CSE 3320
Operating Systems
I/O Devices

Jia Rao

Department of Computer Science and Engineering

http://ranger.uta.edu/~jrao

Recap of Previous Classes

- CPU scheduling
- Memory management
- File Systems
- This chapter → I/O devices (e.g., hard disks)

I/O of an OS

Main Function: Control all the computer’ s I/O devices

- Issue commands to the devices, catch interrupts, and handle errors
- Provide interface between the devices and the rest of system that is simple and easy to use

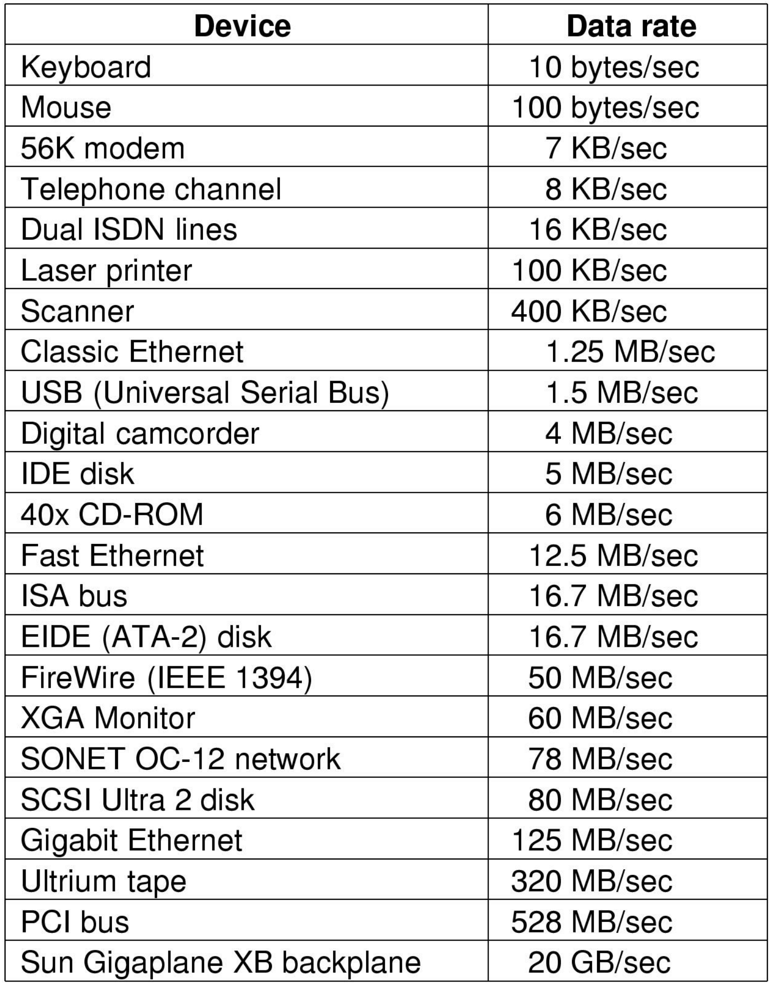
What we care I/O devices most as software designers and programmers?

The interface presented to us:

(1) The commands the hardware accepts
(2) The functions it carries out
(3) The errors that can be reported back

Principles of I/O Hardware: I/O Devices

- Block devices
- Character devices
- Others

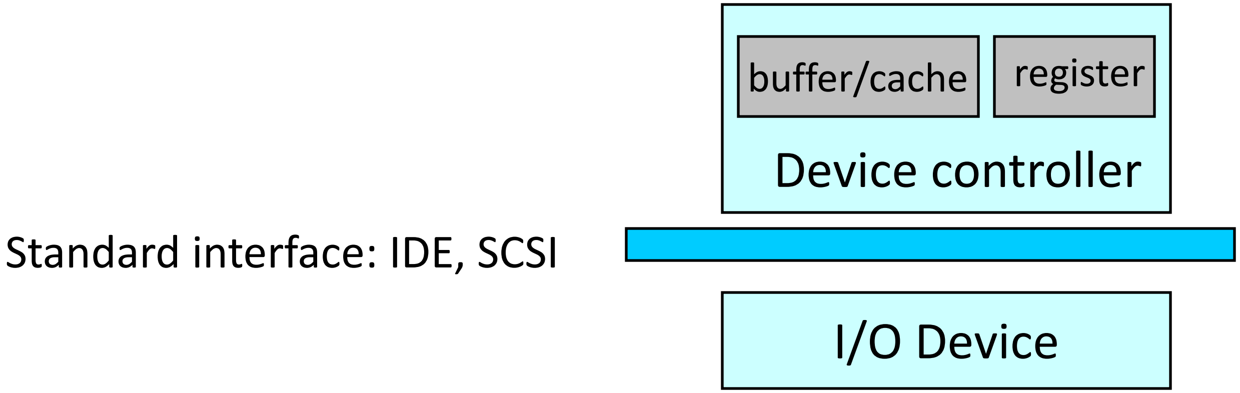


Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

Some typical device, network, and data base rates

Device Controllers

- I/O devices have two components:
  - mechanical component
  - electronic component
- The electronic component is the *device controller*
  - may be able to handle multiple but identical devices
- Controller's tasks
  - convert serial bit stream to block of bytes
  - perform error correction as necessary
  - make blocks available to main memory



CPU→I/O: Special I/O Instructions

- How does the CPU communicate with the control registers and the device data buffers? Two methods are available.
- Method I: Special (assembly) I/O Instructions (IBM 360)
  - Each control register is assigned an I/O *port* number (device #)

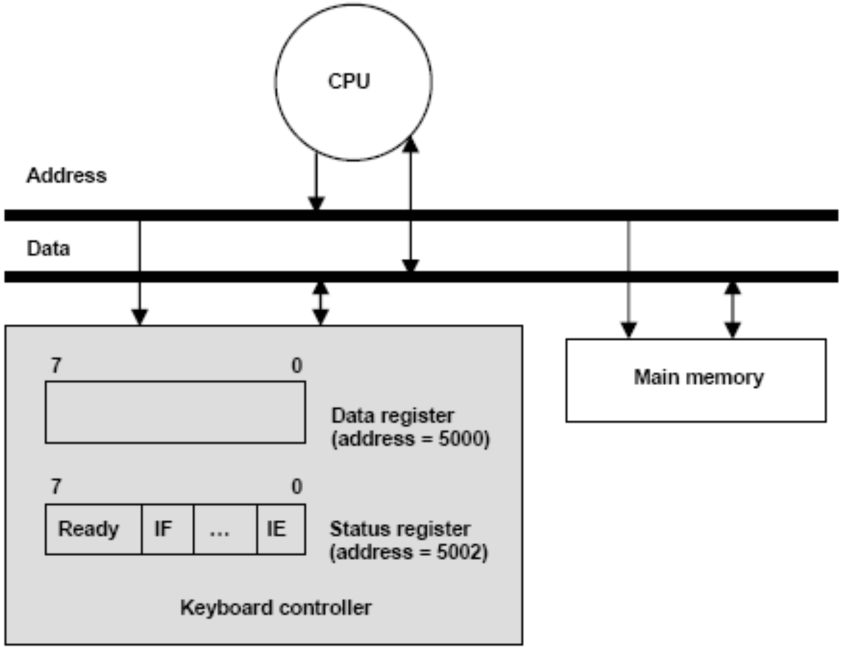
```
IN REG, PORT
OUT PORT, REG
```

  - The memory address space and I/O address space are different

```
IN R0, 4   ; port 4
MOV R0, 4             ; memory word 4
```

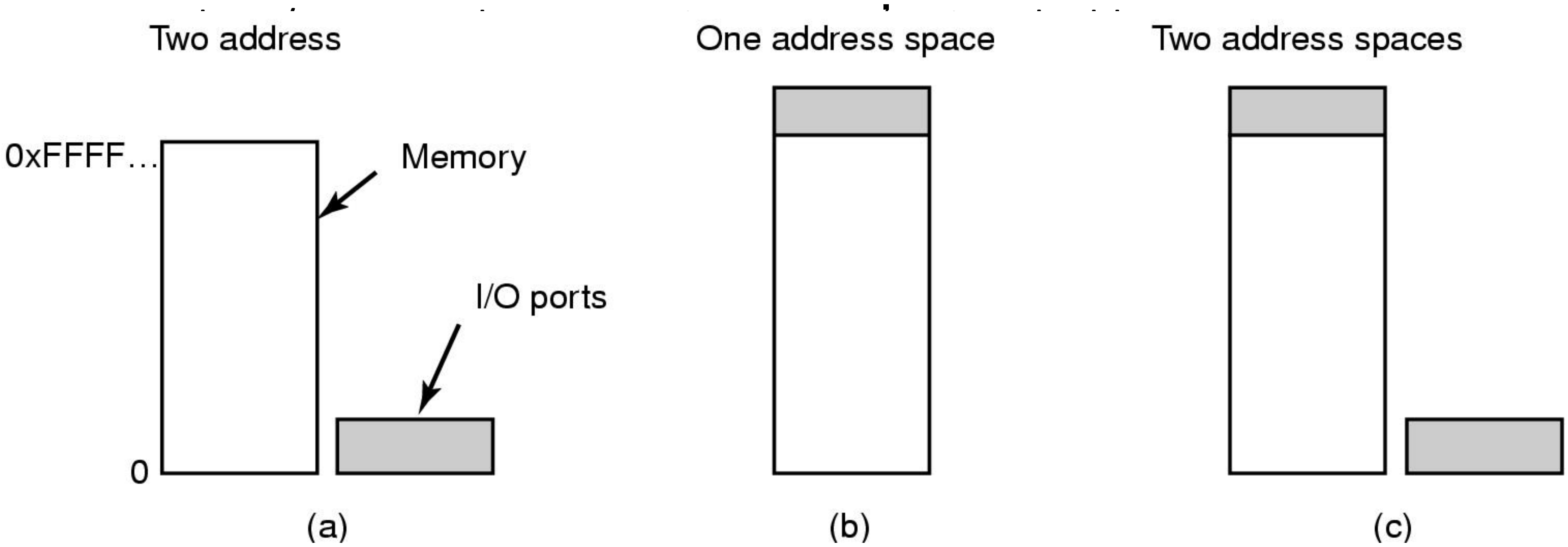
CPU→I/O: Memory-mapped I/O

- Method II: Memory-mapped I/O (Pentium)
  - Each register is assigned a unique memory address to which no memory is assigned; read/write to addresses are I/O operations
  - How it works?



CPU→I/O: Memory-mapped I/O (2)

- Advantages of Memory-mapped I/O
  - An I/O device driver can be written in C, instead of assembly
  - Protection is easy (by address translation); user programs are prevented from issuing I/O operations directly



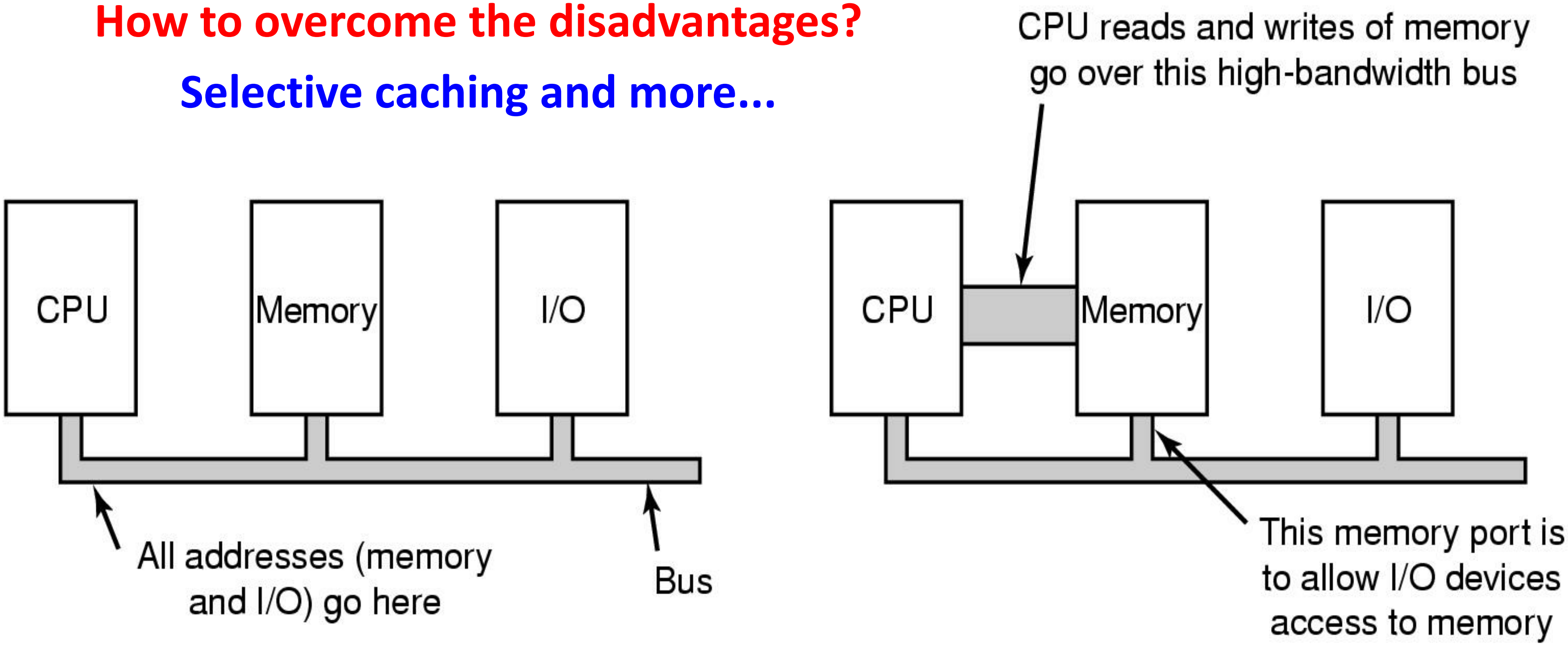
(a) Separate I/O and memory space; (b) Memory-mapped I/O; (c) Hybrid
(PDP-11) (Pentium)

CPU→I/O: Memory-mapped I/O (3)

- Disadvantages of Memory-mapped I/O
  - Caching a device control register would be disastrous
  - Both memory and all I/O devices must examine all memory references to see which ones to respond to, not easy if bus hierarchy

How to overcome the disadvantages?

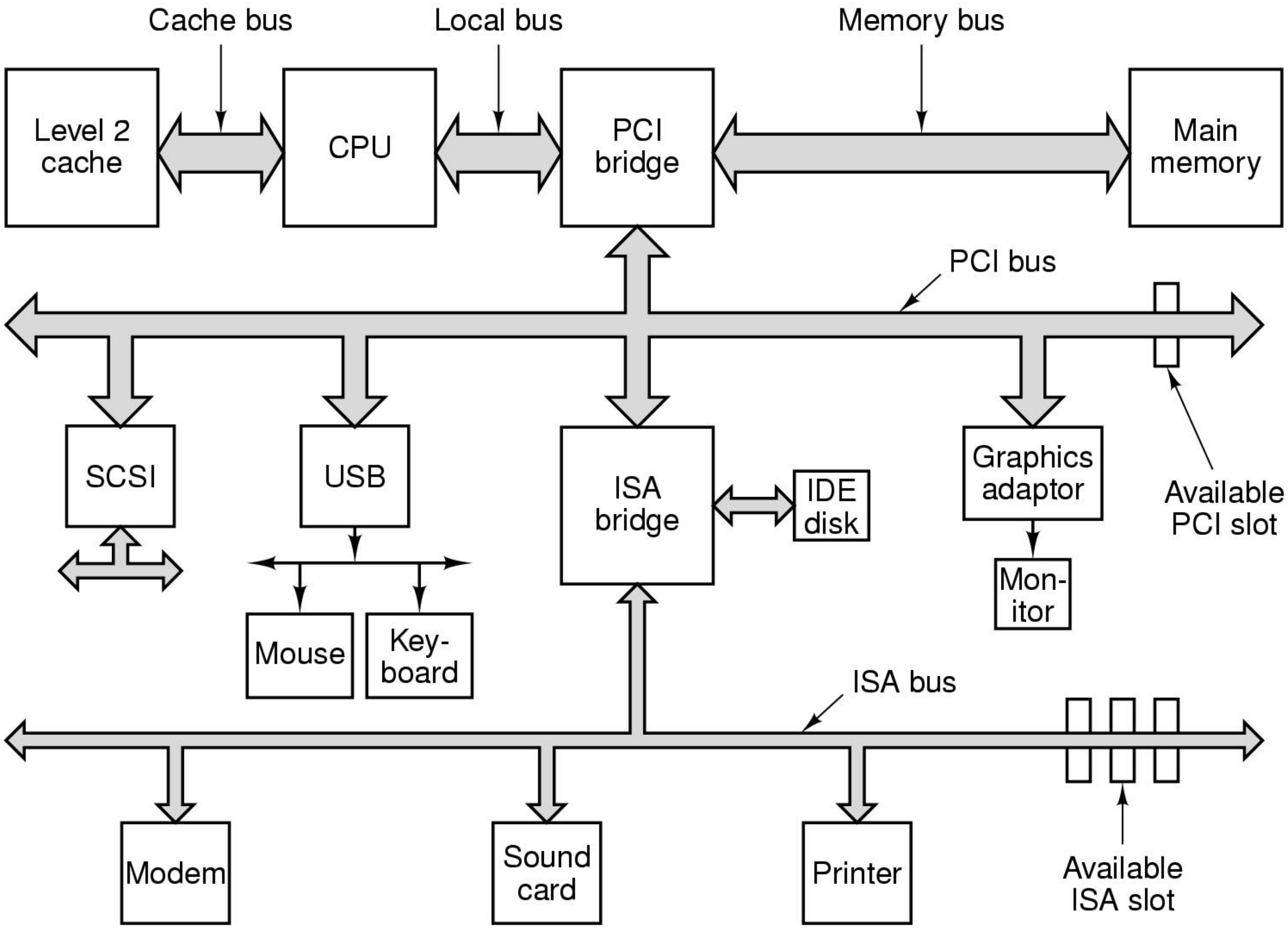
Selective caching and more...


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

CPU→I/O: Memory-mapped I/O (4)

- PCI bridge chip filters addresses to different buses



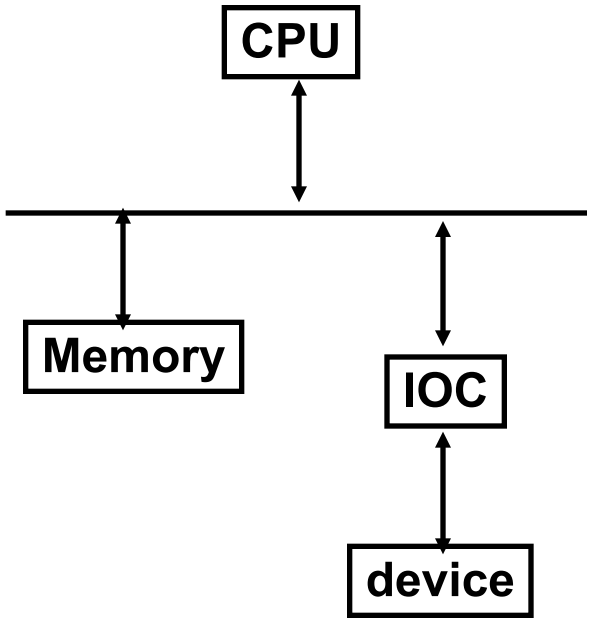
I/O→ CPU: Device Notifying the CPU

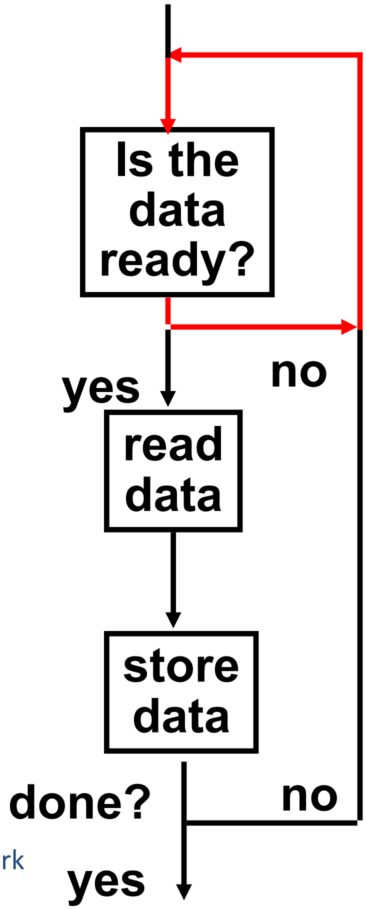
- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- This can be accomplished in two different ways:
  - Polling (Busy Waiting):
    - The I/O device put information in a status register
    - The CPU periodically or continuously check the status register
  - I/O Interrupt:
    - Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing.

In real-time systems, a hybrid approach is often used

- Use a clock to periodically interrupt the CPU, at which time the CPU polls all I/O devices

Data Transfer: Programmed I/O (Busy Waiting)

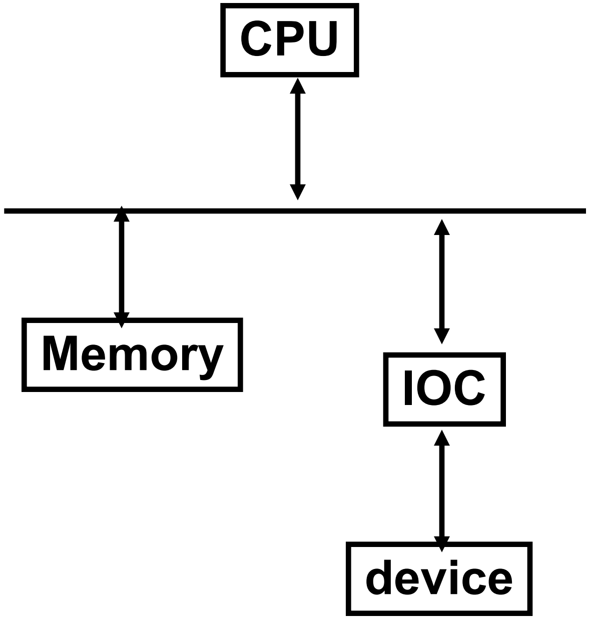


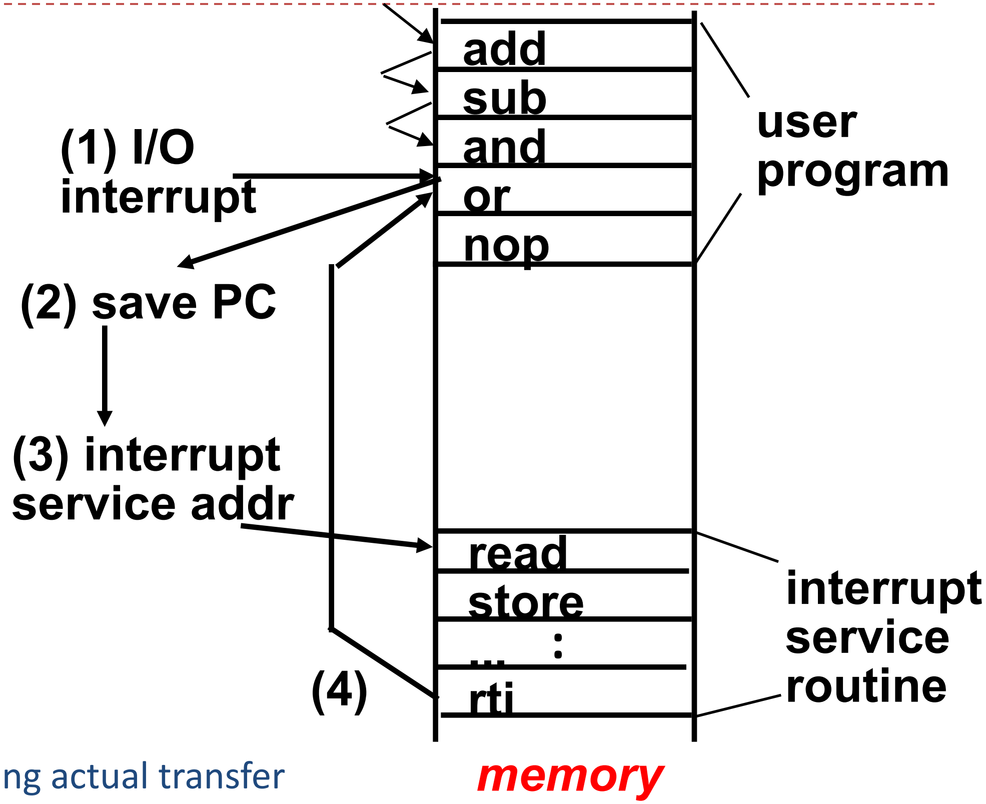


busy wait loop
not an efficient
way to use the CPU
unless the device
is very fast!

- Advantage:
  - Simple: the processor is totally in control and does all the work
- Disadvantage:
  - Polling overhead can consume a lot of CPU time

Data Transfer: Interrupt Driven

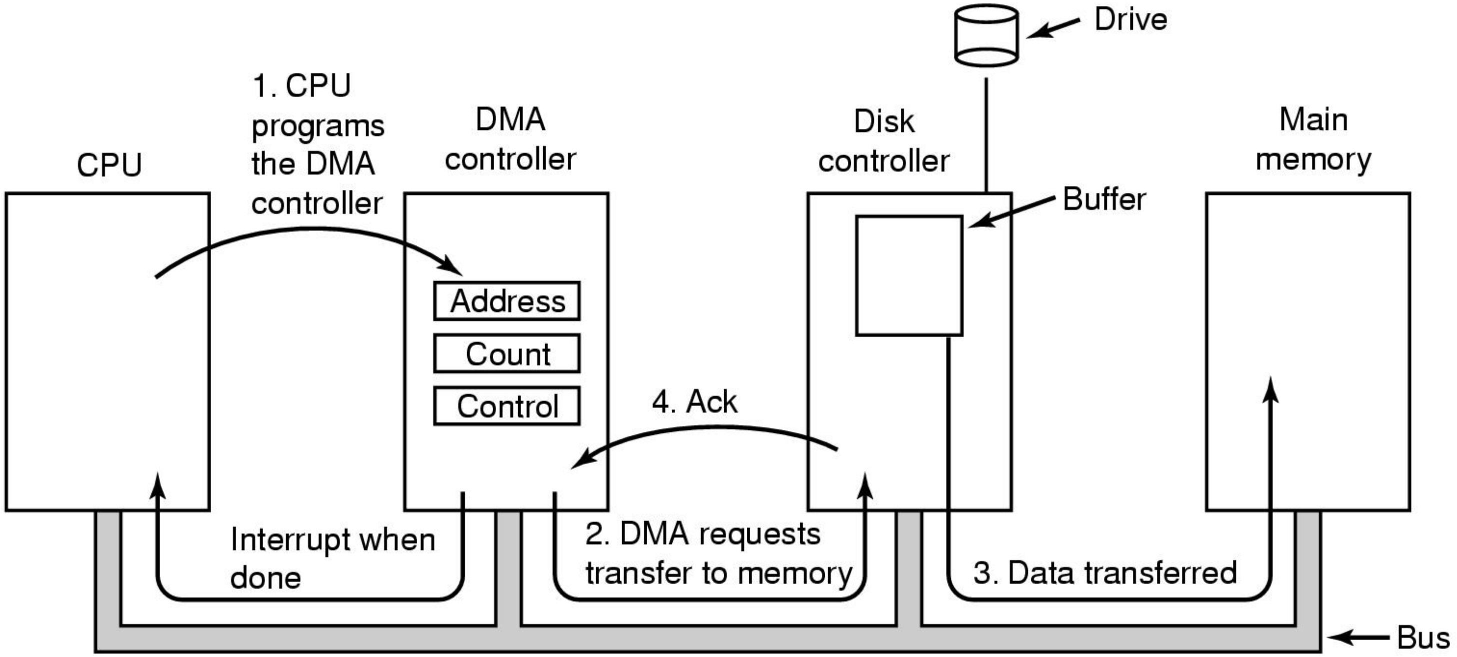




- Advantage:
  - User program progress is only halted during actual transfer
- Disadvantage, special hardware is needed to:
  - Cause an interrupt (I/O device)
  - Detect an interrupt (processor)
  - Save the proper states to resume after the interrupt (processor)

Direct Memory Access (DMA)

- DMA controller has access to system bus independent of CPU
  - A memory address register
  - A byte count register
  - A control register (direction, transfer unit, and transfer mode)
  - Multiple reg. sets if multiple transfer at once

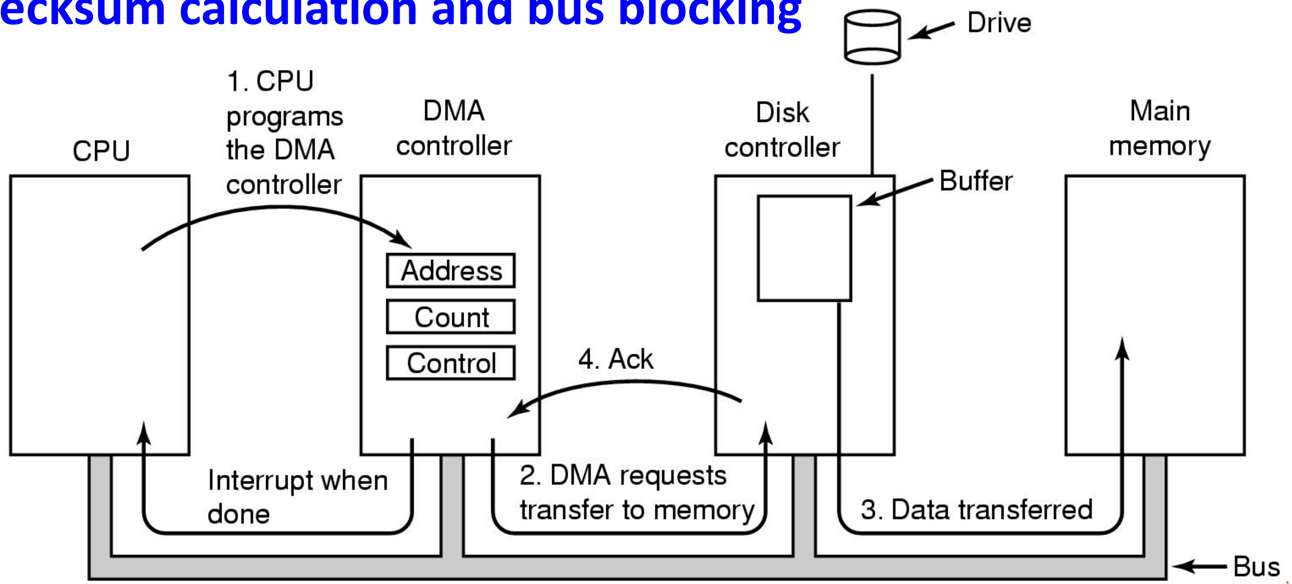


Operation of a DMA transfer

DMA Transfer Modes

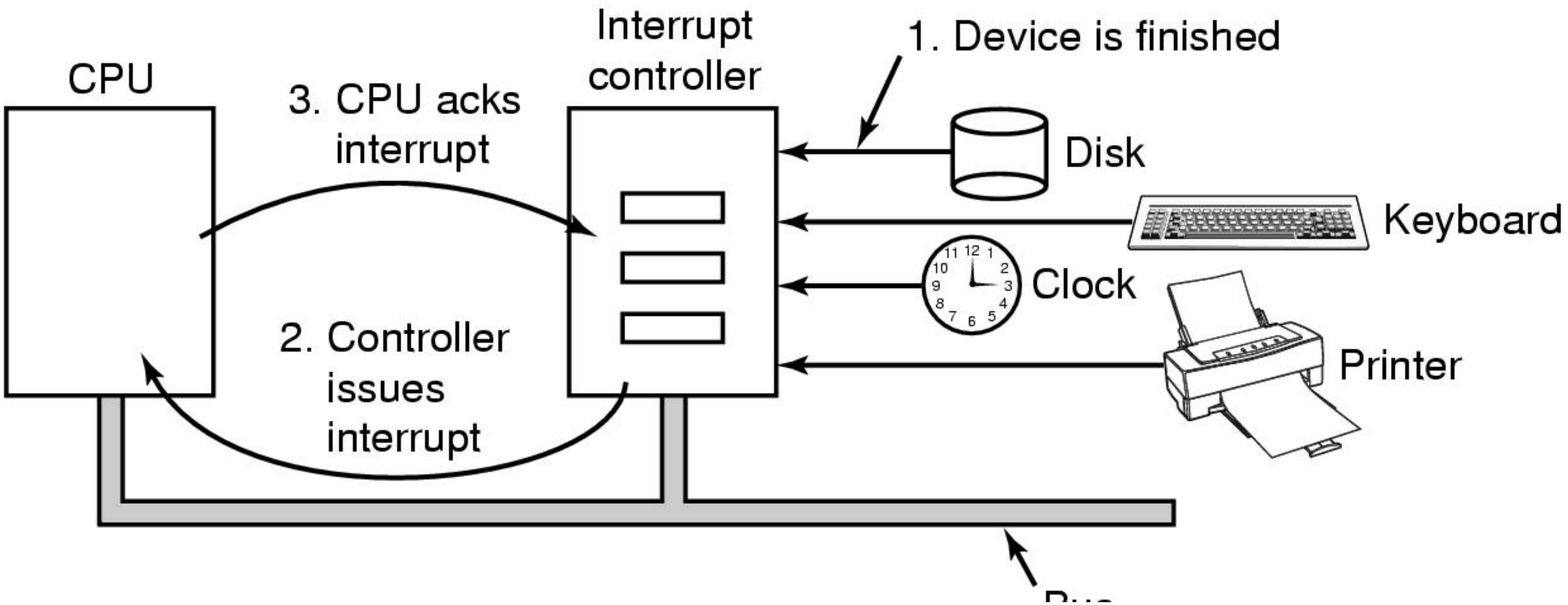
- Cycle stealing mode: word-at-a-time, sneaks in and steals an occasional bus cycle from the CPU once in a while
- Burst mode: asks I/O device to acquire the bus, issue a series of transfers, then release the bus
- Tradeoff: I/O efficiency and CPU blocked
  Why need an internal buffer at the disk controller?

  Checksum calculation and bus blocking



Interrupts Revisited

- One way of how I/O notifies CPU
  - Interrupt vector and interrupt service procedures
  - Priority if multiple simultaneous interrupts
  - CPU delays ACK until it is ready to handle the next interrupt is a way to avoid *race conditions* involving multiple almost simultaneous interrupts



Goals of I/O Software

- Device independence
  - Write programs that can access any I/O device without specifying device in advance
    - (floppy, hard drive, or CD-ROM)
- Uniform naming
  - name of a file or device a string or an integer (e.g., path name)
  - not depending on which machine / device
  - UNIX Mount System
- Error handling
  - handle as close to the hardware as possible, i.e., controller first then device driver next

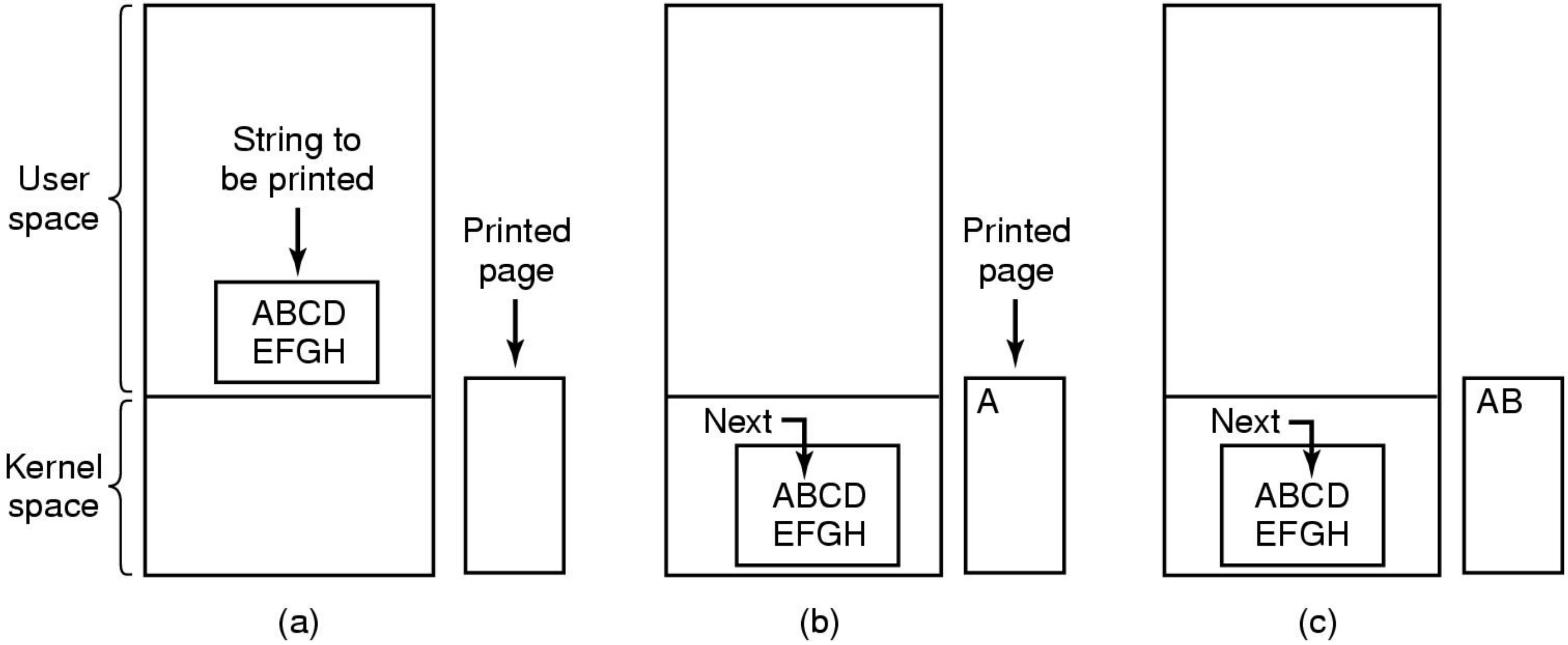
Goals of I/O Software (2)

- Synchronous vs. asynchronous transfers
  - blocked transfers vs. interrupt-driven
  - Logically synchronous to programmers and physically asynchronous
- Buffering
  - data coming off a device cannot be stored *directly* in final destination, e.g., network-I/O operations
    - For checksum and timing constraints
- Sharable vs. dedicated devices
  - disks are sharable
  - tape drives would not be
  - dedication introduces deadlock

What are three fundamental different ways of performing I/O?

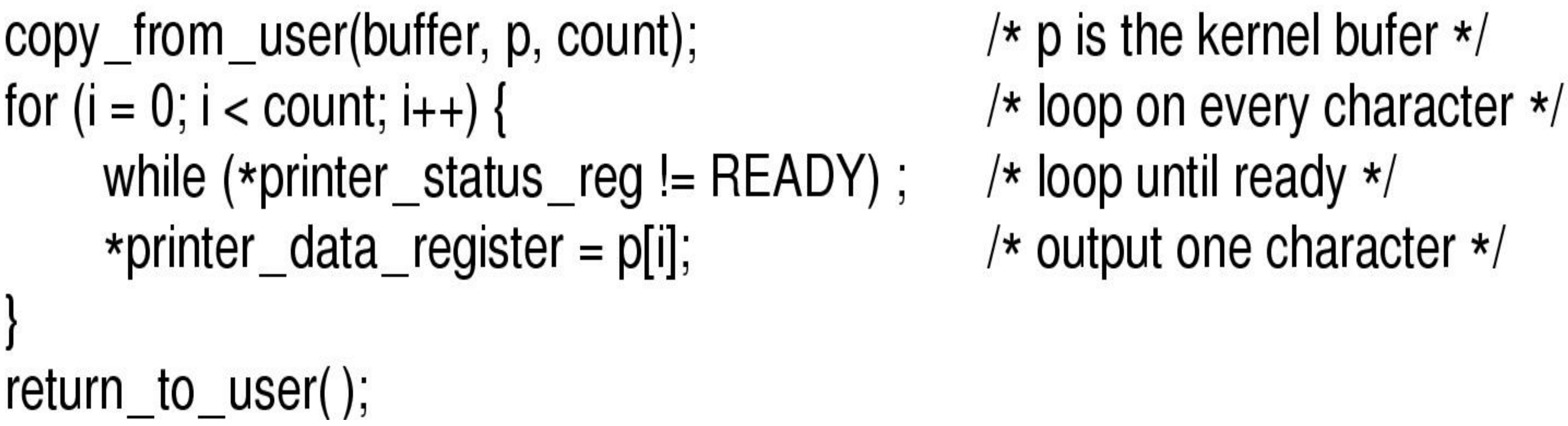
Programmed I/O

- Polling (busy waiting)
  - CPU continuously polls the device (register) to see if it is ready
  - Simple but CPU time wasteful



Steps in printing a string

Programmed I/O (2)



```
copy_from_user(buffer, p, count);                 /* p is the kernel bufer */
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

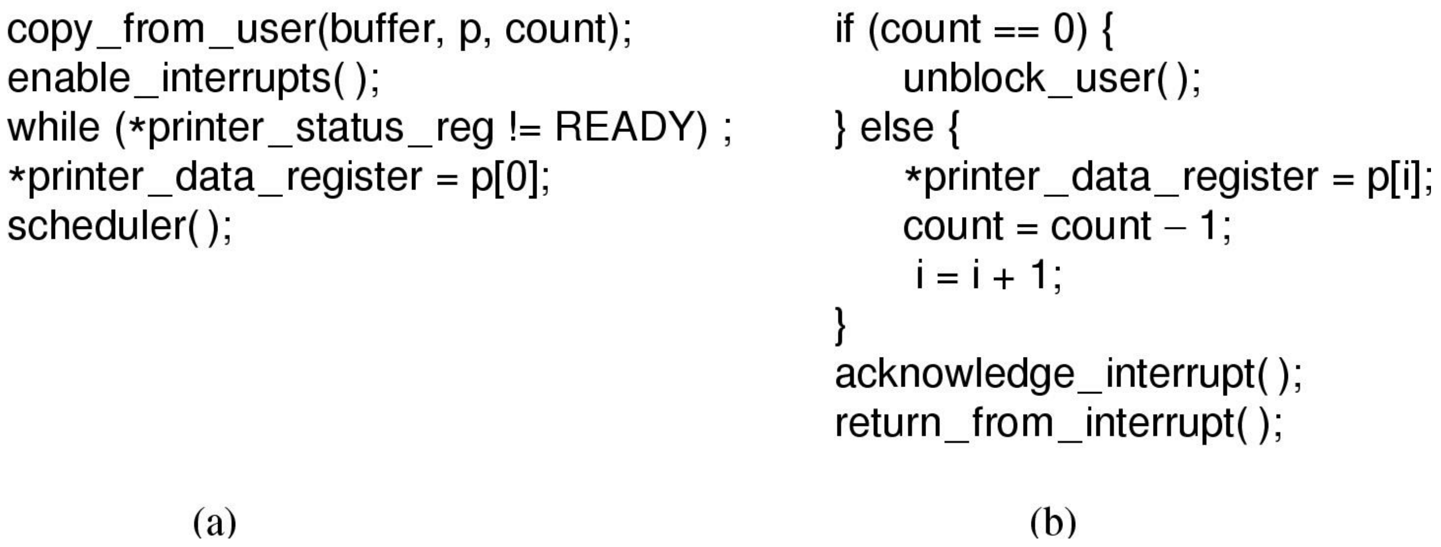
What if the printer can print 100 character/sec?

CPU would be idle for 10ms each loop iteration!
Why not switching to another process?

Interrupt-Driven I/O

- A way to allow the CPU to do something else while waiting for the printer to become ready

Still, an interrupt occurs every character, what to do?



```
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)

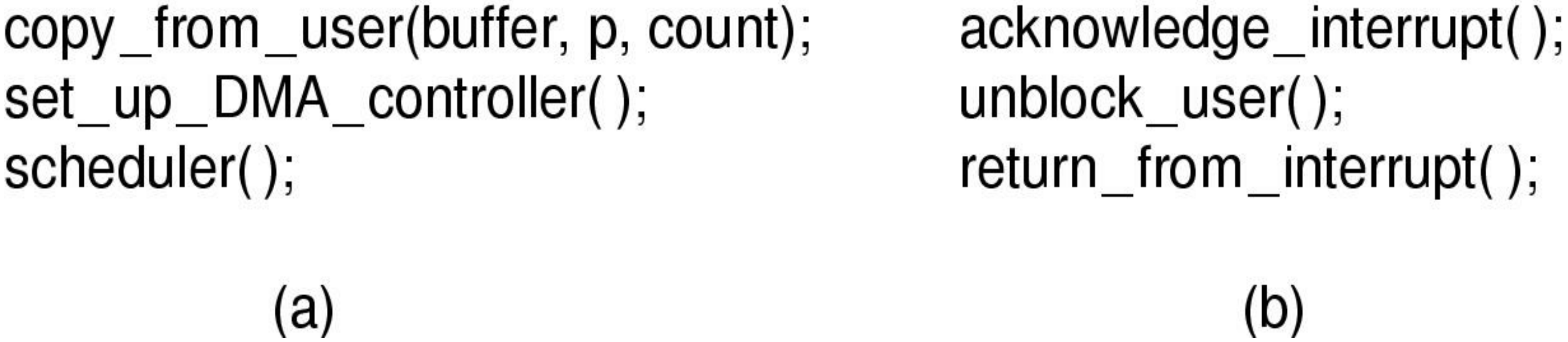
Writing a string to the printer using interrupt-driven I/O

(a) Code executed when print system call is made

(b) Interrupt service procedure

I/O Using DMA

- A way to reduce the number of interrupts from one per character to one per buffer printed
  - Transfers a block of data directly to or from memory
  - An interrupt is sent when the task is complete
  - The processor is only involved at the beginning and end of the transfer



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

(a)

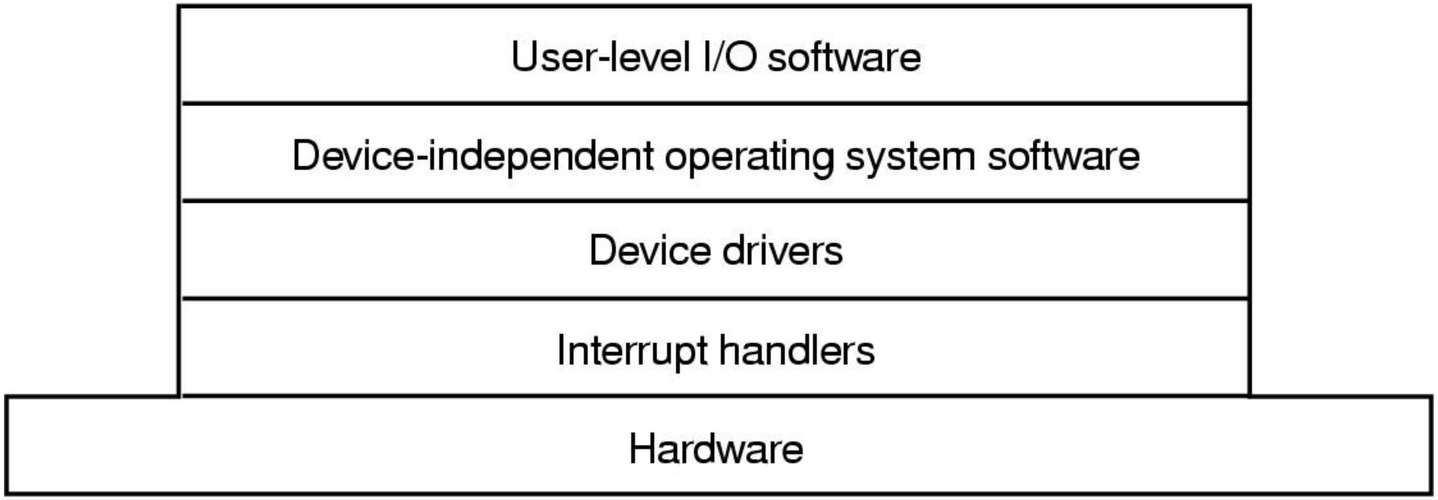
```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

- Printing a string using DMA
  - code executed when the print system call is made
  - interrupt service procedure

I/O Software Hierarchy

One objective: as little of OS as possible knows about the interrupts



User-level I/O software

Device-independent operating system software

Device drivers

Interrupt handlers

Hardware

Layers of the I/O Software System

How about the following jobs:

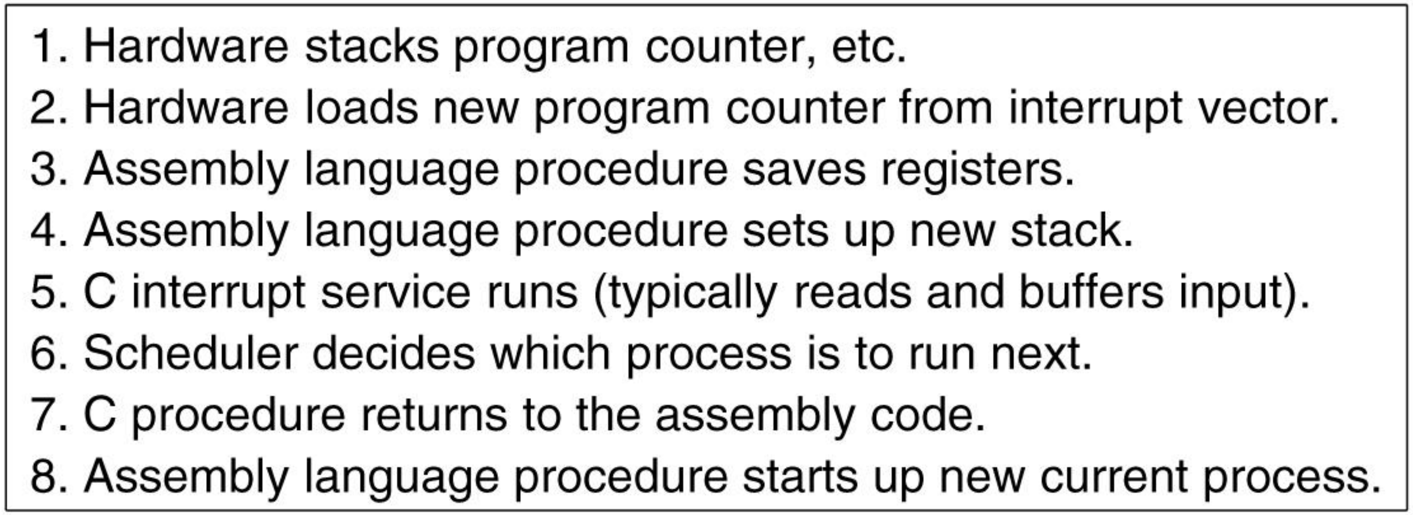
(1) Check if the user is permitted to use the device

(2) Writing commands to the device registers

(3) Formatting the output data for printing

OS Interrupt Handling

- Interrupt vector
  - contains the address of the interrupt service procedures
  - Jump table



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

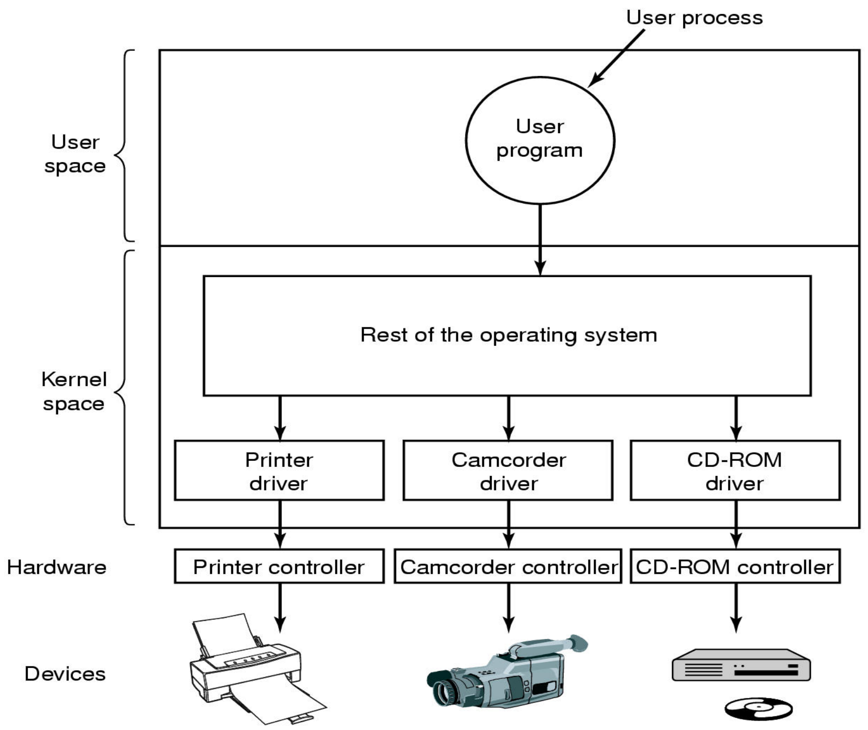
Skeleton of what lowest level of OS does when an interrupt occurs when a process is running

Interrupt Handlers

- Interrupt handlers are best hidden
  - have driver starting an I/O operation block until interrupt notifies of completion
- Interrupt procedure does its task, handling interrupts
  - then unblocks driver that started it
- But actually, many OS steps must be performed in software after *hardware interrupt* has completed
  1. Save regs not already saved by interrupt hardware
  2. Set up context for interrupt service procedure
  3. ……

Device Drivers

- Device drivers: provided by device's manufacturer, device-specific code for controlling the device (via device controller registers)



- Logical position of device drivers is shown here
- Communications between drivers and device controllers go over the bus

Device Drivers Installation

How to install device drivers to OS kernel?

- Static installation and recompiling (Unix)
  - Recompile the kernel with the new driver
- Static registration and rebooting (early Windows)
  - Make an entry in an OS file and reboot the system
- Dynamic installation (Windows & Unix)
  - On-the-fly loading into the system during the execution

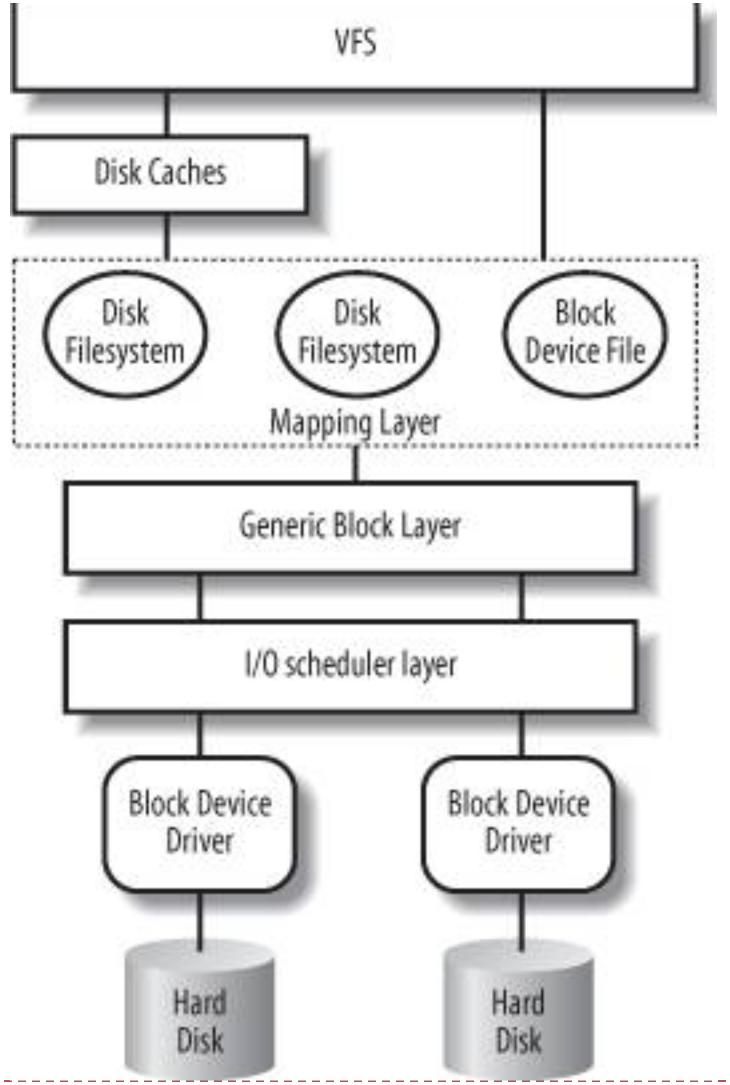
Device-Independent I/O Software

- The basic function is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicate devices
Providing a device-independent block size

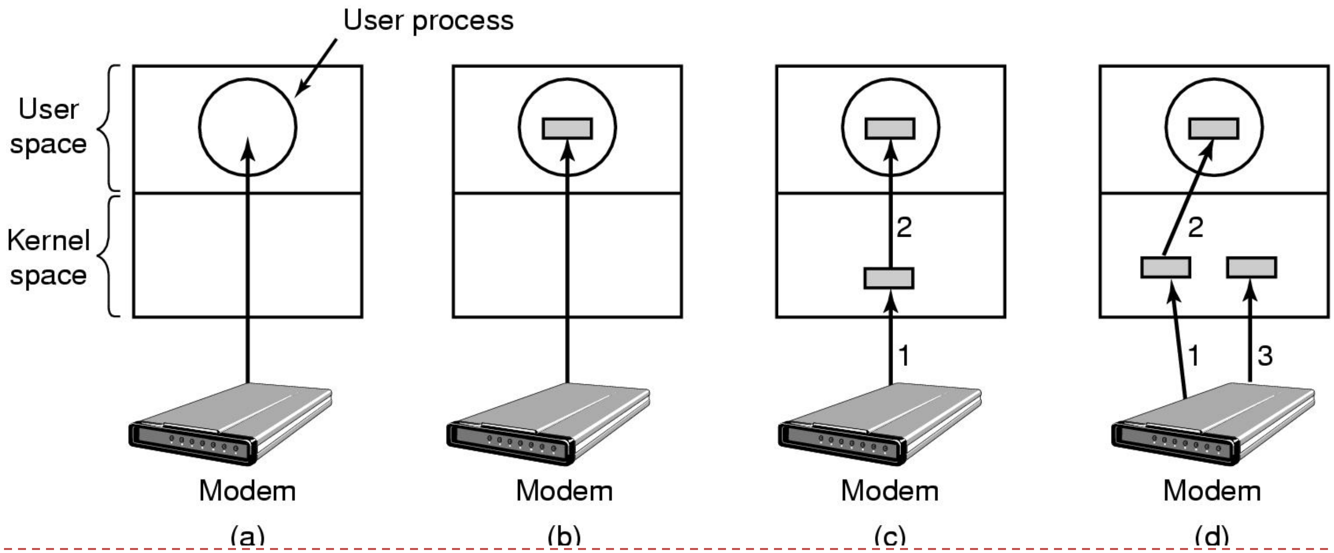
Functions of the device-independent I/O software

An Example of Device-independent I/O Software



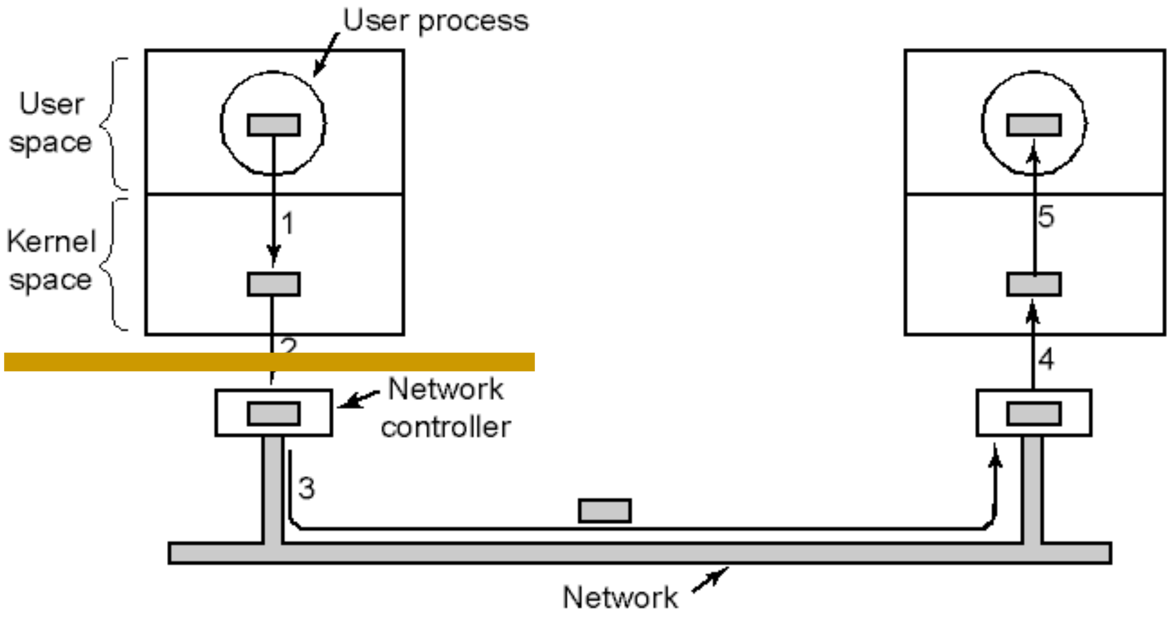
Buffering

- (a) Unbuffered input: doesn’t it have too many interrupts?
- (b) Buffering in user space: What happens if the buffer is *paged out* (to disk) when a character arrives? Go pinning?
- (c) Buffering in the kernel followed by copying to user space
  - What happens to characters that arrive while the page with the user buffer is being brought (in memory) from the disk?
- (d) *Double buffering* in the kernel; two buffers take turns.



Disadvantages of Buffering

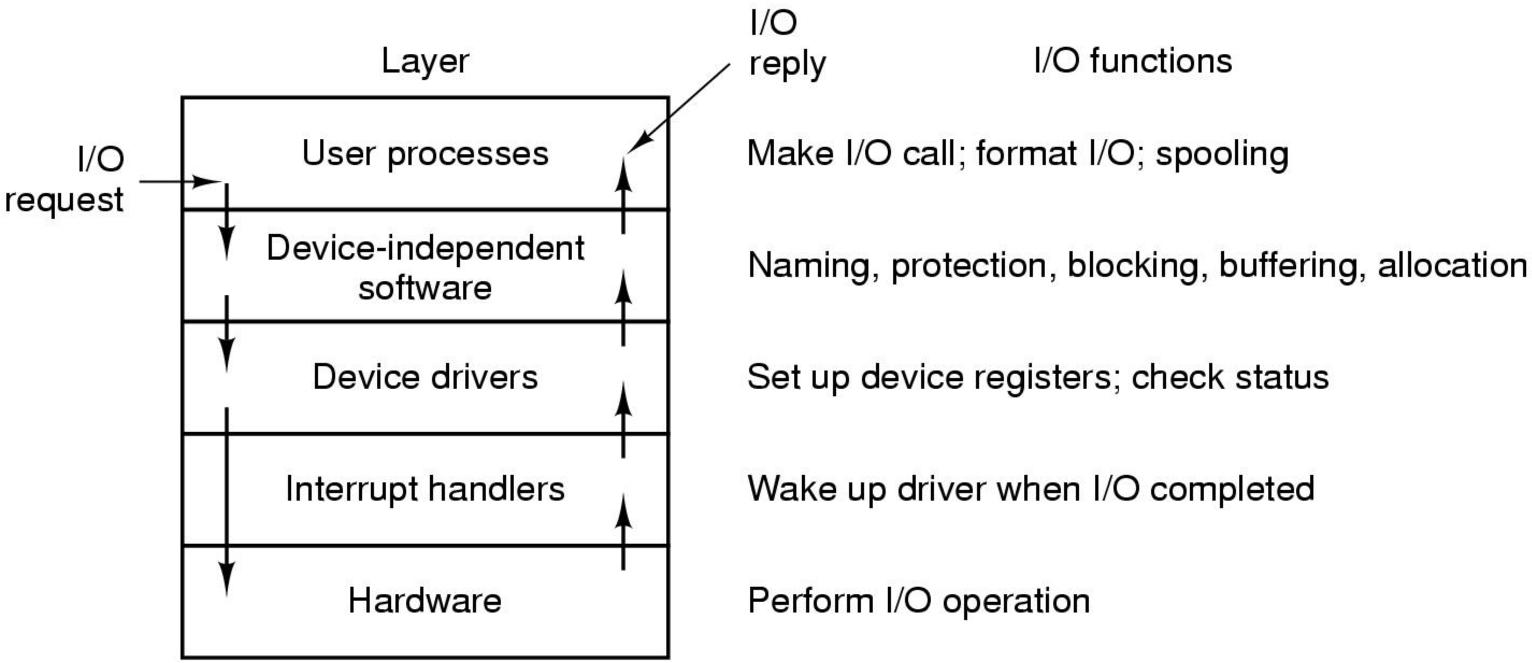
- Why the process does not copy the data to the wire directly from kernel memory?
  - But how about the performance?
    - Why not bit-by-bit from the kernel buffer to the network?



Networking may involve many copies of a packet, and all steps must happen sequentially

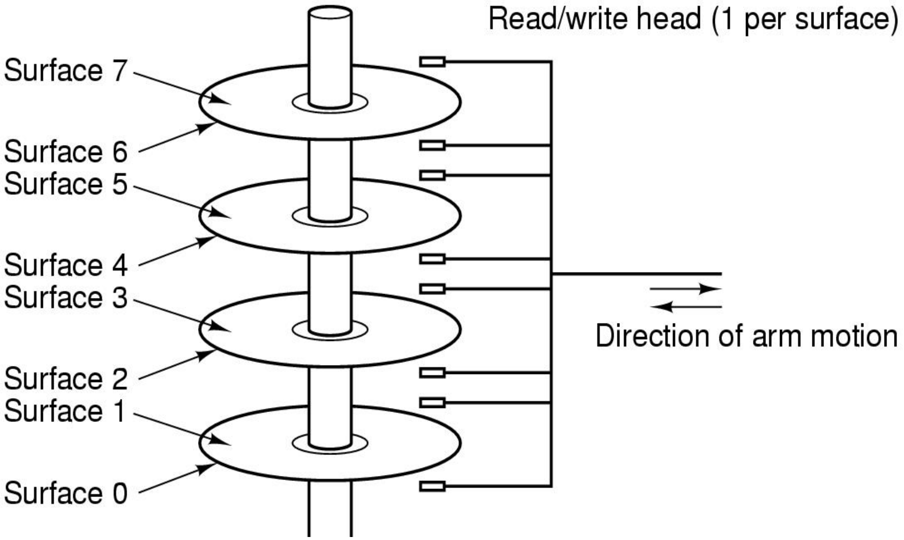
User-Space I/O Software

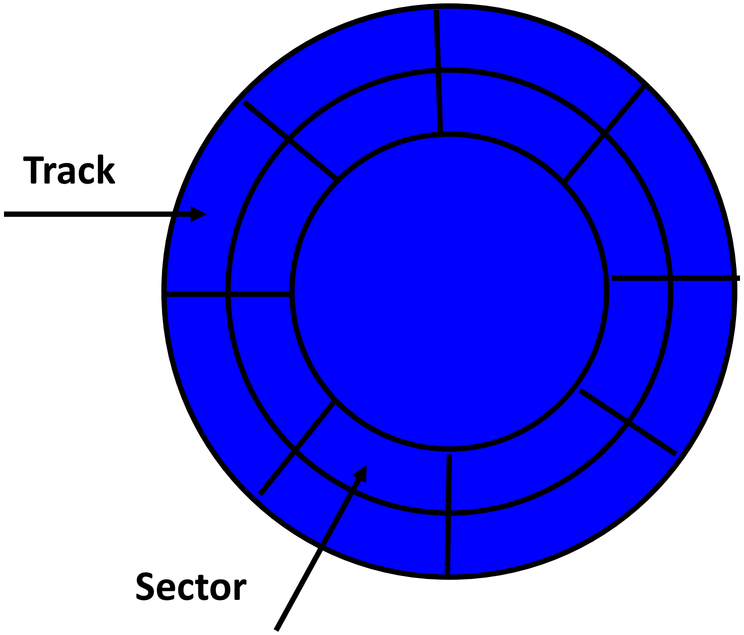
- Some of I/O software consists of user-space libraries
  - The standard I/O library contains a number of procedures that involve with I/O and all run as part of user programs
    - E.g., scanf(), printf() for I/O formatting, Intel DPDK for high-speed NICs


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

Hard Disks





- The conventional *sector-track-cylinder* model
  - A stack of platters, a surface with a magnetic coating
  - Magnetic disks are organized into cylinders, for S/R/W
- Overlapped seeks: one disk controller seeks 2+ drives simultaneously
  - Transfer between the disk and the controller can be simultaneous
  - But one transfer between the controller and the main memory

Three-Stage Disk R/W Process

- Three-stage Disk R/W process: time required to read or write a disk block determined mainly by 3 factors

  1. Seek time
  2. Rotational delay
  3. Actual transfer time

- Seek time dominates

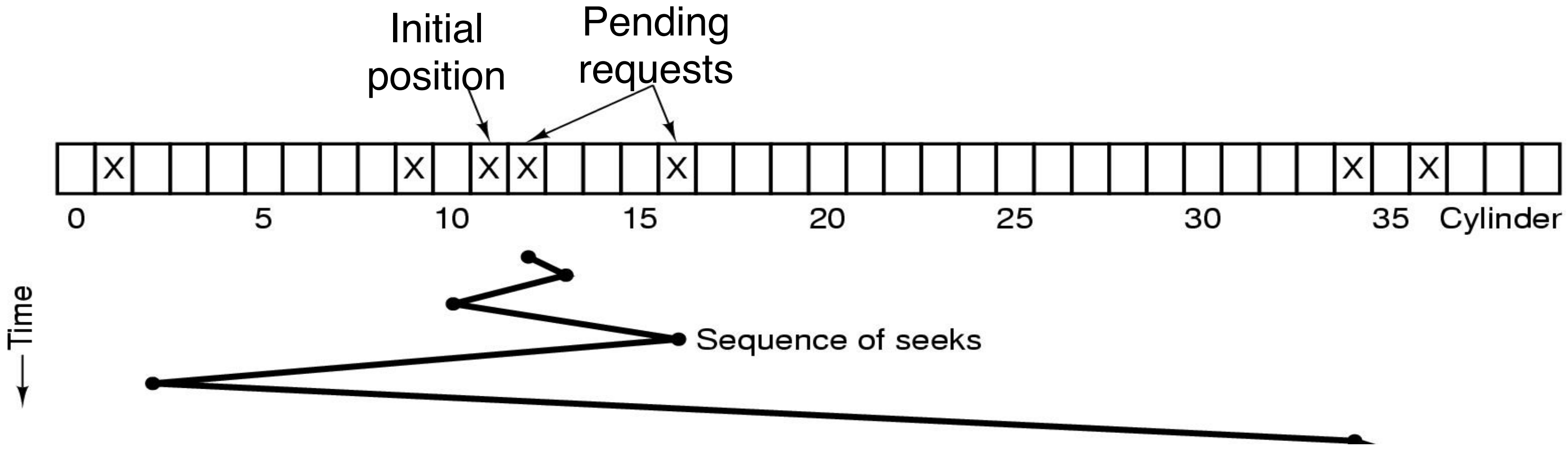
Disk Access Time = Seek time + Rotational Latency + Transfer time + Controller Time + Queuing Delay

How to optimize seek time?

Disk arm scheduling algorithms

SSF Disk Arm Scheduling Algorithm

- How to optimize seek time if a table available, indexed by cylinder number, with all the *pending disk requests* for each cylinder chained in *a linked list* headed by the table entries?
  - Example: 11 (c), 1, 36, 16, 34, 9, 12 by FCFS and SSF
  - SSF cuts average seek time of FCFS almost in half (111→61)

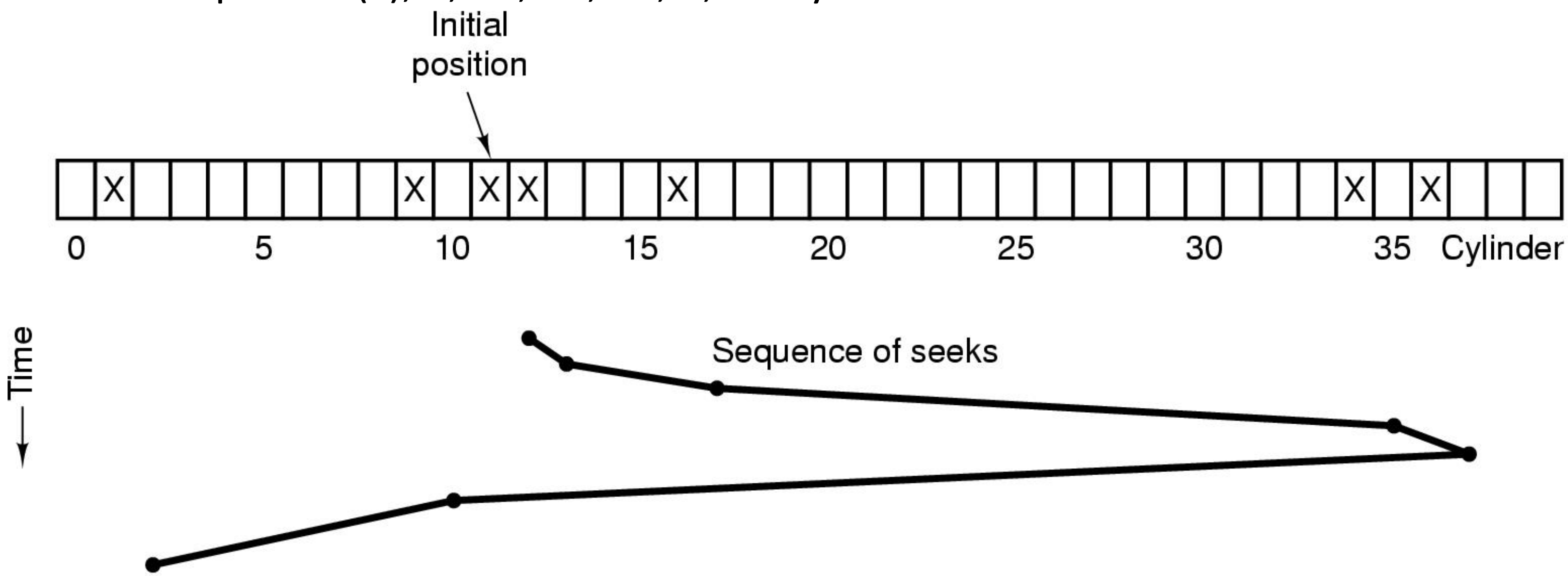


Shortest Seek First (SSF) disk scheduling algorithm

What is the key problem with SSF? How would you deal with it?

The Elevator Disk Arm Scheduling Algorithm

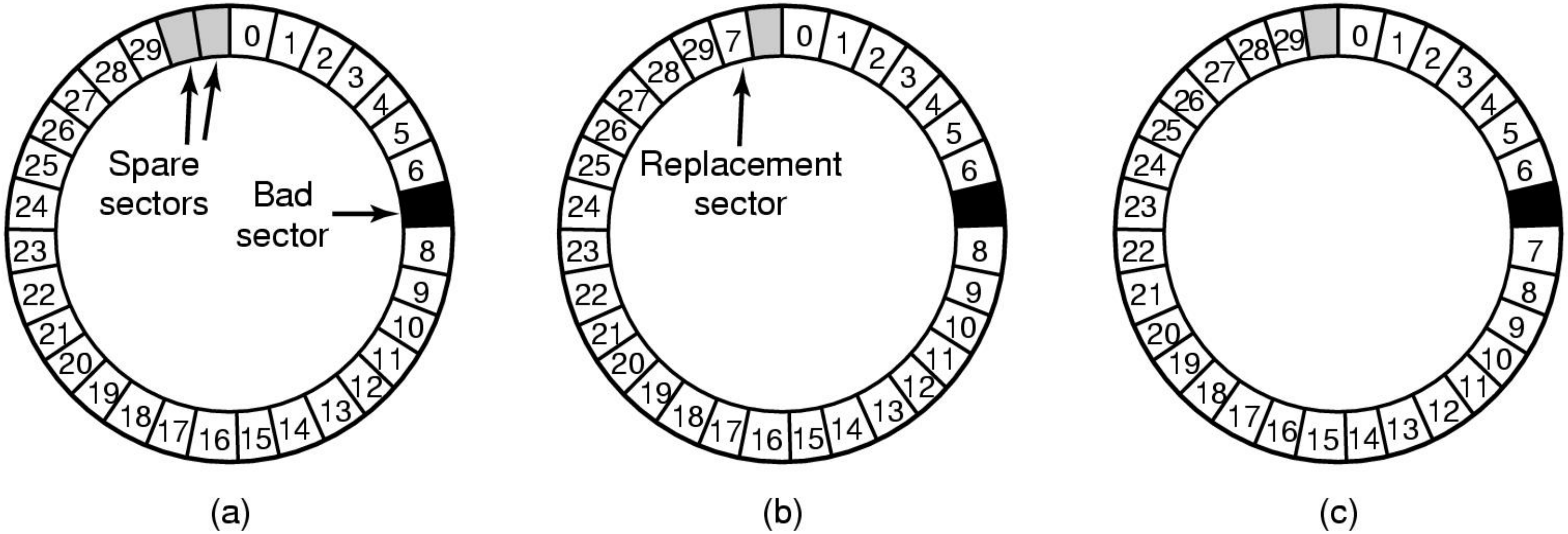
- Elevator algorithm: keep moving in the same direction until there are no more *outstanding* requests in that direction
  - One current direction bit: UP and Down
  - Property: given any collection of requests, the upper bound on the total motion is fixed: twice the number of cylinders
  - Example: 11 (c), 1, 36, 16, 34, 9, 12 by the elevator



The elevator algorithm for scheduling disk requests

Error Handling

- Two general approaches to deal with bad blocks/sectors
  - disk controller based vs. OS based

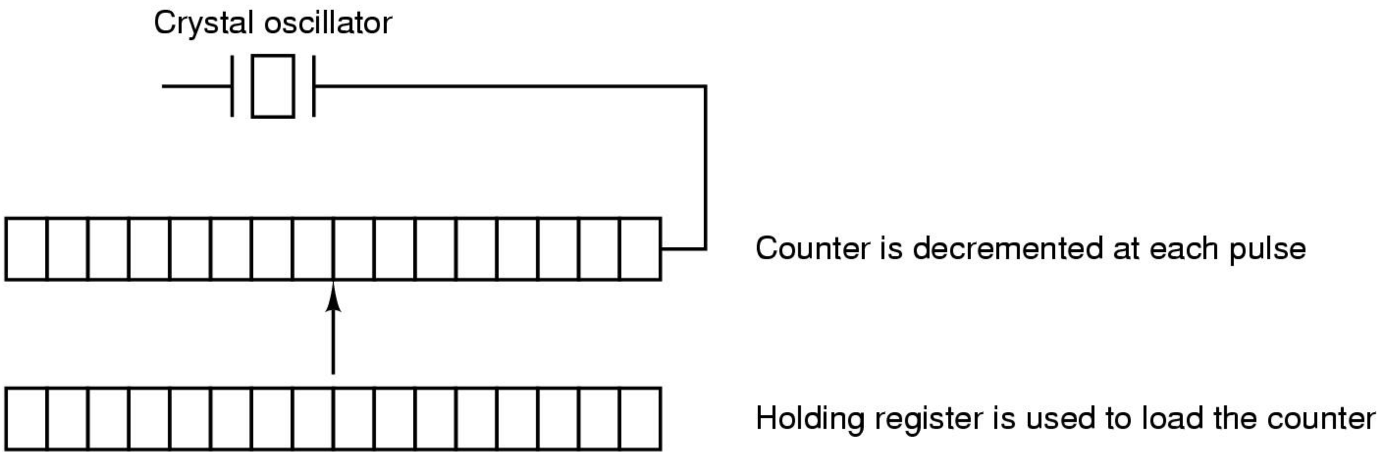


Controller does sector substitution OS does step (c)

- A disk track with a bad sector.
- Substituting a *spare* for the bad sector
- *Shifting* all the sectors to bypass the bad one

Clocks (Timers)

- Clock hardware: generate interrupts at known intervals
  - A programmable clock: its interrupt frequency can be controlled by software
  - A 500 MHz crystal with a 32-bit register can generate interrupt every 2ns to 8.6s


Crystal oscillator
Counter is decremented at each pulse
Holding register is used to load the counter

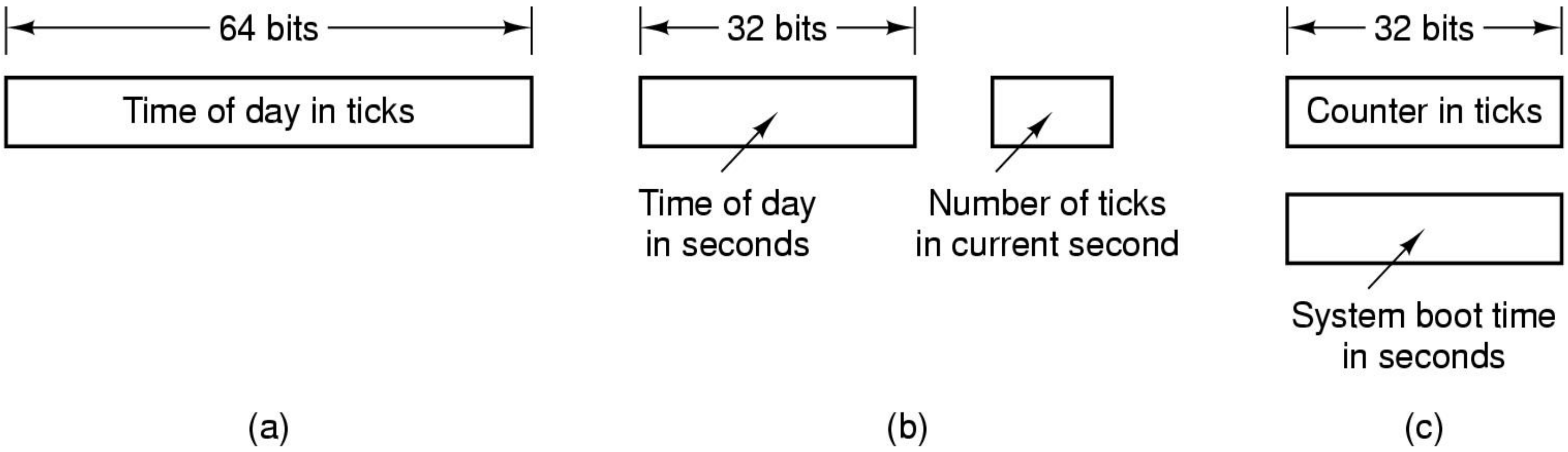
A programmable clock

Clock Software

- Clock Software (clock driver):
  - Maintaining the time of day
  - support time-shared scheduling
  - Accounting CPU usage
  - Handling the *alarm* system call
  - …

Maintaining the Time of Day

- How about incrementing a counter at each clock tick?
  - Given a clock rate of 60 Hz, a 32-bit counter last about 2 years



Three ways to maintain the time of day

(a) Using 64-bits counter

(b) Recording time in terms of seconds with a second counter in ticks

(c) Relative to system boot time, instead of a fixed external moment

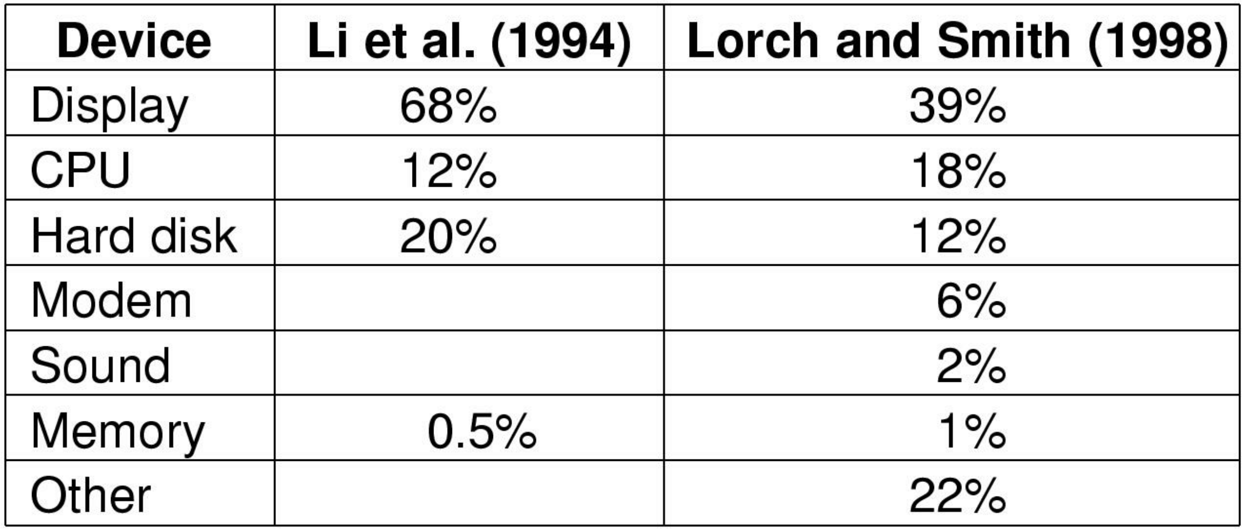
Soft Timers

- There is a need of a second hardware clock to cause timer interrupts at whatever rate a program needs
  - specified by applications
  - no problems if interrupt frequency is low
- Two ways of managing I/O
  - Polling has an average latency of half the polling interval
  - Interrupt can be costly, say 4.45μs on a 300 MHz PII
  - A Giga-bit Ethernet asks for a packet every 12 μs
- Soft timers avoid costly interrupts
  - kernel checks for soft timer expiration before it exits to user mode
  - how well this works depends on rate of kernel entries!

The combination of soft timers and a low-frequency hardware timer may be better than pure interrupt-driven I/O or pure polling

Power Management

- Power consumption is important to desktops and battery-powered computer
- Two General approaches to battery conservation
  - Turn off (down) some parts, top 3 power eaters: display, hard disk, and CPU
  - Application uses less power, multiple states: on/off, sleep, hibernating

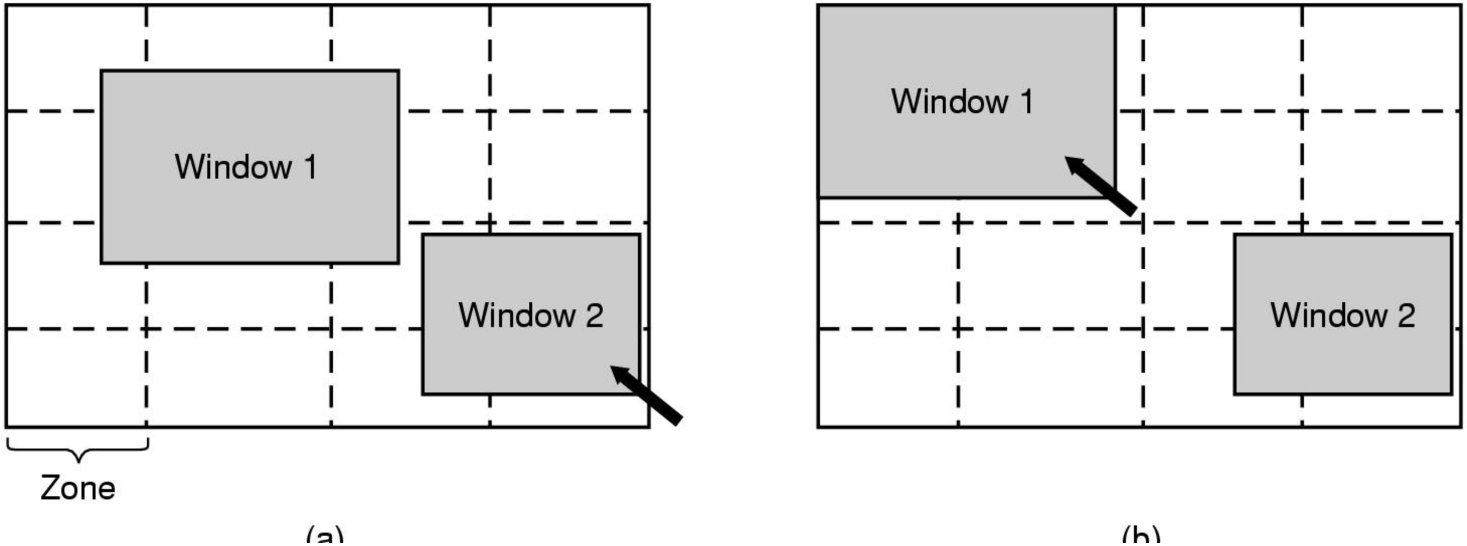


Device	Li et al. (1994)	Lorch and Smith (1998)
Display	68%	39%
CPU	12%	18%
Hard disk	20%	12%
Modem		6%
Sound		2%
Memory	0.5%	1%
Other		22%

Power consumption of various parts of a laptop computer

Power-aware computing in mobile and sensor networks, many tradeoffs!

OS Issues: The Display



- On-off switching
  - Display consist of some zones that can be independently powered up or down

OS Issues: The Hard Disk

- It takes substantial energy to keep it spinning at high speed, even if there are no disk accesses

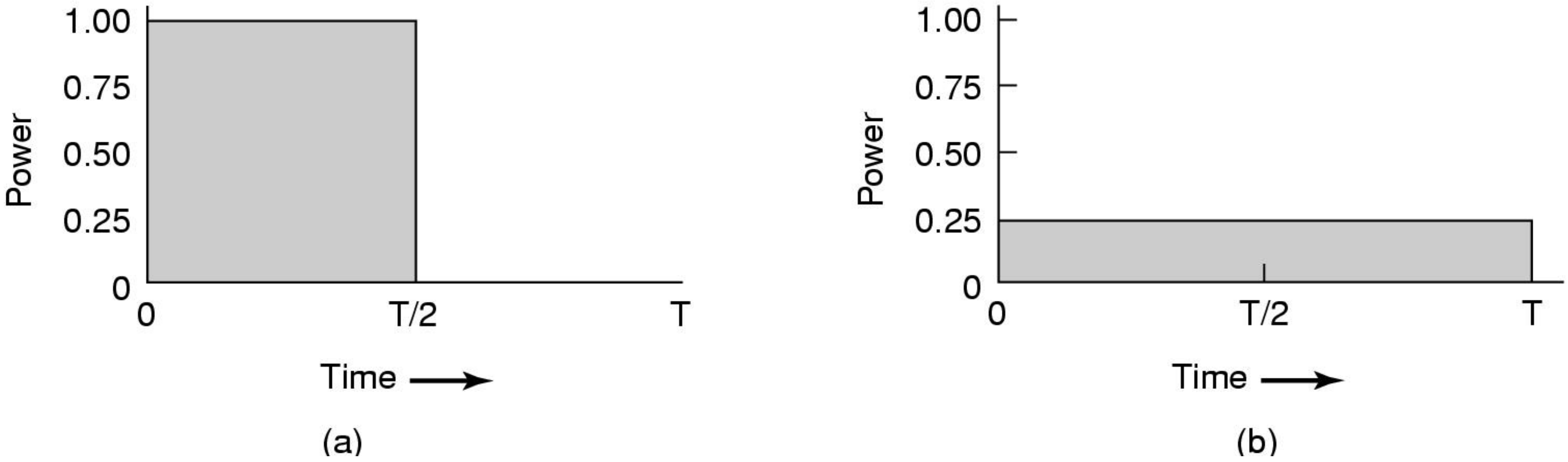
- But Restarting a hard disk consumes considerable energy
  - What is the break-even point?
  - *Td* [5s, 15s]
  - If the next disk access is expected some time *t* in the future, compare *t* to *Td* determines the appropriate disk action

- Alternative: have a substantial disk cache in RAM

- Alternative: OS keeps running programs informed about the disk states, so that some accesses can be delayed (and clustered)

Something can be done to CPU too!

OS Issues: The CPU



- Running at full clock speed (full voltage)
- Cutting voltage by two
    - cuts clock speed by two
    - cuts power by four (power consumption ~ voltage^2)

Summary

- OS responsibilities in I/O operations
  - Protection and Scheduling
  - CPU communicates with I/O devices
  - I/O devices notify OS/CPU
- I/O software hierarchy
  - Interrupt handlers
  - Device drivers
  - Buffering
- Storage Systems
  - Disk head scheduling algorithms
- Power Management

Thank you all !