Imperial College London

Operating Systems

Device Management

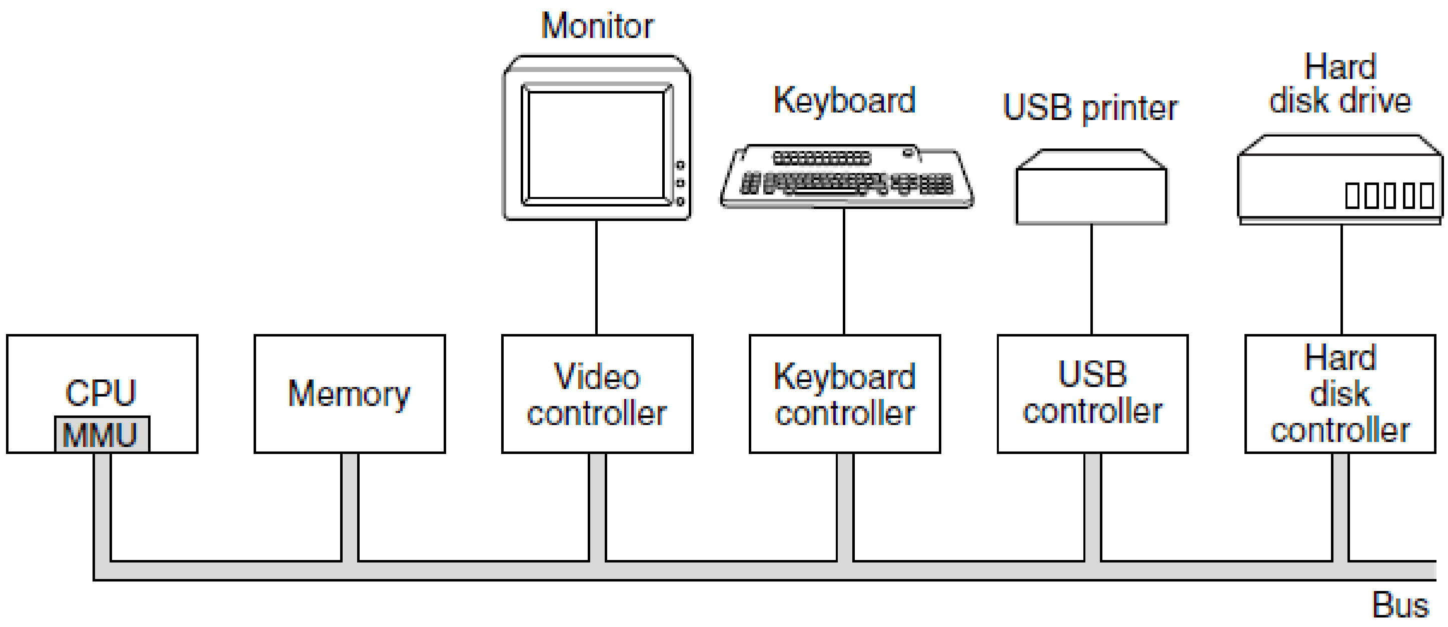
Course 502
Autumn Term 2014-2015

Based on slides by Daniel Rueckert, Peter Pietzuch, and Andrew Tanenbaum

Roman Kolcun

roman.kolcun08@imperial.ac.uk

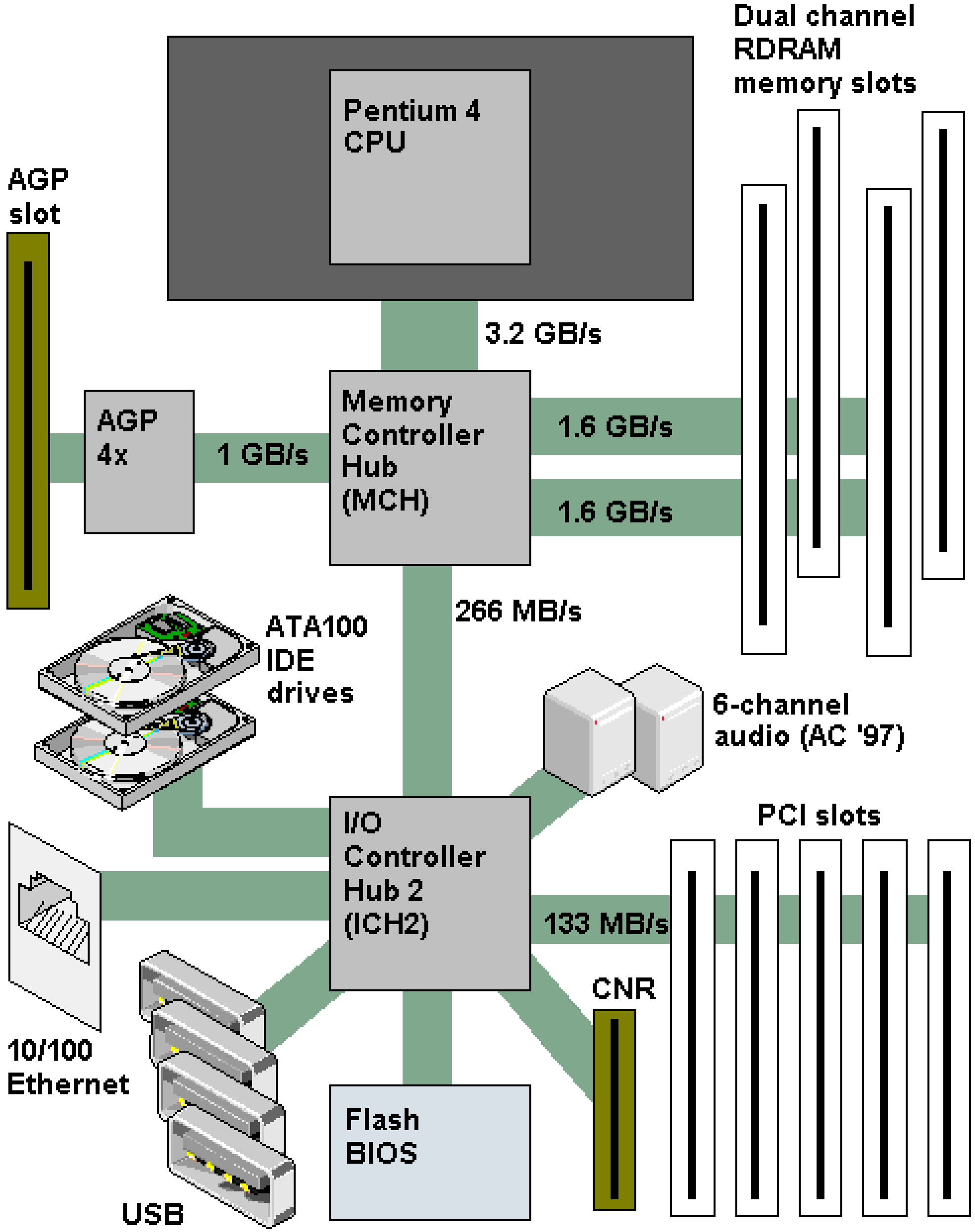
Some Components of a Simple PC



Example: Intel Pentium

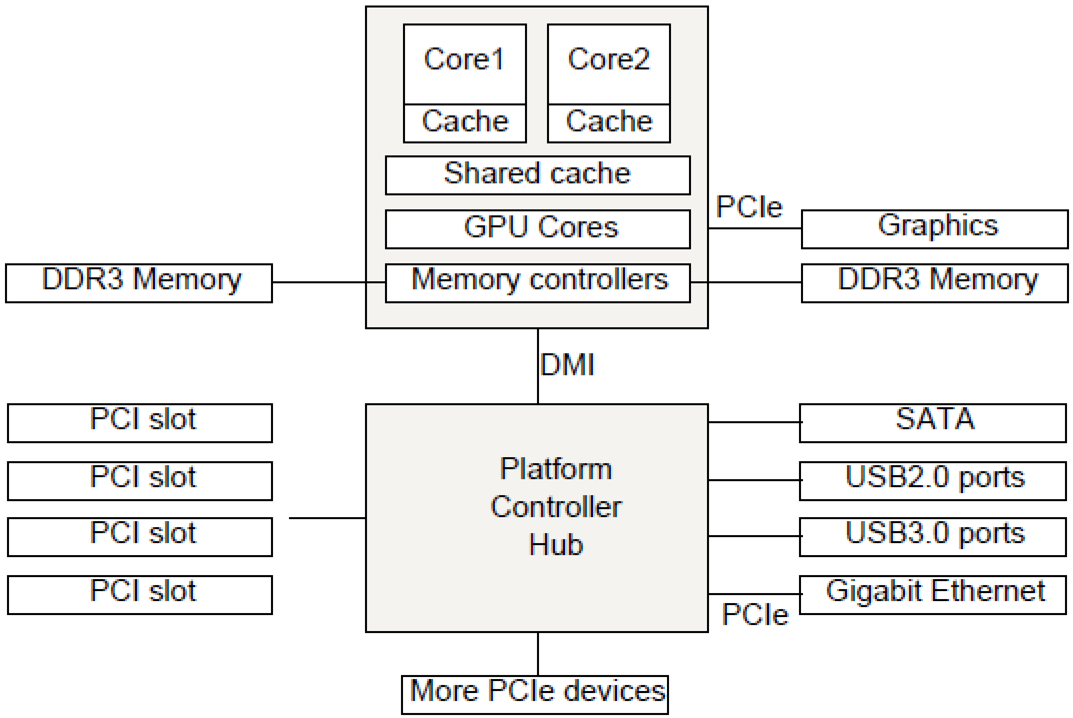
Northbridge

Southbridge



From Computer Desktop Encyclopedia
2001 The Computer Language Co. Inc.

Buses of a Modern Computer



The structure of a 86 system

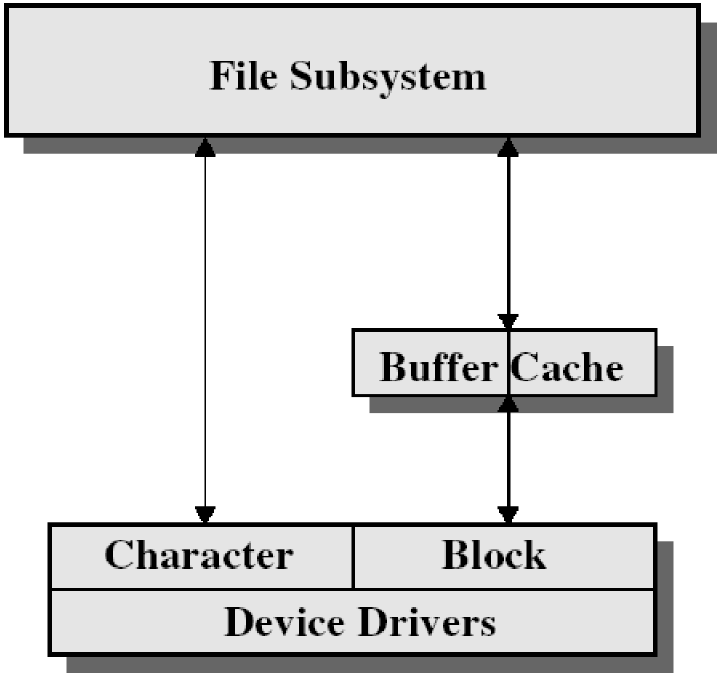
I/O Device Management

Objectives

- Fair access to shared devices
  - Allocation of dedicated devices
- Exploit parallelism of I/O devices for multiprogramming
- Provide uniform simple view of I/O
  - Hide complexity of device handling
  - Give uniform naming and error handling

Device Variations: Character vs. Block

- Block devices
  - Stores information in fixed-size blocks
  - Transfers are in units of entire blocks
- Character devices
  - Delivers or accepts stream of characters, without regard to block structure
  - Not addressable, does not have any *seek* operation

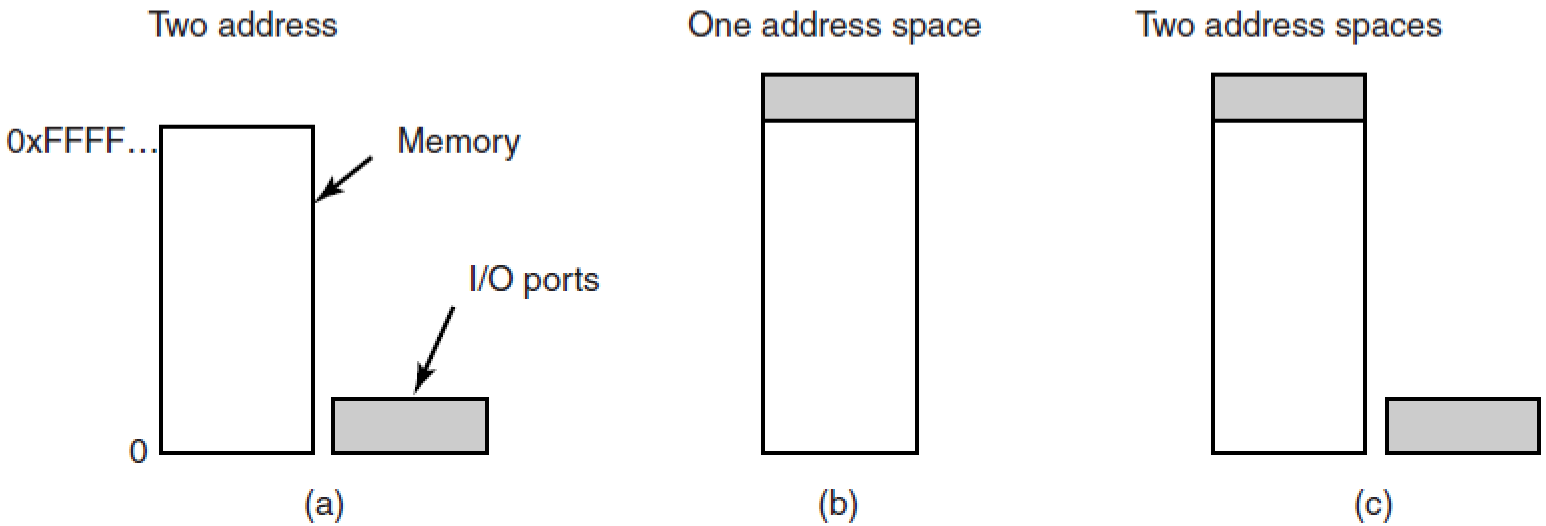


Principles of I/O Hardware

CPU and Devices Communication

- Each controller has a few registers used for communication with the CPU
- OS can write to these registers to command the device to perform some action like:
  - deliver data
  - accept data
  - switch on/off
  - perform some action
- OS can read from these registers to learn about:
  - state of the device
  - whether it is ready to accept commands
  - …

How Controllers Communicate w/ CPU



(a) Separate I/O and memory space.
(b) Memory-mapped I/O. (c) Hybrid.

Separate I/O and Memory Space

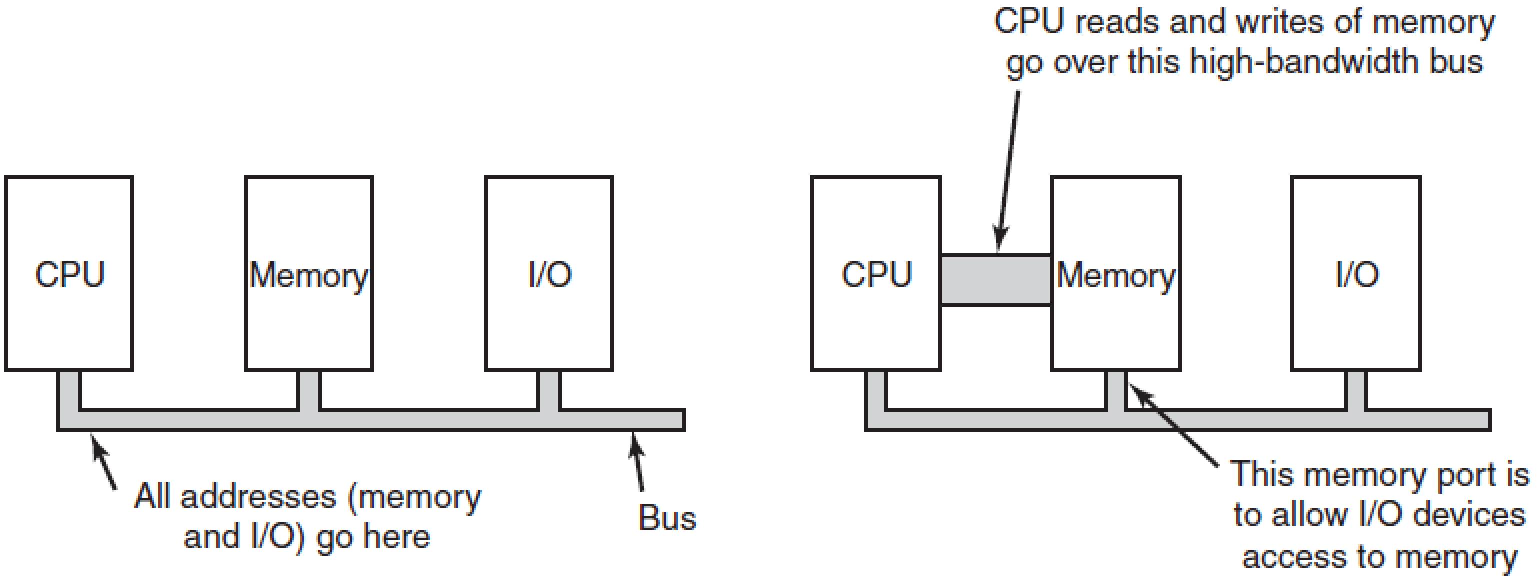
- Each control register is assigned an I/O port number
- The set of all ports form I/O port space which is protected from a user
- Can be accessed only through OS using special instructions like:
  - `IN REG,PORT`
  - `OUT PORT,REG`

Memory Mapped I/O I

- All the control registers are mapped into memory
- Each register has a unique memory address
- Usually these registers are at the top of the address space
- Advantages:
  - No special instruction is needed
  - No special mechanism is needed to keep user processes from performing I/O
  - If each device has its control registers on a different page OS can give access to a user to several devices but not others
  - Instructions for memory manipulation could be used for I/O

Memory Mapped I/O II

- Disadvantages:
  - Caching could be a problem
  - Memory and I/O has to listen to all requests. Each response only to requests within assigned range
  - The problem arises if there is a special bus for communication between CPU and memory

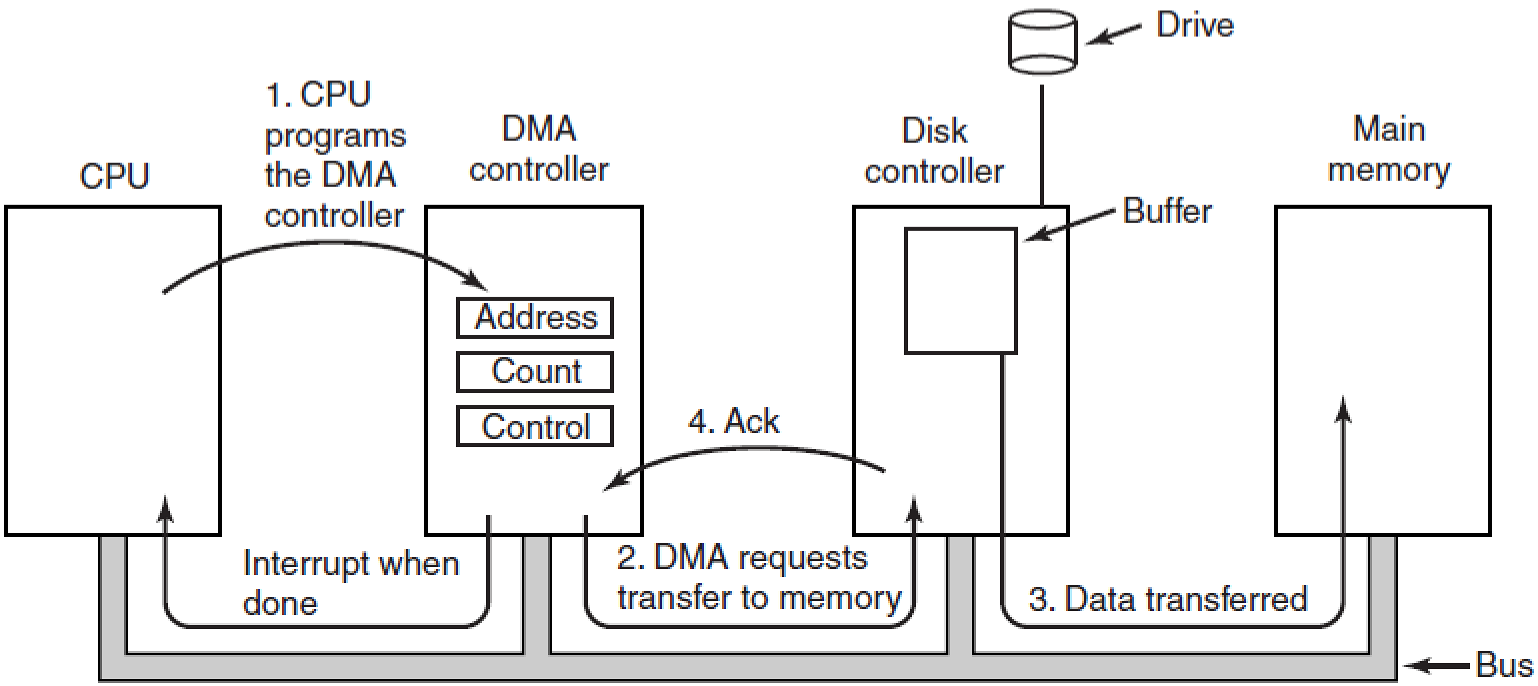


(a) (b)

Direct Memory Access (DMA) I

- Specialised controller
- Could be integrated into disk controller or other controllers
- More usually a single DMA controller is integrated on a motherboard and controls several devices
- More sophisticated DMA controllers can handle multiple transfers in parallel
- Has access to the bus independently from CPU
- Has registers which could be read/written by CPU

Direct Memory Access (DMA) II



Operation of a DMA transfer.

Direct Memory Access (DMA) III

- DMA handling multiple controllers can work in two modes:
  - Word-at-a-time – DMA requests for the transfer of one word and gets the bus. If CPU wants it at the same time it has to wait – this operation is called **cycle stealing**
  - Block mode – a series of transfers is issued by DMA but it blocks the bus for a longer period, hence blocking CPU and other controllers – this operation is called **burst mode**
- Depending on where controller sends data we distinguish between:
  - Fly-by mode – data are written directly to main memory
  - Alternatively data are first stored in DMA controller and then written to memory (support for device-to-device and memory-to-memory transfers)

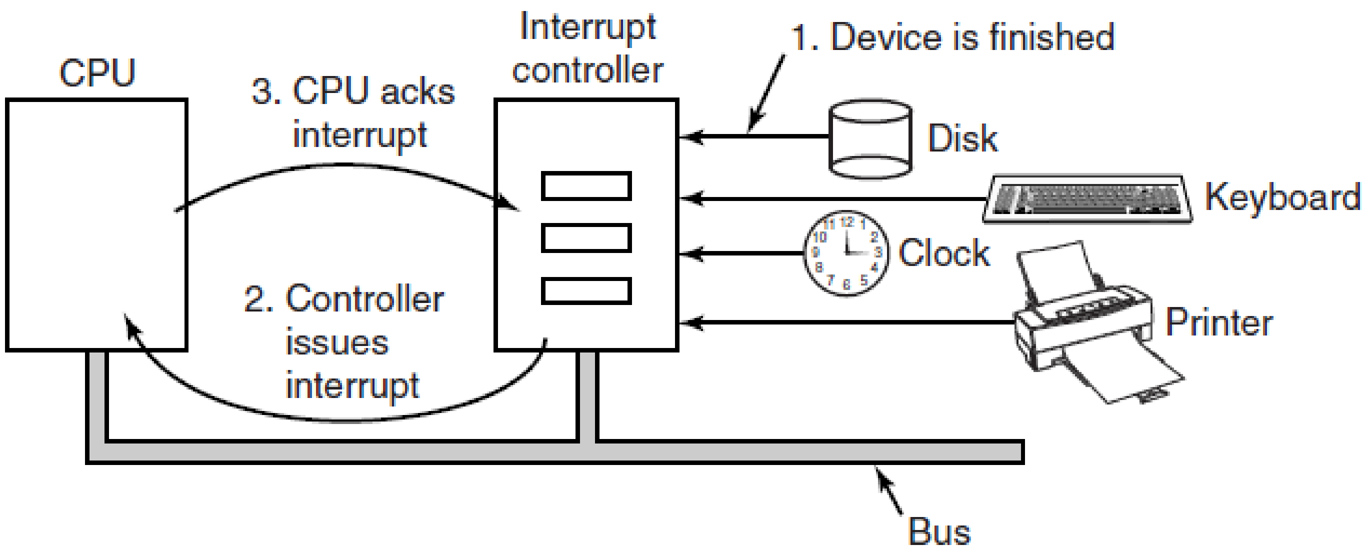
Direct Memory Access (DMA) IV

- Why not to use DMA:
  - CPU is much faster than DMA
  - Does not require additional controller, hence, money could be saved

Interrupts I

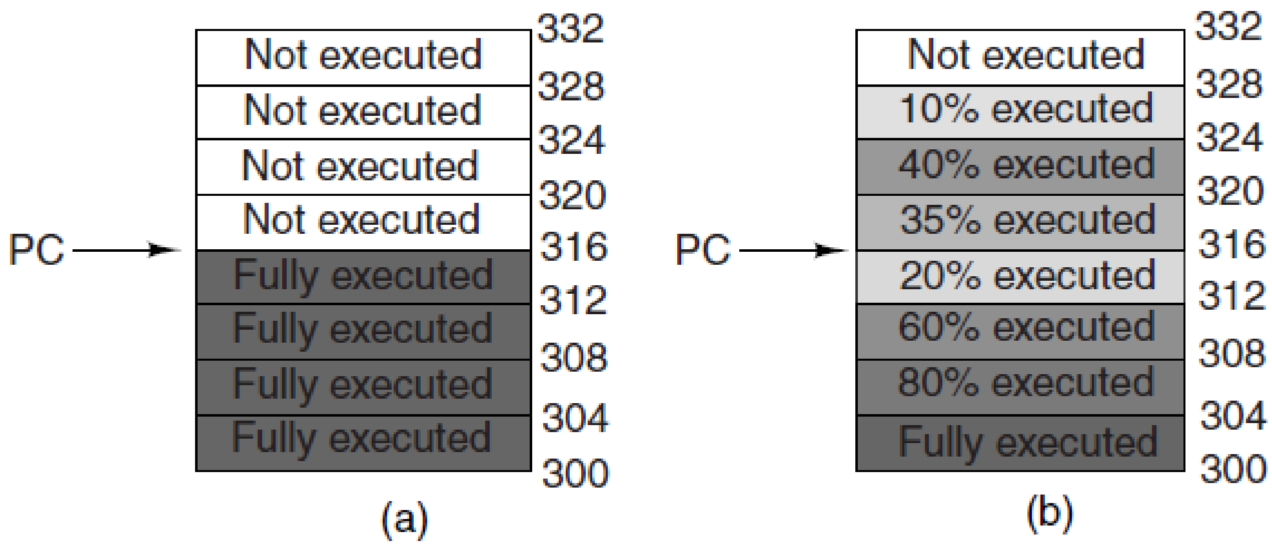
- When I/O device has finished the work given to it, it causes an interrupt by asserting a signal on a bus line assigned to it
- Signal is detected by interrupt controller
- Simultaneous interrupts are processed according to priority of interrupts
- Interrupt controller puts a number on the address lines specifying which device requires attention
- The interrupt signal causes CPU to stop the current work and to handle interrupt
- The number on the address lines is used as an index into a table called the **interrupt vector** which points to the routine handling the interrupt

Interrupts II



How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Precise vs. Imprecise Interrupt



(a) A precise interrupt. (b) An imprecise interrupt.

Principles of I/O Software

Goals of the I/O Software

Device independence from

- Device type (e.g. terminal, disk or DVD drive)
- Device instance (e.g. which disk)

Uniform naming – the name of a file should be a string or integer and not depend on the device in an way

Device variations

- Unit of data transfer: character or block
- Supported operations: e.g. read, write, seek
- Synchronous or asynchronous operation
- Speed differences
- Sharable (e.g. disks) or single user (e.g. printer, DVD-RW)
- Error handling
- Buffering

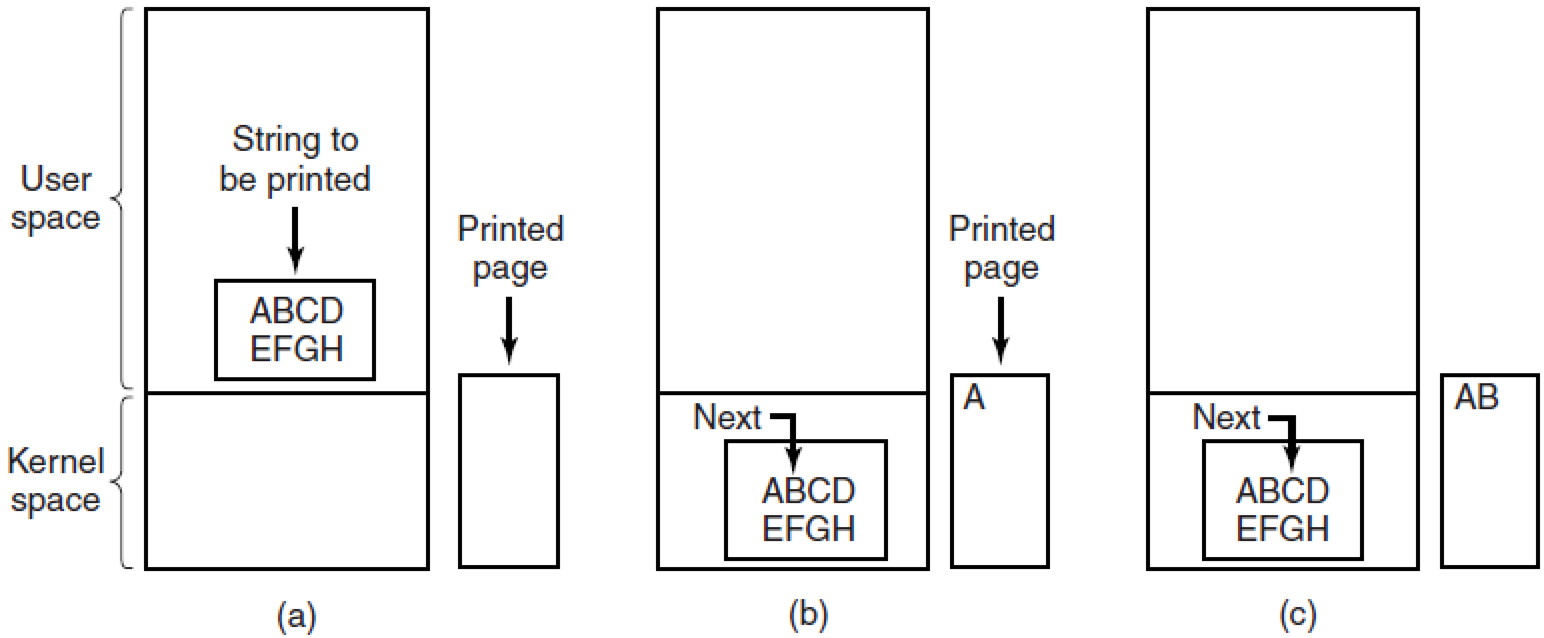
Ways to do I/O

Programmed I/O

Interrupt Driven I/O

Direct Memory Access I/O

Programmed I/O I



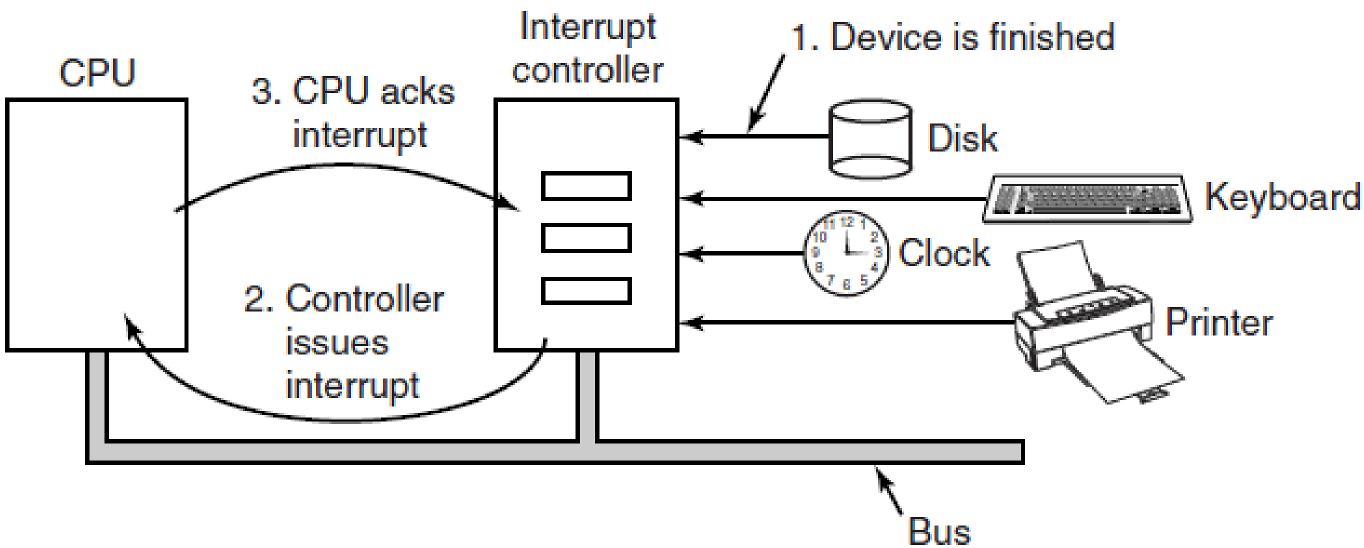
Steps in printing a string.

Programmed I/O II

```
copy_from_user(buffer, p, count);          /* p is the kernel buffer */
for (i = 0; i < count; i++) {              /* loop on every character */
    while (*printer_status_reg != READY) ; /* loop until ready */
    *printer_data_register = p[i];         /* output one character */
}
return_to_user( );
```

Writing a string to the printer using programmed I/O.

Interrupt-Driven I/O I



How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

Interrupt-Driven I/O II

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

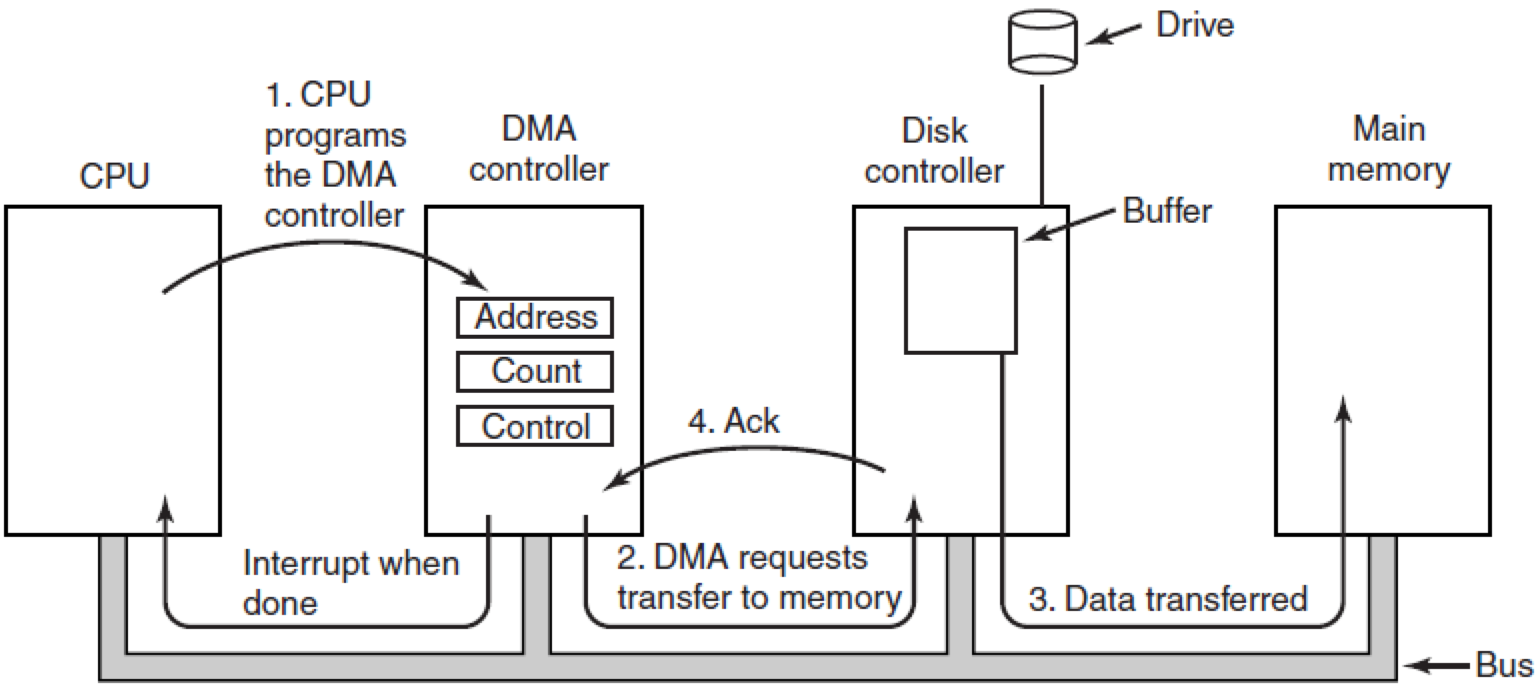
(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Writing a string to the printer using interrupt-driven I/O. (a) Code executed at the time the print system call is made. (b) Interrupt service procedure for the printer.

I/O Using DMA I



Operation of a DMA transfer.

I/O Using DMA II

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

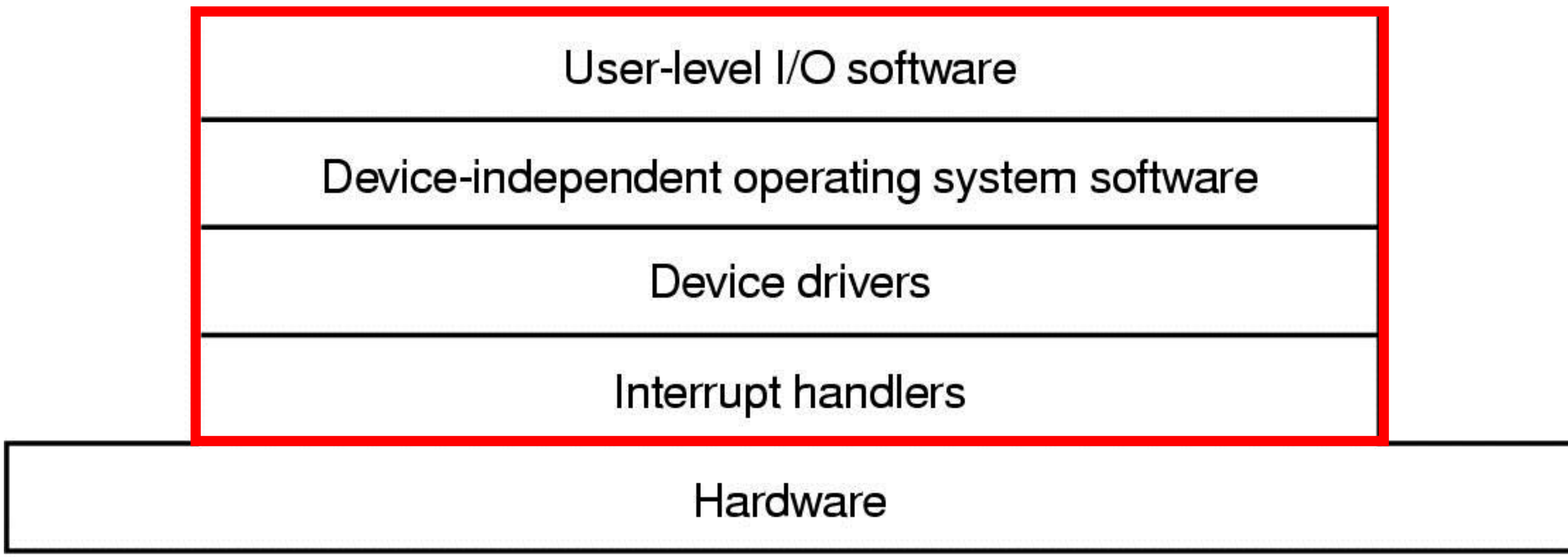
```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt service procedure.

I/O Software Layers

I/O Layers: Overview



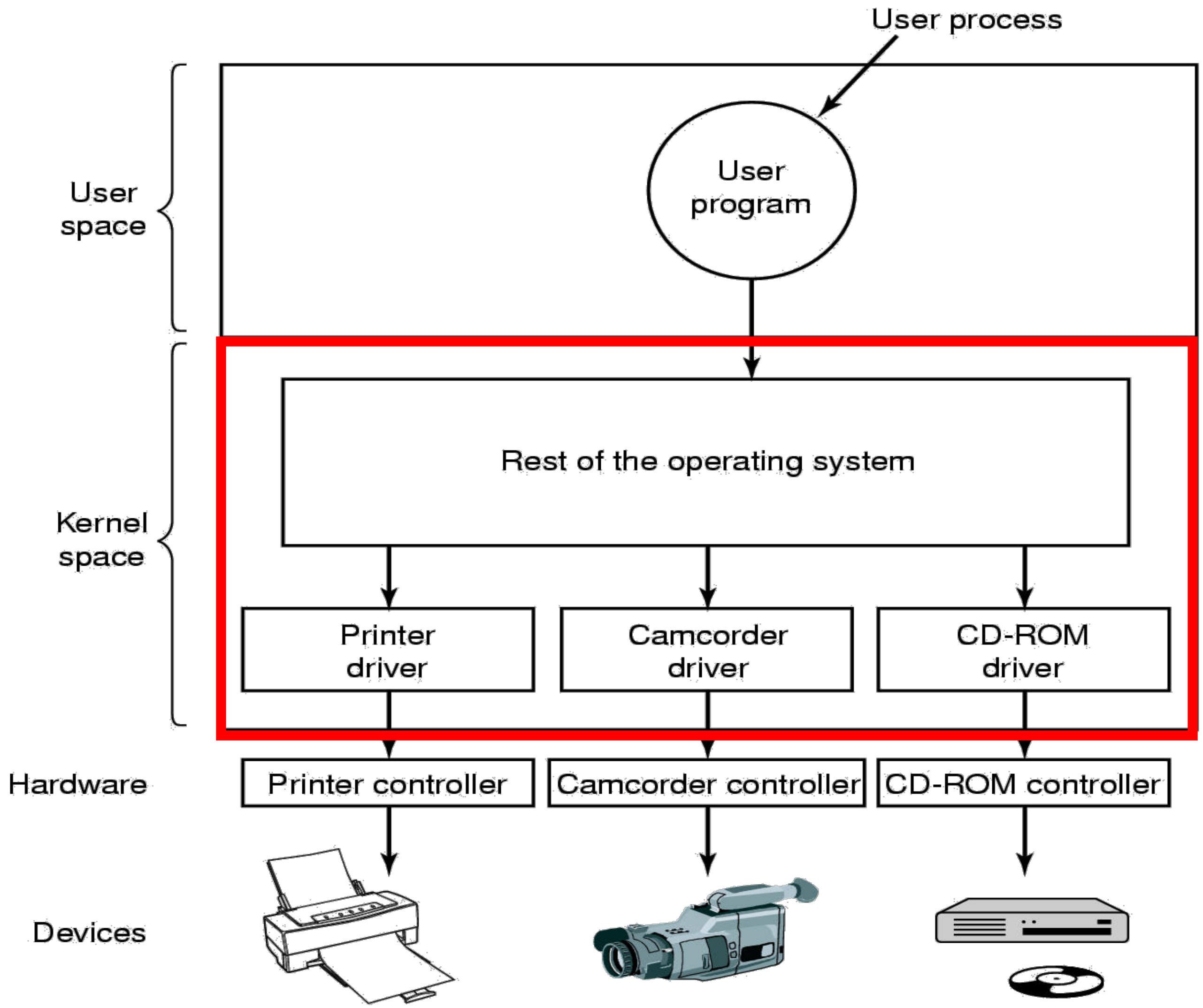
Interrupt Handlers

- Driver starts an I/O operation block until the I/O has completed
- Meanwhile the driver blocks itself doing **down** on a semaphore, a **wait** on a condition variable, a **receive** on a message, or something similar
- When interrupts happens, the interrupt procedure does whatever it has to in order to handle the interrupt
- Then it will unblock the driver that started it

Device Driver

- Is a device-specific code for controlling an I/O device
- A driver for a mouse differs from a driver for an HDD
- Handles one device type, or at most, one class of closely related devices
- Is part of kernel, therefore a buggy driver can cause crash of a system
- Is positioned below the rest of the OS with a direct access to controllers
- Most OS define a standard interface (between OS and the driver) for **block devices** and **character devices**
- Must be flexible and be able to handle errors several interrupts, etc.
- Is allowed to call only a handful of system calls, e.g. to allocate memory for a buffer

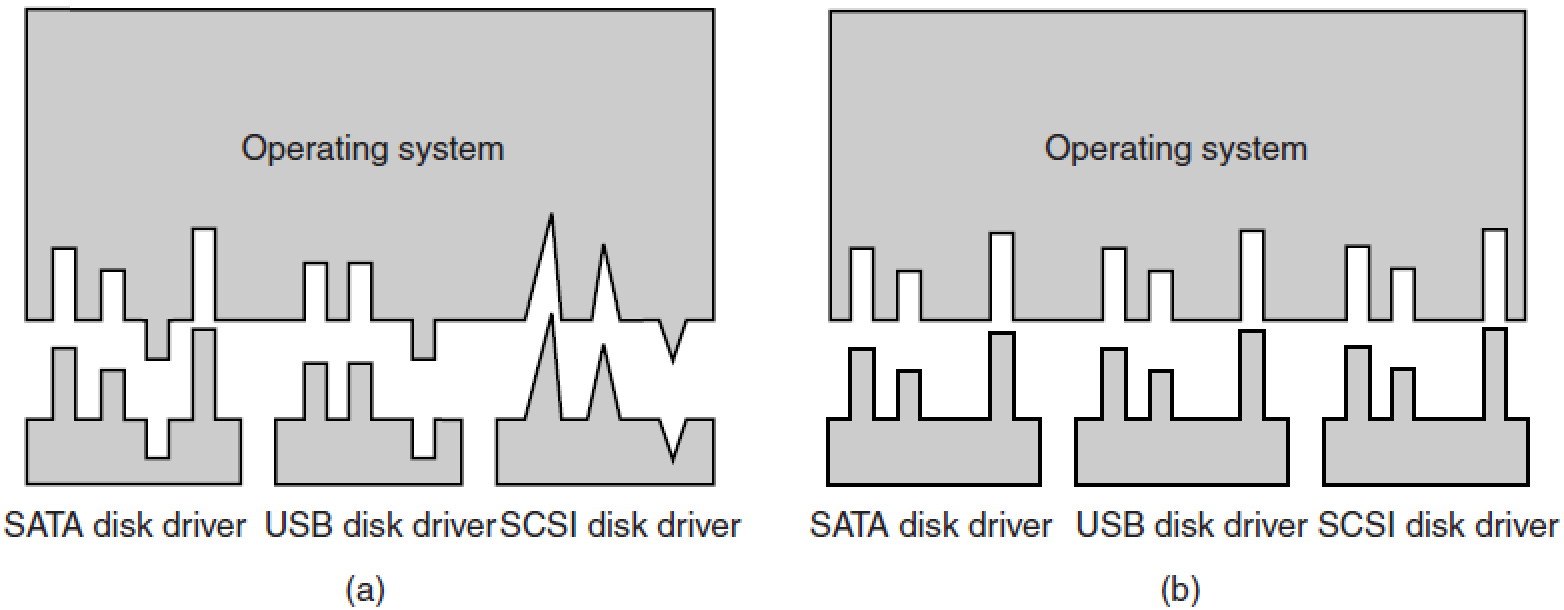
I/O Layering


User process
User space
User program
Kernel space
Rest of the operating system
Printer driver
Camcorder driver
CD-ROM driver
Hardware
Printer controller
Camcorder controller
CD-ROM controller
Devices

Device-Independent I/O Software

- Some parts are device-specific but others are device independent
- There is no strict boundary between device-specific and device independent software and varies between Oss
- Most common device independent functions are:
  - Uniform interfacing for device drivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - Providing a device-independent block size

Uniform Interfacing for Device Drivers I

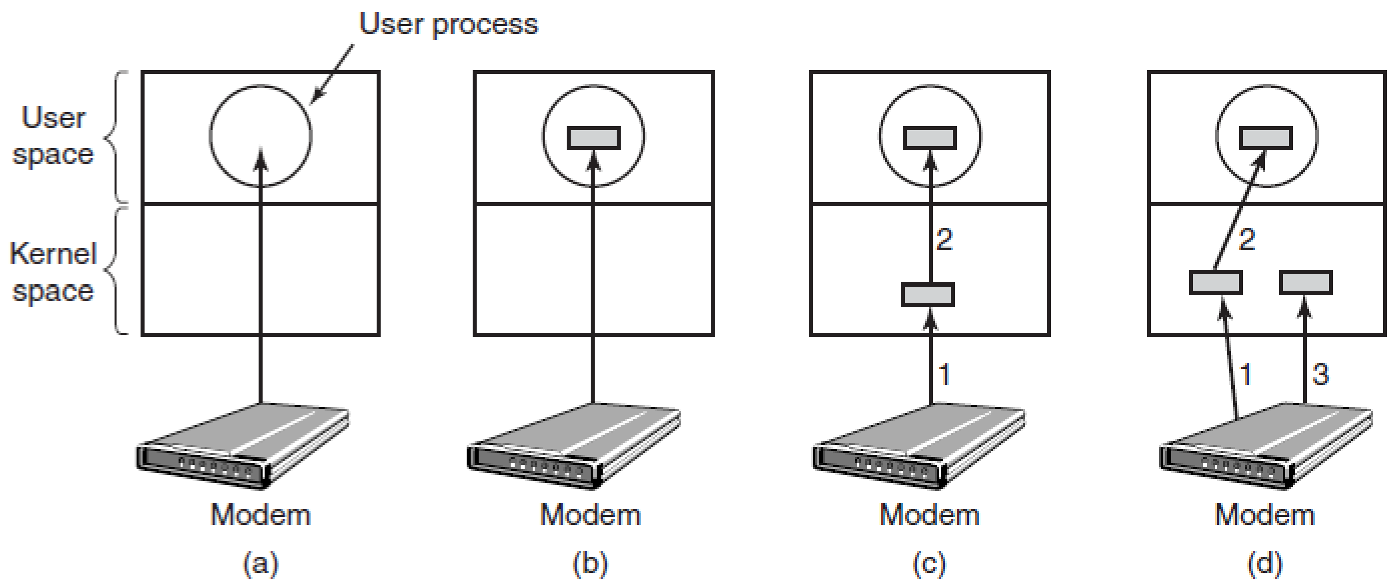


(a) Without a standard driver interface.
(b) With a standard driver interface.

Uniform Interfacing for Device Drivers II

- The interface between the driver and OS is defined
- The OS can install new driver easily and the writer of the driver knows what it can expect from the OS
- In practice, not all devices are absolutely identical, but there are only a small number of device types
- For each class of devices (e.g. disks or printers) the OS defines a set of functions that the driver must supply
- Often the driver contains a table with pointers into itself for these functions
- OS uses records the address of the table when the driver is loaded and makes indirect calls via this table
- Another aspect of having a uniform interface is how I/O devices are named: each device has a **major device number** and **minor device number.**
- Closely related to naming is protection – devices appear as files in the file system, so usual protection rules could be used

Buffering I



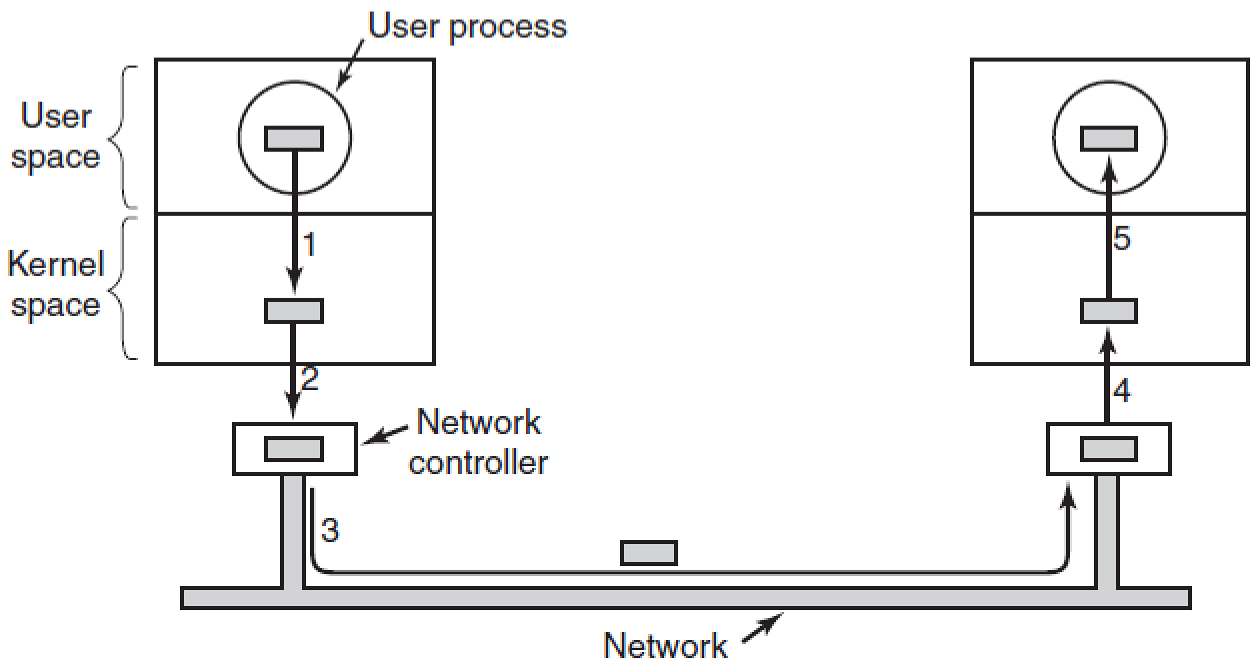
(a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Buffering II

Ways to handle data streams from I/O devices:

- Each interrupt may wake-up user's process
- OS writes into user space's buffer and wakes up user's process once the buffer is full. What happens if the buffer is paged out when a character arrives?
- OS writes into a buffer in kernel's memory space and then copies data to user's space. What happens if a character arrives at the time when the buffer is being copied to user's space?
  - Double buffering
  - Circular buffering
- Buffering is also important for output – e.g. when sending data over a slow telephone line

Buffering III



Networking may involve many copies of a packet.

Error Reporting

- Errors are far more common in the context of I/O than in other contexts
- Many errors are device-specific and must be handled by appropriate driver, but the framework for error handling is device independent
- Classes of errors:
  - Programming errors: write to a keyboard, read from printer, read from invalid buffer address, read from disk 3 when there are only two disks. Solution: just report back an error code to the caller
  - Actual I/O errors: write to a disk block that has been damaged or read from a camera that is turned off. Solution: it is up to the driver what to do whether to try to solve the problem or report back the error code

Allocating and Releasing Dedicated Devices

- Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment
- It is up to OS to examine requests for a device and to accept or reject them
- OS may require a process to **open** a device and to **close** it when it is finished. If the device is not available the call will fail
- Alternatively OS may have a special mechanism for requesting devices and put requests to a queue. A request for an unavailable device will be blocked, instead of failing

Device-Independent Block Size

- Different disks may have different sector sizes
- It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers
- Some devices deliver data one byte at a time (e.g. modems), while others deliver theirs in larger units (e.g. network interfaces)

User-Space I/O Software I

- Most of the I/O software is within the OS
- Small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel
- An example could be count = write(fd, buffer, nbytes)
- This procedure put their parameters in the appropriate place for the system call
- Other procedures do actual work: e.g. formatting of a string
- Not all user-level I/O software consists of library procedures – another important category is the spooling system

User-Space I/O Software II

Blocking user access to allocated, nonsharable devices?

– Causes delays and bottlenecks

Spool to intermediate medium (disk file)

Spooling is a way of dealing with dedicated I/O devices in a multiprogramming system.

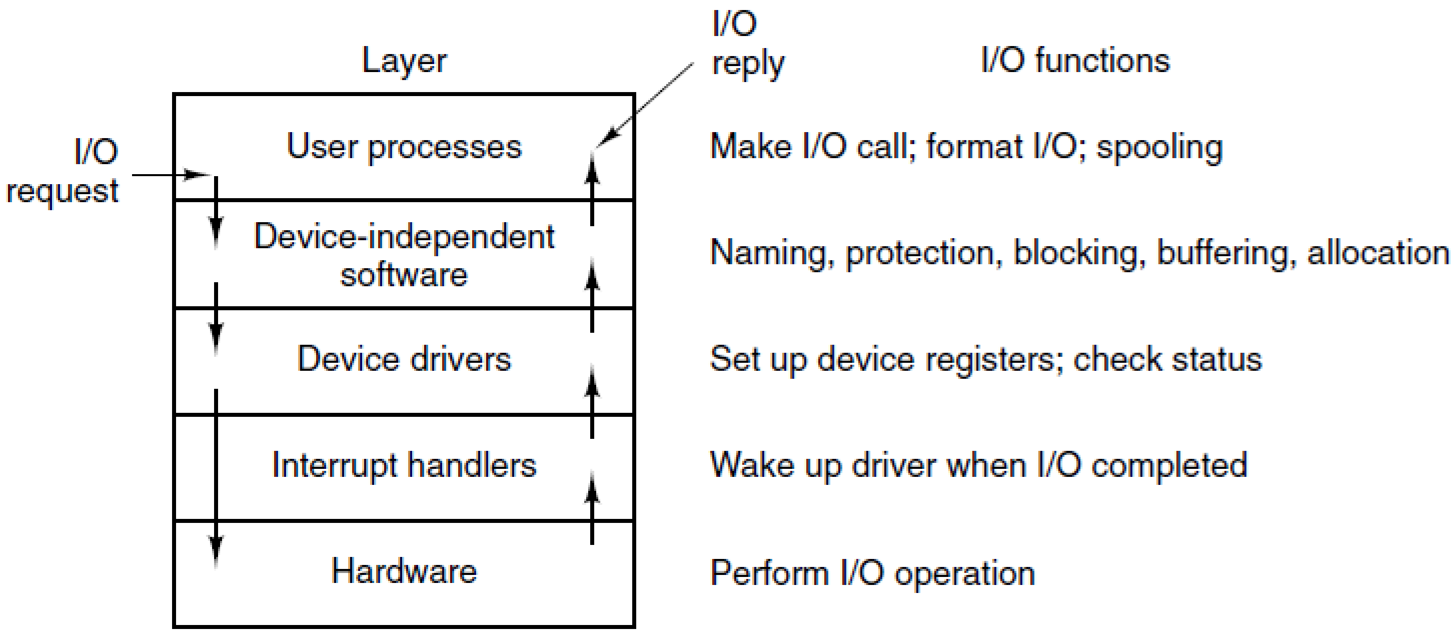
Spooled devices (e.g. printers)

1. Printer output saved to disk file
2. File printed later by **spooler daemon**
   - Printer only allocated to spooler daemon
   - No normal process allowed direct access

– Provides sharing of nonsharable devices

– Reduces I/O time ➔ gives greater throughput

User-Space I/O Software III



Layers of the I/O system and
the main functions of each layer.

Linux: Loadable Kernel Module (LKM)

Loadable kernel modules provide device drivers

- Contain object code, loaded on-demand
  - Dynamically linked to running kernel
  - Provided by hardware vendors or independent developers
- Require binary compatibility
  - Modules written for different kernel versions may not work

Kmod

- Kernel subsystem managing modules without user intervention
- Determines module dependencies
- Load modules on demand

Linux: Basic LKM module

Every LKM consists of two basic functions (minimum):

```
int init_module(void)  /* used for all initialisation code */
{
...
}
void cleanup_module(void)  /* used for clean shutdown */
{
...
}
```

Load module by issuing following command:

```
insmod module.o
```

– Normally restricted to root

Linux I/O Management

Linux I/O Management

Kernel provides common interface for I/O system calls

Devices grouped into **device classes**

- Members of each device class perform similar functions
- Allows kernel to address performance needs of certain devices (or classes of devices) individually

Major and **minor** identification numbers

- Used by device drivers to identify their devices
- Devices with same major num controlled by same driver
- Minor nums enable system to distinguish between devices of same class

Linux: Device Drivers

Device special files

- Most devices represented by device special files
- Entries in **`/dev`** directory that provide access to devices
- List of devices in system can be obtained by reading contents of **`/proc/devices`**:

Character devices:

```
  1 mem
  2 pty
  4 ttyS
  5 cua
 10 misc
 13 input
109 lvm
136 pts
162 raw
180 usb
```

Block devices:

```
 1 ramdisk
 2 fd
 3 ide0
 7 loop
 8 sd
 9 md
58 lvm
65 sd
66 sd
```

Linux: /dev

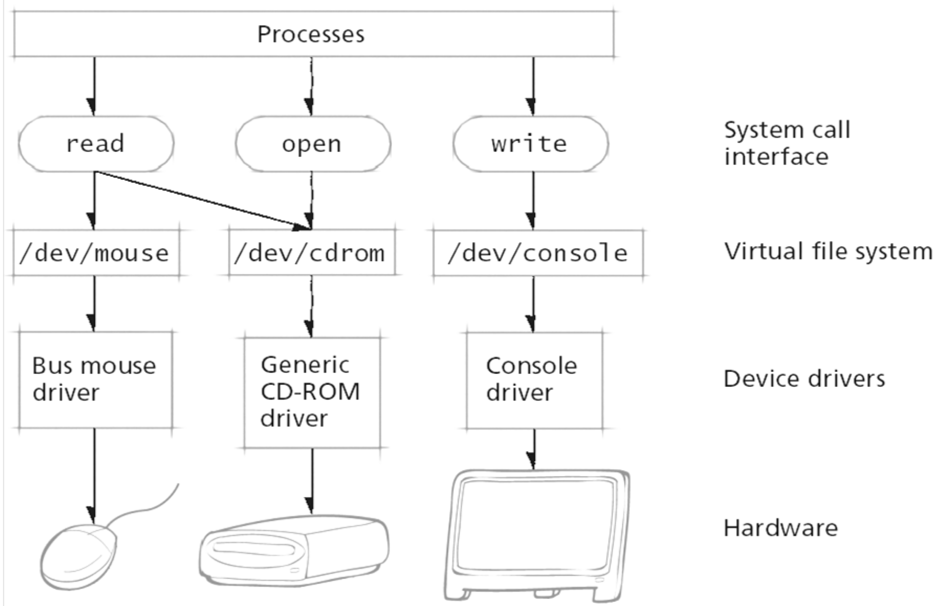
c/b   major minor   file name

```
crw-------    1 root     root      5,   1 Dec 27 16:09 console
brw-rw-rw-    1 root     disk      2,   0 May 21  2001 fd0
brw-rw-rw-    1 root     disk      2,   4 May 21  2001 fd0d360
brw-rw-rw-    1 root     disk      2,   8 May 21  2001 fd0h1200
brw-rw-rw-    1 root     disk      2,  40 May 21  2001 fd0h1440
crw-rw----    1 root     lp        6,   0 May 21  2001 lp0
crw-rw----    1 root     lp        6,   1 May 21  2001 lp1
crw-rw----    1 root     lp        6,   2 May 21  2001 lp2
crw-rw----    1 root     lp      180,   0 May 21  2001 usblp0
crw-rw----    1 root     lp      180,   1 May 21  2001 usblp1
crw-rw----    1 root     lp      180,   2 May 21  2001 usblp2
lrwxrwxrwx    1 root     root          10 Dec  6 06:53 mouse -> /dev/psaux
crw-rw-r--    1 root     root     10,   1 May 21  2001 psaux
lrwxrwxrwx    1 root     root           3 Nov 30  2001 cdrom -> hdc
brw-rw-rw-    1 root     disk      3,   0 May 21  2001 hda
brw-rw-rw-    1 root     disk      3,  16 May 21  2001 hdb
brw-rw-rw-    1 root     disk      3,  32 May 21  2001 hdc
```

Linux: Device Access

Device files accessed via **virtual file system (VFS)**

- System calls pass to VFS, which in turn issues calls to device drivers
- Most drivers implement common file operations
  - e.g. read, write, seek



I/O subsystem overview

Linux provides **ioctl** system call

- Supports special tasks:
  - Ejecting CD-ROM tray
    `ioctl(cdrom, CDROMEJECT, 0)`
  - Retrieving status information from printer

Linux: Character Device I/O I

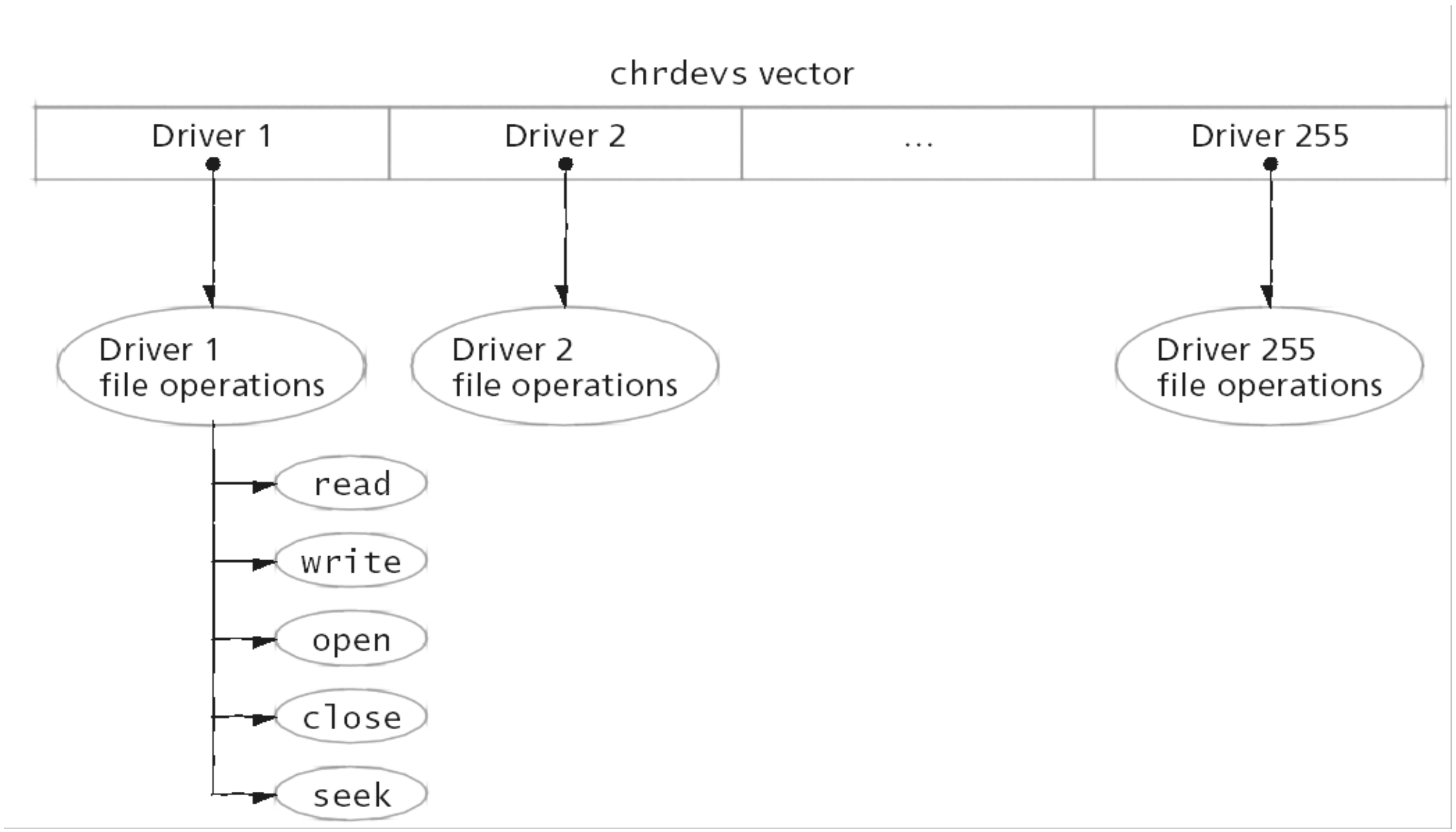
Character device

- Transmits data as <u>stream of bytes</u>
- Represented by **`device_struct`** structure contains:
  - Driver name
  - Pointer to driver's **`file_operations`** structure
- All registered drivers referenced by **`chrdevs`** vector

`file_operations` structure

- Maintains operations supported by device driver
- Stores functions called by VFS when system call accesses device special file

Linux: Character Device I/O II



Linux: Block Device I/O

Block I/O subsystem

- Kernel's block I/O subsystem contains number of layers
- Modularise block I/O operations by placing common code in each layer

Two primary strategies used by kernel to minimise amount of time spent accessing block devices:

- Caching data
- Clustering I/O operations

Linux: Block Device Caching

When data from block device requested, kernel first searches **cache**

- If found, data copied to process's address space
- Otherwise, typically added to request queue

Direct I/O

- Driver bypasses kernel cache when accessing device
- Important for databases and other applications
  - Kernel caching inappropriate and may reduce performance/consistency

Linux I/O API

Linux: I/O API I

I/O classes

Character (unstructured):	Files and devices
Block (structured):	Devices
Pipes (message):	Interprocess communication
Socket (message):	Network interface

I/O calls

```
fd = create(filename, permission)
```

Opens file for reading/writing; **`fd`** is index to file descriptor, permission is used for access control

```
fd = open(filename, mode)
```

Mode is 0, 1, 2 for read, write, read/write

Linux: I/O API II

`close(fd)`

Close file or device

`numbytesread = read(fd, buffer, numbytes)`

read **`numbytes`** from file or device referenced by **`fd`** into memory buffer; returns number of bytes actually read in **`numbytesread`**

`numbyteswritten = write(fd, buffer, numbytes)`

write **`numbytes`** to file referenced by **`fd`** from memory buffer; returns number of bytes actually written in **`numbyteswritten`**

Linux: I/O User Interface API III

`pipe(&fd[0])`

Creates pipe; **`fd`** is an array of two integers: **`fd[0]`** is for reading, **`fd[1]`** for writing

`newfd = dup(oldfd), dup2(oldfd, newfd)`

Duplicate file descriptor

`ioctl(fd, operation, &termios)`

Used to control devices; e.g. **`&termios`** is array of control chars

`fd = mknod(filename, permission, dev)`

Creates new special file e.g. character or block device

Linux: File Descriptors

Each process has its own **file descriptor table**

- Each process has 3 file descriptors when created:

file descriptor	input/output
0	stdin
1	stdout
2	stderr

- By default, all three file descriptors refer to terminal from which program was started

Linux: I/O Example I

```
#include <stdlib.h>
#define BUFSIZE 512

int main( int argc, char ** argv){

  int fd, n, stdin, stdout, stderr;
  char buffer[BUFSIZE];

  /* Standard input always corresponds to fd = 0 */
  stdin = 0;

  /* Standard output always corresponds to fd = 1 */
  stdout = 1;

  /* Standard error always corresponds to fd = 2 */
  stderr = 2;

  /* Open file */
  fd = open(argv[1], O_RDONLY);
```

Linux: I/O Example II

```
  if (fd < 0) {

     write(stderr, "Can't open file", 15);

  } else {

     do {
        n = read(fd, buffer, BUFSIZE);
        if (n < 0) {
           write(stderr, "Error while reading", 19);
        } else {
           write(stdout, buffer, n);
        }
     } while (n > 0);

  }

  /* Close file */
  close(fd);
}
```

Blocking vs. Non-blocking I/O

Blocking I/O

- I/O call returns when operation completed
- Process suspended ➔ I/O appears "instantaneous"
- Easy to understand but leads to multi-threaded code

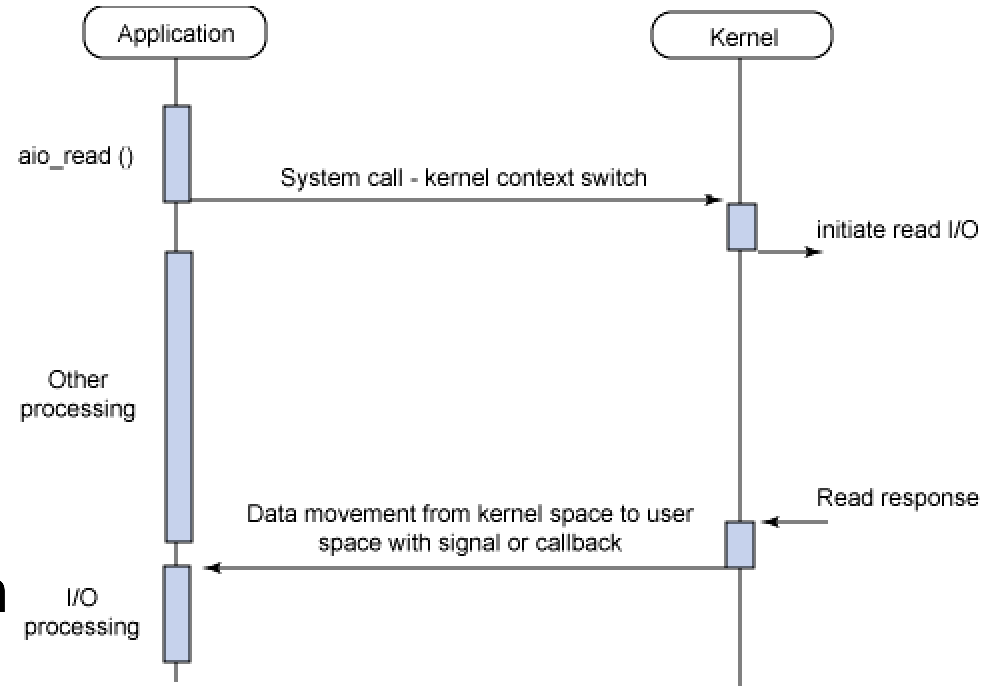
Non-blocking I/O

- I/O call returns as much as available
  (e.g. `read` with 0 bytes)
- Turn on for file descriptor using `fcntl` system call
- Provides application-level polling for I/O (how?)

Asynchronous I/O

Asynchronous I/O

- Process executes <u>in parallel</u> with I/O operation
  - No blocking in interface procedure
- I/O subsystems notifies process upon completion
  - Callback function, process signal, ...



Source: IBM Developerworks

- Supports check/wait if I/O operation completed
- Very flexible and efficient
- Harder to use and potentially less secure (why?)

Linux: AIO Example I

AIO: Support for asynchronous I/O in Linux 2.6

```
#include <aio.h>
...
  int fd, ret;
  struct aiocb my_aiocb;

  fd = open("myfile", O_RDONLY );

  /* Allocate buffer for aio request */
  my_aiocb.aio_buf = malloc(BUFSIZE + 1);

 /* Initialise aio control structure */
  my_aiocb.aio_fildes = fd;
  my_aiocb.aio_nbytes = BUFSIZE;
  my_aiocb.aio_offset = 0;
```

Linux: AIO Example II

/* Initiate read request */

```
ret = aio_read(&my_aiocb);
```

/* Wait for read to finish (more usefully do something else)
Also possible to register signal notification or thread callback */

```
while (aio_error(&my_aiocb) == EINPROGRESS);
```

/* Check result from read */

```
if ((ret = aio_return(&my_iocb)) > 0) {
    /* Successfully read ret bytes */
} else {
    /* Read failed, check errno*/
}
```