

Lecture 12

Input and output

Many computer tasks are actually I/O centric: what we really want is the I/O, and the CPU processing is just incidental.

In the MOA course, you (has been or will be) taught I/O mechanisms like interrupts and DMA. In this course we study how the OS use these mechanisms to allow users to perform I/O efficiently.

References:

- [OSC] Chapter 13
- [ULK] Chapter 13 (pp. 378–393).

POS(0230A)

POS(0230A)-12.1

Objective of OS support: device sharing

- All I/O operations require kernel mode. They can only be done by the operating system, not normal processes. This makes sure that **different users will not interfere each other using a device**.
- The OS defines an interface that allow different processes to **share the same device**. E.g.,
 - A printer can be shared by using a server process which get print jobs from other processes and dispatch them to the printer one by one.
 - A disk can be shared by building a filesystem on it and letting each users to write their information under one directory.
 - A timer device can be shared by keeping a list of timeouts.
 - A sound card can be shared by mixing together the sound waves.
 - A network device can be shared by adding tags to network packets to indicate the network endpoints to which processes attach.

Efficiency issues

Since all I/O operations must be done within kernel, all the efficiency tricks are also done in the kernel.

The objective is simple: **improve the response time** of I/O operations, and **reduce CPU overhead** required for them.

- **Buffering**: read and write to temporary location in RAM to **allow for speed mismatch between I/O devices and processes**.
- **Caching**: after **read** request, **keep the result** in memory (cache) so that next read does not need I/O. After **write** request, do not write to device immediately but **wait until the device has nothing else to do**.
- **Scheduling**: for some devices like storage device, the **exact ordering of read and write operations is not (very) important**. In this case, the OS is free to reorder, or “schedule”, the operation in an order that will improve efficiency.

POS(0230A)-12.2

POS(0230A)-12.3

Different roles of hardware and OS designers

- The I/O **hardware designers** dictates **how the device operates**. For example, for reading from a hard disk,
 - How a computer asks disk reads to be performed?
 - How the computer is notified when the disk reading is completed?
 - Where the result of the read is stored?
- The **OS designers** implement the above specification, and design how processes can access the devices efficiently, and how they can share it.
- The hardware designers will design their hardware so that such OS interface can be easily made, and the CPU load is reasonably low.

General hardware interface

How to perform them?

- I/O through **ports**, by writing 8, 16 or 32 bit numbers to ports, which are themselves addressed by a short integer. Natural, and most “small” I/O uses it.
- **Memory mapped I/O**: a part of the physical memory is allocated for I/O operations. By reading and writing memory there using normal memory instructions, I/O is performed. Usually for devices with large address space, e.g., video memory.

What is being read or written?

- The **data** itself. E.g., the pixel color for a screen, the key pressed from the keyboard.
- **Control** information to operate the device. E.g., address of the memory for DMA, an indicator specifying whether a complete screen is just finished displaying.

POS(0230A)-12.4

General hardware interface (cont'd)

How is I/O completion communicated?

- **Immediate**: completion can be assumed once the port or memory reading or writing is done. Only for very fast devices.
- **Control register**: when I/O is completed, a flag is set in a control register of the device, which can be read or written from an I/O port. Either for polling, or for finding the device generating interrupts.
- **Interrupt**: an interrupt signal is sent to the CPU at I/O completion time. The CPU can then use the control register to find out which device is interrupting.

It is not very efficient to check all the devices (especially if the computer has a lot of them). So there are many “types” of interrupts, each invokes a specific handler. Thus the specific handler has fewer devices to scan for I/O completion.

POS(0230A)-12.5

General hardware interface (cont'd)

Where are the operands and results stored?

- **Device registers:** the result of an operation can be stored somewhere in the device, and can be retrieved and stored using the I/O port interface or the memory mapped interface.

Good if I/O is not very frequent, e.g., keyboard, modem, etc.

- **Direct memory access:** the device is directly connected to the memory of the computer, so that it can read and write into physical memory to fetch operands and store results.

Since memory is written directly, the CPU does not need to copy the result from device to memory. Thus CPU intervention is not needed until I/O completion.

Usually used when I/O is very frequent, e.g., disk, sound card, network card.

POS(0230A)-12.6

What the CPU (i.e., the kernel code) need to do for I/O?

For operations that won't block:

1. **Setup** the I/O by writing to the control registers.
2. **Perform** the I/O by reading or writing the data.
3. For polling, repeatedly read the control register until **completion** is notified.
4. If necessary, **reset** the I/O by writing to the control register.

For operations that would block:

1. **Setup** the I/O by writing to the control registers. Setup interrupt and DMA if needed.
2. **Perform** the I/O by reading or writing the data register.
3. Continue, or put the process into wait queue.

Interrupt handler

4. **Find** the completed operation.
5. Put the process back to **ready**.
6. If needed, **reset** the I/O by writing control registers.

POS(0230A)-12.7

Interrupt handling

Hardware interrupts must be handled quickly. E.g., for a serial device, if the CPU cannot read the old data before new data come, the new data is lost.

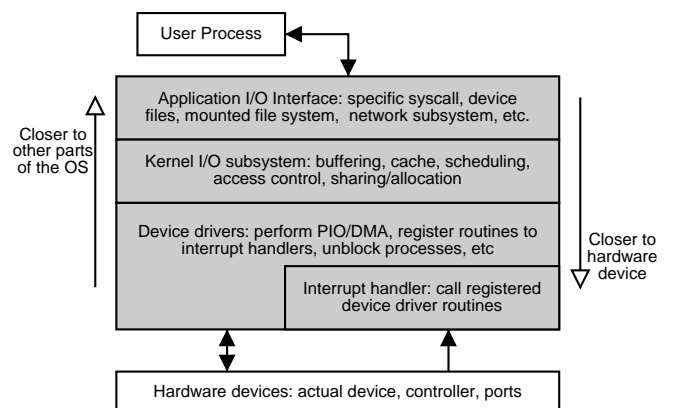
It is thus essential to **reduce the time when interrupt is disabled**, i.e., during the interrupt handler.

Typical handlers have two parts: read the device to free up the device, and move the process to ready queue, deal with the buffers and cache, etc. **Only the first part need interrupt disabling.** Two strategies:

- **Solaris:** the second part is **done in a high priority kernel threads** instead of the interrupt handler.
- **Linux:** the second part is **done in "bottom-halves"**, done only before switching back to user-mode, with interrupts enabled.

POS(0230A)-12.8

Overall picture of kernel I/O subsystem



POS(0230A)-12.9

Application I/O interface

Various ways in which processes can perform I/O:

- **Explicitly through a device file.** By calling system calls like *read* and *write* on a file mapped to a device, I/O is performed. E.g., sound card.
- **Implicitly through a mounted filesystem.** By calling system calls like *read* and *write* on a FS mounted on a storage device (e.g., disk), I/O is implicitly performed.
- **Implicitly through the use of other kernel abstractions.** Some kernel abstractions, e.g. the network, interval timer, etc., that requires the use of hardware devices, e.g., network device, timer, etc.
- **Implicitly through the use of a networked filesystem.** Same as "mounted filesystem" above, except the FS is mapped to a network "disk". Network operation is done.

POS(0230A)-12.10

Example (Unix): Device files

- Like symbolic links and FIFOs, **device files** are **special directory entries** of the filesystem.
- There are two kinds of device files: block devices and character devices. The former are for **storage devices** that **communicate a fixed number of bytes** each time. Buffers and cache are used on these devices.
In practice, this means that only disks, disk partitions and pseudo-devices emulating disks are block devices.
- Instead of storing a list of direct blocks and indirect blocks in the inode, a device file contains two numbers: **major and minor device numbers**.
- When a device is accessed, a **device driver** is found and invoked based on the major and minor device numbers.
- Intuitively, each **class of devices** has a **major device number**, so that there is only a few device drivers for each major device number.

POS(0230A)-12.11

Example: Linux major numbers

Device number	Character device	Block device
1	Memory devices (e.g., mem, null, zero, full, random)	RAM disk
2	Terminal devices (master side)	Floppy
3	Terminal devices (slave side)	IDE
4	Virtual consoles, Serial port	
6	Parallel port	
8		SCSI disks
10	Mouse, Power management	
11	Raw keyboard	SCSI CDs
13	Joystick etc	IDE
14	Sound	
65-71		SCSI disks

POS(0230A)-12.12

How does an application access device files?

- In principle, a **device file works just the same as a normal file**. The application *open()* the device, *read()* from and *write()* to it, and *close()* it at the end. Only block devices allow *seek()*.
- Unluckily, such simple file-like interactions do not cater for special needs of I/O devices to **control the device**.
- For example, if we want to read data from the */dev/dsp* device (i.e., reading sound samples), we would first want to set the sample rate, etc.
- So there is one more call specific to I/O devices: *ioctl()*. It is a "catch-all" system call: all device control interactions are done using *ioctl()*.
- Each device driver defines a set of *ioctl* operations it recognizes, and assign a number to it. E.g., the CDROM device driver recognizes the operation 0x5301 to pause a CD, 0x5309 to eject a CD, etc.
- See *ioctl_list(2)* for a complete list.

POS(0230A)-12.13

Example: ejecting a CD

The following program ejects a CD from the CD-ROM drive.

```
#include <sys/fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <unistd.h>

int main() {
    int cd_fd = open("/dev/cdrom", O_RDONLY);
    if (cd_fd == -1)
        exit(1);
    if (ioctl(cd_fd, CDROMEJECT) == -1) // This is 0x5309
        exit(2);
}
```

This is particularly simple, since the *CDROMEJECT* *ioctl* operation needs no argument. But if it requires some, just add them after the operation code.

POS(0230A)-12.14

Non-blocking I/O

- Unlike reading and writing files, reading and writing a device usually needs to wait for a long time. We say the I/O **blocks** until completion.
- Many programs **cannot afford to suspend all other activities**. E.g., many GUI programs must continue responding to the user during I/O.
- Various methods are available for dealing with such scenarios:
 - Create a **different "I/O" kernel thread** for input operation. While the I/O thread is blocked, the main thread is still working.
 - Set the file to **non-blocking**, using *fcntl*. Now if a read or write operation needs to wait, the operation **returns an error immediately**.
 - Use a system call (*select()*, *poll()*) which **waits for all the events** that the program responds to, including the availability of I/O data.
 - Perform the blocking I/O anyway, but setup the other events to interrupt the blocking I/O. (e.g., signals interrupt most I/O.)

POS(0230A)-12.15

Non-blocking example: clear the input

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <fcntl.h>
using namespace std;
void clear_input() {
    long flag = fcntl(0, F_GETFL);
    fcntl(0, F_SETFL, flag | O_NONBLOCK); // Nonblocking I/O now
    char c;
    while (read(0, &c, 1) != -1); // Read all available characters
    fcntl(0, F_SETFL, flag); // Reset I/O to blocking
}
int main() {
    sleep(8); // supposed to be doing other things
    clear_input();
    cout << endl << "Type a line: ";
    string line; getline(cin, line);
    cout << "Got " << line << endl;
}
```

POS(0230A)-12.16

Interrupting input example: timed input

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <signal.h>
using namespace std;
void do_nothing(int) {}
int main() {
    string str;
    cout << "Give me a line in 10 seconds, or die. ";
    struct sigaction act, oact; // Stop SIGALRM from killing process
    act.sa_handler = do_nothing;
    sigaction(SIGALRM, &act, &oact);
    alarm(10); // Generate SIGALRM in 10 seconds
    getline(cin, str);
    if (!cin)
        cout << "You're dead now. BANG" << endl;
    sigaction(SIGALRM, &oact, 0);
}
```

POS(0230A)-12.17

Buffering

Buffering is introduced when we discuss IPC, as temporary storage of data in RAM when the data sent by the sender is not yet received.

A buffer thus cope with speed mismatch between data producer and data consumer. This is still true for I/O. Other reasons for I/O buffering:

- I/O devices can have different block or message sizes, and **a buffer provides an intermediate place**. E.g., when a computer needs to route a network message, the OS might need to reconstruct a fragmented message, using a buffer.
- Buffers **allow I/O to be delayed**, since we no longer rely on the user memory to service the request. Delaying I/O operations give more room for requests to be re-ordered.

The last reason means that in most OS, all data transfer is done in buffers.

POS(0230A)-12.18

Cache

- To avoid having to read a block again when a block is requested again, the **OS does not immediately deallocate** a buffer after a **block read** is copied to the user memory.
- The buffer is instead **moved to the disk cache**.
- If a disk read is already in the disk cache, the device is not accessed at all, and the process is not blocked.
- When a **disk write** is performed, the buffer used for writing would **replace (or invalidate) the cache copy**, if any.
- When **memory becomes tight**, the virtual memory system start **finding pages that are not accessed for a long time**, and free them for servicing memory requests.
- For this reason, it can be expected that **for most of the time, the RAM is nearly completely filled** for any good OS.

POS(0230A)-12.19

Read-ahead

- When accessing a **sequential, contiguous file**, it is normal that a lot of consecutive blocks of a device is requested.
- Normally, each of them will involve **setting up** the I/O controller, **waiting** for the I/O completion, and **waiting** until the scheduler decides to run the process again. This is not very efficient.
- Most of this inefficiency can be mostly eliminated if, after the request is satisfied, the low-level read operation would **immediately asks the next few blocks to be loaded into the OS buffers**—read-ahead.
- On the other hand, **read-ahead is wasteful for random access**. How do we know if access is random or sequential?
- Again, we **interpolate the future from the past**. E.g., we store the last address accessed for each device. if a request is accessing the next block after the last request, it is assumed that the access is sequential, and read-ahead is done.

POS(0230A)-12.20

Summing all: disk device reading in Linux

When reading is **requested** by the process (writing is similar):

1. If disk can be serviced from cache, done.
2. Create a disk buffer and a request to the low level driver.
3. Schedule a check before returning to user mode.
4. Put the process into the wait queue of the disk buffer, run scheduler.

Before **returning to user mode**:

1. If the disk is servicing other request, return.
2. Find the first request of the device, and start the low-level I/O.

Before **Interrupt**:

1. Copy the buffer to the user, move the buffer to cache.
2. Wake up all waiting process of the buffer. Schedule a check before returning to user mode.

POS(0230A)-12.21

Summary

- OS support allows devices to be shared among processes.
- Hardware designers uses a combination of Memory-mapped I/O, Interrupt and DMA techniques to allow the OS to operate devices efficiently.
- Some devices can be accessed explicitly by user using a device file. Other devices are accessed implicitly through OS abstractions.
- Interrupts handlers only do a few things, so that other interrupts can continue to be serviced. A kernel thread or bottom half completes the I/O.
- Device files are identified with a major and minor number.
- Non-blocking I/O are supported by the OS by various methods.
- The kernel I/O subsystem optimizes block device accesses by various techniques including buffering, caching and reordering.

POS(0230A)-12.22