I/O

Anne Bracy

CS 3410

Computer Science

Cornell University

[K. Bala, A. Bracy, S. McKee, E. Sirer, and H. Weatherspoon]
Big Picture: Input/Output (I/O)

How does a processor interact with its environment?

Computer System =  
Memory + Datapath + Control + Input + Output

- Keyboard
- Network
- Display
- Disk
I/O connected with I/O Controllers

**high-performance interconnect:** processor, memory, display

**lower-performance interconnect:** disk, keyboard, network
Bus Types

Processor – Memory ("Front Side Bus")

- Short, fast, & wide
- Mostly fixed topology, designed as a “chipset”
  - CPU + Caches + Interconnect + Memory Controller

I/O and Peripheral busses (PCI, SCSI, ...)

- Longer, slower, & narrower
- Flexible topology, multiple/varied connections
- Interoperability standards for devices
- Connect to processor-memory bus through a bridge
I/O Device API

Typical I/O Device API

• a set of read-only or read/write registers

Command registers

• writing causes device to do something

Status registers

• reading indicates what device is doing, error codes, ...

Data registers

• Write: transfer data to a device
• Read: transfer data from a device

Every device uses this API
1. Programmed I/O:
   special instructions talk over special busses
   Specify: device, data, direction
   • inb $a, 0x64 (keyboard status register)
   • outb $a, 0x60 (keyboard data register)
   • Protection: only allowed in kernel mode (expensive)

2. Memory-Mapped I/O:
   map registers into virtual address space
   • Accesses to certain addresses redirected to I/O devices
   • Data goes over the memory bus (faster!)
   • Protection: via bits in pagetable entries
   • OS+MMU+devices configure mappings
Memory-Mapped I/O

Virtual Address Space

Physical Address Space

Agreed-upon locations for communication

vs. less-favored alternative = Programmed I/O:
  • Syscall instructions that communicate with I/O
  • Communicate via special device registers

I/O Controller

Display

Disk

Keyboard

Network

Partitioned I/O

Controller

0xFFFF FFFF

0x00FF FFFF

0x0000 0000

0x0000 0000

Programmed I/O

```c
char read_kbd()
{
    do {
        sleep();
        status = inb(0x64);
    } while(!(status & 1));

    return inb(0x60);
}
```

Memory Mapped I/O

```c
struct kbd {
    char status, pad[3];
    char data, pad[3];
};

kbd *k = mmap(...);

char read_kbd()
{
    do {
        sleep();
        status = k->status;
    } while(!(status & 1));

    return k->data;
}
```

**Clicker Question:** Which is better?

(A) Programmed I/O  
(B) Memory Mapped I/O  
(C) Both have syscalls, both are bad
I/O Data Transfer

How to talk to device?
  • Programmed I/O or Memory-Mapped I/O

How to get events?
  • Polling or Interrupts

How to transfer lots of data?

```c
disk->cmd = READ_4K_SECTOR;
disk->data = 12;
while (!(disk->status & 1)) {}
for (i = 0..4k)
    buf[i] = disk->data;
```

Very, Very, Expensive
1. Programmed: Device $\longleftrightarrow$ CPU $\longleftrightarrow$ RAM Transfer

for $(i = 1 \ldots n)$
- CPU issues read request
- Device puts data on bus & CPU reads into registers
- CPU writes data to memory

2. Direct Memory Access (DMA): Device $\longleftrightarrow$ RAM

- CPU sets up DMA request
- for $(i = 1 \ldots n)$
  Device puts data on bus & RAM accepts it
- Device interrupts CPU after done
Programmed I/O vs Memory Mapped I/O

Programmed I/O
- Requires special instructions
- Can require dedicated hardware interface to devices
- Protection enforced via kernel mode access to instructions
- Virtualization can be difficult

Memory-Mapped I/O
- Re-uses standard load/store instructions
- Re-uses standard memory hardware interface
- Protection enforced with normal memory protection scheme
- Virtualization enabled with normal memory virtualization scheme
Polling vs. Interrupts

How does program learn device is ready/done?

1. **Polling**: Periodically check I/O status register
   - Common in small, cheap, or real-time embedded systems
   - Predictable timing, inexpensive
   - Wastes CPU cycles

2. **Interrupts**: Device sends interrupt to CPU
   - Cause register identifies the interrupting device
   - Interrupt handler examines device, decides what to do
   - Only interrupt when device ready/done
   - Forced to save CPU context (PC, SP, registers, *etc.*)
   - Unpredictable, event arrival depends on other devices’ activity

Clicker Question: Which is better?
(A) Polling  (B) Interrupts  (C) Both equally good/bad