

Input/output, Files and Directories

- Issues
- Low Level Input/Output (I/O)
 - Programmed
 - Interrupt driven
 - Direct memory access (DMA)
 - Channel
- Virtual I/O
 - File, disk and directory organisation
- Reference:
 - Tanenbaum (5th ed), sections 5.5.7, 6.2 & 6.4.3
 - Bryant and O'Hallaron, Chapter 11

Programmed I/O

- Simplest form of I/O: One character is in-/output at a time
- To output:
 - Wait until **terminal device** is ready
 - Give character to terminal device
- To input:
 - Wait until **keyboard device** is ready
 - Get character from keyboard device
 - Set **ReadyToReceiveData** flag
- These **tight** loops are referred to as **busy wait** loops
- Very wasteful of CPU resources (i.e. doesn't allow other users to use CPU during I/O)
- PeANUt I/O (trap #2, trap #3) is programmed I/O

Issues

- Device Differences:
 - Speed: from disk drive to keyboard
 - Unit of transfer: characters, words, bytes, blocks or records
 - Data representation: different encoding
 - Allowed operations: rewind or no rewind
 - Error conditions: detection and action
- Design Objectives
 - Character code independent
 - Device independent: “any disc drive will do”
 - Efficient
 - Uniform device treatment: (as far as possible)

Interrupt driven I/O

- Slightly more sophisticated: One character I/O at a time
- To output:
 - Request **interrupt** from terminal device when ready
 - Do something ... (until interrupt is received)
 - Give character to terminal device
- To input:
 - Request interrupt from **keyboard device** when ready
 - Do something ... (until interrupt is received)
 - Get character from keyboard device
- Good: Can productively **multiprocess** while waiting
- Bad: **Process switching** is expensive, so some time is still wasted

DMA (Direct memory access) I/O

- Usually multiple characters (or other data) are sent in sequence
- With **DMA**, a special purpose external processor does the **programmed I/O** for us, and then tells us when it is finished
- Transfers may often be in Kilobytes or Megabytes
- Number of interrupts is reduced dramatically over that of interrupt-driven I/O
- Each transfer uses the bus, thus inhibiting the CPU from using it (this effect is not serious, unless vast amounts of I/O occur)

Channel I/O

- On heavily I/O-bound machines, DMA still generates too many interrupts
- For complex I/O transactions involving multiple DMAs, a small program is created and sent to a **channel processor** (dedicated to I/O)
- This processor executes the program, only raising an interrupt at the end, thereby reducing the interrupt frequency

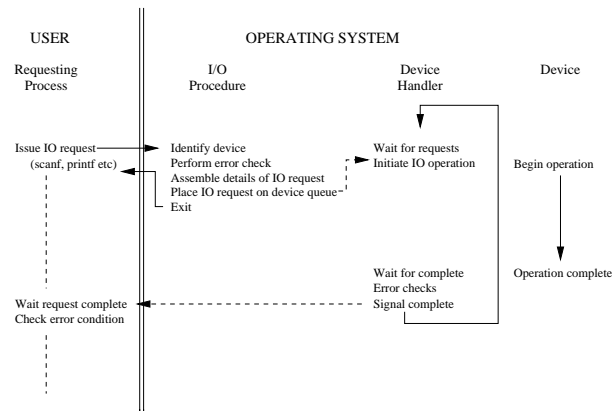
DMA I/O – Example

- If we were to output 2,048 characters from address 1,024 onwards to a disk:
 - Send start address 1,024 and the length 2,048 to the **DMA controller**
 - Send a code (number) for the disk device (e.g. **4**), and a **write** code
 - The DMA controller transfers the data from memory to the disk using programmed I/O
 - When finished, the DMA controller sends an interrupt signifying completion

Responsibility: Virtual I/O

- Level two (ISA level) I/O instructions
 - Allow users to read and write **any** data
 - This would leave the machine without any data security
 - Also somewhat primitive, requiring the user to have substantial knowledge of the structure of the data
 - When reading and writing, many different (correctable) errors may occur
- Level three (O/S level) I/O instructions
 - Users usually don't want to know (just want a successful transfer)
 - Hide much of this complexity from the user and provide a level of security

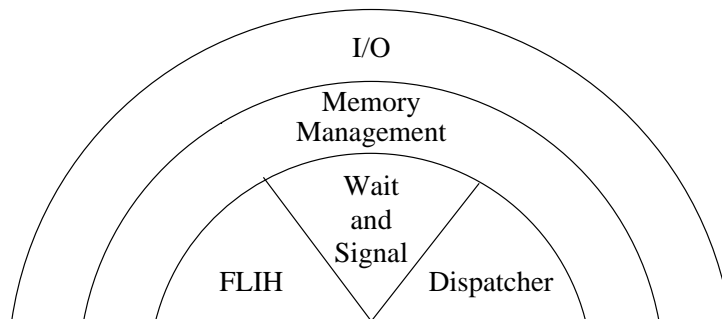
Basic Input/Output Operation



I/O Organisation

- Files provide a means of organising and accessing data in a convenient manner
- A file is a linear sequence of 8 bit bytes indexed from 0 to some maximum (e.g. $2^{32}-1$ or $2^{64}-1$)
- Associated with every open file is a pointer to the next byte to be read or written
- In UNIX, even devices such as printers and terminals are treated as (special) files, e.g. `/dev/lp` `/dev/tty`; also selected kernel data (`/proc`)

OS Structure



File Types

- **Sequential Files:**
 - Five basic operations: Open, read, write, rewind, close
 - Read/write from/to next item in file
 - *rewind* operation permits return to beginning of file
 - Access analogous to that of a magnetic tape
- **Random Access Files:**
 - File position (pointer) can be explicitly set

Principal Unix File System Calls

System Call	Meaning
creat(name,mode)	Create a file; <i>mode</i> specifies the protection mode
unlink(name)	Delete a file (assuming that there is only 1 link to it)
open(name,mode)	Open or create a file and return a file descriptor
close(fd)	Close a file
read(fd,buf,count)	Read <i>count</i> bytes into <i>buf</i>
write(fd,buf,count)	Write <i>count</i> bytes from <i>buf</i>
lseek(fd,offset,w)	Move the file pointer as required by <i>offset</i> and <i>w</i>
stat(name, buf)	Return information about a file
chmod(name,mode)	Change the protection mode of a file
fcntl(fd,cmd,...)	Do various control operations such as locking (part of)
mmap(buf,count,mode,flags,fd,offset)	Map VM pages from <i>buf</i> to <i>buf + count-1</i> to pages in file <i>fd</i> from <i>offset</i>

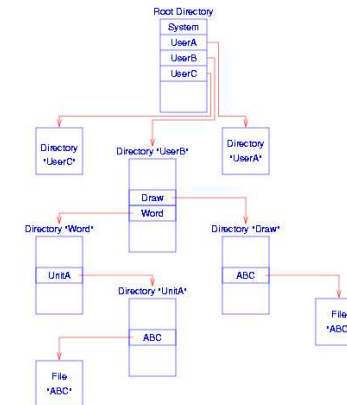
File Management: Directories

- Directories are just files with special properties
 - Are files that contain pointers to files
 - Allow information to be organised within the file system
 - Some early O/S did not support directories (e.g. **MS-DOS**)
- **Directories** are maintained by the operating system
- Must account for:
 - Creation, deletion, renaming, protection (access rights)

Memory Mapping: Files in Virtual Memory

- instead of file access via `read()` / `write()`, can directly *map* pages in a file into the Virtual Memory of a process
 - thus, reading and writing to page in a file can be achieved by normal (to the user program) memory accesses
 - if page written to, corresponding file page is updated when swapped out of VM system
- in Unix, the **loader** in fact does not copy the ELF file sections into memory; it just sets up page mappings
 - when such a page is touched, a copy is made in Phys. Mem.
 - if page is writable, a (VM) copy is made upon write
 - pages of shared libraries (dynamic linking) also *mmap*'ed

Typical directory structure



Principal Unix Directory Management Calls

System Call	Meaning
<code>mkdir(name,mode)</code>	Create a new directory
<code>rmdir(name)</code>	Delete an empty directory
<code>opendir(name)</code>	Open a directory for reading
<code>readdir(dirpointer)</code>	Read the next entry in a directory
<code>closedir(dirpointer)</code>	Close a directory
<code>chdir(dirname)</code>	Change working directory to <i>dirname</i>
<code>link(name1, name2)</code>	Create a directory entry <i>name2</i> pointing to <i>name1</i>
<code>unlink(name)</code>	Remove <i>name</i> from its directory

File and Directory Implementation (UNIX)

- Each file/directory is associated with a 64 byte block of information called an index-node or **i-node**
- i-nodes
 - tell who owns the file, permissions, location of data etc
 - are located in predefined locations in numerical sequence, ie. so they can be easily located
- A directory is a file and an i-node number. On the command


```
open("foo.c",0)
```

 - system searches working directory for file "foo.c" to find its i-node number
 - reads i-node and accesses file

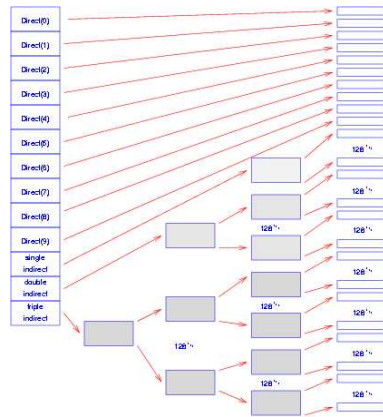
Typical I-Node Content

1. File type (ordinary file, directory, special file, blocked or unblocked), the 9 RWX protection bits and a few other bits
2. Number of links to the file (directory entries)
3. Owners ID
4. Owners Group
5. File length (bytes)
6. Thirteen disk addresses
7. Last time read
8. Last time written
9. Last time i-node changed

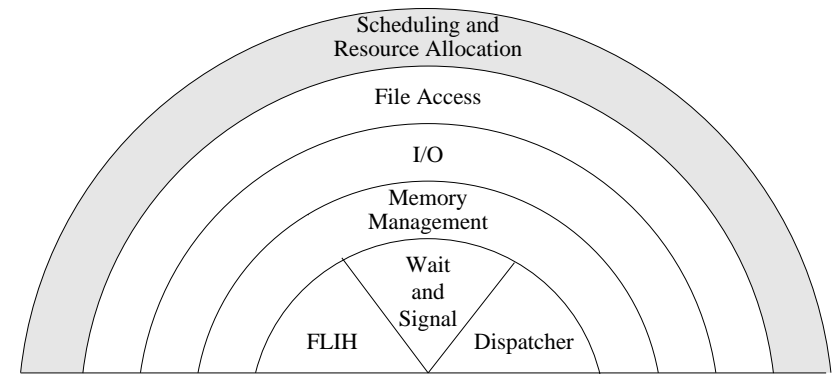
Disk Based I/O Devices

- Disks are organised into **cylinders, tracks** and **sectors**
- Disk store allocation strategy
 - Files are partitioned into fixed-length units called **blocks**
 - Blocks allocated consecutive **sectors** where possible
 - **Fragmentation** problems arise from variable file lengths and isolated unused blocks
- Block addressing scheme
 - Addresses of first 10 blocks are explicitly given in the i-node
 - i-node address 11 points to a disk block containing (block size/address size) further addresses (indirect block)
 - i-node address 12 points to a block containing pointers to blocks containing address (double indirection) which blocks are allocated to a file
 - i-node address 13 gives triple indirection!
 - With a 1024 block size UNIX i-nodes can reference files up to 17 GB

UNIX I-node Addressing Scheme



Typical OS Structure



Windows File System

More complicated - read Tanenbaum 6.4.3!!