CS 61C: Great Ideas in Computer Architecture (Machine Structures)

*Operating Systems, Interrupts, Virtual Memory*

**Instructors:**

John Wawrzynek & Vladimir Stojanovic

http://inst.eecs.berkeley.edu/~cs61c/
CS61C so far...

**CPU**

**Caches**

**Memory**

---

**C Programs**

```c
#include <stdlib.h>
int fib(int n) {
    return fib(n-1) + fib(n-2);
}
```

**MIPS Assembly**

```mips
.lfoo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
```

**Project 1**

**Project 2**

**Labs**
So how is this any different?
Adding I/O

C Programs

MIPS Assembly

#include <stdlib.h>

int fib(int n) {
    return fib(n-1) + fib(n-2);
}

.lfoo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop

Project 1

Project 2

CPU

Caches

Memory

I/O (Input/Output)

Screen

Keyboard

Storage

I/O (Input/Output)
Raspberry Pi ($40 on Amazon)
It’s a real computer!
But wait...

• That’s not the same! When we run MARS, it only executes one program and then stops.

• When I switch on my computer, I get this:

Yes, but that’s just software! The Operating System (OS)
Well, “just software”

- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:

![Codebases diagram](http://www.informationisbeautiful.net/visualizations/million-lines-of-code/)

Codebases (in millions of lines of code). CC BY-NC 3.0 — David McCandless © 2013
What does the OS do?

• One of the first things that runs when your computer starts (right after firmware/bootloader)
• Loads, runs and manages programs:
  – Multiple programs at the same time (time-sharing)
  – Isolate programs from each other (isolation)
  – Multiplex resources between applications (e.g., devices)
• Services: File System, Network stack, etc.
• Finds and controls all the devices in the machine in a general way (using “device drivers”)
Agenda

• Devices and I/O
• OS Boot Sequence and Operation
• Multiprogramming/time-sharing
• Introduction to Virtual Memory
Agenda

- Devices and I/O
- OS Boot Sequence and Operation
- Multiprogramming/time-sharing
- Introduction to Virtual Memory
How to interact with devices?

• Assume a program running on a CPU. How does it interact with the outside world?
• Need I/O interface for Keyboards, Network, Mouse, Screen, etc.
  – Connect to many types of devices
  – Control these devices, respond to them, and transfer data
  – Present them to user programs so they are useful
Instruction Set Architecture for I/O

• What must the processor do for I/O?
  – Input: reads a sequence of bytes
  – Output: writes a sequence of bytes

• Some processors have special input and output instructions

• Alternative model (used by MIPS):
  – Use loads for input, stores for output (in small pieces)
  – Called Memory Mapped Input/Output
  – A portion of the address space dedicated to communication paths to Input or Output devices (no memory there)
Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices
Processor-I/O Speed Mismatch

• 1GHz microprocessor can execute 1B load or store instructions per second, or 4,000,000 KB/s data rate
  • I/O data rates range from 0.01 KB/s to 1,250,000 KB/s
• Input: device may not be ready to send data as fast as the processor loads it
  • Also, might be waiting for human to act
• Output: device not be ready to accept data as fast as processor stores it
• What to do?
Processor Checks Status before Acting

• Path to a device generally has 2 registers:
  • Control Register, says it’s OK to read/write (I/O ready) [think of a flagman on a road]
  • Data Register, contains data

• Processor reads from Control Register in loop, waiting for device to set Ready bit in Control reg (0 → 1) to say it’s OK

• Processor then loads from (input) or writes to (output) data register
  • Load from or Store into Data Register resets Ready bit (1 → 0) of Control Register

• This is called “Polling”
I/O Example (polling)

• Input: Read from keyboard into $v0

```assembly
lui $t0, 0xffff #ffff0000
Waitloop:
    lw $t1, 0($t0) #control
    andi $t1,$t1,0x1
    beq $t1,$zero, Waitloop
    lw $v0, 4($t0) #data
```

• Output: Write to display from $a0

```assembly
lui $t0, 0xffff #ffff0000
Waitloop:
    lw $t1, 8($t0) #control
    andi $t1,$t1,0x1
    beq $t1,$zero, Waitloop
    sw ____ $a0, 12($t0) #data
```

“Ready” bit is from processor’s point of view!
Cost of Polling?

• Assume for a processor with a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling
  – Mouse: polled 30 times/sec so as not to miss user movement
  – Hard disk: assume transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed. (we’ll come up with a better way to do this)
% Processor time to poll

• Mouse Polling [clocks/sec]
  = 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]

• % Processor for polling:
  \[\frac{12 \times 10^3 \text{ [clocks/s]}}{1 \times 10^9 \text{ [clocks/s]}} = 0.0012\%\]
  ▶ Polling mouse little impact on processor
Clicker Time

Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. No transfer can be missed. What percentage of processor time is spent in polling (assume 1GHz clock)?

- A: 2%
- B: 4%
- C: 20%
- D: 40%
- E: 80%
% Processor time to poll hard disk

- Frequency of Polling Disk
  \[16 \text{ [MB/s]} / 16 \text{ [B/poll]} = 1 \text{M [polls/s]}\]

- Disk Polling, Clocks/sec
  \[1 \text{M [polls/s]} * 400 \text{ [clocks/poll]}\]
  \[= 400 \text{M [clocks/s]}\]

- % Processor for polling:
  \[400*10^6 \text{ [clocks/s]} / 1*10^9 \text{ [clocks/s]} = 40\%\]

  - Unacceptable

  (Polling is only part of the problem – main problem is that accessing in small chunks is inefficient)
What is the alternative to polling?

- Wasteful to have processor spend most of its time “spin-waiting” for I/O to be ready
- Would like an unplanned procedure call that would be invoked only when I/O device is ready
- Solution: use exception mechanism to help I/O. **Interrupt** program when I/O ready, return when done with data transfer
- Allow to register (post) **interrupt handlers**: functions that are called when an interrupt is triggered
Interrupt-driven I/O

1. Incoming interrupt suspends instruction stream
2. Looks up the vector (function address) of a handler in an interrupt vector table stored within the CPU
3. Perform a jal to the handler (needs to store any state)
4. Handler run on current stack and returns on finish (thread doesn’t notice that a handler was run)

```
handler:  lui    $t0, 0xffff
         lw      $t1, 0($t0)
         andi   $t1, $t1, 0x1
         lw      $v0, 4($t0)
         sw      $t1, 8($t0)
         ret
```

**CPU Interrupt Table**

<table>
<thead>
<tr>
<th>Interrupt(SPI0)</th>
<th>SPI0</th>
<th>handler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Administrivia

• Project 4, Performance Programming due on Wednesday 11/18

• Project 4 competition:
  – We will post the competition rules soon (After Wednesday).
  – We will allow you to use any optimizations that you want, and the top performers will receive extra credit, and their names will be engraved on the 61c website for all time.
  – Competition will end 11/30.
Agenda

• Devices and I/O
• OS Boot Sequence and Operation
• Multiprogramming/time-sharing
• Introduction to Virtual Memory
What happens at boot?

- When the computer switches on, it does the same as MARS: the CPU executes instructions from some start address (stored in Flash ROM).

\[ \text{PC} = 0x2000 \text{ (some default value)} \]

\[ \begin{align*}
0x2000: & \quad \text{addi } $t0, $zero, 0x1000 \\
& \quad \text{lw } $t0, 4($r0) \\
& \quad \ldots \\
& \quad (\text{Code to copy firmware into regular memory and jump into it})
\end{align*} \]
What happens at boot?

- When the computer switches on, it does the same as MARS: the CPU executes instructions from some start address (stored in Flash ROM)

1. **BIOS**: Find a storage device and load first sector (block of data)

2. **Bootloader** (stored on, e.g., disk): Load the OS kernel from disk into a location in memory and jump into it.

3. **OS Boot**: Initialize services, drivers, etc.

4. **Init**: Launch an application that waits for input in loop (e.g., Terminal/Desktop/...
Launching Applications

• Applications are called “processes” in most OSs.
• Created by another process calling into an OS routine (using a “syscall”, more details later).
  – Depends on OS, but Linux uses fork to create a new process, and execve to load application.
• Loads executable file from disk (using the file system service) and puts instructions & data into memory (.text, .data sections), prepare stack and heap.
• Set argc and argv, jump into the main function.
Supervisor Mode

• If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?
• The OS may need to enforce resource constraints to applications (e.g., access to devices).
• To help protect the OS from the application, CPUs have a supervisor mode bit.
  – A process can only access a subset of instructions and (physical) memory when not in supervisor mode (user mode).
  – Process can change out of supervisor mode using a special instruction, but not into it directly – only using an interrupt.
Syscalls

• What if we want to call into an OS routine? (e.g., to read a file, launch a new process, send data, etc.)
  – Need to perform a syscall: set up function arguments in registers, and then raise software interrupt
  – OS will perform the operation and return to user mode

• Also, OS uses interrupts for scheduling process execution:
  – OS sets scheduler timer interrupt then drops to user mode and start executing a user task, when interrupts triggers, switch into supervisor mode, select next task to execute (& set timer) and drop back to user mode.

• This way, the OS can mediate access to all resources, including devices and the CPU itself.
Agenda

• Devices and I/O
• OS Boot Sequence and Operation
• Multiprogramming/time-sharing
• Introduction to Virtual Memory
Multiprogramming

- The OS runs multiple applications at the same time.
- But not really (unless you have a core per process)
- Switches between processes very quickly. This is called a “context switch”.
- When jumping into process, set timer interrupt.
  - When it expires, store PC, registers, etc. (process state).
  - Pick a different process to run and load its state.
  - Set timer, change to user mode, jump to the new PC.
- Deciding what process to run is called scheduling.
Protection, Translation, Paging

• Supervisor mode does not fully isolate applications from each other or from the OS.
  – Application could overwrite another application’s memory.
  – Also, may want to address more memory than we actually have (e.g., for sparse data structures).

• Solution: Virtual Memory. Gives each process the illusion of a full memory address space that it has completely for itself.
Agenda

• Devices and I/O
• OS Boot Sequence and Operation
• Multiprogramming/time-sharing
• Introduction to Virtual Memory
“Bare” 5-Stage Pipeline

- In a bare machine, the only kind of address is a physical address
Dynamic Address Translation

Motivation
Multiprogramming, multitasking: Desire to execute more than one process at a time (more than one process can reside in main memory at the same time).

Location-independent programs
Programming and storage management ease
⇒ base register – add offset to each address

Protection
Independent programs should not affect each other inadvertently
⇒ bound register – check range of access

(Note: Multiprogramming drives requirement for resident supervisor (OS) software to manage context switches between multiple programs)
Simple Base and Bound Translation

Load X

Program Address Space

Base and bounds registers are visible/accessible only when processor is running in supervisor mode
Can fold addition of base register into (register+immediate) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.
Paged Memory Systems

- Processor-generated address can be split into:

  - page number
  - offset

- A page table contains the physical address of the base of each page

Page tables make it possible to store the pages of a program non-contiguously.
Private Address Space per User

- Each user has a page table
- Page table contains an entry for each user page
Where Should Page Tables Reside?

• Space required by the page tables (PT) is proportional to the address space, number of users, ...
  ⇒ Too large to keep in cpu registers

• Idea: Keep PTs in the main memory
  – Needs one reference to retrieve the page base address and another to access the data word
  ⇒ doubles the number of memory references!
Page Tables in Physical Memory

User 1 Virtual Address Space

User 2 Virtual Address Space

Physical Memory
In Conclusion

• Once we have a basic machine, it’s mostly up to the OS to use it and define application interfaces.
• Hardware helps by providing the right abstractions and features (e.g., Virtual Memory, I/O).
• If you want to learn more about operating systems, you should take CS162!
• What’s next in CS61C?
  – More details on I/O
  – More about Virtual Memory