

Operating Systems

(Sistemas Operativos)

Memory Virtualization Abstractions and Mechanisms

University of Minho
2024-2025



What will we learn?

Memory Management

- ◎ Physical memory can be seen as a simple array of bytes. Programs (running) need to store their instructions and data structures in memory
 - How does the OS abstract and manage memory accesses made by programs?
 - How is it possible to have multiple programs safely sharing the same physical memory?
 - How does the OS provide the illusion of having more memory, than the physically available one, for running programs?
- ◎ Let's study more **mechanisms** and **structures**!

Memory Management

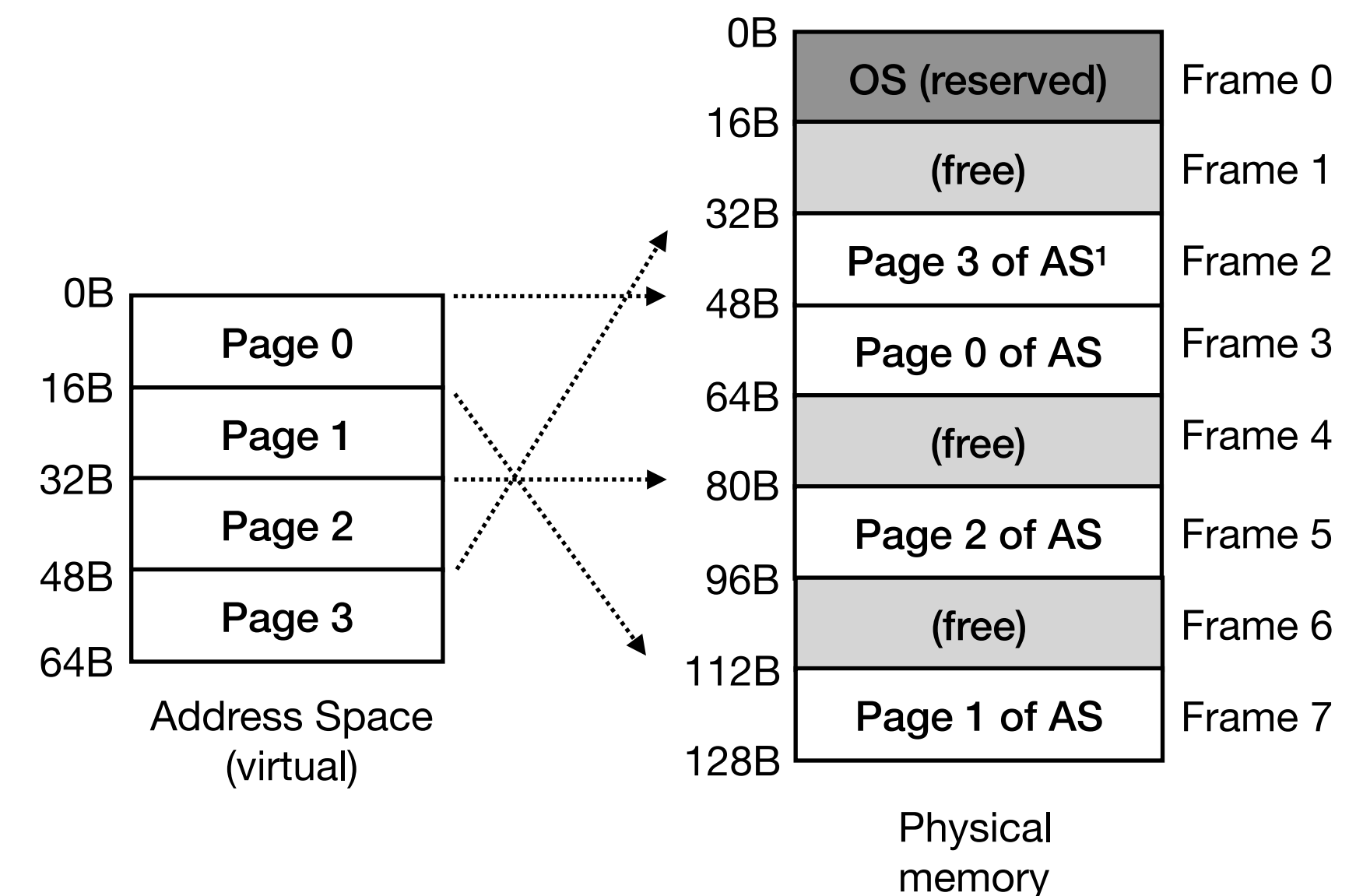
Challenge with segmentation

- ◎ With **segmentation**, the OS chops physical memory into variable-sized chunks
 - Chunks with varied sizes can lead to **external fragmentation**, making memory allocation challenging over time
- ◎ How can one avoid this challenge of using segmentation?
 - What **new techniques** are required?
 - How can one make such techniques **efficient performance-** and **space-wise**?

Paging

Pages and frames

- The virtual address space is divided into fixed-size chunks called virtual **pages**
- Physical memory is divided into fixed-size chunks, named page **frames**
 - Each physical page **frame** contains a single virtual **page** (i.e., to avoid fragmentation)
- Advantages
 - **Simplicity** of free-space management
 - The free list contains free frames, the OS grabs the first one(s) when pages must be allocated
 - **Flexibility** in supporting the address space abstraction
 - Independent from how processes use the address space (e.g., we don't need to make assumptions about how the heap and stack grow and how these are used)



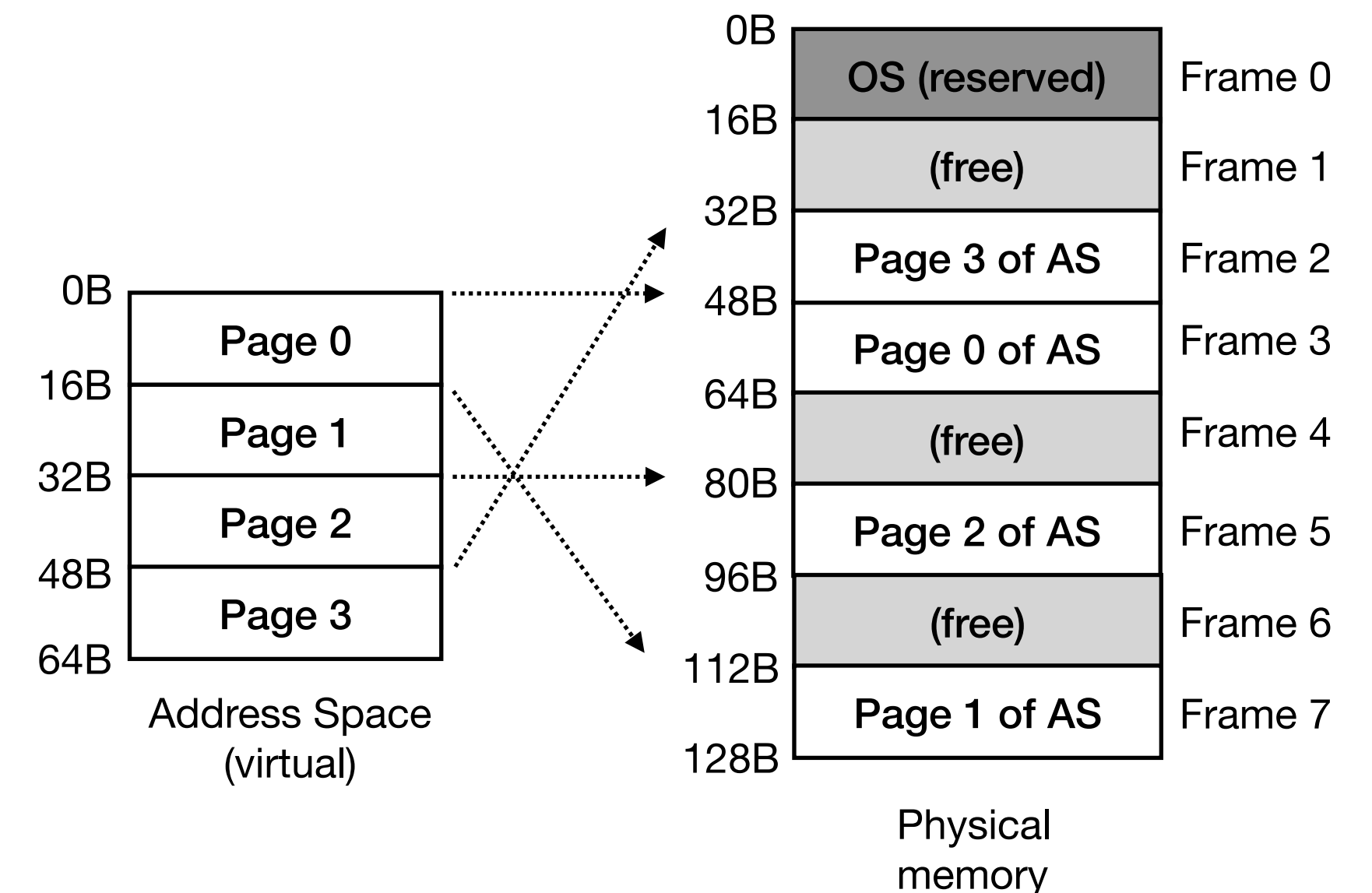
Address Translation

Page table

- **Address translation** must still be done!
 - **Virtual pages** are part of the virtual address space, their **actual content is somewhere mapped into physical memory** (or even disk¹) (i.e., this help the OS building the private process memory abstraction)
- Each **virtual page address** must be translated into the corresponding **physical page frame address**
- A **page table** is used to record where each page is placed in physical memory
 - **Important:** The page table is a **per-process data structure**² (i.e., each process has its own page table)

VPN 0	→	PFN 3
VPN 1	→	PFN 7
VPN 2	→	PFN 5
VPN 3	→	PFN 2

Page Table of Process
Translation from **Virtual Page Number (VPN)** to
Physical Frame Numbers (PFN)



¹We will talk about it later on

²This will be true for all the page table designs we will study, except only for the **inverted page table structure**

Address Translation

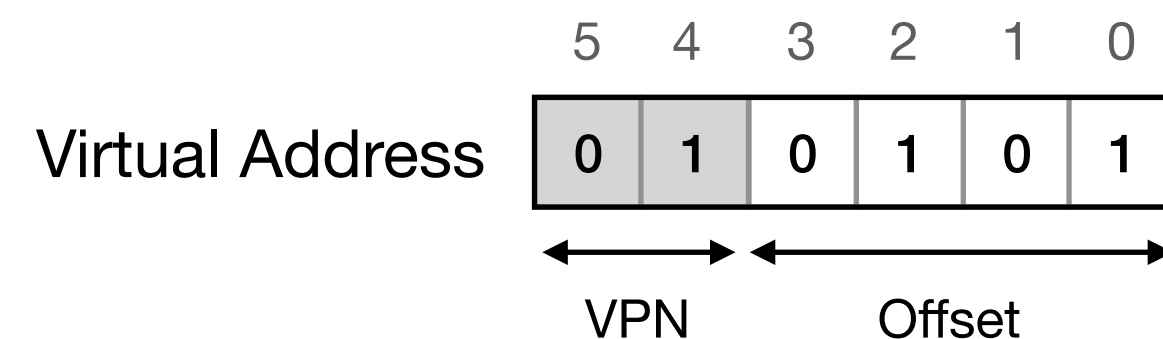
Example

● **Example:** address space has 64 bytes and page size is 16 bytes (unrealistically small...)

- ▶ We need **6 bits** ($2^6 = 64$ bytes) for our **virtual address**

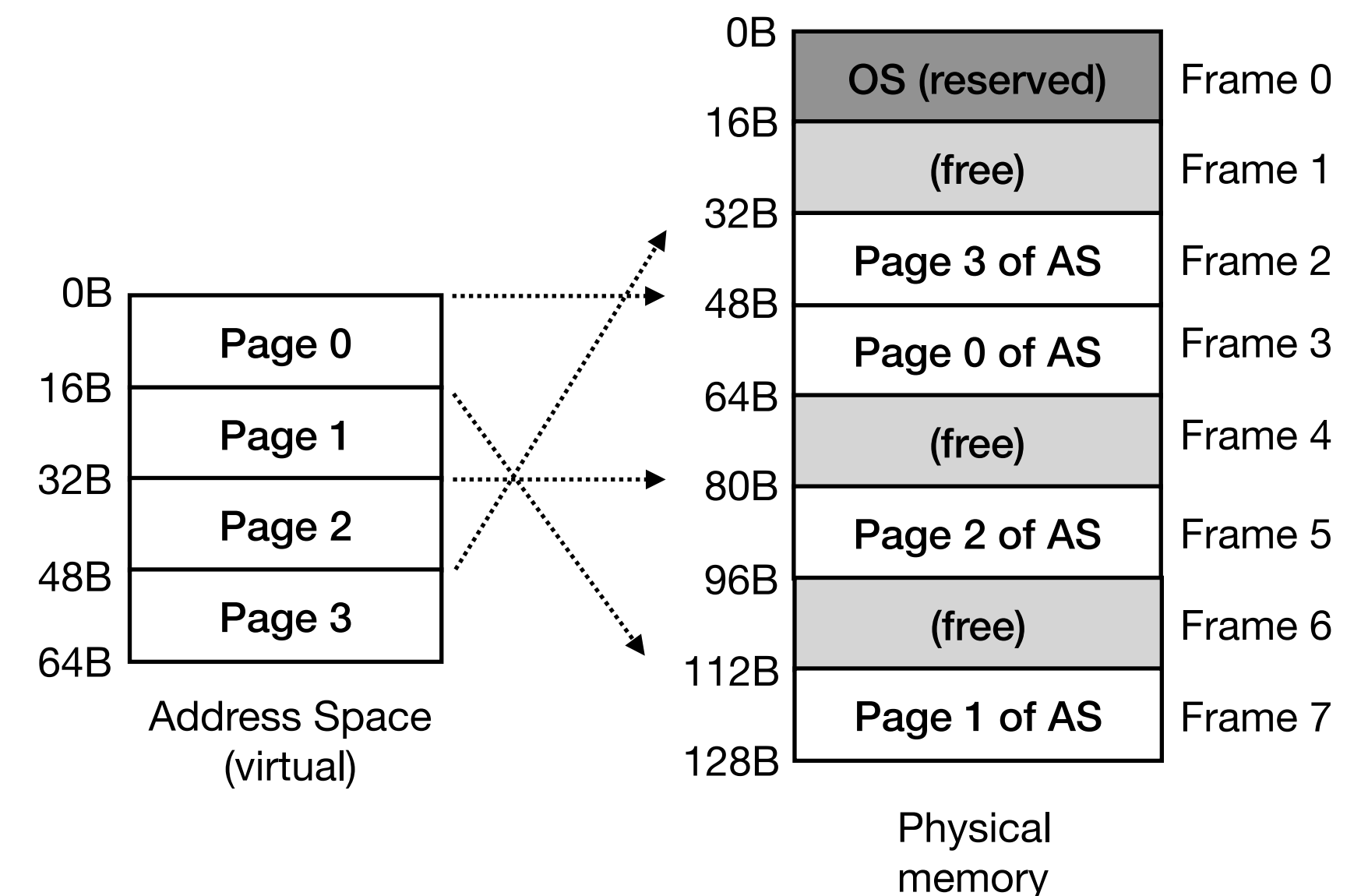
● A request to **virtual address 21 bytes** is done (*010101* in binary). The virtual address is divided into

- ▶ **Virtual Page Number (VPN):** identifies the page number (top bits) (top 2 bits in this example, since we need to refer to 4 distinct pages)
- ▶ **Offset:** identifies the specific bytes inside the page to access (4 bits are required to represent the offset of a 16 (2^4) byte page)
- ▶ In our example (below), the virtual address 21 is on the 5th (*0101* in binary) byte of VPN 1 (*01* in binary)



VPN 0	→	PFN 3
VPN 1	→	PFN 7
VPN 2	→	PFN 5
VPN 3	→	PFN 2

Page Table of Process
Translation from **Virtual Page Number (VPN)** to
Physical Frame Numbers (PFN)



Address Translation

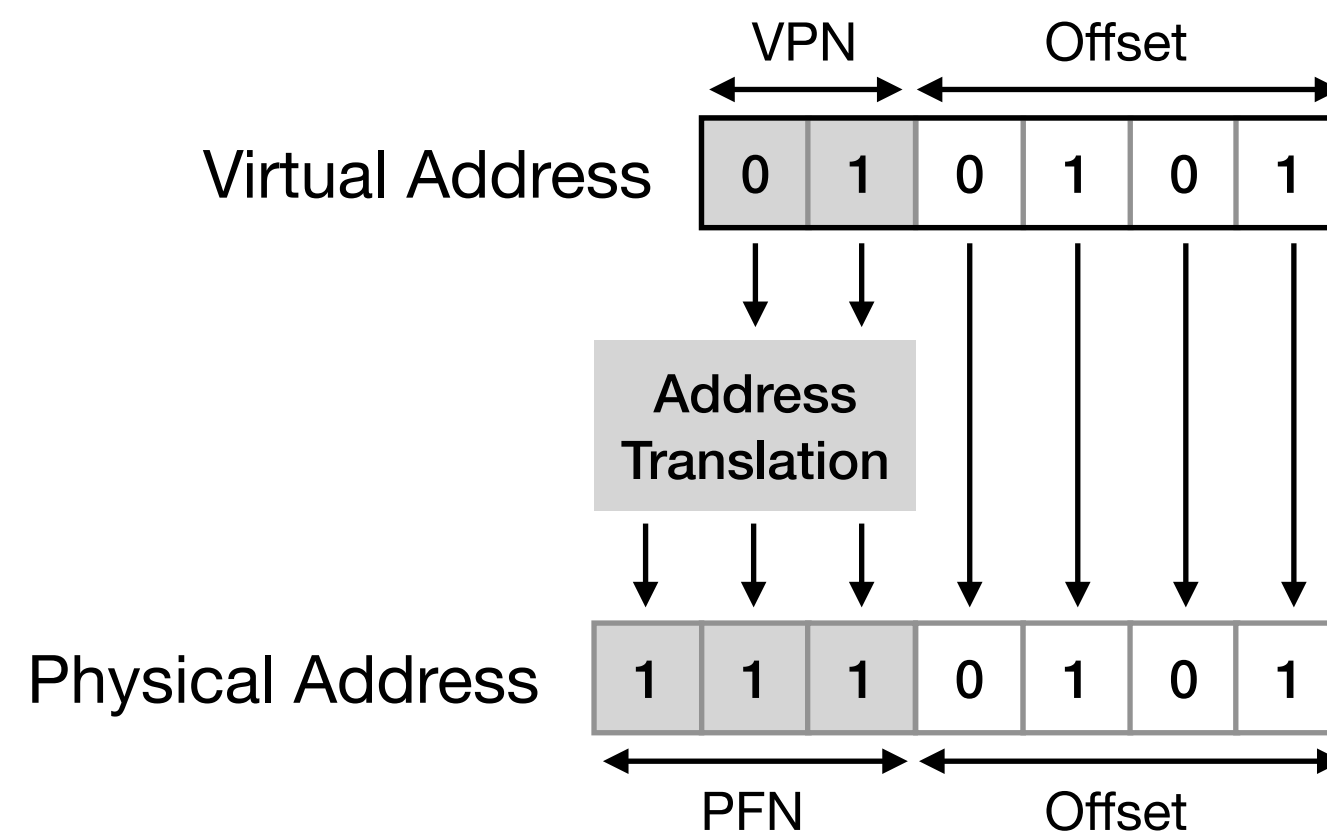
Example

● But, how is the translation into the physical memory address done?

- ▶ Let's assume our physical memory has 128 bytes

● The **page table** is used to map the **VPN** into a **Physical Frame Number (PFN)**¹

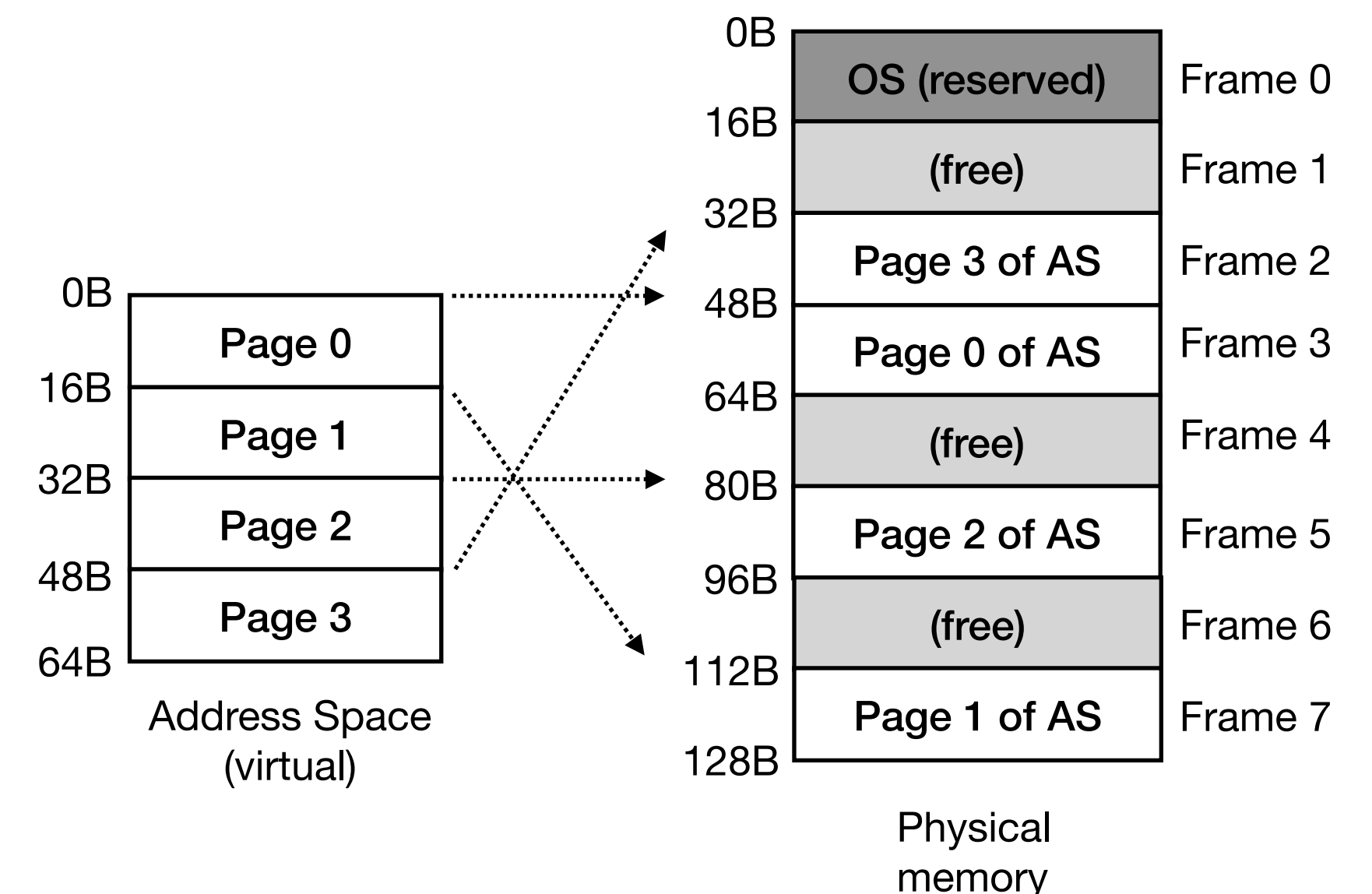
- ▶ VPN 1 is mapped to PFN 7 (111 in binary) in our example
 - 1110000 (binary) corresponds to 112 in decimal (Frame 7 address!)
- ▶ The physical address (adding the offset) is 117 (1110101 in binary)



¹PFN is also known as **Physical Page Number (PPN)** in the literature

VPN 0	→	PFN 3
VPN 1	→	PFN 7
VPN 2	→	PFN 5
VPN 3	→	PFN 2

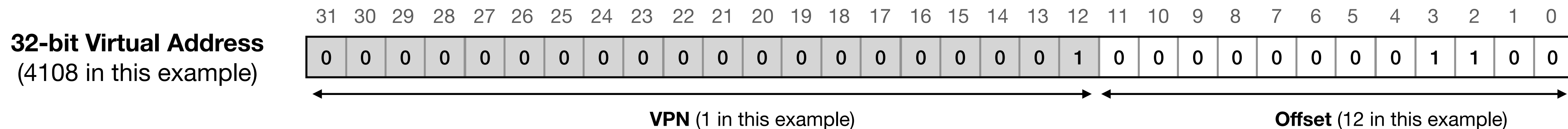
Page Table of Process
Translation from **Virtual Page Number (VPN)** to
Physical Frame Numbers (PFN)



Page Table

Where are they stored?

- For a **32-bit virtual address** and **4KB pages**, we need a **12-bit offset**¹
- With **20 bits per VPN**, the OS manages **2^{20} translations per process** (around 1 million entries)
 - Assuming each **Page Table Entry (PTE)** needs 4 bytes (i.e., for the PFN and other useful information)
 - 1 million entries, each with 4 bytes, leads to a page table with around **4 MB**
 - If **100 processes** are loaded into memory, **400MB of memory** are being wasted with page tables
- If we do the calculations for a **64-bit virtual address** (like we have in most of our computers)...
 - **12 bits for the offset** (assuming 4KB pages) and **52 bits for the VPN**, meaning 2^{52} PTEs!
- Given the sheer size of page tables, these must be stored in physical memory (not at the MMU)



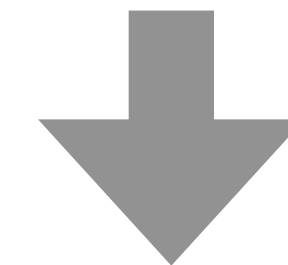
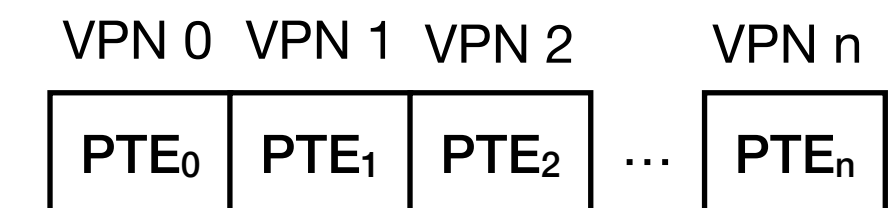
¹To represent bits for a page of 4KB (from 0 to 4095) we need 12 bits ($2^{12} = 4096$)

Page Table

What information do these contain?

- The page table is just a data structure that maps **VPNs** to **PFNs**
 - The simplest design is a **linear page table**, e.g., an array indexed by the VPN
- Each element of the array is a **Page Table Entry (PTE)** that contains
 - The corresponding **PFN**
 - A **valid bit** indicating whether the translation is valid
 - Unused space is marked as **invalid** (bit set to 0).
If an invalid virtual page is accessed, an exception (trap) is generated
 - This bit allows supporting **sparse address spaces**,
i.e., no need to allocate physical **frames** for invalid pages!
 - **Protection bits** indicating if the page can be read, written to, or executed
 - A **present bit** indicating if the page's content resides on memory or disk¹
 - A **dirty bit** indicating if the page was modified after being brought to memory¹
 - A **reference bit** (aka **use bit**) tracks whether a page has been accessed¹

Linear page table
implementation



PFN	valid	prot	present	dirty	reference
10	1	R-X	1	0	1
-	0	-	-	-	-
-	0	-	-	-	-
...					
4	1	RW-	1	1	0

Linear page table
logical view

(row 0 corresponds to **VPN 0**,
row 1 to **VPN 1**, and so on)

¹We will talk about these later on

Page Table

Address translation is slow

- ◎ For **each memory access** done by a process, the **hardware must**
 1. Know where the process's page table is loaded in physical memory, e.g., through another CPU register (**page-table base register**)
 2. Fetch the corresponding PTE from memory to the processor
 3. Check if the request is valid, calculate the PFN and concatenate it with the offset
 4. Access the memory location
- ◎ Every address translation now requires **two accesses to main memory**
 - To the page table and to the memory requested by the process
 - Slower than segmentation (base and bounds for each segment were kept at CPU registers). Accessing a CPU register is faster than accessing physical memory!

Translation Look-aside Buffer (TLB)

Address Translation is slow

- ◎ How can one avoid the extra memory access required by paging?
 - The OS needs help from the hardware
- ◎ The **Translation Look-aside Buffer** (TLB) is a component of the **MMU**
 - Basically, it is a **hardware cache** of **popular** virtual-to-physical **address translations**
 - Accessing the TLB cache is significantly faster than going to main memory!

VPN	PFN
10	101
20	170
2	130

Simple example
of TLB entries

Translation Look-aside Buffer (TLB)

Basic Algorithm

- When a virtual address translation is requested, the hardware must
 1. Extract the VPN from the virtual address
 2. Check for the VPN at the TLB
 - **TLB Hit:** if the TLB holds the translation
 3. The PFN is extracted, concatenated into the offset to form the physical address, and it's done!
(No need to consult the page table, meaning that an extra memory access is spared)
 - **TLB Miss:** if the TLB does not hold the translation
 3. The page table (in memory) must be consulted to find the translation
 4. The TLB must be updated with that translation
 5. The hardware retries the memory access instruction
- Like all caches, the TLB is built on the premise that most accesses will result in **hits**
 - TLB misses result in the extra memory access we want to avoid!

Translation Look-aside Buffer (TLB)

Example

- Assuming an array with 10 elements (4 byte integers), starting at virtual address 36
 - Pages of 16 bytes and we are ignoring other memory accesses (i.e., to vars `i` and `sum`)
 - Also, we are starting this example with an empty TLB

Simple example
of TLB entries

VPN	PFN

Code

```
int sum = 0,
for (i=0; i<10; i++){
    sum += a[i];
}
```

	0	4	8	12	16
VPN 0					
VPN 1					
VPN 2		a[0]	a[1]	a[2]	
VPN 3	a[3]	a[4]	a[5]	a[6]	
VPN 4	a[7]	a[8]	a[9]		
VPN 5					
VPN				

Linear page table

Translation Look-aside Buffer (TLB)

Example

- Assuming an array with 10 elements (4 byte integers), starting at virtual address 36
 - Pages of 16 bytes and we are ignoring other memory accesses (i.e., to vars `i` and `sum`)
 - Also, we are starting this example with an empty TLB
- When `a[0]` is accessed, a **TLB miss** occurs, the translation is fetched to the TLB

Simple example
of TLB entries

VPN	PFN

Miss

Code

```
int sum = 0,
for (i=0; i<10; i++){
    sum += a[i];
}
```

	0	4	8	12	16
VPN 0					
VPN 1					
VPN 2		a[0]	a[1]	a[2]	
VPN 3	a[3]	a[4]	a[5]	a[6]	
VPN 4	a[7]	a[8]	a[9]		
VPN 5					
VPN				

Linear page table

Translation Look-aside Buffer (TLB)

Example

- Assuming an array with 10 elements (4 byte integers), starting at virtual address 36
 - Pages of 16 bytes and we are ignoring other memory accesses (i.e., to vars i and sum)
 - Also, we are starting this example with an empty TLB
- When a[0] is accessed, a **TLB miss** occurs, the translation is fetched to the TLB

Simple example
of TLB entries

VPN	PFN
2	100

Code

```
int sum = 0,
for (i=0; i<10; i++){
    sum += a[i];
}
```

	0	4	8	12	16
VPN 0					
VPN 1					
VPN 2		a[0]	a[1]	a[2]	
VPN 3	a[3]	a[4]	a[5]	a[6]	
VPN 4	a[7]	a[8]	a[9]		
VPN 5					
VPN				

Linear page table

Translation Look-aside Buffer (TLB)

Example

- Assuming an array with 10 elements (4 byte integers), starting at virtual address 36
 - Pages of 16 bytes and we are ignoring other memory accesses (i.e., to vars i and sum)
 - Also, we are starting this example with an empty TLB
- When a[0] is accessed, a **TLB miss** occurs, the translation is fetched to the TLB
 - When a[1] and a[2] are accessed, the translation is already at the TLB (hit!)
 - a[3] and a[7] are on different pages, so a **TLB miss** occurs when these are accessed. Accesses to the remaining elements of the array result in **hits**

Simple example
of TLB entries

VPN	PFN
2	100

Hit

Code

```
int sum = 0,  
for (i=0; i<10; i++){  
    sum += a[i];  
}
```

	0	4	8	12	16
VPN 0					
VPN 1					
VPN 2		a[0]	a[1]	a[2]	
VPN 3	a[3]	a[4]	a[5]	a[6]	
VPN 4	a[7]	a[8]	a[9]		
VPN 5					
VPN				

Linear page table

Translation Look-aside Buffer (TLB)

Example

- Assuming an array with 10 elements (4 byte integers), starting at virtual address 36
 - Pages of 16 bytes and we are ignoring other memory accesses (i.e., to vars `i` and `sum`)
 - Also, we are starting this example with an empty TLB
- When `a[0]` is accessed, a **TLB miss** occurs, the translation is fetched to the TLB
 - When `a[1]` and `a[2]` are accessed, the translation is already at the TLB (hit!)
 - `a[3]` and `a[7]` are on different pages, so a **TLB miss** occurs when these are accessed. Accesses to the remaining elements of the array result in **hits**
- The pattern is therefore: **miss**, hit, hit, **miss**, hit, hit, hit, **miss**, hit, hit
 - TLB hit rate:** $\frac{\text{hits}}{\text{accesses}} = \frac{7}{10} = 0.7 = 70\%$
 - We want **hit rates close to 100%** to **amortize** the cost of accessing memory
- If the array is accessed again, and entries are still at the TLB, then we have 10 hits!

Simple example of TLB entries

VPN	PFN
2	100
3	50
4	101

Code

```
int sum = 0,
for (i=0; i<10; i++){
    sum += a[i];
}
```

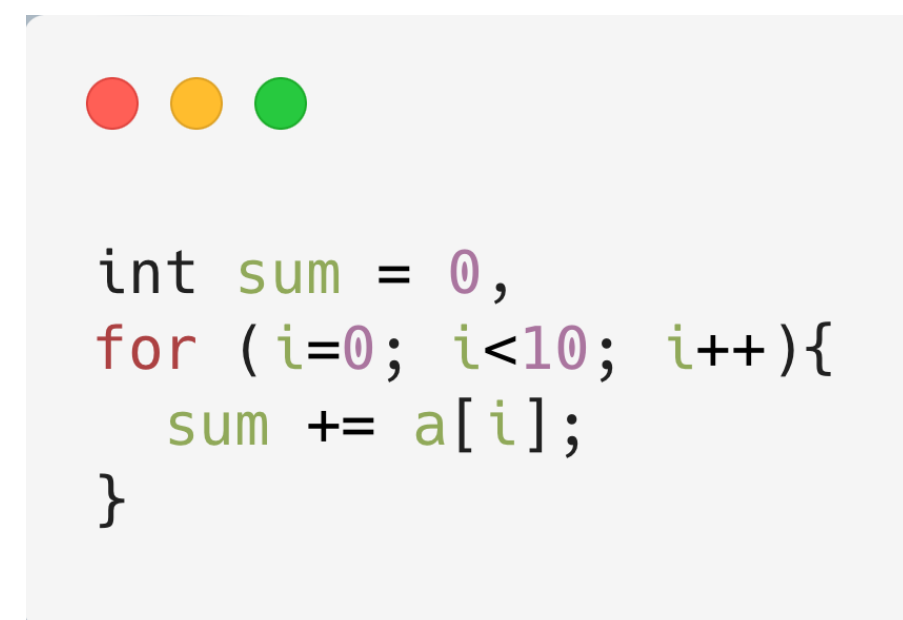
	0	4	8	12	16
VPN 0					
VPN 1					
VPN 2		a[0]	a[1]	a[2]	
VPN 3	a[3]	a[4]	a[5]	a[6]	
VPN 4	a[7]	a[8]	a[9]		
VPN 5					
VPN				

Linear page table

Translation Look-aside Buffer (TLB)

Temporal and spatial locality

- ◎ The **page size** plays an **important role** in the number of **hits** and **misses**
 - A typical page size of 4KB can hold several elements of dense arrays and achieve few misses
- ◎ The TLB is taking advantage of two properties of memory accesses in the previous example
 - **Spatial locality:** If a program accesses address x , it is probable that will then access addresses near x
e.g., a for loop iterating over an array
 - **Temporal locality:** If a program accesses a given memory item (address, data, instruction) it is likely to access it again in the near future
e.g., our variables i and sum or the code inside the loop
- ◎ Many caches (e.g., for data, instructions) take advantage of these properties



```
int sum = 0,  
for (i=0; i<10; i++){  
    sum += a[i];  
}
```


Translation Look-aside Buffer (TLB)

Content

- A typical TLB might have 32, 64 or 128 entries
 - Small size (it is common for faster memory to be smaller...)
 - The TLB is **fully-associative**, i.e., a given translation can be anywhere in the TLB and must be searched by the hardware, typically in parallel
- Each TLB entry has
 - The **VPN** and corresponding **FPN**
 - A **valid bit** indicating whether the address translation is valid
 - **Protection bits** indicating if the page can be read, written to or executed
 - A **dirty bit** indicating if the page was modified after being brought to memory
 - An **address-space identifier** (ASID), used to identify different processes

VPN	FPN	Valid	Prot	ASID
10	100	1	R-X	1
-	-	0	-	-
50	101	1	RW-	2

Updated TLB example

Translation Look-aside Buffer (TLB)

Handling TLB misses and context switching

● TLB Misses can be handled by

- ▶ **The hardware:** the hardware requires direct access to page tables in memory (e.g., through a **page table base register**)
- ▶ **Software:** On a miss the hardware raises an exception, jumps to the OS code, the OS handles the exception, adds the translation to the TLB (special hardware instruction), and then returns from trap

● What about when processes are switched (**context switching**)?

- ▶ The TLB entries of the process being switched cannot be used by the new process
 - **Option 1:** flush the TLB entries of the process being switched. But, if the OS schedules the process again in the near future, several TBL misses may occur...
 - **Option 2:** use the **ASID** field (similar to the PID field but with less bits) to indicate the process the translation refers to. The ASID value is set by the OS (in a special register)

● **Sharing pages across processes** is possible by having **different VPNs** "pointing" to the **same FPN**

Same VPN and different FPNs, the ASID field is used to distinguish the two processes

VPN	PFN	Valid	Prot	ASID
10	100	1	R-X	1
-	-	0	-	-
10	101	1	RW-	2
11	30	1	R-X	1
50	30	1	R-X	2

Updated TLB example

Different VPNs (processes), pointing (sharing) the same FPN

Translation Look-aside Buffer (TLB)

Replacement Policy

- ◎ A final concern is, when the TLB is full and a new entry must be added, what old entry should be replaced (**evicted**)?
- ◎ This requires a **replacement policy**
 - The main goals should be **maximizing the hit rate** and **being efficient**
 - There are several algorithms such as **FIFO**, **random**, **least-recently used (LRU)**.
We will study these when talking about about memory swapping

Linear Page Table

The size issue

- We still have another problem to solve, the **size of page tables**!
 - **Important:** we are concerned with the size that page tables occupy in memory, remember that these structures need to be stored there!
 - **Don't be confused** with the **physical memory (frames) allocated by programs**. We already solved such problem with the **valid bit** at the process's page table that avoids allocating frames for invalid pages (i.e., enables **sparse address spaces**)
- Remember our example from slide 8
 - For a 32-bit address with 4KB pages and a PTE of 4 bytes, each page table uses around 4MB
 - Even worse for 64-bit addresses :(
- **Linear page tables** are thus inefficient structures space-wise (specially when the page tables of multiple processes must be loaded into memory)
- Do you have any ideas on how to reduce the page table size?

Page Table Designs

Bigger pages

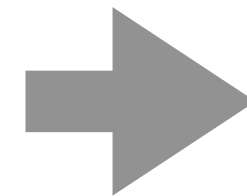
- ◎ Assuming the same 32-bit address space but with **16KB pages**
 - We have a **14-bit offset** ($2^{14} = 16\text{KB}$) and a **18-bit VPN**
 - The linear page table would have **262 144** (2^{18}) **PTEs**
 - If each PTE takes **4 bytes**, each page table occupies around **1MB**
 - A 4x reduction in size when compared to a page size of 4KB
- ◎ While big pages address our size problem, these lead to higher **internal fragmentation**
 - Many processes will allocate a small amount of memory (e.g., in the order of bytes or few KBs)
 - Since processes must allocate at least one page, there will be **wasted space inside the page**
- ◎ To avoid severe internal fragmentation, most OSs use smaller page sizes such as 4KB and 8KB

Page Table Designs

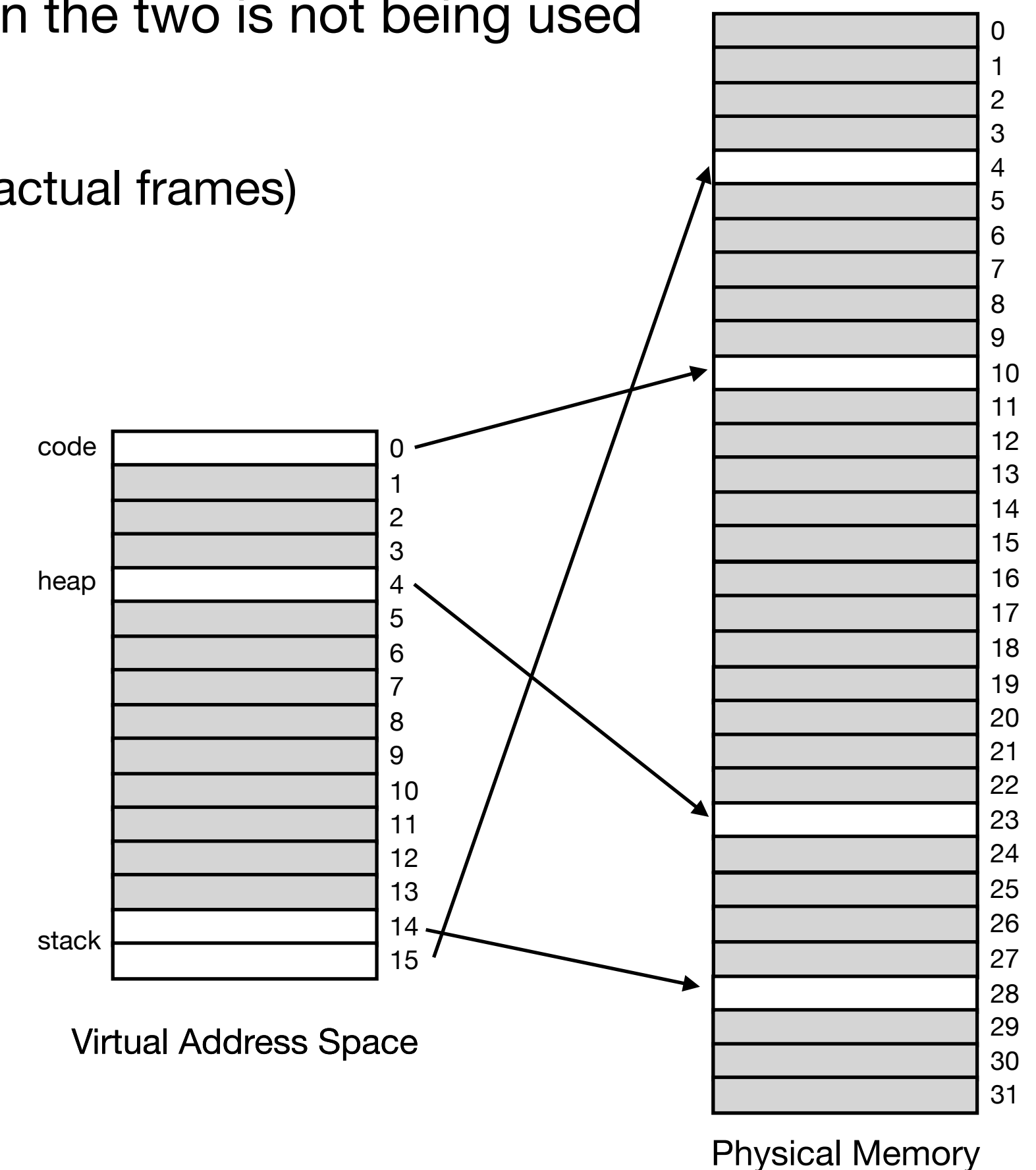
Invalid entries at linear page tables

- Many processes use a small stack and heap space, thus the space between the two is not being used
- Let's look at the 16KB address space with 1KB pages below
 - Only pages 0, 4, 14 and 15 are allocated in physical memory (i.e., mapped to actual frames)
 - Our **linear page table** has several PTEs marked as **invalid** (what a waste having these PTEs occupying memory...)

Page table for the 16KB address space
(row 0 corresponds to **VPN 0**,
row 1 to **VPN 1** and so on)



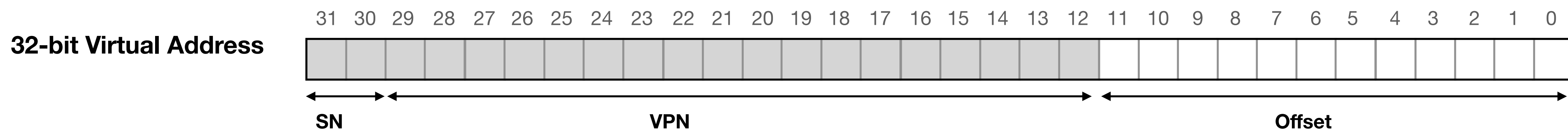
PFN	valid	prot	present	dirty
10	1	R-X	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
23	1	RW-	1	1
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
28	1	RW-	1	1
4	1	RW-	1	1



Page Table Designs

Mixing paging and segmentation

- © What about **combining paging and segmentation** into an hybrid approach?
 - Each segment (code, heap and stack) has its own page table
 - The **base and bounds registers** of each segment now point to the **physical address of the page table** (of that segment) and to the **table's limits**, respectively. Accesses for addresses over the limit lead to exceptions!
- © **Example:** a 32 bit virtual address with 4KB pages
 - **SN (segment number)** is used to find the the base and bounds pairs to use (look again at the segmentation slides if needed!)
 - **VPN** is used to find the **FPN** at the segment's page table, and the offset is added to generate the physical address



Page Table Designs

Mixing paging and segmentation

● Since we have a **page table per segment** if, for example, the heap only uses the first three pages (0, 1, and 2), its linear page table only requires the space for 3 PTEs at physical memory!

- ▶ The same can be done for the code and stack segments
- ▶ One avoids allocating PTEs for the free space between the heap and stack

	PFN	valid	prot	present	dirty
(VPN 0)	10	1	R-X	1	0
(VPN 1)	12	1	R-X	1	0
(VPN 2)	100	1	R-X	1	0

Contiguous page table with the first 3 pages being used
The memory space required is 12 bytes
(assuming each PTE uses 4 bytes)

● There are, however challenges with this hybrid approach

- ▶ If the heap, for example, is **sparsely allocated across several pages** (e.g., pages 1, 100, and 1500), the linear page table must still allocate 1500 PTEs, although only 3 are valid...
- ▶ Further, **page tables** now have **variable sizes**, and allocating memory for these leads to **external fragmentation**

	PFN	valid	prot	present	dirty
(VPN 0)	5	1	R-X	1	0
(VPN 1)	-	0	-	-	-
...	...				
(VPN 100)	10	1	RW-	1	1
(VPN 101)	-	0	-	-	-
...	...				
(VPN 1500)	1000	1	R-X	1	0

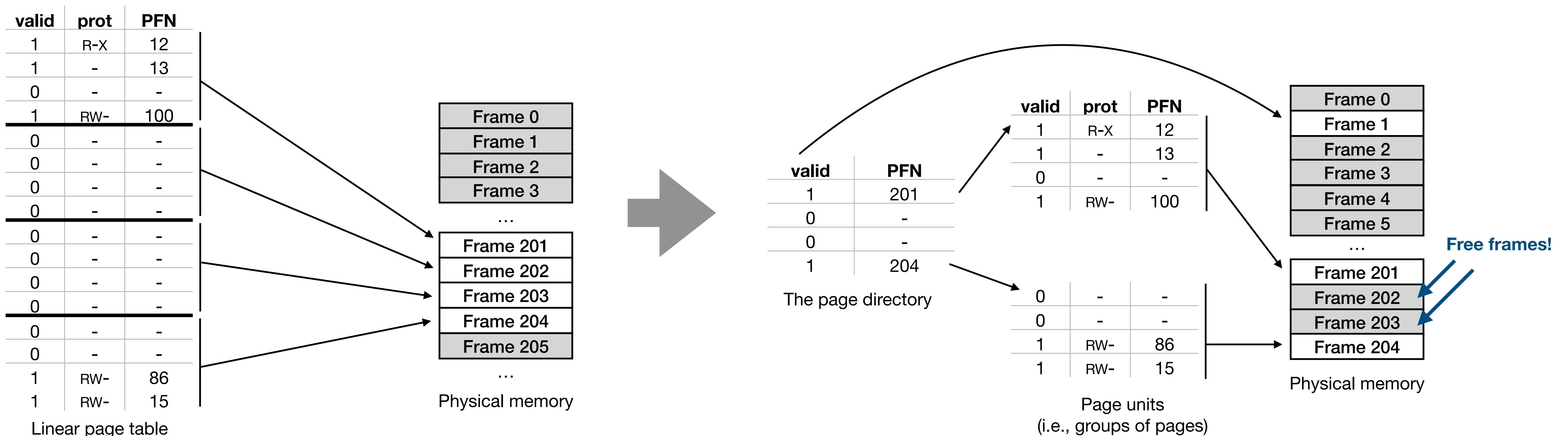
Sparse page table with 1500 PTEs
The memory space required is 6000 bytes
(assuming each PTE uses 4 bytes)

● Is there any other way to avoid having (invalid) PTEs at physical memory?

Page Table Designs

Multi-level page tables

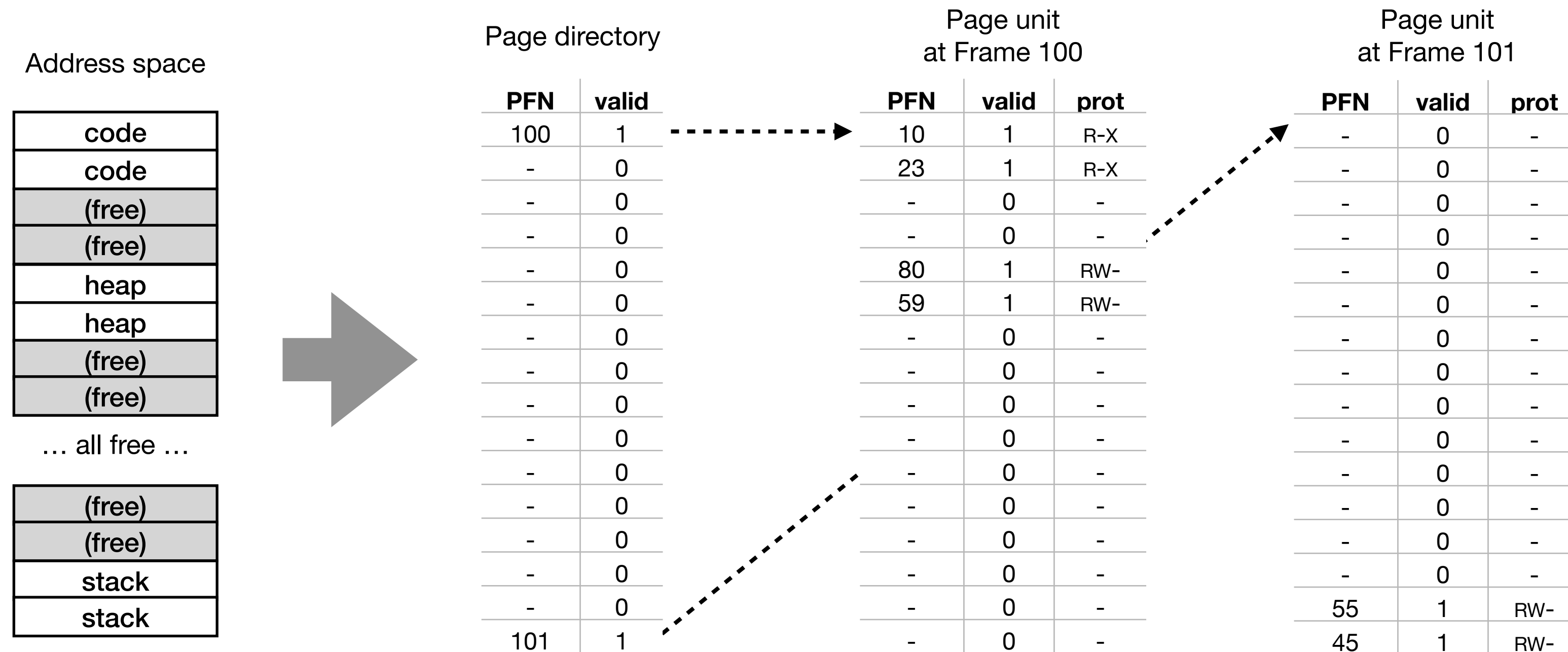
- Chop up the linear page table into **several page-sized units**
 - **Each unit is only allocated** at physical memory (i.e., at a given frame) if it contains **at least one valid PTE**
- A new structure, called **page directory**, is responsible for indexing the **page units** (i.e., it keeps the unit's PFN)
 - It contains a **Page Directory Entry (PDE)** per indexed unit. Each PDE has a **valid bit** and a **PFN**
 - **Example:** Assuming a page size of 16 bytes and PTEs with 4 bytes



Page Table Designs

Two-level page table example

- **Another example:** address space of 16KB, with 64-byte pages and PTE size of 4 bytes
 - 14-bit virtual address with 8 bits for VPN and 6 bits for the offset
 - A linear page table would require 2^8 (256) PTEs, i.e., using around 1KB of memory (256 PTEs x 4 bytes)
- Let's build a **two-level page table!**
 - We chop the linear page table into **page units, each containing 16 PTEs** (64 bytes page / 4 bytes)
 - To represent the full address space, we need up to **16 page units** (256 PTEs in total / 16 PTEs per page table unit)



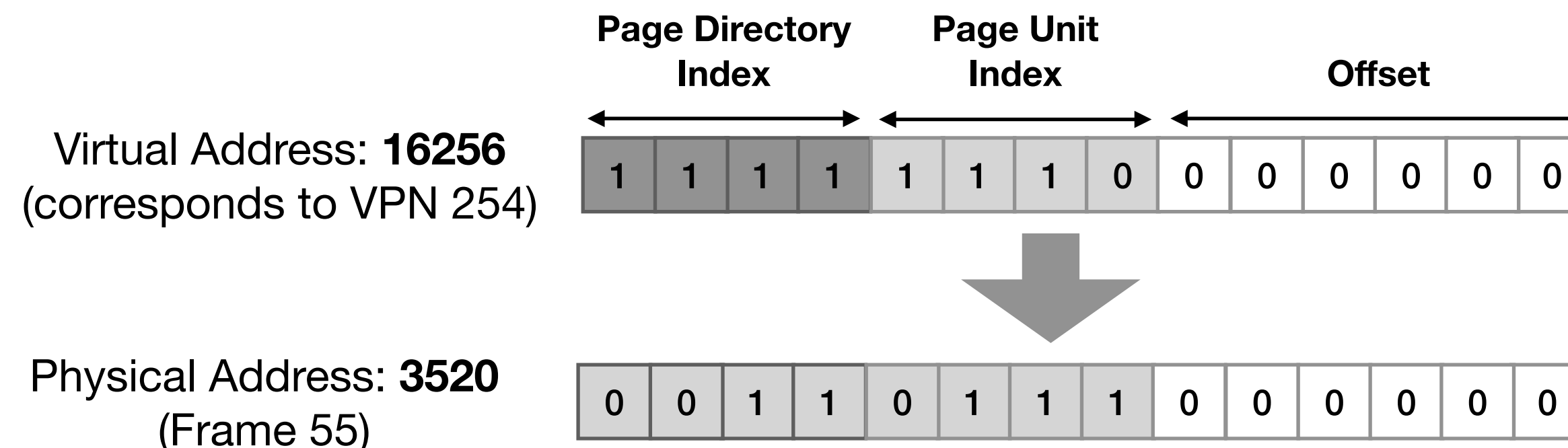
The two-level page table **reduces memory usage to 3 frames** (i.e., 1 for the directory and 2 for page units)

192 bytes instead of 1KB!

Page Table Designs

Two-level page table example

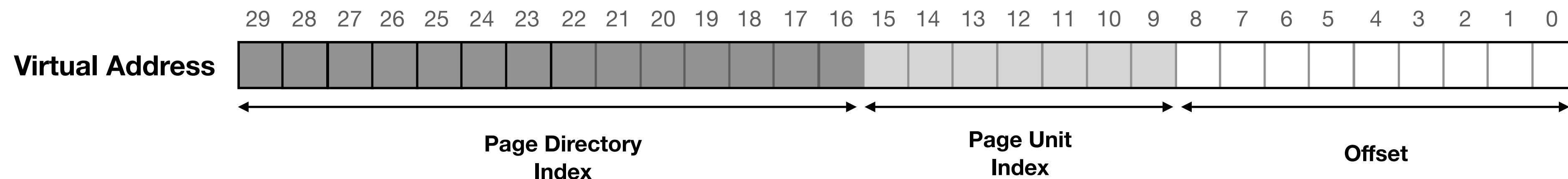
- Address translation is similar to what we saw previously. Let's use the virtual address **16256** as an example
- Access the page directory (physical address is kept at a **page directory base register**), and use the **top 4 bits** at the virtual address to **find the desired PDE** and extract the **PFN of the page unit**
 - 1111 corresponds to the last entry (15th) of the page directory, pointing to PFN 101 (memory location of the page unit!)
 - Access the page unit (at frame 101), and use the **next 4 bits** of the virtual address to **find the desired PTE** and extract the **PFN of our virtual address**
 - At the page unit (PFN 101), the 14th (1110) entry is consulted, mapping to FPN 55 (00110111, i.e., the frame corresponding to our virtual address request!)
 - **Add the offset**, which is 0 in this case, and we get the **physical address** (i.e., the 0th byte of Frame 55!)



Page Table Designs

Multi-level page tables

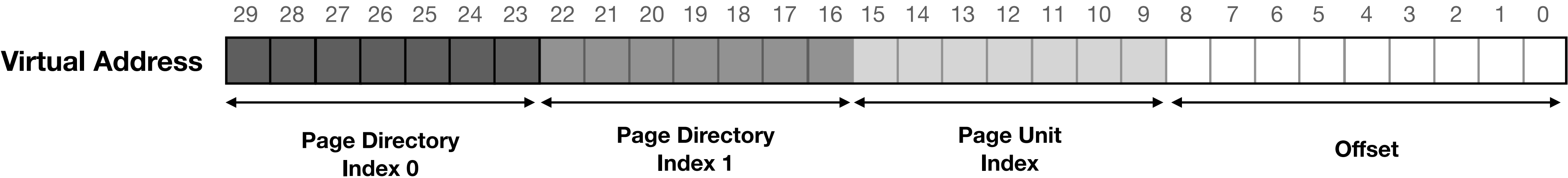
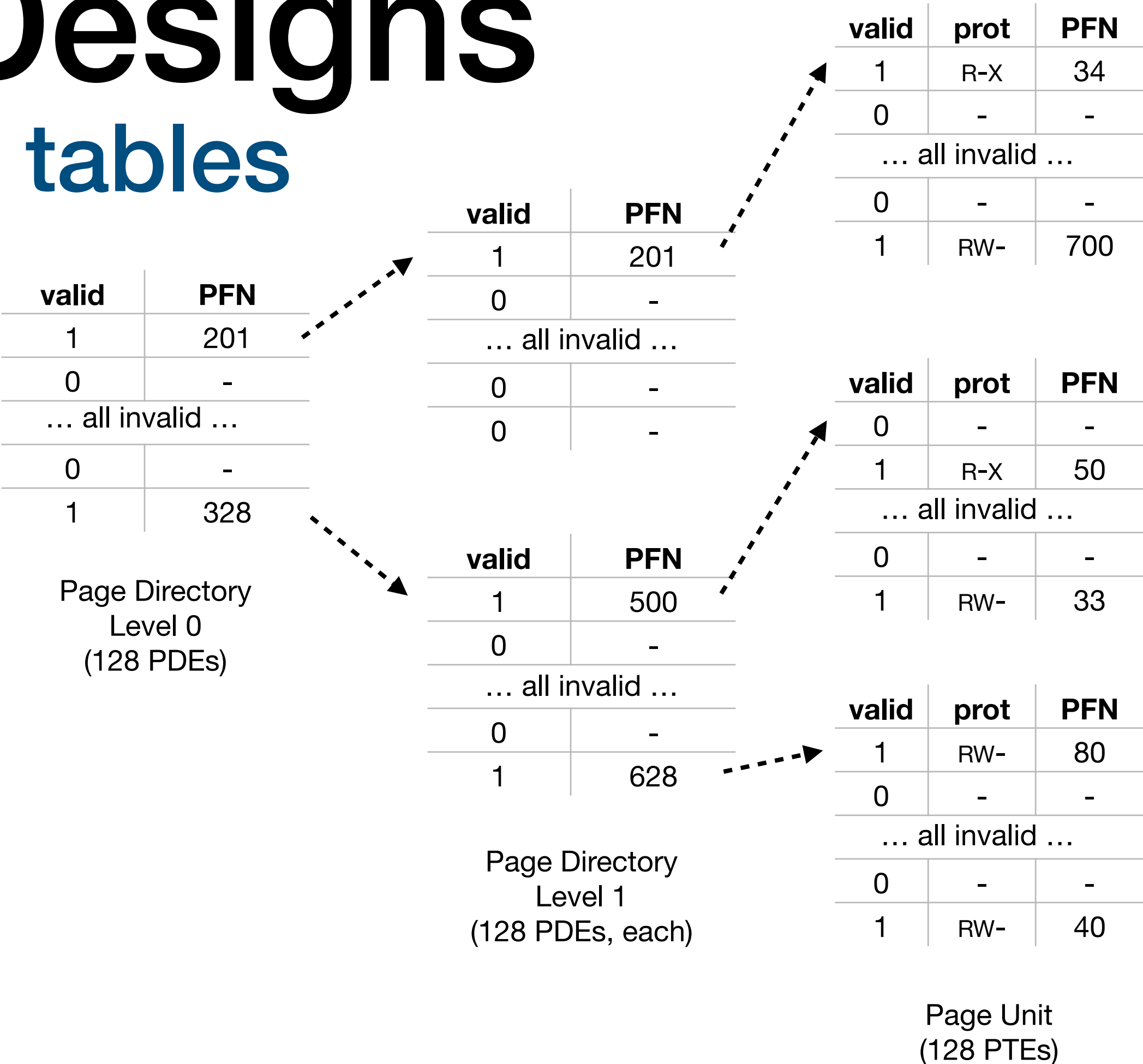
- **Yet another example:** 30-bit virtual address, with a 512-byte page size and a PTE of 4 bytes
 - 9 bits for the offset and 21 bits for the VPN
 - Each **page unit** holds 128 PTEs ($512 / 4$), so we need 7 bits (2^7) to index the PTEs
- Our **page directory** has 14 bits now. This means it holds $2^{14} = 16384$ entries, spanning across 128 pages
 - If we do the math for 64-bit virtual addresses, we end up with a page directory containing 2^{55} entries!
- And therefore, we may have an **incredibly large page directory** that will require a lot of memory, even if many PDEs are marked as invalid
 - How can we solve this issue?



Page Table Designs

Multi-level page tables

- **Solution:** break the tree into more than 2 levels
 - In our previous example, if we use 3 levels, the **page directory at level 0 fits exactly into a single frame** (i.e., contains 128 PDEs)
 - **Level 1 directories**, also fitting in a single frame, are only allocated **if these contain at least one valid entry**
- Space-wise, having each directory and page unit fit into a single frame is the best option!
 - However, every time a TLB miss occurs, we **multiply the number of memory accesses by the number of levels** (indirections)
- There is a **trade-off** between **space efficiency** and **performance**
 - Having many levels may not be worth it!



Page Table Designs

Inverted page table

- An alternative design to spare memory space is having one **page table shared across all processes!**
- **Each frame (PFN) has one entry at the table** mapping to
 - The **VPN**
 - The **process identifier (PID)**
- Finding the correct entry requires a linear scan **through the list**
 - A **hash table** can be built over the base structure to **speedup lookups**

PFN	VPN	PID
0	10	1
1	11	1
2	12	1
4	10	0
5	20	2
6	30	0
7	13	1

...

Inverted table
(logical view)

Supporting Larger Address Spaces

Memory hierarchy

- ◎ We have studied different ways to **reduce the memory used by address spaces and page tables**
- ◎ Even with the previous techniques, if many programs are running, we may want to further reduce memory usage without compromising
 - **Convenience:** Programmers should not be worried about writing programs whose data structures must fit into physical memory
 - **Performance:** We still want programs to run fast (i.e., access these data structures efficiently)
- ◎ What if the OS saves (**swaps**) portions of the address space (e.g., **pages**) that are not in great demand into another location with higher capacity, such as the disk device?
 - Actually, swapping can also be applied for **portions of page tables**, further reducing the memory used by these!
- ◎ This seems a good idea but comes with challenges
 - **Disk devices are slower than physical memory**, and they sit at the bottom of the memory hierarchy

Supporting Larger Address Spaces

The swap space

- If pages are going to be swapped from memory to disk, we must reserve some space in the latter. This location is called the **swap space**
- The size allocated for the **swap space** is important because, when combined with the **physical memory space**, it dictates the total **number of pages usable in the system**
 - For now, let's assume the swap space is very large
- **Example**
 - Processes 0, 1, and 2 are running while having some pages in memory and others in swap space
 - Process 3 has all pages on disk (swap space); thus, probably, it is not running

	PFN 0	PFN 1	PFN 2	PFN 3								
Physical Memory	Proc 0 [VPN 0]	Proc 1 [VPN 1]	Proc 1 [VPN 3]	Proc 2 [VPN 0]								
	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7				
Swap Space	Proc 0 [VPN 1]	Proc 0 [VPN 2]	[Free]	Proc 1 [VPN 0]	Proc 1 [VPN 1]	Proc 3 [VPN 0]	Proc 2 [VPN 1]	Proc 3 [VPN 1]				

Supporting Larger Address Spaces

Present bit and page fault

- When the OS or hardware (depending on the approach) are looking in the page table for a translation, the PTE entry must indicate if the page is on physical memory or disk
 - The PTE must have a **present bit** set to 1 if the page is in memory or 0 if it is on disk
- The act of accessing a page that is not in memory is referred to as a **page fault**
 - Under a page fault, the OS must be invoked to run a **page-fault handler** (i.e., to bring the page from disk into memory)
 - But where is the page located on the disk?
The **PFN** (at the PTE) can be used to **hold the disk address!**

Page table

PFN	valid	prot	present	dirty
1000	1	R-X	0	0
-	0	-	-	-
-	0	-	-	-
4	1	RW-	1	1
...				

Disk address! →

Supporting Larger Address Spaces

Page fault handler

- When invoked, the **page fault handler code** (at the OS) must
 - **Find a free frame in memory** for the page (what if no frames are available?)
 - **Read the page from disk** (I/O request) and place it in memory
 - When the I/O is complete, the corresponding **PTE must be updated**
i.e., the present bit set to 1 and the PFN updated to the physical memory frame number
 - **Retry** the memory access **instruction**
 - This retry may generate a **TLB miss** that must be handled by adding the new translation to the TLB
(an optimization is to update the TLB when handling the page fault to avoid this extra step)
- When the page is being read from disk, the process state is **blocked**, and the OS can run another program
 - Good for optimizing the use of hardware (**overlapping I/O and CPU time**)

Supporting Larger Address Spaces

What if memory is full?

- ◎ If memory is full (i.e., no frames are available), then the OS may need to first **page out** (i.e., swap to disk) one or more pages to make room for the new one(s)
 - Only then can it read the new page into physical memory (to a frame)
- ◎ The process of choosing what pages to replace (evict) is known as the **page-replacement policy**
- ◎ However, **waiting for the memory to be full** to replace pages is a **bad decision**
 - Remember that the page fault handler needs to wait for pages to be freed to proceed with its work...
 - What if pages are freed in the background?

Supporting Larger Address Spaces

Swap daemon

- ◎ A **swap daemon** (or **page daemon**) thread runs in the background and frees memory
 - But **when?** When the amount of **available frames** is **below a given number (low watermark - LW)**
 - **How many** pages should be freed? As many as needed for reaching **another number (high watermark - HW)**
- ◎ Having a daemon changes the fault handler workflow and allows for interesting optimizations
 - The **page fault handler** is **no longer responsible for replacing pages**. It just checks if there are free frames and, if not, informs the daemon and waits for a notification from it
 - The daemon can **group** several pages to evict into a single flush request to the disk
- ◎ All the mechanisms described below are done in a **transparent** way to processes
 - Each process accesses its own private memory without being concerned if the page is residing on disk or memory. The OS does all the job!
- ◎ And therefore, we have our **virtual memory** abstraction that **extends beyond physical memory!**

Page Replacement Policies

Challenge

- With **virtual memory**, the physical memory can be seen as a cache of pages in the system (the other pages are stored on disk!)
- The OS **goal** is to **minimize the number of cache misses** (i.e., the number of page faults!)
 - Alternatively, we can say that the OS wants to **maximize cache hits** (i.e., the number of times a page is found in memory)
- Let's use as our **metric** (to evaluate our policies) the **Average Memory Access Time (AMAT)**
 - **TM**: cost of accessing the memory
 - **TD**: cost of accessing the disk
 - **P_{MISS}**: probability of not finding the data in memory (a miss). **P_{MISS}** varies from 0.0 to 1.0

$$AMAT = TM + (P_{MISS} \times TD)$$

- **Intuition**: you always pay the cost of accessing memory when fetching a page but when you miss, you also pay the cost of fetching data from the disk

Page Replacement Policies

Example

- Assume **10 virtual pages** (pages 0 to 9), all in memory except for **page 3**. If a program accesses each of these pages once
 - The following cache behavior will happen: hit hit hit **miss** hit hit hit hit hit hit ($P_{\text{Miss}} = 0.1$)
 - The cost of accessing the disk is 10 milliseconds while accessing memory takes 100 nanoseconds (0.0001 ms)

$$\mathbf{AMAT} = 0.0001 \text{ ms} + (0.1 \times 10) \text{ ms} = 1.0001 \text{ ms}$$

- If P_{Miss} is 0.001 (i.e., 99.9% hit rate probability)

$$\mathbf{AMAT} = 0.0001 \text{ ms} + (0.001 \times 10) \text{ ms} = 0.0101 \text{ ms} = 10.1 \text{ microseconds}$$

- Since the cost of accessing disk is much higher than accessing memory, one must have very low miss rates (i.e., one needs **good page replacement policies!**)

Page Replacement Policies

Optimal Policy

- Is there an **optimal page replacement policy** (i.e., leading to the **fewest number of misses overall**)?
 - Yes, Belady proposed it many years ago (originally called MIN)
- **Strategy:** *replace the page that will be accessed **furthest in the future**!*
- Let's assume a program that accesses the following stream of pages:
0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1
 - The cache has **3 available frames** and **starts with an empty state**, i.e., the first time a different page is requested, it results in a miss (known as a **cold-start miss**)
- **Hit rate:** $\frac{\text{hits}}{\text{accesses}} = \frac{6}{11} = 0.545 = 54.5\%$ (85.7%, excluding cold-start misses)
- **Problem:** the future is not generally known, and therefore, we cannot build this policy for a general-purpose OS
 - But we can use the optimal policy for comparison purposes!

Access	Hit/Miss?	Evict	Resulting Cache State
0	MISS		0
1	MISS		0, 1
2	MISS		0, 1, 2
0	HIT		0, 1, 2
1	HIT		0, 1, 2
3	MISS	2	0, 1, 3
0	HIT		0, 1, 3
3	HIT		0, 1, 3
1	HIT		0, 1, 3
2	MISS	3	0, 1, 2
1	HIT		0, 1, 2

Page Replacement Policies

FIFO (first-in, first-out)

- Early systems employed very simple replacement policies like FIFO
- **Strategy:** Replace the page that has been in memory for a longer time
- Easy to implement! Add pages to a queue when they enter the system and replace the one at the tail of the queue
- Assuming the same stream of pages accesses

► **Hit rate:** $\frac{\text{hits}}{\text{accesses}} = \frac{4}{11} = 0.364 = 36.4\%$ (85.7%, excluding cold-start misses)

Access	Hit/Miss?	Evict	Resulting Cache State
0	MISS		First-in→ 0
1	MISS		First-in→ 0, 1
2	MISS		First-in→ 0, 1, 2
0	HIT		First-in→ 0, 1, 2
1	HIT		First-in→ 0, 1, 2
3	MISS	0	First-in→ 1, 2, 3
0	MISS	1	First-in→ 2, 3, 0
3	HIT		First-in→ 2, 3, 0
1	MISS	2	First-in→ 3, 0, 1
2	MISS	3	First-in→ 0, 1, 2
1	HIT		First-in→ 0, 1, 2

Page Replacement Policies

Belady Anomaly (side-note)

● What if we try FIFO with the following stream of page accesses:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

● Strange result!

- ▶ With 3 frames available - 3 hits!
- ▶ With 4 frames available - 2 hits!

● **Conclusion:** with FIFO, increasing the cache size may not mean a better hit rate

Access	Hit/Miss?	Evict	Resulting Cache State
1	MISS		First-in→ 1
2	MISS		First-in→ 1, 2
3	MISS		First-in→ 1, 2, 3
4	MISS	1	First-in→ 2, 3, 4
1	MISS	2	First-in→ 3, 4, 1
2	MISS	3	First-in→ 4, 1, 2
5	MISS	4	First-in→ 1, 2, 5
1	HIT		First-in→ 1, 2, 5
2	HIT		First-in→ 1, 2, 5
3	MISS	1	First-in→ 2, 5, 3
4	MISS	2	First-in→ 5, 3, 4
5	HIT		First-in→ 5, 3, 4

With 3 frames

Access	Hit/Miss?	Evict	Resulting Cache State
1	MISS		First-in→ 1
2	MISS		First-in→ 1, 2
3	MISS		First-in→ 1, 2, 3
4	MISS		First-in→ 1, 2, 3, 4
1	HIT		First-in→ 1, 2, 3, 4
2	HIT		First-in→ 1, 2, 3, 4
5	MISS	1	First-in→ 2, 3, 4, 5
1	MISS	2	First-in→ 3, 4, 5, 1
2	MISS	3	First-in→ 4, 5, 1, 2
3	MISS	4	First-in→ 5, 1, 2, 3
4	MISS	5	First-in→ 1, 2, 3, 4
5	MISS	1	First-in→ 2, 3, 4, 5

With 4 frames

Page Replacement Policies

Random

◎ **Strategy:** Replace a random page

◎ Very easy to implement, like FIFO. The ratio of hits and misses depends on making lucky or unlucky choices!

◎ Going back to our original stream of page accesses

▸ **Hit rate:** $\frac{\text{hits}}{\text{accesses}} = \frac{5}{11} = 0.455 = 45.5\%$ (62.5%, excluding cold-start misses)

◎ A random strategy will exhibit different behaviors if one repeats the same stream of accesses several times

▸ Take a look at Section 22.4 of Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, 2018. for further details!

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	HIT		0, 1, 2
1	HIT		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	HIT		2, 3, 0
1	Miss	3	2, 0, 1
2	HIT		2, 0, 1
1	HIT		2, 0, 1

Page Replacement Policies

Using frequency and recency

- ◎ FIFO and Random are simple to implement but do not look at the **importance of pages**
 - As we did for process scheduling, let's look at the past to make better decisions in the future!
 - **Frequency:** if a page has been accessed many times in the past, maybe it is worth not replacing it
 - **Recency:** if a page was recently accessed, maybe it will be accessed again in the near future
- ◎ Policies looking at these metrics are based, again, on the principle of locality
 - **Spatial locality:** if a page P is accessed, the pages around it (say $P - 1$ or $P + 1$) will also be accessed (e.g., iterating through an array)
 - **Temporal locality:** pages accessed in the near past are likely to be accessed again in the near future (e.g., accessing the same variables inside a loop)
 - Note that locality may not be present in all programs

Page Replacement Policies

Least Recently Used (LRU)

● And so, two main policies arise

- **Least Frequently Used (LFU)**: replace the least-frequently used page
- **Least Recently Used (LRU)**: replace the least-recently used page
- The opposite policies also exist: **Most Frequently Used (MFU)** and **Most Recently Used (MRU)**. In most cases (not all), these policies do not work well as they do not embrace locality

● Let's look at **LRU** for our running example

● Assuming the same stream of pages accesses

- **Hit rate:** $\frac{\text{hits}}{\text{accesses}} = \frac{6}{11} = 0.545 = 54.5\%$ (75%, excluding cold-start misses)

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Page Replacement Policies

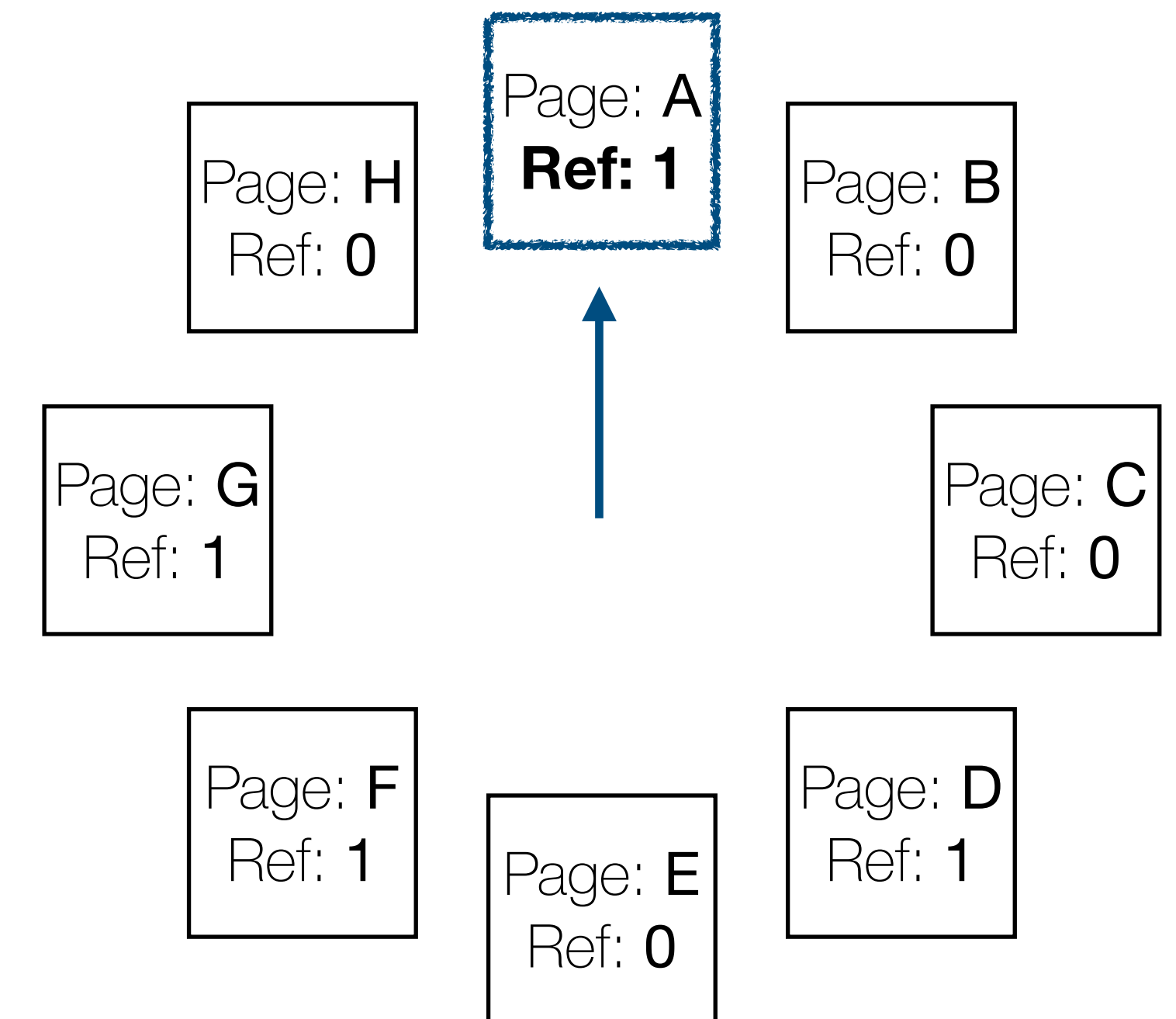
Implementing LRU

- ◎ LRU provides the best results so far. However, implementing it **perfectly** is challenging
 - With some help from the hardware, the timestamps of page accesses could be saved, for example, at page table(s). Then, the OS would need to **scan the full list of timestamps** for all page tables to evict the least recently used page
 - Another option would be to have a queue like in FIFO, but when a page is accessed one **must search for it in the list** and promote it to the list's head
- ◎ Both approaches require too much work from the OS and would delay page eviction (the OS must be efficient!)
 - Imagine scanning through millions of pages...

Page Replacement Policies

Approximating LRU

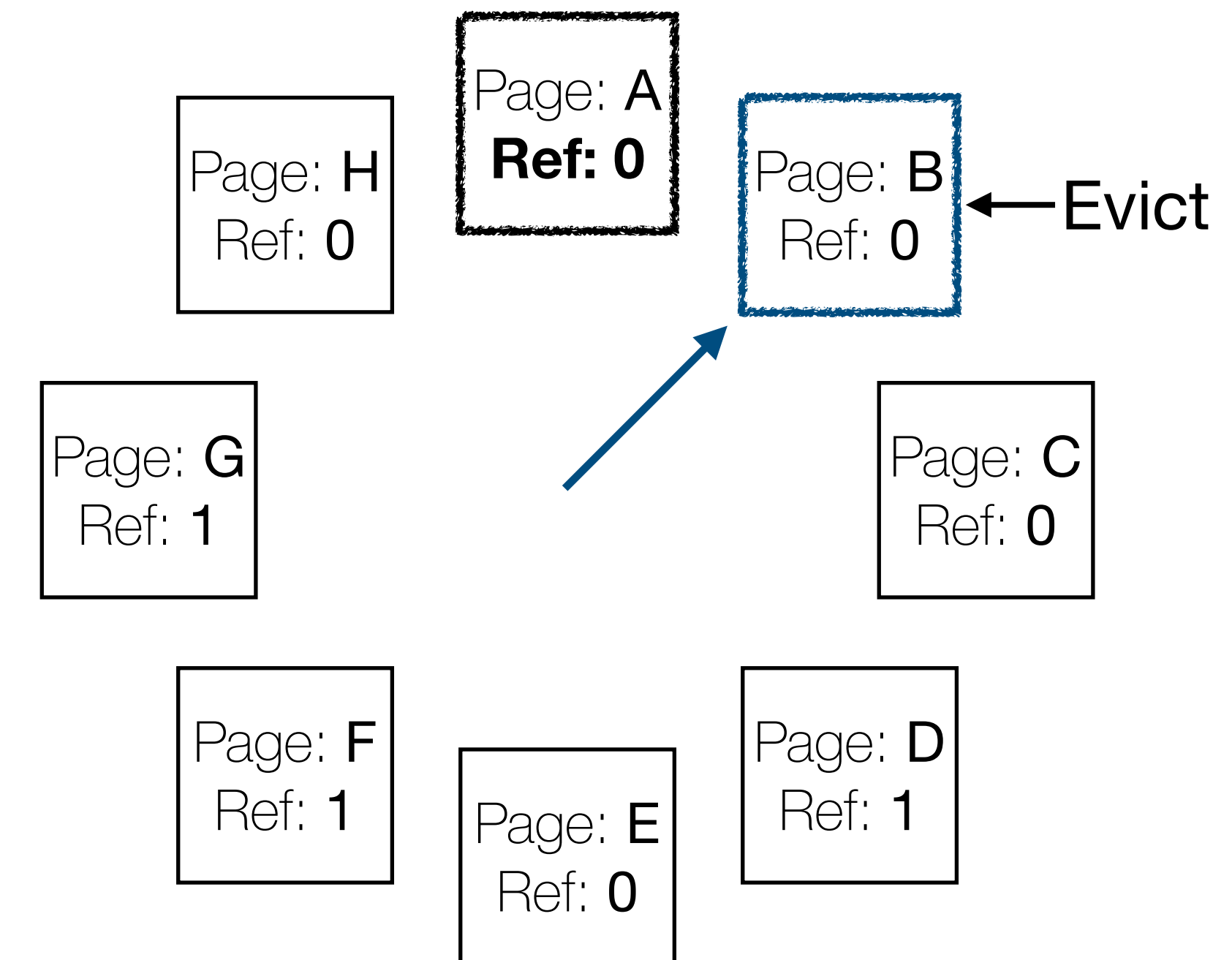
- Modern systems approximate LRU behavior
- A **reference bit** (also called **use bit**) at the PTE is set by the hardware to 1 when the corresponding page is accessed
- The **clock algorithm**
 - Imagine all pages of the system organized into a circular list
 - A **clock hand** points to one page
 - When a page must be replaced, the OS looks at the page pointed by the clock hand
 - If the reference **bit is 1**, the **page was used recently**, so the bit is set to 0 by the OS, and our clock algorithm advances to the next page (moves the clock hand)
 - **If the bit is 0**, then **the page is evicted**



Page Replacement Policies

Approximating LRU

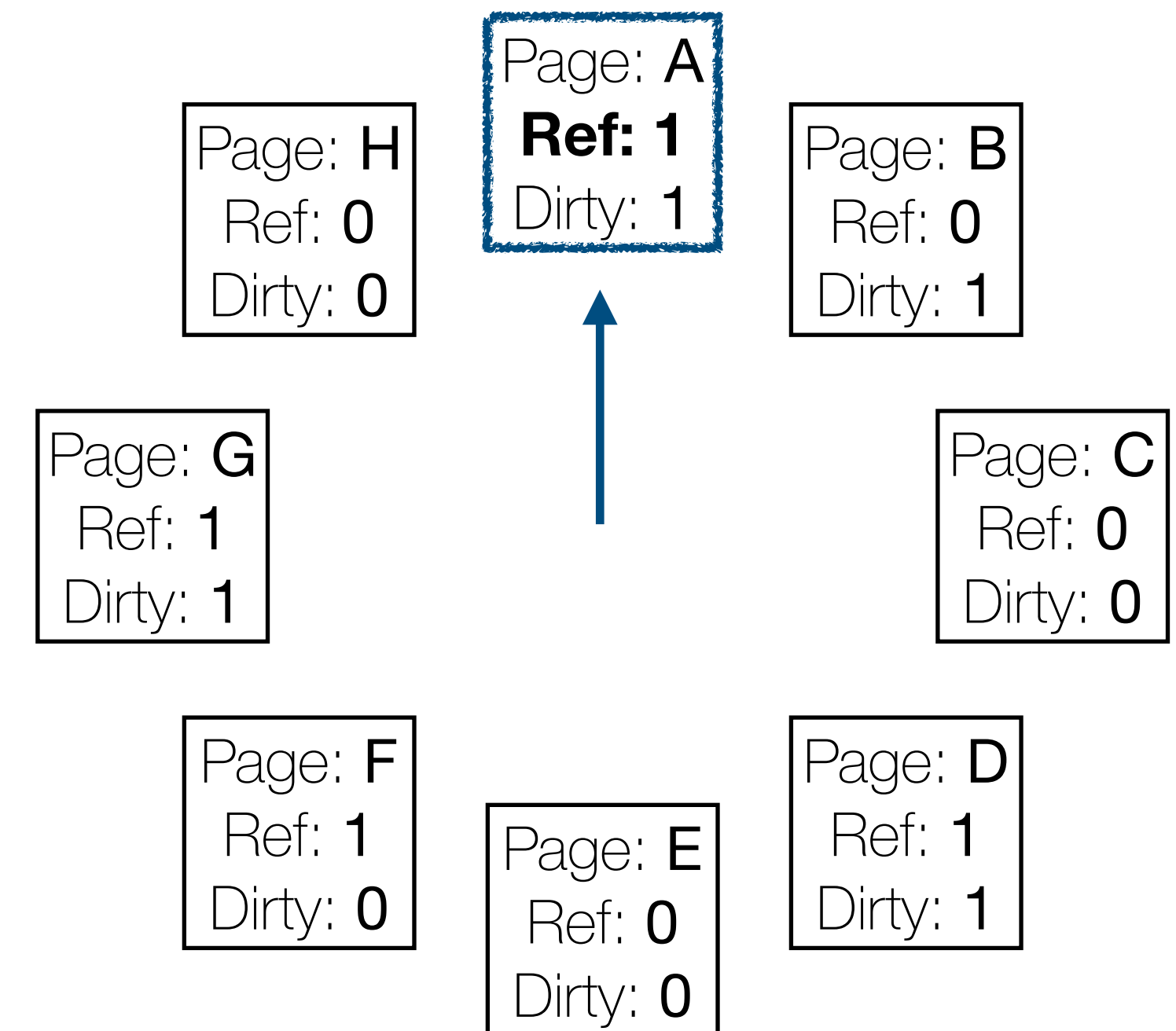
- Modern systems approximate LRU behavior
- A **reference bit** (also called **use bit**) at the PTE is set by the hardware to 1 when the corresponding page is accessed
- The **clock algorithm**
 - Imagine all pages of the system organized into a circular list
 - A **clock hand** points to one page
 - When a page must be replaced, the OS looks at the page pointed by the clock hand
 - If the reference **bit is 1**, the **page was used recently**, so the bit is set to 0 by the OS, and our clock algorithm advances to the next page (moves the clock hand)
 - **If the bit is 0**, then **the page is evicted**



Page Replacement Policies

Approximating LRU

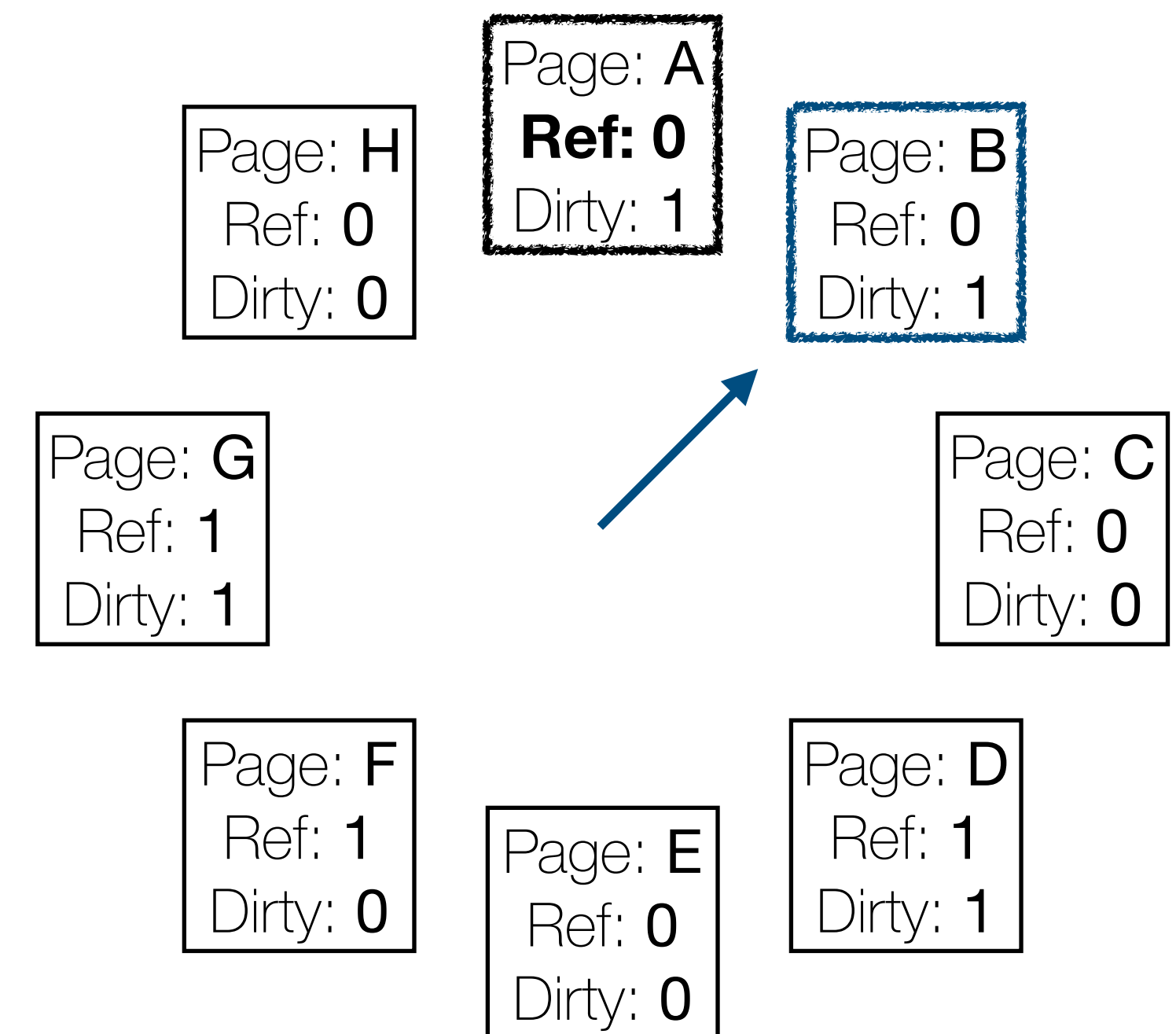
- A small improvement to the previous algorithm considers if the page content was modified after bringing (swapping in) the page to memory
 - The PTE holds a **dirty** bit set to 1 by the hardware when the page is modified
- Some pages will be swapped in and out of memory several times. If there is enough space at the swap area, the OS can leave the content of pages there, even when these are swapped back into memory. With such trick
 - **Evicting an unmodified** (non-dirty) **page** does not require an extra disk I/O operation (i.e., the most recent content of the page is already at the swap area)
 - **Evicting a dirty page** requires writing the modified content back to disk (i.e., updating the content of the page at the swap area)
- Clock algorithm (improved)
 1. First, try evicting pages that were not accessed recently or modified
 2. Then, choose pages that were not accessed recently but that are dirty
 3. And so on, ...



Page Replacement Policies

Approximating LRU

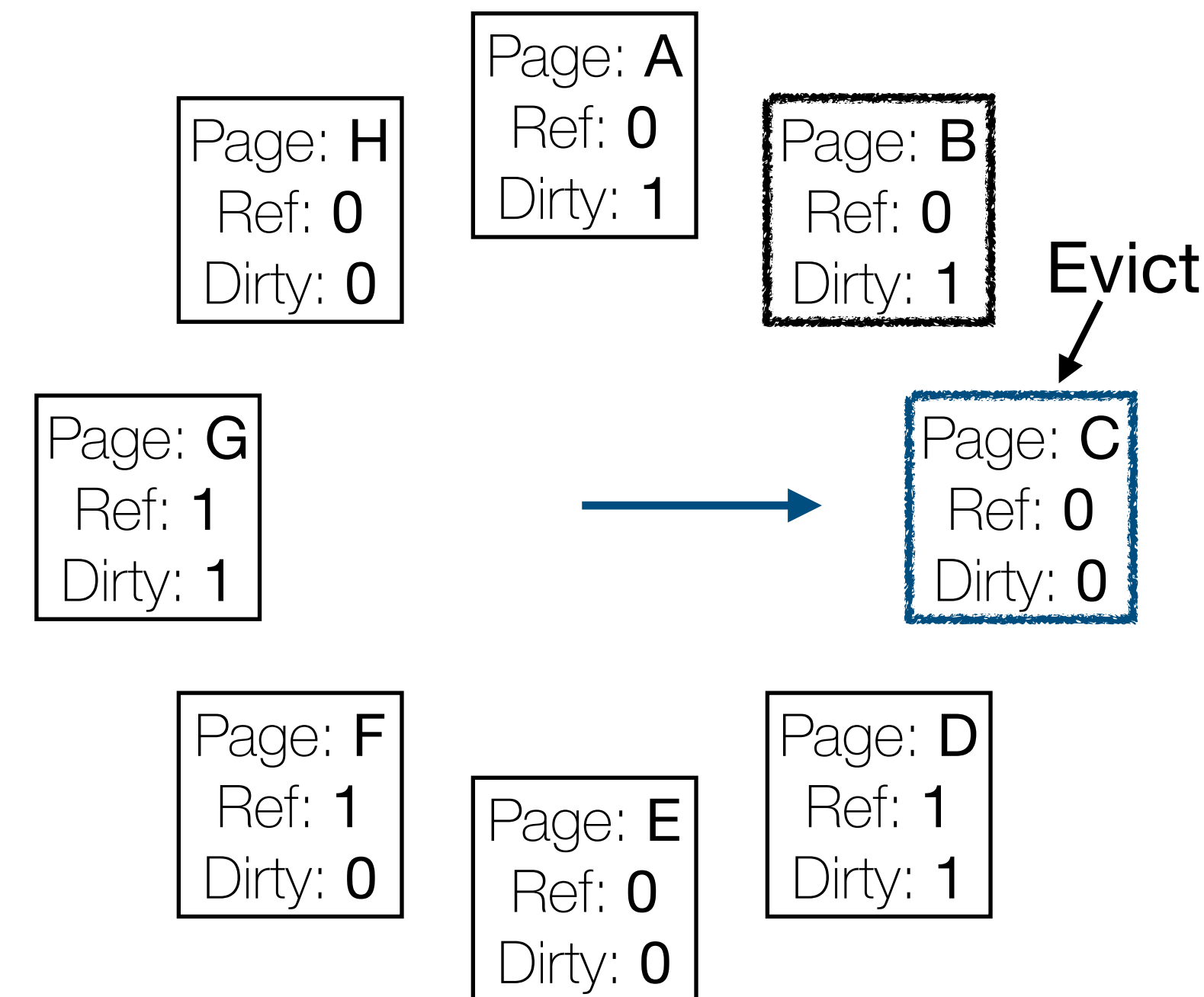
- A small improvement to the previous algorithm considers if the page content was modified after bringing (swapping in) the page to memory
 - The PTE holds a **dirty** bit set to 1 by the hardware when the page is modified
- Some pages will be swapped in and out of memory several times. If there is enough space at the swap area, the OS can leave the content of pages there, even when these are swapped back into memory. With such trick
 - **Evicting an unmodified** (non-dirty) **page** does not require an extra disk I/O operation (i.e., the most recent content of the page is already at the swap area)
 - **Evicting a dirty page** requires writing the modified content back to disk (i.e., updating the content of the page at the swap area)
- Clock algorithm (improved)
 1. First, try evicting pages that were not accessed recently or modified
 2. Then, choose pages that were not accessed recently but that are dirty
 3. And so on, ...



Page Replacement Policies

Approximating LRU

- A small improvement to the previous algorithm considers if the page content was modified after bringing (swapping in) the page to memory
 - The PTE holds a **dirty** bit set to 1 by the hardware when the page is modified
- Some pages will be swapped in and out of memory several times. If there is enough space at the swap area, the OS can leave the content of pages there, even when these are swapped back into memory. With such trick
 - **Evicting an unmodified** (non-dirty) **page** does not require an extra disk I/O operation (i.e., the most recent content of the page is already at the swap area)
 - **Evicting a dirty page** requires writing the modified content back to disk (i.e., updating the content of the page at the swap area)
- Clock algorithm (improved)
 1. First, try evicting pages that were not accessed recently or modified
 2. Then, choose pages that were not accessed recently but that are dirty
 3. And so on, ...



Page Replacement Policies

Comparison

- ◎ When programs **do not exhibit the locality property**, **FIFO**, **Random**, and **LRU** perform similarly. The major factor to improve hit ratio is the size of the cache
- ◎ When programs exhibit **high locality**, **LRU** performs **better**. For instance, when a large percentage of requests (e.g., 80%) is repeatedly done to a small set of pages (e.g., 20%) that fit into the cache space
- ◎ When programs **sequentially scan large portions of memory** (e.g., a very large array) and the cache size is inferior to the pages being scanned, then **FIFO** and **LRU** perform **badly** (worst-case scenario for these strategies). Random performs a bit better
 - Some modern strategies tweak LRU to better handle these scan workloads (e.g., the ARC algorithm)
- ◎ Take a look at Section 22.6 of Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, 2018. for further details about these different workloads!

Other Techniques

Prefetching and clustering

- Besides replacing pages, the virtual memory manager also employs other **policies** and **optimizations**
 - For many pages, the OS applies **demand paging**, meaning that on-disk pages are only fetched to memory when accessed
 - However, in other cases, the OS may guess that a page is about to be used and may **prefetch** it from disk into memory (e.g., useful for pages containing the code of programs)
- Another policy that we discussed before consists of **grouping (clustering)** several pages to be written to disk at once

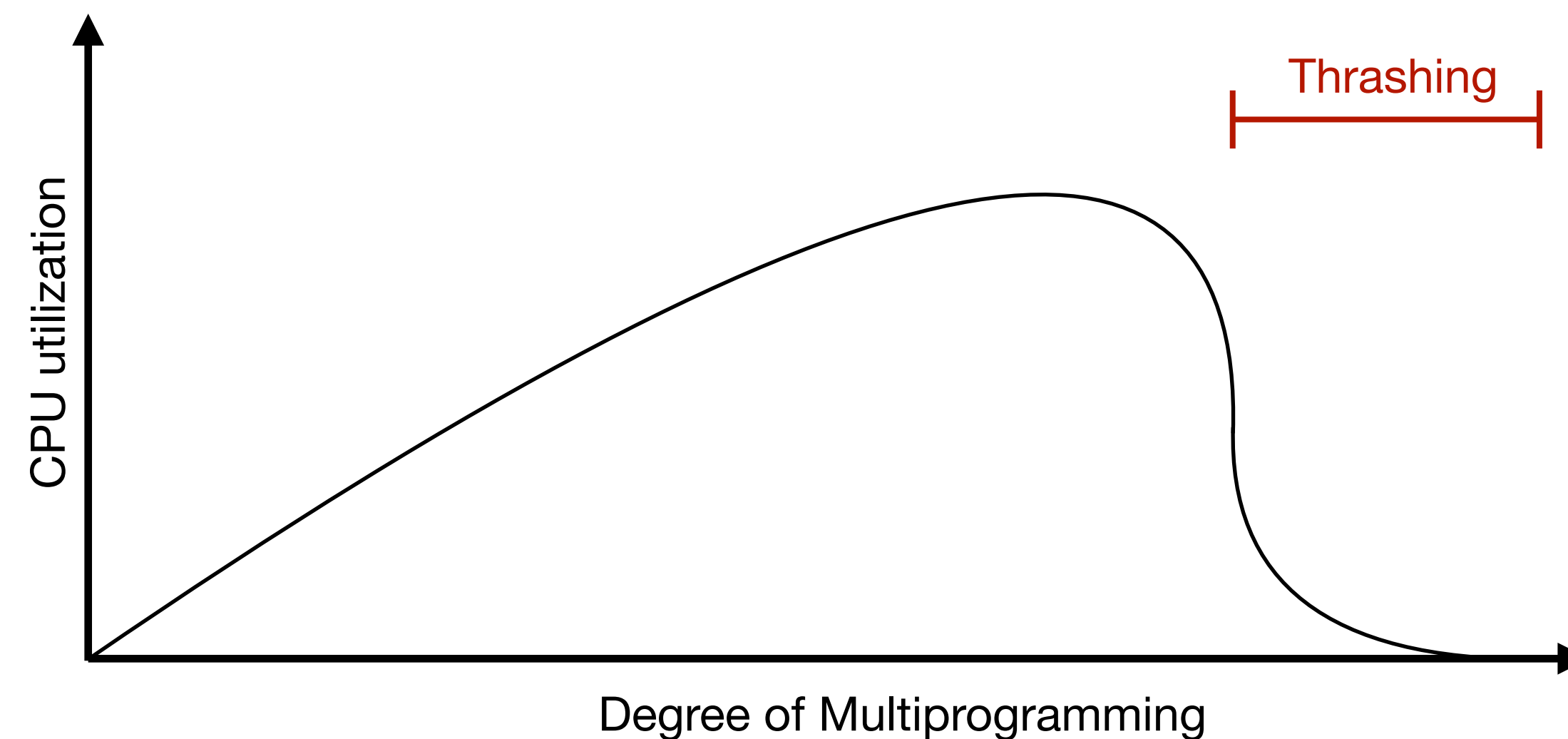
Thrashing

Phenomenon

- Sometimes, the **set of running processes exceeds the available memory**, and there are **no good candidate pages to evict**
- Worst, if the OS decides to evict processes' pages that are “immediately” going to be asked next (e.g., the scheduler decides to run these processes again), this will lead to more page faults, which will lead to other page faults, ...
 - This phenomenon, known as thrashing, makes the OS spend most of its time handling page faults and swapping pages instead of running processes!

Thrashing Scheduler

- Actually, if the OS process scheduler is naive, it may lead to the following scenario
 - As the system is thrashing, **CPU utilization drops** because **more disk I/O** requests are being made (i.e., page fault handling)
 - The scheduler sees a low CPU usage and decides to **run more processes**, leading to a more accentuated **thrashing phenomenon**
 - This means that **looking only at the CPU** is a bad idea. The scheduler should **also look** at the **page fault rate** for instance!



Thrashing

Solution

- ◎ Early systems would try to estimate the **working set** (i.e., the number of pages actively used) of all processes and run a subset of these that would not exceed the available memory
 - This approach, known as **admission control**, is based on the idea that doing less work and well is better than doing everything at once but poorly
- ◎ Modern systems typically run an **out-of-memory killer** that chooses and kills a memory-intensive process. This is a bit problematic if the process is the X server (leaving applications using the display unusable...)

More Information

- **Chapters 18 to 22** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.
- Want to know even more about **complete virtual memory systems**?
 - **Chapter 23** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, 2018.

Questions?