Overview of MINIX I/O Software

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE351

MINIX I/O and Layered Structure
MINIX Interrupt Handlers

◆ Most interrupt handlers generate and send *wake-up* messages for blocked device tasks, as described in Ch 2

◆ For Disk devices, the handler may be as simple as:
  
  ```c
  w_status = in_byte(w_wn->base+REG_STATUS);
  interrupt(WINCHESTER);
  return 1;
  ```

◆ However, not all work this way due to the message passing overhead of this methodology.

Clock Handlers

◆ Clock Handler does intermediate work to reduce message passing overhead
  
  » Accumulates ticks in `pending_ticks`
  
  » Sends message to clock task when
    
    ♦ An alarm expires, or
    
    ♦ Scheduling change required (quantum expires)

◆ If the handler doesn’t not notify the clock task of every clock tick, does that mean the clock is not accurate?
Keyboard Handler and other Terminal Device Interrupt Handlers

- Sends no messages!
- Reads data from keyboard and filters events
  - How?
  - What is an event?
- Adds significant events/codes to a buffer and updates `tty_timeout` (i.e., clears it)
- Clock handler sends message to the terminal task when `tty_timeout` expires
- TTY task processes the queue of keyboard events and all other terminal device queues as well (e.g., RS-232)

Device Drivers in MINIX

- Separate I/O task (device driver) for each class of I/O devices
- Communicate via the file system
- Simple drivers are in their own file
- More complex drivers are subdivided into device dependent (e.g., RAM Disk, hard disk, floppy disk, and terminal) code and device independent/common code (driver.c or tty.c)
- Still separate task for each type of device
  - Why?
- Device drivers are linked into the kernel
  - Why?
### Process Structured vs. Monolithic Structured

(a) Process-structured system
- Processes
- User process
- File system
- Device driver

1–4 are request and reply messages between three independent processes.

(b) Monolithic system
- User-space part
- Device driver
- File system
- A process

The user-space part calls the kernel-space part by trapping. The file system calls the device driver as a procedure. The entire operating system is part of each process.

### Generic Message Formats

#### Requests

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.m_type</td>
<td>int</td>
<td>Operation requested</td>
</tr>
<tr>
<td>m.DEVICE</td>
<td>int</td>
<td>Minor device to use</td>
</tr>
<tr>
<td>m.PROC_NR</td>
<td>int</td>
<td>Process requesting the I/O</td>
</tr>
<tr>
<td>m.COUNT</td>
<td>int</td>
<td>Byte count or ioctl code</td>
</tr>
<tr>
<td>m.POSITION</td>
<td>long</td>
<td>Position on device</td>
</tr>
<tr>
<td>m.DEVICE</td>
<td>int</td>
<td>Minor device to use</td>
</tr>
</tbody>
</table>

#### Replies

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.m_type</td>
<td>int</td>
<td>Always TASK_REPLY</td>
</tr>
<tr>
<td>m.REP_PROC_NR</td>
<td>int</td>
<td>Same as PROC_NR in request</td>
</tr>
<tr>
<td>m.REP_STATUS</td>
<td>int</td>
<td>Bytes transferred or error number</td>
</tr>
</tbody>
</table>
Generic Device Driver Structure

```c
message mess; /* message buffer */
void io_task() {
    initialize(); /* only done once */
    while(TRUE){
        receive(ANY, &mess); /* wait for a request for work */
        caller = mess.source; /* process sending msg */
        switch(mess.type){
            case READ: rcode = dev_read(&mess);break;
            case WRITE: rcode = dev_write(&mess);break;
            /* Other cases go here, e.g., OPEN, CLOSE, IOTCTL */
            default: rcode = ERROR;
        }
        mess.type = TASK_REPLY;
        mess.status = rcode; /* result code */
        send(caller, &mess); /* send reply to caller */
    }
}
```

Block Devices in MINIX

- MINIX always has at least three block device tasks compiled into the kernel:
  - RAM disk driver
  - Floppy disk driver
  - Hard disk driver(s)
- Each block device driver does device specific initialization and then calls a shared I/O function that implements the main loop
  - A data structure that points to the device specific routines to handle reads, writes, etc. is passed as an input parameter
MINIX Main I/O Loop
Block Device Shared Function

message mess; /* message buffer */
void shared_io_task(struct driver_table *entry_points) {
  /* initialization is done before calling this routine */
  while(TRUE){
    receive(ANY, &mess); /* wait for a request for work */
    caller = mess.source; /* process sending msg */
    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, e.g., OPEN, CLOSE, IOTCTL */
      default:  rcode = ERROR;
    }
    mess.type = TASK_REPLY;
    mess.status = rcode; /* result code */
    send(caller, &mess); /* send reply to caller */
  }
}

Six Operations Supported by
MINIX Block Device Drivers

1. OPEN
2. CLOSE
3. READ
4. WRITE
5. IOCTL
6. SCATTERED_IO
Common Block Device SW

- The driver structure that contains the pointers to device specific routines is defined in `driver.h`
- The main loop (shared I/O function) is defined in `driver.c`
  - It does not return to the caller
- Device specific code is in separate files
  - `at_wini.c`
  - `floppy.c`
  - `memory.c`

Driver Library

- “Files `drvlib.h` and `drvlib.c` contain system-dependent code that supports disk partitions on IBM PC compatible computers.”
- Reasons to partition a disk:
  - Large disks are cheaper/byte than small disks
    - Use one disk for multiple OS rather than use two disks
  - Put different file system types (for different OS) on one disk
  - OS disk size limits, e.g., 1-GB file system limit
  - Convenient to put a portion of a file system in its own partition