Introduction to Operating Systems

Chapter 7: Input/Output

Manuel

Fall 2017

Outline

1. Hardware I/O

2. Software I/O

OS and Input/Output

OS controls all I/O:

- Issue commands to the devices
- Catch interrupts
- Handle errors

The OS provides a simple and easy way to use interfaces for the rest the system

Representing devices

Two main categories:

- Block devices:
    - Stores information in blocks of fixed size
    - Can directly access any block independently of other ones
- Character devices:
    - Delivers or accepts a stream of characters
    - Not addressable, no block structure

Representing devices

Two main categories:

- Block devices:
    - Stores information in blocks of fixed size
    - Can directly access any block independently of other ones
- Character devices:
    - Delivers or accepts a stream of characters
    - Not addressable, no block structure
- Others: ex. clock: cause interrupts at some given interval

Device controller

Most devices have two parts: **mechanical**, the device itself and **electronic** that allows the communication with the device.

The electronic part is called the device controller:

- Allows to handle mechanical part in an easier way
- Performs error corrections for instance in the case of a disk
- Prepares and assemble blocks of bits in a buffer
- The blocks are then copied into the memory

Memory-mapped I/O

The CPU communicates with the device using **control registers**:

- OS writes on registers to: deliver/accept data, switch device on/off
- OS reads from registers to: know device's state

Modern approach:

- Map the buffer to a memory address
- Map each register to a unique memory address or I/O port

Memory-mapped I/O

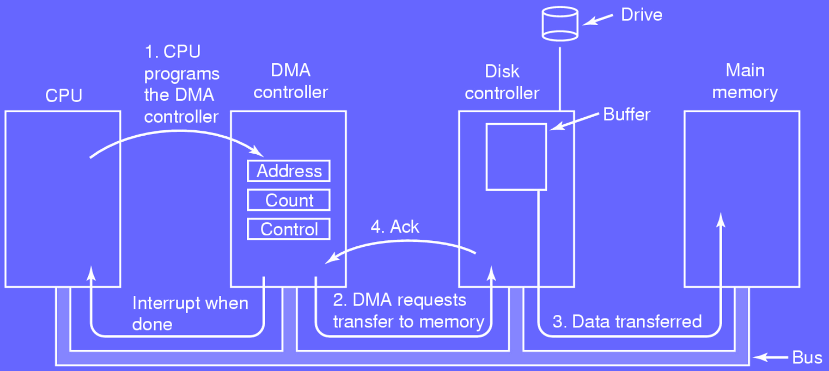
Pro – Cons

Strengths:

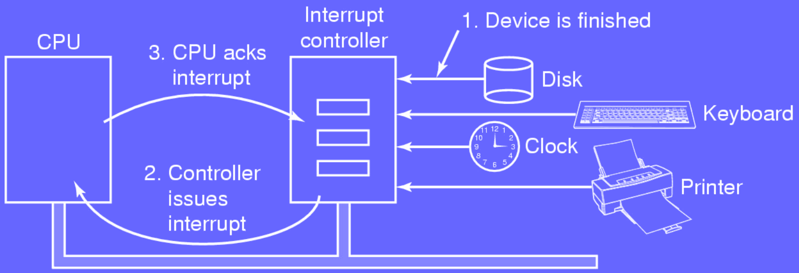
- Access memory not hardware → no need for assembly
- No special protection required → control register address space not included in the virtual address space
- Flexible → a specific/privileged user can be given access to a particular device
- Different drivers in different address spaces → reduces kernel size + no interference between drivers

Weakness: memory words are cached → what if the content of control register is cached?

Direct Memory Access



Interrupts


CPU
3. CPU acks interrupt
Interrupt controller
1. Device is finished
Disk
Keyboard
Clock
Printer
2. Controller issues interrupt

Precise vs. imprecise interrupts

Initial setup: on an interrupt push Program Counter (PC) and PSW on the stack, handle interrupt, retrieve program counter and PSW and resume process.

New setup: pipelined or superscalar CPU. What consequences?

Precise vs. imprecise interrupts

Initial setup: on an interrupt push Program Counter (PC) and PSW on the stack, handle interrupt, retrieve program counter and PSW and resume process.

New setup: pipelined or superscalar CPU. What consequences?

Precise interrupt:

- PC saved in a known place
- All instructions before the one pointed by PC have been executed
- No instruction after the one pointed by PC has be executed
- Execution state of the instruction pointed by PC is known

An interrupt which is not precise is called **imprecise interrupt**.

Dealing with imprecise interrupts

Difficult to figure out what happened and what has to happen:

- Instructions near PC are in different stages of completion
- General state can be recovered if given many details on the internal state
- Code to resume process is complex
- Many details implies much memory used

Conclusion: slow interrupts

Dealing with imprecise interrupts

Difficult to figure out what happened and what has to happen:

- Instructions near PC are in different stages of completion
- General state can be recovered if given many details on the internal state
- Code to resume process is complex
- Many details implies much memory used

Conclusion: slow interrupts

Possible to get precise interrupts at the cost of complex interrupt logic within the CPU. CPU area used to get precise interrupts is wasted.

Outline

1. Hardware I/O

2. Software I/O

Key concepts

Main goals on the design of I/O softwares:

- Device independence: whatever the support, files are handled the same way
- Uniform naming: devices organised by type with a name composed of string and number
- Error handling: fix error at lowest level possible
- Synchronous vs. asynchronous: OS decides if interrupt driven operations look blocking to user programs
- Buffer: need some temporary space to store data
- Shared vs. dedicated devices: more than/only one user can use a device at a time

Communications strategies

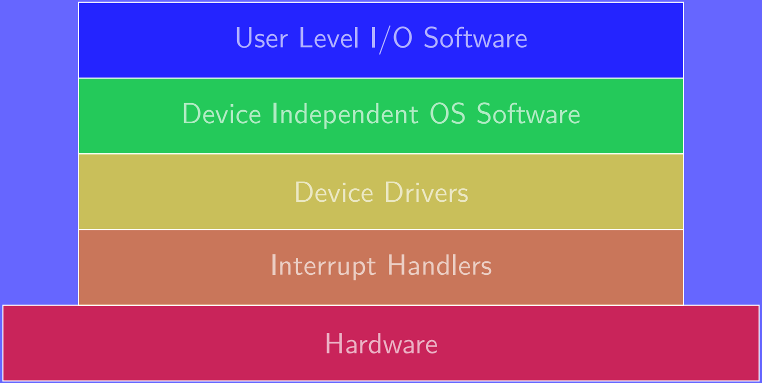
Three communications strategies:

Communications strategies

Three communications strategies:

1. **Programmed I/O:** copy data into kernel space, then fill up device register and wait in tight loop until device register is empty, fill it up. . .
2. **Interrupt I/O:** copy data into kernel space, then fill up device register. The current process is blocked so the scheduler is called to let another process run. When the register is empty an interrupt is sent, the new current process is stopped and register is filled up. . .
3. **DMA:** similar to programmed I/O, but DMA does all the work.

Layers



User Level I/O Software

Device Independent OS Software

Device Drivers

Interrupt Handlers

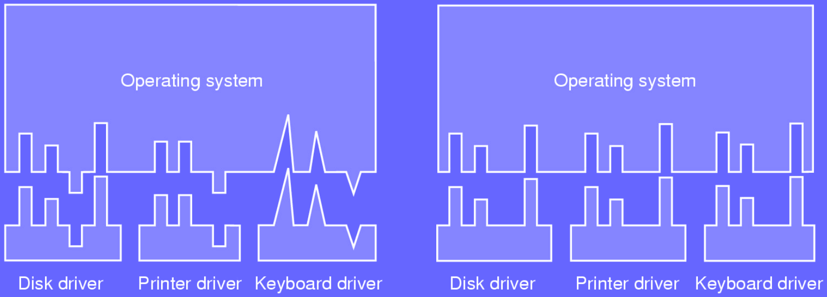
Hardware

Interrupt handlers

Actions to performs on an interrupt:

1. Save registers
2. Setup a context for handling the interrupt
3. Setup a stack
4. Acknowledge interrupt controller + re-enable interrupts
5. Load registers
6. Extract information from interrupting device's controller
7. Choose a process to run next
8. Setup MMU and TLB for next process
9. Load new process registers
10. Run new process

Device drivers interface



- Same class of device has a common basic set of functionalities
- OS defines which functionalities should implemented
- Use a table of function pointers to interface device driver with the rest of the OS
- Uniform naming at user level

Job of device drivers

Basic functions of a driver:

- Initialization
- Accept generic requests (e.g. read/write)
- Log events
- Retrieve device status
- Handle device specific errors
- Specific actions depending on the device

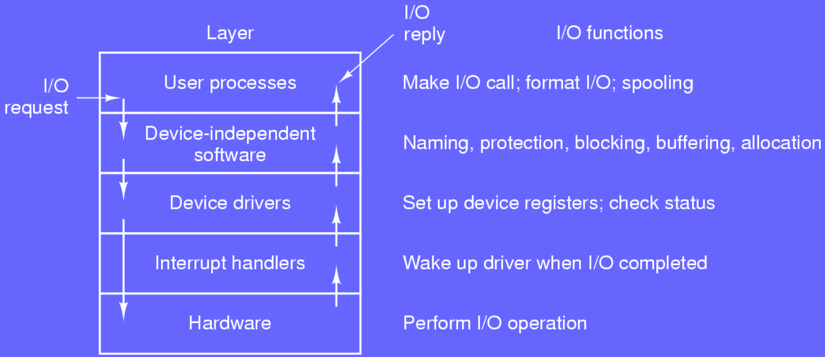
Device driver should react nicely even under special circumstances

Drivers in the OS

General remarks on drivers:

- Location: user or kernel space
- Drivers can be compiled in kernel
- Drivers can be dynamically loaded at runtime
- Drivers can call certain kernel procedures (manage MMU, timers, DMA etc. . .)
- I/O errors framework is device independent
- Clean and generic API such that it is easy to write new drivers

Software I/O and OS



I/O request

I/O reply

Layer	I/O functions
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

Key points

- What are the two main categories of devices?
- Explain the difference between precise and imprecise interrupts
- List three communication strategies
- Why should implementation be done using layers?

Thank you!