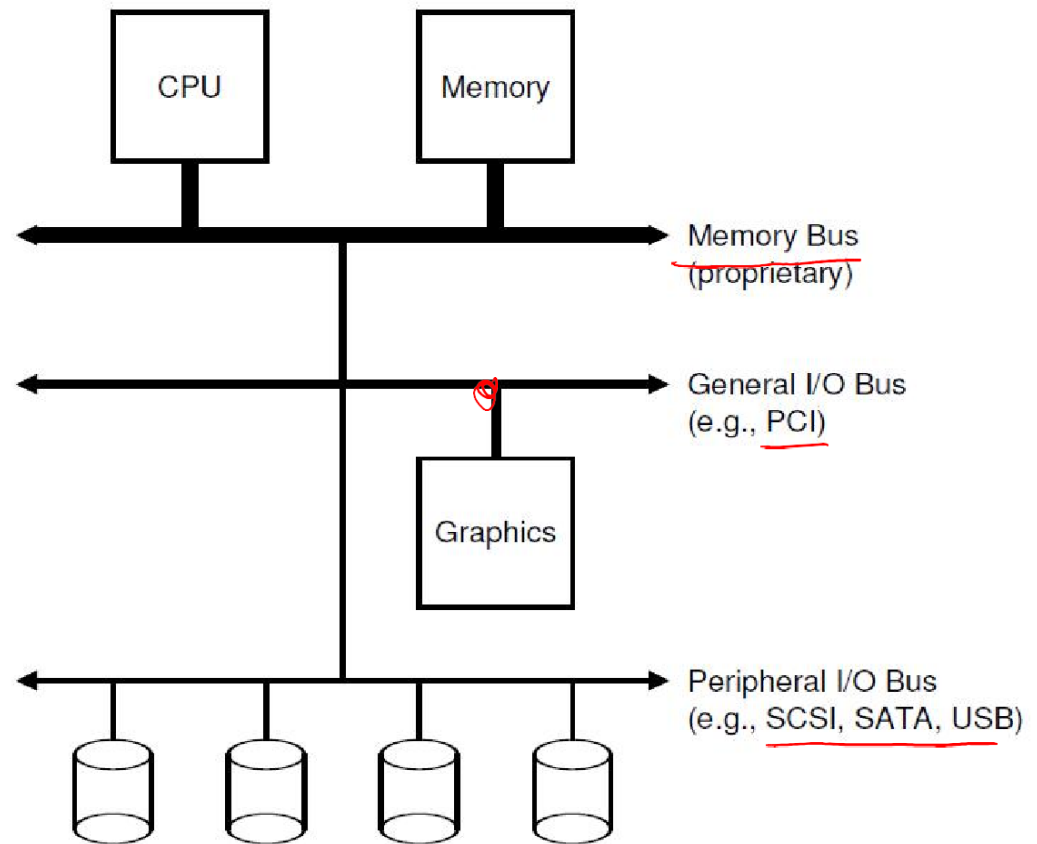


# Communication with I/O Devices && Files and Directories

# Input/Output Devices

- I/O devices connect to the CPU and memory via a bus
  - 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
- The internals of device are usually hidden

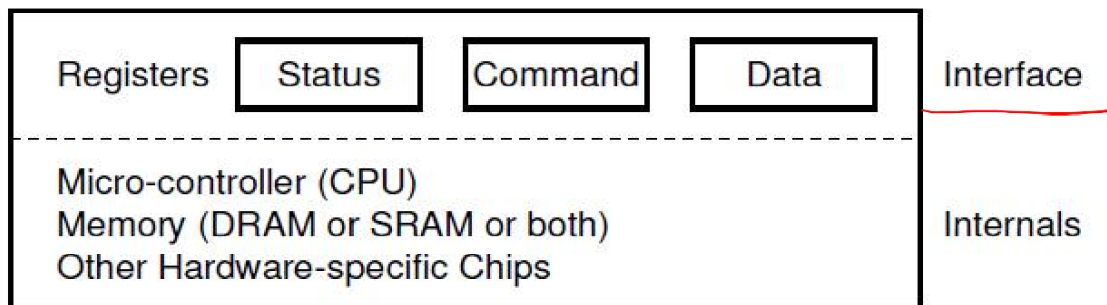


Figure 36.2: A Canonical Device

# How does OS read/write to registers?

- How does OS read/write to registers like status and command?
- 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

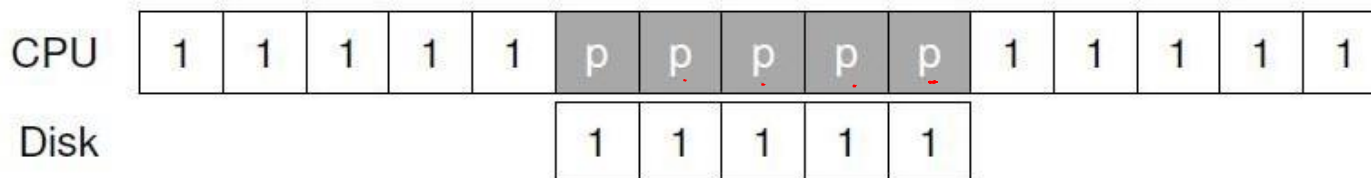
# A simple execution of I/O requests

```
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
```

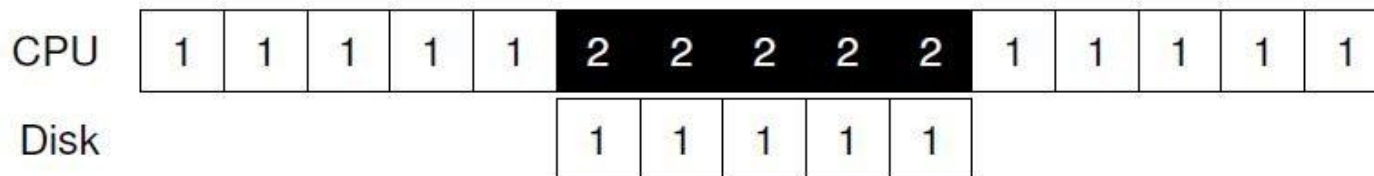
- Polling status to see if device ready – wastes CPU cycles
- Programmed I/O – CPU explicitly copies data to/from device

# Interrupts

- Polling wastes CPU cycles



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



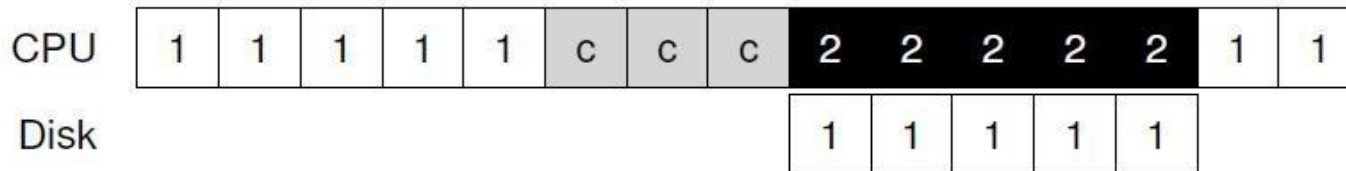
- When I/O request completes, device raises interrupt

# Interrupt handler

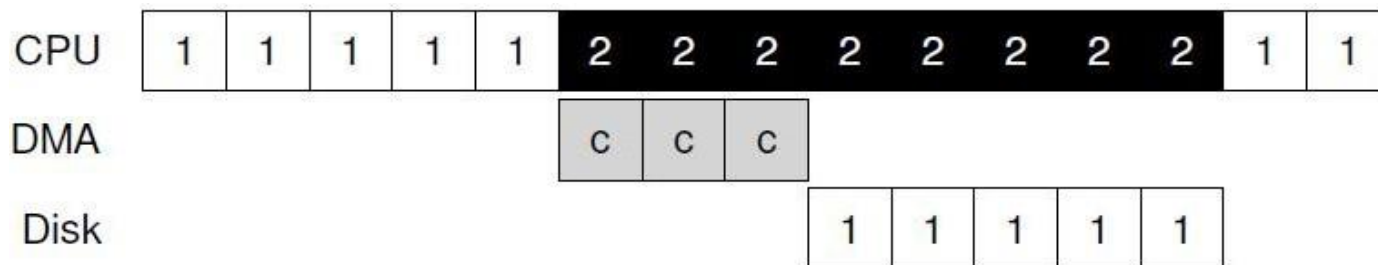
- Interrupt switches process to kernel mode
- Interrupt Descriptor Table (IDT) stores pointers to interrupt handlers (interrupt service routines)
  - Interrupt (IRQ) number identifies the interrupt handler to run for a device
- Interrupt handler acts upon device notification, unblocks the process waiting 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)

- CPU cycles wasted in copying data to/from device



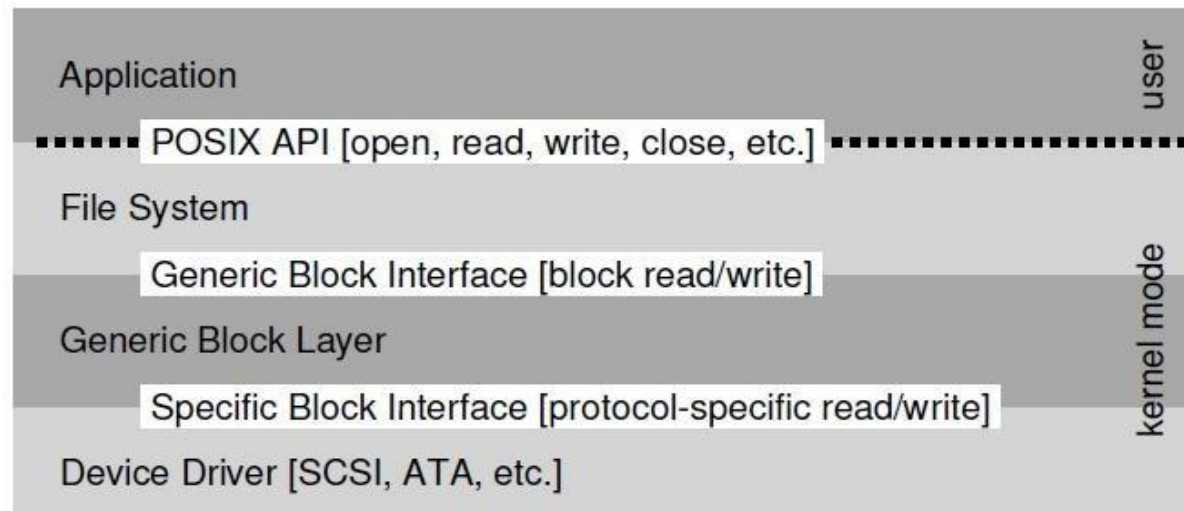
- Instead, a special piece of hardware (DMA engine) copies from main memory to device
  - CPU gives DMA engine the memory location of data
  - In case of read, interrupt raised after DMA completes
  - In case of write, disk starts writing after DMA completes





# Device Driver

- Device driver: part of OS code that talks to specific device, gives commands, handles interrupts etc.
- Most OS code abstracts the device details
  - E.g., file system code is written on top of a generic block interface



# The file abstraction

- File – linear array of bytes, stored persistently
  - Identified with file name (human readable) and a OS-level identifier (“inode number”)
  - Inode number unique within a file system
- Directory contains other subdirectories and files, along with their inode numbers
  - Stored like a file, whose contents are filename-to-inode mappings

# Directory tree

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

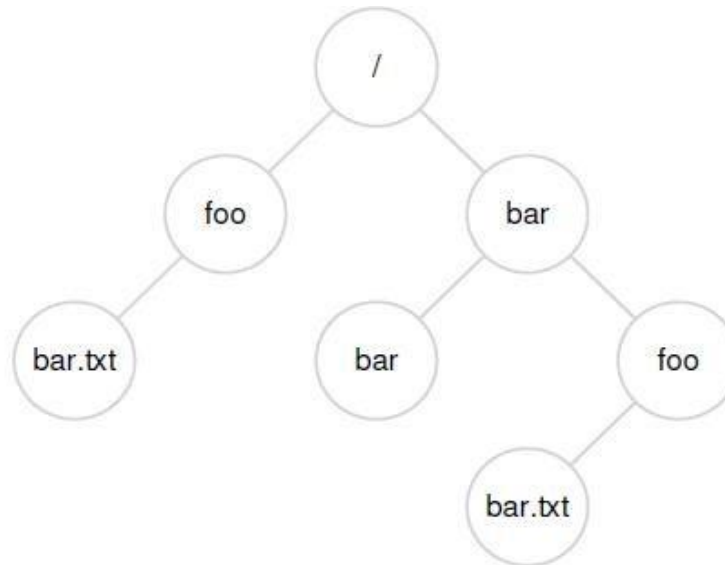


Figure 39.1: An Example Directory Tree


# Operations on files (1)

- Creating a file

- `open()` system call with flag to create
- Returns a number called “file descriptor”

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- Opening a file

- Existing files must be opened before they can be read/written
- Also uses `open` system call, and returns `fd` 

- All other operations on files use the file descriptor

- `close()` system call closes the file

# Operations on files (2)

- Reading/writing files: `read()` / `write()` system calls
  - Arguments: file descriptor, buffer with data, size
- Reading and writing happens sequentially by default
  - Successive read/write calls fetch from current offset
- What if you want to read/write at random location
  - `lseek()` system call lets you seek to random offset
- Writes are buffered in memory temporarily, so `fsync()` system call flushes all writes to disk
- Other operations: rename file, delete (unlink) file, get statistics of a file

# Operations on directories

- Directories can also be accessed like files
  - Operations like 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)
```

# Memory mapping a file

- Alternate way of accessing a file, instead of using file descriptors and read/write syscalls
- `mmap()` allocates a page in the virtual address space of a process
  - “Anonymous” page: used to store program data
  - File-backed page: contains data of file (filename provided as arg to `mmap`)
- When file is mmaped, file data copied into one or more pages in memory, can be accessed like any other memory location in program