**CS323 Operating Systems**

# Input/Output II

Yuanyuan Zhou

Lecture 20

3/7/2003

## Content of this lecture

- Administrative announcements
- I/O software
- Summary

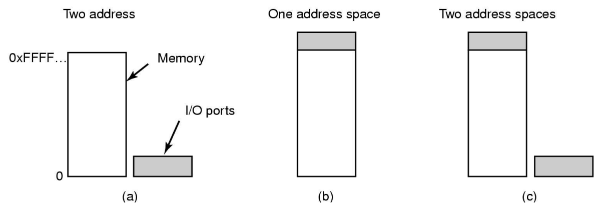
## Administrative

- MP2
- Midterm1
  - Time: 3/10, 7-8pm,
    - DCL 1320, students with last name A-L
    - MSEB 100, students with last name M-Z
  - Conflict exam: 3/11, 5-6pm, DCL 1320

## Review

- I/O basic concepts
- Interrupts
- DMA
- Memory Mapped I/O

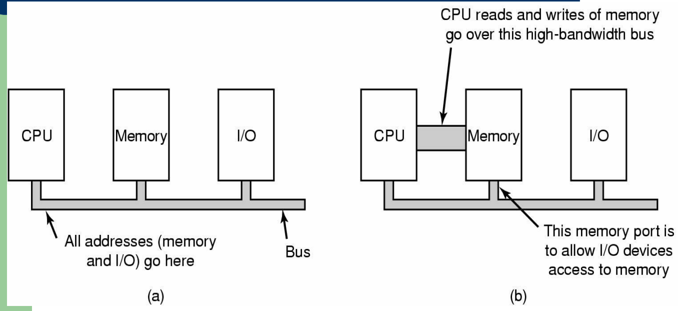
## Memory-Mapped I/O (1)



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

5 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Memory-Mapped I/O (2)


CPU reads and writes of memory go over this high-bandwidth bus
CPU
Memory
I/O
CPU
Memory
I/O
All addresses (memory and I/O) go here
Bus
This memory port is to allow I/O devices access to memory
(a)
(b)

(a) A single-bus architecture
(b) A dual-bus memory architecture

6 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## I/O Software: Principles

- device independence
  - possible to write programs that can access any IO device without having to specify the device in advance.
  - read file as input on a floppy, hard disk, or CD-ROM, without modifying the program for each device
- uniform naming
  - the name of a file or a device should simply be a string or an integer and not depend on the device in any way
  - Example: In Unix all files and devices are addressed the same way by a path name.
- Synchronous (blocking) vs asynchronous (interrupt-driven) transfers
  - Example: most devices are asynchronous. User programs are easier to write if the IO operations are synchronous. Hence, it is up to OS to make asynchronous operations look like synchronous to the user programs.

7 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Principles (cont.)

- Buffering
  - data might need to be buffered because they might not be stored immediately
  - Example: Network packets come in off the network, and need to be buffered for protocol processing, audio buffering between audio speakers and network.
- shared vs dedicated devices
  - allowing for sharing, considering two users having open files on the same device, or allowing dedicated devices and deal with deadlock problems.
  - Example: Disk, audio devices.

8 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Programmed I/O

- Three ways to perform I/O
  - Interrupt-driven IO
  - IO using DMA
  - **programmed IO**.
    - Every IO operation is programmed and controlled.
    - Example: printing a file from user program to the printer means that data is first copied to the kernel, then the OS enters a tight loop outputting the characters one at a time.
    - Essential aspect of programmed IO is that the CPU continuously polls the device to see if it is ready to accept another character. It uses **polling** (or busy waiting) behavior.
    - Programmed IO is simple but required CPU full time until all the IO is done.

## Host-Controller Interface: Polling

- Consider a producer-consumer relation between controller and host
- Controller indicates through busy bit (status bit) if it is busy to work or not
- Host signals its wishes via command-ready bit (command register)
- Polling behavior

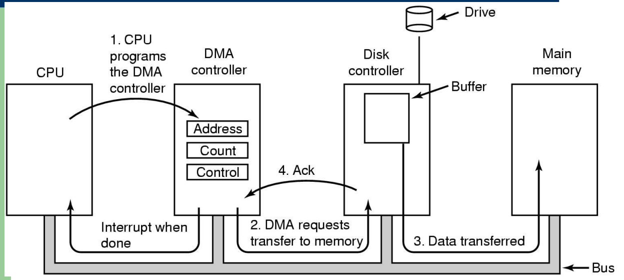
## Handshaking Protocol

- Host reads busy bit (poll) until bit is clear
- Host sets write bit in command register and writes byte into the data-out register
- Host sets the command-ready bit in command register
- When controller notices command-ready bit, it sets busy bit and starts to work
- Controller reads the command register and sees the write command. It reads data-out and performs I/O
- Controller clears the command-ready bit, error bit and busy bit

## Interrupt-Driven IO

- CPU hardware has the interrupt report line that the CPU senses after executing every instruction.
  - Device raises an interrupt
  - CPU catches the interrupt and saves the state (e.g., instruction pointer)
  - CPU dispatches the interrupt handler
  - Interrupt handler determines the cause of the interrupt, services the device and clears the interrupt.

## Direct Memory Access (DMA)



DMA controller feeds the characters to the printer one at the time, without CPU being bothered. DMA is actually the programmed IO, only with DMA controller doing the work.

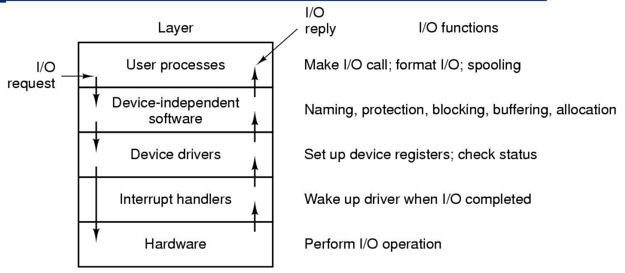
## Discussion

- Tradeoffs between
  - Programmed I/O
  - Interrupt-driven I/O
  - I/O using DMA

## Tradeoffs

- Interrupt-drive IO:
  - Pro: save CPU time for busy polling
  - Con:  triggering interrupt takes time, too.
- Programmed I/O
  - Pro: require no interrupt or DMA support
  - Con: waste CPU time
- I/O using DMA:
  - Pro: relieve CPU from I/O operation
  - Con: DMA is slower than CPU

## I/O Software



| 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 |

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

## I/O Software Layer: Principle

- Interrupts are facts of life, but should be hidden away, so that as little of the OS as possible knows about them.
- The best way to hide interrupts is to have the driver starting an IO operation block until IO has completed and the interrupt occurs.
- When interrupt happens, the interrupt handler handles the interrupt.
- Once the handling of interrupt is done, the interrupt handler unblocks the device driver that started it.
- This model works if drivers are structures as kernel processes with their own states, stacks and program counters.

## Device Drivers

- Device-specific code to control an IO device, is usually written by device's manufacturer
  - Each controller has some device registers used to give it commands. The number of device registers and the nature of commands vary from device to device (e.g., mouse driver accepts information from the mouse how far it has moved, disk driver has to know about sectors, tracks, heads, etc).
- A device driver is usually part of the OS kernel
  - Compiled with the OS
  - Dynamically loaded into the OS during execution
- Each device driver handles
  - one device type (e.g., mouse)
  - one class of closely related devices (e.g., SCSI disk driver to handle multiple disks of different sizes and different speeds.).
- Categories:
  - Block devices
  - Character devices

## Functions in Device Drivers

- Accept abstract read and write requests from the device-independent layer above;
- Initialize the device;
- Manage power requirements and log events
- Check input parameters if the are valid
- Translate valid input from abstract to concrete terms
  - e.g., convert linear block number into the head, track, sector and cylinder number for disk access
- Check the device if it is in use
- Control the device by issuing a sequence of commands. The driver determines what commands will be issued.

## Device Driver Protocol

- After driver knows which commands to issue, it starts to write them into controller's device registers
- After writing each command, it checks to see if the controller accepted the command and is prepared to accept the next one.
- After command have been issues, either (a) the device waits until the controller does some work and it blocks itself until interrupt comes to unblock it; or (b) the device doesn't wait because the command finished without any delay.

## Device-Independent IO Software

- Functions :
  - Uniform interfacing for device drivers
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - providing a device-independent block size

21 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Uniform Interfacing for Device Drivers:

- Goal: how to make all IO devices and drivers look similar
- Problem: interface between drivers and the rest of OS might differ, which might mean that each new driver would require a lot of new programming effort
- Solution: There are specifications which functions and kernel calls to use in device drivers to make the drivers pluggable. Not all devices are absolutely identical, but usually there are only small number of device types.
- Example: block and character devices, but they also have many functions common

22 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Naming of IO devices

- The device-independent software takes care of mapping symbolic device names onto the proper driver.
- Example
  - in UNIX, the symbolic device name dev/disk0 uniquely specifies the disk

23 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Buffering

- Buffer is a memory area that stores data while they are transferred between two devices or between a device and an application.
- Reasons of buffering:
  - cope with speed mismatch between the producer and consumer of a data stream - use *double buffering*
  - adapt between devices that have different data-transfer sizes
  - support of copy semantics for application I/O - application writes to an application buffer and the OS copies it to the kernel buffer and then writes it to the disk.
  - Unbuffered input strategy is ineffective, as the user process must be started up with every incoming character.
  - Buffering is also important on output.
- **Caching**
  - cache is region of fast memory that holds copies of data and it allows for more efficient access.
- Difference?

24 3/6/2003 CS 323 - Operating Systems, Yuanyuan Zhou

## Buffering Issues

- Pros:
  - Unbuffered input strategy is ineffective, as the user process must be started up with every incoming character.
- Buffering in user space
  - if the buffer is paged out, where should the character be put? One solution would be to pin the pages into the real memory, but this could degrade performance if many processes start to lock pages in memory.
- Buffering in kernel followed by copying to user space:
  - far more efficient than the previous methods, but again the problem occurs if the kernel buffer becomes full and the user buffer page is paged out. In this case again there is no space to put the incoming data.
- Double buffering in the kernel
  - as long as one kernel buffer takes incoming data, the other buffer is copied to the user space.
- Note: if data get buffered too many times, performance suffers.

## Error Reporting

- Programming IO Errors
  - these errors occur when a process asks for something impossible (e.g., write to an input device such as keyboard, or read from output device such as printer).
- Actual IO Errors
  - errors occur at the device level (e.g., read disk block that has been damaged, or try to read from video camera which is switched off)
- The device-independent IO software detects the errors and responds to them by reporting to the user-space IO software.

## Allocating and releasing Dedicated Devices

- Some devices can be used only by single process at any given time
  - e.g., CD-ROM recorders, some audio devices
- Two approaches:
  - perform open on the special file for devices directly. If device is not available, open fails.
  - A process requests a device. If it is not available, the process is blocked, instead failing, and waits until the device become available.

## Device Independent Block Size

- Different disks have different sector sizes.
- The device independent IP software should hide this fact and provide uniform block size to higher layers.
  - introduce the same logical block at the user-space IO level and do the mapping between the logical block size and the underlying several sectors in the device independent IO software.

## User-Space IO Software

- Applications in user space call IO system calls to access IO.
- IO system calls are normally made by library procedures.
- Steps to perform an IO system call:
  - User-space library calls the device-independent IO software
  - Device-independent I/O software calls the corresponding device driver
  - Device driver accesses the device controller and through it the corresponding device.

## More on System Calls

- Interface between running application and operating system
- Assembly language macros or subroutines
- Need to use an SVC instruction
- Pass parameters through registers, in memory tables, or on stack
- Unix has about 32 system calls:
  - Read(), write(), open(), close(), fork(), exec(), ioctl(),

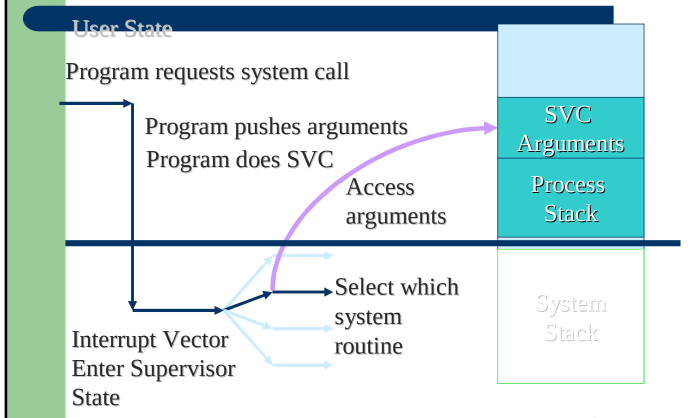
## Spooling

- Another way to do IO is the **spooling system**.
- Spooling is an approach to deal with IO devices in a multiprogramming system
  - Example: printer (spooled device)
- Spooling includes a special process, called *daemon*, and special directory, called *spooling directory*.
- Example: To print a file, the printing process creates the entire file, puts it in the spooling directory, and the daemon then takes out the file and prints it.

## How System Calls Work

1. SVC Causes a Trap
2. System Privilege Mode Entered
3. Trap Handler Decodes Which SVC
4. Branch to System Function
5. Access Arguments From the Stack

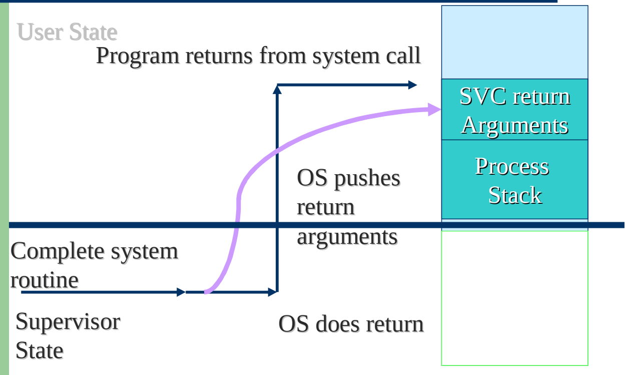
## Animation



## How System Return Work

6.Push Results on the Stack
7.Return to Trap Handler
8.Resume Application Privilege Mode
9.Return to Instruction Following SVC

## Animation



## Blocking and Non-blocking I/O

- When an application calls a **blocking** system call, the execution of the application is suspended.
  - The application waits for the I/O result.
  - The physical action performed by I/O devices is generally asynchronous (they take unpredictable/varying amount of time)
- **Non-blocking** I/O system call means that the application calls the I/O system call, and returns quickly with the returned value
  - Examples: video application which reads from the disk and simultaneously displays video frames

## Asynchronous System Call

- Alternative to non-blocking system call is asynchronous system call.
- An asynchronous system call returns immediately, without waiting for the I/O to complete.
  - The application continues to execute its code and the completion of the I/O call is communicated to the application in the future
- Multi-threaded applications benefit from non-blocking system calls

## Performance

- Heavy demands on CPU to execute device driver code and to schedule processes fairly and efficiently as they block and unblock.
- Several principles to improve the efficiency of I/O:
  - reduce the number of context switches
  - reduce the number of times that data must be copied in memory while passing between device and application
  - reduce frequency of interrupts (user large transfers, smart controllers, polling, etc)
  - increase concurrency by using DMA controllers
  - move processing primitives into hardware
  - balance CPU, memory subsystem, bus and I/O performance
  - Caching
- Place to implement I/O functionality
  - device hardware, device driver, kernel, application?
  - Device-functionality progression (the lower level implementation, the better performance, efficiency, however also increased development cost, and decreased flexibility.

## Discussion

- Tradeoff between
  - Blocking I/O
  - Non-blocking I/O
  - Asynchronous I/O

## Reminder

- MP2
- Midterm1