BCIT CST 操作系统笔记 刘航

1. 操作系统的五个层级(由低到高)
   Digital Level (包括 AND, OR, NOT 逻辑电路)
   Micro architecture Level (用于控制 CPU 的机器码在这一层)
   Instruction Set Architecture (ISA) Level (定义了电脑支持的基本操作指令)
   Operating System (OS) Level (内存管理、进程管理、资源保护等)
   Assembly Language Level (汇编语言写成的代码)
   High Programming Language Level (高级语言写的代码)

2. CPU 的组成
   Control Unit (控制单元)
   ALU (算法逻辑单元)
   Resisters (寄存器)

3. 操作系统的定义
   a. 是个软件
   b. 一个扩展过的机器
   c. 一个可以提供抽象层的资源管理器

4. 内核模式和用户模式 (略)

5. 作为资源管理器的操作系统
   根据时间或空间管理资源

6. 操作系统历史
   a. 第 0 代，机械式计算机
   b. 第一代，真空管计算机(ENIAC)，1945 年由冯诺依曼提出构想
   c. 第二代，晶体管计算机(IBM7094)，可靠可生产，人员职能划分明确，OS 及批处理
   d. 第三代，集成电路，并发执行与任务池
   e. 第四代，大规模集成电路(Intel x86), DOS, Windows
   f. 第五代，移动设备

7. ALU 算法表，应该不会考

Logic gates

ALU (list of operations)

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B − A
1	1	0	1	1	0	B − 1
1	1	1	0	1	1	−A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	−1

8. 控制单元的作用
   FDE: Fetch, Decode, Execute

9. Hardwired 和 Micro program
   区别仅在于是硬件实现还是代码实现，也是 RISC 和 CISC 的区别

10. 提升 CPU 速度的方案
    a. 流水线 pipelining
    b. 指令集的并发
    c. 多线程
    d. 多核
    e. 多处理器
    f. 多电脑

11. 寄存器种类
    PC: 存放下一个要执行的寄存器
    Stack Pointer: 存放当前栈顶的地址
    PSW: 存放状态位，这些状态位一般是比，较指令的结果、内核或用户模式，以及其他值组成。在系统 call 和 io 里扮演重要角色。

12. 计算缓存的 set 格式
    缓存大小 / line 大小 / 每个 set 的 line 个数

13. 缓存的方式
    a. 直接映射
    b. 关联映射
    c. 组关联映射

14. 缓存系统的难点
    何时、何处进行增删改查
    替代算法：LRU/FIFO/LFU/Random/Clock

15. IO 设备
    包括控制器和设备本身
    读写方法：busy waiting, using interrupt, use of special hardware

16. 总线类型
    SCSI(Small Computer System Interface), USB, DMI, PCI, PCI-E

17. 进程
    进程就是运行着的程序，有关联的地址空间，信息保存在系统的进程表里，内存分为三段（text，data，stack）

18. 文件
    i-number 是 i-node 表的索引

19. 文件保护
    Unix 的机制是 9 个 bits 代表 rwx-rwx-rwx，用户，用户所在的组，其他人

20. 系统调用
    Providing abstractions to user programs and managing the computer's resources.

21. 区分软硬链接

22. 单位表

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

23. 多进程系统
    a. CPU 在进程之间快速切换
    b. CPU 执行每个进程数十到数百毫秒
    c. 严格来将，CPU 在同一时刻只运行一个进程

24. 进程的创建
    a. 系统初始化，部分是服务，部分是界面，后台进程成为 daemons
    b. 运行中的进程创建别的进程
    c. 用户执行
    d. 批处理创建

25. 进程的退出
    a. 正常退出
    b. 发生错误
    c. 致命错误(因 bug 导致，比如野指针)
    d. 被其他进程杀死

26. Unix 进程体系
    进程可以创建子进程，子进程还能创建子进程，他们被归为一组。用户的信号可以被这个进程组的所有进程捕获，但具体怎么反应得看进程自身

27. 进程三状态
    Running, Ready, Blocked

28. CPU 利用率的计算

    利用率 = 1 – (IO 满载率)^(进程个数)

    浪费率 = (IO 满载率)^(进程个数)

29. 线程的定义

    A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A light weight process providing a way to improve application performance through parallelism.

30. 线程的作用

    a. 减少阻塞的影响
    b. 创建的成本比创建进程更小
    c. 加速程序
    d. 充分利用 CPU 的核数

31. 线程可以访问进程内任何资源，虽然有自己的 PC、Stack，但线程之间无保护，原因是

    a. 不可能
    b. 没必要

32. 线程状态有四种，比进程多一种 terminated

33. 多线程

    CPU 在多个线程之间快速切换

34. PThread 库实现了 POSIX 的 60 个函数

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

35. 用户空间线程

    优点:

    a. 无需 trap，无需切换，无需刷新内存，让线程调度更快
    b. 可以有自定义的调度算法
    c. 扩展性更好，可以创建大量的线程

    缺点：

    a. 如果一个用户态线程被 blcoked，其他线程都会被卡死
    b. 只有正在运行的线程主动让出执行权，其他线程才有机会被执行
    c. 程序员在线程经常被 blcok 的时候要精确控制线程

36. 内核空间线程

优点：

a. 无需新的、无阻塞的系统调用
b. 当进程的一个线程阻塞(如缺页)时，内核会检查和切到当前进程的其他线程

缺点：

创建、销毁线程的系统调用消耗太大

37. 用户空间线程和内核空间线程的不同

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

38. 在内核中的进程表和线程表是分开放的

39. 混合的线程实现（multiplex 复用）

开发者可以自行决定要创建多少个内核级线程，以及每个内核线程上有多少个用户级线程来复用这些内核线程

40. 弹出式线程

每次有新消息到达，就立刻启动一个新线程来处理。从消息到达直到新线程被创建的延迟很短。新线程没有历史的寄存器、堆栈需要被恢复，所以创建起来很快。

41. 条件竞争(Race Conditions)

两个进程读写一个共享的数据时，读写结果取决于其访问的先后次序，就是条件竞争

42. 临界区(Critical Regions，又叫做 Critical Section)

互斥(Mutual Exclusion)：当一个进程正在使用一个数据或文件时，另一个进程被禁止读写相同的数据或文件。被共享的这个数据称为临界区

43. 临界区的要求

a. 多个进程里，只有一个进程可以进入临界区
b. 临界区与 CPU 的速度和个数无关

c. 运行在临界区之外的进程不能阻塞其他进程
d. 进程不能永远等待进入临界区

44. 自旋锁
持续检测直到一个变量出现的行为称为忙等待(busy waiting)

45. 互斥的实现
a. 进程进入临界区时禁用所有中断，并在离开临界区时恢复中断
b. 只有运行了禁止中断指令的那个 CPU 会受影响，其他 CPU 不受影响

46. Peterson 的互斥实现

```
#define FALSE  0
#define TRUE   1
#define N      2

int turn;
int interested[N];

void enter_region(int process);
{
    int other;

    other = 1 – process;
    interested[process] = TRUE
    turn = process;
    while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

interested = [FALSE, FALSE]
process = 0
other = 1
interested[0] = TRUE
trun = 0
0 0
interested[1] = FALSE
interested[0] = FALSE

```
#define FALSE  0
#define TRUE   1
#define N      2

int turn;
int interested[N];

void enter_region(int process);
{
    int other;

    other = 1 – process;
    interested[process] = TRUE
    turn = process;
    while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

interested = [TRUE , FALSE]
process = 1
other = 0
interested[1] = TRUE
trun = 1
1 1
interested[0] = TRUE
interested[1] = FALSE

47. TSL(Test Set Lock)的互斥实现
一种由硬件支持的方式，有两种实现方式：TSL 和 XCHG，代码不同，目的一致。缺点也一致：耗费 CPU 做循环。

```
enter_region:
    TSL REGISTER,LOCK       | copy lock to register and set lock to 1
    CMP REGISTER,#0         | was lock zero?
    JNE enter_region        | if it was nonzero, lock was set, so loop
    RET                     | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0            | store a 0 in lock
    RET                     | return to caller
```

```
enter_region:
    MOVE REGISTER,#1        | put a 1 in the register
    XCHG REGISTER,LOCK      | swap the contents of the register and lock variable
    CMP REGISTER,#0         | was lock zero?
    JNE enter_region        | if it was non zero, lock was set, so loop
    RET                     | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0            | store a 0 in lock
    RET                     | return to caller
```

48. Sleep 和 Wakeup 的问题

当消费者看到可供消费的项目为 0 时，正准备休眠，此时 CPU 切换到生产者，生产者生产了一个可供消费的项目，并立即尝试唤醒消费者，但是消费者此时没有休眠(正准备进入休眠)，这时候的 Wakeup 就失效了，双双进入休眠状态。

49. 信号量(Semaphores)

为解决 48 里面的问题，操作系统引入信号量，生产者生产后增加信号(当信号量满了就暂停生产，直到信号量不满)，消费者消费后减少信号(当信号量空了就暂停消费，直到信号量为非空)

信号量有两种。一种是只有 1 个信号的，称为 binary(因为要么有信号，要么没信号)，可以直接当 Mutex（互斥）用。另一种可以放多个信号。

生产和消费的过程，最内层用 Mutex 包裹，外层用信号量包裹。

50. 当 TSL 或 XCHG 可用时，在用户空间也可实现 Mutex

通常，0 表示无锁，其他数字均表示有锁。当检测到有锁时，主动让出 CPU 去执行其他线程，回来后再检测是否有锁，直到无锁才 return

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

51. pthread 里的 mutex

pthread_mutex_init/destroy/lock/trylock/unlock

52. pthread 提供另一种同步方式 condition variables(条件变量)

a. 条件变量允许在某个条件没满足时，将进程阻塞
b. Mutex 和条件变量通常成对使用(等待条件变量时自动离开和进入 mutex)
c. 与信号量 semaphores 不同，条件变量不额外占内存
d. 如果给条件变量发信号，但是没有线程在等待这个变量，那么信号丢失
e. 相关函数如下

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

53. 管程 Monitor
    a. 是一个由过程、变量和数据结构组成的一个集合
    b. 不能在管程之外的过程里直接访问管程内的数据结构，只能调用管程中的过程
    c. 在任意时刻，管程中只能有一个活跃进程，由编译器保证
    d. 是一个概念，大多数语言没有管程，Java 有

54. 消息传递 Message Passing
    信号量太低级(原始/底层)，实现管程的语言又很少，所以分布式系统需要消息传递(其实就是 TCP 里的 send 和 receive)，可以用开缓存或不开缓存的方式解决生产和消费的问题，因为缓存满了就自动阻塞。一个被广泛使用的消息传递系统是 MPI(Message Passing Interface)

55. 屏障(Barriers)
    一种用于处理一组带有不同阶段(phase)进程的同步机制(Synchronization Mechanism)，确保所有进程准备好进入下一个阶段后才允许它们通过屏障。

56. 避免锁(Avoid Locking 读-复制-更新)
    a. 最快的锁就是没有锁，常规情况下难以保证写的同时，读的数据不会出错。
    b. 但是在某些情况下是可以做到的，比如更新一棵树。增加节点时，更新好新增的节点后，再把这个节点的父节点指过来。删除时正好相反，先把父节点的指向转移走(可能是指向被要删除节点的子节点)，等没有人读这个要被删除的节点了，再删除。

57. 进程行为
    a. 计算密集型 Compute-Bound，又称作 CPU 密集型 CPU-Bound，有较长的 CPU 突发(burst)和较短的 IO 等待
    b. IO 密集型 IO-bound，有较短的 CPU 突发(burst)和较长的 IO 等待

58. 调度的类型
    非抢占式调度(nonpreemptive)：运行一个程序直到阻塞或主动放弃 CPU
    抢占式调度(preemptive)：挑选进程让其最多运行到一个最大时间，之后调度程序将其暂停并切换到另一个进程

59. 调度的算法分类
    a. 批处理(batch)
    b. 交互式(interactive)
    c. 实时(real time)

(翻到下页)

60. 调度算法的目标

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

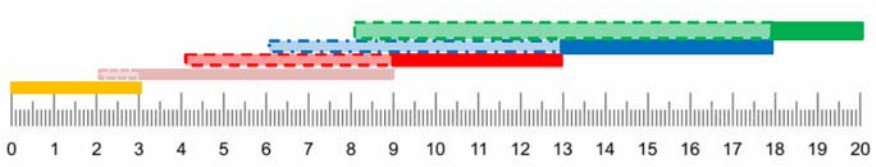
61. Turnaround Time

从提交任务到任务被完成的总时间：等待时间+处理时间

62. 批处理系统中的调度

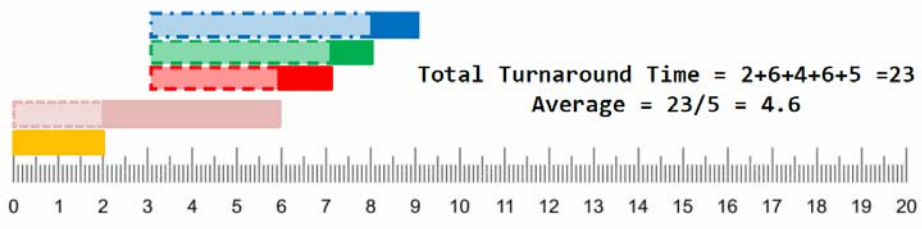
a. 先来先服务(First-come, First-serve) 非抢占，先提交的任务先执行

Process	Arrival Time	Processing Time	Delay	Turnaround Time
1	0	3	0	3
2	2	6	1	7
3	4	4	5	9
4	6	5	7	12
5	8	2	10	12



b. 最短作业优先(Shortest Job First) 非抢占，一个任务执行完成后看剩下的哪个最短

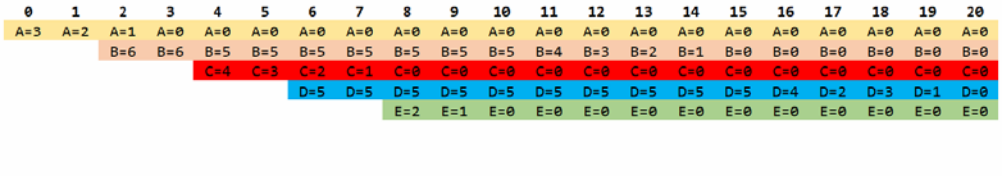
Process	Arrival Time	Processing Time	Delay	Turnaround Time
A	0	2	0	2
B	0	4	2	6
C	3	1	3	4
D	3	1	5	6
E	3	1	4	5



c. 最短剩余时间优先(Shortest Remaining Time Next)，抢占，每个周期都检测最短剩余

Example:

Process	Arrival Time	Processing Time	Delay	Turnaround Time
A	0	3	0	3
B	2	6	7	13
C	4	4	0	4
D	6	5	9	14
E	8	2	0	2



Delay B = (3-2) + (10 - 4) = 7
Delay D = 15 - 6 = 9

63. 交互式系统的调度

a. 轮转调度(Round-Robin Scheduling) 抢占式

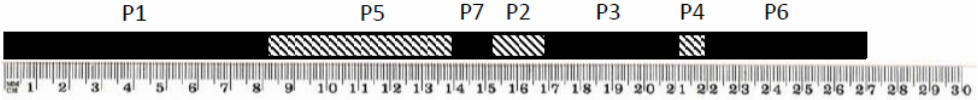
每个进程被分到等额的时间来运行，时间到了没有运行完也立刻切走，如果提前结束，则立刻切走。时间片(Time Quantum)设置为 20-50ms 是比较适合的，太短了切换太多，有浪费，太长了会导致后启动的进程等待时间太久

b. 优先级调度(Priority Scheduling) 抢占式/非抢占式

每个进程被赋予一个优先级，优先级高的可运行进程先运行（优先被调度，而不是得到更多时间）。但是为了防止高优先级的一直运行，低优先级的一直得不到运行，有时需要动态调整优先级。一个简单的算法是将优先级设置为 1/f，f 是进程在上一个分配给他的时间片上所用来计算的时间，而不是 IO 等待时间。例如一个进程在 50ms 的时间片上只用了 1ms 就被 IO 阻塞，那么他的 f 是 1/50，则优先级为 50.

1 - Highest priority
6 - Lowest priority

Process	AT	BT	Priority	Delay	Turnaround Time
P1	0	8	3	0	8
P2	1	2	4	14	16
P3	3	4	4	14	18
P4	4	1	5	17	18
P5	5	6	2	3	9
P6	6	5	6	16	21
P7	10	1	1	4	5



Average Turnaround time = (8+16+18+18+9+21+5)/7 = 13.57
Average delay time = (0+14+14+17+3+16+4)/7 = 9.7

c. 彩票调度(Lottery Scheduling)

持有彩票越多的进程，在下一轮抽奖时，得到运行的机会越多。例如，一个 CPU

每秒抽奖 50 次，这样被抽中的进程可以被运行 20ms。如果本次抽奖共有 100 张彩票，而进程 A 拥有其中 20 张，那么进程 A 在一个连续的时间里，大概能得到 20%的运行时间

d. 公平分享调度(Fair-Sharing Scheduling)
   与其他调度机制不同，这个方式是按照用户来分配时间的。如果有两个用户，甲有 4 个进程 ABCD，乙有 1 个进程 E
   1) 如果用 Round-Robin，则调度顺序为：A E B E C E D E A E B E C E D E
   2) 如果给甲的时间是乙的两倍，则顺序为：A B E C D E A B E C D E

64. 实时系统的调度
   a. 硬实时：所有的实现限制必须被完全满足
   b. 软实时：不希望偶尔错过截止时间，但可以容忍
   c. 实时系统中的时间可以分为周期事件(Periodic，规则发生)和非周期(APeriodic，无法预测地发生)

65. 实时系统的周期事件
   如果有 m 个事件，其中事件 i 以周期 Pi 发生，并且需要 Ci 秒来处理，那么必须满足下面的公式，否则系统无法调度。

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

   例一、 三个事件分别以 100，200 和 500 的周期发生，每个事件需要的 CPU 时间是 50，30 和 100，则系统是否可调度？
   50/100 + 30/200 + 100/500 = 0.5+0.15+0.2 < 1 可调度

   例二、 条件同例一，但加入一个周期为 400 的事件，要消耗 CPU 的时间是 x，那么 x 在什么条件下，这个系统依然是可以调度的？
   50/100 + 30/200 + 100/500 + x/400 = 0.5+0.15+0.2 + x/400 < 1
   x < 60

66. 线程调度
   a. 用户级线程：最大可以运行任意长时间，直到内核切到其他进程。同进程内的多个用户线程在进程时间片内也可以互相切，直到内核切换到其他进程。实际上，用户及现成往往采用轮询和优先级算法来调度。
   b. 内核级线程：如果系统内核任意选取一个线程，而不在意它属于哪个进程，那么在一个时间片内，从进程 A 的线程切到进程 B 的线程也是有可能的。但因为切换进程代价更待，所以内核往往先在进程内切线程，直到进程时间片用完。

期中考试之后的内容

67. 存储管理器是操作系统中用于管理“分层存储体系”(Memory Hierarchy)的部分
    a. 有效管理内存：记录哪些被使用，哪些是空闲的
    b. 进程需要时分配(Allocate)内存
    c. 进程结束后释放(Deallocate)内存

68. 无存储器抽象(Without Memory Abstraction)
    a. 早期电脑没有存储器抽象，程序可以直接访问物理内存
    b. 可以同时运行多个程序
       例如 IBM 360，系统把内存分成 2Kb 的区块，每个区块都对应有一个 4-bits 的 protection key(多个 block 的 key 值可以相同，但要存到不同寄存器里)，这样可以最多同时加载 2^4 = 16 个程序。如果内存有 1MB 内存，那么可以用 1MB/ 2KB = 512 个 CPU 寄存器，512 * 0.5 = 256B，即可管理这 1MB 内存

69. 重定位(reallocation)问题和解决
    a. 问题：在无存储器抽象的系统中，连续加载多个程序，会导致位于高地址的程序执行出现问题，因为这些程序的很多指令是基于绝对地址的
    b. 解决：加载程序时，对代码内的绝对地址进行重定位，让这些地址加上一个常数。但这不是一个通用的解决办法，而且会减慢加载程序的速度

70. 地址空间
    要解决上述问题，就要处理两个问题：保护(Protection)和重定位(Relocation)
    地址空间:
    a. 是一个进程用于寻址内存的地址集合 (An address space is the set of addresses that a process can use to address memory.)
    b. 每个进程的地址空间是独立的(但在特殊情况下空间可能重合，用于共享内存)

71. 基址寄存器和界限寄存器(Base and Limit Register)
    原理：系统把程序加载到内存中的某个空间，这个空间的起始地址就被放到 Base Register，程序大小被放到 Limit Register。程序的指令被执行，指令里的内存地址被送到内存之前，会被 CPU 自动加上基址，来实现重定位。
    缺点：要对每次寻址进行加法计算和比较，耗费性能

72. 内存重载(Memory Overloading)
    操作系统要运行多个程序，占用内存之和往往大于可用内存，可用如下两个方案解决：
    a. 交换技术：完整的把程序加载到内存里，运行一段时间后，放到磁盘。不运行的内存都放到磁盘里，因此不占用多余内存
    b. 虚拟内存：即使只加载了一部分的程序，也允许其运行起来

73. 内存交换(Swapping)
    会导致多个内存空洞，解决方法有两种
    a. 用位图来管理空闲内存
       1) 内存划分成单位(比如几 k)，1 代表被占用，0 代表空闲

2) 单位划分小，则位图占用空间大；反之位图占用小，但可能造成浪费(因为程序填不满分配的内存)
3) 要加载大小为 k 个单位的程序到内存，需要从前到后搜索整个位图，很慢

b. 用链表来管理空闲内存
1) 链表的节点分两种，第一种记录了被占用的基址和大小，第二种记录空洞基址和大小
2) 当有程序退出，则记录此程序占用的链表节点自动向左右合并

74. 内存管理算法
a. 首次适配(First Fit)：存储管理器沿着链表搜索，直到找到第一个足够大的区域，有必要的话把这个空闲区域拆成两部分，一部分给程序，一部分生成新的空闲节点
b. 下次适配(Next Fit)：与首次适配相同，但是每次找到合适区域时记录位置，方便下次的搜索，而不是每次都从头开始，比首次适配略慢
c. 最佳适配(Best Fit)：搜索整个链表，找到一个能容纳进程的最小区域，速度很慢，产生的碎片还更多
d. 最差适配(Worst Fit)：搜索整个链表，找到一个最大区域来容纳进程
e. 快速适配(Quick Fit)：为常用大小的空闲区维护单独链表，比如 8k/12k/20k/21k 等。找制定大小的空闲空间很快，但是进程退出时合并费时

75. 虚拟内存(Virtual Memory)
动机：有些程序太大，无法在内存放下(Fit)，且 Swapping 太慢。1960 年代把程序分成若干片段(overlay)，加载一个 overlay 执行完就用下一个 overlay 覆盖第一个(或者占用更高地址)。分片由程序员来做，耗时、无聊且容易出错。
解决：虚拟内存是分片的提升版
a. 每个程序有自己的地址空间，被分割为块(chunks)，称为页(page)
b. 内存里与页对应的单元称为页帧(page frame)
c. 页和页帧一般一样大
d. 页由连续的地址组成
e. 页被映射到内存里，但不是所有的页都被装入了才能运行程序
f. MMU 来实现虚拟地址和物理地址的转换
g. 页表(page table)维护虚拟地址和物理地址的映射关系
h. 没有虚拟内存的电脑上，虚拟内存地址就是物理内存地址

76. 分页(Paging)
a. CPU 把虚拟地址发到 MMU，MMU 把对应的物理地址发到内存
b. 现在的 MMU 在 CPU 里，以前的 MMU 是个独立部件
c. 虚拟地址的计算：Y = 新基址 A – 老基址 a + 老偏移 x

77. 页的个数 / 页帧的个数如何计算
页的个数 = 虚拟内存大小 / 页的大小
页帧的个数 = 物理内存大小 / 页的大小
例如：虚拟内存有 64KB (2^16B)，物理内存有 32KB()，而页的大小是 4KB(2^12B)。那么页的个数是 64KB / 4KB = 16 个，页帧的个数有 32KB / 4KB = 8 个

78. 缺页错误(Page Fault)

当程序引用一个未映射的页面（访问的虚拟地址并没有对应的映射到一个物理地址），MMU 将导致 CPU 进入操作系统的中断，这个陷阱称为缺页中断。操作系统找到一个很少使用的页，把里面的内容写到磁盘上，然后把之前需要访问的页面读到刚刚回收的页帧里，并修改映射关系，然后跳出陷阱，继续执行刚刚引起中断的指令。

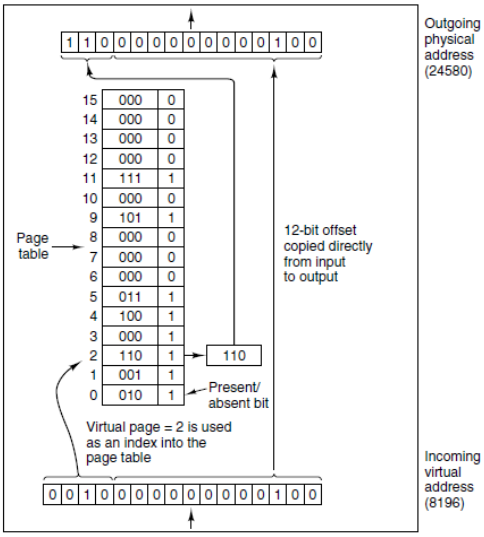
79. MMU 是怎么工作的？

a. 一个虚拟地址被划分为两部分，高位是虚拟页的编号，低位是偏移

b. 虚拟页的编号是一个索引，用于在页表中查找虚拟页的项(Entry)

c. 得到虚拟页的入口后，页帧的编号(如果有)就找到了，此编号即物理地址的前几位

d. 把页帧的编号放到虚拟地址的高几位，与偏移组合，组成最终要访问的物理地址



80. 页表项的结构

a. 页表项的大小：各异，32bits 是一个常见的值，其中最重要的是页帧的编号

b. 存在位(Present/Absent bit)：当此位为 1 时，此项可访问，为 0 时引发缺页中断

c. 保护位(Protection bit)：用来确定该页帧的权限

1) 如果是一位，那么 0 是可读写，1 是只读

2) 如果是三位，则为 rwx

d. 修改位(Modified bit)：用于确定这个页面是否被修改过，也叫做脏位(Dirty Bit)

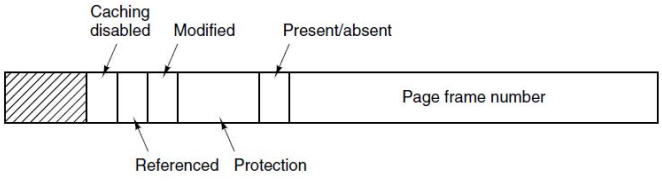
当操作系统要回收这个页时，如果页已被修改，要把页写回磁盘，否则直接丢弃

e. 访问位(Referenced bit)：

只要页面被访问过，系统就将其置 1，没被访问过的页优先被回收

f. 禁止高速缓存位(Caching disabled)

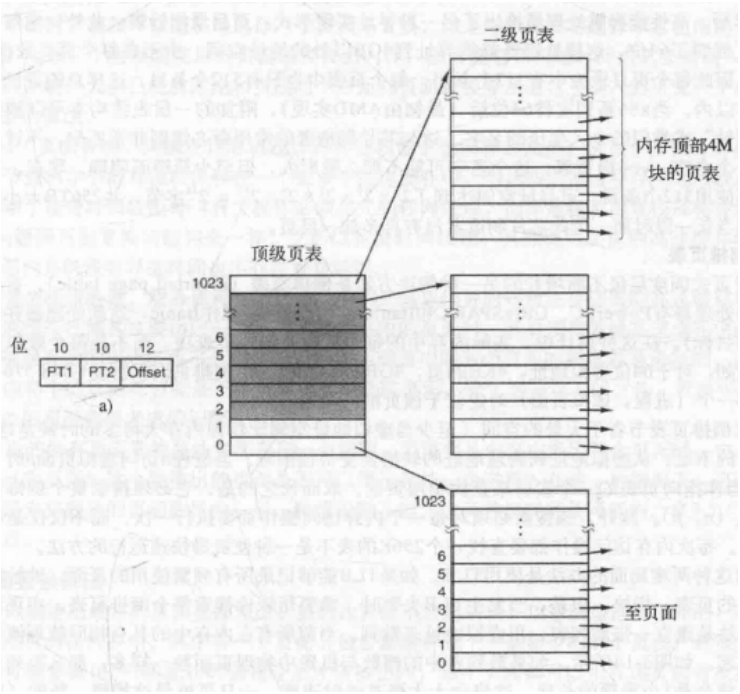
某些虚拟页被映射到了设备寄存器，这些页面要被禁止加载到高速缓存，以免操作系统读到错误的旧数据。有独立 IO 空间而不使用内存映射的机器，不用这一位。



81. 分页的问题
    a. 虚拟地址到物理地址的转换速度必须要非常快(访问内存是较慢的)
    b. 如果虚拟地址空间很大，那么页表的项也会很多

82. 加速分页过程基于一个观察：大部分程序只对少数页表进行多次访问
    a. 转换检测缓冲区 TLB(Translation Lookaside Buffer，又叫做 Associative Memory) 是一种硬件，用于把虚拟地址直接转换为物理地址，而不需要经过页表
    b. 它通常在 MMU 中，通常包括 8 个表项，但很少超过 256 个
    c. 每个表项记录了一个页面相关信息，包括虚拟的页号、页面的修改位、保护位和物理页帧。与页表项基本一一对应，且有一个有效位来记录这个表项是否可用
    d. MMU 把收到的虚拟地址与 TLB 里的所有表项同时进行匹配，发现匹配的有效表项且访问操作不违反保护位(否则产生错误)，则直接把页帧号从表中取出，不需要访问页表，否则正常检查页表，还要淘汰一个 TLB 项，将其从页表内载入。淘汰项的修改位要复制到页表内。

83. 针对大内存的页表
    a. 多级页表
    虚拟地址被分为多个部分，比如两级：PT1 10bits 和 PT2 10bits，偏移占 12bits(可索引 2^12 即 4KB)，先从 Page Table 1 里用 PT1 找到二级页表的入口，再用 PT2 在页表 2 里找到具体的页表项，得到页帧数。
    例如：一个占内存 12M 的(4M Text，4M Data，4M Stack)只要 4 个页表即可描述(一个顶级页表，以及三个二级页表，每个二级页表可以索引 4M 的空间)



    b. 倒排页表(Inverted Page Table)
    与前面所讲的相反，它以物理内存分页并建立表项。例如电脑是 64 位，但内存只有 4G，那么只为 4G 建立表项，需要 4G/4K = 2^20 个表项。用之前的办法要 2^64 / 2^10 = 2^54 个，可以节省大量空间。表项里记录了 PID 和虚页表数，但每次转换地址要搜索整个页表，速度极慢，用 TLB 可有效缓解，但不能彻底解决。

84. 页面置换算法

a. 最优页面置换算法(Optimal Algorithm)
选中最晚可能被访问的页面进行替换，但因为无法预知每个页面什么时候被执行，所以这是一个不现实的算法

b. 最近未使用页面置换算法(Not Recently Used / NRU)
R 位监控读写，W 位监控写，可以有 4 种组合(2^2)，R 位被定期清零
第 0 类：没读，没写 (初始状态)
第 1 类：没读，有写
第 2 类：有读，没写
第 3 类：有读，有写
替换时，随机去选优先级 0 的页，找不到就去 1 类里找，以此类推

c. 先进先出页面置换法(FIFO)
链表加载满后，先被加载到内存的页最先被淘汰

d. 第二次机会页面置换法(Second Chance)
基于先进先出，首次加载后 R 是 0，直到链表加载满
后续加载时，如果头部节点 R 位是 1，则清零并移动到尾部，否则直接淘汰。
如果全部都是 1，会导致全部置 0，然后淘汰头部，与 FIFO 相同

e. 时钟页面置换算法(Clock)
基于第二次机会算法，但使用一个环形链表，有一个指针指向**最老页面**。当发生缺页时，指向的节点如果是 0，则淘汰；如果是 1，则置零并指向下一个节点，直到找到一个 0 为止。

f. 最近最少使用置换算法(LRU/Least Recently Used)
替换未使用时间最长的页面。CPU 每运行一个指令，就把一个 64 位的计数寄存器加 1，这样页表项里必须有一个容纳计数器的区域，每次被访问缺页发生时，找到这个值最小的页表项，将其淘汰。如下图

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3
		1	1	1	3	3	3	2	2	2	2	2	2	2
F	F	F	F		F		F	F	F	F			F	

g. 工作集算法(Working Set)
Demand Paging：进程启动时，所有的页都无效，页只在缺页发生时才加载
Locality of Reference：进程在运行过程中，只用到相对较少的一部分页面
Working Set：进程工作时需要的 Page，如果工作集都在内存，那么不会发生缺页
Thrashing：每执行一些指令就发生交换，导致变慢，一般发生于内存小于工作集
理论：在任意时刻 t，访问过 k 次的页面总会少于 k+1 次的页面。
方案：缺页发生时，挑一个不在工作集的页进行替换，所以要提前辨别进程的“工作集”。但是这个方案代价很大，所以没人用。
Current Virtual Time：进程实际运行时间
对工作集的新定义：实际运行之间之前的 t 秒访问过的页集合
替代方案：每个表项至少有两个信息，上次访问的近似时间和请求位 R，Age 等于运行时间减去上次访问时间。滴答更新 R，缺页时扫描整个表，如果 R 等于 1，则更新访问时间；如果是 R=0 且 Age>t 则替换；如果 R 都是 0 且 Age<t，则选择其中

Age 最大的替换；如果 R 都是 1，则随机挑选一个替换(最好选 M=0 的)

h. 工作集时钟页面替换算法(WSClock)

普通工作集算法非常耗时间，此算法结合了时钟算法，速度快，应用更广泛

所有页表项放到环形列表，缺页发生时，检查指针所指的项

如果 R 是 1，则置零，并指向下一个，继续找

如果 R 是 0，且 Age<t，则跳过

如果 R 是 0，M 也是 0，且 Age>t，则置换此页

如果 R 是 0，M 是 1，且 Age>t, 则加入回写计划任务(M 在回写后清零)，最大回写 n 个

转了一圈还没找到，就随便挑一个 M=0 的替换

85. 局部分配策略和全局分配策略(Local/Global Allocation Policy)

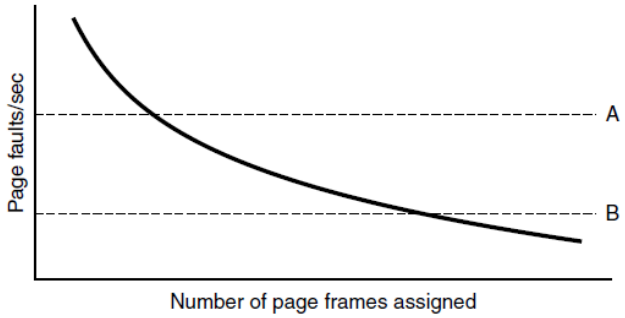
当多个进程 A、B、C 正在运行，当 A 发生缺页时：

a. 局部分配策略是只从 A 里选择一页进行置换
b. 全局分配策略是从 ABC 任意一个进程里选择一页进行置换
c. 通常全局算法工作得比局部算法好，尤其是工作集随时间变化时
d. 如果用局部算法，当工作集增长时，会发生颠簸(Thrashing，比如进程 A 需要的内存很多，其他进程 BC 需要的内存少，结果进程 A 不停触发缺页，BC 的空闲页却没法给它用)。而当工作集收缩时，空闲的内存也得不到使用，造成浪费。

86. 缺页中断率算法(Page Fault Frequency)

PFF 就是每秒的缺页发生次数。

下图中，PFF 超过 A 线，则需要分配更多页帧；低于 B 线，说明页帧分配多了，要回收。



87. 负载控制(Load Control)

当所有进程的工作集大于内存容量时，就会发生颠簸。唯一解决办法就是把其中某些进程整个交换到硬盘，并释放它们占用的全部页面。

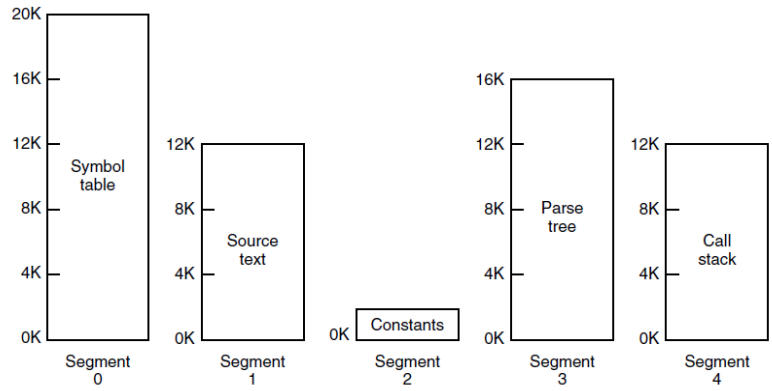
88. 页面大小(Page Size)

a. 页面太大的话会导致内存浪费
b. 页面太小的话会导致页表过大
c. 平均情况下，最后一个页面的一半都是空的(内部碎片)
d. 最佳页面大小算法：$p=\sqrt{2se}$ ，s 是进程平均大小，e 是单个页表项的大小(单位是 byte)
e. 商用计算机的页大小一般是 512B 到 64K，以前典型值是 1K，现在普遍是 4K

89. 分离的指令空间和数据空间(Separated Instruction Space and Data Space)
有些时候虚拟地址太小，无法同时放下指令和数据，所以被分为两个地址空间，一个是 I-space(页表称为 I-space page table)，一个是 D-space(页表称为 D-space page table)

90. 分段(Segment)
问题：当一个编译器编译程序时，符号表、代码、语法分析树等，虽然在一个页表的不同部分，但是随着占用的增长，可能会互相碰撞。解决这个问题的办法是分段。
a. 把内存被划分为可变大小的部分(也叫段，属于不同的地址空间)，并划分给进程，这个技术叫做分段
b. 段是线性序列地址组成，地址从 0 开始到某个允许的最大值
c. 每个段的长度可以从 0 开始，到最大允许的地址值
d. 不同的段可能有相同的长度
e. 段的长度可能在执行过程中发生改变
f. 每个段有独立的地址空间
g. 每个段的增长和缩减是独立的，不会影响其他段

91. 段的地址
段是被当做二维存储来管理的，一个虚拟地址包含了段的编号和在段内的地址(偏移)。这样一来，就算其中某部分有变化，也不至于重新编译全部的段(如动态链接库 dll 和 so)



考查点	分页	分段
需要程序员了解正在使用这种技术吗?	否	是
存在多少线性地址空间?	1	许多
整个地址空间可以超出物理存储器的大小吗?	是	是
过程和数据可以被区分并分别被保护吗?	否	是
其大小浮动的表可以很容易提供吗?	否	是
用户间过程的共享方便吗?	否	是
为什么发明这种技术?	为了得到大的线性地址空间而不必购买更大的物理存储器	为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护

图3-32 分页与分段的比较

92. 纯分段的实现

程序长期运行后，由于段的加载、卸载和替换，会导致内存空洞，这些空洞叫棋盘形碎片(checkerboarding)或者外部碎片(external)，导致浪费，可通过压缩(compaction)解决。

分段和分页的结合：MULTICS 系统

a. 过于巨大的段是不方便的，甚至是无法把它们整个放到内存里
b. 仅仅把段真正需要的页加载进来
c. 每个段被划分成页，每个段都有一些页组成
d. 每个 MULTICS 程序有一个段表，每个段有一个描述符(descriptor)

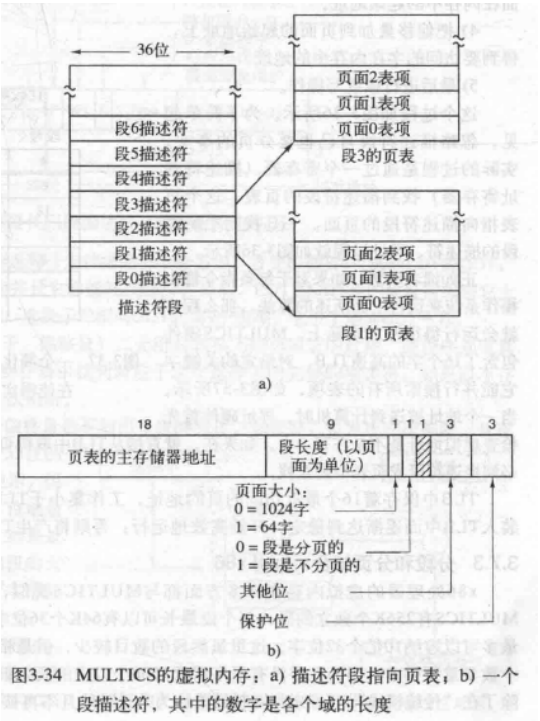


图3-34 MULTICS的虚拟内存：a) 描述符段指向页表；b) 一个段描述符，其中的数字是各个域的长度

e. 一个 MULTICS 的虚拟地址由段号和段内地址构成，段内地址由页号和偏移量构成。先由段号去段表找段的页表(没加载会进入系统陷阱，加载页表)，然后在页表里查找页号(没加载就触发缺页)，最后用页表项里的页帧号加偏移地址，得到物理地址。

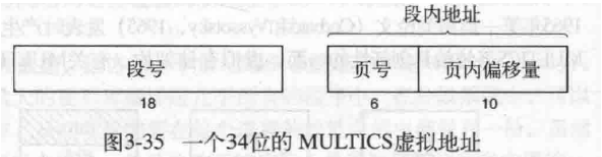


图3-35 一个34位的 MULTICS虚拟地址

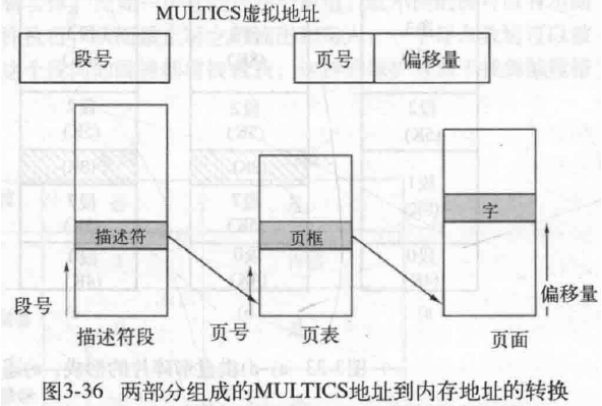


图3-36 两部分组成的MULTICS地址到内存地址的转换

f. MULTICS 也有一个 TLB，直接拿到页帧号，加速虚拟地址转换

这个表项是否在使用?

比较域

段号	虚拟页面	页框	保护	生存时间	
4	1	7	读/写	13	1
6	0	2	只读	10	1
12	3	1	读/写	2	1
					0
2	1	0	只执行	7	1
2	2	12	只执行	9	1

图3-37 一个简化的MULTICS的TLB，两个页面大小的存在使得实际的TLB更复杂

93. 长期存储信息的三个基本要求
    a. 能够存储大量信息
    b. 使用信息的进程终止时，信息依然存在
    c. 必须能使多个进程并发访问有关信息

94. 什么是文件
    文件是进程创建的逻辑信息单元

95. 文件名
    a. 很多系统支持长达 255 个字符(不是字节)的文件名
    b. 文件名由多个字符和扩展名组成，扩展名通常是 1-3 个字符(也可以更多)
    c. Windows 上文件扩展名被注册到系统，并为每个扩展名指定了可以打开它的“主人”

96. 三种不同的文件结构
    a. 无结构的字节序列：用户程序定义文件结构，操作系统(Unix/Windows)不关心结构
    b. 有内部结且长度固定的序列：普通操作系统一般不采用，以前打孔机用
    c. 用树来记录数据：每个记录长度不必相同，每个记录的固定位置有一个 Key 用于排序和快速查找，主要用于处理商业数据的大型计算机(Large Mainframe Computers)

97. 普通文件
    a. ASCII 文件(Regular File)
       可以被直接输出或打印，可以被文本编辑器编辑
    b. 二进制文件(Binary File)
       打印出来的二进制文件是无法被理解的，充满混乱字符的一张表。
       有自己的内部结构，使用这些文件的程序才知道这些结构
       例如：
       1) 可执行文件是一个有正确格式的字节序列，有 5 个区别：header, text, data, relocation bits, 和 symbol table. 头部有一个魔术数，来表明该文件是可自行文件
       2) 存档文件，由一系列已编译但没链接的文件组成，文件中包含名称、创建时间、所有者、保护代码、大小。

98. 文件访问
    a. 顺序访问(Sequential Access)，例如磁带
       从头到位读取，无法跳过部分内容，可以返回到起点，可以多次读取，早期 OS 用
    b. 随机访问(Random Access)
       可以以任意的顺序读取文件中的字节或记录，可以指示可以从何处开始读取文件(seek/read)

99. 文件属性(Attribute)又可称为元数据(Metadata)
    包括了文件的创建时间、文件大小、权限等各种信息

100. 文件操作
    创建、删除、打开、关闭、读取、写入、追加、Seek、获取属性、设置属性、重命名

101. 目录系统
    a. 一级目录系统(Single-Level Directory System)
       一个目录中包含所有的文件，称为根目录
    b. 层次目录系统(Hierarchical Directory System)
       一种把相关文件分组的方案，用树来组织目录和文件

102. 路径名
    a. 根目录
       Windows: \
       *NIX: /
       MULTICS: >
    b. 工作目录：当前工作的目录(.)
    c. 绝对路径：从根目录到文件
    d. 相对路径：相当于工作目录的路径

103. 目录操作
    打开、关闭、读取(获取目录里的下一个文件，用于遍历文件)、改名
    链接(link，软/硬链接)，硬链接增加 i-node 的计数器，软链接类似快捷方式 shortcut
    删除(unlink)

104. 文件系统布局
    a. 文件系统放在磁盘上
    b. 0 号扇区(sector)称为 MBR 主引导记录(Master Boot Record)，用于引导计算机
    c. MBR 的结尾是分区表(给出了每个分区的起止地址)，其中一个分区被标记为活动
    d. 每个分区可以有不同的文件系统(FAT/NTFS 等)
    e. MBR 确定活动分区，读入它的第一个块(引导块 Boot Block)，并执行
    f. 引导块将装载该分区中的操作系统
    g. 引导块之外的磁盘分区布局因文件系统不同而不同

105. 超级块 Superblock

包含了文件系统的所有关键参数。在计算机启动或者文件系统第一次被访问时，加载到内存。其中可能包括：文件系统的 Magic Number，块的数量和其他重要的管理信息。

106. 文件的实现

连续分配：每个文件作为一连串连续的数据块存储在磁盘上。

优点：

a. 实现简单，只需要两个数字，一个是第一个块的地址，另一个是文件长度
b. 读操作性能好，单个操作即可读出整个文件

缺点：随着时间的推移，磁盘会变得零碎(Fragmentation)

链表分配：每个文件用一个链表块来存。块首字(Word)指向下一个块，其他部分存数据

优点：

a. 每个块都可以被用到
b. 只在每个文件到最后一个块可能有内部碎片

缺点：

a. 存储数据的字节数不是 2 的整数次幂，有一些效率问题
b. 速度慢，读取第 N 块需要先遍历 N+1 个块

内存中的表进行链表分配：把指针从磁盘放到内存的表(FAT: File Allocation Table)里

优点：目录项中仅需记录一个整数(起始块的编号，注意不是地址)

缺点：整个表都要放在内存里。如果磁盘是 1T，块大小是 1K，则有 10 亿个项

i 节点分配：i-node 是一个数据结构，包含了文件属性和文件块的磁盘地址

优点：

a. 只有当文件被打开时，i-node 才需要被加载到内存
b. 打开 i-node 大小为 k 的 n 个文件只占 k*n 个字节的内存)

实现：

a. 需要在内存中有一个数组，数组大小决定了最大可打开文件的个数
b. 每个 i 节点的最后一项指向的块不包含数据，而包含一堆额外的块地址
c. 对 b 更高级的解决方案是
    1) 可以有两个或更多个包含磁盘地址的块
    2) 或者跟多级页表类似，指向其他存放地址的磁盘块的磁盘块

107. i-node 例题

一个 i-node 可以存放 10 个直接地址和 1 个间接地址，每个地址是 8 字节，每个块是 1024 字节。请问文件最大大小？

10 * 1024 + (1024/8) * 1024 = 138 * 1024 = 141312 B = 138KB

108. 目录的实现

a. 目录包含了一个长度固定的列表
b. 每个项对应一个文件，包括了一个长度固定的文件名，一个文件的属性结构，还有一个或多个磁盘块地址
c. 如果采用 i-node 方案，那么每个项只需要一个文件名和一个 i-node 编号即可

d. 文件名长度通常是 255 个字符
e. 如果每个项的大小一致，文件名包含在项中，文件名短的会浪费较多空间
f. 如果每个项的大小不一致，文件名包含在项中，会导致文件被移除时，新文件项目可能塞不下，或还是有浪费
g. 如果每个项的大小一致，文件名不在项里，而全部在项后面的堆(heap)里，这样文件被移除了，后面的新文件也可以重复用这个项

109. 常见的日志文件系统

NTFS，Ext3，ReiserFS

110. 计算机崩溃导致文件系统产生一致性的问题

比如，删除一个文件需要这几步操作

a. 在目录中删除文件 (崩在这后面，无法把 i-node 和磁盘块分配给其他文件)
b. 释放 i-node 到空闲 i-node 池(崩在这后面，无法把磁盘块分配给其他文件)
c. 归还磁盘块给空闲磁盘块池

111. 日志文件系统解决一致性问题

a. 在操作前保留一个日志，这样就算系统崩溃，重启后也能根据日志继续操作
b. 为了让日志文件系统工作，被写入日志的操作必须是幂等的(重复操作无害)
c. 跟数据库类似，操作放在事务开始和结束之间，要么全部完成，要么全部放弃

112. 虚拟文件系统(Virtual File System, VFS)

a. 目的是把不同的文件系统放到统一的有序结构里(如 Linux 里的 mount)
b. 上层接口被用户进程调用，比如 read/seek 等函数，即有名的 POSIX 接口
c. 下层接口即 VFS 接口，用于操作实际的文件系统，由文件系统实现者向 VFS 提供

113. VFS 如何工作

a. 系统启动时，根文件系统在 VFS 中注册
b. 其他的文件系统在启动或操作过程中向 VFS 注册
c. 向 VFS 注册的过程就是提供一个包含 VFS 所需要的全部函数的地址列表
d. VFS 调用预定 Index 的函数，即可进行特定的操作，比如读一个块

114. 如何决定块的大小

a. 块太小会浪费时间(读一个文件要连续读多个块)，块太大会浪费空间(占不满)
b. 历史上块的大小是 1-4k，现在的磁盘大了，选 64k 虽然有点浪费，但可能更好

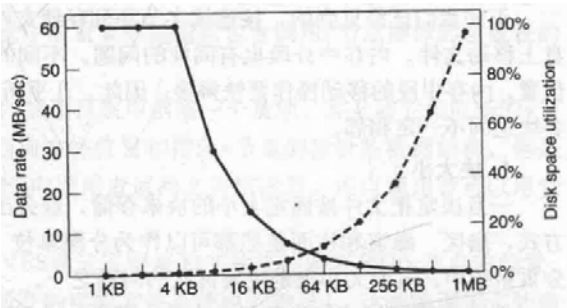


图4-21 虚线（左边标度）给出磁盘数据率，实线（右边标度）给出磁盘空间利用率（所有文件大小均为4KB）

115. 记录空闲块(Keeping Track of Free Blocks)有两个办法

a. 把空闲表放在磁盘块链表中。

比如一个块是 1K，磁盘块号要 32 位(4 字节)，那么可以放下 1KB/4B=256B 个块号，其中 255 个是空闲块的块号，最后一个指向下一个存放空闲表的块号。1T 的硬盘需要 400 万的块存储全部块号。

b. 位图

空闲块用 1 表示，非空闲块用 0 表示，找一个给定位置附近的空闲块更容易。1T 的硬盘需要 131072 个 1KB 的块来存储。(2^30bits / (1024*8))

只有当几乎没有空闲空间的时候，a 方法占用才会小于 b 方法，否则 b 方法很省空间

116. 文件系统备份(备份是 backup，转储是 dump)

a. 解决两个问题：灾难恢复和错误操作

b. 方式有两个：

1） 物理转储：备份全部块，简单速度快但无法跳过或还原指定项

2） 逻辑转储：从一个或多个目录开始递归备份，可做增量，可还原指定项

117. 逻辑转储算法分为四步

a. 从顶层目录开始扫描，“被修改过文件”和“全部目录”被标记出来

b. 再次扫描，把“不包含文件修改的目录”取消标记

c. 按照 i-node 编号顺序扫描 i-node，转储所有“被标记的目录”(没有包括文件)

d. 转储所有“被标记的文件”(这次不包括目录)

118. 文件系统的一致性问题

原因：系统在有修改的块都被写完前崩溃了，文件系统进入了不一致的状态

fsck：file system consistency check

119. 两种一致性检查方式

a. 块的一致性

1) 检查程序构建两张表，每张表为每个块建立一个计数器，都设置为 0
2) 第 1 张表统计每个块在一个文件里出现的次数
3) 第 2 张表统计每个快在空闲块列表(或位图)里出现的次数
4) 表 1 和表 2 的同一块数据应该是 1 和 0 相反
5) 如果表 1 表 2 同一个块都为 0，则此 block 丢失，应该加入到空闲表
6) 如果表 1 为 2，表 2 为 0，则此 block 被多个文件使用(不应出现)，应复制此块
7) 如果表 1 为 0，表 2 为 2，则此 block 在空闲区内重复，要重建空闲表

b. 文件的一致性

1) 检查目录系统，用一个表统计每个文件出现的次数
2) 从根目录出发，递归目录树
3) 对每个目录里的每个 i-node，计数器加 1
4) 硬链接计数器可能超过 1，软连接不计数
5) 得到一个以 i-node 为索引的表，表示每个文件在多少个文件夹里出现过
6) 把上表中的数和 i-node 本身存储的数进行比较
7) 若计数器项大于 i-node 本身的计数，则应把 i-node 中的计数进行修正
8) 如果 i-node 本身的计数大于计数器的计数，也应该修正 i-node 中的计数

120. 文件系统性能

a. 高速缓存缓存

目的：用块高速缓存(block cache)或缓冲区高速缓存(buffer cache)减少磁盘访问

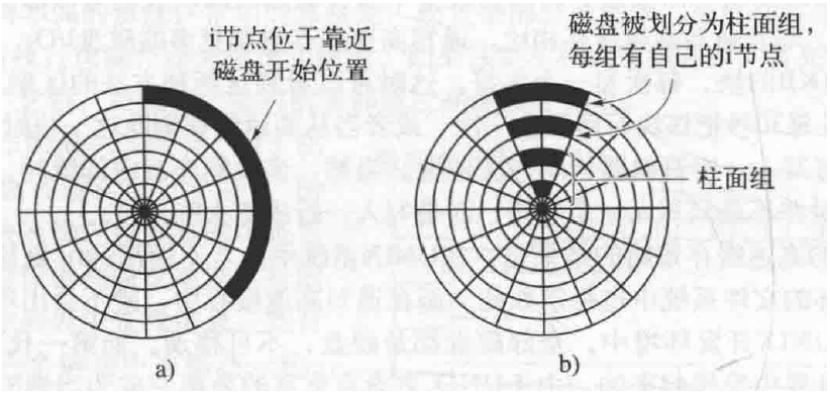
定义：一系列的块，逻辑上属于磁盘，但实际上基于性能的目录被保存在内存中

原理：读磁盘块时首先检查缓存，如果不在缓存中，则读到缓存，再复制到目的地

查找：缓存块很多，为了加速查找，将设备和地址 Hash，在 Hash 表里找缓存地址

b. 减少磁盘臂运动

1) 在机械磁盘里，把 i-node 放在磁盘的中部，而不是磁盘的开头，这样从 i-node 到第一个 block 的寻道时间是原来的一半；将每个磁盘分为多个柱面组，每个柱面有自己的 i-node、数据块、空闲表。如下图



2) 固态硬盘使用了和闪存(flash cards)一样的技术，可以随机访问，跟机械硬盘的顺序访问速度接近。缺点是写入的次数是有限的。

121. MS-DOS 文件系统

a. 历史: 是 Win98/Me 的主要文件系统，也被后续的 Windows 所支持，不是最新的 PC 标准，但是它和它的扩展(FAT32)至今被广泛使用，比如数码相机、MP3、iPod

b. 机制：文件分配表 FAT 放在内存里。而目录项(目录里记录每个文件的项)为 32 字节，包含了文件第一个块的块号(不是地址)，块号用于在一个有 64K 个项(不定，见下表 FAT16)的文件分配表里找到文件第一个块的地址，继而找到全部地址(是链表)

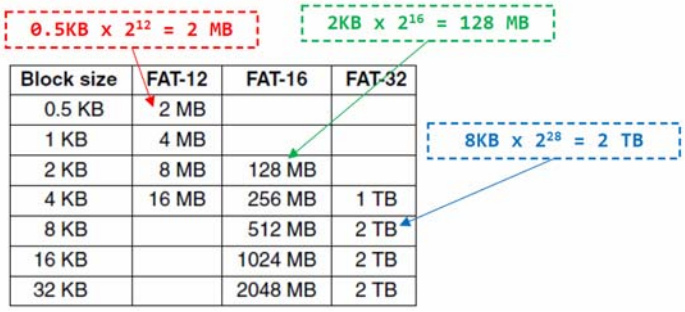
c. 支持 4 个分区，常见的分区大小限制如下，计算公式为：块大小 * 2^寻址位数

FAT12 的块大小最大是 2^12(4K)，FAT 表项是 2^12 个(4K)，单个分区最大 16MB

FAT16 的块大小最大是 2^15(32K)，FAT 表项是 2^16 个(64K)，单个分区最大 2G

FAT32 的块大小最大是 2^15(32K)，FAT 表项是 2^28 个(64K)，单个分区最大 2T(单个磁盘也只能是 2T，原因是一个磁盘只能有 2^32 个逻辑扇区，每个扇区 512 字节，这是硬件限制的)

d. 一个 8G 的硬盘，需要 FAT16 的 4 个分区，或 FAT32 的最少 1 个分区



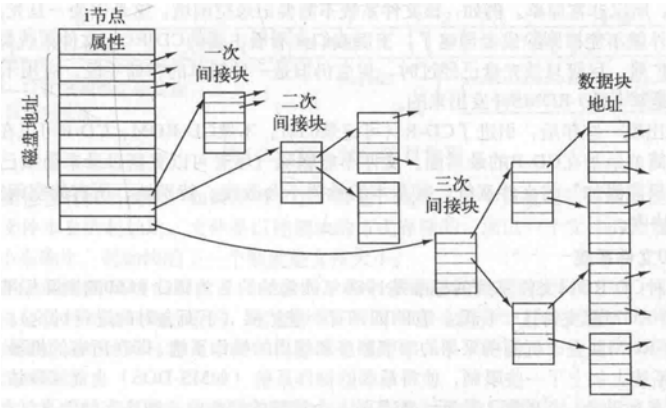
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

122. Unix v7 文件系统

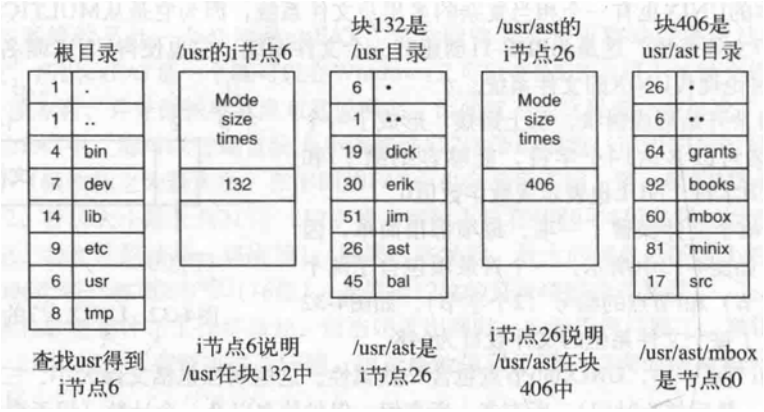
a. 每个目录项由 2 字节(16 位)的 i-Node 编号和 14 字节文件名组成(最多 2^64 个文件)

b. i-Node 包含文件属性，包括大小，三个时间，所有/在组，保护信息，计数

c. i-Node 后面是 10 个直接地址，余下三个分别是 1/2/3 次间接地址，如下图



d. 根目录的 i-node 在磁盘的固定位置，查找文件时从根目录开始，逐层查找，如下图



123. I/O 设备

a. 块设备(例如磁盘、U 盘)

信息存储在固定大小的块中，块有自己的地址

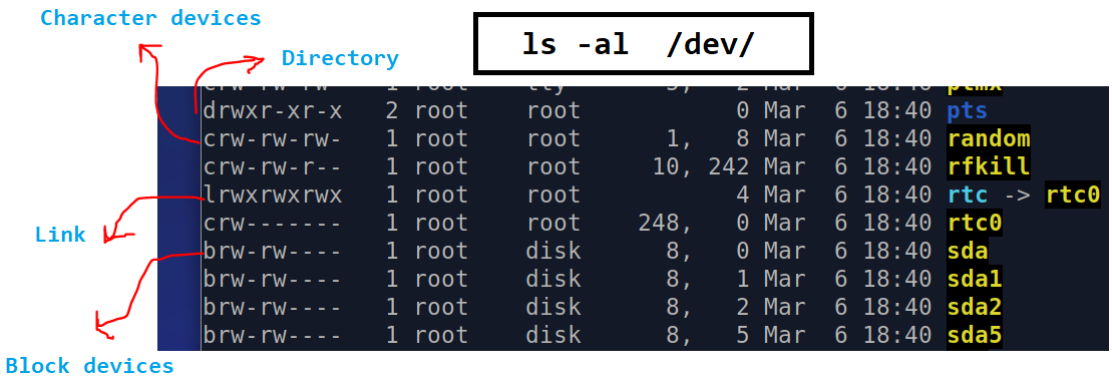
每次传输 1 个到多个连续块

块的大小从 512 字节到 64K 不等

b. 字符设备(例如打印机、网卡、鼠标)

接收或发送字符流，不考虑任何块结构

不可寻址，无法进行 seek 操作



124. Linux 下的设备编号分为 Major Number 和 Minor Number
    a. Major Number 用于定位合适的驱动程序
    b. Minor Number 用于传参给驱动程序，指定要读写的单元，如/dev/sda2

```
OS:>> ls -g /dev/sda*
brw-rw-rw- 1 AD+Group(513) 8, 0 Aug  8 14:39 /dev/sda
brw-rw-rw- 1 AD+Group(513) 8, 1 Aug  8 14:39 /dev/sda1
brw-rw-rw- 1 AD+Group(513) 8, 2 Aug  8 14:39 /dev/sda2
OS:>>
```

125. I/O 设备控制器(I/O Controller，又叫适配器 Adapter)
    I/O 设备一般由机械组件和电气组件构成，电器组件就是 IO 控制器
    例如硬盘的磁头磁盘都是机械组件，给程序提供读写控制的电路就是 IO 控制器

126. I/O 设备控制器的任务
    a. CPU 和设备之间的接口
    b. 开始/停止设备的活动
    c. 把串行的比特流转换为字节块
    d. 检测和修正错误
    e. 把数据传输到主存(一般是内存)
    f. 有些控制器可以控制多个相似的设备
    g. 每个控制器有一些寄存器用来与 CPU 沟通
        向寄存器写数据，可以给 I/O 设备发命令，使其传输或接收数据，启动或停止等等
        从寄存器读数据，可以得到设备的当前状态

127. CPU 如何与 I/O 设备通讯
    a. I/O 端口号
        每个控制寄存器被分配一个 IO 端口号(8 位或 16 位)
        通过特殊的指令，可以通过 IO 端口号对 IO 寄存器进行读写，如下图

```
Read
IN  REG, PORT
IN  REG, 4          (CPU register REG = The content of port 4)

Write
OUT PORT, REG
OUT 4, REG          (The content of port 4 = CPU register REG)
```

    b. 内存映射 I/O
        把 I/O 寄存器全部映射到内存中
        每个寄存器有一个独立的内存地址，不会分配给其他设备或程序
        这些地址一般都是在内存的顶端，比如靠近或等于 0xFFFFFFFF 的区域
    c. 混合方案：同时混用 ab 两个方案，书上没怎么讲，看图

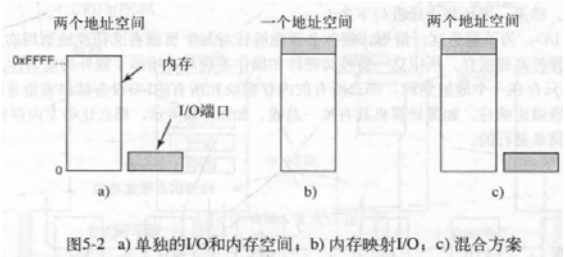


图5-2 a) 单独的I/O和内存空间，b) 内存映射I/O，c) 混合方案

128. 上述方案的优缺点

a. I/O 端口
   必须使用特殊的汇编代码进行读写

b. 内存映射
   I/O 寄存器是内存的一部分，可以直接用类似 C 语言的办法进行寻址和读写
   不需要特殊的保护机制来阻止用户进程进行 I/O 读写
   引用内存的每一种指令，也可以用来引用 I/O 寄存器

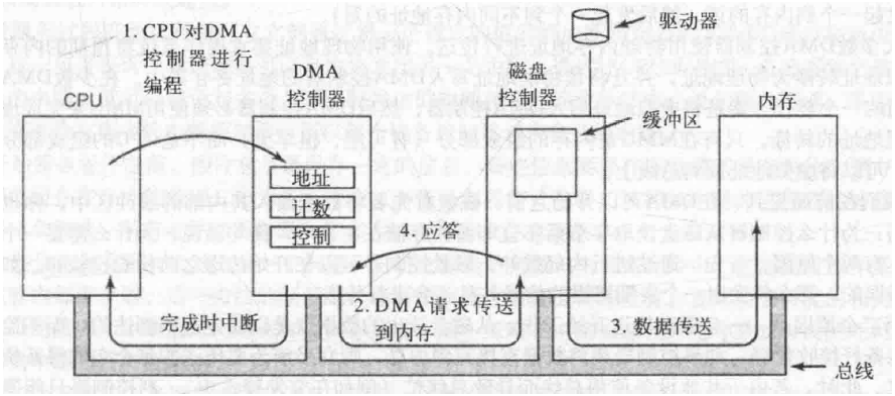
129. 内存映射 I/O 如何工作

a. CPU 想要读一个字，就把地址放在 Bus 地址线上，并在 Bus 控制线发出读信号
b. 用一个 Bus 信号告知是用 I/O 空间还是内存空间
c. 如果是内存空间，内存负责响应这个请求
d. 如果是 I/O 空间，I/O 设备负责响应这个请求

130. DMA 直接存储器存取

CPU 直接操作 IO 控制器，每次只能取一字节的数据，但是这样浪费 CPU 时间
DMA 可以把整块的数据读写到内存，CPU 可以不用参与，步骤如下

a. CPU 对 DMA 控制器进行编程
b. DMA 向通往磁盘的 Bus 发起读请求，来启动一个读传输
c. 要写的内存在 Bus 的地址线上，IO 控制器从内部缓冲读到一个字，再写到这个地址，写操作也是一个标准的 bus 周期
d. 写完后，IO 控制器在 Bus 上发送一个应答信号给 DMA
e. DMA 对地址做加法，并对要读的字节数做减法，重复 b-d，直到要读的字节数为 0
f. DMA 发送一个中断信号给 CPU，告知 CPU 读取已完成



131. DMA 的两种模式

a. 周期窃取(Cycle Stealing)
   DMA 获取 Bus 的使用权来请求传输一个字
   此时 CPU 如果要用总线，就得等，CPU 被拖慢，不高效
   IO 设备偶尔会偷偷溜入，并偷走一个总线周期

b. 突发模式(Burst Mode)
   DMA 发起一连串的传送然后释放总线
   比周期窃取的效率高

132. 重温中断

a. IO 设备完成工作后，在 Bus 上发出信号引发中断
b. 中断控制器立刻检测到中断，并放一个数字到地址线，标明哪个设备要中断 CPU
c. CPU 停止现有操作，并使用 b 中的数字作为索引查找**中断向量表**，获得一个 Program Counter (就是中断处理函数 interrupt-service procedure 的首地址)
d. 中断处理函数运行后，向中断控制器的某个 I/O 端口写一个特定的值，告知其可以自由的发出另一个中断

133. 精确中断(Precise Interruption)

a. 老式机器上，中断发生时，微程序或硬件保证迄今为止的指令全部被执行完了，且后续的指令都没有被开始执行
b. 新式的机器 CPU 更加复杂，可能有超标量，导致中断发生时，有些指令没执行完
c. 把机器留在一个明确状态的中断叫做精确中断，它具有 4 个特征
    1) PC 程序计数器保存在一个已知的地方
    2) PC 所指向的指令之前的指令全执行完了
    3) PC 所指向的指令之前的指令全部没有执行
    4) PC 所指向的指令的执行状态是已知的
d. 不满足 c 条件的叫做非精确中断(Imprecise Interruption)

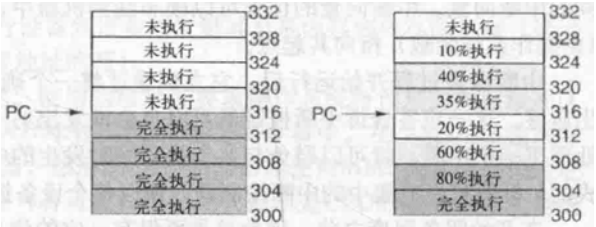


图5-6 a) 精确中断；b) 不精确中断

134. I/O 软件的目标

a. 设备独立性(Device Independence)
    可以访问任意 IO 设备而无需事先制定设备。比如读写磁盘/CD 无需指定设备类型
b. 统一命名(统一命名)
    一个文件或一个设备的名字应该是一个字符串或整数，不依赖于设备类型
    例如 Linux 的目录体系，可以挂在新分区到任意一个目录下，直接访问
c. 错误处理
    应该尽可能在接近硬件的层面进行处理(IO 控制器->驱动程序->上层代码)
d. 同步和异步
    IO 设备本身是异步的，但操作系统把中断驱动的操作变为用户看来是阻塞的操作
e. 缓冲
    数据离开设备之后，通常不能直接存放到它的目的地
f. 共享设备和独占设备
    基于硬件特性，分为独占和共享。磁盘可以由多用户使用，而磁带却不能
    操作系统必须能处理这两种设备，避免发生问题

135. 程序控制 I/O

所有的工作由 CPU 直接承担，实现简单，进行 IO 操作时 CPU 会轮询或忙等待

136. 中断驱动 I/O

中断允许 CPU 在等待 IO 操作时做别的事情

137. 使用 DMA 的 I/O

CPU 给 DMA 控制器发指令，DMA 控制器跟 I/O 设备交互，I/O 交互期间 CPU 不参与

138. I/O 软件从上到下层次为

用户层 IO 软件，与设备无关的操作系统软件，驱动程序，中断处理程序，硬件

139. 设备驱动程序

a. 每个控制器有一些寄存器，CPU 可以通过他们发指令或/和读取 IO 设备的状态。每个设备都需要由厂商提供特定的代码来控制，这就是驱动
b. 通常驱动是系统的一部分，但是在用户空间的驱动也是可能的

140. 与设备无关的 I/O 软件，执行对所有设备公共的 I/O 功能，并向用户层提供接口

a. 设备驱动程序的统一接口：可以让添加新驱动变得简单
b. 缓冲
    1) 无缓冲，用户程序直接读 IO，系统拿到一字节就中断，用户程序来读，很慢
    2) 纯用户空间缓冲，速度快了，但缓冲区可能被换页
    3) 用户缓冲 + 单个内核缓冲，满了无法处理新数据
    4) 用户缓冲 + 两个内核缓冲，可解决上述问题
c. 错误报告
    1) 编程错误：请求一些不可能的操作，比如写入鼠标等输出设备，或读取打印机
    2) 实际的 I/O 错误：比如读写一个坏磁盘块，驱动决定怎么干，否则向上层(与设备无关的软件)报错
d. 分配与释放专用设备

    有些设备只能被一个进程使用，操作系统可以根据设备状态决定接受还是拒绝请求
e. 与设备无关的块大小

    不同设备有不同的扇区大小，与设备无关的软件负责隐藏这个事实，向高层提供统一的块大小

141. 用户空间的 I/O 软件

大部分 IO 软件在系统内核，但一小部分在用户空间。系统调用，包括 I/O 系统调用，通常由库实现

142. 磁盘扇区数的计算

柱面数 * 单柱面的磁道数 * 每个磁道的扇区数

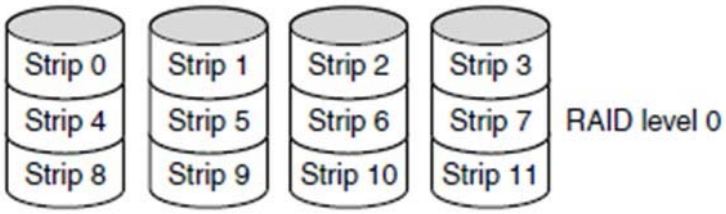
143. 现代磁盘

a. 被划分成环带(Zone)，外层环带比内层环带拥有更多的扇区
b. 把虚拟的几何规格(Virtual Geometry)呈现给系统，隐藏每个磁道有多少扇区的细节

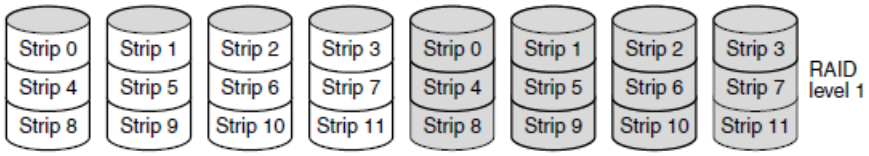
144. RAID (Redundant Array of Inexpensive Disk)
    a. SLED (Single Large Expensive Disk)
    b. 系统把数据发到 RAID 控制器，RAID 控制器再分发到磁盘
    c. 软件不用做修改，就可以直接使用
    d. 数据可以并行的分发到多个磁盘
    e. 系统像用一个磁盘一样来用 RAID，性能和可靠性提高了

145. 条带(Strip)
    a. 把数据分发到多个磁盘
    b. 1 strip = k sectors (如果 k=2，那么每个条带是 2 个扇区)

146. RAID 0
    a. 容量是全部硬盘容量的总和
    b. 对于大数据量的请求性能很好
    c. 对每次请求一个扇区(不到一个条带)的系统，无法并发，性能最差
    d. 没有冗余，不算真正的 RAID



147. RAID 1
    a. 复制了所有的磁盘
    b. 一半磁盘是主磁盘，一半磁盘是备份磁盘
    c. 执行一次写操作，每个条带都被写两次
    d. 与 b 不同的是，读的时候，可以从主备任选一个磁盘
    e. 是真正的 RAID
    f. 写的性能不变，读性能是两倍
    g. 容错性大大提高，一个磁盘坏了，直接读备用的那个就行
    h. 恢复也很简单，直接把好的那个磁盘的数据全部复制过去



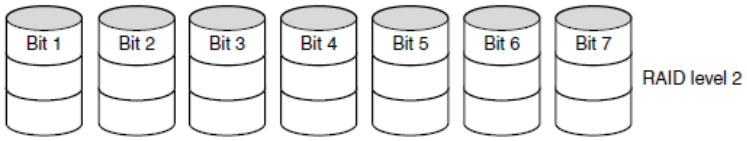
148. 奇偶校验
    a. 奇校验：数据里 1 的个数必须是奇数
    b. 偶校验：数据里 1 的个数必须是偶数
    c. 向数据添加 1 个位，使得其 CodeWord 可以进行奇偶校验，具体哪个错了不知道

149. 汉明码

a. m 是数据的位数，r 是要添加的位数，必须满足 $(m + r + 1) \leq 2^r$
b. Code Word 是数据(Data)和校验位(Extra Bits)生成的新数据
c. 错误的位可以被检测和修正
d. 校验码必须放在 2 的 n 次方位，Index 从 1 开始而不是 0
e. 处于第 n 位的校验码，从自身起校验，然后跳过 n 位，又校验一位，直到结束

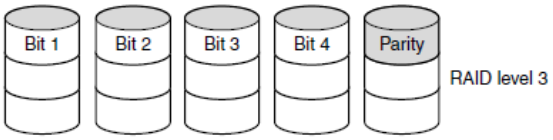
150. RAID 2

a. 假设有 7 个磁盘，数据被分割成 4bits，再加 3 个校验位，总共 7bits
b. 如果丢失了一个磁盘，不会有问题，因为汉明码可以将其修正
c. 所有驱动器的磁盘臂位置和旋转度数必须同步
d. 会浪费很多空间用于存储校验位
e. 这里的汉明码是用异或



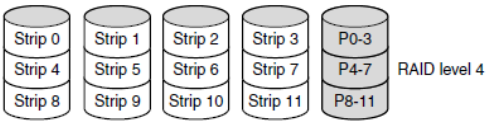
151. RAID3

a. 是 RAID2 的简化版本，但只用一个单独的磁盘计算奇偶校验位
b. 所有驱动器的磁盘臂位置和旋转度数必须同步
c. 如果有一磁盘坏了，就假设它的数据是 0，如果出现了奇偶错误，则其应该是 1



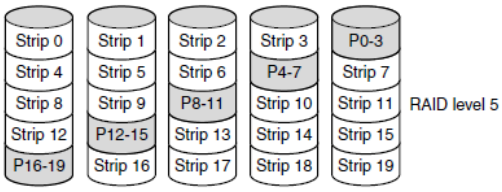
152. RAID4

a. 用条带为单位，而不是单个 bit 为单位，进行异或操作的奇偶校验
b. 校验结果放到其中一个磁盘，可以恢复一个条带的错误
c. 不再需要同步磁盘臂位置和旋转度数
d. 一个扇区被修改，所有磁盘上对应条带都要被依次读取，计算奇偶校验位，再回写
e. 奇偶校验位所在的磁盘任务繁重，会成为 IO 瓶颈



153. RAID5

a. 全部跟 RAID4 一样，但是均匀的把奇偶校验条带分布到不同磁盘 RAID4 瓶颈
b. 如果磁盘挂了，数据恢复是一个复杂的过程，要重新构造数据很复杂



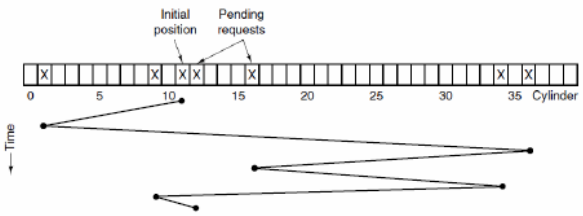
154. RAID6
   a. 与 5 差不多，但是每个奇偶校验条带在下一个驱动器都有备份
   b. 写数据的代价高一点，但是读不会变坏，且更可靠

155. 磁盘格式化
   a. 磁盘使用之前必须低级格式化
   b. 格式包含一系列的同心圆，每个圆有一系列扇区，扇区之间有微小间隔
   c. 一般是每个扇区 512B，包含前导码(Preamble)和 ECC(16B)
   d. 为提升性能，磁道有柱面斜进(Cylinder Skew)，0 扇区与上一磁道末扇区有间隔
   e. 例子：10000RPM(10000 转每分钟，即 60ms 一转)，每磁道 300 扇区，则 20us/扇区。如果寻道时间要 800us，寻道时，有 40 个扇区会被转过磁道，所以斜进是 40 扇区

156. 单交错与双交错(interleaving)
   a. 为了给控制器和磁盘缓存喘息的时间，扇区的编号可以是交错的，间隔 1 个扇区的是单交错，间隔 2 个扇区编号的是双交错
   b. 现代磁盘往往可以缓存一个磁道，这样就不需要交错了

157. 磁盘臂调度算法
   a. 读写磁盘块需要三个时间决定：寻道时间、旋转延迟、实际传输时间。其中寻道时间占主导，所以要最小化寻道时间
   b. 先来先服务(FCFS)：每次接收一个请求并按顺序完成，则基本不可能优化，如下图

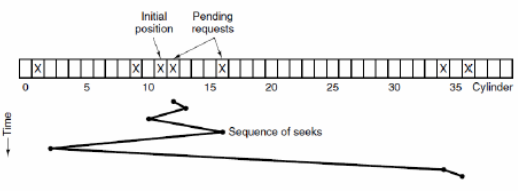
```
11, 1, 36, 16, 34, 9, and 12 ⇒ 11, 1, 36, 16, 34, 9, and 12
```




```
Arm motions = (11-1) + (36-1) + (36-16) + (34-16) + (34-9) + (12-9)
            = 10 + 35 + 20 + 18 + 25 + 3 = 111
```

   c. 最短寻道优先(SSF)：每次都处理跟当前扇区最近的请求，可能磁道两极总不能被读

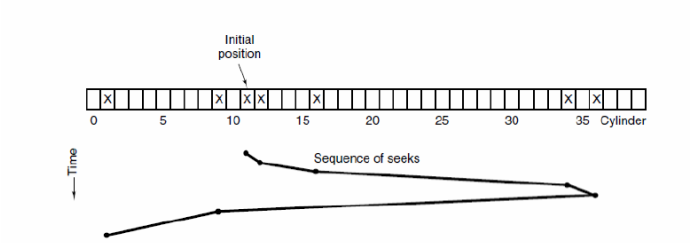
```
11, 1, 36, 16, 34, 9, and 12 ⇒ 11, 12, 9, 16, 1, 34, and 36
```




```
Arm motions = (12-11) + (12-9) + (16-9) + (16-1) + (34-1) + (36-34)
            = 1 + 3 + 7 + 15 + 33 + 2 = 61
```

d. 电梯算法：有个二进制位来确定是前进还是后退，依然找路线上的最近扇区

```
11, 1, 36, 16, 34, 9, and 12 ⇒ 11, 12, 16, 34, 36, 9, and 1
```




```
Arm motions = (12-11) + (16-12) + (34-16) + (36-34) + (36-9) + (9-1)
            = 1 + 4 + 18 + 2 + 27 + 8 = 60
```

158. 错误处理

a. 小的错误可以用 ECC 修正

b. 用备用扇区来取代坏的扇区，如下图

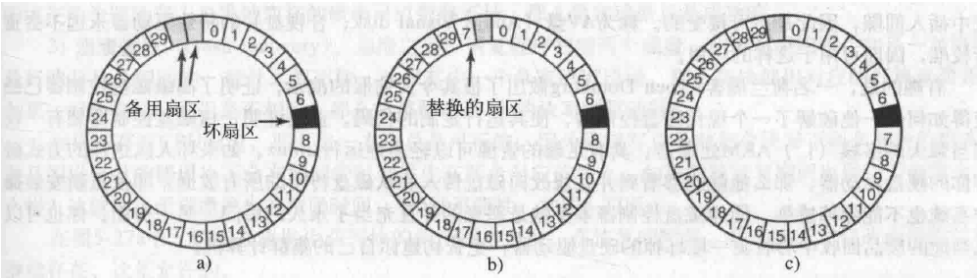


图5-26 a) 具有一个坏扇区的磁盘磁道；b) 用备用扇区替换坏扇区；c) 移动所有扇区以回避坏扇区