CS24: Introduction to Computing Systems

Spring 2015

Lecture 22

Last Time: Process Scheduling

- Began discussing how to schedule processes
  - When a running process is blocked, suspended or terminated, scheduler chooses another process to run
  - Must be fair: all processes must receive CPU time
  - Must perform its scheduling quickly

- Context-switches are slow
  - Frequently take approx. 5-10μs
  - *Multiple tens of thousands of instructions!*

Context-Switch Operations

- *Lots* of work to do in a context switch:
  - Process is interrupted via exceptional flow control
  - The kernel is invoked…
    - Switch from process code to kernel code
    - Change protection-level (and stack/data areas being used)
  - Kernel must save the process' context into its PCB
  - Kernel picks another process to start running
    - (This is where scheduler does its work. Hopefully quickly.)
  - Kernel must restore the process' context from its PCB
  - Return back from kernel to the next process
    - Again, change protection level (and stack/data areas)
  - Resume executing the next process

Last Time: Process Behaviors

- Processes can exhibit a wide variety of behaviors
  - Interactive processes
  - Compute-intensive processes
  - Real-time processes
  - Makes it more challenging to schedule effectively
  - (Plus, programs can change behavior over time…)
- Covered two simple scheduling algorithms
  - Round-robin scheduling – good for compute-intensive processes, bad for interactive ones
  - Shortest jobs first scheduling – good for interactive processes, bad/unfair for compute-intensive ones

SCHEDULING CHALLENGES!

- Generally, if tasks are uniform, it's much easier to solve the scheduling problem
  - If all tasks are interactive, just use Shortest Job First approach
  - If all tasks are compute-intensive, use Round Robin approach with a large time-slice

- The challenges arise when the scheduler must deal with different kinds of processes
  - Must properly balance needs of interactive processes with needs of long-running processes

Earliest Deadline First Scheduling

- Real-time processes are usually deadline-driven
  - *Video player needs to draw 30 frames per second, no matter what!*
- Also tend to be cyclic in their execution
  - Typically don't need the CPU for very long
- Scheduler can estimate several values:
  - When is the deadline for each process? (T_{dl})
  - What is the average run-time for each process? (T_{run})
- Scheduler chooses the process with the smallest $T_{dl} - (T_{now} + T_{run})$
  - (the process most in danger of missing its deadline ☺)
- Variations on this theme
  - Minimize average lateness, minimize max lateness

A General-Purpose Scheduler?

- Problem:
  - Interactive processes should run at a high priority, but should have a relatively small time-slice
    - "Well-behaved" interactive processes can be expected to become IO-bound before using up entire time
  - Compute-intensive tasks should run at a low priority, but should have a longer time-slice
    - Minimize context-switch overhead for long-running process
  - Real-time tasks should run at a very high priority to satisfy timing requirements, but expected to be short

- Solution:
  - As always, build a general-purpose scheduler by combining these strategies

Multilevel Feedback Queues

- Most operating systems use a *multilevel feedback queue* strategy for process scheduling
  - Windows NT/XP/Vista, Solaris, BSD variants
  - (Not MacOS X, or Linux 2.5+)
- Idea:
  - Scheduler uses multiple queues to segregate processes of varying priorities and behaviors
  - Each queue has its own maximum time-slice size, and even its own scheduling algorithm if needed
  - Move processes between the queues based on their observed behaviors
    - As the process executes, the scheduler moves it into the best queue for how the process is behaving
    - If process behavior changes, scheduler adapts to this easily

Multilevel Feedback Queues (2)

- Example queues:
  - Q1: time-slice of 5ms
  - Q2: time-slice of 15ms
  - Q3: time-slice of 30ms
  - Within each queue, it's "first come, first served"
- General rules:
  - A new process is put into the highest priority queue
  - If a process uses its entire time-slice, it is preempted by the kernel and demoted to the next lower queue
  - If a process yields to the kernel before its time-slice is up, it goes to the end of its current queue
  - A process can also be promoted if it regularly blocks or yields within the next higher queue's time-slice

MULTILEVEL FEEDBACK QUEUES (3)

- These rules very quickly categorize processes based on their behavior!
  - Different queues contain processes with different behaviors
- Additionally:
  - Want scheduler to give preference to shorter jobs, and to IO-bound processes
  - Want to give longer-running jobs larger time-slices to reduce context-switch overhead
- Scheduler executes Ready processes in Q1 first
  - Then, if Q2 contains Ready processes, these will be executed
  - Finally, if Q3 contains Ready processes, these are executed in round-robin order

Multilevel Feedback Queues (4)

- Example queues:
  - Q1: time-slice of 5ms
  - Q2: time-slice of 15ms
  - Q3: time-slice of 30ms
- Q1 ends up with processes that work in <5ms bursts
  - High priority since they will be done fast
  - Are very likely to be interactive processes
- Q2 ends up with processes that work in 5-15ms bursts
  - Lower priority, but longer time-slice
- Q3 contains processes that run in >15ms
  - If a process uses 30ms, it is preempted and sent to back of Q3 (i.e. round robin implementation)
  - If a process starts being regularly IO-bound, it will be promoted back up the queues, based on its run-times

Multiple Scheduling Algorithms

- Most operating systems that use multilevel feedback queues also support real-time processes
  - Real-time queue levels are higher priority than standard queue levels
  - The scheduling algorithms are different
  - Real-time processes aren't demoted to below the real-time range of queue levels
- Examples:
  - Windows NT-based systems: levels 0-15 are "normal" processes, 16-31 are soft real-time processes
  - Linux pre-2.5: levels 0-99 are real-time processes, 100-140 are "nice" task levels
  - (Linux versions 2.5+ use various other schedulers)

Summary: Scheduling

- UNIX process API is relatively straightforward…
- Implementation is significantly more complex
  - Many details to keep track of for each process
  - Resources that a process is using, pending requests that a process is blocked on, other state information
- Processes can have *very* different performance characteristics!
  - Simple scheduling techniques can handle specific kinds of process behavior
  - Creating a generic process scheduler is more involved
  - Adapt scheduling choices based on past process behaviors
  - Also need to ensure that scheduling is fair! ☺

Process Abstraction: The Scorecard

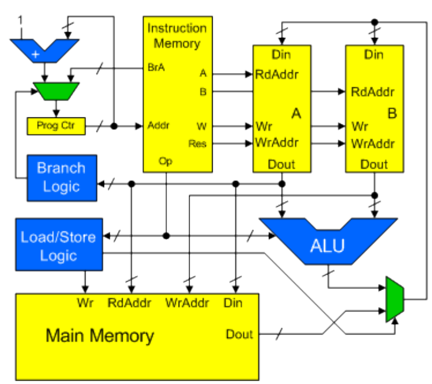
- So far, have covered many aspects of how to implement the process abstraction
- Q: How to enforce differences between kernel and application processes?
  - A: Processor operating modes, hardware protection levels
- Q: What state information to manage for processes? How do we organize and manage it?
  - A: Registers, flags
  - *(What about program memory?)*
  - Use a Process Control Block structure to manage this state
- Q: How to interrupt a running processes?
  - A: Interrupts and exceptions to interrupt logical program flow and let kernel perform various tasks
- Q: How to choose which process should run next?
  - A: Use generalized, fair scheduling algorithms that can manage processes with varying behaviors

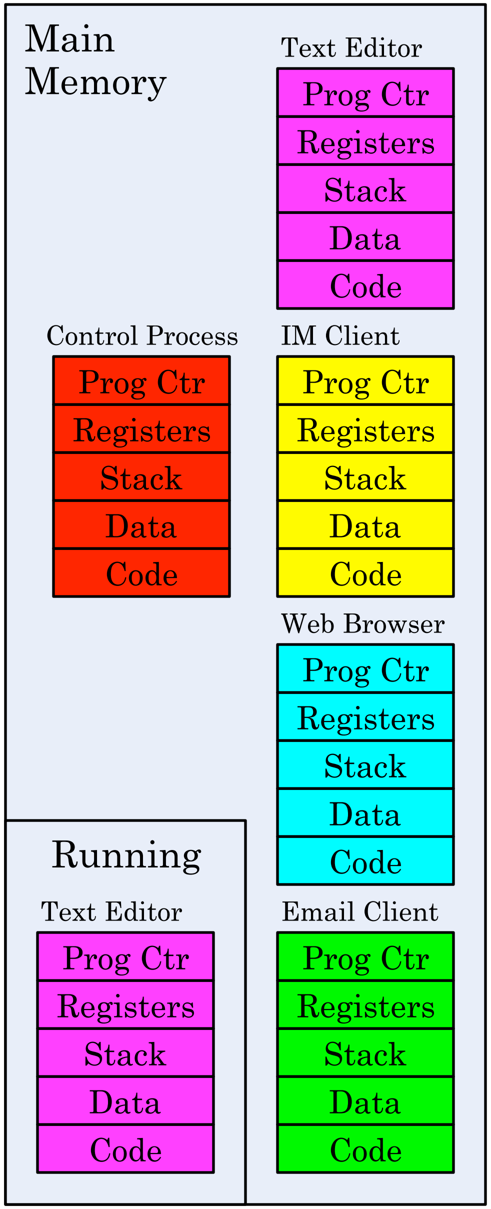
Process Abstraction: Missing Pieces

- Glossed over some pretty important questions!
  - All relating to *how we manage process memory*
- How do we isolate the address spaces of different processes from each other?
- How to provide faster context-switches?
- Some other important questions too:
  - How do we provide access to the kernel and shared libraries in an efficient and uniform way?
  - How do we let processes share memory and coordinate with each other?
  - Very important if our OS is going to support powerful applications and services to be implemented on it

Process Abstraction

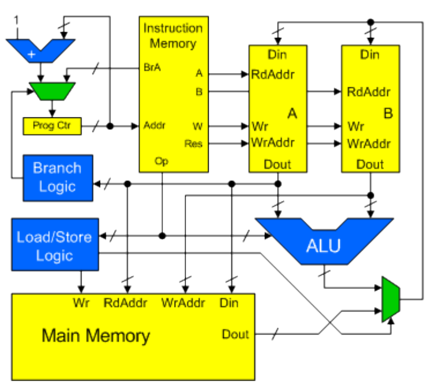
- Previous approach:
  - Running processes use a region of memory at bottom of address space
  - Suspended processes occupy other areas
- Context-switch:
  - Kernel interrupts the running program
  - Copies context of running process back to the process' memory area
  - Next, kernel copies context of another process into the running area
  - Finally, starts running the new process for another time-slice

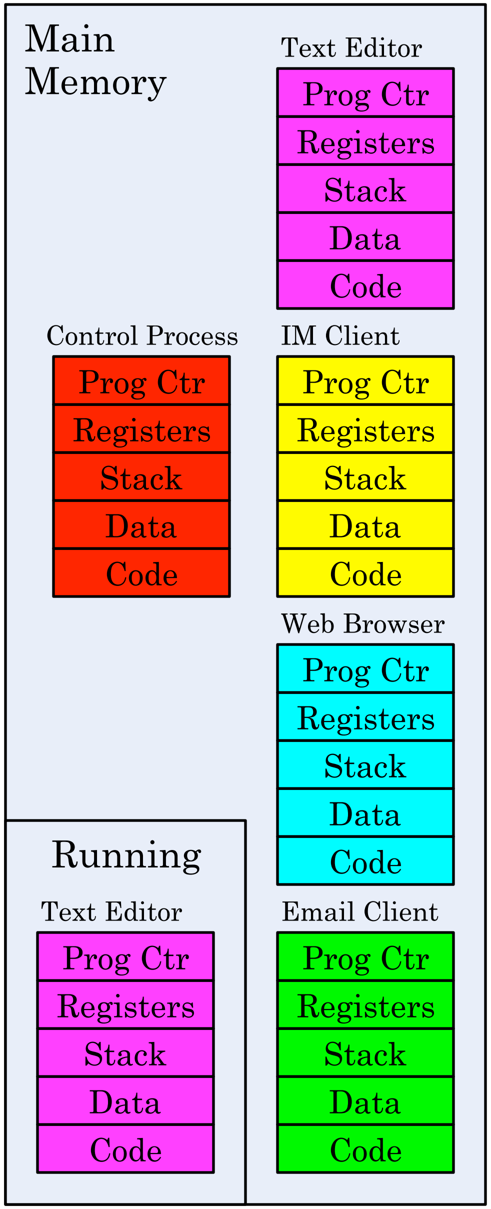




Process Abstraction (2)

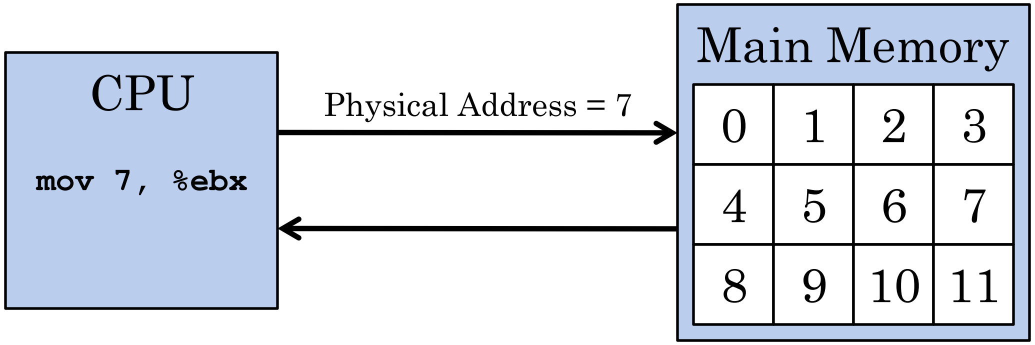
- Several big drawbacks with this approach!
- Copying process data will be very slow
  - Accessing DRAM main memory from the CPU takes 50-100ns!
  - Clearly unacceptable for systems running many processes, or ones with large memory footprints
- All processes must fit in main memory
  - System can't handle a large number of processes at same time
  - Severely limits ability of processes to work with large amounts of data





Physical Addressing

- Main memory is an array of *M* contiguous bytes
  - Each location has its own physical address
  - Size of main memory is fixed
- When a program accesses memory:
  - Instruction refers to a physical address (direct or indirect)
  - Processor sends this address directly to main memory to retrieve the associated value



Virtual Addressing

- Instead of directly addressing physical memory, introduce a level of indirection
  - Instructions use *virtual* addresses instead of physical addresses
  - Translate virtual addresses into physical addresses as instructions are executed
- Given an address a:
  - Instead of directly accessing M[a], introduce a mapping table T, which maps virtual addresses to physical addresses
  - Use T to translate addresses: M[T[a]]
    - Each virtual address a is mapped to a physical address
    - The physical address is used to access physical memory

Virtual Addressing (2)

- Clearly prohibitive to map every single virtual address to a distinct physical address…
  - Mapping table would require much more space than the actual memory for the computer
  - Also, programs typically access memory in blocks, exploiting locality
- Divide memory up into <u>pages</u> of size P, $P = 2^p$
  - Choose a page size much larger than 1 byte or 1 word
  - (page-size considerations are discussed shortly…)
- Map each virtual page to a physical page
  - Mapping is specified using a <u>page table</u>
  - Each <u>page table entry</u> maps one virtual page to one physical page

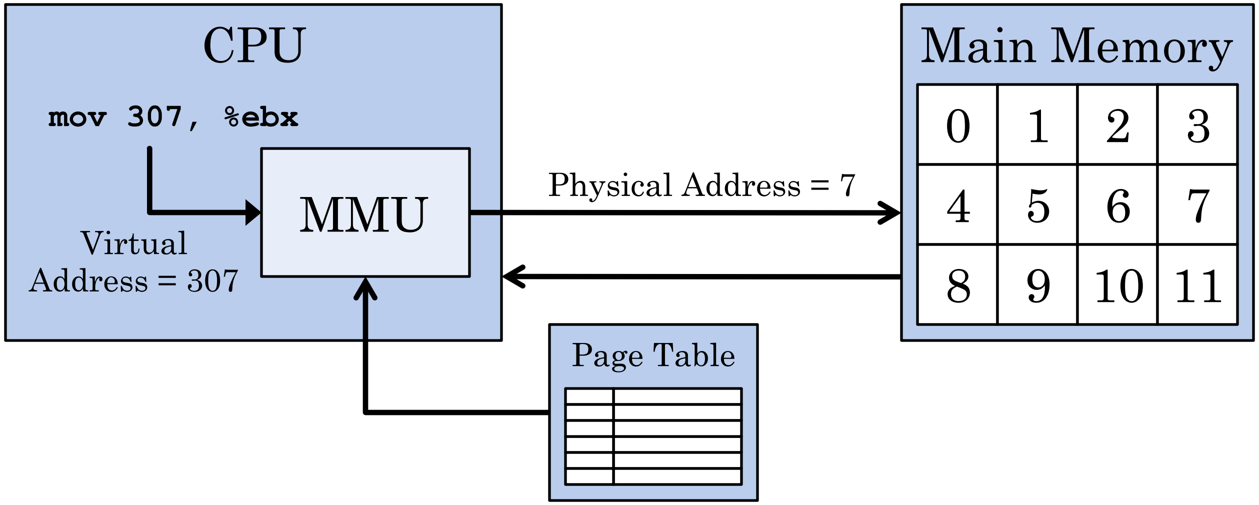
Virtual Memory

- Main memory provides a *physical address space*
  - Size of M bytes (frequently, $M = 2^m$, but not required)
  - Computer provides an m-bit address space
- Define a *virtual address space* of size N bytes
  - $N = 2^n$, so this is an n-bit virtual address space
  - Not required that $M = N$, but for now we will assume this is the case

- Virtual memory system must map virtual pages (pages in virtual address space) to physical page frames (pages in physical address space)

VIRTUAL ADDRESSING

- Performing this address translation in software would be horribly slow…
- CPU provides *hardware* support for virtual memory and address translation
  - CPU has a Memory Management Unit (MMU) that performs this address translation on the fly
  - The MMU uses a page table to perform translation



Virtual and Physical Pages

- Virtual addresses specify a virtual page number (VPN) and a virtual page offset (VPO)
  - Offset is lowest p bits (pages contain 2^p bytes)
  - Virtual page number is upper $n - p$ bits in virtual address

n-1 … p	p-1 … 0
Virtual Page Number	Virtual Page Offset

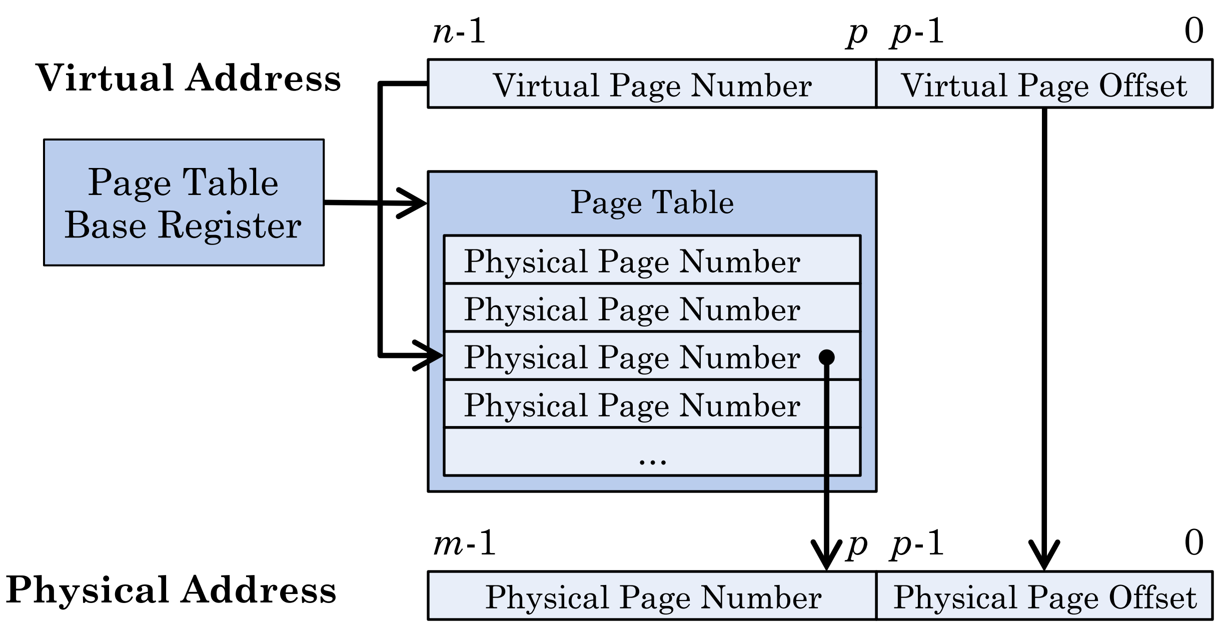
- Physical addresses specify a physical page number (PPN) and a physical page offset (PPO)
  - Offset is lowest p bits, as before
  - Similarly, physical page number is upper $m - p$ bits

m-1 … p	p-1 … 0
Physical Page Number	Physical Page Offset

- Page table maps virtual pages to physical pages

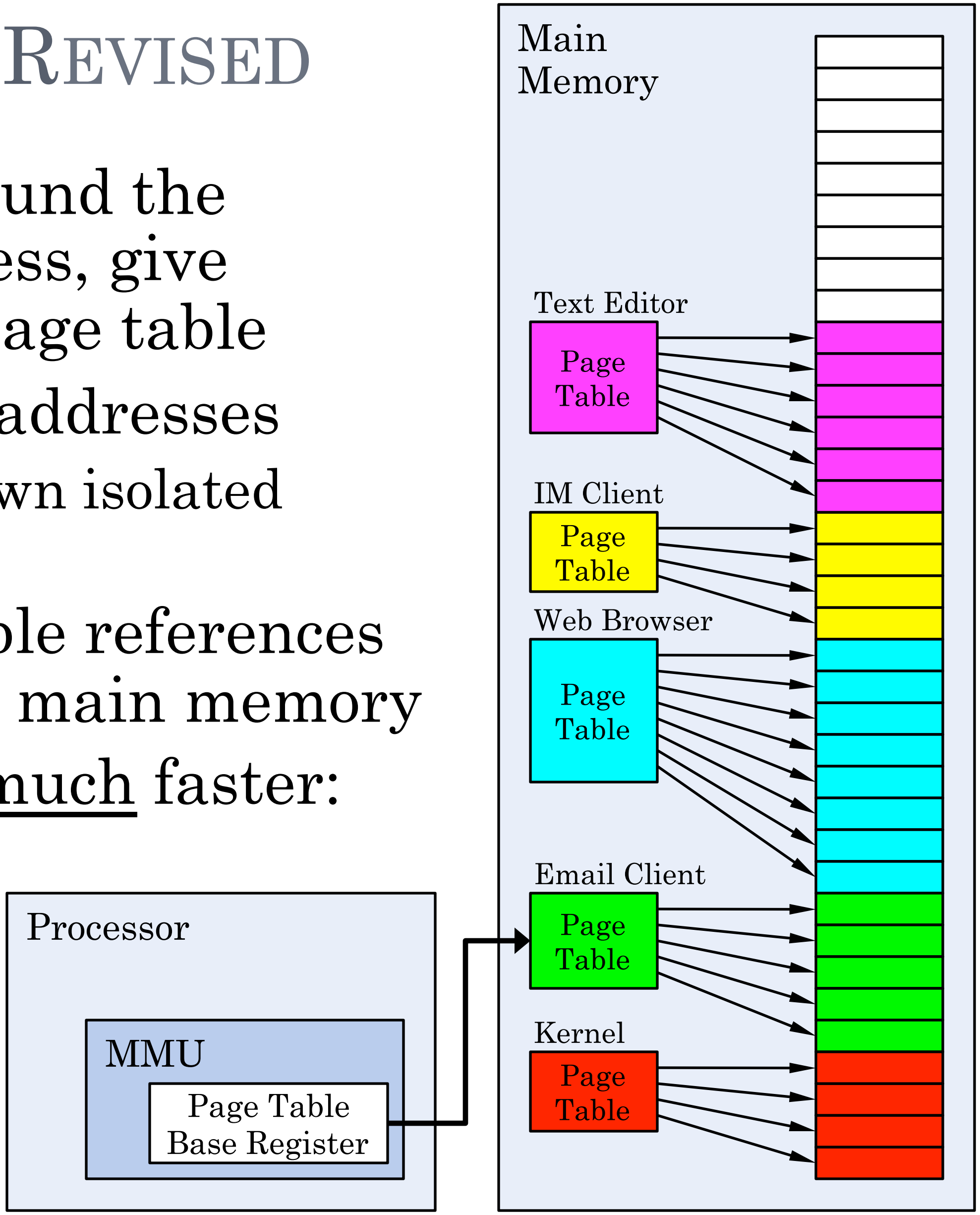
Address Translation

- Page table is indexed with virtual page number
  - Page table entry contains the physical page number
  - Combine physical page number with virtual page offset to get physical address
- Start of page table specified in a control register
  - MMU uses this address to look up page table entries



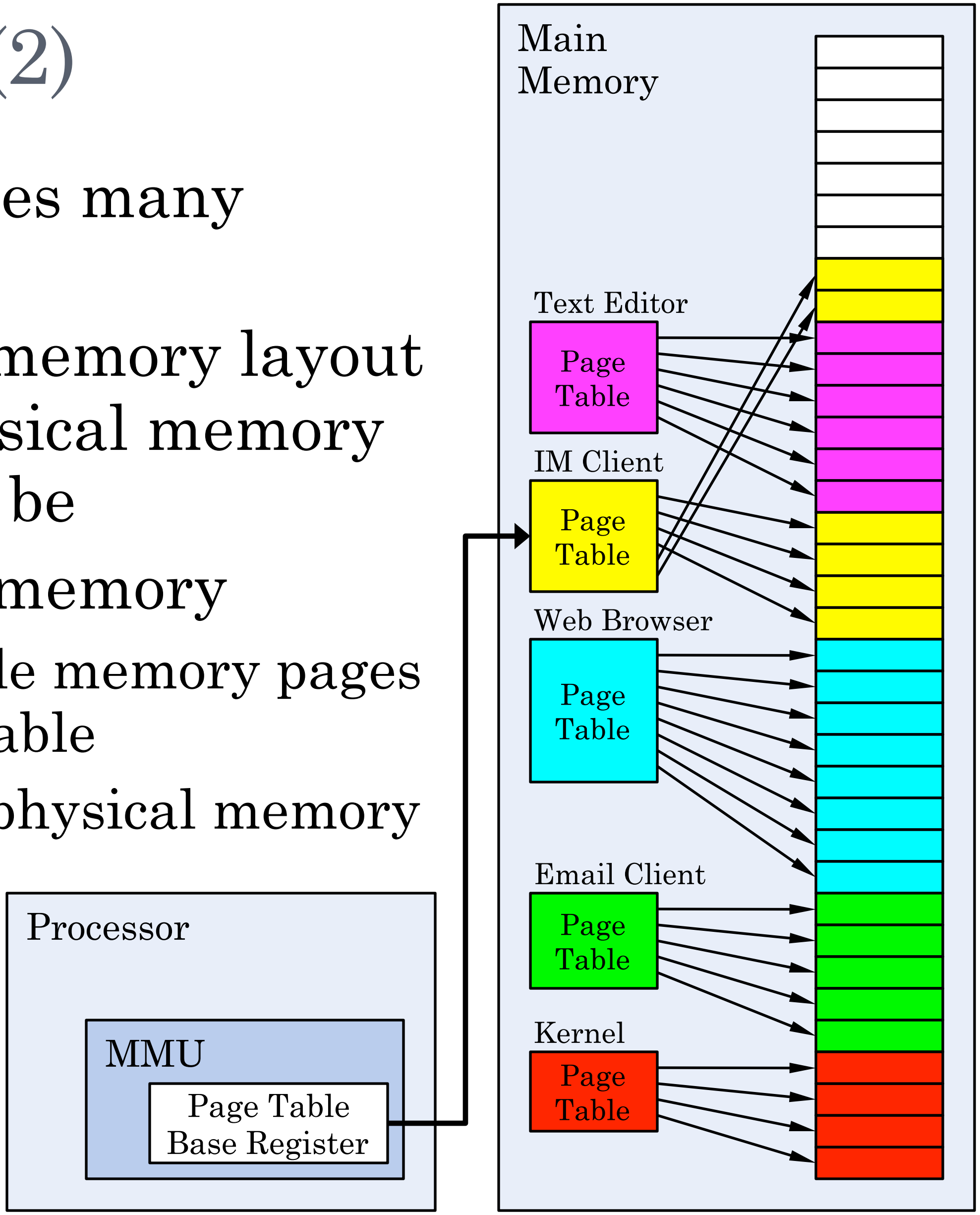
Process Memory, Revised

- Instead of copying around the memory for each process, give each process its own page table
- Programs use virtual addresses
  - Each process has its own isolated address space
- Each process' page table references its own set of pages in main memory
- Context-switching is <u>much</u> faster:
  - Simply change Page Table Base Register to reference the new process' page table!



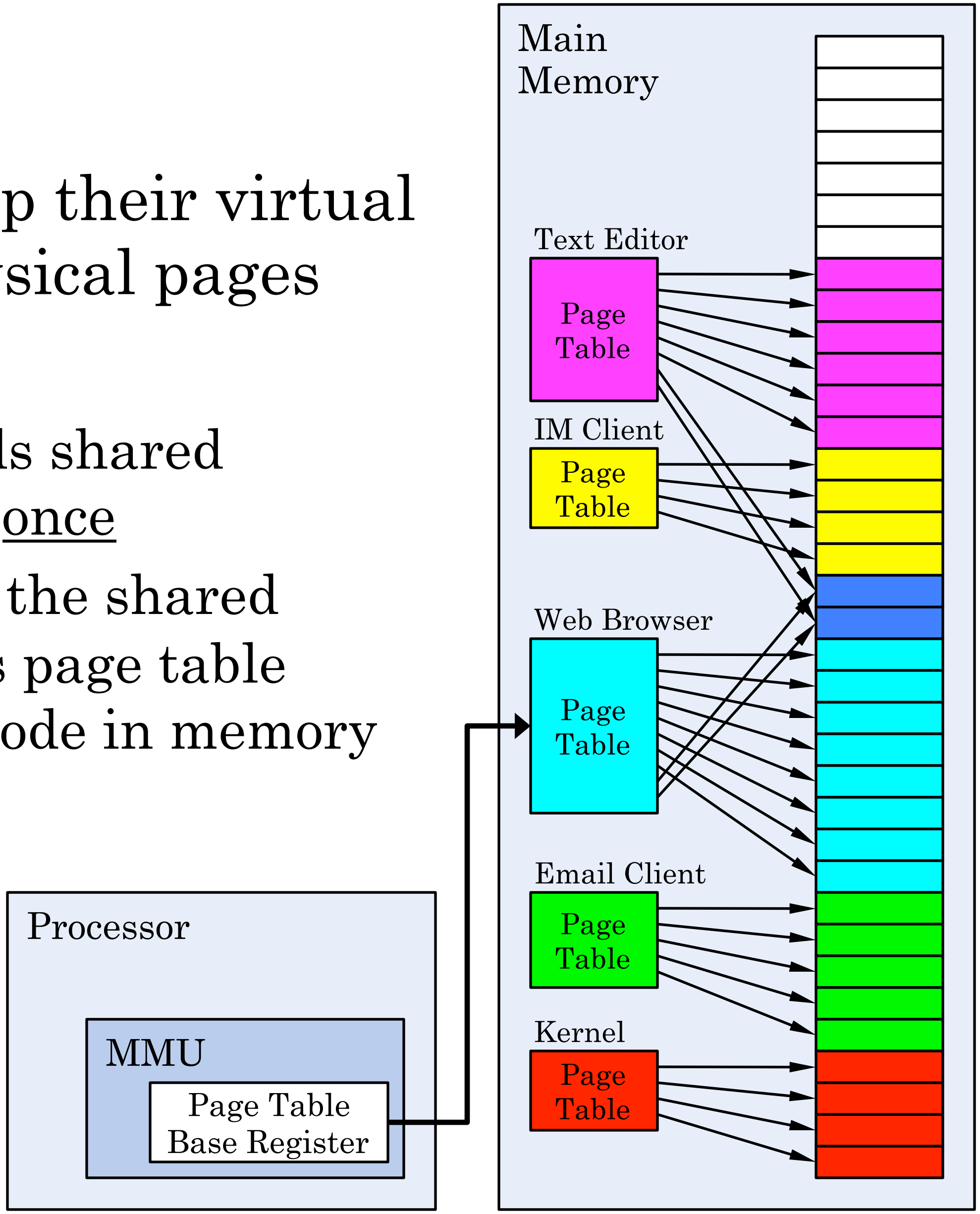
Process Memory (2)

- Virtual memory enables many other useful features
- Each process' virtual memory layout is contiguous, but physical memory layout doesn't have to be
- IM client needs more memory
  - Simply assign available memory pages to that process' page table
  - Process doesn't know physical memory isn't contiguous
  - CPU takes care of virtual-to-physical address translation



SHARED PAGES

- Two processes can map their virtual pages to the same physical pages
- Shared libraries:
  - Operating system loads shared libraries into memory <u>once</u>
  - When a process needs the shared library, just update its page table to reference library's code in memory
- Shared memory:
  - Multiple processes collaborate by working in the same memory area



Virtual Memory System So Far…

- Already achieved a lot with this simple idea!
- <u>Much</u> easier to manage memory in processes
  - Context-switches are *much* faster
  - Processes have their own isolated virtual address spaces
  - CPU handles mapping of virtual addresses to physical addresses automatically
  - A process' physical memory layout doesn't have to be contiguous
  - Can map multiple virtual pages to the same physical page
- Problem:
  - Still have to divide up the limited main memory amongst <u>all</u> processes, whether running or stopped
- We know that not all processes are always active…
  - e.g. a process can be stopped and resumed
  - Also, a process might not always be using all of its memory

Next Time

- Incorporate virtual memory into the memory hierarchy:
  - Virtual memory becomes a *cache* for program data stored on disk