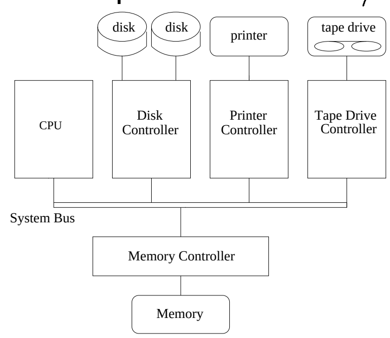
Today: I/O Systems

- How does I/O hardware influence the OS?
- What I/O services does the OS provide?
- How does the OS implement those services?
- How can the OS improve the performance of I/O?



Architecture of I/O Systems

- Key components
  - **System bus:** allows the device to communicate with the CPU, typically shared by multiple devices.
  - A device **port** typically consisting of 4 registers:
    - **Status** indicates a device busy, data ready, or error condition
    - **Control:** command to perform
    - **Data-in:** data being sent from the device to the CPU
    - **Data-out:** data being sent from the CPU to the device
  - **Controller:** receives commands from the system bus, translates them into device actions, and reads/writes data onto the system bus.
  - The device itself
- Traditional devices: disk drive, printer, keyboard, modem, mouse, display
- Non-traditional devices: joystick, robot actuators, flying surfaces of an airplane, fuel injection system of a car, ...

I/O Services Provided by OS

- Naming of files and devices. (On Unix, devices appear as files in the /dev directory)
- Access control.
- Operations appropriate to the files and devices.
- Device allocation.
- Buffering, caching, and spooling to allow efficient communication with devices.
- I/O scheduling.
- Error handling and failure recovery associated with devices (command retries, for example).
- Device drivers to implement device-specific behaviors.

Communication using Polling

- CPU busy-waits until the status is idle.
- CPU sets the command register and data-out if it is an output operation.
- CPU sets status to command-ready $\Rightarrow$controller sets satus to busy
- Controller reads the command register and performs the command, placing a value in data-in if it is an input command.
- If the operation succeeds, the controller changes the status to idle.
- CPU observes the change to idle and reads the data if it was an input operation.
- Good choice if data must be handled promptly, like for a modem or keyboard
- What happens if the device is slow compared to the CPU?

Communication using Interrupts

- Rather than using busy waiting. the device can interrupt the CPU when it completes an I/O operation.
- On an I/O interrupt:
  - Determine which device caused the interrupt.
  - If the last command was an input operation, retrieve the data from the device register.
  - Start the next operation for that device.

Direct Memory Access

- For devices that transfer large volumes of data at a time (like a disk block), it is expensive to have the CPU retrieve these one byte at a time.
- **Solution:** Direct memory access (DMA)
  - Use a sophisticated DMA controller that can write directly to memory. Instead of data-in/data-out registers, it has an address register.
  - The CPU tells the DMA the locations of the source and destination of the transfer.
  - The DMA controller operates the bus and interrupts the CPU when the entire transfer is complete, instead of when each byte is ready.
  - The DMA controller and the CPU compete for the memory bus, slowing down the CPU somewhat, but still providing better performance than if the CPU had to do the transfer itself.

Application Programmer's View of I/O Devices

- The OS provides a high-level interface to devices, greatly simplifying the programmer's job.
  - Standard interfaces are provided for related devices.
  - Device dependencies are encapsulated in device drivers.
  - New devices can be supported by providing a new device driver.
- **Device characteristics:**
  - Transfer unit: character or block
  - Access method: sequential or random access
  - Timing: synchronous or asynchronous.
    - * Most devices are asynchronous, while I/O system calls are synchronous.
    - ⇒ The OS implements *blocking I/O*
  - Sharable or dedicated
  - Speed
  - Operations: Input, output, or both
  - Examples: keyboard (sequential, character), disk (block, random or sequential)

I/O Buffering

I/O devices typically contain a small on-board memory where they can store data temporarily before transferring to/from the CPU.

- A disk buffer stores a block when it is read from the disk.
- It is transferred over the bus by the DMA controller into a buffer in physical memory.
- The DMA controller interrupts the CPU when the transfer is done.

Why buffer on the OS side?

- To cope with speed mismatches between device and CPU.
  - Example: Compute the contents of a display in a buffer (slow) and then zap the buffer to the screen (fast)
- To cope with devices that have different data transfer sizes.
  - Example: ftp brings the file over the network one packet at a time. Stores to disk happen one block at a time.
- To minimize the time a user process is blocked on a write.
  - Writes $\Rightarrow$copy data to a kernel buffer and return control to the user program. The write from the kernel buffer to the disk is done later.

Caching

- Improve disk performance by reducing the number of disk accesses.
  - **Idea:** keep recently used disk blocks in main memory after the I/O call that brought them into memory completes.
  - **Example:** Read (diskAddress)
If (block in memory) return value from memory
Else ReadSector(diskAddress)
  - **Example:** Write (diskAddress)
If (block in memory) update value in memory
Else Allocate space in memory, read block from disk, and update value in memory
- What should happen when we write to a cache?
  - write-through policy (write to all levels of memory containing the block, including to disk). High reliability.
  - write-back policy (write only to the fastest memory containing the block, write to slower memories and disk sometime later). Faster.

Putting the Pieces Together - a Typical Read Call

1. User process requests a read from a device.
2. OS checks if data is in a buffer. If not,
   (a) OS tells the device driver to perform input.
   (b) Device driver tells the DMA controller what to do and blocks itself.
   (c) DMA controller transfers the data to the kernel buffer when it has all been retrieved from the device.
   (d) DMA controller interrupts the CPU when the transfer is complete.
3. OS transfers the data to the user process and places the process in the ready queue.
4. When the process gets the CPU, it begins execution following the system call.

Summary

- I/O is expensive for several reasons:
  - Slow devices and slow communication links
  - Contention from multiple processes.
  - I/O is typically supported via system calls and interrupt handling, which are slow.
- Approaches to improving performance:
  - Reduce data copying by caching in memory
  - Reduce interrupt frequency by using large data transfers
  - Offload computation from the main CPU by using DMA controllers.
  - Increase the number of devices to reduce contention for a single device and thereby improve CPU utilization.
  - Increase physical memory to reduce amount of time paging and thereby improve CPU utilization.