I/O Devices

Nima Honarmand
(Based on slides by Prof. Andrea Arpaci-Dusseau)
Hardware Support for I/O

- CPU
- RAM
- Network Card
- Graphics Card
- Memory Bus
- General I/O Bus (e.g., PCI)
Canonical Device

- OS communicates with the device by reading/writing to **Device Registers**
  - Don’t think of them as storage locations like CPU registers; they are communication interfaces

- Internal device hardware interprets these reads/writes in a device-specific way
Example Write Protocol

Device Registers:

Hidden Internals:

while (STATUS == BUSY) // 1
;

Write data to DATA register // 2

Write command to COMMAND register // 3

while (STATUS == BUSY) // 4
;
A wants to do I/O

CPU: A

Disk: C

while (STATUS == BUSY)  // 1
    ;

Write data to DATA register  // 2

Write command to COMMAND register  // 3

while (STATUS == BUSY)  // 4
    ;
while (STATUS == BUSY) // 1
;
Write data to DATA register // 2
Write command to COMMAND register // 3
while (STATUS == BUSY) // 4
;

How to avoid wasting CPU time with **polling**?
Use Interrupts instead of Polling

while (STATUS == BUSY)           // 1
    context switch and wait for interrupt;

Write data to DATA register      // 2

Write command to COMMAND register // 3

while (STATUS == BUSY)           // 4
    context switch and wait for interrupt;
while (STATUS == BUSY)  // 1
    context switch and wait for interrupt;

Write data to DATA register  // 2

Write command to COMMAND register  // 3

while (STATUS == BUSY)  // 4
    context switch and wait for interrupt;
Interrupts vs. Polling

• Are interrupts ever worse than polling?
  • Fast device: Better to spin than take interrupt overhead
  • Device time unknown? Hybrid approach (spin then use interrupts)

• Flood of interrupts arrive
  • Can lead to livelock (always handling interrupts)
  • Better to ignore interrupts while make some progress handling them

• Other improvement
  • Interrupt coalescing (batch together several interrupts)
# Protocol Variants

Device Registers:

<table>
<thead>
<tr>
<th>Status</th>
<th>COMMAND</th>
<th>DATA</th>
</tr>
</thead>
</table>

Hidden Internals:

- Microcontroller (CPU+RAM)
- Extra RAM
- Other special-purpose chips

- **Status check**: polling vs. interrupt
- **Transferring data**: Programmed IO (PIO) vs. DMA
while (STATUS == BUSY)    // 1
  context switch and wait for interrupt;

Write data to DATA register    // 2

Write command to COMMAND register    // 3

while (STATUS == BUSY)    // 4
  context switch and wait for interrupt;

What else can we optimize?
PIO vs. DMA

- *Programmed IO (PIO)*
  - OS code transfers every byte of data to/from device
  → CPU is directly involved with—and burns cycles on—data transfer

- *Direct Memory Access (DMA)*
  - OS prepares a buffer in RAM
    - If writing to device, fills buffer with data to write
    - If reading from device, initial buffer content does not matter
  - OS writes buffer’s physical address and length to device
  - Device reads/writes data directly from/to RAM buffer
  → No wasting of CPU cycles on data transfer
while (STATUS == BUSY) // 1
    context switch and wait for interrupt;

Write data to DATA register // 2

Write command to COMMAND register // 3

while (STATUS == BUSY) // 4
    context switch and wait for interrupt;
Prepare the buffer // 0

while (STATUS == BUSY) // 1
    context switch and wait for interrupt;

Write data to DATA register // 2

Write command to COMMAND register // 3

while (STATUS == BUSY) // 4
    context switch and wait for interrupt;
Protocol Variants

- Status check: polling vs. interrupt
- Transferring data: Programmed IO (PIO) vs. DMA
- Communication: special instructions vs. memory-mapped IO
How OS Reads/Writes Dev. Registers

- **Special instructions**
  - Each device register is assigned a *port number*
  - Special instructions (in and out in x86) communicate read/write ports

- **Memory-Mapped I/O**
  - Each device register is assigned a physical memory
  - Normal memory loads/store instruction (mov in x86) used to access registers

- OSTEP claims does not matter which one you use; I disagree
  - MMIO far better and more flexible
  - Modern devices exclusively use MMIO
xv6 code review

• IDE disk driver in xv6
Protocol Variants

- Status check: polling vs. interrupt
- Transferring data: Programmed IO (PIO) vs. DMA
- Communication: special instructions vs. memory-mapped IO
Variety is a Challenge

• Problem:
  • Many, many devices
  • Each has its own protocol

• How can we avoid writing a slightly different OS for each H/W combination?
  • Extra level of indirection: use a device abstraction

• Keep OS code mostly device-independent
  • Device drivers deal with devices and provide generic interfaces used by the rest of the OS
  • Most of a modern OS source code is its device drivers
    • E.g., drivers are about 70% of Linux source code
Example: Storage Stack

Application

Virtual file system

Concrete file system

Generic block layer

Driver

Disk drive

Build common interface on top of all disk drivers

Different types of drives: HDD, SSD, network mount, USB stick
Different types of interfaces: ATA, SATA, SCSI, USB, NVMe, etc.
A Few Points on MMIO Programming
Memory-Mapped I/O

• MMIO allows you to map device interface to C struct and use it conveniently in C code
  • Subject to side-effect caveats

• Example: MMIO for our canonical device
  • Lets say the three registers are mapped to three consecutive integers in physical address space

```c
typedef struct {
    int status;
    int command;
    int data;
} mydev_interface;

mydev_interface* dev = (mydev_interface*) <dev_addr>;
while (dev->status & D_BUSY);
for (i=0; i<data_len; i++)
    dev->data = data[i];
dev->command = COMMAND;
while (dev->status & D_BUSY);
```
Programming Mem-Mapped IO

• A memory-mapped device is accessed by normal memory ops
  • E.g., the \texttt{mov} family in x86

• But, how does compiler know about I/O?
  • Which regions have side-effects and other constraints?
  • It doesn’t: programmer must specify!
Problem with Optimizations

• Recall: Common optimizations (compiler and CPU)
  • Compilers keep values in registers, eliminate redundant operations, etc.
  • CPUs have caches
  • CPUs do out-of-order execution and re-order instructions

• When reading/writing a device, it should happen immediately
  • Should not keep it in a processor register
  • Should not re-order it (neither compiler nor CPU)
  • Also, should not keep it in processor’s cache

• CPU *and* compiler optimizations must be disabled
volatile Keyword

• **volatile** on a variable means this variable can change value at any time
  • So, do not register allocate it and disable all optimizations on it
  • Send all writes directly to memory
  • Get all reads directly from memory

• **volatile** code blocks are not re-ordered by the compiler
  • Must be executed precisely at this point in program
  • E.g., inline assembly
Fence Operations

• Also known as Memory Barriers

• `volatile` does not force the CPU to execute instructions in order

```c
Write to <device register 1>
mb(); // fence
Read from <device register 2>
```

• Use a `fence` to force in-order execution
  • Linux example: `mb()`
  • Also used to enforce ordering between memory operations in multi-processor systems
Dealing with Caches

• Processor may cache memory locations
  • Whether it’s DRAM or MMIO locations
  • Because the CPU does not know which is which

• Often, memory-mapped I/O should not be cached
  • Why?

• \textit{volatile} does not affect caching
  • Because compilers don’t know about caching

• Solution: OS marks ranges of memory used for MMIO as \textit{non-cacheable}
  • Basically, disable caching for such memory ranges
  • There are PTE flags for this (e.g., PCD flags in x86 PTEs)
Correct Code for Our Example

```c
make_uncacheable(dev_addr);

volatile mydev_interface* dev =
  (volatile mydev_interface*)dev_addr;

while (dev->status & D_BUSY);  
  mb();  
  for (i=0; i<data_len; i++)
    dev->data = data[i];
  mb();
  dev->command = COMMAND;
  mb();
  while (dev->status & D_BUSY);
```

Notes:

1) `make_uncacheable` is a made-up name; each kernel has a different set of functions for this purpose.

2) Some of the `mb()` calls in this code are unnecessary in x86; but better safe than sorry.