概述

计算机系统分为四大部分：硬件、OS、应用程序、用户。操作系统是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。

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

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-taskiing。区别?

Time Sharing system 分时系统

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.任务要在指定时间内完成.

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(Dual)：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.Fault:如除 0 缺页;Trap:事先安排的如 syscall;abort 类似硬件中断,严重

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。

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： ELF 文件的 .interp 段会指向动态链接器（loader）

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。

静态链接：在 execve 调用后立刻执行 _start

动态链接：先调用动态链接器 ld.so，完成符号解析和链接，再执行_start

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。

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，使得每个层只使用上一层的功能和服务。

微内核结构 microkernel system

只有最基本的功能（通讯、内存管理、进程管理）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。相比宏内核效率低 Windows NT windows 8 10 mac OS L4

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

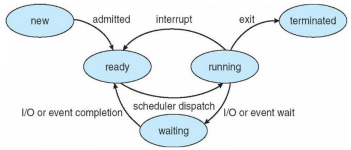
大多数现代操作系统都实现了内核模块。面向对象，内核部件分离，通过已知接口进行沟通，都是可以载入到内核中的。总而言之很像层次结构但是更加灵活。Linux solaris。

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:资源分配的基本单位

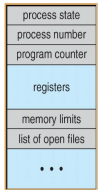
进程包括：PC,寄存器,数据段(全局 data),栈(临时 data),堆(alloc)。



进程状态：new, running, ready, waiting, terminated.

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。

有几个核就**最多**有几个进程处于 running 状态；ready 进程构成就绪队列；wait 进程构成多种等待队列。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：

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: 父的 reg 栈堆复制一份，之后互不相关

fork()：父进程返回子进程 pid，子返回 0

wait()：父等待子结束再运行,返回子 pid

waitpid()：等待特定子进程结束再运行

execXX()：覆盖进程地址空间，执行其他程序

Zombies：child 逻辑上已经终止，但是在等待 parent 调用 wait()，此时 child 为僵尸进程

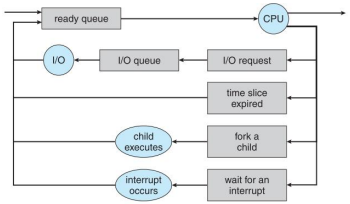
Orphans: parent 终止了都没调用 wait()，此时 child 成为孤儿进程，父进程被设置为 init(pid 1)

Daemon:连续调用两次 fork()，再终止第一个进程，那么第二个进程就会被 init 接管，成为守护进程

进程调度

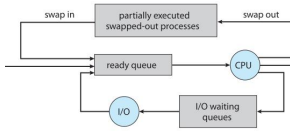
调度队列有：job 队列 ready 队列 设备队列 进程在这些队列内移动。下图是队列图，新进程开始就处于就绪队列

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。



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。

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),也即 CPU 密集

上下文切换：CPU 切换进程时，必须保存当前状态再载入新状态的过程。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。

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

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().

Message passing: basic: send(Q, msg), recv(Q, msg). 如何实现通信 link：通过 mailbox 接受信息（新的操作：create mailbox, destroy mailbox）Message synchronization: blocking – synchronous, non-blocking: asynchronous

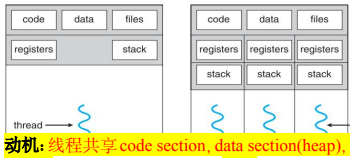
Signals, Pipes: Ordinary pipes – unidirectional 单向, parent-child relationship (fd[0] is the read end; fd[1] is the write end); Named pipes: bidirectional

RPC remote procedure calls.客户端把参数发给服务端.服务端执行返回结果

Thread 线程:CPU 调度的基本单位

进程概念体现了两个特征：资源拥有单位/调度单位 unit of resource 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 LWP lightweight process.资源拥有单元被称为进程任务。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)



动机：线程共享 code section, data section(heap), signals。(reg 和 stack 不共享)**优点**：创建新线程耗时少,context switch 开销小(no cache flush), 线程间通讯可以通过 shared memory; 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。

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，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）

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：用现有县城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 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

CPU 调度时机：run->wait, run->ready, wait->ready, terminate 等.

主动让出 CPU(1,4)叫非抢占

(Non-Preemptive),2,3 情况叫抢占

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。

dispatcher 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：**周转时间 turnaround time**(进程从**开始建立**到**完成**所用的所有时间)；

带权周转时间=周转时间/CPU 执行时间，是一个相对指标；

响应时间 response time(进程从发出请求到**第一次响应**的时间，反应"及时性")；

等待时间 waiting time(在 ready queue 中等待时间，等于周转时间 - 运行时间)；

面向系统：**吞吐量 throughput**(单位时间内完成的进程数)； CPU utilization =CPU 使用时间 / 总时间。**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：按照进程先来后到的顺序进行调度，非抢占，最简单，利于长进程，不利于短进程，利于 CPU 不利于 IO 型，

Shortest-Job-First (**SJF**) Scheduling：选择需要**运行时间最短**的进程先运行。SJF 是最优算法，能保证平均等待时间最小，但难以实现。一种解决方法：用 exponential average 预测时长，

新预测=(1-a)*第 n 次时长+ a*原预测

非抢占式(也叫 Shortest-Next-CPU-Burst)：允许当前进程运行完再运行最短的。

抢占式(也叫 Shortest-Remaining-Time-First ，**SRTF**)：总是执行**剩余**运行时间最短的进程。

优先级调度

总是把 CPU 分配给就绪中最高优先级的进程。**一般数字越小优先级越高**(看题)

HRRN：**高响应比优先调度算法** (Highest Response Ratio Next) 响应比=作业周转

时间/作业处理时间=（作业处理时间+作业等待时间）/作业处理时间

SJF 其实就是把执行时间作为优先级的优先级调度特例。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)

就绪中的进程按照 FCFS；每次调度时将 CPU 分给队首进程,执行一个片;每用完一个时间片就让出 CPU（状态变为 Ready）。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 CPU 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，上下文切换开销过大。

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；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

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 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。优点：提高吞吐量降低平均周转而照顾断进程；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

由两个信号产生竞争，其竞争情况影响最终结果的情况，被称为**竞态条件(race condition)**，我们要避免这种竞争的出现

The Critical-Section Problem

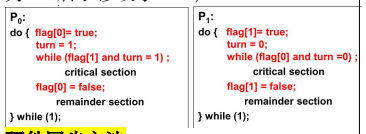
只能被至多一个用户占有的资源为**临界资源**，程序中访问临界资源的代码段为**临界区段(critical section, CS)**。

CS 问题：如何保证只有一个进程进入临界区。解决方案必须满足：**互斥(mutual exclusion)**，操作同一临界资源的临界区段应当互相排斥；**选择时间有限(progress)**，只有现在需要进入临界区域的进程需要被选择，且选择时间有限；**等待时间有限(bounded wait)**，进程等待被允许进入 critical section 的时间有限，不能无限等待。

Peterson 算法

是一种软件解决方法，只适用于两个进程的情况，并且在现代 OS 中不适用。

这里的 turn 标记了“先来后到”，如果 flag 都 ready 且 turn=0, 说明 P0 是先来的(因为 P1 后手修改了 turn)



硬件同步方法

单处理器：在临界区禁止中断。

多处理器：**Memory barriers** 进程对内存做的修改立刻对其它处理器可见

Hardware instruction

Test_and_set()：将目标设为 true，同时返回其旧值。该操作是原子性的。

compare_and_swap(int * target, int expected, int new_val)：如果目标旧值为 expected，则赋值为 new_val。最终返回旧值。该操作是原子性的。

显然，t_a_s(target) 就是 c_a_s (target, 0, 1)。

Test_and_set 存在忙等的情况，因为会一直卡在 while 循环里。

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

      /* critical section */

   lock = false;

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

这种使用忙等待的互斥锁，也被称为自旋锁(spinlock)

虽然自旋锁的忙等会浪费 CPU 性能，但在等待时间并不长的情况下，是一种很好的选择（**是多核系统的首选，但不能用于单核系统**）避免了切换进程的 overhead。

下图为使用 c_a_s()完成的原子自增操作

```
increment(atomic_int * v) {
    int tmp;
    do {
        tmp = *v;
    } while ( tmp != compare_and_swap(v, tmp, tmp+1) );
}
```

优点：进程数随意，简单；

缺点：无法保证等待时间有限，可能饥饿（解决方法：让当前释放锁的进程手动选择下一个进入 critical section 的进程），会引起忙等。

信号量 semaphores

注意<atomic> <atomic>

wait(reference S) { signal(reference S) {

while (S <= 0); S++;

S--;} }

计数信号量：大于等于 0 即可；初始值代表资源个数（同时允许访问的进程）。

二值信号量：只能是 01，也叫互斥锁(mutex locks)。

wait(P)和 signal(V)成对出现，互斥操作就出现在同一进程，同步操作就出现在不同进程。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。

原始的信号量实现会导致忙等。可以引入一个链表来维护等待中的进程，从而避免忙等，此时负数的绝对值代表等待该信号量的进程数。

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
    add this process to S->list;
    block();    } } (进程状态变为 wait
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P); } }(进程状态变为 ready
```

优先级倒置问题(priority inversion)

优先级较低的进程(PL)持有优先级较高进程(PH)所需的锁，导致 PH 无法优先执行。解决方法：**优先级继承（Priority Inheritance）**:临时让 PL 继承 PH 的高优先级，以迅速完成其工作释放锁，释放后恢复原来的低优先级

Dining-Philosophers Problem 哲学家进餐

典型的同步问题。N 个哲学家坐在圆桌，两人之间有一根筷子（共 N 根）；吃饭要用身边的两只筷子一起吃。

一种朴素的解法是：

```
process() {
    wait(chopstick[i]);
    wait(chopstick[ (i+1) % N ]);
    // eat rice
    signal(chopstick[i]);
    signal(chopstick[ (i+1) % N ]);
}
```

但这很显然会导致死锁，如果 5 个哲学家同时饥饿，同时拿起左手筷子，就死锁了。

一些解决方法：①最多只允许 4 个哲学家同时拿起筷子②必须同时拿起两根筷子，不能抓一支等一支③奇数先拿左手，偶数先拿右手。

Bounded-Buffer Problem 有限缓冲区生产者-消费者问题

Producer 生产数据放入 buffer，consumer 从 buffer 取出数据使用。但需要保证 0≤Buffer<n

所以需要 full 和 empty 两个信号量控制上下界，同时用 lock 互斥锁控制 buffer 的读写。

```
semaphore lock = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;
// 保证 empty + full = BUFFER_SIZE
producer() {
    wait(empty); // if buffer is full(empty = 0), wait
    wait(lock);
    add_to_buffer(next_produced);
    signal(lock);
    signal(full);  // full++
}
consumer() {
    wait(full); // if buffer is empty(full = 0), wait
    wait(lock);
    take_from_buffer();
    signal(lock);
    signal(empty);  // empty++
}
```

Readers-Writers Problem

数据库读写的抽象问题。主要需要解决两个冲突：reader 和 writer 的冲突；writer 和 writer 的冲突；（reader 和 reader 之间没有冲突）

用 rw_mutex 控制 readers 和 writer 之间的互斥，用计数器 read_count 表示目前有多少读者，同时用 mutex 互斥锁维护 read_count。

```
writer() {
    wait(rw_mutex);
    /* critical section */
    signal(rw_mutex);
}
reader() {
    wait(mutex);
    read_count++;
    // 只有第一个读者需要获取 rw_mutex
    if (read_count == 1) wait(rw_mutex);
    signal(mutex);
    /* critical section */
    wait(mutex);
    read_count--;
    // 只有最后一个读者需要释放 rw_mutex
    if (read_count == 0) signal(rw_mutex);
    signal(mutex);
}
```

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。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。

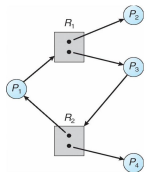
四个必要条件：

①**互斥(mutual exclusion)**：死锁中的资源必须是非共享的

②**持有并等待(hold&wait)**：死锁中的进程必须至少占有一个资源并且等待另一个资源

③**非抢占(no preemption)**

④**循环等待(circular wait)**：死锁中的进程存在环状的等待资源关系，即 wait for graph 中存在环；



资源分配图，由点 V 和边 E 组成，V 被分为两部分：系统活动进程的集合 系统所有资源类型的集合。进程 Pi 到资源 Rj 的有向边记为 Pi->Rj，表示进程 Pi 已经申请了资源类型 Rj 的一个实例，叫请求边；资源 Rj 到进程 Pi 的有向边表示资源类型 Rj 的一个实例已经分配给了进程 Pi，叫做分配边。进程用圆表示，资源类型用方表示，资源实体是方内部的点。

如果分配图无环，必然无死锁；如果有环**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(预防) avoidance(避免)；允许进入死锁但是需要恢复：detection(检测) recovery(恢复)。

Unix Linux Windows 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

破坏死锁的四个条件中的某一个：

Mutual exclusion：不可能

Hold and wait：可行：进程一旦申请资源，就必须一次性获取所有资源；如果没法一次性获取所有资源，就释放已经申请到的资源。

No preemption：允许抢占，但不常用

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。但是扩展性较差

死锁避免

通过获取一些额外的事先信息（prior information）从而避免死锁。

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。

N 个进程,每个进程最多请求 M 个。不发生死锁的最小资源个数为 N(M-1)+1

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 可等待知道所有 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：

每个进程需要说明最大需求；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。

need=max-alloction 分配条件是 need<available，然后 available+=allocatio

	Allocation				Max				Need			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	0	0	1	2	0	0	1	2	0	0	0	0
P1	1	0	0	0	1	7	5	0	0	7	5	0
P2	1	3	5	4	2	3	5	6	1	0	0	2
P3	0	6	3	2	0	6	5	2	0	0	2	0
P4	0	0	1	4	0	6	5	6	0	6	4	2

①系统一开始的 Available = (1, 5, 2, 0)

②P0: Need = (0, 0, 0, 0) < (1, 5, 2, 0)，可满足。完成 P0 后，Available = (1, 5, 2, 0) + (0, 0, 1, 2) = (1, 5, 3, 2)

③P2: Need = (1, 0, 0, 2) < (1, 5, 3, 2)，可满足。完成 P2 后，Available = (1, 5, 3, 2) + (1, 3, 5, 4) = (2, 8, 8, 6)

④剩下的所有进程的 Need 均小于 Available，故肯定可以完成，所以系统处于安全状态

⑤安全序列：P0 -> P2 -> P1 -> P3 -> P4

如果此时 P1 再请求(0,4,2,0)，则 P1 那一行会变成 P1：1 4 2 0；1 7 5 0；0 3 3 0，即从 Need 中把这部分给 Allocation。

此时要重新使用银行家算法进行检测，如果不再安全，则要拒绝该请求。

死锁检测

允许系统进入死锁状态的话，但是需要处理死锁，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：

跟银行家算法类似。

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

①终止所有死锁进程，开销很大
②一次终止一个进程，直到不死锁。影响终止进程顺序的因素：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

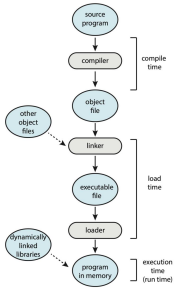
层次存储中，主存(main memory)、cache、寄存器都是 volatile 易失的。

逻辑地址(logical)/虚(virtual)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(base and limit register)控制的。如果 base 寄存器为 300040，limit 寄存器为 120900，那么程序可以合法访问[300040,420910]。

地址绑定的三种情况：



・通常来说，在 compile time，compiler 会将代码中的 symbol 转为可重定位地址(relocatable address)

・在 load time 会把 relocatable address 转为绝对地址(absolute address)。如果进程起始地址发生改变，我们只需要重新装载即可。

・如果进程在 execution time 时允许被移动，那转为绝对地址这一步就要推迟到 execution time 才能做。(需要 base and limit 寄存器支持，绝大多数操作系统都使用这种方案)

・如果在 compile time 就已经知道进程最终被放在哪里，也可以直接生成绝对地址，只不过如果此时起始地址发生改变，就需要重新编译。如果使用虚拟内存,地址绑定在运行阶段

Memory-Management Unit (MMU)

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。

Contiguous Allocation（连续分配）

内存通常分为两个区域：一个驻留 resident 操作系统，一个用于用户进程，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：分区式管理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(内碎片)和动态分区两种。

动态分区(外碎片)：

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-fit（选第一个足够大的）
Best-fit（选所有足够大的里面最小的）
Worst-fit（选最大的）
Next-fit（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）。

First 和 Best 的时空利用率都比 worst 好。

碎片 fragmentation

first 和 best 都存在外部碎片的问题。

外碎片指所有的总可用内存可以满足请求，但是并不连续。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。

内碎片是进程内部无法使用的内存，这是由于块大小固定造成的，比如块大小 8B，进程有 9B，那么不得不给他 16B 的内存，就出现了 7B 的内碎片。

分页存储管理(内碎片)

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。

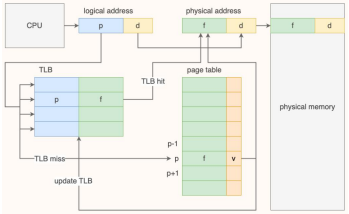
OS 需要跟踪所有空闲帧，叫帧表。

页表的实现

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。这些信息应该被存储在 PCB 中，在线程切换的时候也要切换。

这样的模式下每次内存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。



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
TLB 命中率=a
那么 EAT=(t1+t2) + t2*(1-a)，因为 TLB miss 后要再去 page table 里读一次 (内存访问一次)

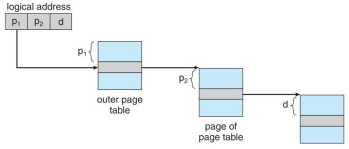
保护 protection

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。

共享页(shared page)：多个页可以对应同一个帧，可以用来提高代码重用率。例如，多个进程可能会使用同一个库，那么这个库就可以被共享，而不需要每个进程都各自在物理内存中准备一份。

分级页表 Hierarchical page table

由于现代计算机逻辑地址空间很大，导致页表会很大，而且页表还要连续，所以不现实。因此要将页表划分变小。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。

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(外碎片)

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 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。

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。

虚存是一种虚拟的存储器系统，具有请求调入功能和置换功能，仅把进程的一部分装入内存便可运行进程，能从逻辑上对内存容量进行扩充的。

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。

可以想象，在程序开始的时候会产生大量的 page fault，但这是一种预期内的 page fault。

在 demand paging 系统里，页是动态地被映射到帧的，所以我们需要维护一个**可用帧列表(free-frame list)**，用来记录当前哪些帧是空闲的。

页错误 Page fault

非法地址访问，或访问不在主存 or 无效的页都会引发 page fault。

Page fault rate 等于 1 不代表所有 page 都是 page fault，因为这个值通常是一个时间段内的统计值。

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

为了计算 EAT，必须知道需要花多少时间处理 page fault，page fault 会引起以下动作的产生：
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.从磁盘读页到内存的空闲帧(包含磁盘队列中的等待 磁盘的寻到 旋转延迟 磁盘的传输延迟)
6.在等待过程中的 CPU 调度
7.IO 中断
8.保存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.修正页表和其他相关表，所需页已经在内存中
11.等待 CPU 再次分配给本进程
12.恢复用户寄存器、进程状态和新页表，重新执行。
其中的三个主要的部分是：缺页中断服务时间，缺页读入时间，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。

COW 加快了进程创建速度。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。

页面置换

当 free-frame list 为空，但用户仍然需要 frame 来进行 page in 时，就需要进行**页置换(page replacement)**，将并没有正在被使用的页腾出来给需要 page in 的内容用，而这个“被要求腾出地方”的页，我们称之为**牺牲帧(victim frame)**。

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（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），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。

置换 replacement 分为 local 和 global 两种：

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。大多数 OS 采用这种。

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。

页面置换算法

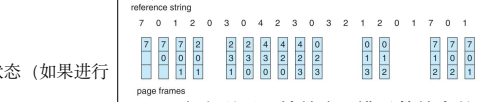
采用最小页错误率的置换算法。

评价标准：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。



FIFO 会出现可用帧越多，错误数越大的问题，这种现象叫 Belady’s Anomaly

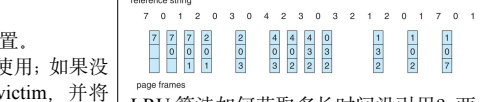
Optimal Page Replacement OPT 最佳页面置换

所有算法中页错误率最低的算法，且绝对没有 Belady’s Anomaly。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。实际中无法实现

M 个 frame, N 个 page，P 次访问，Page fault 的次数下限是 N。上限是 P

Least Recently Used LRU 最近最久使用

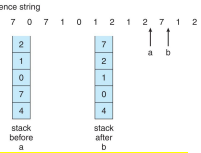
LRU 选择内存中最久没有引用的页面，考虑的是局部性原理，性能最接近 OPT，但是需要记录页面的使用时间，硬件开销太大。就是向过去看的算法。



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栈顶部，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，往往是做一个近似：为页表中的每项加一个引用位 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)，否则就按照 FIFO。

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，值越大说明越最近使用。其中最小值的页是 LRU 页，被置换。

Second chance 二次机会/clock 算法

FIFO 的改进版：选择 victim 页时，**循环遍历** frames 并检查引用位，如果为 0 直接置换。如果为 1，将其引用位清零，给该页第二次机会，选择下一个 FIFO 页。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/NRU(Not Recently Used)增强二次机会/clock 算法

唯一一个考虑了 dirty 位的算法。思路在于给未引用但是修改了的页更高优先级，降低了 IO 数。

同时考虑 reference bit 和 dirty bit，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)

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

每个进程需要的最小数目的页，取决于操作系统。

预调页 prepaging

为了减少冷启动时大量的页错误，会在刚开始预先分配一些页。

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 page fault 的成本是否大于其他 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。

两种分配模式：平均分配，比例分配

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。

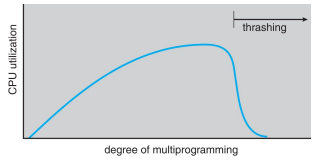
优先级分配 proportional allocation with priority 同样按比例分配，但是引入优先级权重。

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抖动，也就是一个进程不断换入换出页



抖动会导致 CPU 利用率低，然后 OS 认为 CPU 太闲了需要增加多道程序程度，结果进一步引发抖动，CPU 利用率进一步降低。

本质原因：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。

解决方法：使用 priority replacement algorithm，只允许高优先级的进程置换低优先级的 frame

工作集合模型 Working set model

WS 工作集：最近 dleta 个页的引用。

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=在最近的 delta 内总的页面引用次数

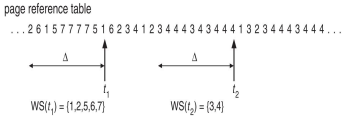
D=所有 WSS_i 求和=帧的总需求量。

m=帧的总可用量。

如果 D>m 就会发生颠簸，此时暂停一个进程。该进程的页面被换出，且其帧可以被分配给其他进程。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，最老引用会丢失。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 pre-paging，但是控制颠簸不是很灵活，更直接的方法是用 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

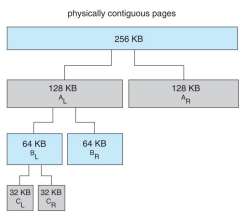
与对待用户内存不同，内核内存从空闲内存池中获取，两个原因：1.内核需要为不同大小的数据结构分配内存。2.一些内核内存需要连续。

Buddy 系统

从物理上连续的大小固定的段上进行分配。如果要求大小为 n，那么分配 2^k，其中 k=log2(n)

优点：可以通过合并快速形成更大的段。

缺点：容易产生内碎片，因为要调整到下一个 2 的幂。



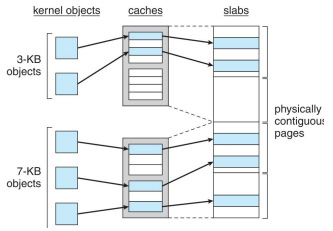
通过不断二分的方式来寻找一块合适的内存

Slab 分配

Slab 分配的大概思路是预先了解到 kernel 内的常见数据结构（被称为各种 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了苹果小盒、冰箱大箱、超大车库，然后根据要放的东西的大小选择合适的空间来放。

优点：避免碎片引起的内存浪费；内存请求可以快速满足。



页大小的影响

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 应该位于 TLB 中，否则就会有不通过 TLB 调页导致的大量 IO

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

IO 互锁

IO 互锁指页面必定有时被锁在内存中。

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(metadata)

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

文件操作

Syscall 实现，文件是 ADT 抽象数据类型，其操作有：(1) Create：在文件系统中为文件分配一块空间，然后在目录中创建对应的条目；(2) Delete：在目录中找到对应条目并删除该条目，如果此时对应的文件没有其它硬链接，还需要释放空间 (3) Open：除 create/delete 的操作都要先调用 open，成功则返回 handle 作为其他调用的参数；(4) Read / Write 维护 Current-File-Position Pointer 表示当前操作的位置，在对应位置上做读写操作; (5) 文件内重定位 Seek：按给定值重定位指针；(6) Truncate：清空文件内容，保留文件属性

打开文件

每个打开的文件都有以下信息：

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：①mandatory lock:根据目前锁请求与占有情况拒绝 access；②advisory lock:进程查看锁情况来决定访问策略

文件内部结构 File Structure

None 字节的序列 流文件结构

Simple record structure 记录文件结构：lines, fixed length, variable length

Complex Structures：formatted document, relocatable load file

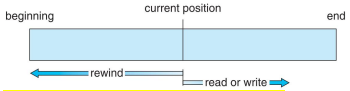
可以通过插入适当的控制字符，用第一种方法模拟最后两个

这些模式由 OS 和程序所决定。

访问方法

Sequential access 顺序访问

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/随机访问

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。

Indexed block access 索引顺序访问访问

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

是顺序访问和直接访问的结合,建立在直接访问上。文件中的数据按顺序存储，但同时维护一个索引,指向各个块,用于快速定位特定记录。优点：可以顺序访问也可以直接访问，支持高效查找（索引通常存储在内存中，查找速度较快）

对于大文件,索引文件可能体大,则建立 Primary index 指向 Secondary index,

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

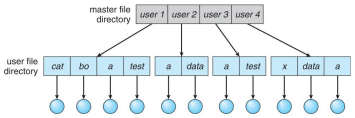
搜索文件 创建文件 删除文件 遍历 list 目录 重命名文件 遍历 traverse 文件系统

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

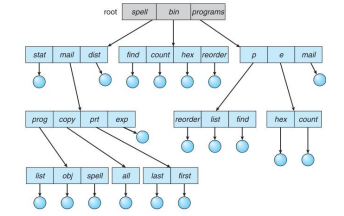
二级目录

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 可用于 solve name collision)

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

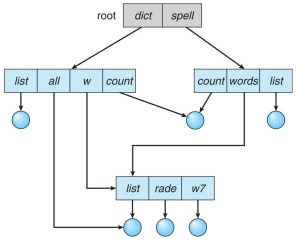
树形结构禁止共享文件和目录。无环图允许目录含有共享子目录和文件。

实现文件盒目录共享, UNIX 采用创建一个叫做链接的新目录条目。链接实际上是另一个文件的指针。连接通过使用路径名定位真正文件。

注意，无环图目录倒置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。符号链接(软链接)存在 dangling pointer 问题: 删除一个文件后指向该文件的其他链接成为 dangling。

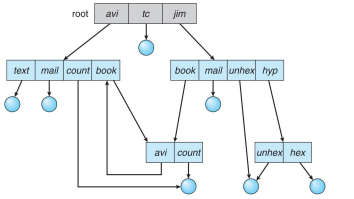
软链接又称符号链接(Symbolic Link)，是一个指向文件的指针，类似于快捷方式。删除被软链接指向的那个文件，并不会连带地处理软链接，但是原先的这个软链接已经失效了。从本质上来看，软链接是特殊的文件。

硬链接是复制链接文件目录项的所有元信息，存到目标目录中，此时文件平等地属于两个目录。在文件元信息被更新时，需要保证在若干目录下该文件的信息是一致的。删除被硬链接的文件并不会直接导致文件被删除。只有当用来记录「被硬链接数量」的 Reference Counter 减至 0 时，文件才会被删除；其他情况下都只需要在当前目录中删除该表项，并将 Reference Counter 减 1，更新相关元信息即可。从本质上来看，硬链接是目录表项。



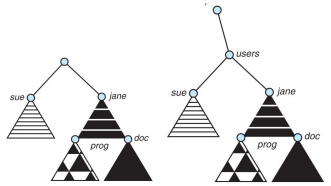
普通图目录 General graph

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。

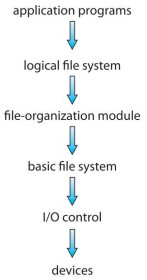
File System Implementation 文件系统实现

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。

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：包含文件的属性，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

应用程序：发出文件请求者。

逻辑文件系统：管理元数据：文件系统的所有结构数据，而不包括文件的实际顺序；根据给定符号文件名来管理目录结构；**逻辑文件系统通过 FCB 来维护文件结构。**

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：向合适的设备驱动程序发送一般命令就 可对磁盘上的物理块进行读写

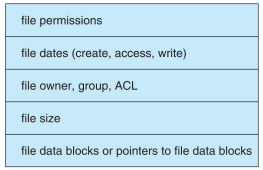
IO 控制：由设备驱动程序和中断处理程序组成，实现内存与磁盘之间的信息转移

On-Disk FS structure

File control block(FCB) (per file)

Directory (per FS)：文件名到 FCB 的映射，包含文件名和对应 FCB 的标识号。在 NTFS 中，也被存在 master file table 中。

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。



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

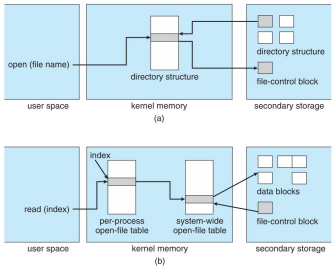


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

System-Wide Open-File Table：记录所有被 load 到内存中的 FCB inode；Per-Process Open-File Table：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）。

一个 **open** 操作，需要：①用 filename 在 FS 中的目录结构中查找这个文件。找到后，其 FCB 会被复制进 System-wide open-file table（若被其他进程打开过，直接在系统级打开表搜索）②在当前进程的 Per-process open-file table 中创建一个 entry，指向刚才那个 entry，同时该 entry 的 reference counter 加 1，表示该文件被打开。返回一个指向这个 entry 的指针③之后，对文件的所有操作都依赖这个指向 per process open-file table 中 entry 的指针。该指针在 UNIX 中被称为 file descriptor，在 Windows 中被称为 file handler。

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 linear list:

使用储存文件名和数据块指针的线性表。哈希表：线性表与哈西结构，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。

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

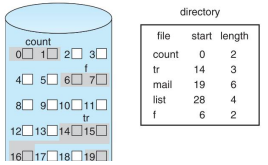
每个文件在磁盘上占有一组连续的块。

优点：支持 random access，访问很容易，只需要起始块位置和块长度就可以读取。

缺点：浪费空间，存在外碎片问题；文件大小不可增长，存在动态存储分配问题。

First-fit 和 Best-fit 表现差不多，但 first 时间快很多。

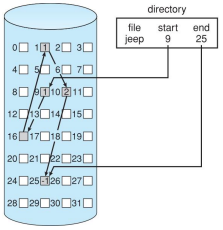
变种：基于长度的系统。利于 Veritas FS 采用。解决了文件大小无法增长的问题，增加了另一个叫做 extent 的连续空间给空间不够的文件，然后与原文件块之间有个指针。一个文件可以有多个 extent。



Linked Allocation 链接分配

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。

LA/(512-1)得到商 Q 和余 R, 商作为块的索引下标，余数作为块内的 offset。因为每个物理块的末尾存储下一个物理块的地址，所以这里要除以 512-1。



FAT File allocation table 文件系统

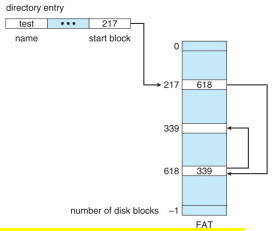
磁盘空间分配用于 MS-DOS 和 OS/2，FAT32 引导区记录被扩展为包括重要数据结构的备份，根目录为一个普通的簇链，其目录项可以放在文件区任何地方。原本的链接分配有问题，指针在每个块中都会占空间，可靠性也不高，任何指针丢失都会导致文件其余部分丢失。FAT 采用单独的磁盘区保存链接。

为了提高可靠性，**文件分配表（FAT）**存储在磁盘上一个 **单独的、专用的区域**。

①引导扇区（Boot Sector）是第一个扇区，用于存储文件系统的基本信息。其中 BPB（BIOS Parameter Block）用于存储文件系统的基本参数。②保留扇区（Reserved Sector）保留给文件系统使用，不存储文件数据。主要包含了引导代码和文件系统的一些元数据③FAT：指示每个 cluster 是否被占用，并为一个文件中的所有 data cluster 建立一个链表。

每个 cluster 会被分配一个号码，FAT 表存储着这些 cluster 的映射关系，FAT 表一般有多个，用于冗余和容错④数据区：存储了所有文件数据。根目录区（Root Directory Region）可以在数据区任何地方，所在的第一个 cluster 的位置由 BPB.root_clus 指定。

计算机系统启动流程：①首先执行的是 BIOS 引导程序，完成自检②并加载主 boot record 和分区表③然后执行 boot record，由它引导激活分区 boot record 并执行④加载各分区的操作系统⑤最后执行操作系统，配置系统。

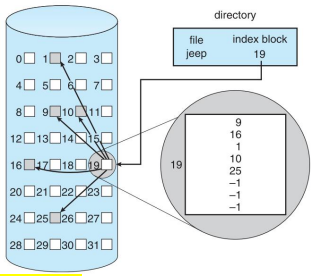


Indexed Allocation 索引分配

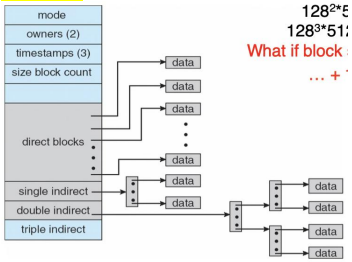
每个文件记录一个 索引块 (index block)，记录这个文件的第 i 个块在磁盘中的哪个块。每个文件都有索引块，是一个磁盘块地址的数组。当首次写入第 i 块时，先从空闲空间管理器获得一块，再将其地址写到索引块中的第 i 个条目。对于小文件，大部分索引块被浪费。如果索引块太小，可以多层索引、然后互相连接。访问需要索引表，支持 randomaccess，动态访问没有外碎片，但是有索引开销。LA/512 得商 Q 和余数 R，Q = displacement into index table R = displacement into block

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 时索引表的块号，R1/512 得到商 Q2 和余数 R2，Q2 是 displacement into block of index table，R2 是 displacement into block of file。

如果是二级索引，那么 LA/(512*512)得到 Q1 和 R1，Q1displacement into outer-index，R1/512 得到 Q2 和 R2，Q2 是 displacement into block of index table，R2displacement into block of file:



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。

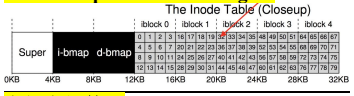
一级索引：一个索引块可以存 4KB/4B=1K 个索引地址，每个索引地址直接引到文件块，所以最大 1K*4KB=4MB。

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

AFS example with Inode region



空闲空间管理

位图 **bitmap**：空闲置 0 占有置 1，块数计算 (number of bits per word)*(number of 0-value words)+offset of first 1bit

位向量所需空间的计算：disk size/block size 便于查找连续文件。

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。

对空闲链表的改进是将 n 个空闲块的地址存到第一个空闲块中。这样可以快速找到大量空闲块的地址。
还有**计数**的方法：不记录 n 个空闲块的地址，而是记录第一各空闲块和紧跟着的空闲块的数量 n。

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

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除以服务请求到传递结束的总时间。
平均旋转延迟=一圈/2

$$\text{Average I/O time} = \underbrace{\text{average seek time} + \text{rotational latency}}_{\text{average access time}} + \underbrace{\frac{\text{data to transfer}}{\text{transfer rate}}}_{\text{transfer time}} + \text{controller overhead}$$

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。
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

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

三级存储 Tertiary storage device

Low cost is the defining characteristic of tertiary storage. Generally, tertiary storage is built using removable media Common examples of removable media are floppy disks and CD-ROMs; other types are available 硬盘闪存是二级存储,内存,寄存器,cache 是一级存储

Swap space

虚存使用硬盘空间作为主存。两种形式：普通文件系统：win 都是 pagefile.sys 独立硬盘分区 linux solaris 都是 swap 分区。还有一种方法：创建在 raw 的磁盘分区上，这种速度最快。

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。

IO system

IO 方式

轮询 polling
中断 CPU 硬件有一条中断请求线 IRL，需要 IO 时就申请中断。(如鼠标,键盘)
DMA direct memory access
CPU 告知 DMA 控制器源地址、目的地址和字节数。DMA 控制器直接在设备和内存间传输数据，仅在结束时中断 CPU. DMA 只做大量的、以 blocks 为单位的数据传输。
磁盘用 DMA

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(driver)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

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：键盘鼠标打印机串口，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

习题

系统设计了文件控制块（FCB）结构来管理文件。通常，当调用开放系统调用时，会在磁盘上创建文件控制块。

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 $2^{32}\times(1KB-4B)=4080GB$

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。