Agenda

- I/O architecture overview
- Basic I/O device model
- I/O protocol variants
- E.g., x86/xv6 IDE disk driver
§ I/O architecture overview
The last piece: I/O

- Computers are arguably useless without I/O!
- Vast number of different types of I/O devices across many different categories
  - Keyboards, displays, printers, network cards, …
- Need an abstract interface that can accommodate different types and combinations of devices
- API & communication protocol
I/O architecture

- CPU
- RAM
- Graphics

memory bus

general I/O bus (e.g., PCI)

peripheral I/O bus (e.g., SCSI, SATA, USB)
Basic I/O device model
Basic I/O device

- Two parts: API + implementation
  - Minimal API:
    - Status, Data, and Command registers
  - Implementation:
    - Microcontroller + RAM
    - Firmware (ROM / EEPROM)
    - Other special purpose hardware
Basic protocol

while (Status == Busy)
    ; // spin
Write data to Data register
Write command to Command register
while (Status == Busy)
    ; // spin
Basic protocol

#1: while (\textbf{Status} == \textbf{Busy})
    ; // spin
#2: Write data to \textbf{Data} register
#3: Write command to \textbf{Command} register
#4: while (\textbf{Status} == \textbf{Busy})
    ; // spin

process A initiates protocol

\begin{center}
\begin{tikzpicture}
    \node (CPU) at (0,0) {CPU};
    \node (Device) at (1,0) {Device};
    \node (A) at (0.5,0.5) {A};
    \path[draw,->] (CPU) edge node[above] {A} (Device);
\end{tikzpicture}
\end{center}
Basic protocol

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin
Basic protocol

#1: while (Status == Busy)
     ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
     ; // spin

CPU    #1 #2

Device A A
Basic protocol

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin

CPU

Device
Basic protocol

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin
Basic protocol

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin

I/O complete
Basic protocol

#1: while (√Status == Busy) 
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (√Status == Busy) 
    ; // spin

Questions/Issues?
Basic protocol issues

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin

Steps #1 & #4 burn CPU time just polling the status register …
Basic protocol issues

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin

Step #2 forces the CPU to copy data between buffers (very tedious!)
Basic protocol issues

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin

How to access status, data, command registers?
§ I/O protocol variants
Polled vs. Interrupt-driven I/O

#1: while (Status == Busy)
    ; // spin
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    ; // spin

Instead of polling, device can notify CPU via interrupt when status has changed
Polled vs. Interrupt-driven I/O

#1: \textbf{while} (\texttt{Status} == \texttt{Busy})
  
  Wait for interrupt;

#2: Write data to \texttt{Data} register

#3: Write command to \texttt{Command} register

#4: \textbf{while} (\texttt{Status} == \texttt{Busy})
  
  Wait for interrupt;

\begin{center}
\begin{tikzpicture}
  \node (cpu) at (0,0) {CPU};
  \node (device) at (0,-1) {Device};
  \draw[->] (cpu) -- (device) node[midway,above] {A};
\end{tikzpicture}
\end{center}
Polled vs. Interrupt-driven I/O

#1: while (Status == Busy)
    Wait for interrupt;

#2: Write data to Data register

#3: Write command to Command register

#4: while (Status == Busy)
    Wait for interrupt;

---

![Diagram](image)
Polled vs. Interrupt-driven I/O

#1: while (\textbf{Status} == \textbf{Busy})
  Wait for interrupt;
#2: Write data to \textbf{Data} register
#3: Write command to \textbf{Command} register
#4: while (\textbf{Status} == \textbf{Busy})
  Wait for interrupt;
Polled vs. Interrupt-driven I/O

#1: while \( \text{Status} == \text{Busy} \)
   Wait for interrupt;
#2: Write data to \text{Data} register
#3: Write command to \text{Command} register
#4: while \( \text{Status} == \text{Busy} \)
   Wait for interrupt;

CPU

Device

interrupt

interrupt
Polled vs. Interrupt-driven I/O

#1: while (Status == Busy)
   Wait for interrupt;
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
   Wait for interrupt;

![Diagram showing CPU and Device operations]

- CPU: A B A B A B
- Device: A
  - #1: CPU starts
  - #2: CPU writes data to Data register
  - #3: CPU writes command to Command register
  - #4: CPU waits for interrupt
  - Interrupt signal

#1: while (Status == Busy)
   Wait for interrupt;
Polled vs. Interrupt-driven I/O

caveat: context switch time not accounted for!
Polled vs. Interrupt-driven I/O

- Interrupts are not always better!
  - Fast devices can lead to very frequent interrupts — flood CPU with interrupt handlers (livelock)
  - May use a hybrid polled & interrupt-driven approach
    - Also: interrupt coalescing
    - Compromise between system overhead and responsiveness
Programmed I/O vs. Direct Memory Access

1: while (Status == Busy)
   Wait for interrupt;
2: Write data to Data register
3: Write command to Command register
4: while (Status == Busy)
   Wait for interrupt;

May be able to have device copy data from RAM directly (CPU just provides address as part of command)
Programmed I/O vs. Direct Memory Access

1. While (Status == Busy)
   - Wait for interrupt;
2. Write data to Data register
3. Write command to Command register
4. While (Status == Busy)
   - Wait for interrupt;

Diagram:
- CPU: A B A B A B
- Device: A

Interrupts: 1 3 4
Polled + PIO vs. Interrupts + DMA

Polled + PIO

Interrupts + DMA
I/O programming API

#1: while (Status == Busy)
    Wait for interrupt;
#2: Write data to Data register
#3: Write command to Command register
#4: while (Status == Busy)
    Wait for interrupt;

How to access status, data, command registers?

- Two approaches: (1) special instructions & (2) memory-mapped I/O
  - (1) Special assembly instructions that address device-specific “ports”
  - (2) Hardware registers map to special address ranges
Protocol variants summary

- Checking hardware status: **polling** vs. **interrupts**
- Data transfer: programmed I/O (**PIO**) vs. direct memory access (**DMA**)
- Control API: **special instructions** vs. **memory-mapped I/O**
- To encapsulate and simplify access to hardware with different protocol variations, we write separate **device drivers**
  - Also allows us to mix and match devices in different OS modules
E.g., x86/xv6 IDE disk driver
xv6 idewait routine

// wait for hard disk to be ready
static int idewait() {
    int r;
    while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY) ;
    return 0;
}

- Busy loop polling for disk to be ready
- Note special x86 “in” & “out” instructions
- 0x1f7 = IDE status register
// add new request to queue
void iderw(struct buf *b) {
    acquire(&idelock);

    struct buf **pp;
    for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
    *pp = b; // add new request to end

    if(idequeue == b) // if queue was empty
        idestart(b); // send request to disk

    while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
        sleep(b, &idelock); // block until complete
    }
    release(&idelock);
}

- Manages a queue of disk buffer requests (reads/writes)
- If empty, starts the new request, else queues it
- Uses sleep to block
// Start hard disk request
static void idestart(struct buf *b) {
    idewait();
    outb(0x3f6, 0); // enable interrupt
    outb(0x1f2, sector_per_block); // number of sectors
    outb(0x1f3, sector & 0xff); // logical block address
    outb(0x1f4, (sector >> 8) & 0xff); // ...
    outb(0x1f5, (sector >> 16) & 0xff); // ...
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
    if(b->flags & B_DIRTY){
        outb(0x1f7, write_cmd); // write command
        outsl(0x1f0, b->data, BSIZE/4);
    } else {
        outb(0x1f7, read_cmd); // read command
    }
}  

- Sends command details to device (when ready)
- Writes data (PIO) if needed
- Interrupt is delivered when operation completes (note: not DMA!)
xv6 ideintr routine

// Hard disk interrupt handler
void ideintr(void) {
    struct buf *b;
    acquire(&idelock);

    // Read data if needed.
    if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
        insl(0x1f0, b->data, BSIZE/4);

    // Wake process waiting for this buf.
    b->flags |= B_VALIDATE;
    b->flags &= ~B_DIRTY;
    wakeup(b);

    if(idequeue != 0)
        idestart(idequeue); // start next request

    release(&idelock);
}

- If read request, load data from device to buffer
- Wake up blocked process
- Queue up next request