Input/Output, Files and Directories

- refs:
  - [Tanembaum, sects 5.5.7, 6.2 & 6.4.3]
  - [O'H&Bryant, ch 11] [Null&Lobur, sect 7.1–7.4]
- issues
- low-level input/output (I/O)
  - programmed
  - interrupt-driven
  - direct memory access (DMA)
  - channel
- virtual I/O
  - file, disk and directory organisation
- other issues:
  - Assignment 2: feedback, solution up
  - Assignment 3: a taste of research in computer systems!

Issues in I/O

- 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:
  - efficient
  - character code independent
  - device independent: "any disc drive will do"
  - uniform device treatment: (as far as possible)

  $\Rightarrow$ requires suitable abstractions!

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`) can be thought of as programmed I/O

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: (frequent) 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 (DMA controller) does the programmed I/O for us, and then tells us when it is finished
- transfers may often be in KBs or MBs
- 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)
- example: output 2048 characters from address 1024 onwards to a disk:
  - send start address 1024 and the length 2048 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

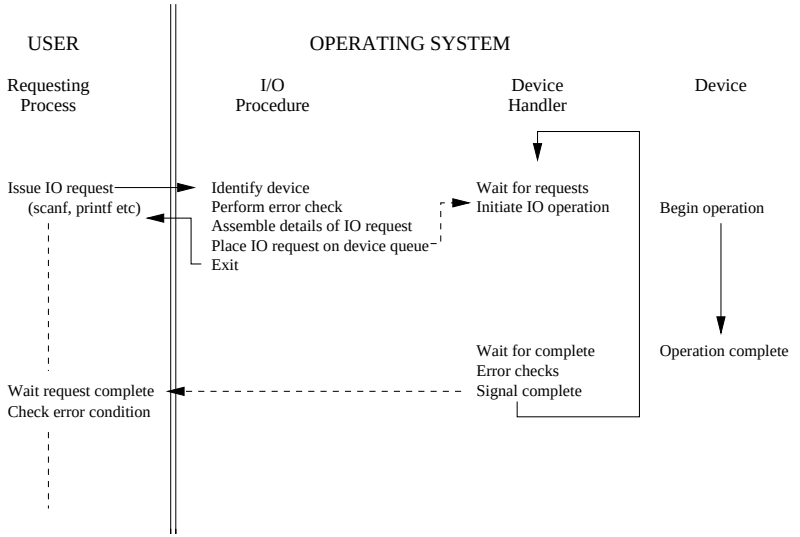
Channel I/O and Device Access

- 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
- when directly communicating with a device (e.g. channel processor or DMA controller), the CPU reads/writes to registers on that device:
  - this could be done via special instructions (drawbacks?)
  - *or* memory-mapped I/O: mapping these registers to special locations in physical memory
  - recall [O'H&Bryant, fig 1-4]
    - thus memory can provide an abstraction to devices

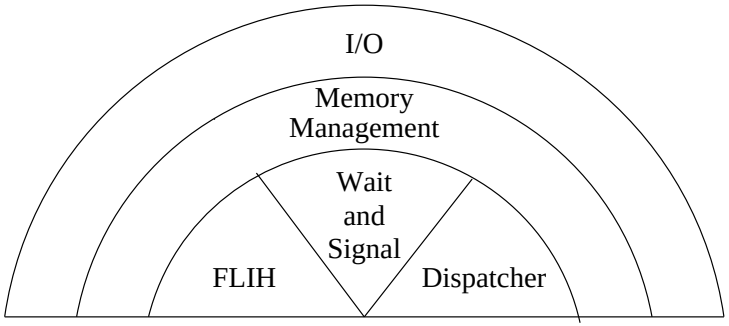
Responsibility: Virtual I/O

- if access to I/O devices was allowed at user-level:
  - inconvenient: requires users to have substantial knowledge of the devices
  - also when reading and writing, many different (correctable) errors may occur
  - arbitration problems: what if two users tried to access the disk simultanously?
  - this would leave the machine without any data security!
- instead, O/S level I/O instructions (traps) provide an abstraction to I/O:
  - 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
- the parts of the OS which access devices are called device drivers

Basic Input/Output Operation



OS Structure



File Organisation and Types

- 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`)
  - this enables these to be read/written to as files - a powerful abstraction
- 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

- a system call is the HLL interface to the OS
  - thus, the system call function's code executes a corresponding trap instruction
- supports both sequential and random access file types

system call	meaning
`creat(name, mode)`	create a file; `mode` 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 `count` bytes into `buf`
`write(fd, buf, count)`	write `count` bytes from `buf`
`lseek(fd, offset, w)`	move the file pointer as required by `offset` and `w`
`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) a file
`mmap(buf, count, mode, flags, fd, offset)`	memory map VM pages from `buf` to `buf + count-1` to pages in file `fd` from `offset`

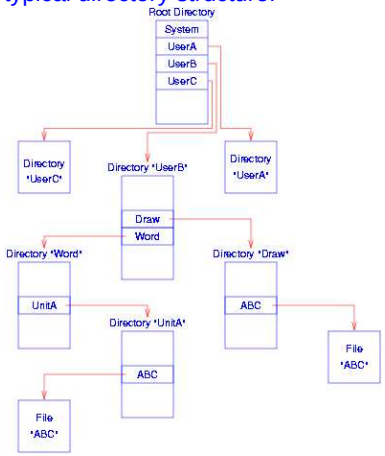
Memory Mapping: Files in Virtual Memory

- instead of file access via `read()` / `write()`, we 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 a page is written to, the corresponding page in file is updated when swapped out of VM system
  - note: in VM, the 'master' image of a page is on disk anyway
- in Unix, the link loader in fact does not copy the ELF image file sections into memory; it just sets up page mappings
  - when such a page is touched, a copy is made in physical memory
  - even the BSS section (uninitialized global data) is `mmap`'ed to an anonymous file (of zeroes)
  - if page is writable, a (VM) copy is made upon write (so the ELF file does not get modified)
  - (dynamically linked) pages of shared libraries are also `mmap`'ed
- again, provides a powerful abstraction

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)

typical directory structure:



Principal Unix Directory Management Calls

system call	meaning:
`mkdir(name,mode)`	create a new directory
`rmdir(name)`	delete an empty directory
`opendir(name)`	open a directory for reading
`readdir(dirpointer)`	read the next entry in a directory
`closedir(dirpointer)`	close a directory
`chdir(dirname)`	change working directory to `dirname`
`link(name1, name2)`	create a directory entry `name2` pointing to `name1`
`unlink(name)`	remove `name` from its directory

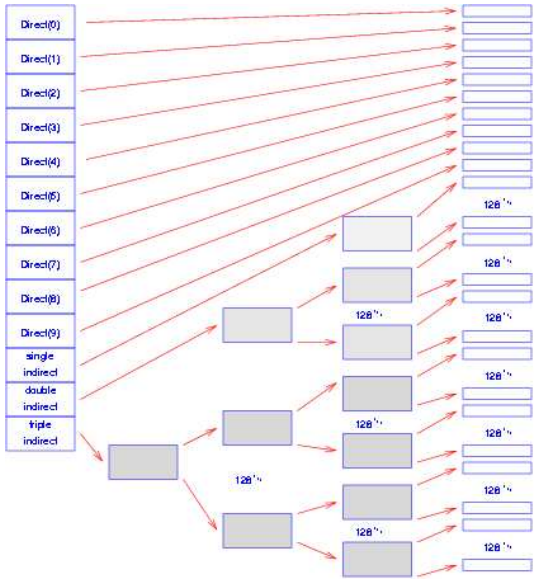
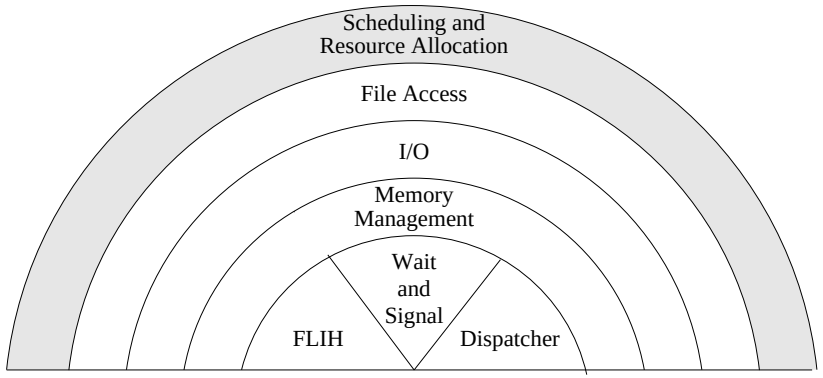
File and Directory Implementation (UNIX)

- 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, i.e. so they can be easily located
- a directory is a file and an i-node number. On the call `open("foo.c", 0);`
  - the OS searches the working directory for file `foo.c` to find its i-node number
  - reads i-node and accesses file
- typical i-node contents:
  - file type (ordinary file, directory, special file, blocked or unblocked), the 9 RWX protection bits and a few other bits
  - number of links to the file (directory entries)
  - owners ID and group ID
  - file length (bytes)
  - thirteen disk addresses
  - last time file read / written / i-node changed

Disk Based I/O Devices

- recall that disks are organised into cylinders, tracks and sectors
- disk storage 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