Operating Systems

I/O

Overview

- Devices and Controllers
- I/O Subsystem
- Device Drivers

I/O Controllers

- A device controller is attached to the system or integrated into the motherboard or SoC
- The peripheral itself attaches to the controller
  – RS-232, SCSI, SATA, SAS, USB
- Convert a string of bits into bytes or blocks of bytes
  – Even disks are strings of bits
- The controller has registers mapped into memory
  – Read and written to control and check status

I/O Ports vs Memory Mapping

- I/O ports in a dedicated namespace
- Accessed with special I/O instructions

```
outb  %al,$18
```

- As opposed to memory space which is accessed with standard load, store, move

```
$ cat /proc/ioports
0000-0cf7 :PCI Bus
0000:00  0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-0060 : keyboard
0064-0064 : keyboard
0070-0071 : rtc0
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
02f8-02ff : serial
0378-037a : parport0
03c0-03df : vga+
03f2-03f2 : floppy
```

The I/O Subsystem

- Devices have a complicated low-level interface
  - Control and data registers mapped into memory
  - Header files and documentation to understand registers and bit fields
- Goal: provide a high level interface so that programs don't have to be rewritten
  - I/O devices do mostly the same thing – they input, they output
  - Design: a small set of of abstract routines can encapsulate various devices

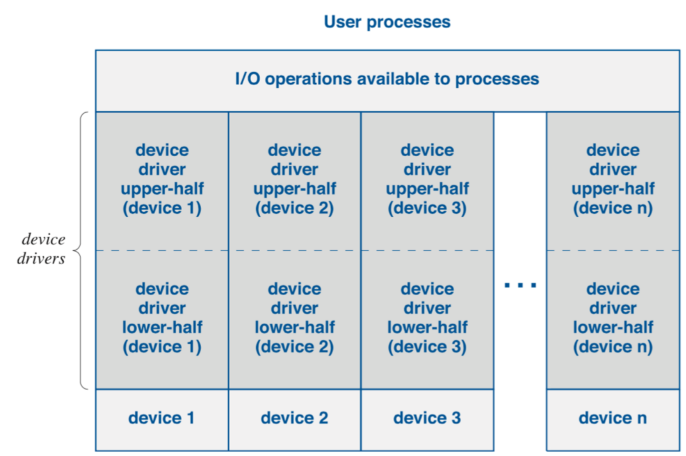
I/O Interfaces

- Another purpose of the I/O interface is to protect shared I/O resources (devices, buffers)
- Safe and fair access
- Policies can be applied at the high level interfaces and can be generalized over various devices
- The Unix abstraction is that "everything is a file"

Organization of the I/O Subsystem

- High level I/O functions to abstract the details of hardware and provide general entry points
  - Design challenge is to capture diversity of devices with a generic interface
- Device drivers interact with specific devices
- Drivers have a upper and lower half
- The upper half interacts with process requests
- The lower half responds to interrupts with handler functions
  - service interrupts, initiate new operations as necessary

Organization of the I/O Subsystem



I/O and Driver Abstractions

- Synchronous vs Asynchronous
  - Synchronous: the requesting process is blocked until I/O completes – easier to program
  - Asynchronous: the process can continue to execute – more control of overlap of communication and computation
- Asynchronous I/O interfaces must notify the process
  - Deliver a signal
  - Spawn a thread
  - Check (poll) the status of an I/O action, or read from a queue of completion actions

POSIX AIO

- Allows applications to initiate one or more asynchronous I/O operations
  - signal, instantiate a thread, no notification
- aio_read(struct aiocb *aiocbp), aio_write()
- aio_return() – to check the return status of an AIO operation

```
struct aiocb { /* The order of these fields is implementation-dependent */
       int aio_fildes;            /* File descriptor */
       off_t aio_offset;          /* File offset */
       volatile void *aio_buf;    /* Location of buffer */
       size_t aio_nbytes;         /* Length of transfer */
       int aio_reqprio;           /* Request priority */
       struct sigevent aio_sigevent;    /* Notification method */
.....
```

I/O and Driver Abstractions

- Format and size of data transfers
  - Bytes, strings, blocks
- Block vs character interfaces
  - Look in /dev on your favorite *nix system
  - The question is whether chunks of data (blocks) are independently addressable
- Buffering can be used to adapt between the two
- How much state is preserved between requests?
  - Specify a starting point and read successive blocks or specify a block with each read?

Abstract I/O Interface

Operation	Purpose
close	**Terminate use of a device**
control	**Perform operations other than data transfer**
getc	**Input a single byte of data**
init	**Initialize the device at system startup**
open	**Prepare the device for use**
putc	**Output a single byte of data**
read	**Input multiple bytes of data**
seek	**Move to specific data (usually a disk)**
write	**Output multiple bytes of data**

Open, Read, Write, Close

- Common paradigm (Xinu, Unix, Windows)
- Before a process can use a device, it must open it
  - Manage exclusive access
  - Check permissions
  - Set up state in system data structures
- Close when finished
  - Clean up state
  - The device could be powered down

Control

- Control interface allows for configuration of device driver parameters
- Can also manage device-specific interactions that are not possible with the standard interfaces
  - Buffering or caching behavior
- ioctl() on Unix

Binding Operations and Devices

- Abstract interfaces need to act on specific devices
- Must be mapped to device driver functions
- The OS provides a virtual I/O environment, passing operations through to devices via drivers
- Unix embeds devices in the filesystem, providing names to specific devices
- General-purpose OSes construct this dynamically, but embedded systems often statically configure it

Device Names in Xinu

- Specify a set of devices when the system is configured
- Assign an integer device descriptor
- For instance, CONSOLE is device 0
- Programs don’t need to be rewritten when devices change, but the system does need to be reconfigured and recompiled

Xinu's Device Switch Table

- The OS must forward I/O operations to the correct driver function
- The device ID is used as an index into a table of device-specific functions
- Each entry in the table contains information about the device and function pointers to functions that implement operations
- To write to a device, find the device entry and invoke the specific write function
- Xinu's approach is simple but is fundamentally the same as e.g. Unix

Xinu Example

	open	close	read	write	getc	
CONSOLE	conopen	conclose	conread	conwrite	congetc	
ETHER	ethopen	ethclose	ethread	ethwrite	ethgetc	. . .
DISK	dskopen	dskclose	dskread	dskwrite	dskgetc	
			⋮			

- Uniform interface hiding the differences of underlying hardware

Multiple Instances of a Device

- Multiple instances of a device can share a driver
- Multiple instances in the device table that are largely the same, differing in only a few aspects
- Each instance will have its own control and status registers
- Can also be distinguished by the "minor" device number

Device Table Entry

```
/* Device table entry */
struct  dentry  {
  int32   dvnum;
  int32   dvminor;
  char    *dvname;
  devcall (*dvinit) (struct dentry *);
  devcall (*dvopen) (struct dentry *, char *, char *);
  devcall (*dvclose)(struct dentry *);
  devcall (*dvread) (struct dentry *, void *, uint32);
  devcall (*dvwrite)(struct dentry *, void *, uint32);
  devcall (*dvseek) (struct dentry *, int32);
  devcall (*dvgetc) (struct dentry *);
  devcall (*dvputc) (struct dentry *, char);
  devcall (*dvcntl) (struct dentry *, int32, int32, int32);
  void    *dvcsr;
  void    (*dvintr)(void);
  byte    dvirq;
};
```

Some Devices

```
extern  struct  dentry  devtab[]; /* one entry per device */

/* Device name definitions */

#define CONSOLE     0       /* type tty      */
#define NULLDEV     1       /* type null     */
#define ETHER0      2       /* type eth      */
#define NAMESPACE   3       /* type nam      */
#define RDISK       4       /* type rds      */
#define RAM0        5       /* type ram      */
#define RFILESYS    6       /* type rfs      */
#define RFILE0      7       /* type rfl      */
#define RFILE1      8       /* type rfl      */
#define RFILE2      9       /* type rfl      */
#define RFILE3      10       /* type rfl      */
#define RFILE4      11       /* type rfl      */
#define RFILE5      12       /* type rfl      */
#define RFILE6      13       /* type rfl      */
```

read()

```
syscall read(
    did32   descrp,   /* Descriptor for device  */
    char    *buffer,  /* Address of buffer     */
    uint32  count    /* Length of buffer    */
  )
{
  intmask   mask;    /* Saved interrupt mask   */
  struct dentry *devptr;   /* Entry in device switch table */
  int32   retval;    /* Value to return to caller  */

  mask = disable();
  if (isbaddev(descrp)) {
    restore(mask);
    return SYSERR;
  }
  devptr = (struct dentry *) &devtab[descrp];
  retval = (*devptr->dvread) (devptr, buffer, count);
  restore(mask);
  return retval;
}
```

control()

```
syscall control(
    did32   descrp,   /* Descriptor for device  */
    int32   func,   /* Specific control function  */
    int32   arg1,   /* Specific argument for func */
    int32   arg2    /* Specific argument for func */
  )
{
  intmask   mask;    /* Saved interrupt mask    */
  struct dentry *devptr;  /* Entry in device switch table */
  int32   retval;    /* Value to return to caller  */

  mask = disable();
  if (isbaddev(descrp)) {
    restore(mask);
    return SYSERR;
  }
  devptr = (struct dentry *) &devtab[descrp];
  retval = (*devptr->dvcntl) (devptr, func, arg1, arg2);
  restore(mask);
  return retval;
}
```

open() and close()

- Implemented identically to read() and control()
- Explicit opening and closing allows the system to maintain a *reference count* of processes using a device
  - Again, the system may power a device down when not in use

Null Entries in Devtab

- Note that each of the high-level functions calls the device-specific function without checking its validity
- Not all operations make sense on all devices
  - You can't seek on the console, or getc() on a network device
- ionull() returns OK
- ioerr() returns SYSERR

Initialization

- General operating systems can dynamically initialize devices
  - recall the discussion of USB devices
- Embedded systems like Xinu use static configuration
- Xinu specifies devices and functions in a file called Configuration, and generates a C file and a header with appropriate values

Configuration

```
/* Entries for a device specify the functions that handle each of the */
/* high-level I/O operations as follows:       */
/*                   */
/*  -i    init  -o    open  -c    close     */
/*  -r    read  -w    write -s    seek      */
/*  -g    getc  -p    putc  -n    control     */
/*  -intr int_hndlr -csr  csr -irq  irq     */
/*                   */
/*************************************************************************/

/* Type Declarations for both real- and pseudo- device types */

/* type of a null device */
null:
  on nothing
    -i ionull -o ionull -c ionull
    -r ionull -g ionull -p ionull
    -w ionull -s ioerr
```

Configuration

```
/* type of a tty device */
tty:
  on uart
    -i ttyinit      -o ionull       -c ionull
    -r ttyread      -g ttygetc      -p ttyputc
    -w ttywrite     -s ioerr        -n ttycontrol
    -intr ttyhandler

/* type of a ethernet device */
eth:
  on am335x_eth
    -i ethinit  -o ioerr  -c ioerr
    -r ethread  -g ioerr  -p ioerr
    -w ethwrite -s ioerr  -n ethcontrol
    -intr ethhandler
```

conf.c

```c
struct  dentry  devtab[NDEVS] =
{
/**
 * Format of entries is:
 * dev-number, minor-number, dev-name,
 * init, open, close,
 * read, write, seek,
 * getc, putc, control,
 * dev-csr-address, intr-handler, irq
 */

/* CONSOLE is tty */
  { 0, 0, "CONSOLE",
    (void *)ttyinit, (void *)ionull, (void *)ionull,
    (void *)ttyread, (void *)ttywrite, (void *)ioerr,
    (void *)ttygetc, (void *)ttyputc, (void *)ttycontrol,
    (void *)0x44e09000, (void *)ttyhandler, 72 },
```

conf.c

```c
/* CONSOLE is tty */
  { 0, 0, "CONSOLE",
    (void *)ttyinit, (void *)ionull, (void *)ionull,
    (void *)ttyread, (void *)ttywrite, (void *)ioerr,
    (void *)ttygetc, (void *)ttyputc, (void *)ttycontrol,
    (void *)0x44e09000, (void *)ttyhandler, 72 },

/* NULLDEV is null */
  { 1, 0, "NULLDEV",
    (void *)ionull, (void *)ionull, (void *)ionull,
    (void *)ionull, (void *)ionull, (void *)ioerr,
    (void *)ionull, (void *)ionull, (void *)ioerr,
    (void *)0x0, (void *)ioerr, 0 },

/* ETHER0 is eth */
  { 2, 0, "ETHER0",
    (void *)ethinit, (void *)ioerr, (void *)ioerr,
    (void *)ethread, (void *)ethwrite, (void *)ioerr,
    (void *)ioerr, (void *)ioerr, (void *)ethcontrol,
    (void *)0x0, (void *)ethhandler, 0 },
```

Embedded Linux - DeviceTree

- Embedded Linux uses something called DeviceTree
- A Flattened Device Tree (FDT) is shipped so that a kernel image can be configured appropriately for hardware
- Required for Linux on new ARM SoCs
- Identifies the type of CPU and describes devices in the system very similarly to what we have discussed

Summary

- Complex operating systems have more functionality between the abstract interfaces and the device drivers
- Caching
- Security and policy
- This indirection mechanism forms the basis however