CS 61C:
Great Ideas in Computer Architecture

Lecture 22:
Operating Systems

John Wawrzynek & Nick Weaver
http://inst.eecs.berkeley.edu/~cs61c
CS61C so far…

C Programs

```
#include <stdlib.h>

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

RISC-V Assembly

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

CPU

Project 1

Project 2

Caches

HW 1

Memory

2
So How is a Laptop Any Different?
Adding I/O

C Programs

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

RISC-V Assembly

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

HW 1

Project 1

Project 2

CPU

Caches

Screen

Keyboard

Storage

I/O (Input/Output)

Memory

4
Raspberry Pi (Less than $40 on Amazon in 2018)
It’s a Real Computer!
But Wait…

• That’s not the same! Our 61c experience isn’t like the real world. When we run VENUS, 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 (in millions of lines of code).

CC BY-NC 3.0 — David McCandless © 2013
http://www.informationisbeautiful.net/visualizations/million-lines-of-code/
Operating System
What Does the OS do?

• OS is first thing that runs when computer starts (and intermittently runs as long as computer is on)
• Finds and controls all I/O devices in the machine in a general way
  − Relying on hardware specific “device drivers”
• Starts services (100+)
  − File system,
  − Network stack (Ethernet, WiFi, Bluetooth, …),
  − …
• 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)
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
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 different 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: read a sequence of bytes
  − Output: write a sequence of bytes

• Interface options
  a) Special input/output instructions & hardware
  b) Memory mapped I/O
    ▪ Portion of address space dedicated to I/O
    ▪ I/O device registers there (no memory)
    ▪ Use normal load/store instructions, e.g. lw/sw
    ▪ Very common, used by RISC-V
Memory Mapped I/O

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory-mapped I/O</th>
<th>Program &amp; Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>control reg.</td>
<td></td>
</tr>
<tr>
<td>0x7FFFFFFF</td>
<td>data reg.</td>
<td></td>
</tr>
<tr>
<td>0x8000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFFFFFF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Processor-I/O Speed Mismatch

• 1 GHz microprocessor I/O throughput:
  – 4 Gi-B/s (lw/sw)
  – Typical I/O data rates:
    ▪ 10 B/s (keyboard)
    ▪ 100 Ki-B/s (Bluetooth)
    ▪ 60 Mi-B/s (USB 2)
    ▪ 100 Mi-B/s (Wifi, depends on standard)
    ▪ 125 Mi-B/s (G-bit Ethernet)
    ▪ 550 Mi-B/s (cutting edge SSD)
    ▪ 1.25 Gi-B/s (USB 3.1 Gen 2)
    ▪ 6.4 Gi-B/s (DDR3 DRAM)
  – These are peak rates – actual throughput is lower

• Common I/O devices neither deliver nor accept data matching processor speed
KEEP CALM
IT'S BREAK TIME
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Processor Checks Status before Acting

- Device registers generally serve two functions:
  - **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)
  - Indicates “data available” or “ready to accept data”

- Processor then loads from (input) or writes to (output) data register
  - I/O device resets control register bit (1 → 0)

- Procedure called “Polling”
I/O Example (Polling)

• Input: Read from keyboard into \texttt{a0}

\begin{verbatim}
lui t0, 0x7fffff #7fffff000 (io addr)
Waitloop: lw t1, 0(t0) #read control
andi t1, t1, 0x1 #ready bit
beq t1, zero, Waitloop
lw a0, 4(t0) #data
\end{verbatim}

• Output: Write to display from \texttt{a1}

\begin{verbatim}
lui t0, 0x7fffff #7fffff000
Waitloop: lw t1, 8($t0) #write control
andi t1, t1, 0x1 #ready bit
beq t1, zero, Waitloop
sw a1, 12(t0) #data
\end{verbatim}

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

Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7ffff000</td>
<td>input control reg</td>
</tr>
<tr>
<td>7ffff004</td>
<td>input data reg</td>
</tr>
<tr>
<td>7ffff008</td>
<td>output control reg</td>
</tr>
<tr>
<td>7ffff00C</td>
<td>output data</td>
</tr>
</tbody>
</table>
Cost of Polling?

• Assume for a processor with
  – 1 GHz clock rate
  – Taking 400 clock cycles for a polling operation
    ▪ Call polling routine
    ▪ Check device (e.g., keyboard or wifi input available)
    ▪ Return
  – What’s the percentage of processor time spent polling?

• Example:
  – Mouse
  – Poll 30 times per second
    ▪ Set by requirement not to miss any mouse motion
      (which would lead to choppy motion of the cursor on the screen)
% Processor time to poll

- Mouse Polling [clocks/sec]
  \[= 30 \text{ [polls/s]} \times 400 \text{ [clocks/poll]} = 12K \text{ [clocks/s]}\]

- % Processor for polling:
  \[12\times10^3 \text{ [clocks/s]} / 1\times10^9 \text{ [clocks/s]} = 0.0012\%
  \]
  \[\Rightarrow\text{ Polling mouse little impact on processor}\]
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 (rate at which chunks come could off disk)
  = 16 [MB/s] / 16 [B/poll] = 1M [polls/s]

• Disk Polling, Clocks/sec
  = 1M [polls/s] * 400 [clocks/poll]
  = 400M [clocks/s]

• % Processor for polling:
  400*10^6 [clocks/s] / 1*10^9 [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?

• Polling wastes processor resources
• Akin to waiting at the door for guests to show up
  – What about a bell?
• Computer lingo for bell:
  – **Interrupt**
  – Occurs when I/O is ready or needs attention
    ▪ Interrupt current program
    ▪ Transfer control to special code “interrupt handler”
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Traps/Interrupts/Exceptions:
altering the normal flow of control

An external or internal event that needs to be processed - by another program – the OS. The event is often unexpected from original program’s point of view.
**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 *(save PC in special MEPC* register)*
4. Handler run on current stack and returns on finish (thread doesn’t notice that a handler was run)

**Handler Execution**

| Stack Frame |

**Stack Frame**

**Stack Frame**

Label:  
- `s1 l t1, s3, 2`
- `add t1, t1, s5`
- `lw t1, 0(t1)`
- `or s1, s1, t1`
- `add s3, s3, s4`
- `bne s3, s2, Label`

**CPU Vector Interrupt Table**

<table>
<thead>
<tr>
<th>SPI0</th>
<th>handler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*MEPC: Machine Exception Program Counter*
Terminology

In CS61C (other definitions in use elsewhere):

- **Interrupt** – caused by an event external to current running program
  - E.g., key press, disk I/O
  - Asynchronous to current program
    - Can handle interrupt on any convenient instruction
    - “Whenever it’s convenient, just don’t wait too long”

- **Exception** – caused by some event during execution of one instruction of current running program
  - E.g., divide by zero, bus error, illegal instruction
  - Synchronous
    - Must handle exception precisely on instruction that causes exception
    - “Drop whatever you are doing and act now”

- **Trap** – action of servicing interrupt or exception by hardware jump to “interrupt or trap handler” code
Precise Traps

• Trap handler’s view of machine state is that every instruction prior to the trapped one (e.g., overflow) has completed, and no instruction after the trap has executed.

• Implies that handler can return from an interrupt by restoring user registers and jumping back to interrupted instruction
  – Interrupt handler software doesn’t need to understand the pipeline of the machine, or what program was doing!
  – More complex to handle trap caused by an exception than interrupt

• Providing precise traps is tricky in a pipelined superscalar out-of-order processor!
  – But a requirement, e.g., for
    ▪ Virtual memory to function properly (see next lecture)
Exceptions are handled like pipeline hazards

1) Complete execution of instructions before exception occurred
2) Flush instructions currently in pipeline (i.e., convert to \texttt{nop}s or “bubbles”)
3) Optionally store exception cause in status register
   - Indicate type of exception
   - Note: several exceptions can occur in a single clock cycle!
4) Transfer execution to trap handler
5) Optionally, return to original program and re-execute instruction
Trap Pipeline Diagram

\begin{itemize}
  \item (I_1) 096: DIV
  \item (I_2) 100: XOR
  \item (I_3) 104: SUB
  \item (I_4) 108: ADD
  \item (I_5) Trap Handler code*
\end{itemize}

*MEPC = 100 (instruction following offending ADD)
KEEP CALM
IT'S BREAK TIME
Administrivia

• Only 2 more weeks to go!

• HW 4 due tomorrow evening
• HW 5 (final homework!) released next week due Friday of last week of classes

• Project 4: Performance Programming
  – Due Monday, Monday April 16 (this coming Monday)

• Guerrilla Session tonight in 7-9pm in Mulford 159
  – Topics covered: Parallelism & MapReduce
  – Second to last guerrilla session—come and get valuable exam practice!

• Project 5 (WSC related) will be released by next week
  – You’ll get at ~1.5 weeks to complete it - due Monday RRR week
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
What Happens at Boot?

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

PC = 0x2000 (some default value)
What Happens at Boot?

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/...)

*BIOS: Basic Input Output System
Launching Applications

• Applications are called “processes” in most OSs
  – Thread: shared memory
  – Process: separate memory
  – Both threads and processes run (pseudo) simultaneously
  – Many user applications actually comprise multiple threads and/or processes

• Apps are started by another process (e.g., shell) calling an OS routine (using a “syscall”)
  – Depends on OS, but Linux uses `fork` to create a new process, and `execve` (execute file command) to load application

• Loads executable file from disk (using the file system service) and puts instructions & data into memory (.text, .data sections), prepares stack and heap

• Set argc and argv, jump to start of main

• Shell waits for main to return (join)
Supervisor Mode

• If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?

• The OS enforces resource constraints to applications (e.g., access to memory, devices)

• To help protect the OS from the application, CPUs have a supervisor mode (e.g., set by a status bit in a special register)
  – 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
  – Supervisory mode is a bit like “superuser”
    • But used much more sparingly (most of OS code does not run in supervisory mode)
    • Errors in supervisory mode often catastrophic (blue “screen of death”, or “I just corrupted your disk”)
Syscalls

• What if we want to call an OS routine? E.g.,
  – to read a file,
  – launch a new process,
  – ask for more memory (“sbreak” used by malloc),
  – send data, etc.

• Need to perform a syscall:
  – Set up function arguments in registers,
  – Raise software interrupt (with special assembly instruction)

• OS will perform the operation and return to user mode
• This way, the OS can mediate access to all resources, and devices
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
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 (on human time scale) – 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 alone is not sufficient to fully isolate applications from each other or from the OS
  – Application could overwrite another application’s memory.
  – Typically programs start at some fixed address, e.g. 0x8FFFFFFFF
    ▪ How can 100’s of programs share memory at location 0x8FFFFFFFF?
  – 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
Modern Virtual Memory Systems
Illusion of a large, private, uniform store

Protection
• several users (processes), each with their private address space and one or more shared address spaces

Demand Paging
• Provides the ability to run programs larger than the primary memory
• Hides differences in machine configurations

The price is address translation on each memory reference
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)
Paged Memory Systems

• Processor-generated address can be split into:

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
</table>

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

Address Space of User-1

Page Table of User-1

Physical Memory

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

PT User 1

PT User 2

Physical Memory
And, in Conclusion, ...

- **Input / output (I/O)**
  - Memory mapped: appears like “special kind of memory”
  - Access with usual load/store instructions (e.g., `lw, sw`)

- **Exceptions**
  - Notify processor of special events, e.g. divide by 0, page fault (next lecture)
  - “Precise” handling: immediately at offending instruction

- **Interrupts**
  - Notification of external events, e.g., keyboard input, disk or Ethernet traffic

- **Multiprogramming and supervisory mode**
  - Enables and isolates multiple programs

- **Take CS162 to learn more about operating systems**