Operating Systems

Steven Hand

Michaelmas / Lent Term 2008/09

17 lectures for CST IA

Handout 4

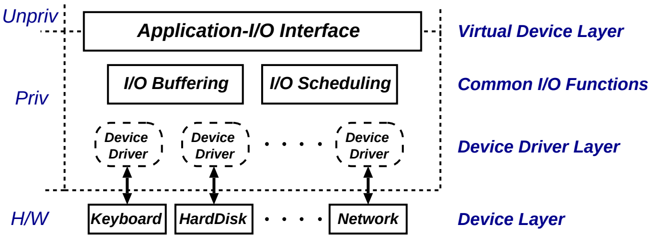
I/O Hardware

- Wide variety of 'devices' which interact with the computer via I/O:
  - Human readable: graphical displays, keyboard, mouse, printers
  - Machine readable: disks, tapes, CD, sensors
  - Communications: modems, network interfaces
- They differ significantly from one another with regard to:
  - Data rate
  - Complexity of control
  - Unit of transfer
  - Direction of transfer
  - Data representation
  - Error handling

⇒ hard to present a uniform I/O system which masks all complexity

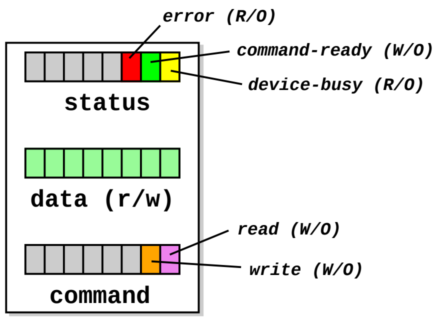
I/O subsystem is generally the 'messiest' part of OS.

I/O Subsystem



- Programs access virtual devices:
  - terminal streams not terminals
  - windows not frame buffer
  - event stream not raw mouse
  - files not disk blocks
  - printer spooler not parallel port
  - transport protocols not raw ethernet
- OS deals with processor–device interface:
  - I/O instructions versus memory mapped
  - I/O hardware type (e.g. 10's of serial chips)
  - polled versus interrupt driven
  - processor interrupt mechanism

Polled Mode I/O



- Consider a simple device with three registers: `status`, `data` and `command`.
- (Host can read and write these via bus)
- Then polled mode operation works as follows:
  - Host repeatedly reads `device_busy` until clear.
  - Host sets e.g. `write` bit in `command` register, and puts data into `data` register.
  - Host sets `command_ready` bit in `status` register.
  - Device sees `command_ready` and sets `device_busy`.
  - Device performs write operation.
  - Device clears `command_ready` & then `device_busy`.
- What's the problem here?

Interrupts Revisited

Recall: to handle mismatch between CPU and device speeds, processors provide an interrupt mechanism:

- at end of each instruction, processor checks interrupt line(s) for pending interrupt
- if line is asserted then processor:
  - saves program counter,
  - saves processor status,
  - changes processor mode, and
  - jump to a well known address (or its contents)
- after interrupt-handling routine is finished, can use e.g. the `rti` instruction to resume where we left off.

Some more complex processors provide:

- multiple levels of interrupts
- hardware vectoring of interrupts
- mode dependent registers

Interrupt-Driven I/O

Can split implementation into low-level *interrupt handler* plus per-device *interrupt service routine*:

- interrupt handler (processor-dependent) may:
  - save more registers
  - establish a language environment (e.g. a C run-time stack)
  - demultiplex interrupt in software.
  - invoke appropriate interrupt service routine (ISR)
- Then interrupt service routine (device-specific but not processor-specific) will:
  1. for programmed I/O device:
     - transfer data.
     - clear interrupt (sometimes a side effect of tx).
  1. for DMA device:
     - acknowledge transfer.
  2. request another transfer if there are any more I/O requests pending on device.
  3. signal any waiting processes.
  4. enter scheduler or return.

Question: who is scheduling who?

Device Classes

Homogenising device API completely not possible

⇒ OS generally splits devices into four *classes*:

1. Block devices (e.g. disk drives, CD):
   - commands include `read`, `write`, `seek`
   - raw I/O or file-system access
   - memory-mapped file access possible
2. Character devices (e.g. keyboards, mice, serial ports):
   - commands include `get`, `put`
   - libraries layered on top to allow line editing
3. Network Devices
   - varying enough from block and character to have own interface
   - Unix and Windows/NT use *socket* interface
4. Miscellaneous (e.g. clocks and timers)
   - provide current time, elapsed time, timer
   - `ioctl` (on UNIX) covers odd aspects of I/O such as clocks and timers.

I/O Buffering

- Buffering: OS stores (its own copy of) data in memory while transferring to or from devices
  - to cope with device speed mismatch
  - to cope with device transfer size mismatch
  - to maintain "copy semantics"
- OS can use various kinds of buffering:
  1. single buffering — OS assigns a system buffer to the user request
  2. double buffering — process consumes from one buffer while system fills the next
  3. circular buffers — most useful for bursty I/O
- Many aspects of buffering dictated by device type:
  - character devices ⇒ line probably sufficient.
  - network devices ⇒ bursty (time & space).
  - block devices ⇒ lots of fixed size transfers.
  - (last usually major user of buffer memory)

Blocking v. Nonblocking I/O

From the programmer's point of view, I/O system calls exhibit one of three kinds of behaviour:

1. Blocking: process suspended until I/O completed
   - easy to use and understand.
   - insufficient for some needs.
2. Nonblocking: I/O call returns as much as available
   - returns almost immediately with count of bytes read or written (possibly 0).
   - can be used by e.g. user interface code.
   - essentially application-level "polled I/O".
3. Asynchronous: process continues to run while I/O executes
   - I/O subsystem explicitly signals process when its I/O request has completed.
   - most flexible (and potentially efficient).
   - . . . but also most difficult to use.

Most systems provide both blocking and non-blocking I/O interfaces; modern systems (e.g. NT, Linux) also support asynchronous I/O, but used infrequently.

Other I/O Issues

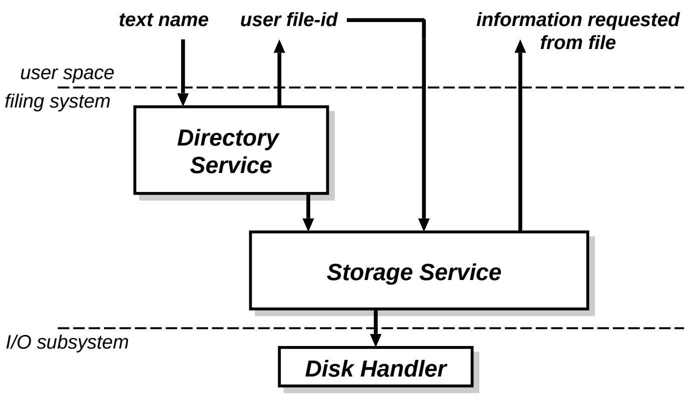
- Caching: fast memory holding copy of data
  - can work with both reads and writes
  - key to I/O performance
- Scheduling:
  - e.g. ordering I/O requests via per-device queue
  - some operating systems try fairness. . .
- Spooling: queue output for a device
  - useful for "single user" devices which can serve only one request at a time (e.g. printer)
- Device reservation:
  - system calls for acquiring or releasing exclusive access to a device (careful!)
- Error handling:
  - e.g. recover from disk read, device unavailable, transient write failures, etc.
  - most I/O system calls return an error number or code when an I/O request fails
  - system error logs hold problem reports.

I/O and Performance

- I/O is a major factor in overall system performance
  - demands CPU to execute device driver, kernel I/O code, etc.
  - context switches due to interrupts
  - data copying, buffering, etc
  - (network traffic especially stressful)
- Improving performance:
  - reduce number of context switches
  - reduce data copying
  - reduce # interrupts by using large transfers, smart controllers, adaptive polling (e.g. Linux NAPI)
  - use DMA where possible
  - balance CPU, memory, bus and I/O for best throughput.

Improving I/O performance is a major remaining OS challenge

File Management



Filing systems have two main components:

1. Directory Service
   - maps from names to file identifiers.
   - handles access & existence control
2. Storage Service
   - provides mechanism to store data on disk
   - includes means to implement directory service

File Concept

What is a file?

- Basic abstraction for non-volatile storage.
- Typically comprises a single contiguous logical address space.
- Internal structure:
  1. None (e.g. sequence of words, bytes)
  2. Simple record structures
     - lines
     - fixed length
     - variable length
  3. Complex structures
     - formatted document
     - relocatable object file
- Can simulate 2,3 with byte sequence by inserting appropriate control characters.
- All a question of who decides:
  - operating system
  - program(mer).

Naming Files

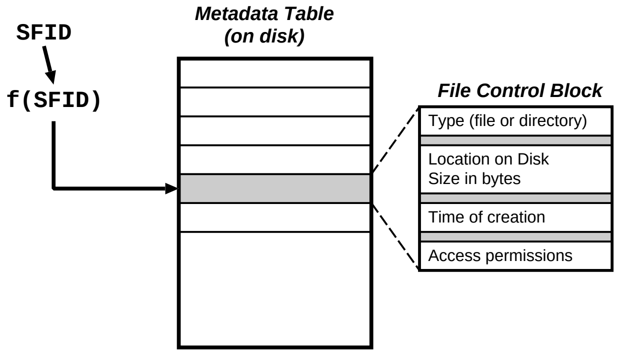
Files usually have at least two kinds of 'name':

1. system file identifier (SFID):
   - (typically) a unique integer value associated with a given file
   - SFIDs are the names used within the filing system itself
2. human-readable name, e.g. `hello.java`
   - what users like to use
   - mapping from human name to SFID is held in a *directory*, e.g.

Name	SFID
hello.java	12353
Makefile	23812
README	9742

   - directories also non-volatile ⇒ must be stored on disk along with files.
3. Frequently also get user file identifier (UFID)
   - used to identify *open* files (see later)

File Meta-data



As well as their contents and their name(s), files can have other attributes, e.g.

- Location: pointer to file location on device
- Size: current file size
- Type: needed if system supports different types
- Protection: controls who can read, write, etc.
- Time, date, and user identification: for protection, security and usage monitoring.

Together this information is called **meta-data**. It is contained in a file control block.

Directory Name Space (I)

What are the requirements for our name space?

- Efficiency: locating a file quickly.
- Naming: user convenience
  - allow two (or more generally N) users to have the same name for different files
  - allow one file have several different names
- Grouping: logical grouping of files by properties (e.g. all Java programs, all games)

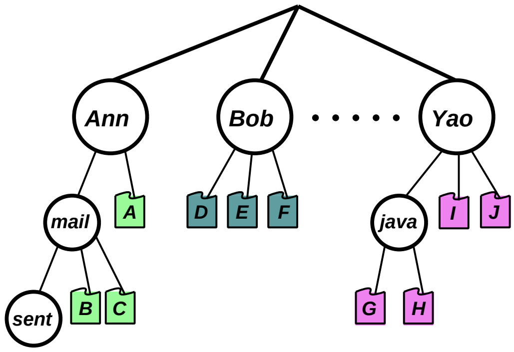
First attempts:

- Single-level: one directory shared between all users

  ⇒ naming problem
  ⇒ grouping problem

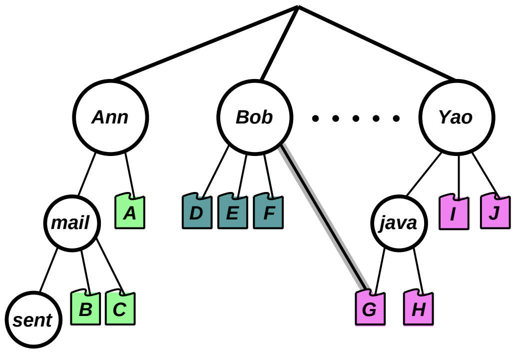
- Two-level directory: one directory per user
  - access via *pathname* (e.g. `bob:hello.java`)
  - can have same filename for different user
  - but still no grouping capability.

Directory Name Space (II)



- Get more flexibility with a general hierarchy.
  - directories hold files or [further] directories
  - create/delete files relative to a given directory
- Human name is full path name, but can get long:
  e.g. /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c
  - offer relative naming
  - login directory
  - current working directory
- What does it mean to delete a [sub]-directory?

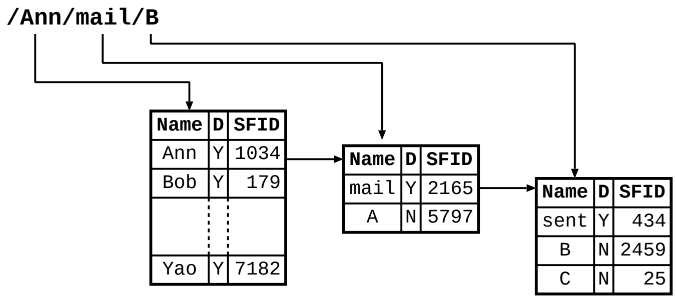
Directory Name Space (III)



- Hierarchy good, but still only one name per file.

⇒ extend to directed acyclic graph (DAG) structure:
  - allow shared subdirectories and files.
  - can have multiple aliases for the same thing
- **Problem**: dangling references
- Solutions:
  - back-references (but require variable size records); or
  - reference counts.
- **Problem**: cycles. . .

Directory Implementation



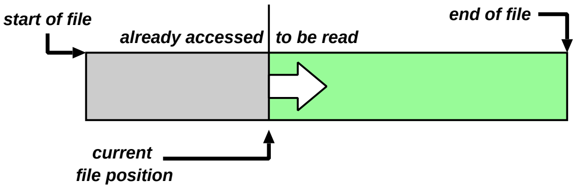
- Directories are non-volatile ⇒ store as “files” on disk, each with own SFID.
- Must be different types of file (for traversal)
- Explicit directory operations include:
  - create directory
  - delete directory
  - list contents
  - select current working directory
  - insert an entry for a file (a “link”)

File Operations (I)

UFID	*SFID*	*File Control Block (Copy)*
1	*23421*	*location on disk, size,...*
2	*3250*	*" "*
3	*10532*	*" "*
4	*7122*	*" "*
⋮	⋮	

- Opening a file: `UFID = open(<pathname>)`
  1. directory service recursively searches for components of `<pathname>`
  2. if all goes well, eventually get `SFID` of file.
  3. copy file control block into memory.
  4. create new `UFID` and return to caller.
- Create a new file: `UFID = create(<pathname>)`
- Once have `UFID` can read, write, etc.
  - various modes (see next slide)
- Closing a file: `status = close(UFID)`
  1. copy [new] file control block back to disk.
  2. invalidate `UFID`

File Operations (II)



- Associate a cursor or file position with each open file (viz. UFID)
  - initialised at open time to refer to start of file.
- Basic operations: *read next* or *write next*, e.g.
  - `read(UFID, buf, nbytes)`, or `read(UFID, buf, nrecords)`
- Sequential Access: above, plus `rewind(UFID)`.
- Direct Access: *read* N or *write* N
  - allow "random" access to any part of file.
  - can implement with `seek(UFID, pos)`
- Other forms of data access possible, e.g.
  - append-only (may be faster)
  - indexed sequential access mode (ISAM)

Other Filing System Issues

- Access Control: file owner/creator should be able to control what can be done, and by whom.
  - normally a function of directory service ⇒ checks done at file *open* time
  - various types of access, e.g.
    - ∗ read, write, execute, (append?),
    - ∗ delete, list, rename
  - more advanced schemes possible (see later)
- Existence Control: what if a user deletes a file?
  - probably want to keep file in existence while there is a valid pathname referencing it
  - plus check entire FS periodically for garbage
  - existence control can also be a factor when a file is renamed/moved.
- Concurrency Control: need some form of *locking* to handle simultaneous access
  - may be mandatory or advisory
  - locks may be shared or exclusive
  - granularity may be file or subset