

CS 347M (Operating Systems Minor)

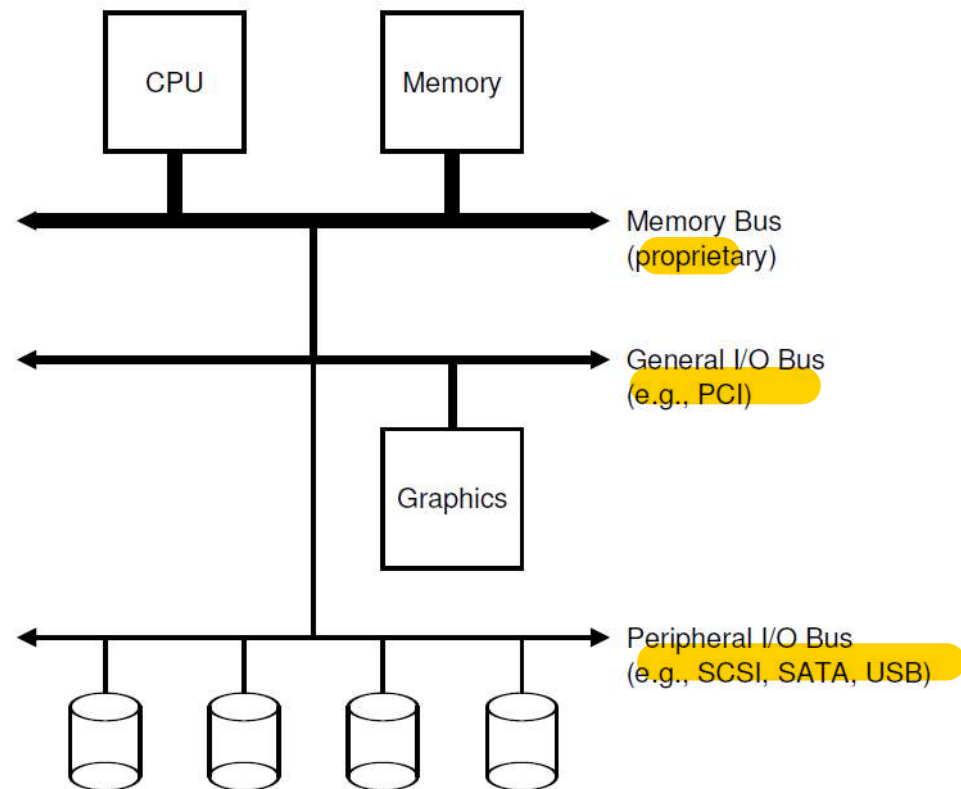
Spring 2022

# Lecture 17: I/O and Filesystems

Mythili Vutukuru  
CSE, IIT Bombay

# Input/Output Devices

- CPU and memory connected via high speed system (memory) bus
  - Bus = set of wires carrying signals
- I/O devices connect to the CPU and memory via other separate buses
  - High speed bus, e.g., PCI
  - Other: SCSI, USB, SATA
- Point of connection to the system: port



# Simple Device Model

- Block devices store a set of numbered blocks (disks)
- Character devices produce/consume stream of bytes (keyboard)
- Devices expose an interface of memory registers
  - Current status of device
  - Command to execute
  - Data to transfer
- Device controller manages device, internals of device are usually hidden

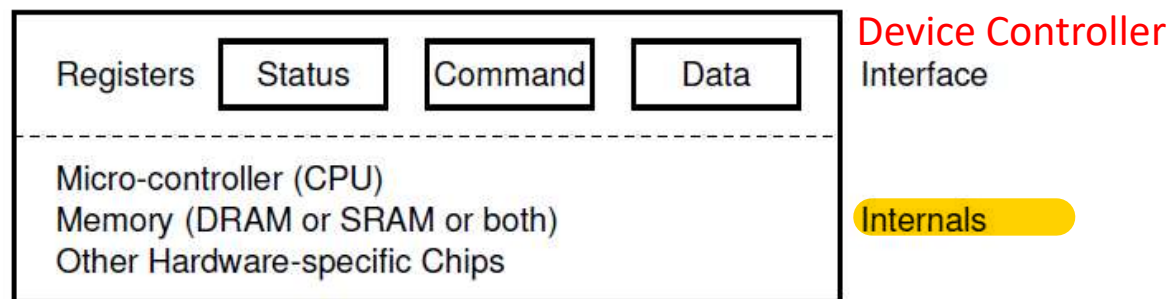



Figure 36.2: A Canonical Device

# How does OS read/write to device registers?

- OS communicates with device via device controller
- OS reads/writes registers in the I/O device: how?
- **Explicit I/O instructions**
  - E.g., on x86, `in` and `out` instructions can be used to read and write to specific registers on a device
  - Privileged instructions accessed by OS
- **Memory mapped I/O**
  - Device makes registers appear like memory locations
  - OS simply reads and writes from memory
  - Memory hardware routes accesses to these special memory addresses to devices

# A simple execution of I/O requests

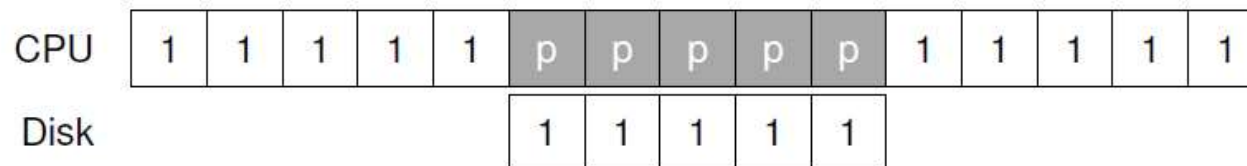


```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

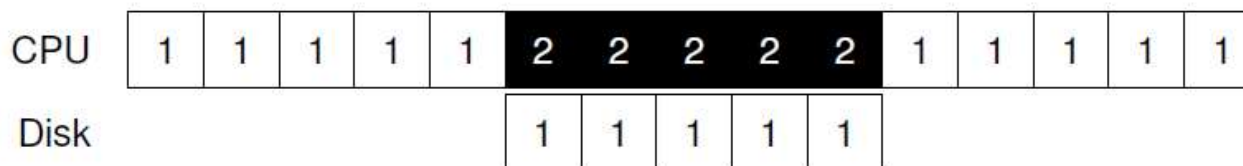
- Simple model of I/O, e.g., read/write block of data from disk
  - Give command to device via command register
  - Write: transfer data to device via data register
  - Poll status register to see if I/O operation completes
  - Read: Copy data from data register to main memory after I/O completes
- Polling status to see if device ready constantly – wastes CPU cycles

# Interrupts

- Polling wastes CPU cycles



- Instead, OS can put process to sleep and switch to another process



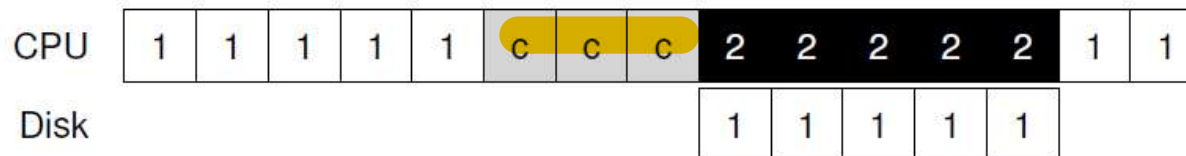
- When I/O request completes, device raises interrupt, OS can switch back to original process after that request has completed
  - Note: context switch to original process need not be immediate

# Interrupt handler

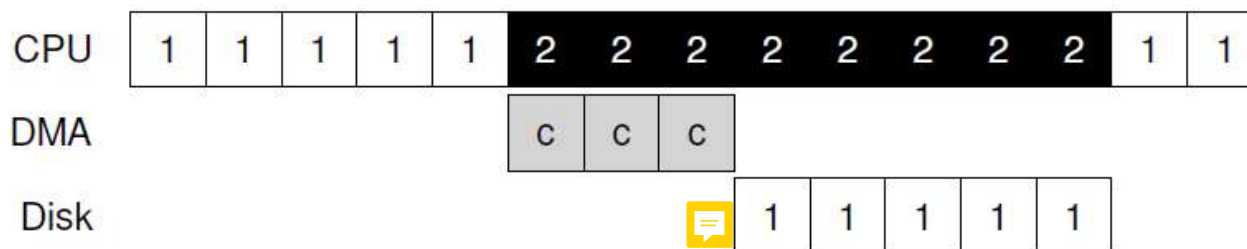
- Interrupt from I/O device causes trap, switches process to kernel mode
- Interrupt Descriptor Table (IDT) stores pointers (value of PC) to OS interrupt handlers (interrupt service routines)
  - Interrupt (IRQ) number identifies the interrupt handler to run for a device
- Interrupt handler processes notification from device, unblocks the process waiting for I/O (if any), and starts next I/O request (if any pending)
- Handling interrupts imposes kernel mode transition overheads
  - Note: polling may be faster than interrupts if device is fast

# Direct Memory Access (DMA)

- In spite of interrupt, CPU cycles wasted in copying data to/from device



- Instead, a special piece of hardware (DMA engine) copies from main memory to device and vice versa, without involving CPU
  - CPU gives DMA engine the memory location of data
  - In case of disk read, device copies data via DMA to RAM and then raises interrupt
  - In case of write, device copies data via DMA from RAM and then starts writing





# Summary of disk read with interrupt + DMA

- Process P1 makes read system call to read data from disk
- OS gives command to disk via command register
- OS switches to another process P2 (P1 cannot run anymore, blocked)
- Disk completes reading data, copies data into main memory directly via DMA, then raises interrupt
- OS handles interrupts, disk data is ready, marks P1 as ready to run
  - Interrupt handled in the kernel mode of P2
- OS scheduler switches back to P1 at a later time

# File abstraction

```
fd = open("/home/foo/a.txt")  
read(fd, ..)  
write(fd, ..)  
close(fd)
```

- **File**: sequence of bytes, stored persistently on disk
- **Directory**: container for files and other sub-directories
- Steps to access a file
  - Open a file using system call, get a file descriptor
  - File descriptor is a handle to refer to file for read/write
  - Close file when done accessing it
- **Filesystem**: OS subsystem that stores files and directories persistently on secondary storage like hard disks
  - File-related system calls are exposed to users to access files

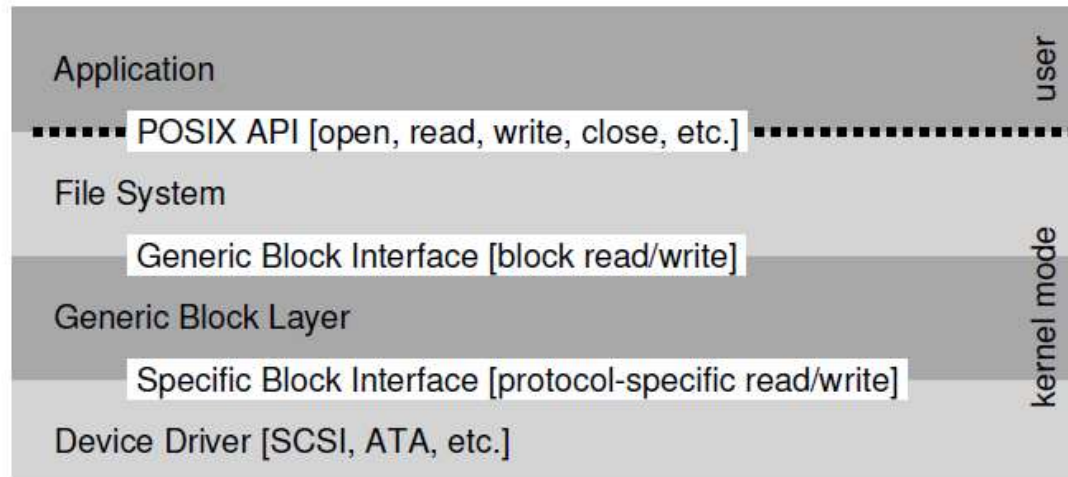
# Reading and writing a file

```
fd = open("/home/foo/a.txt")
char buf[64]
n = read(fd, buf, 64)
buf[0] = ...
n = write(fd, buf, 64)
```

- After opening a file, process can read/write to a file as a stream
  - File descriptor used as handle to refer to the open file stream
  - Read system call reads specified number of bytes into a user-defined buffer, returns number of bytes read
  - Write system call writes specified number of bytes from a user-defined buffer, returns number of bytes written
- Read and write system calls update the offset into a file
  - After reading N bytes, next read will return the next set of bytes
  - Can also update the offset from which to read/write using a seek system call
  - Every open file descriptor will read/write file as independent stream, with independent offsets

# Filesystem and device driver

- Device driver: part of OS code that talks to specific device, gives commands, handles interrupts etc.
  - Most OS code abstracts out the device-specific details
- File system code is built as layers: system calls, block read/write, and device driver that actually communicates with disk



# Directory tree

- Files and directories arranged in a tree, starting with root ("/")

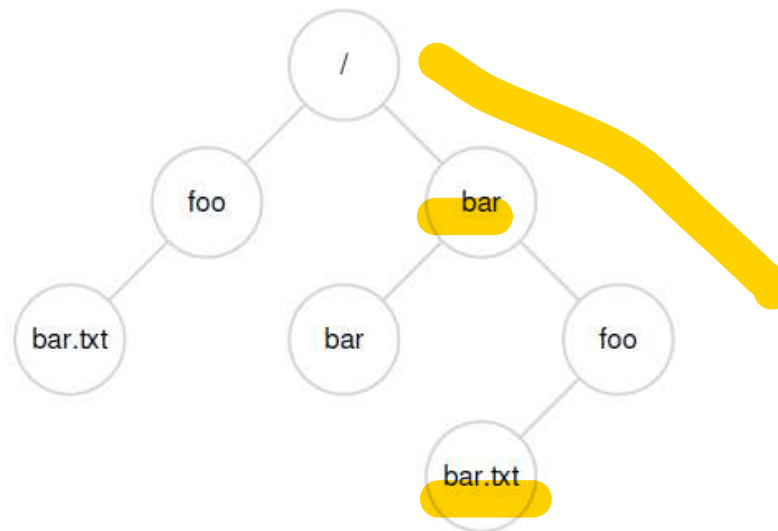


Figure 39.1: An Example Directory Tree

# Operations on directories

- Every file is identified by a unique **inode number** in filesystem
- Directory is a special file that contains mappings between filenames and inode number of the file
- Directories can also be accessed like files, e.g., create, open, read, close
- For example, the “ls” program opens and reads all directory entries
  - Directory entry contains file name, inode number, type of file (file/directory) etc.

```
int main(int argc, char *argv[]) {  
    DIR *dp = opendir(".");  
    assert(dp != NULL);  
    struct dirent *d;  
    while ((d = readdir(dp)) != NULL) {  
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);  
    }  
    closedir(dp);  
    return 0;  
}
```

# Hard links

- Hard linking creates another file that points to the same inode number (and hence, same underlying data)
- If one file deleted, file data can be accessed through the other links
- Inode maintains a link count, file data deleted only when no further links to it
- You can only unlink, OS decides when to delete

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

# Soft links or symbolic links

- Soft link is a file that simply stores a pointer to another filename

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r----- 1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

- If the main file is deleted, then the link points to an invalid entry: dangling reference

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```



# Mounting a filesystem

- Mounting a filesystem connects the files to a specific point in the directory tree

```
prompt> mount -t ext3 /dev/sda1 /home/users  
prompt> ls /home/users/  
a b
```

- Several devices and file systems are mounted on a typical machine, accessed with `mount` command

```
/dev/sda1 on / type ext3 (rw)  
proc on /proc type proc (rw)  
sysfs on /sys type sysfs (rw)  
/dev/sda5 on /tmp type ext3 (rw)  
/dev/sda7 on /var/vice/cache type ext3 (rw)  
tmpfs on /dev/shm type tmpfs (rw)  
AFS on /afs type afs (rw)
```