CS111, Lecture 23

Paging

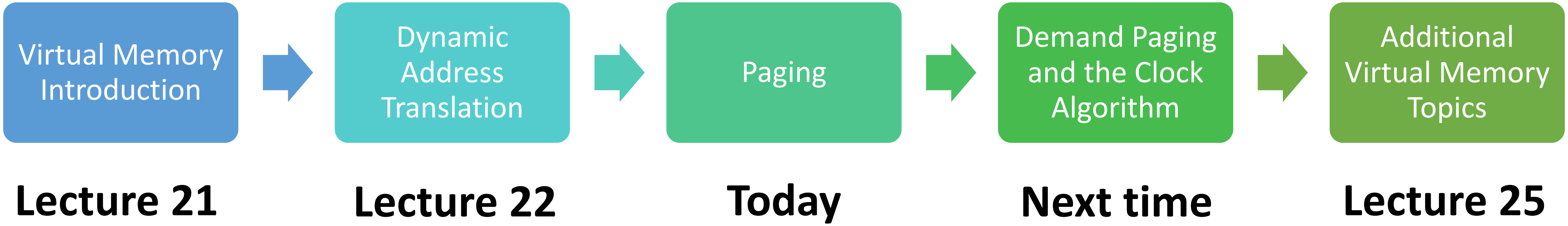
😷 masks recommended

Announcements

- Mid-quarter grade updates posted prior to Thanksgiving Break
- assign5 due Wed. 11/30
- Midterm regrade requests due Thurs. 12/1 5PM
- assign6 released Wed 11/30, due Fri. 12/9 (no late days permitted)

<u>Topic 4: Virtual Memory</u> - How can one set of memory be shared among several processes? How can the operating system manage access to a limited amount of system memory?

CS111 Topic 4: Virtual Memory



assign6: implement *demand paging* system to translate addresses and load/store memory contents for programs as needed.

Learning Goals

- Reason about the tradeoffs in different ways to implement dynamic address translation
- Understand the paging mechanism to map virtual addresses to physical addresses via fixed-size *pages* of memory.
- Learn about page maps and how they help translate virtual addresses to physical addresses

Plan For Today

- **Recap:** dynamic address translation so far
- Approach #3: Paging
- Page Maps

Plan For Today

- **Recap: dynamic address translation so far**
- Approach #3: Paging
- Page Maps

Virtual memory is a mechanism for multiple processes to simultaneously use system memory.

Dynamic Address Translation

Dynamic address translation: translate addresses dynamically during every memory reference.

- The OS intercepts every memory reference and handles it.
- The OS can stop processes from accessing prohibited addresses (e.g. OS memory or another process's memory)
- Behind the scenes, the OS can choose how it maps each process's virtual addresses to real ("physical") addresses
- As a result, a process's virtual address space may look very different from how the memory is really laid out in the physical address space.
- Every process can think it starts at address 0 and is the only process in memory
- Hardware support – MMU – to do address translation quickly

Dynamic Address Translation

Key question: how do the MMU / OS translate from virtual addresses to physical ones? Three designs we'll consider:

1. **Base and bound**
2. **Multiple Segments**
3. **(Today) Paging**

Approach #1: Base and Bound

Key Idea: every process's virtual address space is mapped to a contiguous region of physical memory, tracked via a **base** register and **bound** register.

- "base" is physical address starting point – corresponds to virtual address 0
- "bound" is one greater than highest allowable **virtual** memory address
- Each process has own base/bound. Stored in PCB and loaded into two registers when running.
- Base/bound can be updated: e.g. new physical memory location, larger bound.

On each memory reference:

- Compare virtual address to bound, trap if >= (invalid memory reference)
- Otherwise, add base to virtual address to produce physical address

Approach #1: Base and Bound

What are some benefits of this approach?

- Inexpensive translation – just doing addition
- Doesn’t require much additional space – just per-process base + bound
- The separation between virtual and physical addresses means we can move the physical memory location and simply update the base, or we could even *swap* memory to disk and copy it back later when it’s actually needed.

What are some drawbacks of this approach?

- One contiguous region per program
- Fragmentation
- Growing can only happen upwards with the bound
- Doesn’t support read-only regions of memory within a process

Approach #2: Multiple Segments

Key Idea: Each process is split among several variable-size areas of memory, called *segments*, each mapped individually with their own **base** and **bound**.

- Process's *segment map* stores info for each segment: base+bound plus a *protection* bit that indicates whether it's read/write or read-only.
- Flexibility: each segment can have its own permissions, grow/shrink independently, be swapped to disk independently, be moved independently, and even be shared between processes (e.g. shared code).

Approach #2: Multiple Segments

On each memory reference:

- Look up info for the segment that address is in (**how?**)
- Compare virtual address to that segment's bound, trap if >= (invalid memory reference)
- Add segment's base to virtual address to produce physical address

Problem: how do we know which segment a virtual address is in?

Approach #2: Multiple Segments

Problem: how do we know which segment a virtual address is in?

One Idea: make virtual addresses such that the top bits of the address specify its segment, and the low bits of the address specify the offset in that segment.

Segment #	*Offset*
0x122	**0x456**

Virtual Address

Example: PDP-10 computer had design with 2 segments, and the most-significant bit in addresses encoded which one was being referenced.

Approach #2: Multiple Segments

What are some benefits of this approach?

- Flexibility – can manage each segment independently
- Can share segments between processes
- Can move segments to compact memory and eliminate fragmentation

What are some drawbacks of this approach?

- Variable-length segments result in memory fragmentation – can move, but creates friction
- Typically small number of segments
- Encoding segment + offset rigidly divides virtual addresses (how many bits for segment vs. how many for offset?)

Idea: what if we broke up the virtual address space not into variable-length segments, but into fixed-size chunks?

Plan For Today

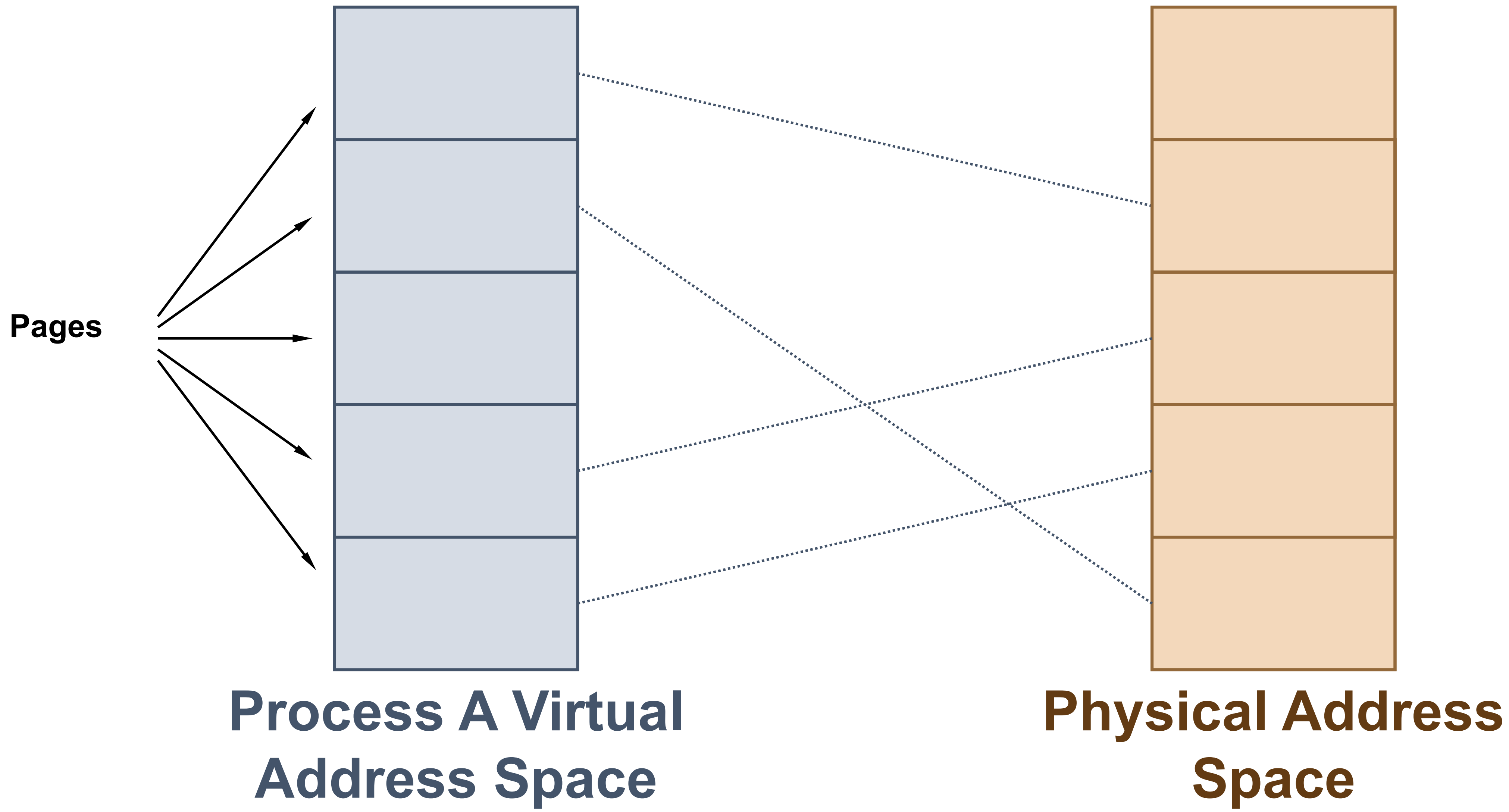
- **Recap:** dynamic address translation so far
- **Approach #3: Paging**
- Page Maps

Approach #3: Paging

Key Idea: Each process's virtual (and physical) memory is divided into fixed-size chunks called *pages*. (Common size is 4KB pages).

- A "page" of virtual memory maps to a "page" of physical memory. No partial pages
- The **page number** is a numerical ID for a page. We have virtual page numbers and physical page numbers.
- A virtual address is comprised of the virtual page # and offset in that page.
- A physical address is comprised of the physical page # and offset in that page.
- Each process has a *page map ("page table")* with an entry for each virtual page, mapping it to a physical page number and other info such as a protection bit (read-only or read-write).

Paging


Pages
Process A Virtual Address Space
Physical Address Space

Plan For Today

- **Recap:** dynamic address translation so far
- Approach #3: Paging
- **Page Maps**

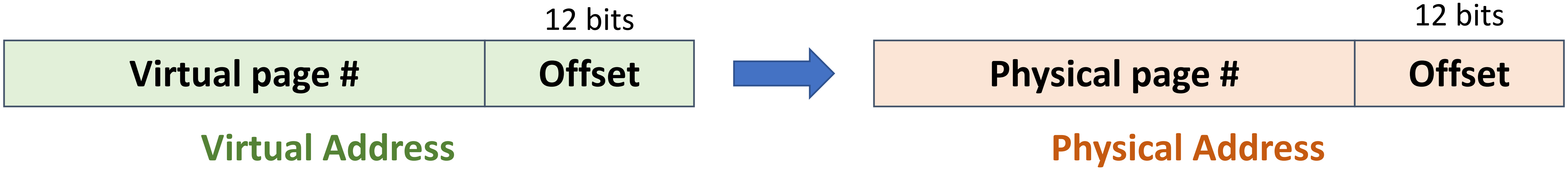
Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page # = index

Page Map

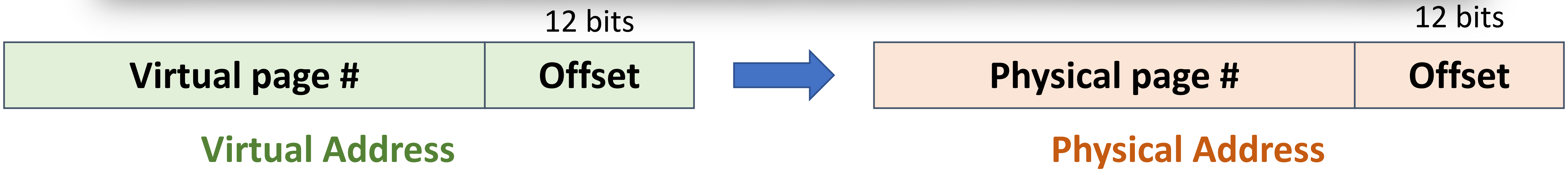
Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1

For 4KB pages (4096 bytes), the offset can be 0-4095. Thus, we can store the offset in 12 bits (the amount needed to represent any number 0-4095). 12 bits = 3 hexadecimal digits.



Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page #	*Offset*
0x3	**0x400**

Virtual Address

0x3400

→

Physical page #	*Offset*
???	**???**

Physical Address

???

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page #	*Offset*
0x3	**0x400**

Virtual Address

0x3400

Physical page #	*Offset*
???	**???**

Physical Address

???

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page #	*Offset*
0x3	**0x400**

Virtual Address

0x3400

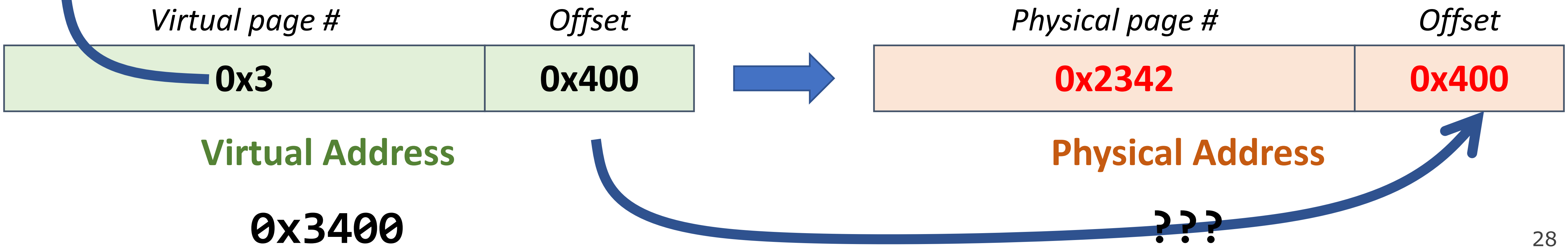
Physical page #	*Offset*
0x2342	**???**

Physical Address

???

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page #	*Offset*
0x3	**0x400**

Virtual Address

`0x3400`

Physical page #	*Offset*
0x2342	**0x400**

Physical Address

`0x2342400`

Practice: What is the physical address?

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Respond on PollEv: pollev.com/cs111 or text CS111 to 22333 once to join.

Virtual page #	*Offset*
???	???

Virtual Address

0x1456

→

Physical page #	*Offset*
???	???

Physical Address

???

What physical address corresponds with virtual address 0x1456 in this example?

Practice: What is the physical address?

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page #	*Offset*
0x1	**0x456**

Virtual Address

0x1456

→

Physical page #	*Offset*
0x13241	**0x456**

Physical Address

0x13241456

Practice: What is the physical address?

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

unused (16 bits)	Virtual page # (36 bits)	Offset (12 bits)

x86-64 64-bit Virtual Address

Physical page # (40 bits)	Offset (12 bits)

x86-64 52-bit Physical Address

x86-64 with 4KB pages has 36-bit virtual page numbers and 40-bit physical page numbers.

Paging

On each memory reference:

- Look up info for that virtual page in the page map
- If it's a valid virtual page number, get the physical page number it maps to, and combine it with the specified offset to produce the physical address.

Problem: what about invalid page numbers? I.e. how do we know/represent which pages are valid or invalid?

Solution: have entries in the page map for *all* pages, including invalid ones. Add an additional field marking whether it's valid ("present").

Page Map

Index	Physical page #	Writeable?	Present?
…	…	…	…
3	0x2342	1	1
2	XXX	X	0
1	0x13241	0	1
0	XXX	X	0

Page Map

Index	Physical page #	Writeable?	Present?
…	…	…	…
3	0x2342	1	1
2	XXX	X	0
1	0x13241	0	1
0	XXX	X	0

If there is a memory access in virtual pages 0 or 2 here, it would trap due to an invalid memory reference.

Paging

How do we provide memory to a process?

- Keep a global free list of physical pages – grab the first one when we need one
- Update process page table for a virtual page to map to this physical page, and mark present / set permission bit

In this way, we can represent a process's segments (e.g. code, data) as a collection of 1 or more pages, starting on any page boundary.

Approach #3: Paging

Key Idea: Each process's virtual (and physical) memory is divided into fixed-size chunks called *pages*. (Common size is 4KB pages).

- A "page" of virtual memory maps to a "page" of physical memory. No partial pages
- Each process has a *page map ("page table")* with an entry for every virtual page, mapping it to a physical page number and other info such as a protection bit (read-only or read-write) and whether it is present.
- The page map is stored in contiguous memory

Problem: how big is a single process's page map? You said an entry for *every* page?

Recap

- Recap: dynamic address translation so far
- Approach #3: Paging
- Page Maps

Next time: more about paging, and demand paging

Lecture 23 takeaway: Paging is a design where we chop physical and virtual memory into fixed-size pages. We map virtual pages to physical ones and store these mappings in a page map for each process.