CS 61C: Great Ideas in Computer Architecture

Multiple Instruction Issue/Virtual Memory Introduction

Guest Lecturer/TA: Andrew Luo

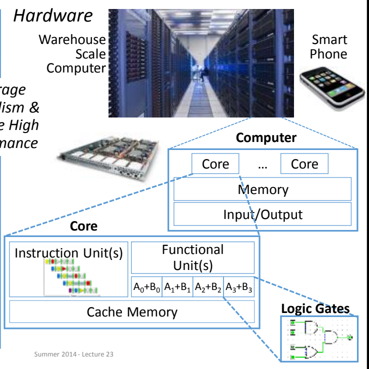
Great Idea #4: Parallelism

Software

- Parallel Requests
  Assigned to computer
  e.g. search "Garcia"
- 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 descriptions
  All gates functioning in parallel at same time

Leverage Parallelism & Achieve High Performance

Hardware



Review of Last Lecture (1/2)

- A hazard is a situation that prevents the next instruction from executing in the next clock cycle
  - Structural hazard
    - A required resource is needed by multiple instructions in different stages
  - Data hazard
    - Data dependency between instructions
    - A later instruction requires the result of an earlier instruction
  - Control hazard
    - The flow of execution depends on the previous instruction (ex. jumps, branches)

Review of Last Lecture (2/2)

- Hazards hurt the performance of pipelined processors
  - Stalling can be applied to any hazard, but hurt performance… better solutions?
  - Structural hazards
    - Have separate structures for each stage
  - Data hazards
    - Forwarding
  - Control hazards
    - Branch prediction (mitigates), branch delay slot

Agenda

- Multiple Issue
- Administrivia
- Virtual Memory Introduction

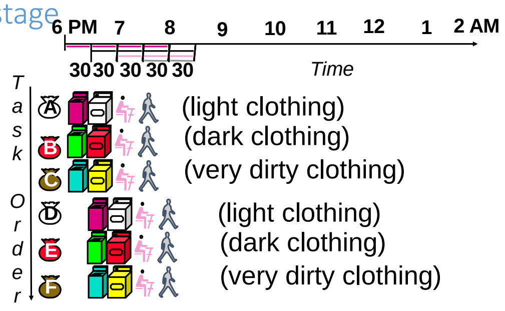
Multiple Issue

- Modern processors can issue and execute multiple instructions per clock cycle
- CPI < 1 (*superscalar*), so can use *Instructions Per Cycle* (IPC) instead
- e.g. 4 GHz 4-way multiple-issue can execute 16 billion IPS with peak CPI = 0.25 and peak IPC = 4
  - But dependencies and structural hazards reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler reorders independent/commutative instructions to be issued together
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines pipeline and chooses instructions to reorder/issue
  - CPU can resolve hazards at runtime

Superscalar Laundry: Parallel per stage



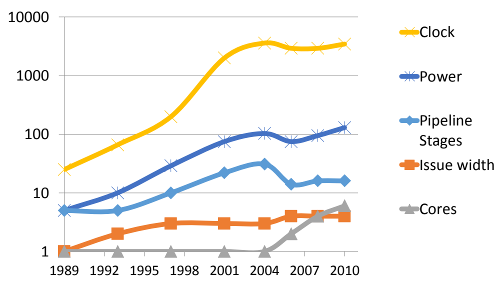
- More resources, HW to match mix of parallel tasks?

Pipeline Depth and Issue Width

- Intel Processors over Time

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Cores	Power
i486	1989	25 MHz	5	1	1	5W
Pentium	1993	66 MHz	5	2	1	10W
Pentium Pro	1997	200 MHz	10	3	1	29W
P4 Willamette	2001	2000 MHz	22	3	1	75W
P4 Prescott	2004	3600 MHz	31	3	1	103W
Core 2 Conroe	2006	2930 MHz	12-14	4*	2	75W
Core 2 Penryn	2008	2930 MHz	12-14	4*	4	95W
Core i7 Westmere	2010	3460 MHz	14	4*	6	130W
Xeon Sandy Bridge	2012	3100 MHz	14-19	4*	8	150W
Xeon Ivy Bridge	2014	2800 MHz	14-19	4*	15	155W

Pipeline Depth and Issue Width



Static Multiple Issue

- Compiler reorders independent/commutative instructions to be issued together (an "issue packet")
  - Group of instructions that can be issued on a single cycle
  - Determined by structural resources required
  - Specifies multiple concurrent operations

Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - *No* dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with `nops` if necessary

Dynamic Multiple Issue

- Used in "superscalar" processors
- CPU decides whether to issue 0, 1, 2, … instructions each cycle
  - Goal is to avoid structural and data hazards
- Avoids need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions *out of order* to avoid stalls
  - But commit result to registers in order
- Example:

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
subu  $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start `subu` while `addu` is waiting for `lw`
- Especially useful on cache misses; can execute many instructions while waiting!

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g. cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined by I/O
- Different implementations of an ISA have different latencies and hazards
  - Forward compatibility and optimizations

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right and roll back if necessary
- Examples:
  - Speculate on branch outcome (Branch Prediction)
    - Roll back if path taken is different
  - Speculate on load
    - Load into an internal register before instruction to minimize time waiting for memory
- Can be done in hardware or by compiler
- Common to static and dynamic multiple issue

Not a Simple Linear Pipeline

3 major units operating in parallel:

- Instruction fetch and issue unit
  - Issues instructions *in program order*
- Many parallel functional (execution) units
  - Each unit has an input buffer called a *Reservation Station*
  - Holds operands and records the operation
  - Can execute instructions *out-of-order (OOO)*
- *Commit unit*
  - Saves results from functional units in *Reorder Buffers*
  - Stores results once branch resolved so OK to execute
  - Commits results *in program order*

Out-of-Order Execution (1/2)

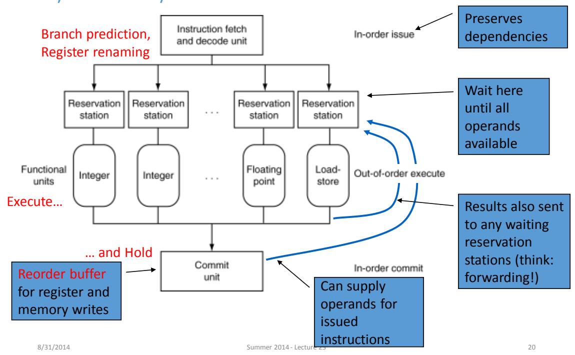
Can also **unroll loops in hardware**

1) Fetch instructions in program order (≤ 4/clock)
2) Predict branches as taken/untaken
3) To avoid hazards on registers, *rename registers* using a set of internal registers (≈ 80 registers)
4) Collection of renamed instructions might execute in a *window* (≈ 60 instructions)

Out-of-Order Execution (2/2)

5) Execute instructions with ready operands in 1 of multiple *functional units* (ALUs, FPUs, Ld/St)
6) Buffer results of executed instructions until predicted branches are resolved in *reorder buffer*
7) If predicted branch correctly, *commit* results in program order
8) If predicted branch incorrectly, discard all dependent results and start with correct PC

Dynamically Scheduled CPU

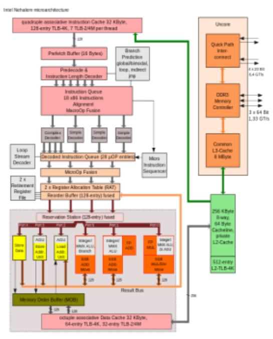

Branch prediction, Register renaming
Instruction fetch and decode unit
In-order issue
Preserves dependencies
Reservation station
Reservation station
Reservation station
Reservation station
Wait here until all operands available
Functional units
Execute…
Integer
Integer
Floating point
Load-store
Out-of-order execute
Results also sent to any waiting reservation stations (think: forwarding!)
… and Hold
Reorder buffer for register and memory writes
Commit unit
In-order commit
Can supply operands for issued instructions

Out-Of-Order Intel

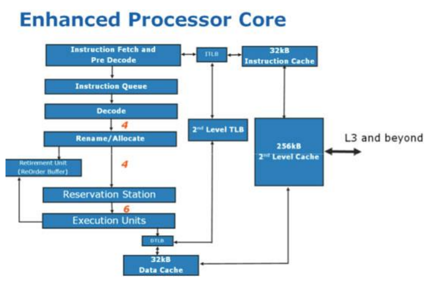
- All use O-O-O since 2001

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25 MHz	5	1	No	1	5W
Pentium	1993	66 MHz	5	2	No	1	10W
Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Core 2 Conroe	2006	2930 MHz	12-14	4*	Yes	2	75W
Core 2 Penryn	2008	2930 MHz	12-14	4*	Yes	4	95W
Core i7 Westmere	2010	3460 MHz	14	4*	Yes	6	130W
Xeon Sandy Bridge	2012	3100 MHz	14-19	4*	Yes	8	150W
Xeon Ivy Bridge	2014	2800 MHz	14-19	4*	Yes	15	155W

Intel Nehalem Microarchitecture

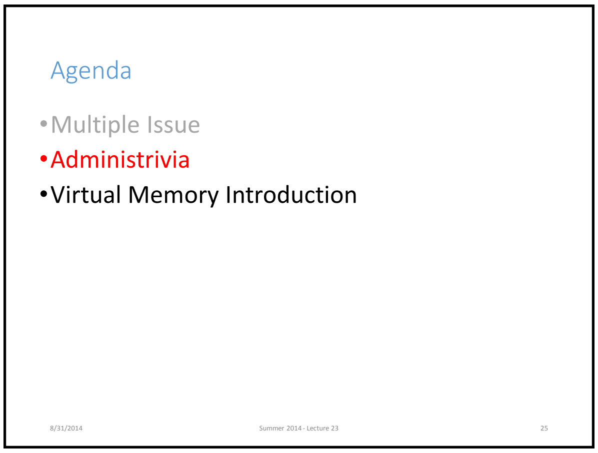


Intel Nehalem Pipeline Flow



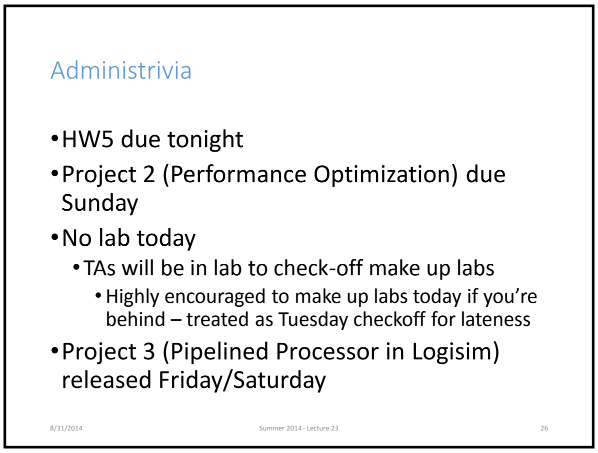
Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g. pointer aliasing (restrict keyword helps)
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well



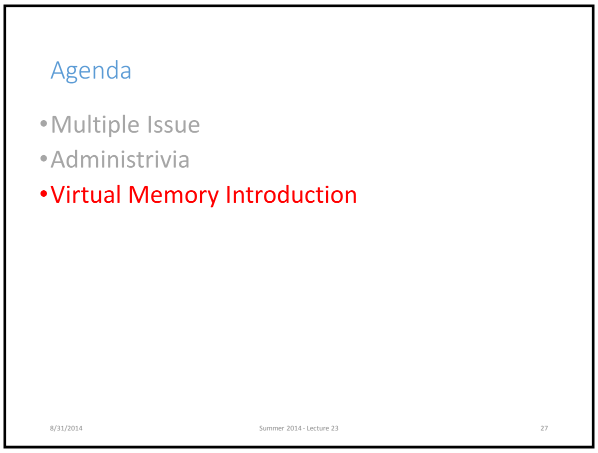
Agenda

- Multiple Issue
- Administrivia
- Virtual Memory Introduction



Administrivia

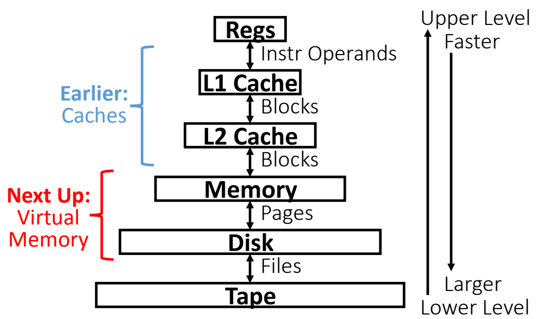
- HW5 due tonight
- Project 2 (Performance Optimization) due Sunday
- No lab today
  - TAs will be in lab to check-off make up labs
    - Highly encouraged to make up labs today if you're behind – treated as Tuesday checkoff for lateness
- Project 3 (Pipelined Processor in Logisim) released Friday/Saturday

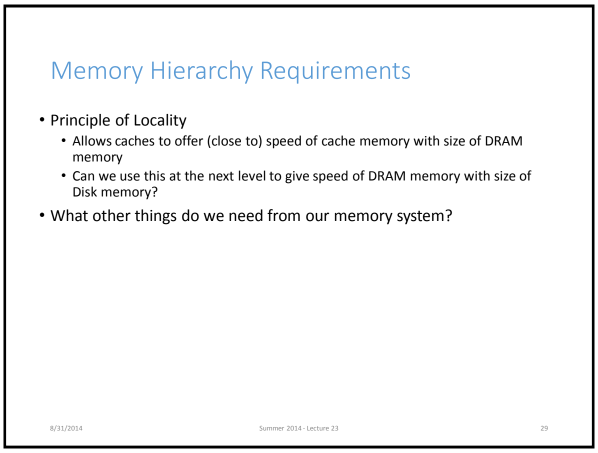


Agenda

- Multiple Issue
- Administrivia
- Virtual Memory Introduction

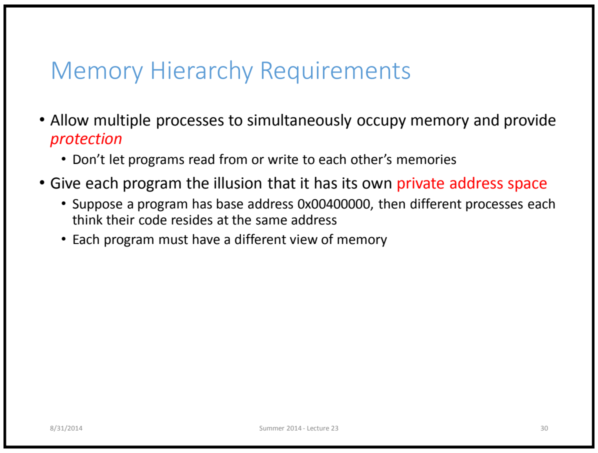
Memory Hierarchy





Memory Hierarchy Requirements

- Principle of Locality
  - Allows caches to offer (close to) speed of cache memory with size of DRAM memory
  - Can we use this at the next level to give speed of DRAM memory with size of Disk memory?
- What other things do we need from our memory system?



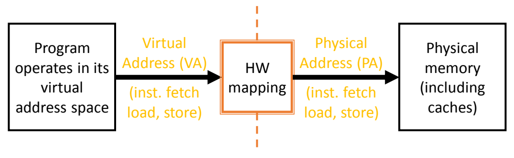
Memory Hierarchy Requirements

- Allow multiple processes to simultaneously occupy memory and provide *protection*
  - Don't let programs read from or write to each other's memories
- Give each program the illusion that it has its own private address space
  - Suppose a program has base address 0x00400000, then different processes each think their code resides at the same address
  - Each program must have a different view of memory

Virtual Memory

- Next level in the memory hierarchy
  - Provides illusion of very large main memory
  - Working set of "pages" residing in main memory (subset of all pages residing on disk)
- **Main goal:** Avoid reaching all the way back to disk as much as possible
- **Additional goals:**
  - Let OS share memory among many programs and protect them from each other
  - Each process thinks it has all the memory to itself

Virtual to Physical Address Translation


Program operates in its virtual address space
Virtual Address (VA)
(inst. fetch load, store)
HW mapping
Physical Address (PA)
(inst. fetch load, store)
Physical memory (including caches)

- Each program operates in its own virtual address space and thinks it's the only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware gives virtual $\rightarrow$ physical mapping

VM Analogy (1/2)

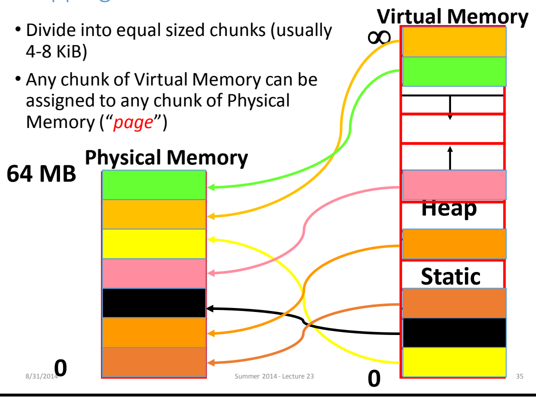
- Trying to find a book in the UCB library system
- Book title is like *virtual address (VA)*
  - What you want/are requesting
- Book call number is like *physical address (PA)*
  - Where it is actually located
- Card catalogue is like a *page table (PT)*
  - Maps from book title to call number
  - Does not contain the actual that data you want
  - The catalogue itself takes up space in the library

VM Analogy (2/2)

- Indication of current location within the library system is like *valid bit*
  - Valid if in current library (main memory) vs. invalid if in another branch (disk)
  - Found on the card in the card catalogue
- Availability/terms of use like *access rights*
  - What you are allowed to do with the book (ability to check out, duration, etc.)
  - Also found on the card in the card catalogue

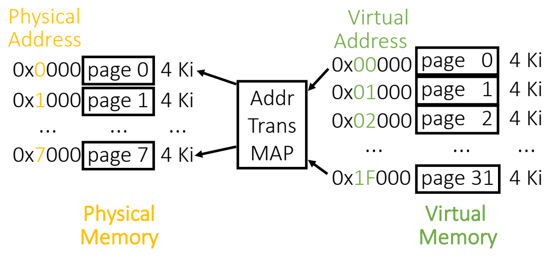
Mapping VM to PM

- Divide into equal sized chunks (usually 4-8 KiB)
- Any chunk of Virtual Memory can be assigned to any chunk of Physical Memory ("*page*")



Paging Organization

- Here assume page size is 4 KiB
  - Page is both unit of mapping and unit of transfer between disk and physical memory

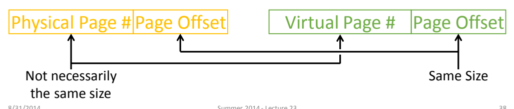


Virtual Memory Mapping Function

- How large is main memory? Disk?
  - Don't know! Designed to be interchangeable components
  - Need a system that works regardless of sizes
- Use lookup table (*page table*) to deal with arbitrary mapping
  - Index lookup table by # of pages in VM (not all entries will be used/valid)
  - Size of PM will affect size of stored translation

Address Mapping

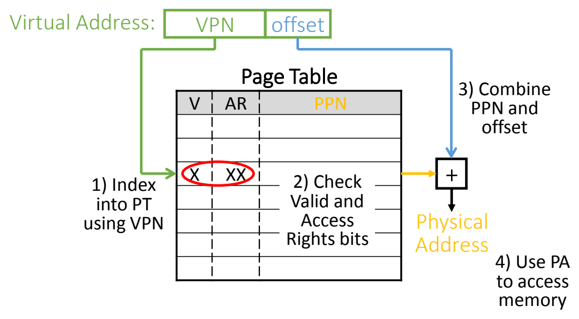
- Pages are aligned in memory
  - Border address of each page has same lowest bits
  - Page size (P bytes) is same in VM and PM, so denote lowest PO = $\log_2(P)$ bits as *page offset*
- Use remaining upper address bits in mapping
  - Tells you which page you want (similar to Tag)



Address Mapping: Page Table

- **Page Table functionality:**
  - Incoming request is Virtual Address (VA), want Physical Address (PA)
  - Physical Offset = Virtual Offset (page-aligned)
  - So just swap Virtual Page Number (VPN) for Physical Page Number (PPN)
- **Implementation?**
  - Use VPN as index into PT
  - Store PPN and management bits (Valid, Access Rights)
  - Does NOT store actual data (the data sits in PM)

Page Table Layout


Virtual Address:
VPN
offset
Page Table
V
AR
PPN
X
XX
1) Index into PT using VPN
2) Check Valid and Access Rights bits
3) Combine PPN and offset
+
Physical Address
4) Use PA to access memory

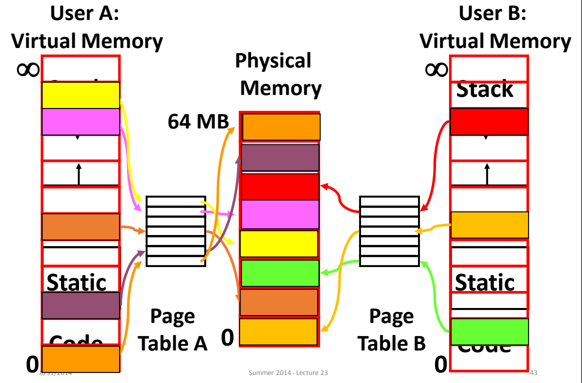
Page Table Entry Format

- Contains either PPN or indication not in main memory
- Valid = Valid page table entry
  - 1 → virtual page is in physical memory
  - 0 → OS needs to fetch page from disk
- Access Rights checked on every access to see if allowed (provides protection)
  - *Read Only:* Can read, but not write page
  - *Read/Write:* Read or write data on page
  - *Executable:* Can fetch instructions from page

Page Tables

- A page table (PT) contains the mapping of virtual addresses to physical addresses
- Where should PT be located?
  - Physical memory, so faster to access and can be shared by multiple processors
- The OS maintains the PTs
  - Each process has its own page table
    - "State" of a process is PC, all registers, and PT
  - OS stores address of the PT of the current process in the *Page Table Base Register*

Paging/Virtual Memory Multiple Processes



Caches vs. Virtual Memory

Caches	Virtual Memory
Block	Page
Cache Miss	Page Fault
Block Size: 32-64B	Page Size: 4KiB-8KiB
Placement: Direct Mapped, N-way Set Associative	Fully Associative (almost always)
Replacement: LRU or Random	LRU
Write Thru or Back	Write Back

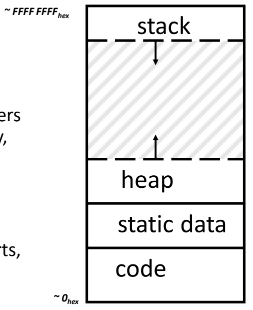
Technology Break

Notes on Page Table

- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory

Why would a process need to "grow"?

- A program's *address space* contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via **malloc()** ; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines).

Virtual Memory and Caches

- Physical memory is slow, so we cache data
  - Why not do the same with the page table?
- Translation Lookaside Buffer (TLB) is the equivalent of cache for the page table

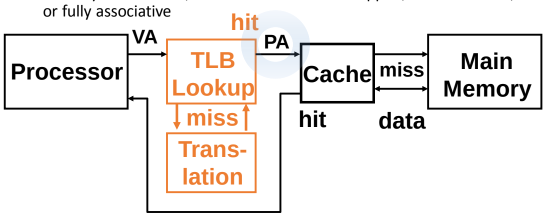
Virtual to Physical Address Translation

1. Check TLB
2. Check page tables (in main memory)
3. Page fault – OS handles

4. OS: Check disk
5. OS: Swap on disk/Segmentation fault

Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative



On TLB miss, get page table entry from main memory

Context Switching and VM

- Context Switching now requires that both the TLB and caches be flushed
  - In reality, TLB entries have a context tag

Peer Instruction

1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM

2) VM helps both with security and cost

12	
blue)	FF
green)	FT
purple)	TF
yellow)	TT

Peer Instruction Answer

1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM

FALSE

1. No. Both for VM and cache

2) VM helps both with security and cost

TRUE

2. Yes. Protection and a bit smaller memory

12	
blue)	FF
green)	FT
purple)	TF
yellow)	TT

Summary

- More aggressive performance options:
  - Longer pipelines
  - Superscalar (multiple issue)
  - Out-of-order execution
  - Speculation
- Virtual memory bridges memory and disk
  - Provides illusion of independent address spaces to processes and protects them from each other
  - VA to PA using Page Table
  - TLB