第三章　I/O系统

翁楚良

https://chuliangweng.github.io

2023 春　ECNU

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

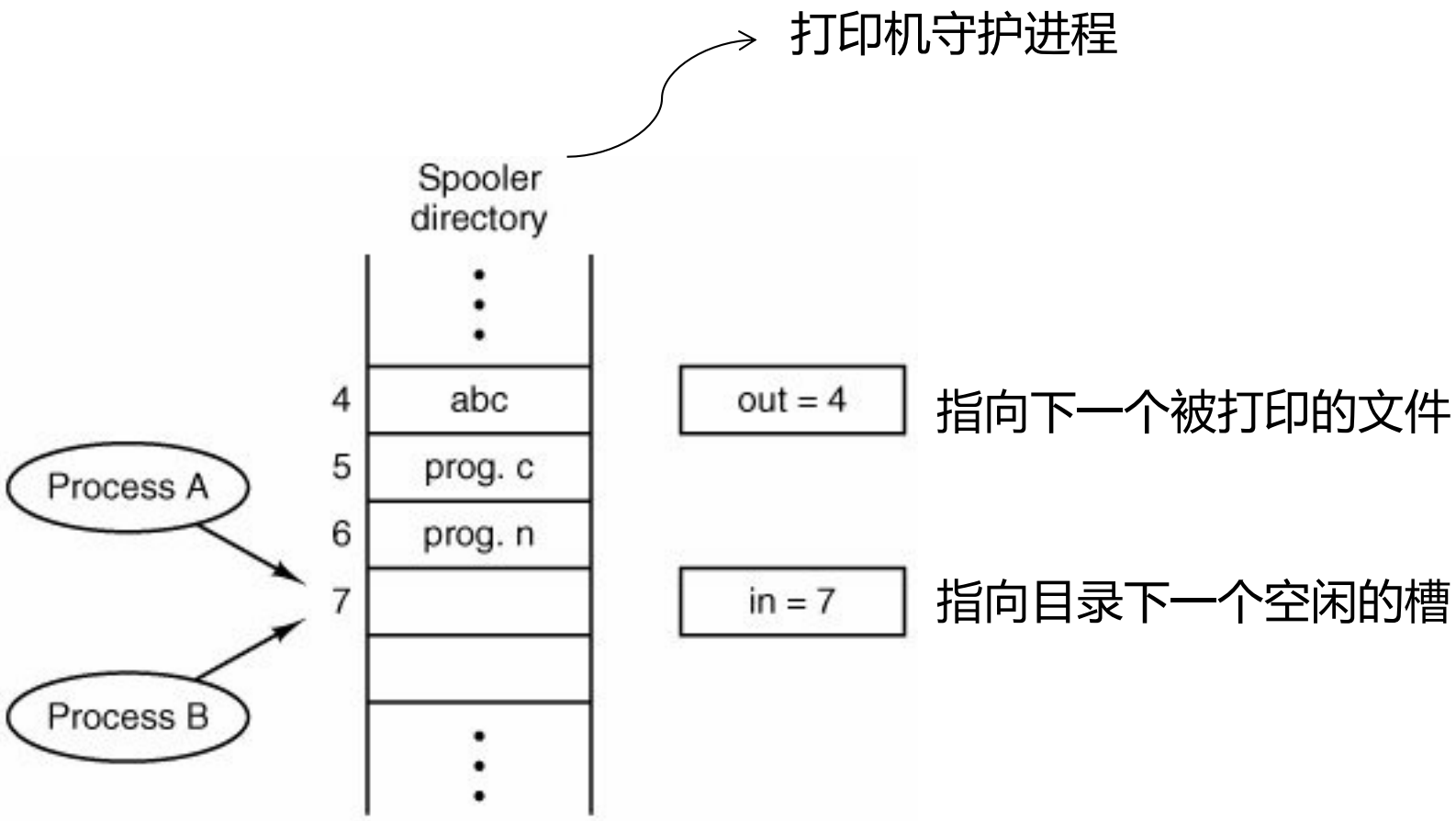
- 资源
- 死锁原理
- 鸵鸟算法
- 死锁检测与恢复
- 死锁预防
- 避免死锁

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

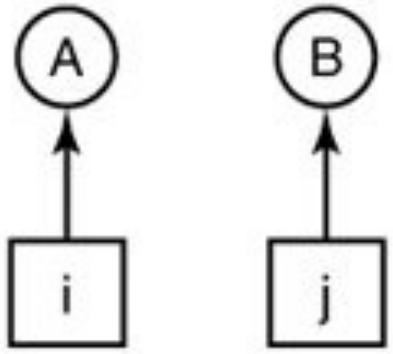

打印机守护进程
Spooler directory
4
abc
5
prog. c
6
prog. n
7
Process A
Process B
out = 4
指向下一个被打印的文件
in = 7
指向目录下一个空闲的槽

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

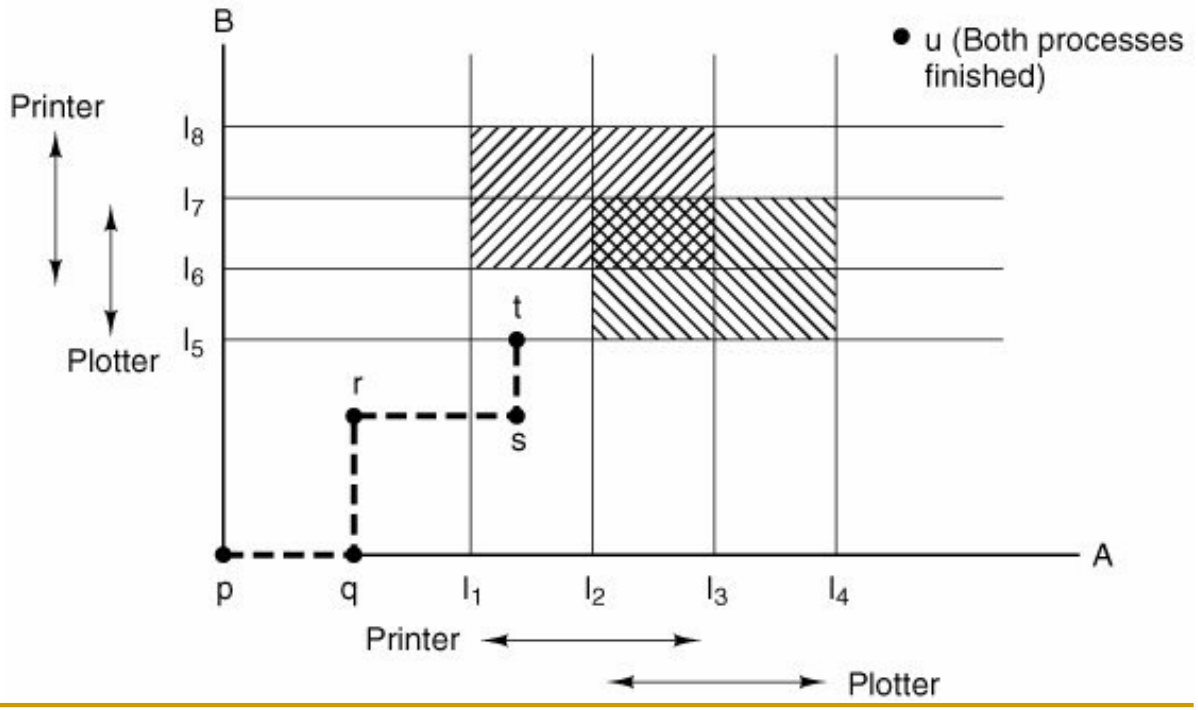
(c)

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。

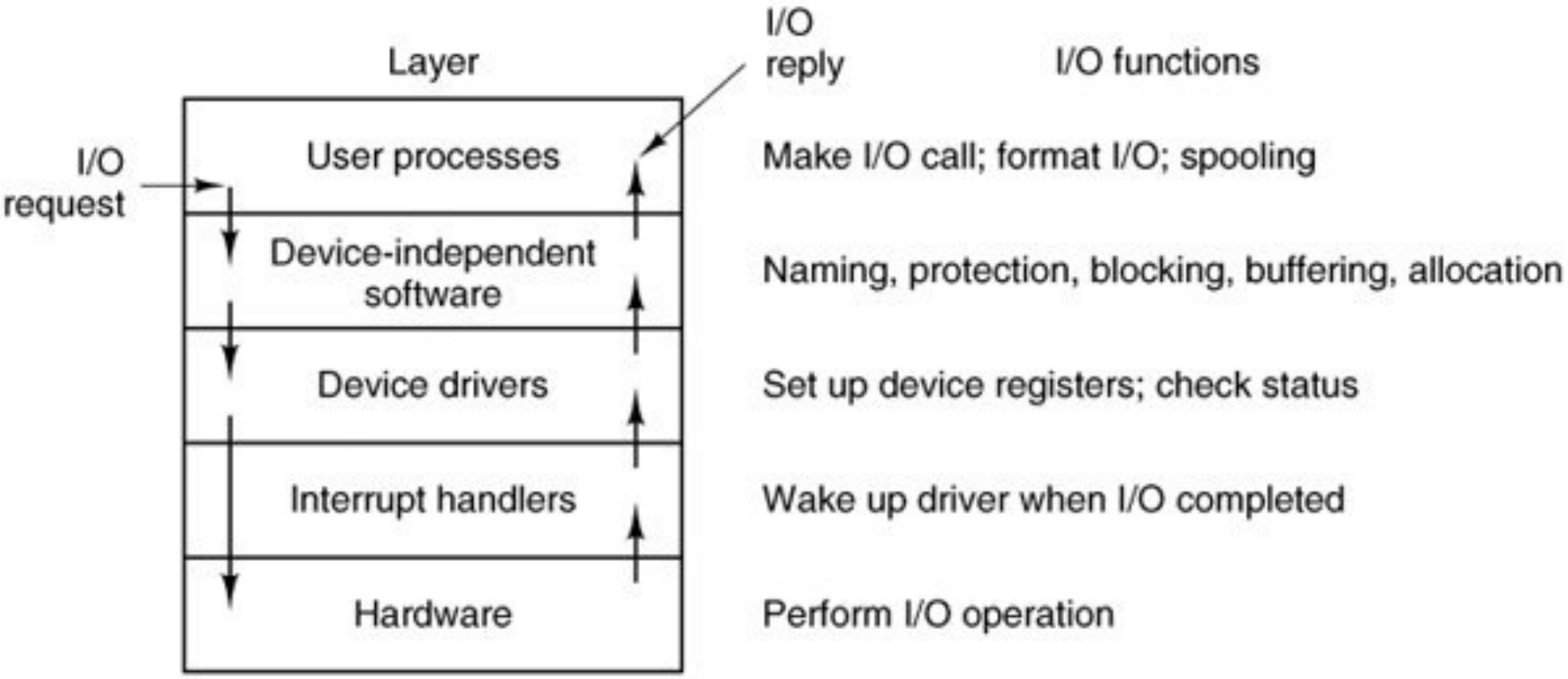
两阶段加锁法

- 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

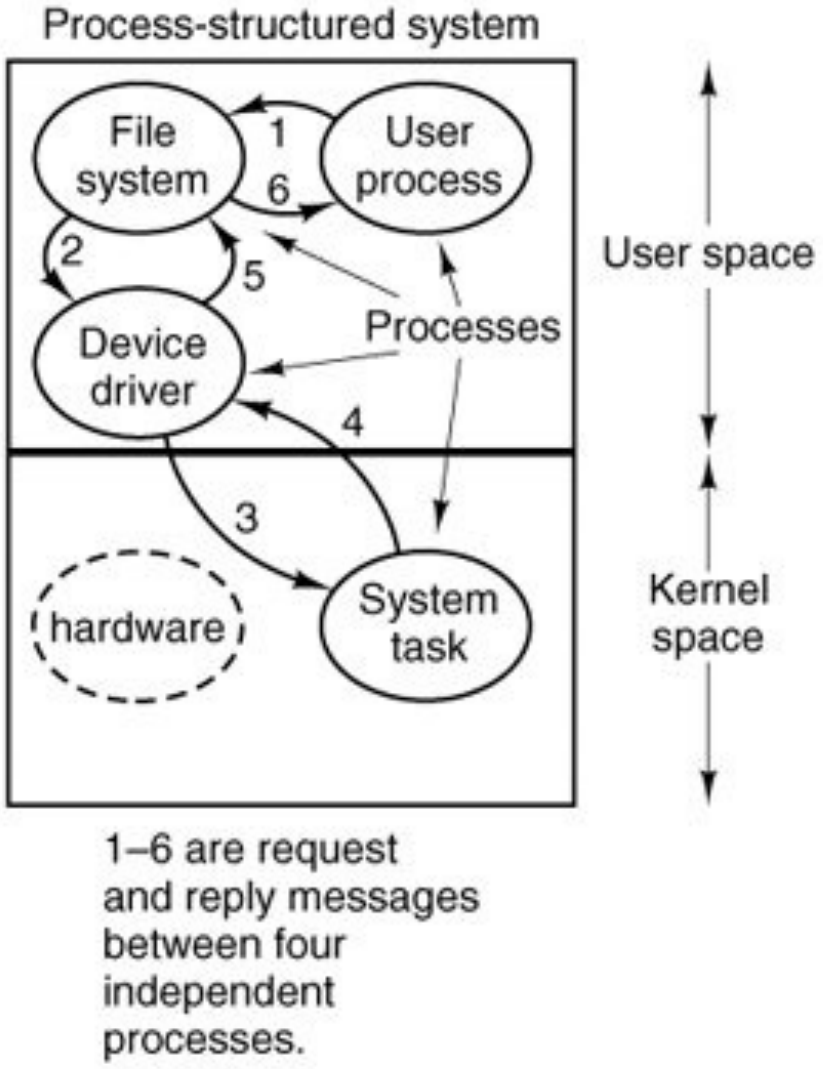
3.4 MINIX3 I/O概述

- 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。