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

Operating Systems and Virtual Memory

cs61c.org
Machine Structures

Software

Hardware

Application (ex: browser)
Compiler
Assembler
Processor
Memory
I/O system
Datapath & Control
Digital Design
Circuit Design
Transistors
Fabrication

Operating System (Mac OS X)

Instruction Set Architecture

Hardware

Software

CS61C

Berkeley

RISC-V (4)
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**

**Hardware descriptions**

- All gates work in parallel at same time
Great Idea #3: Principle of Locality / Memory Hierarchy

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

- **DRAM chip** (e.g. DDR3/4/5, HBM/HBM2/3)
  - Faster
  - Expensive
  - Small capacity

- **SSD, HDD Drives**
  - Fast
  - Cheap
  - Large capacity

- **Physical Memory**
  - Random-Access Memory (RAM)
  - Fast
  - Medium capacity

- **Virtual Memory**
  - Solid-State Memory (Flash)
  - Fast
  - Large capacity

- **Magnetic Disks**
  - Slow
  - Cheap
  - Large capacity
RISC-V (7)

C Programs

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

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

Project 1

Project 2

Project 3

CS61C so far...

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

RISC-V Assembly

```c
#include <stdlib.h>

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

C Programs

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

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

Screen  Keyboard  Storage

I/O (Input/Output)

Memory

Program  Bytes  Data
Raspberry Pi ($35)

- Storage I/O (Micro SD Card)
- Wireless I/O (WiFi)
- Screen I/O (HDMI)
- Memory
- Serial I/O (USB)
- Network I/O (Ethernet)
It’s a Real Computer!

thepihut.com
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:
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:

[Image: Codebases (in millions of lines of code). CC BY-NC 3.0 — David McCandless © 2015 http://www.informatioinsbeautiful.net/visualizations/million-lines-of-code/]
Operating System

Lines of code in Linux kernel

[Graph showing the growth of lines of code in Linux kernel versions from 1991 to 2021, with millions of lines on the y-axis and versions on the x-axis.]
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)

- Memory mapped

- 0x0002000: Code to copy firmware into regular memory and jump into it.)
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”)

Garcia, Nikolić
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
Trapping can alter 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!
Trap Handling

- 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
  4) Transfer execution to trap handler
  5) Optionally, return to original program and re-execute instruction
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 (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
Virtual Memory Concepts
Virtual Memory

- Virtual memory - Next level in the memory hierarchy:
  - Provides program with illusion of a very large main memory: Working set of “pages” reside in main memory - others are on disk
  - Demand paging: Provides the ability to run programs larger than the primary memory (DRAM)
  - Hides differences between machine configurations

- Also allows OS to share memory, protect programs from each other

- Today, more important for protection than just another level of memory hierarchy

- Each process thinks it has all the memory to itself

- (Historically, it predates caches)
Great Idea #3: Principle of Locality / Memory Hierarchy

- **Processor chip**
  - Extremely fast
  - Extremely expensive
  - Tiny capacity
- **DRAM chip – e.g. DDR3/4/5, HBM/HBM2/3**
  - Faster
  - Expensive
  - Small capacity
- **SSD, HDD Drives**
  - (Not so) Fast
  - Cheap
  - Large capacity
- **CPU Core**
  - Registers
- **CPU Cache**
  - Level 1 (L1) Cache
  - Level 2 (L2) Cache
  - Level 3 (L3) Cache
- **Physical Memory**
  - Random-Access Memory (RAM)
- **Virtual Memory**
  - Solid-State Memory (Flash)
- **Magnetic Disks**

**Berkeley UNIVERSITY OF CALIFORNIA**

**RISC-V [36]**

Garcia, Nikolić
Virtual vs. Physical Addresses

- Processes use virtual addresses, e.g., 0 … 0xffff,ffff
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g., 0 … 0xffff,ffff)
  - Memory manager maps virtual to physical addresses

Many of these (soft & hardware cores) 0000 0000_{hex}

One main memory
Address space = set of addresses for all available memory locations

Now, two kinds of memory addresses:

- **Virtual Address Space**
  - Set of addresses that the user program knows about
- **Physical Address Space**
  - Set of addresses that map to actual physical locations in memory
    - Hidden from user applications

Memory manager maps (‘translates’) between these two address spaces
Bora’s Laptop

```plaintext
bora@DESKTOP-QAB1DCR:~$ lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: LittleEndian
Address sizes:
  CPU(s): 39 bits physical, 48 bits virtual
  On-line CPU(s) list: 0-7
  Thread(s) per core: 2
  Core(s) per socket: 4
  Socket(s): 1
  Vendor ID: GenuineIntel
  CPU family: 6
  Model: 126
  Model name: Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
  Stepping: 5
  CPU MHz: 1497.604
  BogoMIPS: 2995.20
  Hypervisor vendor: Microsoft
  Virtualization type: full
  L1d cache: 192 K1B
  L1i cache: 128 K1B
  L2 cache: 2 M1B
  L3 cache: 8 M1B
```
Analogy

- Book title like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call #
- On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk (storage)
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights
Memory Hierarchy Requirements

- Allow multiple processes to simultaneously occupy memory and provide protection – don’t let one program read/write memory from another

- Address space – give each program the illusion that it has its own private memory
  - Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same address. So each program has a different view of memory.
Physical Memory and Storage
### Memory

- **Memory (DRAM)**
  - **Desktop/server**
  - **MS Surface Book**

- **Apple A12 Bionic (DRAM goes on top)**

**Volatile**

- Latency to access first word: ~10ns (~30-40 processor cycles)
- Each successive (0.5ns – 1ns)
- Each access brings 64 bits
- Supports ‘bursts’
Storage – “Disk”

Attached as a peripheral I/O device; Non-volatile

- **SSD**
  - Access: 40-100μs (~100k proc. cycles)
  - $0.05-0.5/GB

- **HDD**
  - Access: <5-10ms (10-20M proc. cycles)
  - $0.01-0.1/GB
Aside ... Why are Disks So Slow?

- 10,000 rpm (revolutions per minute)
- 6 ms per revolution
- Average random access time: 3 ms (~10^7 processor cycles)
How Hard Drives Work?

- Nick Parlante’s https://cs.stanford.edu/people/nick/how-hard-drive-works/
  - Several YouTube videos as well
What About SSD?

- Made with transistors
- Nothing mechanical that turns
- Like “Ginormous” register file
  - Does not "forget" when power is off (non-volatile)
- Fast access to all locations, regardless of address
- Still much slower than register, DRAM
  - Read/write blocks, not bytes
  - Potential reliability issues
- Some unusual requirements:
  - Can’t erase single bits – only entire blocks
Flash Memory

- 3D array of bit cells (up to 256 layers!)

3D NAND Architecture

Western Digital/Semiengineering
Memory Manager
In a ‘bare metal’ system (w/o OS), addresses issued with loads/stores are real physical addresses

- In this mode, any process can issue any address, therefore can access any part of memory, even areas which it doesn’t own
  - Ex: The OS data structures
- We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - a translation mechanism
  - Check that process has permission to access a particular part of memory
100+ Processes, managed by OS

- 100’s of processes
  - OS multiplexes these over available cores
- But what about memory?
  - There is only one!
  - We cannot just “save” its contents in a context switch …
Processes use virtual addresses, e.g., $0 \ldots 0xffff,ffff$
- Many processes, all using same (conflicting) addresses

Memory uses physical addresses (also, e.g., $0 \ldots 0xffff,ffff$
- Memory manager maps virtual to physical addresses
Conceptual Memory Manager

Concept; Real memory managers use more complex mappings
Responsibilities of Memory Manager

1) Map virtual to physical addresses

2) Protection:
   - Isolate memory between processes
   - Each process gets dedicated “private” memory
   - Errors in one program won’t corrupt memory of other program
   - Prevent user programs from messing with OS’s memory

3) Swap memory to disk
   - Give illusion of larger memory by storing some content on disk
   - Disk is usually much larger and slower than DRAM
     - Use “clever” caching strategies