CS61C
Great Ideas in Computer Architecture
(a.k.a. Machine Structures)

Operating Systems and Virtual Memory

UC Berkeley
Teaching Professor
Dan Garcia

UC Berkeley
Professor
Bora Nikolić

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org
Machine Structures

Instruction Set Architecture

Hardware

Software

Application (ex: browser)

Operating System

Compiler

Assembler

Processor

Memory

I/O system

Datapath & Control

Digital Design

Circuit Design

Transistors

Fabrication

CS61C

RISC-V
Machine Structures

CS61C

Software

Hardware

Application (ex: browser)

Operating System (Mac OSX)

Compiler

Assembler

Processor

Memory

I/O system

Datapath & Control

Digital Design

Circuit Design

Transistors

Fabrication

Instruction Set Architecture

RISC-V (3)
**New-School Machine Structures**

### Software
- **Parallel Requests**
  - Assigned to computer
  - e.g., Search “Cats”

- **Parallel Threads**
  - Assigned to core e.g., Lookup, Ads

- **Parallel Instructions**
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions

- **Parallel Data**
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words

### Hardware
- **Harness Parallelism & Achieve High Performance**

- **Warehouse Scale Computer**

- **Computer**
  - Core
  - Memory (Cache)
  - Input/Output
  - Exec. Unit(s)
  - Functional Block(s)

- **Main Memory**

- **Logic Gates**

### Hardware descriptions
- All gates work in parallel at same time

---

*Berkeley University of California*

*Garcia, Nikolić*
Great Idea #3: Principle of Locality / Memory Hierarchy

- **Processor chip**
  - Extremely fast
  - Extremely expensive
  - Tiny capacity

- **CPU Core**
  - Faster
  - Expensive
  - Small capacity

- **Registers**
  - Fast
  - Priced reasonably
  - Medium capacity

- **CPU Cache**
  - Faster
  - Expensive
  - Small capacity

- **Level 1 (L1) Cache**
  - Fast
  - Cheap
  - Large capacity

- **Level 2 (L2) Cache**
  - Faster
  - Expensive
  - Large capacity

- **Level 3 (L3) Cache**
  - Fast
  - Cheap
  - Large capacity

- **Physical Memory**
  - Fast
  - Priced reasonably
  - Medium capacity

- **Random-Access Memory (RAM)**
  - Faster
  - Expensive
  - Large capacity

- **Virtual Memory**
  - Fast
  - Priced reasonably
  - Medium capacity

- **Solid-State Memory (Flash)**
  - Faster
  - Expensive
  - Large capacity

- **Magnetic Disks**
  - Faster
  - Expensive
  - Large capacity

- **SSD, HDD Drives**
  - Faster
  - Expensive
  - Large capacity

- **DRAM chip – e.g. DDR3/4/5, HBM/HBM2/3**
  - Faster
  - Expensive
  - Large capacity

- **Solid-State Memory (Flash)**
  - Faster
  - Expensive
  - Large capacity
CS61C so far...

RISC-V Assembly

```
#include <stdlib.h>

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

```
.foo
lw t0, 4(a0)
addi t1, t0, 3
beq t1, t2, foo
nop
```

CS61C (7)
So How is a Laptop Any Different?

Keyboard

Screen

Storage
Adding I/O

C Programs

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

RISC-V Assembly

```
.foo
lw t0, 4(a0)
addi t1, t0, 3
beq t1, t2, foo
nop
```

This module: Memory (DRAM), Storage (Disk) and I/O

Screen, Keyboard, Storage

I/O (Input/Output)

Memory
Raspberry Pi ($35)

- CPU
- Storage I/O (Micro SD Card)
- Screen I/O (HDMI)
- Memory
- Wireless I/O (WiFi)
- Serial I/O (USB)
- Network I/O (Ethernet)
It's a Real Computer!
CS61C with Raspberry PI?

- Lot’s of concepts from 61C covered in a book, with Raspberry Pi exercises

(and it is free to download if you are a Cal student:
But Wait...

- That’s not the same! 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)
Operating System Basics
Well, “Just Software”

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

![Chart showing codebases in millions of lines of code](http://www.informativisbeautiful.net/visualizations/million-lines-of-code/)

Codebases (in millions of lines of code). CC BY-NC 3.0 — David McCandless © 2015

http://www.informationisbeautiful.net/visualizations/million-lines-of-code/
Operating System

Lines of code in Linux kernel

Millions of lines

versions

RISC-V (16)
What Does the OS do?

- OS is the (first) thing that runs when computer starts
- Finds and controls all devices in the machine in a general way
  - Relying on hardware specific “device drivers”
- Starts services (100+)
  - File system,
  - Network stack (Ethernet, WiFi, Bluetooth, ...),
  - TTY (keyboard),
  - ...
- 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)
What Does the Core of the OS Do?

- Provide *isolation* between running processes
  - Each program runs in its own little world
- Provide *interaction* with the outside world
  - Interact with "devices": Disk, display, network, etc...
What Does OS Need from Hardware?

- **Memory translation**
  - Each running process has a mapping from "virtual" to "physical" addresses that are different for each process.
  - When you do a load or a store, the program issues a virtual address... But the actual memory accessed is a physical address.

- **Protection and privilege**
  - Split the processor into at least two running modes: "User" and "Supervisor".
    - RISC-V also has "Machine" below "Supervisor".
  - Lesser privilege *can not* change its memory mapping.
    - But "Supervisor" *can* change the mapping for any given program.
    - And supervisor has its own set of mapping of virtual->physical.

- **Traps & Interrupts**
  - A way of going into Supervisor mode on demand.
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
Operating System Functions
Launching Applications

- Applications are called “processes” in most OSs
  - Thread: shared memory
  - Process: separate memory
  - Both threads and processes run (pseudo) simultaneously

- Apps are started by another process (e.g., shell) calling an OS routine (using a “syscall”)
  - Depends on OS; 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”)
What if we want to call an OS routine? E.g.,
- to read a file,
- launch a new process,
- ask for more memory (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
We need to transition into Supervisor mode when "something" happens

**Interrupt**: Something external to the running program
  - Something happens from the outside world

**Exception**: Something done by the running program
  - Accessing memory it isn't "supposed" to, executing an illegal instruction, reading a csr not supposed at that privilege

**ECALL**: Trigger an exception to the higher privilege
  - How you communicate with the operating system: Used to implement "syscalls"

**EBREAK**: Trigger an exception within the current privilege
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., memory error, bus error, illegal instruction, raised exception
  - 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
### Trap Handling

- Altering the regular execution flow

An external or internal event that needs to be processed - by another program; often handled by OS. The event is often unexpected from original program's point of view.
Precise Traps

- Trap handler’s view of machine state is that every instruction prior to the trapped one (e.g., memory error) 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 for things to actually work right!
Exceptions in a 5-Stage Pipeline

PC -> IMEM -> Dec. (PC address exception)
IF -> Reg -> Dec. (Illegal opcode)
ID -> ALU -> Dec. (Data address exceptions)
EX -> DMEM -> MA -> WB

Garcia, Nikolić
Exceptions are handled *like pipeline hazards*

1. Complete execution of instructions before exception occurred.
2. Flush instructions currently in pipeline (i.e., convert to *nops* or "bubbles").
3. Optionally store exception cause in status register.
   - Indicate type of exception.
4. Transfer execution to trap handler.
5. Optionally, return to original program and re-execute instruction.
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 (we will call this ‘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
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. 0x8FFFFFFF
  - How can 100’s of programs share memory at location 0x8FFFFFFF?
- 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