Computer Systems

CSE 410 Winter 2022

18 –Course Summary

The Final Exam

- Some “why” questions
  - Example: Why do systems implement virtual memory? What problem(s) does it solve?
- Some “what” questions
  - Example: What mechanism do operating systems use to protect memory?
- Some “demonstrate” questions
  - Example: Express the meaning of this C language for loop in RISC-V assembler instructions. (Assume nothing useful is in registers when your code starts.)

Part 1: Some Themes

- "Simpler is faster"
- Static vs. Dynamic Evaluation
- Representation and Translation
- Interfaces vs. Implementation
  - Layers, not options
  - Policy vs. mechanism
  - Interposition to evolve functionality
- Naming / Virtualization
- Parallelism / Concurrency
  - ~~Atomicity~~
- Trading space for time

Simpler is Faster

- RISC-V ISA
  - load-store architecture (all operations are on values in registers)
  - instructions are all 32 bits long (vs. variable length instructions)
  - simple operations (load, store, add, and, shift, branch, jal, etc.)
    - no hardware notion of stack
    - no hardware notion of procedure call (beyond jal instruction)
    - no hardware notion of type

- Languages
  - C vs. Java (functionality/performance)

Simpler is Faster: Layering

- OS over hardware
  - abstracts memory to address space
  - abstracts processor to thread
  - abstracts disk to file system
  - (abstracts network to sockets/connections)

- Virtual Memory
  - Hardware provides mechanism (address translation and page faults), OS provides policy (management of physical memory, setting access rights in virtual memory)

- Language over ISA / OS abstractions
  - programmer works to language specification, not ISA/OS specification
    - more efficient for programmer
    - portable across ISAs

- ISA over Hardware
  - hardware is boolean circuits, ISA is an interface specification

Layering

- Caches / Memory Hierarchy
  - Main memory cache
  - Small/fast
  - Transparent

- Registers
  - a “programmer controlled cache”
  - pushes issue of deciding which values are most important right now to the compiler (see Static vs. Dynamic)

Static vs. Dynamic

- We often prefer to do things statically – at compile/build time – because then we don't pay the cost of doing it at each execution
  - Example: type checking
- Additionally, we can "view the entire program" statically, whereas we see only the current instruction at run time
  - Compiler can perform optimizations statically that you couldn't perform dynamically
    - E.g., dead code removal (computing a value that will never be used)
- Some things can be done only dynamically
  - E.g., optimizing instruction sequences across branches (e.g., inserting bubbles in pipeline when needed, but only when needed)
  - All computations/optimizations involving input

Representation and Translation

- Everything is bits
  - Why base 2? Why not base 3, say?
- Integers
  - signed and unsigned (why?)
  - overflow
- Floats
- Characters
- Strings
- Objects

- Instructions
  - Want compact encoding of instructions. Why?
  - Must be able to encode every possible instruction

Representation and Translation: Instructions

- Instruction formats
- Why have an immediate format instruction?
- base-displacement memory addressing
  - Why?
- PC relative branch format
  - Why?

- Assembly language
  - “human readable” assembly language
  - Role of the assembler
    - What does it do, what doesn’t it do (compared with a compiler, say)?

Interfaces vs. Implementations

- ISA as an interface
  - One-at-a-time instruction execution model
- Pipelining as an implementation
  - Many instructions in execution at once (Instruction Level Parallelism)
  - Pipelining is made more effective by careful design of the ISA (RISC-V)
- Pipeline hazards
  - Load hazards
  - Control hazards

Layers, Not Options

- Programs on top of compilers on top of OS on top of hardware
- User level threads on top of kernel threads

Policy vs. Mechanism

- Example Mechanism: trap handler mechanism
  Example Policy: Whatever OS decides to do in response

- (Same Example) Mechanism: signals
  (Same Example) Policy: whatever app decides to do in response

- The most general way to defer policy to a layer above is to allow that layer to execute code when an event that requires a policy decision occurs

Interposition / Naming

- Interposition is finding an existing interface and inserting new functionality that conforms to the existing interface
- Example: Main memory caches
- Example: Virtual memory
- Example: copy-on-write fork()

Parallelism / Concurrency

- Instruction level parallelism
  - Pipelines
- Processes
  - More than one application running concurrently
- Threads
  - A single application using more than on core concurrently
  - A single application dividing it’s control flow into simple, relatively independent paths

Trading Space for Time

- Caches
    - more space, less time
- Virtual memory
    - Less space, more time
- Pipelines
    - More space, less time

Part 2: Skills (Things You Can Do)

- Compile C-like code to RISC-V assembler
- Decompile RISC-V assembler to C-like code
- Encode/decode RISC-V assembler/machine instructions
- Manually simulate the result of executing a RISC-V assembler instruction sequence
- Follow subroutine calling conventions in RISC-V assembler to call a function and to return a value from function to the caller
  - including caller/callee saved registers
- Manually optimize a sequence of assembler instructions by re-writing them to an equivalent sequence that runs in fewer cycles

More Things You Can Do: Boolean Circuits

- Can define truth tables for a Boolean function
- Can convert a truth table into a Boolean circuit
- Can convert a Boolean circuit into a truth table

- Understand how Boolean circuits can do binary addition
  - half-adders and full-adders

More Things You Can Do: Machine Organization

- Determine how many cycles it will take to issue a sequence of RISC-V instructions into the standard five stage pipeline
  - Including dealing with control and load hazards when the pipeline implements forwarding

- Determine the RAW, WAW, and WAR dependences that limit possible parallel execution of a sequence of instructions

More TYCD: Caches

- Understand when caches will be effective
    - Temporal and spatial locality
- Informally evaluate code in terms of how much spatial and/or temporal locality it has
- Apply some simple re-writes to code to improve it’s spatial and/or temporal locality
    - We used block matrix multiply as an example in class
- Given a cache design (block/cache line size, number of lines, and set associativity) and a memory address, determine where in the cache to look for a possibly cached copy of the value stored at that address

More TYCD: OS / Security

- Understand the mechanism the OS uses to protect the CPU
- Understand the mechanism the OS uses to protect I/O devices
- Understand the mechanism the OS uses to protect memory
- Be able to explain how the OS on klaatu keeps you from running a program that reads private files owned by other users

More TYCD: OS / Processes

- Understand fork/exec
  - What they do
  - Why giving a chance for the parent’s code to run in the context of the child’s process is a good idea
  - How is “inheritance” used (and useful) in fork()?
- Understand/write code for a simple shell that does input/output redirection or creates pipes between processes
- Understand how it is that the OS protects

More TYCD: Virtual Memory

- Understand the mechanism:
  - address translation: mapping from a virtual address to a physical address through a page table
    - finding the page table index to locate a page table entry
    - checking the valid bit
    - extracting the physical frame number
    - combining it with the offset from the address to create the physical address
- Page faults
  - because the valid bit is false
  - because the page table entry doesn't permit access of the type being attempted (read, write, execute)
- Holes in the virtual address space
  - How?
  - Why?

More TYCD: Processes / Address Spaces

- How do you create a process?
  - What has to happen to create a process?
  - What doesn’t happen?
    - Role of exec()
- Memory layout for process address space
  - stack / heap / static data / text
  - permissions for each section of the address space
- Process control blocks and process state
  - running / runnable / blocked
- The context switch mechanism
  - Why must the hardware save the PC at the time of an interrupt/trap/exception? Why can’t the software do it?
- Copy on write
  - What is it used for?
  - How does it work?

More TYCD: Threads

- Why aren’t processes enough?
- Creating a thread (in Java)
- join()’ing with a thread

- Why use both kernel threads and user level threads?

More TYCD: Threads / Synchronization

- Recognizing code that has a race condition
- Recognizing code that is a critical section
  - And code that isn't
- Resolving critical sections through mutual exclusion
  - in general, using locks
  - in Java, using "synchronized" methods
- Recognizing when deadlock might occur
- *Can multiple processes have a race condition (among them)? Can they deadlock?*

Limits to Parallelism / Concurrency

- Number of cores on system
- Number of threads in the application

- Depth of pipeline
- Dependences among instructions (RAW, WAW, WAR)

- Another TYCD:
  Amdahl’s Law (relating “inherently sequential” fraction of execution time to maximum possible speedup)

That’s It

- Thank you for this quarter.
- The final will be online (Canvas) next Wednesday, 2:30-4:20.
  I will be online for email’ed questions.