个清晰的VFS接口，将文件系统的通用操作和实现分开。VFS提供多个通用接口，允许透明的访问本地安装的不同类型的文件系统。
2. 它提供了一种机制，唯一表示网络上的文件。VFS 基于称为**虚拟节点**的文件表示结构，虚拟节点包含一个数字指示符，唯一表示网络上的一个文件。UNIX中采用inode。内核为每个活动节点保存一个vnode结构。



**Figure 12.4** Schematic view of a virtual file system.

VFS根据文件系统的类型调用特定的操作以便处理本地请求。通过NFS协议程序来处理远程请求。

## 目录的实现

数据结构的选择

1. 线性列表：不合适。频繁的搜索称为问题。
2. 哈希表：哈希表根据文件名获取一个值，并返回线性列表内的一个元素指针。减少了目录搜索的时间。当发生冲突的时候可以使用链式哈希结构。

## 分配方法

创建文件时，如何在空闲的磁盘上为一个新文件分配空间？

一般由是那种方法：连续，链接和索引。

### 连续分配：

连续分配表示每个文件在磁盘上占有一组连续的块。磁盘地址为一组线性结构。

问题：在给每个分配的时候，容易在中间产生碎片。还有无法估算一个文件有多大，估算的很小，没有办法扩展。

### 链接分配：

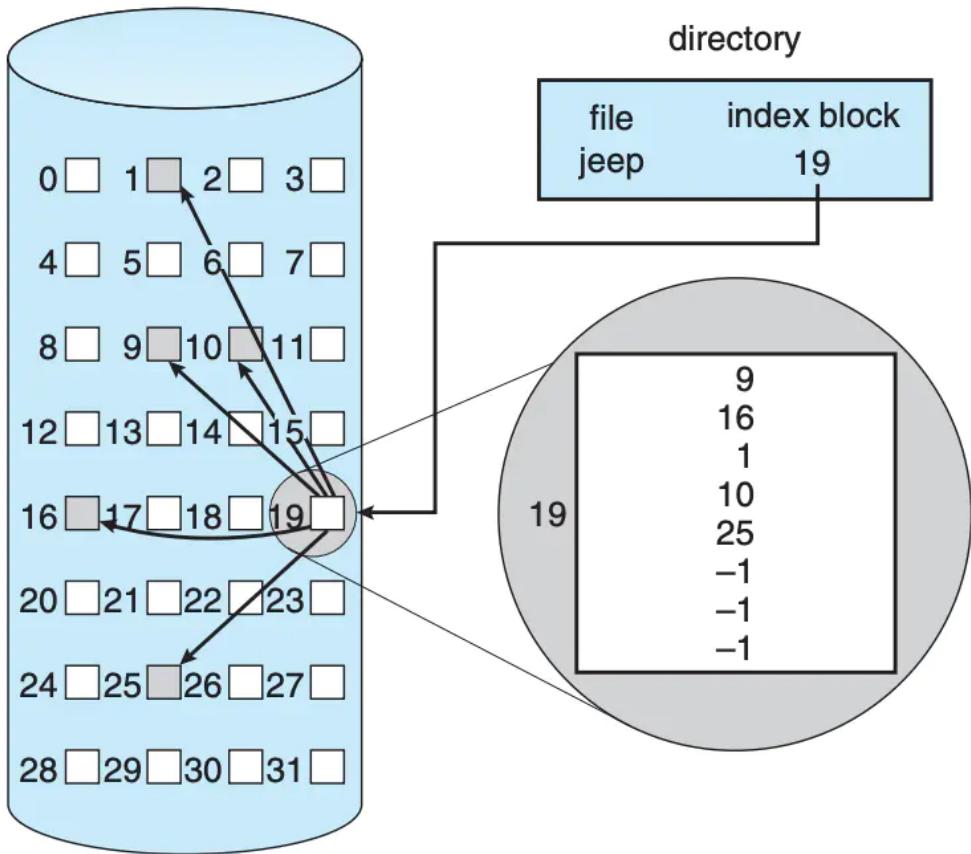
解决了连续分配的所有问题，每个文件都是磁盘块的链表。文件的磁盘块可能分布在磁盘的任何地方，通过链表来进行连接。

问题：只能顺序的访问文件，要找到第*i*个块，必须从头开始。还有每次磁盘块的最后4个字节需要保存下一个磁盘的地址。而且如果链断了，就全部完了。

### 索引分配：

索引分配通过将所有指针放在一起，称为索引块。

每个文件都有自己的索引块。这是一个磁盘块地址的数组。索引块的第一个条目指向文件的第*i*块。目录包含索引块的地址。



**Figure 12.8** Indexed allocation of disk space.

## 空闲空间管理

由于磁盘空间有限，如果可能，需要将删除文件的空间重新用于新文件。为了跟踪空闲磁盘空间。系统需要维护一个**空闲空间列表**。空闲空间列表记录了所有空闲磁盘空间。

### 位图：

空闲空间列表按位图(bitmap)来实现。每个块用一个位来表示。块是空闲的，位为1；如果块是分配的，位为0。

### 链表：

空闲空间将所有的空闲磁盘块用链表连接起来。

### 组

空闲列表方法的一个改进是，在第一个空闲块中存储n个空闲块的地址。

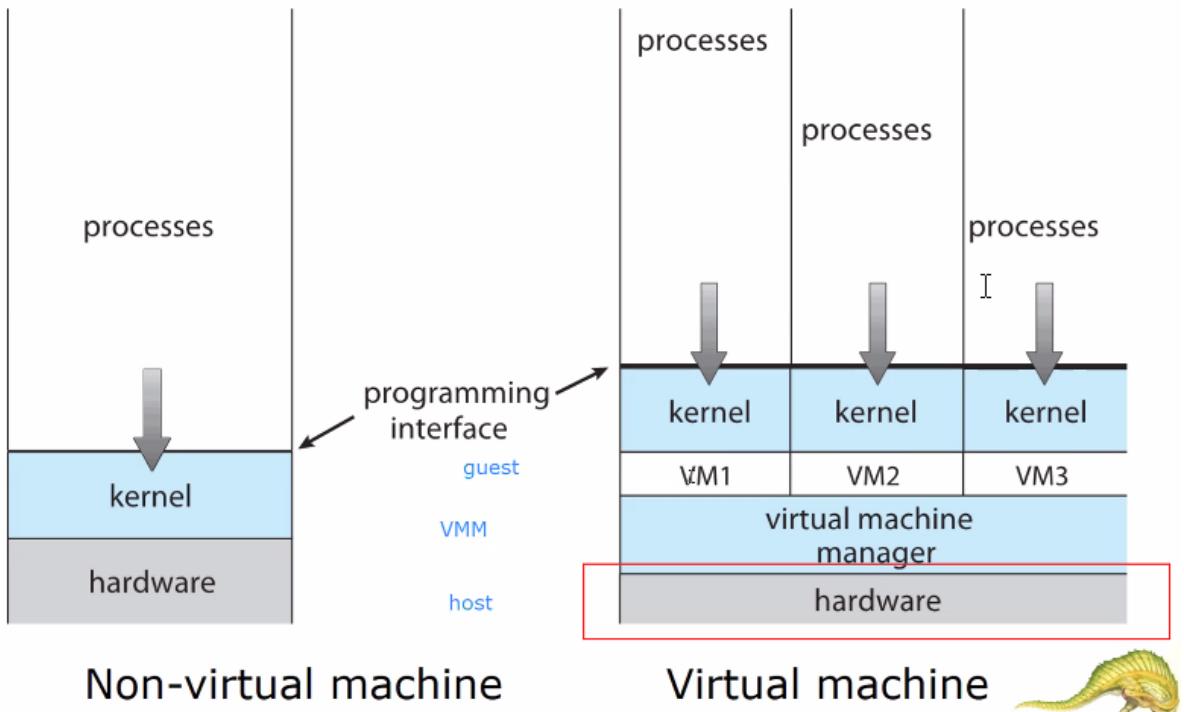
这些块的前n-1个确实为空。

最后一块包含另外n个空闲块的地址。

# Ch 18 虚拟机 virtual machines

## 组成

- host
- VMM
- guest



## 虚拟机层次

- **Type 0 hypervisors** - Hardware-based solutions that provide support for virtual machine creation and management via firmware
  - ▶ IBM LPARs and Oracle LDOMs are examples
- **Type 1 hypervisors** - Operating-system-like software built to provide virtualization
  - ▶ Including VMware ESX, Joyent SmartOS, and Citrix XenServer
- **Type 1 hypervisors** – Also includes general-purpose operating systems that provide standard functions as well as VMM functions
  - ▶ Including Microsoft Windows Server with HyperV and RedHat Linux with KVM
- **Type 2 hypervisors** - Applications that run on standard operating systems but provide VMM features to guest operating systems
  - ▶ Including VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox