第三章　I/O系统

翁楚良

https://chuliangweng.github.io

2023 春　ECNU

第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

I/O硬件分类

- 按交互对象分类
  - 人机交互设备：视频显示设备、键盘、鼠标、打印机
  - 与计算机或其他电子设备交互的设备：磁盘、磁带、传感器、控制器
  - 计算机间的通信设备：网卡、调制解调器
- 按交互方向分类
  - 输入（可读）：键盘、扫描仪
  - 输出（可写）：显示设备、打印机
  - 输入/输出（可读写）：磁盘、网卡
- 按外设特性分类
  - 使用特征：存储、输入/输出、终端
  - 数据传输率：低速(如键盘)、中速(如打印机)、高速(如网卡、磁盘)
  - 信息组织特征：单个字符或数据块
    - 字符设备(如打印机)
    - 块设备(如磁盘)

I/O设备的特点

- 种类多
- 差异大(控制和速度)
  - 在速率相差多个数量级的不同设备上保持良好的性能，这对操作系统提出了很高的要求

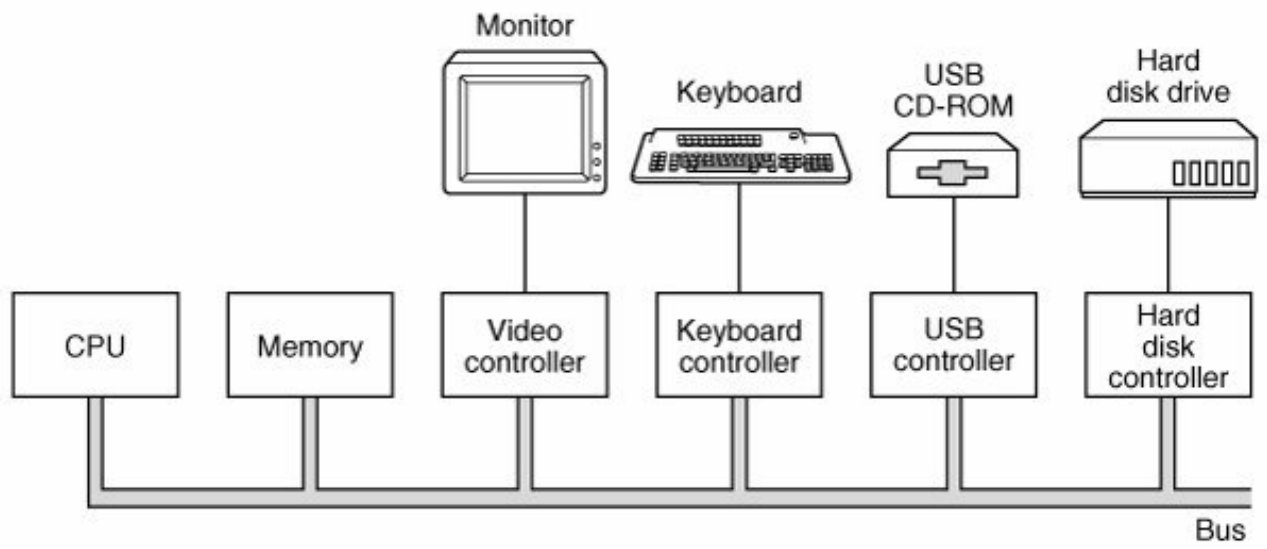
Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	4 MB/sec
52x CD-ROM	8 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
Gigabit Ethernet	125 MB/sec
Serial ATA disk	200 MB/sec
SCSI Ultrawide 4 disk	320 MB/sec
PCI bus	528 MB/sec

块设备和字符设备

- 块设备将信息存储在可寻址的固定大小的数据块中
  - 通常数据块大小的范围从512字节到32768字节不等
  - 块设备的主要特征是能够独立地读写单个的数据块
  - 磁盘是最常见的块设备
- 字符设备发送或接收的是字符流，而不是块结构
  - 字符设备无法编址，也不存在任何寻址操作
  - 如：打印机、网络接口、鼠标等

设备控制器

- I/O设备通常包含一个机械部件和一个电子部件。为了达到设计的模块性和通用性，一般将其分开。电子部分称为设备控制器或适配器。在个人计算机中，它常常是一块可以插入主板扩展槽的印刷电路板。机械部分则是设备本身。



设备控制器

- 控制器负责将驱动器读出来的比特流转换成字节块并在需要时进行纠错。
- 通常该字节块是在控制器中的一个缓冲区中逐个比特汇集而成。在对检查和进行校验证实数据正确之后，该块数据随后被拷贝到主存中。

I/O控制技术

- 程序控制I/O
- 中断驱动方式
- 直接存储访问方式
- 通道控制方式

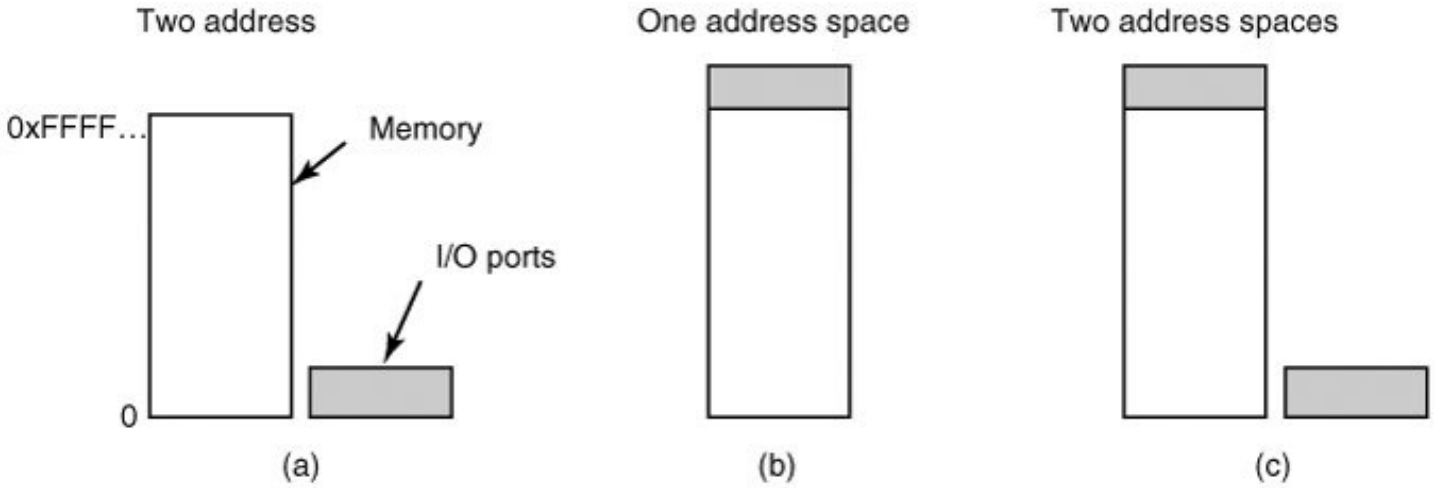
程序控制I/O

- I/O操作由程序发起，并等待操作完成。数据的每次读写通过CPU。
- 每个控制器都有一些用来与CPU通信的寄存器及数据缓冲区。
  - 通过写入寄存器，操作系统可以命令设备发送数据、接收数据、开启或关闭、或其它操作
  - 通常设备有一个数据缓冲区，以供操作系统读写数据

程序控制I/O

- 设备编址方式
  - I/O端口
    - 设备寄存器单独编址，即I/O端口号
  - 内存映射I/O
    - 设备数据缓冲区按内存地址空间进行统一编址

IN REG, PORT
OUT PORT, REG



IBM PC兼容机：0-64KB作为I/O端口；640KB-1MB保留给数据缓冲区

I/O控制技术

- 程序控制I/O
- 中断驱动方式
- 直接存储访问方式
- 通道控制方式

中断驱动方式

- I/O操作是否可以开始或完成
  - 检测设备控制寄存器中的状态标志位
  - 使用中断方式通知CPU
- 中断驱动方式
  - I/O操作由程序发起，在操作完成时（如数据可读或已经写入）由外设向CPU发出中断，通知该程序。数据的每次读写通过CPU。
  - 优点
    - 在外设进行数据处理时，CPU不必等待，可以继续执行该程序或其他程序。
  - 缺点
    - CPU每次处理的数据量少（通常不超过几个字节），只适于数据传输率较低的设备。

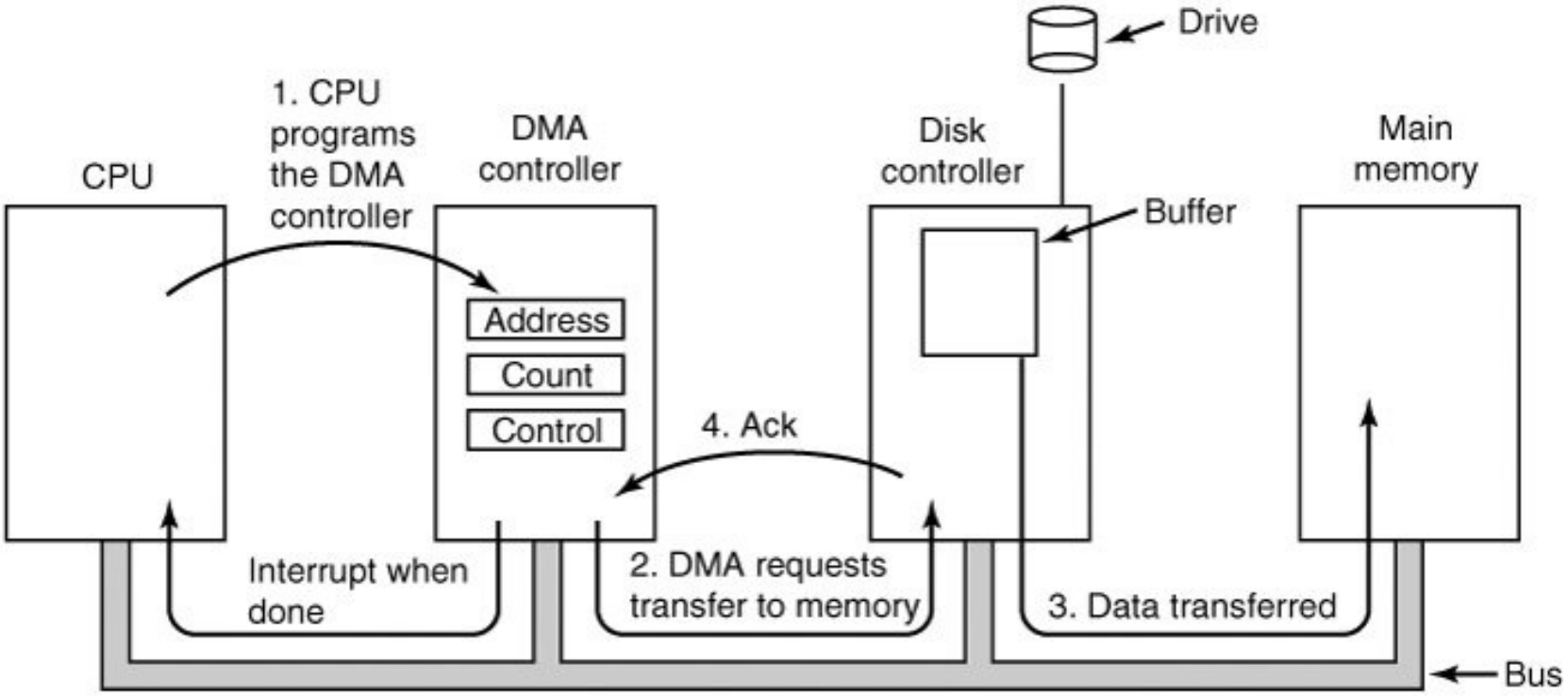
I/O控制技术

- 程序控制I/O
- 中断驱动方式
- 直接存储访问方式
- 通道控制方式

直接存储访问方式

- 由程序设置DMA(Direct Memory Access)控制器中的若干寄存器值（如内存始址，传送字节数），然后发起I/O操作，而后者完成内存与外设的成批数据交换，在操作完成时由DMA控制器向CPU发出中断。
- 优点
  - CPU只需干预I/O操作的开始和结束，而其中的一批数据读写无需CPU控制，适于高速设备。

DMA传送操作

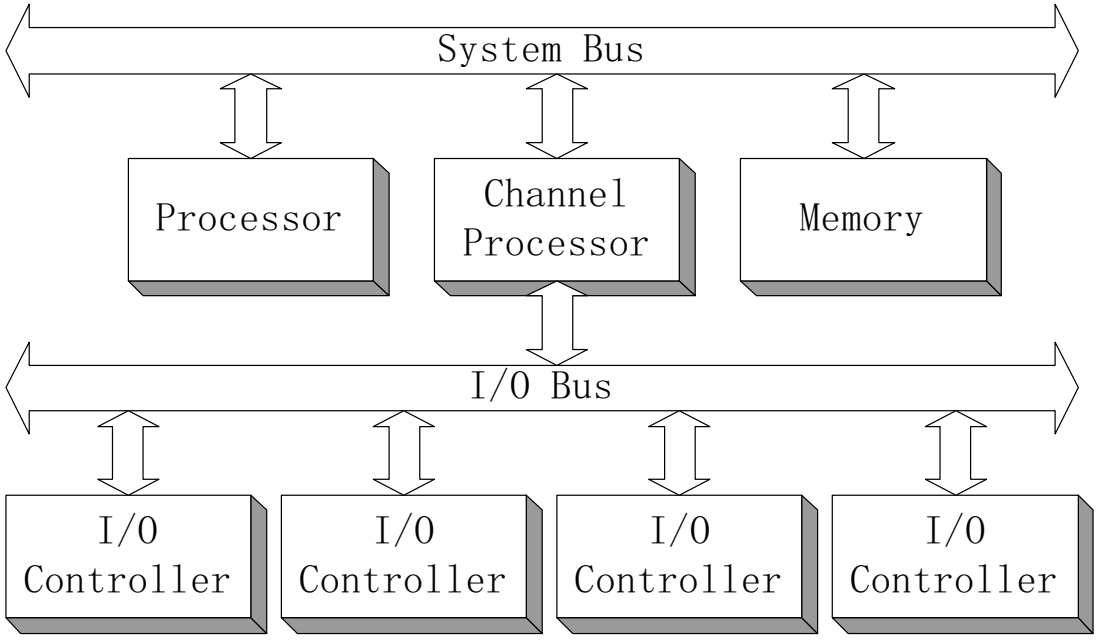

Drive
1. CPU programs the DMA controller
DMA controller
Disk controller
Main memory
CPU
Buffer
Address
Count
Control
4. Ack
Interrupt when done
2. DMA requests transfer to memory
3. Data transferred
Bus

I/O控制技术

- 程序控制I/O
- 中断驱动方式
- 直接存储访问方式
- 通道控制方式

通道控制方式

- 通道控制器(Channel Processor)有自己的专用存储器，可以执行由通道指令组成的通道程序，因此可以进行较为复杂的I/O控制。
- 通道程序通常由操作系统所构造，放在内存里。
- 优点：执行一个通道程序可以完成几批I/O操作



选择通道(selector channel)：可以连接多个外设，而一次只能访问其中一个外设

多路通道(multiplexor channel)：可以并发访问多个外设。分为字节多路(byte)和数组多路(block)通道。

第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

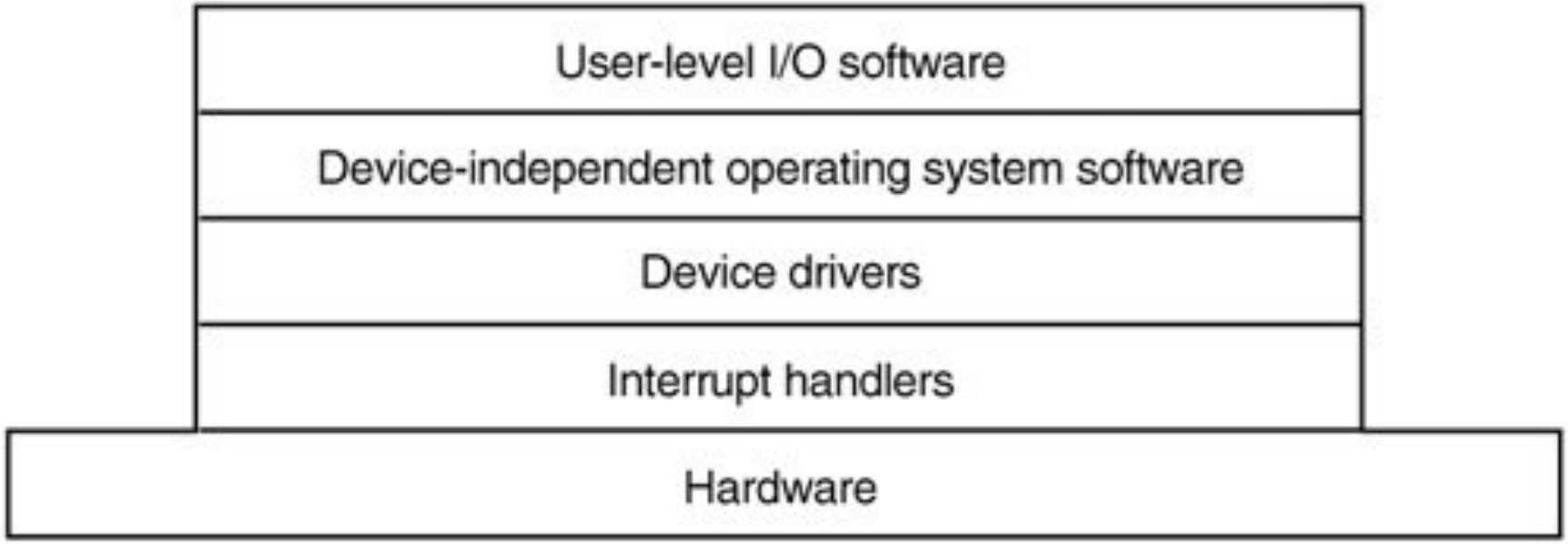
I/O软件原理

- I/O软件目标
- 中断处理器
- 设备驱动程序
- 设备无关I/O软件
- 用户空间I/O软件

I/O软件目标

- 设备无关性
  - 程序员写出的软件无需修改便能读出软盘、硬盘以及CD-ROM等不同设备上的文件
- 统一的命名
  - 一个文件或设备名将简单地只是一个字符串或一个整数，而完全不依赖于设备。
  - 在UNIX和MINIX 3中，所有的磁盘可以以任何方式集成到文件系统层次结构中去，用户也不必知道哪个各字对应着哪个设备。
- 容错功能
  - 错误应在尽可能接近硬件的地方处理，低层软件可以自行处理错误，尽可能向上层软件透明
- 协同同步与异步传输
  - 多数物理I/O是异步传输，用户接口是阻塞的，需要使之协同
- 设备共享
  - 某些设备可同时被多个用户使用，另一些设备则在某一时刻只能供一个用户专用

I/O软件系统的层次结构



I/O软件原理

- I/O软件目标
- 中断处理器
- 设备驱动程序
- 设备无关I/O软件
- 用户空间I/O软件

中断处理器

- 中断需要尽量加以屏蔽，需将其放在操作系统的底层进行处理，以便其余部分尽可能少地与之发生联系。
  - 屏蔽中断的最好方法是将每一个进行I/O操作的进程挂起，直至I/O操作结束并发生中断。
  - 进程自己阻塞的方法有：执行信号量的DOWN操作、条件变量的WAIT操作；或接收一条消息等等。
- 当中断发生时，中断处理程序执行相应的操作，然后解除相应进程的阻塞状态。
  - 一些系统中是执行信号量的UP操作，在管程中可能是对一个条件变量执行SIGNAL操作，另一些系统则可能是向阻塞的进程发一条消息，总之其作用是将刚才被阻塞的进程恢复执行。

I/O软件原理

- I/O软件目标
- 中断处理器
- 设备驱动程序
- 设备无关I/O软件
- 用户空间I/O软件

设备驱动程序

- 设备驱动程序中包括了所有与设备相关的代码。每个设备驱动程序只处理一种设备，或者一类紧密相关的设备
- 设备驱动程序的功能是从与设备无关I/O软件中接收抽象的请求，并负责执行该请求。
- 设备驱动程序放在系统内核中，可以获得良好的性能，但会影响系统的可靠性；MINIX3将其作为用户模式的进程，以提高其可靠性

驱动程序的工作流程

- 执行一条I/O请求的第一步，是将它转换为更具体的形式。
  - 例如对磁盘驱动程序，它包含：计算出所请求块的物理地址、检查驱动器电机是否在运转、检测磁头臂是否定位在正确的柱面等等。
  - 简言之，它必须确定需要哪些控制器命令以及命令的执行次序。
- 一旦决定应向控制器发送什么命令，驱动程序将向控制器的设备寄存器中写入这些命令。
  - 某些控制器一次只能处理一条命令，另一些则可以接收一串命令并自动进行处理。
- 这些控制命令发出后，存在两种情况
  - 驱动程序需等待控制器完成一些操作，所以驱动程序阻塞，直到中断信号到达才解除阻塞。
  - 另一种情况是操作没有任何延迟，所以驱动程序无需阻塞。
    - 例如：在有些终端上滚动屏幕只需往控制器寄存器中写入几个字节，无需任何机械操作，所以整个操作可在几微秒内完成。

I/O软件原理

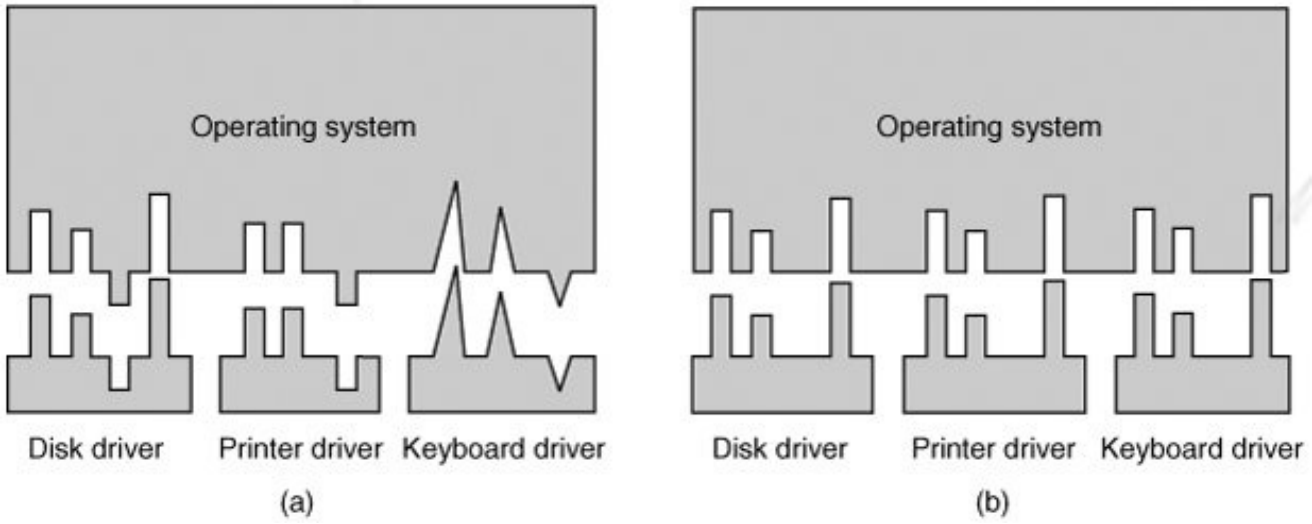
- I/O软件目标
- 中断处理器
- 设备驱动程序
- 设备无关I/O软件
- 用户空间I/O软件

设备无关I/O软件

- 设备无关软件的基本功能是执行适用于所有设备的常用I/O功能，并向用户层软件提供一个一致的接口。
  - 设备无关软件和设备驱动程序之间的精确界限在各个系统都不尽相同。对于一些以设备无关方式完成的功能，在实际中由于考虑到执行效率等因素，也可以考虑由驱动程序完成。
- 设备无关I/O软件的功能包括
  - 设备驱动程序的统一接口
  - 缓冲
  - 错误报告
  - 分配和释放专用设备
  - 提供与设备无关的块大小

设备驱动程序的统一接口

- I/O设备和驱动程序的对外接口需要尽可能的相同
- 在标准接口情况下，可以方便地加入新的驱动程序



设备驱动程序的统一接口

- 设备无关I/O软件负责将设备名映射到相应的驱动程序
  - 在UNIX中，一个设备名，如/dec/tty00 唯一地确定了一个i-节点，其中包含了主设备号（major device number），通过主设备号就可以找到相应的设备驱动程序。i-节点也包含了次设备号（minor device number），它作为传给驱动程序的参数指定具体的物理设备。
- 设备作为命名对象出现在文件系统中，通过对文件的常规保护规则，确保对设备的访问权限。

缓冲

- 对于块设备，硬件每次读写均以块为单元，而用户程序则可以读写任意大小的单元。如果用户进程写半个块，操作系统将在内部保留这些数据，直到其余数据到齐后才一次性地将这些数据写到盘上。
- 对字符设备，用户向系统写数据的速度可能比向设备输出的速度快，所以需要进行缓冲。超前的键盘输入同样也需要缓冲。

错误报告

- 错误处理多数由驱动程序完成，多数错误是与设备紧密相关的，因此只有驱动程序知道应如何处理（如重试、忽略、严重错误）。
  - 一种典型错误是磁盘块受损导致不能读写。驱动程序在尝试若干次读操作不成功后将放弃，并向设备无关软件报错。从此处往后错误处理就与设备无关了。
    - 如果在读一个用户文件时出错，则向调用者报错即可。
    - 如果是在读一些关键系统数据结构时出错，比如磁盘使用状况位图，则操作系统只能打印出错信息，并终止运行。

分配和释放专用设备

- 一些设备，如CD-ROM记录器，在同一时刻只能由一个进程使用。
- 这要求操作系统检查对该设备的使用请求，并根据设备的忙闲状况来决定是接受或拒绝此请求。
  - 一种简单的处理方法是通过直接用OPEN打开相应的设备文件来进行申请。若设备不可用，则OPEN失败。关闭独占设备的同时将释放该设备。

提供与设备无关的块大小

- 不同磁盘的扇区大小可能不同，设备无关软件屏蔽了这一事实并向高层软件提供统一的数据块大小
  - 如将若干扇区作为一个逻辑块，这样上层软件只和逻辑块大小相同的抽象设备交互，而不管物理扇区的大小。
  - 如有些字符设备对字节进行操作(Modem)，另一些则使用比字节大一些的单元（网卡），这类差别也可以进行屏蔽。

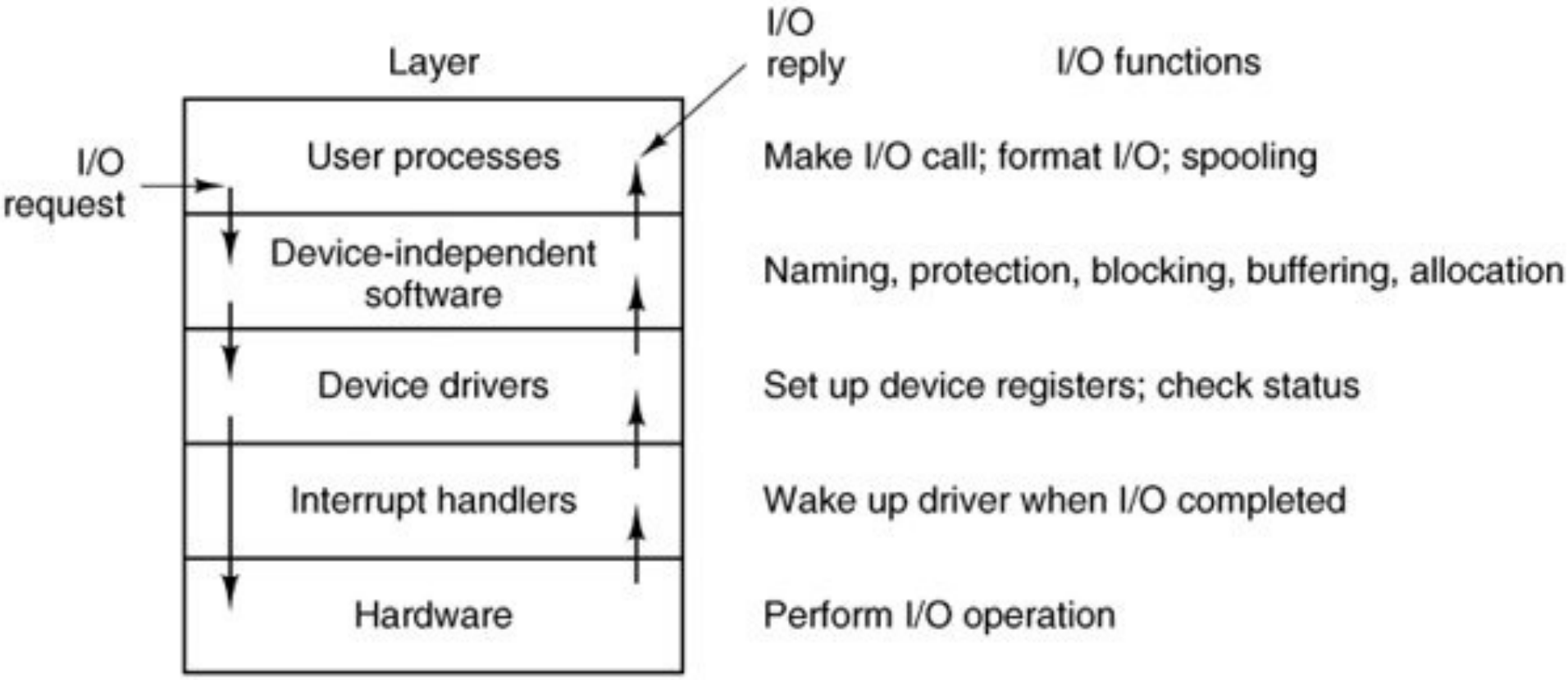
I/O软件原理

- I/O软件目标
- 中断处理器
- 设备驱动程序
- 设备无关I/O软件
- 用户空间I/O软件

用户层I/O软件

- I/O相关的库例程
  - 主要工作是提供参数给相应的系统调用并调用之
- 假脱机系统(spooling)
  - Spooling是在多道程序系统中处理专用I/O设备的一种方法
    - 创建一个特殊的守护进程daemon以及一个特殊的目录，称为 Spooling 目录。
    - 打印一个文件之前，进程首先产生完整的待打印文件并将其放在Spooling目录下。
    - 由该守护进程进行打印，只有该守护进程能够使用打印机设备文件。

I/O系统的层次结构及功能



第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

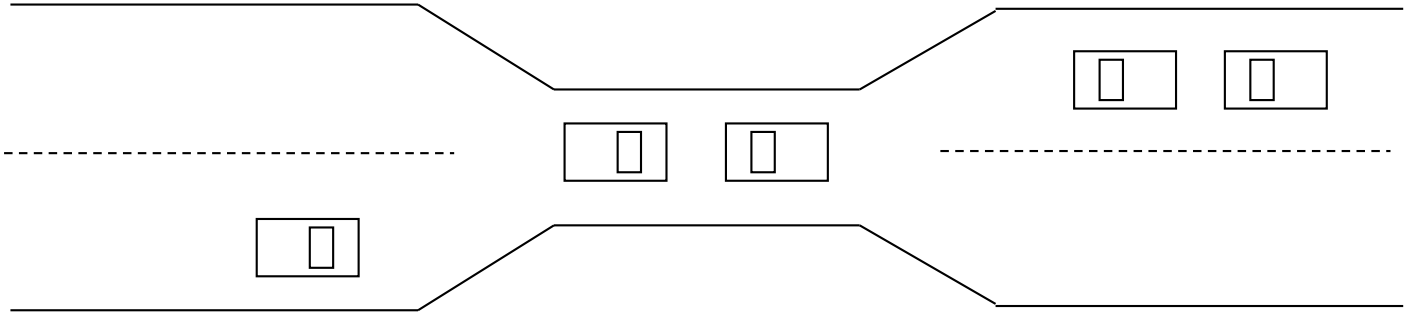
概述

- 死锁是指系统中多个进程无限制地等待永远不会发生的条件
- Example
  - System has 2 tape drives.
  - P0 and P1 each hold 1 tape drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

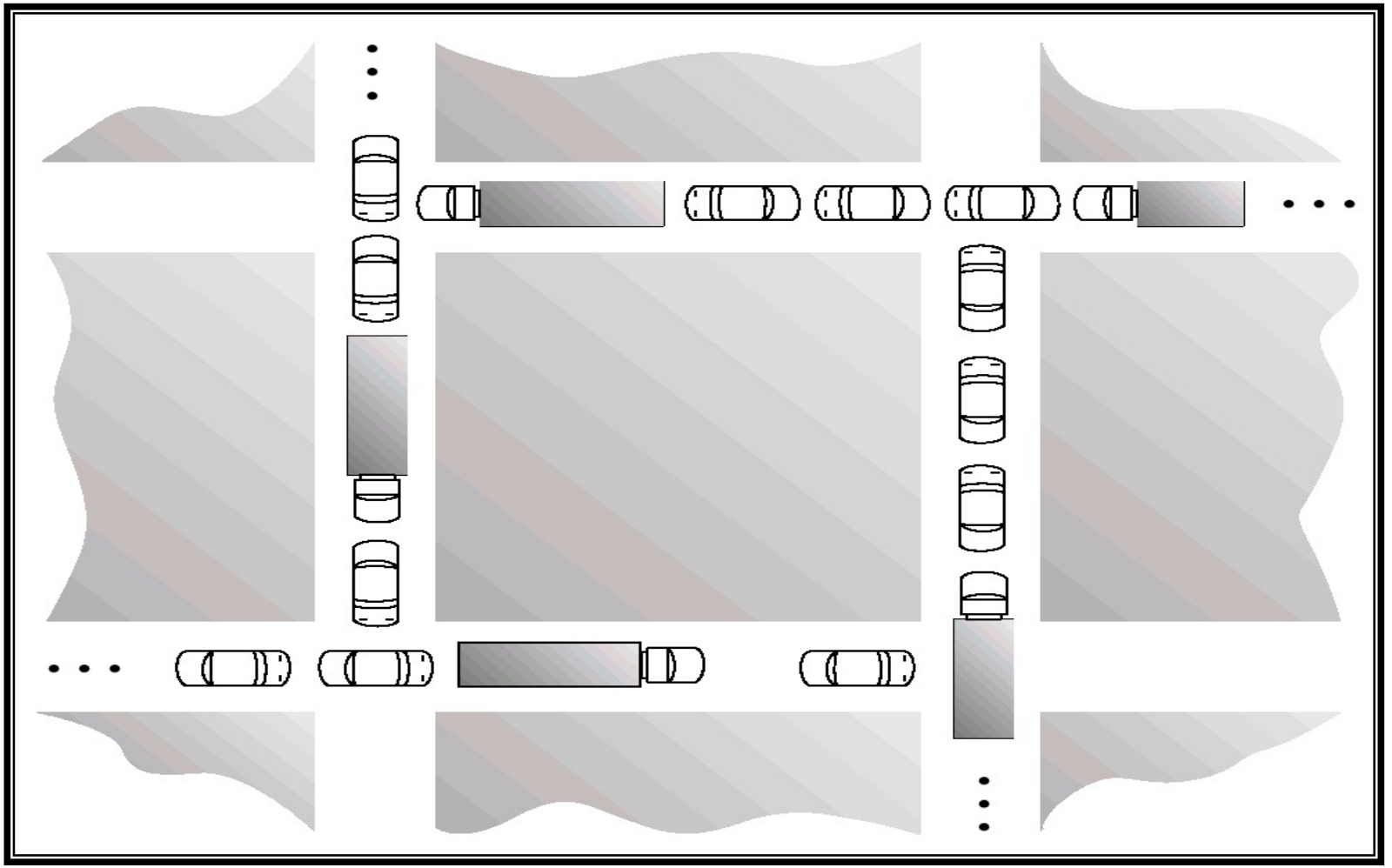
P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

Bridge Crossing Example

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.



Traffic Gridlock/Deadlock

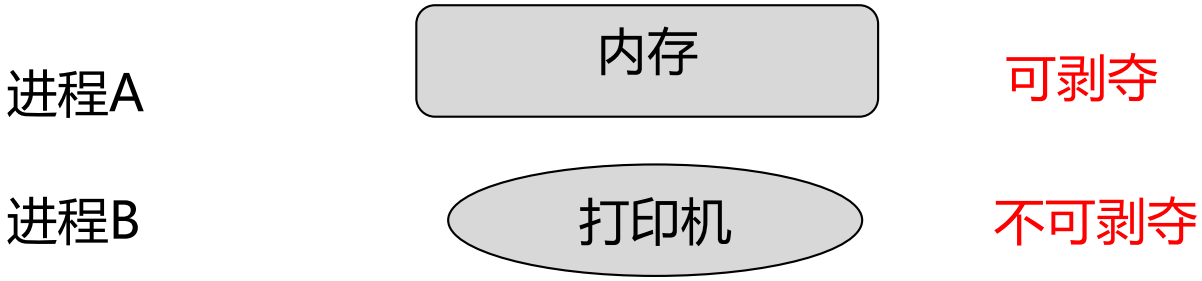


死锁

- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

资源

- 进程对设备、文件等获得独占性的访问权时有可能会发生死锁，为了尽可能地通用化，将这种需排它使用的对象称为资源。
- 资源有两类：可剥夺式和不可剥夺式
  - 可剥夺式资源可从拥有它的进程处剥夺而没有任何副作用，存储器是一类可剥夺资源。
  - 不可剥夺资源无法在不导致相关计算失败的情况下将其从属主进程处剥夺



死锁

- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

System Model

- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release
- deadlock: A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

死锁的必要条件

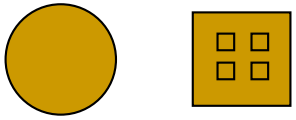
- 死锁发生的必要条件
  - 互斥：任一时刻只允许一个进程使用资源
  - 请求和保持：进程在请求其余资源时，不主动释放已经占用的资源
  - 非剥夺：进程已经占用的资源，不会被强制剥夺
  - 环路等待：环路中的每一条边是进程在请求另一进程已经占有的资源。

Resource-Allocation Graph (1)

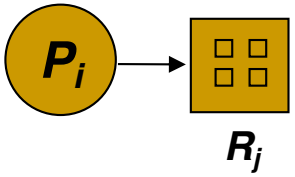
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (2)

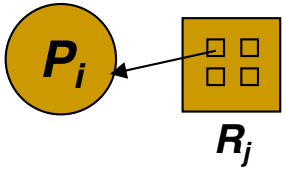
- Process



Resource Type with 4 instances

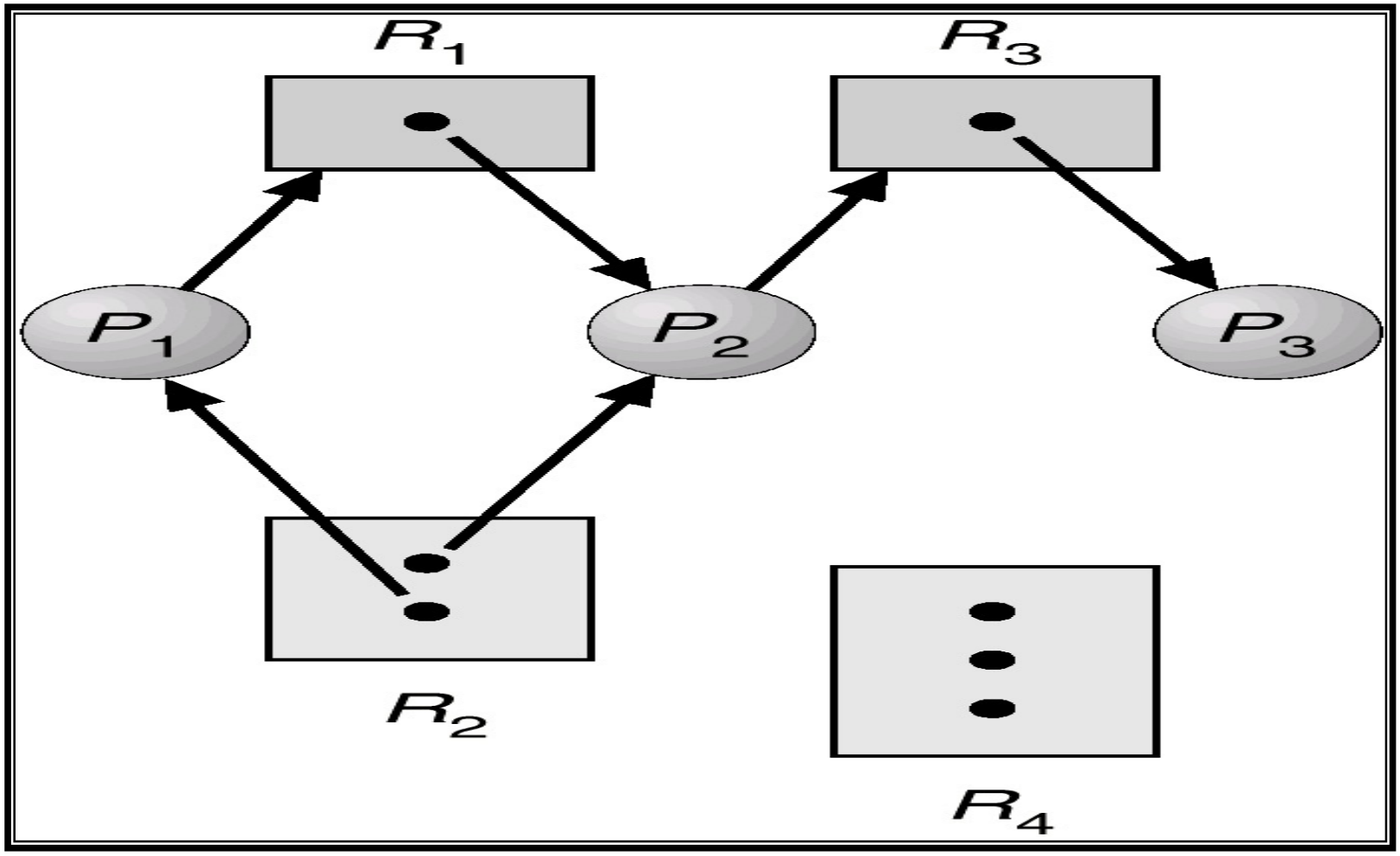


- P_i requests instance of R_j

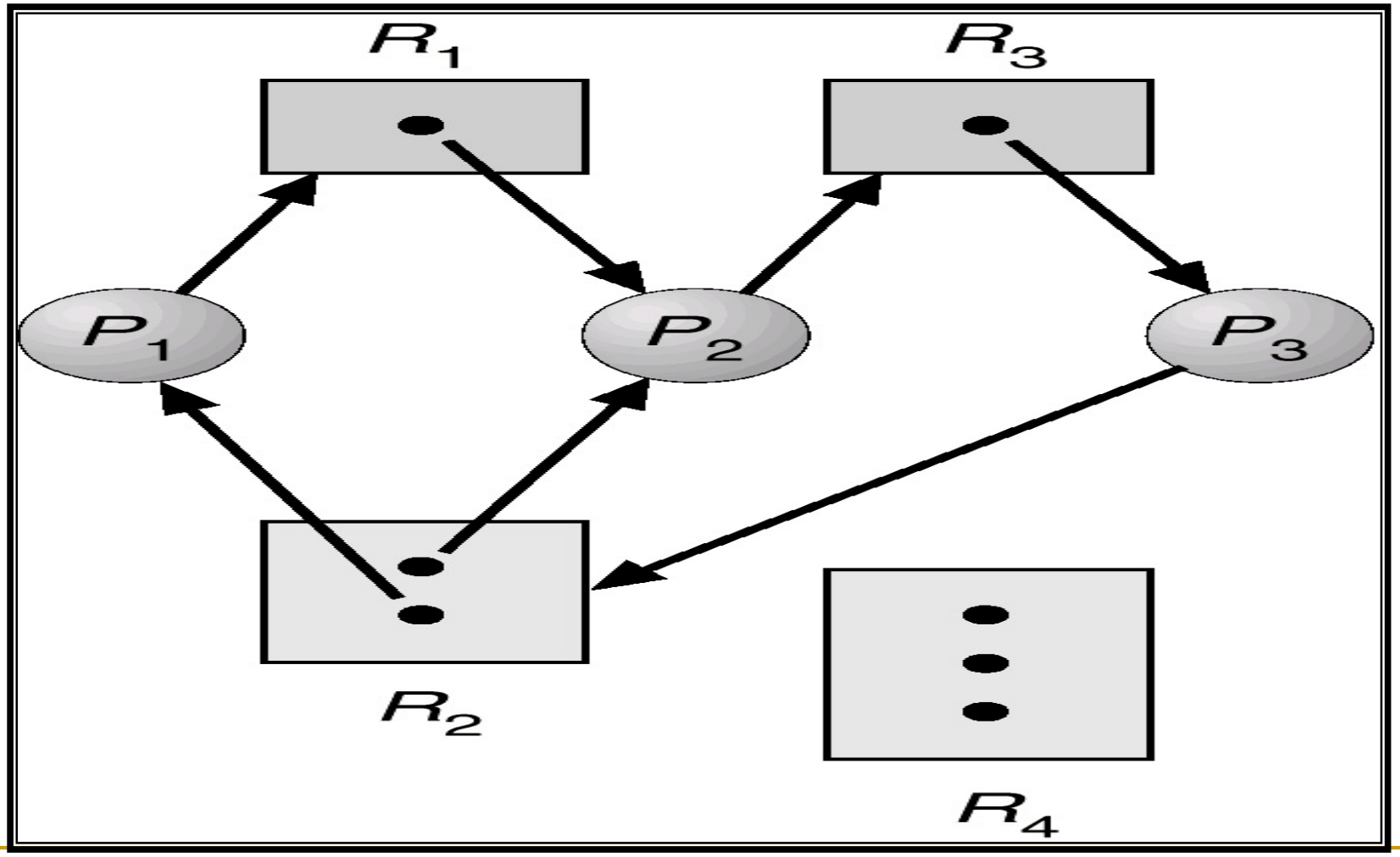


- P_i is holding an instance of R_j

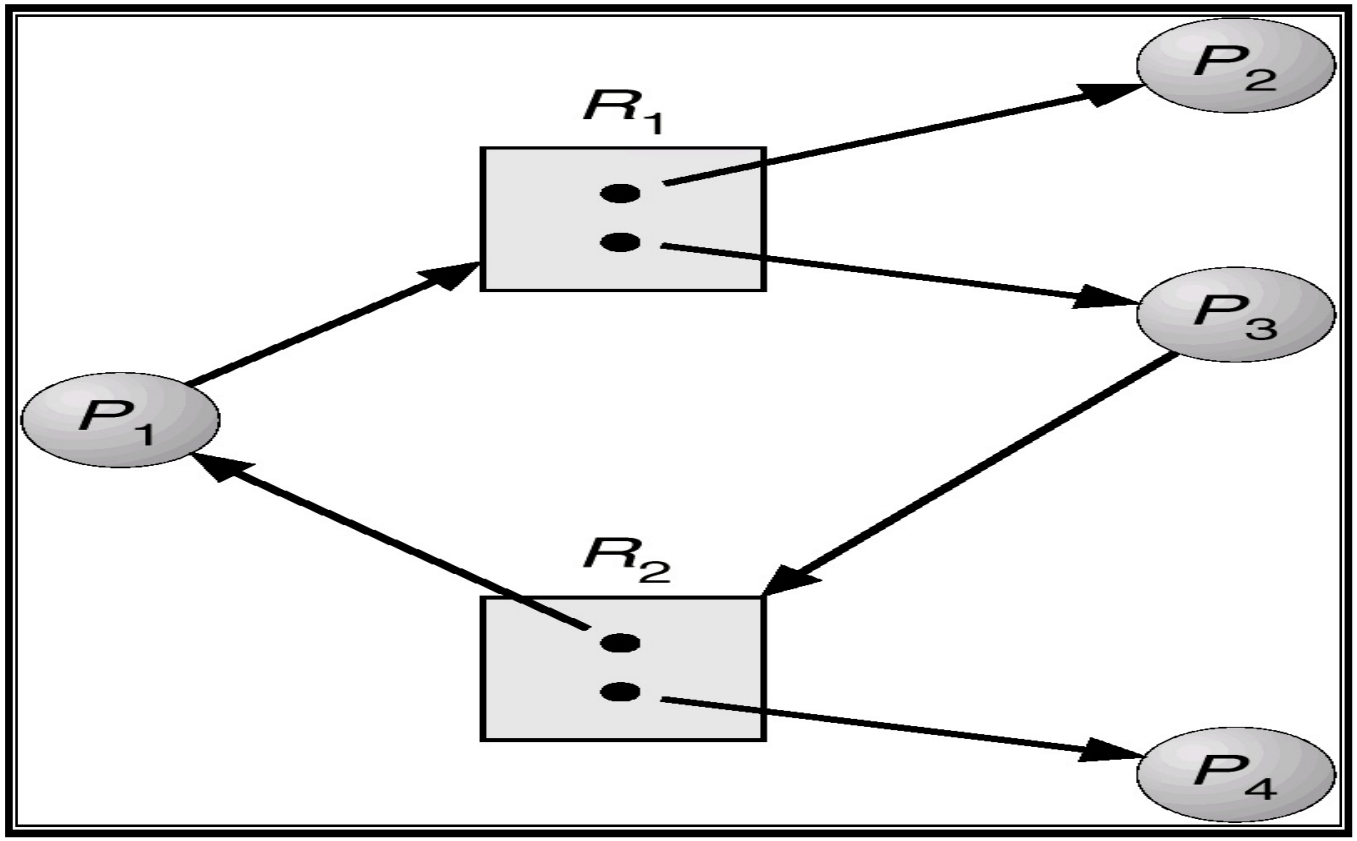
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph with a cycle but no deadlock



Basic Facts

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

处理死锁的有效方法

方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的；宁可资源闲置（从机制上使死锁条件不成立）	一次请求所有资源<条件 2>	适用于作突发式处理的进程；不必剥夺	效率低；进程初始化时间延长 剥夺次数过多；多次对资源重新起动 不便灵活申请新资源
		资源剥夺<条件 3>	适用于状态可以保存和恢复的资源	
		资源按序申请<条件 4>	可以在编译时（而不必在运行时）就进行检查	
避免 Avoidance	是“预防”和“检测”的折衷（在运行时判断是否可能死锁）	寻找可能的安全的运行顺序	不必进行剥夺	使用条件：必须知道将来的资源需求；进程可能会长时间阻塞
检测 Detection	宽松的；只要允许，就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间；允许对死锁进行现场处理	通过剥夺解除死锁，造成损失

死锁

- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

鸵鸟算法

- 最简单的方法是象鸵鸟一样对死锁视而不见。
- 对该方法各人的看法不同。
  - 数学家认为不管花多大代价也要彻底防止死锁的发生
  - 工程师们则要了解死锁发生的频率、系统因其他原因崩溃的频率、以及死锁有多严重
    - 如果死锁平均每5年发生一次，而系统每个月会因硬件故障、编译器错误或操作系统错误而崩溃一次，那么大多数工程师不会不惜工本地去消除死锁。

死锁

- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

死锁检测与恢复

- 保存资源的请求和分配信息，利用某种算法对这些信息加以检查，以判断是否存在死锁。死锁检测算法主要是检查是否有循环等待。
- 死锁的恢复
  - 通过撤消代价最小的进程，以解除死锁。
  - 挂起某些死锁进程，并抢占它的资源，以解除死锁。
- 撤消进程的原则
  - 进程优先级
  - 系统会计过程给出的运行代价

死锁

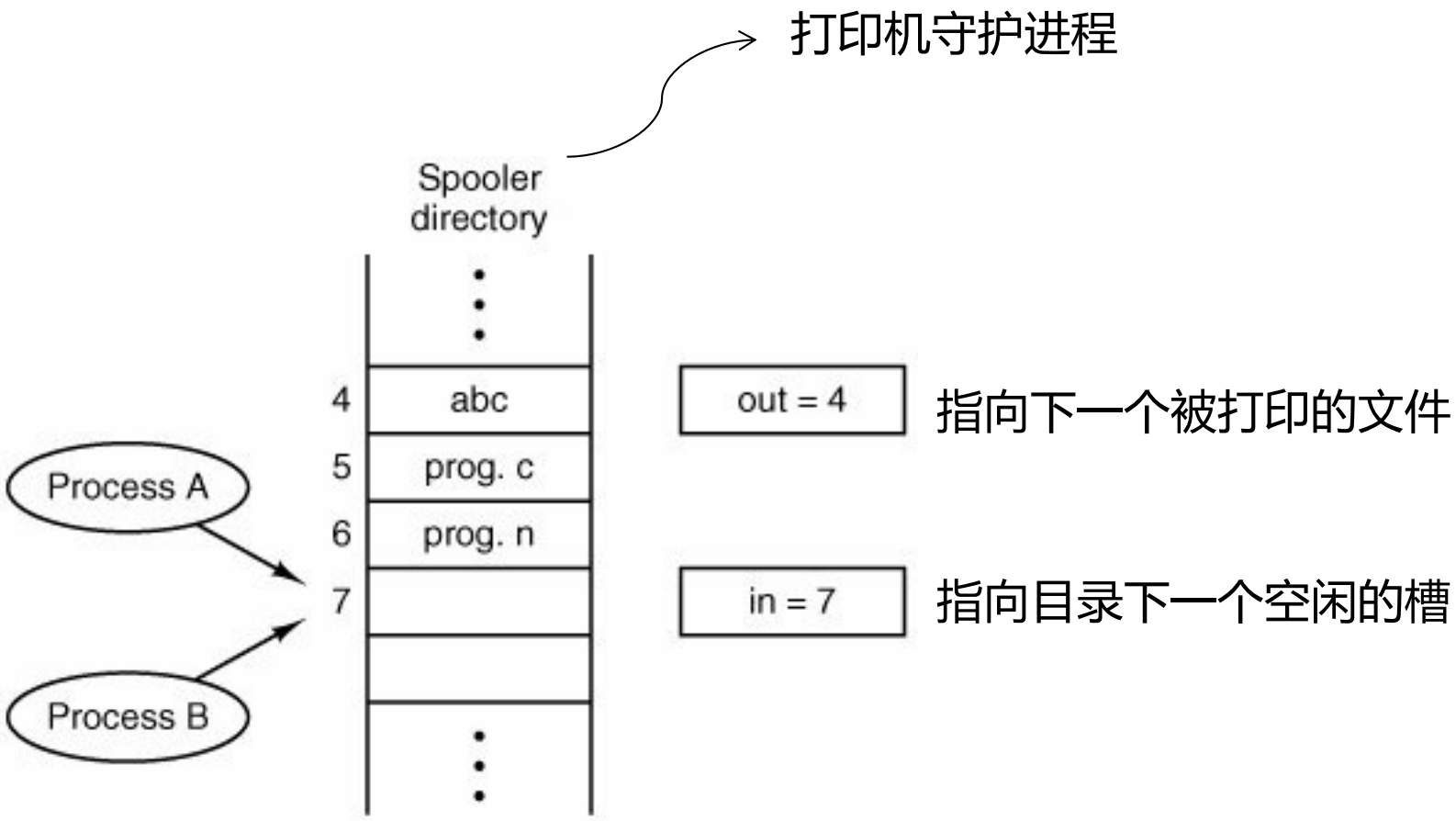
- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

死锁预防

- 对进程施加适当的限制以从根本上消除死锁
  - 使死锁发生的四个必要条件至少有一个不成立，则死锁将不会发生

条件	方法
互斥	对所有资源进行spooling
保持并等待	初始时申请所有资源
不可剥夺	将资源剥夺
循环等待	对资源进行编号

假脱机打印

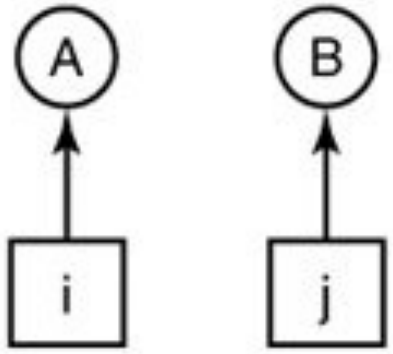

打印机守护进程
Spooler directory
4
abc
5
prog. c
6
prog. n
7
Process A
Process B
out = 4
指向下一个被打印的文件
in = 7
指向目录下一个空闲的槽

全局编号

- 所有资源赋予一个全局编号，进程申请资源必须按照编号顺序
  - 进程可以先申请扫描仪，后申请磁带机，但不可以先申请绘图仪，后申请扫描仪
- 改进：不允许进程申请编号比当前所占有资源编号低的资源
  - 若一个进程起初申请9号和10号资源，随后将其释放，它实际上相当于从头开始，所以没有必要阻止它现在申请1号资源。

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive

(a)



(b)

死锁

- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

单种资源的银行家算法

- 基本思想
  - 一个小城镇的银行家向一群客户分别承诺了一定的贷款额度，考虑到所有客户不会同时申请最大额度贷款，他只保留较少单位的资金来为客户服务
  - 将客户比作进程，贷款比作设备，银行家比作操作系统
- 算法
  - 对每一个请求进行检查，检查如果满足它是否会导致不安全状态。若是，则不满足该请求；否则便满足。
  - 检查状态是否安全的方法是看他是否有足够的资源满足某一客户。如果可以，则这笔投资认为是能够收回的，然后检查最接近最大限额的客户，如此反复下去。如果所有投资最终都被收回，则该状态是安全的，最初的请求可以批准。

例子

- 三种资源分配状态
  - (a) 安全
  - (b) 安全
  - (c) 不安全

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

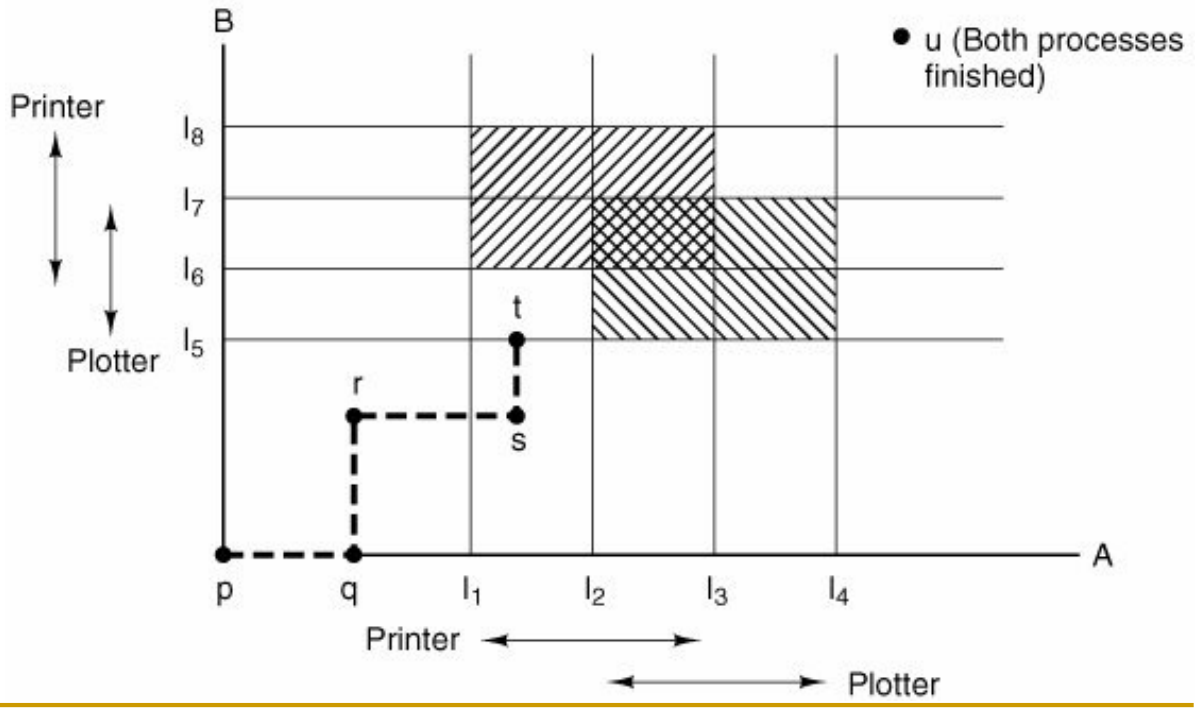
(c)

资源轨迹图

- 处理两个进程和两种资源（打印机和绘图仪）
- 横轴表示进程A的指令执行过程，纵轴表示进程B的指令执行过程。
- 进程A在I_1处请求一台打印机，在I_3处释放，在I_2处申请一台绘图仪，在I_4处释放。进程B在I_5到I_7之间需要绘图仪，在I_6到I_8之间需要打印机。

如果系统一旦进入由I_1、I_2和I_5、I_6组成的矩形区域，那么最后一定会到达I_2和I_6的交叉点，此时就发生死锁。

在t处唯一的办法是运行进程A直到I_4，过了I_4后则可以按任何路线前进，直到终点u。



多种资源的银行家算法

- 资源轨迹图的方法很被难被扩充到系统中有任意数目的进程、任意种类的资源，并且每种资源有多个实例的情况，但银行家算法可以被推广用来处理这个问题
- 两个矩阵
  - 左边的显示出对5个进程分别已分配的各种资源数
  - 右边的则显示了使各进程运行完所需的各种资源数。
- 三个向量分别表示总的资源E、已分配资源P，和剩余资源A

目前的状态是安全的。

假设进程B现在在申请一台打印机，可以满足它的请求，而且保持系统状态仍然是安全的（进程D可以结束，然后是A或E，剩下的进程最后结束）。

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

多种资源的银行家算法

- 检查一个状态是否安全的步骤如下：
  - 查找右边矩阵中是否有一行，其未被满足的设备数均小于或等于向量A。如果找不到，则系统将死锁，因为任何进程都无法运行结束。
  - 若找到这样一行，则可以假设它获得所需的资源并运行结束，将该进程标记为结束，并将资源加到向量A上。
  - 重复以上两步，直到所有的进程都标记为结束。若达到所有进程结束，则状态是安全的，否则将发生死锁。
  - 如果在第1步中同时存在若干进程均符合条件，则不管挑选哪一个运行都没有关系，因为可用资源或者将增多，或者在最坏情况下保持不变。

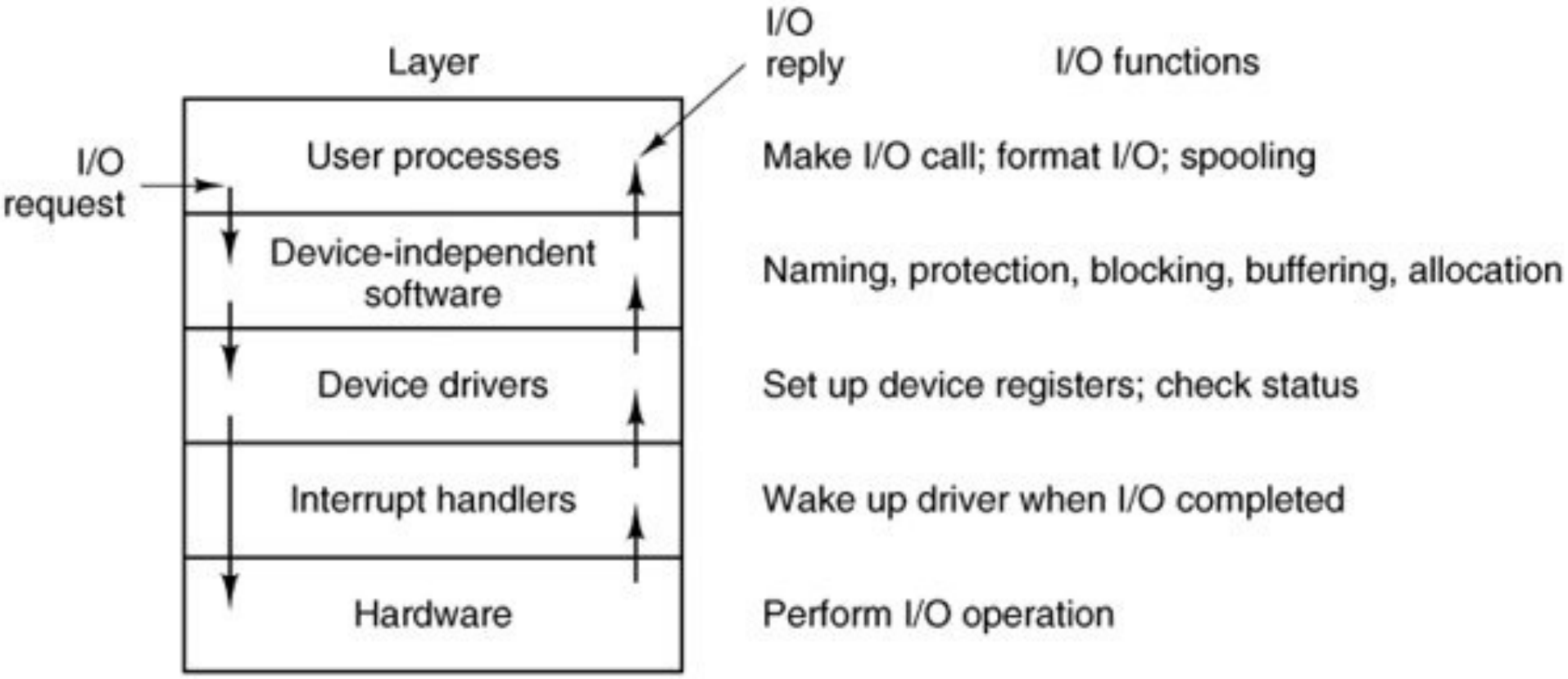
两阶段加锁法

- 死锁预防的方案过于严格，死锁避免的算法又需要无法得到的信息
- 实际情况采取相应的算法
  - 例如在许多数据库系统中，常常需要将若干记录上锁然后进行更新。当有多个进程同时运行时，有可能发生死锁。常用的一种解法是两阶段加锁法。
    - 第一阶段，进程试图将其所需的全部记录加锁，一次锁一个记录。
    - 若成功，则开始第二阶段，完成更新数据并释放锁。
    - 若有些记录已被上锁，则它将已上锁的记录解锁并重新开始第一阶段

第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

MINIX3 I/O结构


Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

3.4 MINIX3 I/O概述

- MINIX3中断处理器和I/O访问
- MINIX3设备驱动程序
- MINIX3设备无关IO软件
- MINIX3用户级I/O软件
- MINIX3死锁处理

MINIX3中断处理器和I/O访问

- 时钟中断
  - 无须每次时钟中断都向时钟任务发送消息，而是设定一个计算器，定期向时钟任务发送消息
- 用户空间的设备驱动程序不同层次的I/O访问及中断处理
  - 驱动程序访问正常数据空间以外的内存
  - 驱动程序读写I/O端口
  - 驱动程序响应预期的中断
  - 驱动程序响应不可预测的中断
  - 由系统任务提供的内核调用实现，中断处理方式有所不同

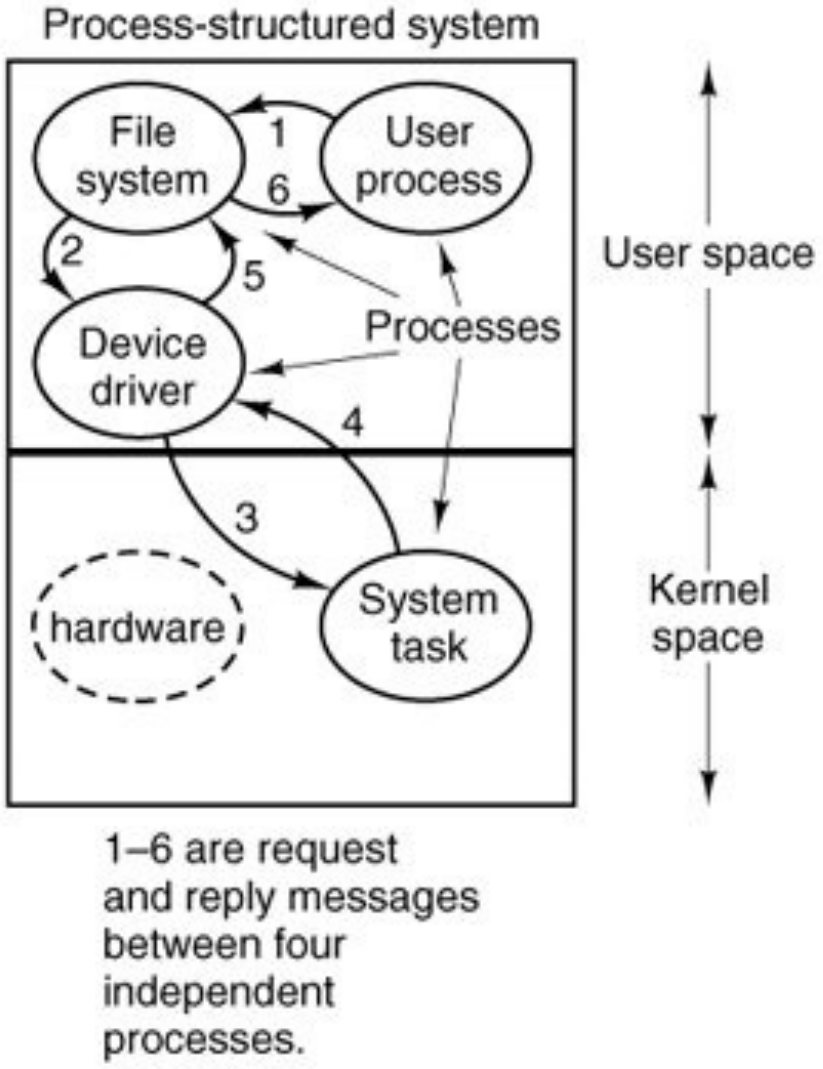
3.4 MINIX3 I/O概述

- MINIX3中断处理器和I/O访问
- MINIX3设备驱动程序
- MINIX3设备无关IO软件
- MINIX3用户级I/O软件
- MINIX3死锁处理

MINIX3设备驱动程序

- MINIX中的每一类设备都有一个单独的I/O设备驱动程序
  - 这些驱动程序是完整的进程，每个都有其自己的状态、寄存器、堆栈等。
  - 设备驱动程序采用消息传递机制与文件系统进行通信
- 驱动程序分为两部分
  - 设备相关部分
    - RAM、硬盘和软盘有各自的设备相关代码
    - 键盘、串口线、虚拟终端等分别有各自的设备相关代码
  - 设备无关部分
    - 支持所有块设备类型的通用例程放在driver.c和drvlib.c
    - 终端驱动程序设备的无关代码放在tty.c
- 驱动程序的运行
  - 不同类型块设备的驱动程序各自作为独立的进程运行
  - 单个进程支持所有终端设备

I/O流程


Process-structured system
File system
User process
1
6
2
5
Device driver
Processes
4
3
hardware
System task
User space
Kernel space
1–6 are request and reply messages between four independent processes.

驱动程序的消息结构

- 驱动程序与MINIX3系统中其它部分的交互
  - 将请求的消息发送给驱动程序
  - 驱动程序执行收到的请求，并返回应答

请求

域	类型	含义
m.m_type	int	请求的操作
m.DEVICE	int	使用的次设备
m.PROC_NR	int	请求I/O的进程
m.COUNT	int	字节计数或IOCTL码
m.POSITION	long	在设备上的位置
m.ADDRESS	char *	在请求进程中的地址

应答

域	类型	含义
m.m_type	int	总是TASK_REPLY
m.REP_PROC_NR	int	与请求消息中的PROC_NR相同
m.REP_STATUS	int	已传输的字节数或错误码

文件系统发送到块设备驱动程序的消息的各个域，以及应答消息的各个域

块设备驱动程序的主程序

```
message mess;                                   /* message buffer*/

void io_driver() {
  initialize();                                 /* only done once, during system init.*/
  while (TRUE)  {
        receive(ANY, &mess);                    /* wait for a request for work*/
        caller = mess.source;                   /* process from whom message came*/
        switch(mess.type) {
            case READ:      rcode = dev_read(&mess); break;
            case WRITE:     rcode = dev_write(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL*/
             default:       rcode = ERROR;
        }
        mess.type = DRIVER_REPLY;
        mess.status = rcode;                    /* result code*/
        send(caller,&mess);                     /* send reply message back to caller*/
  }
}
```

3.4 MINIX3 I/O概述

- MINIX3中断处理器和I/O访问
- MINIX3设备驱动程序
- MINIX3设备无关IO软件
- MINIX3用户级I/O软件
- MINIX3死锁处理

MINIX3设备无关IO软件

- 所有与设备无关的代码均包含在文件系统进程中。
  - 处理与驱动程序、缓冲和数据块分配等
  - 共它文件系统的保护和管理等。

3.4 MINIX3 I/O概述

- MINIX3中断处理器和I/O访问
- MINIX3设备驱动程序
- MINIX3设备无关IO软件
- MINIX3用户级I/O软件
- MINIX3死锁处理

MINIX3用户级I/O软件

- I/O相关的库例程
  - 主要工作是提供参数给相应的系统调用并调用之
- 假脱机系统(spooling)
  - Spooling是在多道程序系统中处理专用I/O设备的一种方法
    - 守护进程lpd处理打印文件
    - 通过lp命令提交打印文件

3.4 MINIX3 I/O概述

- MINIX3中断处理器和I/O访问
- MINIX3设备驱动程序
- MINIX3设备无关IO软件
- MINIX3用户级I/O软件
- MINIX3死锁处理

MINIX3死锁处理

- MINIX继承了UNIX的死锁处理办法：仅仅简单地忽略它。
- 对消息及共享资源加入了一些限制，以有效降低死锁的发生。
  - 消息

第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

MINIX3块设备驱动概述

- 驱动工作流程
  - 初始化
    - RAM盘驱动程序要预留一些内存空间、硬盘驱动程序要确定硬盘参数等等。
  - 调用一个公共主循环的函数
    - 该循环将一直执行下去，不会返回到调用者
    - 主 循环执行的工作是：接收一条消息，执行相应的操作，然后发回一条应答消息

使用间接调用的I/O驱动程序主过程

```
message mess;                                   /* message buffer*/

void shared_io_driver(struct driver_table *entry_points){
/* initialization is done by each driver before calling this */
   while (TRUE) {
        receive(ANY, &mess);
        caller = mess.source;
        switch(mess.type) {
            case READ:      rcode = (*entry_points->dev_read)(&mess); break;
            case WRITE:     rcode = (*entry_points->dev_write)(&mess); break;
            /* Other cases go here, including OPEN, CLOSE, and IOCTL */
            default:        rcode = ERROR;
        }
        mess.type = DRIVER_REPLY;
        mess.status = rcode;            /* result code* /
        send(caller, &mess);
  }
}
```

驱动程序的操作

- READ操作将从设备读取一个数据块到调用进程的内存区域，在数据传输完成前将一直阻塞
- WRITE系统调用一般很快就返回调用进程，操作系统有可能将数据暂存在缓冲区中，随后再真正将其传送到设备
- OPEN 操作将验证设备是否可用，当不可用时则返回一条错误消息
- CLOSE将确保把先前延迟写的数据真正写到设备上
- IOCTL对I/O设备的一些操作参数，进行检查或修改
  - 在MINIX中，检查和改变磁盘设备的分区是用IOCTL完成的
- SCATTERED_IO用于设置每次读写多个块，并且多个读块或写块进行排序，然后一次读多个块或写多个块

块设备的公共代码

- 块设备驱动程序的数据结构定义
  - drivers/libdriver/driver.h
  - 数据结构driver
- 主函数及其它函数
  - drivers/libdriver/driver.c
  - 在硬件执行完必要的初始化后，驱动程序都调用driver_task，同时向其传入一个driver结构作为参数。
  - 在获得一个供DMA操作使用的缓冲区地址后进入主循环，该循环将一直执行下去，不返回到调用进程。

数据结构drive

```
/* Info about and entry points into the device dependent code. */
struct driver {
  _PROTOTYPE( char *(*dr_name), (void) );
  _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( struct device *(*dr_prepare), (int device) );
  _PROTOTYPE( int (*dr_transfer), (int proc_nr, int opcode, off_t position,
                  iovec_t *iov, unsigned nr_req) );
  _PROTOTYPE( void (*dr_cleanup), (void) );
  _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
  _PROTOTYPE( void (*dr_signal), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( void (*dr_alarm), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_cancel), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_select), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_other), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_hw_int), (struct driver *dp, message *m_ptr) );
};
```

代码示例

```
/* Get a DMA buffer. */
init_buffer();

/* Here is the main loop of the disk task.  It waits for a message, carries
 * it out, and sends a reply.
 */
while (TRUE) {

  /* Wait for a request to read or write a disk block. */
  if(receive(ANY, &mess) != OK) continue;

  device_caller = mess.m_source;
  proc_nr = mess.PROC_NR;

  /* Now carry out the work. */
  switch(mess.m_type) {

  case DEV_OPEN:      r = (*dp->dr_open)(dp, &mess);break;

  case DEV_CLOSE:         r = (*dp->dr_close)(dp, &mess);     break;

  case DEV_IOCTL:     r = (*dp->dr_ioctl)(dp, &mess); break;

  case CANCEL:        r = (*dp->dr_cancel)(dp, &mess);break;
  case DEV_SELECT:    r = (*dp->dr_select)(dp, &mess);break;


  case DEV_READ:

  case DEV_WRITE:   r = do_rdwt(dp, &mess); break;
  case DEV_GATHER:
  case DEV_SCATTER: r = do_vrdwt(dp, &mess);   break;
```

函数指针
间接调用

调用：
(*dp->dr_prepare)
(*dp->dr_transfer)
完成实际的传送

驱动程序库

- 支持IBM PC兼容机的磁盘分区
  - drivers/libdriver/drvlib.h & drvlib.c
- 分区的好处
  - 大容量磁盘单位价格便宜。
  - 操作系统能够处理的设备的大小可能有限。
  - 一个操作系统可能使用两个或更多的文件系统。
  - 将一个系统的一部分文件放在一个独立的逻辑设备上可能方便一些。

第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

RAM盘

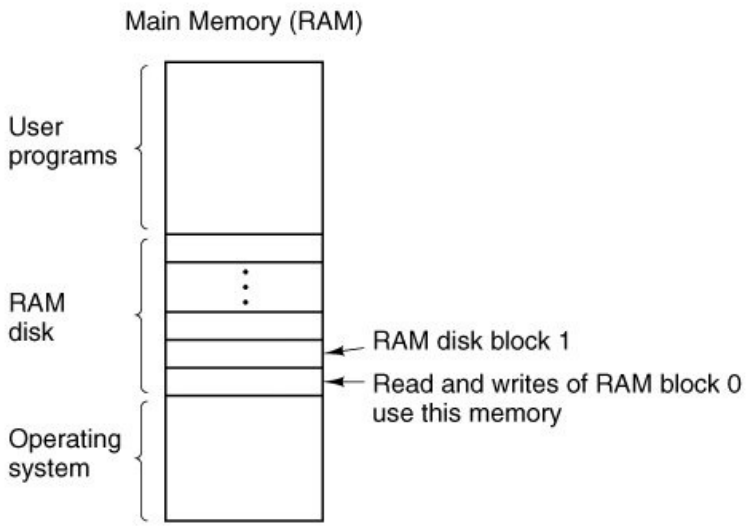
- 在内存中保留一部分存储区域，使其象普通磁盘一样使用，即RAM盘
  - RAM盘不提供永久存储，但可以快速访问

RAM盘硬件和软件

- 基本思想
  - 根据为RAM盘分配内存的大小，RAM盘被分成n块，每块的大小和实际磁盘块的大小相同。
  - 当驱动程序接收到一条读写一个数据块的消息时，它只计算被请求的块在RAM盘存储区的位置，并读出或写入该块。

数据的传输是通phys_copy汇编语言调用加以实现

一个RAM盘驱动程序可以支持多个RAM盘



MINIX3中RAM盘驱动概述

- RAM盘驱动对应六个RAM相关次设备
  - /dev/ram, /dev/kmem/, /dev/boot, /dev/mem/, /dev/null, /dev/zero
  - /dev/ram, /dev/mem, /dev/kmem, 和/dev/boot 由同一代码实现，区别在于它们作用的存储区域有所不同
    - 存储区域由数组ram_origin和ram_limit确定，通过次设备号进行索引

MINIX3中RAM盘的实现

- RAM盘驱动程序代码
  - 设备无关代码：driver.c实现主循环 (同其它磁盘驱动一样)
  - 设备相关代码：memory.c

```
/*===========================================
 *                    main
 *===========================================
PUBLIC int main(void)
{
/* Main program.  Initialize the memory dri
 m_init();
 driver_task(&m_dtab);
 return(OK);
}
```

```
/* Entry points to this driver. */
PRIVATE struct driver m_dtab = {
 m_name,     /* current device's name */
 m_do_open,  /* open or mount */
 do_nop,     /* nothing on a close */
 m_ioctl,    /* specify ram disk geometry */
 m_prepare,     /* prepare for I/O on a given minor device */
 m_transfer,    /* do the I/O */
 nop_cleanup, /* no need to clean up */
 m_geometry, /* memory device "geometry" */
 nop_signal,    /* system signals */
 nop_alarm,
 nop_cancel,
 nop_select,
 NULL,
 NULL
};
```

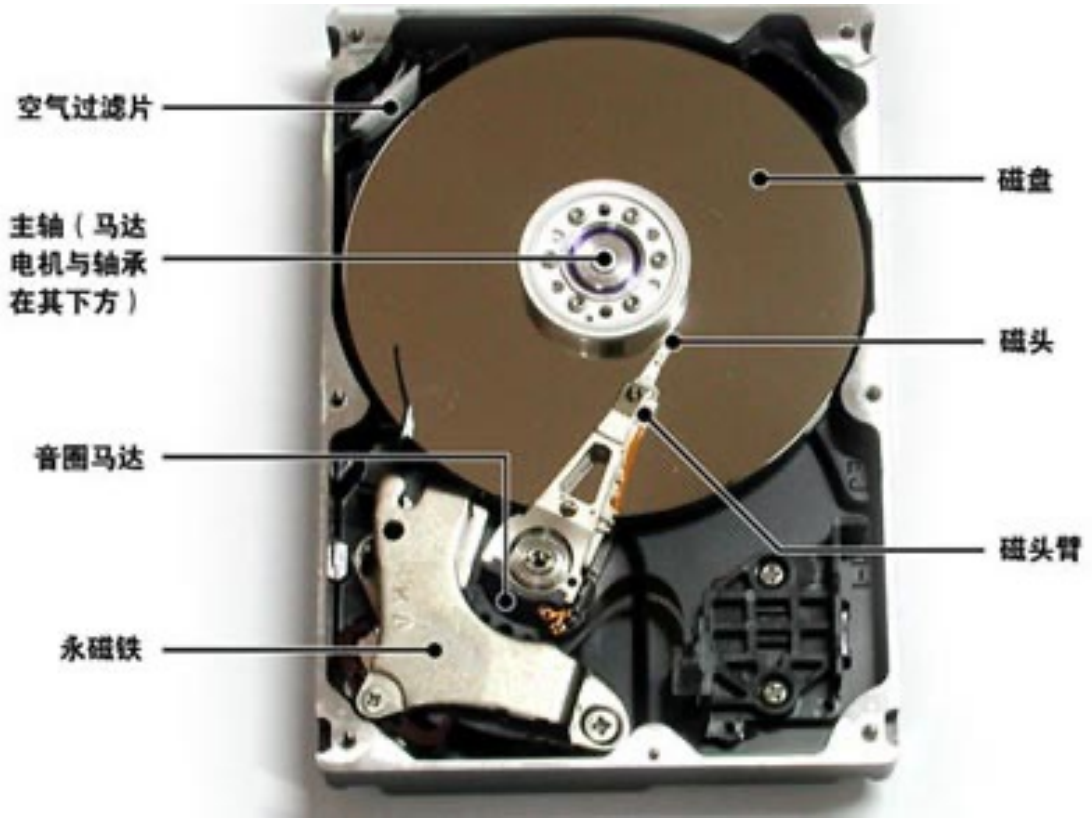
第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

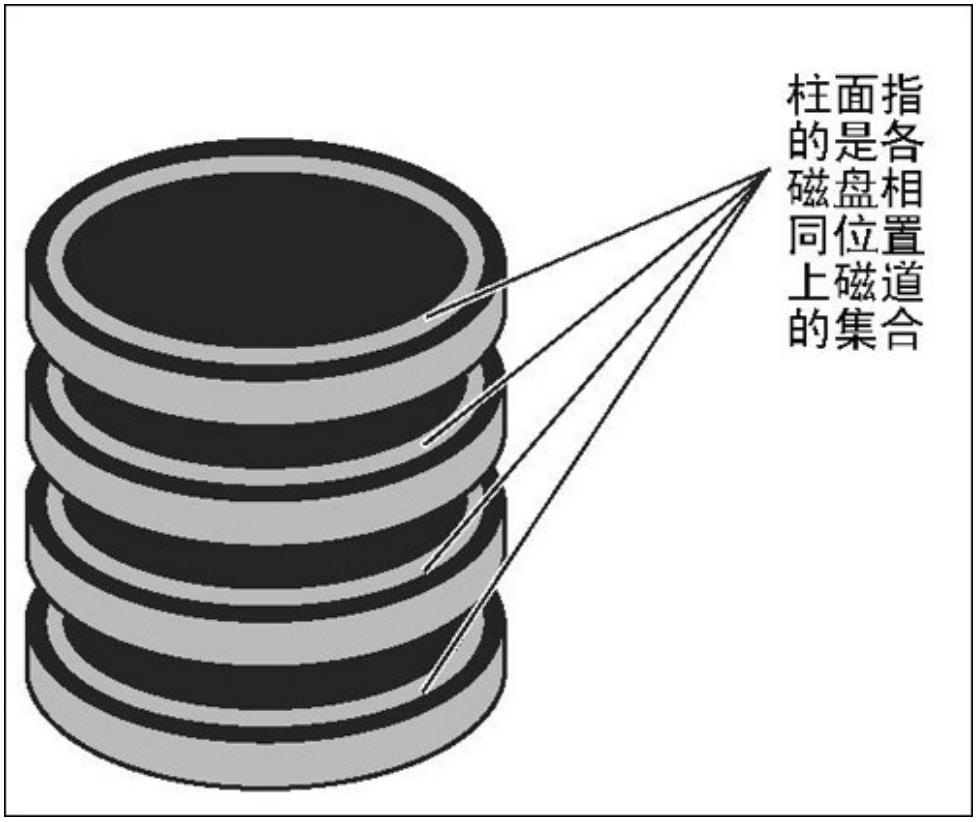
3.7 磁盘

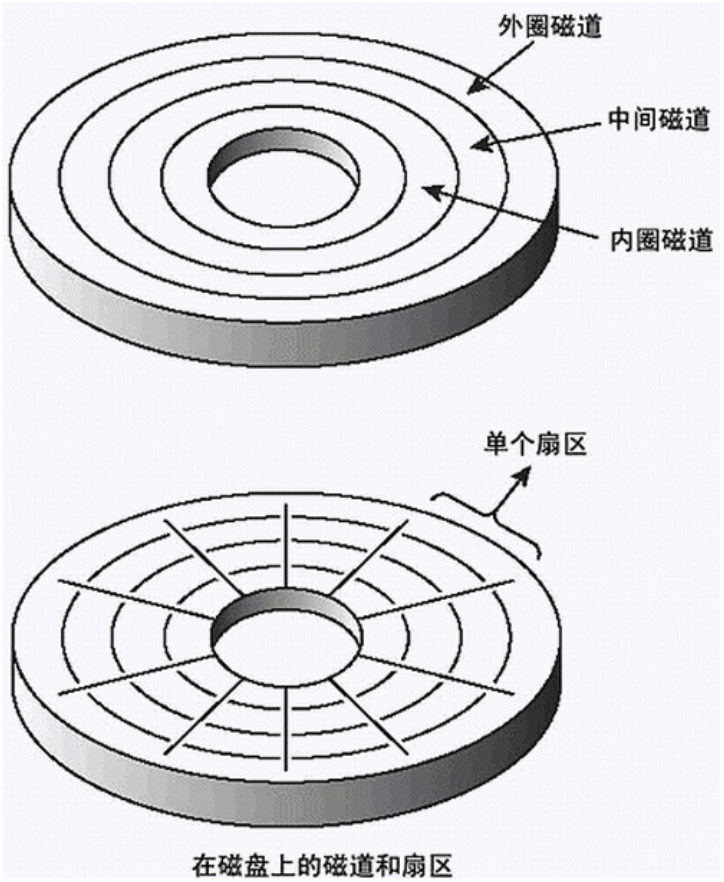
- 磁盘硬件
- RAID
- 磁盘软件
- MINIX3硬盘驱动简介
- MINIX3硬盘驱动实现
- 软盘处理

磁盘硬件



磁盘硬件



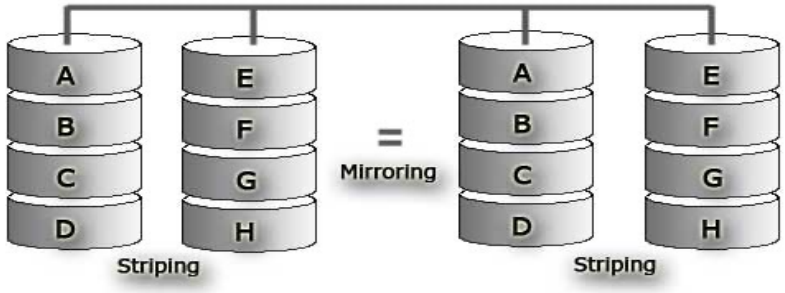


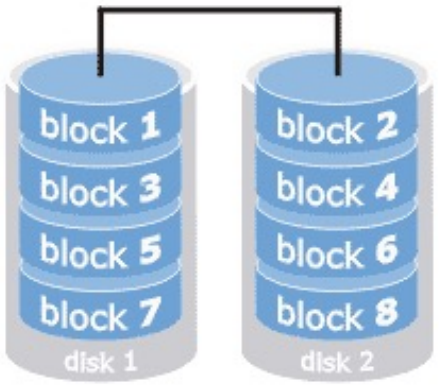
在磁盘上的磁道和扇区

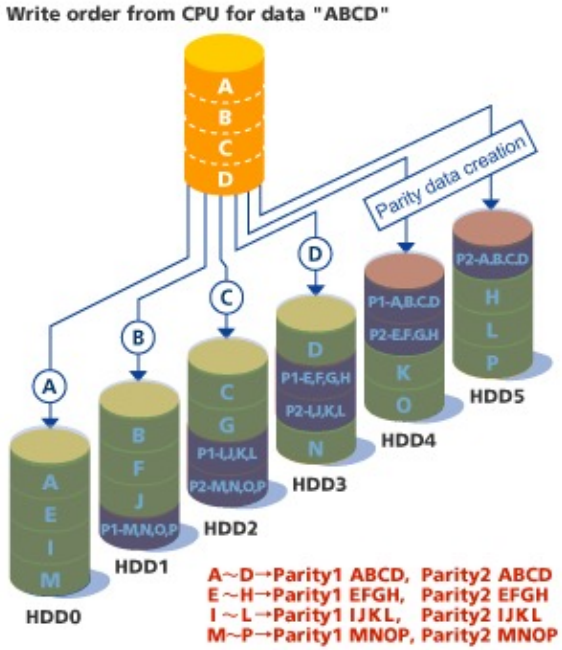
3.7 磁盘

- 磁盘硬件
- RAID
- 磁盘软件
- MINIX3硬盘驱动简介
- MINIX3硬盘驱动实现
- 软盘处理

RAID


A
B
C
D
E
F
G
H
=
Mirroring
Striping
Striping


RAID 0
striping
block 1
block 3
block 5
block 7
block 2
block 4
block 6
block 8
disk 1
disk 2


Write order from CPU for data "ABCD"
A
B
C
D
Parity data creation
HDD0
HDD1
HDD2
HDD3
HDD4
HDD5
A~D→Parity1 ABCD, Parity2 ABCD
E~H→Parity1 EFGH, Parity2 EFGH
I~L→Parity1 IJKL, Parity2 IJKL
M~P→Parity1 MNOP, Parity2 MNOP

3.7 磁盘

- 磁盘硬件
- RAID
- 磁盘软件
- MINIX3硬盘驱动简介
- MINIX3硬盘驱动实现
- 软盘处理

磁盘软件

- 读写一个磁盘块需要的时间由下面三个因素决定：
  - 寻道时间（将磁头臂移动到相应的柱面所需的时间）
  - 旋转延迟（相应的扇区旋转到磁头下面所需的时间）
  - 实际的数据传输时间。

磁盘IO时间

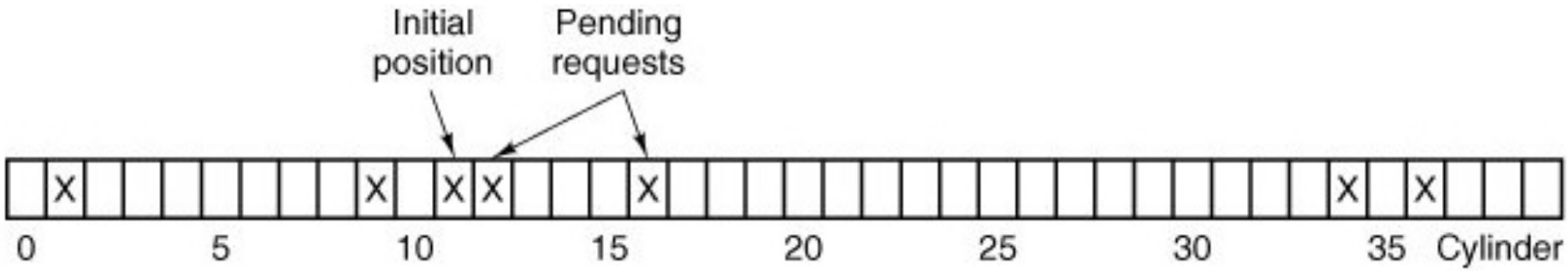
- 由于柱面定位时间在访问时间中占主要部分，合理组织磁盘数据的存储位置可提高磁盘I/O性能。

- 例子：读一个128KB大小的文件：
  - (1)文件由8个连续磁道(每个磁道32个扇区)上的256个扇区构成：
    - 20ms+(8.3ms+16.7ms)*8=220ms;
    - 其中，柱面定位时间为20ms，旋转延迟时间为8.3ms，32扇区数据传送时间为16.7ms；
  - (2)文件由256个随机分布的扇区构成：
    - (20ms+8.3ms+0.5ms)*256=7373ms;
    - 其中，1扇区数据传送时间为0.5ms；

- 随机分布时的访问时间为连续分布时的33.5倍。

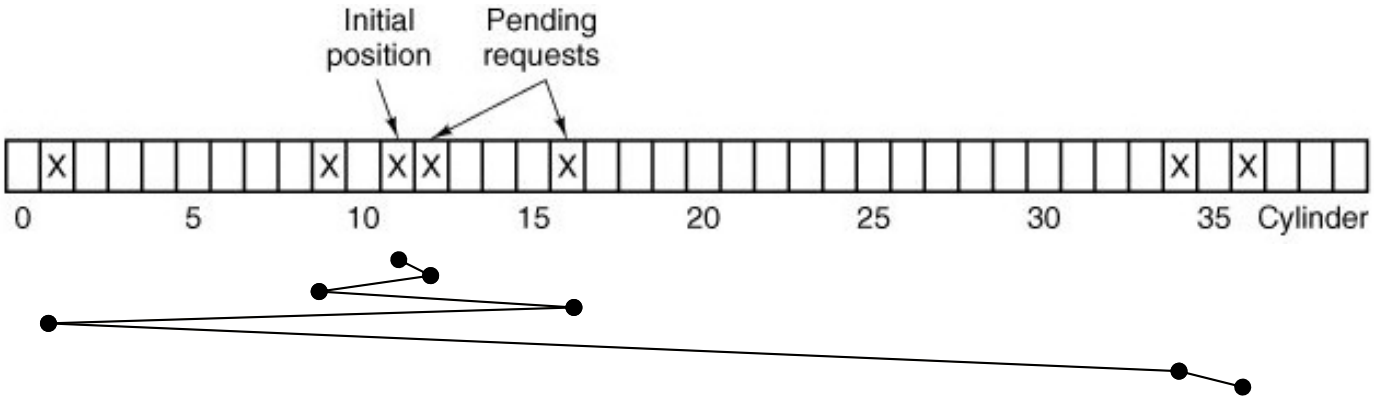
磁盘臂调度算法--FCFS算法

- 如果磁盘驱动程序每次接收一个请求并按照接收顺序执行，即先来先服务（FCFS）
  - 考虑一个具有四十个柱面的磁盘。假设一个请求到达请求读柱面11上的一个数据块，当对柱面11寻道时，又顺序到达了新请求要求寻道1、36、16、34、9和12，则它们被安排进入请求等待表，每一个柱面对应一个单独的链表。
  - 该算法中磁盘臂分别需要移动10、35、20、18、25和3个柱面，总共需要移动111个柱面。



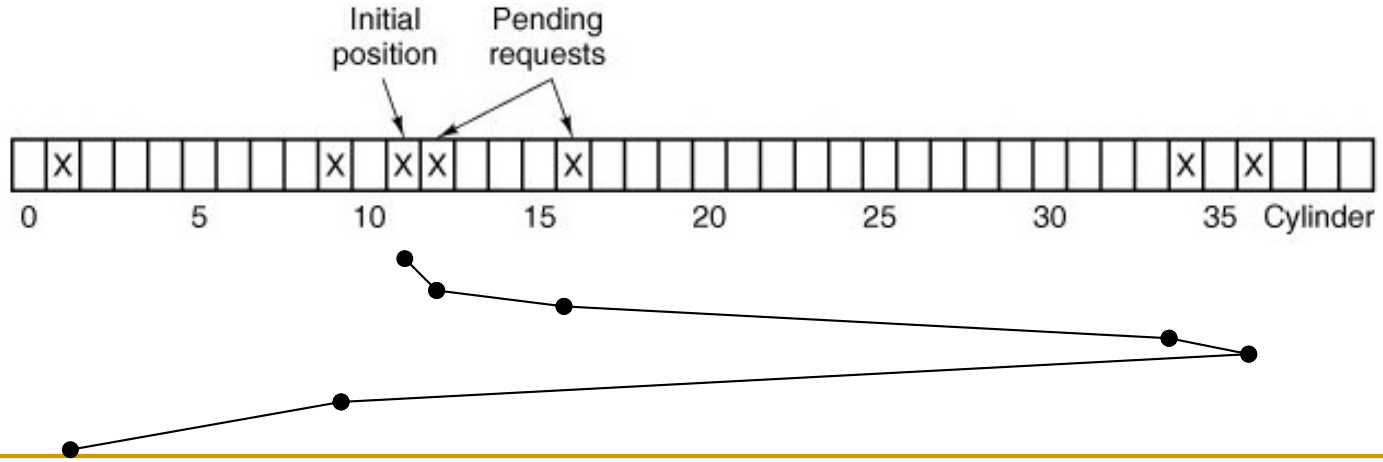
磁盘臂调度算法--最短寻道算法

- 在寻道时，选择与磁盘臂最接近的柱面请求，即最短寻道算法(SSF)
  - 考虑一个具有四十个柱面的磁盘。假设一个请求到达请求读柱面11上的一个数据块，当对柱面11寻道时，又顺序到达了新请求要求寻道1 、 36 、 16 、 34 、 9 和12，则它们被安排进入请求等待表，每一个柱面对应一个单独的链表。
  - 该算依次为12 、 9 、 16 、 1 、 34和36。按照这个顺序，磁盘臂分别需要移动1 、 3 、 7 、 15 、 33和2个柱面，总共需要移动61个柱面。



磁盘臂调度算法--电梯算法

- 电梯调度思想：电梯保持按一个方向运动，直到在那个方向上没有更远的请求为止，然后改变方向。
- 维护一个二进制位，即当前的方向：向上(外)或是向下(内)。
  - 当一个请求结束之后，磁盘驱动程序检查该位，如果是向上，磁盘臂移至下一个更高(向上)的等待请求。如果更高的位置没有请求，就翻转方向位。
  - 如果方向位设置为向下，同时存在一个低位置的请求，则移向该位置
- 假设初始方向位为向上，则各柱面获得服务的顺序是 12 、 16 、 34 、 36 、 9和1，磁盘臂分别移动1 、 4 、 18 、 2 、 27和8个柱面，总共移动60个柱面。

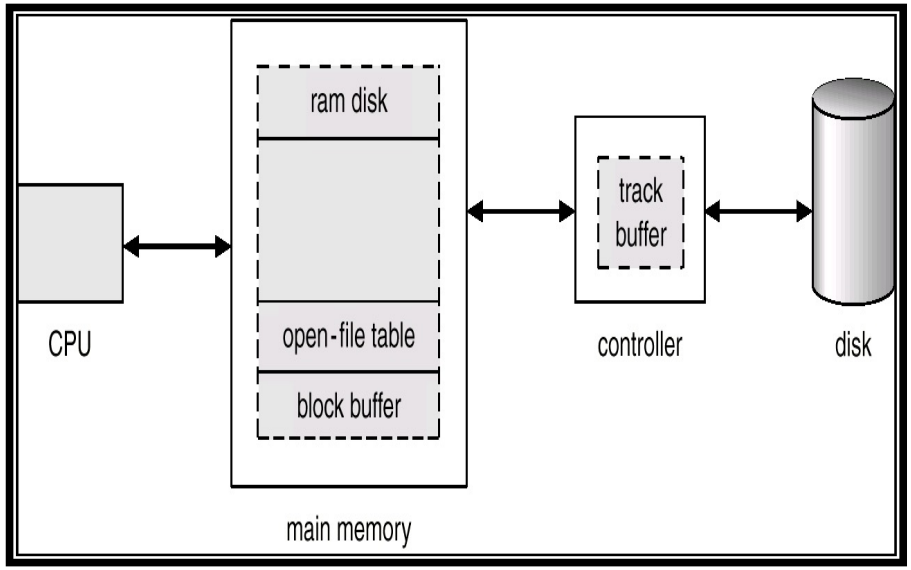


错误处理

- 磁盘驱动程序需尽可能地处理如下错误
  - 程序性错误（例如请求读写不存在的扇区）。
  - 暂时性校验和错（例如由磁头上的灰尘引起）。
  - 永久性校验和错（例如磁盘块的物理损坏）。
  - 寻道出错（例如磁盘臂应定位在第6柱面，但却到了第7柱面）。
  - 控制器错（例如控制器拒绝接受命令）。

每次一道缓冲

- 驱动程序需要把磁盘臂移到某个位置，那么读一个扇区还是读一条磁道都差不多
- 有些磁盘驱动程序通过维护一个秘密的每次一道的高速缓冲来充分利用这一特性，并且不被独立于设备的软件所感知。如果需要的扇区位于高速缓冲里，则不需要磁盘数据传输。
- 一些控制器将这个过程更进一步，在它们自己内部的存储器中实现每次一道缓冲，而对驱动程序透明。



3.7 磁盘

- 磁盘硬件
- RAID
- 磁盘软件
- MINIX3硬盘驱动简介
- MINIX3硬盘驱动实现
- 软盘处理

确盘驱动程序

- 主循环采用共享代码实现
  - DEV_OPEN：选择分区、子分区等
  - DEV_READ、DEV_WRITE、DEV_GATHER、DEV_SCATTER：准备和传输两阶段
    - do_rdwt、do_vrdwt → dp->dr_prepare、dp->dr_transfer
  - DEV_CLOSE：关闭
- 对于多个块的读写调度，由文件系统确定，由磁盘驱动根据请求队列，执行相应的数据读写操作。

硬盘驱动类型

- 针对硬盘异构性，几种处理方法
  - 为需要支持的每种硬盘控制器重新编译一个操作系统版本。
  - 在核心中编译几个不同的硬盘驱动程序，由核心在启动时自动决定使用哪一个。
  - 在核心中编译几个不同的硬盘驱动程序，提供一种方法使用户决定使用哪一个。

3.7 磁盘

- 磁盘硬件
- RAID
- 磁盘软件
- MINIX3硬盘驱动简介
- MINIX3硬盘驱动实现
- 软盘处理

MINIX3硬盘驱动实现

- 硬盘设备相关代码
  - drivers/at_wini/At_wini.c

```
/*===========================================================================*
 *                at_winchester_task                         *
 *===========================================================================*/
PUBLIC int main()
{
/* Set special disk parameters then call the generic main loop. */
  init_params();
  driver_task(&w_dtab);
  return(OK);
}
```

```
/* Entry points to this driver. */
PRIVATE struct driver w_dtab = {
  w_name,       /* current device's name */
  w_do_open,        /* open or mount request, initialize device */
  w_do_close,       /* release device */
  do_diocntl,   /* get or set a partition's geometry */
  w_prepare,    /* prepare for I/O on a given minor device */
  w_transfer,   /* do the I/O */
  nop_cleanup,      /* nothing to clean up */
  w_geometry,       /* tell the geometry of the disk */
  nop_signal,   /* no cleanup needed on shutdown */
  nop_alarm,    /* ignore leftover alarms */
  nop_cancel,       /* ignore CANCELs */
  nop_select,   /* ignore selects */
  w_other,      /* catch-all for unrecognized commands and ioctls */
  w_hw_int      /* leftover hardware interrupts */
};
```

- 设备无关代码
  - drivers/libdriver/driver.c

3.7 磁盘

- 磁盘硬件
- RAID
- 磁盘软件
- MINIX3硬盘驱动简介
- MINIX3硬盘驱动实现
- 软盘处理

软盘处理

- 简单的设备对应简单的控制器，为此操作系统就必须考虑更多的内容
  - 廉价的、缓慢的、低容量的软盘驱动器不值得配置硬盘所使用的复杂的集成控制器，因此驱动程序软件就不得不处理一些在硬盘中被隐藏于硬盘驱动器的操作。
    - 寻道（SEEK）操作需要显式地编程
  - 使软盘驱动程序复杂化的一些因素
    - 可移动介质
    - 多种磁盘格式
    - 电机控制

第三章 I/O系统 提纲

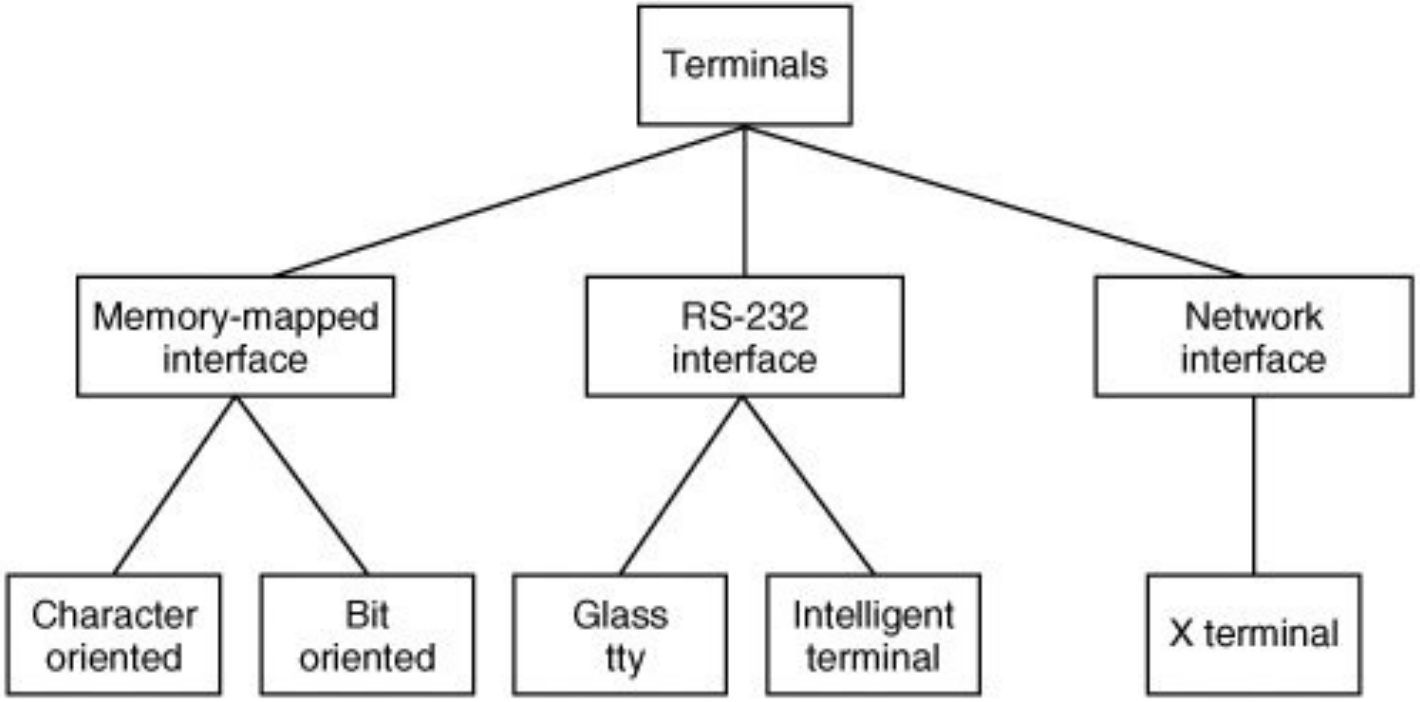
- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

终端

- 终端硬件
- 终端软件
- MINIX3终端驱动程序概述
- 设备无关终端驱动程序
- 设备相关软件

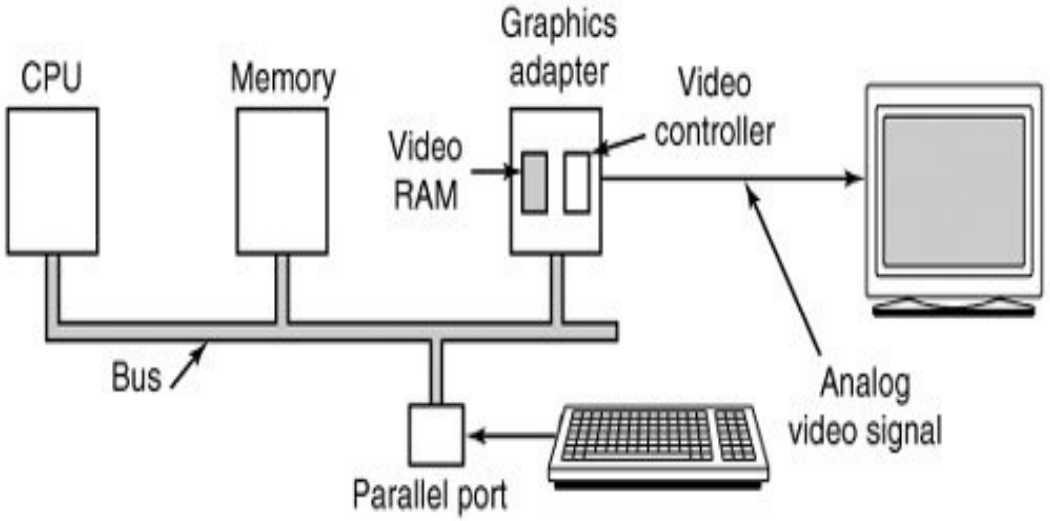
终端硬件

- 终端：用户与计算机交互的工具，包括键盘和显示器



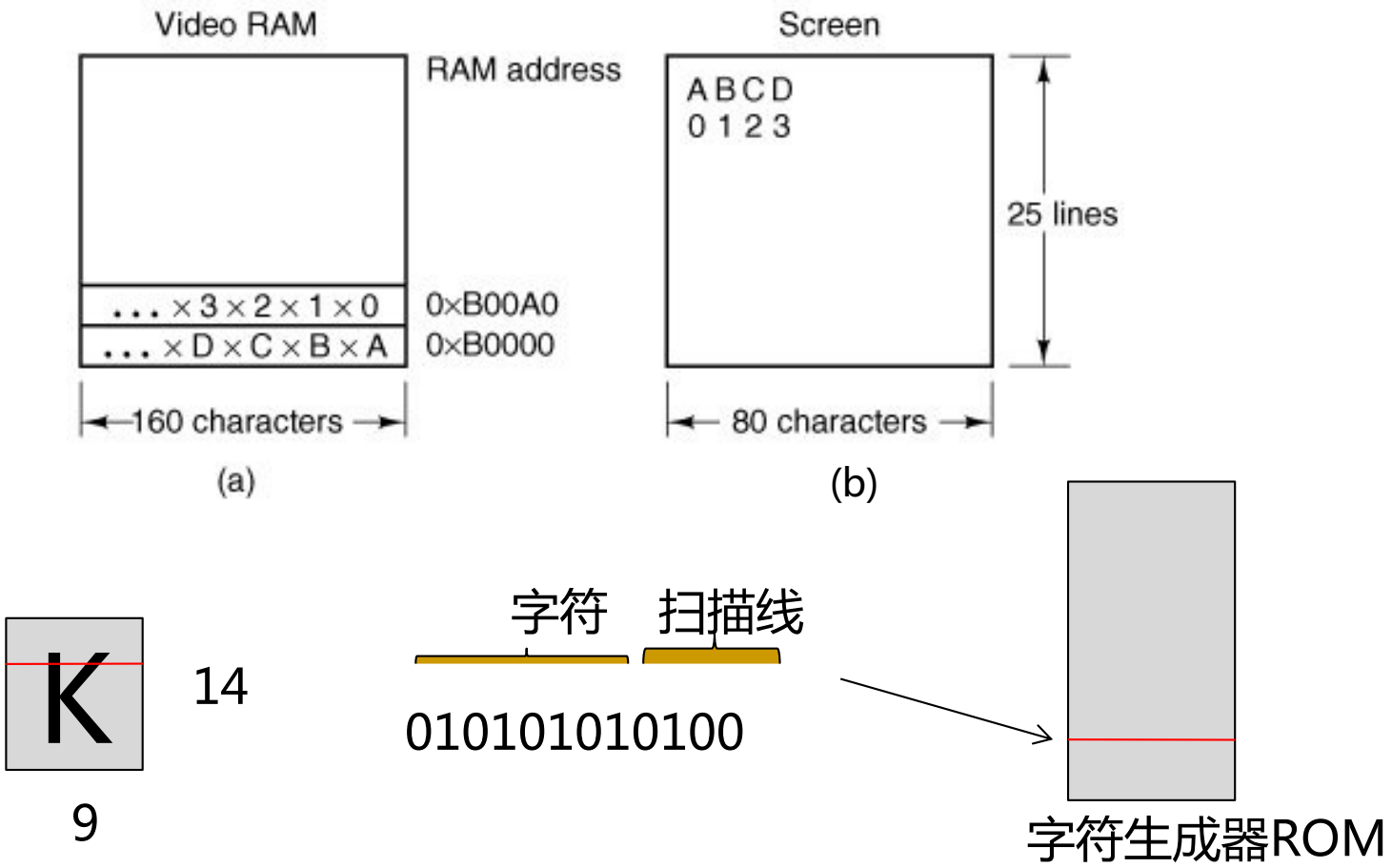
内存映射终端

- 视频RAM是计算机地址空间中的一部分，可以按地址进行访问
- 视频控制器：依据从视频RAM中取出的字符，产生视频信号



对于存储器映像显示器，键盘是与显示器分开的，它可能通过一个串行口或并行口和计算机相连。对于每一个键动作，产生CPU中断，键盘中断程序通过读I/O口取得键入的字符。

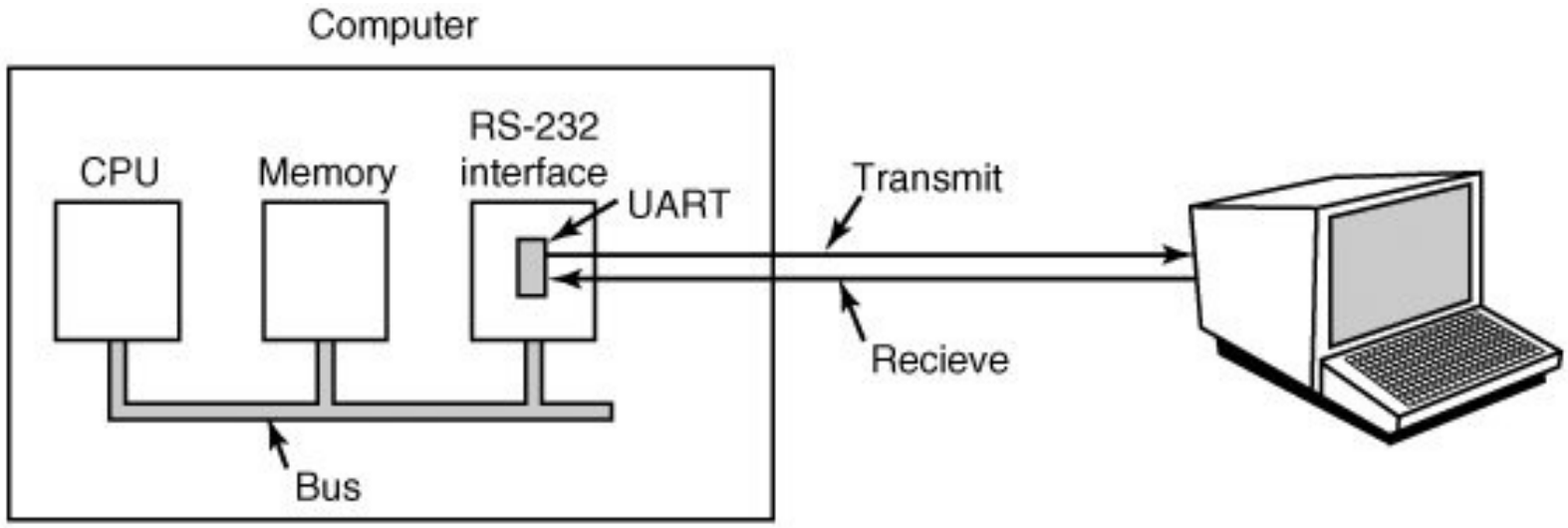
字符映射显示模式



另一种模式， 即位图模式显示，每个像素独立控制

RS-232终端

- RS－232终端是包括一个键盘和一个显示器的设备，通过一次传输一位的串行口与计算机通讯
- 为了向RS－232终端发一个字符，计算机必须一次传输一位，在字符前面加一起始位，后接一个或两个终止位为字符定界
- 一般传输速率为9600、19200或38400 bps



终端

- 终端硬件
- 终端软件
- MINIX3终端驱动程序概述
- 设备无关终端驱动程序
- 设备相关软件

输入软件

- 键盘驱动程序的工作模式
  - 生模式(非规范模式)
    - 驱动程序的工作仅仅是接收输入，并且不经任何修改就向上层传递

例：如果用户键入了dste而不是date，然后键入三个退格建和ate键来对此进行修正，最后键入一个回车键，那么用户程序将收到键入的全部11个ASCII码

  - 熟模式(规范模式)
    - 驱动程序处理行内的编辑，仅仅向用户程序传输正确的一行

输出软件

- RS－232终端
  - 输出被首先拷贝到缓冲，每一字符通过传输线送至终端
- 内存映像终端
  - 需要打印的字符在某一时刻从用户空间中取出，直接放入视频RAM中
  - 跟踪在视频RAM中当前位置，以便在那里打印可打印字符，然后向前移动输出位置

终端

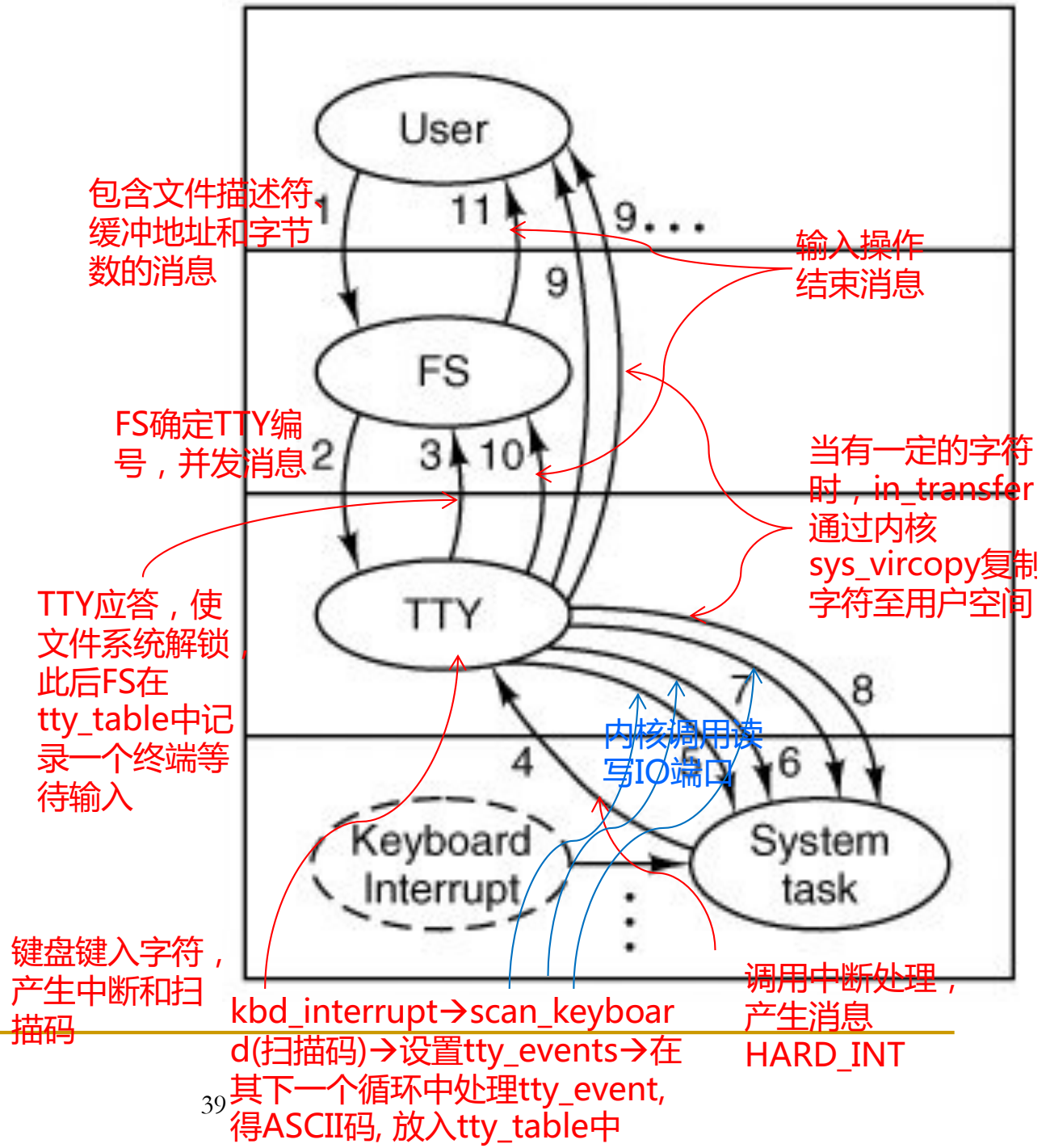
- 终端硬件
- 终端软件
- MINIX3终端驱动程序概述
- 设备无关终端驱动程序
- 设备相关软件

MINIX3终端驱动程序概述

- 终端驱动程序包含在几个文件中，构成了MINIX中最大的驱动程序。终端驱动程序既处理键盘，也处理显示器。
- 终端驱动程序接收七种消息类型：
  - 从终端读（来自FS，它代表用户进程）
  - 向终端写（来自FS，它代表用户进程）
  - 为IOCTL设置终端参数（来自FS，它代表用户进程）
  - 键盘中断发生（有键被按下或者释放）
  - 终止上一个请求（当信号发生时，来自文件系统）
  - 打开设备
  - 关闭设备

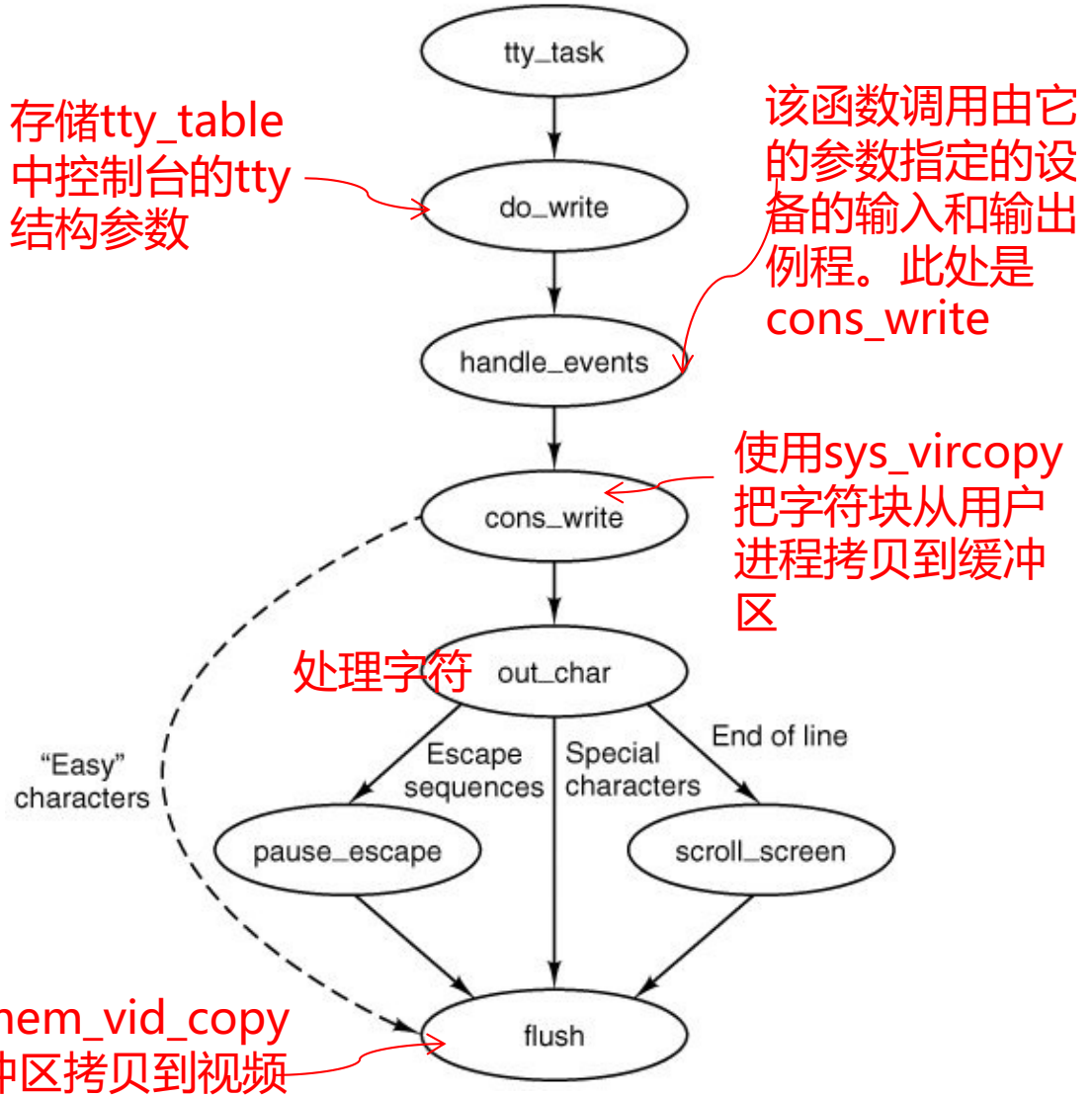
终端输入

- 终端驱动程序工作流程，以处理字符输入为例。
  - 当用户在控制台上登录时，系统为其创建一个shell，它用/dev/console作为标准输入、标准输出和标准出错。shell启动并试图通过调用库过程read读标准输入



终端输出

- 终端输出流程以处理输出字符串为例
  - 一个进程在试图显示时一般要调用printf。
  - printf调用WRITE向文件系统发送一条消息。该消息包含一个指向待显示字符序列的指针（不是字符序列本身）
  - 文件系统向终端驱动程序发送一条消息
  - 终端驱动程序取出字符并拷贝到视频RAM中。



终端

- 终端硬件
- 终端软件
- MINIX3终端驱动程序概述
- 设备无关终端驱动程序
- 设备相关软件

设备无关终端驱动程序

- 一个终端任务要支持几个不同类型的终端设备
- 数据结构tty
  - 每个终端设备都有一个这样的结构与之对应（控制台显示和键盘看作一个单一的终端）
- 主要的终端任务和设备无关的支持函数都在tty.c中定义

```
typedef struct tty {
  int tty_events;       /* set when TTY should inspect this line */
  int tty_index;        /* index into TTY table */
  int tty_minor;        /* device minor number */

  /* Input queue.  Typed characters are stored here until read by a program.
  u16_t *tty_inhead;        /* pointer to place where next char goes */
  u16_t *tty_intail;        /* pointer to next char to be given to prog */
  int tty_incount;      /* # chars in the input queue */
  int tty_eotct;        /* number of "line breaks" in input queue */
  devfun_t tty_devread;         /* routine to read from low level buffers *
  devfun_t tty_icancel;     /* cancel any device input */
  int tty_min;          /* minimum requested #chars in input queue */
  timer_t tty_tmr;      /* the timer for this tty */

  /* Output section. */
  devfun_t tty_devwrite;    /* routine to start actual device output */
  devfunarg_t tty_echo;         /* routine to echo characters input */
  devfun_t tty_ocancel;         /* cancel any ongoing device output */
  devfun_t tty_break;       /* let the device send a break */

  /* Terminal parameters and status. */
  int tty_position;     /* current position on the screen for echoing */
  char tty_reprint;         /* 1 when echoed input messed up, else 0 */
  char tty_escaped;         /* 1 when LNEXT (^V) just seen, else 0 */
  char tty_inhibited;       /* 1 when STOP (^S) just seen (stops output) */
  char tty_pgrp;        /* slot number of controlling process */
  char tty_openct;          /* count of number of opens of this tty */

  /* Information about incomplete I/O requests is stored here. */
  char tty_inrepcode;       /* reply code, TASK_REPLY or REVIVE */
  char tty_inrevived;       /* set to 1 if revive callback is pending */
  char tty_incaller;        /* process that made the call (usually FS) */
  char tty_inproc;      /* process that wants to read from tty */
  vir_bytes tty_in_vir;     /* virtual address where data is to go */
```

tty.c

```
if (tp < tty_addr(NR_CONS)) {
        scr_init(tp);
        tp->tty_minor = CONS_MINOR + s;
} else
if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
        rs_init(tp);
        tp->tty_minor = RS232_MINOR + s-NR_CONS;
} else {
        pty_init(tp);
        tp->tty_minor = s - (NR_CONS+NR_RS_LINES) + TTYPX_MINOR;
}
```

在console.c中

```
/* Fill in TTY function hooks. */
tp->tty_devwrite = cons_write;
tp->tty_echo = cons_echo;
tp->tty_ioctl = cons_ioctl;
```

具体终端处理函数

```
/*===========================================================================*
 *                tty_task                   *
 *===========================================================================*/
PUBLIC void main(void)
{
/* Main routine of the terminal task. */

  message tty_mess;       /* buffer for all incoming messages */
  unsigned line;
  int s;
  char *types[] = {"task","driver","server", "user"};
  register struct proc *rp;
  register tty_t *tp;

  /* Initialize the TTY driver. */
  tty_init();

  /* Get kernel environment (protected_mode, pc_at and ega are needed). */
  if (OK != (s=sys_getmachine(&machine))) {
    panic("TTY","Couldn't obtain kernel environment.", s);
  }

  /* Final one-time keyboard initialization. */
  kb_init_once();

  printf("\n");

  while (TRUE) {

      /* Check for and handle any events on any of the ttys. */
      for (tp = FIRST_TTY; tp < END_TTY; tp++) {
          if (tp->tty_events) handle_events(tp);
      }

      /* Get a request message. */
      receive(ANY, &tty_mess);

      /* First handle all kernel notification types that the TTY supports.
       *  - An alarm went off, expire all timers and handle the events.
       *  - A hardware interrupt also is an invitation to check for events.
       *  - A new kernel message is available for printing.
       *  - Reset the console on system shutdown.
       * Then see if this message is different from a normal device driver
       * request and should be handled separately. These extra functions
       * do not operate on a device, in constrast to the driver requests.
       */
      switch (tty_mess.m_type) {
```

终端

- 终端硬件
- 终端软件
- MINIX3终端驱动程序概述
- 设备无关终端驱动程序
- 设备相关软件

设备相关软件

- 尽管控制台显示器和键盘看作一个单一的终端，但支持它们的物理设备是完全独立的
  - 在一个标准的桌面系统中显示器使用一块插在底板上的适配卡。
  - 键盘由设计在母板上的电路支持，通过该电路与键盘内部的一个8位单片机接口。
  - 两个子设备需要完全独立的软件支持，即文件keyboard.c和console.c。

第三章 I/O系统 提纲

- 3.1 I/O硬件原理
- 3.2 I/O软件原理
- 3.3 死锁
- 3.4 MINIX3 I/O概述
- 3.5 MINIX3 块设备
- 3.6 RAM盘
- 3.7 磁盘
- 3.8 终端

作业

- 3, 4, 11, 12, 14, 15, 17, 22, 28