第二章　进程管理

翁楚良

https://chuliangweng.github.io

2023 春　ECNU

第二章进程管理 提纲

- 2.1 进程
- 2.2 进程间通信
- 2.3 经典并发问题
- 2.4 进程调度
- 2.5 MINIX3进程概述
- 2.6 MINIX3进程实现
- 2.7 MINIX3系统任务
- 2.8 MINIX3时钟任务

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

源码组织

- src/include -- 头文件目录
- src/kernel -- 调度、消息、时钟和系统任务
- src/drivers -- 驱动程序(磁盘、打印、…)
- src/servers -- 服务器进程(进程管理器、文件系统管理器、…)
- src/lib -- 库例程(open, read, …)
- src/boot -- 引导和安装系统的代码
- src/tools -- Makefile和脚本 (用于编译系统)

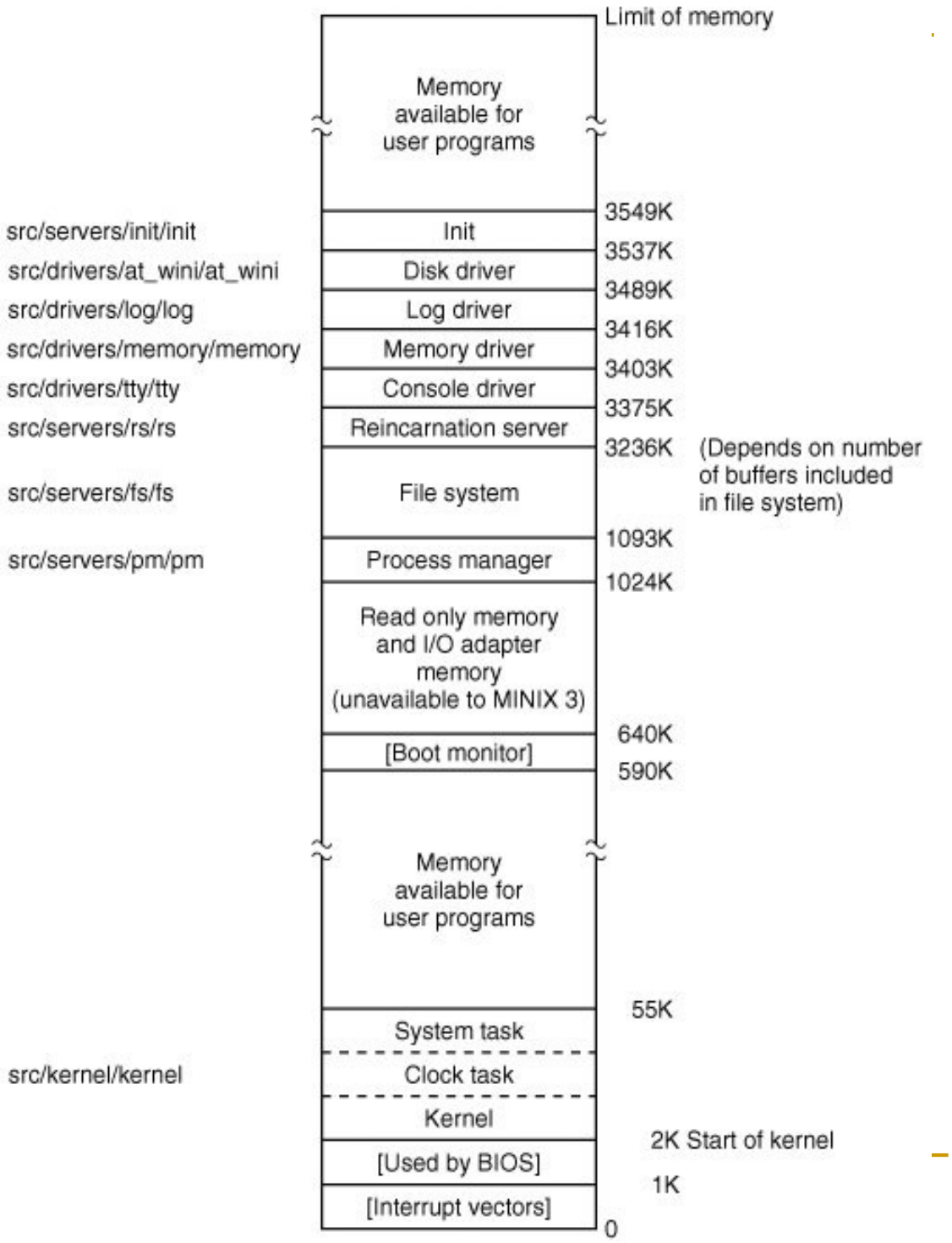
源码学习内容

- src/kernel
  - 实现MINIX3的第一层功能，包括内核、时钟任务、系统任务，
  - 编译为同一个二进制文件，但以三个进程独立运行
- src/drivers
  - 实现设备驱动层，第三章
- src/server/pm
  - 实现进程管理器，第四章
- src/server/fs
  - 实现文件系统管理器，第五章

编译工具

- 源码编译
  - make, Makefile
  - src/tools
  - 分别进入每个源码目录，为每个部分生成相应的二进制可执行文件
- 引导映像
  - installboot负责生成引导映像文件
  - 在src/tools/中可以找到新编译的引导映像文件
  - 需要放至/boot/image或/boot中

内存分布


Limit of memory
Memory available for user programs
3549K
Init
src/servers/init/init
3537K
Disk driver
src/drivers/at_wini/at_wini
3489K
Log driver
src/drivers/log/log
3416K
Memory driver
src/drivers/memory/memory
3403K
Console driver
src/drivers/tty/tty
3375K
Reincarnation server
src/servers/rs/rs
3236K
(Depends on number of buffers included in file system)
File system
src/servers/fs/fs
1093K
Process manager
src/servers/pm/pm
1024K
Read only memory and I/O adapter memory (unavailable to MINIX 3)
640K
[Boot monitor]
590K
Memory available for user programs
55K
System task
Clock task
src/kernel/kernel
Kernel
2K Start of kernel
[Used by BIOS]
1K
[Interrupt vectors]
0

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

头文件

- 公共头文件
  - src/include
  - src/include/sys
  - 由POSIX标准规定
- MINIX头文件
  - src/include/minix
    - config.h -- 与硬件相关的配置
  - src/include/ibm
    - 在IBM类型机器

头文件说明

- #include <filename>
  - 到默认头文件目录去查找文件
- #include “filename”
  - 首先在源文件所在的目录中查找，若没找到，再到默目录中去查找

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

进程表(进程控制块)

- 在MINIX中，进程表被分为三部分
  - 内核、进程管理器、文件系统服务器各占其一
- 内核进程表
  - src/kernel/proc.h
  - the process' registers, stack pointer, state, memory map, stack limit, process id, accounting, alarm time, and message info

```
05516   struct proc {
05517     struct stackframe_s p_reg;    /* process' registers saved in stack frame */
05518     reg_t p_ldt_sel;              /* selector in gdt with ldt base and limit */
05519     struct segdesc_s p_ldt[2+NR_REMOTE_SEGS]; /* CS, DS and remote segments */
05520
05521     proc_nr_t p_nr;               /* number of this process (for fast access) */
05522     struct priv  *p_priv;         /* system privileges structure */
05523     char p_rts_flags;             /* SENDING, RECEIVING, etc. */
05524
05525     char p_priority;              /* current scheduling priority */
05526     char p_max_priority;          /* maximum scheduling priority */
05527     char p_ticks_left;            /* number of scheduling ticks left */
05528     char p_quantum_size;          /* quantum size in ticks */
05529
05530     struct mem_map p_memmap[NR_LOCAL_SEGS];   /* memory map (T, D, S) */
05531
05532     clock_t p_user_time;          /* user time in ticks */
05533     clock_t p_sys_time;           /* sys time in ticks */
05534
05535     struct proc  *p_nextready;    /* pointer to next ready process */
05536     struct proc  *p_caller_q;     /* head of list of procs wishing to send */
05537     struct proc  *p_q_link;       /* link to next proc wishing to send */
05538     message *p_messbuf;           /* pointer to passed message buffer */
05539     proc_nr_t p_getfrom;          /* from whom does process want to receive? */
05540     proc_nr_t p_sendto;           /* to whom does process want to send? */
05541
05542     sigset_t p_pending;           /* bit map for pending kernel signals */
05543
05544     char p_name[P_NAME_LEN];      /* name of the process, including \0 */
05545   };
```

进程的特权级

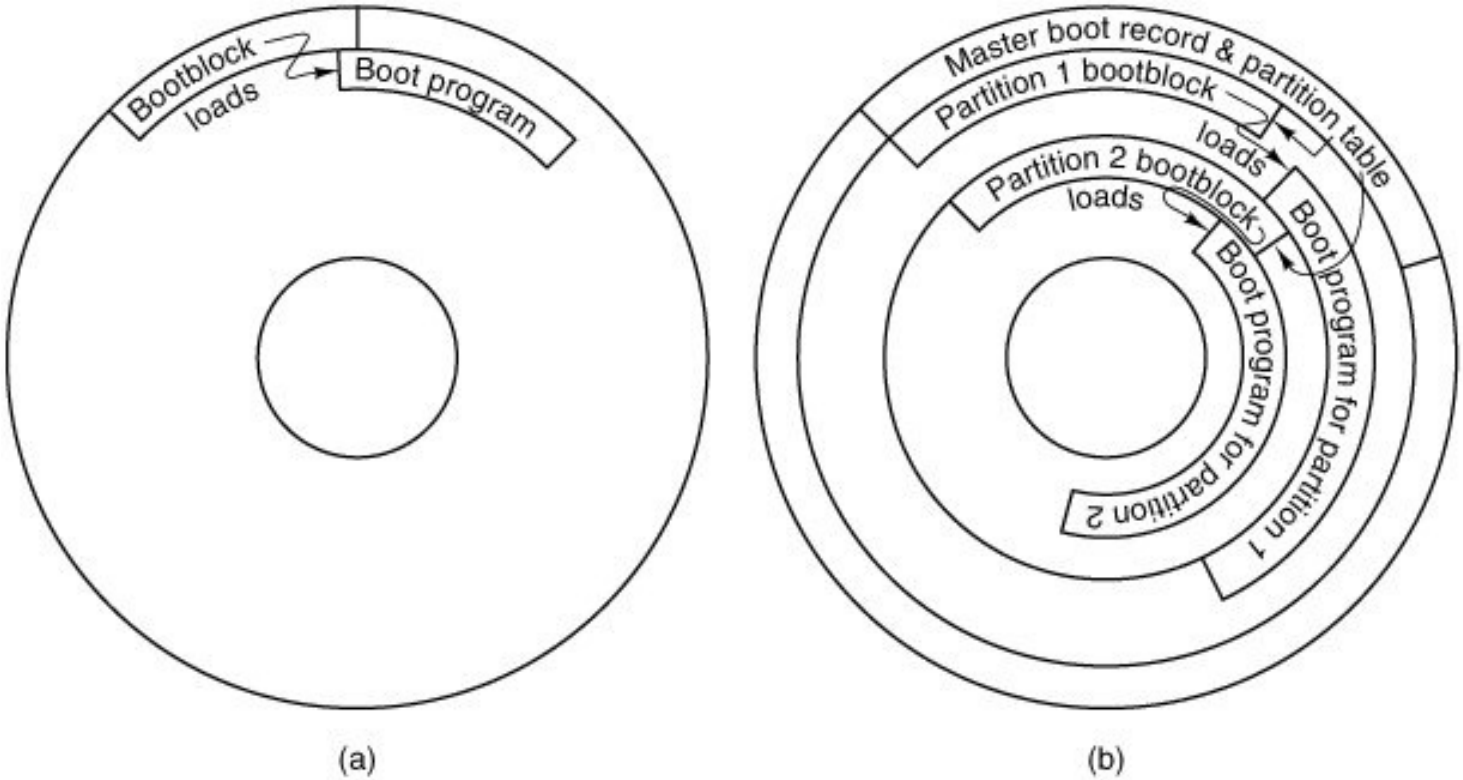
- 32位intel系列处理器提供四个特权级
- MINIX3 用到三个特权级

```
05842 /* Privileges. */

05843 #define INTR_PRIVILEGE 0 /* kernel and interrupt handlers */

05844 #define TASK_PRIVILEGE 1 /* kernel tasks */

05845 #define USER_PRIVILEGE 3 /* servers and user processes */
```

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

磁盘结构


Bootblock loads
Boot program
(a)
Master boot record & partition table
Partition 1 bootblock loads
Partition 2 bootblock loads
Boot program for partition 2
Boot program for partition 1
(b)

软盘

硬盘

Boot与bootblock

- boot是MINIX的次级装入程序，它不仅可以装入操作系统，而且作为一个监控程序，它允许用户改变、设置和保存不同的参数。
- boot从它所在分区的第二个扇区中寻找一套可使用的参数。
  - MINIX保留每个硬盘设备的前1K字节作为一个引导块(bootblock)
  - 其中只有一个扇区(512字节)被ROM引导程序或主引导扇区装入
  - 另外512字节可以用来保存设置信息，这些信息控制引导操作，并且被传到操作系统本身。缺省的设置显示一个只有一个选择项的菜单，即引导MINIX。
  - 但该设置信息可以被改变，以显示一个更复杂的菜单。这样便允许引导其他操作系统（通过装入并执行其他分区的引导扇区）或使用不同的选择项来引导MINIX。缺省设置也可以被修改，以旁路掉该菜单而直接引导MINIX。
- boot并不是操作系统的一部分，但它很精巧，可以使用文件系统的数据结构来找到操作系统映像

MINIX3引导映像

Component	Description	Loaded by
kernel	Kernel + clock and system tasks	(in boot image)
pm	Process manager	(in boot image)
fs	File system	(in boot image)
rs	(Re)starts servers and drivers	(in boot image)
memory	RAM disk driver	(in boot image)
log	Buffers log output	(in boot image)
tty	Console and keyboard driver	(in boot image)
driver	Disk (at, bios, or floppy) driver	(in boot image)
init	parent of all user processes	(in boot image)

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

系统初始化(对应教材OS版本)

- 在MINIX3内核被成功加载到内存后，控制权会转移到kernel/mpx.s文件，根据机器字长为16位或32位来分别加载mpx88.s或者mpx386.s文件并执行。
- 对于32位PC机，MINIX3内核会跳转到mpx386.s文件的MINIX标签处开始执行
  - start.c文件中的cstart函数，初始化全局描述符、中断描述符等等
  - 以跳转至main入口结束(main.c文件中的main)

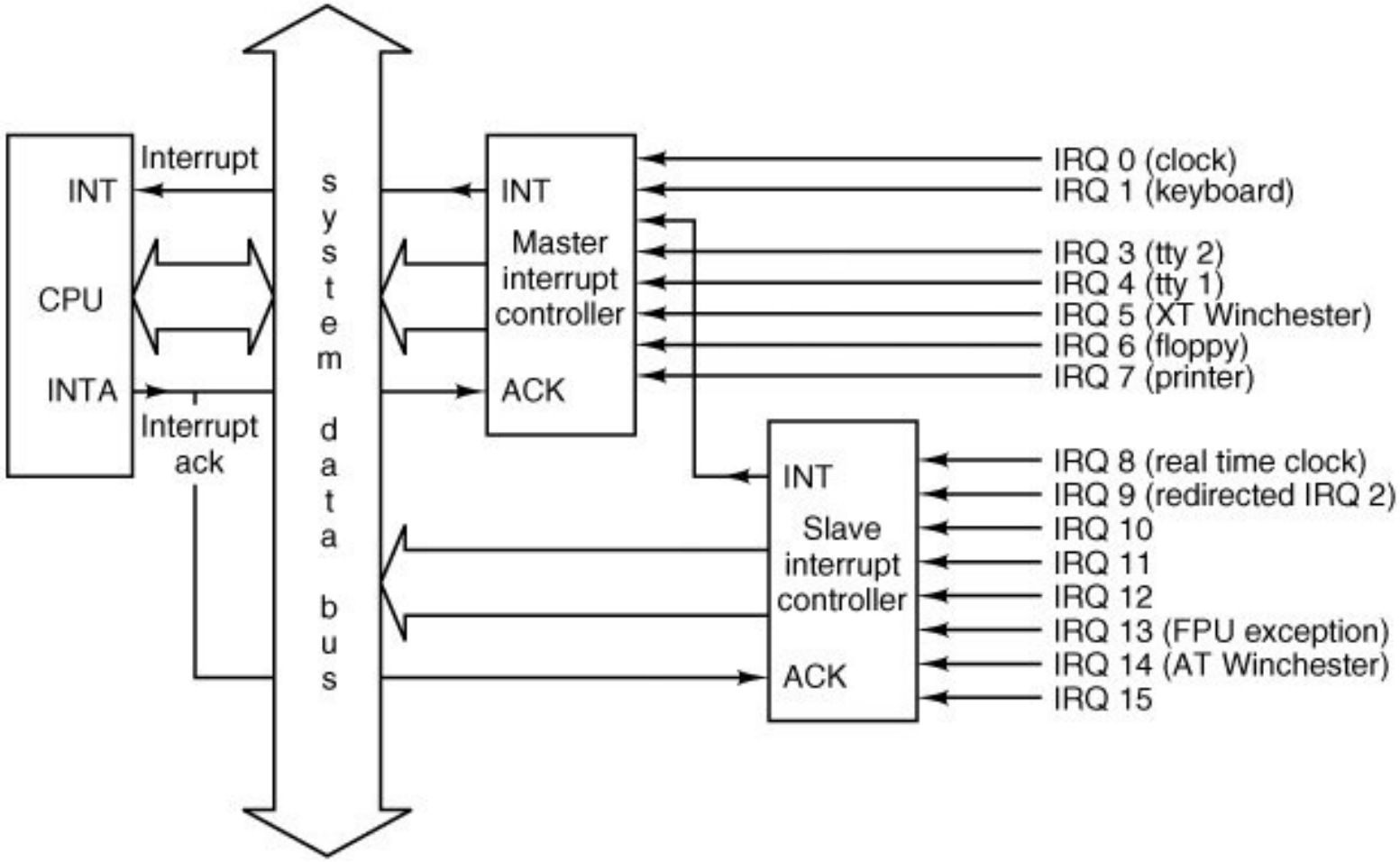
main

- Main初始化流程
  - 初始化中断控制器
  - 初始化进程表与特权控制表
  - 初始化boot image中的进程
  - 设置IDLE为第一个运行的进程
  - 显示欢迎信息
  - 调用mpx386.s文件中的_restart函数开始进程调度(上下文切换)等

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

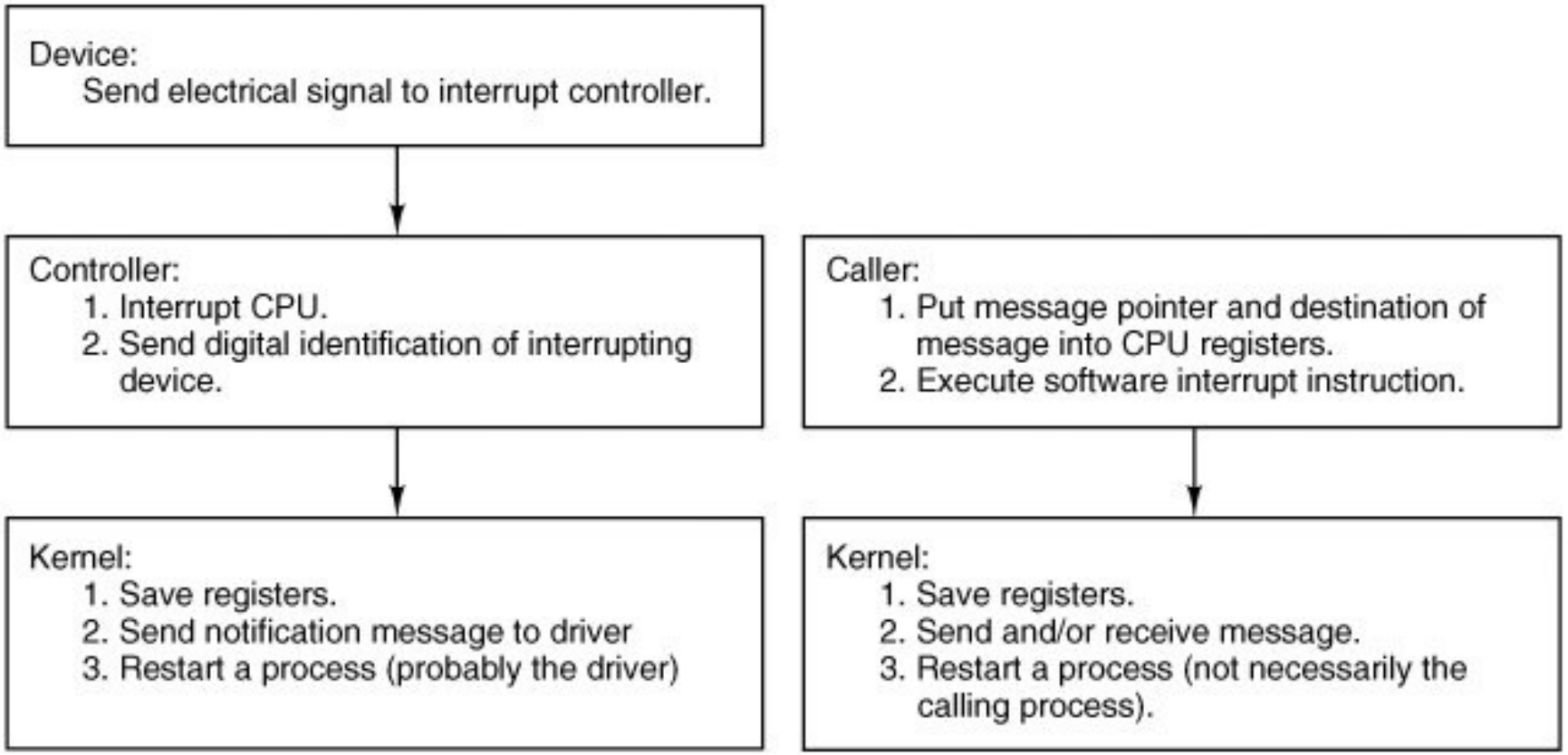
32位 Intel PC中断处理硬件



硬件中断处理

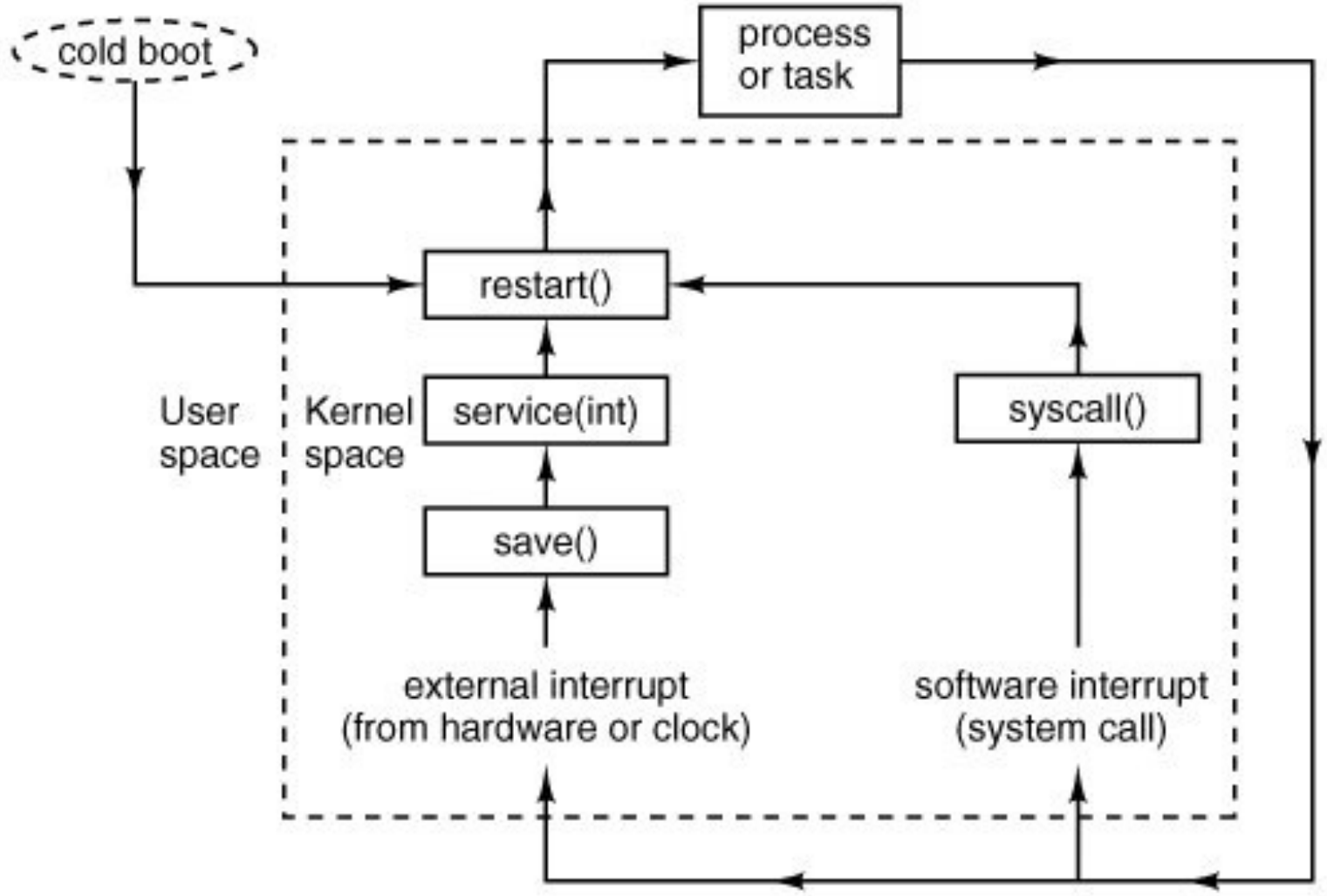
- 中断信号出现在中断控制器芯片右侧的IRQn信号线上，通过CPU INT管脚的连接线通知CPU发生了中断
- CPU发出INTA（中断应答）信号
- 中断控制器芯片将数据放在系统数据总线上并通知处理器应执行哪个服务例程
  - 在系统初始化期间，当main调用intr_init时，对中断控制器进行编程
  - 确定了对应于各条输入线的信号将向CPU送出什么样的数据
  - 数据是一个8位的数值，它用作对一个表格的索引，该表格最多可包含256项
  - MINIX的表格中含有56个表项，其中实际用到35项，其余21项保留供MINIX将来扩展使用
  - 在32位的Intel处理器上，这张表中包含中断门描述符，每个中断门描述符是一个含有若干域的8字节结构
- CPU执行被选中的描述符所指向的代码，与执行如下的汇编语言指令完全一样(软件中断)
  - int <nnn>

中断处理过程



(a) 硬件中断处理过程　　(b) 系统调用处理过程

_restart (mpx386.s) (对应教材OS版本)



Restart the current process or the next process if it is set

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

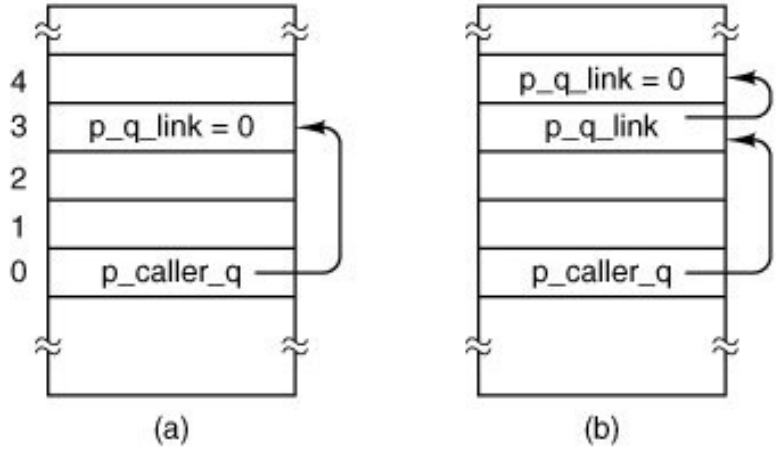
进程间通信

- MINIX中的进程使用消息进行通信，这里使用到进程会合的原理。
- 当一个进程执行SEND时，内核检查目标进程是否在等待从发送者（或任一发送者）发来的消息。
  - 如果是，则该消息从发送者的缓冲区拷贝到接收者的缓冲区，同时这两个进程都被标记为就绪态。
  - 如果目标进程未在等待消息，则发送者被标记为阻塞，并被挂入一个等待将消息发送到接收进程的进程队列中。
- 当一个进程执行RECEIVE时，内核检查该队列中是否存在向它发送消息的进程。若有，则消息从被阻塞的发送进程拷贝到接收进程，并将两者均标记为就绪；若不存在这样的进程，则接收进程被阻塞，直到一条消息到达。

实现

- sys_call (proc.c)
  - 检查并保证该消息指定的源进程和目标进程合法性及消息指针的合法性。
  - 验证目标进程正在运行,没有启动终止过程
  - 调用mini_send, mini_receive, mini_notify
  - 防止发送死锁

```
/* Process is now blocked.  Put in on the destination's queue. */
xpp = &dst_ptr->p_caller_q;     /* find end of list */
while (*xpp != NIL_PROC) xpp = &(*xpp)->p_q_link;
*xpp = caller_ptr;              /* add caller to end */
caller_ptr->p_q_link = NIL_PROC;   /* mark new end of list */
```

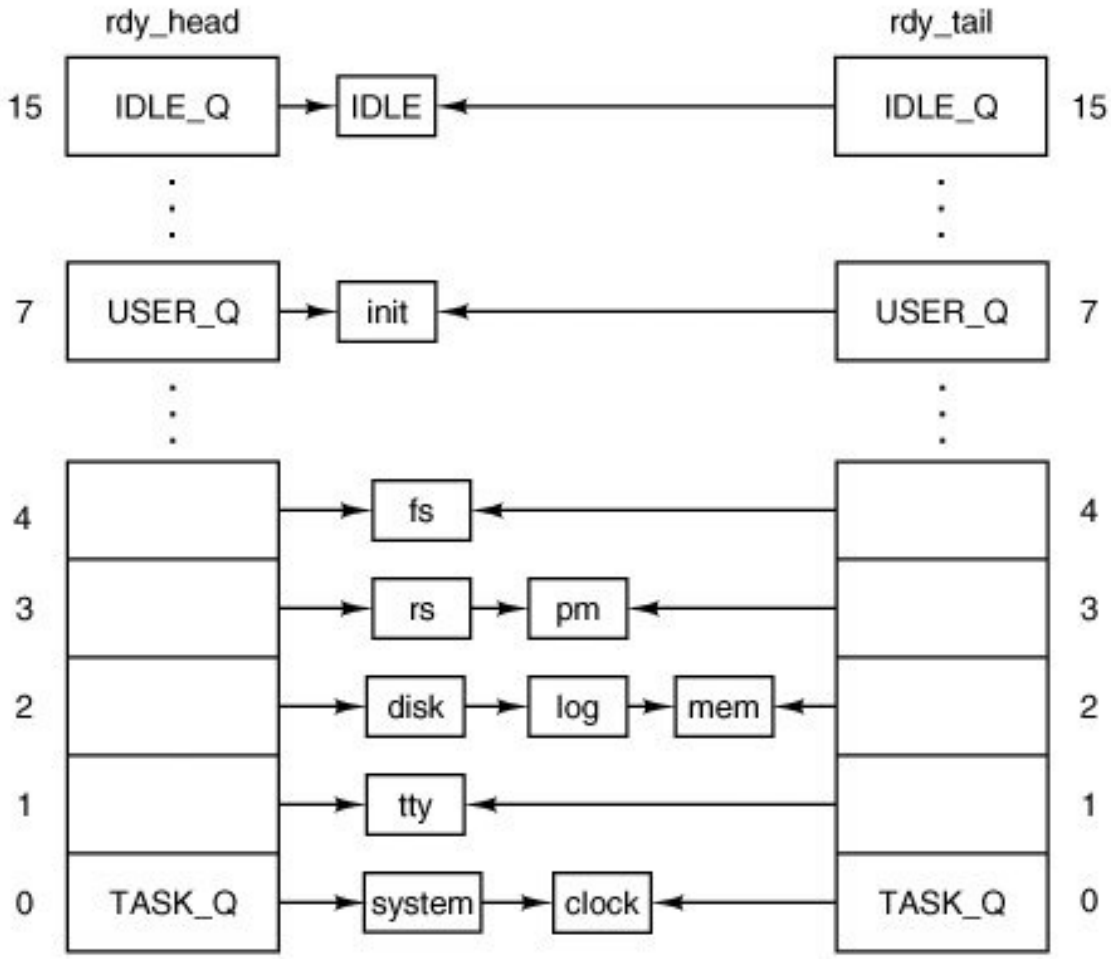


试图向进程0发送消息的进程被排队, 形成一个链表,链表头由进程0的p_caller_q域记录

MINIX3进程实现

- 源码组织与编译
- 头文件
- 进程数据结构
- 系统引导
- 系统初始化
- 中断处理
- 进程间通信
- 进程调度

多个级别的运行进程队列



MINIX启动后的初始进程队列

实现 (proc.c)

- enqueue
  - 将进程加入到某级运行队列中
    - 调用sched
    - 根据需要修改rdy_head, rdy_tail
- dequeue
  - 进程状转为非就绪态时，需要调用
  - 将阻塞的进程及受其影响也变为阻塞的进程移出队列
- sched
  - 确定一新就绪的进程放在哪个队列上，是放其首还是其尾
- pick_proc
  - 设置下一个运行进程，即设置next_ptr的值

第二章进程管理 提纲

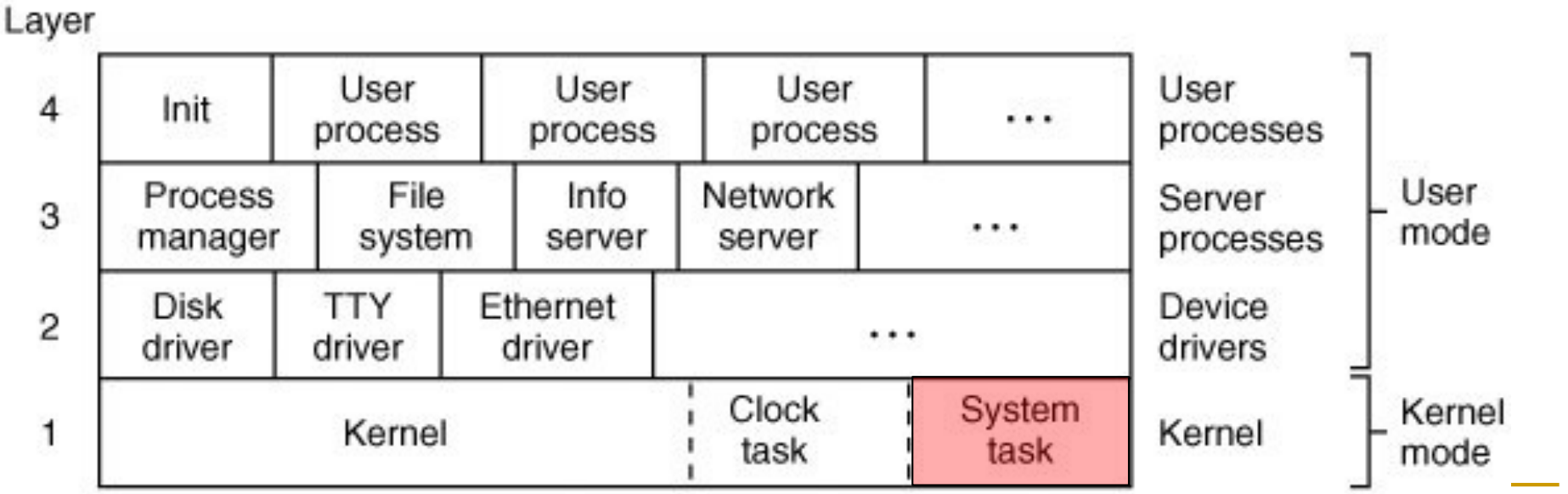
- 2.1 进程
- 2.2 进程间通信
- 2.3 经典并发问题
- 2.4 进程调度
- 2.5 MINIX3进程概述
- 2.6 MINIX3进程实现
- 2.7 MINIX3系统任务
- 2.8 MINIX3时钟任务

2.7 MINIX3系统任务

- 系统任务概述
- 系统任务实现
- 系统库的实现

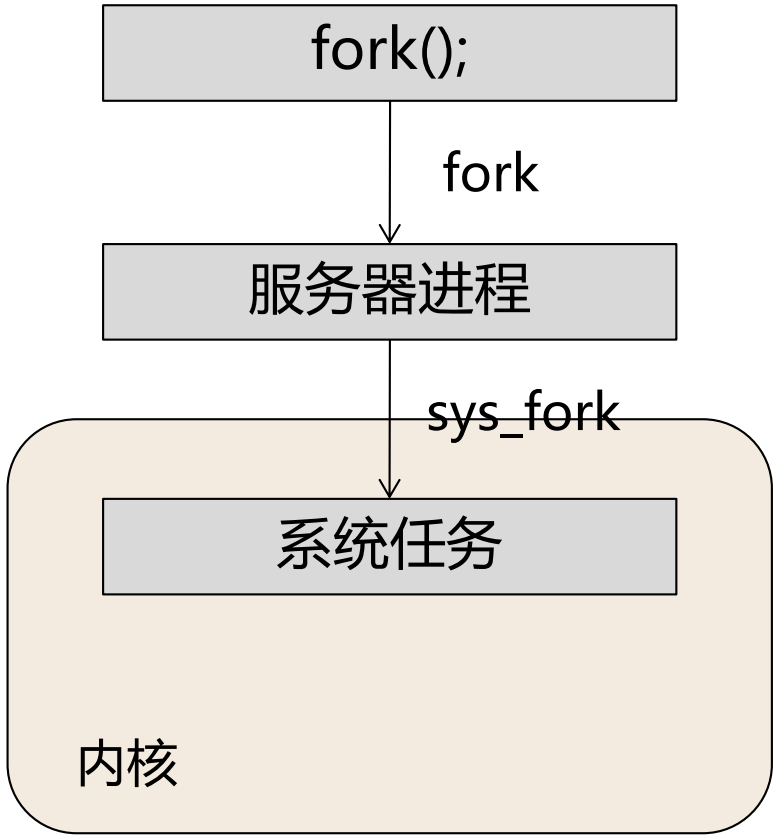
系统任务

- 系统任务作为内核中的一个进程被独立调度
- 系统任务接收所有来自驱动程序和服务器层进程的相关服务请求
  - 读写I/O端口、跨地址空间复制数据等



系统调用

- 在整体式内核的操作系统中，系统调用是指对内核提供服务的调用
- 在MINIX3中，用户进程发出的系统调用将被转换为发往服务器进程的消息，服务器进程处理部分工作后，将给系统任务进程发消息，然后由系统任务完成剩下的工作
- 系统任务接收对于内核服务的请求，称为内核调用



系统任务接收的消息类型(1)

- 系统任务接收28条消息，每一种消息对应于一个内核调用
  - 进程管理类
    - sys_fork, sys_exec, sys_exit, sys_trace,与POSIX系统调用相对应
    - sys_nice改变进程的属性
    - sys_privctl改变进程特权，被再生服务器使用
  - 信号类
    - sys_kill与系统调用kill相对应
    - sys_getksig, sys_endksig, sys_sigsend用于协助处理信号

系统任务接收的消息类型(2)

- 系统任务接收28条消息，每一种消息对应于一个内核调用
  - 内存类
    - sys_newmap 用于更新进程表的内核部分
    - sys_segctl, sys_memset 用于访问进程数据空间之外的内存
      - 当一个新进程被创建时，进程管理器使用它清理内存，以免新进程可以读取其它进程遗留数据
    - sys_umap 用于虚拟地址转换物理地址
    - sys_vircopy, sys_physcopy 用于使用虚拟地址或物理地址进行内存复制
  - 时间类
    - sys_times 对应于time, sys_setalarm设置一个定时器
  - 系统控制
    - sys_abort 在正常关机请求或系统崩溃后，进程管理器产生
    - sys_getinfo 响应对内核信息的不同请求

2.7 MINIX3系统任务

- 系统任务概述
- 系统任务实现
- 系统库的实现

实现代码

- system.h, system.c
  - 系统任务的主框架
  - 通过函数指针数组call_vec，将消息类型(用数字表示)作为数组索引，确定相应的函数以响应服务请求
  - 系统任务的顶层是sys_task过程，其在初始化函数指针数组之后，进入一个循环以等待消息。在对消息做有效性验证之后，根据消息类型调用相应的函数。
- kernel/system 目录
  - 包含了实现每个响应函数的源文件

```
09756   PUBLIC void sys_task()
09757   {
09758   /* Main entry point of sys_task.  Get the message and dispatch on type. */
09759     static message m;
09760     register int result;
09761     register struct proc *caller_ptr;
09762     unsigned int call_nr;
09763     int s;
09764
09765     /* Initialize the system task. */
09766     initialize();
09767
09768     while (TRUE) {
09769         /* Get work. Block and wait until a request message arrives. */
09770         receive(ANY, &m);
09771         call_nr = (unsigned) m.m_type - KERNEL_CALL;
09772         caller_ptr = proc_addr(m.m_source);
09773
09774         /* See if the caller made a valid request and try to handle it. */
09775         if (! (priv(caller_ptr)->s_call_mask & (1<<call_nr))) {
09776             kprintf("SYSTEM: request %d from %d denied.\n", call_nr,m.m_source);
09777             result = ECALLDENIED;                   /* illegal message type */
09778         } else if (call_nr >= NR_SYS_CALLS) {           /* check call number */
09779             kprintf("SYSTEM: illegal request %d from %d.\n", call_nr,m.m_source);
09780             result = EBADREQUEST;                   /* illegal message type */
09781         }
09782         else {
09783             result = (*call_vec[call_nr])(&m);    /* handle the kernel call */
09784         }
```

[Page 757]

```
09785
09786       /* Send a reply, unless inhibited by a handler function. Use the kernel
09787        * function lock_send() to prevent a system call trap. The destination
09788        * is known to be blocked waiting for a message.
09789        */
09790       if (result != EDONTREPLY)  {
09791           m.m_type  = result;                   /* report status of call */
09792           if (OK != (s=lock_send(m.m_source, &m))) {
09793               kprintf("SYSTEM, reply to %d failed: %d\n", m.m_source, s);
09794           }
09795       }
09796     }
```

2.7 MINIX3系统任务

- 系统任务概述
- 系统任务实现
- 系统库的实现

系统库的实现

- 在系统任务主程序system.c中每一个名字为do_xyz形式的函数的源码在kernel/system/do_xyz.c文件中
- 这些函数可以分为两大类
  - 代表用户态进程访问内核数据结构，如：system/do_setalarm.c
  - 一些系统调用，大部分工作在用户空间进程处理，需要在内核空间下作些操作，实现这些操作的函数，如：system/do_exec.c

初始化：建立消息与函数库的关联

```
09822    for (i=0; i<NR_SYS_CALLS; i++) {
09823         call_vec[i] = do_unused;
09824    }
09825
09826    /* Process management. */
09827    map(SYS_FORK, do_fork);              /* a process forked a new process */
09828    map(SYS_EXEC, do_exec);              /* update process after execute */
09829    map(SYS_EXIT, do_exit);              /* clean up after process exit */
09830    map(SYS_NICE, do_nice);              /* set scheduling priority */
09831    map(SYS_PRIVCTL, do_privctl);        /* system privileges control */
09832    map(SYS_TRACE, do_trace);            /* request a trace operation */
09833
09834    /* Signal handling. */
09835    map(SYS_KILL, do_kill);              /* cause a process to be signaled */
09836    map(SYS_GETKSIG, do_getksig);        /* PM checks for pending signals */
09837    map(SYS_ENDKSIG, do_endksig);        /* PM finished processing signal */
09838    map(SYS_SIGSEND, do_sigsend);        /* start POSIX-style signal */
09839    map(SYS_SIGRETURN, do_sigreturn);    /* return from POSIX-style signal */
09840
09841    /* Device I/O. */
09842    map(SYS_IRQCTL, do_irqctl);          /* interrupt control operations */
09843    map(SYS_DEVIO, do_devio);            /* inb, inw, inl, outb, outw, outl */
09844    map(SYS_SDEVIO, do_sdevio);          /* phys_insb, _insw, _outsb, _outsw */
```

[Page 758]

```
09845    map(SYS_VDEVIO, do_vdevio);          /* vector with devio requests */
09846    map(SYS_INT86, do_int86);            /* real-mode BIOS calls */
09847
09848    /* Memory management. */
09849    map(SYS_NEWMAP, do_newmap);          /* set up a process memory map */
09850    map(SYS_SEGCTL, do_segctl);          /* add segment and get selector */
09851    map(SYS_MEMSET, do_memset);          /* write char to memory area */
```

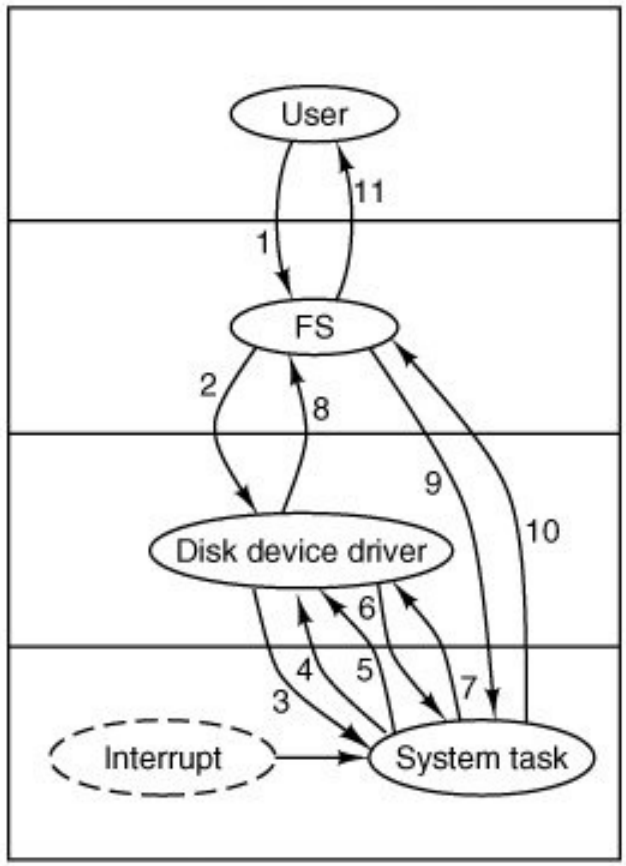
例: exec -> do_exec

- 进程管理器完成exec系统调用的大部分工作
- 为设置进程表内核数结构中的栈指针，由do_exec来处理
  - 栈指针被设置在进程表的内核部分
  - 如果该进程使用到了附加段，将调用函数phy_memeset，以擦除在这个内存区域中可能留有先前应用中的残留数据
- 调用exec的进程, 发送消息给进程管理器后阻塞
  - 不同于其它系统调用，响应结果将解除阻塞, exec没有响应
  - do_exec为进程解除阻塞，并使其成为就绪

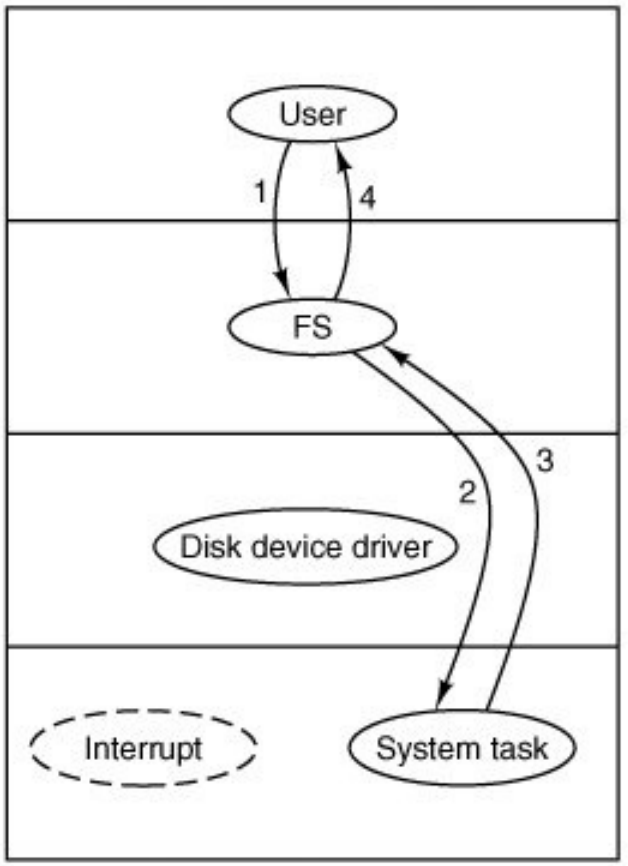
实例：系统任务的角色

- read系统调用
  - 最坏情况：用户执行read调用后，文件系统检查它的缓冲区以确定是否有所需要的数据，发现没有后，它会向磁盘驱动程序发送消息，以请求将该数据块加载到缓冲区，然后文件系统发送一条消息给系统任务以通知将该数据块复制到用户进程
  - 最好情况：用户执行read调用后，文件系统检查它的缓冲区以确定是否有所需要的数据，发现有，于是请求系统任务将数据复制给用户，系统任务复制数据

实例：系统任务的角色



(a)



(b)