09. I/O Systems

9th ed: Ch. 13
10th ed: Ch. 12
Objectives

• To understand the general structure of the I/O subsystem
• To know different ways of performing I/O including polling, interrupts, and direct memory access
• To know of different types of device
• To be aware of other issues including caching, scheduling, and performance
Outline

• I/O subsystem
• I/O devices
• Kernel data structures
Outline

• I/O subsystem
  • Polling
  • Interrupts
  • Interrupt handling
  • Direct Memory Access (DMA)

• I/O devices

• Kernel data structures
Computers and computation rely on I/O

- Need input data to process, and means to output results
- There is a huge range of I/O devices
  - **Human readable**: graphical displays, keyboard, mouse, printers
  - **Machine readable**: disks, tapes, CD, sensors
  - **Communications**: modems, network interfaces, radios
- All differ significantly from one another in several ways:
  - **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**
- I/O management is therefore a major component of an OS
  - New devices come along frequently
  - I/O performance is critical to system performance
  - Also wish to present a homogenous API
I/O subsystem

• Incredible variety of I/O devices but there are commonalities
  • Signals from I/O devices interface with computer
  • A device has at least one connection point, or **port**
  • Devices interconnect via a **bus**, either daisy-chained or shared direct access
  • Devices have integrated or separate controllers (host adapters) containing processor, microcode, private memory, etc that operate the device, handle bus connections, any ports

• Typically device will have registers to hold commands, addresses, data
  • E.g., Data-in register, data-out register, status register, control register

• Devices have addresses and are used by either
  • Direct I/O instructions, usually privileged, or
  • Memory-mapped I/O, where device registers are mapped into processor address space, especially when large (e.g., graphics cards)
Polling

• Consider a simple device
  • Three registers: status, data and command
  • Host can read and write registers via the bus

• Polled mode operation is as follows, for every byte:
  • Host repeatedly reads device-busy until clear
  • Host sets read or 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 requested operation, executing transfer
  • Device clears command-ready and any error bit, and then clears device-busy

• Step 1 is polling – a busy-wait cycle, waiting for some I/O from device
  • This is ok if the device is fast but very inefficient if not
  • If the CPU switches to another task it risks missing a cycle leading to data being overwritten or lost
Interrupts

- More efficient than polling when device is relatively infrequently accessed
- Device triggers **interrupt-request line**
  - Checked by the CPU after each instruction
  - Aligns interrupts with instruction boundaries
- **Interrupt handler** receives the interrupt unless masked
- **Interrupt vector** dispatches interrupt to correct handler
  - Context switch required before and after
  - Priorities applied, and some interrupts may be **non-maskable**
An operating system has other good uses for an efficient hardware and software mechanism that saves a small amount of processor state and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Another example is found in the implementation of system calls. Usually, a program calls system calls. The library routine checks the arguments given by the application, builds a data structure to convey the arguments to the kernel, and then executes a special instruction called a software interrupt, or trap. This instruction has a operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.

Interrupts can also be used to manage the flow of control within the kernel. For example, consider one example of the processing required to complete the process.
Handling interrupts

• Split the implementation into two parts:
  • Bottom half, the **interrupt handler**
  • Top half, **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

• But who is scheduling whom? Consider, e.g., network **livelock**
Direct Memory Access (DMA)

• Used for high-speed I/O devices able to transmit information at close to memory speeds
  • Interrupts good but (e.g.) *livelock* a problem
  • Better if devices can read and write processor memory *directly* – Direct Memory Access (DMA)

• Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention with generic DMA “command” include, e.g.,
  • Source address plus increment / decrement / do nothing
  • Sink address plus increment / decrement / do nothing
  • Transfer size
Direct Memory Access (DMA)

• Only generate one interrupt per block rather than one per byte
• DMA channels may be provided by dedicated DMA controller, or by devices themselves
  • E.g. disk controller passes disk address, memory address and size, and read/write
• All that’s required is that a device can become a bus master
  • Requires ability for arbitration as not just CPU driving the bus
  • Involves cycle stealing as taking the bus away from the CPU
• Scatter/Gather DMA chains multiple requests, e.g., of disk reads into set of buffers
Outline

• I/O subsystem

• I/O devices
  • Device characteristics
  • Blocking, non-blocking, asynchronous I/O
  • I/O structure

• Kernel data structures
I/O device characteristics

• **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
  - On Unix, *ioctl* covers other odd aspects of I/O

<table>
<thead>
<tr>
<th>aspect</th>
<th>variation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-transfer mode</td>
<td>character block</td>
<td>terminal disk</td>
</tr>
<tr>
<td>access method</td>
<td>sequential random</td>
<td>modem CD-ROM</td>
</tr>
<tr>
<td>transfer schedule</td>
<td>synchronous asynchronous</td>
<td>tape keyboard</td>
</tr>
<tr>
<td>sharing</td>
<td>dedicated sharable</td>
<td>tape keyboard</td>
</tr>
<tr>
<td>device speed</td>
<td>latency seek time transfer rate delay between operations</td>
<td></td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only write only read–write</td>
<td>CD-ROM graphics controller disk</td>
</tr>
</tbody>
</table>
Blocking, non-blocking, asynchronous I/O

• From programmer perspective, I/O system calls exhibit one of three behaviours
  
  **Blocking**
  • Process suspended until I/O completed
  • Easy to use and understand but insufficient for some needs
  
  **Nonblocking**
  • I/O call returns all available data, immediately
  • Returns count of bytes read/written, maybe 0
  • `select` following `read/write`
  • Relies on multi-threading

  **Asynchronous**
  • Process continues running while I/O executes with I/O subsystem explicitly signalling I/O completion
  • Most flexible and potentially most efficient, but also most complex to use
I/O structure

• **Synchronous**
  - After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing

• **Asynchronous**
  - After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** allows application to request to the OS to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt
I/O request lifecycle

• Consider process reading a file from disk:
  • 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
Outline

• I/O subsystem
• I/O devices
• Kernel data structures
  • Vectored I/O
  • Buffering
  • Other issues
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 critical 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
Vectored I/O

• Enable one system call to perform multiple I/O operations
  • E.g., Unix *readve* accepts a vector of multiple buffers to read into or write from
• This *scatter-gather* method better than multiple individual I/O calls
  • Decreases context switching and system call overhead
• Some versions provide atomicity
  • Avoids, e.g., worry about multiple threads changing data while I/O occurring
Buffering

- So OS can deal with mismatches between devices, e.g., speed, transfer size), different buffering strategies can be used
  - **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
    - Details often dictated by device type: character devices buffer by line; network devices are very bursty; block devices often the major user of IO buffer memory
- Can smooth peaks/troughs in data rate but can’t help if on average:
  - Process demand > data rate – the process will spend time waiting, or
  - Data rate > capability of the system – the buffers will all fill and data will spill
- However, buffering can introduce jitter which is bad for real-time or multimedia applications
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, ...)
- **Protection**: process might attempt to disrupt normal operation via illegal I/O operations so all such instructions must be privileged and memory-mapped and I/O port memory locations protected, with I/O performed via system calls
- **Performance**: I/O really affects performance through demands on CPU to execute device driver, kernel I/O code, context switches due to interrupts, data copying
Summary

• I/O subsystem
  • Polling
  • Interrupts
  • Interrupt handling
  • Direct Memory Access (DMA)

• I/O devices
  • Device characteristics
  • Blocking, non-blocking, asynchronous I/O
  • I/O structure

• Kernel data structures
  • Vectored I/O
  • Buffering
  • Other issues