I/O Systems

Chapters 12.1, 12.2, 12.3, 12.4, 12.5

# Overview

* Wide array of methods needed to support a variety of I/O devices: methods form I/O subsystem of kernel.
* Two conflicting trends:
  + Increasing standardization of software and hardware interfaces, making it easier to incorporate improving I/O devices into existing OSs and computer systems.
  + Broadening variety of I/O devices, some of which are so new that incorporating them into computers and OSs are more challenging
  + **Device Driver**: uniform device-access interface to I/O subsystem for a device.

# I/O Hardware

* Devices communicate with a computer system via a **port**
* A **bus** is a set of wires and a rigidly defined protocol specifying the kinds of messages that can be sent via the wires.
* Diagram

  Description automatically generatedA series of devices connected linearly to a computer system is a **daisy chain**.
* **PCIe bus**: connects processor-memory subsystem to fast devices.
* **Expansion bus**: connects relatively slow devices like keyboard etc.
* Four disks on bottom left connected via **Serial-attached SCSI** (**SAS**).
* **Controller**: connection of electronics operating a port, bus, or device.
  + **Serial-port controller**: simple device controller. Single chip in the computer controlling signals on wires.
  + **Fibre Channel** (**FC**) bus controller: extremely complex. Used in data centres, implemented as separate circuit board called a **Host Bus Adapter** which connects to a bus in computer, helps process FC protocol messages.

## Memory-Mapped I/O

Table

Description automatically generated

* I/O transfer accomplished via communicating with registers in controller for data and control signals.
* Communication can be achieved by using special **I/O instructions** specifying transfer of byte or word to an I/O port address.
* This triggers bus lines to select the proper device and move bits in or out of device register.
* Alternatively use **memory-mapped I/O**.
* Device control registers are mapped into the address space of processor.
* CPU then executes I/O requests using standard data-transfer instructions to read and write device-control registers at their mapped locations in physical memory.
* I/O Device control consists of **four registers** typically 1-4 bytes in size.:
  + **Data-in register**: read by host to get input
  + **Data-out register**: written by host to send output
  + **Status Register**: contains bits indicating states that can be read by the host e.g. if current command is complete, if bye is available to be read from data-in register, whether device error has occurred etc.
  + **Control/Command Register**: written by host to start command or change mode of device. These modes include:
    - Choosing between full-duplex and half-duplex communication
    - Parity Checking
    - Setting words length (7 or 8 bits)
    - Select speeds supported by serial port.

## Polling

* Protocol for interaction between host and controller intricate, but basic handshaking notion is simple.
* Say we have a 2 bits to coordinate the producer-consumer relationship between the host and controller:
  + one **busy bit** for status register to indicate status of I/O device
  + one **command-ready** bit in the command register
* Here is an example of the host writing an output through a port and coordinating with the controller.

1. Host repeatedly reads busy bit until cleared
2. Host sets write bit in command register and writes a byte into data-out register
3. Host sets command-ready bit
4. Controller notices set command-ready bit, and sets busy bit
5. Controller reads command register and sees **write** command. Then reads data-out register to get byte and does I/O to device
6. Controller clears command-ready bit, clears error bit it signals successful I/O, and clears busy bit.

* Step 1 is also known as **busy-waiting** or **polling**. It is a loop checking the status register.
* While this works for fast-responding devices, but polling can be waste time if it is attempted repeatedly but rarely finds a device ready for service, when there are more important tasks at hand.
* Instead we can do the opposite: have the device **interrupt** the host to notify it that its ready.

## Interrupts

1. Diagram

   Description automatically generatedDevice controller **asserts** a signal on the **interrupt-request line** (**raises** interrupt).
2. CPU detects (**catches**)controller has asserted signal, and:
   1. performs a **save state**
   2. Jump to **Interrupt-Handler routine** (**dispatches**) at fixed address in memory
3. Interrupt handler determines cause of interrupt, performs necessary processing, performs state restore and executes return (**clears**) from interrupt instruction

* This procedure enables CPU to respond to asynchronous event. For modern Oss, more sophisticated features necessary
* This is managed by **interrupt-controller hardware**, which takes care of the following:
  + deferring interrupts during critical processing
  + Dispatching to interrupt handler for device without polling all to find raising device.
  + Inputs hierarchy to assign urgency
  + Faults and errors (traps) separate from I/O requests
* Two types of interrupt line:
  + **Non-maskable**: reserved for urgent events e.g. unrecoverable memory errors
  + **Maskable**: for device controllers to service request. Can be turned off by CPU during critical processing
* Interrupts accept an **address** requesting specific interrupt-handling routine. This reduces the need for a single handler to work through all possible sources of list to find correct one.
* Modern OSs have more devices than address space, so **interrupt chaining** is used.
  + This is where each element in the interrupt vector points to the head of a list of interrupt handlers, which is run through before finding correct handler.
  + This compromises huge interrupt table and inefficiency of single interrupt handler.
* Interrupt handler also manages **exceptions**. These are processes that induce the OS to execute an urgent, self-contained routine.
* Also given resource and time constraints, modern OSs use two handlers for multiple interrupts:
  + **First-hand Interrupt Handler** (**FLIH**) for context switch, state storage and queueing of handling operation
  + **Second-level Interrupt Handler** (**SLIH**): Performs handling of requested operation
* Interrupts also used for :
  + Virtual memory paging
  + Page fault exceptions
  + System calls (via **software interrupts** or **trap**)
  + Managing flow of control within kernel operations e.g. reading from disk
* To summarise, interrupts are used to trigger kernel routines, and interrupt driving I/O is very popular. It will switch to polling if I/O rate is high

## Direct Memory Access

* **DMA** used for large transfers involving watching the status bit and feeding data into the control register one byte at a time.
  + **DMA Command Block** written to memory to initiate. Contains:
    - Pointers to source and destination of transfer
    - No of bytes to be transferred
  + This can be contiguous and complex, containing multiple transfers from a single DMA command. Known as a **scatter-gather method**
* DMA controller operates memory bus directly with help from the CPU when DMA command initiated. Standard for most computers.
* Most straightforward for target to be in kernel address, as user space data can be changed during transfer. This would result in **double buffering**, pulling back what was transferred.
* Handshaking between DMA and device controllers performed via **DMA-request** and **DMA-acknowledge** wires.

1. Signal asserted on DMA-request wire when word is ready for transfer
2. DMA controller seizes memory bus, places desired address on memory bus wire and places signal on DMA-acknowledge wire.
3. DMA acknowledge received by device controller and transfers word of data into memory. DMA-request signal removed.
4. DMA controller interrupts CPU when transfer is finished

* This process involves **cycle stealing**, where the CPU is denied access to main memory.
* This slows down CPU computation but improves overall system performance.
* **Direct Virtual memory access** (**DVMA**) sometimes used where virtual addresses that undergo translation to physical ones are used.
  + This makes CPU intervention and main memory use unnecessary.
* On protected-mode kernels, the OS exports functions to allow privileged processes to access low-level operations. Without this layer of protection, processes can access device controllers directly.

## Summary

* Main components:
  + Bus
  + Controller
  + An I/O port and registers
  + Handshaking relationship between host and device controller
  + Execution of handshaking via polling or interrupts
  + Offloading work to DMA controllers for large transfer.

# Table Description automatically generatedApplication I/O Interface

* Device driver layer: hides differences among device controllers from the I/O subsystem
* Separate I/O subsystem makes OS developer’s job easier
* Either:
  + design new devices to be compatible with existing host controller interface
  + Write device drivers to interface new hardware to popular OSs
* This means support code from OS developer not necessary to attach peripheral.
* Devices can vary in the following ways:
  + **Data-transfer mode**: **Character Stream** or **block**
  + **Access method**: **Sequential** or **random access**
    - Host can either access data sequentially in order determined by device, or access any available data storage location
  + **Transfer Schedule**: **Synchronous** or **Asynchronous**
    - Synchronous: performs data transfer with predictable response times, coordinated with host.
    - Asynchronous: Irregular, unpredictable response times, not coordinates with other computer events.
  + **Sharing**: **Shared** or **Dedicated**:
    - Sharable device used concurrently by several processes or threads.
    - Dedicated cannot
  + **Device Speed** of Operation: varying from few bytes per second to gigabytes per second
  + **I/O Direction**: **Read-write**, **read only**, **write once**: Differences between performing input and output, one data transfer direction or only modifiable once.
* Devices are grouped to conventional types by OS. The major access conventions include:
  + block I/O
  + Character Stream I/O
  + Memory-mapped file access
  + Network Sockets
* Most OSs have **escape** (**backdoor**) for passing arbitrary commands to device driver.

## Block and Character Devices

* Captures all aspects necessary for accessing disk drives and other block-oriented devices.
* Device must understand read(), write() (and seek() if random access and not sequential).
* These instructions capture essential behaviours of block-storage devices, insulating applications from low-level differences among devices.
* **Raw I/O**: when applications or OS want to access block device as simple linear array of blocks
* To stop the OS and application from conflicting while accessing the device (either from buffering or locking blocks), the application is given full control of the device. This means no OS services are performed on the device.
* Another compromise, **direct I/O**, uses a mode of operation on a file, disabling buffering and locking
* Alternatively, memory mapped file access layered on top of block-device drives instead of read/write operations.
* Memory-mapped interface offers access to disk storage via array of bytes in main memory
* Syscall that maps file into memory returns virtual memory address containing copy of file
* Actual data transfer performed when accessing memory image (copy of virtual address space in file).
* Mechanisms for data transfers are same as ones used for demand-paged virtual memory access, so memory-mapped I/O is efficient.
* **Character stream interface**: example includes keyboard
* Using this interface, an application can get() or put() one character.
* Libraries built on top of interface that offer line-at-a-time access with buffering and editing services e.g. backspace, remove preceding char from input stream.
* Good for unpredictable inputs.

## Network Devices

* Performance and addressing very different between disk and network I/O, so read()-write()-seek() interface does not work
* Instead UNIX and Windows use **network socket** interface
* System calls in socket interface allow applications to create a socket to connect to a remote address.
* The application can then listen for a remote application to “plug into” the local socket to send and receive packet over the connection
* Comes with select() command which eliminates necessity for polling and busy waiting. It lists:
  + Sockets that have a packet that’s waiting to be received
  + Sockets that have room to accept a packet to be sent
* Windows provides interface to network

## Clocks and Timers

* Most computers have hardware clocks and timers. Three main functions:
  + Gives current time
  + Gives elapsed time
  + Sets a timer to trigger operation at time
* These functions are used widely but are **not standardised** across all systems
* Hardware that measures elapsed time and triggers operations: **programmable** **interval timer:**
  + Can generate interrupt after certain amount of time
  + Can be set to do this once or repeat process for periodic interrupts
* Different OS functionalities use this, including:
  + Scheduler to generate interrupt pre-empting a process at end of time slice.
  + Disk I/O subsystem: to invoke periodic flushing of dirty cache buffers (modified but not written to disk) to disk.
  + Network subsystem: cancel operations proceeding too slowly because network congestion or failures
* OS can also:
  + provide interface for user processes to use clocks
  + simulate virtual clocks when there are more requests than channels
    - Accomplished by sorting interrupts in ascending order of time
    - Each time the timer interrupts, the kernel signals the requester and reloads timer with next earliest time.
* **High-performance event timer** (**HPET**) used by most PCs for processes.
* When HPET trigger generates interrupt, OSs clock management systems determine what it was for and what action to take
* Precision limited by:
  + Resolution of timer
  + Overhead of maintaining virtual clocks
* If timer ticks are used to maintain system time-of-day clock, system clock can drift.
* This is remedied by protocols such as **network time protocols**, using sophisticated latency calculations to keep clock accurate to atomic-clock levels

## Nonblocking and Asynchronous I/O

* System calls can be **blocking**, **non-blocking** and **asynchronous**
* **Blocking** syscalls does not return control to application until I/O is complete move calling thread to wait queue, and continue it when it is eligible to resume execution.
* **Nonblocking** returns immediately with data gathered from I/O
* **Asynchronous** syscalls trigger I/O without waiting for completion. Thread continues to execute code and completion of I/O is indicated via some variable in address space or signal/software interrupt

## Vectored I/O

* Allows one system call to perform multiple I/O operations using multiple locations
* Can be more efficient e.g. transferring content of multiple buffers, avoiding context switching and syscall overhead.
* Scatter-gather method

# Kernel I/O Subsystem

## I/O Scheduling

* Essence of I/O scheduling is minimising overhead of hardware movement and optimising efficiency via ordering tasks.
* Every time application issues blocking I/O syscall, scheduler rearranges queue order to improve system efficiency and average response time for applications.
* Scheduler attempts to be fair by:
  + ensuring no application receives especially poor service
  + giving priority service for delay-sensitive requests e.g. requests from virtual memory.
* For **asynchronous** I/O, OS attached wait queue to a **device-status table**. Kernel manages table.
* Each table entry indicates device’s type, address and state.
* Diagram

  Description automatically generatedIf device is busy with request, table entry stored for request and parameters

## Buffering

* **Buffer**: memory area storing data between either two devices, or a device and an application.
* Three reasons for buffering:
  + Cope with speed mismatch between producer and consumer of data stream
    - e.g. downloading from internet, drive much faster than network speed, so buffer is made to accumulate bytes from network
    - **Double buffering** can be used, so while full buffer transfers to disk, the other fills via network data.
    - This decouples producer from consumer, relaxing time requirements between them.
  + Provide adaptations for devices with different data-transfer sizes.
    - e.g. receiving fragmented message and placing in reassembly buffer to get image of source data.
  + Support copy semantics for application I/O
    - **Copy semantics**: when writing data from buffer to disk, version of data is guaranteed to be version at time of application syscall.
    - Simple way of guaranteeing copy semantics is for write() syscall to copy application data into a kernel buffer before returning control to application. Disk write then performed at kernel buffer, so subsequent application changes have no effect.
    - Common despite increased overhead because of clean semantics.

## Caching

* **Cache**: region of fast memory holding copies of data.
* Cache holds a copy on faster storage of an item residing elsewhere, whereas buffers hold existing copy of data item.
* Buffers also used in cache to improve I/O efficiency for files that are shared by applications or being written to and reread rapidly.
* Disk writes also accumulated in buffer cache for seconds so large transfers are gathered to allow efficient write schedules.

## Spooling and Device Reservation

* **Spool**: buffer that holds output for device
* Allows concurrent applications demanding the device can be ordered into a linear fashion i.e. it coordinates concurrent outputs.
* Works for devices like printers

## Error Handling

* Using protected memory to guard against hardware and application errors, such that complete system failure results from major mechanical malfunction.
* SCSI device uses SCSI protocol to report errors in 3 levels:
  + **sense key**: identifies general nature of failure
  + **additional sense code**: states category of failure
  + **additional sense-code qualifier**: gives more detail e.g. which command parameter was wrong, or which hardware subsystem failed its self test

## I/O Protection

* All I/O instructions are privileged, to avoid user processes disrupting the normal operation of the system.
* I/O instructions hence executed through the OS via syscalls.
* OS in **monitor** (system/kernel) **mode** checks if request is valid and executes appropriately
* OS must also protect memory-mapped and I/O port memory locations using memory protection system.
* If certain user processes need direct access to I/O, kernel provides locking mechanism, allowing section of graphics memory allocated to one process at a time i.e. assigning memory.
  + E.g. games or video-editing software needing direct access to memory-mapped graphics)

## Kernel Data Structures

* Kernel must keep state information about use of I/O components

## Power Management

* OSs play a role in power use, specifically heat generation and cooling
* By use of monitoring and management tools, hardware that is unnecessary for system load can be powered off until required.
  + **System load**: measure of amount of computational work a computer system performs
* CPU cores can be suspended when system load does not require them, and resumed when load increases.
  + Their state is saved on suspend and restored on resume
  + Necessary as servers use multiple cores, so disabling them is necessary to avoid wasted energy.
* Extremely important for mobile devices, to increase longevity of battery life. There are three key main features for maximising battery life:
  + **Power Collapse**: the ability to put a device into a very deep sleep state.
    - Idea is that computer uses only a little more power as if it were turned off, but responds to external stimuli instantly.
    - Achieved by powering off many individual components within device so they consume no power, and places CPU in lowest sleep state
  + **Component level power management**: infrastructure understanding how components relate and whether each is in use
    - Android achieve this by using device tree to create hierarchy.
    - Device drivers track usage of components, accounted for by OS.
  + **Wakelocks**: functionality allowing applications to temporarily prevent system from entering power collapse
    - e.g. updating application or watching video
* At boot time, firmware system analyses system hardware and builds device tree in RAM
* Kernel uses tree to load device drivers and manage devices.
* To manage power, changing state devices etc. modern computers use **advanced configuration and power interface** (**ACPI**)
  + Provides code that runs as routines, callable by kernel for:
    - device state recovery and management
    - Error management
    - Power management
  + e.g. when kernel needs to quiesce (silence) device, it calls device driver, which calls ACPI routines, which then talk to the device.

## Summary

* I/O subsystem coordinates extensive collection of services, supervising these procedures:
  + Management of name space for files and devices
  + Access control to files and devices
  + Operational control (e.g. modem cannot seek())
  + File-system space allocation
  + Device allocation
  + Buffering, caching and spooling
  + I/O scheduling
  + Device-status monitoring, event handling and failure recovery
  + Device-driver configuration and initialisation
  + Power management of I/O devices
* Upper levels of I/O subsystem access devices via uniform interfaces provided by device drivers.

# Transforming I/O request to Hardware Operations

* How does the OS connect an application request to a set of network wires of to a specific disk sector?
* E.g. for obtaining a file, applications use filename which is used to obtain the space allocation for the file.
* How is connection made from file name to disk controller (hardware port address or memory-mapped control registers)?
* **MS-DOS for FAT** method can be used.
  + C: first part of every filename on primary hard disk. This is built into the OS and maps to a specific port address via the device table
  + Colon separator means device name space is separate from file-system name space.
  + Makes it easy for OS to associate extra functionality for each device e.g. invoking spooling on any files written to printer.
* Alternatively device name space can be incorporated into regular file-system name space
  + Devices would have owners and access control
  + Interface providing access to I/O system split on two levels:
    - names used to access device themselves
    - names to access files stored on device
  + UNIX does this: their default path names do not include any clear separation of the device portion desired
    - UNIX uses **mount table** associating prefixes of path names with specific device names
    - To resolve path names, UNIX looks up name in mount table to find longest matching prefix, and corresponding entry gives device name.
    - When this name is looked up in file-system directory structures, two <major,minor> names are given
      * major corresponds to device driver, called to handle I/O device
      * minor is passed to device driver, into device table, and corresponding entry gives port address or memory-mapped address of device controller.
* This functionality allows the introduction of new devices and drivers without recompiling kernel.
* Devices added after boot can be detected by error they cause (e.g. interrupt-generated with no associated interrupt handler) which prompts kernel to inspect device details and load appropriate device drivers dynamically
* Requires more complex kernel algorithms, device-structure locking, error handling etc.
* Typical life cycle of a blocking read request:

Diagram

Description automatically generated

1. Process issues a blocking read() syscall to file descriptor of file opened previously
2. Syscall code in kernel checks parameters for correctness. If the device is input and the data is in buffer cache, the data is returned to process and request complete
3. Otherwise physical I/O performed. Process removed from run queue and placed on wait queue for the device, and I/O request is scheduled. Eventually I/O subsystem sends request to device driver. Depending on OS request is sent via subroutine call or an in-kernel message
4. Device driver allocates kernel buffer space to receive the data and schedule the I/O. Eventually driver sends commands to the device controller by writing into device-control registers.
5. Device controller operates device hardware to perform data transfer
6. Drive may poll for status and data, or have set up a DMA transfer into kernel memory. We assume transfer is managed by DMA controller, which generates interrupt when transfer complete.
7. Correct interrupt handler receives interrupt via interrupt vector table, stores any necessary data, signals device driver and returns from interrupt.
8. Device driver receives signal, determines which I/O request has completed, determined request status and signals kernel I/O that request has been completed.
9. Kernel transfers data or return codes to address space of requesting process, and moves process from wait back to ready queue.
10. Moving the process to ready queue unblocks process. When the scheduler assigns the process to the CPU, the process resumes execution at completion of syscall.