Lecture 10:
I/O Systems

www.cl.cam.ac.uk/Teaching/2001/OSFounds/

Today’s Lecture

Today we’ll cover:
- How does OS manage and control I/O operations and devices?
  - I/O hardware (revision).
  - Interrupts.
  - Classes of devices.
  - I/O services.

I/O Hardware
- Wide variety of ‘devices’ which interact with the computer via I/O, e.g.
  - 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
- difficult to present a uniform I/O system which hides all the 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:
  - H repeatedly reads device_busy until clear.
  - H sets e.g. write bit in command register, and puts data into data register.
  - H sets command_ready bit in status register.
  - D sees command_ready and sets device_busy.
  - D performs write operation.
  - D clears command_ready & then device_busy.
- What's the problem here?

Interru ts 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 well known address (or its contents)
- after interrupt-handling routine is finished, can use e.g. the rti instruction to resume.

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.
  - demultiplex interrupt in software.
  - invoke appropriate interrupt service routine (ISR)
- Then ISR (device- 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):
   - 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 (a copy of) data in memory while transferring between 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 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 runs 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; fewer support asynchronous I/O.

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 if device is “single user” (i.e. can serve only one request at a time), e.g., printer.
- **Device reservation**:
  - system calls for acquiring or releasing exclusive access to a device (care required)
- **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 a major factor in system performance
  - demands CPU to execute device driver, kernel I/O code, etc.
  - context switches due to interrupts
  - data copying
  - network traffic especially stressful.
- **Improving performance**:
  - reduce number of context switches
  - reduce data copying
  - reduce # interrupts by using large transfers, smart controllers, polling
  - use DMA where possible
  - balance CPU, memory, bus and I/O performance for highest throughput.

Improving I/O performance is one of the main remaining systems challenges...