CS61C: Great Ideas in Computer Architecture (Machine Structure)

Instructors: Rosalie Fang, Charles Hong, Jero Wang

Lecture feedback: https://tinyurl.com/fyr-feedback

Come sit at the front 🥺🥺🥺

So far in 61C

- Multi-threading, multi-processing

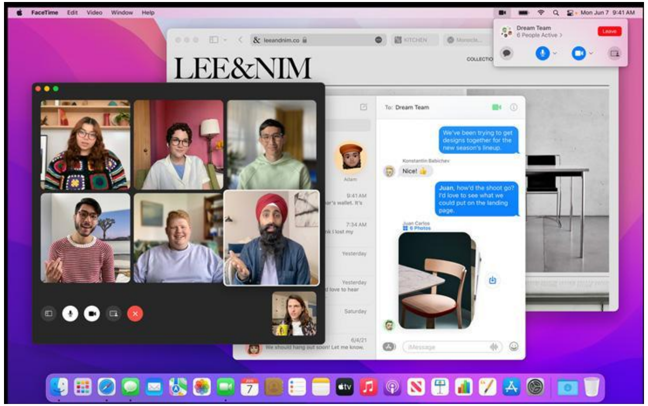
Screen

Keyboard

Storage

1. Multiple ***I/O devices*** (input-output)

when switched on…



To: Dream Team

We've been trying to get designs together for the new season's lineup.

Konstantin Babichev

Nice! 👍

Juan, how'd the shoot go? I'd love to see what we could put on the landing page.

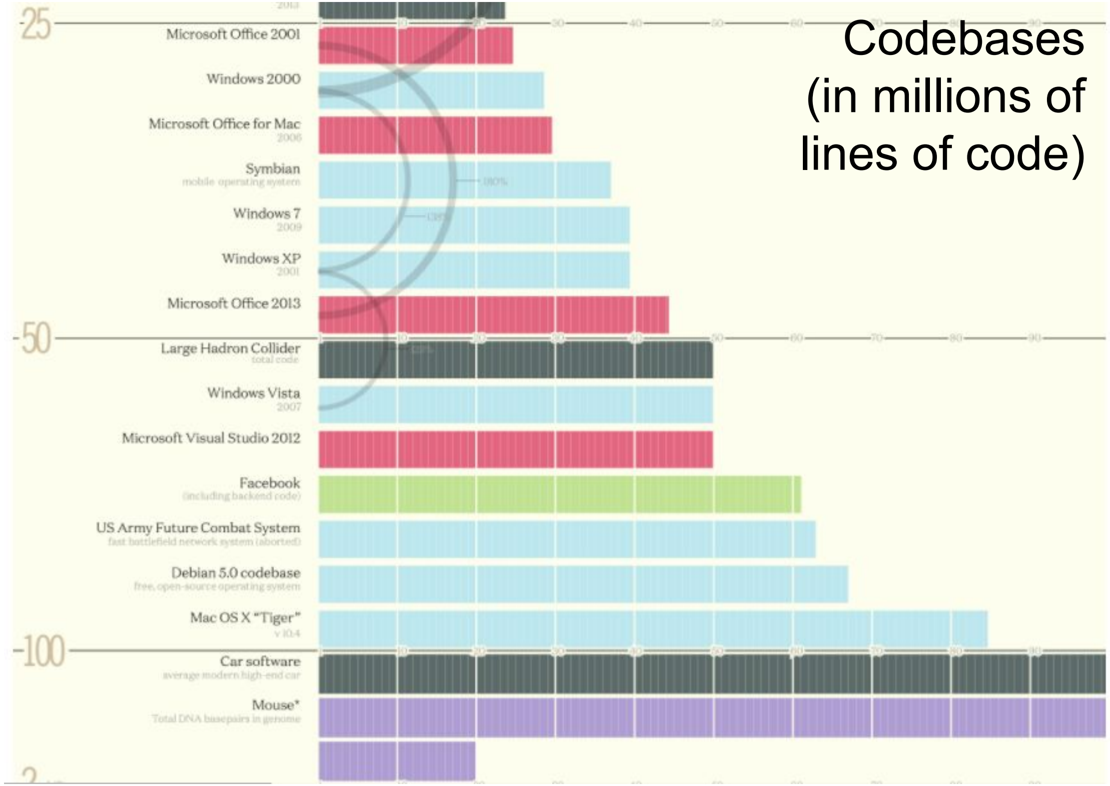
Juan Carlos

6 Photos

2. Multiple programs running “simultaneously” “on” a software program called the ***Operating System (OS)***

The OS is “Just Software”

- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:



Codebases
(in millions of
lines of code)

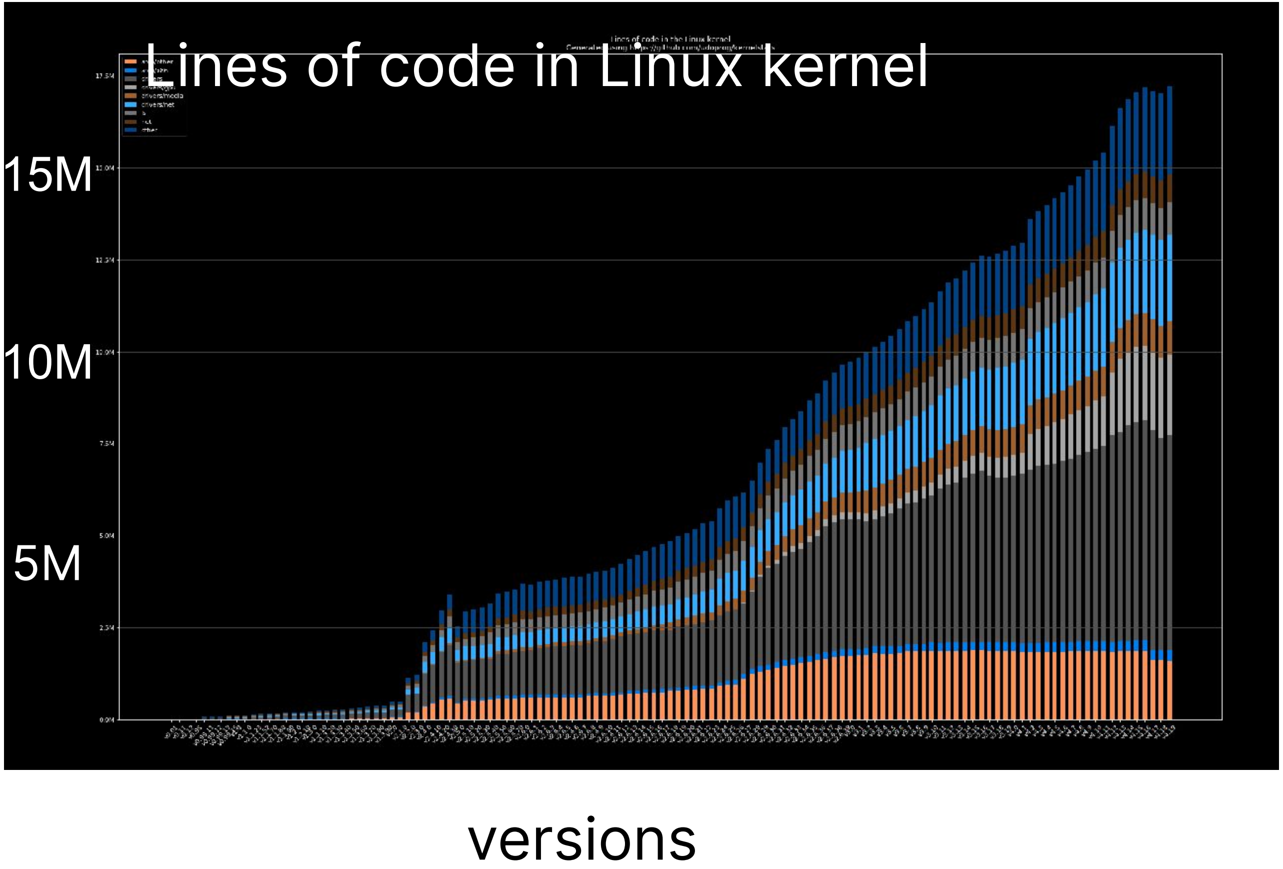
http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Linux Kernel Over Time

In CS61C,
OS ≈ kernel.

- Other components, e.g., User Interface, not covered.

Take CS162 for more!



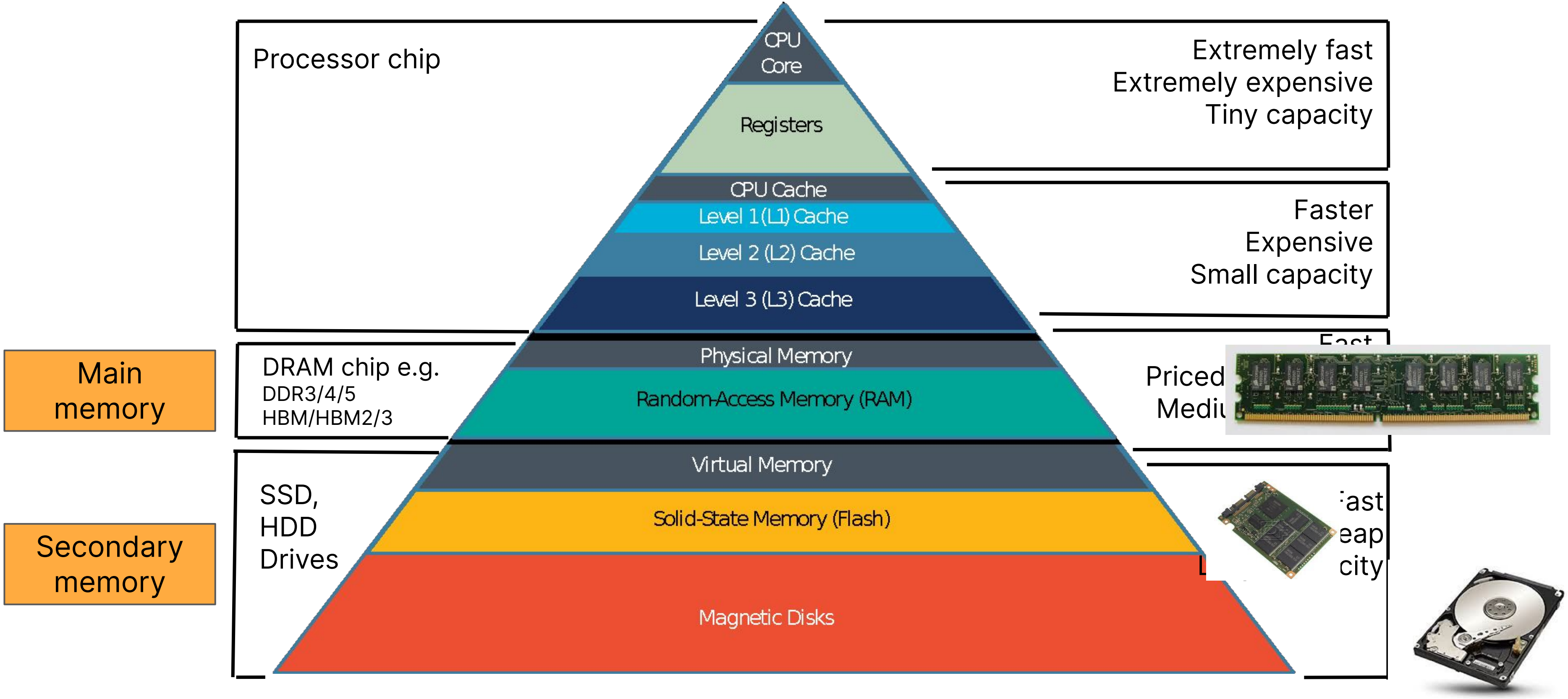
What does the core of the OS do?

- OS is the (first) thing that runs when computer starts.
    - Starts services (100+).
        - File system, Network stack (Ethernet, WiFi, Bluetooth, ...), TTY (keyboard), etc.
- Provides interaction with the outside world:
    - Finds and controls all devices in the machine in a general way:
        - Relies on hardware specific “device drivers”
- Loads, runs and manages programs:
    - ***Isolation***: Each program runs (i.e., appears to run) in its own little world.
    - Resource-sharing: Multiple programs share the same resources:
        - Memory
        - I/O devices: disk, keyboard, display, network, etc.
    - Time-sharing: Processor (CPU) runs multiple processes.

Multiprogramming at a High Level

- The OS manages multiprogramming, which is running multiple applications (processes) “simultaneously” on one CPU.
  - (vs. multiprocessing: running processes simultaneously on different CPUs. The OS also manages this.)
- This is achieved via OS context switches, i.e., switches between processes very quickly (on the human time scale):
  - Save current process state (program counter, registers, etc.)
  - Load next process state to execute next instruction on CPU
  - Do not switch out data between main memory and disk! Too costly…

Great Idea #3: Principles of Locality / Memory Hierarchy


Main memory

Secondary memory

Processor chip

DRAM chip e.g.
DDR3/4/5
HBM/HBM2/3

SSD,
HDD
Drives

Extremely fast
Extremely expensive
Tiny capacity

Faster
Expensive
Small capacity

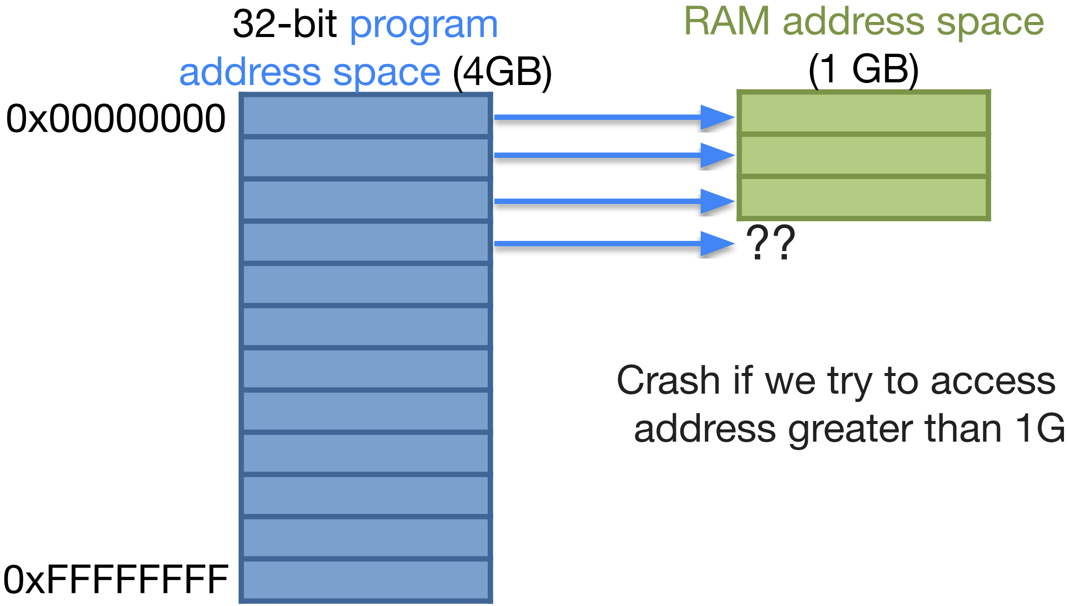
Fast
Priced
Mediu

ast
eap
city
L

Problems with Memory

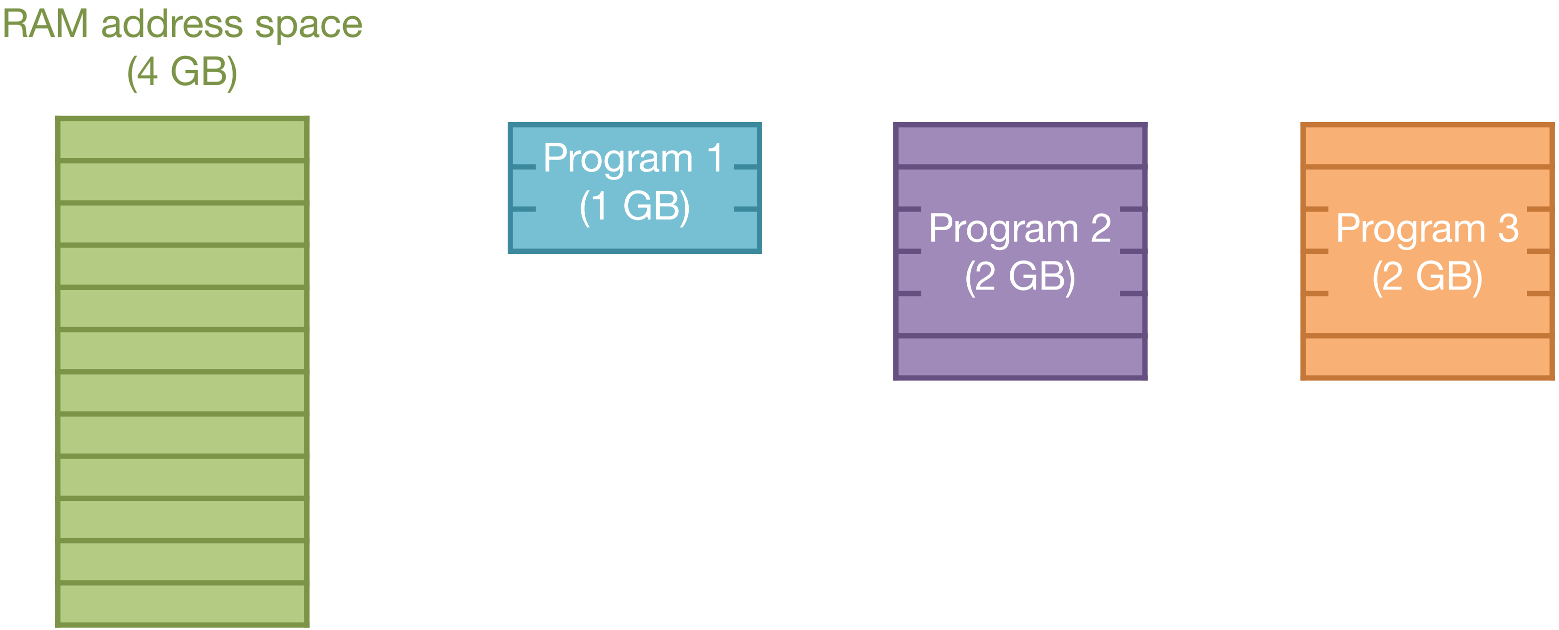
#1: Not Enough Space

- RISC-V32 provides a 32-bit address space
  - Q: How much memory can I access with a 32-bit address?
    - 2^{32} bytes = 4GB

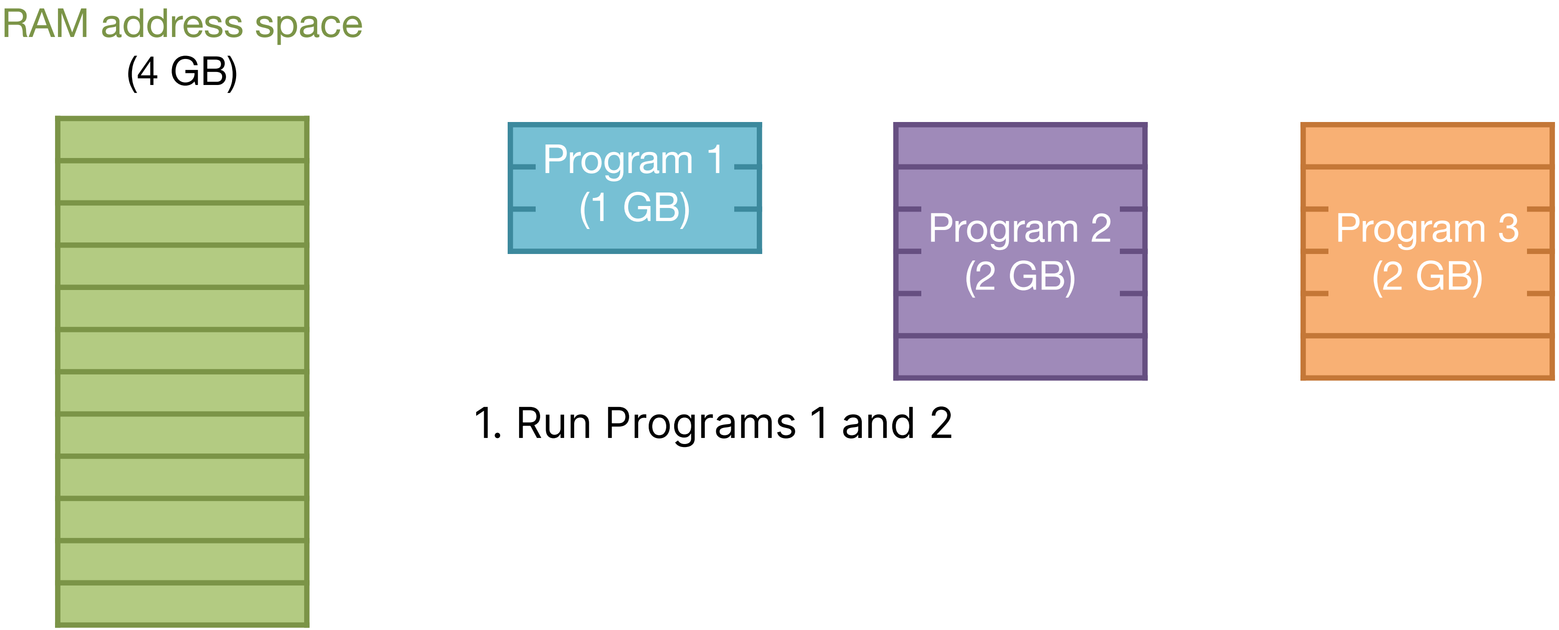


Crash if we try to access an address greater than 1GB

#2: Holes in Address Space



#2: Holes in Address Space



RAM address space
(4 GB)

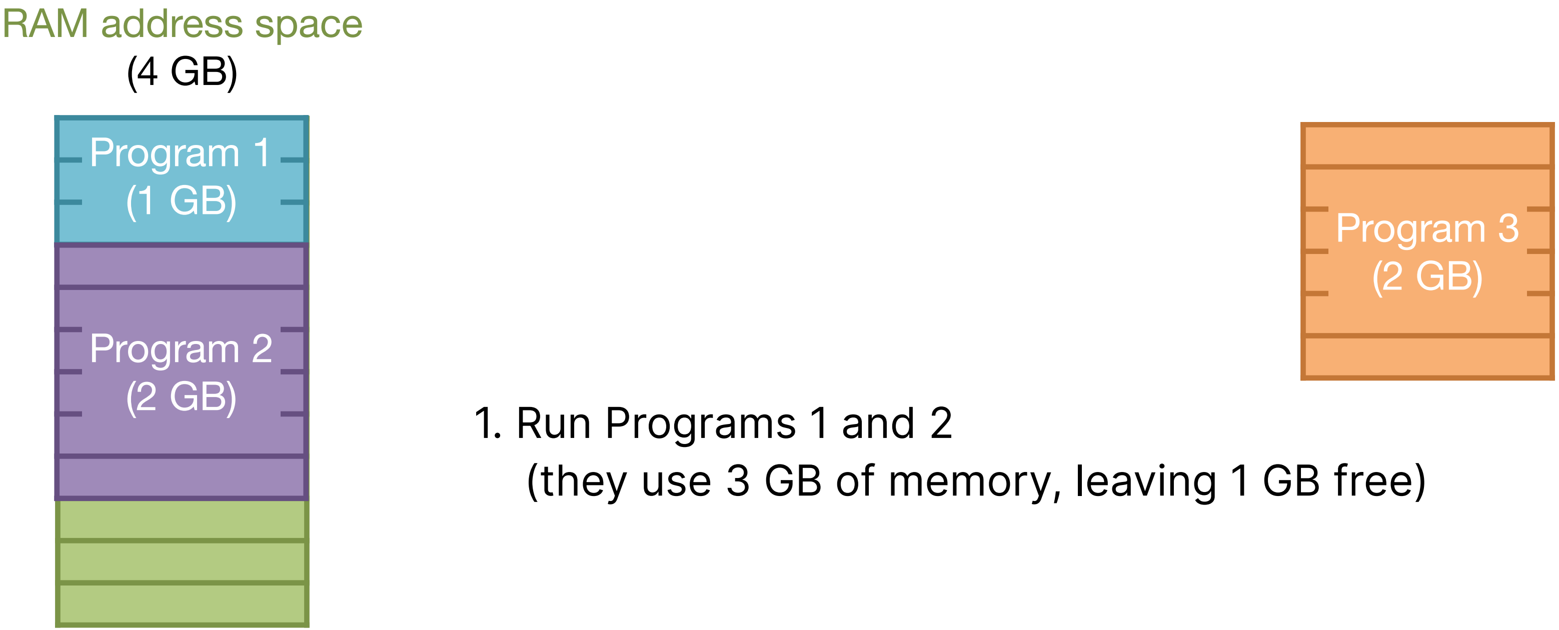
Program 1
(1 GB)

Program 2
(2 GB)

Program 3
(2 GB)

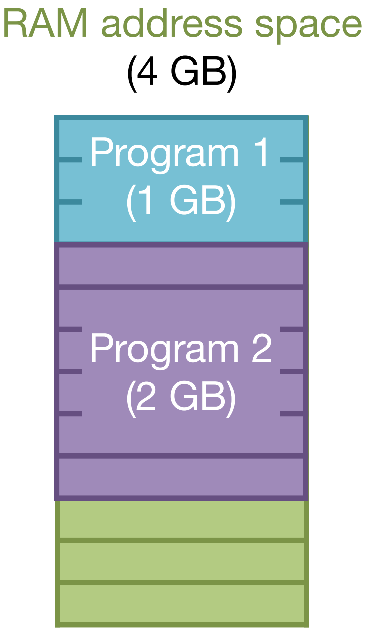
1. Run Programs 1 and 2

#2: Holes in Address Space



1. Run Programs 1 and 2
   (they use 3 GB of memory, leaving 1 GB free)

#2: Holes in Address Space

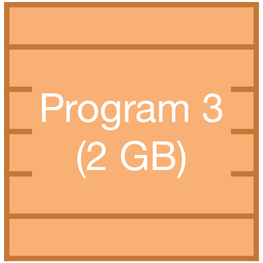


RAM address space

(4 GB)

Program 1
(1 GB)

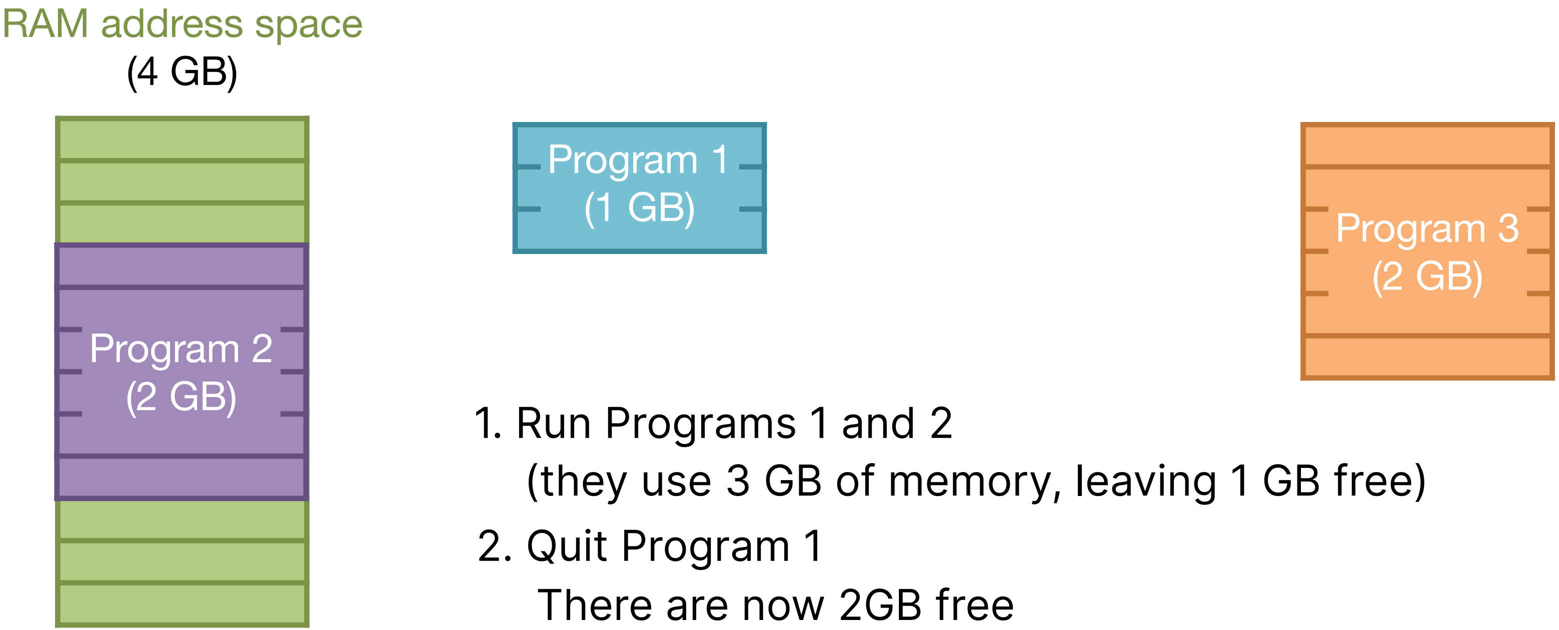
Program 2
(2 GB)



Program 3
(2 GB)

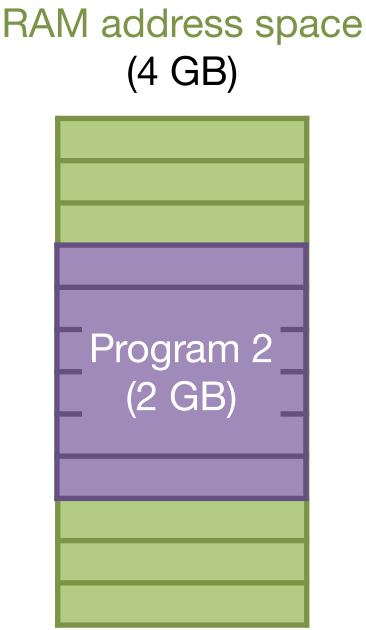
1. Run Programs 1 and 2
   (they use 3 GB of memory, leaving 1 GB free)
2. Quit Program 1

#2: Holes in Address Space

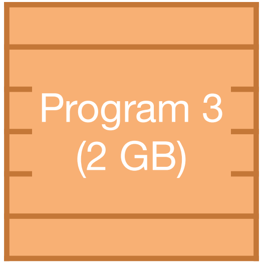


1. Run Programs 1 and 2
   (they use 3 GB of memory, leaving 1 GB free)
2. Quit Program 1
   There are now 2GB free

#2: Holes in Address Space



Program 1
(1 GB)



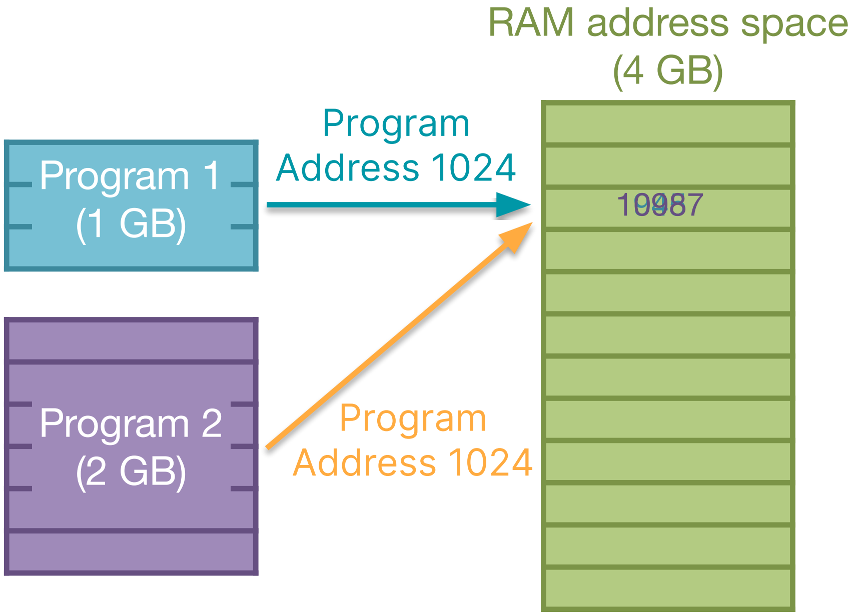
1. Run Programs 1 and 2
   (they use 3 GB of memory, leaving 1 GB free)
2. Quit Program 1
   There are now 2GB free
3. Try to run Program 3
   We can’t, even though there is enough space!

Memory Fragmentation

#3: Ensuring Protection from Other Programs

- Each program can access any 32-bit memory address
- What if multiple programs access the same address?
- They can corrupt or crash each other

1. Program 1 stores your bank account balance at address 1024
2. Program 2 stores your video game score at address 1024



How do we solve these problems?

Virtual Memory

Virtual Memory

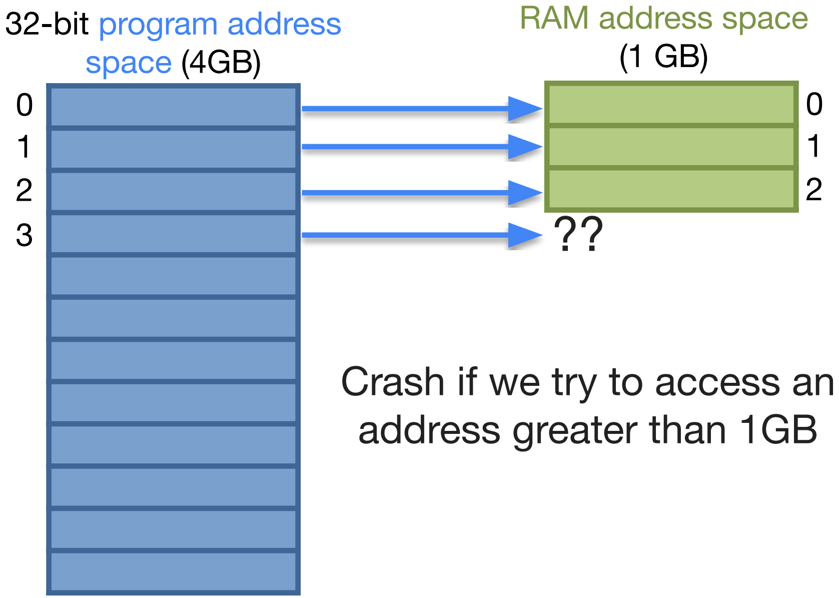
- Virtual memory is the next level in the memory hierarchy:
  - Give each process the **illusion** of a full memory address space that it has completely for itself.
  - Under the hood: working set of pages reside in main memory; other pages are in disk.
- Benefits:
  - Demand paging provides the ability to run programs larger than the primary memory (DRAM).
  - OS can share memory and protect programs from each other.
  - Hides differences between machine configurations.
- Today, more important for protection than space management.
  - (Historically, virtual memory predates caches.)

Virtual Memory: Indirection

Virtual memory takes program addresses and maps them to RAM addresses

No Virtual Memory

program address = RAM address



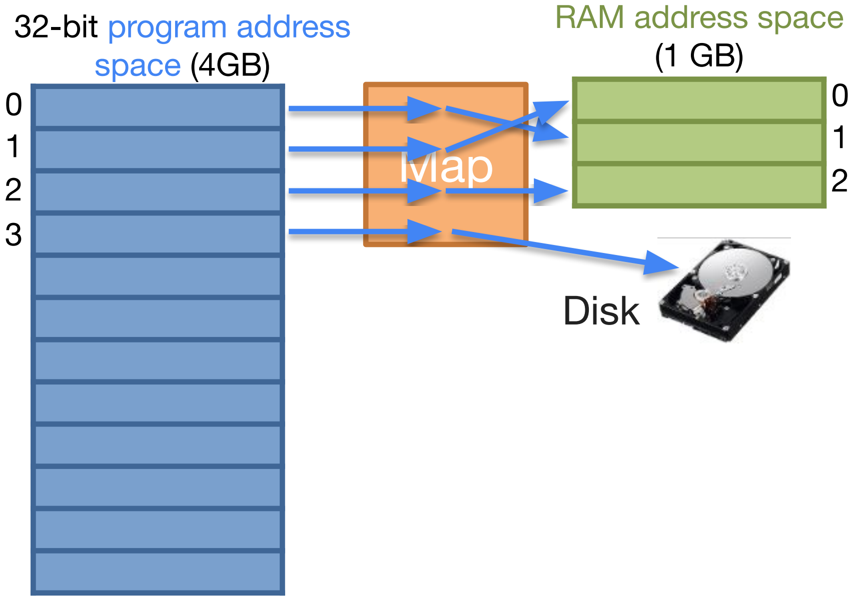
32-bit program address space (4GB)

RAM address space (1 GB)

Crash if we try to access an address greater than 1GB

Virtual Memory

program address maps to RAM address



32-bit program address space (4GB)

RAM address space (1 GB)

Virtual vs. Physical Addresses

- Address Space: set of addresses for all available memory locations.
  - Now, two kinds of memory addresses!

Virtual Address Space

- Set of addresses that the user program knows about
- Size is determined by what the programs access

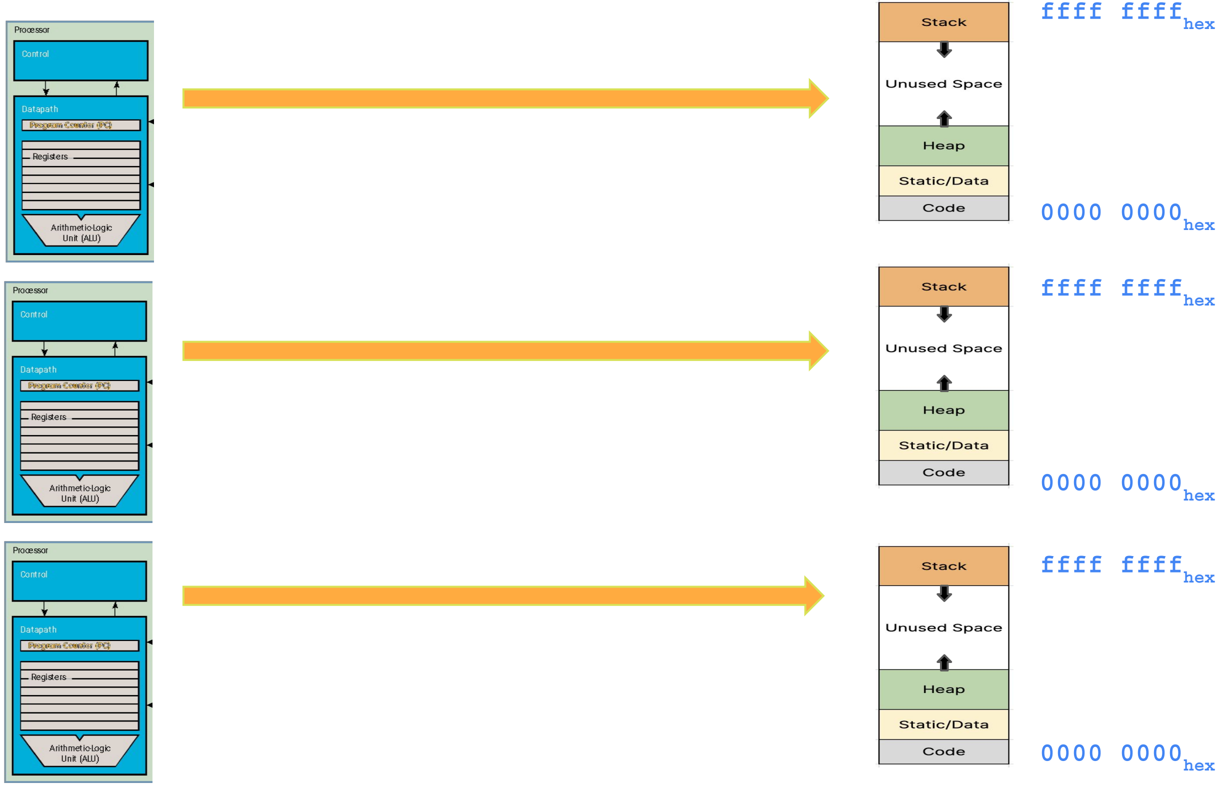
Physical Address Space

- Set of addresses that map to actual physical locations in memory
- Hidden from user applications
- Size is decided by how many bytes of memory we actually have

For each program, a memory manager maps (translates) between these two address spaces.

Virtual Address Space Illusion

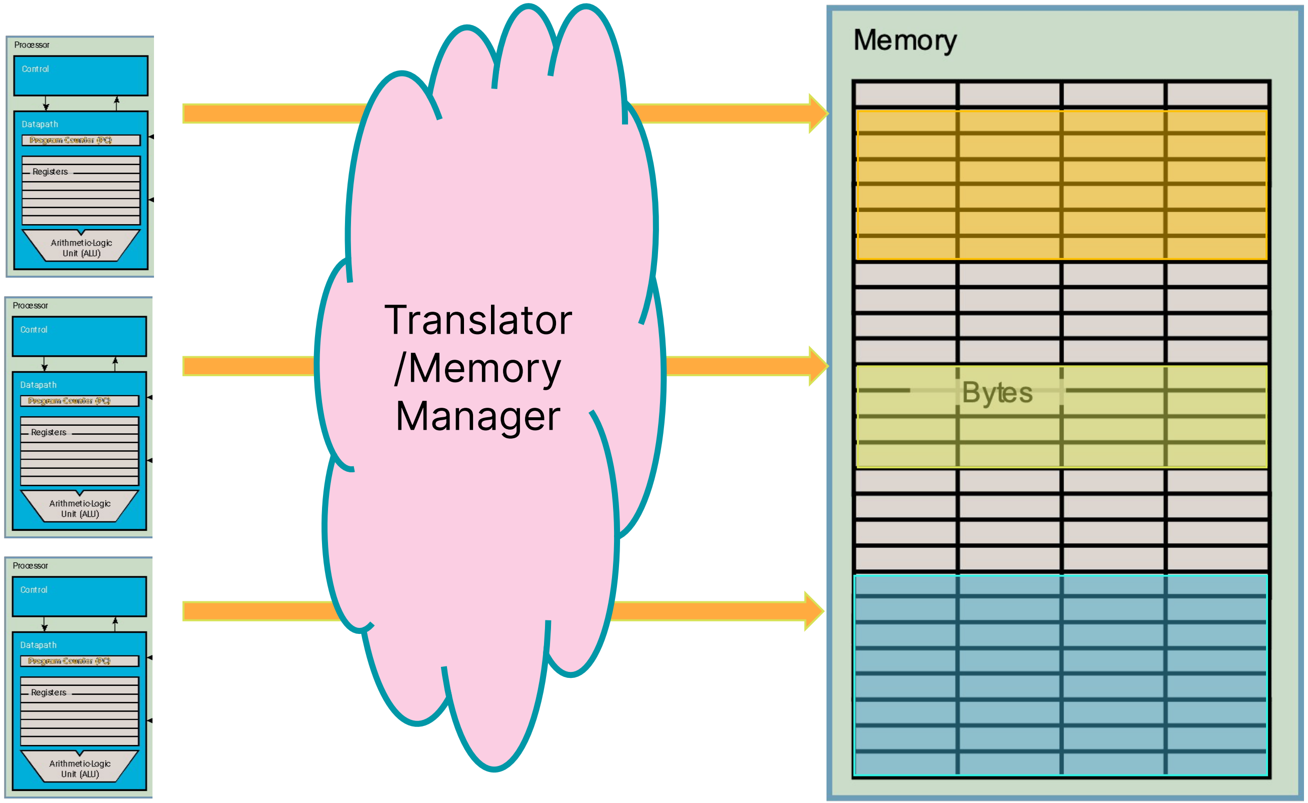
Different processes run simultaneously



Processes use **virtual addresses.**

Many processes, all using **same** (conflicting) addresses

Conceptual Memory Manager in OS



`7fff ffff`$_{hex}$

Memory uses **physical addresses**.

`0000 0000`$_{hex}$

Paged Memory

Let’s just… assume that caches don’t exist for now! (We’ll reintroduce it later, with Translation Lookaside Buffer)

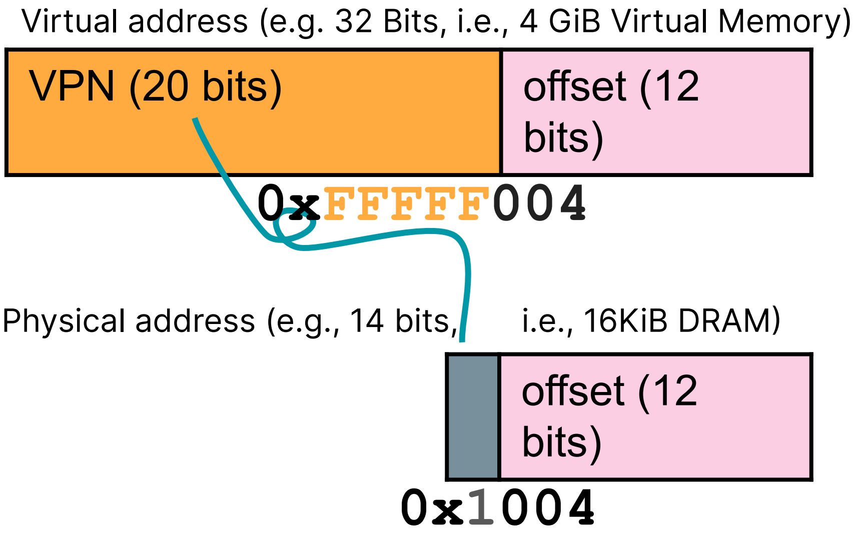
OS Virtual Memory Management Responsibilities

1. Map virtual addresses to physical addresses.
2. Use both memory and disk.
   - Give illusion of larger memory by storing some content on disk.
   - Disk is usually much larger and slower than DRAM.
3. Protection:
   - Isolate memory between processes.

Paged Memory

- The concept of "paged memory" dominates:
  - Physical memory (DRAM) is broken into pages.
  - A disk access loads an entire page into memory.
  - Typical page size: 4 KiB+ (on modern OSs). Let's assume it's 4KiB...
    - Need 12 bits of page offset to address all 4 KiB.
  - It's just another unit - just like how 1 byte is 8 bits.
  - Why? What would happen if we try to re-map every single word?

Memory translation maps Virtual Page Number (VPN) to a Physical Page Number (PPN).



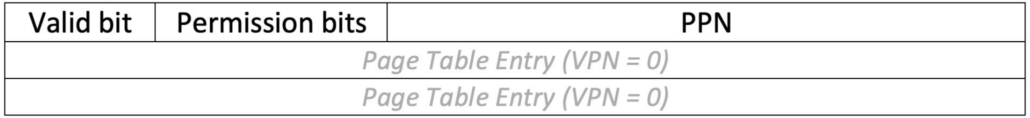
Virtual address (e.g. 32 Bits, i.e., 4 GiB Virtual Memory)
VPN (20 bits)
offset (12 bits)
0xFFFFF004
Physical address (e.g., 14 bits, i.e., 16KiB DRAM)
offset (12 bits)
0x1004

Page Tables

Page Tables

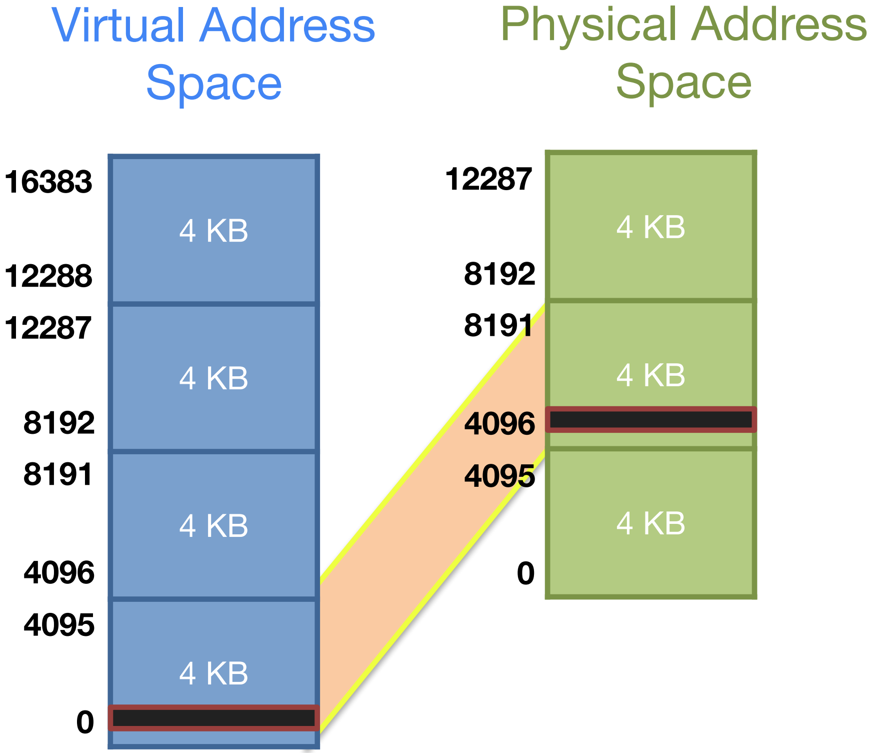
- Page Table: the map from **Virtual Addresses (VA)** to **Physical Addresses (PA)**
- Should we have one Page Table Entry (PTE) for every Virtual Address?
- If we have one entry for every word in our address space, how many entries would we have?
  - 2^{30} = 1 billion!

- Page Table: indexed by **virtual page number**



Valid bit	Permission bits	PPN
Page Table Entry (VPN = 0)		
Page Table Entry (VPN = 0)		

How do we map addresses with pages?



Page Table
VA -> PA

Map

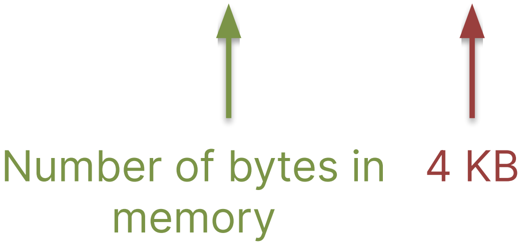
VA	PA
0-4095	4096-8191
…	…
…	…

Q: What is the physical address of virtual address 4?

4096 + 4 = 4100

Page Size

- Today, page tables are usually 4KB (1024 words)
- Q: How many pages do we need in our page table with 4KB pages on a 32-bit machine?
  - 2^{32} bytes / 2^{12} bytes = 2^{20} = 1 million



Number of bytes in memory

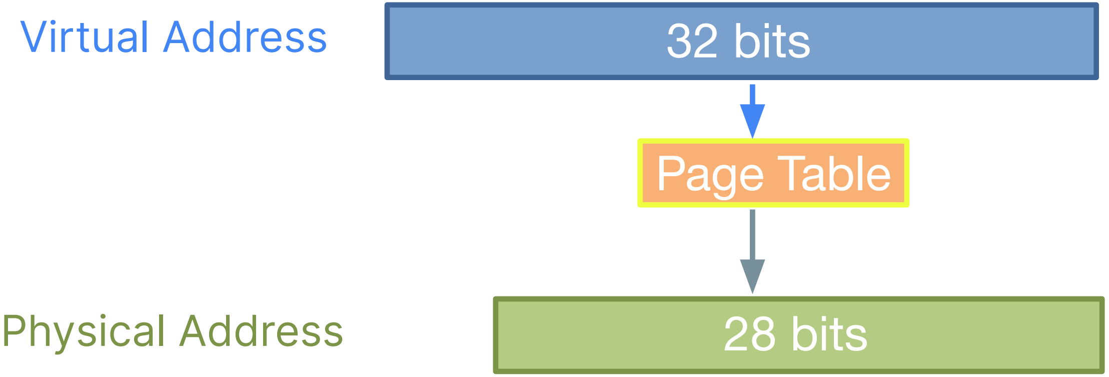
4 KB

- Q: How many entries do we have in our page table?
  - 1 million

Address Translation

Address Translation

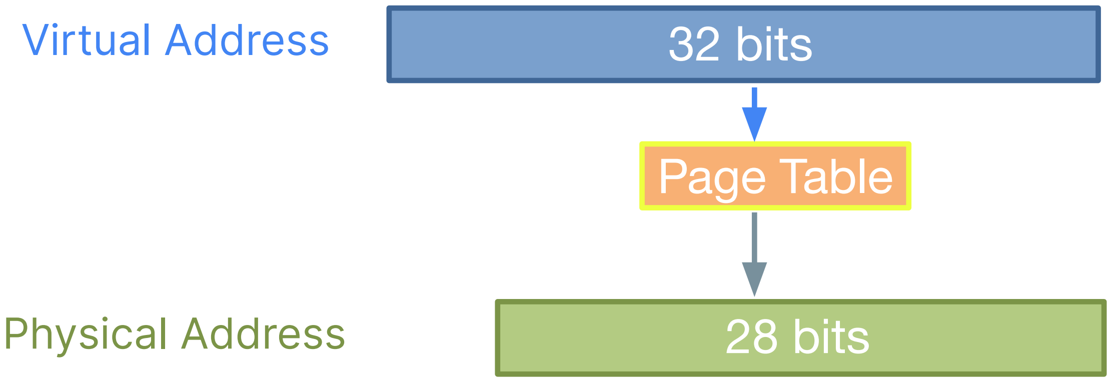
- What is the size of our virtual address and physical address on a 32 bit machine with 256 MB of RAM and 4KB pages?
  - VA size = 32 bits
  - PA size = $\log_2(256\text{ MB}) = \log_2(2^8 * 2^{20}) = 28$ bits
  - Offset size = 12 bits

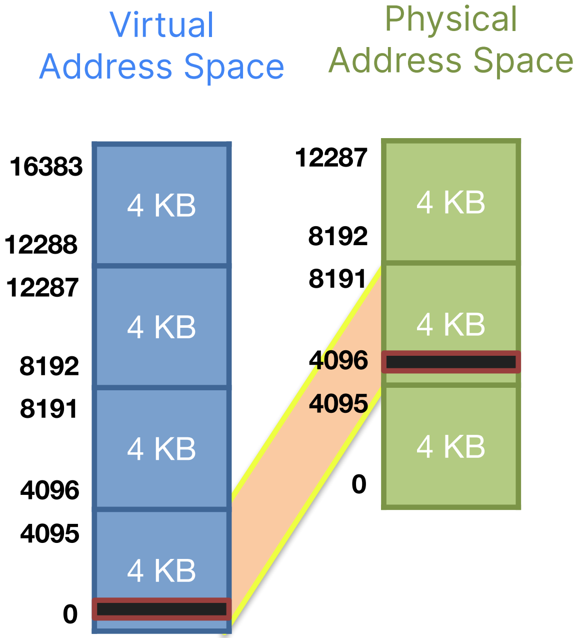


Why do we have more bits for the VPN than the PPN?
The virtual address space is larger than the physical address space! Not necessarily true all the time.

Address Translation

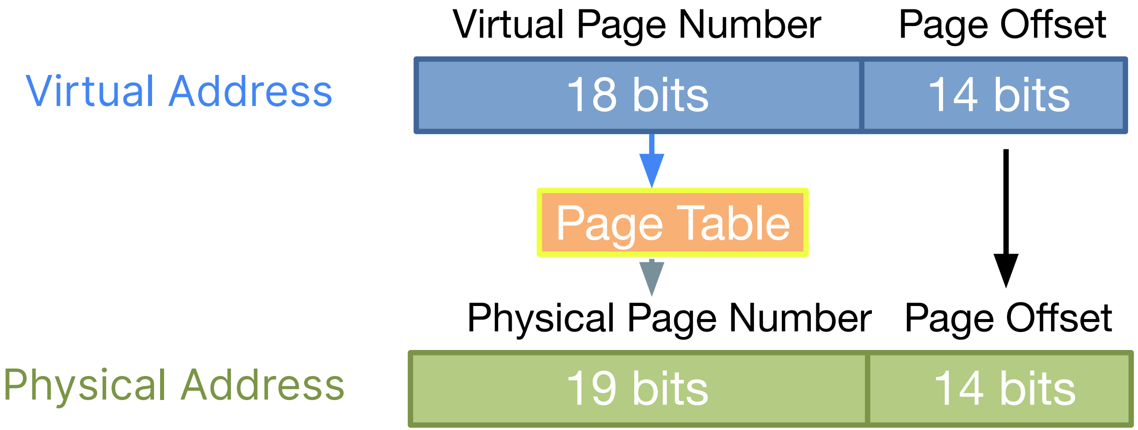
- What is the size of our virtual address and physical address on a 32 bit machine with 256 MB of RAM and 4KB pages?
    - VA size = 32 bits
        - VPN = 32 - 12 = 20 bits
    - PA size = $\log_2(256 \text{ MB}) = \log_2(2^8 * 2^{20}) = 28$ bits
        - PPN = 28 - 12 = 16 bits



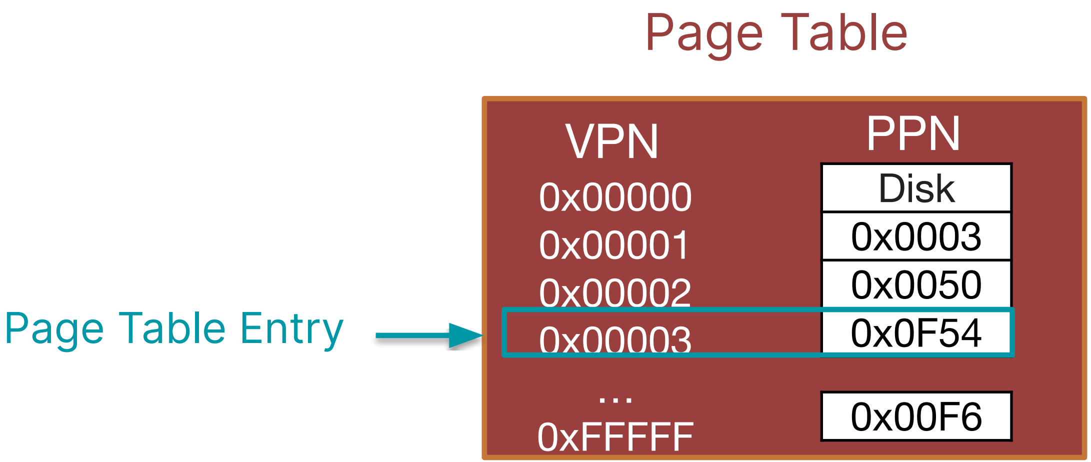


Your turn

- Q: How many bits would there be for the VPN, PPN, and page offset on a 32-bit machine with 8GB of RAM and 16KB pages?
    - Number of page offset bits = $\log_2(16\text{ KB}) = \log_2(2^4 * 2^{10}) = 14$
    - Number of VPN bits = 32 - 14 = 18
    - Number of PPN bits = $\log_2(8\text{GB}) - 14 = \log_2(2^3 * 2^{30}) - 14 = 33 - 14 = 19$



Translation Walk-Through



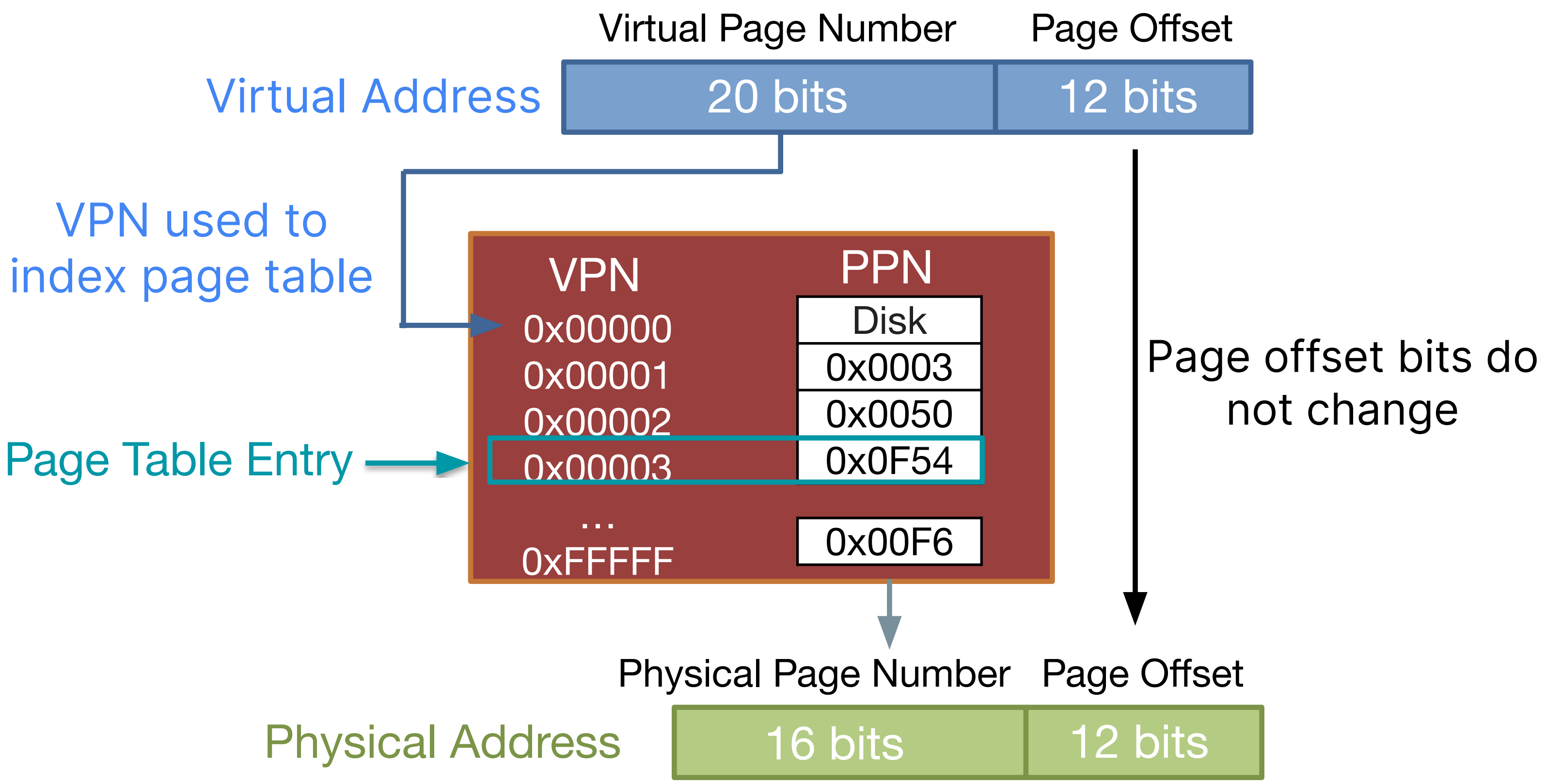
Page Table

VPN	PPN
0x00000	Disk
0x00001	0x0003
0x00002	0x0050
0x00003	0x0F54
…	
0xFFFFF	0x00F6

Page table contains mapping of every VPN to PPN

Each process has its own page table

Translation Walk-Through



Virtual Address

Virtual Page Number	Page Offset
20 bits	12 bits

VPN used to index page table

Page Table Entry

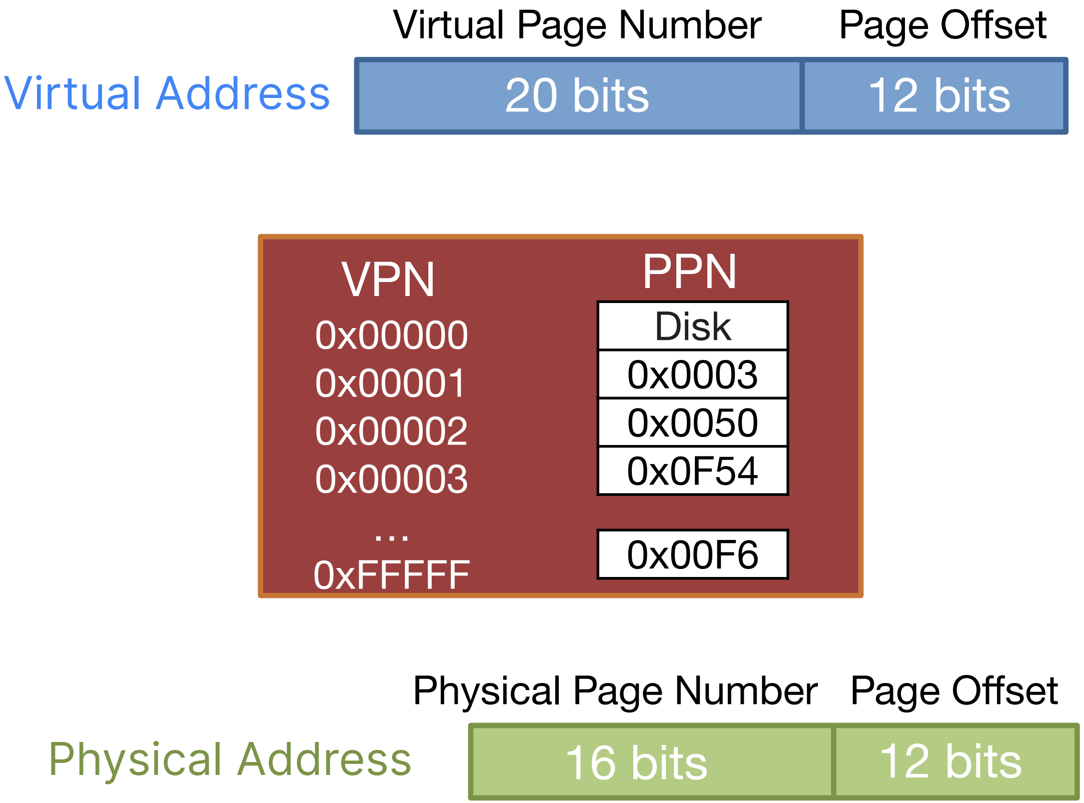
VPN	PPN
0x00000	Disk
0x00001	0x0003
0x00002	0x0050
0x00003	0x0F54
...	
0xFFFFF	0x00F6

Page offset bits do not change

Physical Address

Physical Page Number	Page Offset
16 bits	12 bits

Example Translation #1



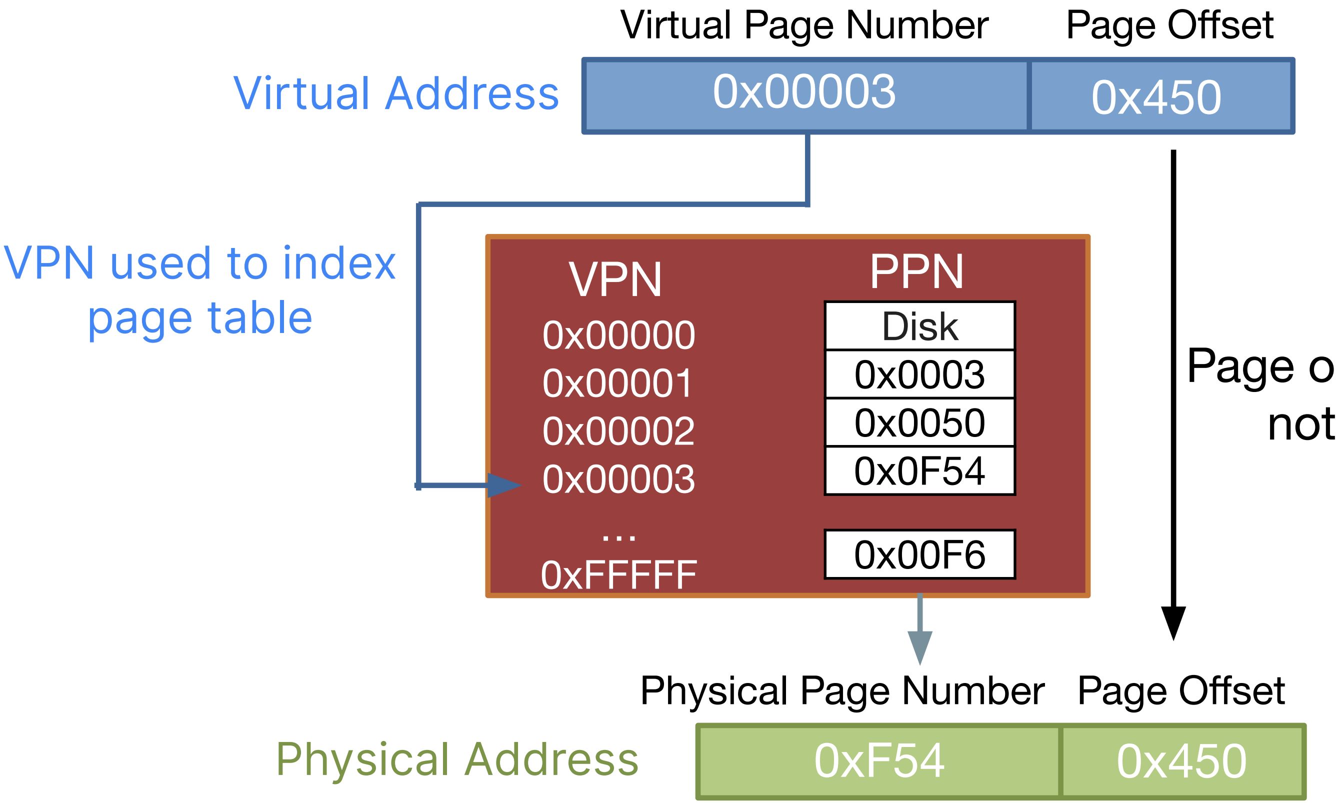
	Virtual Page Number	Page Offset
Virtual Address	20 bits	12 bits

0x00003450

VPN	PPN
0x00000	Disk
0x00001	0x0003
0x00002	0x0050
0x00003	0x0F54
…	
0xFFFFF	0x00F6

	Physical Page Number	Page Offset
Physical Address	16 bits	12 bits

Example Translation #1



VPN used to index page table

VPN	PPN
0x00000	Disk
0x00001	0x0003
0x00002	0x0050
0x00003	0x0F54
…	
0xFFFFF	0x00F6

Page offset bits do not change

0x00003450

↓

0x0F54450

OS Virtual Memory Management Responsibilities

✅ Map virtual addresses to physical addresses.

✅ Use both memory and disk.

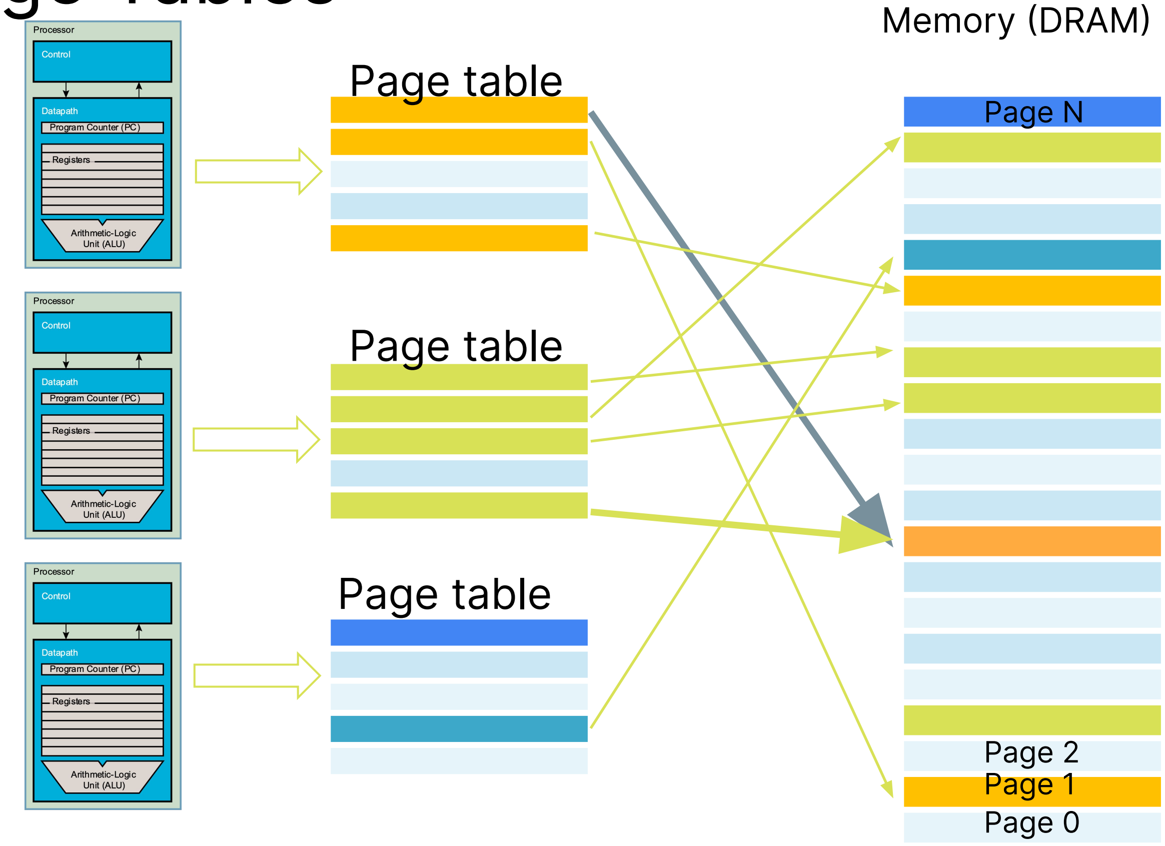
- Give illusion of larger memory by storing some content on disk.
- Disk is usually much larger and slower than DRAM.

??? Protection:

- Isolate memory between processes.
- Ideas?

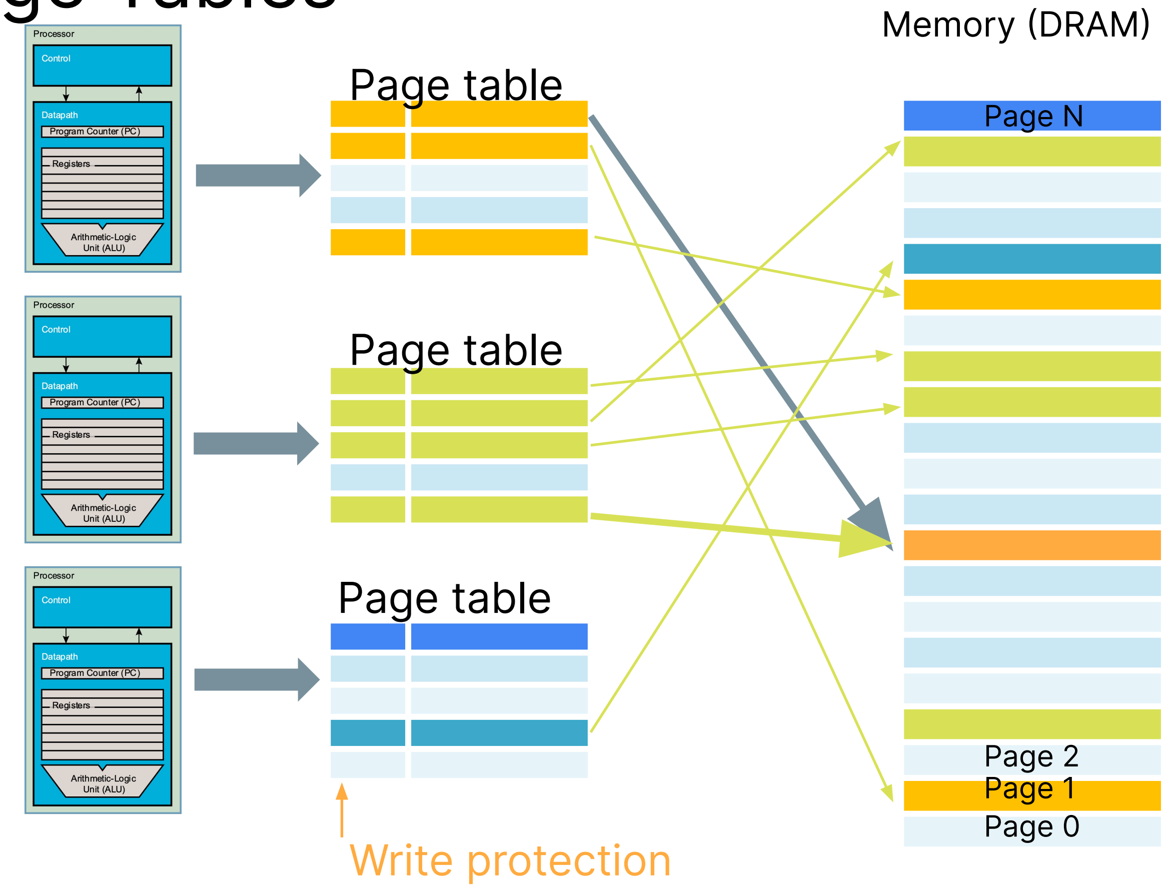
Protection with Page Tables

- Each process has a dedicated page table.
  - OS keeps track of which process is active.
- Isolation: Assign processes different pages in DRAM
  - Prevents accessing other processors’ memory
  - Page tables managed by OS
- Sharing is also possible:
  - OS may assign same physical page to several processes, e.g., system data



Protection with Page Tables

- Page Table Entry also includes a write protection bit.
- If on, then page is "protected":
  - e.g., program code, system data, etc.
  - Writing to a protected page triggers an exception. E.g. Segfault!
  - Exceptions are handled by OS. (more later)



Page Faults

- Page table entries store status to indicate if the page is in memory (DRAM) or only on disk.
  - On each memory access, check the page table entry “valid” status bit.
  - The “valid” bit indicates whether there is already a mapping.
- Valid ⇒ In DRAM
  - Read/write data in DRAM
- Not Valid ⇒ On disk
  - Triggers a **Page Fault**; OS intervenes to allocate the page into DRAM.
  - If out of memory, first evict a page from DRAM.
  - Store evicted page to disk.
  - Read requested page from disk into DRAM.
  - Finally, read/write data in DRAM.

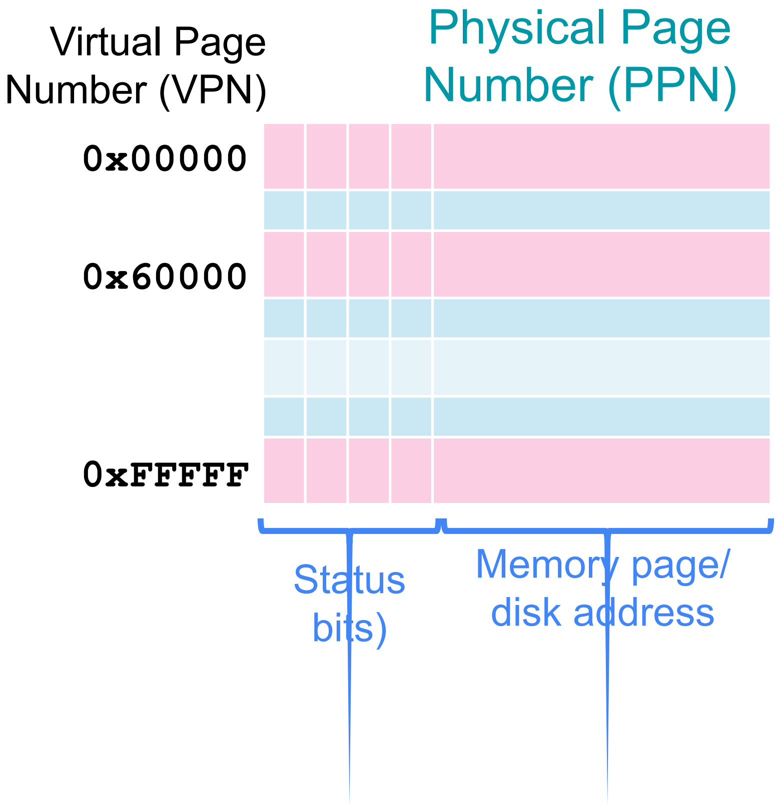
The *page replacement policy* (e.g., LRU/FIFO/random) is usually done in OS/software; this overheard << disk access time.

Memory’s Write Policy?

- DRAM acts like a “cache” for disk.
    - Should writes always go directly to disk (write-through), or
    - Should writes only go to disk when page is evicted (write-back)?
- Answer: All virtual memory systems use write-back.
    - Disk accesses take too long!

Page Table Metadata: Status Bits

- Write Protection Bit
  - On: If process writes to page, trigger exception… segfault!!
- Valid Bit
  - On: Page is in RAM
- Dirty Bit
  - On: Page on RAM is more up-to-date than page on disk



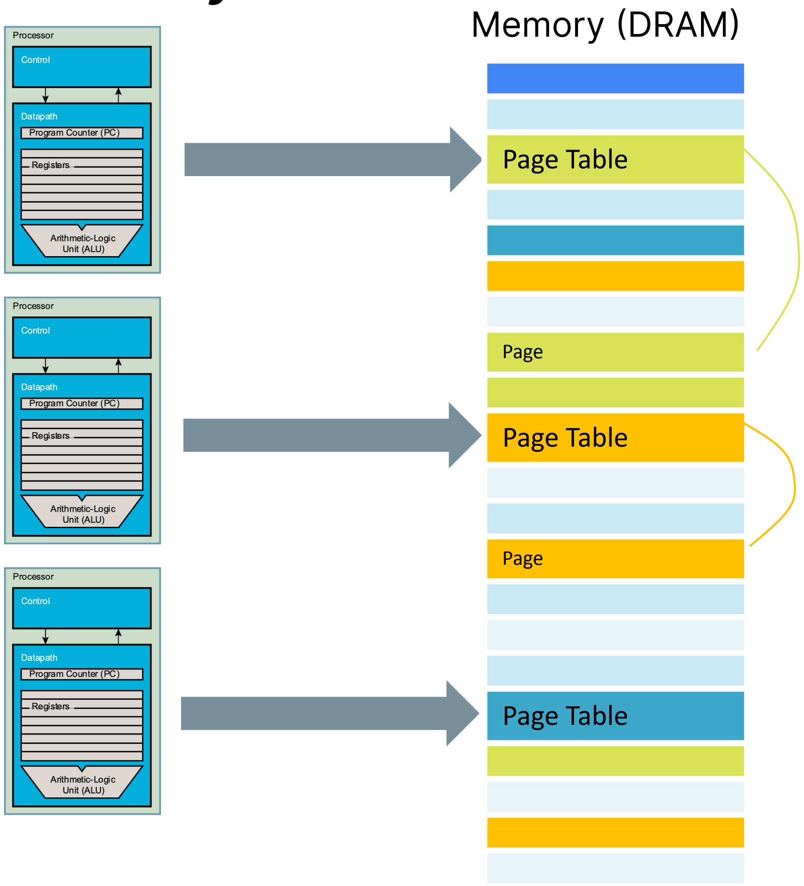
Virtual Page Number (VPN)
Physical Page Number (PPN)
0x00000
0x60000
0xFFFFF
Status bits)
Memory page/ disk address

Page Tables Are Stored in Memory

- E.g., 32-Bit virtual address space, 4-KiB pages
    - Single page table size (suppose each entry is 4B, including status bits):
        - 4 × 2^{20} Bytes = 4-MiB
        - 0.1% of 4-GiB memory. Not bad. But much too large for a cache!
- For now, store page tables in memory (DRAM).
    - Caveat: Two (slow) memory accesses per lw/sw on cache miss!

Page Tables Are Stored in Memory

- Caveat: `lw`/`sw` then requires two memory accesses:
  - Read page table (stored in main memory) to translate to physical address
  - Read physical page, also in main memory
- To minimize the performance penalty:
  - Use a cache for frequently used page table entries … (more later, TLB)



Next Time

- How to make the process of memory translation more efficient!
    - What if we add a cache for translation?
    - What else can we do?

OS: Supervisor mode, exceptions

Supervisor Mode vs. User Mode

- If an application goes wrong (or rogue, e.g., malware), it could crash the entire machine!
- CPUs have a hardware supervisor mode (i.e., kernel mode).
  - Set by a status bit in a special register.
  - An OS process in supervisor mode helps enforce constraints to other processes, e.g., access to memory, devices, etc.
  - Supervisor mode is a bit like “superuser”...
    - Errors in supervisory mode are often catastrophic (blue “screen of death”, or “I just corrupted your disk”).
- By contrast, in user mode, a process can only access a subset of instructions and (physical) memory.
  - Can change out of supervisor mode using a special instruction (e.g. sret).
  - Cannot change into supervisor mode directly; instead, HW interrupt/exception.
  - The OS mostly runs in user mode! Supervisor mode is used sparingly.

Exceptions and Interrupts

Exceptions

- Caused by an event during the execution of the current program.
- Synchronous; must be handled immediately.
- Examples:
  - Illegal instruction
  - Divide by zero
  - Page fault
  - Write protection violation

Interrupts

- Caused by an event external to the current running program.
- Asynchronous to current program; does not need to be handled immediately (but should be soon).
- Examples:
  - Key press
  - Disk I/O

Traps Handle Exceptions/Interrupts

- The trap handler is code that services interrupts/exceptions.

asynchronous,
external

synchronous,
during (e.g. page fault)

1. Complete all instructions before the faulting instruction.
2. Flush all instructions after the faulting instruction.
    - Like pipeline hazard: convert to noops/”bubbles.”
    - Also flush faulting instruction.
3. Transfer execution to trap handler (runs in supervisor mode).
    - Optionally return to original program and re-execute instruction.

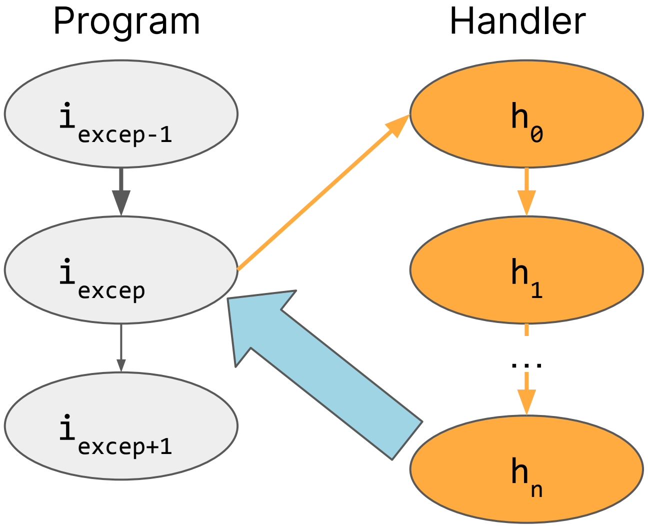
If the trap handler returns, then from the program’s point of view it must look like nothing has happened!

The Trap Handler

1. Save the state of the current program.
   - Save ALL of the registers!
2. Determine what caused the exception/interrupt.
3. Handle exception/interrupt, then do one of two things:

Continue execution of the program:

4. Restore program state.
5. Return control to the program.

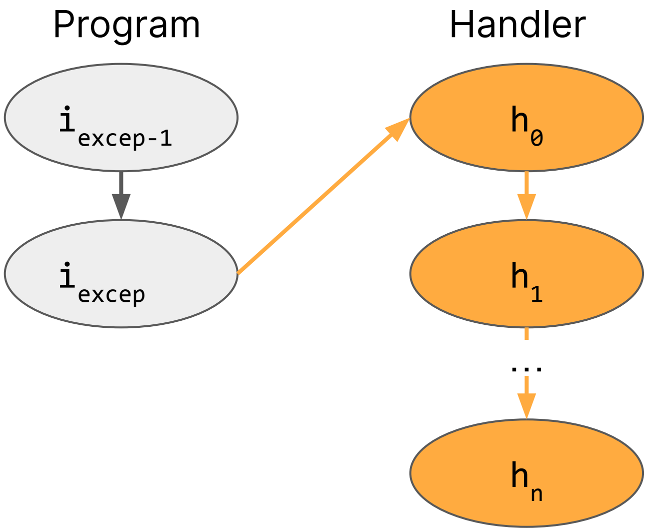


The Trap Handler

1. Save the state of the current program.
   - Save ALL of the registers!
2. Determine what caused the exception/interrupt.
3. Handle exception/interrupt, then do one of two things:

Continue execution of the program:

4. Restore program state.
5. Return control to the program.



Terminate the program:

4. Free the program resources, etc.
5. Schedule a new program.

Handling Context Switches

- Recall the context switch:
  - OS switches between processes (i.e., programs) by changing the internal state of the processor.
  - Allows a single processor to “simultaneously” run many programs.
- At a high-level:
  - The OS sets a timer. When it expires, perform a hardware interrupt.
  - Trap handler saves all register values, including:
    - Program Counter (PC)
    - Page Table Register (SPTBR in RV32I)
      - The memory address of the active process’s page table.
  - Trap handler then loads in the next process’s registers and returns to user mode.

Handling Page Faults

- Recall page faults:
  - An accessed page table entry has valid bit off ⇒ data is not in DRAM.
- Page faults are handled by the trap handler.
  - The page fault exception handler initiates transfers to/from disk and performs any page table updates.
  - (If pages needs to be swapped from disk, perform context switch so that another process can use the CPU in the meantime.)
    - (ideally need a “precise trap” [recoverable] so that resuming a process is easy.)
  - Following the page fault, re-execute the instruction.
- Side note: Write protection violations also trigger exceptions.