概述-蒋奕-3210103803

操作系统作为用户和硬件之间的中间人。

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

计算机系统分为：硬件、OS、应用程序和用户。

OS 是用户与计算机硬件之间的接口。OS 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 system call

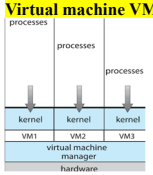
OS 是计算机系统资源的管理者,是资源分配者是控制程序,是**扩充裸机的第一层系统软件**。

program running all times on computer is **kernel**

大型机系统 Mainframe System

OS 有：批处理系统、分时系统。批处理分为多道和单道。Multiprogramming:多个 job 存在 memory 中，CPU 在它们间进行切换。发展：no software->resident moniters->muli-programming->multi-tasking

Virtual machine VMM/virtualization



OS natively compiled for CPU, running **guest** OS also natively compiled. **VMM can run natively, in which case they are also the host**

Time Sharing/multitasking 系统-RR 调度算法

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 timeslice.进程数(用户数)为 n，每个进程的运行时间片为 q，则系统的 response time＝nq

集群系统 Clustered：一组互联主机构成的统一的计算机资源,用于高性能计算。用 **storage-area network (SAN)**分享存储。对称: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。

实时系统 Real Time 有软实时 硬实时

手持 Handheld：PDA cellular telephone

嵌入式.

操作系统市场格局 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

Dual mode 用户态：执行应用程序。内核态：执行操作系统程序时。

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

Interrupt：Caused by external event 可在指令执行任意时刻产生 Exceptions: Caused by executing instruction 一条指令终止执行后 CPU 才发出

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。每个 **system call** 对应一个 **wrapper routine**，唯一目的就是发布系统调用

system call 是进程和 OS 内核间的程序接口,一般用 C 写，大多数由程序提供的叫做 **API** 应用程序接口，而非直接 system call。常见的 API 是 win32 API，POSIX API 和 JAVA API。

Time(),debugger,lock 也都是 system call

system call 3 种传参方式：(i)寄存器传参(ii)参数存在 memory 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 **(iii)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

操作系统启动过程：

1.从 **ROM** 读 **BIOS/EFI**：硬件 **POST 自检**有无问题，按照 **Boot Sequence** 优先级高低启动硬件.**2.** 主引导记录 **M**aster **B**oot **R**ecord 是排第 1 的储存设备最前面 **512B；引导计算机到硬盘特定位置找 OS。3.**硬盘启动 **3case:(i)**读激活分区的**第一个扇区** Volume **B**oot **R**ecord**(ii)读取扩展分区第 1 个扇区** Extended **B**oot **R**ecord 接着读第 2 个逻辑分区的第 1 个扇区直到某个逻辑分区的分区表只包含它自身 1 个分区.**扩展分区可包含多个逻辑分区(iii)**运行已安装 **boot loader** 由用户选择启动哪个 **OS.4.OS 内核加载，init 进程运行，最终被 load 到 RAM。**

Bootstrap Loads operating system kernel and starts execution

计算机系统启动时，首先执行 BIOS 引导程序，完成自检，并加载 MBR 和分区表，然后执行 MBR，由 MBR 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。

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)

代码编译**(i)**.c+complier->.o**(ii)**.o+**linker**->.exe**(iii)**.exe+**loader**+DLL->prog in mem

Dynamic linking loader 链接 ELF 与 lib, .interp

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。

Static linking _start 地址在 execve 调用后立刻执行；动态链接：先执行 ld.so

OS architecture

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：(效率不高)OS 被划为很多层，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功能/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。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**开销大效率低**。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

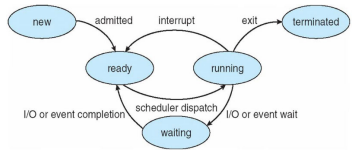
大多数现代操作系统不是单一模型。Linux 和 solaris 是 monolithic+module。Windows 大部分是 monolithic 加上 microkernel。Mac 是层次 Hybrid

Cloud Computing

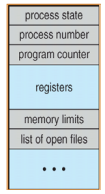
Delivers computing, storage, even apps as a service across a network.分为 **SaaS 软件即服务**(i.e. word processor);**PaaS 平台即服务** (i.e a database server); **IaaS 基础设施即服务** (i.e. storage available for backup use)

Process 进程

进程包括： PC,register,text section(code),data section(global data),stack(临时 data),heap(alloc)。



进程状态：waiting 往往没获得资源(读取文件失败/等待 IO).会因 system call,OS(调度),interrupt 改变状态。Wait->run 和 ready->wait 一般不发生。N 核理器下，最多 n 个 run，ready 进程构成就绪队列，wait 进程构成多种等待队列。单处理器 run 最多 1 最少 0,wait 最多 n 最少 0,ready 最多 n-1 最少 0。



一个程序可对应多个进程，一个进程可包括多个程序

PCB 表示进程，包括

Process state, Program counter, CPU registers,CPU scheduling information, Memory-management information

Accounting information File management I/O status information

Linux 的 PCB 保存在 struct task_struct 里

Fork: fork() -> exec*() [replace memory space] -> parent calls wait() for children to terminate

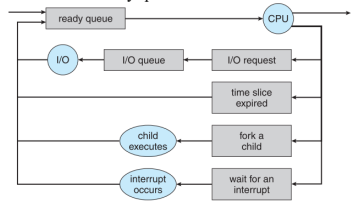
wait(): block until any child terminates. waitpid(): block until a specific child completes

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

Zombies&Orphans: Zombie 子进程运行终止，父进程尚未调用 wait(). Orphan 父进程没有调用 wait()就终止 is “adopted” by process with pid 1.

进程调度

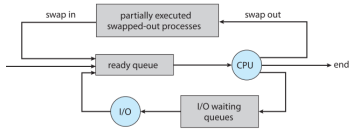
调度队列有： jobqueue,readyqueue,devicequeue 进程在这些队列内移动。下图是队列图，新进程开始就处于 ready queue



长程/作业调度:选择应被带入 readyqueue 进程，频率低,controls degree of multiprogramming **(内存中进程数)**,现代 OS unix win 没有长程

短程/CPU 调度：从接下来要执行的进程中选择并分配 CPU，频率高。

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



处于临界区(CS)进程也可以被调度

进程分为 IO 型(many short CPU bursts),CPU 型(few very long CPU bursts)

Context Switch：CPU 切换进程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**。上下文切换是 overhead，过程中不能进行其他事务。**不影响 global 变量,影响 memory**

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 不能 share same address space

Shared memory: POSIX 用 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).实现 communication link：通过 **mailbox** 接受信息(新操作：create mailbox, destroy mailbox）Message synchronization: blocking – synchronous, non-blocking: asynchronous

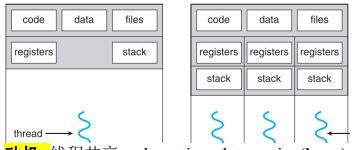
Pipe: Ordinary pipe(anonymous) – unidirectional, parent-child relationship (fd[0]= read end; fd[1]=write end); **Named** pipes: bidirectional

RPC remote procedure call 与 local call 不同

Thread 线程拥有少量的系统资源

Process 是 unit of resouce ownership.thread 是 unit of dispatch，OS 将它们分别处理，调度单元被称为 thread/LWP lightweight process.资源拥有单元被称为 process。线程就是进程内一个执行单元或可调度的实体。 **heavyweight/traditional thread= task with only 1 thread**

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。**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

User thread：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/阻塞则整个进程在等待。

Kernel thread：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**影响其他线程。时间片分配给线程，多线程的进程获得更多 CPU 时间。

Multithread model

多对一：多个用户级线程映射到 1 个内核线程。由 **thread library** 在 **user space** 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：线程操作有 **High overhead**,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）

2 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。**Signal Handling**：信号由 particular event 产生，信号**必须**要发给 **process**，信号被发送后要被处理。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。**Thread pool** 用现有线程处理请求比等待创建新线程快；限制可用线程数量。**Thread Specific Data** 线程自己保存数据拷贝，在无法控制创建线程时有用(用线程池时)

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 提供从内核到线程库的通信机制 **upcalls**,保证了应用可以维持正确数量的线程。Win xp linux

WIN XP 实现 1 对 1 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 线程叫 **task**，除了 **fork**，额外提供线程创建通过 **clone** 系统调用完成,它允许子任务和父任务共享地址空间

CPU/process Scheduling

调度时机:**run->wait/terminate**(只有这 2 个**非**抢占),run->ready(高优先级抢占),wait->ready..调度只发生在时间片用尽叫 non-preemptive，否则 preemptive

dispatcher 将 CPU 的控制交给由**短**程调度选择的进程。功能有：上下文切换 切换到用户模式 跳转到用户程序的合适位置来重启程序。

分派程序停止一个进程而启动另一个所需要花费的时间叫 **dispatcher latency**

高响应比优先调度算法 Highest Response Ratio Next(**HRRN**)响应比 R = (等待时间+ 要求执行时间)/ 要求执行时间

调度算法的选择准则和评价：

Waiting time = turnaround time – burst time

turnaround time = termination time – arrival time

面向用户：**turnaround time**(进程从 commit 到完成所用时间 包括 ready+wait 时间 **带权周转时间=周转时间/CPU 执行时间**) 响应时间 **waiting time**(在 ready queue 中等待时间) 截止时间 公平性 优先级。面向系统：吞吐量 throughput 处理机利用率 CPU utilization 设备均衡利用。调度算法自身：易于实现 开销较小。最佳算法准则：CPU 利用率 吞吐量 周转时间 等待时间 响应时间 公平

First-Come, First-Served (FCFS)：根据就绪状态的先后分配 CPU，非抢占，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)：对预计执行时间短的作业优先分派 CPU，分为抢占式：如果新来的进程时间比当前的还短，抢占，这种 SJF 叫做 **Shortest-Remaining-Time-First (SRTF)**。非抢占：允许当前进程运行完再运行最短的。SJF 给出最小平均等待时间。而 **SJF 无法实现因为不知道下一个 CPU 脉冲 burst 时长**。

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

新预测=(1- α)第 n 次时长+ α 原预测

优先级调度

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

时间片轮转调度 Round Robin(RR)

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

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

Multilevel Queue Scheduling 将就绪队列根据性质/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

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. 新进程创建后，先放入队列 1 的末尾，按 RR 算法调度；若按队列 1 一个时间片未能执行完，则降低投入到队列 2 的末尾，同样按 RR 算法调度；如此下去，降低到最后的队列,则按 FCFS 算法调度直到完成。仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢占执行新进程，并把被抢占的进程投**

入原队列的末尾。

Thread Scheduling 可以 FCFS/SJF

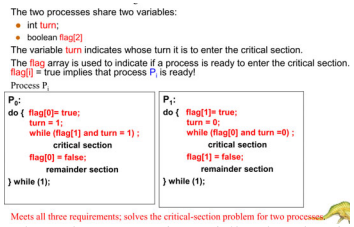
Process synchronization

The Critical-Section Problem

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

Peterson 算法

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



现代 OS 中不适用(若编译器交换两条指令则奇)

硬件同步方法

单处理器：在 CS 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 原语

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

    return rv;
}
```

```
while(1) {
    key = TRUE;
    while (key == TRUE)
      Swap(&lock,&key);
    critical section
    lock = false;
    remainder section
}
```

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

   /* critical section */

   lock = false;

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

Test_and_set 存在 **busy waiting**。会忙等待的信号量是 **spin lock**

抽象出 2 个硬件实现的**不可以被中断的原子操作**：赋值和交换，然后来解决临界区，testandset 的共享变量是 lock 初始 false，swap 也是一样，但是多了局部变量 key 不是共享的。

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

semaphores

atomic 操作 wait(P,不用 waiting queue 会 busywaiting)，signal(V,不会 busywaiting)

count 信号量，值域不受限；

binary 信号量=互斥锁 **mutex**，只能是 01(可以被>2 个线程使用,**不会 block,会 busywaiting**).

为消除 wait(P)时的 busy waiting，对信号量增加 block(run->wait)和 wakeup(wait->ready)来**避免忙等** wait(semaphore *S) { S->value--; if (S->value < 0) { add this process to S->list; block();} } signal(semaphore *S) { S->value++; if (S->value <= 0) { remove a process P from S->list; wakeup(P);} }具有忙等的信号量值非负，但是这种实现可以为负，**负数绝对值代表等待该信号量的进程数，0 代表无资源可用.**

Mutex of 2 concurrent processes has value
0: **1** process entered CS, **0** process blocked
Count S of 3 concurrent processes has value
1: **0** process entered CS, **0** process blocked
0: **1** process entered CS, **0** process blocked
-1: **1** process entered CS, **1** process blocked
-2: **1** process entered CS, **2** process blocked

Wait 和 signal 成对出现，互斥操作就在同一进程出现，同步操作在不同进程。**连续的 wait 顺序是需要注意**的，但连续 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

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.

Monitor 管程

一次只有 **1** 个进程能在管程内活动.内部可定义 **condition** 类型的变量以提供同步机制

Bounded-Buffer Problem 有限缓冲区 生产者-消费者问题很多相互合作进程

设置 N 个缓冲项；信号量 **mutex** 初始化为 1，用来保证访问 buffer 的互斥要求； **full** 初始化为 0，表示满缓冲项的个数； **empty** 初始化为 N 表示空缓冲项的个数(这里不能只用一个 count 表示 buffer 空的个数，否则就无法实现锁)

生产者：do {…produce an item…**wait(empty); wait(mutex);** …add to buffer …**signal(mutex); signal(full);**} while (1);

消费者：do {**wait(full); wait(mutex);** …remove an item from buffer…**signal(mutex); signal(empty);** …consume the item …} while (1);

Readers-Writers Problem 数据库

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

共享数据有访问的数据、**reader_count_lock** 初始 1,保证更新 **readcount** 时互斥; **write_lock** 初始 1,为读写公用，供写者作为互斥信号量，被第一个进入 CS 和最后一个离开 CS 读者使用，其他读者不适用;**Readcount** 初始 0，跟踪多少进程正在读。

写进程：do {**wait (write_lock) ;** …writing… **signal (write_lock) ;**} while (1);

读进程：do {
wait(reader_count_lock); readcount++;
if (readcount == 1) **wait(write_lock);**
signal(reader_count_lock); …reading …
wait(reader_count_lock); readcount--;
if (readcount == 0) **signal(write_lock);**
signal(reader_count_lock);} while (1);

Dining-Philosophers Problem 哲学家进餐

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

共享数据：一碗米饭；共享变量 **chopstick[5]**初始为 1 表示筷子未被持有；**lock** 初始化为 1 表示某个哲学家是否可以同时拿起左右两根筷子。

解决方案之一：**只允许在 CS 内同时拿起两根筷子**；轮询每人是否能够拿起两根筷子，如果能则拿起，如果不能则需要等待那些筷子放下

do{**wait(lock);wait(chopstick[i]);wait(chopstick[(i+1) % 5]); signal(lock);**…eat…**signal(chopstick[i]);signal(chopstick[(i+1) % 5]);** … think …} while (1);

防止死锁的可能解决：最多只允许 4 个哲学家坐在桌上 do {…thinking…
P(count);/*初始 count=4，是否超过四人准备进餐*/**P(chopstick [i]); P(chopstick [(i+1)% 5]);**…eat..
V(chopstick [i]); V(chopstick [(i+1)%5]); V(count);
}while(1);

使用非对称的解决方法：奇数先拿左手，偶数先拿右手 do {…thinking…
if(i%2 == 1){ **P(chopstick [i]);**//判断左边的筷子是否可用.**P(chopstick [(i+1)%5]);**//判断右边筷子是否可用}else{**P(chopstick [(i+1)%5]);**//判断右边的筷子是否可用.**P(chopstick [i]);**//判断左边的筷子是否可用}…eat..
V(chopstick [i]); V(chopstick [(i+1)%5]);
}while(1);

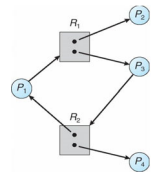
Deadlock 死锁

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

四个必要条件：

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

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

P4 可能释放 R2 的实例，这个资源分配给 P3，这样就打破了死锁。

死锁处理

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

死锁预防 Prevention

通过限制请求的方式来预防死锁。

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

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

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

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

死锁避免

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

最简单和最有效的模型要求每个进程声明它**可能需要的每种类型的资源的最大数量**。死锁避免算法动态检查资源分配状态，确保永远不会出现循环等待。资源分配状态由可用和已分配资源的数量以及进程的最大需求定义。

安全状态：对于所有进程，如果存在一个安全序列，那么系统就处于安全状态。对于进程序列 P1,P2,…,Pn，如果对于每一个 Pi,Pi 仍然可以申请的资源数小于当前可用的资源加上所有进程 Pj(i>j)所占有的资源，那么这一序列是安全序列。这种情况下，进程 Pi 的资源即使不能立即可用，那么 Pi 可等待知道所有 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：

每个进程实现说明最大需求；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。

数据结构：

N 进程数，m 资源类型的种类数；Available：长度为 m 的向量，表示每种资源的现有实例数量，available[j]=k 表示 j 型资源还有 k 个；Max：n*m 的矩阵，定义每个进程的最大需求，max[i][j]=k 表示进程 Pi 最多可以申请 k 个 Rj 型资源；Allocation：n*m 的矩阵，表示每个进程所分配的各种资源类型的实例数，allocation[i][j]=k 表示已经为 Pi 分配了 k 个 Rj 型实例；Need：n*m 矩阵，表示每个进程还需要的剩余的资源，need[i][j]=k 表示进程 Pi 还可能继续申请 k 个 Rj 型的实例。Need=max-allocation。

安全状态检测算法：

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

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

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

4.如果所有的 finish 都是 true，那么系统处于安全状态。

算法需要 m*n*n 的操作数量级确定系统状态

资源请求算法：

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

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

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

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

死锁检测

允许系统进入死锁状态的话，那么系统就需要提供检测算法和恢复算法。

等待图，单实体资源类型检测算法：

等待图是资源分配图的变形，节点都是进程，Pi->Pj 表示 Pi 在等待 Pj 释放 Pi 所需的资源。当且仅当等待图中有一个环，系统死锁，检测环的算法需要 n*n，n 为点数。

多实体资源类型检测算法：

数据结构：Available，alloction 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：死锁发生频率以及思索发生时受影响的进程数。如果经常发生死锁，那么就要经常调用检测。如果在不确定的时间调用检测算法，资源图可能有很多环，通常不能确定哪些造成了死锁

死锁恢复

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

进程终止

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

抢占资源

抢占资源需要处理三个问题：

选择一个牺牲品 victim：要代价最小化

回滚：回退到安全状态，但是很难，一般需要完全终止进程重新执行

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

Main Memory 主存

层次存储中主存 cache 寄存器为 volatie 易失的。

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

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

Address binding3 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 **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)

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 **relocation register**，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

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，所有子程序都以 relocatable form 存在 disk，需要的时候装入 memory**。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 **backing store** 上，当需要再次执行时再调回。需要 **dynamic relocate** 支持

backing store：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。有固定分区和动态分区两种。

动态分区：

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

碎片 fragmentation

first 和 best 都存在 external frags。外碎片指所有的总可用内存可以满足请求，但是并不连续。外碎片可通过 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。

Internal frags 是进程内部无法使用的内存，这是由于零头和块大小造成的，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

分页允许进程的 PA 空间非连续；将物理内存分

为固定大小的块，叫做帧 frame/物理块/页框，将逻辑内存也分为同样大小的块叫做页 page，Linux Win(x86)是 4KB。
OS 需要跟踪所有空闲帧，叫帧表。
运行一个 n 页的程序就需要找到 n 个空帧然后装载进去。
页表项数和内存大小相关 2^m B 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 B 那么逻辑地址长度为 m，页表大小 2^n B，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，然后得到物理真号拼上 offset，然后进入到内存中找 frame。

Page table-放在 memory 里面

PTBR page-table base reg 指向页表，切换页表只需要改变 PTBR 就可 PRLR page-table length register 说明页表长度，这样的模式下每次数据/指令访问都需要 2 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，引入特殊的转换表缓冲区 TLB，是一种硬件 cache。部分 TLB 维护了 **ASID addressspace identifier 用来唯一地标识进程**，为进程提供空间保护。否则每次切换根页表需要 flush TLB。*Associative memory:一种支持并行搜索的内存，如果虚页号与其中键匹配上，则直接返回物理帧。

Effective Access Time EAT

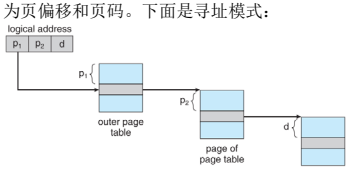
Associative lookup=t1:查 TLB 表的时间
Memory access time=t2:内存访问时间
α:TLB 命中率
那么 EAT=(t1+t2)* α +(t1+t2+t2)*(1- α)
查 TLB miss 后，需要进内存查一次页表，再去取一次数据，命中就直接取数据。
Eg (hit=80%,E(TLB)=2 ns , memory access time=100 ns, EAT = (100 + 2)*0.8 + (100+100+2)*(1-0.8)=122 ns

保护 protection

内存保护通过与每个 frame 关联的保护位实现。Valid bit 存在 page table 中的每一个 entry 上。
Shared code：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 Private code and data 即可。

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。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

Page size=2^{10}B，entry size=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。

Hash page table

超过 32 位 LA 地址空间时，一般采用哈希页表，将 **VPN 的哈希值**作为哈希表索引 key，哈希表每一项都链着哈希值相同的(VPN,PPN)对,然后在查表时用 VPN 与链表中的每个元素进行比较从而查 PPN

Inverted page table（整个系统只有 1 个）

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**。对每个物理内存的帧也只有一条相应的条目。时间换空间，需要为页表条目中添加一个地址空间标识符 **ASID**

分段 Segmentation

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

段页式 存储管理

程序(VM)被划分成若干 segment,每一段给予 segment-number 标识符，然后，对每一 segment 分成若干个固定大小的 page.
LA 结构：

Segment number	Logical Page number	Page offset

Virtual Memory 虚存

虚存将用户的路基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。
locality：时间：指令的一次执行和下次执行 数据的一次访问和下次访问都集中在一个较短时期内；空间：当前指令和邻近的指令 当前数据和邻近的数据都集中在一个小区域内。
虚存是是具有请求调入功能和置换功能，能仅把进程的一部分装 入内存便可运行进程的存储管理系统，它能从逻辑上对内存容量进行 扩充的一种虚拟的存储器系统。

按需调页 Demand Paging

指在需要时才调入相应的页的技术,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

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

更完整的页表项 请求分页中

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

Effective memory-access time 有效访问时间

EAT=(1-p)*memory access time + p*page fault time
Page fault time 包括 page fault overhead, swap page out, swap page in, restart overhead 等
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 page fault 时间是缺页中断服务时间 缺页读入时间和重启时间

写时复制 copy-on-write COW

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**时才拷贝一份该页面进行写。
COW 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.Win linux solaris 都用了 COW

页面置换

寻找一些内存中没有使用的页换出去。内存的过度分配 over-allocation 会导致 page fault 调页后发现所有页都在使用。
使用 dirty/modify 位来减少页传输的开销，只有脏页才需要写回硬盘。

基本页面置换过程：

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

页面置换算法

采用最小页错误率的置换算法。
评估方法：
针对特定的内存引用序列，运行算法，计算出页错误数。引用序列叫做 reference string。
注意两个事实：给定页大小，只需要关心页码，不用管完整地址；紧跟页 p 后面对页 p 的引用不会引起页错误。

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

最简单的页面置换算法。必须置换一页时，选择最旧的。不需要记录时间，只需要 FIFO 队列来管理页即可。15 次缺页

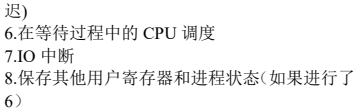
FIFO 会出现可用帧越多，错误数越大的问题，这叫 **Belady's Anomaly**

Optimal Page Replacement OPT 最佳页面置换

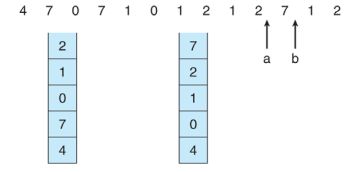
OPT 时所有算法中页错误率最低，且绝对没有 Belady 异常。置换最长时间不会使用的页。或者说选择未来不再使用/在离当前最远位置上出现的页置换。这个使用时长看下一次该页号出现的距离即可。就是向未来看

Least Recently Used LRU 最近最久使用

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



LRU 算法如何获取多长时间没引用？两种方法：
计数器 counter：每一个页表条目都有一个 counter，每次被引用，就把时钟信息复制到 counter。当置换时，置换时间最小的页，最近越使用，clock 越大。
栈实现：维护一个页码栈，栈由双向链表实现。引用页面时将该页面移动到顶部，需要改变 6 个指针。替换时直接替换栈底部就是 LRU 页。



LRU Approximation LRU 近似

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

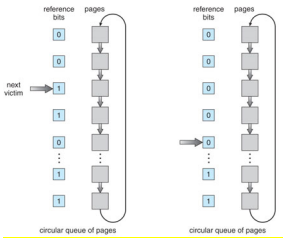
Additional reference bits 附加引用位算法

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

Second chance 二次机会/clock 算法 NRU

基本算法是 FIFO.当一个页获得二次机会时，引用位清零，且将到达时间设为当前时间。因此，获得二次机会的页在所有其他页置换或获得二次机会之前，是不会被置换的。
1.每个 frame 有一个 reference bit
2.当一个 frame 被引用到，它的 reference bit=1。
下次置换先跳过这个 frame，再给它一次机会留在内存中。
3.当新页面被读到 mem 中，它的 reference bit=0。
需要替换 frame 时，使用轮询的方式来查找：
如页面 reference bit=1，那么置为 0，继续查找；
如页面 reference bit=0，那么将该页面置换出去。

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



Enhanced Second chance 改进 clock 增强二次机会

通过将引用位和脏位作为有序对来考虑，可以改进二次机会算法。两个位有四种可能：
(0,0)无引用无修改，置换的最佳页
(0,1)无引用有修改，置换前需要写回脏页
(1,0)有引用无修改，很可能会继续用
(1,1)有引用有修改，很可能会继续用且置换前须要写回脏页
淘汰次序(0,0)-> (0,1)-> (1,0)-> (1,1)
当页面需要被置换时，使用时钟算法，置换(0,0)的页，在进行置换前可能要多次搜索循环队列。改进的点子在于给未引用但是修改了的页更高优先级，降低了 IO 数。
Macintosh 使用

Counting 基于计数的置换算法

为每个页保存一个用于记录引用次数的计数器，具体方案有两种：
Least frequently used LFU：置换计数最小的。但是有问题：一个页可能一开始狂用，但是后来不用了，他的计数可能很大，但是不会被替换。解决方法是定期右移次数寄存器。
Most frequently used MFU：置换计数最大的，因为最小次数的页可能刚调进来，还没来得及用。这两种很没用，实现开销很大，而且还很难近似 OPT。

Page Buffering 页面缓冲

通过被置换页面的缓冲，有机会找回刚被置换的页。
被置换页面的选择和处理：用 FIFO 选择置换页，把被置换的页面放到两个链表之一。即：如果页面无修改，将其归入空闲页链表，否则归入已修改页面链表。
需要调入新页面时，将新页面内容读入空闲页面链表的第一项所指的页面，然后将其删除。

帧分配 allocation of frames

每个进程都需要最小数目的页。两种分配模式：

平均分配算法 Equal allocation

每个如果有 100 个帧 5 个进程，每一个进程获得 20 个帧。

按比例分配 Proportional allocation

根据进程的大小按比例分配。

优先级分配 Priority allocation

同样按比例分配，但是是用优先级进行比例分配。

全局置换 global allocation

允许一个进程从所有帧中选择一个帧进行替换，不管该帧是否已分配给其他进程。

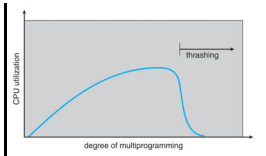
局部置换 local allocation

每个进程只能从自己的分配帧中进行置换选择。固定分配局部置换 可变分配全局置换 可变分配局部置换

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

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

Thrashing

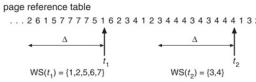


频繁的页调度行为叫做颠簸，会导致：CPU 利用率低、OS 认为多道程序程度需要增加、其他进程进入到系统中
颠簸就等价于一个进程不断换入换出页

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

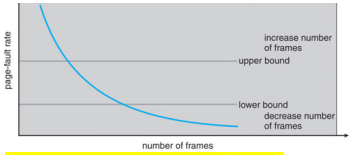
工作集合模型 Working set model

WS 工作集：最近 δ 个 page 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，如果 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=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。如果 D<m 就暂停一个进程。
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。刮起的进程可以在以后重启。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，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，但是控制颠簸不是很灵活，更直接的方法是 PFF。
可以为所期望的页错误设置一个上限和下限，如果页错误率超过上限，那么分配更多的帧，如果低于下限，那么可以从进程中移走帧。



Memory-Mapped Files 内存映射文件

使用虚存技术来讲文件 IO 作为普通文件访问

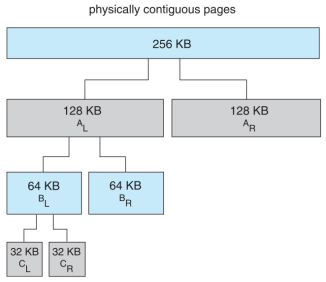
的技术。
开始的文件访问按照普通按需请求调度，会出现页错误。这样，一页大小的部分文件从文件系统中读入物理页，以后的文件访问就可以按照通常的内存访问来处理，这样就可以用内存操作文件，而非 read write 等系统调用，简化了文件访问和使用。多个进城可以允许将同一文件映射到各自的虚存中，达到数据共享的目的。

Allocating Kernel Memory 内核内存分配

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

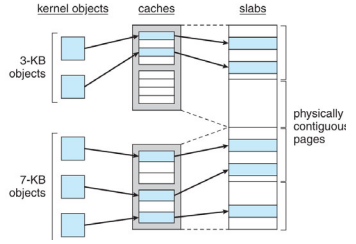
Buddy system

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。优点是可以通过合并快速形成更大的段。缺点是由于调整到下一个 2 的幂容易产生 **internel** frags。



Slab 分配

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



预调页 prepaging

为了减少冷启动时大量 page fault。
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：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
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 互锁指页面必定有时被所在内存中。
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

File attributes

name identifier(**唯一**标识该文件:File metadata)
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 disk。

文件操作

文件是 ADT 抽象数据类型，其操作有：创建 写 读 文件内重定位 删除 截短 truncate **Open(Fi)在硬盘上寻找目录结构并且移动到内存中 Close(Fi)将内存中的目录结构移动到磁盘中**。

Open

每个打开文件都有以下信息：文件指针：跟踪上次读写位置作为当前文件位置指针
每次访问文件前都要 **open** 文件.**create(首次 open)会在 disk 上创建 FCB,之后 open 从 disk 读 FCB**
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

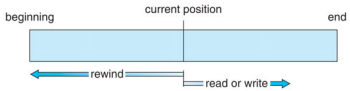
文件内部结构 File Structure

None 字 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

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。

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

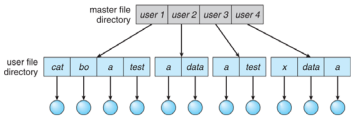
搜索文件 创建文件 删除文件 遍历 list 目录 重命名文件 遍历 traverse 文件系统

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

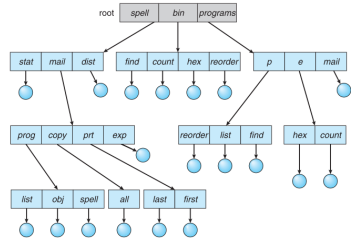
Two Level Directory

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



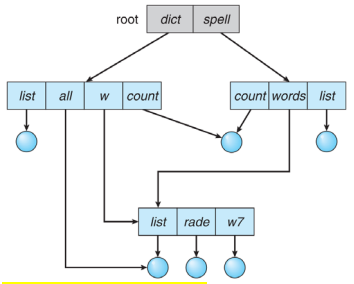
树形目录(avoid name collision)

将二级目录拓展即可。搜索高效 有分组能力。



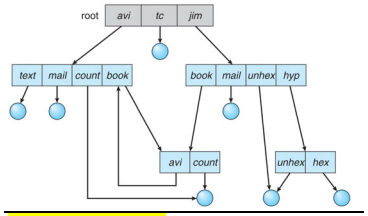
无环图目录 Acyclic graph

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。存在 **dangling pointer**：删除一个文件后指向该文件的其他链接成为 dangling。
对于采用**符号链接**实现共享的系统，删除链接并不影响源文件，如果文件本身被删除，链接也被删除。
删除的另一方法是保留文件知道删除其全部引用，所以为文件引入了计数，删除一次链接或者条目就计数-1，到 0 时完全删除文件，UNIX 的**硬链接采用这种方法**，在 **inode** 中保留一个 **reference count**。通过禁止对目录的多重引用，可以维护无环图结构。



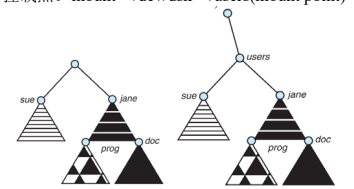
普通图目录 General graph

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

FS 在访问前必挂载。没挂载的 FS 会在 mount point 挂载。左图是未安装的卷，右图的 users 为挂载点。mount /dev/dsk /users(mount point)



Linux disk partion

装置	装置在Linux内的文件名
IDE硬盘机	/dev/hd[a-d]
SCSI/SATA/USB硬盘机	/dev/sd[a-p]
USB快闪碟	/dev/sd[a-p](与SATA不同)
软盘驱动器	/dev/fd[0-1]
打印机	25针: /dev/lp[0-2] USB: /dev/usb/lp[0-15]
鼠标	PS2: /dev/psaux USB: /dev/usb/mouse[0-15]
当前CDROM/DVDROM	/dev/cdrom
当前的鼠标	/dev/mouse

一块磁盘硬盘分区表 64B，一块分区占 16B，主分区最多 4 个(扩展分区也是一个主分区)，扩展分区最多 1 个，扩展分区上可有多个逻辑分区(必须建立在扩展分区上)。主分区范围是 1-4，逻辑分区从 5 开始。

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

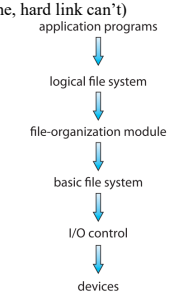
3 种用户类型：
拥有者 owner access 组 group access 其他 public access。UNIX 一个类型有 rwx 3 个权限，所以一个文件需要 3*3=9 位说明文件访问权限

File System Implementation

文件系统：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。
Hard link: a directory entry
Soft link: a file containing path name of another file. (soft link can cross volume, hard link can't)
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

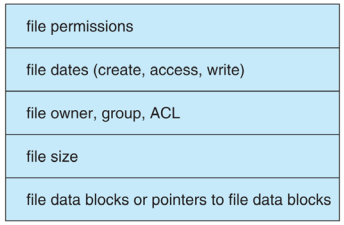
应用程序：发出文件请求者。
逻辑文件系统：管理元数据：文件系统的所有结构数据，而不包括文件的实际顺序；根据给定符号文件名来管理目录结构；**逻辑文件系统通过 FCB 来维护文件结构。**
文件组织模块知道文件逻辑块到物理块的映射，包括空闲空间管理器。做 translation。
基本文件系统：向合适的设备驱动程序发送一般命令就 可对磁盘上的物理块进行读写
IO 控制：由设备驱动程序和中断处理程序组成，实现内存与磁盘之间的信息转移



文件系统实现

On-Disk FS structure

磁盘上，文件系统可能包括：如何启动所存储的 OS，总块数，空闲块数目和位置，目录结构及各个具体文件。
磁盘结构包括：每个卷的 **boot control block** 包括从该卷引导操作系统所需信息；每个卷的 **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(不含 file name)**



In-Memory FS structure

In-memory 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

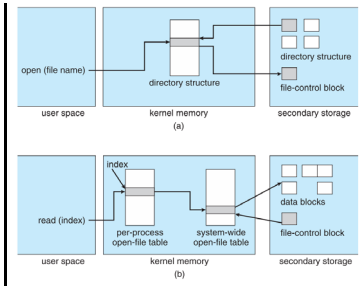


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

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)。

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

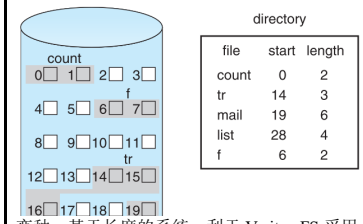
线性列表 linear list：
使用储存文件名和数据块指针的线性表。
哈希表：线性表与哈西结构，哈希表根据文件名得到一个值返回一个指向线性表中元素的指针。

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

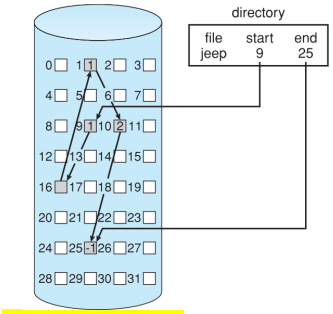
每个文件在磁盘上占有一组连续的块。优点：访问很容易，只需要起始块位置和块长度就可以读取。支持寄存器 random access。但是浪费空间，存在动态存储分配问题。First 和 best 表现差不多，first 时间快很多。存在外碎片问题，此外文件大小不可增长。
逻辑到物理的映射：LA/512 分为两部分商 Q 和余数 R，Block to be accessed = Q + starting address Displacement into block = R。LA 是存取文件逻辑地址，512 是块大小



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。

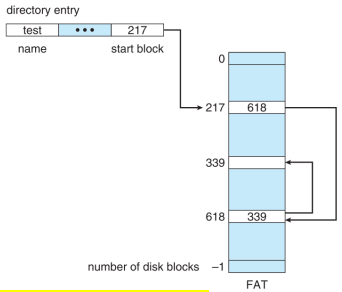
Linked Allocation 链接分配

解决了连续分配的所有问题。每个文件都是磁盘块的链表。访问起来只需要一个起始地址。没有空间管理问题，不会浪费空间，但是不支持 random access。
地址映射：LA/(512-**1**)得到商 Q 和余 R，Block to be accessed is the Qth block in the linked chain of blocks representing the file. Displacement into block = R + **1**。因为每个索引块的末尾节点是用来链接下一个索引块的，不链数据块，所以要 512-1



FAT File allocation table

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**。每个 entry 存下一个 disknumber。Startblock 和 FAT 常被读入 memory

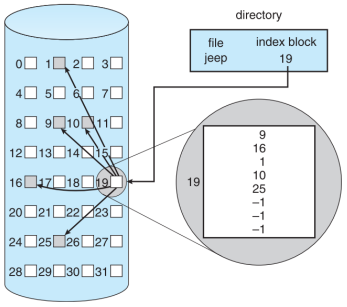


Indexed Allocation 索引分配

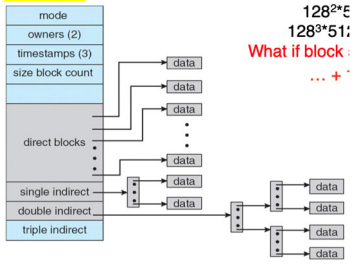
索引分配把所有指针放在一起，通过索引块解决这个问题。每个文件都有索引块，是一个磁盘块地址的数组。当首次写入第 i 块时，先从空闲空间管理器获得一块，再将其地址写到索引块中的第 i 个条目。对于小文件，大部分索引块被浪费。如果索引块太小，可以多层索引、然后互相连接。访问需要索引表，支持 randomaccess，动态访问没有外碎片，但是有索引开销。LA/512 得商 Q 和余数 R，Q = displacement into index table R = displacement into block

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。

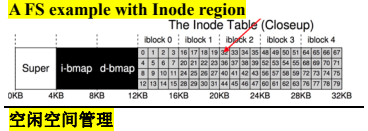
一级索引：一个索引块可以存 4KB/4B=1K 个索引地址，每个索引地址直接引到文件块，所以最大 1K*4KB=4MB。

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

A FS example with Inode region



空闲空间管理

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。

用 linked list：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 效率低因为要遍历。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非磁盘块缓冲到虚存。

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，否则文件系统洒还没更新。

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

Disk bandwidth=传递的总字节数/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，适合 **flash memory**。

SSTF 最短寻道时间优先：处理靠近当前磁头位置的请求，本质和 SJF 一样，有可能请求会饿死，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(不是磁盘的极限)就不继续走，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 **section** 才能进行读写,为每个 **section** 采用特殊的数据结构(如 ECC)。**逻辑格式化**：创建文件系统(建立 FS 根目录,初始化空闲磁盘块数据结构)。

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；后为提升效率将块集中到一起成为 **cluster** Eg. cluster size=1KB, section size=512B,file size=1025B,系统给该文件分配的磁盘空间大小=2* cluster size(按照 cluster 大小分配)

bootstrap 存在 **ROM(firmware)**系统启动顺序：ROM 中代码(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:无容错 1:镜像 2:纠错码 3:专门 1 盘放校验码,**byte**-stripping 适合连续不适合随机读写 4:与 3 类似，**block**-Striping 适合随机读不适合随机写 5:校验码 1 个盘分布式存储(随机写好但小 IO 性能差)6:校验码 2 个盘分布式存储 **1+0 0+1(底层+顶层) 0+1 最少坏 2 个盘就报废**

三级存储 Tertiary storage device

Low cost is the defining characteristic of tertiary storage. tertiary storage is built using **removable** media.examples of removable media are **floppy disks(软盘不是 hard disk),CD-ROMs,tapes**; other types are available

Swap space 三级存储 Tertiary storage device

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。还可创建在 **raw partition 上(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

Polling(device 为了 IO, busy waiting) **Interrupt** CPU 硬件有一条中断请求线 **IRL**，IO 设备触发，需要 IO 时申请中断。设备控制器通过 IRL 发信号而引起中断,CPU 捕获中断并派遣到中断处理程序，中断处理程序通过处理设备清除中断。

DMA direct memory access(disk 主要用这个,直接在 IO 设备和 memory 数据交互)

对需要进行大量 IO 的设备，为避免 programmed I/O(PIO)，将一部分任务给 **DMA**controller，在 DMA 开始传输时，主机向 memory 写入 DMA 命令块。然后 CPU 在写入后继续干别的，DMA 去自己操作内存总线，然后就可以向内存进行传输。DMA 传输完每个 **block** 后发信号给 CPU 而不是每个 byte

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

Device driver

是 uniform device-access interface to I/O subsystem.多个**同一种类的物理设备(i.e 多个 printer)只需要对应 1 个逻辑驱动程序**

用户程序的设备独立性:用户程序不直接使用物理设备名/设备的物理地址，而只能用逻辑设备名；而系统在实际执行时，将逻辑设备名转换为某个具体的物理设备名，实施 I/O 操作

I/O 软件的设备独立性:除了直接与设备打交道的低层软件外，其他部分的软件并不依赖于硬件,提高设备管理软件的设计效率

IO 应用接口

Linux 把 IO 设备看做 special file

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

设备分类:

Sequential(moderm) or random-access(CD-ROM)

Sharable(keyboard) or dedicated(tape)

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

read-write(disk), read-only(CD-ROM), or write only(graphics controller)

Synchronous(tape) or asynchronous(keyboard)

Block device：包括 IDE harddisk，disk drives，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。内存映射文件访问建立在 **block device driver** 之上。内存映射接口不提供 read,write 操作，而提供通过内存中的字节数组来访问磁盘存储

Character device：keyboards, mice, serial ports, terminal 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=blocking IO+non-blocking I/O

blocking IO:进程挂起直到 IO 完成，容易使用和理解，但是不能满足某些需求

non-blocking IO：IO 调用**立刻返回**尽可能多的数据(**立刻返回**读/写的字节数)。用多线程实现。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。

非阻塞和异步系统调用区别：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 所调用的传输一定要完整执行完,其具体执行可以是将来某个特定时间

Buffer 作用:维持 copy semantics 要求,解决 CPU 与 I/O 之间速度不匹配的矛盾

Spooling:用户需要打印一个文件时，该文件先送到 spool 目录，再由负责打印的后台进程把这些数据送入打印机(Printing 打印机虽然是独享设备，通过 SPOOLing，可以将它改造为可供多个用户共享的设备)

Device reservation provides exclusive access to a device,也可能造成死锁

Stream:是用户级进程和设备之间的 full-duplex communication channel.(interface head with user process, drive end with device).Each module contains a **read queue** and a **write queue**

VFS

VFS 并不是一种实际的文件系统。ext2 等物理文件系统是存在于外存空间的，而 VFS 仅存在于内存。文件系统的源代码可以在 linux/fs 中找到.

超级块对象 superblock :存储已安装文件系统的信息，通常对应磁盘文件系统的文件系统超级块或控制块。

索引节点 inode：存储某个文件的信息。通常对应磁盘文件系统的文件控制块

目录项 dentry：主要是描述一个目录项，是路径的组成部分。

文件对象 file object:存储一个打开文件和一个进程的关联信息。只要文件一直打开，这个对象就一直存在与内存

物理文件系统的 inode 在外存中并且是长期存在的， VFS 的 inode 对象在内存中，它仅在需要时才建立，不再需要时撤消.物理文件系统的 inode 是静态的，而 VFS 的 inode 是一种动态结构 VFS 的 inode 与某个文件的对应关系是通过设备号 i_dev 与 inode 号 i_ino 建立的，它们唯一地指定了某个设备上的一个文件或目录。

VFS 的 inode 是物理设备上的文件或目录的 inode 在内存中的统一映像。这些特有信息是各种文件系统的 inode 在内存中的映像。如 EXT2 的 ext2_inode_info 结构。

i_lock 表示该 inode 被锁定，禁止对它的访问。

i_flock 表示该 inode 对应的文件被锁定。i_flock 是个指向 file_lock 结构链表的指针， 该链表指出了一系列被锁定的文件。

VFS 的 inode 组成一个双向链表，全局变量 first_inode 指向链表的表头。在这个链表中，空闲的 inode 总是从表头加入，而占用的 inode 总是从表尾加入。

Inode operations

Create:只适用于目录 inode，当 VFS 需要在“inode”里面创建一个文件（文件名在 dentry 里面给出）的时候被调用。VFS 必须已经检查过文件名在这个目录里面不存在。

lookup:用于检查一个文件（文件名在 dentry 里面给出）是否在一个 inode 目录里面。

Link:在 inode 所给出的目录里面创建一个从第一个参数 dentry 文件到第三个参数 dentry 文件的硬链接（hard link）。

Unlink:从 inode 目录里面删除 dentry 所代表的文件。

symlink:用于在 inode 目录里面创建软链接(soft link)。

Mkdir:用于在 inode 目录里面创建子目录。

Rmdir:用于在 inode 目录里面删除子目录。

Mknod:用于在 inode 目录里面创建设备文件。

目录项对象 dentry object

每个文件除了有一个索引节点 inode 数据结构外，还有一个目录项 dentry 数据结构。 每个 dentry 代表路径中的一个特定部分。如：/、bin、vi 都属于目录项对象。目录项也可包括安装点，如：/mnt/cdrom/foo，/、mnt、cdrom、foo 都属于目录项对象。inode 结构代表的是物理意义上的文件，记录的是物理上的属性，对于一个具体的文件系统，其 inode 结构在磁盘上就有对应的映像.一个索引节点对象可能对应多个目录项对象.目录项对象作用是帮助实现文件的快速定位，还起到缓冲作用

VFS 的 dentry cache 与 inode cache

为了加速对经常使用的目录的访问， VFS 文件系统维护着一个目录项的缓存。为了加快文件的查找速度 VFS 文件系统维护一个 inode 节点的缓存以加速对所有装配的文件系统的访问。用 hash 表将缓存对象组织起来。

File 对象

文件对象 file 表示进程已打开的文件，只有当文件被打开时才在内存中建立 file 对象的内容。该对象由相应的 open()系统调用创建，由 close()系统调用销毁。

file_operations

llseek：用于移动文件内部偏移量。

read：读文件。 aio_read：异步读，被 io_submit 和其他的异步 IO 函数调用。

write：写文件。

aio_write：异步写，被 io_submit 和其他的异步 IO 函数调用。

readdir：当 VFS 需要读目录内容的时候调用这个函数。

poll：当一个进程想检查一个文件是否有内容可读写的时候，VFS 调用这个函数；一般来说，调用这个函数之后进程进入睡眠，直到文件中有内容读写就绪时被唤醒。详情请参考 select 和 poll 系统调用。

ioctl:被系统调用 ioctl 调用。

unlocked_ioctl：被系统调用 ioctl 调用；不需要 BKL（内核锁）的文件系统应该使用这个函数，而不是上面那个 ioctl。

compat_ioctl：被系统调用 ioctl 调用；当在 64 位内核上使用 32 位系统调用的时候使用这个 ioctl 函数。

mmap：被系统调用 mmap 调用。

open：通过创建一个新的文件对象而打开一个文件，并把它链接到相应的索引节点对象。

flush：被系统调用 close 调用，把一个文件内容写回磁盘。

release：当对一个打开文件的最后引用关闭的时候，VFS 调用这个函数释放文件。

fsync：被系统调用 fsync 调用。

fasync：当对一个文件启用异步读写（非阻塞读写）的时候，被系统调用 fcntl 调用。

lock：fcntl 系统调用使用命令 F_GETLK，F_SETLK 和 F_SETLKW 的时候，调用这个函数。

内核文件 bzImage 在/usr/src/linux/arch/i386/boot。

make modules_install 安装模块

make install 使用命令 make install 将 bzImage 和 System.map 拷贝到/boot 目录下。这样，Linux 在系统引导后从/boot 目录下读取内核映像到内存中

添加系统调用号 在系统调用表中修改或添加相应项

添加系统调用：

system_call()函数实现了系统调用中断处理程序：

1.它首先把系统调用号和该异常处理程序用到的所有 CPU 寄存 器保存到相应的栈中，SAVE_ALL

2.把当前进程 task_struct （thread_info）结构的地址存放在 ebx 中

3.对用户态进程传递来的系统调用号进行有效性检查。若调 用号大于或等于 NR_syscalls，

系统调用处理程序终止。（sys_call_table）
4.若系统调用号无效，函数就把-ENOSYS 值存放在栈中 eax 寄存器所在的单元，再跳到 ret_from_sys_call()
5.根据 eax 中所包含的系统调用号调用对应的特定服务例程

添加文件系统：

文件类型：
普通文件（文件名不超过 255）
目录文件
字符设备文件和块设备文件：
fd0 (for floppy drive 0)
hda (for harddisk a)
lp0 (for line printer 0)
tty(for teletype terminal)
管道(FIFO)文件 链接文件 socket 文件
文件系统分三大类：基于磁盘的文件系统，如 ext2/ext3/ext4、VFAT、NTFS(win10)等。网络文件系统，如 NFS 等。特殊文件系统，如 proc 文件系统、devfs、sysfs（/sys）等。
Linux 以 ext2/ext3 做为基本的文件系统，所以它的虚拟文件系统 VFS 中也设置了 inode 结构。物理文件系统的 inode 在外存中并且是长期存在的，VFS 的 inode 对象在内存中，它仅在需要时才建立，不再需要时撤消。物理文件系统的 inode 是静态的，而 VFS 的 inode 是一种动态结构。
dd：用指定大小的块拷贝一个文件，并在拷贝的同时进行指定的转换
命令语法：dd [选项]
if = 输入文件（或设备名称）
of = 输出文件（或设备名称）
bs = bytes 同时设置读/写缓冲区的字节数（等于设置 ibs 和 obs）
count=blocks 只拷贝输入的 blocks 块
conv = ucase 把字母由小写转换为大写
conv = lcase 把字母由大写转换为小写。
例：dd if=/dev/zero of=myfs bs=1M count=1
/dev/zero：零设备“0”
/dev/loop：loopback device（回环设备、或虚拟设备）指是用文件来模拟块设

习题

Banker algorithm

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 2 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: 3+3+(3+1)=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: 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：文件系统有一个 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 4KB，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：由于块地址为 4B(4*8=32bit)，所以能表示的最多块数为 2^32=4G，而每个盘块中存放文件大小为 4096-4=4092B。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 4B，所以最多的链接索引块数为 2^32=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=2^32，算出 n=2^22，链接索引分配可管理的最大文件为：4M*1023*4KB=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：1.**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 1 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。

inverted page table

Assume we have a simple, demand paging environment, with no segmentation. Processes P1 and P2 both have logical memory addresses in the range 0 . . . 99, inclusive. **page size is 5**. hash table portion of the structure uses a hash function which calculates **index** by adding numerical portion of **process identifier and logical page number**. hash table and IPT shown are below.

Hashtable

Index	Process ID / logical page	Frame ID
…	…	…
3		
4	P2 2	1
5	P1 3	4
…	…	…
18	P1 17	2
19	P1 18	3

Inverted Page Table

Index	Logical address (minimized for sim
0	
1	P2:RW
2	P1:RW
3	P1:R
4	P1:RW

(1) Calculate **physical address** for the following logical addresses. Assume first page of a process is page number 0, and that a mechanism exists to find a free frame in memory.

P1:17 Logical page=17/5 = 3,offset = 2. hash table index = 3 + 1(pid)=4,P1 本来 hash table 里对应 frameID=1，但是 **IPT 里面 frameID=1 映射到的是进程 P2 而非 P1**，所以 P1 在 hash table 里面映射后移 1,即 hash index=4+1==> **follow chain to entry 5**. From hash table, physic frame=4. Address = (frame)*(frame size) + offset = 4*5 + 2 = 22

P1:92 Logical page=92/5=18,offset = 2. Hash table index = 19. From lookup logical frame = 3. Address = 17

(2) When process P2 attempts to read logical address 97, what happens?

This causes page fault. Frame 0 is allocated, and the entry in IPT is updated with logical address information for P2 frame 19. Next the hash table entry at index 21 (19 + 2) is mapped from P2:19 to frame 0. After this processing is complete, the P2 resumes execution as before.

进程同步与死锁

4 processes share a buffer space named B: R1 reads a number from the keyboard and saves it into B, so that the saved number is only consumed by W1, which prints the number to the screen. R2 reads a character from a mouse and also saves character into B, so that only W2 can print character to a printer. Please write the synchronization code for the four processes so that no race condition may arise among them.

Semophere S=1(B 为空，初始为 1), S1=0(R1 做完，W1 可以做), S2=0(R2 做完，W2 可以做)

计数信号量 S

若干进程对 S 进行 28 次 P 操作和 18 次 V 操作后，信号量 S 的值为 0(此时无资源可用)，然后又对信号量 S 进行了 3 次 V 操作,此时有 0 个进程等待在信号量 S 的队列中

FAT 计算

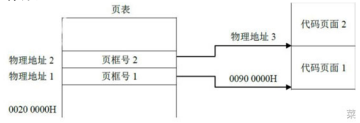
FAT 的 entrySize=2B，则 FAT 最大 size=2^16*2B(2B=16bit->2^16)

Waiting time/turnaround time cal.

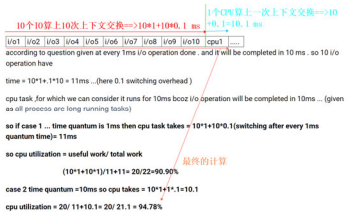
Turn around time=waiting time+burst time

Address computing in page table(VPN→PPN)

PA,VA 均为 32bit,pageSize=4KB,**pageTableEntrySize=4B**.代码段起始 VA=0x8000，长度=8KB，被装载到从 PA=0x900000 开始的**连续**主存空间。页表从主存 0x200000 开始的物理地址处连续存放



如图所示,代码页面 2 的起始 PA=0x900000+8K。代码页面 1 对应 VA1=0x8000->VPN1=8→物理地址 1=0x200000+**VPN1*pageTableEntrySize**=0x200020,同时 PA1=0x900000->PPN1=0x900




```
pid = fork();
if (pid == 0)
{
    /* childprocess */
    fork();
    thread_create(. . .);
}
    fork();
```

a. How many unique processes are created? __6___（包括第一次运行该程序的进程）

第一次运行该程序的进程 P0 fork() 创建一个 sub process P1，没有创建 thread

P1 在 pid==0 的代码块里面创建 1 个 sub process P2，P1P2 一共创建 2 个 thread

P0P1P2 在外面的 fork 各自创建一个 sub process 分别是 P3,P4,P5

b. How many unique threads are created? __2___ (没有主线程)