[08] IO SUBSYSTEM

OUTLINE

- Input/Output (IO)
  - Hardware
  - Device Classes
  - OS Interfaces
- Performing IO
  - Polled Mode
  - Interrupt Driven
  - Blocking vs Non-blocking
- Handling IO
  - Buffering & Strategies
  - Other Issues
  - Kernel Data Structures
  - Performance

INPUT/OUTPUT

- **Input/Output (IO)**
  - **Hardware**
  - **Device Classes**
  - **OS Interfaces**
- Performing IO
- Handling IO

IO HARDWARE

Very wide range of **devices** that interact with the computer via **input/output** (IO):

- Human readable: graphical displays, keyboard, mouse, printers
- Machine readable: disks, tapes, CD, sensors
- Communications: modems, network interfaces, radios

All differ significantly from one another with regard to:

- **Data rate**: orders of magnitude different between keyboard and network
- **Control complexity**: printers much simpler than disks
- **Transfer unit** and **direction**: blocks vs characters vs frame stores
- **Data representation**
- **Error handling**

IO SUBSYSTEM

Results in IO subsystem generally being the "messiest" part of the OS

- So much variety of devices
- So many applications
- So many dimensions of variation:
  - Character-stream or block
  - Sequential or random-access
  - Synchronous or asynchronous
  - Shareable or dedicated
  - Speed of operation
  - Read-write, read-only, or write-only

Thus, completely homogenising device API is not possible so OS generally splits devices into four classes

DEVICE CLASSES

Block devices (e.g. disk drives, CD)

- Commands include `read`, `write`, `seek`
- Can have **raw** access or via (e.g.) filesystem ("cooked") or memory-mapped

Character devices (e.g. keyboards, mice, serial):

- Commands include `get`, `put`
- Layer libraries on top for line editing, etc

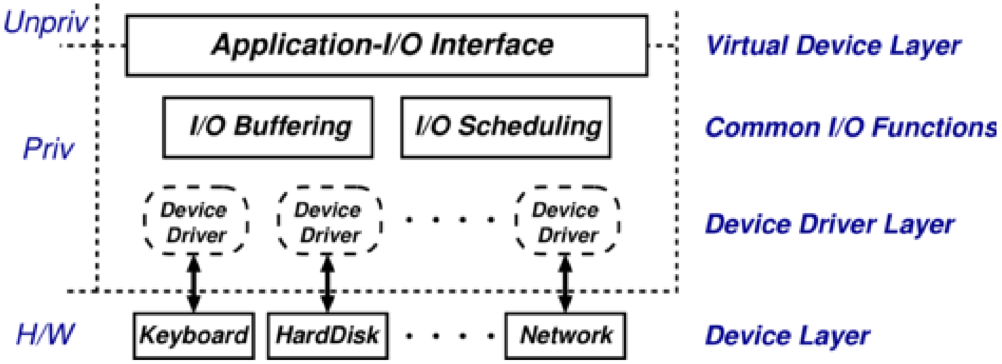
Network Devices

- Vary enough from block and character devices to get their own interface
- Unix and Windows NT use the **Berkeley Socket** interface

Miscellaneous

- Current time, elapsed time, timers, clocks
- (Unix) `ioctl` covers other odd aspects of IO

OS INTERFACES



Programs access **virtual devices**: Terminal streams not terminals, windows not frame buffer, event streams not raw mouse, files not disk blocks, print spooler not parallel port, transport protocols not raw Ethernet frames

OS handles the processor-device interface: IO instructions vs memory mapped devices; IO hardware type (e.g. 10s of serial chips); Polled vs interrupt driven; CPU interrupt mechanism

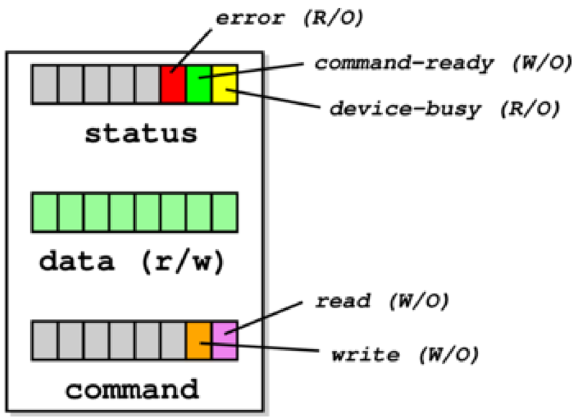
Virtual devices then implemented:

- In kernel, e.g. files, terminal devices
- In daemons, e.g. spooler, windowing
- In libraries, e.g. terminal screen control, sockets

PERFORMING IO

- Input/Output (IO)
- **Performing IO**
  - **Polled Mode**
  - **Interrupt Driven**
  - **Blocking vs Non-blocking**
- Handling IO

POLLED MODE



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 clears `device-busy`

What's the problem here?

INTERRUPT DRIVEN

Rather than polling, processors provide an interrupt mechanism to handle mismatch between CPU and device speeds:

- At end of each instruction, processor checks interrupt line(s) for pending interrupt
  - Need not be precise (that is, occur at definite point in instruction stream)
- If line is asserted then processor:
  - Saves program counter & processor status
  - Changes processor mode
  - Jumps to a well-known address (or contents of a well-known address)
- Once interrupt-handling routine finishes, can use e.g. `rti` instruction to resume
- More complex processors may provide:
  - Multiple priority levels of interrupt
  - **Hardware vectoring** of interrupts
  - Mode dependent registers

HANDLING INTERRUPTS

Split the implementation into two parts:

- At the bottom, the **interrupt handler**
- At the top, N **interrupt service routines** (ISR; per-device)

Processor-dependent interrupt handler may:

- Save more registers and establish a language environment
- Demultiplex interrupt in software and invoke relevant ISR

Device- (not processor-) dependent interrupt service routine will:

- For programmed IO device: transfer data and clear interrupt
- For DMA devices: acknowledge transfer; request any more pending; signal any waiting processes; and finally enter the scheduler or return

Question: *Who is scheduling whom*?

- Consider, e.g., network livelock

BLOCKING VS NON-BLOCKING

From programmer's point of view, IO system calls exhibit one of three kinds of behaviour:

- **Blocking**: process suspended until IO completed
  - Easy to use and understand.
  - Insufficient for some needs.
- **Nonblocking**: IO 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 IO"
- **Asynchronous**: process runs while IO executes
  - IO subsystem explicitly signals process when its IO request has completed
  - Most flexible (and potentially efficient)
  - Also most complex to use

HANDLING IO

- Input/Output (IO)
- Performing IO
- **Handling IO**
  - **Buffering & Strategies**
  - **Other Issues**
  - **Kernel Data Structures**
  - **Performance**

IO BUFFERING

To cope with various **impedance mismatches** between devices (speed, transfer size), OS may **buffer** data in memory. Various buffering strategies:

- **Single buffering**: OS assigns a system buffer to the user request
- **Double buffering**: process consumes from one buffer while system fills the next
- **Circular buffering**: most useful for bursty IO

Buffering is useful for smoothing peaks and troughs of data rate, but can't help if on average:

- Process demand > data rate (process will spend time waiting), or
- Data rate > capability of the system (buffers will fill and data will spill)
- Downside: can introduce **jitter** which is bad for real-time or multimedia

Details often dictated by device type: character devices often by line; network devices particularly bursty in time and space; block devices make lots of fixed size transfers and often the major user of IO buffer memory

SINGLE BUFFERING

OS assigns a single buffer to the user request:

- OS performs transfer, moving buffer to userspace when complete (remap or copy)
- Request new buffer for more IO, then reschedule application to consume (**readahead** or **anticipated input**)
- OS must track buffers
- Also affects swap logic: if IO is to same disk as swap device, doesn't make sense to swap process out as it will be behind the now queued IO request!

A crude performance comparison between no buffering and single buffering:

- Let t be time to input block and c be computation time between blocks
- Without buffering, execution time between blocks is $t + c$
- With single buffering, time is $\max(c, t) + m$ where m is the time to move data from buffer to user memory
- For a terminal: is the buffer a line or a char? depends on user response required

DOUBLE BUFFERING

- Often used in video rendering
- Rough performance comparison: takes $\max(c, t)$ thus
  - possible to keep device at full speed if $c < t$
  - while if $c > t$, process will not have to wait for IO
- Prevents need to suspend user process between IO operations
- ...also explains why two buffers is better than one buffer, twice as big
- Need to manage buffers and processes to ensure process doesn't start consuming from an only partially filled buffer

CIRCULAR BUFFERING

- Allows consumption from the buffer at a fixed rate, potentially lower than the **burst rate** of arriving data
- Typically use circular linked list which is equivalent to a FIFO buffer with queue length

OTHER ISSUES

- **Caching**: fast memory holding copy of data for both reads and writes; critical to IO performance
- **Scheduling**: order IO requests in per-device queues; some OSs may even attempt to be fair
- **Spooling**: queue output for a device, useful if device is "single user" (e.g., printer), i.e. can serve only one request at a time
- **Device reservation**: system calls for acquiring or releasing exclusive access to a device (care required)
- **Error handling**: generally get some form of error number or code when request fails, logged into system error log (e.g., transient write failed, disk full, device unavailable, ...)

KERNEL DATA STRUCTURES

To manage all this, the OS kernel must maintain state for IO components:

- Open file tables
- Network connections
- Character device states

Results in many complex and performance criticial data structures to track buffers, memory allocation, "dirty" blocks

Consider reading a file from disk for a process:

- Determine device holding file
- Translate name to device representation
- Physically read data from disk into buffer
- Make data available to requesting process
- Return control to process

PERFORMANCE

IO a major factor in system performance

- Demands CPU to execute device driver, kernel IO code, etc.
- Context switches due to interrupts
- Data copying

Improving performance:

- Reduce number of context switches
- Reduce data copying
- Reduce number of interrupts by using large transfers, smart controllers, polling
- Use DMA where possible
- Balance CPU, memory, bus and IO performance for highest throughput.

Improving IO performance remains a significant challenge...

SUMMARY

- Input/Output (IO)
    - Hardware
    - Device Classes
    - OS Interfaces
- Performing IO
    - Polled Mode
    - Interrupt Driven
    - Blocking vs Non-blocking
- Handling IO
    - Buffering & Strategies
    - Other Issues
    - Kernel Data Structures
    - Performance