More VM: Paging

- Topics
  - TLBs and the kernel
  - Page faults
  - Loading programs
  - Page replacement
  - Working sets
- Learning Objectives:
  - Identify strategies for efficiently sharing physical memory.
  - Define a page fault and explain how they occur and are handled.
  - Explain the MIN, LRU, Clock, and Working set paging algorithms.
  - Tackle Assignment 3.

Address Translation and the Kernel

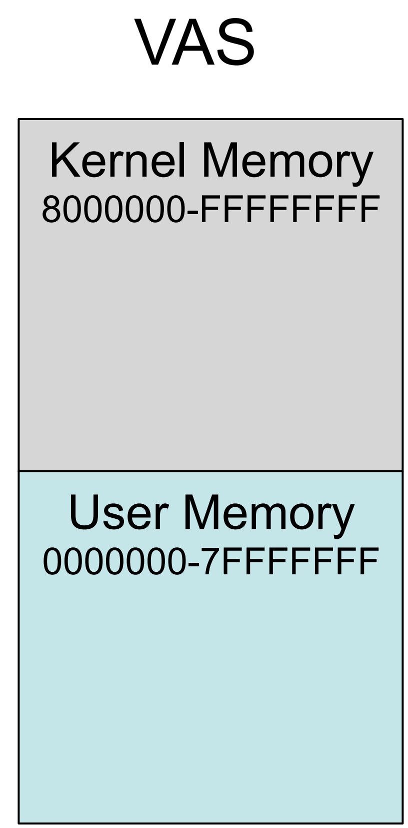
- Exercises:
  - If everything gets translated via the TLB, how does the operating system manage physical memory?
  - How do we make sure that a user process does not mess with the kernel's memory?

Address Translation and the Kernel

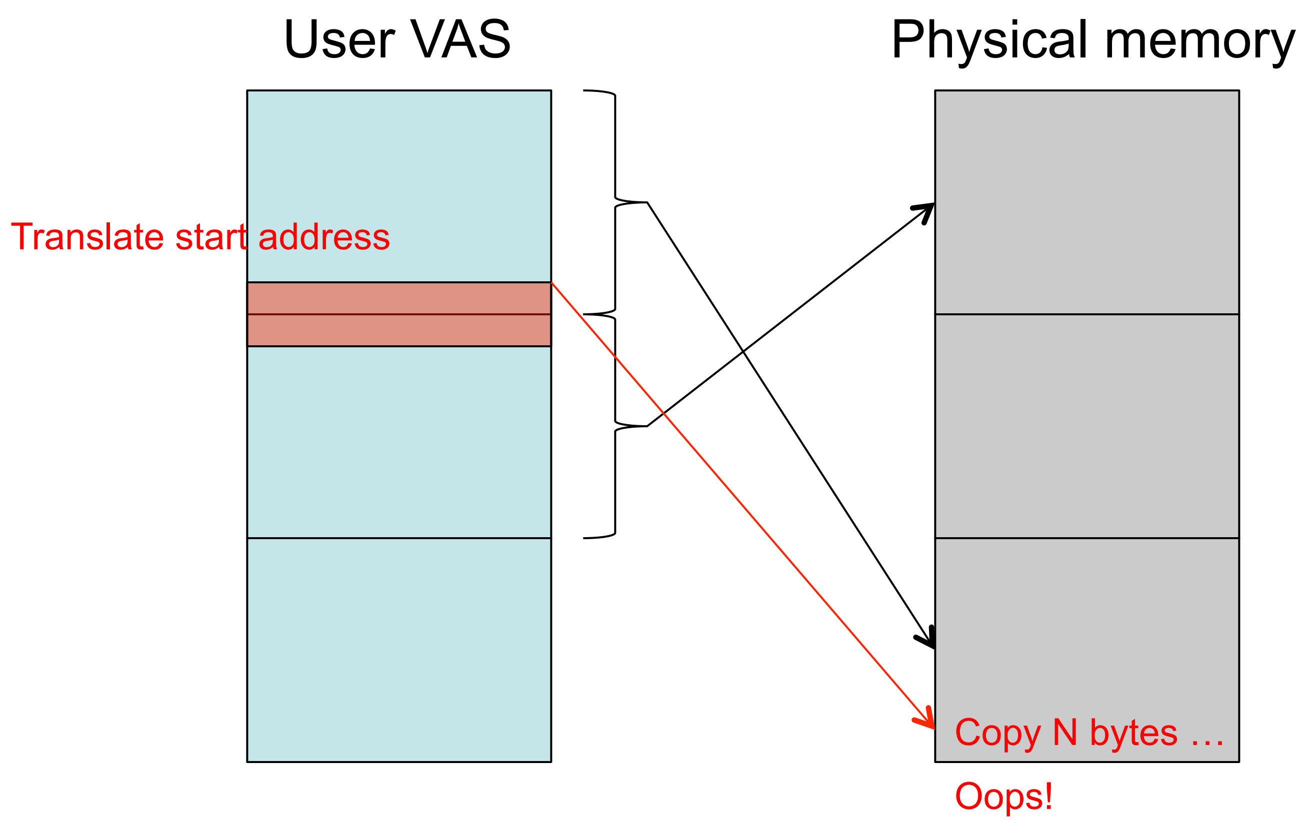
- Exercises
  - If everything gets translated via the TLB, how does the operating system manage physical memory?
  - How do we make sure that a user process does not mess with the kernel's memory?
  1. Run the OS unmapped (with VM turned off)
  2. Divide the address space into parts, some of which get mapped for the kernel and some of which get mapped for user processes.
  - In either case, the kernel needs to be able to read a user process's page tables and translate (to access data in a user process's address space).
  - One other problem …

Alternatives to an unmapped kernel

- Hardware reserves a portion of the virtual address space for the kernel.
- Kernel effectively mapped into ***every*** process.
- Kernel uses process's page tables to access process mappings.
- User process not able to access kernel TLB entries and mappings, but kernel can use process ones.



When User Memory Spans Pages



Copyin/Copyout

- Recall that we mentioned that the kernel had two functions, copyin and copyout that it used to move things from a user address space into the kernel's address space?
- You do not want to have to translate every address, but you need to make sure that you don't fall off a page boundary and end up somewhere you shouldn't.
- You can make this work, but it's tedious.

Paging Overview

- The MMU separates the programmer's view of memory from the system's view, providing:
  - Flexibility in placing processes in memory.
  - Multiple mechanisms by which processes may share memory.
- While we have discussed the possibility of virtual pages not being resident in memory, we have not discussed why this might happen, and precisely how to deal with it.
- EXERCISE: Why might we want to run processes when some of their pages aren't in memory?

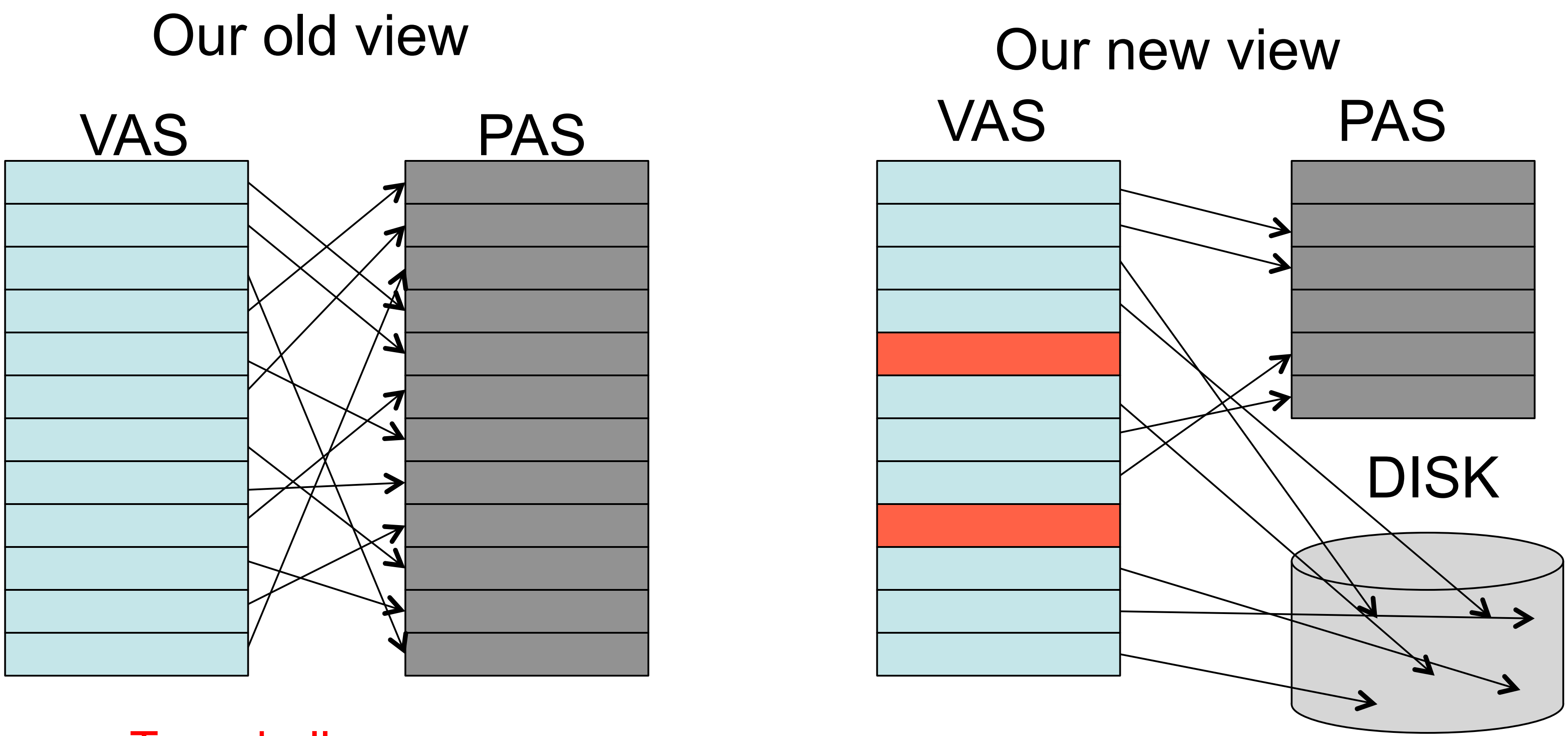
Paging Overview

- The MMU separates the programmer's view of memory from the system's view, providing:
  - Flexibility in placing processes in memory.
  - Multiple mechanisms by which processes may share memory.
- While we have discussed the possibility of virtual pages not being resident in memory, we have not discussed why this might happen, and precisely how to deal with it.
- EXERCISE: Why might we want to run processes when some of their pages aren't in memory?
  - Processes can be very large.
  - Programs exhibit locality, so some pages may be unnecessary.
  - You only use a subset of a process's pages at any one time.
  - You can start processes more quickly if you don't need to preload everything.

What is Paging

- The mechanism by which we allow processes to run with only some of their pages resident in memory.
- In a demand paging system, virtual pages can be in one of three states:
  - Unmapped: there is nothing present at a virtual address.
  - Memory resident
  - Disk resident
- Pages in main memory are frequently called page frames.
- Pages on disk are frequently called backing frames.
- Our goal is to provide the illusion that main memory is as large as disk and as fast as memory.
  - When things go wrong, you get the feeling that memory is as small as memory and as slow as disk!
  - Fortunately, locality saves us (in most cases).

Our New View of Memory



- Two challenges:
  - How to run processes with some pages are missing
  - How to schedule which page are in main memory?

Page Faults

- Extend page table entry (PTE) to include a present bit.
- If virtual to physical translation yields a page table entry in which present is not set, the reference results in a trap, called a page fault.
- Any page not in main memory has a present bit of 0.
- When a page fault occurs:
  - Operating system brings page into memory.
  - Page table is updated; present bit is set.
  - TLB is updated.
  - The process that faulted continues execution.
- Continuing a process is extremely tricky.
  - Page fault may have occurred in the middle of an instruction.
  - Need to make the fault invisible to the user process.

Page Fault Handling (1)

- Typically, the PC is incremented at the **beginning** of the instruction cycle. Therefore, if you do not do anything special, you will continue running the process at the instruction **after** the faulting one and it will appear as if the faulting instruction got skipped.
  - Users probably will not like this behavior.
  - "Hi, we're giving you virtual memory. Oh by the way, sometimes we skip instructions."
- You have three options:
  - Restart the instruction: undo whatever the instruction may have already done and then reissue the instruction.
    - Used by PDP-11, **MIPS R3000**, and most modern architectures.
  - Complete the instruction: continue where you left off.
    - Used in the Intel x86.
  - Test for faults before issuing the instruction.
    - Used in the IBM 370.

Page Fault Handling (2)

- Without hardware support, you should either forget about paging or use complex (and disgusting) solutions.
  - MC68000, Intel 8086 and 80286: could not restart instructions.
  - Apollo systems (used Motorola CPUs) had two CPUs.
    - One executed user code.
    - If it took a fault, the user CPU stalled while the OS CPU fetched the page.
    - Once it got the page, the user CPU was un-stalled.
- Even with hardware support, the page fault handler must be able to recover the cause of the fault and enough of the machine state to continue the program.
- EXERCISE: Food for thought:
  - Where to you find missing pages that you need?
    -
    -
    -

Page Fault Handling (2)

- Without hardware support, you should either forget about paging or use complex (and disgusting) solutions.
  - MC68000, Intel 8086 and 80286: could not restart instructions.
  - Apollo systems (used Motorola CPUs) had two CPUs.
    - One executed user code.
    - If it took a fault, the user CPU stalled while the OS CPU fetched the page.
    - Once it got the page, the user CPU was un-stalled.
- Even with hardware support, the page fault handler must be able to recover the cause of the fault and enough of the machine state to continue the program.
- Food for thought:
  - Where to you find missing pages that you need?
    - In the executable file?
    - Create them?
    - In swap space?

Scheduling Decisions

- Page Selection: When do you bring pages into memory?
- Page Replacement: When you need to evict a page from memory, how do you select the page to evict?

Page Selection

- Preloading
  - Before execution begins, load in a few pages to get started: e.g., full program text, statically allocated data, a stack page.
- Prepaging
  - Bring a page into memory just before it is referenced
  - Typically best guess is sequential.
  - Unfortunately, programs aren't necessarily sequential on a page-wide basis.
  - May read a lot of pages you didn't really need.
  - When might it work OK?
    -
    -
- Request paging
  - Make users request pages that they need to run.
  - This is the "Oh please, Operating System, I really, really, really, need the following pages.
- Demand paging
  - Start execution with no valid mappings.
  - On each page fault, load in a page.

Page Selection

- Preloading
  - Before execution begins, load in a few pages to get started: full program text, statically allocated data, a stack page.
- Prepaging
  - Bring a page into memory just before it is referenced
  - Typically best guess is sequential.
  - Unfortunately, programs aren't necessarily sequential on a page-wide basis.
  - May read a lot of pages you didn't really need.
  - When might it work OK?
    - Preloading: see above.
    - Boot time.
- Request paging
  - Make users request pages that they need to run.
  - This is the "Oh please, Operating System, I really, really, really, need the following pages.
- Demand paging
  - Start execution with no valid mappings.
  - On each page fault, load in a page.

Page Replacement

- Random
  - Pick any page to evict.
  - Works surprisingly well!
- FIFO
  - Throw out page that has been in memory the longest.
  - The basic idea is that you give all pages equal residency.
- MIN
  - Predict the future.
  - Evict the page that will not be referenced for the longest time.
  - Tough to implement.
  - Good for comparison.
  - Defined by Laszlo Belady (known as Belady's algorithm).
- LRU
  - As usual, use past to predict future.
  - Evict page that has been unreferenced the longest.
  - With locality, this is a good approximation to MIN.
- What makes implementing some of these difficult? What other metrics/ statistics might you want to keep about your pages?

Page Replacement

- Random
  - Pick any page to evict.
  - Works surprisingly well!
- FIFO
  - Throw out page that has been in memory the longest.
  - The basic idea is that you give all pages equal residency.
- MIN
  - Predict the future.
  - Evict the page that will not be referenced for the longest time.
  - Tough to implement.
  - Good for comparison.
  - Defined by Laszlo Belady (known as Belady's algorithm).
- LRU
  - As usual, use past to predict future.
  - Evict page that has been unreferenced the longest.
  - With locality, this is a good approximation to MIN.
- What makes implementing some of these difficult? What other metrics/statistics might you want to keep about your pages?
  - LRU is recency; requires a single queue
  - Frequency is easier (sorting is hard).

Playing pager (3 memory frames)

Reference stream	**A**	**B**	**C**	**A**	**B**	**D**	**A**	**D**	**B**	**C**	**B**
FIFO	A										
		B									
			C								
MIN	A										
		B									
			C								
LRU	A										
		B									
			C								

Playing pager (3 memory frames)

Reference stream	A	B	C	A	B	D	A	D	B	C	B
FIFO	A					D				C	
		B					A				
			C						B		
MIN	A									C	
		B									
			C			D					
LRU	A									C	
		B									
			C			D					

- Just like STCF, MIN is optimal, but not implementable.
- Just like priority queues or fair-share scheduling, use the past to predict the future. For page replacement, LRU (least recently- used) works remarkably well.

Implementing LRU

- Need hardware to keep track of recently used pages.
- Perfect LRU?
  - Register for every physical page.
  - Store clock on every access.
  - To replace, scan through all the registers.
  - Assessment?
    -
    -
- Approximate LRU
  - Find any *old* page.
  - May not be oldest, but if it's old, it's probably good enough.
  - After all, LRU is an approximation of MIN; what's another level of approximation?
- Clock
  - Maintain a *use* bit for each frame.
  - Set bit on every reference.
  - Operating system sweeps through memory clearing use bits.

Implementing LRU

- Need hardware to keep track of recently used pages.
- Perfect LRU?
  - Register for every physical page.
  - Store clock on every access.
  - To replace, scan through all the registers.
  - Assessment?
    - Expensive!
    - Not very practical.
- Approximate LRU
  - Find any old page.
  - May not be oldest, but if it’s old, it’s probably good enough.
  - After all, LRU is an approximation of MIN; what’s another level of approximation?
- Clock
  - Maintain a use bit for each frame.
  - Set bit on every reference.
  - Operating system sweeps through memory clearing use bits.

Implementing Clock

- When time to replace, replace a page frame with a 0 use bit.
- On page fault — circle around clock.
  - If bit is set, clear it.
  - If bit is not set, replace it.
  - Can this loop infinitely?
  - Can also incorporate *dirty* bit since dirty pages are more expensive to evict than clean ones.
- In clock, what does it mean if the clock hand is sweeping very slowly?
- What if the hand is sweeping very quickly?

Implementing Clock

- When time to replace, replace a page frame with a 0 use bit.
- On page fault — circle around clock.
  - If bit is set, clear it.
  - If bit is not set, replace it.
  - Can this loop infinitely? NO
  - Can also incorporate *dirty* bit since dirty pages are more expensive to evict than clean ones.
- In clock, what does it mean if the clock hand is sweeping very slowly?
  - Plenty of memory.
  - Not many page faults.
  - This is good (desirable).
- What if the hand is sweeping very quickly?
  - Not enough memory.
  - Thrashing.