CS 167: I/O and Booting
This diagram is not intended to be all-inclusive, but suggestive of what the major OS components are.
Examples of hardware-dependent portions of an OS are device drivers, interrupt management, power management, etc.
OS Components: Importance
OS Components: Flow of Control

Applications → OS Kernel

System calls

OS Kernel → External Hardware

Kernel threads

External Hardware

Interrupts
To Be Discussed

- What is the functionality of the components?
- What are the key data structures?
- How is the system broken up into modules?
- To what extent is the system extensible?
- What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- In which execution contexts do the various activities take place?
Finding Devices

unix etc home pro dev

disk1 disk2 ...

device number:
major = 6
minor = 1

cdevsw

read entry point
write entry point
...
Discovering Devices

- You plug in a new device to your computer …
  - OS must notice
    - must find a device driver
      • what kind of device is it?
      • where is the driver?
    - must assign a name
      • how chosen?
    - multiple similar devices
      • how does application choose?
Computer Terminal
The photo shows a Teletype terminal being used on an early Unix system by Ken Thompson. The photo is from http://histoire.info.free.fr/images/pdp11-unix.jpeg, but it is probably owned by Lucent Technologies. “Teletype” is the word from which tty is derived. (According to Wikipedia (August 25, 2010) “Teletype” was a trademark of the Teletype Corporation, but the company no longer exists.)
The image is from http://filmdoctor.co.uk/2013/03/26/the-brit-list-2013/.
Terminals

• Long obsolete, but still relevant
• Issues
  1) characters are generated by the application faster than they can be sent to the terminal
  2) characters arrive from the keyboard even though there isn't a waiting read request from an application
  3) input characters may need to be processed in some way before they reach the application
Physical terminals are rarely used these days ...
Pseudo terminals are currently used (for example, for character I/O to windows on a display).
Network Communication

• Multiple protocol modules
  – pass blocks from one module to the next without copying
  – append headers to the beginning of outgoing packets; remove headers from incoming packets
  – hold on to packets for possible retransmission
  – request and respond to time-out notifications
Two Queued Segments
Passed to the Next Module ...
In this section we address the area of input and output (I/O). We discuss two basic I/O architectures and talk about the fundamental I/O-related portion of an operating system—the device driver.
A very simple I/O architecture is the *memory-mapped* architecture. Each device is controlled by a controller and each controller contains a set of registers for monitoring and controlling its operation. In the memory-mapped approach, these registers appear to the processor as if they occupied physical memory locations. In reality, each of the controllers is connected to a *bus*. When the processor wants to access or modify a particular location, it broadcasts the address on the bus. Each controller listens for a fixed set of addresses and, if it finds that one of its addresses has been broadcast, then it pays attention to what the processor would like to have done, e.g., read the data at a particular location or modify the data at a particular location. The memory controller is a special case. It passes the bus requests to the actual primary memory. The other controllers respond to far fewer addresses, and the effect of reading and writing is to access and modify the various controller registers.

There are two categories of devices, *programmed I/O (PIO)* devices and *direct memory access (DMA)* devices. In the former, I/O is performed by reading or writing data in the controller registers a byte or word at a time. In the latter, the controller itself performs the I/O: the processor puts a description of the desired I/O operation into the controller’s registers, then the controller takes over and transfers data between a device and primary memory.
## PIO Registers

<table>
<thead>
<tr>
<th>GoR</th>
<th>GoW</th>
<th>IER</th>
<th>IEW</th>
<th>Control register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RdyR</th>
<th>RdyW</th>
<th>Status register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Read register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- GoR: Go read (start a read operation)
- GoW: Go write (start a write operation)
- IER: Enable read-completion interrupts
- IEW: Enable write-completion interrupts
- RdyR: Ready to read
- RdyW: Ready to write
The sequence of operations necessary for performing PIO is outlined in the picture. One may choose to perform I/O with interrupts disabled, you must check to see if I/O has completed by testing the ready bit. If you perform I/O with interrupts enabled, then an interrupt occurs when the operation is complete. The primary disadvantage of the former technique is that the ready bit is typically checked many times before it is discovered to be set.
## DMA Registers

<table>
<thead>
<tr>
<th>Go</th>
<th>IE</th>
<th>Op Code</th>
<th>Control register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rdy</td>
<td></td>
<td></td>
<td>Status register</td>
</tr>
</tbody>
</table>

### Memory address register

### Device address register

### Legend:
- **Go**: Start an operation
- **Op Code**: Operation code (identifies the operation)
- **IE**: Enable interrupts
- **Rdy**: Controller is ready
For I/O to a DMA device, one must put a description of the desired operation into the controller registers. A disk request on the simulator typically requires two operations: one must first perform a seek to establish the location on disk from or to which the transfer will take place. The second step is the actual transfer, which specifies that location in primary memory to or from which the transfer will take place.
A device driver is a software module responsible for a particular device or class of devices. It resides in the lowest layers of an operating system and provides an interface to other layers that is device-independent. That is, the device driver is the only piece of software that is concerned about the details of particular devices. The higher layers of the operating system need only pass on read and write requests, leaving the details to the driver. The driver is also responsible for dealing with interrupts that come from its devices.
This snippet of C++ specifies the interface to a generic device driver, as shown in the previous slide.
A more realistic device driver has an asynchronous interface: one calls the `start_read` and `start_write` routines to start operations. At a later time, one calls the `wait` routine to wait for a particular operation to terminate.

```cpp
class device {
public:
    virtual handle_t start_read(request_t);
    virtual handle_t start_write(request_t);
    virtual status_t wait(handle_t);
    virtual status_t interrupt();
};
```
The photo is from http://www.pdp8.net/pdp8i/pdp8i.shtml.
The code in the slide was “toggled in” to the computer by hand. It would load a program provided on a paper tape, then run that program.
VAX-11/780 Boot

- Separate “console computer”
  - LSI-11
  - read boot code from floppy disk
  - load OS from root directory of first file system on primary disk
Configuring the OS (1)

- Early Unix
  - OS statically linked to contain all needed device drivers
    - all device-specific info included with drivers
  - disk drivers contained partitioning description
Configuring the OS (2)

- Later Unix
  - OS statically linked to contain all needed device drivers
    - at boot time, OS would probe to see which devices were present and discover device-specific info
  - partition table in first sector of each disk
The photo is from wikipedia.
Issues

• Open architecture
  – large market for peripherals, most requiring special drivers
  – how to access boot device?
  – how does OS get drivers for new devices?
The Answer: BIOS

- Basic Input-Output System
  - code stored in *non-volatile RAM*
    - CMOS, flash, whatever …
  - configuration data also in NV RAM
    - including set of boot-device names
  - three primary functions
    - power-on self test (POST)
    - load and transfer control to boot program
    - provide drivers for all devices
  - main BIOS on motherboard
  - additional BIOSes on other boards
POST

• On power-on, CPU executes BIOS code
  – located in last 64k of first megabyte of address space
  – initializes hardware
  – counts memory locations
The master boot record (MBR) is stored in the first sector of each disk. This diagram is based on http://www.ibm.com/developerworks/library/l-linuxboot/index.html. BIOS loads the MBR into memory, then passes control to the boot program. It scans the partition table to find one that’s marked as the boot partition, then loads its first sector into memory and passes control to it. This code then loads the initial kernel image from the partition into memory and passes control to it. All disk I/O is performed by calling back to BIOS code.

BIOS is invoked by the x86’s “software interrupt” procedure: the caller uses the x86 *int* (interrupt) instruction, which simulates the occurrence of an interrupt, passing control through an interrupt vector to the appropriate BIOS code. For the case of getting to device drivers, *int 13* is called.
Linux Booting (1)

- Two stages of booting provided by one of:
  - lilo (Linux Loader)
    - uses sector numbers of kernel image
  - grub (Grand Unified Boot Manager)
    - understands various file systems
  - both allow dual (or greater) booting
    - select which system to boot from menu
  - perhaps choice of Linux or Windows
The first `startup_32` routine is in `arch/i386/boot/compress/head.S`; the second is in `arch/i386/kernel/head.S`. The `startup_kernel` routine is in `init/main.c`. 

<table>
<thead>
<tr>
<th>Linux Booting (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>assembler code</strong></td>
</tr>
<tr>
<td>(startup_32)</td>
</tr>
<tr>
<td>• Kernel image is compressed</td>
</tr>
<tr>
<td>– step 1: set up stack, clear BSS, \n  uncompressed kernel, then transfer \n  control to it</td>
</tr>
<tr>
<td><strong>assembler code</strong></td>
</tr>
<tr>
<td>(different startup_32)</td>
</tr>
<tr>
<td>• Process 0 is created</td>
</tr>
<tr>
<td>– step 2: set up initial page tables, \n  turn on address translation</td>
</tr>
<tr>
<td><strong>C code</strong></td>
</tr>
<tr>
<td>(start_kernel)</td>
</tr>
<tr>
<td>• Do further initialization</td>
</tr>
<tr>
<td>– step 3: initialize rest of kernel, \n  create init process (#1)</td>
</tr>
<tr>
<td>– invoke scheduler</td>
</tr>
</tbody>
</table>
Beyond BIOS

- BIOS
  - designed for 16-bit x86 of mid 1980s

- Open Firmware
  - designed by Sun
  - portable
    - drivers, boot code in Forth
  - compiled into bytecode
  - used on non-Intel systems

- UEFI (Unified Extensible Firmware Interface)
  - improved BIOS originally from Intel
  - also uses bytecode