Device Management

Anandha Gopalan
(with thanks to D. Rueckert, P. Pietzuch, A. Tannenbaum and R. Kolcun)
axgopala@imperial.ac.uk

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

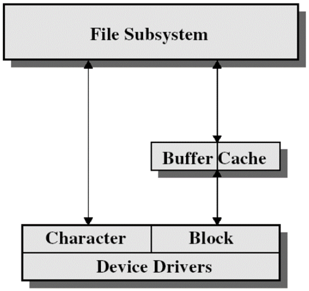
Character vs. Block

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



How does the OS actually communicate with the hardware?

CPU and Devices Communication

Each hardware controller has a few registers used for communication with the CPU

OS can write to these registers to command the device

- 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
- . . .

Imperial College London

I/O Software
Goals

Device independence from

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

Uniform naming → name of a file should be a string or integer and not depend on the device in any 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

Techniques

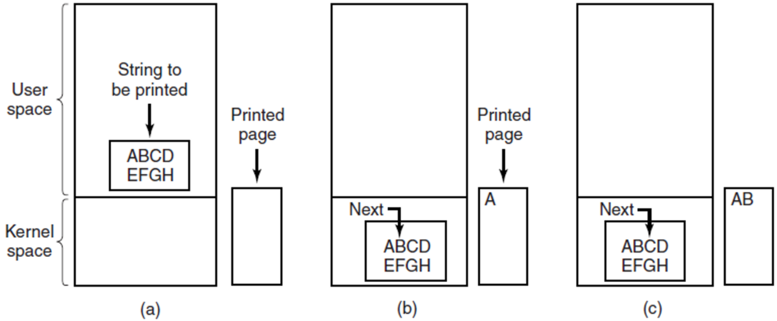
Programmed I/O

Interrupt Driven I/O

Direct Memory Access I/O

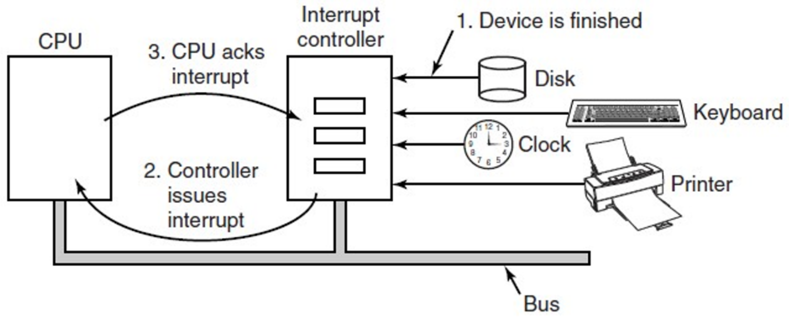
Example used: Printing a string

Programmed I/O



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

Interrupts



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

Imperial College London

Interrupt Driven I/O

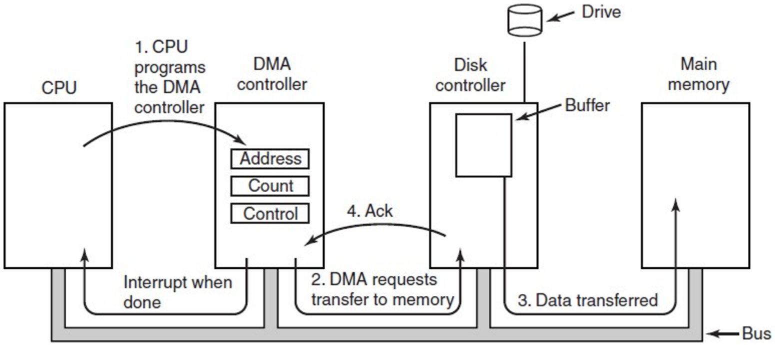
```
copy_from_user (buffer, p, count);
enable_interrupts ();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler ();
```

Code executed at the time the print system call is made

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

Interrupt service procedure for the printer

Direct Memory Access (DMA)



Operation of a DMA transfer

Imperial College London

DMA I/O

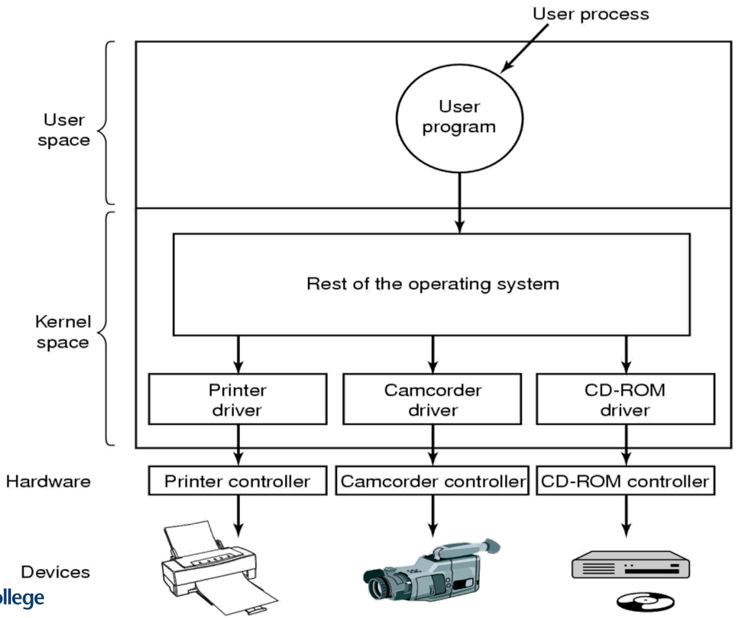
```
copy_from_user (buffer, p, count);
set_up_DMA_controller ();
scheduler ();
```

Code executed when the print system call is made

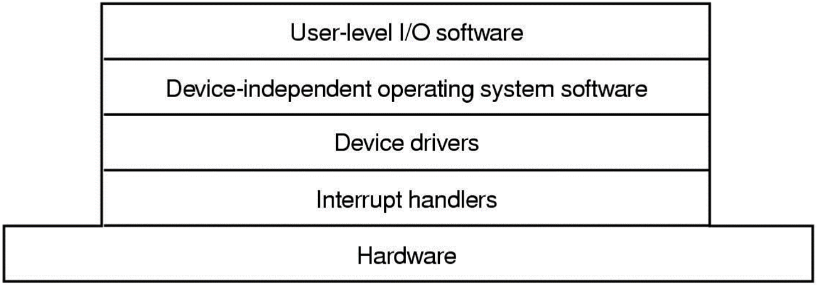
```
acknowledge_interrupt ();
unblock_user ();
return_from_interrupt ();
```

Interrupt service procedure

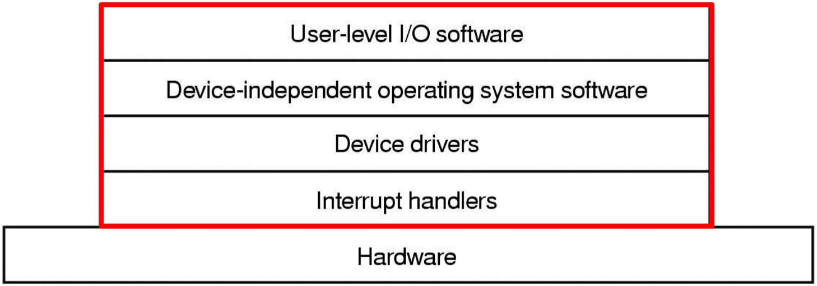
I/O Layering


User process
User program
User space
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



I/O Layers
Overview



Interrupt Handlers
Steps

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

Device Drivers

Device-specific code for controlling an I/O device

A driver for a mouse differs from a driver for a HDD

Handles one device type, or at most, one class of closely related devices

Part of kernel → a buggy driver can cause crash of the system

Positioned below the rest of the OS with direct access to hardware controllers

Most OSs 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.

Allowed to call only a handful of system calls, e.g. to allocate memory for a buffer

Device-Independent I/O Software

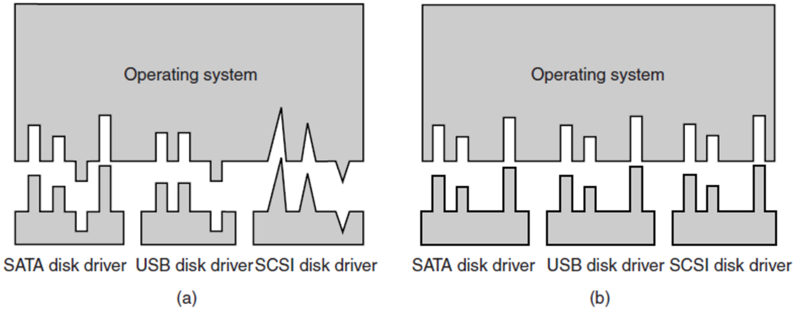
Some parts are device-specific but others are device independent

There is no strict boundary between device-specific and device independent software and varies between OSs

Most common device independent functions

- 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

Imperial College London

Uniform Interfacing for Device Drivers II

Interface between the driver and OS is defined

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

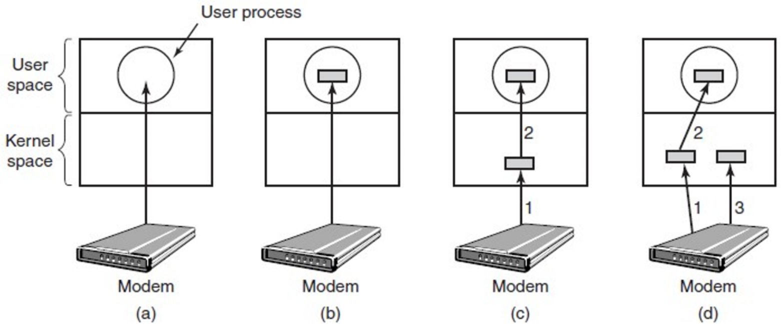
Often the driver contains a table with pointers into itself for these functions

OS 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 $\rightarrow$ 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

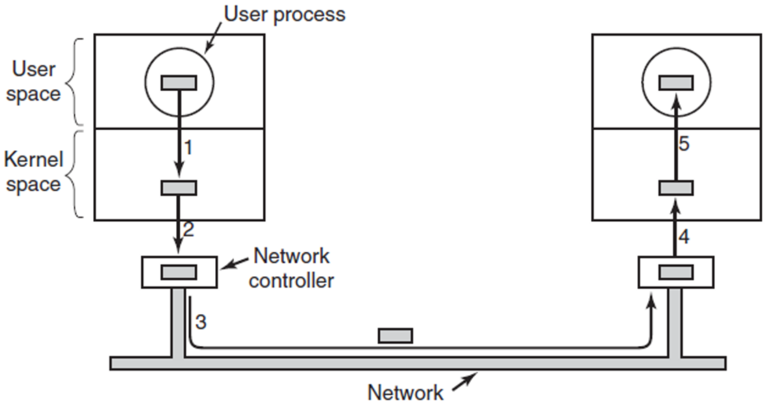
Buffering II

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 $\rightarrow$ e.g. when sending data over a slow telephone line

Imperial College London

Buffering III



Networking may involve multiple 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 keyboard, read from printer, read from invalid buffer address, read from disk 3 when there's only two
  - 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 to decide what to do, whether to try to solve the problem or report back the error code

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)

Imperial College London

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

Example → `count = write (fd, buffer, nbytes)`

Procedure puts parameters in the appropriate place for the system call

Other procedures may 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

Some devices, such as CD-ROM recorders, can be used only by a single process at any given moment and cannot be shared

- Blocking user access to these devices causes delays and bottlenecks

Spool to intermediate medium (disk file)

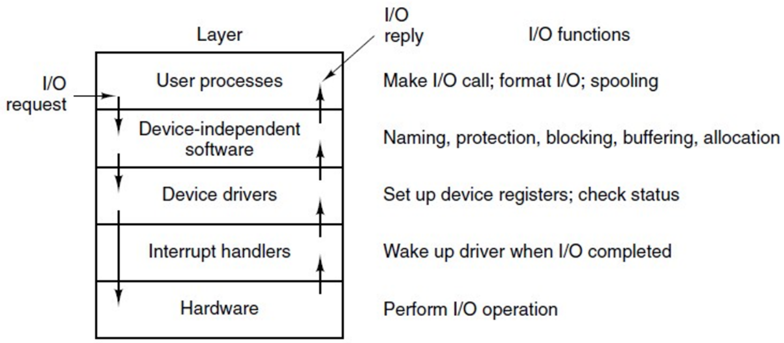
Spooling is a way of dealing with dedicated I/O devices in a multiprogramming system

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 non-sharable devices
- Reduces I/O time $\rightarrow$ gives greater throughput

User-Space I/O Software III


Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

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
- Loads modules on demand

Every LKM consists of two basic functions (minimum)

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

Load module by using the **insmod** command → normally restricted to root

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 number controlled by same driver
- Minor numbers enable system to distinguish between devices of same class

Linux
Device Drivers and Access

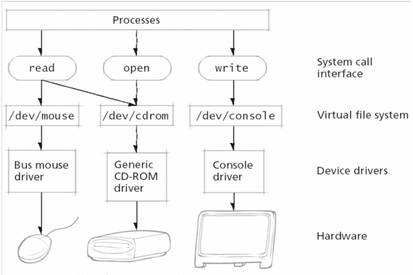
Most devices represented by device special files

Device files accessed via virtual file system (VFS) (/dev)

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

List of devices in system → /proc/devices

Linux provides the `ioctl` system call that supports special tasks → retrieving status information from printer



I/O subsystem overview

Character Device

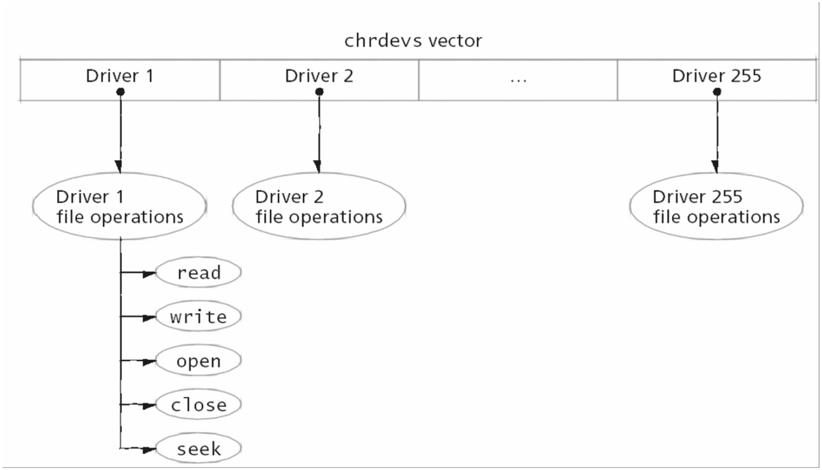
- Transmits data as stream of bytes
- Represented by `device_struct` structure, which contains
  - Driver name
  - Pointer to driver's `file_operations` structure
- All registered drivers referenced by `chrdevs` vector

`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



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

When data from block device requested, kernel first searches cache

- If found, data copied to processs address space
- Otherwise, typically added to request queue

Direct I/O

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

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

I/O API Calls I

`fd = create (filename, permission)`

- Opens file for reading/writing
- `fd` is the index to the file descriptor
- Permission is used for access control

`fd = open (filename, mode)`

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

`close (fd)`

I/O API Calls II

`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 `numbytesread`

I/O API Calls III

`pipe (&fd[2])`

- Creates pipe
- `fd` is an array of 2 integers (`fd[0]` 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 an array of control chars

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

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

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

Blocking vs. Non-blocking I/O

Blocking I/O

- Call returns when operation completed
- Process suspended $\rightarrow$ 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

Asynchronous I/O

Process executes in parallel with I/O operation

- No blocking in interface procedure

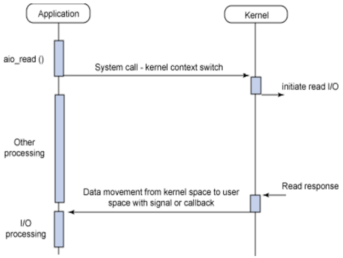
I/O subsystems notifies process upon completion

- Callback function, process signal, ...

Supports check/wait if I/O operation completed

Very flexible and efficient

Harder to use and potentially less secure



Application
Kernel
aio_read ()
System call - kernel context switch
initiate read I/O
Other processing
Read response
Data movement from kernel space to user space with signal or callback
I/O processing

Imperial College London