概述

操作系统作为用户和硬件之间的中间人

操作系统目标：①用户需求：易学易用，可靠性高，速度快②系统需求：易于设计实现维护，灵活可靠错误少高效

计算机系统：硬件、OS、应用程序和用户

操作系统是一组有效控制和管理计算机各种硬件和软件资源，合理地组织计算机的工作流程，以及方便用户的程序的集合

操作系统是用户与计算机硬件之间的接口，分为：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

OS 有：批处理系统、分时系统。批处理可以分为多道和单道两种。多道程序：多个作业保存在内存中，CPU 在它们之间进行切换。发展：no software→resident moniters→muli-programming->multi-tasking。

Time Sharing system 分时系统

分时系统是多个用户分时共享，把系统资源分割，每个用户轮流使用一个时间片。

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 *I/O 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 external events.

Exceptions: Caused by executing instructions.

操作系统结构

操作系统服务：

①用户界面：CLI GUI②程序执行③I/O 操作④文件系统操作⑤通信⑥错误检测⑦资源分配⑧统计⑨保护和安全

操作系统的用户界面

操作系统接口：①命令接口：CLI/GUI②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

常见 API: win32 API&POSIX API&JAVA API

Time()是一个系统调用

系统调用传参方式：①寄存器传参②参数存在内存块和表中，将块和表地址通过寄存器传递，linux 和 solaris 用这种③参数通过堆栈传递

strace -xf [command]: 查看某命令执行过程中发生的所有系统调用情况

Types of syscalls: Process control, File management, Device management, Information maintenance, Communications, Protections

ELF: .text: code, .rodata: initialized read-only data, .data: initialized data, bss: uninitialized data (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。 Windows NT windows 8 10 mac OS L4

To make the system more efficient is not an advantage that the microkernel structure has compared with themonolithic structure

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。如 OS/360，VMS Linux

模块 modules

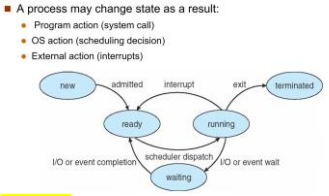
大多数现代操作系统都实现了内核模块。面向对象，内核部件分离，通过已知接口进行沟通，都是可以载入到内核中的。总而言之很像层次结构但是更加灵活。Linux solaris。

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

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

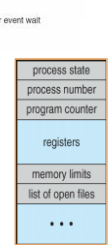


进程状态

单处理器下最多一个 run，就绪进程构成就绪队列，等待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

PCB

①进程状态②pc③寄存器④CPU 调度信息⑤内存管理信息⑥技术信息⑦文件管理⑧I/O 信息



Linux 的 PCB 保存在 struct task_struct 里

Fork:

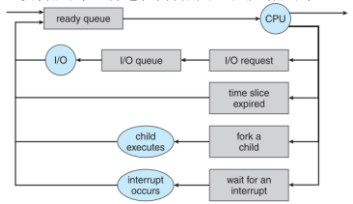
wait(): block until any child terminates. waitpid(): block until a specific child completes

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

Zombies&Orphans: Zombie die until its parent has called wait() or its parent dies. Orphan is "adopted" by the process with pid 1.

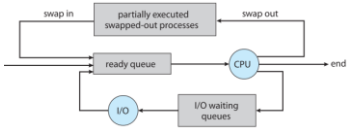
进程调度

调度队列有：进程队列(Job Queue)&就绪队列&设备队列，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。进城可以被换出，之后再换入。增加了中程调度的队列图



进程可以分为 I/O 型和 CPU 型(CPU-bound)

上下文切换：进程上下文储存在 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。

合作进程：独立进程：运行期间不会受到其他进程影响。**进程合作优点**：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

共享内存 Shared memory: shm_open(): creates a shn segment; mmap(): memory-map a file pointer to the shared memory object; Reading and writing to shared memory is done by using the pointer returned by mmap().

消息传递 Message passing: send(Q, msg), recv(Q, msg). 实现通信 link：通过 mailbox 接受信息（create mailbox, destroy mailbox）

Message synchronization: blocking – synchronous，non-blocking: asynchronous

Signals, Pipes: Ordinary pipes – unidirectional, parent-child relationship (fd[0] is the read end; fd[1] is the write end); Named pipes: bidirectional

RPC remote procedure calls

Thread 线程

进程是资源拥有单位，线程是调度单元

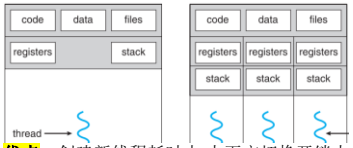
线程是进程内一个执行单元或可调度的实体

重量型/传统线程=单线程的进程

线程只拥有其在运行中必不可少的资源（**程序计数器、一组寄存器和栈**），但它可与同一进程中的其它线程共享进程拥有的全部资源

共享：①进程地址空间②代码段③数据段④堆

私有：①栈②线程局部存储(TLS)③寄存器



优点：创建新线程耗时少,上下文切换开销小(no cache flush),线程间通讯可通过共享内存;responsiveness, scalability **缺点**：weak isolation; 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提供创建&同步&调度&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，整个进程等待

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：多用户级线程对应一个内核线程。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，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，*win NT/XP/2000 linux Solaris 9 later*。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。IRIX HP-UX Tru64 Solaris 8 earlier

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）

线程取消：Asynchronous：立即终止目标线程;**Deferred**：目标线程不断检查是否该终止

信号处理：信号由特定事件产生，必须发送给进程，被发送后需要被处理①发送信号到信号产生的线程②到进程内的每个线程③进程内特定线程④创建专门处理信号的线程

线程池：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：线程自己保存数据，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

WIN XP 实现一对一模型，但是通过 fiber 库也支持多对多。每个线程包括：ID 寄存器集 用户栈和内核栈 私有数据存储区。后面三个都是线程的上下文。主要数据结构 ETHREAD 执行线程块 KTHREAD 内核线程块 TEB 线程执行环境块 后者在用户空间 前两者内核空间。

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

调度类型：

①高级调度，从后备队列中将进程调入内存，创建进程分配资源放入就绪队列②中级调度，将不能运行进程调至外存挂起等待③低级调度，从就绪队列中选取进程进入处理机执行

CPU 调度时机：

run->wait, run->ready,wait->ready, terminate 等

分派程序(dispatcher)将 CPU 的控制交给由短程调度选择的进程，①上下文切换②切换到用户模式③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：①**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) ②响应时间，进程提出请求到首次被响应的时间③**waiting time**(在 ready queue 中等待时间) ④截止时间

面向系统：①吞吐量 throughput②处理机利用率 CPU utilization③设备均衡利用，CPU 繁忙型与 I/O 繁忙型搭配

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

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

①抢占式：如果更短的进程则抢占，这种 SJF 叫做 Shortest-Remaining-Time-First (SRTF)

②非抢占：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**。

估计CPU运行时间为

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

其中，

- t_n是第n个进程的实际CPU运行时间
- τ_{n+1}为预计的下一个进程的CPU运行时间
- $0 \le \alpha \le 1$

可能导致饥饿

优先级调度

确定进程优先级：①静态优先权：创建时确定好②动态优先权。**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**。抢占/非抢占

饥饿，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)

提高并发性和响应时间，提高资源利用率

①将就绪中的进程按照 FCFS 排队②每次调度时将 CPU 分给队首进程，让其执行一个时间片③时间片结束时发生时钟中断④调度程序暂停当前进程执行将其送至就绪队列尾，然后上下文切换至新的队首⑤进程可以未使用完完一个时间片就让出 CPU(如阻塞)

如果就绪队列中有 n 个进程时间片为 q，那么每个进程得到 1/n 的 CPU 时间，长度不超过 q。每个进程必须等待的时间不超过(n-1)q，知道下一个时间片位置。例如 5 个进程，200ms 时间片，那么每个进程每 100ms 不会得到超过 20ms 的时间。

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

时间片长度影响因素：①就绪进程越多，时间片越小②应当使用户输入能在一个时间片内完成，否则平均周转和平均带权周转都会延长

一般来说 RR 比 SJF 平均周转更长,无饥饿，响应时间更好。时间片固定时，用户越多响应时间越长，适合前台进程

多级队列调度

将**就绪队列**根据性质或类型的不同分为多个独立队列区别对待，综合调度。每个作业固定归入一个队列。不同队列可能有不同的优先级时间片 调度算法。例如系统进程、用户交互、批处理等这样的队列分法。

一般，分成前台 foreground(交互式 interactive)和后台(批处理)，后台一般 RR，前台 FCFS。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

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，设置多个就绪队列，优先级和时间皮不同，优先级越低时间片越长

I/O 型进程，进入最高优先级队列，处理完 I/O 请求后转入阻塞队列；CPU 型进程，执行完时间片后进入更低级队列

I/O 完成时提高优先级；时间片用完时降低

提高吞吐量降低平均周转，照顾短进程；提高 I/O 设备利用率和降低响应时间；**I/O bound 高优先级，CPU bound 低优先级**

Windows 2000/XP 采用严格抢占式动态优先级调度，根据优先级和分配时间配额进行调度，调度对象是线程

Linux 采用抢占式调度，分时技术，相同优先级进程采用时间片轮转法，动态优先级

Process synchronization

进程间的竞争：互斥、死锁和饥饿

The Critical-Section Problem

临界区问题的解决必须满足三个要求：①**互斥(mutual exclusion)②空闲让进(progress)③有限等待(bounded wait)。让权等待不是必须的进程最基本的特性：并发性，独立性**

Peterson 算法 – 进程互斥的软件实现方法

适用两个进程，假设 load 和 store 是原子操作

①**单标志法**：一个进程退出临界区后，标志改变，另一个进程马上进入临界区，如此交替

强制进程进入临界区，没有考虑进程的实际需要，资源利用不均匀

当一个进程让出临界区之后，另一个进程使用临界区之前，这个让出临界区的进程不可能再次使用临界区，**违反了空闲让进原则**

②**双标志法**：只有当前进程想要进入临界区且另一个进程不在临界区时才进入

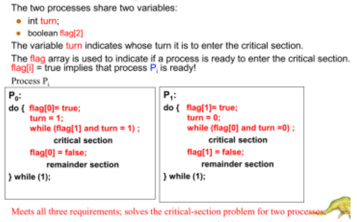
可能导致两个进程同时进入临界区，因为进程进入临界区前先检查能否进入，然后再进入，

在这个间隙可能两个进程同时做这一步，就会导致同时进入，违反了互斥原则

改进：先设置标志再进入，双标志后检查法，可能导致两个进程都进入不了临界区，**违反了有限等待原则**

③Peterson

每个进程先将各自的标志记录，表达想要进入临界区的想法，然后等待单标志轮到自己进程进入时，二者都满足才进入临界区



现代OS 中不适用(若编译器交换两条指令则寄)

Lamport 面包房算法

N 个进程先进行取号登记顺序然后按照顺序进行访问临界区，如果取号一致则按照字典序，执行完后号码归零，重新排队

每个事件对应一个 lamport 时间戳，初始值为 0

-如果事件在节点内发生，时间戳加 1

-如果事件属于发送时间，时间戳加 1，并在消息中带上该时间戳

-如果事件属于接收事件，时间戳为本地时间戳和消息中的时间戳的最大值加 1

硬件同步方法

单处理器：在临界区禁止中断。

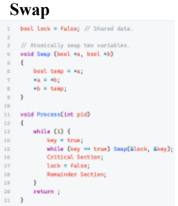
多处理器：**Memory barriers**(an instruction forcing any change in memory to be propagated (made visible) to all other processors)

Hardware instruction

抽象出两个硬件实现的原子操作：赋值和交换

硬件方法优点：进程数随意，简单，支持多个临界区；缺点：**无法让权等待，可能饥饿，可能死锁。会引起忙等**

Test-And-Set **Swap**



Test_and_set 存在 busy waiting 的情况

Bounded-Waiting Mutual Exclustion with TS

可以解决所有临界区问题

自旋锁：当一个进程想要访问已被其它进程锁定的资源时，进程循环检测锁是否释放，拥有自旋锁的线程永远不会被抢占

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），二值信号量互斥锁(mutex locks)。整型信号量可能导致忙等，违反了让权等待原则

增加 block(run->wait)和 wakeup(wait->ready)来避免了忙等。

负数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①互斥操作，在同一进程里出现②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**: temporary assign the highest priority of waiting process to the process holding the lock.

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

①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②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

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

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。

解决：①最多只允许 4 个哲学家坐在桌上②临界区内必须拿起两根筷子③使用非对称的解决方法：奇数先拿左手，偶数先拿右手

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

进程使用系统资源顺序：①申请②使用③释放

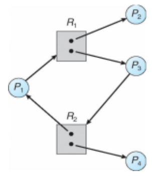
四个必要条件：

①Mutual exclusion：一次只能有一个进程使用该资源②hold 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③No preemption：被占用的资源不能被抢占，只能在进程使用完成后释放④circular wait：进程间循环等待资源

资源分配图，G(V, E)，V 分为①系统活动进程集合 P②系统所有资源类型的集合 R

请求边 Pi→Rj 分配边 Rj→Pi

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 P3，这样就打破了死锁.**死锁定理：S 为死锁状态当且仅当 S 状态的资源分配图是不可完全简化的**

死锁处理

保证系统不进入死锁：①prevention②avoidence；允许进入死锁但恢复：③detection④recovery

死锁预防 Prevention

预防死锁发生条件的发生

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**。静态预分配方式：只有所需全部资源都就绪才执行进程，否则不能占有任何资源，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，其已分配的资源马上释放，可以被抢占。被抢占资源分配到进程的资源等待列表中。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

死锁避免

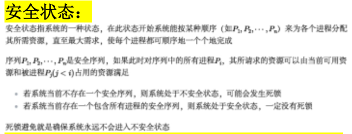
死锁预防降低了吞吐率，可以获取一些额外事先信息从而避免死锁 prior information。

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

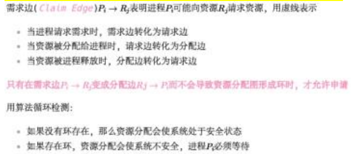
资源类型只有一种实例，使用资源分配图算法

资源类型有多个实例，使用银行家算法

安全状态：



资源分配图，single instance 死锁避免算法：



Banker 银行家算法：

每个进程事先声明最大需求量且不能超过系统最大资源量；进程请求资源时可能会等待；进程拿到资源后必须在有限时间内释放它们

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

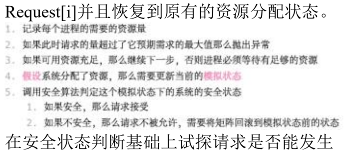
资源请求算法：

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：

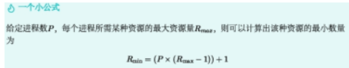
1.如果 request[i]<=need[i]跳到第二步，否则出错，因为进程 Pi 已经超过了其最大需求。

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：Available=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。



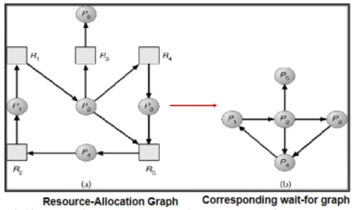
在安全状态判断基础上试探请求是否能发生



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 n*n，n 为点数。

多实体资源类型检测算法：

数据结构：Available，allocation 是一样的，request：n*m 的矩阵，表示当前各进程的资源请求状况，request[i][j]=k 表示 Pi 正在请求 k 个资源 Rj。

算法：1.设 work 和 finish 分别是长度为 m 和 n 的向量，初始化：work=available，如果 allocation[i]非 0，finish[i]=false 否则初始化为 true；

2.寻找 i 满足 finish[i]=false 且 request[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.如果某个 finish 是 false，那么系统处于死锁状态，且对应下标的进程 Pi 死锁。

算法需要 m*n*n 的操作数量级确定系统状态

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：①通知管理员②系统自己恢复

打破死锁的两种方法：①抢占资源②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。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。

基址寄存器存放了逻辑地址首地址，限长寄存器存放逻辑地址的结束地址或逻辑地址长度

地址绑定的三种情况：

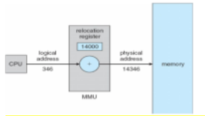
编译时间：如果编译时就知道进程在内存中的地址，那么就可以生成绝对代码 absolute code。

装载时间：编译时不知道在哪，那么编译器生成可重定位代码 relocatable code。

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)

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



Dynamic Loading（动态加载）

进程大小会受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储存 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。

Contiguous Allocation（连续分配）

Single-partition Allocation：内存通常分为两个区域：用户区+操作系统区，**由于中断向量一般位于低内存，所以 OS 也放在低内存**。重定位寄存器用于保护各个用户进程以及 OS 的代码和数据不被修改。Base 是 PA 的最小值；limit 包含了 LA 的范围，每个 LA 不能超过 Limit。MMU 地址映射是动态的。

Multiple-partition allocation：将内存划分为多个连续区域叫做分区，每个分区放一个进程。有固定分区和动态分区两种。外碎片

动态分区：

分区大小恰好符合需要。操作系统维护空闲内存和已占用内存

算法：①first-fit②best-fit③worst-fit④next-fit

First 和 best 在时间和空间利用率都比 worst 好。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

碎片 fragmentation

内碎片：进程分区内部，不能被使用

外碎片：进程内存间碎片，可运行进程数减少

first 和 best 都存在外部碎片的问题。外碎片可以通过紧凑 compaction 拼接 defragmentation 减少。重定向是动态并且在执行时间完成可以进行紧凑操作 重新排列内存来将碎片拼成一个大块，但是拼接的开销很大。

分页存储管理

将物理内存分为 frame，将逻辑内存分为 page

Linux Win(x86)是 4KB

分页式管理是用户视角内存和物理内存分离的内存分配管理方案

分页式管理会产生内部碎片

OS 需要跟踪所有空闲帧，叫帧表

2^m 字节的内存大小对应需要 m 项的页表，物理地址长度为 m

逻辑地址和页表大小及虚存空间有关：虚存大小 2^m 字节那么逻辑地址长度为 m，页表大小 2^n 字节，则页偏移位数 n，页号位数 m-n。

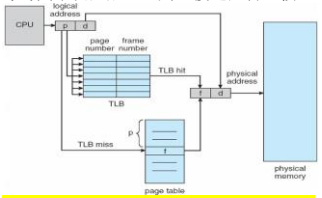
地址映射过程：逻辑页号拼上 offset 经过页表查到物理页号，然后得到物理帧号拼上 offset，然后进入到内存中找 frame。

页表的实现

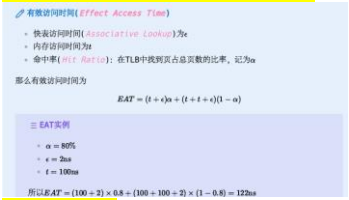
页表放在内存中。PTBR page-table base reg 指向页表，切换页表只需要改变这个寄存器

PRLR page-table length register 说明页表长度

两次访存，一次查页表一次查数据/指令。为了加速，引入了专用硬件 cache **TLB**。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



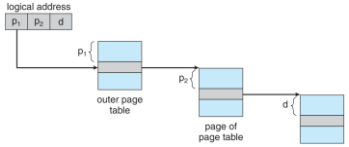
保护 protection

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

现代计算机逻辑地址空间很大导致页表会很大且连续，不现实。因此要将页表划分变小。方法：页表再分页，即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

某计算机采用二级页表的分页存储管理方式，按字节编址，页大小为2^{10}字节，页表项大小为2字节，逻辑地址结构为：

页目录号	页号	页内偏移量

逻辑地址空间大小为2^{16}页，则表示整个逻辑地址空间的页目录表中包含表项的个数至少是

A. 64　　B. 128　　C. 256　　D. 512

页大小为 2^{10}B，页表项大小为 2B，采用二级页表，一页可存放 2^9 个页表项，逻辑地址空间大小为 2^{16} 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 $2^{16}/2^9=2^7=128$ 个页面保存页表项，即页目录表中包含的个数最少为 128。

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 segment-number 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。分段有外碎片，无内碎片

Virtual Memory 虚拟内存

局部性原理：时间：指令执行和下次执行、数据访问和下次访问都集中在一个较短时期内空间：当前指令和邻近指令、当前数据和邻近数据都集中在一个小区域内。

虚拟存储器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

需要页时才调入相关页，采用 lazy swapper，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、不在内存、无效页都会缺页

缺页率等于 1 不代表 every page is a page fault

更完整的页表项 请求分页中

物理帧号|状态位 P（页是否已调入）|访问字段 A（页面访问次数）|修改位 M|外存地址

Effective memory-access time 有效访问时间 EAT=(1-p)*访存时间+p*缺页异常服务时间

Page fault time= page fault overhead+swap page out & in+restart overhead

缺页异常服务流程：

1.陷入 trap 到 OS

2.保存用户 reg 和进程状态

3.确定中断是否为 page fault

4.检查页引用是否合法并确定所在磁盘位置

5.磁盘读页到内存空闲帧(在等待队列中等待磁盘服务、磁盘寻道旋转延迟、向内存传输页)

6.在等待过程中让出 CPU 给其它进程，调度

7.磁盘 IO 完成，中断返回

8.保存让出的 CPU 的其他用户寄存器和进程状态（如果进行了 6）

9.确定中断是否来自磁盘

10.修正页表及相关表，所需页已在内存

11.等待 CPU 再次分配给本进程

12.恢复寄存器、进程状态和新页表，重新执行

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、重启时间**

如果有 m 个物理帧初始为空，引用串长度为 p，包含 n 个不同页访问，n≤缺页数≤p

写时复制 copy-on-write

父子进程开始时共享同一页面，某个进程修改共享页时才会拷贝一份该页

加快进程创建速度。当确定一个页采用 COW 时，这些空闲页在进程栈或堆必须拓展时可用于分配 或用于管理 COW 页。OS 此案用按需填 0 zero fill on demand 按需填零页需要在分配前填 0.*Windows&linux&solaris 都用了 COW*

页面置换

内存过度分配→换出内存中未使用的页

使用脏位，只有脏页写回硬盘，减少缺页开销

基本页面置换过程：

1.查找所需页的磁盘位置 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。

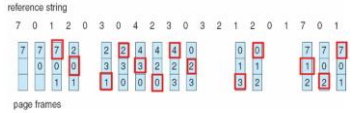
页面置换算法

目标：最小缺页率的置换算法

引用串，如 *0x432 属于页 4(100Byte/page)*

First-In-First-Out Algorithm FIFO

最简单的页面置换算法　　15 次缺页↓



Belady's Anomaly:越多帧，缺页越多

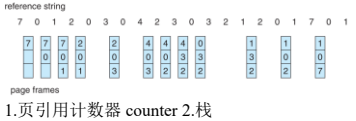
Optimal Page Replacement OPT

缺页率最低且无 Belady 异常。置换最长时间不使用的页、未来不再使用、离当前最远位置上的页。页使用时长看下一次该页出现的距离

Least Recently Used LRU 最近最久使用

局部性原理，性能最接近 OPT

需要记录页面使用时间，硬件开销大



1.页引用计数器 counter 2.栈

3.移位寄存器：引用时最高位置 1，定期右移

LRU Approximation LRU 近似 NRU

页表项中设立引用位，初值 0，被访问置 1

替换访问位为 0 的页

Additional reference bits 附加引用位算法

移位寄存器，通常 8bit，换出 reg 值最小的页

Second chance 二次机会/clock 算法 NRU

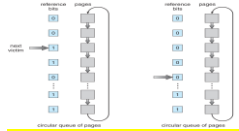
选择页时检查引用位，若 0 直接置换；若 1，给第二次机会，选择下一个 FIFO 页

一个页获得二次机会时，引用位置 0，且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

一种实现是循环队列，用一个指针表示下一次要置换的页。当需要一个帧时指针向前移动直到找到一个引用位 0 的页，移动过程中置 0 经过页的引用位

最坏情况：所有帧都二次机会，所有页引用位置 0 后选择页进行置换，此时二次机会=FIFO



Enhanced Second chance 改进 clock 增强二次机会

引用位+脏位的有序对

(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，很可能继续用

(1,1)有引用有修改，很可能继续用且需要写回

淘汰次序(0,0)-> (0,1)-> (1,0)-> (1,1)

使用时钟算法置换(0,0)页，在置换前可能要多次搜索循环队列。改进：给未引用的脏页更高优先级，降低 IO 次数 *Macintosh 系统使用*

Counting 基于计数的置换算法

使用计数器记录页被访问的次数

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

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

这两种开销很大且还很难近似 OPT

Page Buffering 页面缓冲

设置缓冲，有机会找回刚被置换的页

用 FIFO 选择置换页，把被置换页放入两个链表之一：若页面无修改，归入空闲页链表，否则归入已修改页链表

调入新页时将新页内容读入空闲页链表第一项所指的页面，然后将其（第一项）删除

空闲页和已修改页仍停留在内存中一段时间

已修改页达一定数目后一起调出到外存，然后将它们归入空闲页面链表

Windows 和 Linux 采用

帧分配 allocation of frames

每个进程都需要最小数目的页

分配策略：固定分配+可变分配

平均分配算法 Equal allocation

按比例分配 Proportional allocation

优先级分配 Priority allocation

按比例分配，但是用优先级进行比例分配

页置换策略：

全局置换 global allocation

从内存所有帧中选择一个进行替换

局部置换 local allocation

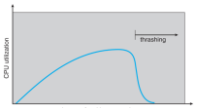
从自己的分配帧中进行置换选择

组合策略：固定分配局部置换|可变分配全局置换|可变分配局部置换

Reclaiming Pages 空闲页数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



频繁的页调度行为会导致：CPU 利用率低、OS 认为多道程序道数需要增加、其他进程进入到系统中

等价于一个进程不断换入换出页

按需调页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

L=S 准则：产生缺页平均时间=系统处理时间

工作集合模型 Working set model

WS 工作集：每一个进程在一个 delta 时间窗口内引用的页的集合

Delta=工作集窗口=固定数目的页引用，例如 10000 条指令

WSS_i 进程 Pi 的工作集大小=在最近 delta 内总页面引用次数，若 delta 太小，不能包含整个局部；delta 太大，可能包含过多局部；delta 无穷工作集为进程执行所接触到的所有页的集合

D=$\sum WSS_i$ 求和=帧总需求量，m=帧总可用量。若 D>m 就会发生颠簸，此时挂起一个进程降低多道程序的道数

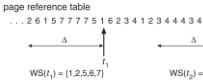
跟踪工作集合模型

OS 跟踪每个进程的 WS，并为进程分配大于其 WS 的帧数，如果还有空闲帧，那么可以启动另一个进程，如果所有 WS 的和的增加超过了可用帧的总数，那么 OS 会暂停一个进程，该进程的页面被换出且其帧可以被分配给其他进程。挂起的进程可以稍后重启。这样的策略防止了颠簸，提高了多道程序的程度，优化了 CPU 使用率。

WS 窗口是一栋窗口，每次引用时会增加新引用并丢失最老引用。如果一个页在 WS 窗口内被引用过，那么他就处于 WS 中

通过固定定时中断和引用可以模拟 WS 模型

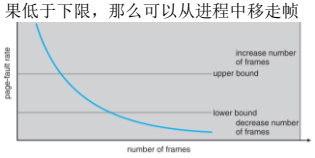
假设 delta=10000 个引用，且每 5000 个会出现中断。当中断出现，先复制再清除所有页的引用位。出现 page fault 后，可以检查当前引用位和位于内存内的两个位，确定在过去的 10000 到 15000 个引用之间该页是否被引用过。如果使用过，至少有一个位为 1.如果没有使用过，3 个位全是 0。只要有一个 1，那么可以认为处于 WS 中。这种安排并不完全准确，因为并不知道在 5000 个引用的何处出现了引用。通过增加历史位的位数和终端频率可以降低不确定性，但是开销也会变大。



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，如果缺页率超过上限，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

虚存技术将文件 IO 作为普通文件访问的技术

初始文件访问按照普通按需请求调度，会出现缺页。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。多个进程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

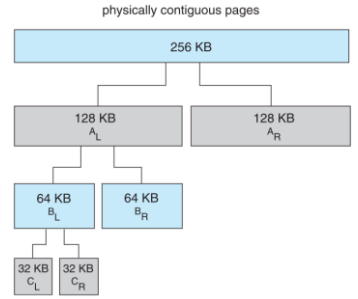
与对待用户内存不同：内核内存从空闲内存池中获取，两个原因：1.内核需要为不同大小的数据结构分配内存。2.一些内核内存需要连续

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。不断二分寻找合适的内存

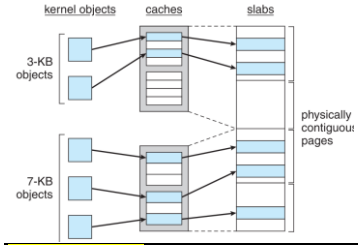


Slab 分配

预先了解内核常见数据结构（称为各种 object）的大小并预先准备好对应粒度的小内存块，注册到每类 object 的 cache 里。当一个 object 需要使用内存时，就查询对应的 cache 里是否有空闲的内存块，如果有就分配给它，如果没有就向 Buddy system 申请

相当于预先把内存分成不同尺寸然后根据所需内存尺寸选择合适的空间来放

Slab 优点：没有碎片引起的内存浪费；内存请求可以快速满足



预调页 prepaging

为了减少冷启动时大量的缺页

同时将所有需要的页一起调入内存，但是如果预调页没有被用到，那么 IO 就被浪费了。

假设 s 页被预调到内存，其中 a 部分被用到了。问题在于节省的 s*a 个页错误的成本是大于还是小于其他 s*(1-a)不必要的预调页开销。如果

a 接近于 0，调页失败，a 接近 1，调页成功。

页大小

页大小的考虑因素：碎片↓、页表大小↑、IO 开销↑、局部性↓、page fault 数量↑、TLB 大小及效率↑

TLB 范围 TLB reach

TLB 范围指通过 TLB 可以访问到的内存量
TLB Reach=TLB size * Page Size
理想情况下每个进程的 WS 应位于 TLB 中，否则会有不通过 TLB 调页导致大量 IO 导致缺页率高
增大页大小来缓解 TLB 压力；但可能会导致不需要大页表的进程带来的内碎片
提供多种页大小的支持：那么 TLB 无法硬件化，性能降低。

IO 互锁

IO 互锁指页面有时必须被锁在内存中
必须锁住用于从设备复制文件的页，以便通过页面置换驱逐。

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

名称&标识符(唯一标识该文件的数字)&类型&位置&大小&保护&时间日期&用户标识
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

文件是抽象数据类型：创建&写&读&文件内重定位&删除&截断
Open(Fi)①在硬盘上寻找目录结构②移动到内存中；Close(Fi)将内存中目录结构移到磁盘中

打开文件

每个打开文件都有以下信息：
文件指针：上次读写位置作为当前文件指针
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

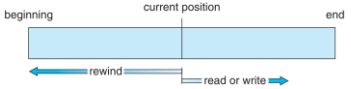
文件内部结构 File Structure

无结构 None：字/字节序列，流文件结构
简单记录结构：lines, fixed length, variable length
复杂：formatted-document, relocatable-load-file
可以通过插入适当的控制字符，用第一种方法模拟最后两个
这些模式由 OS 和程序所决定

访问方法

Sequential access 顺序访问

最常用，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/Relative/Random access 直接访问

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。

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

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
在 Windows 中，目录和文件由不同的系统调用来控制，而在 Linux 中目录是一个特殊文件
磁盘结构
磁盘可装多种文件系统、分区或片 minidisk slice。

目录操作

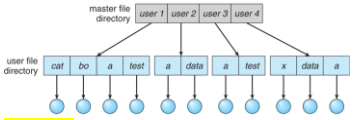
搜索文件&创建文件&删除文件&遍历 list&重命名文件&遍历 traverse 文件系统

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

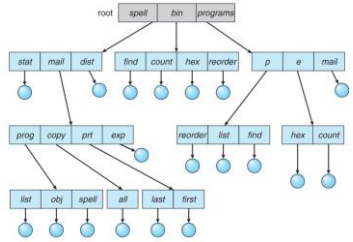
二级目录

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



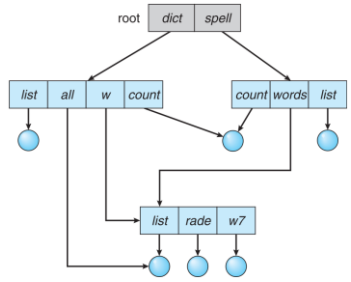
树形目录

拓展二级目录，搜索高效，有分组能力



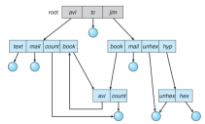
无环图目录 Acyclic graph

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。
实现文件和目录共享，***UNIX 采用创建链接***
链接实际上是另一个文件的指针。链接通过使用路径名定位真正文件
无环图目录导致一个文件可以有多个绝对路径名，不同文件名可能表示同一文件，出现了**别名问题**，存在**悬挂指针问题**：删除一个文件后指向该文件的其他链接成为 dangling
软链接（符号链接）：类似于 Windows 快捷方式，删除软链接所指文件不会删除软链接
软链接本质上是特殊的文件
硬链接：复制链接目文件目录项所有元信息，文件平等属于两个目录，文件元信息被更新时需保证在若干目录下该文件的信息一致
删除被硬链接的文件并不会直接导致文件被删除，只有被引用数为 0 才删除文件，否则只更新元信息并删除对应目录表项
硬链接本质上是目录表项，与文件系统相关，只能在同一个文件系统下创建而不能跨越
UNIX/Linux 采用硬链接，在 inode 中保留一个引用计数。通过禁止对目录的多重引用，可以维护无环图结构。
任意路径都有. 和 .. 这两个特殊目录，它们通过硬链接分别指向当前目录和父目录（根目录的父目录也是根目录）



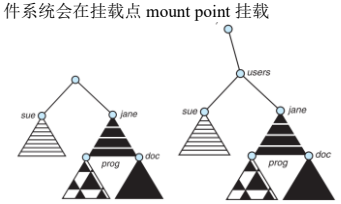
普通图目录 General graph

仅允许向文件链接，允许目录存在环
垃圾回收机制
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。一个未挂载的文件系统会在挂载点 mount point 挂载



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。

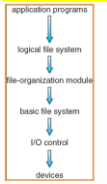
保护 Protection

访问类型：读&写&执行&追加&删除& 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

文件系统是 OS 中以文件方式管理计算机软件资源的软件以及被管理的文件、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（磁盘）中
提供对磁盘的高效且便捷的访问
磁盘：1.可以原地重写 2.可以随机或顺序访问
write() & fsync(): write()为在未来某个时间将数据写回到二级存储中，具体时间由文件系统根据性能决定；fsync()强制将 dirty data 写回 disk。
文件控制块 file control block：包含文件的树形，如拥有者、权限、文件内容的位置
设备驱动控制物理设备

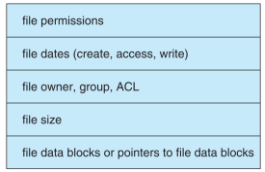
分层设计的文件结构



文件系统实现

On-Disk FS structure

磁盘上，文件系统可能包括：如何启动硬盘中的 OS、硬盘总块数、空闲块数目和位置、目录结构及各个具体文件等
磁盘结构：①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②每个卷的卷控制块 volume control block 包括卷的详细信息③目录结构，用于组织文件和维护元信息④每个文件的 FCB
FCB：维护了被打开的文件的具体信息



包括文件时间戳、文件大小、访问权限，不包括文件名
FCB 在 open 系统调用时在磁盘上创建

In-Memory FS structure

①In-memory patition table 分区表②in-memory directory structure 目录结构③system-wide open-file table 系统打开文件表④per-process open-file table 进程打开文件表

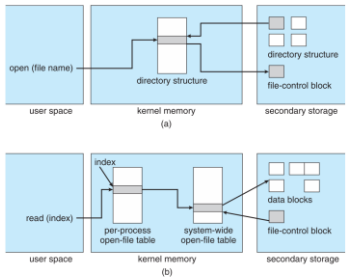


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

System-Wide Open-File Table：记录所有被 load 到内存中的 FCB inode；Per-Process Open-File Table：其表项指向 System-Wide Open-File Table 中的项（包含当前在文件中位置、文件访问模式等信息）

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

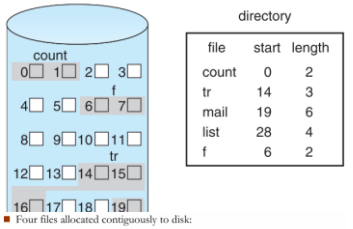
线性列表：使用文件名和数据块指针的线性表
哈希表：根据文件名得到一个值返回一个指向线性表中元素的指针
许多操作系统采用软件缓存来存储最近访问过的目录信息，如：Linux 的目录项对象

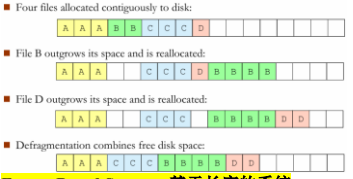
文件物理结构 Allocation Method

主要磁盘空间分配方法：连续、链接和索引

连续分配 Contiguous Allocation

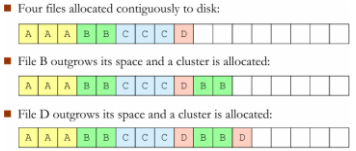
每个文件在磁盘上占有一组连续的块。优点：访问很容易，只需记录起始块位置和块长度，寻道时间最短，支持随机访问，但浪费空间，存在动态存储分配问题。First 和 best 表现差不多，first 时间快很多。存在外碎片问题，此外文件大小不可增长
逻辑到物理的映射：LA/512 得到块号和偏移量
LA 是存取文件逻辑地址，512 是块大小





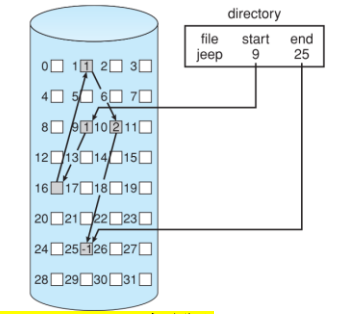
Extent-Based Systems 基于长度的系统

利于 Veritas FS 采用，解决了文件大小无法增长的问题
开始分配一块连续空间，空间不够时链式分配连续空间，称为扩展。
文件块的位置(文件属性之一)为开始地址、块数、加上一个指向下一扩展的指针



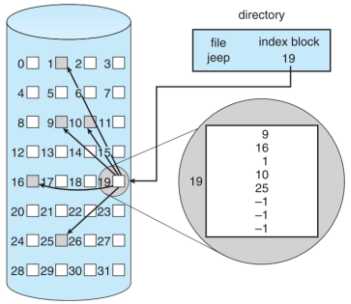
Linked Allocation 链接分配

解决了连续分配的所有问题。每个文件都是磁盘块的链表。只需起始地址即可访问，无空间管理问题、浪费空间，但是不支持随机访问，块与块之间的链接指针需要占用空间
簇：将多个连续块组成簇，磁盘以簇为单位进行分配，节省指针占用空间，***Windows 采用***
地址映射：LA/(512-1)得到块号和偏移量
每个索引块的尾节点用来链接下一个索引块，不链数据块，所以要 512-1

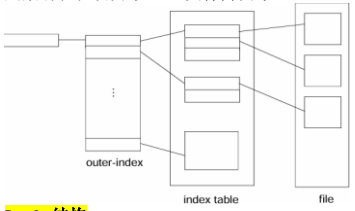


Indexed Allocation 索引分配

把所有指针放在一起通过索引块访问
每个文件都有索引块，一个磁盘块地址数组
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
小文件的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，支持随机访问，没有外碎片，但有索引开销。LA/512 得索引表块号和表项号
Unix inode 采用组合索引：直接寻址+一级索引+二级索引+三级索引

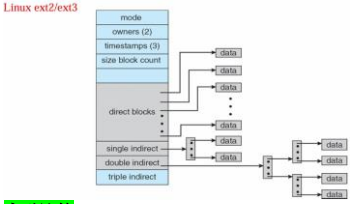


链接索引 Linked scheme，把索引块链接起来，LA/(512*511)得到商 Q1 和余数 R1，商 Q1 为索引表的块号，R1/512 得到商 Q2 和余数 R2，Q2 是索引表表项块号，R2 是文件内块号
如果是二级索引，那么 LA/(512*512)得到 Q1 和 R1，Q1 外索引，R1/512 得到 Q2 和 R2，Q2 是索引表表项块号，R2 文件内块号



Inode 结构

Linux ext2/ext3



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，<u>Linux 中一个索引节点共有 15 个指针在 inode 中，前面 12 个直接指向文件块，其他三个指针分别指向一级索引、二级索引、三级索引间接块</u>，因此有 48KB 可以直接访问，最大文件的大小为：

12*4KB+1K*4KB+1K*1K*4KB+1K*1K*1K*4KB=48KB+4MB+4GB+4TB

文件 F 由 200 条记录组成，记录从 1 开始编号。用户打开文件后，欲将内存中的一条记录插入到文件 F 中，作为其第 30 条记录。请回答下列问题，并说明理由。

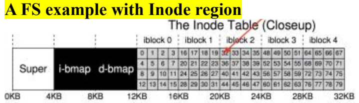
1. 若文件系统采用**连续分配**方式，每个磁盘块存放一条记录，文件 F 的存储区域**前后**均有足够空闲的磁盘空间，则完成上述插入操作最少需要访问几次存储块？F 的文件控制块内容会发生哪些改变？
2. 若文件系统采用**链接分配**方式，每个磁盘块存放一条记录和一个链接指针，则完成上述插入操作需要访问几次磁盘块？若每个磁盘块大小为 1 KiB，其中 4 B 存放链接指针，则该文件系统支持的文件最大长度是多少 GiB？

1.前移 1-29 块，29*2+1=59(读取后写，2 次 IO)，关于文件控制块的变化，与文件存储直接相关的是：文件起始地址改变（向前移动一块）以及文件长度改变（增加一条记录）

2.31

- 这种问题乍一眼看可能会觉得：「链表不是不限长度吗，为什么会有最大大小」，但是需要注意，**指针是定长的**，地址是有限的，因此决定了链节也是有限的。
- 所以，这里的「其中 4 B 存放链接指针」包含了两个信息：① 每个磁盘块有效存储空间为 1 KiB - 4 B，② 最多只有 2^{32} 个链节。
- 因此，该文件系统支持的文件最大长度为 2^{32} * (1 KiB - 4 B) = 4080 GiB。

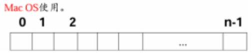
A FS example with Inode region



空闲空间管理

位图（n块）（位向量）

- bit[i] = 0 → block[i]空闲
- bit[i] = 1 → block[i]被占用
- 第一个空闲块的计算：
  - 一个字的位数 × 值为1的字数 + 第一个值为0的位的偏移
- 位图需要额外的空间
  - 设块大小为2^{12}字节
  - 磁盘大小为2^{30}字节 (1GB)
  - $N = 2^{30} / 2^{12} = 2^{18}$ (即32K bytes)
- 容易得到连续的文件
- Windows、Liunx、Mac OS使用。



空闲链表：将所有空闲块链接起来，将指向第一个空闲块的指针保存在磁盘的特殊位置且还存在内存中，但 IO 效率低，因为需要遍历

分组：将 n 个空闲块的地址存到第一个空闲块中，可以快速找到大量空闲块的地址

计数：通常有多个连续块需要同时分配或释放因此记录第一块的地址和紧跟第一块的连续空闲块的数量 n

页面缓冲 page buffer

统一缓冲区将文件数据页到虚存而不是磁盘块，因为虚拟内存相关接口比文件系统相关接口更快

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，否则文件系统就还没更新。

文件系统实例

- FAT (MS-DOS文件系统)
- FAT32 (VFAT), exFAT (64位)
- NTFS (NT文件系统)
- ReFS(Resilient File System)复原文件系统 (Windows 8后、server)
- S51K/S52K (AT&T UNIX sysv)
- ext (minix文件系统)
- ext2、ext3、ext4 (linux文件系统、Android)
- proc、sysfs (linux虚拟文件系统)
- yaffs (Yet Another Flash File System)
- ReiserFS (Linux一种日志文件系统)
- HPFS (OS/2高性能文件系统)

- UFS(BSD UNIX的一种文件系统)
- HFS+ (Mac OS , iOS文件系统)
- iso9660 (通用的光盘文件系统)
- NFS (网络文件系统)
- VFS (Linux虚拟文件系统)
- VFS是物理文件系统与服务之间的一个接口，它屏蔽各类文件系统的差异，给用户和程序提供一个统一的接口
- ZFS (Open Solaris文件系统)
- LTFS(Liner Tape File System, 线性磁带文件系统) 为磁带提供了一种通用、开放的文件系统。
- APFS (Apple File System) 苹果新一代的文件系统，MAC OS 10.13以后、iOS 10.3以后

FAT File allocation table 文件系统

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。

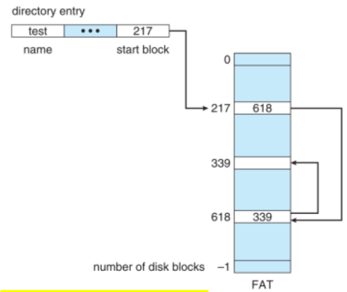
计算机系统启动时，首先执行的是 BIOS 引导程序，完成自检，并加载主引导记录和分区表，然后执行主引导记录，由它引导 激活分区引导记录，再执行分区引导记录，加载操作系统，最 后执行操作系统，配置系统。

FAT 的每个目录项为 32 个字节

FAT32 长文件名目录项由几个 32B 表项组成。用一个表项存放短文件名和其他属性（包括簇号、文件大小，最后修改时间和最后修改日期、创建时间、创建日期和最后存取日期），短文件名的属性是 0x20。

用连续若干个表项存放长文件名，每个表项存放 13 个字符（使用 Unicode 编码，每个字符占用 2 个字节。）

长文件名的表项首字节的二进制数低 5 位值，分别为 00001 、00010 、 00011 、……，表示它们的次序，左起第 2 位为 1（也就是在低 5 位基础上加 40H）表示该表项是最后一项。最后项存放 13 个字符位置多余时，先用 2 个字节 0 表示结束 ，再用 FFH 填充。长文件名的属性是 0x0F。长文件名项的第 13、27、28 字节为 0x00，第 14 字节为短文件名校验和。

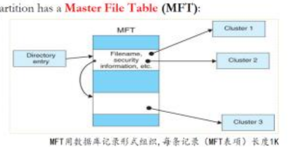


NTFS File System 文件系统

NTFS卷布局：

引导扇区	主文件表(MFT)（前16个为系统文件）	文件区	MFT前16个元数据文件备份	文件区

Each partition has a Master File Table (MFT):



MFT用数据库记录形式组织，每条记录（MFT表项）长度1K

MFT由一个个MFT项（也称为文件记录）组成，每个MFT项占用1024字节的空间。

MFT前16个记录用来存放元数据文件的信息，它们占有固定的位置。

每个MFT项的前部几十个字节有着固定的头结构，用来描述本MFT项的相关信息。后面的字节存放着文件属性等。

每个文件或目录的信息都包含在MFT中，每个文件或目录至少有一个MFT项。

MFT表项信息

ReFS 复原文件系统 ， Windows server 版

Microsoft 的最新文件系统，旨在最大限度地提高数据可用性、跨不同的工作负荷高效地扩展到大型数据集，并提供数据完整性，使其能够恢复损坏。它旨在解决存储方案的扩展集问题以及为将来的革新打造基础。

功能	ReFS	NTFS
最大文件名长度	255 个 Unicode 字符	255 个 Unicode 字符
最大路径名长度	32K Unicode 字符	32K Unicode 字符
文件大小上限	35 PB (pb)	256 TB
最大卷大小	35 PB	256 TB

Mass storage system 大容量存储

磁盘的 0 扇区是最外面的第一个磁道的第一个扇区。逻辑块时最小传输单位 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从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，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响应。到达另一端时反向移动继续处理，也叫做电梯算法。

C-SCAN：磁头从一端到另一端，到达另一端立即返回开始处，不响应沿途（非闪现返回）

LOOK：SCAN 优化版，不移动到最远端而是移到最远的请求位置

C-LOOK：C-SCAN 优化版

调度算法选择：

SSTF 一般来说比较好，SCAN 和 C-SCAN 对于高负荷 IO 磁盘表现更好

表现依赖于请求类型和数量；磁盘请求依赖于文件分配策略；磁盘调度算法应该模块化，可自由选择。SSTF/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。

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

交换分区

虚拟内存用磁盘作为交换分区，主存的扩展

Swap-space can be carried out in two forms:

- in the normal file system
  - e.g. Windows family

pagefile.sys文件

- in a separate disk partition
  - e.g. Linux、Unix、solaris

SWAP分区

最快的方法：创建在 raw partition 上

RAID

一种把多块独立的物理硬盘按不同方式组合形成一个逻辑硬盘组从而提供比单个硬盘更高的存储性能和提供数据备份技术

0：无冗余 1：镜像 2：纠错码 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

RAID 5 provides high reliability inexpensively.

三级存储 Tertiary storage device

floppy disks&CD-ROMs&tapes&DVD

性能：

Sustained bandwidth 大传输的平均速率 字节/时间。Effective bandwidth IO 时间下的平均速率。前者是数据真正流动时的速率，后者是驱动器能够提供的能力，一般驱动器带宽指前者。

I/O system

计算机两大工作：I/O+processing

The control of devices connneted to the computer is a major concern of operating-system designers

设备驱动程序为 I/O 子系统提供统一接口

I/O 硬件

I/O 系统组成：PC BUS I/O + 主机 I/O 系统

设备寻址方式：direct I/O instructions+memory-mapped I/O

I/O 接口：串行、并行、USB、键盘、硬盘

I/O 端口：设备与计算机通信连接点端口地址

I/O 端口通常有四种寄存器，即状态、控制、数据输入与数据输出寄存器

I/O 方式

轮询 polling：OS 与 I/O 设备交互最简单方式需要进行 busy wait，降低 CPU 利用率

中断：OS 与 I/O 设备交互的主要方式

CPU 硬件有一条中断请求线 IRL，由 I/O 设备触发，需要 IO 时就申请中断

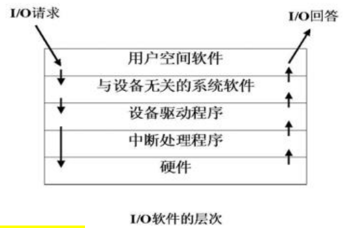
DMA direct memory access：允许内存直接与 I/O 设备交互不经过 CPU，提高 I/O 性能对于需要进行大量 I/O 的设备，为了避免程序控制 IO 即 PIO，将一部分任务交给 DMA 控制器，在 DMA 开始传输时，主机向内存中写入 DMA 命令块。然后 CPU 在写入后继续干别的，DMA 操作内存总线，然后就可以向内存进行传输

磁盘通常使用 DMA 实现 I/O 设备

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**)，为内核 I/O 子系统隐藏设备控制器之间的差异，将来自上层软件的与设备无关的抽象请求转为具体请求



I/O软件的层次

设备多维度：①Sequential or random-access 顺序或随机访问设备②Sharable or dedicated 共享或独占设备③操作速度（快速、中速、慢速）④read-write, read only, or write only 读写、只读、只写设备⑤同步异步

块设备和字符设备

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：包括阻塞 I/O 和非阻塞 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

缓冲区：为了解决 CPU 与 I/O 之间速度不匹配

假脱机技术 SPOOLing：保存设备输出的缓冲

打印机可通过 SPOOLing 改造为多用户共享

习题

进程间通信不能通过(共享文件/数据库/共享内存/全局变量) D

12. Which of the following components of program state are shared across threads in a multithreaded process?

多选题 (6 分) 6分

- A. Global variables
- B. Stack memory
- C. Register values
- D. Heap memory

一个由于等待键盘输入而不能运行的进程处于等待状态

12. 下列进程调度算法中，综合考虑进程等待时间和执行时间的是？ 回答错误

单选题 (5 分) 0分

- A. 时间片轮转调度算法
- B. 短进程优先调度算法
- C. 先来先服务调度算法
- D. 高响应比优先调度算法

正确答案: D

14. 下列选项中，满足短任务优先且不会发生饥饿现象的调度算法是？ 回答错误

单选题 (5 分) 0分

- A. 先来先服务
- B. 高响应比优先
- C. 时间片轮转
- D. 非抢占式短任务优先

正确答案: B

5. Which of the following Critical Section problem solutions results in busy-waiting? 回答错误

单选题 (5 分) 0分

- A. Semaphore
- B. Special machine instruction
- C. critical region
- D. Monitor

正确答案: B

7. The mutual exclusion semaphore of two concurrent processes has the value 0 (zero) at this moment. It indicates that__________. 回答错误

- A. a process has entered the critical-section,another process is waiting to enter the critical-section
- B. two processes have entered the critical-section
- C. a process has entered the critical-section, and no process is being blocked
- D. no process has entered the critical-section

正确答案: C

8. Critical section can be enforced with ageneral semaphore whose initial value is greater than 1. 回答错误

判断题 (4 分) 0分

- A. TURE
- B. FALSE

正确答案: B

假设一个正在运行的进程对信号量 S 进行了 P（WAIT）操作后，信号量 S 的值变为-1，此时该进程将<u>转为等待状态</u>

1. There are N processes which share M mutualexclusive resources, each process can hold W resources at most. Which of thefollowing condition may cause a deadlock?

2. 总体上说，请求分页(demand-paging)是个很好的虚拟内存管理策略。但是，有些程序设计技术并不适合于这种环境。例如，________。

- C. 二分法搜索

正确答案: C

4. Considering a system, which uses virtual memory. At what point can address binding be done? 回答错误

单选题 (3 分) 0分

- A. canbe any of the above
- B. compile time
- C. execution time
- D. loadtime

正确答案: C

3. In order to solve name collision, the file system normally adopts _________. 回答错误

单选题 (4 分) 0分

- A. indexing
- B. pathnames
- C. tree-like directory structures
- D. conventional naming methods

正确答案: C

17. In a file system, the basic operation in a "Open" system call is _________.

单选题 (4 分) 4分

- A. reading file contents from outer storageinto memory
- B. reading file control information from outerstorage into memory
- C. reading disk block from disk into memory
- D. reading file's FAT table from outer storageinto memory

10. The I/O control of disk devices mainlyadopt following method.

单选题 (9 分) 9分

- A. DMA
- B. polled
- C. interrupt
- D. channel

Suppose a system had 12 resources, 3 processes P0, P1 and P2.

	max	current	need
P0	10	5	5
P1	4	2	2
P2	9	3	6

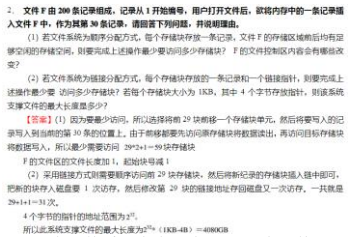
Currently there are two resources available. This system is in an unsafe state as process P1 could complete, thereby freeing a total of four resources. But we cannot guarantee that processes P0 and P2 can complete. However, it is possible that a process may release resources before requesting any further. For example, process P2 could release a resource, thereby increasing the total number of resources to five. This allows process P0 to complete,which would free a total of nine resources, thereby allowing process P2 to complete as well.

A system has 3 concurrent processes, each of which requires 4 items of resource R. What is the minimum number of resource R in order to avoid the deadlock. Answer: 10

The system design the structure File Control Block (FCB) to manage the files. Commonly, File control block is created on disk when the open system call is invoked.

Which kind of swap space is fastest?

A raw partition



2、 **文件 F 由 200 条记录组成，记录从 1 开始编号，用户打开文件后，欲将内存中的一条记录插入文件 F 中，作为其第 30 条记录，请回答下列问题，并说明理由。**

（1）若文件系统为顺序分配方式，每个存储块存放一条记录，文件 F 的存储区域前后均有足够空闲的存储空间，则要完成上述操作最少要访问多少存储块？ F 的文件控制区内容会有哪些改变？

（2）若文件系统为链接分配方式，每个存储块存放的一条记录和一个链接指针，则要完成上述操作最少要 访问多少存储块？若每个存储块大小为 1KB，其中 4 个字节存放指针，则该系统支撑文件的最大长度是多少？

【答案】(1) 因为要最少访问，所以选择将前 29 块前移一个存储块单元，然后将要写入的记录写入到当前的第 30 条的位置上。由于前移都要先访问原存储块将数据读出，再访问目标存储块将数据写入，所以最少需要访问 29*2+1＝59 块存储块

F 的文件区的文件长度加 1，起始块号减 1

(2) 采用链接方式则需要顺序访问前 29 块存储块，然后将新纪录的存储块插入链中即可，把新的块存入磁盘要 1 次访存，然后修改第 29 块的链接地址存回磁盘又一次访存，一共就是 29+1+1=31 次。

4 个字节的指针的地址范围为 2^{32}。

所以此系统支撑文件的最大长度为 2^{32}*（1KB-4B）=4080GB

Q：一个文件系统中有一个 20MB 大文件和一个 20KB 小文件，当分别采用连续、链接、链接索引、二级索引和 LINUX 分配方案时，每块大小为 4096B，每块地址用 4B 表示。问：

（1）各文件系统管理的最大文件是多少？

（2）每种方案对大、小两文件各需要多少专用块来记录文件的物理地址(说明各块的用途)?

（3）如需要读大文件前面第 5.5KB 的信息和后面第（16M＋5.5KB）的信息，则每个方案各需要多少次盘 I/O 操作?

A:（1）

连续分配：理论上是不受限制，可大到整个磁盘文件区。

隐式链接：由于块的地址为 4 字节，所以能表示的最多块数为 232=4G，而每个盘块中存放文件大小为 4092 字节。链接分配可管理的最大文件为：4G×4092B=16368GB

链接索引：由于块的地址为 4 字节，所以最多的链接索引块数为 232=4G，而每个索引块有 1023 个文件块地址的指针，盘块大小为 4KB。假设最多有 n 个索引块，则 1023×n+n=232，算出 n=222，链接索引分配可管理的最大文件为：4M10234KB=16368GB

二级索引：由于盘块大小为 4KB，每个地址用 4B 表示，一个盘块可存 1K 个索引表目。

二级索引可管理的最大文件容量为 4KB×1K×1K＝4GB。

LINUX 混合分配：LINUX 的直接地址指针有 12 个，还有一个一级索引，一个二级索引，一个三级索引。因此可管理的最大文件为 48KB＋4MB+4GB＋4TB。

（2）

连续分配：对大小两个文件都只需在文件控制块 FCB 中设二项，一是首块物理块块号，另一是文件总块数，不需专用块来记录文件的物理地址。

隐式链接：对大小两个文件都只需在文件控制块 FCB 中设二项，一是首块物理块块号，另一是末块物理块块号；同时在文件的每个物理块中设置存放下一个块号的指针。

一级索引：对 20KB 小文件只有 5 个物理块大小，所以只需一块专用物理块来作索引块，用来保存文件的各个物理块地址。对于 20MB 大文件共有 5K 个物理块，由于链接索引的每个索引块只能保存（1K－1）个文件物理块地址（另有一个表目存放下一个索引块指针），所以它需要 6 块专用物理块来作链接索引块，用于保存文件各个的物理地址。

二级索引：对大小文件都固定要用二级索引，对 20KB 小文件，用一个物理块作第一级索引，用另一块作二级索引，共用二块专用物理块作索引块，对于 20MB 大文件，用一块作第一级索引，用 5 块作第二级索引，共用六块专用物理块作索引块。

LINUX 的混合分配：对 20KB 小文件只需在文件控制块 FCB 的 i_addr[15]中使用前 5 个表目存放文件的物理块号，不需专用物理块。对 20MB 大文件，FCB 的 i_addr[15]中使用前 12 个表目存放大文件前 12 块物理块块号（48K），用一级索引块一块保存大文件接着的 1K 块块号（4M），剩下还有不到 16M，还要用二级索引存大文件以后的块号，二级索引使用第一级索引 1 块，第二级索引 4 块（因为 4KB×1K×4=16 M）。总共也需要 6 块专用物理块来存放文件物理地址。

(3)

连续分配：为读大文件前面和后面信息都需先计算信息在文件中相对块数，前面信息相对逻辑块号为 5.5K / 4K=1（从 0 开始编号），后面信息相对逻辑块号为（16M＋5.5K）/4K=4097。再计算物理块号＝文件首块号＋相对逻辑块号，最后化一次盘 I/O 操作读出该块信息。

链接分配：为读大文件前面 5.5KB 的信息，只需先读一次文件头块得到信息所在块的块号，再读一次第 1 号逻辑块得到所需信息，共 2 次。而读大文件 16MB＋5.5KB 处的信息，逻辑块号为（16M＋5.5K）/4092=4107，要先把该信息所在块前面块顺序读出，共化费 4107 次盘 I / O 操作，才能得到信息所在块的块号，最后化一次 I/O 操作读出该块信息。所以总共需要 4108 次盘 I / O 才能读取（16MB+5.5KB）处信息。

链接索引：为读大文件前面 5.5KB 处的信息，只需先读一次第一个索引块得到信息所在块的块号，再读一次第 1 号逻辑块得到所需信息，共化费 2 次盘 I / O 操作。为读大文件后面 16MB+5.5KB 处的信息，(16MB+5.5KB)/(4KB×1023)=4,需要先化 5 次盘 I / O 操作依次读出各索引块，才能得到信息所在块的块号，再化一次盘 I/O 操作读出该块信息。共化费 6 次盘 I / O 操作。

二级索引：为读大文件前面和后面信息的操作相同，首先进行一次盘 I / O 读第一级索引块，然后根据它的相对逻辑块号计算应该读第二级索引的那块，第一级索引块表目号=相对逻辑块号 / 1K，对文件前面信息 1 / 1K＝0，对文件后面信息 4097 / 1K＝4，第二次根据第一级索引块的相应表目内容又化一次盘 I / O 读第二级索引块，得到信息所在块块号，再化一次盘 I / O 读出信息所在盘块，这样读取大文件前面或后面处信息都只需要 3 次盘 I / O 操作。

LINUX 混合分配：为读大文件前面 5.5KB 处信息，先根据它的相对逻辑块号，在内存文件控制块 FCB 的 i_addr 第二个表目中读取信息所在块块号，而只化费一次盘 I / O 操作即可读出该块信息。为读大文件后在（16MB＋5.5KB）信息，先根据它的相对逻辑块号判断要读的信息是在二级索引管理范围内，先根据 i_addr 内容化一次盘 I / O 操作读出第一级索引块，再计算信息所在块的索引块号在第一级索引块的表目号为（4097-12-1024）/ 1024＝2，根据第一级索引块第 3 个表目内容再化费一次盘 I / O 操作，读出第二级索引块，就可以得到信息所在块块号，最后化一次盘 I / O 读出信息所在盘块，这样总共需要 3 次盘 I / O 操作才能读出文件后面的信息。