CPSC 313 - Computer Hardware and OS

Jan. 6th - Lecture 1: Introduction

- Office Hours: Tue 13:30 - 14:30 (Zoom); Wed 16:00 - 17:00 (DLC Table 3); Fri 9:30 - 10:30 (DLC Table 3)
- cpsc313-admin@cs.ubc.ca
- Grading
    - Final: 36%
    - Quizzes: 36% (6 quizzes - in CBTF)
    - Labs: 22% (10 labs, each worth the same)
    - Participation: 6% (Pre-class + In-class + Tutorials)

Jan. 8th - Lecture 2: The y86 (Sequential Processor)

- Mostly review
- y86 in a single slide
    - Registers: `%rax, %rcx, %rdx, %rbx, %rsp, %rbp, %rsi, %rdi` + `%r8 - %r14`
    - Flags: ZF (Zero-flag), SF (Sign-flag), OF (Overflow-flag)
    - Status Register
    - Program Counter
    - DMEM (DRAM): Memory
    - Instructions
        - 1-byte instructions: op + fun
        - 2-byte instructions: op + fun + rA + rB
        - 9-byte instructions: op + fun + 64-bit address
        - 10-byte instructions: op + fun + rA + rB + 64-bit address
- Representing Data in Memory

        - Swapping two values in registers
        - Swapping two values in memory
- Little Endian Representation
    - Assume the value we wish to store in the register to be: `0xDEBC0A8967452301`
    - Then instruction code that we use to store it into register `0` is the following:

      30 F0 01 23 45 67 89 0A BC DE
        - where `30` represent `irmovq`
        - `F0` represents moving the value to register `0`
        - the rest is data represented in Little Endian
    - We do not store strings or arrays backwards
- Expressing Negative Numbers
    - 2's complement: write the positive number in binary, flip all the bits, add 1

Jan. 10th - Lecture 3: The y86 (Cont.d) (ALU Instructions and Control Flow)

- Logical Operations: A refresher
    - `and`: outputs 1 only when both inputs are 1
    - `xor`: outputs 1 only when both inputs are different
    - Practice
        - `0xEEEE & 0x1111` = `0x0000`
        - `0xEEEE ^ 0x1111` = `0xFFFF`
        - `0x9393 & 0xA0A0` =
          `1001 0011 1001 0011 & 1010 0000 1010 0000` = `0x8080`
        - `0x9393 ^ 0xA0A0` = `0x3333`
- Arithmetic Fundamentals
    - Addition is traditional. However, if we assume 16-bit values:
        - `0x5` + `0x3` = `0x8`
        - `0xF` + `0x1` = `0x10`
        - `0xFFFF` + `0x0001` = `0x0000` (!)

        - How does hardware perform subtraction
            - `0x5` $-$ `0x3` $=$ `0x5` $+ (-$`0x3`$)$ $=$ `0101 + 1101` $=$ `0010`
    - Condition code/flag fundamentals
        - Z flag: the last ALU operation gives the result of 0
        - S flag: the last ALU operation produced a number with a 1 in the high order bit
        - O flag: the last ALU operation produced an arithmetic overflow
    - Using condition codes to jump
        - Reason we have this is to implement control structures, such as `if-then`, `if-then-else`, and loops
        - Writing assembly: which ALU to use to set the condition flags so that the proper conditional jump is used

Jan. 13th - Lecture 4: The y86 (Cont.d) (Execution)

- Quickly review `call`
    - Part 1: `pushq PC`
        - `R[%rsp] <- R[%rsp] - 8`
        - M_8`[R[%rsp]] <- PC`
    - Part 2: change the `PC`
        - `PC <- Dest`
- `ret`
    - Part 1: `popq PC`
        - `PC <-` M_8`[R[%rsp]]`
        - `R[%rsp] <- R[%rsp] + 8`
- y86 Instruction Reference Sheet
- y86 Implementation: Steps in Execution
    - `mrmovq 4(%r8), %r9`
    - Fetch
        - PC: program counter $\rightarrow$ Instruction Register
        - $\rightarrow$ `rA` $=$ `%r9`, `rB` $=$ `%r8`, `valC` $= 4$
    - Decode
        - decode `valB` $=$ `R[rB]`

        - do nothing about `valA`: as we want to put value into this regardless of previous values
    - Execute
        - With condition flags (though not set in this instruction)
        - ALU takes `valB` and `valC` to yield `valE`
    - Memory
        - Use `valE` to read from memory, and gives us the value at the desired memory location, denoted as `valM`
    - Writeback
        - we write `valM` to `R[rA]`

Jan. 15th - Lecture 5: The y86 (Cont.d) (Buffer Overflows)

- Buffer Overflows
    - When we exploit the fact that you can (sometimes) write outside of a buffer, thus corrupting memory around it
    - Several examples: The Internet Worm; Heartbleed; WhatsApp VOIP

Jan. 17th - Lecture 6: The y86 (Cont.d) (Instructions Implementation)

- Understanding Execution
    - We understood each instruction in logical form - what it did
    - we saw how to realize instructions in hardware, using wiring diagrams
    - execution as a set of formal specification statements (what must happen in each phrase for each instruction)
    - we will examine execution as a phased process routing signals
- Calling conventions
    - Why?
        - C functions are often compiled independently
        - what if `fputc` was updated and now changes `%rbx`
            - We would need to recompile every function that calls `fputc`, directly or indirectly

                - some of them might be in libraries we do not have code for
        - Parameter passing
        - Caller-save registers
            - functions are allowed to modify them if they like
            - if
                - function f calls function g, and
                - f wants to use the value in `%rcx` after g returns
            - then
                - f needs to save `%rcx` before calling g, and
                - restore it afterwards
            - because g might modify `%rcx`
        - Callee-save registers
            - functions can rely on the value of the register not changing
            - if
                - function g wants to use `%rbp`
            - then
                - g needs to save `%rbp` before modifying it
                - g needs to restore the old value of `%rbp` before returning
            - because
                - the function that called g expected `%rbp` not to change
- `rrmovq` three ways
    - what it does: `R[rB] <- R[rA]`
    - from implementation video: wiring diagram
    - by specification
        -
            ```
            # Fetch
            icode:ifun = M1[PC]
            rA:rB = M1[PC + 1]
            valP = PC + 2
            ```
        -
            ```
            # decode
            valA = R[rA]
            ```

        - ```
          # execute
          valE = valA + 0
          ```
        - ```
          # memory
          ```
        - ```
          # writeback
          R[rB] = valE
          ```
        - ```
          # update PC
          PC = valP
          ```
- From `rrmovq` to `cmovxx`
    - ```
      # fetch
      icode:ifun = M1[PC]
      rA:rB = M1[PC + 1]
      valP = PC + 2

      # decode
      valA = R[rA]

      # execute
      valE = valA + 0
      Cnd = cond(CC, ifun)

      # memory


      # writeback
      if Cnd then R[rB] = valE

      # PC update
      PC = valP
      ```

- Allowed in specifications
    - Fetch: `icode, ifun, rA, rB, valC, valP, PC, Memory` (for instruction fetch)
      ```

- Decode: `rA, rB, valA, valB`
- Execute: `valC, valE, valA, valB, CC, Cnd, cond, 0, 8, -8, ifun`
- Memory: `valA, valB, etc`
- Writeback
- PC Update

Jan. 20th - Lecture 7: The y86 (Sequential Wrap-up)

- Why cannot we have the instruction `addm offset(rB), rA`?
    - We need to do two additions - using ALU twice.
    - The clock cycle will be prolonged if there exists a second ALU.
- `AluA`
    - Inputs: `valA`, `valC`, `icode`, ± 8
    - Outputs: input to ALU
    - Uses of the inputs
        - `valA`: `cmov*, OPq`
        - `valC`: `irmovq, mrmovq, rmmovq`
        - ± 8: : `pushq, call, popq, ret`
- Digression: the `? :` operator in C
    - the if-then-else operator `expr1 ? expr2 : expr3`, means `if (expr1) then expr2 else expr3`

Jan. 22nd - Lecture 8: Introduction to Pipelining

- The key observation is that most of the time, most of the circuitry in our sequential CPU is idle
- We want to avoid such idle in hardware
- Terminology
    - Stage (phase): each piece of the pipeline is a stage, the unit of computation that the processor performs in a single clock cycle
    - Instruction statuses
        - In-flight: somewhere in the pipeline
        - Retired: instruction has completed execution

        - Pipeline Register
            - The extra work we incur by pipelining
            - a register in the processor that captures the set of signals (values) on which a stage will execute
            - Each stage has its own pipeline register
        - Hazard
            - The "problems" that we run into when we naively add pipelining
- Performance: Latency
    - Sequential: add time of each stage together and the register delay
    - Pipelined: add the time of the longest stage and the register delay, and then multiply the time with the number of stages
- Performance: Retirement Latency
    - The amount of time for which the current instruction waits in order to complete because of previous instruction's execution.
- Performance: Throughput
    - The rate at which instructions complete. Typically expressed in GIPS (Giga instructions per second)
- Pipeline Issues
    - For a pipelined processor, throughput is the inverse of the retirement latency

Jan. 27th - Lecture 10: Pipelining - Hazards

- Pipeline Execution
    - Pipeline registers hold values for their stage
    - Stages execute in parallel
    - Signals flow from the pipeline register and do not enter the next pipeline register until the clock ticks
    - During a stage, we use the contents of that stage's pipeline register to compute the values that will go into that stage
- Describing Instruction Execution
    - `UPPERCASE_val`: indicates the stage's input
    - `lowercase_val`: indicates the stage's computation result

- What can go wrong? Hazards
    - Problems that arise in pipelining when information needed by an instruction is not available when it needs it - unless mitigated, hazards can lead to incorrect results
    - Types
        - Data hazard: when data dependencies (e.g. one instruction reads from a register that another one writes) potentially lead to incorrect behavior
        - Control hazard: when control dependencies (e.g. a conditional jump determines the address of the next instruction to execute) potentially lead to incorrect behavior
- Approaches to Hazard Handling
    - Pure software
    - Delaying execution in hardware
    - Value forwarding in hardware (addresses data hazards)
    - Branch prediction

Jan. 29th - Lecture 11: Pipelining - Forwarding

- Forwarding
    - Consider

```
addq %rax, %rbx
subq %rbx, %rcx
```

    - We **have** the value for `%rbx`, which is what we need, before it is written into the register file, so we need to somehow use the value `e_valE`
    - We simply take the `e_valE` value and **forward** it just before the execution register for `valA, valB`
    - Things that do not work
        - Forward `e_valE` into the register file
        - Forward `M_valE` to the new logic
        - Forward `e_valE` into the beginning of the execute stage
    - However, consider

```
addq %rax, %rbx
irmovq 10, %rdx
subq %rbx, %rcx
```

    - In this code block, the value needed is the output of the Memory stage
- Forwarding lets us to mitigate some of the hazards that occur due to pipelining

Jan. 31st - Lecture 12: Pipelining - Branch Prediction

- Dealing with Control Hazards
    - `jmp, call`: we know where to go in Fetch
    - `ret`: we have to stall for three cycles
    - `jXX`: we guess
        - always taken - assume that the nextPC is going to be valC
        - never taken - assume that the nextPC is going to be valP
        - other fancier ways
- Branch Prediction examples
    -
```
j = 1;
do {
    j++;
} while j < 10;
```

```
# assume j is in %rsi
    irmovq 1, %rax      # incr = 1
    irmovq 1, %rsi      # j = 1
Loop:
    addq %rax, %rsi
    irmovq 10, %rdi     # rdi = 10
    subq %rsi, %rdi     # set CC
    jg Loop
```

        - 8 times where we always take a jump, only 1 time where we won't jump

- ```
  if (p == NULL) {
      return -1;
  }
  ```

  ```
  ###
      andq %rbp, %rbp
      je error
  error:
      irmovq -1, %rax
      ret
  ```

    - We are not jumping very often
  - Backward jumps are often used to repeat a loop body, so they are usually taken
  - Forward jumps are usually conditionals, and may or may not be taken
- Steering Branch Prediction (Linux Kernel)
  - Linux has two macros

    ```
    #define likely(x) __builtin_expect(!!(x), 1)
    #define unlikely(x) __builtin_expect(!!(x), 0)
    ```

- More complex strategies
  - Need to introduce prediction logic
  - The predictor takes the PC increment, and instruction memory as input
  - Earlier conditional jumps' misprediction should prevent later conditional jumps from being taken
- Facts about Branch Prediction
  - Some processors devote hardware to branch prediction
    - 1 bit: do whatever was done last time
    - more bits for more complicated schemes
    - entries for branches at different values of the PC
  - Other processors let you execute instructions while you are waiting to determine what is going to happen with the jump - these instructions occupy the **branch delay slots**
- What happens if we mis-predict?
    ```

- There can exist instructions in the pipeline that should not have been executed at all
- We can cancel or squash or quash them by replacing these instructions by `nop`s
- In the worst case, it is no worse than if no prediction is done

Feb. 3rd - Lecture 13: Pipelining - Performance (Cycle counting)

- Consider the following program

```
    xorq %rax, %rax     # rax = 0
    xorq %rsi, %rsi     # rsi = 0
    irmovq 1, %rbx      # to increment
Loop:
    irmovq 5, %rdi      # rdi = 5
    subq %rsi, %rdi     # check end cond
    jle Done            # if 5-j <= 0, jmp
    addq %rbx, %rsi     # j++
    addq %rbx, %rax     # count++
    jmp Loop
Done:
    halt
```

```
count = 0;
for (int j = 0; j < 5; j += 1) {
    count += 1;
}
```

- Assume no data forwarding and no branch prediction
    - Data hazards occurs on `subq %rsi, %rdi`, we need 3 stalls
    - Control hazard occurs on `jle Done`, we need 2 stalls
    - 5 stalls per iteration $\times$ 6 iterations $=$ 30 bubbles produced
    - The CPI:
        - 3 instructions before the loop, execute 1 time
        - check loop condition 6 times, 3 instructions
        - body of the loop 5 times, 3 instructions

        - end of loop is 1 instruction
        - 37 instructions, each instruction take 1 cycle
        - We need 4 clock cycles to load the pipeline
        - 30 cycles for stalls
        - The sum is 71 cycles, the average is $\frac{71}{37} \approx 1.9$
    - Assume **data forwarding** and no branch prediction
        - This time we don't need to stall for data hazard
        - The stalls are still needed for control hazard
        - Number of bubbles produced: $2 \times 6 = 12$
        - CPI: $\frac{37+4+12}{37} \approx 1.4$
    - Assume **forwarding** and **always taken** branch prediction
        - The branch will only be taken once, so 5 mispredictions with each misprediction having 2 squashes
        - CPI: $\frac{37+4+10 \text{ mispredict}}{37}$
    - Assume **forwarding** and **never taken** prediction
        - The branch will only have 2 mis-predictions
        - CPI: $\frac{37+4+2}{37} \approx 1.2$
- Assume no forwarding and **always taken** branch prediction

```
    irmovq 1, %r11      # no stalls
    irmovq 3, %rax      # no stalls
loop:
    subq %r11, %rax     # 3 stalls due to data hazard on rax
                        # 2 stalls due to data hazard on rax
(2 times)
    jg loop             # 2 squashes due to branch
prediction failure
    halt                # no stalls
```

- Consider

```
2    irmovq 9, %rax     # rax = 9 - init value
3    irmovq 1, %rbx     # rbx = 1 - condition
4    irmovq 1, %rcx     # rcx = 1 - decrement

    nop
```

```
loop_start:
7     rrmovq %rax, %rdi  # need 1 stall
      nop
8     subq   %rcx, %rax  # need 2 stalls
      nop
9     subq   %rbx, %rdi  # need 2 stalls (9 times)
10    jg     loop_start      # if (i - 1) > 0
                             # mispredicts 8 times
                             # squashes 2?

loop_end:
13    subq   %r8, %r11
14    subq   %r12, %r13
15    xorq   %r9, %r10
16    mulq   %r8, %r9
      nop
      nop
      nop
17    rrmovq   %r9, %rbx    # need 3 stalls
18    rrmovq   %r11, %r8
```

Feb. 5th - Lecture 14: Pipelining - Parallelism

- What is really inside a single chip?
    - Memory $\iff$ (L2 Cache, L1 Cache, Execution Core)
- Our pipeline is very much simplified
    - Not all ALU operations take a single cycle
        - Division is particularly difficult
    - There can be multiple Functional Units
        - More than one adder/multiplier/divider etc
    - Super Scalar architectures
        - Issue more than one instruction per cycle
    - Out-of-Order Execution and other tricks
        - Reorder instructions so they execute faster
        - move memory reads earlier or dependent instructions later
        - Renaming registers on the fly to avoid "output" dependencies

        - Memory accesses can take many clock cycles, not all accesses take the same time
- What a real Intel execution unit looks like?
    - How instructions are executed
        - multiple instructions are fetched at once
        - Instructions can be reordered, registers can be renamed to avoid name conflicts
        - Instructions wait in the scheduler (also called reservation station) until operands are ready
        - Then they are sent through one of multiple execution units
- Limits to the pipeline
    - clock frequency - as high as possible
        - pipeline depth, heat, clock skew
    - cpi - as low as possible
        - pipeline tricks, amount of ILP (instruction-level parallelism), 1?
    - other types of parallelism
- Some other kinds of parallelism
    - Multitasking (or multiprogramming)
        - Software makes it look like many things are happening at once, but quite likely the operating system is simply rapidly switching among many different jobs
    - Multiprocessing
        - Place multiple processing units (CPUs) on a machine so you can run different programs on each unit
    - Parallel runtimes that can leverage multiprocessing or other forms of parallelism
        - can be figured out either manually or automatically
        - Need a runtime system to dispatch the different tasks to different processing elements
    - Multithreading
        - Allow different threads to run on different processing units
        - Can have multiple software threads that still run on a single HW core
    - Thread-level Parallelism

        - Programmers Express this Explicitly
        - Granularity
        - Exploited by
        - multi-core processor...
    - Hyper Threading
        - Memory latency is slow, so use fast, hardware-implemented, thread switch while waiting for memory
        - Each has N hyper-threads (usually 2)
        - implemented by having N copies of all registers
        - exploited by virtual cores
    - Distributed Systems
        - Multiple machines communicating over a network
        - They don't share memory, each machine runs a different program
        - Each machine has its own OS, managing threads locally
        - Distributed ???
    - Matrix multiplication?
        - Fine-grain parallelism
    - Machine-learning algorithm?
- Flynn's Taxonomy: `x`L`x`D
    - M: multiple, S: single
    - Conventional Uniprocessor: Single Instruction Single Data (SISD)
    - Multicore/Multiprocessor: Multiple Instruction Multiple Data (MIMD)
    - Single Instruction, Multiple Data (SIMD)
        - apply same instruction to many data: matrix multiplication
    - Multiple instruction, single data (MISD)
        - different functional units do different things to the same data

Feb. 10th - Lecture 16: Introduction to Cache

- The Memory Hierarchy

        - Registers (2 KB, 0.3 ns), L1 Cache (640 KB, 1.25 ns), L2 Cache (10 MB, 3.7 ns), L3 Cache (30 MB, 17 ns), L4 Cache, Main Memory (32 GB, 80 ns, 3.15/GB), Solid State (Flash) Drive (200 GB - 1 TB, 0.1 ms, 0.1/GB), Disk Drive (2 - 5 TB, 3 ms, 0.007/GB)
        - We see factors of 10 or 100 in size, performance, and price
        - 1956: $\frac{\text{price of memory}}{\text{price of disk}} \approx \frac{411M}{9200} \approx 44673$

          2024: the ratio is about 450
        - Cost of memory in 1956 vs. 2024: 130 trillion times cheaper; Cost of disk in 1956 vs. 2015: 13 trillion times cheaper
- Caching
    - Definition
        - Colloquially: store away in hiding or for future use
        - Computationally: placing data somewhere it can be accessed more quickly
    - Examples
        - Assembly language: we move data from memory to registers
        - In the hardware: we move data from main memory into memory banks that live on the processor
        - In software: we read things into our program's local buffers and manipulate them there

Feb. 12th - Lecture 17: Cacheline and Performance

- In what unit do I move things into a cache?
    - Spatial locality: having accessed some item x, the tendency to access other items near x in the near future
        - All instructions of a function are close in memory
        - Array elements and fields of a struct are also close to each other
    - The data source has long latency
    - We expect data to have high "spatial locality"
    - Block size: the unit in which data is stored in a cache (64 or 128 bytes)
        - File system caches: 4 KB
        - Object caches: size of the object

- Cache line (block) size tradeoffs
    - Large cache lines can be either good or bad
    - Having a large cache line
        - Pros: Have more data with faster access (if the data exhibit excellent spatial locality)
        - Cons: Cache gets filled up faster
    - Summary
        - suppose cache line size = 8 bytes
        - we divide memory into chunks of 8 bytes each
- How to organize the cache?
    - Cache are split into cache slots, and a slot holds a cache line. **Each cache line can go in one and only one slot**
    - Deciding where to place a cache line
        - Small set of locations in which to store a large set of objects
        - General approach: **hashing** (implementing cache in software, like a file system or a database cache)
        - Hardware constraints: mod by power-of-2 table size
- From address to cache lines
    - Address: Tag + Index (hash) into the cache + Offset in cache line
    - Index: the number of bits in this part are a function of the number of entries in the cache
    - Offset: the number of bits in this part are a function of the cache line size
    - Tag: they disambiguate addresses that have the same index
    - In a hash table: key of cache line - tag; value of cache line -
- Evaluating a Cache
    - Cache hit rate: $\frac{\text{number of cache hits}}{\text{number of cache accesses}}$
    - Average access time:
      $\text{time of 1 hit} \times \text{percent of hits} + \text{time of 1 miss} \times \text{percent of misses}$
    - Counting hits
        - if you touch the same item more than once within a short period, you get a hit (temporal locality)
        - spatial locality

Feb. 14th (Oooooo, Valentine's Day) - Lecture 18: Cache Replacement Policy

- How much is a KByte?
    - Official ISO standard
        - 1 **kilo**byte = 1000 B
        - 1 **kibi**byte = 1024 B
    - Historically, in most of computer science, a KByte is 1024 Bytes
- Associativity Summary
    - A k-way set associative cache with n total slots for lines
        - Has n/k sets, each with k slots
        - Each cache line in memory can go in one and only one set
        - That means
            - a direct-mapped cache is 1-way set associative
            - each set is effectively "its own fully-associative cache" for "its own part of memory"
            - if a set is full, we need a way to decide what to evict when we we bring a new line in
    - The number of sets n/k must be a power of 2. The number of slots per set k and the number of cache lines n may not be
- Eviction in hardware
    - In hardware caches
        - direct mapped: exactly 1 choice
        - k-way set associative: k choices
    - Often use a random replacement policy as other policies would be too expensive to build
- Eviction in software
    - In a perfect world, we would evict the item that is least valuable
    - In the real world, we would like to, but we don't know what that item is
    - Practically, all eviction policies try to approximate the ideal (use heuristics)
        - Least-Recently-Used: find the item that has been unused the longest and get rid of that

        - Least-Frequently-Used: find the item that has been used least frequently and get rid of that
        - Clock - used in virtual memory systems to approximate LRU
        - Something tuned to known access patterns
        - Cool live-updating, real-access-patternn-based, learned policies
- Ideal Eviction Policy: Belady's Algorithm
    - Always evict the item that we use farthest in the future
- Approximating Belady: LRU
    - Evict the item that was used farthest in the past
- Least Frequently Used
    - Keep track of how many times each item is used and **replace the one with the smallest frequency count**
    - When evicted, the count of the item clears to 0
- No perfect policy
    - Assume 2-slot, using only 3 different data items, we can produce an ordering whose hit rate is 0 with LRU replacement
    - e.g. 1, 2, 3, 1, 2, 3, ...

Feb. 24th - Lecture 22: Caching Speedup

- Evaluating Caches
    - Amdahl's Law
        - Originally designed to quantify the opportunities and benefits of parallelism
        - Overall speedup is a function of both
            - The amount by which you improve some part of the system (k)
            - And the fraction of time you spend in that part of the system (α)
        - Given
            - Latency of the original system is T_{old}
            - Can speed up part of a system by a factor of k
            - The part of system being sped up is α

- $T_{\text{new}} = T_{\text{old}} \times (1 - \alpha) + T_{\text{old}} \times \alpha \times \frac{1}{k}$
- The speedup is $\frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1}{(1-\alpha)+\frac{\alpha}{k}}$
- In reality, there are overheads in resource contention, synchronization, communication cost, data transfer cost

- What about writes - How does caching apply writing operations
  - A write cache hit: writing an item, the cache has that item, and the new value is written
    - A hit means we have "dirty" data in the cache, when do we write new data back to the data source?
    - Writeback caching: time passes, and something causes us to evict the line
    - Writethrough caching: the cache asks the data source to write the new value
  - Write Policy Trade-offs
    - Advantages of writeback cache/Disadvantage of writethrough
      - Faster (cache time)
      - Write coalescing: update the same item (or multiple items in a cache line) many times before writing it back to the source
    - Advantages of writethrough cache/Disadvantage of writeback
      - No window of vulnerability
      - Typically simpler to implement
  - A write cache miss: writing an item, the cache does not have that item
    - Write allocate: the cache requests the cache line from data source
    - No Write Allocate
  - Write Allocation Trade-offs
    - Advantages of write-allocate/Disadvantage of no write allocate
      - Faster when modifying multiple values in a cache line
      - Matches read behavior
    - Advantages of no write-allocate/Disadvantage of write-allocate
      - Avoids cluttering the cache with write-only data
      - Avoids consuming a lot of cache ...

Feb. 26th - Lecture 23: Managing and Evaluating Writes

- Cache Recall
    - (Direct Map Cache) Address = Tag + Slot (Index) + Cache line size (Offset)
    - Set Associative Cache
- Anatomy of Addresses and Cache Lines
    - Cache Line Metadata
        - Valid bit: 1 iff contains valid data; if 0, none of the other data/metadata is meaningful
        - Tag: used to determine precisely which address is stored in a cache line - for a hit, the tag bits in the address must match the tag bits in a slot's metadata in the set indicated by the address's index bits
        - (For writeback) Dirty bit: 1 iff line contains dirty data (updates not yet written to data source)
        - (As needed for replacement policy) Replacement metadata
- Write Performance: Hits
    - Write Through Cache: data is written to both the cache and the source
    - Write Back Cache: Data is written only to the cache, but the dirty bit in the cache line metadata is set to 1
- Write Through Cache - Write Miss
    - No Write Allocate
        - Data is written to the source
        - Cache is checked (otherwise we wouldn't know it was a miss) but unchanged
    - Write Allocate
        - Acts like a read miss; read cache line from the source into the cache
        - strange combination: write-through is usually for when simplicity is the goal
        - benefits workloads where read locality mirrors write locality
    - Through/No Allocate

      Example: latency to read/write cache = 2 ns; latency to read/write source = 20 ns

        - latency of write **hit** = 20 ns
        - latency of write **miss** = 20 ns
    - Through/Allocate

      latency of write **hit** = 20 ns

      latency of write **miss** = 40 ns (Write data source, Read data source into cache) (Read and write from data source is sequential)
- Write Back Cache - Write Miss
    - No Write Allocate
        - Write goes to data source, cache is unchanged
        - strange combination: writes to same block, will miss again
        - latency of write miss = 22 ns (Need to attempt to write in cache, then write to data source)

          2 ns + 20 ns
    - Write Allocate
        - Evict nothing or evict a **clean** cache slot

          latency ≈ 1 data source read + 1 cache write
        - Evict a **dirty** cache slot

          latency ≈ 1 data source write + 1 data source read + 1 cache write

          discover the miss, write the evicted cache line, read/update the new cache line
    - Back/Allocate

      latency of write **hit** = 2 ns

      latency of write **miss** + **clean** line evicted = 20 + 2 = 22 ns

      latency of write **miss** + **dirty** line evicted = 20 + 20 + 2 = 42 ns
- Improving write-miss latency
    - None of the designs so far improve write miss latency
    - Write Buffer
        - small, associative buffer
        - on a write miss, the processor writes data into the buffer
            - asynchronously, the buffer does whatever is necessary to complete the write
        - subsequent reads must check the buffer first

- more common with write-back, write-allocate caches

Mar. 3rd - Lecture 25: The Memory Hierarchy, From Arrays to Strided Access

- Spatial Locality and Clustering
    - What aspect of cache design takes advantage of spatial locality?

      Cache line size
    - Under what condition is this effective?

      When we access more than one data per line
    - What does this say about how data should be clustered in memory?

      Associated data, if we access first item of a data, we are likely to access other items of the data
    - Good / Bad example taking advantage of spatial locality

      Good example: Linear Search

      Bad example: Linked list, Binary Search, random array accesses
- Stride: Distance between elements accessed consecutively
    - Measured in elements or bytes. For example, in an array of 4-byte items
        - stride of 1 element is a stride of 4 bytes, and accesses all elements
        - stride of 2 elements is a stride of 8 bytes, and access every other element
- Array Layout in C: Row Major Form
    - (0, 0) | (0, 1) | (0, 2)

      (1, 0) | (1, 1) | (1, 2)
- Column Major Form
    - (0, 0) (1, 0) (2, 0)

      (0, 1) (1, 1) (2, 1)

Mar. 5th - Lecture 26: Multicore, Multiprocessor Caching

- Cache Coherence: what is costly in parallelism
    - Is communication needed?
        - Single-core, single write-back/write-allocate L1 cache
            - Read hit: NO; Read miss: YES; Write hit: NO; Write miss: YES; Evict a **clean** line: NO; Evict a **dirty** line: YES
        - Multi-core, each has a write-back/write-allocate L1 cache, share a single L2 cache, maintain coherency based on valid/dirty bits
            - Read hit: NO; Read miss: YES; Write hit: **YES**; Write miss: YES; Evict a **clean** line: NO; Evict a **dirty** line: YES
        - Multi-core, each has a write-back/write-allocate L1 cache, share a single L2 cache, L1 caches synchronize with MSI
            - Read hit: NO; Read miss: YES; Write hit: **MAYBE**; Write miss: YES; Evict a **clean** line: NO; Evict a **dirty** line: YES
- MSI
    - M: means "modified" but also means "exclusively mine"
    - S: means "shared", clean, may be in multiple caches
    - I: means "invalid", not in this cache
    - A particular state is specific to a particular line in a particular cache
    - Each message sent includes an address' index and tag bits and possibly more
    - A cache operation can fetch data, set a cache line's state, evict a line, and send an "invalidate" request to make other caches evict lines
    - Algorithms that behave otherwise share a lot of memory and so communicate a lot and thus do a bad job of parallelism
    - E: means "exclusive"
- Directory Based Coherence
    - A (centralized) directory stores all the state information
    - Processors consult the directory before updating caches
        - information in the directory is similar to the cache states

Mar. 7th - Lecture 27: File Systems, From the API to the Disk

- File - a stream of bytes
    - The OS maintains an **offset**
    - Reading and writing proceeds from the current offset
    - What information does the kernel need to keep and what role does the file descriptor play?
- What we have discovered so far
    - Each open file needs an offset for reading and writing
    - The file descriptor must allow us to find: the file's data (persistent), the file's metadata (persistent), the file offset
    - Must be able to determine to which logical block a byte belongs (arithmetic)
    - Must be able to map the logical block to the actual disk location (persistent)
    - Directories must map names to a file or directory (persistent)
    - Must be able to find the first (root) directory (it depends)
    - Need to associate access permissions with a file (persistent)
- A steam of bytes → blocks
    - A file is a logical stream of bytes, but data can only be read from disk one block at a time
    - Reading a byte requires that we retrieve a block from the disk
- Offset → byte - which block?
    - Offset divided by block size = block number, but
        - this block number is relative to the start of this file
        - there are lots of files on the disk and lots of blocks/sectors as well
- `open(filename, flags, mode)`
    - A file consists of blocks
    - We need a map from logical block numbers to disk addresses
    - ...
- File name hierarchy
    - form /A/BC/DE/file

        - consists of directories and files
        - a directory maps a name to either another directory or to a file
- File system behaviour
        - if both are allowed to read a file in the file system, should we be able to share the data?
        - if two processes are reading (writing) the same file, should they be using the same file offset?
        - if a process opens the same file twice, have one or two file descriptor?
        - two threads using the same file descriptor, should be using same offset?
        - if one process creates another process, should the child inherit parent's file descriptor? use same offset?

Mar. 10th - Lecture 28: Fun with File Descriptors

- Per-process file descriptor table + Shared across processes open file table and `vnode` table
- Parent and Child have the same File Descriptor table
- `pid_t fork(void)`
        - Creates a new process almost identical to the process that called it
                - the parent's return value will be the pid of the child process
                - The child process's return value will be 0
        - Immediately after the fork call, both processes should check their return values and proceed accordingly
                - The child process often calls a member of the exec family
                - The parent might wait on the child or do something else
- Exec and friends
        - `int execve(const char* path, char* const argv[], char* const envp[])`
                - Typically path is the pathname of a command you want to execute
                - argv is an argument vector -- it is what is passed to main
                - envp is an environment: the environment is a set of name/value pairs that are frequently used to communicate environmental information to processes...
        - `int execvp(const char* path, char* const argv[])`

            - if the parameter path does not begin with /, or ., or .., then it searches for parameter path in each directory listed in the PATH environment variable (just like the shell does)
- dup
    - `int dup(int fd)`, or `int dup2(int fd1, int fd2)`
    - dup2 -- `FDT[fd2] = FDT[fd1]`
    - When might you use it?
        - when want to capture a program's output, can use `ls > my-stdout-file.txt`
        - If you dup into an fd that is open, this automatically closes the file that had been using that fd
        - Even though standard IO is not a system call, ...
- Other fun facts
    - can redirect both stdout and stderr, can redirect stdin, can redirect stdout but also let it display on the screen, can redirect stdout and append to an existing file

Mar. 12th - Lecture 29: Representing Files on Disk

- File system design goals and constraints
    - long-lived and robust
        - many files created, deleted, extended and truncated over time
        - performance should not degrade with time
    - general purpose
        - different file sizes, files can be sparse
        - different access patterns: sequential and arbitrary
    - performance dictated by storage media hardware (rotating disk or SSD)
        - file is collection of disk blocks that store its data and meta-data
        - seeking to a block can be much slower than transferring all of its data
        - where blocks are on the disk can really matter
- Sparse files
    - purpose

        - some files represent data placed at particular offsets to a starting point
        - potentially with large gaps in between
        - rather than continuous streams of data
    - implementation consideration
        - e.g. write one byte at offset 0 and another at offset $2^{30} - 1$
        - file's size is 2^{30} bytes
        - but storing that data requires only 2 disk blocks for the file's data (allocating 2^{30} bytes worth of blocks would waste valuable disk space)
- Layers of abstraction
    - Application: stream of bytes
    - File: sequence of logical blocks, LBN
    - Disk: sequence of physical blocks
- Basic data structures and concepts
    - super block
        - meta-data for entire file system
        - stored at specific disk locations
        - to mount a file system OS must be able to read its super block, so these may be replicated multiple places on disk
    - inode
        - on-disk meta-data that describes a file
        - stores: (root of) mapping to disk block number and some other meta-data (file size, file permissions, etc)
        - does not have a symbolic, human-readable name
    - inumber
        - internal "name" of an inode
        - file system can map inumber to disk block for that file's inode
        - `ls -i` to see file inumbers
- Store something in a new file
    - Steps
        - create: allocate a new inode, thus assigning the file an inumber
        - write: allocate disk blocks and write data into those blocks

- Strategy 1: Single-Extent-Based Allocation
    - Extent
        - Definition: variable-sized contiguous collection of disk blocks
        - Use: one extent per file, stores all of a file's data
    - LBN to PBN map
        - simple and small; just store:

          block number of extent's first block

          total number of blocks in the extent

          or just the size of the file since we can divide to get the number of blocks
    - Pros
        - inode is small for all file sizes
        - sequential access is optimized, matching hardware characteristics
        - random access needs ≤ 2 reads to get disk data (inode + data block)
    - Cons
        - handles sparse files poorly
        - does not match file POSIX API (when creating a file in Unix, you don't tell the OS how big it will be)
        - does not handle file extension or truncation well
        - causes **external fragmentation**
    - External fragmentation
        - extents are variable-sized, created and deleted randomly/arbitrarily
        - over time, large, contiguous free spaces become scarce
        - they get fragmented into many smaller space
        - even though there is plenty of room for our new file overall, no contiguous free space is big enough
- Strategy 2: Block-based allocation
    - Blocks

        - fixed size units of allocation, thus **eliminating external fragmentation**
        - a file might require many blocks
    - LBN to PBN map
        - must store block number of **every block** of the file
        - So, inodes for large files must be large
    - Pros
        - no external fragmentation
        - matches Unix API: files start out empty
        - easy to extend and truncate
        - handles sparse files well
    - Cons
        - not optimized for sequential access
        - ...
    - Picking a block size
        - Big blocks (pros/cons)
            - better performance with sequential accesses
            - smaller inode block maps
            - more internal fragmentation
            - wastes space

Mar. 14th - Lecture 30: Representing files on disk

- From persistent media to memory is really slow, we will use main memory as a cache for persistent data
    - The buffer cache is

      an in-memory cache of persistent data pages

      managed by the file system (part of the OS)

      a software cache (managed by software)
- Constraints on Indexes
    - the index itself must be in persistent storage, or we could not find the blocks of our files on reboot
    - indexes "live" in disk blocks, taking space

        - strict single-extent-based is impractical (the inode has the disk address of the first block in the extent and the length of the extent)
- File representation: flat index
    - the index is a fixed sized array, thus

      the index consumes some number of disk blocks

      growing the index is not possible

      there is some maximum file size (true for most indexes, it's worse here)
    - example

      4096 B blocks, 4 B block numbers, inode: 16384 index entries

      maximum file size $4096 \times 16384 = 64$ MB

      4096 byte blocks @ 4 bytes/entry = 1024 entries/block

      16384 entries @ 1024 entries/block = 16 index blocks
    - Pros

      can prevent sparse files, sequential and random access are efficient in terms of metadata blocks that need to be fetched
    - Cons

      either we have to allocate really big indices or we impose unreasonable constraints on file size

      small and large files consume exactly the same amount of index space
- File representation: Multi-level index
    - The inode stores a disk address

      it may refer to a data block (for a file with one block), in some versions, the inode always references a separate index root, even for 1 (or 0) block files

      if there is more than one data block, it refers to a metadata block packed with data addresses of blocks - an indirect block

      when that fills up, switch to storing the address of a block packed with addresses of indirect blocks: a double-indirect block
    - Pros

      can represent sparse files

      if block size >> disk address size, an access requires few intermediate blocks

      can grow easily

        - Cons

          even for 2-block files, we perform 2 IOs

          the file size determines how deep the index is
- File representation: Hybrid index
    - The index is a fixed-sized array, small enough to fit in the inode

      most entries are direct pointers

      a few entries are single, double, or triple-indirect
    - Pros

      small index

      efficient for small files and sparse files

      files can grow large

      good for random/sequential if block size >> PBN size
    - Cons

      more complicated to map from LBN to PBN

Mar. 17th - Lecture 31: File Systems - Naming

- What problem to solve?
    - Given a name like `/Users/patrice/quiz4-answers.org`
    - `/` means the root directory, ...
- Hierarchical naming
    - generalized tree structure
    - Pros
        - makes names local to a directory
        - reuses whatever file implementation we have for directories
    - Cons
        - Lookup is an iterative (and potentially expensive operation)
- Directory representation
    - they are regular files with a special format
    - a field in a file's inode indicates that the file is of type directory
    - A **directory entry** is a **pair of name** and associated **inode number**
        - User programs can read directories like files

        - Only OS can write directories (users can corrupt the directory structure)
    - directories can contain names of other directories: subdirectories
    - absolute pathname vs. relative pathname
- Traditional Directory Implementation
    - A directory is a file with structured contents (directory entries: `struct dirent`)
        - name, inode, type
    - ...
    - In POSIX, every directory has
        - . entry refers to itself
        - .. refers to parent directory
- The Directory Entry Structure
    - The details of `struct dirent` are specific to a file system
- The root directory (name and inode number only)
    - The root directory is the only time where the . and .. have the same inode number
- Walking a directory path
    - Given a path `/C1/C2/C3`

      Start at the root directory if the path begins with / or the current working directory otherwise

      1 Let `inum` = starting inumber; current component = next component

      2 ...

      3 ...

      4 Find the associated inumber

      5 Read the inode for that inumber

      if this is the last component of the path, we are done

      if it is not a directory and we have more components...
- Hard links
    - Creating a new file
        - creates a link between "its" name and inode (every file name is just a hard-link, the file's real name is its inumber)

        - Adding a hard link: `ln <target-file> <new-name>`
            - Creates an extra name for the file: a new link from a new name to the inode
            - inodes are reference counted, ...
        - Removing a link
            - think of as removing a file, but not quite that
            - `rm foo` ...
            - ...
- Symbolic links
    - a file that contains a pathname to another file

      `ln -s <target-file> <new-name>`
    - Removing a symbolic link, does nothing to the target file
    - Removing the file a symbolic link points to (does nothing to the link, becomes dangling pointer)

Mar. 19th - Lecture 32: Ext2 File System

- File System Metadata: how to find all parameters of the file system
    - `struct ext2_super_block`
    - similar to v6 file system
    - ext2 specific: `s_blocks_per_group` etc
- Disk Layout: what goes where, and how big is it
    - with block size 1KB, superblock is at block 1, block 0 is reserved, and block 2 is the block descriptor table
    - with larger block size, the address of superblock remains the in same place, which means it is in block 0
    - Blocks are grouped into block groups, and within a block group
        - may have super block and block descriptor table (only in groups 0, 1, and multiples of 3, 5, 7)
        - then we have a data block bitmap, and inode bitmap
        - followed with inode blocks
        - and then actual data blocks

        - one block's size is configurable, typically in current systems: 4KB
- Free Space Management: how to allocate space and how to deallocate space
    - in the data block bitmap, each block is 1 bit
    - given the bitmap, we need to build data structures to quickly find a free block, we need to find a sequence of consecutive free blocks
    - choose group size such that all blocks in a group are on the same cylinder (avoid seeking to move between them)
    - 512 byte blocks = 4096 bits = 1 bitmap block representing 4096 inodes
    - 512 byte blocks + 64 byte inodes = 8 inodes per block = need 512 inode blocks
    - 4096 bits = 4096 data blocks + 512 bytes per block = 2 MB of data in data group
- Logical-Physical Mapping: how do I find a particular block within a file
    - per-file metadata: 15 blocks = 12 data blocks + 1 indirect block + 1 double indirect block + 1 triple indirect block
    - maximum file size
        - size of physical block number: 4 bytes
        - say block size is 4 KB
        - number of PBNs per block is 1024
        - direct reference: 1×4 KB
        - indirect reference: 1024×4 KB = 4 MB
        - double indirect: 1024×4 MB = 4 GB
        - triple indirect: 1024×4 GB = 4 TB
        - Total: 4 TB + a bit more
- Directory Structures: how do I make directories/files work
    - directory entries are 4 byte aligned

```
struct ext2_dir_entry {
    _le32 inode;    // inode number
    _le16 rec_len;  // directory entry length
    _le16 name_len; // name length
    char name[256]; // file name
}
```

        - Building a directory
            - the first 3 entries are fixed size part
            - the name of the entry is variable size, and pad the entry (i.e. increase rec_len as necessary) to make each entry 4-byte aligned
- What came after
    - ext3: 2001, added journaling (helps with file system recovery and reduces chances of data loss in case of a system crash), journaling is similar to the logs kept by database systems

      max sizes: 16 GB to 2 TB per file, 4 TB to 32 TB for the file system
    - ext4: 2008, added
        - extents (can have a list of arrays [stored consecutively] of blocks)
        - transparent encryption
        - improvements to how blocks are allocated
        - H-tree indices for very large directories

      max sizes: 16 GB to 16 TB per file, up to 106 TB for the file system

Mar. 24th - Lecture 34: Virtual Memory - Achieving Process Isolation

- Fundamentals
    - Programs run in different processes
    - Each process has its own address space - the locations in memory the process can access
    - Address spaces provide **process isolation**
        - each process behaves as if it controls the entire resources
        - anything one process does should not affect what other processes do, unless the processes agree (e.g. set up a communication channel)
- Illusion
    - 2 different kinds of address
    - Physical address
        - refer to specific locations in memory (DRAM)

        - Virtual address
            - these are the addresses that programs use
            - the hardware and software collaborate to map each virtual address to physical address
        - A machine can have less physical memory than the size of the virtual address space (also a lie)
            - can have 64-bit architecture supporting program with virtual addresses measuring in petabytes, without having a machine with that much memory
- How?
    - If MMU takes the virtual address and simply passes it through, then VA = PA
    - CPU does not know if the address it is using is a VA or PA, nor does it care
- How does it help?
    - Protection: data from different processes are segregated on memory, managed by MMU
- VM: a hardware/software partnership
    - we need hardware support to provide virtual memory
        - speed! invoking the OS on every address access would be much too slow
    - hardware
        - provides a fast mechanism to map a virtual address to a physical address
    - software (OS)
        - sets up the mappings that the hardware uses
        - manages the allocation of physical memory
        - implements policies about how memory is shared
- System calls that manipulate address spaces
    - `pid = fork()`
    - `int execve(const char* path, char* const argv[], ...)`
    - `int execvp(const char* path, char* const argv[])`

Mar. 26th - Lecture 35: Transferring Control to the OS

- VM: a hardware/software partnership
    - we need hardware support to provide VM
        - need virtual-to-physical address translation
        - Software is too slow
    - software
        - sets up the mappings that the hardware will execute
        - manages the allocation of physical memory
        - implements policies about how memory is shared
- Protection boundaries
    - modern hardware has multiple **privilege levels** or **modes**
    - processors typically provide 2+ different modes
        - **user mode**: how regular programs run
        - **kernel mode** or **supervisor mode**: how the OS runs
    - the mode in which a piece of software is running determines
        - what instructions may be executed
        - how addresses are translated
        - what memory locations may be accessed (enforced through translation)
- Constraining the mapping
    - we want the OS to be allowed to do things that normal processes cannot: interact with devices, **read/write any process's memory**
    - Different parts of an address space support different operations
        - read-only text/data
        - data should not be executed
- The hardware
    - map a triple of: virtual address, type of access, privilege level - to a physical address
    - either produce a physical address

      or fault due to invalid virtual address, invalid access, inappropriate process privilege, OS needs fault to force it to find it first
- MIPS

        - two privilege modes: User, Kernel
        - User
            - access to CPU/FPU registers
            - access to flat, uniform virtual memory address space
        - Kernel
            - access to everything accessible in user mode
            - access to memory mapping hardware and special registers
            - can issue privileged instructions
- How do we change the privilege levle
    - Two new instructions: raise privilege level (leave it at kernel) and lower it (leave it at user)
    - To transfer control from less privileged to more privileged code, we use a **trap**
- Kinds of traps
    - System calls: an application asks the OS to act on its behalf
    - Exceptions: an application unintentionally does something requiring OS assistance
    - Interrupts: an asynchronous event
- The OS must guarantee that it eventually runs
    - processors have timers
    - timers have the property that they either count up or down then generate an interrupt
    - selecting a timer interval is part of the **scheduling problem**
        - what process should get to run first
        - what policy should it use to share the processor among multiple processor
        - how long should a process run
    - timer interval (quantum): short enough that the system is responsive, long enough to keep the fraction of our time handling timer interrupts small

Mar. 28th - Lecture 36: How does Virtual Memory work?

- Process isolation - translation/fault supports virtual addressing, trust the resource manager OS (privileged)
- Virtual Memory mapping
    - Goal: we need a map from triples to physical address
    - (VA, access, priv) → PA || Fault
    - physical addresses are not necessarily contiguous or ascending
    - we would like to avoid fragmentation
- First approach
    - store a mapping for every byte in the virtual address space
    - critique
        - mapping table is huge
- Better idea
    - divide virtual address spaces and memory into blocks, we call them **pages**
    - store one mapping for each page
    - critique
        - how large are the pages, how to implement this map?
    - Dividing the bits in an address (x86-32): pages are 4 KB - 12 bits to represent 4 KB
        - The address will be an offset in the page
        - after we map into the physical address space, the offset will remain to be same
        - 20 bit virtual page number + 12 bit offset → 20 bit physical page number + 12 bit offset
    - What if we have a 32-bit VAS, but our machine has more than 4 GB of memory - no problem
    - What if we have a 64-bit VAS while our machine has much less main/physical memory DRAM
        - it's fine (x86-64 runs with a 48-bit address space), processes don't use the entire 48-bit address space, and we can run a process without having all its pages in memory
- Example

        - An instructor machine: 4 KB page size, of the 64-bits in the virtual address space (12 bits are page offset, 16 bits are unused, 36 bits are the virtual page number)
        - 16 GB memory - we need 34 bits of physical page number
        - 48-bit VAS - 34-bit PAS
        - We can have the MMU **cache some mappings**
- Translation Lookaside Buffers (TLB)
    - the simplest form of MMU is basically just a cache of translations (and permissions and protections)
    - VPN - PPN - Permissions - Mode
    - If a **user program** tries to access the page without the proper permission, it will be a **fault**
- Takeaway
    - Max VPN and PPN can be different
    - Virtual pages and physical pages are the same size
    - TLB is a fast, **hardware** cache

Mar. 31st - Lecture 37: Virtual Memory - x86

- So far
    - virtual pages and physical pages are the same size
    - page size is set by the hardware
    - maximum number of physical pages and therefore the size of physical page numbers are also determined by hardware
    - actual number of physical pages is specific to a particular machine
    - virtual page numbers and physical page numbers can be different sizes
- Page Table: Pre-class
    - one flat array, one page-table per process
    - OS must track page tables for all running processes
- Address space observation
    - we know the page size, the address space size, how much our program uses, most programs use just a bit / a lot of address space
    - Most processes won't use all or even most of their address space

        - many only use a few pages
        - page tables are **sparse** (most entries are invalid)
- Sparse address space representation
    - require

      simple enough data structure to work with in hardware, efficient for a large address space, supports gaps in the address space: sparse address spaces
    - similar to problem of representing files: be efficient representing small, large and sparse files
    - essentially use multi-level index, but choose the sizes of those indexes to match the hardware
        - height of the tree is fixed
        - all blocks in the tree match our virtual memory page size
        - we use specific bits in an address to index into each level of the tree
        - each page in memory is 4 KB aligned
- File index to page table
    - instead of an inode, we have hardware register: CR3
    - every indirect block is now a **page table**
    - Page tables always contain PTEs (L1 Page Table), can include other in-memory physical pages and sometimes reference disk blocks
    - x86-64 has 4 levels of page tables
    - all page tables are indexed by bits in the virtual address
    - PTEs are 8 bytes
- x86 Address Translation
    - 64-bit virtual address = 12 offset bits + 16 unused bits + 36 bit virtual page number (4 levels $\times$ 9 bits, L1, L2, L3, L4)
    - L1 $\rightarrow$ L2 $\rightarrow$ L3 $\rightarrow$ L4 $\rightarrow$ physical page number to access
- A look at PTEs
    - 0 - P (child table present in memory or not)

      1 - R/W (read-only or read/write permissions for all reachable pages)

      2 - U/S (user or supervisor mode for all pages reachable by the child table)

3 - WT (write through or write-back for child page table)

4 - CD (are we allowed to cache the page table we reference)

5 - A (reference bit)

6 - D (dirty bit)

7 - PS (page size)

8 - G (global page, is not evicted from TLB on a process switch)

9-11 - Unused

12-51 - Page table physical base address

52-62 - Unused

63 - XD (Execute disable)

- if P = 0, OS can do whatever it wants with 1-63 bits

Apr. 7th - Lecture 40: Virtual Memory Summary & 2-handed Clock Algorithm

- Big picture
    - purpose of virtual memory: process isolation
    - how does VM work: hardware-software partnership, hardware translates whatever addresses it can, software takes over where hardware leaves off
    - other benefits of VM
        - process address space can be larger than physical memory
        - call loading of the VM pages as they are needed **demand paging**
        - Main memory acts like a cache for the sum total of all processes' address spaces (need **replacement policies**)
- Why clock and not LRU
    - OS does not have visibility into the access stream, so it cannot identify LRU page (tracking it would be expensive and impractical)
        - VM pages can be accessed without OS intervention
        - The OS must track which pages are cached and which can/should be evicted
- How to handle modified pages
    - we can add a dirty bit to each PTE that is set on write operations

        - If a page is dirty and will be paged out, it needs to be written to disk before eviction
- 2-handed clock
        - one hand for replacement, one hand for writeback
        - keep the writeback hand a specific distance ahead of the replacement hand, as it move forward, it sends dirty pages to the disk controller so they can be written back to disk
        - critical point: we can do this writing while the disk controller is not otherwise occupied, **in parallel** with work happening in the cores
- Lots of policy decisions
        - how far ahead the writeback hand should be
        - how many pages should we write at a time
        - when should we move the write hand