**概述**

操作系统作为用户和硬件之间的中间人。

操作系统的目标：用户需求：操作系统需要容易学容易用，可靠性高，速度快。系统需求：OS必须易于设计实现维护，灵活可靠错误少高效。

计算机系统分为四大部分：硬件、OS、应用程序和用户。

操作系统时用户与计算机硬件之间的接口。操作系统提供的接口有两类：命令级接口，提供键盘或鼠标命令。程序级接口：提供系统调用。

操作系统是计算机系统资源的管理者。OS是资源分配者是一个控制程序。

OS是扩充裸机的第一层系统软件。

**大型机系统Mainframe System**

**批处理系统, Batch Processing Systems**

Jobs 在内存或者外存里，内存始终有一个 job 在运行，操作系统负责在结束后加载下一个开始运行（我们将加载到内存并运行的程序为 进程, process）。

批处理系统的逻辑简单，弊端也非常明显：当进程执行 I/O 时，CPU 会停下来等待事件完成。I/O 事件有可能是和显示器等设备交互，但更耗时的实际上是等待用户进行输入。在这样的系统里，大多数时间浪费在了等用户输入这件事情上。

**多道批处理系统**

Multiprogramming Batch Systems。在批处理系统基础上，当当前 job 发生 I/O 时，操作系统负责让 CPU 转而运行另一个 job。

多道批处理系统通过这种方式避免了批处理系统等待用户输入等 I/O 事件的 CPU 时间浪费，但是这样的系统仍然对用户不够友好。例如，我们假设视频播放器播放视频需要一个单独的进程，那么如果用户希望同时开两个视频播放器分别播放一个视频，在多道批处理系统中就是不可能实现的，因为在没有 I/O 的情况下 CPU 始终只在执行一个 job，因此只有一个视频能被播放。

**分时系统, Time Sharing Systems**

将 CPU 时间划分为很小的时间片，操作系统负责安排各个 job 轮流运行。

这个设计解决了「两个视频播放器」的问题：由于切换频率很高，远高于人的反应频率，因此可以快速在两个播放进程之间切换，从而让用户看起来「以为」两个视频是同时播放的。

分时系统本身也是一种多道 (multiprogramming) 系统，即允许多个 job 并发 (concurrently) 执行。但是不是批处理 (batch) 系统。

**集群系统Clustered：**一组互联主机构成的统一的计算机资源。有对称集群，多个节点跑程序，互相监视。非对称：一台机器处于热备用模式。集群用于高性能计算。

**实时系统Real Time**有软实时 硬实时。用于工业控制、显示设备、医学影像、科学实验

手持Handheld：PDA cellular telephone

嵌入式.

**操作系统市场格局**，三大体系：Unix服务器 Win桌面 Android 手机

**特权指令：**用户程序不能直接使用，如IO 时钟设置 寄存器设置，系统调用不是特权指令

双模：用户态：执行应用程序时。内核态：执行操作系统程序时。

**仅内核态能做的**：Set value of the system timer, Clear memory, Turn off interrupts, Modify entries in device-status table, Access I/O device

**Interrupt：Caused by externel events.**

**Exceptions: Caused by executing instructions.**

**操作系统结构**

操作系统服务：

用户界面：CLI GUI

程序执行 IO操作 文件系统操作 通信 错误检测 资源分配 统计 保护和安全

操作系统的用户界面

操作系统接口：命令接口和程序接口(系统调用)；命令接口：CLI GUI；程序接口：系统调用指OS提供的服务。

系统调用是进程和OS内核间的程序接口。

一般用C/C++写，大多数由程序提供的叫做API应用程序接口，而不是直接的系统调用。三个常见的API是win32 API，POSIX API 和JAVA API。

Time()是一个系统调用



系统调用有三种传参方式：寄存器传参 参数存在内存的块和表中，将块和表的地址通过寄存器传递，linux和solaris用这种 参数通过堆栈传递。

**系统调用提供的功能:** 进程控制, 文件管理, 设备管理, 信息维护, 通信, 保护

ELF: .text:代码, .rodata:已经初始化的read only数据, .data:已经初始化的数据, bss:未初始化的静态/全局变量在bss段

动态链接： loader链接ELF与lib，.interp段

Running binary：sys\_execve -> do\_execve ->...->load\_elf\_binary->start\_thread，从entry point address作为用户程序起始地址。

静态连接：\_start地址在execve调用后立刻执行；动态链接：先执行ld.so

操作系统结构：

简单结构

MSDOS，小、简单功能有限的系统，没有划分为模块，接口和功能层次没有很好的划分

原始UNIX：受到硬件功能限制，原始UNIX结构受限，分为两部分：系统程序和内核。

UNIX LINUX**单内核结构(Monolithic)**

**层次结构**：OS被划分为很多层，最底层0层是硬件，最高层是用户接口。通过模块化，选择层，使得每个层使用较低层的功能和服务。

**微内核结构microkernel system**

只有最基本的功能直接由微内核实现，其他功能都委托给独立进程。也就是由两大部分组成：微内核和若干服务。好处：利于拓展、容易移植到另一种硬件平台设计。更加可靠（内核态运行的代码更少了），更安全。缺点：用户空间和内核空间的通信开销很大。

**单/宏内核monolithic kernel**

与微内核相反，内核的全部代码，包括子系统都打包到一个文件中。更加简单更加普遍的OS体系。优点：组件之间直接通讯，开销小；缺点：很难避免源代码错误 很难修改和维持；内核越来越大。

**模块modules**

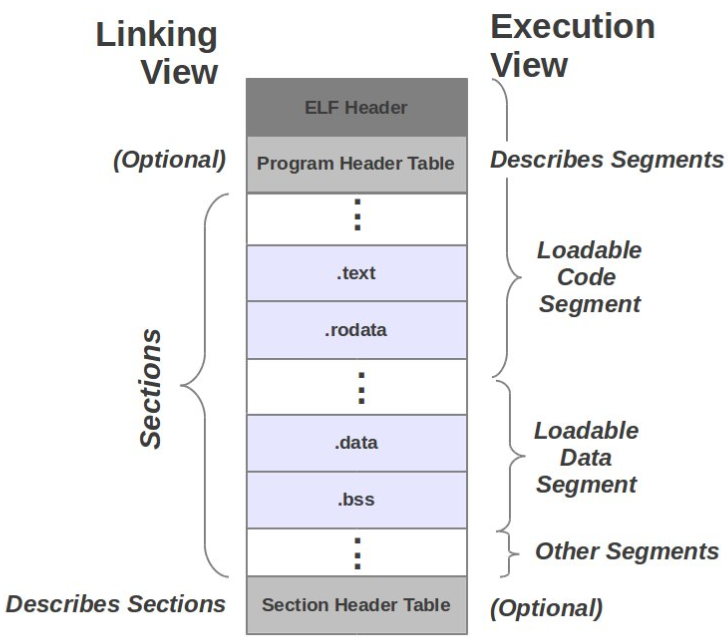
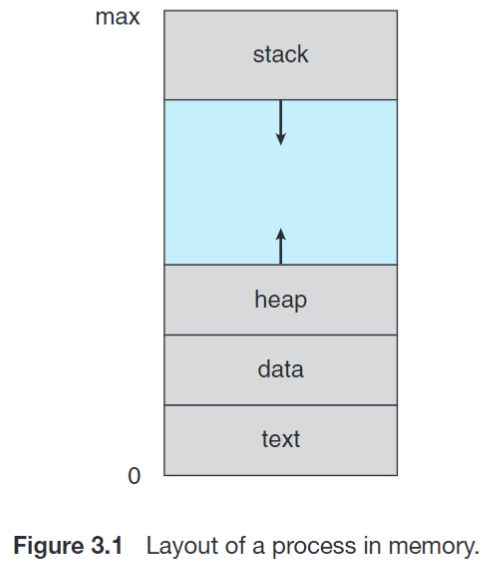
大多数现代操作系统都实现了内核模块。面向对象，内核部件分离，通过已知接口进行沟通，都是可以载入到内核中的。总而言之很像层次结构但是更加灵活。Linux solaris。

**混合系统Hybrid**

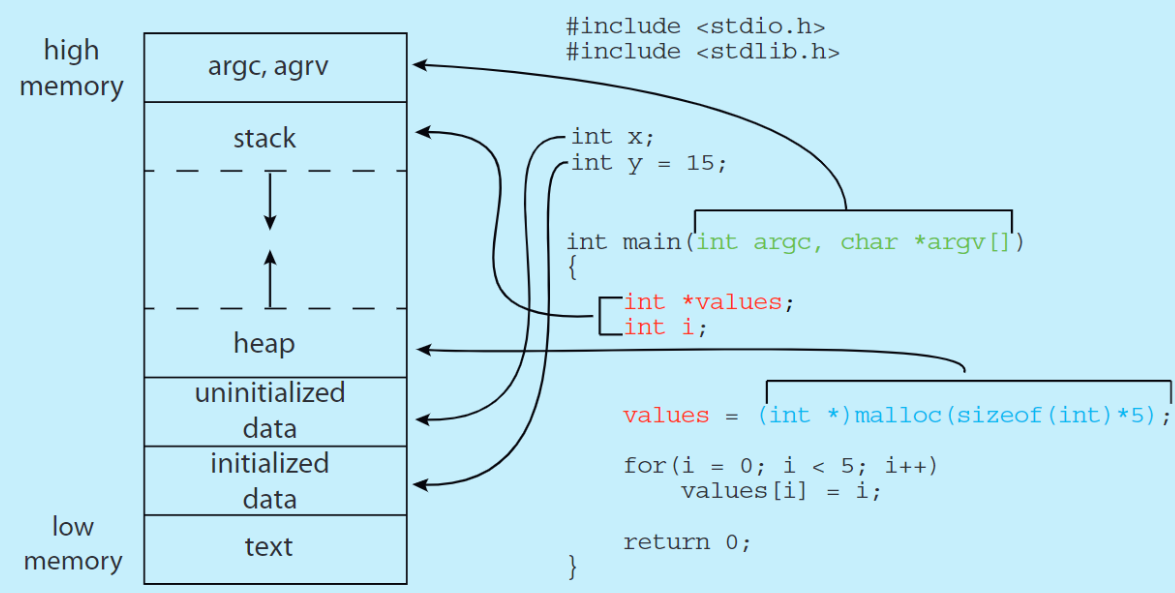
大多数现代操作系统不是单一模型。Linux和solaris是monolithic+module。Windows大部分是monolithic加上microkernel。Mac是层次 Hybrid

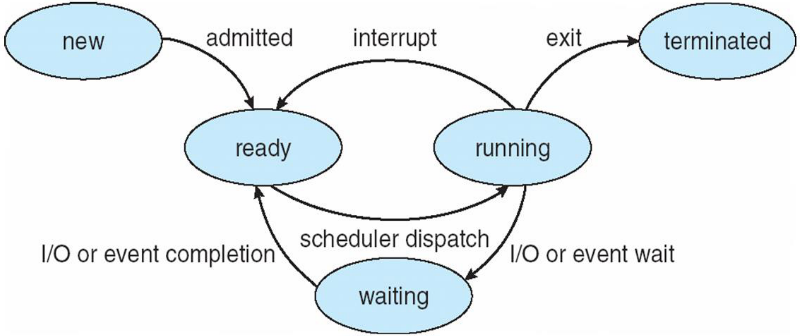
**Process 进程**

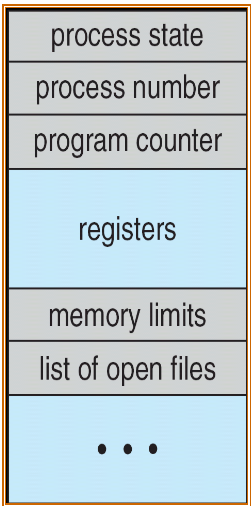
进程 (Process) 是被加载到内存中、正在运行的程序。多个进程可能对应同一个程序。一个正在运行的 OS 中会有多个进程。进程是程序的一次执行过程，是操作系统分配资源的基本单位。

****

stack:临时数据(局部变量,参数,ra,返回值),heap:动态分配内存,data:global virables,text:code



**进程包括**：PC,寄存器,数据段(全局data),栈(临时data),堆(alloc)。

**进程状态**：进程状态会因为程序(系统调用),OS(调度),外部(中断)动作而改变状态。单处理器下，最多一个run，ready进程构成就绪队列，wait进程构成多种等待队列。运行最多1最少0，等待最多n最少0，就绪最多n-1最少0。

OS用**PCB**来表示进程，PCB包括

Process state,Program counter, CPU registers,CPU scheduling information,

Memory-management information

Accounting information File management I/O status information

Linux的PCB保存在struct task\_struct里

Fork: fork() -> exec\*() (给子进程新的程序)-> parent calls wait() for children to terminate

wait(): block until any child terminates.

waitpid(): block until a specific child completes

Signal: signal() system call allows a process to specify what action to do on a signal

Zombies

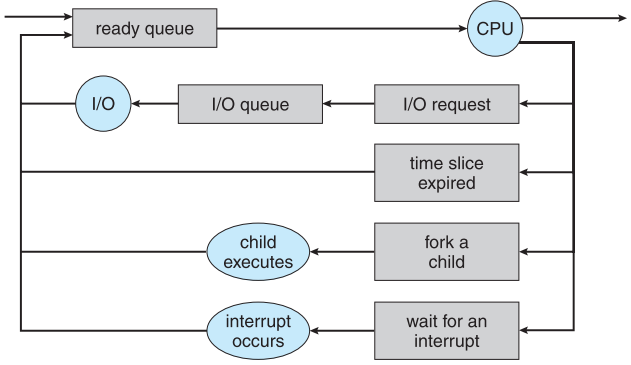
当一个进程终止时，它进入 terminated 状态，它的资源被操作系统回收。但是，操作系统仍然会保存一些信息（例如 PID，结束状态，资源使用情况等2），因为父进程有可能会需要调用 wait() 来获取其一些信息。当子进程已经终止，但父进程在忙，还没有调用 wait()，我们称这样的子进程为 僵尸进程 (zombie processes)，因为前述信息仍然占据了进程表中的一项；如果表满了，就不能创建新的进程了。

Orphans:

当子进程没有结束，或者终止了但父进程没有调用 wait() 的情况下，父进程就结束了，子进程就会成为 孤儿进程 (orphan processes)。一些操作系统会将孤儿进程一同终止掉，但是 UNIX 的做法是让 init 进程收养它们，即 init 进程成为其父进程。init 进程会定期调用 wait()，从而收集孤儿进程的退出状态，并释放进程表条目。

**进程调度**

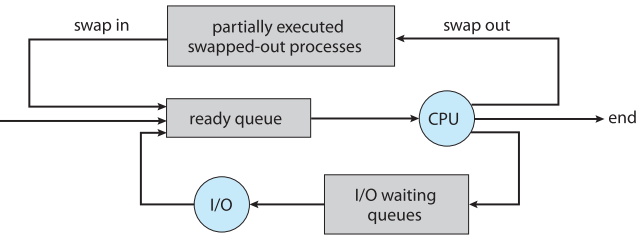
调度队列有：作业job队列 就绪队列 设备队列 进程在这些队列内移动。下图是队列图，新进程开始就处于就绪队列



长程/作业调度：选择应该被带入就绪队列的进程，调度频率低，控制多道程序设计的程度(内存中进程数),现代操作系统没有长程unix win。

短程/CPU调度：从接下来要执行的进程中选择并分配CPU，频率很高。

中程调度：能将进程从内存或CPU竞争中溢出，降低多道程序设计的程度。进程可以被换出，之后再换入。增加了中程调度的队列图



进程可以分为IO型和CPU型(CPU-bound)

上下文切换：CPU切换进程时，必须保存当前状态再载入I新状态的过程。进程的上下文储存在PCB中。上下文切换是overhead，过程中不能进行其他事务。

**进程操作**

父子进程资源共享模式：共享全部资源/部分/不共享；

执行模式：并发执行/父进程等待子进程结束再执行；地址空间：子拷贝父/子进程装入另一个新程序。

进程终止：父进程终止时，不同OS对子进程不同：不允许子进程继续运行/级联终止/继承到其他父进程上。

独立进程：运行期间不会受到其他进程影响。

合作进程：信息共享,运算速度提高,模块化,方便。生产消费问题的两种缓冲：无限缓冲unbounded-buffer生产者可以无限生产;有限缓冲，缓冲满后生产者要等待

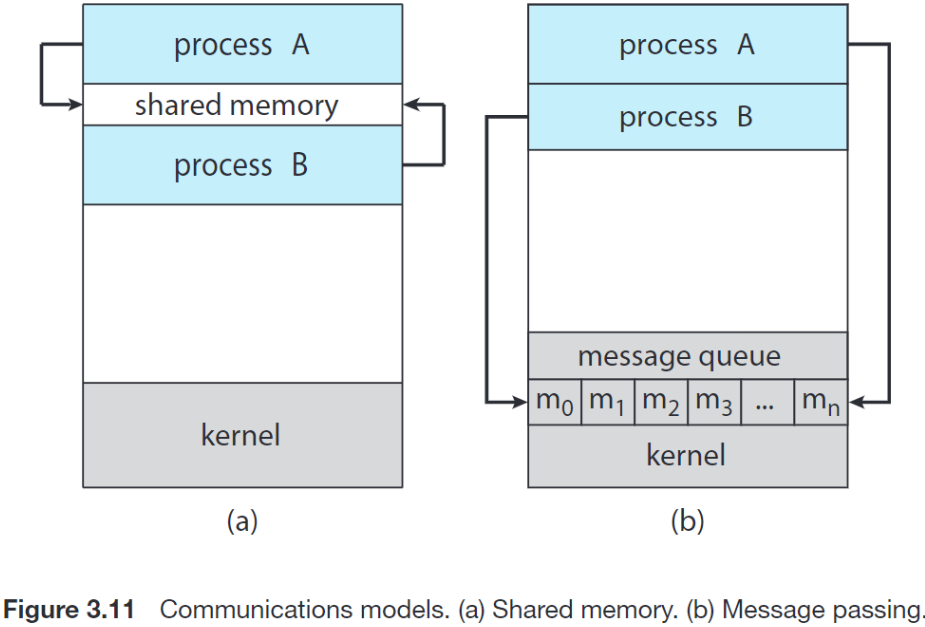
**进程通信IPC**

**Shared memory:** 两个进程各自有一块虚拟内存，映射到同一块物理内存。共享内存也需要信号量等同步手段保护。

**Message passing:** 消息队列，其实就是操作系统维护的一个一个的链表，进程可以新建或者连接到一个消息队列，并写入消息，或读取第一条满足一定条件的消息。

**Signals, Pipes:** 管道的实现其实也是文件，创建管道时操作系统会返回两端的文件描述符；逻辑上实现的是一个半双工的信道。

**RPC remote procedure calls**



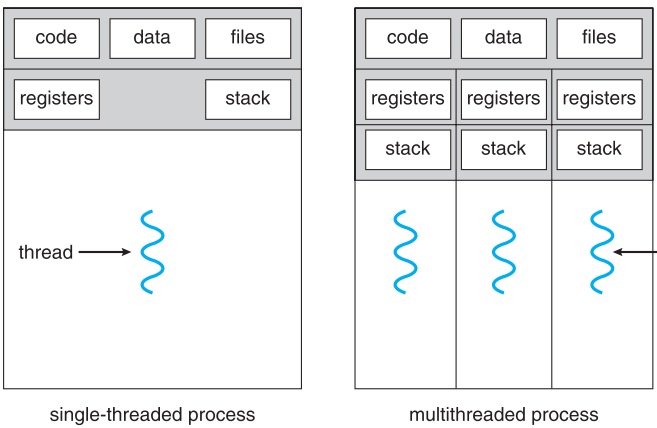
**Thread 线程**

进程概念体现了两个特征：资源拥有单位/调度单位unit of resoutce ownership/unit of dispatch，OS将它们分别处理，调度单元被称为线程或轻量进程。

每个线程都有它自己的 thread ID, PC, register set 和 runtime stack。线程与同一进程的其他线程共享 code section, data section, heap, open files 以及 signals。

线程就是进程内一个执行单元或可调度的实体。

线程能力：有状态转移,不运行时保存上下文,有一个执行栈,有局部变量的静态存储,可以存取所在线程的资源,可以创建撤销其他线程,不拥有系统资源(拥有的少量资源,资源是分配给进程)



**优点**：

Economy: 建立线程相比进程是很经济的，因为 code, data & heap 已经在内存中了；另外在同一进程的线程间进行 context switch 时也会更快，因为我们不需要 cache flush。

Resource Sharing: 同一进程的线程之间天然共享内存，因此我们无需为它们编写 IPC；这也允许我们对同一块内存做并行的处理。但这也会引入风险。

Responsiveness: 多线程的进程会有更好的响应性，即当一个线程 blocked 或者在做一些长时间的操作时，其他线程仍然能完成工作，包括对用户的响应。

Scalability: 在多处理器的体系结构中，多线程进程可以更好地发挥作用，因为每个线程都可以在一个处理器上运行；而单线程进程只能在一个处理器上运行。

**缺点：**weak isolation(内存保护困难); one thread fails, process fails

**用户级线程**：不依赖于OS核心(内核不了解用户线程的存在), 应用进程利用线程库(thread library)提供创建 同步 调度和管理线程的函数来控制用户线程。一个线程发起系统调用而阻塞，则整个进程在等待。同一个进程中的多个用户级线程无法在多核上分别运行

**内核级线程**：依赖于OS核心，由内核的内部需求进行创建和撤销 ，用来执行一个指定的函数。一个线程发起系统调用而阻塞，不会影响其他线程。时间片分配给线程，所以多线程的进程获得更多CPU时间。

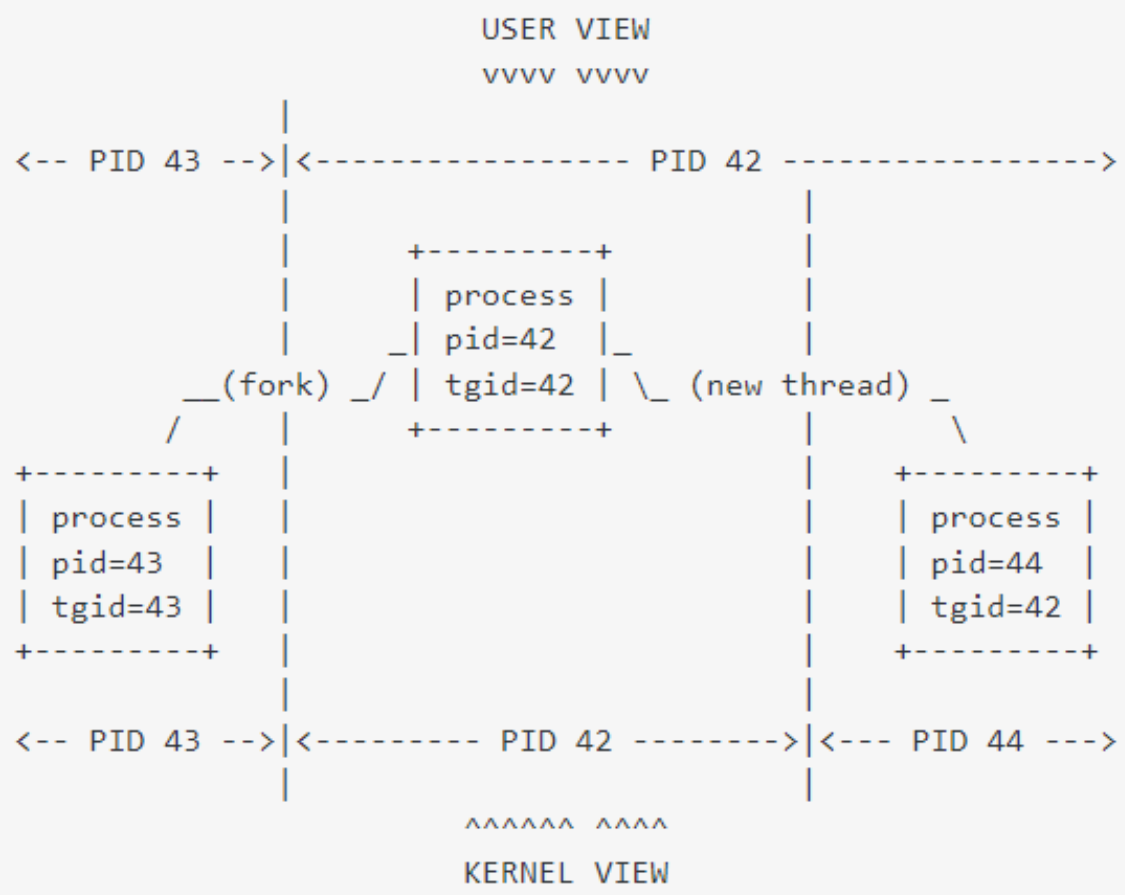
**多线程模型：**

**多对一：**将多个用户级线程映射到一个内核线程。由线程库在用户空间进行。优点：无需OS支持，可以调整(tune)调度策略满足应用需求，无系统调用线程操作开销很低。缺点：无法利用多处理器 不是真并行，一个线程阻塞时整个进程也阻塞

**一对一：**一个用户到一个内核。每个内核线程独立调度，线程操作由OS完成。优点：每个内核线程可以并行跑在多处理器上，一个线程阻塞，进程的其他线程可以被调度。缺点：线程操作开销大,OS对线程数的增多处理必须很好

**多对多：**多用户到多内核，允许OS创建足够多的内核线程，Solaris prior v9 win NT/2000 with ThreadFiber。**两极模型：**多对多的变种，一部分多对多，但是有一个线程是绑定到一个内核上。

线程调用fork：两种情况：仅复制线程、复制整个进程的所有线程（Linux为第一种）



**两种线程取消：Asynchronous**：立即终止目标线程;**Deferred**：目标线程不断检查自己是否该终止。信号处理：信号由特定事件产生，信号必须要发送给进程，信号被发送后需要被处理。选择：发送信号到信号所应用的线程；到进程内的每个线程；进程内的某些线程；规定特定线程接收信号。线程池：优点：用现有线程处理请求比等待创建新线程快；限制了可用线程的数量。线程特有数据：允许线程自己保存数据拷贝，在无法控制创建线程时很有用，比如用线程池的时候。

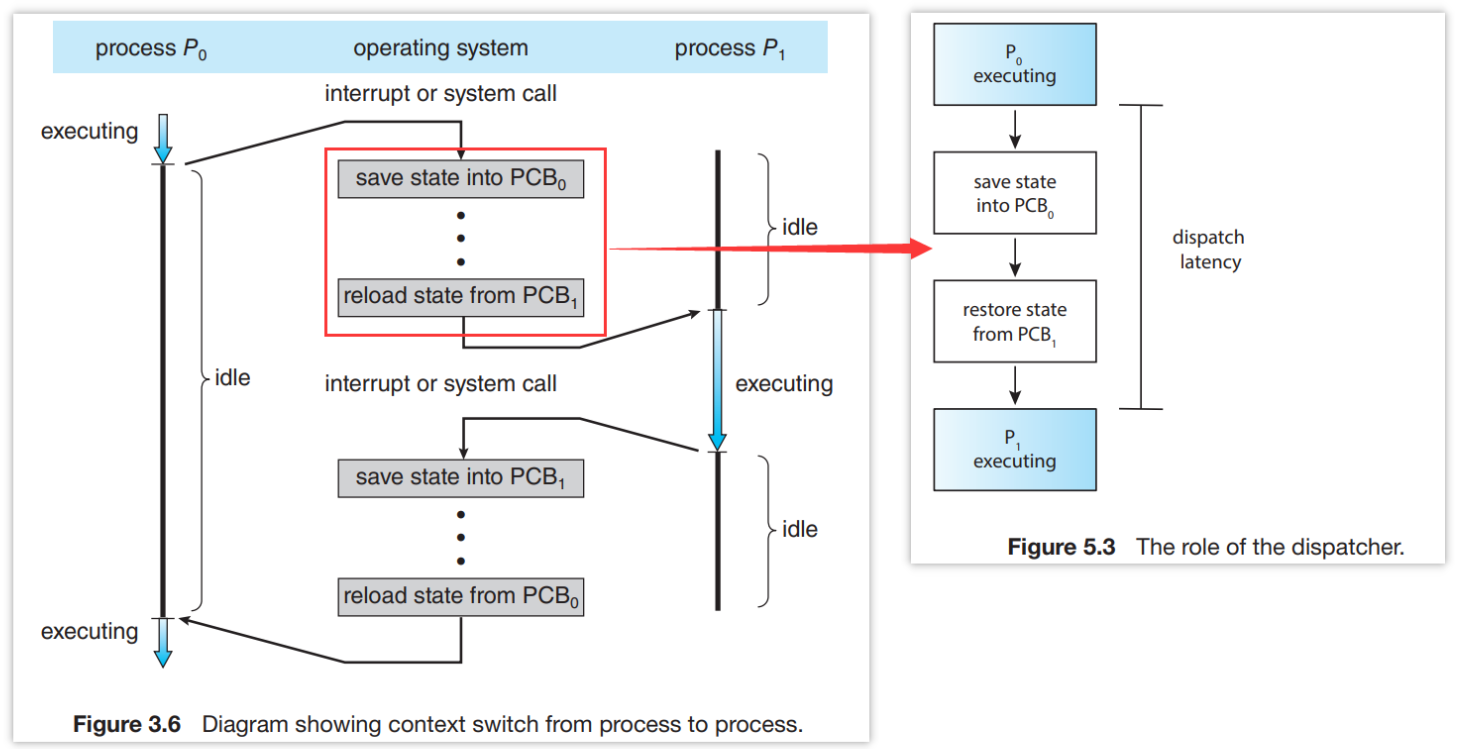
调度程序激活 Scheduler Activations

多对多和两级模型需要通信来维护分配给应用适当数量的线程；SA提供upcalls，一种从内核到线程库的通信机制；这种通信保证了应用可以维持正确数量的线程。Win xp linux

Linux把线程叫做tasks，除了fork，额外提供了线程创建通过clone系统调用完成，它允许子任务和父任务共享地址空间

**CPU Scheduling**

user(pt\_regs)->kernel(context\_regs)->user



调度只能发生在时间片用尽时叫非抢占(non-preemptive)，否则叫做抢占(preemptive)。

**分派程序(dispatcher)**将CPU的控制交给由短程调度选择的进程。功能有：上下文切换 切换到用户模式 跳转到用户程序的合适位置来重启程序。分派程序停止一个进程而启动另一个所需要花费的时间叫分派延迟(dispatcher latency)。

**调度算法的选择准则和评价：**

面向用户：**turnaround time**(进程从commit到完成所用时间 包括就绪和阻塞中的等待时间 带权周转时间=周转时间/CPU执行时间) 响应时间 **waiting time**(在ready queue中等待时间) 截止时间 公平性 优先级。

面向系统：throughput(每个时间单元内完成的进程) CPU utilization 设备均衡利用。调度算法自身：易于实现 开销较小。最佳算法准则：CPU利用率 吞吐量 周转时间 等待时间 响应时间 公平

**调度算法**

First-Come, First-Served (FCFS) Scheduling：根据就绪状态的先后来分配CPU，非抢占，最简单，利于长进程，不利于短进程，利于CPU型不利于IO型。

Shortest-Job-First (SJF) Scheduling：对预计执行时间短的作业优先分派CPU，分为抢占式：如果新来的进程时间比当前的还短，抢占，这种SJF叫做Shortest-Remaining-Time-First (SRTF)。非抢占：Shortest-next-CPU-burst允许当前进程运行完再运行最短的。SJF被证明是最佳算法，它能给出最小平均等待时间。而**SJF是无法实现的，因为不知道下一个CPU脉冲burst时长**。

一种解决：预测burst时长：Exponential averaging of previously observed burst durations：

新预测=(1-\alpha)第n次时长+\alpha原预测

**优先级调度**

每个进程都有优先级数字相关联，总是把CPU分配给就绪中最高优先级的进程。确定进程优先级的两种方法：静态优先权：创建时就确定好 动态优先权：给予某种算法调整。**一般数字越小优先级越高**。SJF是以下一次CPU的脉冲长度作为优先数的优先级调度特例。优先级调度也可以是抢占/非抢占的。问题：starvation，低优先级的进程可能永远无法执行。解决方法：老化(**aging**)逐渐增加在系统中等待时间长的进程的优先级，也就是动态优先级。

**时间片轮转调度Round Robin(RR)** **Preemptive**

通过时间片轮转提高并发性和响应时间，提高资源利用率。算法：将就绪中的进程按照FCFS排队；每次调度时将CPU分给队首进程，让其执行一个时间片；时间片结束时发生时钟中断；调度程序暂停当前进程执行将其送至就绪的队尾，然后上下文切换至新的队首；对称没用完一个时间片的话就让出CPU(如阻塞)

RR算法性能依赖于时间片大小，如果q很大，就和FCFS一样了；如果q很小，那么q也要足够大来保证上下文切换，否则开销过大。时间片长度的影响因素：响应时间一定时，就绪进程越多，时间片越小；应当使用户输入通常能在一个时间片内完成，否则相应 平均周转和平均带权周转都会延长。一般来说RR比SJF有更高的平均周转，无starvation, response time更好。时间片固定时，用户越多响应时间越长。

**多级队列调度**

一般，分成前台队列foreground(交互式interactive)和后台队列(批处理)，后台一般FCFS，前台RR。多级队列在队列间的调度算法有：固定优先级，即先前台后后台，有饥饿；给定时间片，如80%执行前台的RR，20%执行后台的FCFS。

**Multilevel Feedback Queue Scheduling多级反馈队列：Q0:RR(q=8),Q1:RR(q=16),Q2:FCFS,在Q0(8ms内)没跑完的到Q1，Q0全部结束开始Q1，以此类推**

是RR和优先级算法的综合。与多级队列的区别:**允许进程在不同的就绪队列切换**，等待时间长的进程会进入到高优先级队列中。优点：提高吞吐量降低平均周转而照顾短进程；为IO设备利用率和降低响应时间而照顾IO进程型；**IO bound给予高优先级, CPU bound低优先级**

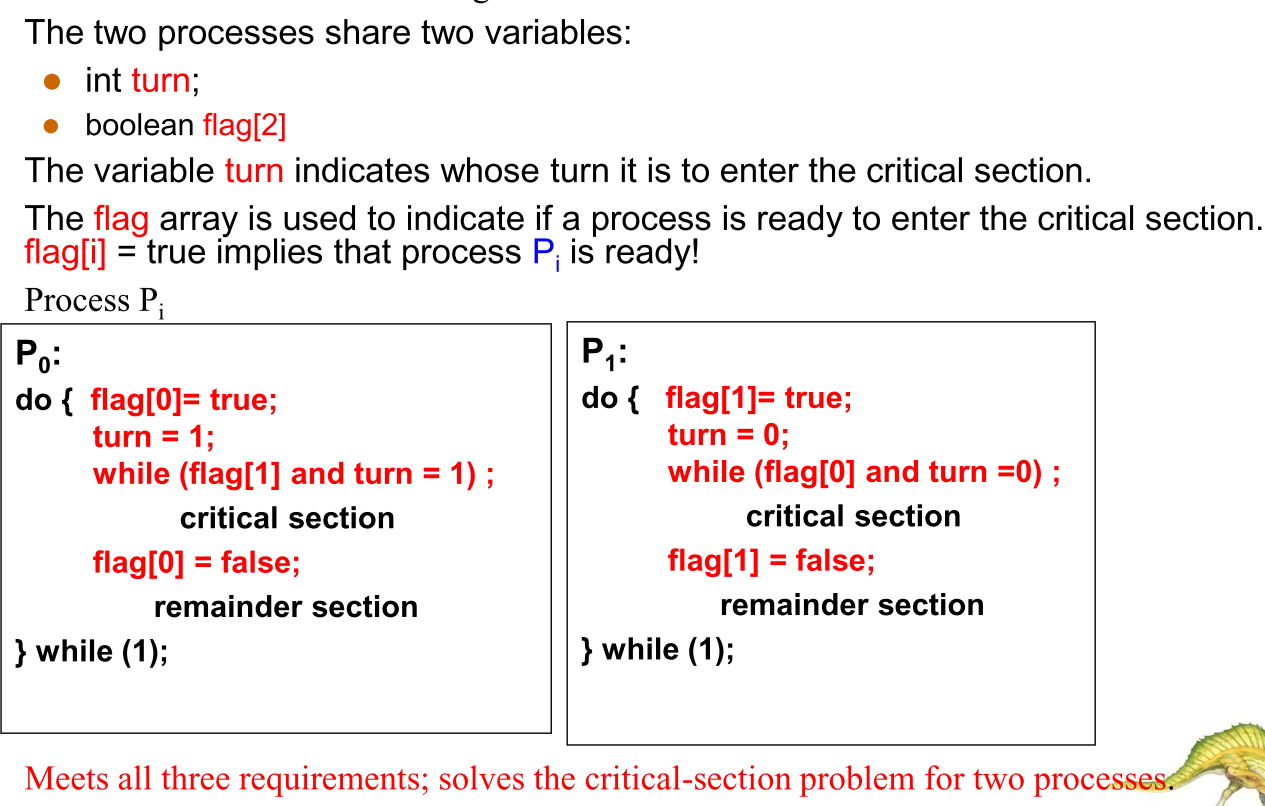
**Process synchronization**

**The Critical-Section Problem**

N个进程都计算同样的共享数据，每个进程都有一个临界区，其中共享数据被访问。问题：需要保证只有一个进程进入临界区。临界区问题的解决必须满足三个要求：**互斥(mutual exclusion)；空闲让进(progress)；有限等待(bounded wait)**。让权等待不是必须的。

**Peterson算法**

只用于两个进程的情况，并且假设load和store是原子操作，是一种软件解决方法。

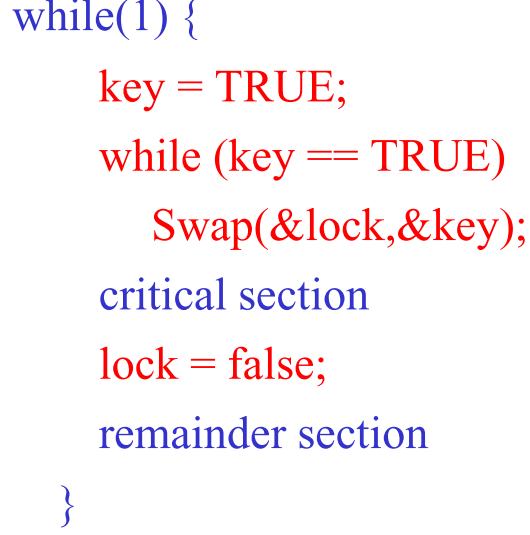


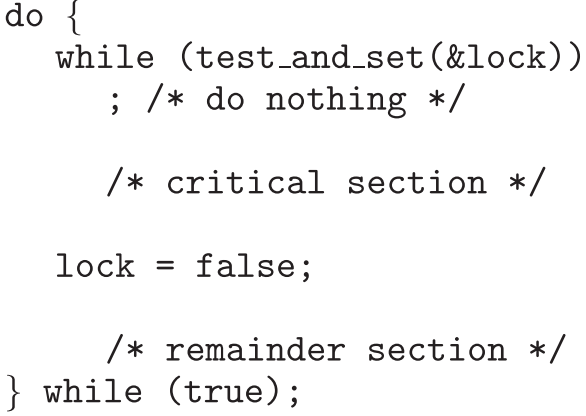
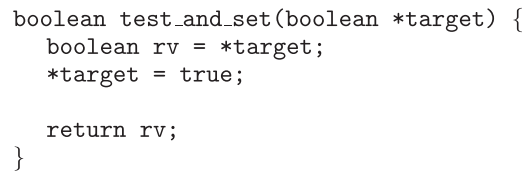
现代OS中不适用(若编译器交换flag和turn指令则寄)

**硬件同步方法**

单处理器：在临界区禁止中断。

多处理器：**Memory barriers**(an instruction forcing any change in memory to be propagated (made visible) to all other processors)

**Hardware instruction**



Test\_and\_set存在busy waiting的情况。

抽象出两个硬件实现的原子操作：赋值和交换，然后来解决临界区，testandset的共享变量是lock 初始false，swap也是一样，但是多了局部变量key不是共享的。

硬件方法优点：进程数随意，简单，支持多个临界区；缺点：不满足bounded waiting，饥饿,死锁。会引起忙等待

**semaphores**

两个操作wait(P)和signal(V)。信号量分为计数信号量，值域不受限制，二值信号量，只能是01所以也叫互斥锁(mutex locks = spinlock)。为了等待资源进行无限循环是忙等，通过对信号量的修改增加了block(run->wait)和wakeup(wait->ready)来避免了忙等。wait(semaphore \*S) { S->value--; if (S->value < 0) { add this process to S->list; block();} } signal(semaphore \*S) { S->value++; if (S->value <= 0) { remove a process P from S->list; wakeup(P);} }具有忙等的信号量值非负，但是这种实现可以为负，负数的绝对值代表等待该信号量的进程数，0代表无资源可用。Wait和signal成对出现，互斥操作就在同一进程出现，同步操作在不同进程。连续的wait顺序是需要注意的，但是连续的signal无所谓。同步wait和互斥wait相邻时，要先同步wait。优点：简单、表达能力强；缺点：不够安全，使用不当会死锁，实现复杂

**优先级倒置(priority inversion)**：当优先级较低的进程持有较高优先级进程所需的锁定时的调度问题。解决方法：**priority inheritance:** 所有正在访问资源的进程暂时获得需要访问这个资源的更高优先级进程的优先级。

信号量至少为1才能执行，否则等待其他进程解锁，wait上锁，signal解锁

**Bounded-Buffer Problem 有限缓冲区 生产者-消费者问题**

是很多相互合作进程的抽象。算法：设置N个缓冲项；信号量mutex初始化为1，用来保证对缓冲池访问的互斥要求；信号量full初始化为0，表示满缓冲项的个数；信号量empty初始化为N表示空缓冲项的个数。生产者：do {…produce an item in nextp …wait(empty); wait(mutex); …add nextp to buffer …signal(mutex); signal(full);} while (1);消费者：do {wait(full); wait(mutex); …remove an item from buffer to nextc …signal(mutex); signal(empty); …consume the item in nextc …} while (1);

**Readers-Writers Problem**

数据库读写的抽象。第一读写问题：允许多个读者同时读，但是只有一个写者，也就是没有读者会因为写者在等待而等待其他读者的完成，写者可能饿死。第二读写问题：写者就绪后，写者就立即开始写操作，也就是说写者等待时，不允许新读者进行操作，读者可能饿死。

共享数据有访问的数据、mutex初始1,保证更新readcount时互斥，wrt初始1,为读写公用，供写者作为互斥信号量，被第一个进入临界区和最后一个离开临界区的读者使用，其他读者不适用。Readcount初始0，用来跟踪多少进程正在读。写进程：do {wait (wrt) ; // writing is performed signal (wrt) ;} while (TRUE); 读进程：do {wait(mutex); readcount++; if (readcount == 1) wait(wrt);signal(mutex); …reading is performed …wait(mutex); readcount--; if (readcount == 0) signal(wrt);signal(mutex)； } while (TRUE);

**Dining-Philosophers Problem 哲学家进餐**

典型的同步问题，问题描述：N个哲学家坐在圆桌，每个哲学家和邻居共享一根筷子；哲学家吃饭要用身边的两只筷子一起吃；邻居不允许同时吃饭；哲学家只会思考或者吃饭。

共享数据：数据集/一碗米饭；共享变量chopstick[5]初始为1；第i个哲学家进程：do {wait(chopstick[i]) wait(chopstick[(i+1) % 5]) … eat …signal(chopstick[i]); signal(chopstick[(i+1) % 5]); … think …} while (1);这个解决方案可以保证没有2个哲学家同时使用1个筷子，但是很显然会导致死锁，如果5个哲学家同时饥饿，同时拿起左手筷子，就死锁了。

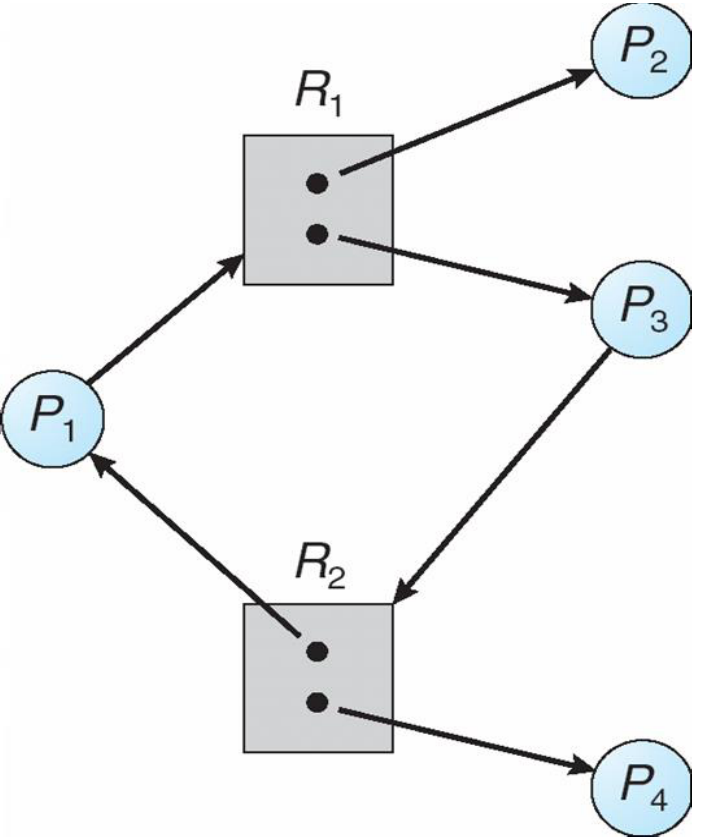
一些其他的可能解决：最多只允许4个哲学家坐在桌上/临界区内必须拿起两根筷子/使用非对称的解决方法：奇数先拿左手，偶数先拿右手。这些额外限制都能防止死锁。

**Deadlock 死锁**

死锁指多个进程因竞争共享资源而造成的一种僵局，若无外力作用，这些进程都将永远不能再向前推进。进程按以下顺序使用资源：申请 使用 释放。申请和释放为系统调用。

**四个条件同时成立，发生死锁：**

Mutual exclusion：至少有一个资源处于共享模式，一次只能有一个进程使用该资源;hold and wait：一个进程必须至少占有一个资源并且等待另一个资源，且该资源被其他进程占有;No preemption：资源不能被抢占，只能在进程使用完成后释放;circular wait：进程间循环等待资源，A等B占的 B等C占的 C等A占的。

**资源分配图**如果分配图无环->没有进程死锁，如果有环，那么**可能**死锁。**如果每个资源恰好只有一个实例，有环则必死锁**。如果环所在的资源类型是只有一个实例的，则必死锁。如果每个资源有多个实例，有环不一定死锁。

P4可能释放R2的实例，这个资源分配给P3，这样就打破了死锁。

**死锁处理**

保证系统不进入死锁：prevention avoidence；允许进入死锁但是需要恢复：detection recovery。U L W三个系统都忽略问题假装没有死锁，是鸵鸟方法。

**死锁预防Prevention**

至少一个必要条件永不成立，来保证系统不会出现死锁

Mutual exclusion：非共享资源必须互斥，例如一台打印机不能被多个进程共享，而共享资源不需要互斥，也不导致死锁，类似只读文件。

Hold and wait：必须保证一个进程**申请一个资源时不能占有其他资源**。进程在执行前就要申请并分配资源，是资源静态预分配的方法；缺点：低资源利用率、可能饥饿。

No preemption：如果一个进程占有资源并且申请了另一个不能立即分配的资源，那么它现已分配的资源都可以被抢占，也就是被隐式释放了。抢占资源分配到进程所等待的资源的列表中。进程需要获取到原有的资源和申请的新资源后才能运行。

Circular wait：给资源设置显式序号，请求必须按照资源序号递增的方式进行，通过资源的有序申请破坏了循环等待条件。然而有时候资源的“顺序”是模糊的（from, to）

**死锁避免**

前面的方法虽然避免了死锁，但是降低了吞吐率，我们可以通过获取一些额外的事先信息从而避免死锁prior information。

最简单和最有效的模型要求每个进程声明它**可能需要的每种类型的资源的最大数量**。死锁避免算法动态检查资源分配状态，确保永远不会出现循环等待。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。

**安全状态：**对于所有进程，如果存在一个安全序列，那么系统就处于安全状态。对于进程序列P1,P2,…,Pn，如果对于每一个Pi,Pi仍然可以申请的资源数小于当前可用的资源加上所有进程Pj(i>j)所占有的资源，那么这一序列是安全序列。这种情况下，进程Pi的资源即使不能立即可用，那么Pi可等待直到所有Pj释放其资源，当它们完成时Pi就可以运行，Pi运行结束后，Pi+1就可以获得到所需的资源，如此进行。

安全状态->没有死锁；不安全状态->可能有死锁；避免->保证系统永远不进入非安全状态。

**资源分配图，single instance死锁避免算法：**

Single instance: 每种资源只有一个

引入一种新边claim edge需求边，Pi->Rj表示进程Pi在未来可能请求资源Rj，用虚线表示。当进程真正请求资源时，用请求边覆盖掉需求边。当资源被分配给进程后，用assignment edge分配边来覆盖掉请求边，当资源被释放后，分配边恢复为需求边。系统必须事先说明需求边。

算法：假设进程Pi申请资源Rj。只有在需求边Pi –>Rj 变成分配边Rj->Pi而**不会导致资源分配图形成环**时，才允许申请。

用该算法循环检测，如果没有环存在，那么资源分配会使系统继续安全状，否则就会不安全，Pi就要等待。

**Banker，多实体资源类型避免算法：**

选取一个 need（的每一项都对应地）小于 available（的对应项）的进程，其运行完后会将 allocation 释放回 available，以此类推。

**安全状态检测算法：**

1. 设work和finish分别是长度为m和n的向量，初始化：work=available，finish[i]=false；

2.寻找i满足finish[i]=false且need[i]<=work，如果i不存在跳到第四步；

3.work=work+allocation[i],finish[i]=true，返回第二步；

4.如果所有的finish都是true，那么系统处于安全状态。

算法需要m\*n\*n的操作数量级确定系统状态

**资源请求算法：**

Request[i]为Pi的请求向量，如果request[i][j]=k那么进程Pi需要资源类型Rj的数量为k。当进程Pi请求资源时，动作如下：

1.如果request[i]<=need[i]跳到第二步，否则出错，因为进程Pi已经超过了其最大需求。

2.如果request[i]<=available跳到第三步，否则Pi必须等待，因为没有可用资源

3.假定系统可以分配给进程Pi请求的资源，进行下面的操作：Avaible=avaible-request[i]; allocation[i]=allocation[i]+request[i];need[i]=need[i]-request[i]；如果产生的资源分配状态是安全的，那么交易完成且进程Pi可以分配到资源，如果新状态不安全，那么进程Pi必须等待Request[i]并且恢复到原有的资源分配状态。

**死锁检测**

允许系统进入死锁状态的话，那么系统就需要提供检测算法和恢复算法。

**等待图，单实体资源类型检测算法：**

等待图是资源分配图的变形，节点都是进程，Pi->Pj表示Pi在等待Pj释放Pi所需的资源。当且仅当等待图中有一个环，系统死锁，检测环的算法需要O(V+E)，最坏情况下O(n^2)

**多实体资源类型检测算法：**

类似银行家算法。如果找不到任何安全序列，则说明系统处于死锁状态。

**死锁检测算法的应用**

检测算法的调用时刻及频率取决于：死锁发生频率以及死锁发生时受影响的进程数。如果经常发生死锁，那么就要经常调用检测。如果在不确定的时间调用检测算法，资源图可能有很多环，通常不能确定哪些造成了死锁

**死锁恢复**

检测到死锁后的措施：通知管理员 系统自己恢复。打破死锁的两种方法：抢占资源 进程终止。

**进程终止**

两种方法来恢复死锁：1.终止所有死锁进程 2.一次终止一个进程直到不死锁。许多因素都影响终止进程的选择：优先级 进程已经计算了多久，还要多久完成 进程使用了哪些类型的资源 进程还需要多少资源 多少进程需要被终止 进程是交互的还是批处理的

**抢占资源**

抢占资源需要处理三个问题：

选择一个牺牲品victim：要代价最小化

回滚：回退到安全状态，但是很难，一般需要完全终止进程重新执行

饥饿：保证资源不会总是从同一个进程中被抢占。常见方法是为代价因素加上回滚次数。

**Main Memory主存**

层次存储中主存cache寄存器为volatie易失的。

逻辑地址/虚地址/相对地址：由CPU生成，首地址为0，逻辑地址无法在内存中读取信息。物理地址/实地址/绝对地址：内存中储存单元的地址，可以直接寻址。

物理地址中的逻辑地址空间是通过一对基址寄存器和界限地址寄存器控制的base and limit register。如果基址寄存器为300040，界限寄存器为120900，那么程序的合法访问从300040到420910(含)的所有地址。

**Address Binding流程：**

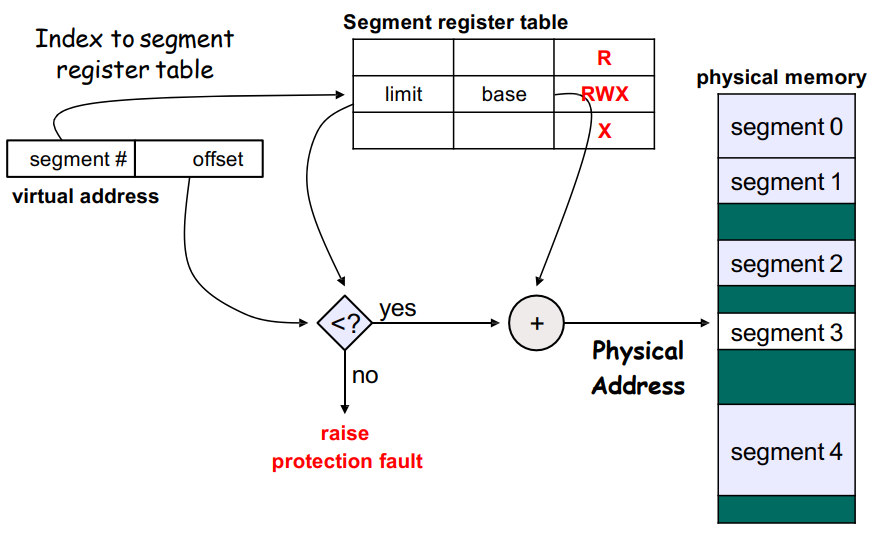
编译时间：如果编译时就知道进程在内存中的地址，那么就可以直接生成绝对代码absolute code。否则编译器会将label绑定到 relocatable addresses。

装载时间：链接器或加载器（linker / loader）会将 relocatable addresses 绑定到 absolute addresses。

执行时间：如果进程在执行时可以移动到另一个内存段，需要硬件支持也就是base and limit目前绝大多数都是采用这种。

**Memory-Management Unit (MMU)分段**

就是将虚拟地址映射到物理地址的硬件设备。在MMU中，base寄存器叫做重定位寄存器，用户进程送到内存前，都要加上重定位寄存器的值。PA=relocation reg+LA。用户程序只能处理LA，永远看不到真的PA。



**Dynamic Loading（动态加载）**

进程大小会收到物理内存大小的限制，为了有更好的空间使用率，采用动态加载。一个子程序只有在调用时才被加载，所有子程序都可以重定位的形式存在磁盘上，需要的时候装入内存中。OS不需要特别支持，是程序设计做的事。当需要大量的代码来处理一些不常发生的事时很有用，如错误处理。

**Swapping（交换技术）**

进程可以暂时从内存中交换到备份存储backing store上，当需要再次执行时再调回。需要动态重定位dynamic relocate

备份存储：是快速硬盘，而可以容纳所有用户的所有内存映像，并为这些内存映像提供直接访问，如Linux交换区 windows的交换文件pagefile.sys

Roll out roll in：如果有一个更高优先级的进程需要服务，内存交换出低优先级的进程以便装入和执行高优先级进程，高执行完后低再交换回内存继续执行。

交换时间的主要部分是转移时间transfer time。总转移时间与所交换的内存大小成正比。系统维护一个就绪的可立即运行的进程队列，并在磁盘上有内存映像。

**Partition**

Partition要求Protection: 保证进程之间不会互相闯入对方的存储。Fast execution: 不能由于 protection 降低访问内存的效率。Fast context switch: 每当进行 context switch 时，可以比较快地找到并访问当前进程的内存。

1.fixed partition：internal fragmentation

2. Variable Partition：external fragmentation

**Dynamic Storage-Allocation Problem(external fragmentation)：**

动态划分内存，在程序装入内存时切出一个连续的区域hole分配给进程，分区大小恰好符合需要。操作系统需要维护一个表，记录哪些内存可用哪些已用。从一组可用的hole选择一个空闲hole的常用算法first best worst-fit三种。分别是分配第一个足够大的/分配最小的足够大的/分配最大的。First和best在时间和空间利用率都比worst好。还有一个next-fit是每次都从上次查找结束的位置开始找，找到第一个足够大的。

**分页存储管理**

分页允许进程的PA空间非连续；将物理内存分为固定大小的块，叫做帧frame/物理块/页框，将逻辑内存也分为同样大小的块叫做页page，Linux Win(x86)是4KB。

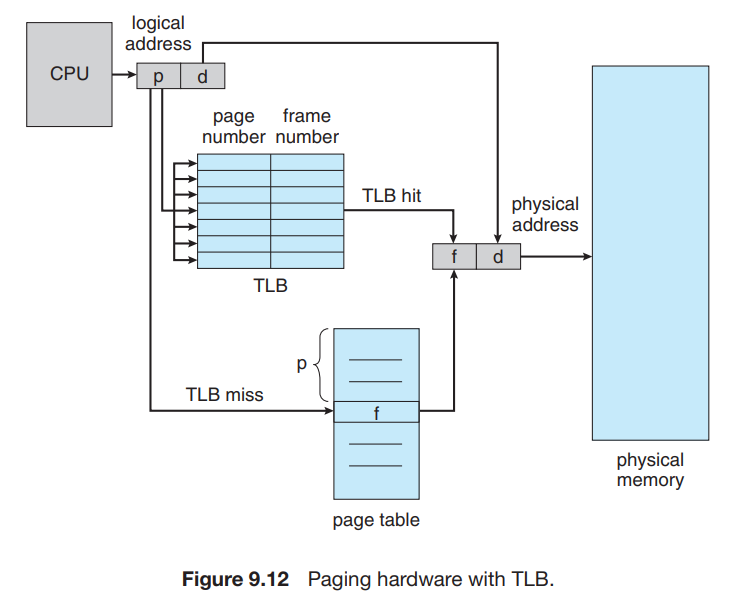
OS需要跟踪所有空闲帧，叫帧表。

运行一个n页的程序就需要找到n个空帧然后装载进去。

地址映射过程：逻辑页号拼上offset经过页表查到物理页号，然后得到物理真号拼上offset，然后进入到内存中找frame。

**页表的实现**

页表放在内存中。PTBR page-table base reg指向页表，切换页表只需要改变这个寄存器就可以 PRLR page-table length register说明页表长度，这样的模式下每次数据/指令访问都需要两次内存访问，一次查页表一次查数据/指令。为了加速这个过程，引入了特殊的转换表缓冲区TLB，是一种硬件cache。部分TLB维护了ASID addressspace identifier，用来唯一地标识进程，为进程提供空间保护。否则每次切换根页表需要flush TLB。\*Associative memory:一种支持并行搜索的内存，如果虚页号与其中键匹配上，则直接返回物理帧。



**Effective Access Time 有效访问时间EAT**

Associative lookup=t1查TLB表的时间

Memory access time=t2 内存访问时间

a:TLB命中率

那么EAT=(t1+t2)\* a+(t1+t2+t2)\* (1-a)

也就是查TLBmiss后，需要进内存查一次页表，再去取一次数据，命中就直接取数据。

**保护protection**

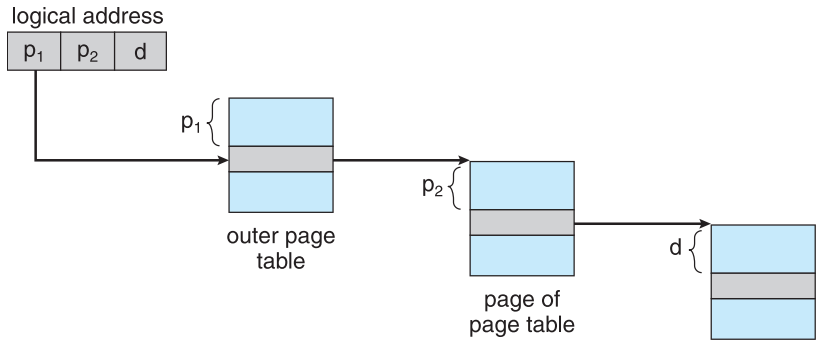
内存保护通过与每个帧关联的保护位实现。Valid bit存在页表中的每一个条目上。

Shared code共享代码：如果代码是可重入即只读代码reentrant code或者是纯代码pure code，可以共享，共享代码在各个进程中的逻辑地址空间相同。然后每个进程再花较小的空间保存私有代码和数据即可。

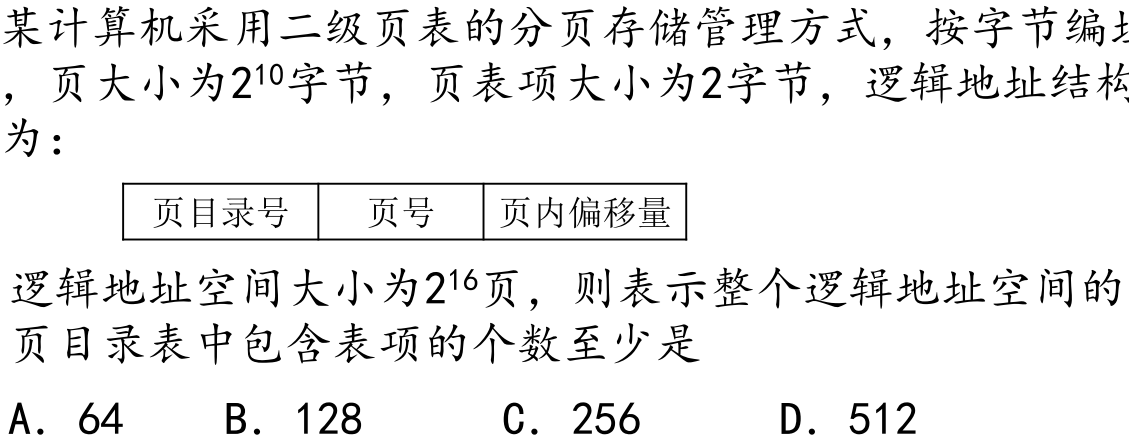
**分级页表Hierarchical page table**

hierarchical paging 其实就是对页表的分页（page the page table）。因此，它避免了 page table 必须处在连续内存的问题，这一问题在 p 比较大时尤其严重。另外，这样做在一般情况下可以节省空间。

简单的实现方法：两页分页算法，就是将页表再分页。就是将页号部分再划分为页偏移和页码。下面是寻址模式：



P1是用来访问外页表的索引，p2是外页表的页偏移，然后d是内页表的偏移。对于一个32位的LA，一般10位外10位内12位偏移



页大小为210B，页表项大小为2B，采用二级页表，一页可存放29个页表项，逻辑地址空间大小为216页，要使表示整个逻辑地址空间的页目录表中包含的个数最少，则需要216/29=27=128个页面保存页表项，即页目录表中包含的个数最少为128。

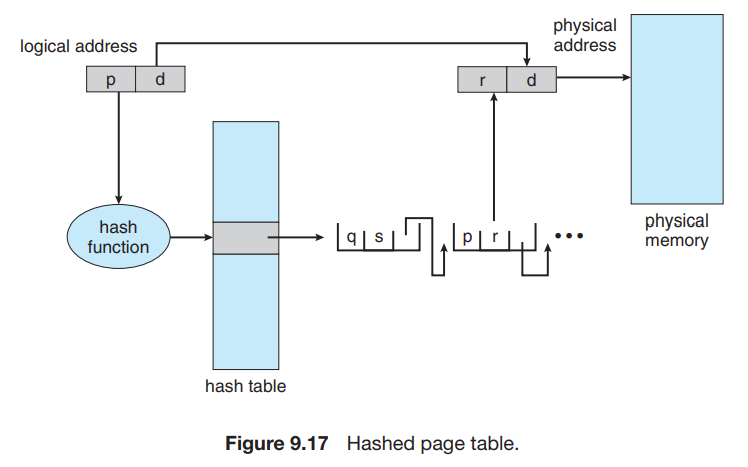
假设一级页表有4MB，页面大小是4KB，页目录项是2B：

有4MB/4KB=2^10个一级页表，因此要10位的二级页表位。每个一级页表可以放4KB/4B = 2^10个entry，因此要10位一级页表位。每个页面大小是4KB，需要12位offset。

三级页表：9+9+9+12 = 39

**哈希页表**

超过32位LA地址空间时，一般采用哈希页表，将虚页号的哈希值存到哈希表里，哈希表的每一项都是链表，链着哈希值相同的页号。然后在查表时用虚页号与链表中的每个元素进行比较从而查物理表号



**反向页表**

对于每个physical frame有一个条目。每个条目包含映射到该frame的虚拟页的**虚地址及拥有该页的进程PID**。因此整个系统只有一个页表，对每个物理内存的帧也只有一条相应的条目。拿时间换空间，需要为页表条目中添加一个地址空间标识符ASID。

**分段 Segmentation**

分页无法避免的是用户视角的内存和物理内存的分离。分段管理支持用户视角的内存管理方案，LA空间是由一组段组成的，每个段都有其名称和长度，地址指定了段名称和段内偏移。因此LA通过有序对<segment-number,offset>构成。

段表将用户定义的二维地址映射成一维，每一个条目包含base和limit。STBR segment table base reg 指向内存中段表的位置，STLR一个程序使用的段长度，用户使用的有序对中的offset必须小于STLR。同样有valid位，还有读写执行的权限设置，也可以进行code share。内存分配是动态存储分配问题。

**Virtual Memory虚存**

虚存将用户的逻辑存储和物理存储分开；LA空间可以大于PA空间；允许PA空间被多个进程共享。

局部性原理：时间：指令的一次执行和下次执行 数据的一次访问和下次访问都集中在一个较短时期内；空间：当前指令和邻近的指令 当前数据和邻近的数据都集中在一个小区域内。

虚存是是具有请求调入功能和置换功能，能仅把进程的一部分装入内存便可运行进程的存储管理系统，它能从逻辑上对内存容量进行扩充的一种虚拟的存储器系统。

**按需调页 Demand Paging**

指在需要时才调入相应的页的技术。采用lazy swapper的方式，除非需要页面，否则不进行任何页面置换。

**页错误 Page fault**

非法地址访问和不在主存或无效的页都会page fault。Page fault rate等于1不代表every page is a page fault。

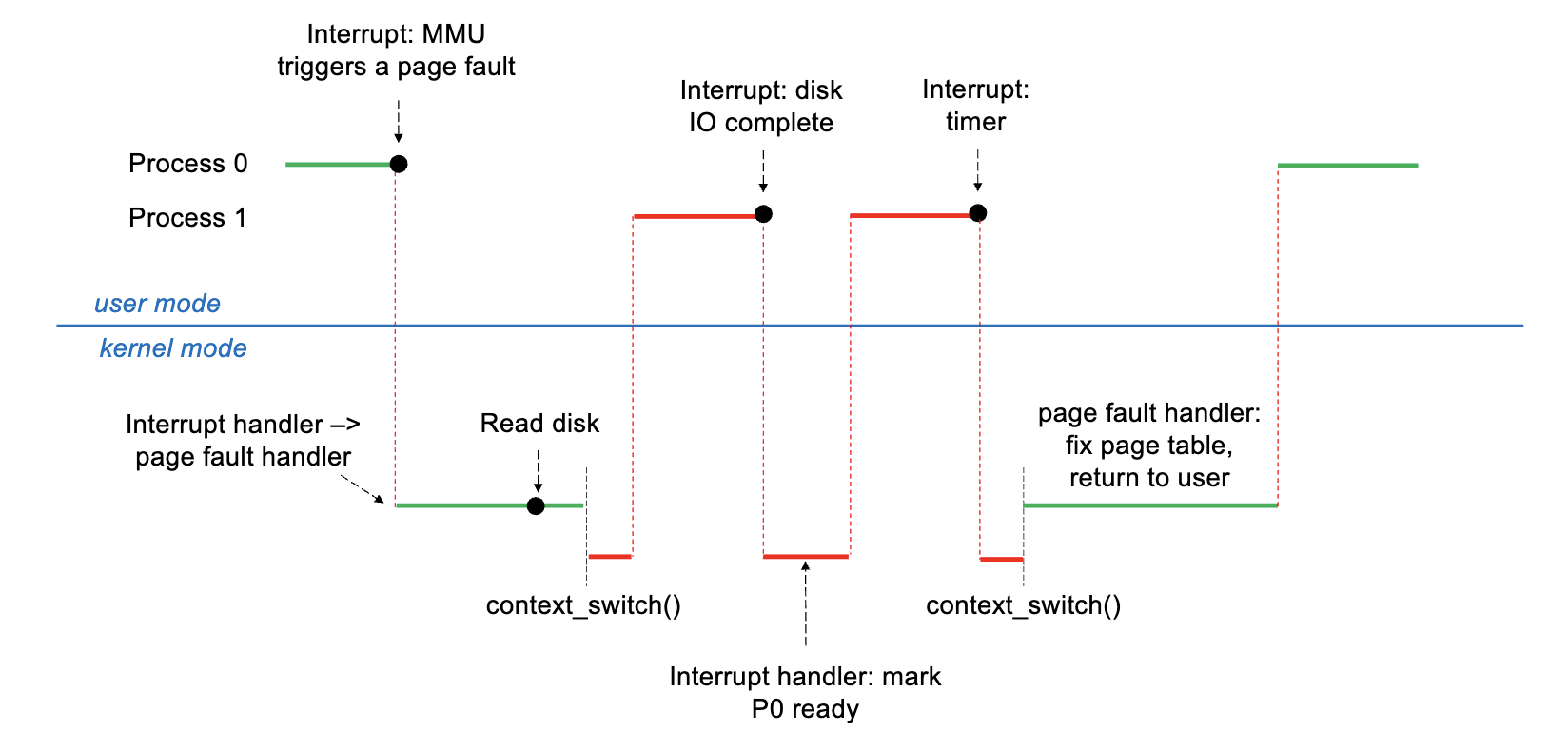
**更完整的页表项请求分页中**

虚拟页号物理帧号状态位P(存在位 页是否已调入内存) 访问字段A(记录页面访问次数) 修改位R/W(调入内存后是否被修改过) 外存地址(用来调页)

**Effective memory-access time有效访问时间**

EAT=(1-p)\*memory access time + p\*page fault time

Page fault time包括page fault overhead, swap page out, swap page in, restart overhead等



其中的三个主要page fault时间是缺页中断服务时间 缺页读入时间和重启时间

**写时复制 copy-on-write**

COW copy on write允许父子进程开始时共享同一页面，在某个进程要修改共享页时，它才会拷贝一份该页面进行写。

Win linux solaris都用了COW

**页面置换**

寻找一些内存中没有使用的页换出去。内存的过度分配over-allocation会导致page fault调页后发现所有页都在使用。

使用dirty/modify位来减少页传输的开销，只有脏页才需要写回硬盘。

**基本页面置换过程：**

1.查找所需页在磁盘上的位置。2.查找空闲帧，如果有直接使用；如果没有就用置换算法选择一个victim，并将victim的内容写回磁盘，改变页表和帧表。3.将所需页读入新的空闲帧，改变页表和帧表。4.重启用户进程。

**页面置换算法**

采用最小页错误率的置换算法。

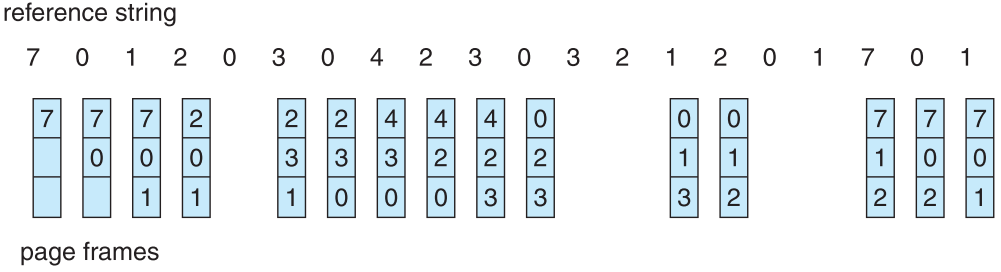
评估方法：

针对特定的内存引用序列，运行算法，计算出页错误数。引用序列叫做引用串reference string。

注意两个事实：给定页大小，只需要关心页码，不用管完整地址；紧跟页p后面对页p的引用不会引起页错误。

**First-In-First-Out Algorithm (FIFO，先进先出算法)**

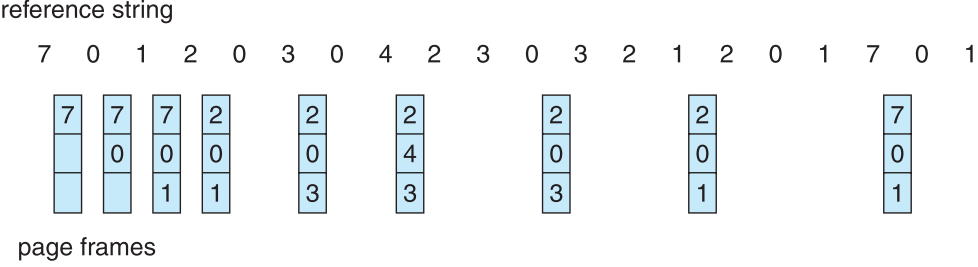
最简单的页面置换算法。必须置换一页时，选择最旧的。不需要记录时间，只需要FIFO队列来管理页即可。



FIFO会出现可用帧越多，错误数越大的问题，这种结果叫Belady’s Anomaly

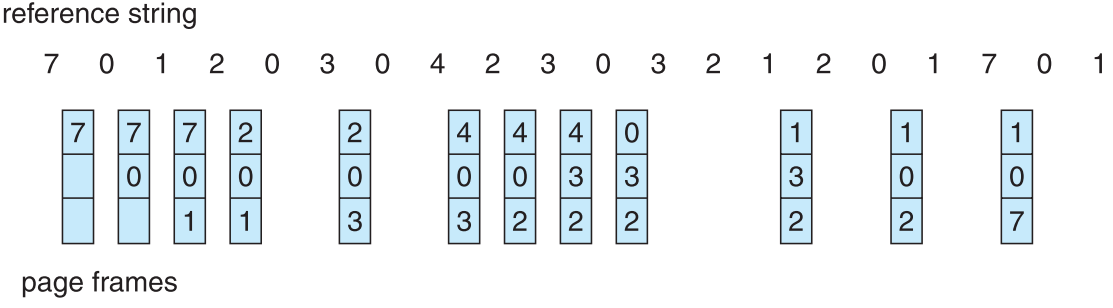
**Optimal Page Replacement OPT 最佳页面置换**

OPT时所有算法中页错误率最低，且绝对没有Belady异常。置换最长时间不会使用的页。或者说选择未来不再使用/在离当前最远位置上出现的页置换。这个使用时长看下一次该页号出现的距离即可。就是向未来看



**Least Recently Used LRU 最近最久使用**

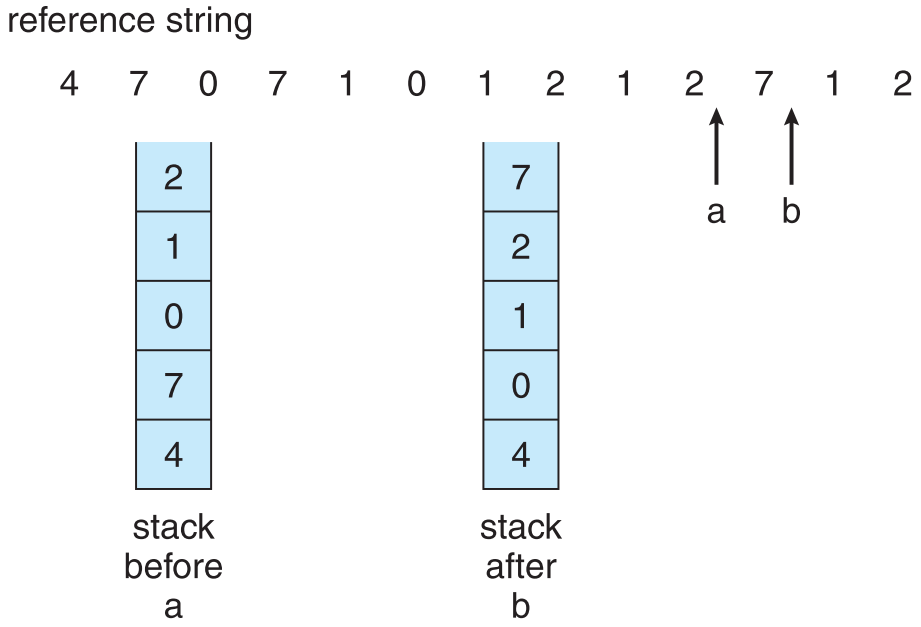
LRU选择内存中最久没有引用的页面，考虑的是局部性原理，性能最接近OPT，但是需要记录页面的使用时间，硬件开销太大。就是向后看的算法。



LRU算法如何获取多长时间没引用？两种方法：

计数器counter：每一个页表条目都有一个counter，每次被引用，就把时钟信息复制到counter。当置换时，置换时间最小的页，最近越使用，clock越大。

栈实现：维护一个页码栈，栈由双向链表实现。引用页面时将该页面移动到顶部，需要改变6个指针。替换时直接替换栈底部就是LRU页。



**LRU Approximation LRU近似**

很少有计算机有足够的硬件支持真正的LRU，因此许多系统为页表中的每项关联一个引用位reference bit，初始化为0，当引用一个页时，读写都可以，对应页面的引用位设为1。替换时替换掉引用位为0的(存在的话)，虽然我们不知道他有多老。

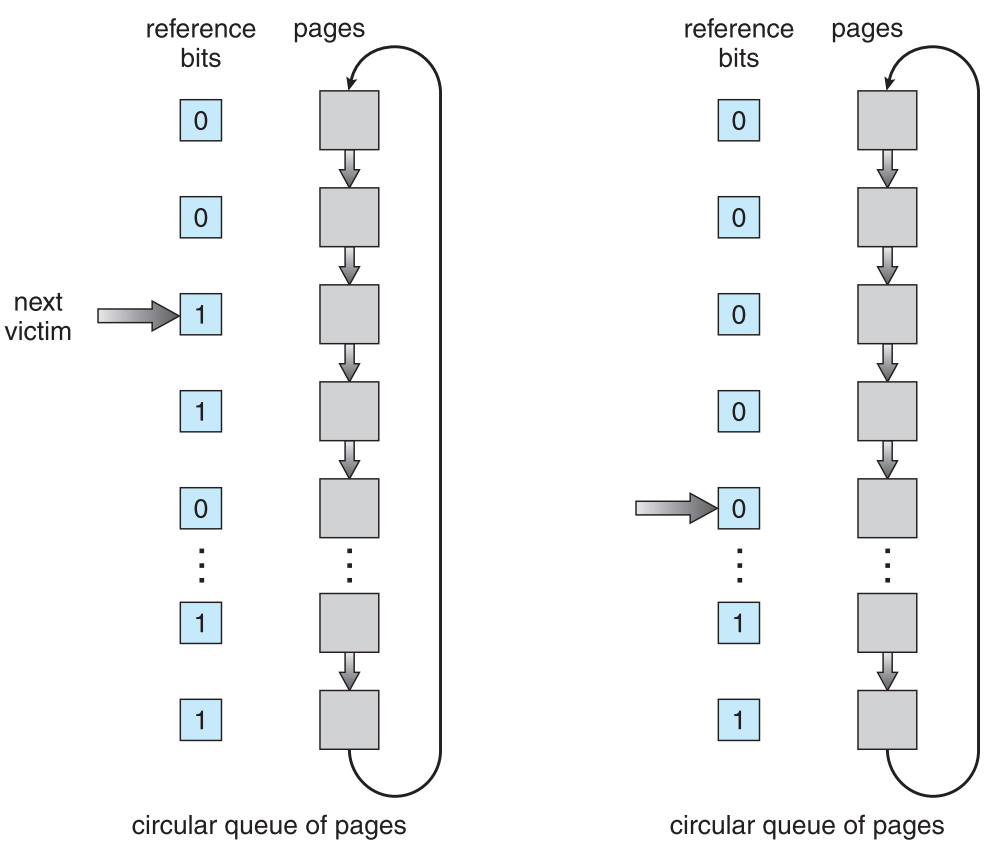
**Additional reference bits附加引用位算法**

在规定时间间隔内记录引用位。在规定的时间间隔内，时钟产生一个中断并且交控制权给OS，OS把每个页的引用位转移到其8位字节的高位，其他位向右移1位，抛弃最低位。这些8位寄存器包含着该页在最近8个周期内的使用情况，全0说明没用过，全1说明每个周期至少都用过1次，值越大越最近使用。有最小值的页是LRU页，被置换。被访问时左边最高位置1，定期右移并且最高位补0.

**Second chance 二次机会/clock算法 NRU**

基本算法是FIFO，选择页时，检查引用位，如果为0直接置换。如果为1，给该页第二次机会，选择下一个FIFO页。当一个页获得二次机会时，引用位清零，且将到达时间设为当前时间。因此，获得二次机会的页在所有其他页置换或获得二次机会之前，是不会被置换的。

一种实现二次机会算法的方法是采用循环队列，用一个指针表示下一次要置换哪一页。当需要一个帧时，指针向前移动知道找到一个引用位0的页，在其向前移动的过程中，它会清楚引用位。最坏情况下所有帧都会被给二次机会，他就会清除所有引用位之后再选择页进行置换，此时二次机会=FIFO。



**Enhanced Second chance 改进clock 增强二次机会**

通过将引用位和脏位作为有序对来考虑，可以改进二次机会算法。两个位有四种可能：

(0,0)无引用无修改，置换的最佳页

(0,1)无引用有修改，置换前需要写回脏页

(1,0)有引用无修改，很可能会继续用

(1,1)有引用有修改，很可能会继续用且置换前须要写回脏页

淘汰次序(0,0)-> (0,1)-> (1,0)-> (1,1)

当页面需要被置换时，使用时钟算法，置换(0,0)的页，在进行置换前可能要多次搜索循环队列。改进的点子在于给未引用但是修改了的页更高优先级，降低了IO数。

Macintosh使用

**Counting 基于计数的置换算法**

为每个页保存一个用于记录引用次数的计数器，具体方案有两种：

Least frequently used LFU：置换计数最小的。但是有问题：一个页可能一开始狂用，但是后来不用了，他的计数可能很大，但是不会被替换。解决方法是定期右移次数寄存器。

Most frequently used MFU：置换计数最大的，因为最小次数的页可能刚调进来，还没来得及用。

这两种很没用，实现开销很大，而且还很难近似OPT。

**Page Buffering 页面缓冲**

通过被置换页面的缓冲，有机会找回刚被置换的页。

被置换页面的选择和处理：用FIFO选择置换页，把被置换的页面放到两个链表之一。即：如果页面无修改，将其归入空闲页链表，否则归入已修改页面链表。

需要调入新页面时，将新页面内容读入空闲页面链表的第一项所指的页面，然后将其删除。

**帧分配 allocation of frames**

每个进程都需要最小数目的页。两种分配模式：

**平均分配算法 Equal allocation**

每个如果有100个帧 5个进程，每一个进程获得20个帧。

**按比例分配 Proportional allocation**

根据进程的大小按比例分配。

**优先级分配 Priority allocation**

同样按比例分配，但是是用优先级进行比例分配。

**全局置换 global allocation**

允许一个进程从所有帧中选择一个帧进行替换，不管该帧是否已分配给其他进程。

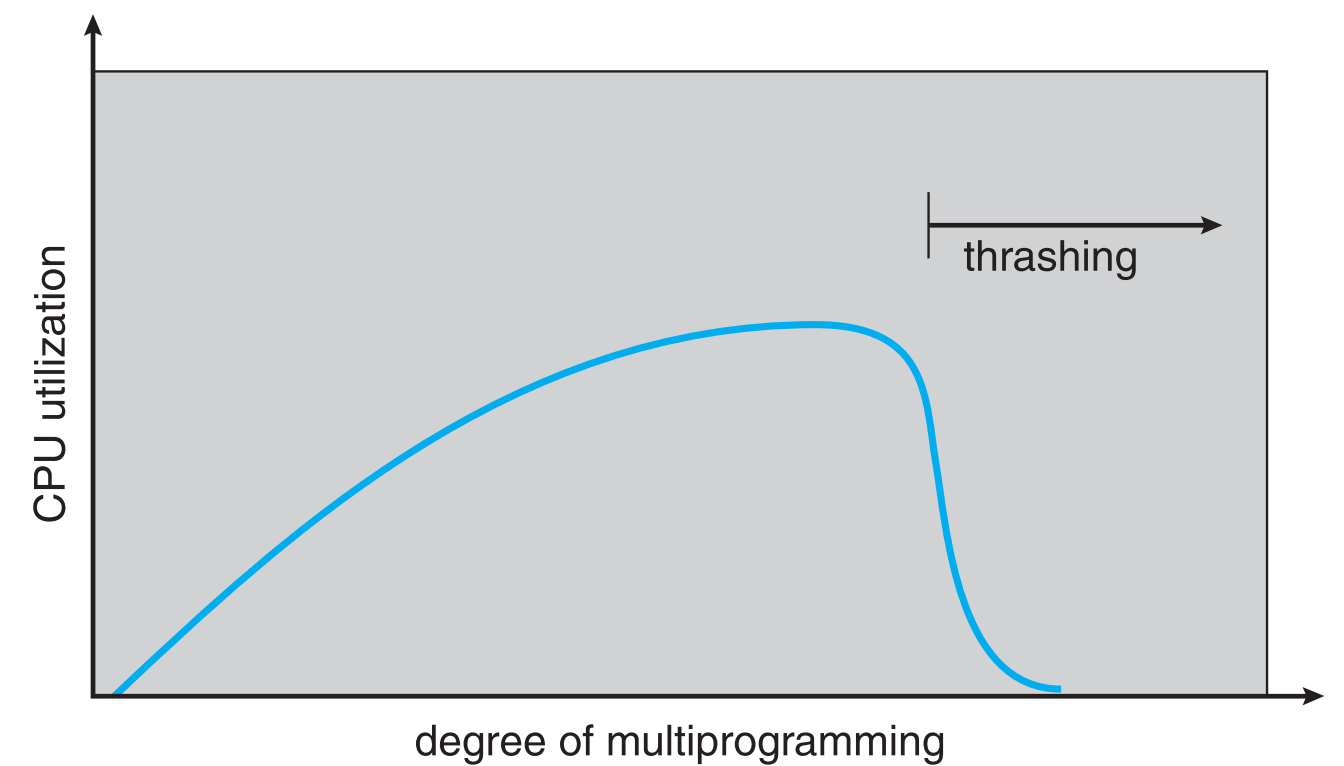
**局部置换 local allocation**

每个进程只能从自己的分配帧中进行置换选择。

固定分配局部置换 可变分配全局置换 可变分配局部置换

**Recaiming Pages** free pages数量降到一定值后提前选择一些page replaced掉。

**Major/Minor Page Fault** Major:访问的页不在内存中；Minor:访问的页在内存中（shared library；某页被reclaimed了但还没实际换出）

**颠簸 抖动 Thrashing**

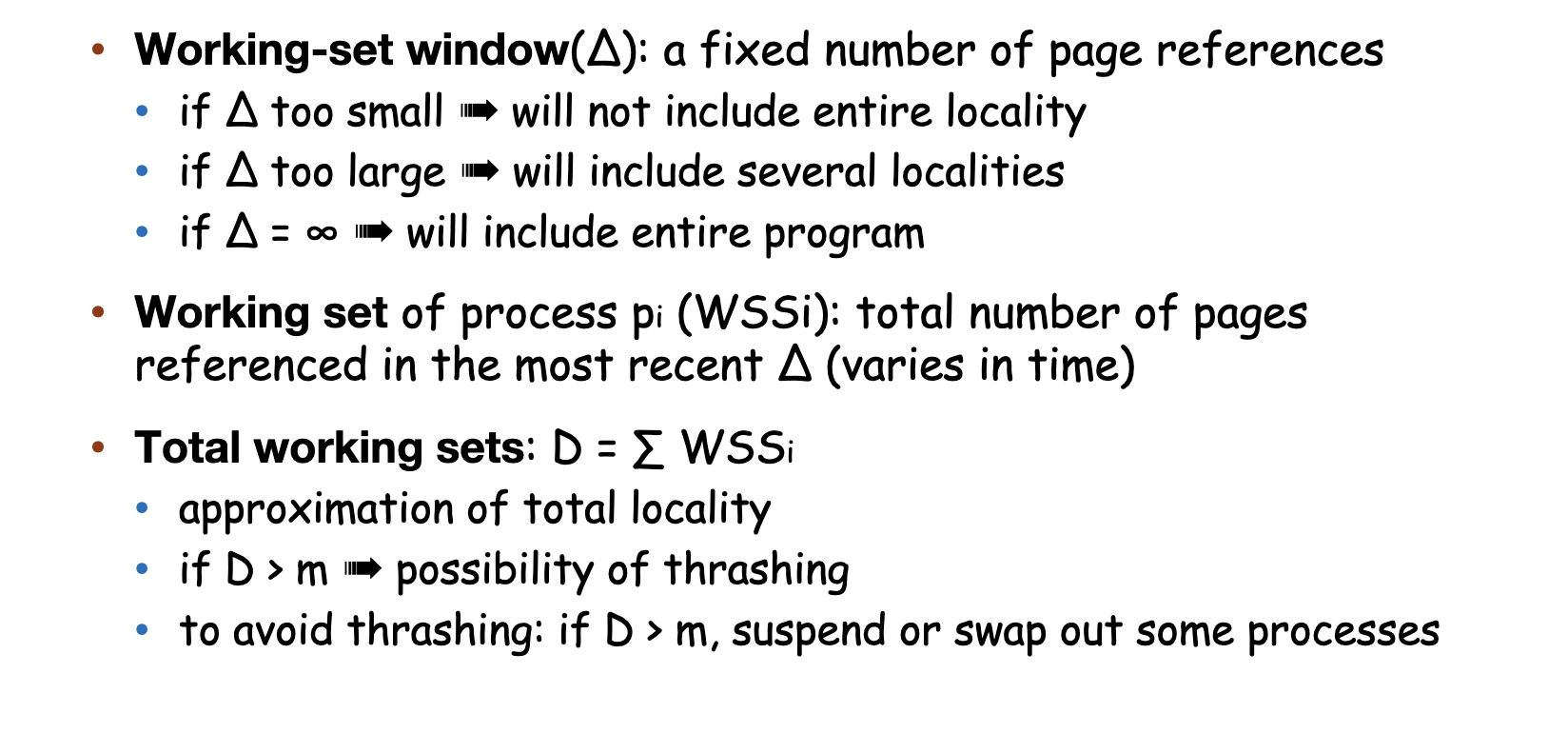
频繁的页调度行为叫做颠簸，会导致：CPU利用率低、OS认为多道程序程度需要增加、其他进程进入到系统中

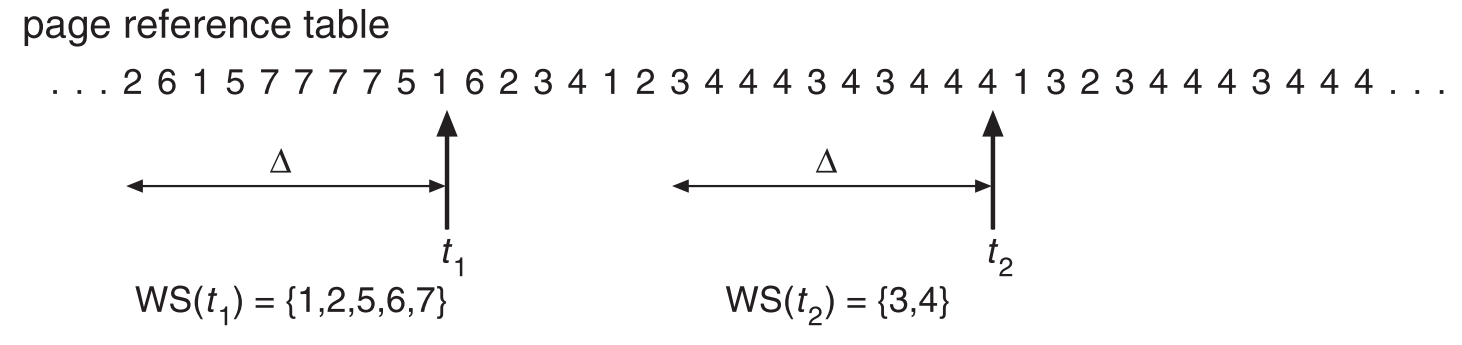
颠簸就等价于一个进程不断换入换出页

按需调页能成的原因是局部性原理，进程从一个局部性移动到另一个，局部性可能重叠。为什么颠簸会发生，因为局部大小大于总内存大小，不能将全部经常用的页放到内存中。

**工作集合模型 Working set model**

用来确定一个进程频繁访问的页面，保证这些页面不被换出；需要调页时从剩余的页面进行交换。如果频繁访问的页面数已经大于了当前进程可用的页面数，操作系统就应当把整个进程换出，以防止出现抖动现象。

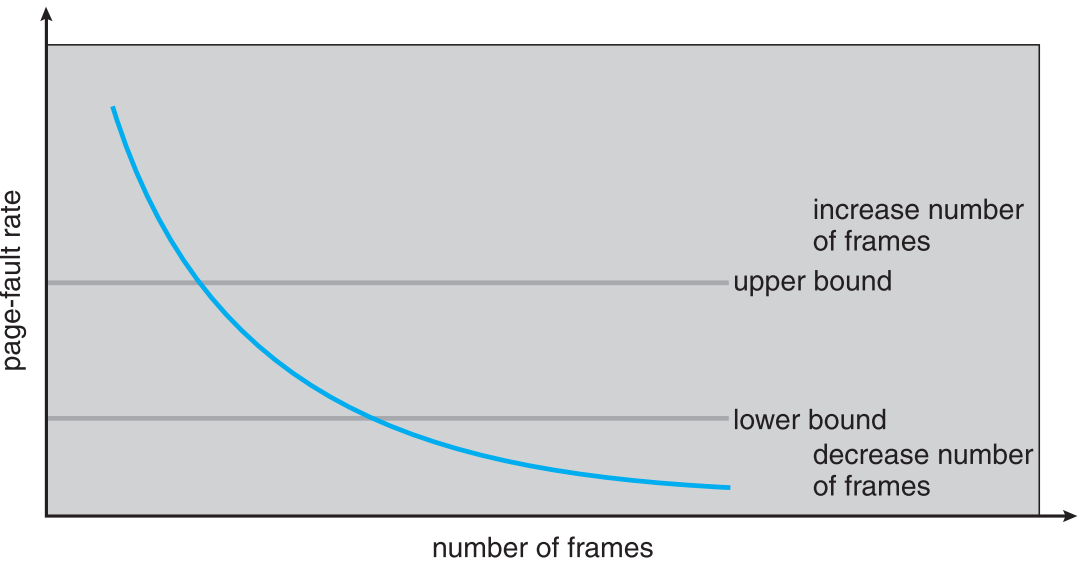




**页错误频率 Page fault frequency schema**

WS模型能用于预先调页，但是控制颠簸不是很灵活，更直接的方法是PFF。

可以为所期望的页错误设置一个上限和下限，如果页错误率超过上限，那么分配更多的帧，如果低于下限，那么可以从进程中移走帧。



**Memory-Mapped Files内存映射文件**

使用虚存技术来讲文件IO作为普通文件访问的技术。

开始的文件访问按照普通按需请求调度，会出现页错误。这样，一页大小的部分文件从文件系统中读入物理页，以后的文件访问就可以按照通常的内存访问来处理，这样就可以用内存操作文件，而非read write等系统调用，简化了文件访问和使用。多个进城可以允许将同一文件映射到各自的虚存中，达到数据共享的目的。

**Allocating Kernel Memory内核内存分配**

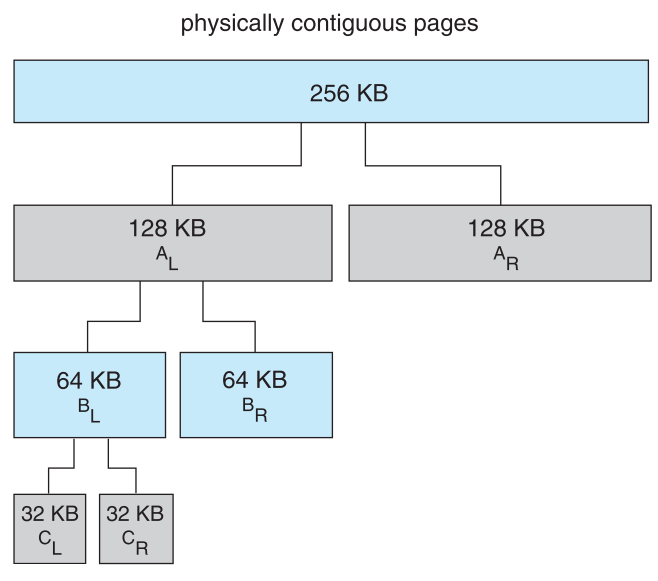
与对待用户内存不同；内核内存从空闲内存池中获取，两个原因：1.内核需要为不同大小的数据结构分配内存。2.一些内核内存需要连续。

**Buddy系统**

从物理上连续的大小固定的段上进行分配。

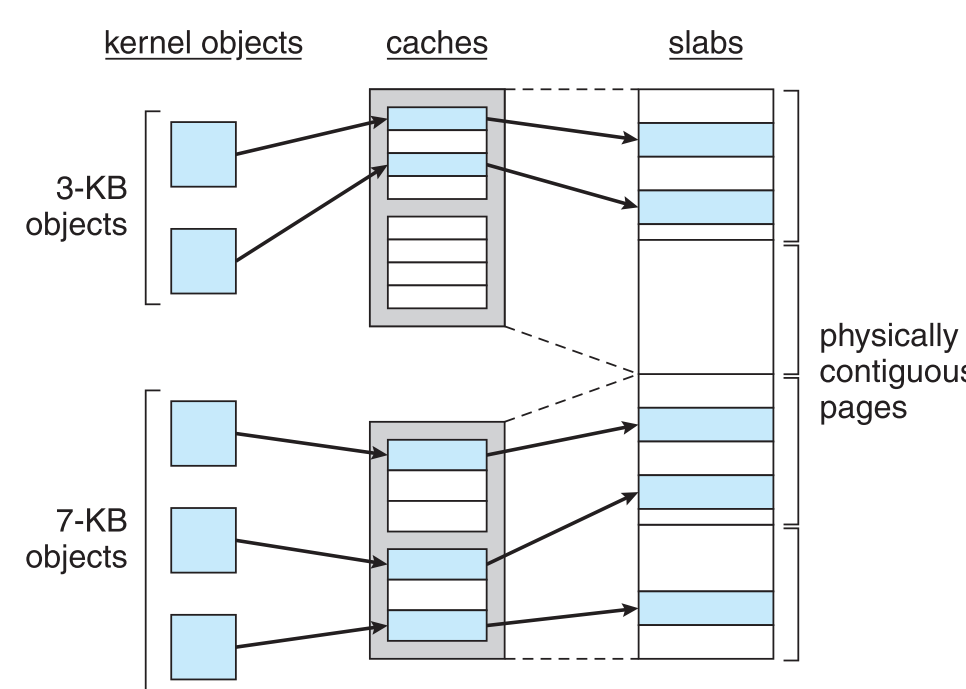
内存分配按2得幂的大小来分配：

请求大小必须是2的幂；如果不是，那么调整到下一个更大的2得幂；当需要比可用的更小的分配时，当前块分成两个下一个较低幂的段。继续这一过程直到适当大小的块可用。Buddy系统的优点是可以通过合并快速形成更大的段。明显缺点是由于调整到下一个2的幂容易产生内碎片。



**Slab分配**

为了解决Buddy碎片损失的问题，slab是由一个或多个物理上连续的页组成的。Cache包含一个或者多个slab。每个内核数据结构都有一个cache。每个cache都含有内核数据结构的对象实例。当cache被创建时，起初包括若干标记为空闲的对象。对象的数量和slab大小有关，12KB的slab(包含三个连续的页)可以存储6个2KB的对象。当需要内核数据结构的对象时，可以直接从cache上取，并将该对象标记为使用 used。Slab首先从部分空闲的slab中分配，如果没有则从全空的slab进行分配。如果没有，从物理连续页上分配新的slab，把他赋给一个cache，再从slab分配空间。Slab优点：没有碎片引起的内存浪费；内存请求可以快速满足。



**File System Interface 文件系统接口**

文件是存储某种介质上的（如磁盘、光盘、SSD等）并具 有文件名的一组相关信息的集合

**文件属性**

Name。这是唯一的以 human-readable 形式保存的信息。

Identifier。一个在当前文件系统中唯一的 tag（通常为 number），文件系统用它来标识文件。

Type。一些文件系统支持不同的文件类型。

Location。标识文件在哪个设备的哪个位置。

Size。当前文件大小，也可能包含文件允许的最大大小。

Protection。Access-control information。

Timestamp。保存创建时间、上次修改时间、上次使用时间等。

User indentification。创建者、上次修改者、上次访问者等。

所有的文件信息都保存在目录结构中，而目录结构保存在外存上。

**文件操作**

创建 写 读 文件内重定位 删除 截断truncate Open(Fi) 在硬盘上寻找目录结构并且移动到内存中 Close(Fi)将内存中的目录结构移动到磁盘中。

**打开文件**

每个打开文件都有以下信息：文件指针：跟踪上次读写位置作为当前文件位置指针

文件打开计数器file-open count：跟踪文件打开和关闭的数量，在最后关闭时，计数器为0，系统可以移除该条目。

文件磁盘位置disk location of file：用于定位磁盘上文件位置的信息

访问权限：访问模式信息

锁机制：mandatory lock:根据目前锁请求与占有情况拒绝access；advisory lock:进程查看锁情况来决定访问策略

**文件锁File Lock**

一些 FS 也会提供 文件锁 (file lock) 来协调文件访问，例如允许一个进程锁定文件来避免其他文件访问它。

文件锁类似 reader-writer lock。有的 FS 提供两种锁：共享锁 (shared lock) 类似于 reader lock，独占锁 (exclusive lock) 类似于 writer lock。有的 FS 只提供独占锁。

同时，文件锁也有两种可能的机制，由操作系统决定。一种是 强制锁定 (mandatory lock)，即一旦进程获取了独占锁，操作系统就阻止任何其他进程访问对应文件，即操作系统本身确保了完整性；另一种是 建议锁定 (advisory lock)，即进程可以自己得知锁的状态然后决定要不要坚持访问。Windows 使用前者，而 UNIX 使用后者。

**文件类型(扩展名)**

文件有不同的类型。操作系统可以选择是否需要识别不同的文件类型。

识别不同文件类型的方式之一是 文件扩展名 (file extension)；例如规定只有扩展名是 .com, .exe, .sh 的文件才能执行。

另外，也可以通过在文件开始部分放一些 magic number 来表明文件类型；但是由于不是所有文件都有 magic number，因此系统不能只基于这种方式判断文件的类型。

也可以采用的一种方式是，操作系统不试图识别文件的类型；当用户尝试运行一个文本文件，或者试图用文本编辑器打开一个二进制文件时，会照常按照预期的编码方式解析；但操作系统以及打开这个文件的应用程序本身的安全设计使得这样的操作不会带来太坏的后果。

UNIX 允许文件扩展名的提示，但是扩展名主要是帮助用户来确定内容类型；操作系统本身并不强制文件扩展名，也并不依赖这些扩展名。应用程序的开发者可以选择是否忽略文件扩展名。

**文件内部结构 File Structure**

文件可以有不同的结构，由操作系统或者用户程序定义。

无结构 (no structure) 文件，它只是 a stream of bytes or words。例如 UNIX operating system defines all files to be simply streams of bytes。

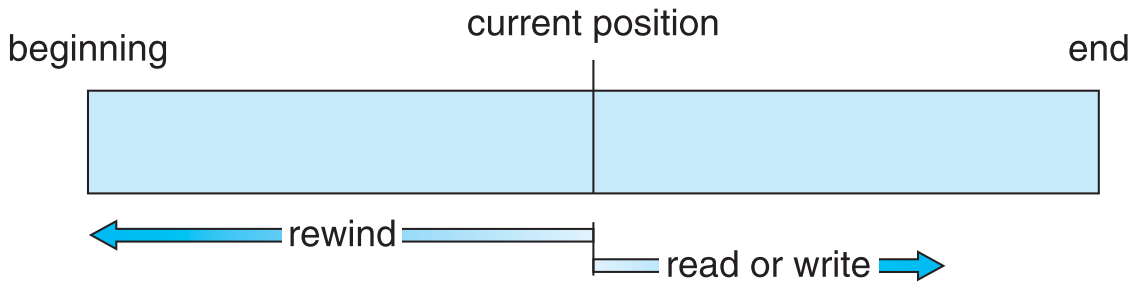
simple record structure 文件，以 record 为单位，每个 record 可以是定长或者变长的，例如数据库。

complex structures 文件。例如 Microsoft Word 文档。

**访问方法**

**Sequential access顺序访问**

文件信息按顺序，一个记录接着一个记录处理。访问模式最常会能够用，编辑器和编译器用这种方式。读操作读取文件下一文件部分，并自动前移文件指针，跟踪IO位置。写操作向文件尾部增加内容，相应文件指针到新文件结尾。顺序访问基于文件的磁带模型，也适用于随机访问设备。可以重新设置指针到开始位置或者向前向后跳过记录。No read after last write。

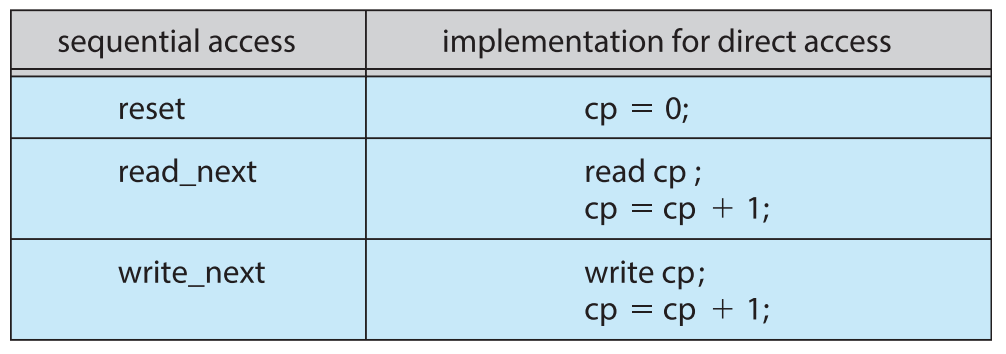


**Direct access直接访问/相对访问 (relative access) /随机访问 (random access)**

支持以几乎相同的时间访问任意位置

文件由固定长度的逻辑记录组成，允许程序按任意顺序进行快速读写，直接访问是基于文件的磁盘模型。文件可作为块或记录的编号序列。读写顺序没有限制。可以立即访问大量信息，DB常用。往往用指向blocks的index实现。

文件操作必须经过修改从而能将块号作为参数，有读n操作，而不是读下一个；写n操作；定位到n；要实现读n只需要定位n再读下一个即可。注意n是相对块号，相对于文件开始的索引号。



**Indexed block access索引顺序访问访问**

**目录结构**

目录是包含所有文件信息节点的集合。目录结构和文件在磁盘上。

**磁盘结构**

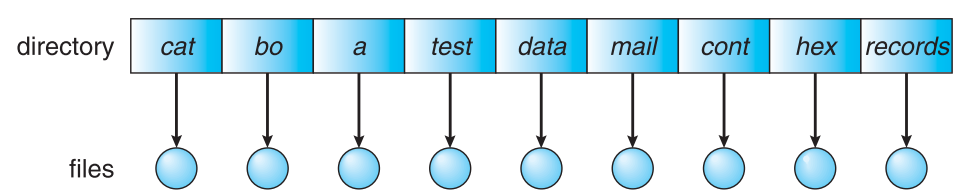
磁盘可以装多种文件系统，分区或片minidisk slice。

**目录操作**

搜索文件 创建文件 删除文件 遍历list目录 重命名文件 遍历traverse文件系统

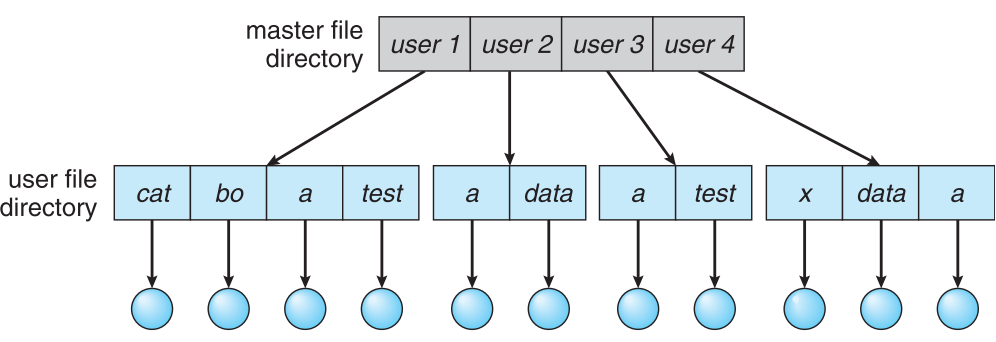
**单级目录**

所有文件包含在同一目录中，一个文件系统提供给所有用户。由于所有文件在同一级，不能有重名，此外存在着分组问题

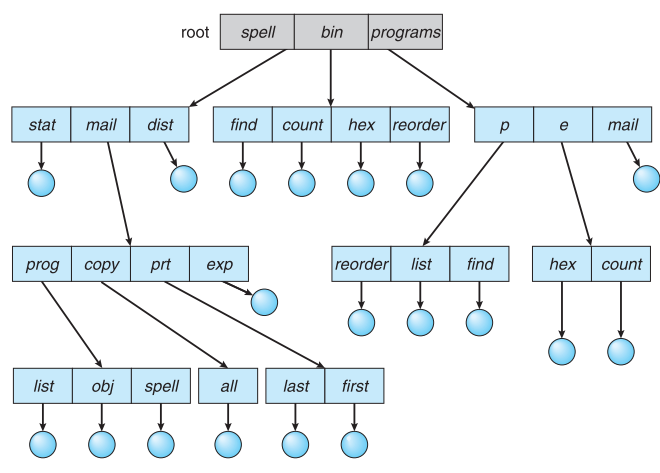


**二级目录**

为每个用户创建独立目录。每个用户都有自己的用户文件目录user file directory UFD。不同用户可以有同名文件，搜索效率高，但是没有分组能力。

**树形目录**

将目录二级目录拓展即可。搜索高效,有分组能力。



**无环图目录 Acyclic graph**

树形结构禁止共享文件和目录。无环图允许目录含有共享子目录和文件。

这个指针的实现方式之一是用链接文件来记录指向文件的绝对或相对路径，这种方式称为 符号链接 (symbolic link) 或者 软链接 (soft link)。当访问这种文件时，操作系统通过该路径名来 resolve 链接，从而定位真实文件。

这种方式存在的一个问题是，如果删除一个被某个 soft link 所引用的文件，则会留下 dangling pointer，即那些 link 现在指向并不存在的文件。UNIX 和 Windows 采用的方式是，当真实文件被删除时，并不试图处理这些 link；用户在试图访问这些 link 时会发现真实文件已经被删除。

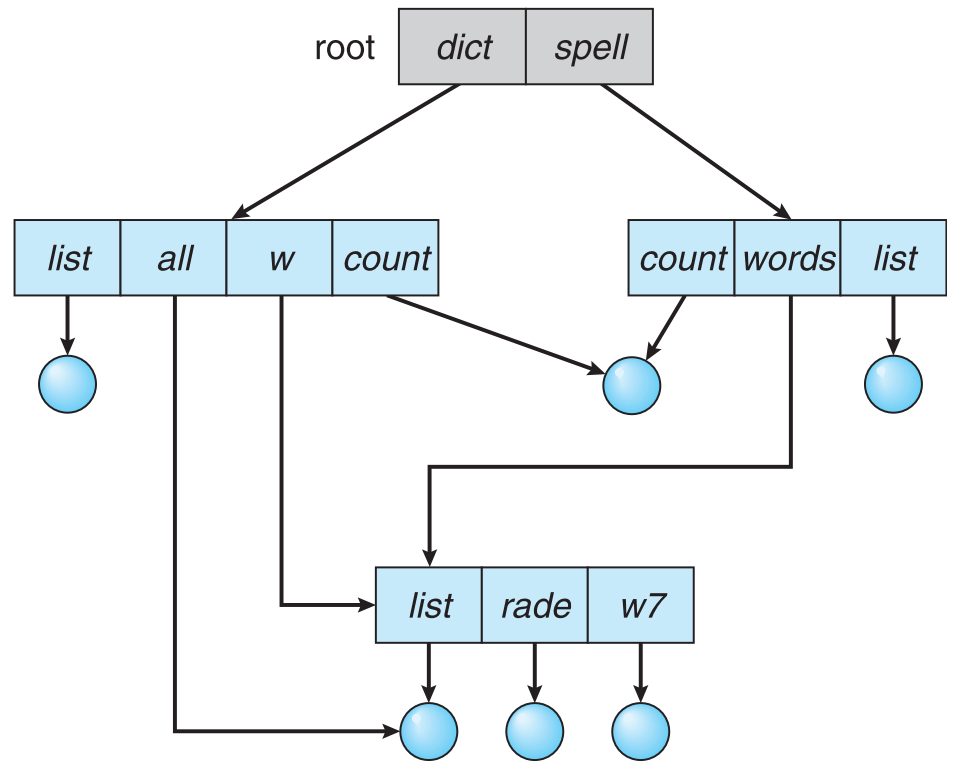
另一种实现方式是复制被引用文件的所有信息，放在当前目录中。由于这里的「信息」中也包含文件存储在外存中的哪个部分，因此事实上这个新的条目和被引用的条目指向的是同一个文件。这种方式称为 硬链接 (hard link)。这个链接和原文件本身相同且相等，因此可能没有办法区分它们。这种实现的主要问题是，在修改文件时要sdf维护一致性，即需要在所有包含这个文件的目录中更新对应的信息。

这种实现给每个文件引入了一个 引用计数 (reference counter)；在每个文件被创建一个新的硬链接时，该计数 +1；而删除一个硬链接或原文件本身时，该计数 -1。当计数为 0 时，文件可以删除。

UNIX 同时支持上述两种链接。为了维护无环图结构，hard link 不允许引用目录，而 soft link 可以。

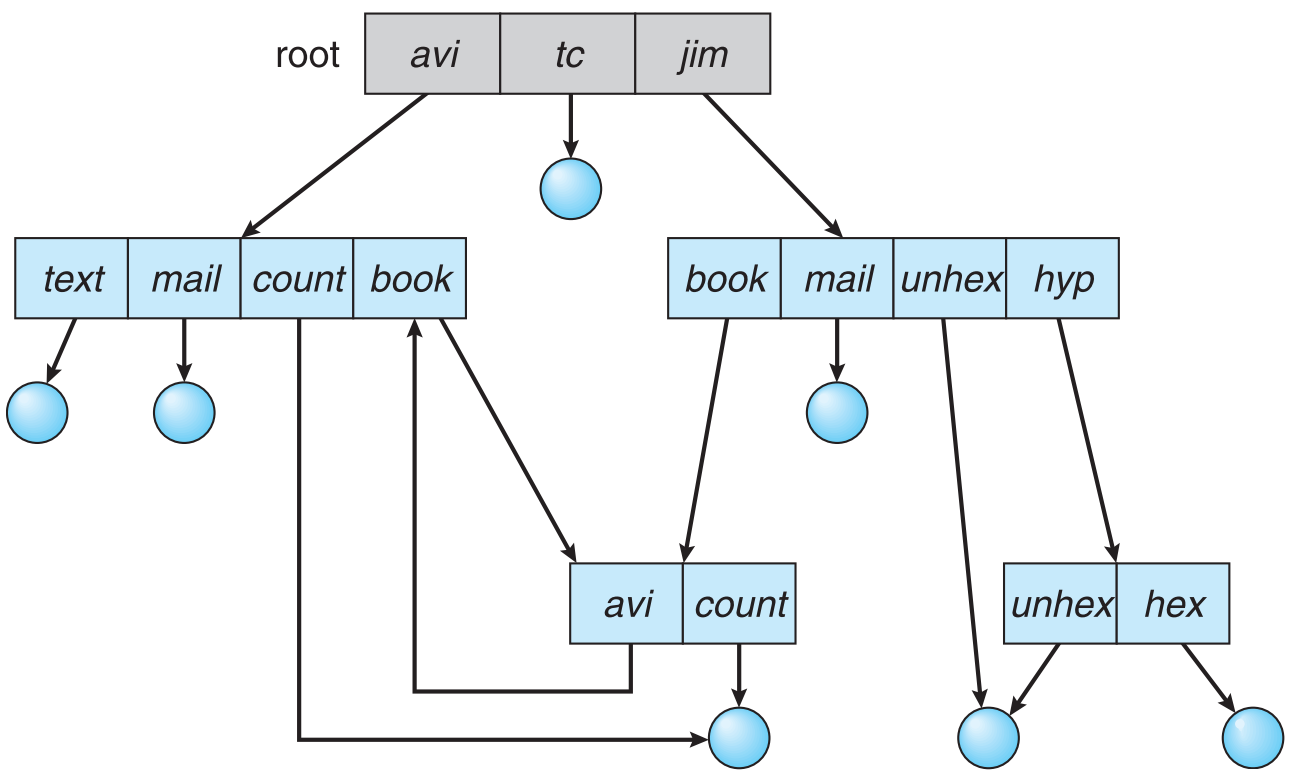
Hard link 是一个 directory entry，而 soft link 是一个 file。

Hard link 不能引用 directory，因为这会带来很多问题，但是 soft link 可以。一个例外是，每个目录中的 file name . 是指向自己的 hard link，而 .. 是指向父目录的 hard link。



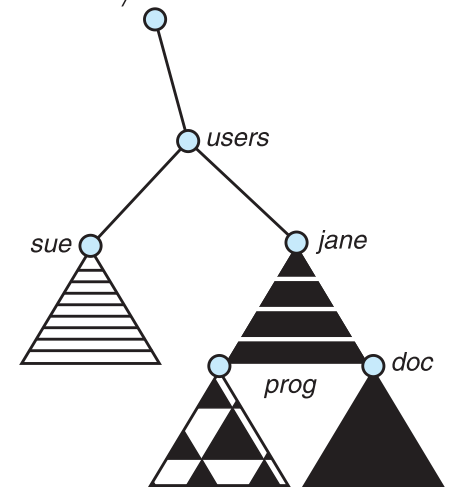
**普通图目录 General graph**

采用这种目录必须确保没有环，仅允许向文件链接，不允许目录，或者允许环，但使用gabage collection回收空间。每次加入链接都要执行环检测算法。



**文件系统挂载mount**

文件系统在访问前必须挂载。一个为挂载的文件系统会在挂载点mount point挂载。左图是未安装的卷，右图的users为挂载点。



**文件共享file sharing**

多用户系统的文件共享很有用。文件共享需要通过一定的保护机制实现；在分布式系统，文件通过网络访问；网络文件系统NFS是常见的分布式文件共享方法。NFS是UNIX文件共享协议 CIFS是WIN的协议。

**保护 Protection**

访问类型：读 写 执行 追加append删除 列表清单list

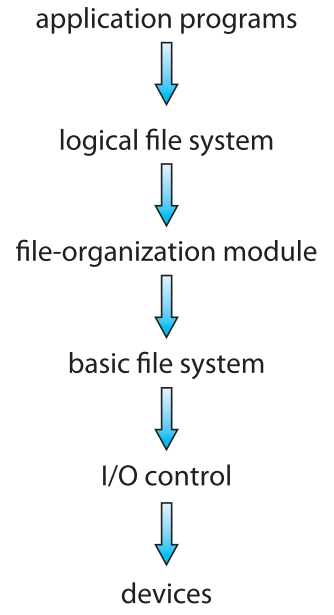
**访问控制列表access-control list ACL**

三种用户类型：

拥有者owner access 组 group access 其他 public access

在UNIX里，一个类型有rwx三个权限，所以一个文件需要3\*3=9位说明文件访问权限。

**File System Implementation文件系统实现**

文件系统：是操作系统中以文件方式管理计算机软件资源的软件和被管理的文件和数据结构（如目录和索引表等）的集合。文件系统储存在二级存储中，磁盘。

**分层设计的文件结构**

应用程序：发出文件请求者。

逻辑文件系统：管理所有文件系统所需的 metadata（即除了文件内容之外的所有东西），例如 directory structure。它通过一个叫做 File Control Block (FCB) 的数据结构保存这些信息，即我们之前所说的 name, ownership, permissions, ref count, timestamps, pointers to data blocks on disk 等。

文件组织模块知道文件逻辑块到物理块的映射，包括空闲空间管理器。做translation。

基本文件系统：管理包含 file-system, directory 和 data blocks 的缓存，从而提高性能。

如果上层的读取或写入 miss 了，那么 basic file system 将对应的指令传递给 I/O control

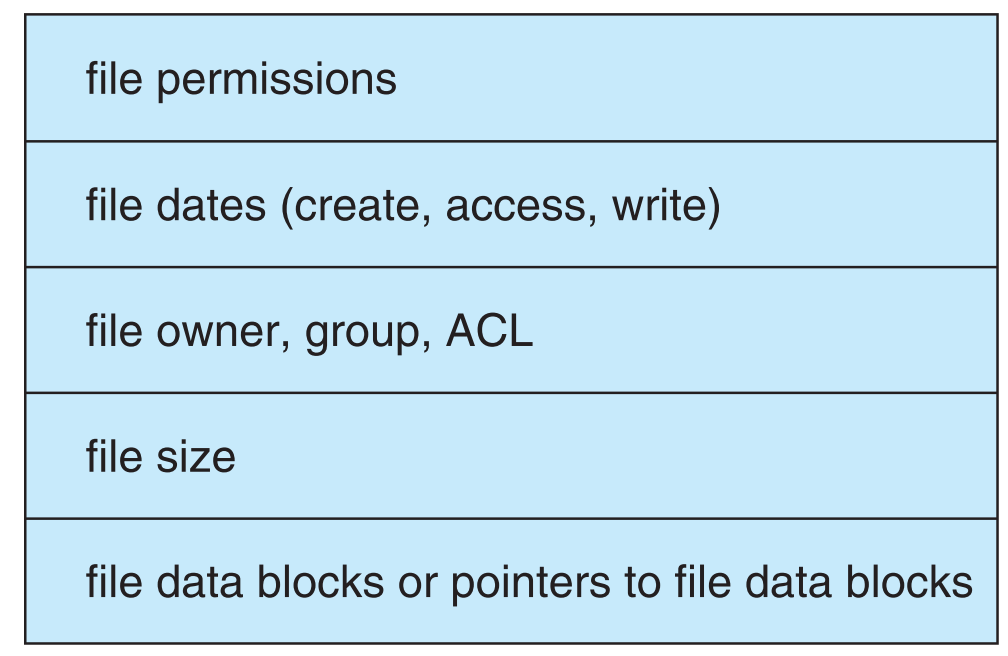
IO控制：由设备驱动程序和中断处理程序组成， 实现内存与磁盘之间的信息转移

**文件系统实现**

**On-Dist FS structure**

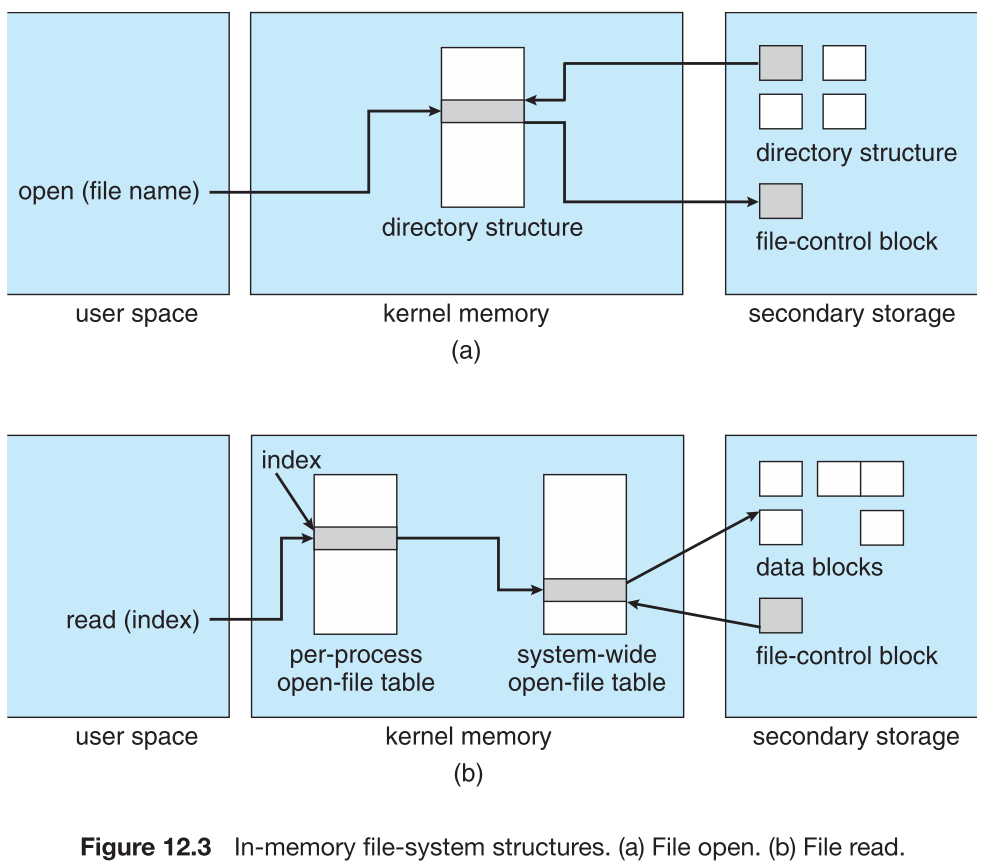
磁盘上，文件系统可能包括：如何启动所存储的OS，总块数，空闲块数目和位置，目录结构及各个具体文件。

磁盘结构包括：每个卷的引导控制块boot control block 包括从该卷引导操作系统所需的信息；每个卷的卷控制块volume control block包括卷的详细信息；目录结构来组织文件；每个文件的FCB(inode)。Directory (per FS)。用来组织文件，包含文件名和对应 FCB 的标识号。



**In-Memory FS structure**

In-mrmory patition table分区表 in-memory directory structure目录结构 system-wide open-file table系统打开文件表 per-process open-file table进程打开文件表



系统调用 open() 将文件名传给 logical file system，后者搜索 system-wide open-file table 以确定该文件是否正在被其他进程使用：

如果不是，则需要在 directory 中找到这个 file name，并将其 FCB 从磁盘加载到内存中，并将其放在 system-wide open-file table 中。然后，在当前进程的 per-process open-file table 中新建一个 entry，指向 system-wide open-file table 中的对应项。

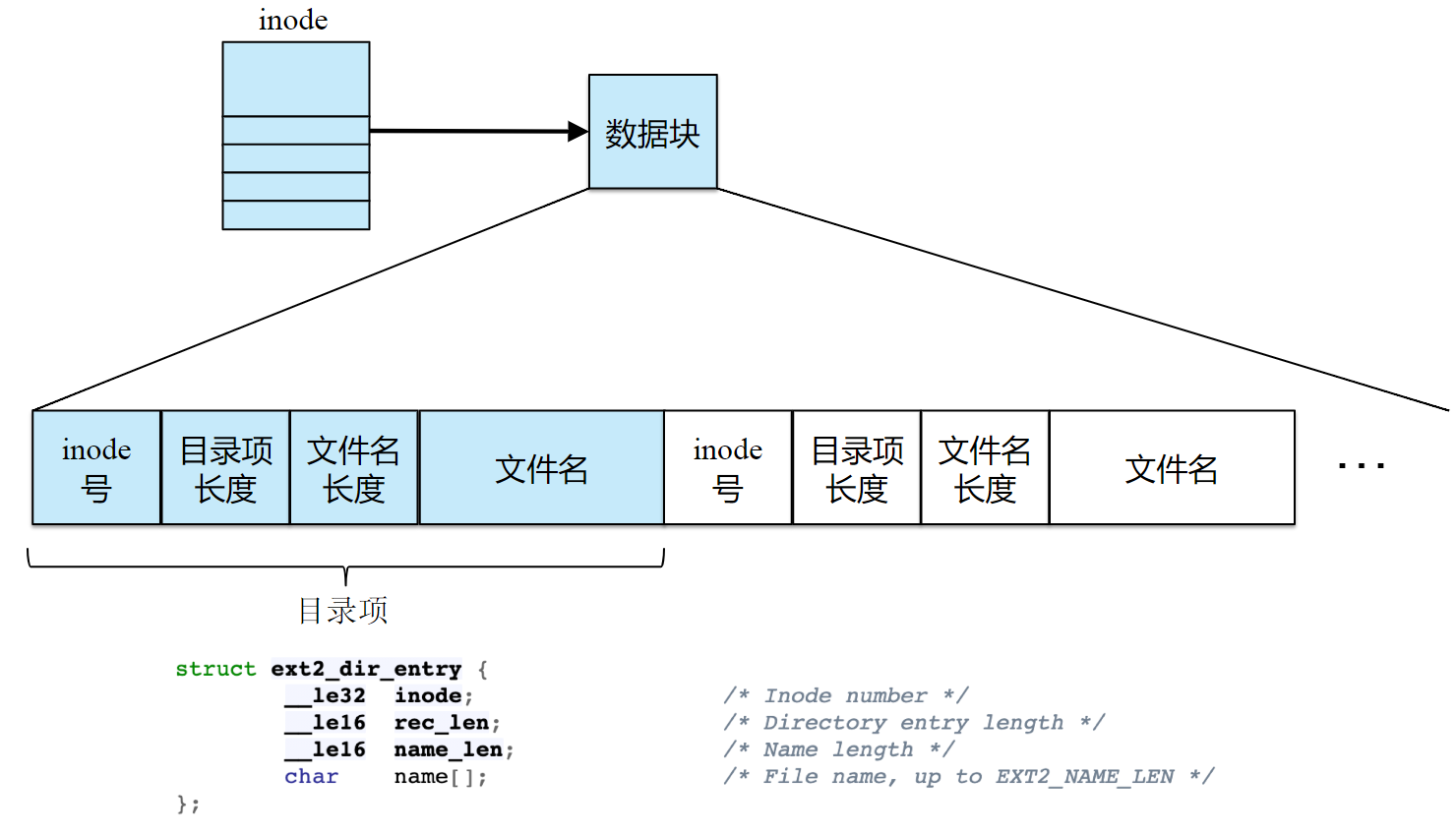
如果是，则直接在当前进程的 per-process open-file table 中新建一个 entry，指向 system-wide open-file table 中的对应项即可。这可以节省大量开销。

**虚拟文件系统VFS**

VFS提供面向对象的方法实现文件系统。允许将相同的系统调用接口（API）用于不同类型的文件系统。(Write syscall -> vfs\_write)

**目录实现**

目录是一种特殊的文件，它保存 file name 到 FCB 的映射关系。

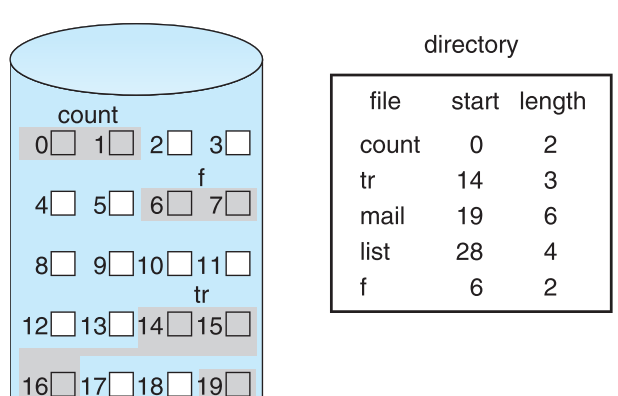


**分配方法Allocation Method**

常见的主要磁盘空间分配方法：连续、链接和索引。

**连续分配Contiguous Allocation**

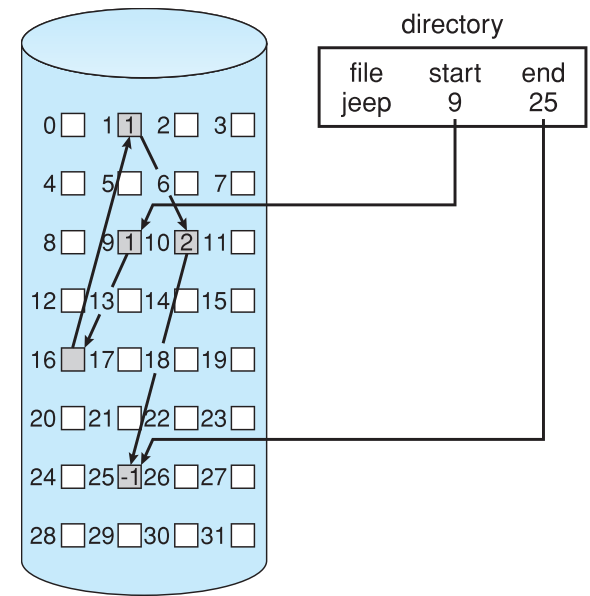
每个文件在磁盘上占有一组连续的块。优点：访问很容易，只需要起始块位置和块长度就可以读取。支持random access。但是浪费空间，存在动态存储分配问题。First和best表现差不多，first时间快很多。存在外碎片问题，此外文件大小不可增长。



变种：基于长度的系统。利于Veritas FS采用。解决了文件大小无法增长的问题，增加了另一个叫做extent的连续空间给空间不够的文件，然后与原文件块之间有个指针。一个文件可以有多个extent。FCB 记录的信息除了起始和长度外，还额外维护下一个 extent 的信息。

**Linked Allocation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。访问起来只需要一个起始地址。没有空间管理问题，不会浪费空间，但是不支持random access。

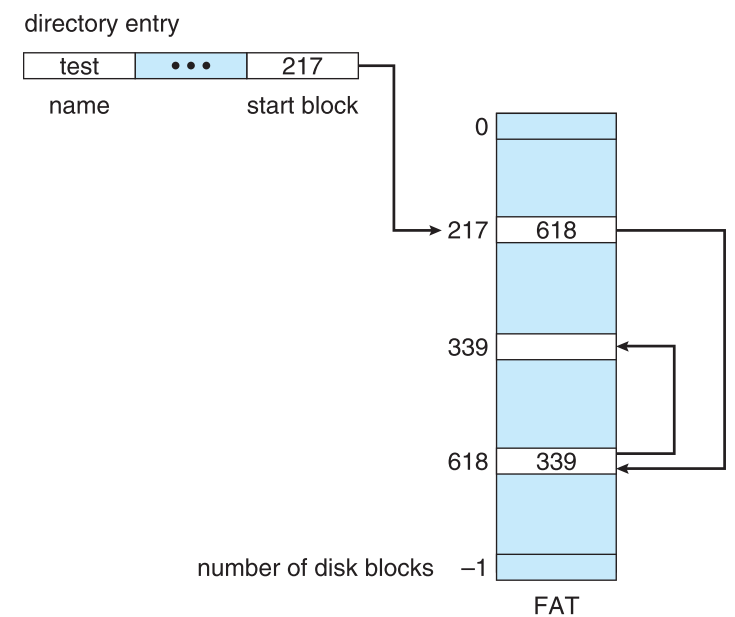


**FAT File allocation table文件系统**

基于linked allocation

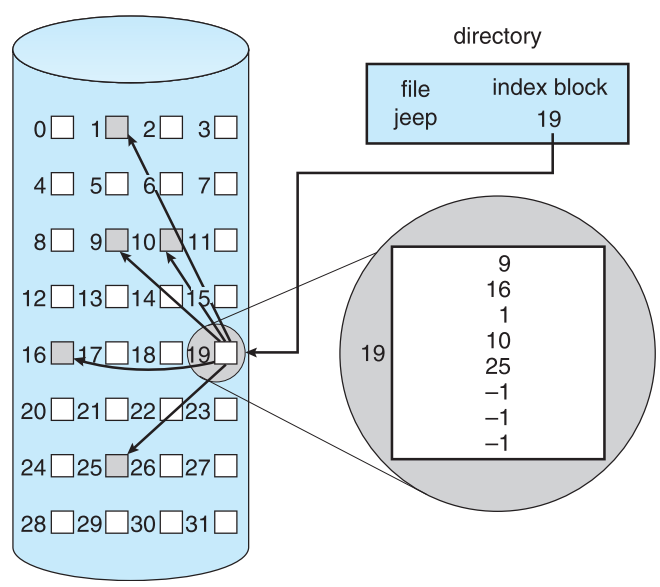
磁盘空间分配用于MS-DOS和OS/2。FAT32引导区记录被扩展为包括重要数据结构的备份，根目录为一个普通的簇链，其目录项可以放在文件区任何地方。原本的链接分配有问题，指针在每个块中都会占空间，可靠性也不高，任何指针丢失都会导致文件其余部分丢失。FAT采用单独的磁盘区保存链接。

计算机系统启动时，首先执行的是BIOS引导程序，完成自检， 并加载主引导记录和分区表，然后执行主引导记录，由它引导 激活分区引导记录，再执行分区引导记录，加载操作系统，最 后执行操作系统，配置系统。

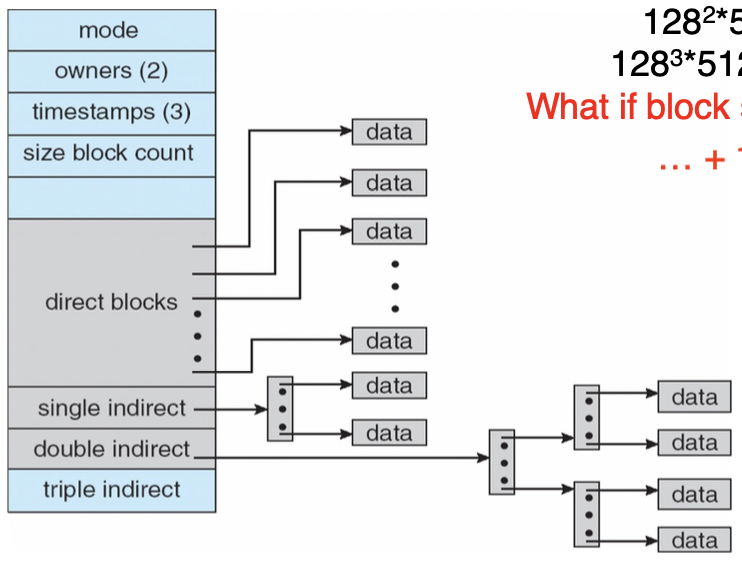


**Indexed Allocation索引分配**

索引分配把所有指针放在一起，通过索引块解决这个问题。每个文件都有索引块，是一个磁盘块地址的数组。当首次写入第i块时，先从空闲空间管理器获得一块，再将其地址写到索引块中的第i个条目。对于小文件，大部分索引块被浪费。如果索引块太小，可以多层索引、然后互相连接。访问需要索引表，支持random access，动态访问没有外碎片，但是有索引开销。



**Inode结构**

****

**索引计算**

连续、链接组织的最大文件大小可以大到整个磁盘文件分区。

考虑每块大小4KB，块地址4B。

一级索引：一个索引块可以存4KB/4B=1K个索引地址，每个索引地址直接引到文件块，所以最大1K\*4KB=4MB。

二级索引：一个索引块可以再继续连接到索引块，因此有1K\*1K\*4KB=4GB的最大文件。

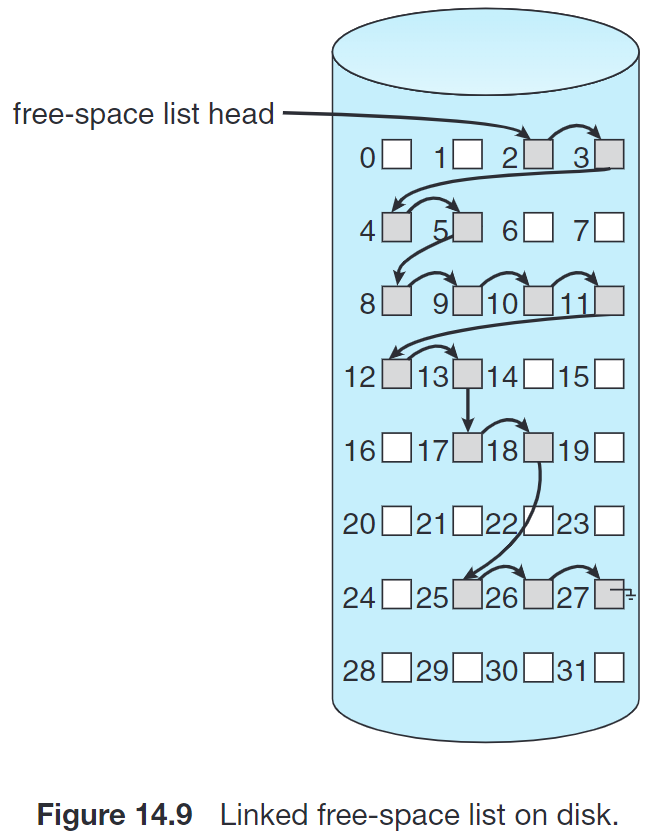
采用Linux分配方案，Linux中共有15个指针在inode中，前面12个直接指向文件块，因此有48KB可以直接访问，其他三个指针指向间接块，第一个间接块指针是指向以及间接块，第二个是二级间接块，第三个是三级间接块。因此最大文件的大小为：

12\*4KB+1K\*4KB+1K\*1K\*4KB+1K\*1K\*1K\*4KB=48KB+4MB+4GM+4TB

**空闲空间管理**

实现方案之一是用 bitmap：每个 block 用 1 bit 表示：1 表示空闲，0 表示占用。为了减少 bitmap 占用的空间，可以以 cluster 为单位分配和表示。

采用链表管理：将所有的空闲块链接起来，将指向第一个空闲块的指针保存在磁盘的特殊位置并且还存在内存中。但是IO效率很低，因为需要遍历。



为了解决IO效率低的问题，引入 grouping 的方案。维护若干个 block 形成的链表，每个 block 保存若干空闲块的地址。这种方案可以减少需要的 I/O。

另外也可以使用 counting 的方案。之前链表的方案维护的是空闲块的链表，而现在维护的是 连续的空闲块 的链表，即链表的每个结点是连续的空闲块的首块指针和连续的长度。

**页面缓冲page buffer**

将文件数据作为页而不是磁盘块缓冲起来到虚存。

**恢复Revoverry**

一致性检查：将目录结构数据与磁盘数据块比较，并且纠正发现的不一致。

用系统程序将磁盘数据备份到另一个设备。然后从该设备恢复。

**日志结构的文件系统**

日志文件系统记录文件系统的更新为事务。事务会被写到日志里。事务一旦写入日志就是已经commit了，否则文件系统酒还没更新。

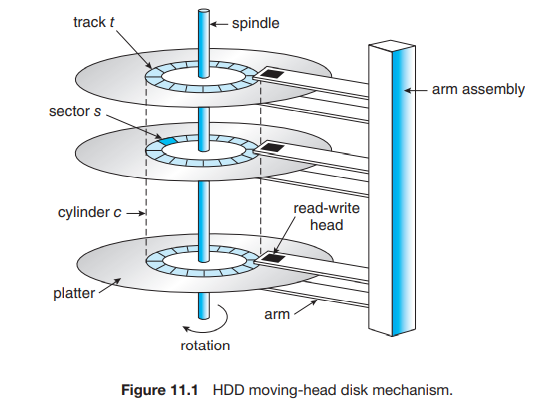
**Mass storage system大容量存储**

现代计算机的大部分二级存储由 hard disk drives (HDDs) 和 nonvolatile memory (NVM) devices 提供。

HDD 就是硬盘驱动器。NVM 设备根据定义是断电后存储的数据还能保持的设备

磁盘的0扇区是最外面的第一个磁道的第一个扇区。逻辑块时最小传出单位512B

**磁盘调度**

****

Seek time 寻道时间，磁头移动到包含目标扇区的柱面的时间。旋转延迟rorational latency：旋转到目标扇区的时间（1/rpm\*60,average latency=1/2\*latency）。传输时间transfer time：数据传输时间

磁盘带宽是传递的总字节数初一服务请求到传递结束的总时间。

平均旋转延迟=一圈/2

FCFS先来先服务：算法公平，但不是最快。

SSTF最短寻道时间优先：处理靠近当前磁头位置的请求，本质上和SJF一样，所以有可能请求会永远无服务(starvation)，时间也不是最优。

SCAN：从磁盘一端到另一端，对所有路上经过的柱面进行服务。到达另一端时改变移动方向，继续处理，也叫做电梯算法。

C-SCAN磁头从一端移动到另一端，到了另一端就马上返回到磁盘开始，返回路径中不服务。

LOOK：磁头从一端到另一端，到达另一端最远的服务就不继续走了，开始折返服务。

C-LOOK：磁头从一端到另一端，到达另一端最远服务就立即返回到磁盘开始的第一个服务，返回路径不服务。

调度算法选择：

SSTF一般来说比较好

LOOK C-LOOK对于高负荷IO磁盘表现更好

表现依赖于请求类型和数量；磁盘请求又依赖于文件分配策略；磁盘调度算法应该模块化，可以随时更换自由选择。SSTF或者LOOK都是很棒的默认算法。

**磁盘管理**

低级格式化/物理格式化：将磁盘划分为扇区才能进行读写。逻辑格式化：创建文件系统。

要是用一个磁盘保存文件，OS需要这么几步：首先分区，然后逻辑格式化，也就是创建文件系统；为了提升效率然后将块集中到一起成为簇cluster。

**引导程序**

一般bootstrap存在ROM里。系统启动顺序：ROM中的代码(simple bootstrap)boot block里的代码(full bootstrap)也就是boot loader如Grub LILO然后是整个OS内核

**RAID**

**Redundant Array of Independent Disks**

0：无冗余 1：镜像 2：海明纠错 3：按bit对每个盘进行奇偶校验，结果放在1个盘 4：与3类似，按块条带化Striping 5：校验值分散到各个盘 6：P+Q冗余，差错纠正码



**IO system**

**IO方式**

轮询 (polling)：当 PC 需要和某个 I/O 设备交互时，它查询该设备对应的 I/O 控制器 (controller) 中状态寄存器的相关位，当该位表示事件（如写入或读取）可以进行时 PC 进行相关操作并通过状态寄存器通知控制器。

中断：是更高效和常用的 I/O 手段，它通过在 interrupt-request line 上发出信号来通知 CPU 响应设备的 I/O 事件。

对于需要大量传输的设备，上面两种操作会过多占用 CPU 资源，因此很多计算机提出了 直接内存访问 (DMA, Direct Memory Access)。这一技术将相关的 I/O 任务交给一个专用的处理器（DMA 控制器）来完成，设备直接与内存交互。

对于需要进行大量IO的设备，为了避免程序控制IO 即PIO，将一部分任务细分给了DMA控制器，在DMA开始传输时，主机向内存中写入DMA命令块。然后CPU在写入后继续干别的，DMA去自己操作内存总线，然后就可以向内存进行传输。

**IO分类：**block I/O(read, write, seek); character I/O (stream, keyboard, clock); memory-mapped file access; network sockets

**IO应用接口**

实现统一的IO接口，设备驱动提供了API来操控IO设备(Linux：最底层为**ioctl**)

设备分成很多种：

Sequential or random-access 顺序或随机访问设备

Sharable or dedicated共享或独占设备

Speed of operation Operating System jjm 操作速度（快速、中速、慢速）

read-write, read only, or write only 读写、只读、只写设备

块设备和字符设备(block/character):

块设备：包括硬盘，一般有读写seek的命令，对其进行raw原始IO或者文件系统访问。内存映射文件访问也OK

字符设备：键盘鼠标串口，命令是get put。库函数提供具有缓冲和编辑功能的按行访问。

Synchronous I/O：包括blocking IO和non-blocking I/O

阻塞IO:进程挂起直到IO完成，很容易使用和理解，但是不能满足某些需求

非阻塞IO：IO调用立刻返回尽可能多的数据。用户接口就是，接收鼠标键盘输入，还要在屏幕上输出，放视频也是，从磁盘读帧然后显示。

Asynchronous I/O：IO与进程同时运行。

非阻塞和异步的区别：非阻塞的read会马上返回，虽然可能读取的数据没有达到要求的，或者就没读到。

异步read一定要完整执行完

