Systems I – CS 6013
Computer Architecture and Operating Systems
Lecture 17: Paging / TLB

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

*(adapted from slides by Ryan Stutsman, Andrea Arpaci-Dusseau, and Sarah Diesburg, and MSD presentations)

Lecture 17 – Topics

- Paging
- Segments
- Address Translations
- Translation Lookaside Buffer
  - Chapter 19 – This will be assigned in next week's reading, but if you are done with this week's readings…

Announcements / Questions

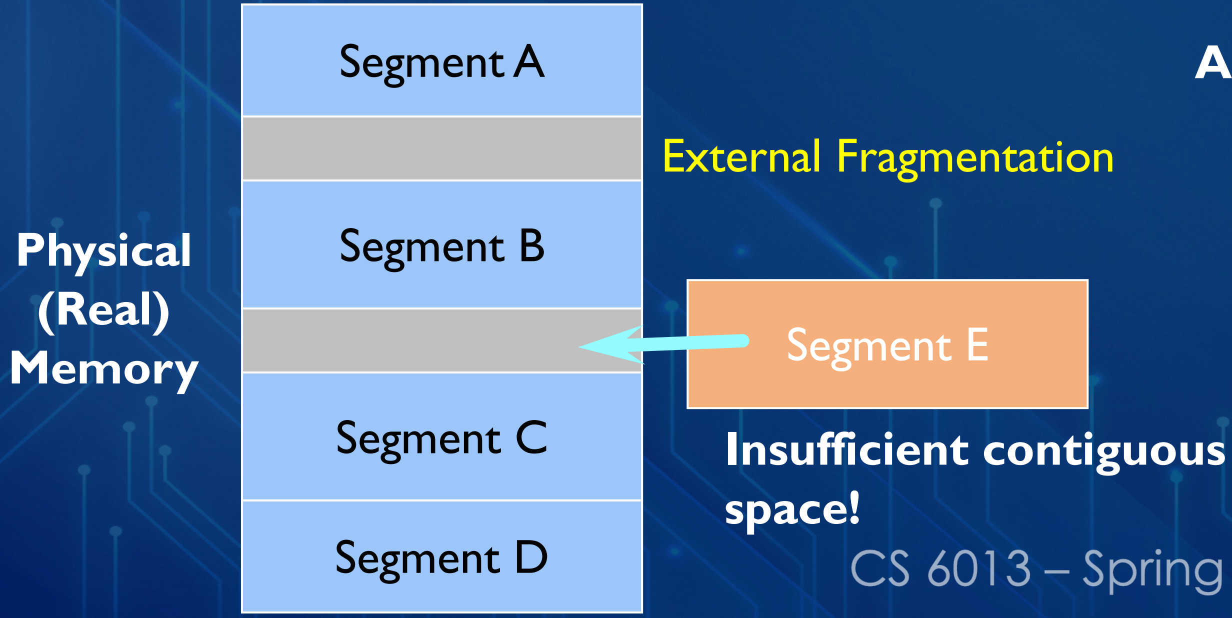
- Monday is a Holiday
- Industry Seminar - Tuesday

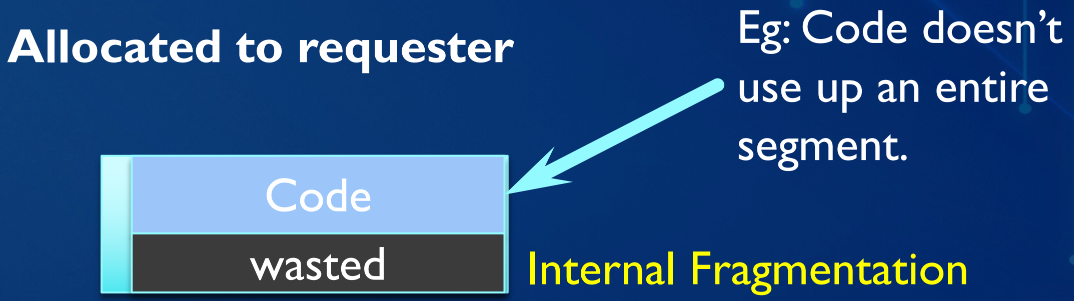
Review: Address Translation

- **Static relocation**
  - rewrite code and addresses before running (when program is loaded)
- **Dynamic Relocation**
  - **Base & Bound**: add base register to all accesses, trap on **virtual addresses** beyond bound
  - **Segmentation**: base, bound, permissions for multiple **address space** logical segments (code, heap, stack)
    - Discontinuous address spaces – supports sparsity/holes without wasting physical resources
    - Per-segment permissions
    - Enables sharing: code segment shared for multiple processes
    - Problem: **external fragmentation**

Problem: Fragmentation

- Definition of Fragmentation: Free memory that can't be usefully allocated
- Why?
  - Free memory (hole) is too small and scattered (external)
  - Unit of allocation does not match unit of need (internal)
- Types of fragmentation
  - **External**: Visible to allocator (e.g., OS)
  - **Internal**: Visible to requester (e.g., must allocate at some granularity)

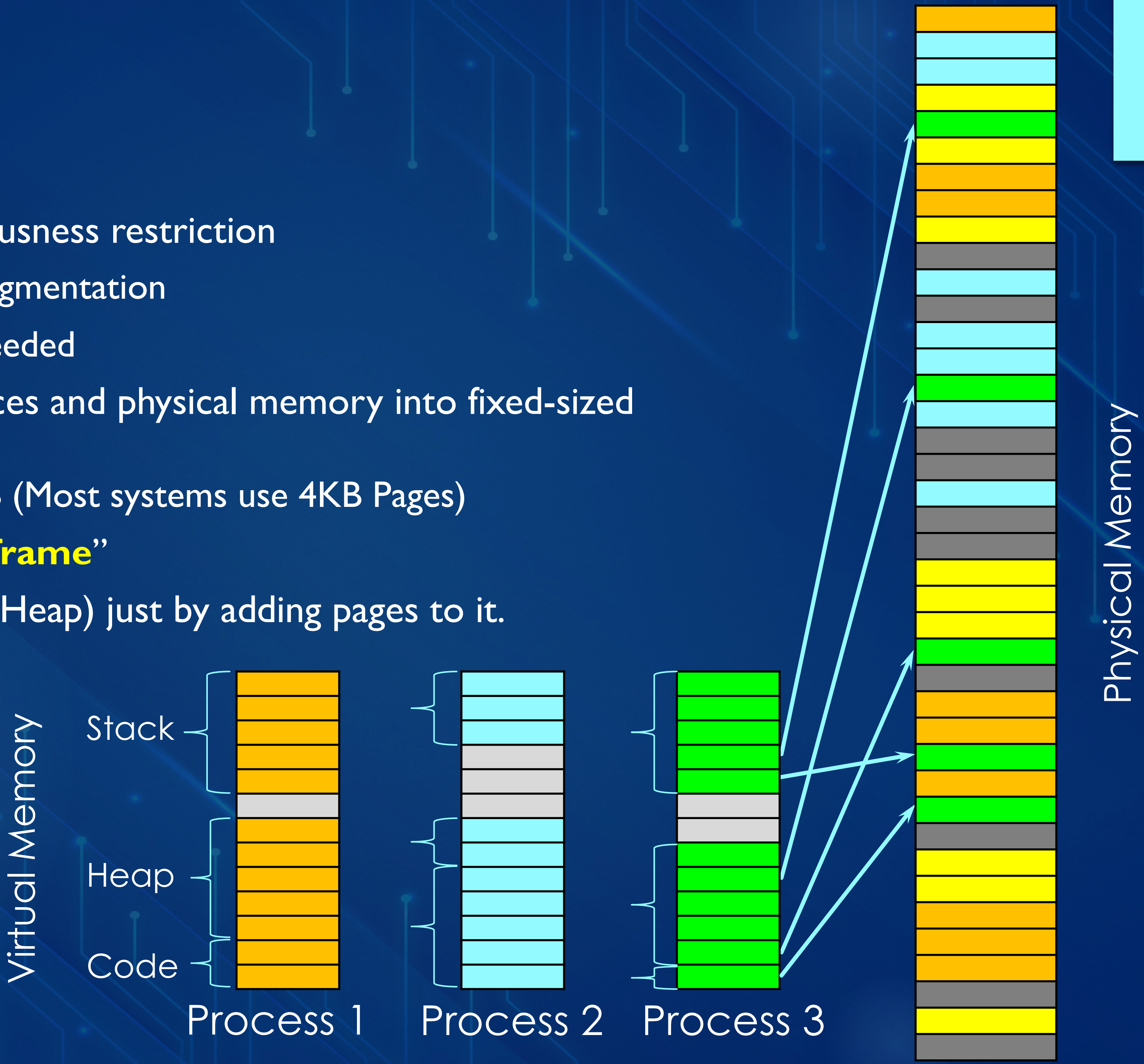




Paging

- Goal: Eliminate contiguousness restriction
  - Eliminate external fragmentation
  - Grow segments as needed
- Idea: Divide address spaces and physical memory into fixed-sized **pages**
  - Size: 2^n, Example: 4KB (Most systems use 4KB Pages)
  - Physical page: "**page frame**"
- Grow a segment (Stack, Heap) just by adding pages to it.

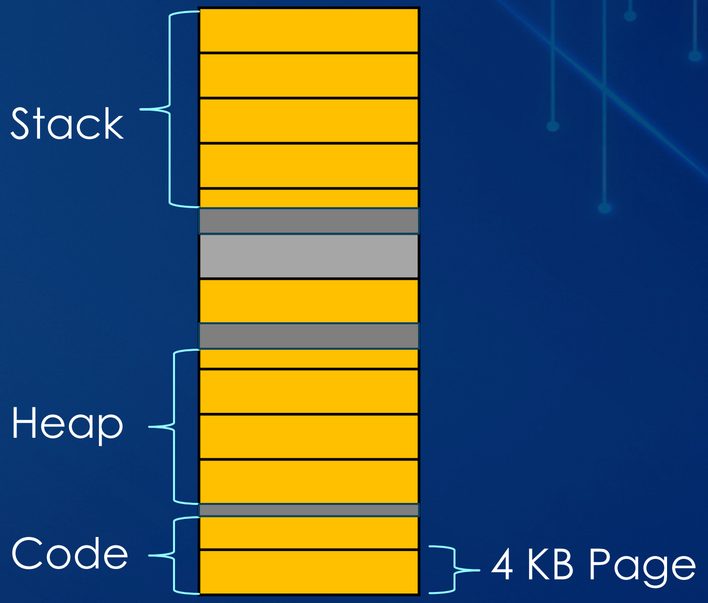
Note: this diagram implies that Physical Memory is much larger than Virtual Memory... However, this is not the case. Discuss…



Segments

- Per-process Virtual Address View
  - Each process has 3 segments (Code, Heap, Stack)
  - But number of pages does not correspond to number of segments

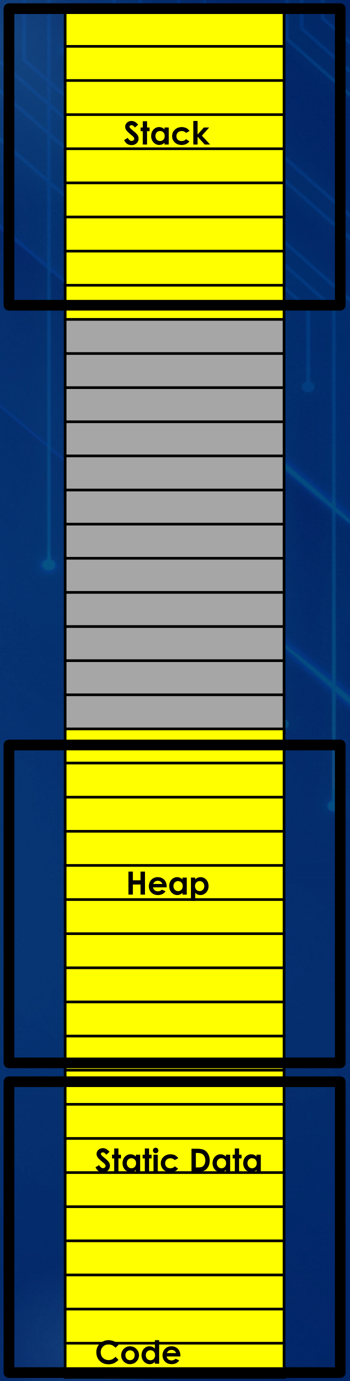
- How many pages is the code segment using?
  - 2 – well almost two.
- How many pages is the Heap using?
  - 4 – well almost four.
- What kind of fragmentation are we seeing here?
  - Internal… How bad is it?



Process 1

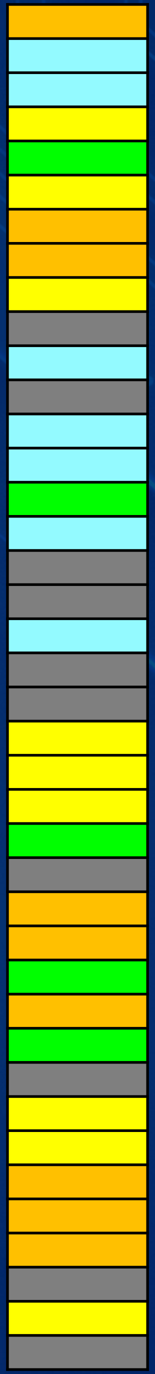
Segments and Internal Fragmentation

- When we consider more realistic pages sizes (with respect to the code/heap/stack sizes)
- We see that there really is minimal internal fragmentation.
  - Note yellow pages are allocated.
- Note: The size of pages as displayed in this diagram are still much too big to paint a fully accurate picture…



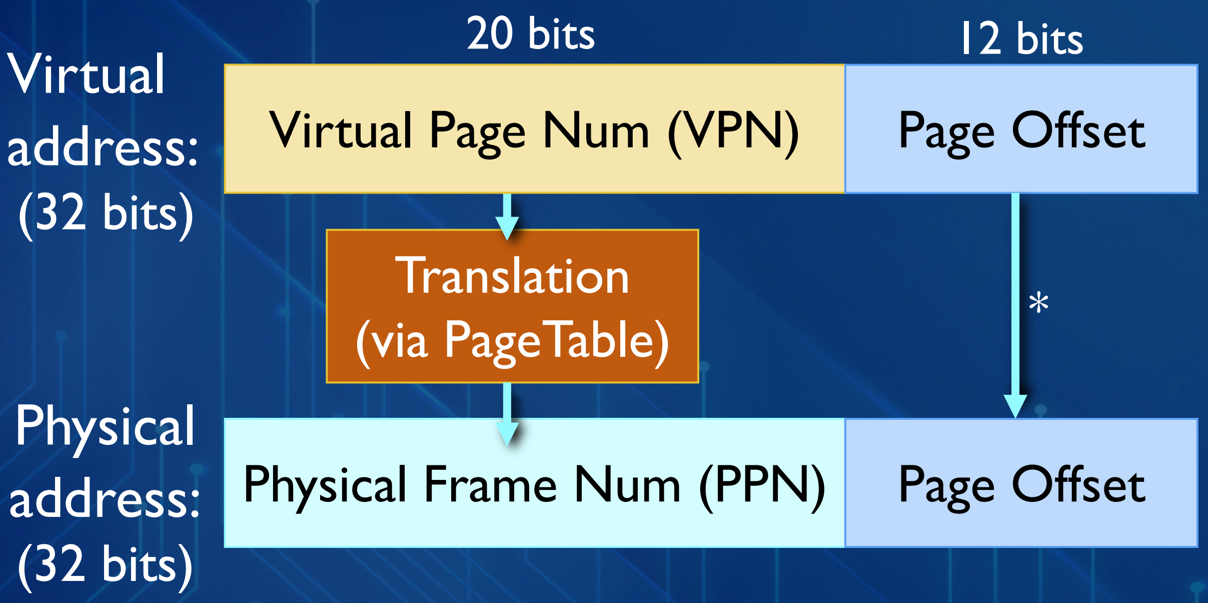
Pages vs Segments

- Segments map directly to logical segments of the Process Address Space (Code / Heap / Stack), Pages don't have to.
  - Code, Heap, Stack will be contained in multiple pages.
- Pages are always a fixed size (while segments do not have to be)
  - This eliminates external fragmentation of Physical memory…
    - Though of course an individual page can have internal fragmentation



Translation of Page Addresses

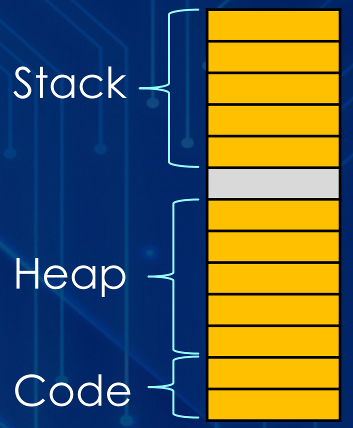
- Translate virtual address to physical address?
  - High-order bits of address designate page number
  - Low-order bits of address designate offset within page

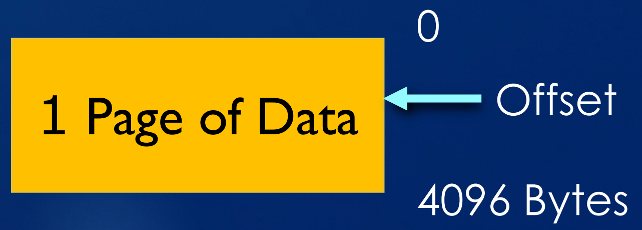


*No addition needed; just append bits correctly…

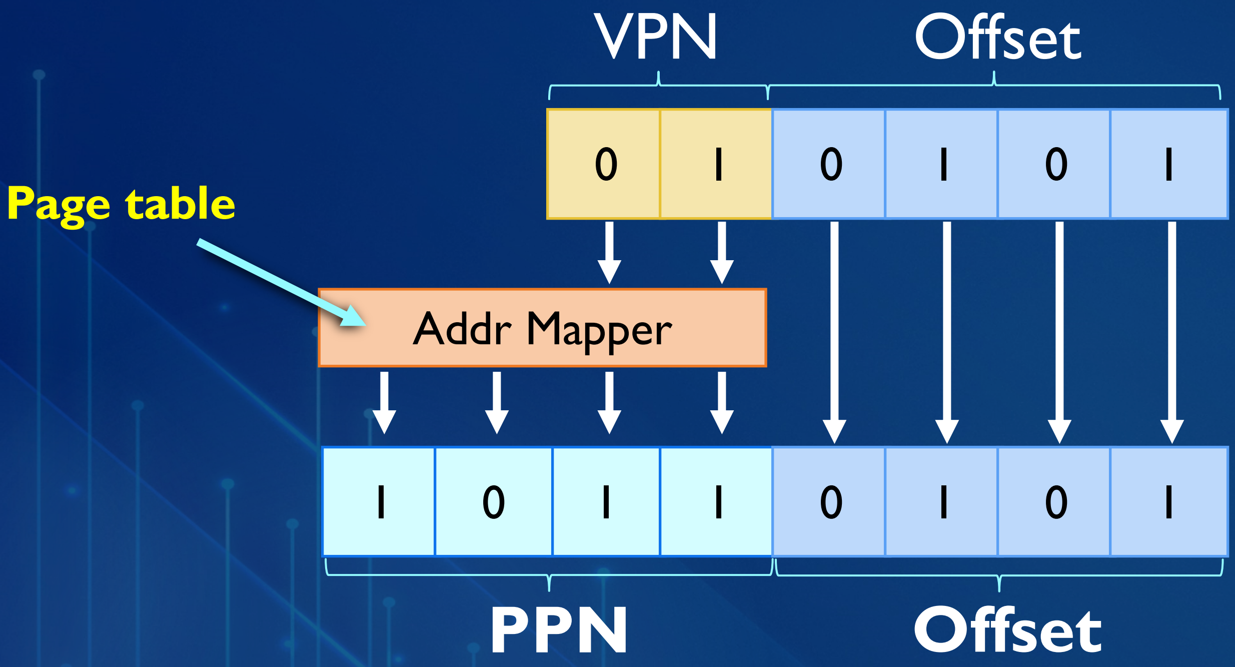
How does format of address space determine number of pages and size of pages?

- 32 bits == how much memory?
  - 4 GB (4,294,967,296)
- Split 32 bits into 20 and 12 bits.
- Why 12 bits for page offset?
  - Pages are usually 4K in size.





Virtual to Physical Page Mapping



Note: Weirdness!
Bits in VA need not equal bits in PA!

What if sizeof(VA) < sizeof(PA)?
Multiple proc's in mem at same time
What if sizeof(VA) > sizeof(PA)?
Only a portion of a proc can fit in real mem.
Benefits? Drawbacks?

Demand Paging

- Only the active "pages" / memory are in physical memory.
- Where are the inactive pages?
  - On disk… Eg: What is you are processing a 8 GB database?
- Have you ever seen a program run out of memory?

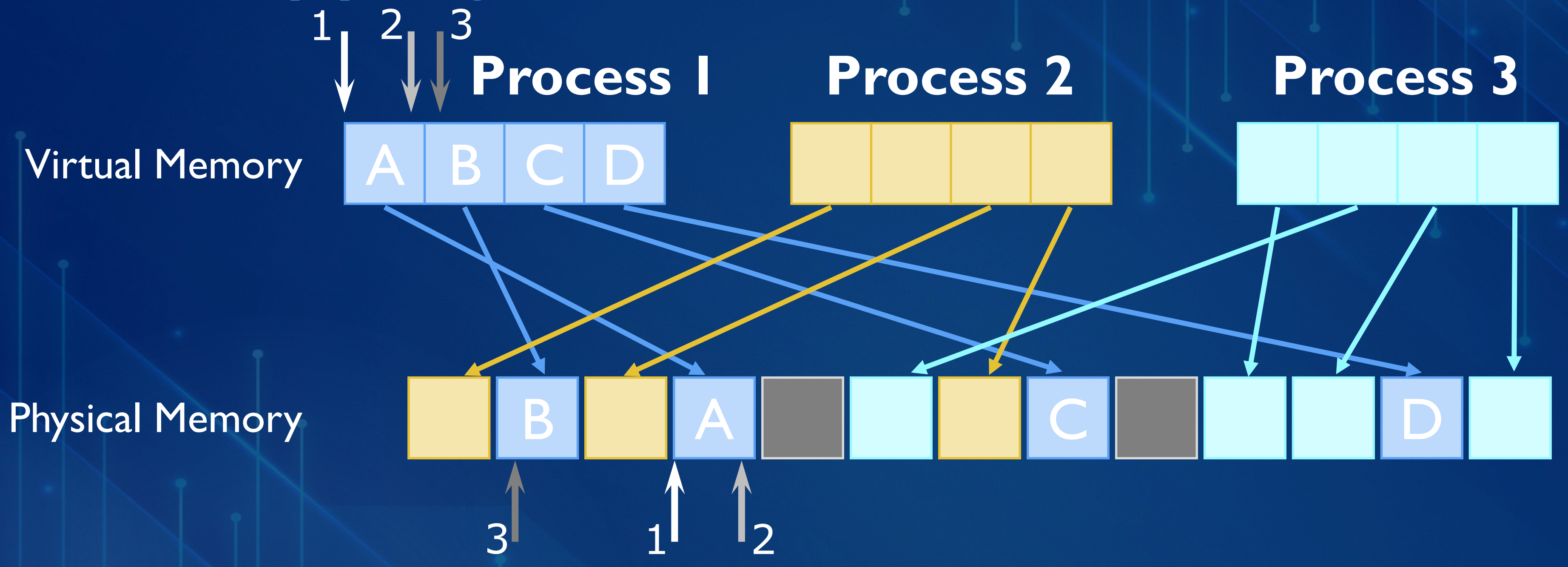
Translate from VPN to PPN?

Segmentation used a formula (e.g., PhysAddr = Base + Offset)

For paging, similar but need **many** more "bases" (no offsets)

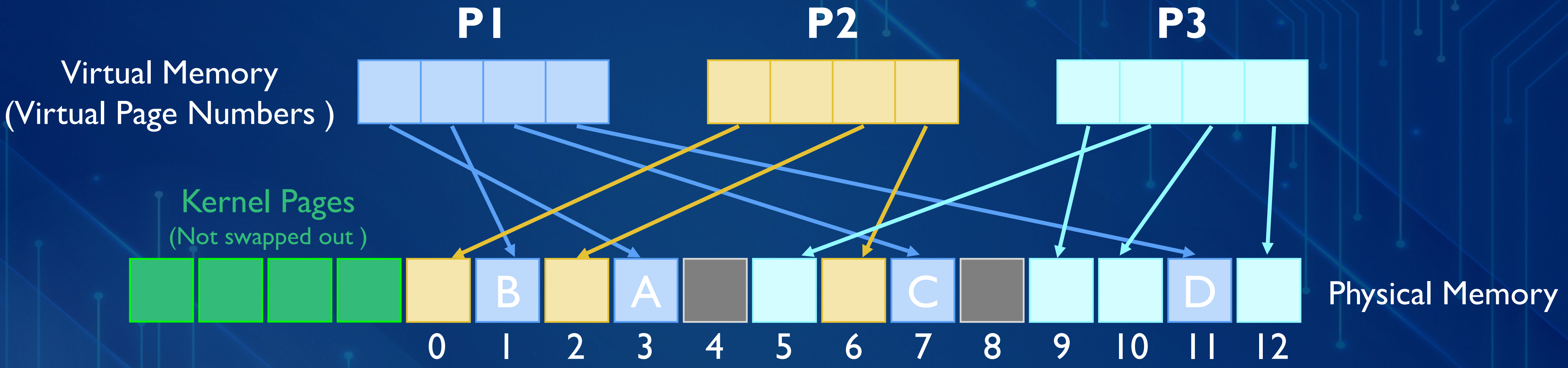
What data structure is good? Big array → **Page table**

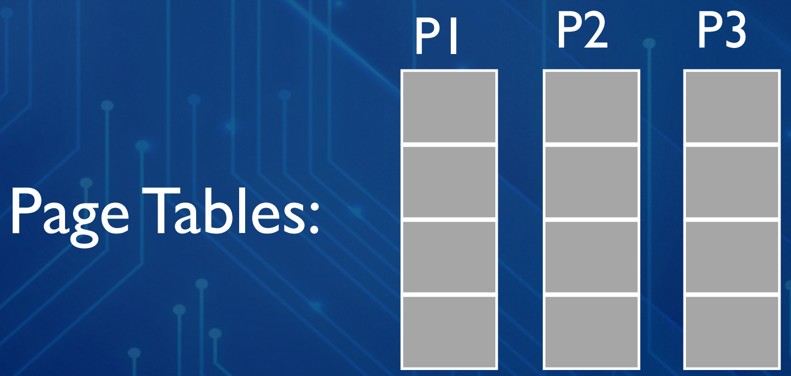
The Mapping



- Virtual address 2 and 3 are very close, but the physical address has changed drastically.
- Ramifications of this? Is this slower?
  - Only if the new page isn't already in memory.
  - Remember (disregarding caching for now), any given address in memory can be accessed at the same speed. (What does RAM stand for / mean?)
  - Remember, the CPU / MMU hardware maps the virtual address to the different physical addresses in a single cycle – it doesn't matter where in physical memory that address is.

Fill in the Page Table

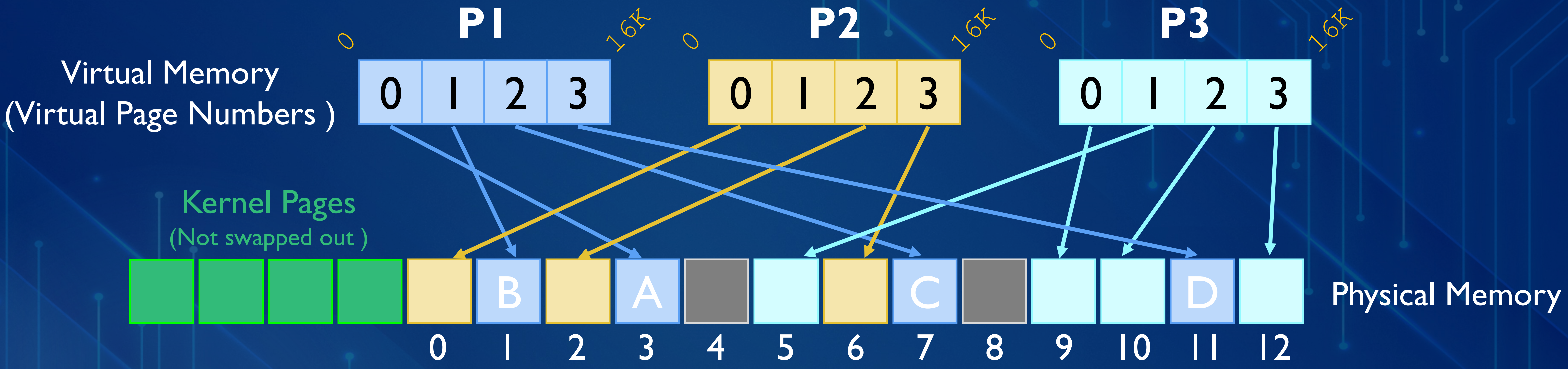




- What are the virtual page numbers for Process 1, Process 2, and Process 3?
- What are the physical page numbers they are mapped to?
  - Ie, what values are in the Page Tables?
- Note, kernel memory (the OS), is usually stored at the beginning of physical memory and is not swapped out.

Fill in the Page Table

- BTW, what is the range of the virtual address space of P1? P2? P3?
  - Ie: What are the memory addresses?

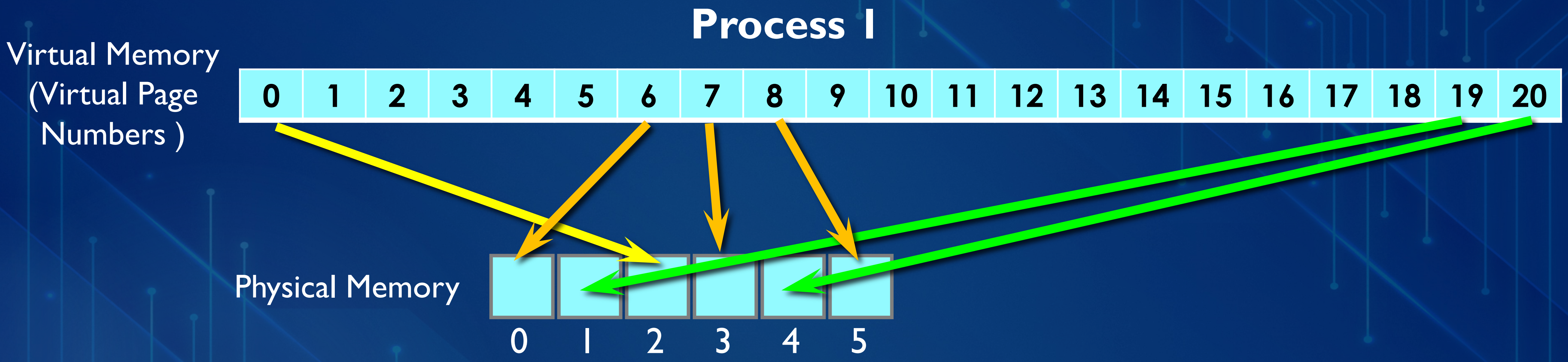


Page Tables:

P1	P2	P3
3	0	9
1	-1	5
7	2	10
11	6	12

- You can think of the page table as one giant table across all processes, or as each process having their own table.
- Notice in this example that the VA (virtual address) space is smaller than the PA (physical address) space.
- What would the picture look like if the VA space was larger than the PA space? What would that imply? See next slide.
- OS does all this work, programmer doesn't ☺

Virtual Address Space Bigger



- VA space bigger than physical memory
- Only parts of program are loaded into memory at any given time… which parts?
  - Some of the Code
  - Some of the Heap
  - Some of the Stack
- Where is the rest of the memory for the process?
  - On disk / not used

Where Are Page Tables Stored?

How big is a typical page table? Assuming…

32-bit VAs, 4 KB pages, and 4-byte **page table entries (PTEs)**

Answer: $2^{(32\text{ - }\log(4\text{ KB}))} * 4 =$ **4 MB** per process

- Page table size = num entries * size of each entry
- Num entries = num virtual pages = $2^{(\text{bits for VPN})}$
- Bits for VPN = 32 – number of bits for page offset

  = $32 - \log_2(4\text{ KB}) \Rightarrow 32 - 12 \Rightarrow 20$
- Num entries = $2^{20} \Rightarrow \sim 1$ million
- Page table size = Num entries * 4 bytes = 4 MB

Virtual address:

20 bits	12 bits
Virtual Page Number	Page Offset

Implication: Store each page table in memory

- Hardware finds **page table base with register** (cr3 on x86)

What happens on a context-switch?

- Save old page table base register in PCB of de-scheduled process
- Repoint page table base register to newly scheduled process

- Note: 64-bit addresses?
  - 52 bits for # of pages
  - 18,014,398,509,481,984
    - 18 Peta Bytes for Page Table
  - Note: Kilo, Mega, Giga, Tera, Peta…

xv6 PCB Process Control Block

```
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  pde_t* pgdir;                 // Page table
  char *kstack;                 // Bottom of kstack
  enum procstate state;         // Process state
  int pid;                      // Process ID
  struct proc *parent;          // Parent process
  struct context *context;      // swtch() here to run
  void *chan;                   // If !0, sleeping on chan
  struct file *ofile[NOFILE];   // Open files
  struct inode *cwd;            // Current directory
  …
};
```

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

- What is the point of the PCB?
  - To store information about a process – including CPU registers when it is swapped out.
- We should understand most of the PCB at this point.
- `pgdir` stores the cr3 value.

Other Page Table Information

What other info is in pagetable entries besides translation?

- **protection** bits (read/write/execute)
- **present** bit (trap if access a VPN that's not present)
- **accessed** bit (h/w sets when page is used)
- **dirty** bit (h/w sets when page is modified)

We'll discuss later when we cover swapping

Page table entries are just bits stored in memory

- Agreement between hardware and OS about interpretation

Memory Accesses with Pages

- Consider the instruction: `mov ax, di[ si ]`
  - `di` == array address, `si` == offset into that array
- We first have to load the mov instruction. Ignore this.
- The virtual address (VA) is `di + si` (think array access).
- The top bits of the VA are the Virtual Page Number (VPN).
- If Page Table (PT) is just an array, we lookup `PT[VPN]`.
- This is the PTE (Page Table Entry) for this VA.

Memory Accesses with Pages

- VA is `di + si`.
- We lookup `PT[VPN]` to get the Page Table Entry (PTE).
- Check permissions, get the PPN from the PTE.
- The physical address (PA) is PPN + lower bits of VA (the offset).

Page Table is slow! Doubles memory references

Memory Access w/ Pages (aka Translation)

Which steps are expensive when executing an assembly instruction?

It's the memory accesses (2 per) that are slow…

H/W algorithm/support for each memory reference:

1. extract **VPN** (virtual page num) from **VA** (virtual address)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory into register

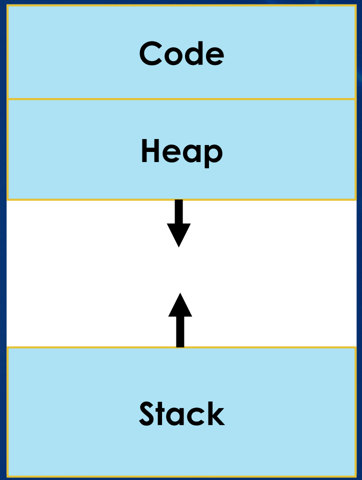
- How do we avoid / speed up these memory accesses?
  - Cache Page Table

Advantages of Paging

- No external fragmentation
  - Any page can be placed in any "slot" (frame) in physical memory
- Fast to allocate and free physical memory
  - Allocation: no searching for suitable free space
  - Free: doesn't have to coalesce with adjacent free space
  - Just keep all free page frames on a linked list
- Simple to "swap-out" pages to disk (future lecture)
  - Page size good match for disk I/O granularity
  - Can run process when some pages are on disk
    - Use "present" bit in PTE

Disadvantages of Paging

- Internal fragmentation: page size may not match process need
  - Tension: large pages → small tables but more int. fragmentation
- Additional memory reference to page table, inefficient
  - Page table must be stored in memory
  - MMU stores only base address of page table
  - Solution: Add TLBs (coming up next!)
- Storage for page tables may be substantial
  - Table is large (4 GB / 4 KB = 1 mil, 4 B PTE so 4 MB)
    - Even if some entries aren't needed
  - External fragmentation issues again:
    page tables must be contiguous in physical memory
  - Solution: multi-level page tables (page the page tables!)



Example: Iterating an Array

```
int sum = 0;
for (i=0; i<N; i++) {
  sum += a[i];
}
Assume 'a' starts at 0x3000
Ignore instruction fetches
```

VAs Loaded?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

PAs Loaded?

load 0x100C ; address of sum

load 0x7000 ; array access

load 0x100C

load 0x7004

load 0x100C

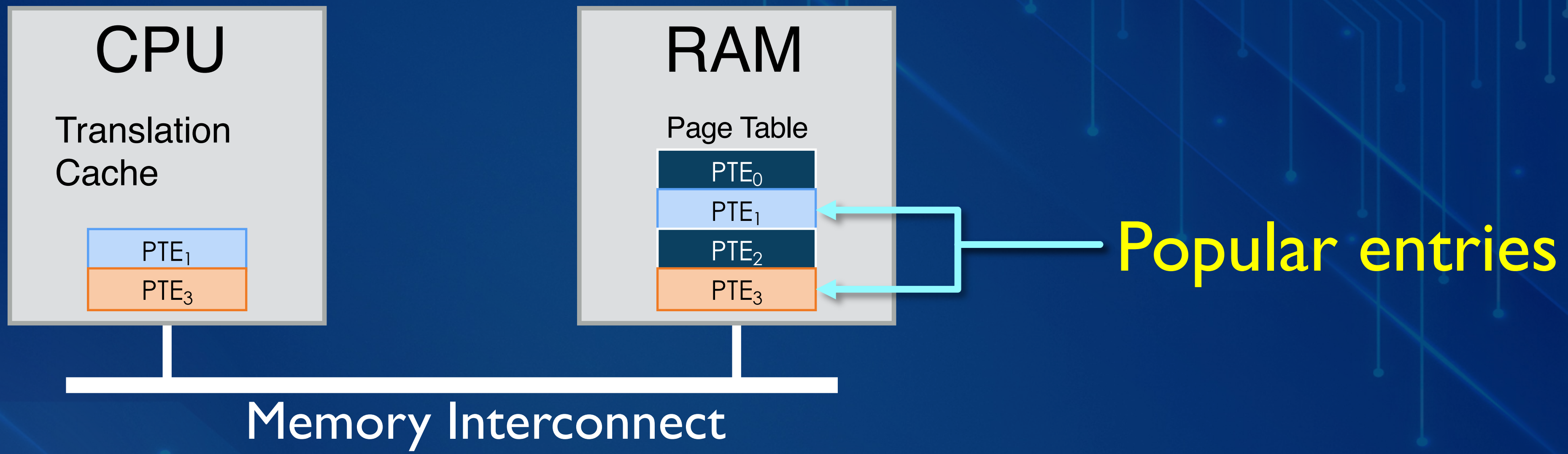
load 0x7008

load 0x100C

load 0x700C

Observation:
Spatial locality: repeated access to same PTE because program repeatedly accesses same virtual page

Caching in the TLB



TLB: Translation Lookaside Buffer

- Caches PTEs: the VPN and PPN, the valid bit, PID, etc.
- On access, if PTE in TLB, skip page table. Else, look at page table.
- If page table entry present, then cache in TLB.

Aside: Cache Associativity

- Caches are divided into blocks.
- “Fully associative” = “cache incoming data into any unused block of the cache”.
- A set is a group of blocks.
- TLBs used to be fully associative, but are now “set associative”.

TLB Performance

- Sequential array accesses almost always “hit” in TLB
  - Very fast!

- What access pattern will be slow?
  - Highly random, with no repeat accesses

Context Switches

- What happens if a process uses cached TLB entries from another process?
- Solutions?

1. Flush TLB on each switch
    - Costly; lose all recently cached translations
2. Track which entries are for which process
    - Address Space Identifier (called PCIDs on Intel)
    - Tag each TLB entry with an 8-bit ASID
        - ASIDs are smaller than PIDs!

TLB Performance

- Context switches are expensive
- Even with ASID, other processes “pollute” TLB
  - Discard process A’s TLB entries for process B’s entries
- Architectures can have multiple TLBs
  - 1 TLB for data, 1 TLB for instructions
  - 1 TLB for regular pages, 1 TLB for “super pages”

HW and OS Roles

Who handles TLB miss? ***Hardware or OS?*** Both have been used

If Hardware: CPU must know where PTs are

- cr3 register on x86
- Page table structure fixed and agreed upon between HW and OS
- HW "walks" the page table and fills TLB

If OS: CPU traps into OS upon TLB miss

- "Software-managed TLB"
- OS interprets page tables as it chooses
- Modifying TLB entries is privileged

Need same protection bits in TLB as page table
(read/write/execute and kernel/user mode access

Summary

- Pages are great, but accessing page tables is slow
- Cache recent page translations → TLB
  - Hardware performs TLB lookup on every memory access
- TLB performance depends strongly on workload
  - Sequential workloads perform well
  - Workloads with temporal locality can perform well
- In different systems, hardware or OS handles TLB misses
- TLBs increase cost of context switches
  - Add ASID to every TLB entry

~ *Fin* ~