CS44110 Lecture 2: HW/SW interface. Devices

- Design of I/O bus
- Programming with I/O
- Multi-programming: Some history

- MMIO vs. Programmed I/O
  - Polling vs. Interrupts
  - DMA
  - Device controllers
  - Processes vs. Programs
  - Process state
dev 0

Buttons (keyboard)

8 bits

Keycode

Logic

dev 1

Mic

CPU

Control

8 bits data

dev id

I/O bus

dev 2

Mouse

to communicate with devices (version 1)
- add new instructions
  eg. get data from kb
  GET IO (or device #)

  send data
  PUT IO to device
Hello World v1 (programmed I/O, polling)

message: "HELLO WORLD"

```
main: PUTIO message screen_dev.id

r1 < 0
while not r1:
    GETIO keyboard_dev.id -> r1
    // key press in r1
```

Polling:
- bad, because CPU can't do useful work while waiting
- might miss input
When an interrupt occurs, jump to code to handle interrupt.

**New instruction:**

```
CONFIGURE: stores address of interrupt handler code.
```
Interrupt vector:

- mapping of devices to addresses of I.H. code.

- `CONFIGURE <device id> <address of code>`, of int. routine

- interrupt controller jumps to appropriate handler on interrupt.

```
<table>
<thead>
<tr>
<th>dev0 (keyboard)</th>
<th>process_keypress</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev1 (mouse)</td>
<td>process_mouse_event</td>
</tr>
</tbody>
</table>
```
Device Controller

- Keyboard needs internal logic
  - Store keypress until int. halr. can fetch
  - Buffer keystrokes, remove "bounce"
  - Convert coordinates to ASCII...

- Hardware/logic inside device: device controller
  - Manage dev. state
  - Communicate w/ CPU, interpret commands
  - Control device operation
  - Can be arbitrarily complex! Modern GPUs, for example, are entire processors.
Hello World v2 (Programmed I/O, Interrupts)

message: "HELLO WORLD"

main:
  PUTIO message screen_dev_id
  SET_INT_HANDLER keyboard_id handler
  while true:
    pass  // (or do other useful work)
  handler:
    GETIO keyboard_id \[
      | \\
    \r
    // keycode m r1.
  ...
Memory-mapped I/O
- one bus for memory & I/O
- use normal load/store
- each device has addr. range.
- Devices don't communicate w/ memory.

Dedicated I/O: bus & instructions add complication

**NEW/MMIO**
- CPU

**OLD/P10**
- CPU

**I/O bus**
- data, control, device ID
  - interrupt

**KB**
- Controlled by copying address to dedicated address range
- has dedicated address range
- OxFFFF - OxFFFF

**Devices**
- Memory
- Ox0000 - 0x0000
- Ox1000 - 0xFFFF
- Ox2000 - 0xFFFFFFFF

Programmed I/O:
- dedicated bus
- special instr.
- more complicated
Hello World v3 (MMIO, interrupts)

Message: "HELLO WORLD"

Main:

```
r1 <- message
r3 <- screen.data_addr
do
  r2 <- *r1
  store r3 <- r2
  r1 <- r1 + 1
  r3 <- r3 + 1
  until r2 == 0
  store screen_control_addr <- screen_print_command
GO TO read
```

Print "HELLO WORLD"

```
load r1 <- kb.data_addr
// r1 has keystroke
```

read:

```
set_int_handler kb_int_id

store kb_control_addr <- kb_input_cmd
while true:
  pass

handler:
  load r1 <- kb_data_addr
```

Blue underlined values are HW-specific constants
DMA (Direct Memory Access)

- Devices communicate with main memory
- Ex: Microphone or webcam or graphics card (lots of data)
- Use dedicated address ranges (as with MMIO) to control device
- Device sends/receives data from memory
- Raises interrupt to signal completion
Hello World v4 (MMIO, DMA, interrupts)

message: "Enter a line & press ENTER"
buffer: "1010 ... 10"

main:
STORE screen_data_addr <- message
STORE screen_ctrl_addr <- Screen_println_cmd
STORE kb_output_ptr <- buffer
SET_INTR_HANDLER kb_dev_id handler
STORE kb_ctrl_addr <- kb readline_cmd
while true:
  pass

handler:
LOAD rl <- kb_status_addr
// input line is in buffer

Screen only needs address of message, can read it itself.

KB can transfer a whole string into a buffer & interrupt when done
Running Hello World v4

Diagram:
- CPU
  - Running task
  - CPU is idle
  - Running handler
- Screen
- KB
  - Collecting/waiting for input
- pr.w1n
- interrupt
Multiprogramming

Store multiple ready to run programs.

process prog. 1

process prog. 2  process prog. 3

either
1. Run prog 1
   IH routine

or 2. Store data to  deliver to
    P1 later.

proc prog 1

proc prog 2

proc prog 3

CPU Dev2 Dev 1

I/O

I/O

I/O

interrupt

go back to prog 3 or to prog 1.
Program: code, global constants.
  i.e. stuff in executable file.

Process: program in motion
  program and
  - local/global variables, registers, stack, heap...

Running a program creates a process
Each process has a state:

- ready (can make progress)
- running
- waiting (for I/O to complete)
- zombie (done, needs cleanup)

One process allocation setup
interrupt (I/O complete)
wait
input
scheduled
exit

P1
ready

P2
running
I/O (waiting)

P3
running
ready
running
ready
Enter the Operating System

- make scheduling decisions
- changing state of CPU / memory to load / unload different processes.
- manage access to devices.
- isolating processes from one another.
- some instructions are disallowed e.g. "update interrupt vector"