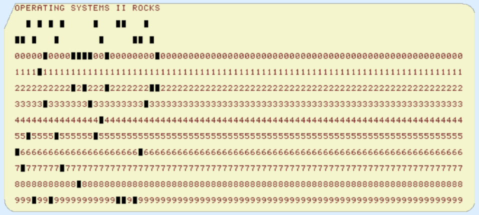
Imperial College London

Operating Systems

Device Management



Course 211
Spring Term 2018-2019

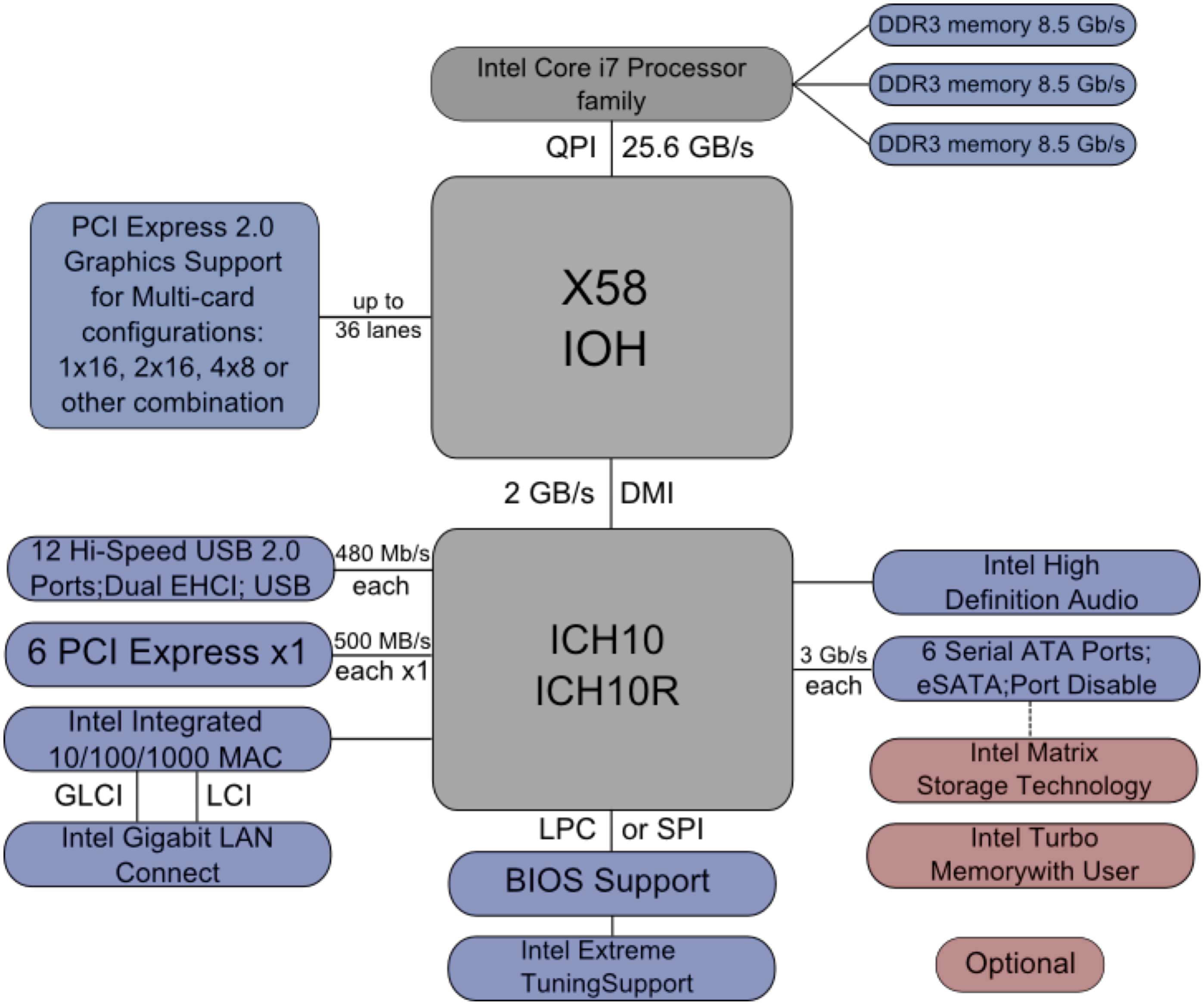
http://www.imperial.ac.uk/computing/current-students/courses/211/calendar/

Based on slides by Daniel Rueckert

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

Example: Intel Architecture



I/O Device Management

Objectives

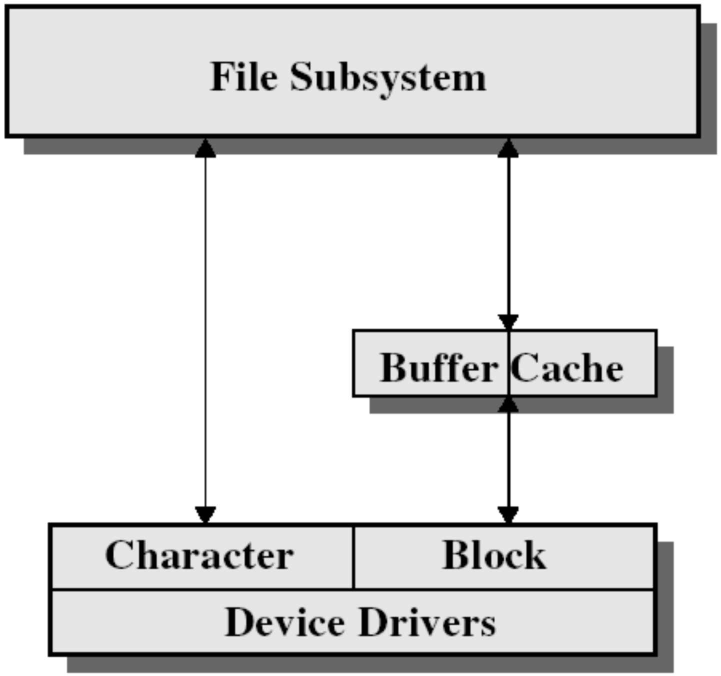
- Fair access to shared devices
  - Allocation of dedicated devices
- Exploit parallelism of I/O devices for multiprogramming
- Provide uniform simple view of I/O
  - Hide complexity of device handling
  - Give uniform naming and error handling

Device Independence

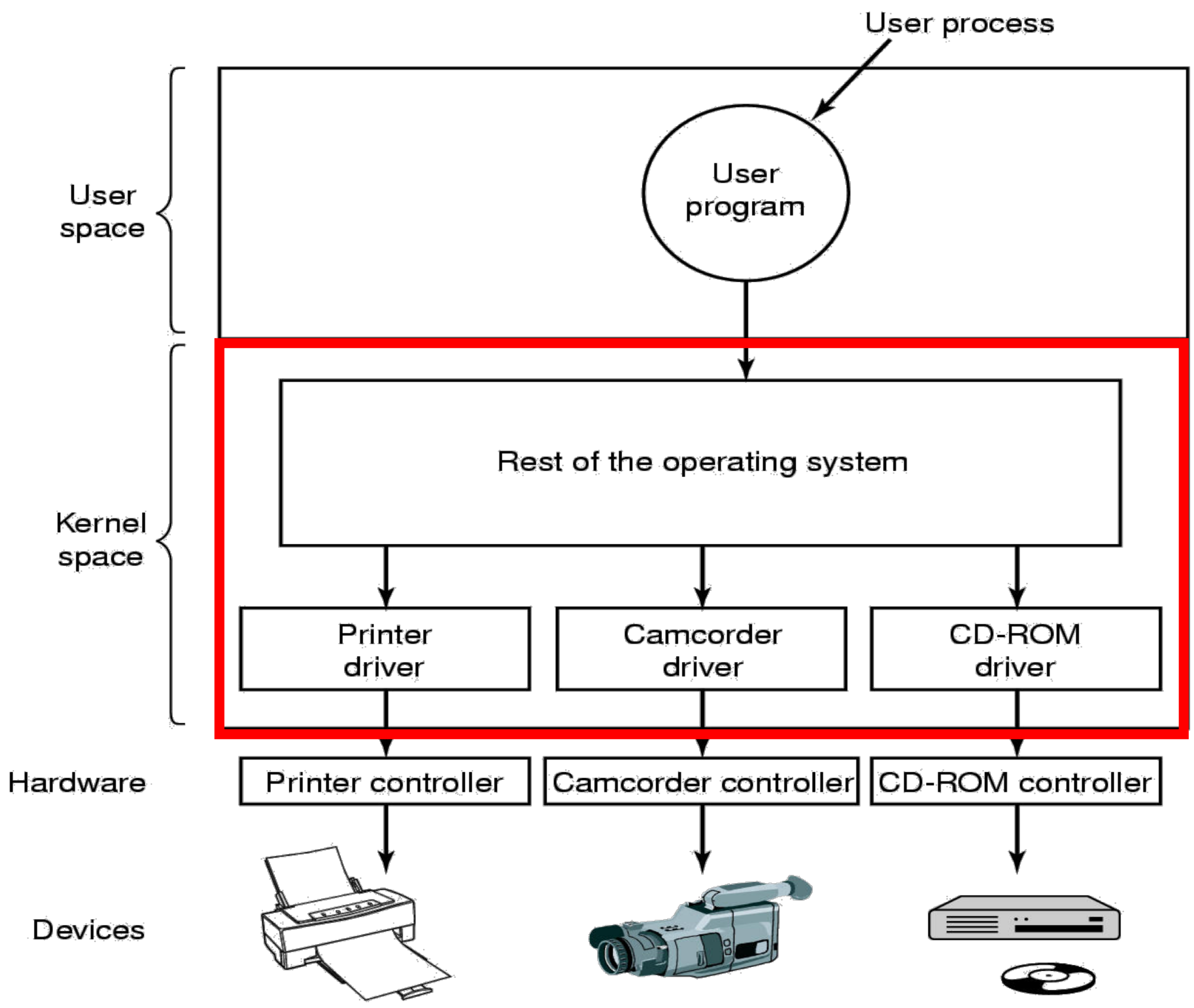
Device independence from
- Device type (e.g. terminal, disk or DVD drive)
- Device instance (e.g. which disk)

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)
- Types of error conditions

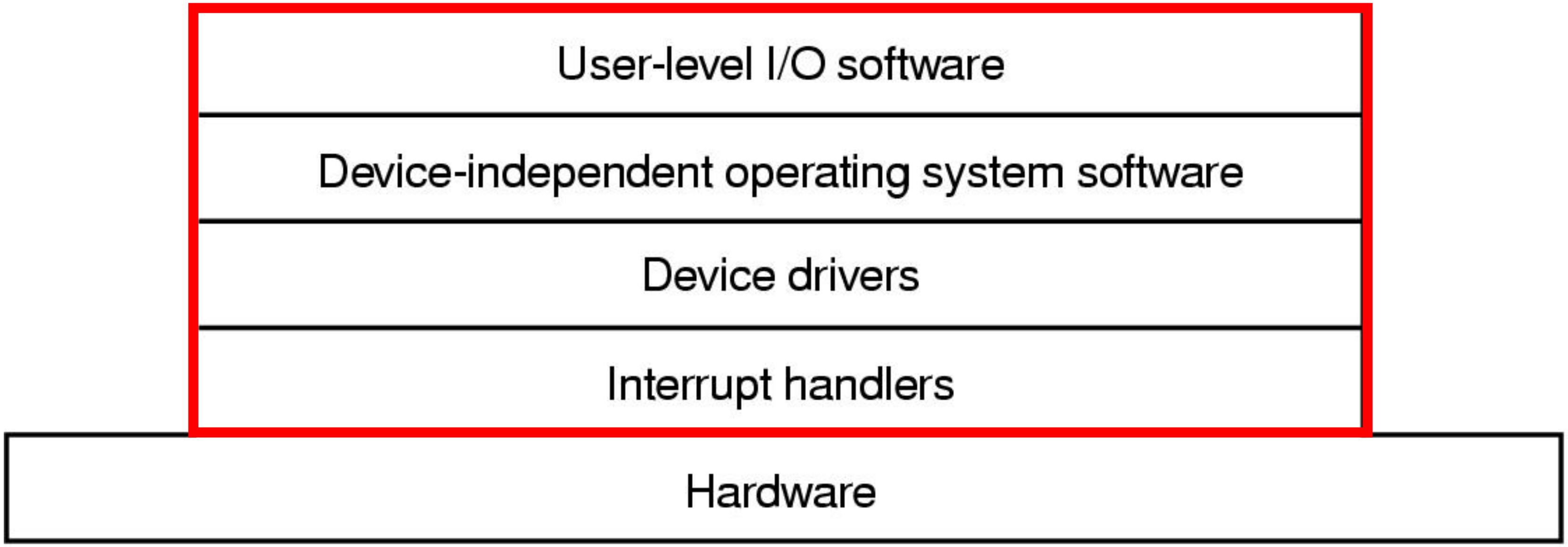
Device Variations: Character vs. Block



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

I/O Layers: Overview



Interrupt Handler

Interrupt handler

- Process each interrupt

- For block devices:
  - on transfer completion, signal device handler
- For character devices
  - when character transferred, process next character

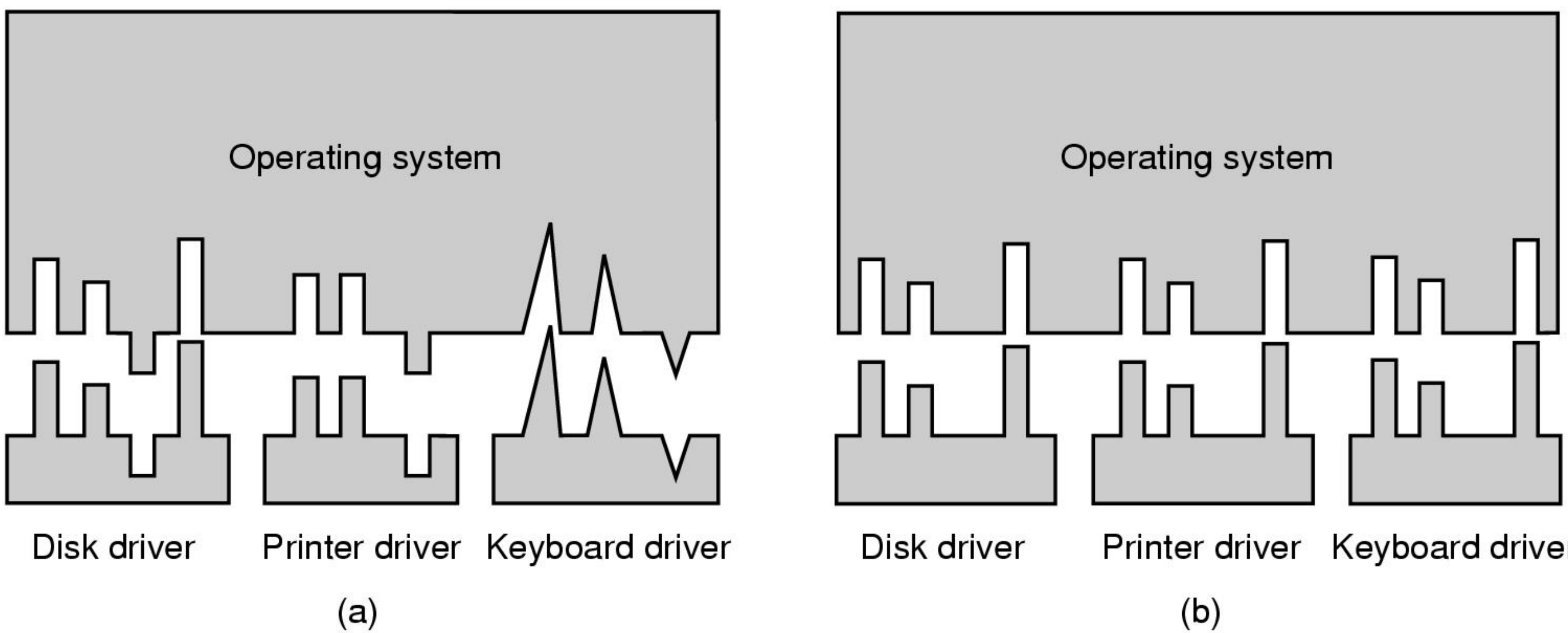
Device Driver

Device handler/driver

- Handles one device type
  - but may control multiple devices of same type

- Implements block read or write
- Access device registers
- Initiate operations
- Schedule requests
- Handle errors

Device Independent OS Layer I



(a) without standard driver interface

(b) with standard driver interface

Device Independent OS Layer II

Device independent layer provides device independence

- Mapping logical to physical devices (naming and switching)
- Request validation against device characteristics
- Allocation of dedicated devices
- Protection/user access validation
- Buffering for performance and block size independence
- Error reporting

Dedicated vs. Shared Device Allocation

Dedicated device (e.g. DVD writer, terminal, printer, ...)

- Simple policy:
  - Open fails if already opened
  - Alternatively, queue open requests
- Allocated for long periods
- Only allocated to authorised processes

Shared device (e.g. disks, window terminals, ...)

- OS provides file system for disks

Device Allocation: Spooling

Blocking user access to allocated, nonsharable devices?

- – Causes delays and bottlenecks

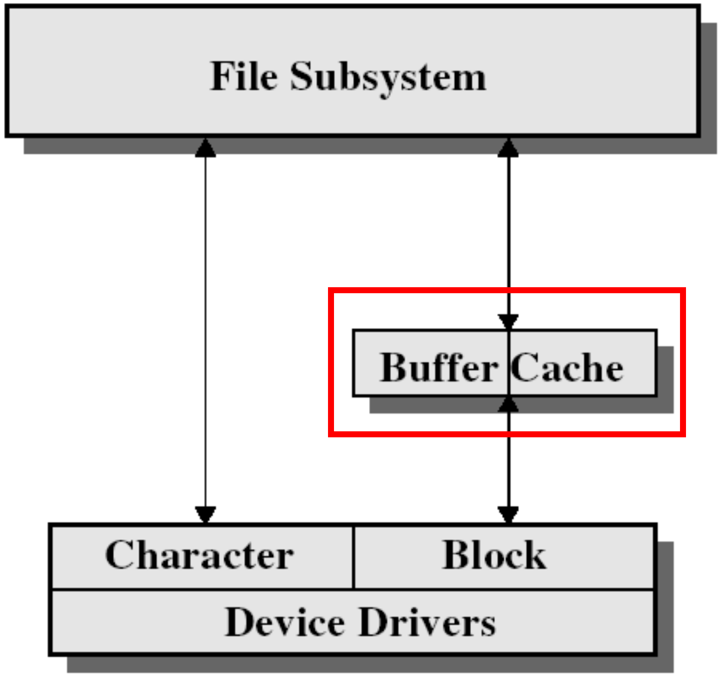
☛ Spool to intermediate medium (disk file)

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

Buffering


File Subsystem
Buffer Cache
Character
Block
Device Drivers

Buffered vs. Unbuffered I/O

Buffered I/O

Output: User data transferred to OS output buffer
Process continues and only suspends when buffer full

Input: OS reads ahead; reads normally satisfied from buffer
Process blocks when buffer empty

- Used to smooth peaks in I/O traffic
- Caters for differences in data transfer units between devices

Unbuffered I/O

- Data transferred directly from user space to/from device
  - Each read/write causes physical I/O
  - Implies device handler used for each transfer
- High process switching overhead (e.g. per character)

User-Level I/O Interface

User interface

- I/O operations: open, close, read, write, seek
- OS I/O library procedures to set up parameters
  - Must be device independent

- Synchronous or asynchronous
- Blocking or non-blocking

- Unix: Access virtual devices as files

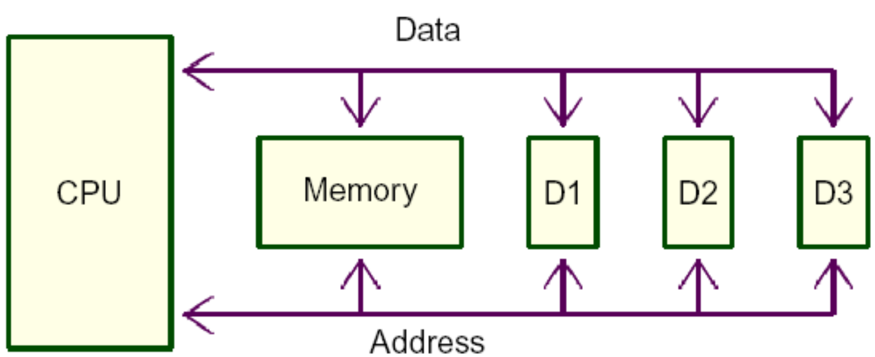
Device Drivers

Memory-Mapped I/O

Device addressed as memory location

Example: Disabling the I2S clock on Raspberry PI:

```
*(clk+0x26) = 0x5A000000;
*(clk+0x27) = 0x5A000000;
```



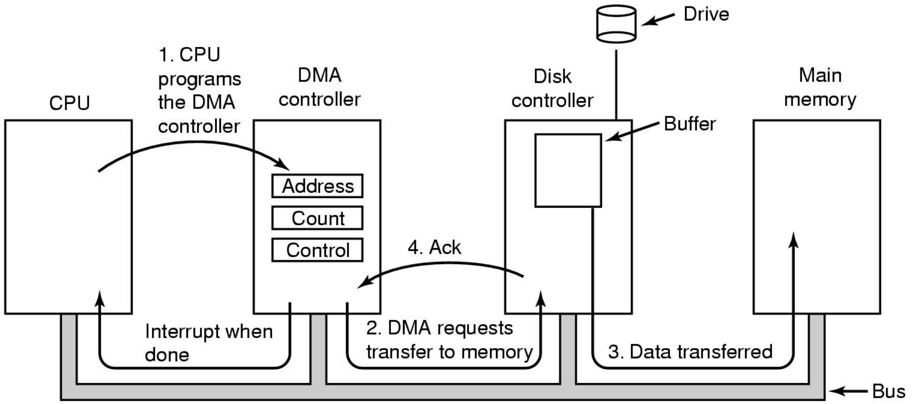
More flexible

Ways to do I/O

1. Programmed I/O

2. Interrupt-Driven I/O

3. I/O using Direct Memory Access (DMA)



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

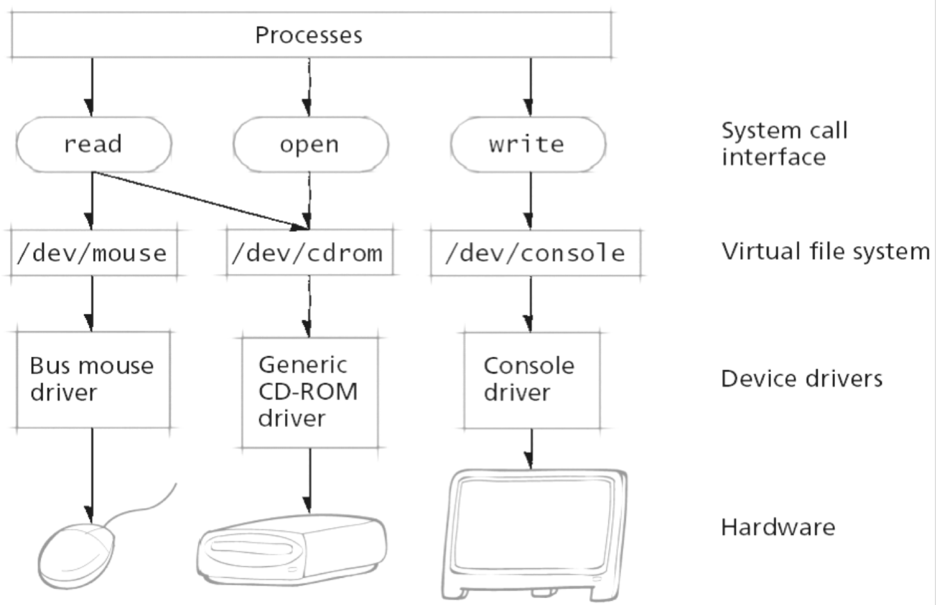
c/b major minor file name

```
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

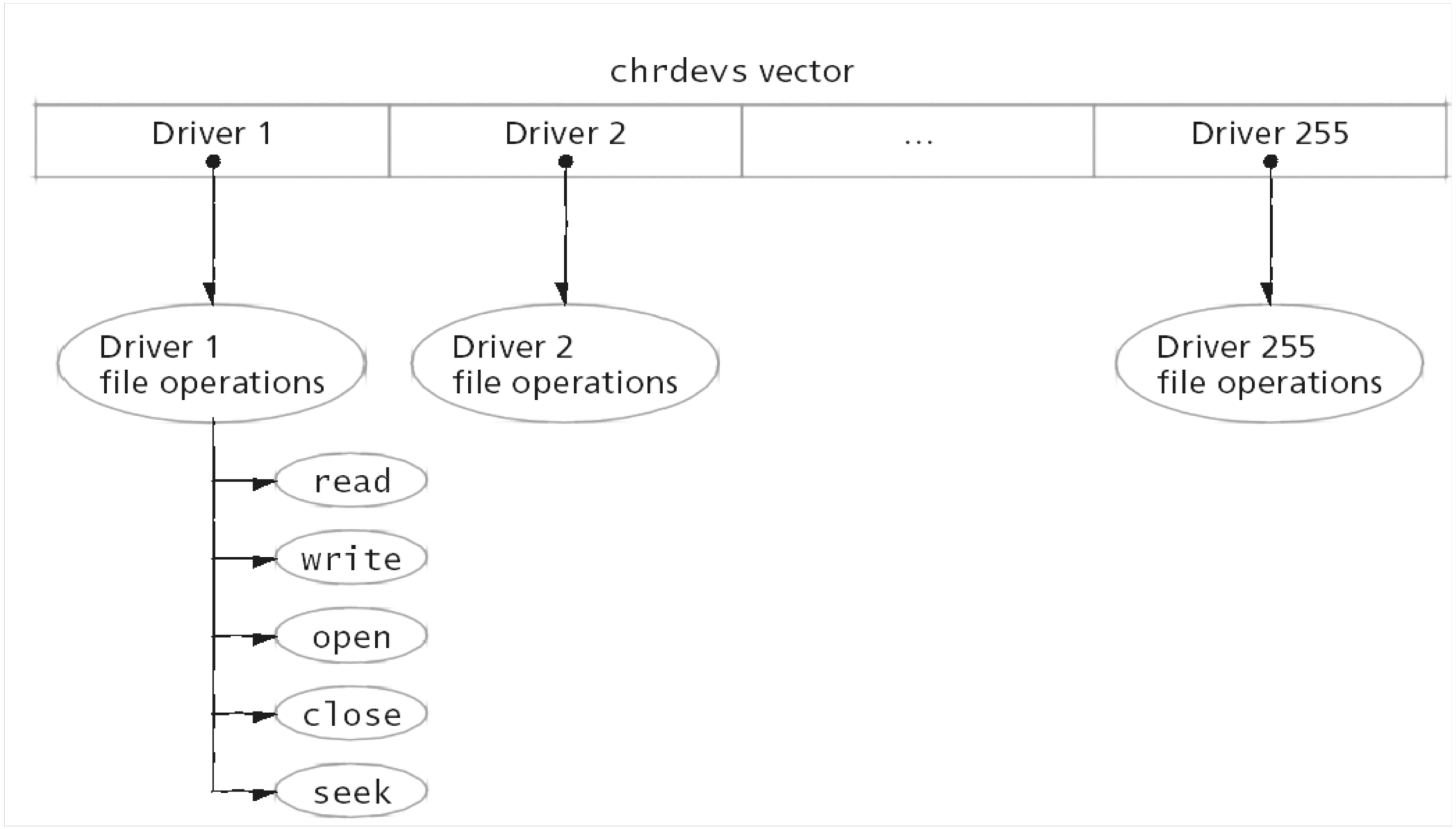
Character device

- Transmits data as **stream of bytes**
- 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 Classes

I/O classes

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

Sockets

Allow **bidirectional** communication

Can be used to exchange information both locally and across a network

- Unlike pipes which are identified by machine specific file descriptors

Two types of sockets:

- **TCP** (stream sockets)
- **UDP** (datagram sockets)

Linux I/O API I

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

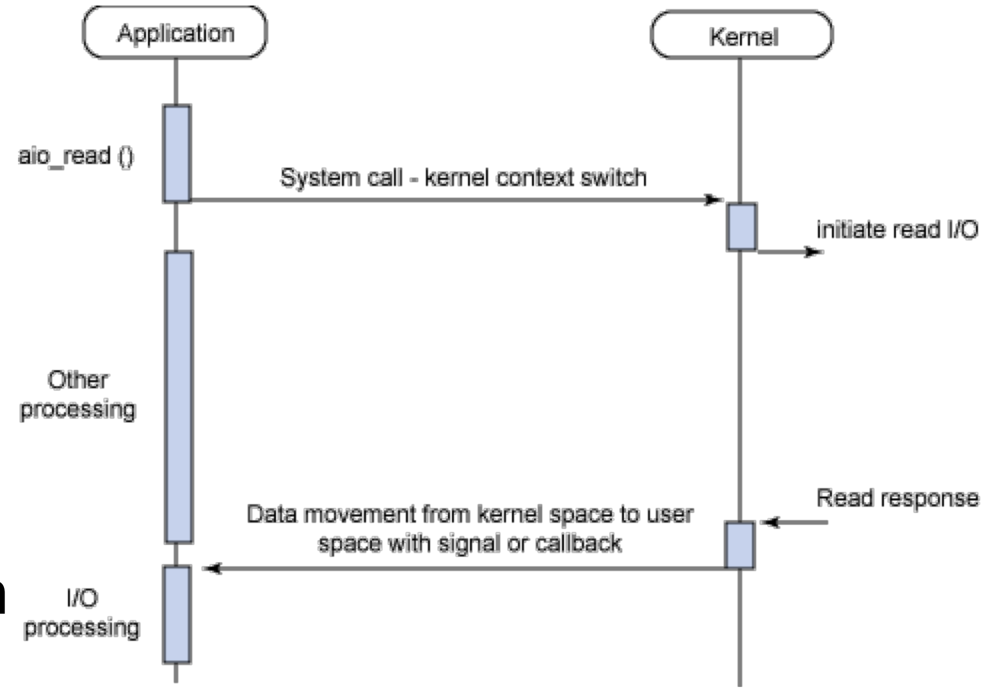
Non-blocking I/O

- 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 in parallel 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*/
}
```

Tutorial Question

In which of the four I/O software layers (user-level I/O software, device-independent OS software, device drivers and interrupt handlers) is each of the following done?

(a) Computing the track, sector and head for a disk read

(b) Maintaining a cache of recently used blocks

(c) Writing commands to the drive registers

(d) Checking to see if the user is permitted to use the device

(e) Converting binary integers to ASCII for printing

Tutorial Answer

(a) Computing the track, sector and head for a disk read

(b) Maintaining a cache of recently used blocks

(c) Writing commands to the drive registers

(d) Checking to see if the user is permitted to use the device

(e) Converting binary integers to ASCII for printing