CS130 操作系统I

Chapter 0 课程介绍

- 给分标准：
  - 签到(**5%**) 随机时间随机形式
  - 期中考试(**10%**) 4.23课上100分钟，考到 Chapter 7 死锁之后，双面A4手写Cheatsheet，允许计算器；单选、简答、可能提供代码段的问题
  - 期末考试(**15%**) 6.16课上100分钟，从 Chapter 8 内存管理开始，考到最后一章，两张A4手写Cheatsheet，允许计算器；和期中格式相似
  - 个人作业(**10%**) 期中期末之前各一次，问答题，不可迟交
  - Project(**10%**+**10%**+**20%**+**20%**) 双人；ddl分别在第5,9,13,16周日23:59；ddl前一周会有讲解，ddl后周一或周二晚 CodeCheck
- 使用*Piazza*和*Gradescope*

Chapter 1 操作系统架构概述

- 操作系统是一种软件，负责管理系统资源
- 计算机系统组织中I/O与CPU协同的核心要点：
  - I/O设备和CPU并发执行
  - 每个设备控制器负责一类设备，带本地缓冲，由对应驱动管理
  - CPU在内存与设备控制器本地缓冲间传输数据
  - I/O数据先到控制器本地缓冲
  - 控制器完成操作后，通过中断通知CPU

系统调用 System Call

- 系统调用提供了程序和操作系统之间联系的接口
- 系统调用大部分情况下是汇编语言编写的，但是现在可以直接用 C/C++ 来编写了
- 运行的程序和操作系统之间传递参数的方式
  - 直接在寄存器中传参
  - 将参数储存在内存的一个表格中，表格的内存地址作为参数传入寄存器
  - 程序将参数压入一个栈里，操作系统弹栈获取

硬件保护 Hardware Protection

双模式操作 Dual-Mode Operation：

- 分成**内核态 Kernel/monitor Mode** 和**用户态 User Mode**，前者有更高的权限，后者只有限制的权限
- 有一个Mode bit，在硬件上表示了当前的模式：kernel (0) 或 user (1)
- User Mode 可以通过 fault（trap中的异常） 或 Interrupt 来切换为 Kernel Mode
- Kernel Mode 可以通过 set user mode 来切换为 User Mode

- Privileged instructions 只能在 Kernel Mode 下运行

输入输出保护 I/O Protection

- 所有的输入输出操作都是privileged操作，必须在kernel mode下才能操作
- 当用户程序需要输入输出时，先向操作系统发起 system call 来 trap to monitor，然后操作系统在Kernel Mode下作出输入输出操作后，返回给用户程序

内存保护 Memory Protection

- 使用两个寄存器来存储一个程序可以访问的内存地址范围：
- Base register 存储了用户能访问的最小物理内存地址
- Limit register 存储了用户能访问的物理内存大小
- 即用户能访问 `[Base, Base + Limit)` 的物理内存地址范围
- 这两个寄存器的数值编辑都是Privileged instructions

CPU保护 CPU Protection

- 同一个CPU-cycle内，只能做一个进程的处理
- 设定一个Timer，每个时间刻减1，到达0时触发中断
- Timer可以用来计算当前的时间
- Load-timer是一个Privileged instruction

Chapter 2 操作系统服务与结构

操作系统服务

操作系统提供了用户程序运行的环境

- 提供了**用户界面 User interface (UI)**
  - 命令行界面 Command-Line (CLI)
  - 图形界面 Graphics User Interface (GUI)
  - 触屏 touch-screen
  - Batch
- 程序执行
  - 加载用户程序进入内存
  - 运行程序
  - 终止程序执行
  - 处理程序错误
- 输入输出操作
  - 运行的程序要求输入输出设备
  - 运行的程序要求访问文件系统
- 文件系统控制
  - 用户程序增删改查文件和目录，需要操作系统管理权限
- 进程通信

    - 通过网络进行通信
    - 在同一台计算机上发生通信（可能是直接发信息或共享内存地址）
- 错误侦查与处理
    - 硬件错误（硬件损坏/突然断电）
    - 软件错误
    - Debug功能
- 高效、安全的资源管理
    - CPU-cycle
    - 内存
    - 文件存储
    - 输入输出设备
- 日志管理
    - 储存用户使用各资源的情况
- 保护与安全
    - **保护**保证所有访问系统资源的行为都被控制
    - **安全**保证外来访问都经过用户的授权，阻止越界的访问

系统调用 System Call

- 系统调用是程序和操作系统交互的接口，主要由C/C++编写
- 系统调用主要由程序借由更高级的Application Programming Interface(API)来调用，而非直接调用，因为这样能有效提高程序的兼容性

系统调用的分类

- 进程控制 Process control
    - create process(调用API而非直接system call), 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

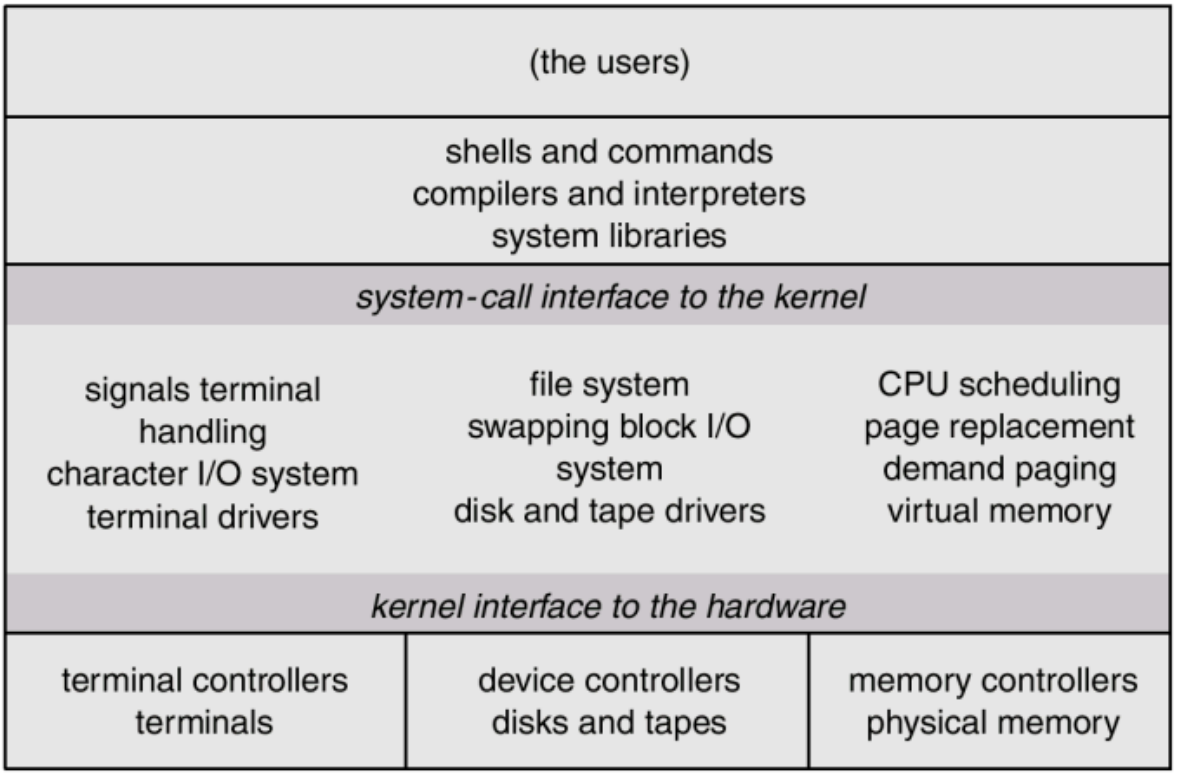
- 设备管理 Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- 信息维护 Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- 交互 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

操作系统结构

分层结构 Layer Structure

- MS-DOS 是一种不完美的分层结构，层间没有严格的屏障，导致用户程序可以直接操作硬件
  - 应用程序 application program
- 常驻系统程序 resident system program
- 设备驱动 MS-DOS device drivers
- ROM BIOS 设备驱动程序 ROM BIOS device drivers
- 分层结构的优缺点
  - 由于分层，从内向外的Debug相对容易
  - 很难定义各个功能的分层关系
  - 在成熟的操作系统中并不常见，但在次级结构中有可能采用分层结构
  - （实际上在TCP-IP协议等网络通信领域，分层结构较为实用）

UNIX 系统结构



- 系统程序 System programs
- 内核 The kernel：所有在系统调用界面之下、物理硬件之上的内容
  - 文件系统 File system
  - CPU调度 CPU scheduling
  - 内存管理 Memory management
  - ...

交互模型

- 两个进程之间可能发生信息交互：
  - 消息传递 Message Passing：进程A将信息发送给内核，然后内核将信息发送给进程B
  - 共享内存 Shared Memory：进程A将信息存入共用内存，进程B从中读取信息

微内核结构

- 把所有非必要的系统服务（文件系统、驱动等）都从内核中转移到用户空间中
- 内核只保留其核心功能：进程间通信(IPC)，内存管理，调度
- 更多使用Message passing来在用户模块之间传递信息，因为这样内核可以控制信息传递过程，更安全
- 优缺点：
  - 内核更小，可移植性强，可以移植到不同的硬件架构上
  - 内核模式下运行的代码更少，因此更可靠、更安全
  - 设计困难，内核与用户功能分野较难确定
  - 内核与用户空间的信息交流过分频繁

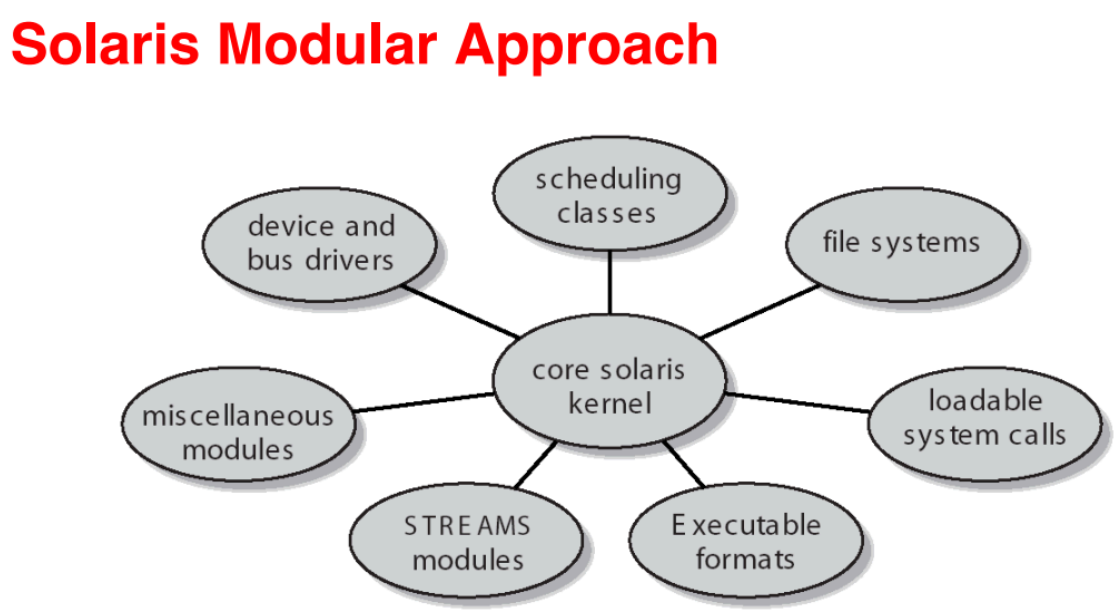
Mac OS X架构

- 内核采用 BSD Unix + mach micro-kernel 的混合架构
- 两种内核架构分别提供了一种 system call 接口

内核模块 Kernel Module

- 大多数现代操作系统都支持添加自定义内核模块
- 内核模块核心组分是独立的
- 内核模块的通信是基于已有的接口的
- 内核模块随时启动/停止，对性能影响较小
- 内核模块的概念类似于分层结构的层，但是灵活性更高

Solaris 模块化方法



虚拟机 Virtual Machine

- 把硬件和操作系统都看作硬件
- 同一台电脑上可以同时跑多个操作系统
- 可以跑多个进程，每个进程的资源管理都是独立的
- 内层的操作系统不能感知到自己在虚拟机上跑
- 优缺点
    - 独立性：虚拟机之间不会相互干扰，但是也不能共享资源
    - 开发与测试内核更加容易
    - 虚拟机的实现较为复杂
    - 虚拟化技术有性能开销，影响整体性能

反虚拟化 Xen Architecture: Paravirtualization

- 内层操作系统可以和外层宿主操作系统交互

系统启动 System Boot

- 在固件(Firmware)上有一小段启动引导程序：**bootstrap loader**
- 这段引导程序可以定位内核，并把内核加载进内存，并启动内核
- 有些电脑存在多级loader

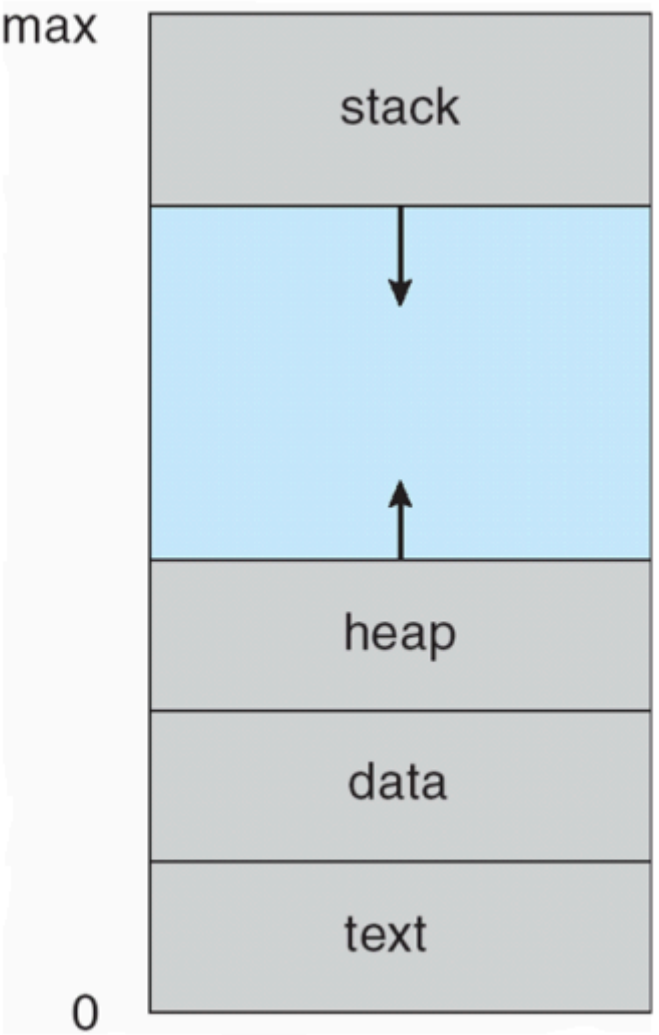
Chapter 3 进程 Process

中断和陷阱 Interrupts and traps

- 中断 Interrupt：异步操作，可以在处理器执行指令的任何时候发生，不受当前程序流程的控制
- 陷阱 traps：同步操作，由当前正在执行的指令直接引发

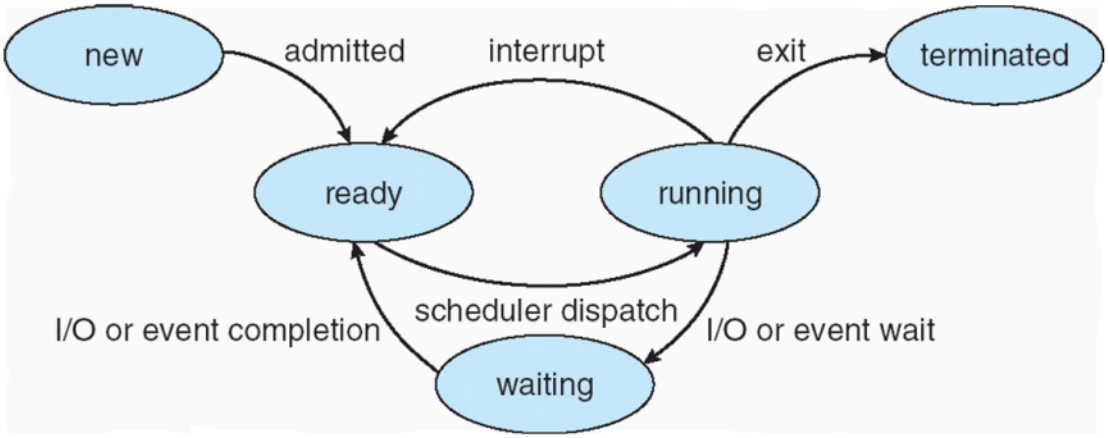
进程

- 一个正在执行的程序，可以看作一个程序的实例化
- 包含一个程序计数器program counter，一个栈stack和一个数据段data section



- heap和stack是动态的，但是data和text是静态区域，执行时大小基本不变
- 进程有多种状态，在运行时改变：
  - new：进程刚被创建
  - running：进程正在运行

    - waiting：进程正在等待某些事件发生
    - ready：进程正在等待被分配给一个处理器
    - terminated：进程终止执行



- 进程的多种状态名称在不同操作系统内不尽相同，且存在某些操作系统有更多的进程状态
- 操作系统需要管理进程
    - 支持进程创建、进程删除等操作
    - 支持进程间交互(interprocess communication, IPC)
    - 支持给进程分配资源
    - 支持多进程交错执行

进程控制块 Process Control Block, PCB

- 在内存中的内核空间 Kernel Space 中
- 储存进程运行所需全部信息的数据结构，包含以下内容：
    - 进程状态 Process state
    - 程序计数器 Program counter
    - CPU寄存器 CPU register
    - CPU调度信息 CPU scheduling information
    - 内存管理信息 Memory-management information
    - 资源记账信息 Accounting information
    - 输入输出状态 I/O status information

上下文切换 Context Switch

- 当CPU切换进入另一个进程时，系统需要储存中断进程的状态，然后载入新进程的状态
    - 复制所有使用的寄存器信息储存到PCB中
    - 需要至少一个临时寄存器(scratch register)：这个寄存器指向进程控制块中用于保存寄存器值的特定内存区域，用于转存时提示目标PCB内存地址
    - 重新恢复新进程的状态，把新进程的PCB寄存器信息重新导入寄存器内
- 一次上下文切换的用时是由硬件决定的（如寄存器数量、寄存器大小等）

进程创建 Process Creation

- 父进程可以创建子进程，从而形成一个进程树

- 每个进程都有一个不同的pid(process identifier)
- 可以有不同的资源分享机制
  - 父子进程共享所有资源
  - 父子进程共享部分资源
  - 父子进程不共享资源
- 可以有不同的执行机制
  - 父子进程并行执行（同步执行）
  - 父进程保持等待，直到子进程结束（异步执行）
- 可以有不同的地址空间分配方式
  - 子进程复制了父进程的所有空间资源 (fork system call：创建新的进程，并返回子进程的pid)
  - 子进程在创建后加载一个新的程序 (exec system call：在fork后执行，用新程序覆盖子进程，然后子进程拥有对新程序的完全控制权；除非新程序发生错误，则控制权交还给父进程)

```
int main()
{
    pid_t pid;
    pid = fork();   // 系统调用，产生一个新的子进程，父子进程都执行到这一行：子进程中返回0，父进程中返回子进程pid（正整数）
    if (pid < 0)    // 发生错误
    {
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0)  // 子进程
    {
        execlp("/bin/ls", "ls", NULL)
        // 操作系统从 /bin/ls 加载 ls 程序，替换当前进程的所有内存内容
        // 设置参数：argv[0] = "ls" ，并开始执行 ls 的主函数
        // 当 ls 运行结束，整个进程终止
    }
    else    // 父进程
    {
        wait(NULL); // 父进程等待，直到子进程以任意状态结束
        printf("Child Complete");
        exit(0);
    }
}
```

进程终止 Process Termination

- 当一个进程运行完最后一条命令，会调用 `exit()` 来让操作系统删除这个进程，此时进程的所有资源将被操作系统回收
- 父进程需要通过 `wait(&state)` 来等待子进程结束并清理子进程PCB，然后将退出状态值存入 `int state` 中；`wait(NULL)` 或 `wait(0)` 可以等待直到子进程以任意状态结束
- 父进程有可能要求终止子进程的执行(`abort()`)，有如子进程资源超限、子进程不再需要、父进程终止等原因

进程间交流 Interprocess Communication (IPC)

- 进程可能会有资源共享或相互影响的情况，这被称为进程间交流(Interprocess Communication, IPC)

进程间交流的原因：

- 信息共享
- 多进程计算加速
- 模块化场景下运行
- 允许数据传递带来方便

进程间交流的方式：

共享内存 Shared memory

- 比消息传递更快
- 可以用于传输大量数据

消息传递 Message passing

- 比共享内存更容易实现
- 一般用于传输少量数据
- 先建立一个交互链接(Communication link)，然后进程A发送消息给内核，内核再把消息发送给进程B
- 支持固定或可变长度的消息
- 直接交流：
  - 进程必须显式地定向收发消息：`send(A, message); receive(Q, message)`
  - 交互链接是自动建立的，固定链接两个进程，且每对进程之间只有一个链接，一般为双向链接
- 间接交流：
  - 消息从邮箱(mailbox/port)收发，仅当两个进程共享一个邮箱时才能交流
  - 原语(Primitive)有 `send(Mailbox A, message)` `receive(Mailbox A, message)`
  - 交互链接可能链接多个进程，每对进程之间可能有多个链接，链接可能是单向或双向
  - 一次仅允许一个进程执行`receive`操作
- 同步 Synchrinization
  - Blocking：一种同步方式，发送者/接收者进程会被阻塞，直到收到消息/消息可用
  - Non-blocking：一种异步方式，发送者一旦发送结束，就可以直接向下执行；接收者不管是否接收到，都继续执行
- 缓冲 Buffering
  - 零容量：发送方必须等待接收方准备好，才能发送信息
  - 有界容量：到达上界后，发送方必须等待消息列表空出来
  - 无界容量：发送方随时发送信息

生产者-消费者问题 Producer-Consumer Problem

- 一种典型的进程交互例子

- 对于临界区的数据，先修改，后标志可以有效防止死锁
- 共享数据：

```
#define BUFFER_SIZE 10
typedef struct
{
    ...
} item;
item buffer[BUFFER_SIZE];   // 建立一个循环队列
int in = 0;      // 生产者下一个放入的位置
int out = 0;     // 消费者下一个取出的位置
```

- 生产者 Producer 进程：

```
while (true)
{
    item next_produced;      // 生产一个 item
    while (((in + 1) % BUFFER_SIZE) == out);     // 如果满了（BUFFER_SIZE - 1
个，留一个空位），就忙等待
    buffer[in] = next_produced;                  // 填入下一个空位
    // 如果在此处发生中断，则消费者程序会等待到时间片用完，然后归还操作权，不会死锁
    in = (in + 1) % BUFFER_SIZE;                 // 指针循环向后移动
}
```

- 消费者 Consumer 进程：

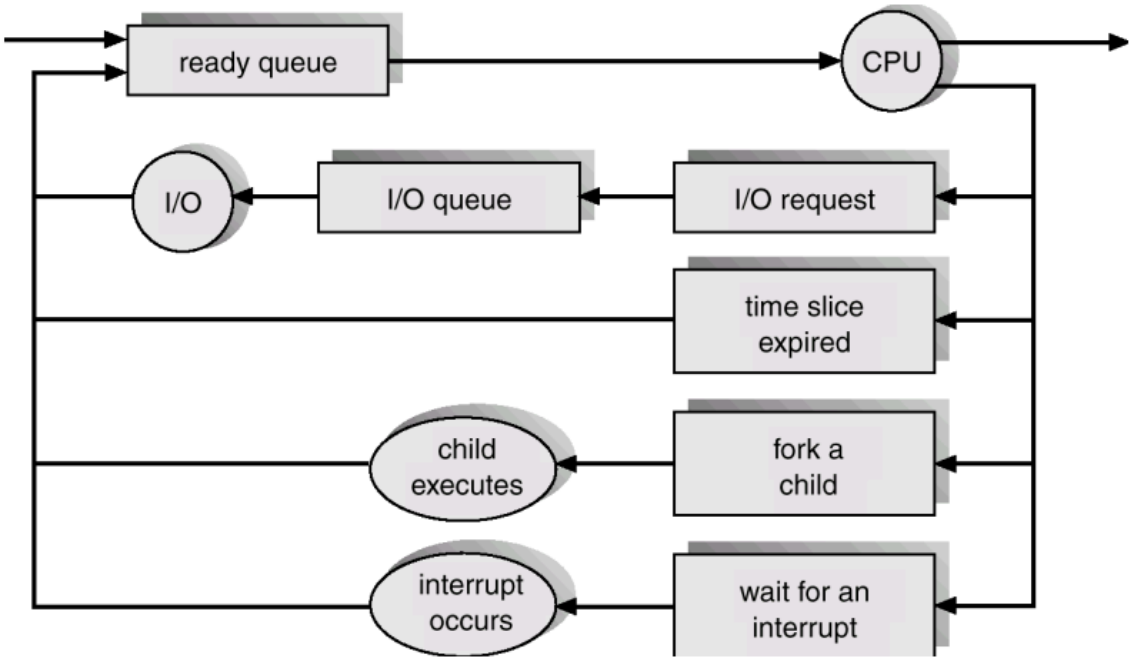
```
while (true)
{
    while (in == out);       // 如果空着，就忙等待
    item = buffer[out];      // 拿一个出来
    out = (out + 1) % BUFFER_SIZE;  // 指针循环向后移动
    return item;
}
```

进程调度 Scheduler

- 长期调度器 Long-term scheduler / job scheduler
  - 解决哪个进程需要进入就绪队列(ready queue)
  - 调用不频繁（秒~分钟）
- 短期调度器 Short-term scheduler / CPU scheduler
  - 解决哪个进程下一个执行
  - 调用极其频繁（毫秒）
- 作业队列 Job queue
  - 一般是链表
  - 所有进程的集合
- 就绪队列 Ready queue
  - 一般是链表
  - 位于主存（内存）中、已准备好执行且等待CPU分配的进程集合

- 设备列表 Device queue
  - 一般是链表
  - 等待输入输出设备的进程集合

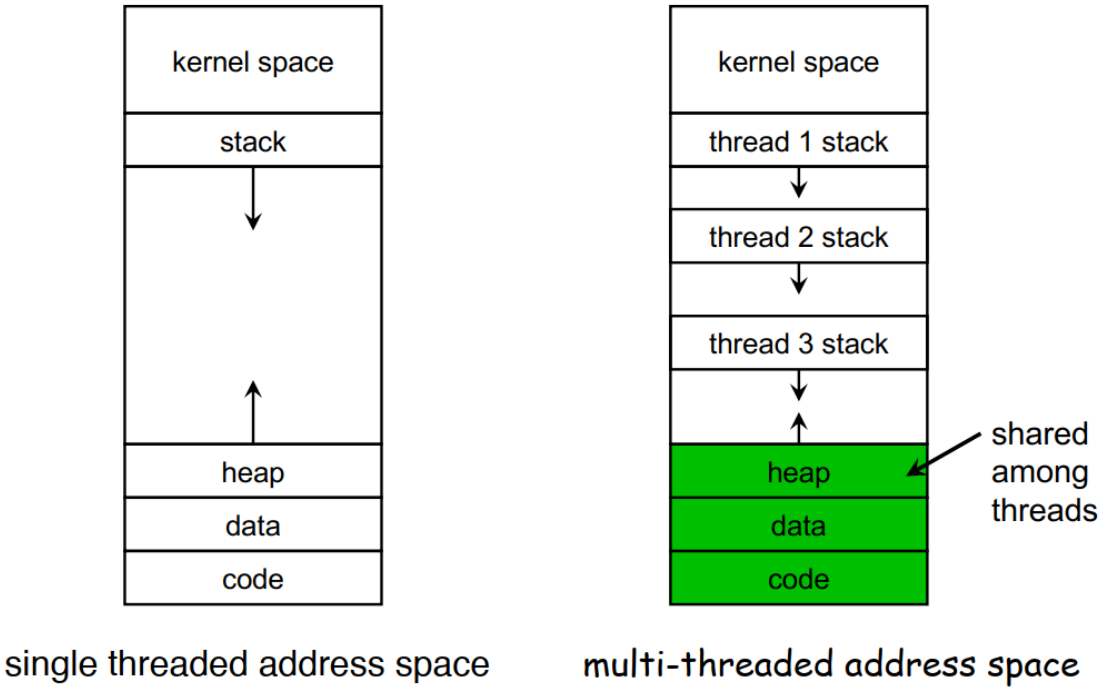
进程调度的表示



Chapter 4 线程 Thread

线程 Thread 与 进程 Process

- 1个进程可以有多个线程并行执行
- 进程和线程都有控制块，分别是PCB(Process Control Block)和TCB(Thread Control Block)
- 进程和线程都有上下文选择，而线程的上下文选择开销更小
- 多线程进程中，线程独立拥有寄存器register、栈空间stack、执行状态(running, ready, blocked, etc)，而同一个进程内的线程之间共享代码code、数据data和文件file（堆空间heap）、网络套接字、用户id、I/O资源、打开的文件、IPC设备等
- 线程是CPU利用率的最小单位
- 在单线程系统中，进程是资源拥有者、调度和执行的最小单元
- 在多线程系统中，线程是执行和调度的最小单元，而进程是资源的容器、线程的集合



single threaded address space　　multi-threaded address space

多线程（而非多进程）的优势

- 线程的创建/终止比进程更快
- 同进程的线程之间上下文切换更快
- 同进程的线程之间的（从共享内存的）交流更加高效

使用线程的场景

- 多处理器的机器
- 需要处理慢速设备
- 有后台操作
- 有图形化界面、窗口系统
- 服务器应用处理多个请求

不使用线程的场景

- 不同的执行单元需要使用不同的用户id或授权，如ssh服务器

线程的上下文切换 Thread Context Switch

- 上下文切换发生在：异步事件发生（如时间配额到期，硬件中断等），同步调用（如线程执行阻塞的系统调用，主动让出执行权等）
- 需要保存当前执行线程的状态：
  - 复制所有用户寄存器到线程控制块(TCB)中
  - 保存控制信息（PC，SP）
- 需要加载下一个运行的线程：
  - 把寄存器的值复制到处理寄存器processor registers中

POSIX 线程（Pthreads）API

- 线程创建与终止

```
pthread_create(&tid, NULL, start_func, arg);
pthread_exit(status);
```

- 线程的连接：阻塞当前线程，直到目标线程 `tid` 终止

```
pthread_join(tid, &status);
```

- 互斥锁

```
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
```

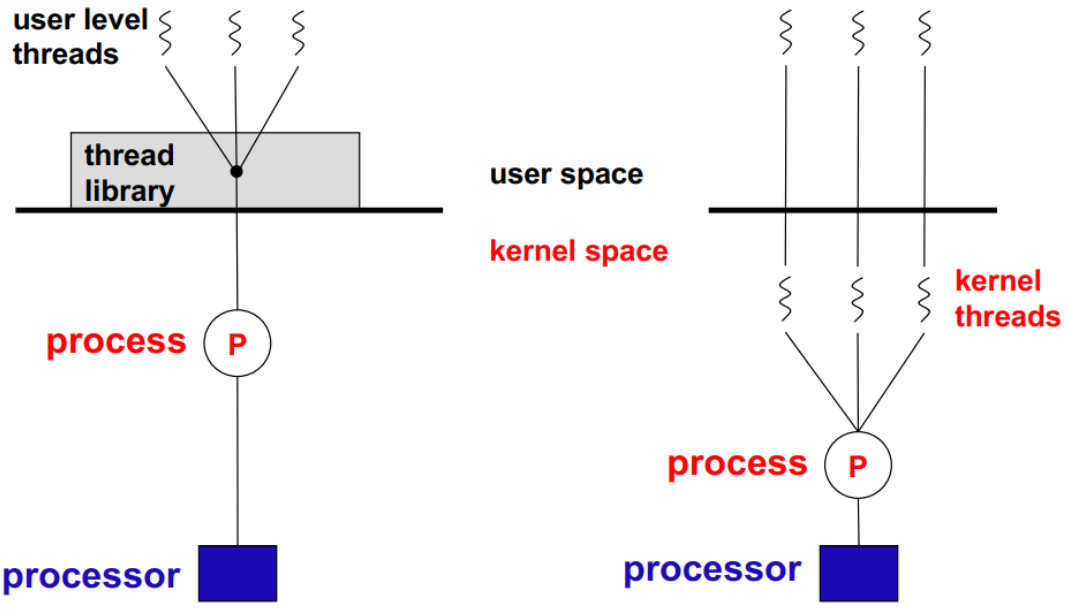
- 条件变量

```
pthread_cond_wait(&cond, &lock);     // 等待
pthread_mutex_signal(&cond);         // 唤醒至少1个
pthread_mutex_broadcast(&cond);      // 唤醒所有
```

线程的实现

用户线程与系统线程

- 用户线程 User level thread
  - 由用户空间线程库 User space thread library 管理
  - 内核不知道线程的活动（从内核的角度看，是一种单线程进程）
  - 可以是抢占式(可以在另一个线程执行中抢先执行)/非抢占式的
- 内核线程 Kernel level thread
  - 由内核管理
  - 内核支持多个执行的内核线程上下文



- 用户线程的优缺点
  - 更低消耗的线程操作与上下文切换（无需内核参与）

    - 调度更灵活，由thread library操控而无需内核参与
    - 无需内核参与，因此可移植性更强
    - 一旦有一个用户线程在内核被阻塞，那么整个进程都会被阻塞
    - 无法充分利用不同处理器来同步处理
- 内核线程的优缺点
    - 阻塞一个线程并不会影响其它线程的状态
    - 同进程的线程可以在不同处理器上同时处理
    - 上下文切换更慢
    - 调度完全依赖于内核调度器的实现
- 几乎所有系统都支持内核线程，而用户线程由可链接的库实现
- 线程管理模式的应用
    - （用户线程：内核线程）多对一：Solaris Green Threads, GNU Portable Threads（现在很少使用了，因为这样无法体现多核处理器的优势）
    - 一对一：Windows NT/XP/2000, Linux, Solaris 9 and later，Mac
    - 多对多：带有 ThreadFiber包的Windows NT/2000
    - 两层模型：在多对多的基础上，支持用户线程绑定到内核线程（一对一的形式）；IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier

多线程信号处理 Signal Handling in Multithreading Process

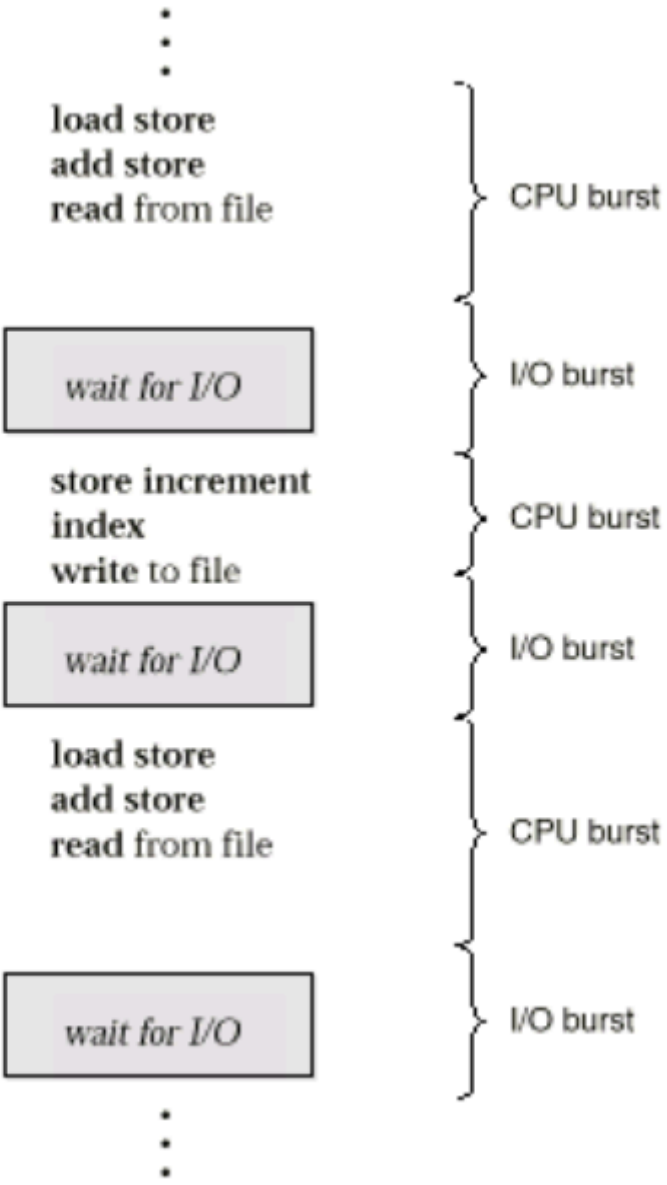
- 信号Signal 是进程级别的概念，信号处理器和整个进程关联
- 信号传递给多线程进程时，只会由其中一个线程处理
- 线程可以选择忽略或屏蔽信号
- 操作系统会在进程控制块PCB里标记：信号应该传递给哪个线程
- 如果信号传入时所有线程都在阻塞，会等待直到有线程被调度器唤醒后，传递给那个线程
- 常见策略：一个线程处理所有信号，其它线程屏蔽所有信号

线程的取消 Thread Cancellation

- 线程在正常结束之前被取消
- 实现机制：
    - 异步取消 Asynchronous Cancellation：立即终止目标线程
    - 延迟取消 Deferred Cancellation：目标线程主动定期检查是否需要被取消；若到达检查点后发现需要被取消，则终止线程

Chapter 5 CPU调度 CPU Scheduling

CPU调度器



- 进程执行的序列是由CPU突发(CPU burst)和输入输出突发(I/O burst)交替排列的
- CPU调度器(CPU Scheduler)可以选择哪个就绪ready状态的进程执行
- CPU调度在以下情况发生：
    - 进程从running切换到waiting状态（非抢占的 nonpreemptive）
    - 进程终止（非抢占的 nonpreemptive）
    - 进程从running切换到ready状态（抢占的 preemptive）
    - 进程从waiting切换到ready状态（抢占的 preemptive）

调度程序 Dispatcher

- 调度程序模块负责将CPU的控制权交给由短期调度程序 short-term scheduler 选中的进程。
- 调度延迟 dispatch latency：需要尽可能小，而减少调度时间
- 现实中多核系统会每个核分别进行调度，而本课程中简化为单核CPU，同时只执行一个线程

调度算法

评价调度的原则

- CPU利用率 CPU utilization：尽可能保持CPU工作(busy)
- 吞吐量 Throughput：单位时间内完成的进程数量
- 周转时间 Turnaround time：执行特定进程所需的时间
- 等待时间 Waiting time：进程在就绪队列里等待的时间
- 响应时间 Response time：从提交请求到第一个响应发生的时间（并非输出完成，而是开始输出）
- 我们希望最大化CPU利用率、最大化吞吐量、最小化周转时间、最小化等待时间、最小化响应时间，但是这些目标很难同时达成，故对于特定的需求，需要设计特定的系统来处理

队列调度 First-Come, First-Served (FCFS) Scheduling

- 进程等待时间很大程度上受到进程运行时间和到达顺序的影响
- 护航效应 Convoy effect: 短进程在长进程的后面执行时，会有较长的等待时间

短进程优先调度 Shortest-Job-First (SJF) Scheduling

- 以进程到下一次CPU突发 CPU burst 的时间长度为关键字，实现小根堆优先队列调度
- 抢占式：也叫最短剩余时间优先调度 Shortest-Remaining-Time-First (SRTF)，是等待时间最少的调度算法
- 非抢占式：必须等待直到正在运行的进程达到CPU突发 CPU burst
- 但是我们不知道进程的下一个CPU突发时长是多久，因此无法实现这个算法
- 但我们可以预测下一次CPU突发时长

预测下一个CPU突发的时长

- t_n 表示第n次CPU突发的时长（测量值）
- τ_{n+1} 表示对第n+1次CPU突发时长的预测
- τ_0 是一个常数，表示初始预测值
- $0 \leq \alpha \leq 1$ 表示一个估计的参数，越大表示上一次历史记录越重要，越小表示初始预测值越重要
- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

优先级调度 Priority Scheduling

- 每个进程有一个优先值 priority number
- 分为抢占式和非抢占式（新进入就绪列表的进程优先级如果最高，是否立刻抢占当前进程）
- 问题：饥饿Starvation：低优先级的进程可能永远无法执行
- 解决：老化Aging：进程会随着时间推移而优先级提升

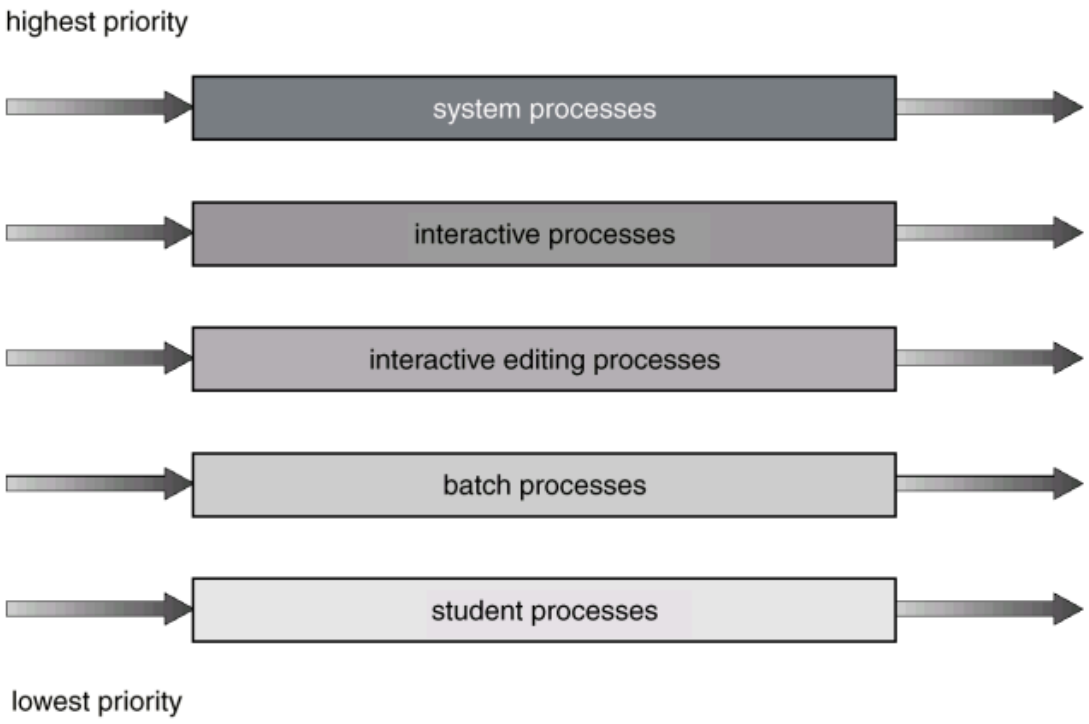
时间片轮转调度 Round Robin (RR)

- 每个进程有一个小的时间片 time quantum，每个进程轮流执行
- 进程运行完一个时间片后，会被强制抢占，并放回就绪队列的末尾，等待下一次轮转调用
- 实现会高度依赖于时间片长度q的设定：
    - 过长的时间片长度会让RR退化为FCFS

- 过短的时间片长度会带来过大的上下文切换开销
- 一般设定为10~100 milisecond

多级队列调度 Multilevel Queue Scheduling

- 就绪队列 ready queue 可以分为多个队列，每个队列可以有不同的调度策略，不同队列也有不同的优先级



- 前台程序 foreground 可以用RR，后台程序 background 可以用FCFS

多级反馈队列 Multilevel Feedback Queue

- 进程会在不同级别的队列里移动，有以下影响因素：
  - 队列数量
  - 每个队列的调度算法
  - 升级一个进程的方法
  - 降级一个进程的方法
  - 需要服务的进程应该去哪个队列
- 可以根据系统的实际目的来改变这些参数

线程调度 Thread Scheduling

- 进程内竞争范围 Process Contention Scope, PCS：只有同一进程内的线程相互竞争CPU
- 系统内竞争范围 System Contention Scope, SCS：整个系统的线程相互竞争CPU

实际操作系统的调度

Linux 调度

- 总体上是 $O(1)$ 时间复杂度，基于两种调度算法：

- 分时调度 Time-sharing：面向普通用户进程；有最多credit的进程优先运行，每次timer interrupt发生时，credit归零，换进程执行；当所有进程均完成执行时，重新分配credit
- 实时调度 Real-time：面向时间敏感的任务；定义了RR和FCFS两种实时调度类，有更高优先级的进程优先运行

Windows XP 调度

- 基于优先级的抢占式调度 Priority-based, preemptive scheduling，保证更高优先级的进程优先运行

	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

- 有API可以手动定义进程的优先级

Chapter 6 同步 Synchronization

临界区与竞态条件 Critical Section and Race Condition

回顾生产者-消费者问题 Producer-Customer Problem

- Consider this execution: initially, count = 5

Producer:	Consumer:	Values:
register1 = count		{register1 = 5}
register1 = register1 + 1		{register1 = 6}
	register2 = count	{register2 = 5}
	register2 = register2 – 1	{register2 = 4}
count = register1		{count = 6}
	count = register2	{count = 4}

- 由于两个线程没有同步，二者同时访问 `count` 变量，导致最后结果出错

概念

- 临界区 Critical Section：多个线程访问共享资源的代码段
- 竞态条件 Race Condition：输出结果取决于线程执行顺序，而可能输出不同结果的情况
- 发生竞态条件的必要条件
  - 并发性 Concurrency

- 共享数据 Shared data
- 干扰 Interference

竞态条件的解决

竞态条件解决的要求

- 互斥 Mutual Exclusion：如果有进程正在执行临界区代码，其它进程不能进入临界区
- 进步 Progress：如果没有任何进程正在执行临界区代码，必须选择一个进程进入临界区
- 有界等待 Bounded Waiting：进程等待进入临界区的时间是有限的

软件算法解决：Peterson's Solution

- 两个进程共享两个变量：
  - `int turn` 当前应该执行进程的编号
  - `Boolean flag[2]` 当前[编号]进程是否准备好执行

```
// Process i
while (1)
{
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);       // Busy waiting
    // Excute the critical section
    flag[i] = FALSE;
    // Excute the remainder section
}

// Process j
while (1)
{
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);       // Busy waiting
    // Excute the critical section
    flag[j] = FALSE;
    // Excute the remainder section
}
```

硬件同步 Synchronization Hardware

- 很多操作系统向临界区代码提供硬件支持
- 在单处理器 Uniprocessors 系统中，支持关闭中断（操作系统不会大规模使用这个方法，因为效率过低）（多核CPU中无法使用，因为其它CPU内核可能修改数据）
- 现代机器提供特殊的原子指令（即不可打断的指令）
- 使用自旋锁 spinlock 的缺点：一旦一个进程占用了锁，其它进程都需要在 `while` 循环里等待（busy waiting），占用了资源

原子指令：TestAndSet 解法

```
// 原子指令 TestAndSet()
// 保存锁的状态，并无条件立刻锁上锁
```

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}

while (TRUE)
{
    while (TestAndSet (&lock)); // 如果刚才没锁上(lock = False)则直接进入临界区
    // 临界区代码
    lock = FALSE;
    // 其它代码
}
```

原子指令：Swap 解法

```
// 原子指令 Swap()
// 直接交换手牌和锁牌
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

while (TRUE)
{
    key = TRUE;      // 进程手握红牌
    while (key == TRUE)      // 检测手牌，直到换成绿牌
    {
        Swap(&lock, &key);  // 把锁上的牌子和手牌交换（原子指令）
    }
    // 临界区代码
    lock = FALSE;        // 把锁上的牌子换成绿牌
    // 其它代码
}
```

信号量 Semaphore

- 一个整数变量，表示可用资源的数量
- 只有两个不可分割的原子操作可以更改
    - `acquire()` / `wait()` / `P()` 申请进入临界区
    - `release()` / `signal()` / `V()` 释放资源，退出临界区

信号量的实现

- 不能用忙等待 busy waiting 实现，否则在高并发场景下执行效率过低
- 每个信号量关联一个等待队列 waiting queue，用链表实现
- 每个队列元素包含一个整数和一个前向指针
- 支持两种原子操作：
    - `block()` 阻塞：把发起操作的进程/线程加入对应等待队列中

    - `wakeup()` 唤醒：从等待队列中移除一个进程/线程，并置入就绪队列
- `value` 正值表示当前可用资源的数量，负值表示等待队列中进程/线程的数量

```
void acquire(value)
{
    value--;
    if (value < 0)
    {
        // Add this process to list
        block();
    }
}

void release(value)
{
    value++;
    if (value <= 0)
    {
        // Remove a process P from list
        wakeup(P);  // 对应直接解除进程 P 的阻塞状态
    }
}
```

信号量的分类

- 计数信号量 Counting semaphore：信号量可以不受限制地增加/减少
- 二元信号量 Binary semaphore（又称互斥锁mutex locks）：信号量只能是0或1

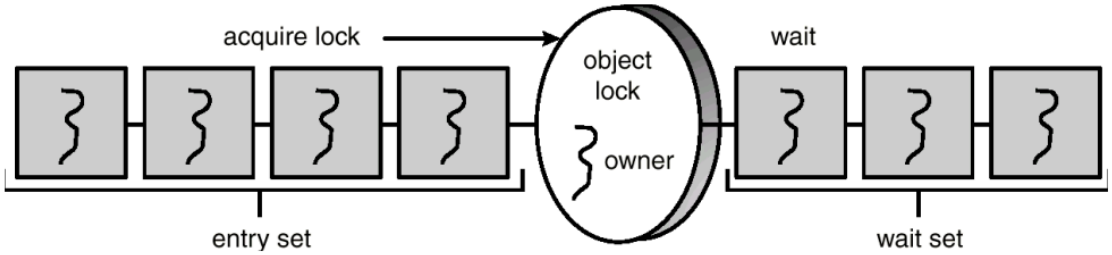
死锁与饥饿 Deadlock and Starvation

- 死锁：两个进程或多个进程无限等待一个事件，而这个事件只能由其中一个等待状态的进程触发
- 饥饿：一个进程永远等不到资源/调度
- 优先级反转：低优先级进程持有高优先级进程所需要的锁时，可能由于低优先级进程长期无法执行，从而间接导致高优先级进程被阻塞

POSIX线程库 Pthread 同步API

- 互斥锁 mutex lock
    - `pthread_mutex_t` 互斥锁类型定义
    - `pthread_mutex_init(&lock, NULL)` 互斥锁初始化
    - `pthread_mutex_lock(&lock)` 互斥锁加锁（`acquire()`）
    - `pthread_mutex_unlock(&lock)` 互斥锁释放（`release()`）
- 条件变量 conditional variable
    - `pthread_cond_t` 条件变量类型定义
    - `pthread_cond_wait(&cond, &lock)` 等待条件变量满足后，重新获取互斥锁
    - `pthread_cond_signal(&cond)` 唤醒一个正在等待该条件变量的线程
    - `pthread_cond_broadcast(&cond)` 唤醒所有正在等待该条件变量的线程

管程 Monitor



- `wait()` 线程等待
  - 线程释放锁
  - 线程状态设置为阻塞 blocked
  - 线程进入等待集合 wait set
- `notify()` 线程唤醒
  - 从等待集合 wait set 里随机选一个线程，置入入口集合 entry set
  - 设置这个线程状态为可运行 runnable
- `notifyAll()` 唤醒全部等待集合里的线程，置入入口集合并设置为可运行 runnable
- 优点：相对简单，不需要使用成对的`acquire()`和`release()`，可以有效防止程序员忘记释放锁
- Java内置了这种实现
- 很多高级语言实现了管程 Monitor

同步经典问题

有界缓冲区 Bounded Buffer (Producer-Consumer)

- 有一个缓冲区 Buffer，其容量有限 (N)
- 有一个生产者 Producer，会向缓冲区内装东西 item；如果缓冲区满了，则等待
- 有一个消费者 Consumer，会从缓冲区里拿东西出来；如果缓冲区为空，则等待
- 定义信号量
  - `mutex` 缓冲区进入权限控制：初始化为1
  - `full` 满槽计数：初始化为0
  - `empty` 空槽计数：初始化为N

```
Item buffer[N];
int in = 0, out = 0;
Semaphore mutex = 1, full = 0, empty = N;

public void insert (Item item)
{
    empty.acquire();     // 瞅一眼货架满不满：不满就补货
    mutex.acquire();     // 把顾客赶走，要补货了
    // 以上两条语句不可交换，否则可能死锁：货架满了，但是顾客被赶走了

    buffer[in] = item;
    in = (in + 1) % N;
```

```
    mutex.release();     // 补货完成了，允许顾客进来
    full.release();      // 通知顾客：有货了
}

public Item remove()
{
    full.acquire();      // 瞅一眼货架有没有货：有货就来买
    mutex.acquire();     // 把补货员赶走，要买货了
    // 同理，以上两句不可交换

    item = buffer[out];
    out = (out + 1) % N;

    mutex.release();     // 买完了，允许补货员来补货
    empty.release();     // 通知补货员：可以补货了
    return item;
}
```

- 同步（管程）解法：

```
monitor Produce-Consumer
{
    // 在类内，关键词 synchronized 标识的所有函数，同时只能有1个线程访问
    // insert() 和 remove() 也不能同时被访问
    public synchronized void insert(Object item)
    {
        while (count == N)
        {
            try { wait(); }
            catch (InterruptedEx e)
            {}
        }
        ++count;
        buffer[in] = item;
        in = (in + 1) % N;
        notify();           // 唤醒正在 wait() 中的消费者
    }

    public synchronized Object remove()
    {
        Object item;
        while (count == 0)
        {
            try { wait(); }
            catch (InterruptedEx e)
            {}
        }
        --count;
        item = buffer[out];
        out = (out + 1) % N;
        notify();           // 唤醒正在 wait() 中的生产者
        return item;
    }
}
```

读写问题 Readers and Writers Problem

- 同时只有一个作家可以进行写操作
- 可以有很多读者，读者之间可能发生并发访问操作
- 读写不能同时进行
- 定义信号量
  - `db` 需要读写的数据区进入权限
  - `mutex` 读者数量权限控制
- 安全条件：`(nr = 0 or nw = 0) and nw <= 1`

```
int nr = 0;             // 正在读的读者数量
Semaphore mutex = 1;    // 读者数量 nr 的变量锁
Semaphore db = 1;       // 数据访问锁

void writer()
{
    while (1)
    {
        P(db);      // Acquire(): db--
        // Write
        V(db);      // Release(): db++
    }
}

void reader()
{
    while (1)
    {
        P(mutex);       // Acquire(): mutex--
        nr++;           // 读者人数+1
        if (nr == 1)    // 如果是第一个读者，请求数据访问锁
        {
            P(db);      // Acquire(): db--
        }
        V(mutex);       // Release(): mutex++
        // Read
        P(mutex);       // Acquire(): mutex--
        nr--;           // 读者数量-1
        if (nr == 0)    // 如果是最后一个读者，释放数据访问锁
        {
            V(db);      // Release(): db++
        }
        V(mutex);       // Release(): mutex++
    }
}
```

哲学家进餐问题 Dining-Philosophers Problem

- 5个哲学家，每个人都有两种操作：吃饭/思考
- 在圆桌边坐成一圈，每个人面前有一个碗，两两之间有一把叉子
- 吃饭操作需要拿起自己左右两把叉子，吃完后会放下

```
monitor DP
{
    typedef struct enum
    {
        THINKING, HUNGRY, EATING
    } state[5];          // 5个哲学家的状态：不饿、饿了想吃、正在吃
    condition self[5];   // 条件变量：等待区

    synchronized void pickup (int i)
    {
        state[i] = HUNGRY;           // 哲学家 i 宣布饿了
        test(i);                     // 检验允不允许吃
        if (state[i] != EATING)      // 如果 i 没吃上（邻居在吃）
        {
            self[i].wait();          // i 睡觉
        }
    }

    synchronized void putdown (int i)
    {
        state[i] = THINKING;         // 哲学家 i 吃完了
        test((i + 4) % 5);           // 检查上一个哲学家允不允许吃，并提醒他
        test((i + 1) % 5);           // 检查下一个哲学家允不允许吃，并提醒他
    }

    synchronized void test (int i)  // 检查哲学家 i 允不允许吃，并提醒他
    {
        if ((state[(i + 4) % 5] != EATING) &&   // 上一个不在吃
            (state[i] == HUNGRY) &&             // 自己饿了
            (state[(i + 1) % 5] != EATING))     // 下一个不在吃
        {
            state[i] = EATING;       // 哲学家 i 吃饭
            self[i].signal();        // 提醒 i 吃饭
        }
    }

    void initialization_code()       // 初始化：每个人都不饿
    {
        for (int i=0; i<5; i++)
        {
            state[i] = THINGKING;
        }
    }
}
```

- 这个解法有饥饿 Starvation 的风险：左右侧哲学家轮流吃饭，自己永远吃不上

Chapter 7 死锁 Deadlock

死锁定义

- 两个进程或多个进程无限等待一个事件，而这个事件只能由其中一个等待状态的进程触发
- 经典类比：过桥问题（狭路相逢）

- 没有任何一个进程可以执行，CPU有可能处于空闲状态
- 对应地，存在活锁 Livelock，指若干进程尝试解决冲突时，同时试图改变自己状态来适应对方变化，而实际上无法解决问题。类比狭路相逢时，两者同时向左/向右而堵住对方，导致进程都在执行，但无法取得有效进展

死锁的必要条件

- 互斥 Mutual exclusion：同时只能有1个进程使用指定的资源
- 持有并等待 Hold and wait：持有资源的进程正在等待其它进程
- 无抢占 No preemption：资源只能被进程自愿释放，而不能被抢占
- 循环等待 Circular wait：一系列进程互相等待其它进程持有的资源

系统模型

- 一共有m中资源类型 $R_1, \cdots, R_m$，每种资源 R_i 有 W_i 个实例
- 每个进程可以有3种利用资源的方式：`acquire`，`use`，`release`

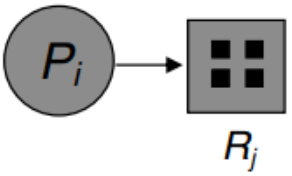
资源分配图 Resource-Allocation Graph

- 有向图
- 有两种类型的节点：
  - 进程 P_i
  - 资源类型 R_j
- 有两种类型的边：
  - 请求边 request edge：$P_i \to R_j$ 进程正在请求资源
  - 分配边 assignment edge: $R_j \to P_i$ 资源（的实例）已经被分配给进程

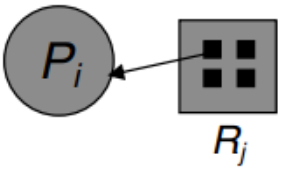
- Process

- Resource Type with 4 instances

- P_i requests instance of R_j



- P_i is holding an instance of R_j



解决死锁的方法

- 避免 Avoid：保证从不进入死锁状态
- 检验 Detect：允许系统进入死锁，然后试图恢复
- 忽略 Ignore：不由系统处理，由程序开发者处理（大部分操作系统的处理方式）

防止死锁

要防止死锁，必须让死锁的四个必要条件之一不成立

- 互斥：让资源可共享，如只读模式（不现实）
- 持有并等待：进程持有资源时不能请求其它资源，且在执行前申请所有资源（不现实，且效率低）
- 无抢占：允许发生抢占，当进程需要无法获取的资源时，直接释放该进程，直到该进程的所有资源需求都满足，才重启该进程（不现实）
- 循环等待：所有进程都必须以资源编号的固有顺序来获取资源
  - 最简单且有效的方法，是要求进程必须提前声明每种资源的最大可能需求量，然后用死锁避免算法来动态监测来保证不会死锁
  - 安全状态：每个进程需要的资源，都可以通过 `当前可用资源` 与 `比它先执行的进程所占有的资源` 来满足，则处于安全状态。安全状态下不会发生死锁

资源分配图方案

- 定义一个声明边 Claim edge $P_i --> R_j$，表示进程有可能需要资源，用虚线表示
- 当实际发出该请求时，声明边转换为请求边
- 当请求的资源被实际分配时，请求边转换为分配边
- 当资源利用完成被释放时，分配边回到声明边
- 进程只有处于初始状态（不包含除了声明边以外的边）时，才能添加声明边
- 对于每种资源类型只有1个实例的问题，做环检测即可解决
- 可以使用 `banker's algorithm` 来解决每种资源类型有多个实例的问题（本课程不要求）

检验与解决死锁

- 每隔一段时间检查一次死锁；死锁发生越频繁，死锁检验越应该频繁触发
- 每次检验就是检验n个进程的等待图 wait-for graph 里有没有环，时间复杂度 $O(n^2)$
- 检验死锁的策略：当CPU利用率降低到阈值以下（如40%）时，会触发死锁检验
- 解决死锁的策略：按顺序每次停止一个进程/线程并回滚 rollback 到初始状态，直到不再死锁
    - 顺序：如优先级低、已经执行时间短、使用资源少、还需要使用的资源多、总共需要终止更少线程的、批处理线程而非交互线程
    - 问题：可能有单个进程被反复回滚，造成饥饿（可以记录每个进程回滚轮次，优先回滚轮次更少的进程）

Chapter 8 内存管理 Memory Management

背景知识

- 程序必须从磁盘 disk 上被加载进入内存 memory，并被置入进程，才能运行
- CPU唯一可以直接访问的存储器是主内存和寄存器
- 寄存器访问只需要1个或更少的CPU时钟周期，而主存则需要数十个时钟周期
- 主存和CPU寄存器之间有缓存 Cache
- 需要有保护操作，才能保证内存不被错误访问

内存保护

- 使用一对寄存器 `base` 和 `limit` 来定义逻辑地址空间 `[base, base + limit]`
- 加载 base 和 limit 寄存器的指令是特权指令

地址绑定 Address Binding

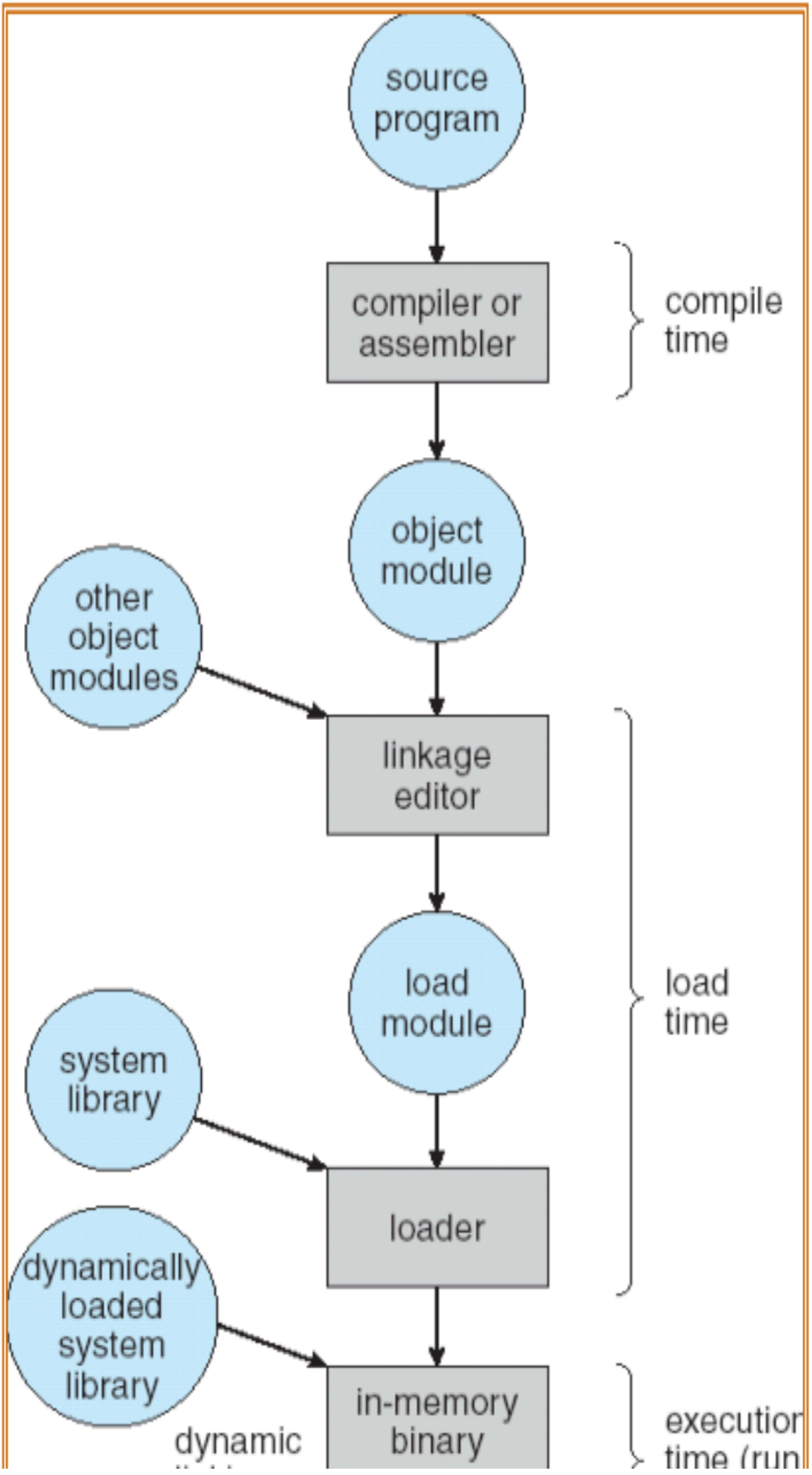
- 等待被加载到内存中的程序会形成一个输入队列
- 如果没有特殊支持，程序会被默认加载到0000地址开始的位置，而这是不方便的（地址冲突；和操作系统相关地址也会冲突）

不同阶段程序地址的表示形式

- 源代码阶段 Source code：地址是符号化的，如变量名

- 编译后阶段 Compiled code：地址被绑定到一个可重定位地址 relocatable addresses，即相对地址，如从模块开头的第14字节
- 链接/加载阶段 Linker & Loader：绝对地址（真实物理地址或虚拟地址）
- 本质上是一种地址的映射/绑定 mapping/binding

处理用户程序的阶段


source program
compiler or assembler
compile time
object module
other object modules
linkage editor
load module
load time
system library
loader
dynamically loaded system library
in-memory binary
execution
dynamic

- 编译时绑定：在编译时直接确定内存地址（地址发生改变时必须重编译，不方便）
- 加载时绑定：如果编译时不能确定内存地址，必须生成可重定位代码（运行时不能移动程序位置）
- 运行时绑定：最灵活的模式，也是大多数现代操作系统的模式，需要内存管理单元MMU的硬件支持，且有一定运行时开销

逻辑地址与物理地址 Logical & Physical Address

- 逻辑地址/虚拟地址 Logical/Virtual Address：由CPU生成的地址
- 物理地址 Physical Address：由内存单元定义的物理内存地址
- 虚拟地址和物理地址在编译时绑定和加载时绑定中都是相同的，而执行时绑定中二者不同

内存管理单元 Memory-Management Unit, MMU

- 一种硬件，负责从虚拟地址到物理地址的映射
- 有一个重定位寄存器 relocation register，其中的值加上虚拟地址得到物理地址
- 每次CPU访问逻辑地址，传递给MMU，然后MMU将地址转换为物理地址，进入内存进行访问

动态加载 Dynamic Loading

- 很多代码在运行时不一定用到，而在运行前将代码全部加载进内存会浪费内存空间
- 动态加载：一个模块 Routine 在被调用时，才被加载进内存中
- 不用操作系统的特殊支持，即可实现动态加载
- 可以提高内存利用率

动态链接 Dynamic Linking

- 动态链接：直到运行时才链接
- 在程序代码中包含一个小代码段 Stub，用于运行时定位内存中已经加载的库函数地址
- 程序首次调用某个库函数时，Stub会找到该函数的地址，并用地址替换自身，后续再次调用时直接跳转
- 在多程序共用库函数时，可以大幅降低内存占用
- 对于库函数更新的适应较为方便
- 可执行文件的大小缩减了
- 运行时链接会有时间开销

内存分配

交换 Swapping

- 内存有限，而进程很多，内存有可能不够用，所以需要交换 Swapping
- 进程可以被临时换出 swapped out 到后备存储区 backing store（例如磁盘 disk），之后可以重新被换入 brought back 内存继续执行
- 交换的主要时间开销在于传输时间，传输时间与需要交换的内存大小成正比

- 操作系统常驻内存，不会被交换出去

内存连续分配

- 洞 Hole：一块连续的可用内存；内存中存在分布各处的、大小不一的洞
- 当进程到达时，会被分配进入一个足够容纳它的洞
- 操作系统需要维护内存已经被分配的部分，以及每个洞的位置、每个洞的大小
- 操作系统一般放在低地址（硬件决定），包含中断向量
- MMU会负责内存动态映射，把虚拟地址转换为物理地址
- 重定位寄存器 Relocation registers （包括基址寄存器 Base register 和界限寄存器 Limit register）负责防止用户进程越界访问

动态存储分配问题

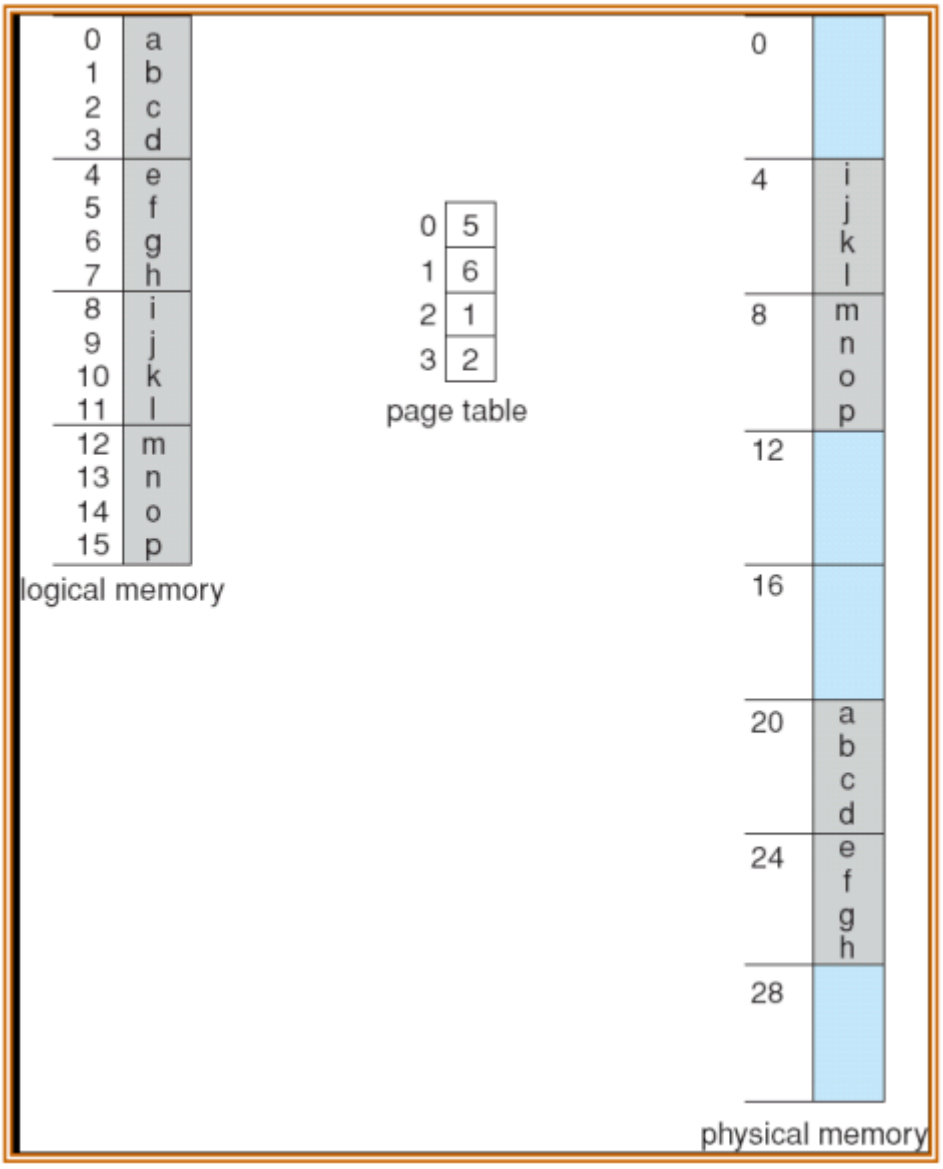
- 如何从一系列洞中找到一个，来满足一个大小为n的内存请求?
- 策略
  - First-fit：分配第一个足够大的洞
  - Best-fit：分配最小的足够大的洞
  - Worst-fit：分配最大的洞
- 在实际使用中，First-fit 和 Best-fit 是更优的算法；First-fit是能保持空间利用率的同时，有较高空间效率的算法，因此更受欢迎
- Best-fit 能产生一个最小的洞，而 Worst-fit 能产生一个最大的洞

内存碎片 Fragmentation

- 外部碎片 External Fragmentation：系统中总空闲内存足够满足某个进程的内存请求，但是由于空闲内存不连续，因此分配失败
- 内部碎片 Internal Fragmantation：分配给进程的内存比进程实际请求的稍大，而进程不会使用，别的进程也无法使用
- 紧凑压缩 Compaction：通过挪动进程位置，来把空闲内存合并到连续块中（前提是操作系统需要支持动态重定位 Relocation）

分页 Paging

- 由于连续内存分配会产生外部碎片，考虑非连续的内存分配方案：分页 Paging
- 帧 Frame：把物理内存分为固定大小（一般是2的幂，512~8192字节，主流是4096字节=4kb）的块
- 页 Page：把虚拟内存分配为和帧相同大小的块
- 当一个进程需要 n 页时，操作系统会给该进程设定一个页表 Page Table，里面存储了每页的逻辑地址到对应帧的物理地址的映射
  - 具体地，从虚拟地址映射到物理地址时，虚拟地址被拆分为页码 Page number 和页偏移 Page offset，然后通过查页表来进行页码对应，页偏移不变
  - 整个地址转换过程发生在内存管理单元 MMU 中，速度很快
- 这样处理只会产生较少的内部碎片，比连续内存分配更优，是现当代操作系统内存管理的基本方法



- 单页大小越小，内部碎片越小，但是对应的页表会更大
- 随着物理内存增大，现代计算机的页表正在越来越大
- 操作系统会维护一个空闲帧列表 Free Frame List；当新的进程请求页时，操作系统会给每页分配一个帧来存储
- 页表 Page table 存储在内存中，而页表基址寄存器 Page-table base register (PTBR) 指向页表在内存中的起始地址，页表长度寄存器 Page-table length register (PTLR) 存储了页表的大小

转换检测缓冲区 TLB

- 按照页表的实现，每次访问数据都需要两次内存访问：访问页表找到物理地址，然后访问物理地址找到数据
- 但内存访问的速度较慢，因此有硬件优化：引入快表（转换检测缓冲区） associative memory / translation look-aside buffers (TLBs)
- 这个硬件实现的缓存能极快地（并行查找）将虚拟页号映射到物理地址，会存储最近使用过的页表项；每次CPU访问内存数据前，先查找TLB；如果找到了 (TLB hit)，则直接一次访问内存找到数据；如果没找到 (TLB miss)，则两次访问内存，先找页表，再找数据
- 为了防止多进程页号映射混乱，有些TLB同时存储地址空间标识符 Address-Space Identifier (ASID)，唯一标识每个进程的虚拟地址空间对应的页表段
- 有效访问时间 Effective Access Time（EAT）：一个衡量访存速度的定量指标
    - TLB的查找时间为 ϵ 时间单位

- 假设内存周期时间为 1 时间单位（内存进行1次读/写操作的时间）
- TLB命中率 Hit ratio 为 α
- 有效访问时间 EAT = $(1+\epsilon)\alpha + (2+\epsilon)(1-\alpha) = 2+\epsilon-\alpha$
- （1 查快表成功 + 1 访存）或（1 查快表失败 + 2 访存）

页的内存保护

- 对物理内存上的每一帧都有独立的保护位，可以决定是否只读、是否可执行等
- 页表中对于每页都有一个是否合法的位 Valid-invalid bit，决定这一页是否属于该进程的合法逻辑地址；页表长度寄存器 PTLR 也能检查这类越界访问
- 所有非法访问都会触发一个内核陷阱 Trap to kernel，然后触发内核的错误处理

共享页

- 共享代码 Shared code
  - 多个进程共享一份只读、可重入 reentrant 代码（不同进程可以同时执行互不影响）
  - 如文本编辑器、窗口系统等
  - 如果允许共享读写 read-write 页，进程间可以通过共享页来通信
- 私有代码和数据 Private code and data
  - 每个进程会维护独立的代码和数据副本

页表结构

多层分页 Hierarchical Paging

- 把页表分成多个小页表，即对页表进行分页
- 以二层分页为例：页表分为外部页表 Outer Page Table 和内部页表 (Inner) Page Table，外部页表里面存储了内部页表的地址，而内部页表里面存储内存物理地址；当内部页表指示的内存地址都没有被使用时，对应的内部页表都不会被生成
- 在32位操作系统（逻辑地址长度与寄存器宽度均为为32 bit的操作系统，内存最大4 GB）中，二层分页是可以正常使用的：外部分页占12位，内部分页占10位，页偏移量 offset 占10位
- 但在64位操作系统中，二层分页还是会导致外部页表过大；因此使用三层分页：第二外层分页 2nd outer page 占32位，外层分页 outer page 占10位，内层分页占10位，页偏移量 offset 占12位
- 过于多层的分页会导致较大的访存延时

哈希页表 Hashed Page Tables

- 使用链表处理哈希冲突，用哈希表把页号映射到地址链表，然后在链表中依次查询，找到物理地址
- 在哈希函数较好的情况下，查找速度最快

倒置页表 Inverted Page Tables

- 不存储“页 - 帧”的映射，而是存储“帧 - 页”映射，即存储每个物理地址上的帧对应的页号、进程PID
- 相比于传统页表，能够节省大量空间；但是查找较慢，且实现内存共享较为困难
- 也可以配合哈希表来加速搜索

Chapter 9 虚拟内存 Virtual Memory

设计背景

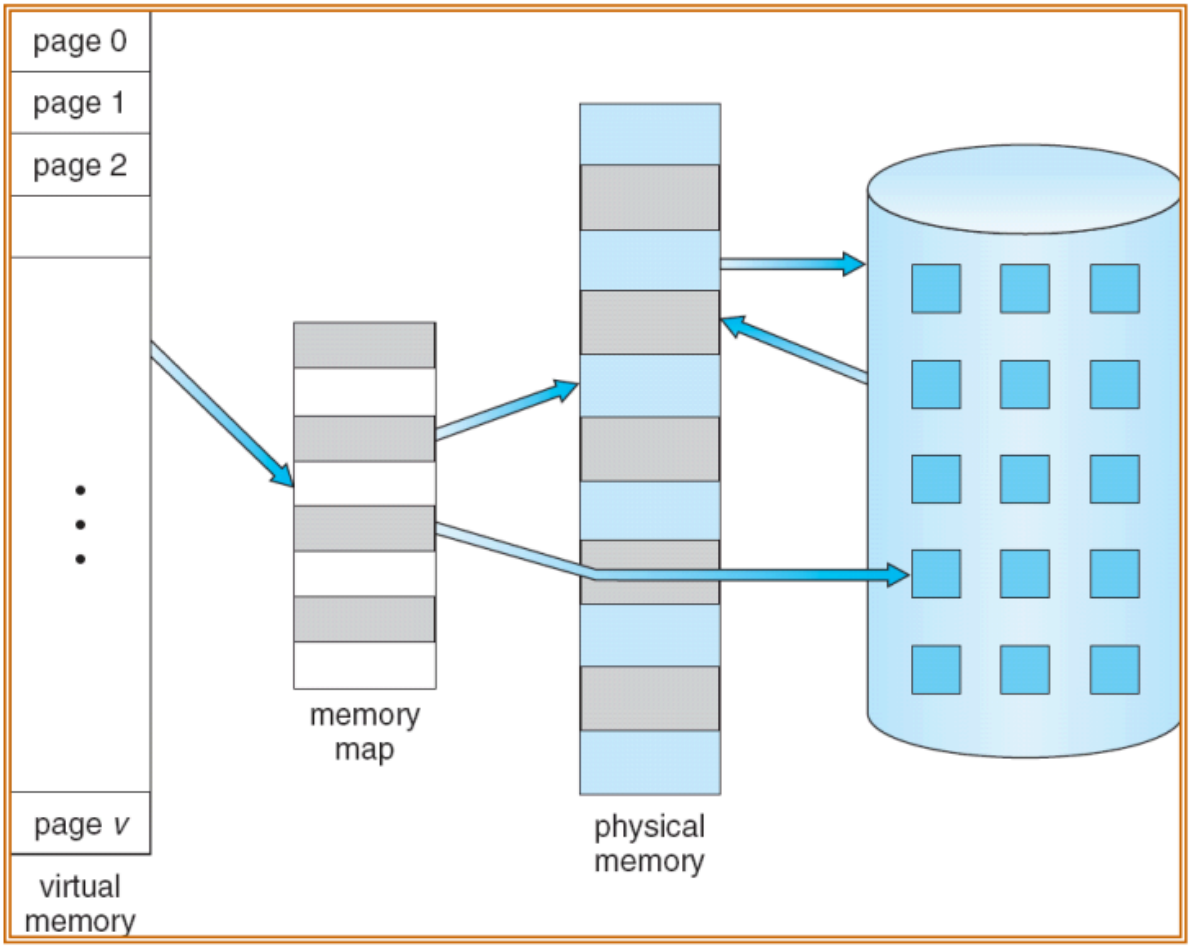
- 程序需要的内存大小超过了物理内存的大小
- 代码需要在内存中才能执行，但是在大多数情况下，整个程序代码不会同时被执行
- 因此可以将不常用的代码暂时存储在硬盘中，只将一小部分正在频繁使用的代码存入内存中

虚拟内存的优势

- 逻辑地址空间可以比物理地址空间大得多
- 不同的逻辑地址可以映射到同一个物理地址上，多个进程可以共享地址空间（如共享库函数）
- 建立新进程更快（不用复制旧进程的整个内存空间了）
- 更高的并发度（同时可以在内存中运行很多程序）
- 减少无效的 I/O 消耗（不需要加载完整的大程序）

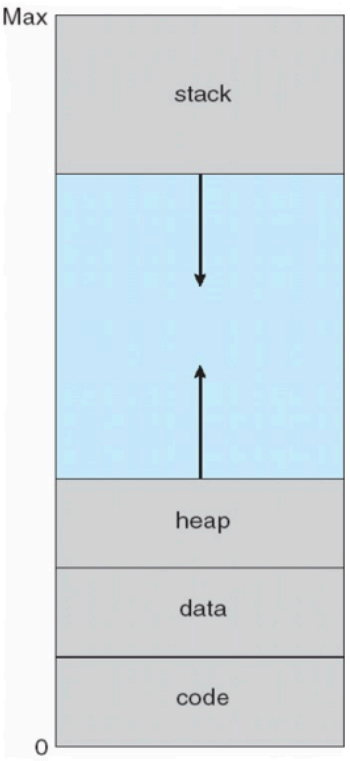
虚拟内存的实现

- 在进程的视角，虚拟地址从0地址开始，是连续的地址空间
- 对应的物理地址散落在各处
- 其中，MMU负责将虚拟地址映射到物理地址
- 虚拟内存可以由请求分页 Demand paging （每一页大小相同）或请求分段 Demand segmentation （每一段基于逻辑意义，长度可变；在现代操作系统中比较少见）实现



- 对于一个进程，其栈空间 stack 存在高地址，堆空间 heap 存在低地址；其间稀疏的 sparse 空洞 hole 在堆/栈增长到新的页面时，不需要重新分配新的物理内存
- 对于系统库，不同的进程可以共享系统库的页，而让不同的进程的“洞”里的某些逻辑地址映射到同一块物理地址上
- `fork()` 创建新进程时，内核会让子进程复制父进程的虚拟地址空间（页表），而物理地址上则是“共享只读”（写时复制 copy-on-write），从而加速了进程创建

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

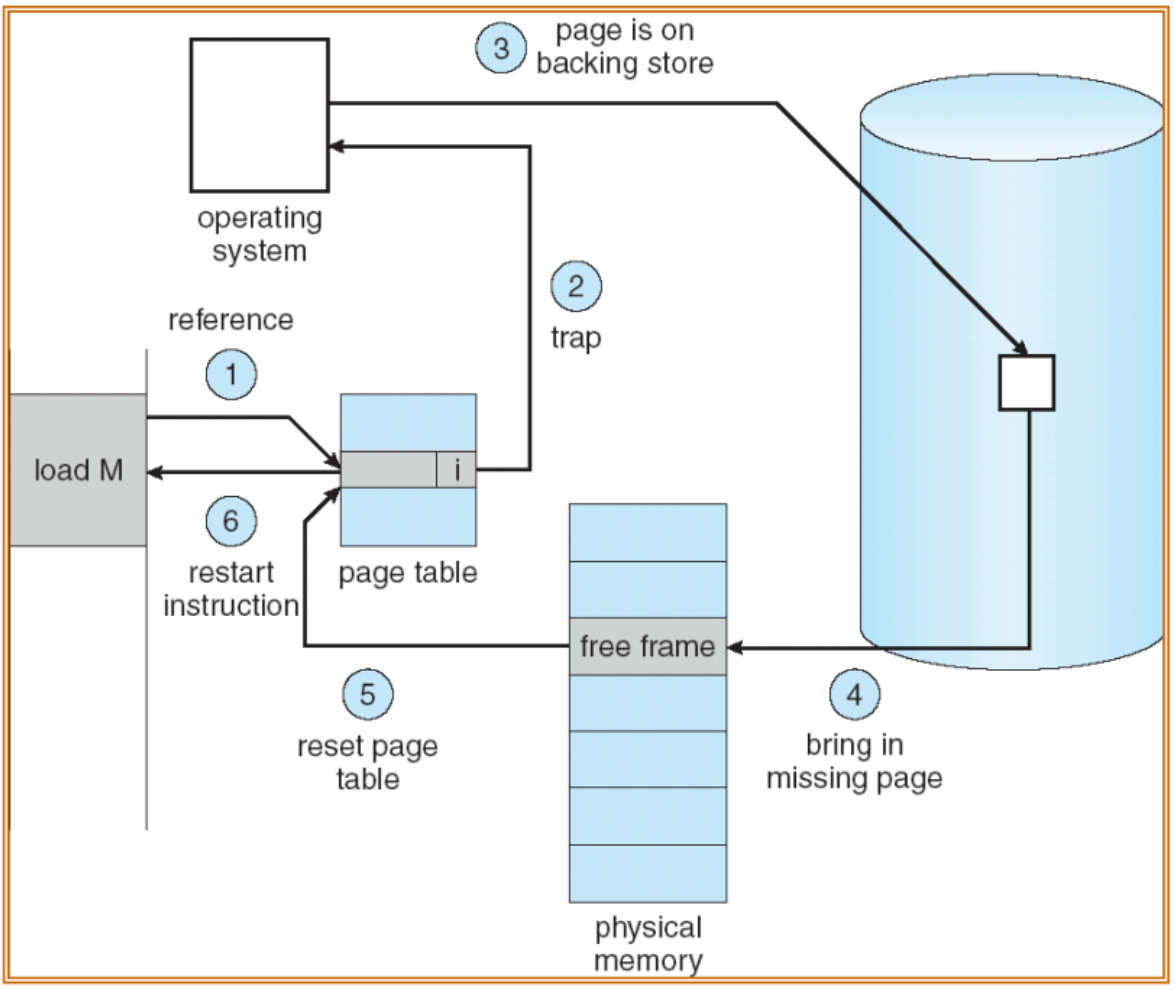


请求分页 Demand Paging

- 仅当需要时，才把某页载入内存
- 优势：减少I/O，减少进程内存需求，支持更多并发进程
- 每个进程页在硬盘上的地址尽可能连续（加快硬盘读写）
- 进程的页表中，每一页都有一个验证位 Valid-Invalid Bit
  - `v` 表示在内存中，`i` 表示不在内存中
  - 在地址翻译时，如果遇到 `i`，则会引发一个缺页中断 page fault

缺页中断 Page fault

- 先检查进程的PCB，检查这次访问是否是非法访问：非法则直接终止程序
- 在物理内存中获取一个空闲帧 empty frame
- 从磁盘将页面换入帧中 Swap page into frame
- 重置页表，并设置验证位 Valid-Invalid Bit 为 `v`
- 重新执行引发缺页中断 Page fault 的指令



- 纯请求分页 Pure demand paging：进程从内存中没有页的情况下加载，所有页都需要从硬盘中读取
- 多重缺页中断 Multiple page faults：一条指令可能需要访问多页（可以由局部性原理缓解：程序通常倾向于在短时间内重复访问一小段内存区域）
- 缺页中断的处理极慢，因为CPU必须停下来，等待内存、二级存储的响应，而且还需要从二级存储向内存传输数据

缺页中断的硬件支持

- 带验证位 Valid-Invalid Bit 的页表（在内存里）
- 二级存储（交换空间 Swap Space）：不直接由CPU访问的存储设备，即外存（硬盘、U盘等）
- 支持指令重启 Instruction Restart

空闲帧列表 free-frame list

- 当缺页中断发生时，操作系统必须从二级存储中把需要的页放入内存中
- 大多数操作系统会维护一个空闲帧列表 free-frame-list，动态存储每一时刻的空闲帧
- 使用 zero-fill-on-demand 原则，即空页被申请之前会被清空

请求分页的性能评估

- 设缺页中断的发生概率为 `p`
- 有效访问时间 Effective Access Time `EAT = (1-p) * memory_access + p * (page_fault_overhead + swap_out + swap_in + restart_overhead)`
- 内存访问的时间大约是 200 ns

- 平均缺页中断的处理时间约为 8 ms = 8×10^6 ns

写时复制 Copy-on-Write

- 当进程使用 `fork()` 创建新进程时，子进程先复制父进程的页表，然后父进程与子进程暂时只读共享内存中的分页；当父进程/子进程需要修改某个页面时，会触发一个异常，导致该页面先复制再修改
- 当进程使用 `vfork()` 创建新进程时，页表不会发生复制，父进程直接停止运行，子进程直接继承父进程的地址空间；直到子进程 `exit()` 或 `exec()`，父进程恢复运行（此时父进程会看到子进程的修改；一般用于子进程直接 `exec()` 的场景）

页交换 Page replacement

- 当内存中没有空闲页时，需要将页换出 swap out
- 内存为每页维护一个 `dirty bit`，存储这页是否被修改过；若未被修改过，则说明硬盘中存在天然备份，发生换出时不再保存，直接清空

基本步骤

- 在二级存储上找到需要的页
- 试图查找一个空闲页：
  - 如果有，直接使用
  - 如果没有，使用页交换算法选定一个牺牲帧 victim frame，并修改页表
- 把需要的页换入空闲页，更新页表
- 指令重启

页交换策略

- Global replacement：进程从全局所有页中选一页替换（会导致进程可能被疯狂抢夺资源，运行时间不可预测）
- Local replacement：进程只能从自己分配的页中替换（不够灵活，无法充分利用内存）

页交换算法

评估标准

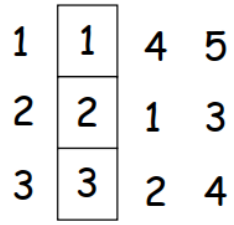
- 我们需要最低的缺页中断率
- 在一个特定的内存引用串 reference string 上运行，计算缺页中断发生的次数

先进先出 FIFO Algorithm

- 把最早进入的页面踢出
- 贝莱迪异常 Belady's Anomaly：当帧增加时，缺页中断数量反而增加了（只有部分算法存在这个现象）

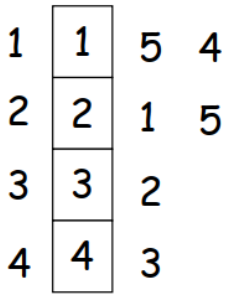
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



9 page faults

- 4 frames



10 page faults

- Belady's Anomaly: more frames ⇒ more page faults

最优算法 Optimal Algorithm

- 是所有情况下最优的算法（贪心的）
- 总是换出未来最长时间内用不到的页
- 现实不可做到

最近最少访问算法 Least Recently Used Algorithm (LRU)

- 总是踢出最长时间没有访问过的页
- 逻辑：短时间内访问过的页会被反复访问
- 不会发生贝莱迪异常
- 实现：每页都对应维护一个时间戳，每次访问时，把当前时间复制进入时间戳
- 是现代操作系统中的主流算法，但存储每个时间戳开销较大，故常用近似算法模拟

引用位法 Reference Bit

- 对于每个页，维护一个引用位 Reference bit，初始值都是 `0`；每当一个页面被访问时，就把对应的引用位设置为 `1`
- 当需要换出页面时，踢出第一个引用位为 `0` 的页面

二次机会算法/时钟算法 Second Chance

- 对于每个页，维护一个引用位 Reference bit，初始值都是 `0`
- 当需要换出页面时，循环扫描内存中的页面：如果引用位是 `0`，则直接踢出并把指针移到下一个位置，结束扫描过程并保留指针位置；如果引用位是1，则"给它第二次机会"，把引用位设为 `0` 并继续检查下一个页面

最低频访问算法 Least Frequently Used Algorithm (LFU)

- 对于每个页面进行访问计数，总是踢出访问次数最少的页面
- 访问次数多意味着更有可能继续被访问

最经常使用算法 Most Frequently Used Algorithm (MFU)

- 对于每个页面进行访问计数，总是踢出访问次数最多的页面
- 访问次数少意味着有可能是新进入内存但还没大量使用的

页分配 Allocation of Frames

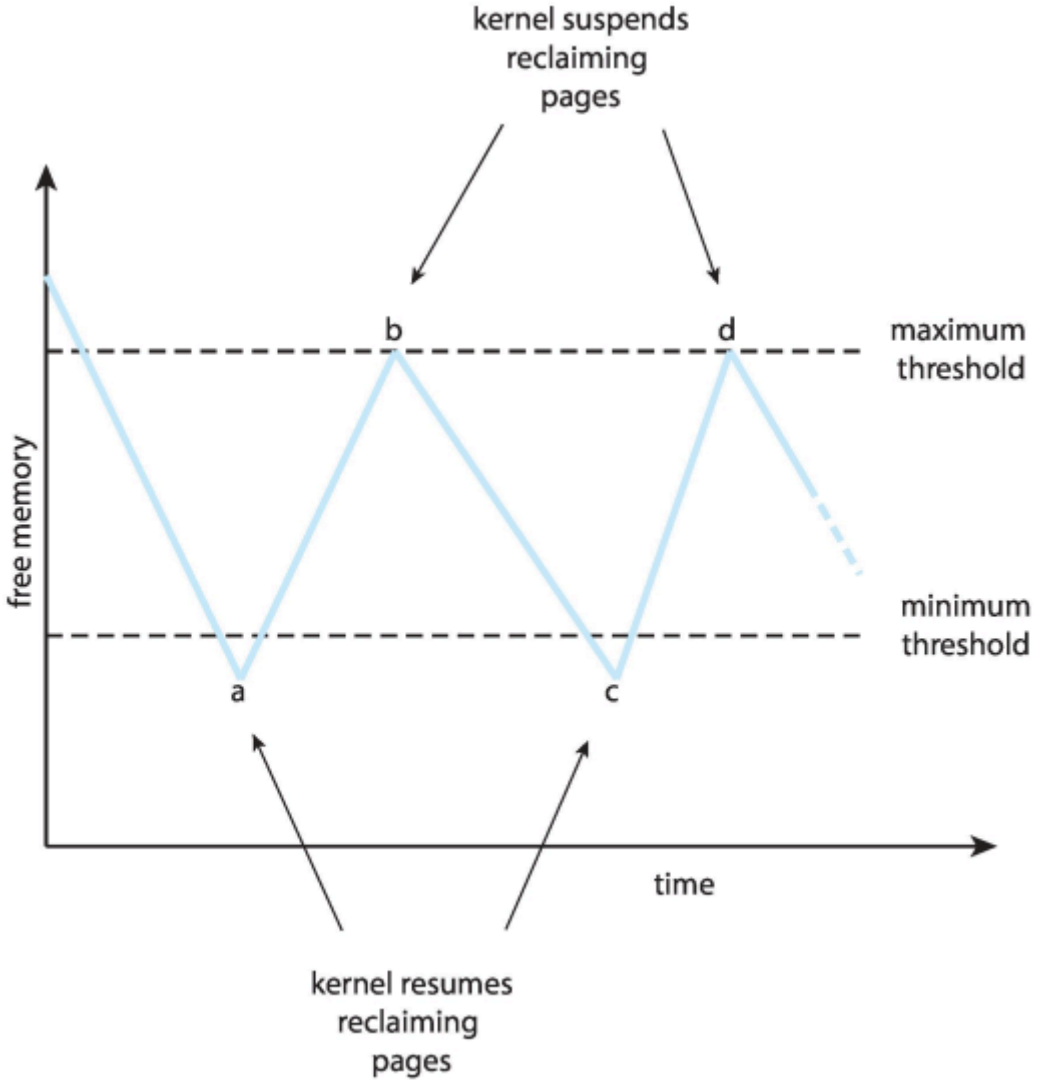
- 每个进程需要最少的页数量，IBM370规定了至少需要6页来处理 `ss move` 指令
- 主要分为固定分配 fixed allocation 和优先级分配 priority allocation 两种

固定分配 Fixed Allocation

- 等量分配 Equal allocation：对每个进程都分配相同大小的内存资源（不合理）
- 成比例分配 Proportional allocation：基于进程当前的大小按比例分配内存资源

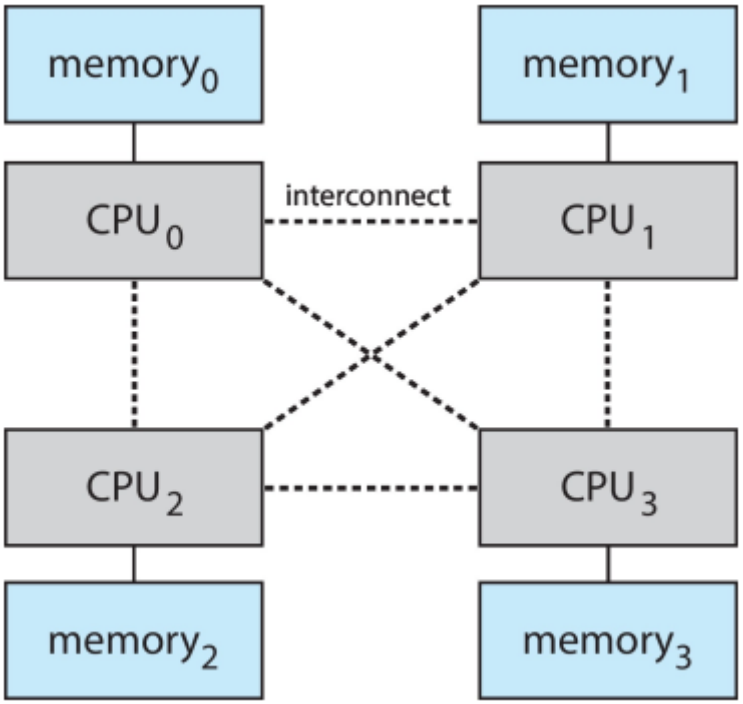
页回收 Reclaiming Pages

- 当空闲页列表长度低于某个阈值时，会自动触发页交换，直到空闲页列表长度回到最大回收阈值以上时，暂停页交换



非均匀内存访问 Non-Uniform Memory Access, NUMA

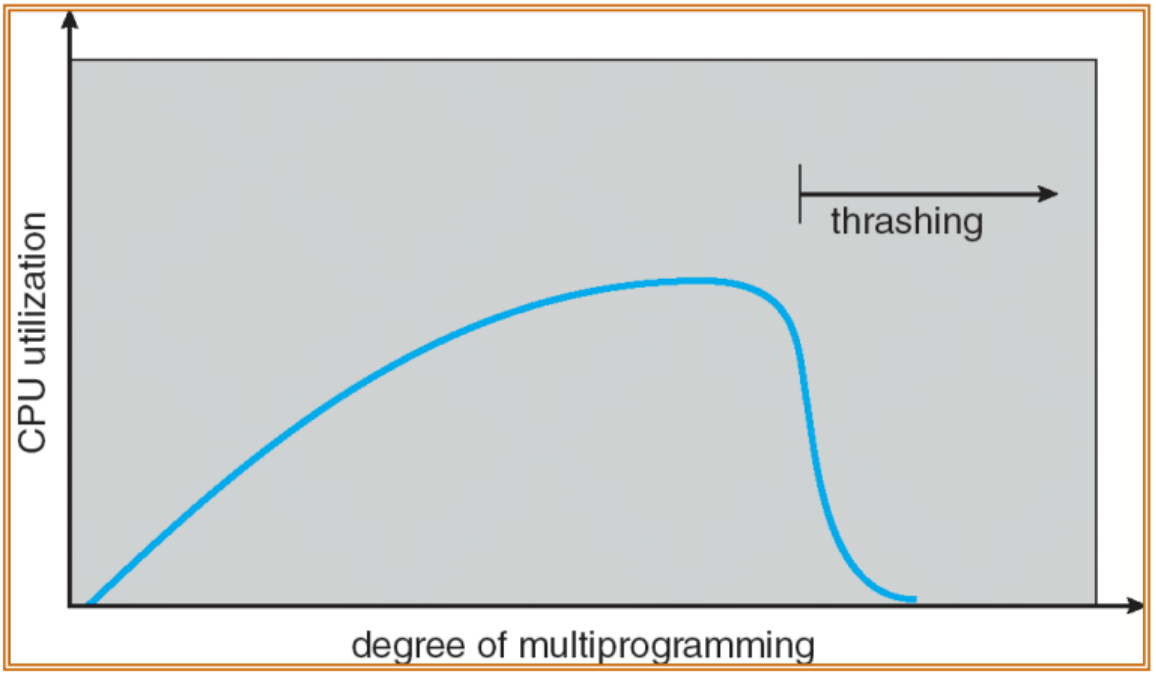
- 我们之前假设不同的内存访问速度都一样，即均匀内存访问 UMA
- 许多系统是非均匀内存访问 NUMA 的，访问不同内存的速度不同



- 现代操作系统会尽可能把进程的物理页分配在本地内存页，即“更近”的内存页
- Solaris 的解决方案：latency group, lgroup
  - 操作系统维护一个数据结构，记录CPU和内存的延迟关系，调度器参照这个结构来决策内存页分配

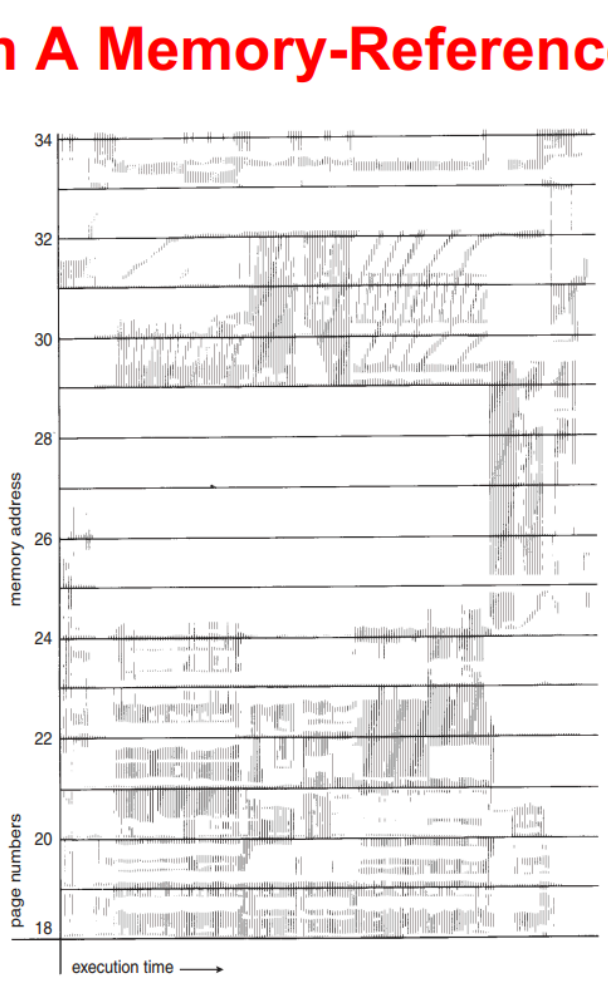
抖动 Thrashing

- 定义：进程不断进行页交换
- 进程没有足够的页，于是一直进行页交换而导致低CPU利用率、高I/O消耗，这种现象叫作抖动 Thrashing
- 进一步，调度器发现CPU利用率降低，于是分配更多任务进入 ready queue，加剧抖动 Thrashing



- 抖动发生的原因：进程的局部性 locality 大小之和 > 总内存大小

局部性 Locality



- 水平条带表示时间局部性（同一页在一段时间内反复使用）
- 竖直条带表示空间局部性（相近的页在一段时间内被一起使用）

工作集模型 Working-Set Model

- 现代操作系统常用的方法
- 用于量化进程当前系统对内存帧数量的需求
- 工作集窗口 working-set window Δ：固定参数，代表最近一段页面引用序列的长度
  - 如果 Δ 太大，窗口太宽，会把进程以前用过而现在不再需要的页面包括进来，造成内存浪费
  - 如果 Δ 太小，窗口太窄，无法覆盖进程运行需要的完整局部性，会导致频繁的缺页中断
- 进程P_i的工作集大小 WSS_i：在最近的 Δ 长度时间窗口内，进程P_i引用不同页面的总数
- 总需求帧数 $D = \sum_i WSS_i$：系统中所有正在运行的进程的工作集大小之和
  - 假设总共有 m 个可用的物理内存帧
  - 如果 $D > m$，说明会发生抖动 Thrashing
  - 如果 $D \leq m$，说明内存充足，系统运行稳定
- 在工程中，记录每一次内存访问对于系统性能的消耗太大了，所以考虑采用定时采样的方法来近似模拟
- 为每一个物理页设计 n 个引用位 Reference bit，当页被访问时，将最近的引用位设置为1
- 设置一个间隔定时器 Interval Timer，每隔固定时间（$\frac{\Delta}{n}$）触发一次中断，检查页在最近这段时间是否被引用过（所有引用位中只要有任何1个是1，就认为最近 Δ 时间被引用过），并把引用位进行循环位移

缺页频率方案 Page-Fault Frequency Scheme, PFF

- 系统为每一个进程设定一个可接受的缺页中断率 page-fault rate 的上下限 Upper/Lower Bound
- 如果进程实际检测到的缺页中断率高于进程可接受的上限，就会分配页给进程
- 如果进程实际检测到的缺页中断率低于可接受的下限，就会从进程中回收页
- 在工程中，进程每次发生缺页中断时，记录和前一次缺页中断的时间间隔

Chapter 10 存储系统 Storage Systems

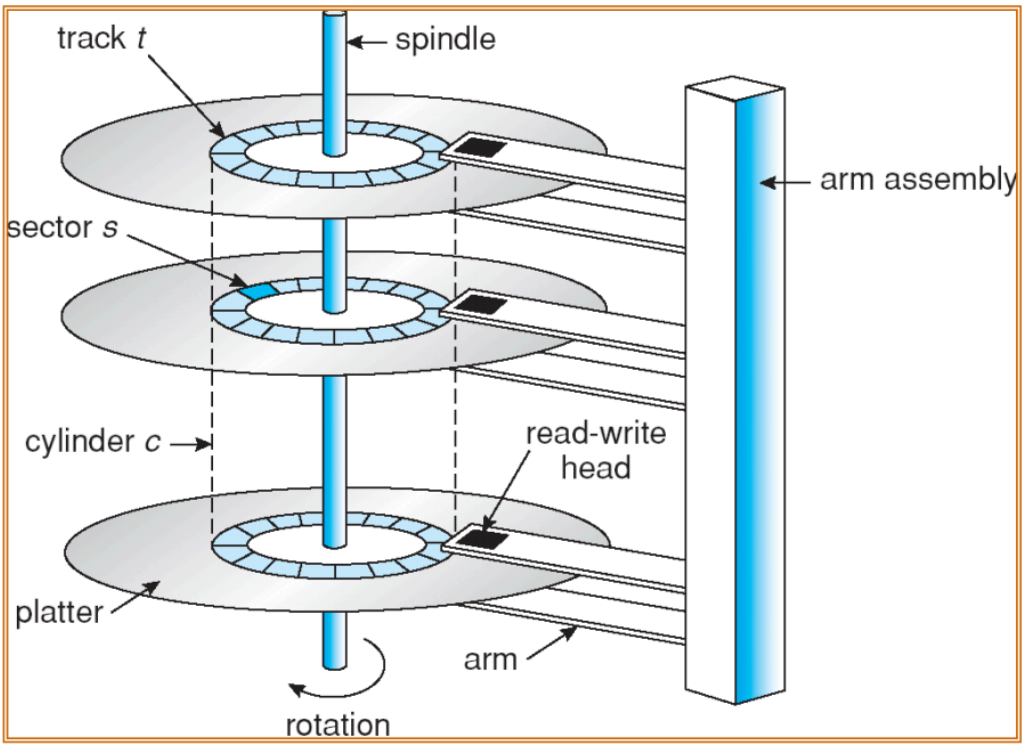
大容量存储结构 Mass Storage Structure

- 现代操作系统的二级存储 secondary storage 由硬盘驱动器 hard disk drives, HDDs 和非易失性存储 nonvolatile memory, NVM 设备组成
- 有些存储设备是可拆卸的

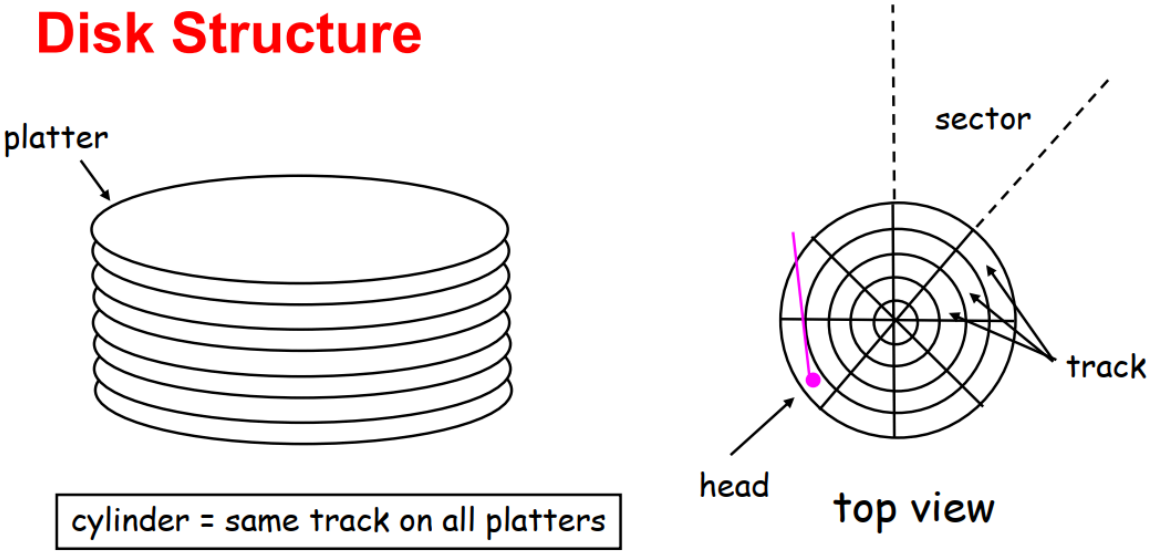
HDD

- HDD 的结构类似于老式唱片机，通过读写头/磁头 read-write head 来读写数据
- 驱动盘片每秒旋转 60-250 次
- 传输速率 Transfer rate 是数据在驱动和电脑之间传输的速率
- 随机访问时间 Positioning time / random-access time 是磁头找到目标数据位置所需的时间，由寻道时间和旋转延迟组成
    - 寻道时间 Seek time：将磁臂移动到正确的磁道（柱面）所需的时间
    - 旋转延迟 Rotational latency：磁头找到正确磁道后，等待盘面旋转到目标扇区出现在磁头下方所需的时间
- 磁头碰撞 Head crash：磁头碰到了盘片表面，会划伤磁性图层，导致硬件损坏，数据永久丢失

Moving-head Disk Mechanism



Disk Structure



Logical addressing: CHS = cylinder, head, sector

- 磁道 track：一个盘片 platter 上的一个圆环
- 扇区 sector：一个磁道上的一个弧段
- 柱面 cylinder：一叠磁道形成的一个柱面
- 在软件设计中，我们会将逻辑地址 Logical addressing 映射到一个一维数组:逻辑块 logical blocks
    - Block 0 是最外侧柱面 cylinder 的第一个磁道 track 的第一个扇区 sector，然后是第二个扇区 sector，存满第一个轨道 track 后是同一个柱面 cylinder 的第二个轨道 track，以此类推

磁盘访问延迟 Disk Access Latency

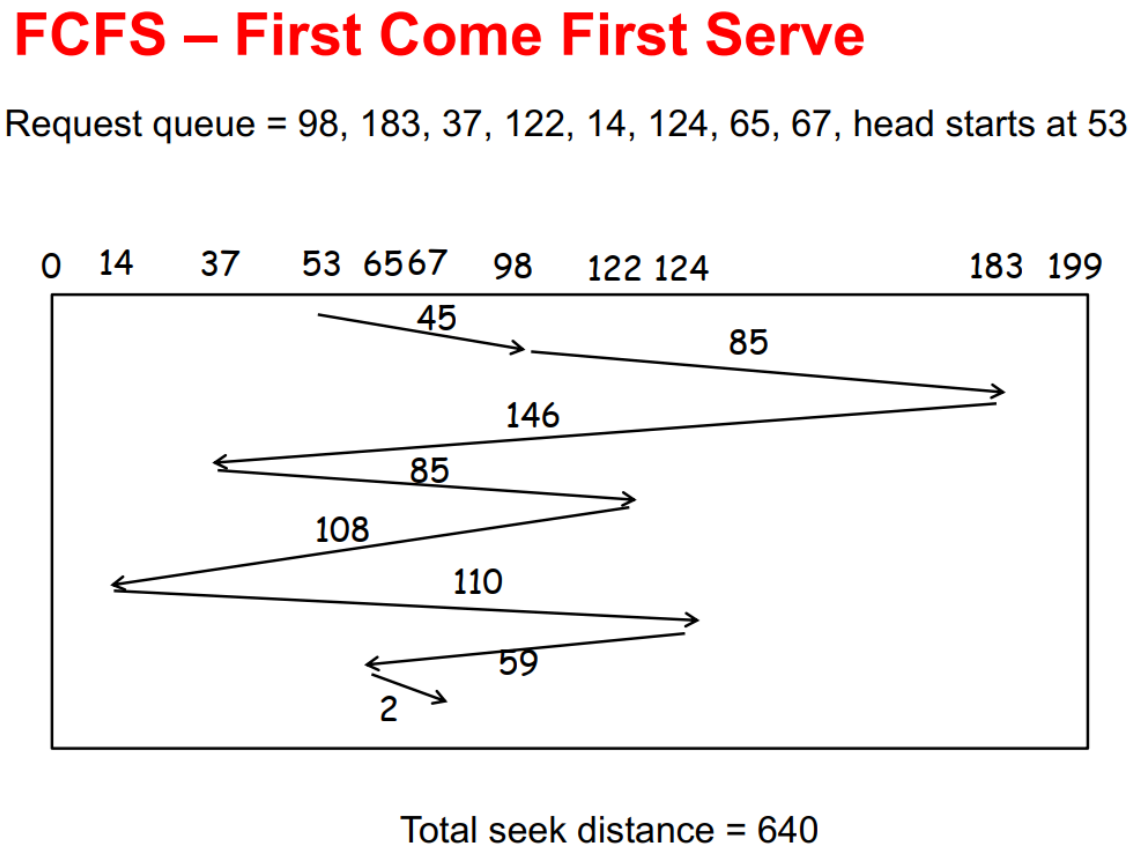
- 寻道时间 Seek time：10~15 ms（延迟的主要来源）
- 旋转延迟 Rotational delay：4.15 ms 对于 7200 rpm（盘片每分钟旋转7200圈）
- 传输速率 Transfer rate：30 MB/s
- 因此，磁盘会按照柱面存储（节省寻道时间），且优先存储外侧柱面（比内侧柱面大）
- 磁盘带宽 Disk bandwidth：总传输的字节数 / 从第一个数据请求，到最后一个数据接收的时间

磁盘调度 Disk Scheduling

- 磁盘至少比内存访问慢了4个数量级，因此是很大的性能瓶颈
- 我们的目标是最小化寻道时间，而寻道时间大约正比于寻道距离 seek distance
- 磁盘调度算法是操作系统的独立模块，允许根据条件选择不同的算法

First Come First Serve (FCFS)

- 对于所有的请求，优先处理先来的请求，按序访问
- 没有进行任何性能优化，性能较低
- 公平性最好

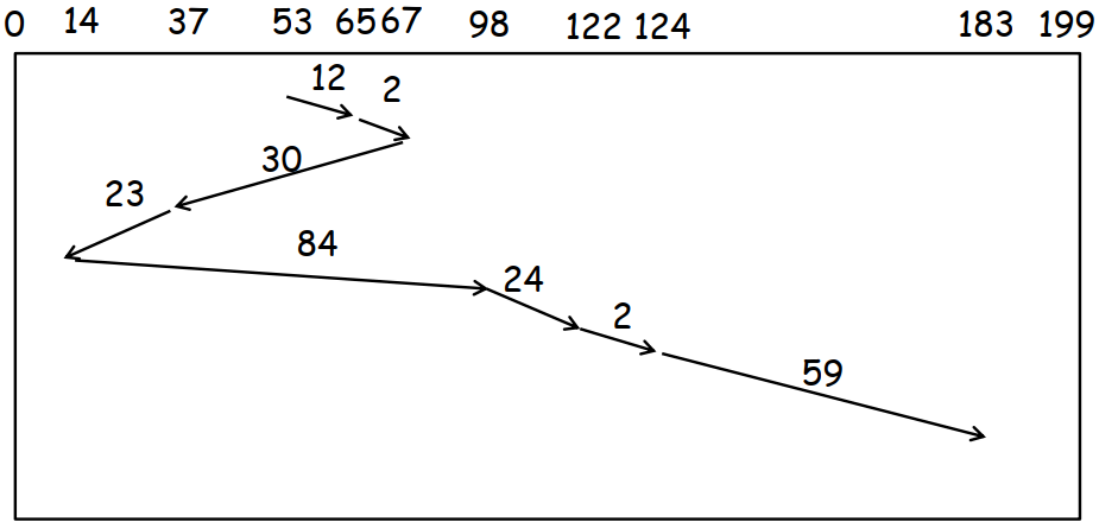


Shortest Seek Time First (SSTF)

- 就近原则，总是优先处理离当前磁头位置最近的请求
- 效率较高
- 可能导致饥饿 Starvation
- 可以作为默认算法，较为常用

SSTF (Shortest Seek Time First)

Request queue = 98, 183, 37, 122, 14, 124, 65, 67, head starts at 53



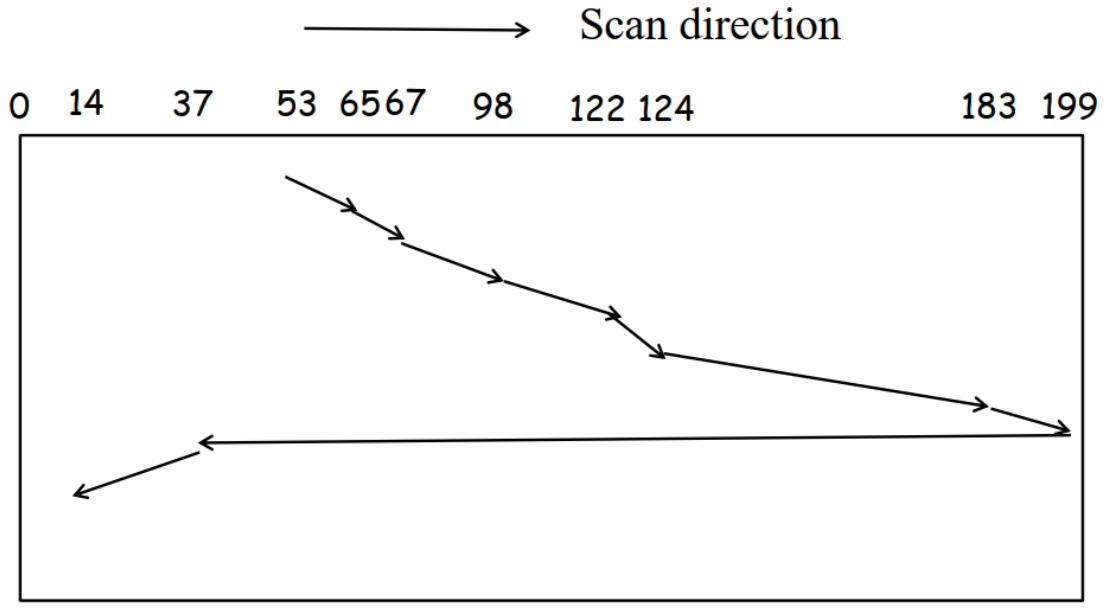
Total seek distance = 236

SCAN / Elevator Algorithm

- 磁头从磁盘的一端移动到另一端，一路处理所有请求；然后再移动回来，一路处理所有请求
- 效率高于 FCFS，公平性强于 SSTF
- 每个请求都有最长等待时间，不会导致饥饿 Starvation
- 在磁盘负载较重的系统中性能较好

SCAN

Request queue = 98, 183, 37, 122, 14, 124, 65, 67, head starts at 53



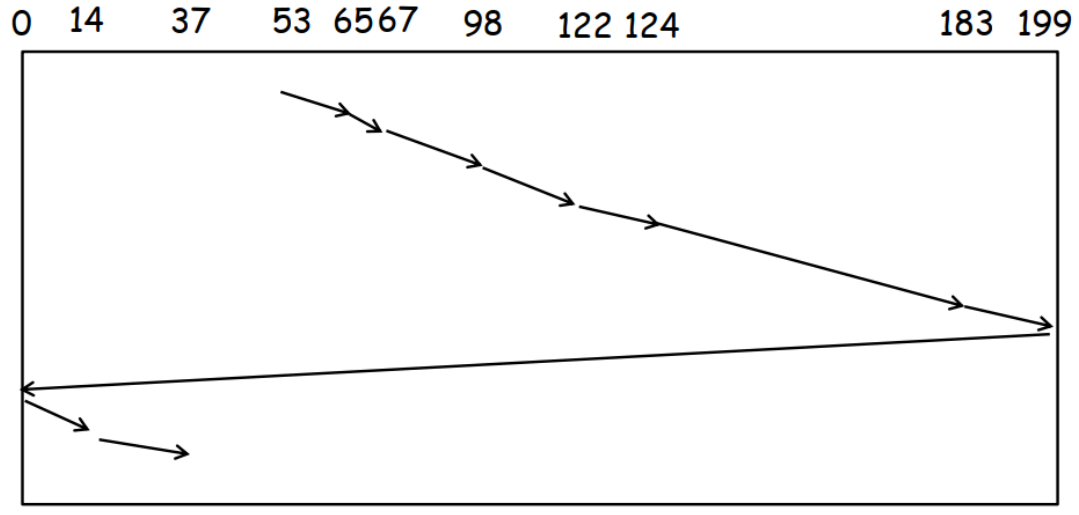
Total seek distance = 331

Circular SCAN (C-SCAN)

- 磁头从磁盘的一端移动到另一端，一路处理所有请求；然后立刻移动回来而不处理请求（相比于处理请求的移动，速度更快），重新开始扫描
- 相比于 SCAN，不会歧视两端附近的请求，更加公平
- 在磁盘负载较重的系统中性能较好

C-SCAN

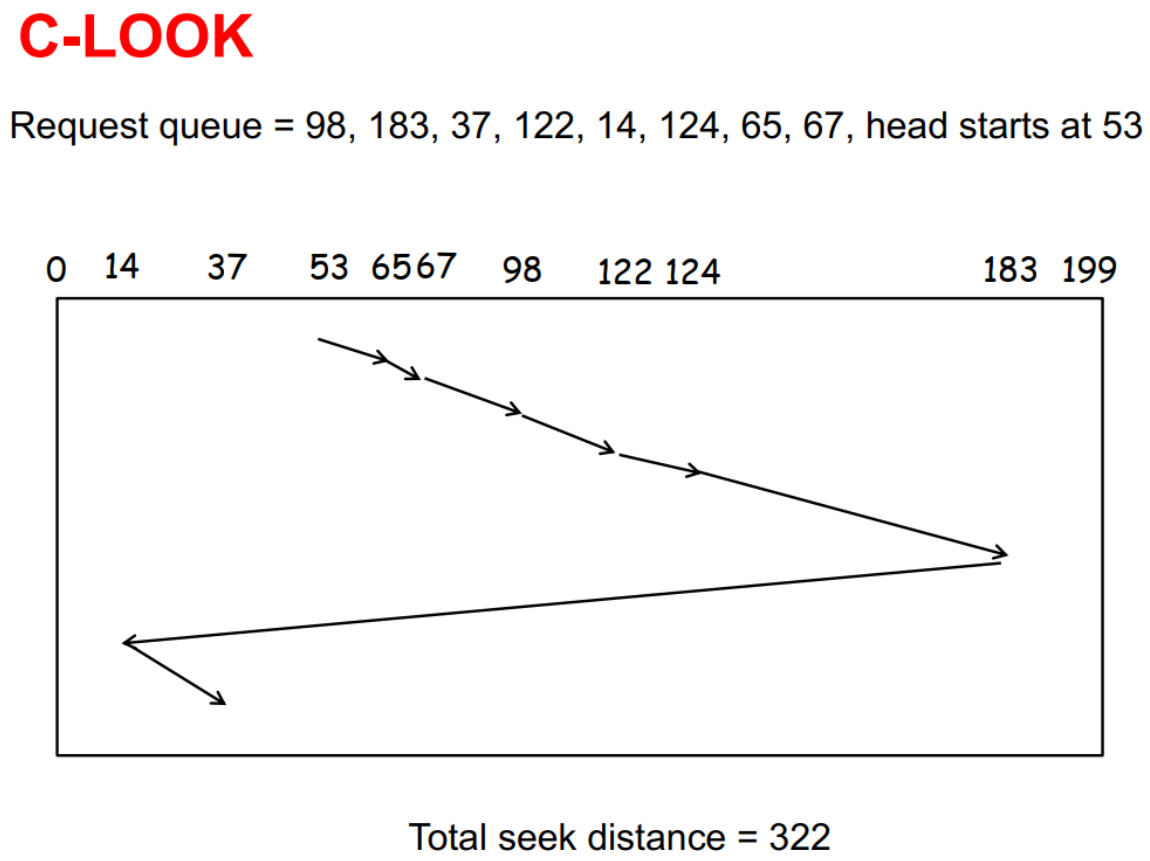
Request queue = 98, 183, 37, 122, 14, 124, 65, 67, head starts at 53



Total seek distance = 383

C-LOOK

- 是C-SCAN的一种优化
- 磁头只移动到一端的最后一个请求处，而不会走到底
- 可以作为默认算法，较为常用



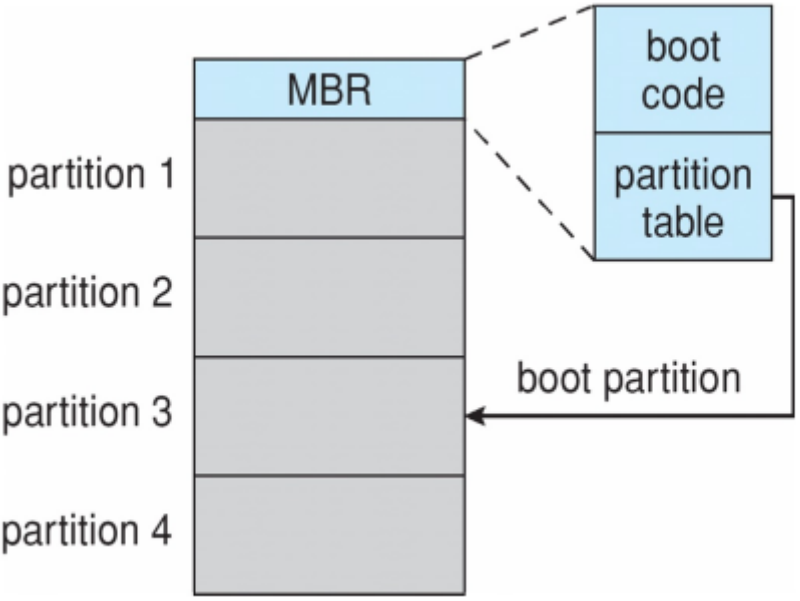
错误检测和纠正 Error Detection and Correction

- 错误检测 Error Detection 是对于磁盘错误的检验，如位翻转 bit flipping 等
- 当发现错误时，可以触发停机 halt
- 可以使用如 checksum, cyclic redundancy check 等校验算法
- 纠错码 Error-correction code, ECC 不仅能检验错误，还能纠正某些错误
- 软错误 Soft error 可以被纠正，硬错误 Hard error 无法被纠正

存储设备管理

- 低级格式化/物理格式化 Low-level formatting / physical formatting：将磁盘划分为可被磁盘控制器读写的扇区
    - 每个扇区可以包含头信息 header information，数据 data 和纠错码 Error-correction code, ECC
    - 每个扇区通常存放512字节的数据，但也可以选择其它大小
- 为了使用磁盘来存储文件，操作系统仍然需要在磁盘上记录自己的数据结构
    - 将磁盘分区 Partition 为一个或多个柱面组，每个组被视为一个逻辑磁盘 logical disk
    - 需要进行逻辑格式化/创建文件系统 Logical formatting / making a file system

    - 为了提高效率，大多数文件系统将块 block 组合成簇 cluster，磁盘 I/O 按块进行，而文件 I/O 按簇进行
- 根分区 Root partition 包含操作系统，在启动时被挂载 Mounted
- 其它分区可以容纳其它操作系统，其它文件系统，或作为原始分区 Raw；可以自动挂载或手动挂载
- 在分区挂载时，会检验文件系统的一致性 file system consistency：所有元数据 metadata 是否正确？若不正确，则试图修复并重试；若正确，则添加到挂载表 mount table，并允许访问
- 引导块 Boot block 可以指向引导卷 boot volume，或指向一组包含从引导文件系统加载内核的引导加载程序 Boot loader / 引导多操作系统的 boot management program
- 提供原始磁盘访问 Raw disk access，使操作系统不干预希望自行管理块的应用程序（如数据库）
- 引导块 Boot block 初始化系统
    - 引导程序 bootstrap 存储在固件 firmware: ROM 中
    - 引导加载程序 bootstrap loader 存储在硬盘中引导分区 boot partition 的引导块 boot block（第一个扇区）中
- 使用扇区备用 sector sparing 等方法来处理坏块



独立磁盘冗余阵列 Redundant Array of Inexpensive Disks, RAID

- 使用多个小容量物理磁盘组合形成一个大容量逻辑磁盘，降低成本
- 通过并行访问，提高吞吐量 throughput
- 可用性更高（冗余存储，故障概率更低）
- 有6个级别的RAID，代表不同的设计方案
- 两大支柱 two RAID aspects
    - 数据条带化 Data striping：提高带宽
    - 数据冗余 Data redundancy：追求安全
- 保护机制：镜像备份 Mirroring, 校验 parity, 其它编码方式 other encoding

数据条带化 Data striping

- 将数据透明地分布在多个磁盘上
- 外观上看起来是一个单一的高速大容量磁盘

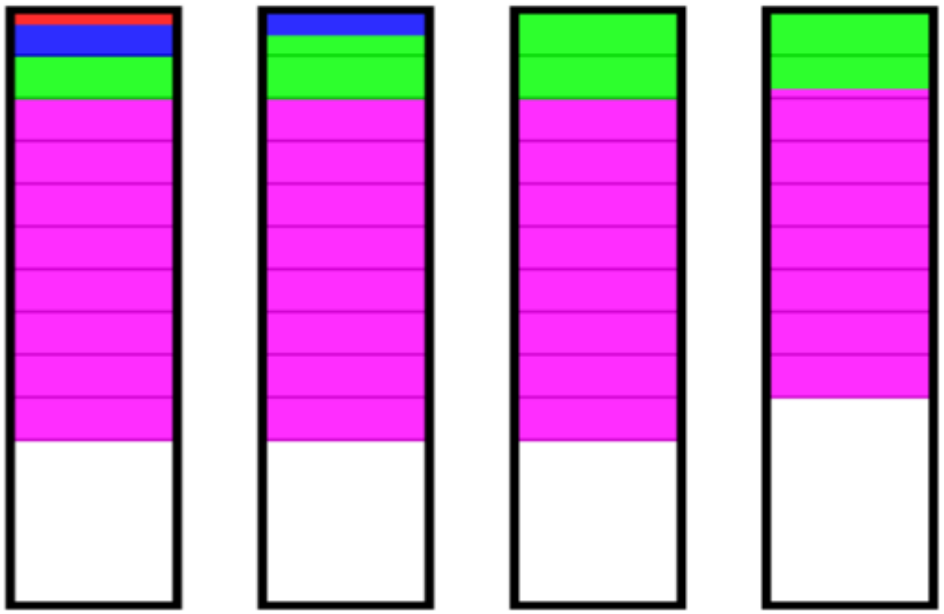
- 允许多个 I/O 操作并行进行
- 数据交错 data interleaving 的粒度 Granularity
    - 细粒度 Fine grained：位交错 bit interleaved
        - 用相对较小的单位，高传输速率
        - I/O 请求会访问所有磁盘阵列中的磁盘
        - 一次只能处理一个逻辑 I/O 请求
        - 所有磁盘在每个请求时都需要花费时间进行定位
    - 粗粒度 Coarse grained：块交错 block-interleaved
        - 用相对较大的单位
        - 小型的 I/O 请求只需要使用少量的磁盘，而只有大型请求会访问到阵列中的全部磁盘

数据冗余 Data redundancy

- 校验冗余信息的方法：奇偶校验 Parity，汉明码 Hamming（可检测单bit的错误，现在少见），里德-所罗门码 Reed-Solomon Codes（可以容忍多磁盘的错误，且能纠正）
- 分布冗余信息的方法
    - 把冗余信息集中存储在少数几块硬盘上：可能造成访问时间瓶颈
    - 把冗余信息分布存储在所有硬盘上：避免热点 hot spots 和其它负载均衡性问题

RAID Level 0

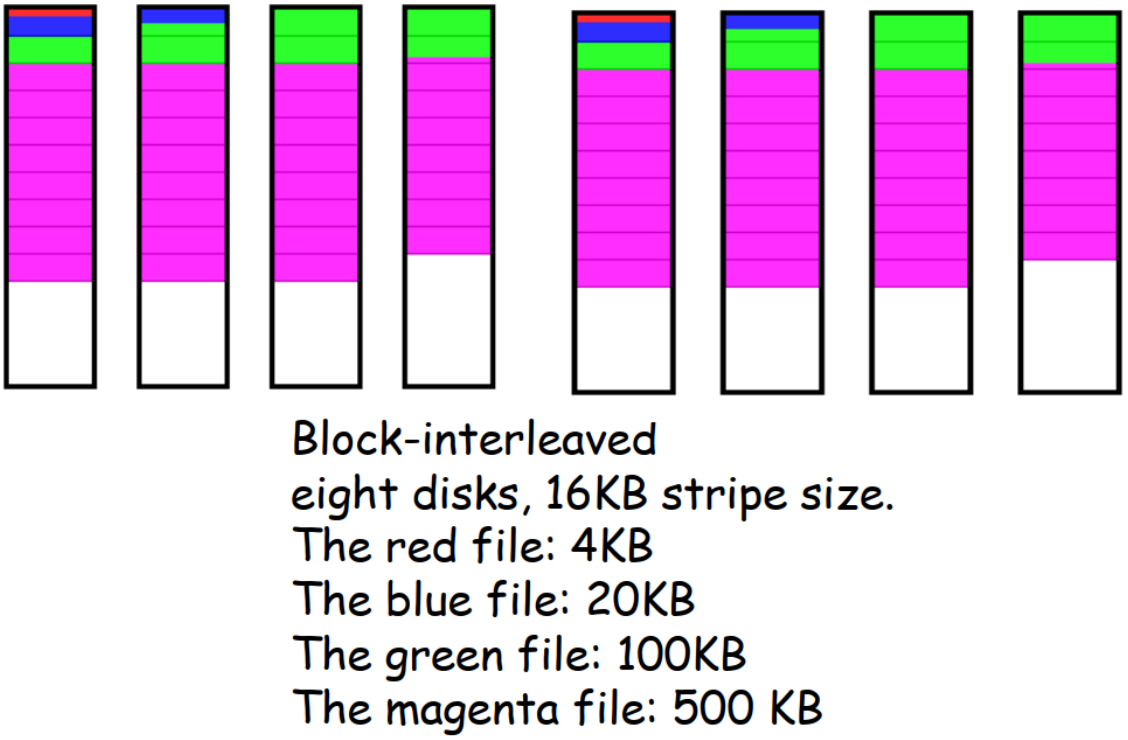
- 最追求高性能，不包括冗余性
- 数据条带化在所有磁盘上，使用 Round-robin 映射到连续的磁盘上
- 由于并行访问，数据传输能力较高，I/O 响应时间较短，性能高
- 由于没有冗余性，只要一块磁盘坏了，数据就会丢失
- 至少 2 块硬盘，空间利用率 100%



Block-interleaved
Four disks, 16KB stripe size.
The red file: 4KB
The blue file: 20KB
The green file: 100KB
The magenta file: 500 KB

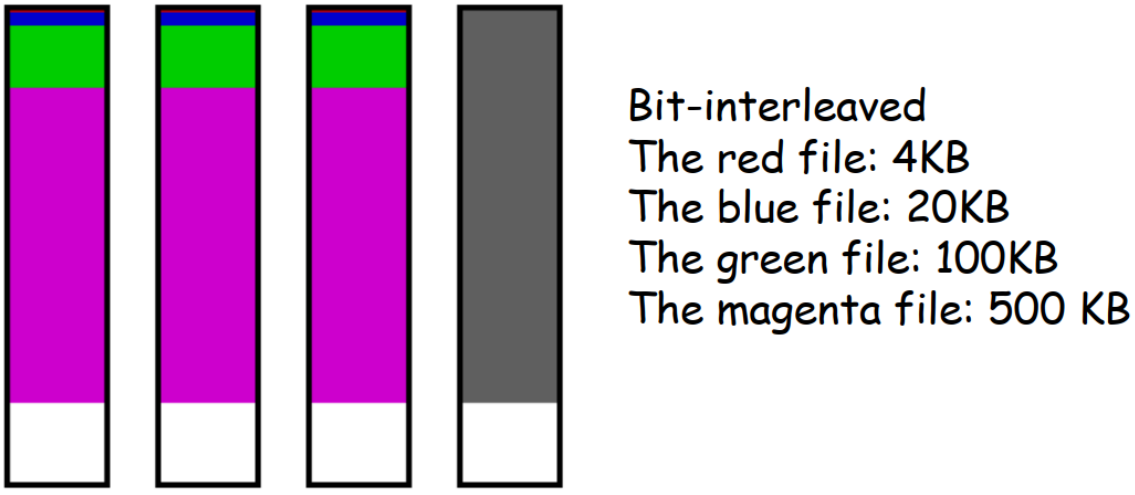
RAID Level 1

- 使用镜像 Mirroring：把所有数据冗余复制一次，即同一份数据在两个磁盘上分别存储
- 写请求需要在镜像的两个存储上同时进行，但由于可以并行执行，所以速度不会慢很多
- 至少2块硬盘，空间利用率 50%



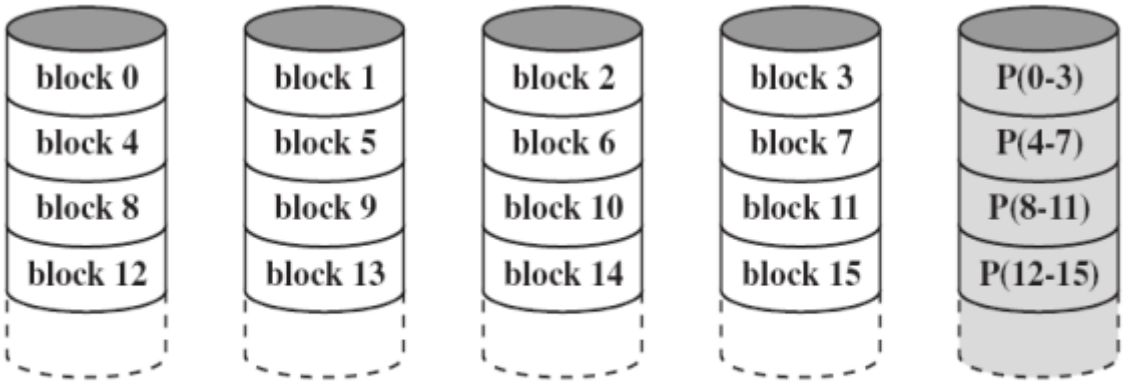
RAID Level 2 & 3

- 使用逐位的数据条带化：位交错 Bit interleaved
- RAID2 采用汉明码进行校验，需要多块专用的校验盘，能发现双位错误，纠正单位错误；比 RAID1 更便宜，但仍然有较大的额外开销；已经极少被使用了
- RAID3 采用简单的奇偶校验，通常只需要一块专用的校验盘，每次只能发现并解决单硬盘故障（异或反算）
- 由于每次请求都需要动用所有硬盘，RAID2&3 在读写大型文件时速度较快（高吞吐量），但事务处理速度较慢（只能一次处理一个读写请求）

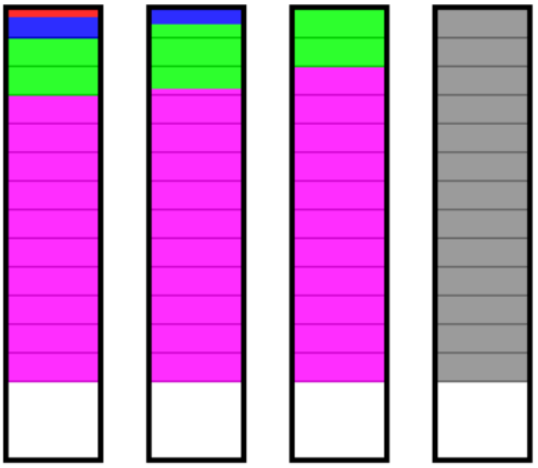


RAID Level 4-6

- 使用逐块的数据条带化：块交错 Block interleaved
- 每个成员盘独立运作，当读写文件小于等于块大小时，只需要读取一块对应硬盘即可，支持并行处理多个小任务
- RAID4 采用了独立的校验盘 Parity disk，在执行写操作时，至少需要2次读+2次写（分别校验+操作），校验盘是性能瓶颈

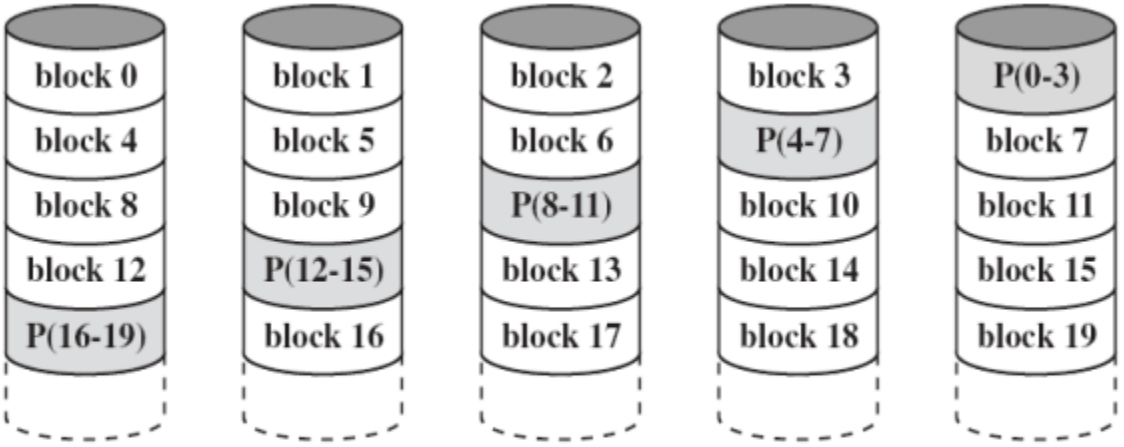


(e) RAID 4 (block-level parity)

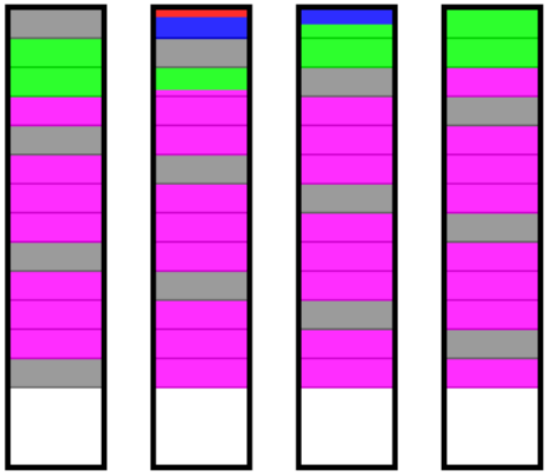


Block-interleaved
The red file: 4KB
The blue file: 20KB
The green file: 100KB
The magenta file: 500 KB

- RAID5 以校验条带的形式将校验位分布存储在各磁盘上；可以发现并纠正单硬盘故障；是“性价比之王”，目前被广泛使用

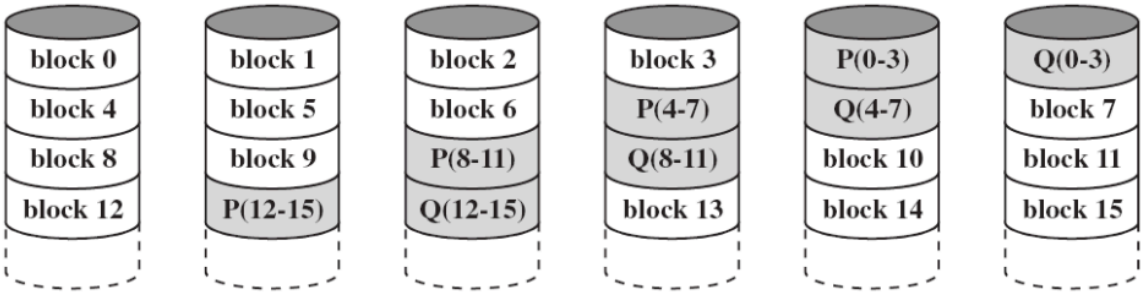


(f) RAID 5 (block-level distributed parity)

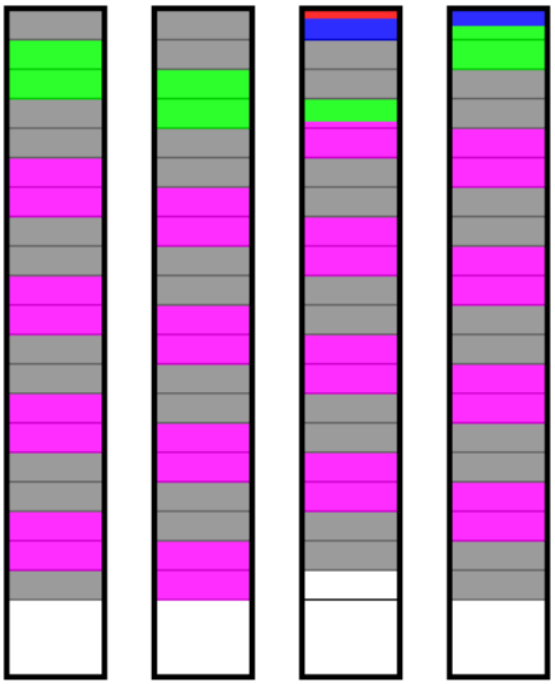


Block-interleaved
The red file: 4KB
The blue file: 20KB
The green file: 100KB
The magenta file: 500 KB

- RAID6 采用了双重校验条带，且分布在各磁盘上；可以发现并纠正双硬盘故障；在大容量（如>20T）硬盘中广泛使用



(g) RAID 6 (dual redundancy)



Block-interleaved
The red file: 4KB
The blue file: 20KB
The green file: 100KB
The magenta file: 500 KB

性能总结

Category	Level	Description	I/O Request Rate (Read/Write)	Data Transfer Rate (Read/Write)	Typical Application
Striping	0	Nonredundant	Large strips: Excellent	Small strips: Excellent	Applications requiring high performance for noncritical data
Mirroring	1	Mirrored	Good/Fair	Fair/Fair	System drives; critical files
Parallel access	2	Redundant via Hamming code	Poor	Excellent	
	3	Bit-interleaved parity	Poor	Excellent	Large I/O request size applications, such as imaging, CAD
Independent access	4	Block-interleaved parity	Excellent/Fair	Fair/Poor	
	5	Block-interleaved distributed parity	Excellent/Fair	Fair/Poor	High request rate, read-intensive, data lookup
	6	Block-interleaved dual distributed parity	Excellent/Poor	Fair/Poor	Applications requiring extremely high availablity

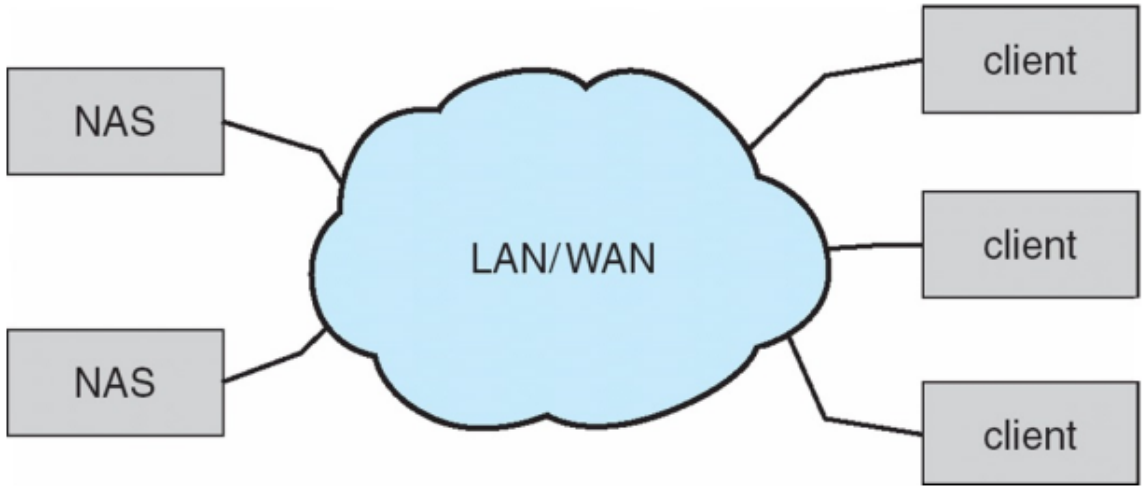
- RAID 1/3/5 是最有研究价值的

磁盘连接 Disk Attachment

- 磁盘常以两种方式连接：
  - 通过 I/O 接口主机连接 Host attached (via an I/O port)：通常主机独占存储，不与其它主机共享
  - 通过网络连接 Network attached (via a network connection)：通常用于集群等的管理共享存储

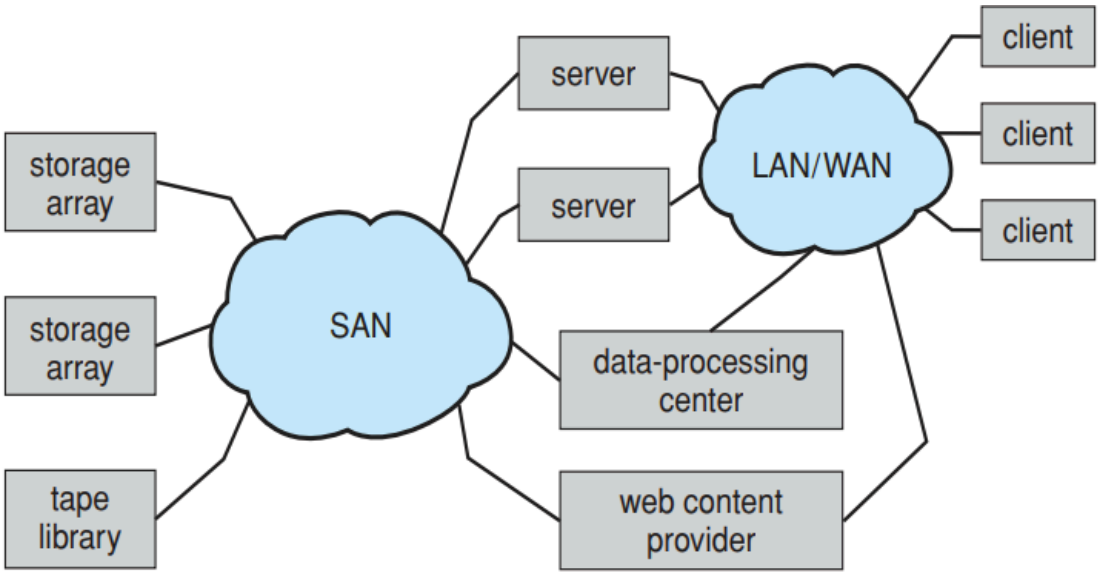
网络附属存储 Network-Attached Storage, NAS

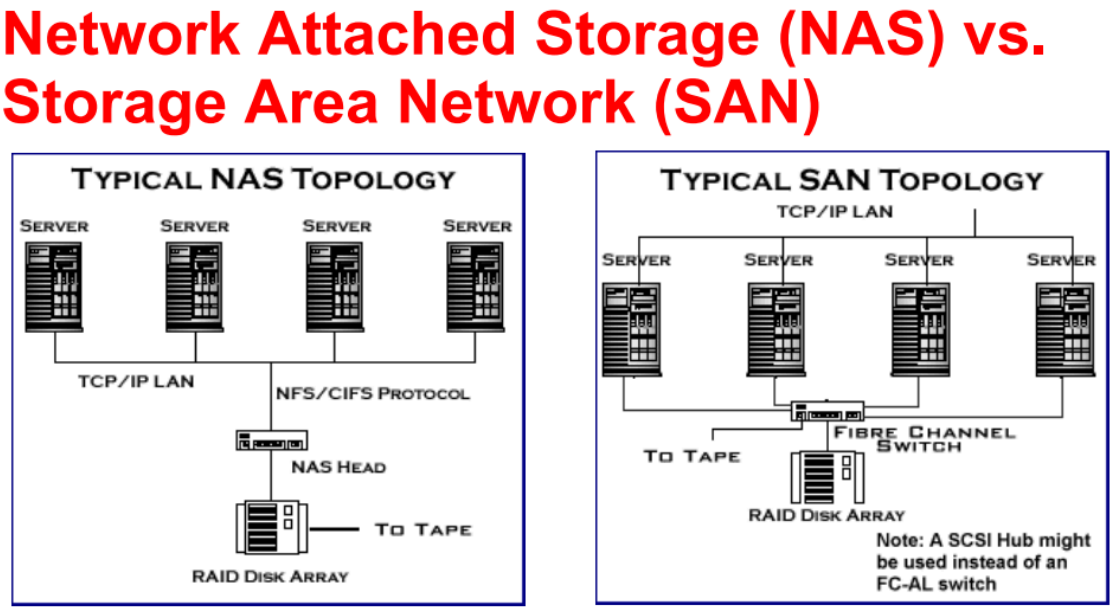
- 通过网络连接（局域网LAN/广域网WAN）而非主机连接，远程接入文件系统
- 常用协议 common protocols：NFS 和 CIFS
- 客户端电脑 Host 和存储设备之间通过远程过程调用 Remote Procedure Calls, RPCs 进行通信
- 在 IP 网络上共享文件
- ISCSI协议允许主机对远端的存储进行管理
- 主要使用 TCP/IP 网络：以太网 / FDDI / ATM 等



存储区域网络 Storage Area Network, SAN

- 常用于大型存储环境，成本较高
- 在专用网络上共享数据块（未格式化的本地硬盘）
- 主要使用光纤通道 Fibre Channel 作为线路，速度快，延迟低
- 使用封装的 SCSI 命令作为协议，服务器像指挥本地硬盘一样指挥远程存储





- The Wires
  - NAS uses TCP/IP Networks: Ethernet, FDDI, ATM
  - SAN uses Fibre Channel
- The Protocols
  - NAS uses TCP/IP and NFS/CIFS/HTTP
  - SAN uses Encapsulated SCSI

	NAS	SAN
Connectivity	Almost any machine	Only server class w/ special ($$) hardware
Data storage & ID	File & Name	Block & Index
Backups & mirrors	By file	By block
General	Simple, limited	Complex, featured

Chapter 11 文件系统 File System

文件的概念

- 文件：相连的逻辑地址空间 Contiguous logical address space
- 文件的基本类型：
  - 数据 Data：数值型 Numeric / 文字型 Character / 二进制 Binary
  - 程序 Program：一组特殊的指令，可以在CPU中执行形成进程

基本文件操作 Basic File Operations

- 打开文件 Open a file
  - `FILE* fopen(const char* filename, const char* mode);`
- 读取文件 Read a file
  - `int fscanf(FILE* stream, const char* format, ...);`
  - `char* fgets(char* str, int num, FILE* stream);`
- 关闭文件 Close a file
  - `int fclose(FILE* stream);`
  - 释放内存资源，同时完成数据更新
- 打开目录 Open a directory
  - `opendir(DIR* drip);`
- 读取目录 Read a directory
  - `struct dirent* readdir(DIR* drip);`
  - 每次返回下一个文件的指针
- 关闭目录 Close a directory
  - `closedir(DIR* drip);`

打开文件 Open Files

对于打开的文件，内核必须维护：

- 一个打开文件表 Open-file table，存储了内核中打开的文件以及操作这些文件的进程

- 文件指针 File pointer，对于每个程序的每个打开的文件，都有独立的文件指针，指向上一次进程读/写的地址
- 文件打开计数器 File-open count：对文件被打开的次数计数，当最后一个进程关闭它时，可以将文件的内存释放
- 文件硬盘地址 Disk location of the file：暂存在内存中，方便下次查找
- 进入权限 Access rights：每个进程对文件的读取/写入/执行权限模式记录

文件锁 File Locking

锁的类型：

- 共享锁 Shared lock：多个进程可以同时读取文件
- 排他锁 Exclusive lock：一个进程写入文件时，不能有其它进程正在读/写

执行模式：

- 强制性锁 Mandatory Locking：操作系统强制执行锁的规则，操作系统会直接拒绝违反规则的访问请求；安全性较高，但效率较低
- 建议性锁 Advisory Locking：操作系统只负责维护锁的状态信息，进程可以查看文件的锁，并自行决定接下来的操作；效率较高，但安全性较低（是Linux / Unix系统常用的方式）
- Java API 文件锁示例：

```
import java.io.*;
import java.nio.channels.*; // 程序与文件中的数据传输管道
public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;
    public static void main(String arsg[]) throws IOException{
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // get the channel for the file
            FileChannel ch = raf.getChannel();
            // the locks of the first half of the file - exclusive
            exclusiveLock = ch.lock(0, raf.lenght() / 2, EXCLUSIVE);
            // Modify the data here
            // Release the lock
            exclusiveLock.release();

            // this locks the second half of the file - shared
            sharedLock = ch.lock(raf.length() / 2 + 1, raf.length(), SHARED);
            // Read the data here
            // Release the lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        } finally {
            if (exclusiveLock != NULL)
                exclusiveLock.release();
            if (sharedLock != NULL)
                sharedLock.release();
        }
```

```
    }
}
```

文件的属性 File Attributes

- 文件名 Name
- 文件类型 Type
- 文件位置 Location：指向文件物理位置的指针
- 文件大小 Size：文件当前的大小
- 文件保护等级 Protection：控制读取 Read / 写入 Write / 执行 Execute 权限
- 时间，日期与用户标识 Time, date and user identification
- 文件的属性存储在目录结构下，在硬盘中保存

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

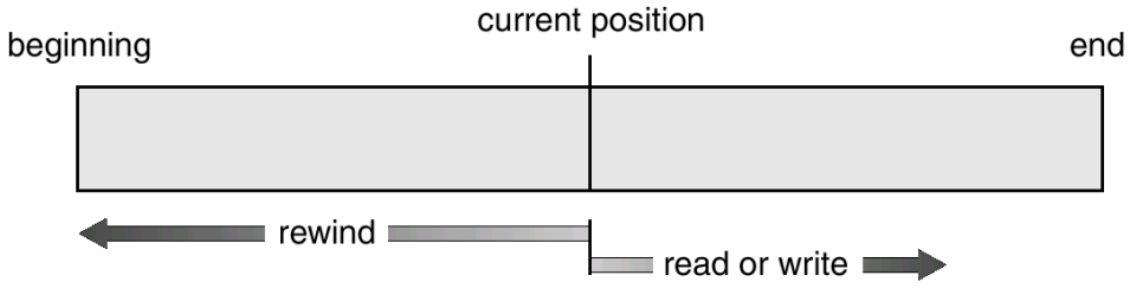
文件的访问方法 Access Methods

- 顺序访问 Sequential Access：每次读/写操作过后，文件指针自动移动到下一个位置
    - read next

    - write next
    - reset
- 直接访问 Direct Access：直接跳到某个位置开始读/写
    - read n（n为从文件开头开始的相对地址）
    - write n
    - position to n
        - read next
        - write next
    - rewrite n

序列访问文件 Sequential-access File

- 是文件系统中最基础的访问模式



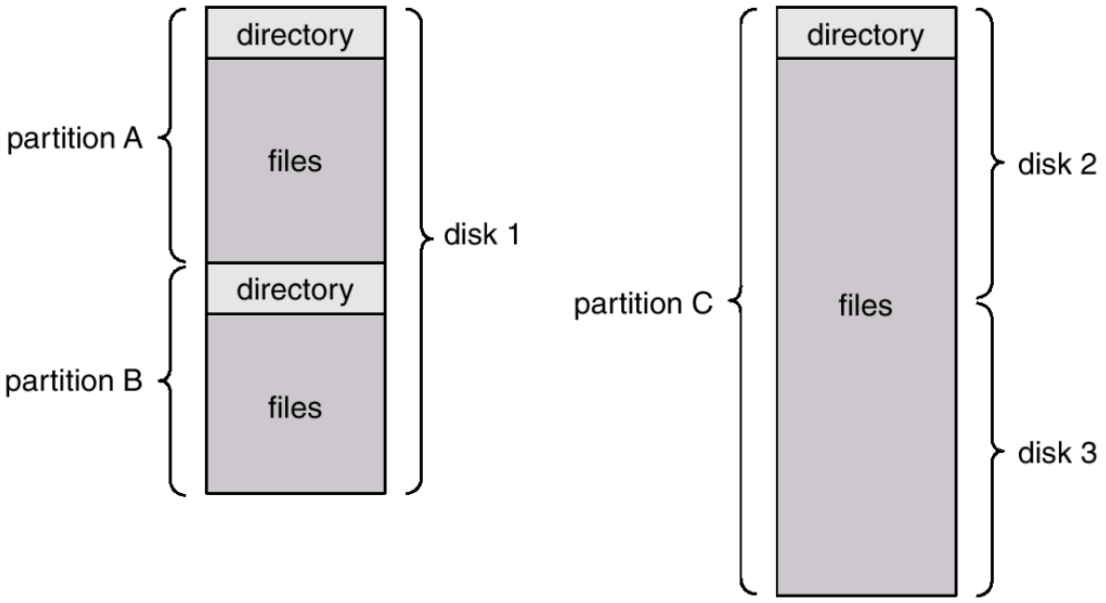
文件目录 File Directories

- 目录包含文件的信息：文件属性 Attributes，文件位置 Location，文件所有者 Ownership
- 目录的本质也是操作系统的文件（一切皆文件）
- 提供了从文件名到文件的映射

磁盘结构 Disk Structure

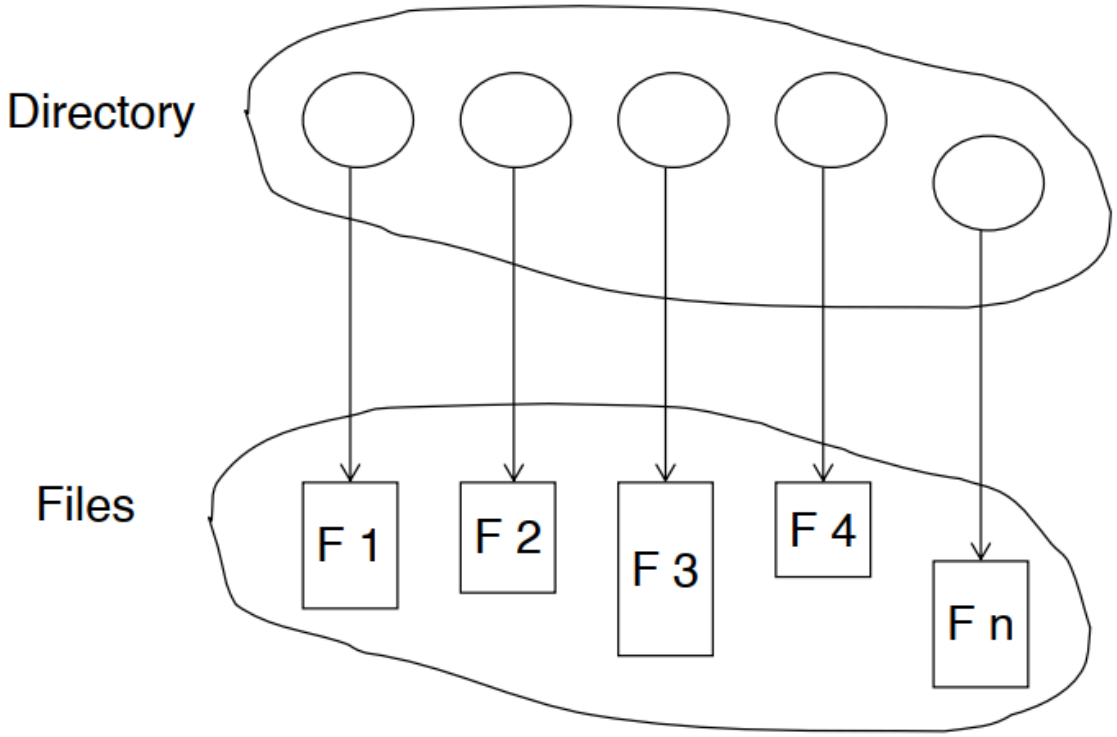
- 硬盘被分为若干个分区 partition / minidisk / slice
- 硬盘或分区可能被 RAID 保护
- 硬盘或分区可能是原始的 raw 裸数据（没有文件系统），也可能是格式化的 formatted（有文件系统）
- 包含文件系统的部分叫做卷 volume；当分区格式化并挂在到操作系统上使用时，它就被称为一个卷
- 每个包含文件系统的卷还会通过设备目录 device directory 或目录表 volume table of contents 来跟踪该文件系统的信息，记录了哪些块是空的，以及每个文件的元数据 Metadata，包含文件的名称、大小和权限
- 除了通用文件系统外，还有专用文件系统，可以存在于同一个操作系统或计算机中

A Typical File-system Organization



目录结构 Directory Structure

- 目录里存储了一系列节点，包括所有文件的信息
- 目录结构和文件都存储在磁盘中；开机后，目录结构会被加载进入内存



- 文件系统一致性校验 File System Consistency Check, FSCK 可以解决“孤儿文件”的问题（只有目录，没有对应实体文件）

目录操作 Operations performed on directory

- 打开一个文件 `fopen()`
- 查找文件
- 创建文件

- 删除文件
- 列举目录
- 重命名文件（只需要改变目录中的文件名信息，不用改变文件）
- 遍历文件系统

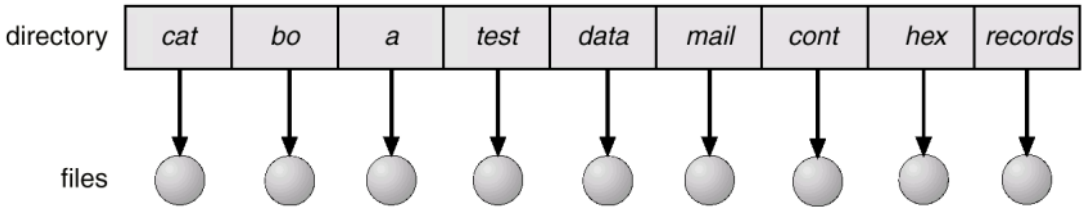
目录结构设计的规划

设计一个文件系统的目录，需要保证：

- 高效性 Efficiency：快速找到文件
- 命名 Naming：需要保证用户友好，不同用户可能有相同名称的文件，相同文件可能有不同的命名
- 分组 Grouping：可以根据性质（用户定的）进行逻辑分组（而非物理地址分组）

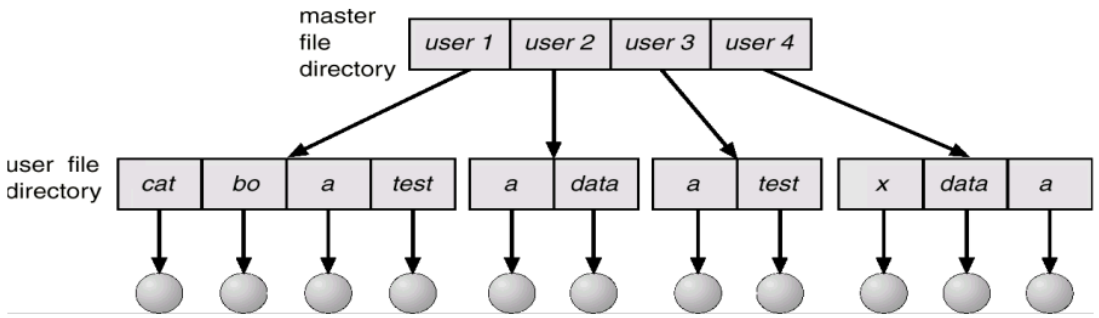
单级目录 Single-Level Directory

- 线性结构，所有用户的所有文件都存储在这个结构中
- 现代操作系统已经不再使用这个结构了
- 问题：
  - 命名问题 naming problem：所有用户的文件不得重名，否则会发生命名冲突
  - 分组问题 grouping problem：难以组织文件夹；在大列表中遍历查找文件，效率较低



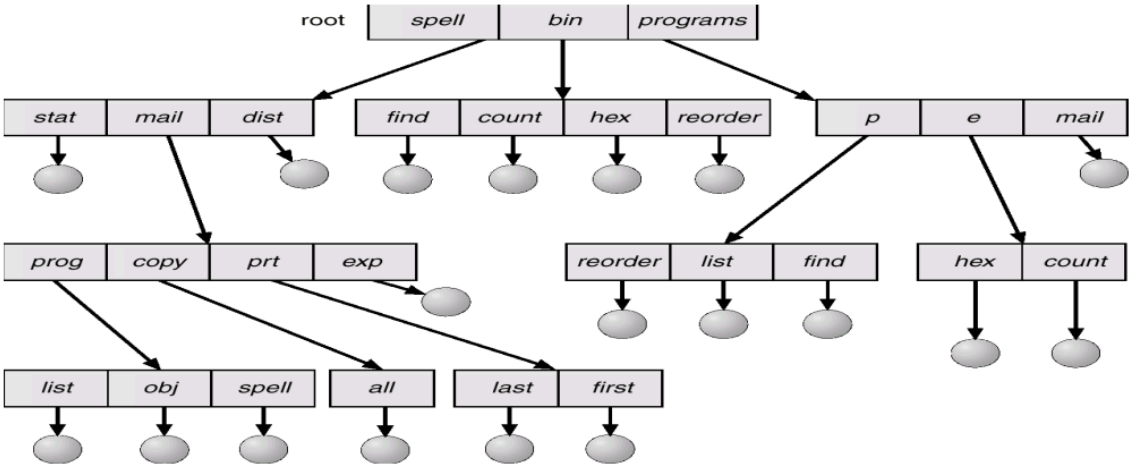
两级目录 Two-Level Directory

- 对每个用户都有独立的单级列表
- 不同用户可以有相同的文件命名
- 查找速度快于单级目录
- 分组问题仍然存在，难以组织文件夹



树状目录 Tree-Structured Directories

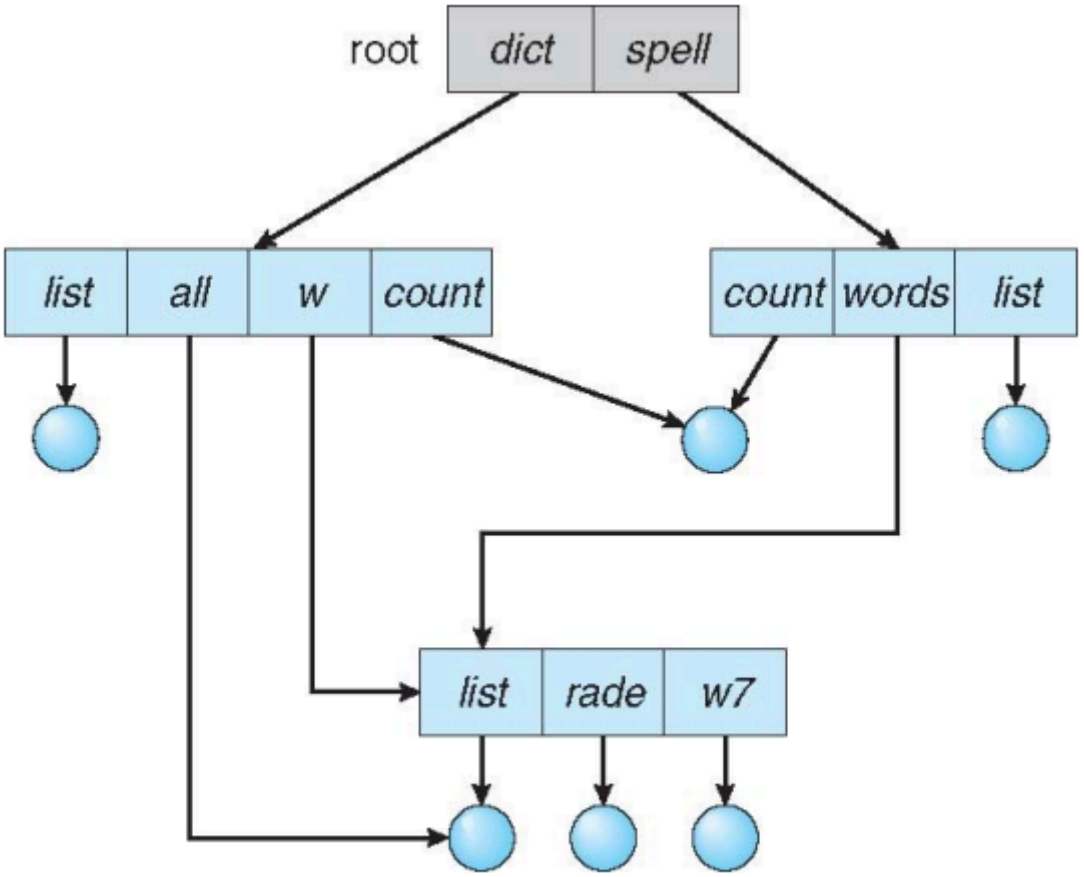
- 可以通过从根目录到文件的完整路径来唯一定位一个文件
- 文件搜索效率很高
- 有分组能力，可以组织文件夹



- 在树状目录中，可以使用绝对路径 Absolute path / 相对路径 Relative path 来定位文件
  - 绝对路径 "/home/jane/homework.html"
  - 相对路径 "./homework.html"
- 支持创建/删除文件、创建/删除子目录指令，其中删除子目录会删除子目录及其所有子目录和文件

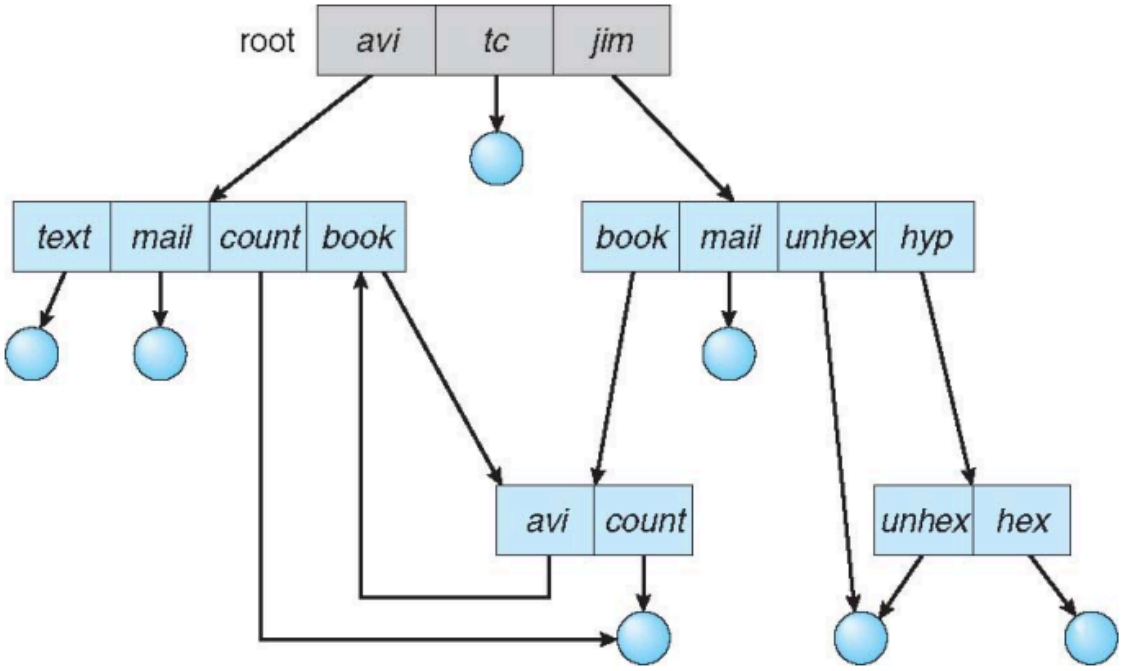
无环图目录 Acyclic-Graph Directories

- 允许文件/目录在目录结构中存在多个父节点
- 同一个文件可能存在多个不同的路径名 aliasing
- 如果一个目录中删除了一个文件，可能导致在另一个目录中的同一个文件出现悬挂指针 dangling pointer
- 增加一个新的目录项类型：链接 Link



通用图目录 General Graph Directory

- 允许文件系统目录结构中含有环
- 一般避免这种情况的发生



文件保护 Protection

核心目标

- 使文件所有者对文件具有绝对的掌控权：谁可以做什么
- 大致分为6种权限类型：
  - Read：读取文件内容
  - Write：写入修改文件
  - Execute：执行文件
  - Append：从文件末尾写入文件
  - Delete：允许从目录结构中删除这个文件，并释放空间
  - List：查看目录下的文件

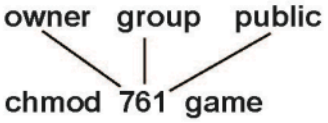
Unix 文件权限设置

Access Lists and Groups in Unix

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a file (say *game*) or subdirectory, define an appropriate access.



- Attach a group to a file

```
chgrp      G      game
```

- `chmod` 设置文件对不同用户/组的权限，`chgrp` 将组关联到某个文件上

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

- 第一列为权限，第一位 `d` 表示是目录，`-` 表示是文件；后续依次表示 owner/group/public权限
- 第二列为该目录关联的硬链接数量
- 第三列为所有者(pbg)
- 第四列为所属组
- 第五列为文件大小，以字节 Byte 为单位

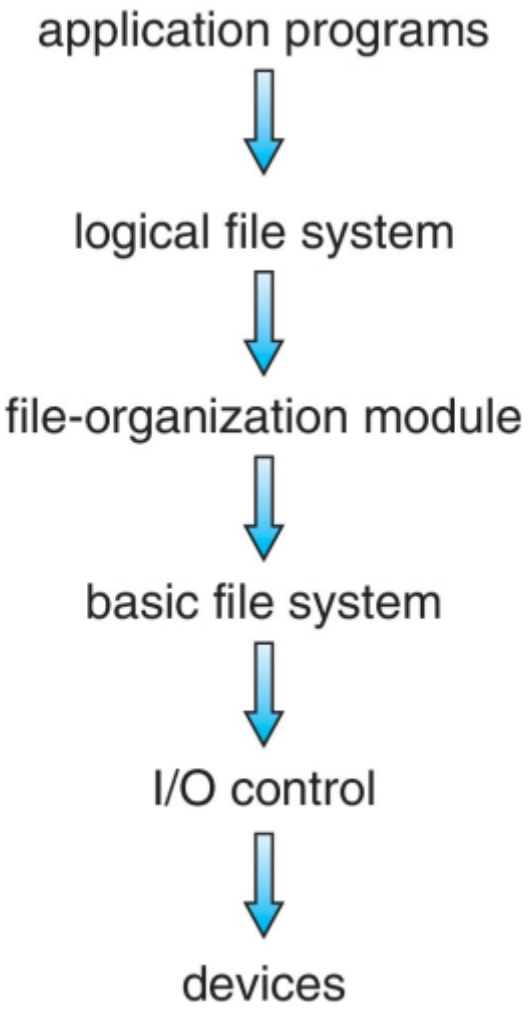
- 第六列为最后修改时间
- 第七列为文件/目录名

文件系统的实现 File System Implementation

文件系统结构 File System Structure

- 文件系统存储在二级存储（一般是硬盘）上，允许数据能被存储、定位和搜索
- 文件控制块 File control block, FCB 存储包含文件信息（元数据 Metadata）的结构
- 设备驱动 Device driver 负责控制物理硬件（硬盘驱动）

分层文件系统 Layered File System



- 设备驱动 Devices Drivers（I/O control 层）负责接收底层指令，并转换为对硬件的控制指令
- 基本文件系统 Basic file system 负责接收通用命令并传递给驱动，也负责管理传输缓冲区 Buffer 和缓存 Cache
- 文件组织模块 File organization module 负责管理从逻辑地址块 logical block 到物理地址块 physical block 的映射，管理空闲空间 free space 和磁盘分配 disk allocation（即新文件放在哪里）
- 逻辑文件系统 Logic file system
    - 将文件名转换为系统识别的文件编号、文件句柄或位置信息
    - 通过维护文件控制块 FCB，管理文件元数据信息 Metadata information
    - 进行目录结构管理

- 检查当前用户读写权限，保护文件

分层文件系统的优缺点

- 降低复杂性，减少冗余
- 每层具体代码实现可以由操作系统设计者自行选择，较为灵活
- 每层都可能产生额外开销 Overhead，可能降低性能

文件系统操作 File System Operations

- 需要有硬盘上的结构和内存中的结构 on-disk and in-memory structures
- 启动控制块 Boot control block 包含
- 柱面控制块 Volume control block / superblock / master file table 记录整个硬盘上关于块的宏观数据：总块数、总空闲块数、块大小、空闲块指针/数组
- 组织文件的目录结构

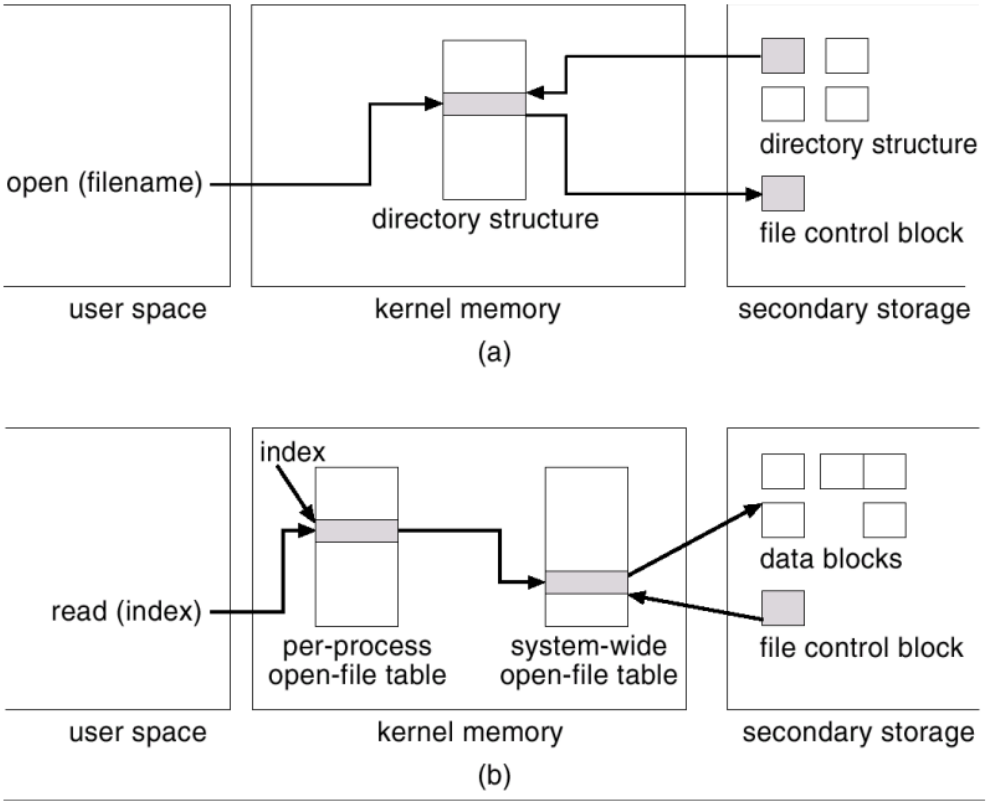
文件控制块 file control block

- 在 Unix 中，一个文件控制块 file control block 被称为一个 i-node
- 包含文件权限 file permissions，文件日期 file dates(create, access, write)，文件所有者 file owner, group, ACL，文件大小 file size，文件数据块 file data blocks

内存文件系统结构 In-memory File System Structures

- 挂载表 Mount table 记录了每个文件系统的类型（识别该分区的格式）、文件系统在整个目录树中的位置、其它信息（如该分区对应的物理设备标识）
- 系统级打开文件表 System-wide open-file table 是一个全局表，存储了当前整个操作系统中被所有进程打开的所有文件，包括文件控制块 FCB 的副本，还包含对每个文件的引用计数 Reference count
- 进程级打开文件表 Per-process open-file table 是每个运行中的进程独立拥有的表，用于管理进程打开的文件，其中包含：
    - 指向系统级打开文件表中对应条目的指针
    - 当前文件偏移量 Current file offset，记录该进程读写文件的到哪个位置
    - 对文件的访问权限
    - 文件描述符 File Descriptor, FD
- 当打开/读取文件时，内存文件系统中会发生的过程如下

(a) opening a file; (b) reading a file;



目录的实现 Directory Implementation

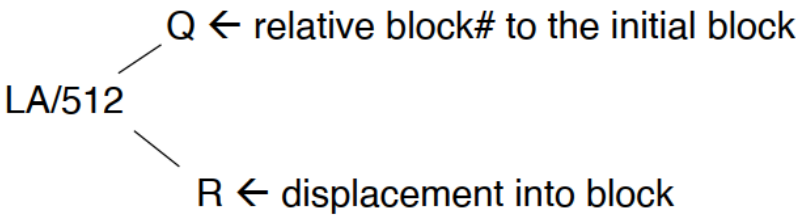
- 可以线性存储文件名与指向数据块的指针，但是文件搜索过慢（线性查找）
- 可以使用哈希表 Hash Table 来存储文件名，保证 $O(1)$ 查找，但是可能发生哈希冲突，也会受限于哈希表的固定大小

分配方法 Allocation Methods

连续分配 Contiguous allocation

- 每个文件必须被分配连续的块
- 存储文件名-起始块编号-长度
- 结构简单
- 支持随机访问 Random access
- 浪费空间（有动态分配问题）
- 文件大小不可增长
- 一般用于只读磁盘
- 逻辑地址到物理硬盘地址的映射：`物理块号 = 逻辑地址 / 块大小 + 文件起始块号` `块内偏移 = 逻辑地址 % 块大小`

- Mapping from logical address (LA) to physical disk address, block size = 512 bytes



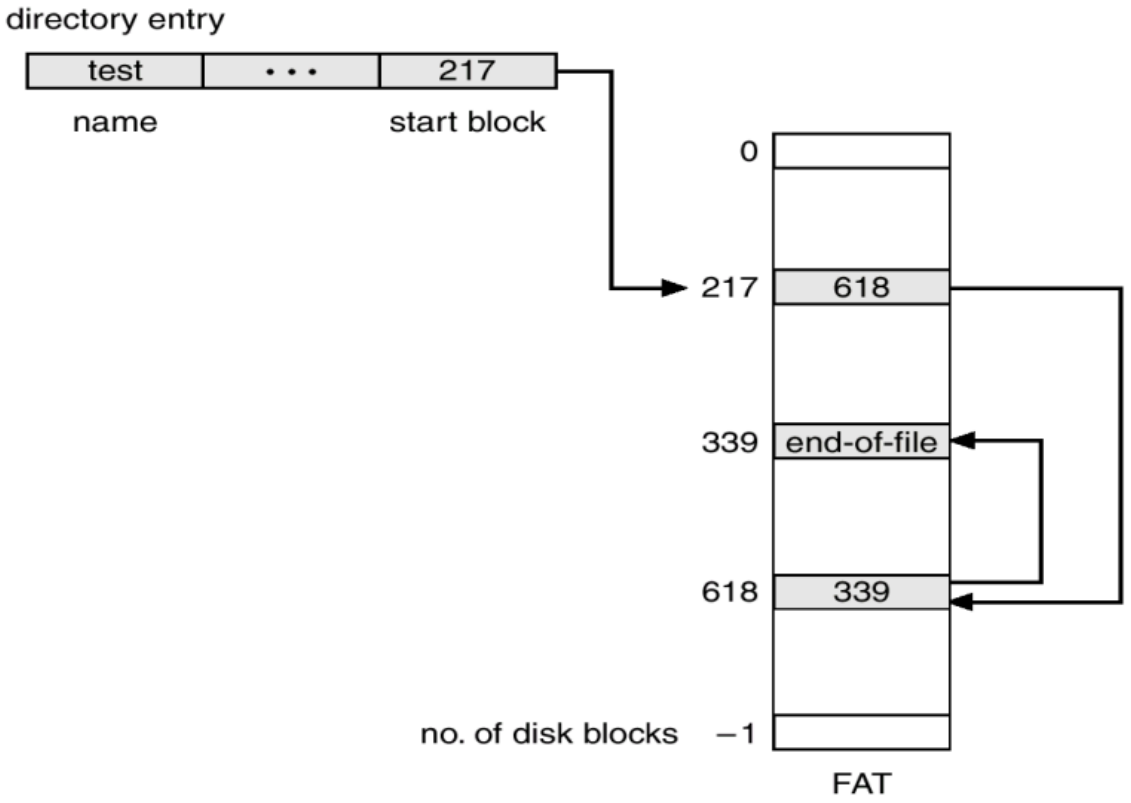
- Block to be accessed = Q + initial block#
- Displacement into block = R

链表分配 Linked allocation

- 每个文件的一个块的开头存放指向下一个块的指针，形成一个链表
- 块不一定连续存储，解决了硬盘空间碎片化的问题
- 结构简单
- 不会浪费硬盘空间
- 不支持随机访问
- 磁盘损坏会造成大面积数据丢失
- 逻辑地址到物理地址的映射：`物理块号 = 链表中的第<逻辑块号>个链表项` `块内偏移 = 4 + 逻辑地址 % 块大小`

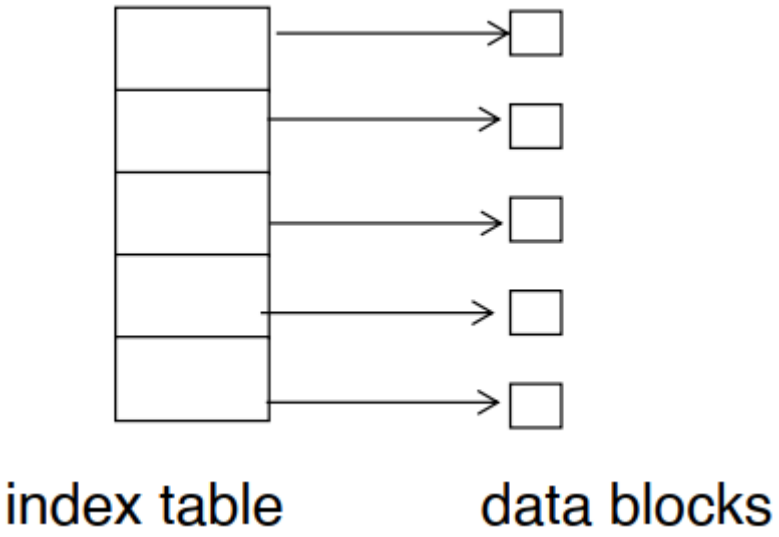
文件分配表 File-Allocation Table, FAT

- 把指向磁盘中的每个块的指针统一写在 FAT 中，即将链表分配中的链表结构独立存储在一个特定的磁盘区域
- 在启动时，FAT将常驻在内存中，以加速访问（随机访问时I/O次数缩减至1）
- 在当时是教科书式方案，但现在已经被取代



索引分配 Indexed allocation

- 为每个文件独立建立一个索引表 index table，按序存放指向每个磁盘块的指针
- 每个文件的索引表存放在每个文件独立的索引块 index block 上
- 存在一个总目录，存放每个文件的索引块地址



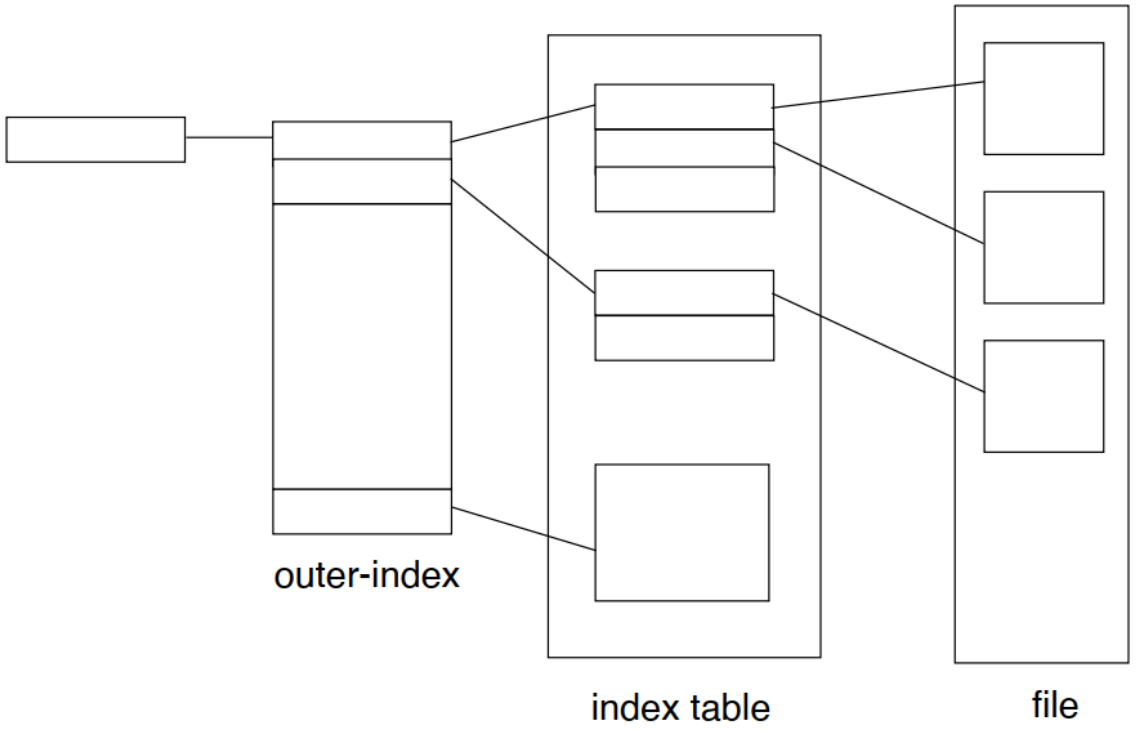
- 支持随机访问
- 有较高的空间利用率
- 需要单独消耗空间存储编号表
- 对单个文件大小有限制
- 逻辑地址到物理地址的映射：`物理块号 = 索引表中的第<逻辑块号>个项` `块内偏移 = 逻辑地址 % 块大小`

链接索引分配 Linked Mapping

- 每个索引块的开头存放指针，指向下一个索引块；每个索引块按序存放指向物理块的指针
- 支持存储大文件，但是需要在索引表上使用链表实现

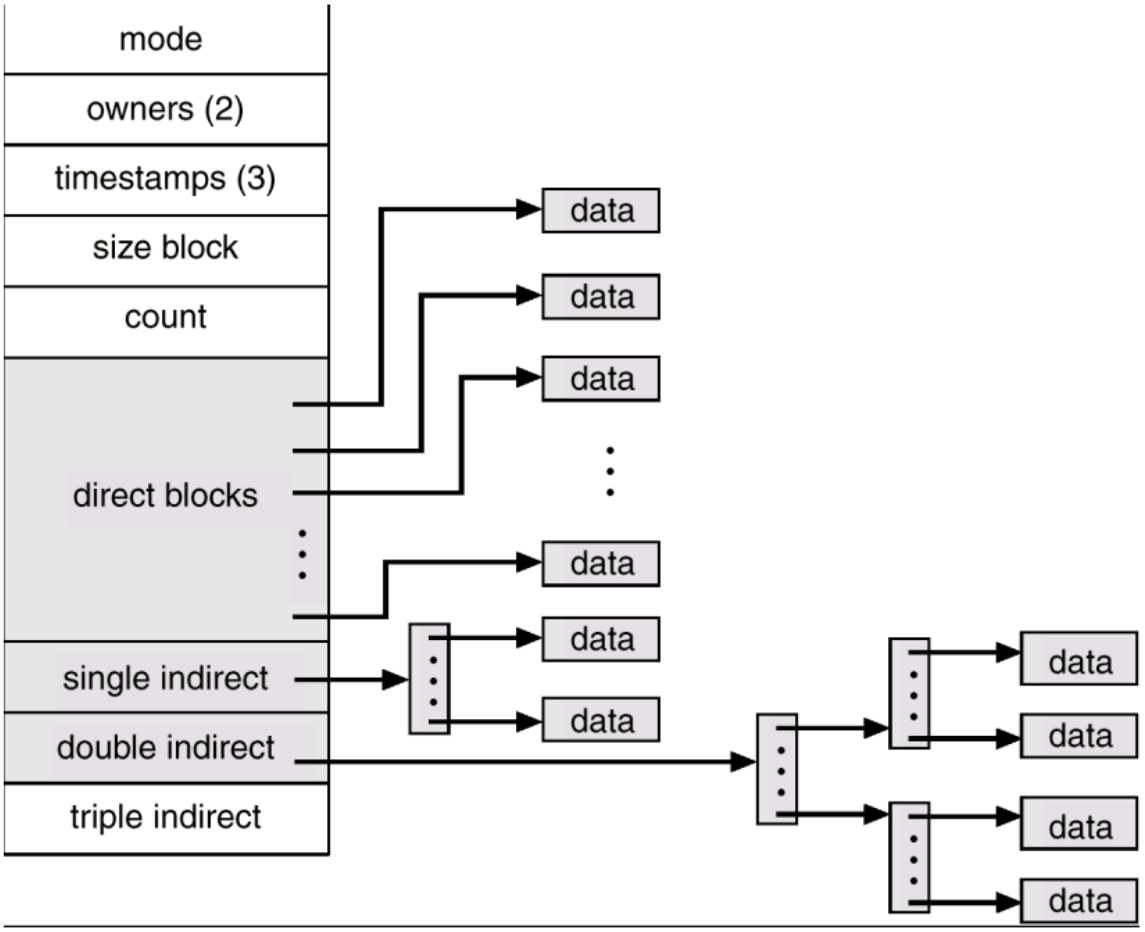
两级索引 Two level index

- 外部索引表按序存放索引块的地址，索引块按序存放指向物理块的指针
- 支持存储大文件，且不需要使用链表实现，支持随机访问
- 每次访问文件，需要进行3次磁盘 I/O，速度较慢



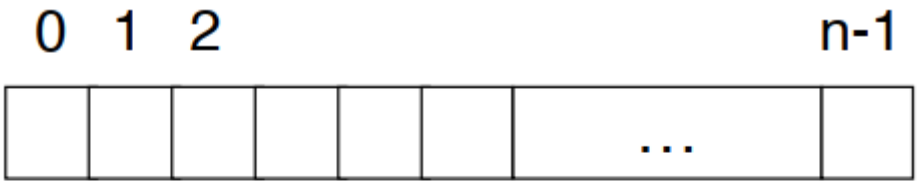
Unix File System (UFS) Allocation Scheme

- 对于大文件和小文件，分别采用不同的存储方式，达到性能的最大利用



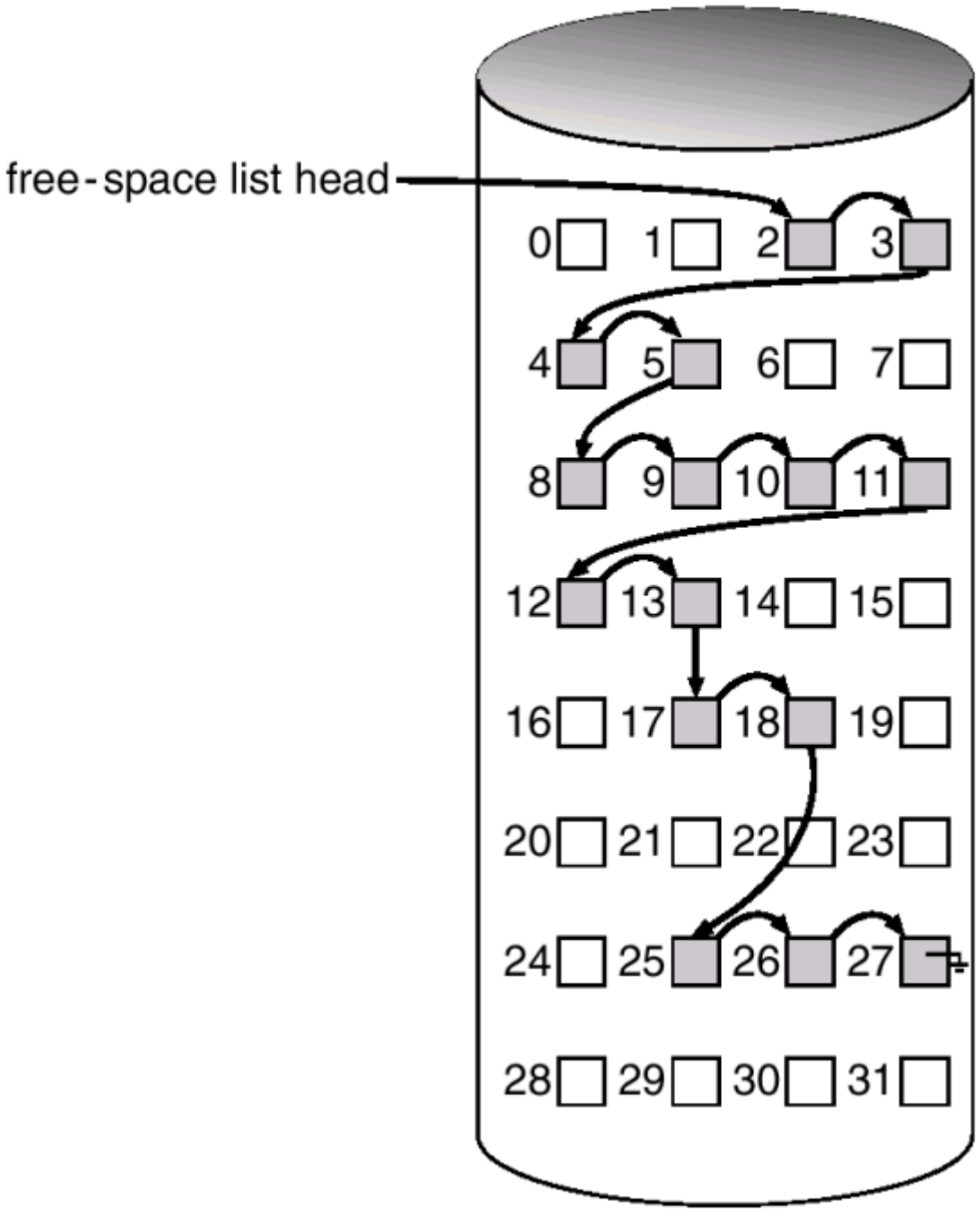
空闲空间管理 Free-Space Management

- Bitmap：对每个块都存储一个是否空闲的状态，合并成一个向量 bit vector



$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

- Linked Free Space List：在磁盘上的每个空闲块 free block 的开头存储下一个 free block 的编号（不维护顺序），在内存中仅储存第一个空闲块的编号
  - 释放产生新的空闲块时，把新的空闲块挂载在链表的头部



日志结构文件系统 Log structured / journaling file systems

- 对文件系统的每一次更新都被记录为一次事务 transaction
- 事务具有原子性
- 所有的事务写入硬盘上的日志 log，写入后视为已提交 committed 状态
- 按照日志，异步完成文件系统的更新，完成后删除对应的日志记录
- 优势：不再需要FSCK等恢复方法，恢复文件速度快

文件系统的性能与效率

效率 Efficiency 影响因素

- 磁盘分配与目录算法 Disk allocation and directory algorithms
- 目录项中保存的数据类型 Types of data kept in file's directory entry

- 元数据结构的预分配/按需分配 Pre-allocation / as-needed allocation of metadata structures：在格式化磁盘时就预先分配好元数据的空间，还是创建新文件时才动态分配元数据空间
- 固定大小/变长数据结构 Fixed-size / varying-size data structures

性能 Performance 优化

- 把数据和元数据存在物理磁盘中相近的位置
- 使用缓冲池 Buffer cache：内存中的专用区域，用于存放最近经常访问的磁盘块内容
- 同步写 Synchronous（在写入磁盘后再反馈写入成功；慢，安全）/ 异步写 Asynchronous（在写入缓冲区后就反馈写入成功，接下来异步完成缓冲区写入操作；较快，断电可能丢数据）
- 丢弃后面与提前读取 Free-behind & read-ahead
- 读取（必须同步操作）通常比写入（可以异步操作）慢

恢复 Recovery

- 使用一致性检查 Consistency checking：比较目录结构中的数据和磁盘上的数据块，试图修复不一致的地方；可能很慢（几小时甚至几天），也可能失败（Linux FSCK工具的原理）
- 使用另一个存储设备进行数据备份

Chapter 12 基础分布式系统概念 Basic Distributed System Concepts

分布式系统 Disributed System

- 互联网/因特网，域名解析系统DNS，Peer to peer，P2P System，Stock Trading System, Cluster, Grid等，都是分布式系统
- 分布式系统是一种实体的集合，其中每个实体都是自动的 autonomous（可以自主控制的），可编程的 programmable，异步的 asynchronous，错误剪枝的 failure-prone
- 设计目标：
    - 可扩展性 Scalability：系统处理不断增长负载的能力
    - 鲁棒性强 Robustness：系统在面临异常情况下的生存能力
    - 用户可访问性 Availability：系统在任何时间都能够正常提供服务的能力
    - 异构性 Heterogeneity：系统处理多样性和差异性设备、协议、结构的能力

分布式系统结构 Distributed System Structures

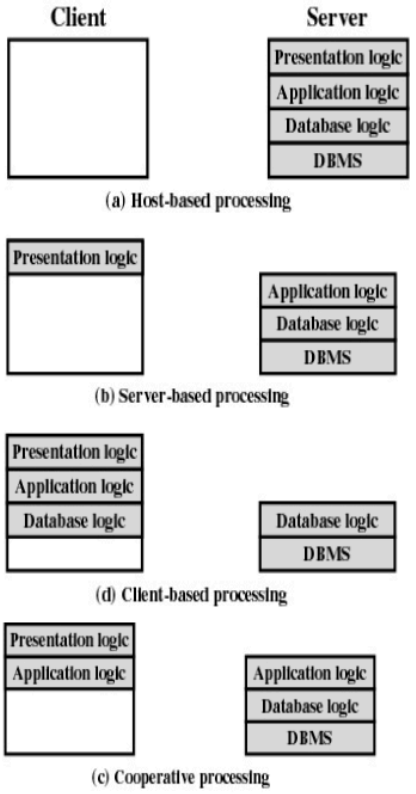
- 客户机-服务器模型 Client/server model：经典结构
- 集中式控制 Centralized control：系统存在一个主节点，负责调度和管理其它节点
- 分布式控制 Decentralized control：没有中心控制点，但是节点协调复杂
- 对等网络 Peer-to-peer, P2P：系统中的每个节点既是客户端也是服务器，每个节点对等
- 事务系统 Transaction System：规模较小，但要求更高的一致性、准确性

客户端/服务器计算 Client/Server Computing

- 客户机提供用户友好的界面，服务器提供数据共享与高性能计算

Classes of Client/Server Applications

1. Host-based processing
   - not true client/server computing
   - traditional mainframe environment
2. Server-based processing
   - Server: all processing
   - User: provides graphical interface
3. Client-based processing
   - Client: all processing data
   - Server: validation /database logic
4. Cooperative processing
   - processing optimized: client/ server
   - complex to set up and maintain

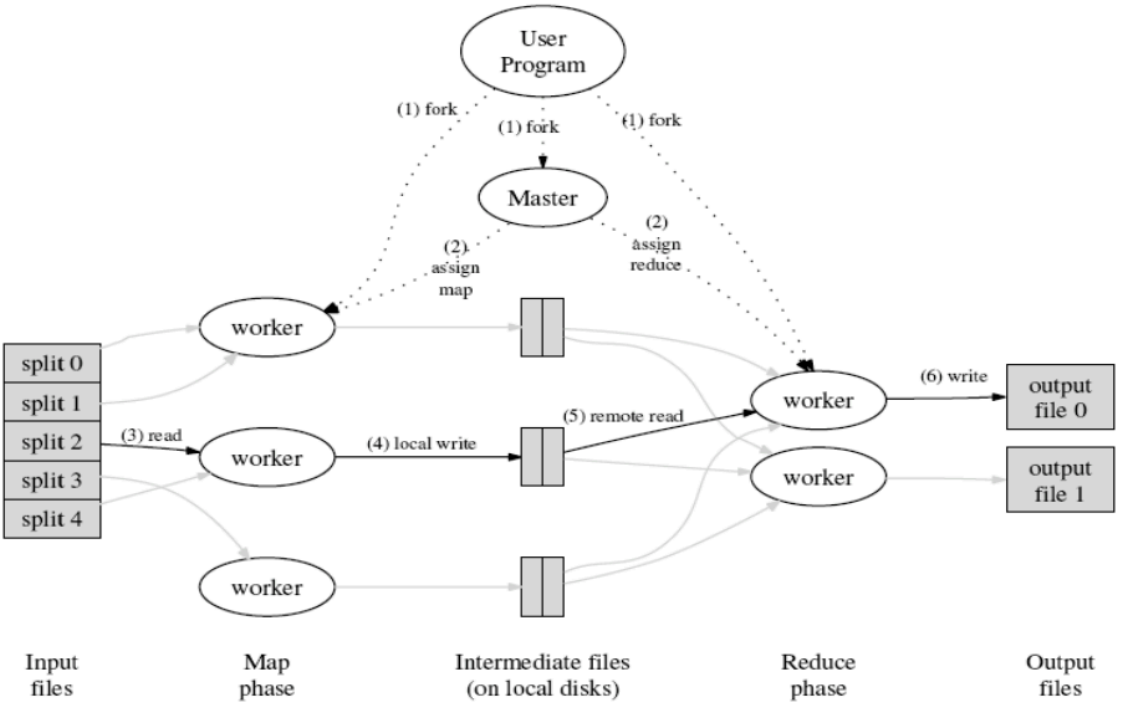


Google 分布式系统架构 Google Distributed System Infrastructure

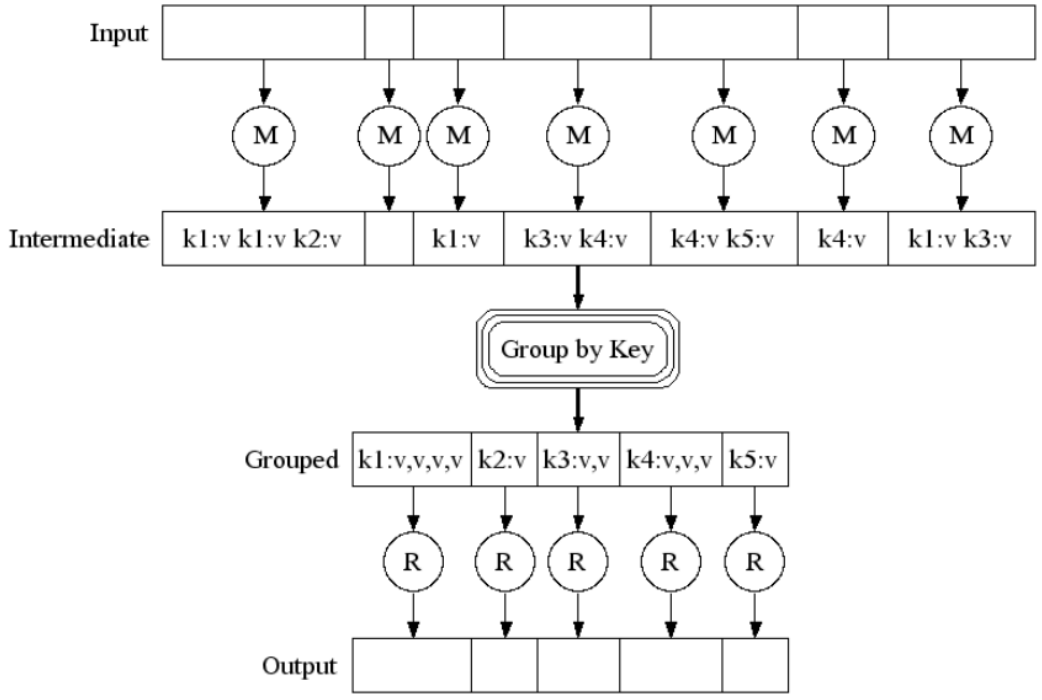
- 底层：廉价的PC硬件、Linux操作系统和物理网络组成的计算平台
- 中层：分布式系统基础设施 Distributed systems infrrastructure
- 顶层：服务与应用，如网页搜索、Gmail、广告系统、地图等

分布式并行编程模型：MapReduce

- MapReduce：一种为了处理和生成大规模数据集的并行编程模型及其实现
- 映射函数 Map：拆分工作，处理一个键值对，产生一组中间键值对
- 归约函数 Reduce：汇总工作，将具有相同中间键的中间值合并
- 这种函数式风格编写的程序会被自动分发，并由普通机器组成的大规模集群上分布式执行
- 使用心跳 Heartbeat 检测故障：当执行出错时，就换一台机器重新执行；仅重新执行已完成/进行中的Map任务，以及进行中的Reduce任务（已完成的Reduce任务一般会分布式存储，存在备份机制，即使机器故障，也可以从其它地方读取结果）



Execution



例子

- 统计一个网址中，指定单词出现的次数
- Map(key=url, val=contents)：对于每个单词，输出 emit 一个 `(w, "1")` 表示单词 `w` 出现了1次
- Reduce(key=word, values=uniq_counts)：对于每个输出，计算它们次数之和，然后输出键值对 `(word, sum)`

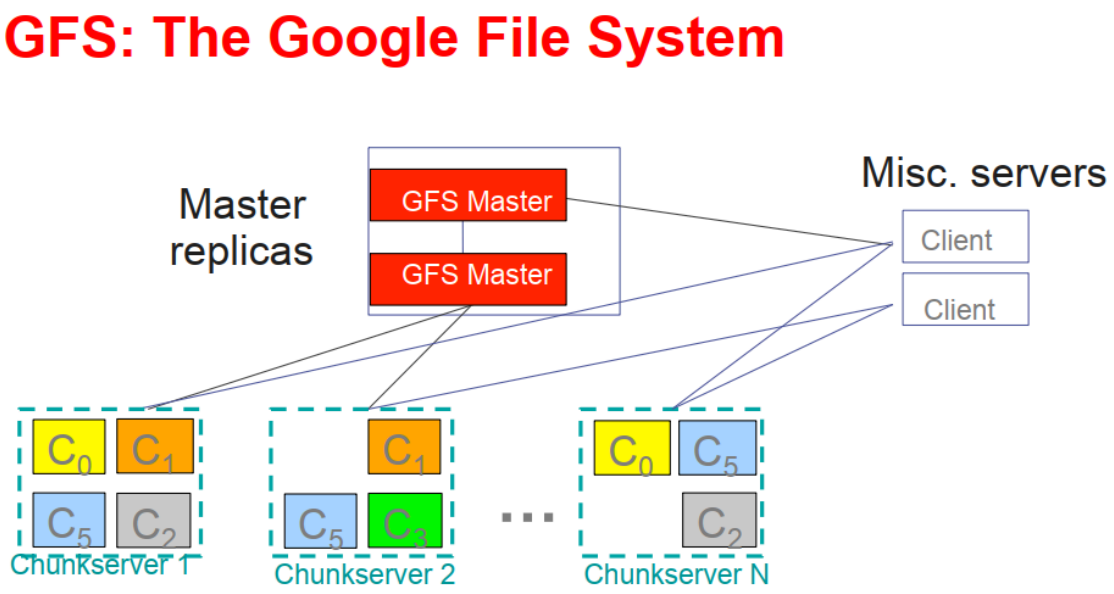
逆向网页链接图

- 一个比较重要的网站，应该有很多其它网站链接到它，因此统计被引用的数量
- Map：对于每个链接到目标网站的链接，输出 <target, source> 对

- Reduce：合并所有来源的 URL，输出 <target, list(source)> 对

GFS：Google File System

- 设计目标：一个可靠的、能处理PB级别数据的分布式存储系统
- 将大文件切分为固定大小的块 Chunks，每块大小 64MB，分散存储在各普通机器的磁盘上
- 副本 Replication：每个数据通常复制3份存储在不同的机器上
- 系统架构：1个Master服务器，多个数据节点 Chunkservers，多个客户端 Clients
- Master只负责管理元数据，实际上的数据传输发生在客户端和数据节点之间



- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety

大表 BigTable

- 一个用于管理结构化数据的分布式存储系统，建立在GFS之上
- 被Google的60多个项目使用
- 使用 `(row, column, timestamp)` 作为索引定位数据
- 许多格子并没有数据，因此面向列存储，避免浪费空间
- 行键 row key 可以是任意字符串，对同一行数据的读写是原子性的（全成功/全失败）
- 行按照字典序排序，相邻的行存储在同一台/少数几台机器上，便于检索
- 与传统数据库的区别：
  - 不支持关系模型：它不是标准的SQL数据库
  - 没有表级的一致性约束
  - 不支持跨行事务：你不能在一个操作中保证同时更新两个不同的行