Week 7

Memory Management: Demand Paging

Oana Balmau

February 14, 2023

Announcements

Graded exercises released and
Assignment 1 grades are released

Ungraded exercises will be released

Week 6 **Memory Management**	feb 6 C Review: C files	feb 7 **Virtual Memory (1/2)** Optional reading: OSTEP Chapters 12 – 18	feb 8 Scheduling Assignment Released	feb 9 **Virtual Memory (2/2)** Scheduling Assignment Overview -- with Jiaxuan	feb 10
Week 7 **Memory Management**	feb 13 **OS Shell Assignment Due** C Review: Working with pthreads I	feb 14 **Demand Paging (1/3)** Optional reading: OSTEP Chapters 19 – 22	feb 15	feb 16 **Demand Paging (2/3)**	feb 17
Week 8 **Memory Management**	feb 20 C Review: Working with pthreads I	feb 21 **Demand Paging (3/3)** Optional reading: OSTEP Chapters 19 – 22 Practice Exercises Sheet: Memory Management	feb 22	feb 23 Mid-semester Q&A – not recorded • Graded Exercises Sheet Released • Grades released for OS Shell Assignment	feb 24
Week 9	feb 27	feb 28	mar 1	mar 2	mar 3

Key Concepts

- TLB
- Page Table for very large address spaces
- Demand paging
  - Page fault
- Page replacement policies

Recap Week 6: Virtual vs. Physical Address Space

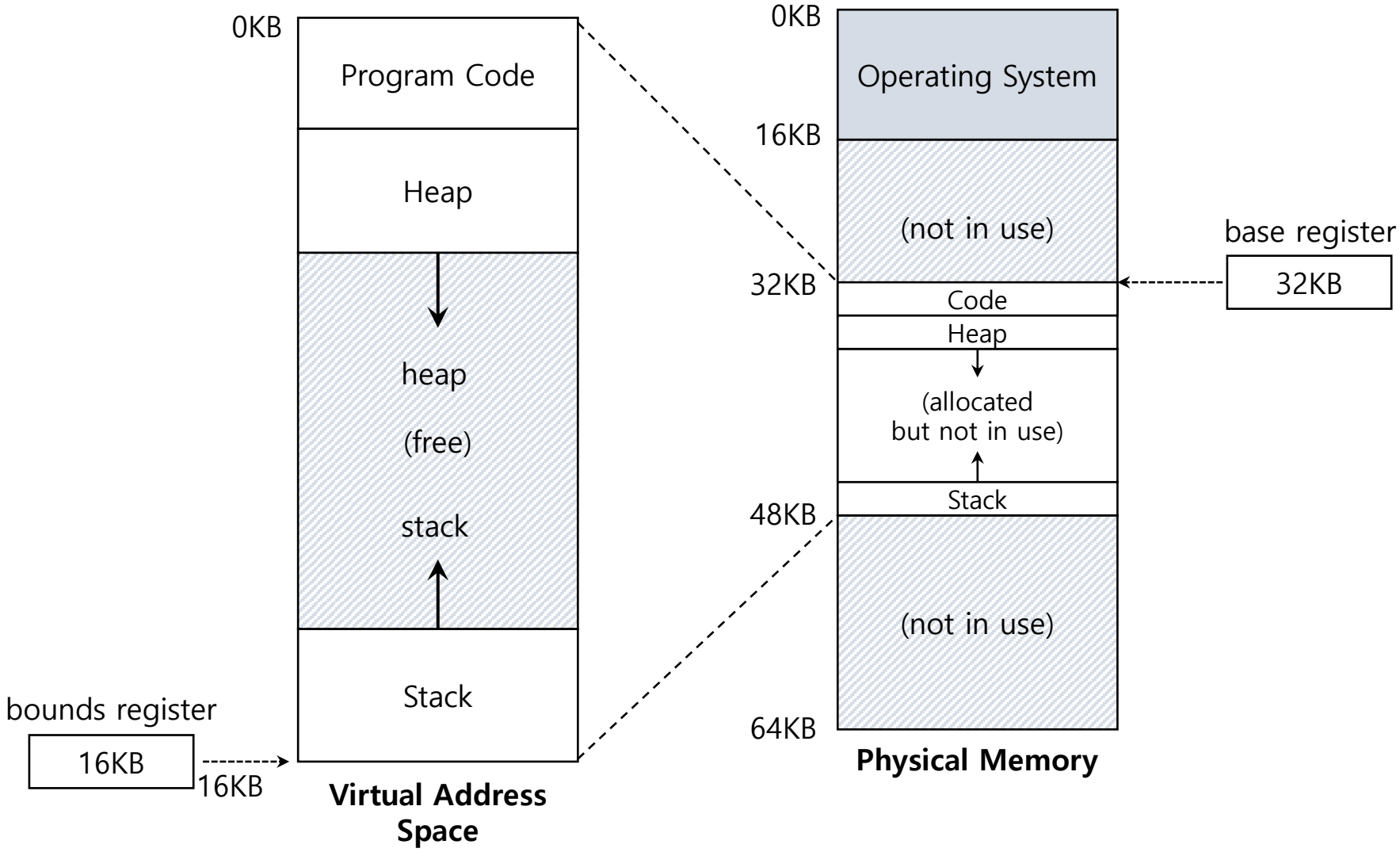
Virtual/logical address space = What the program(mer) thinks is its memory

Physical address space = Where the program actually is in physical memory

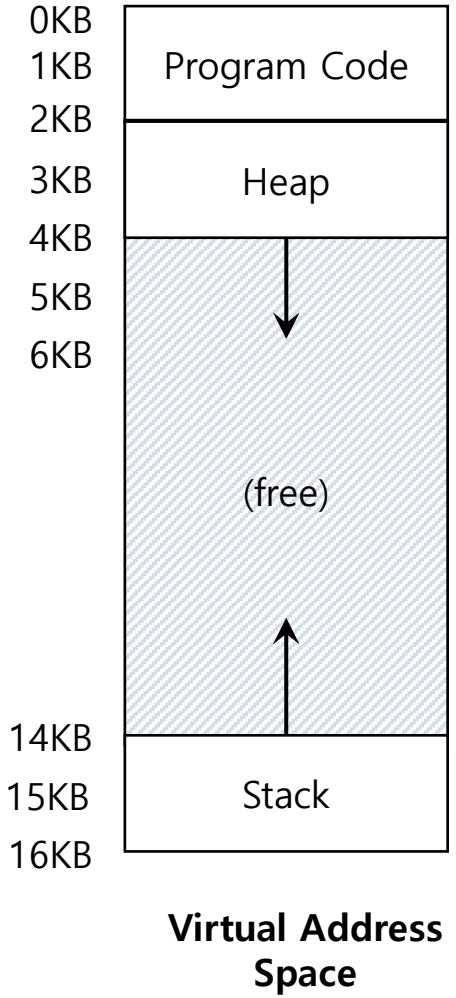
Recap Week 6: Memory Mapping Methods

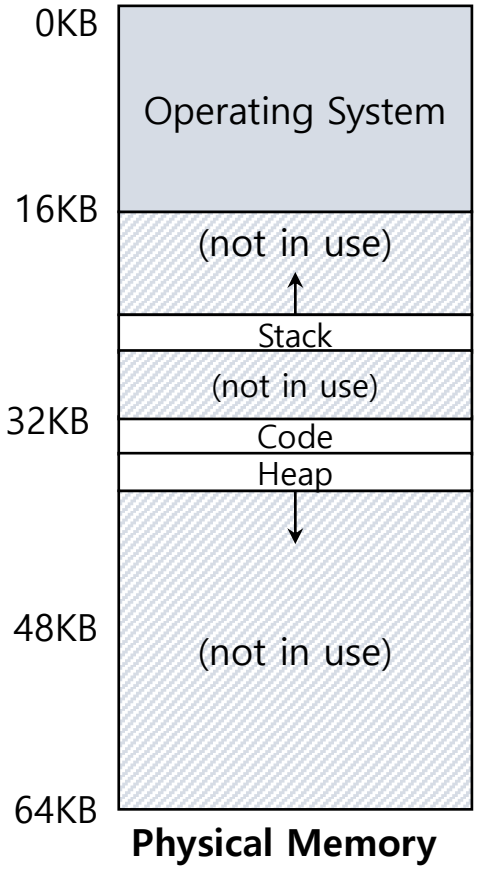
- Base and bounds
- Segmentation

Recap Week 6: Base and Bounds



Recap Week 6: Segmentation





Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Sharing Memory between Processes

Why would we want to do that?

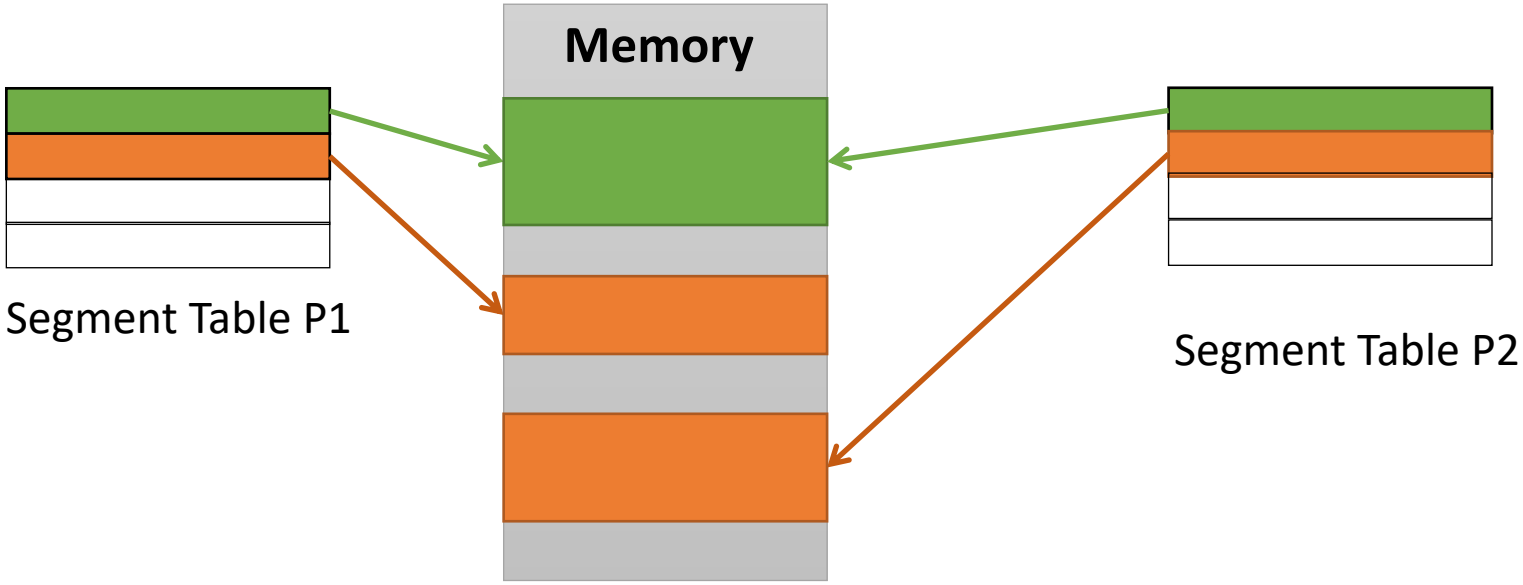
For instance,

- Run twice the same program in different processes
- May want to share code
- Read twice the same file in different processes
- May want to share memory corresponding to file

Sharing not possible with base and bounds, but is possible with segmentation

Segmentation Provides Easy Support for Sharing

- Create segment for shared data
- Add segment entry in segment table of both processes
- Points to shared segment in memory



Segmentation Provides Easy Support for Sharing

Extra **hardware** support is need for form of **Protection bits.**

- **A few more bits** per segment to indicate **permissions** of **read,** write and **execute**.

Example Segment Register Values(with Protection)

Segment	Base	Size	Protection
Code	32K	2K	Read-Execute
Heap	34K	2K	Read-Write
Stack	28K	2K	Read-Write

Main memory allocation with Segmentation

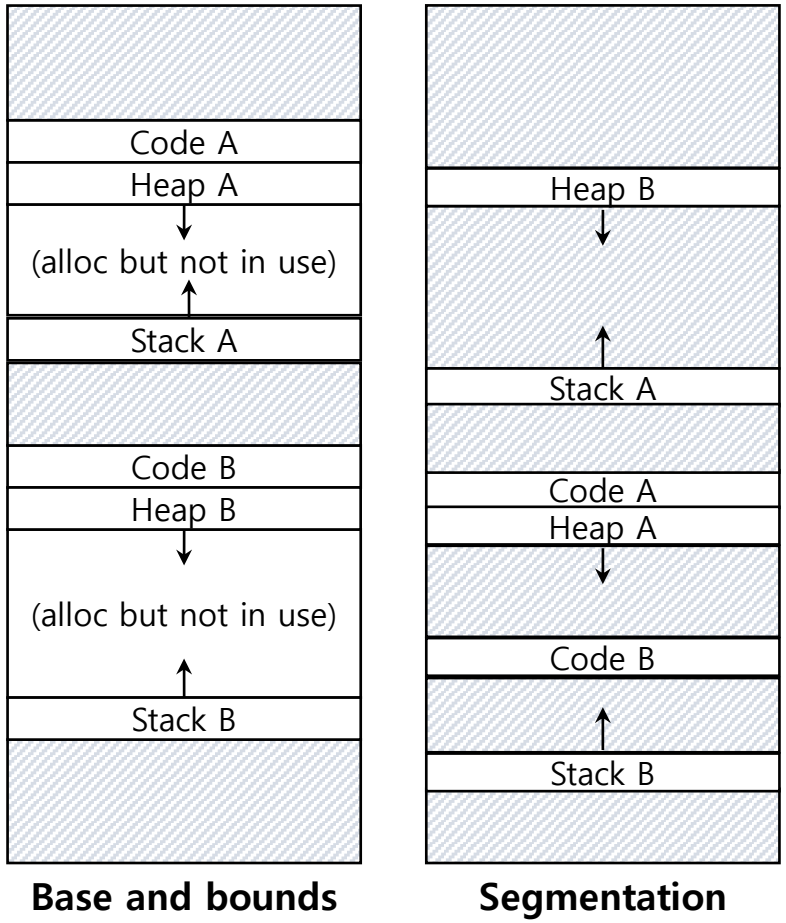
Remember:

Segmentation ~= multiple base-and-bounds

No internal fragmentation inside each segment.

External fragmentation problem is similar.

- Pieces are typically smaller



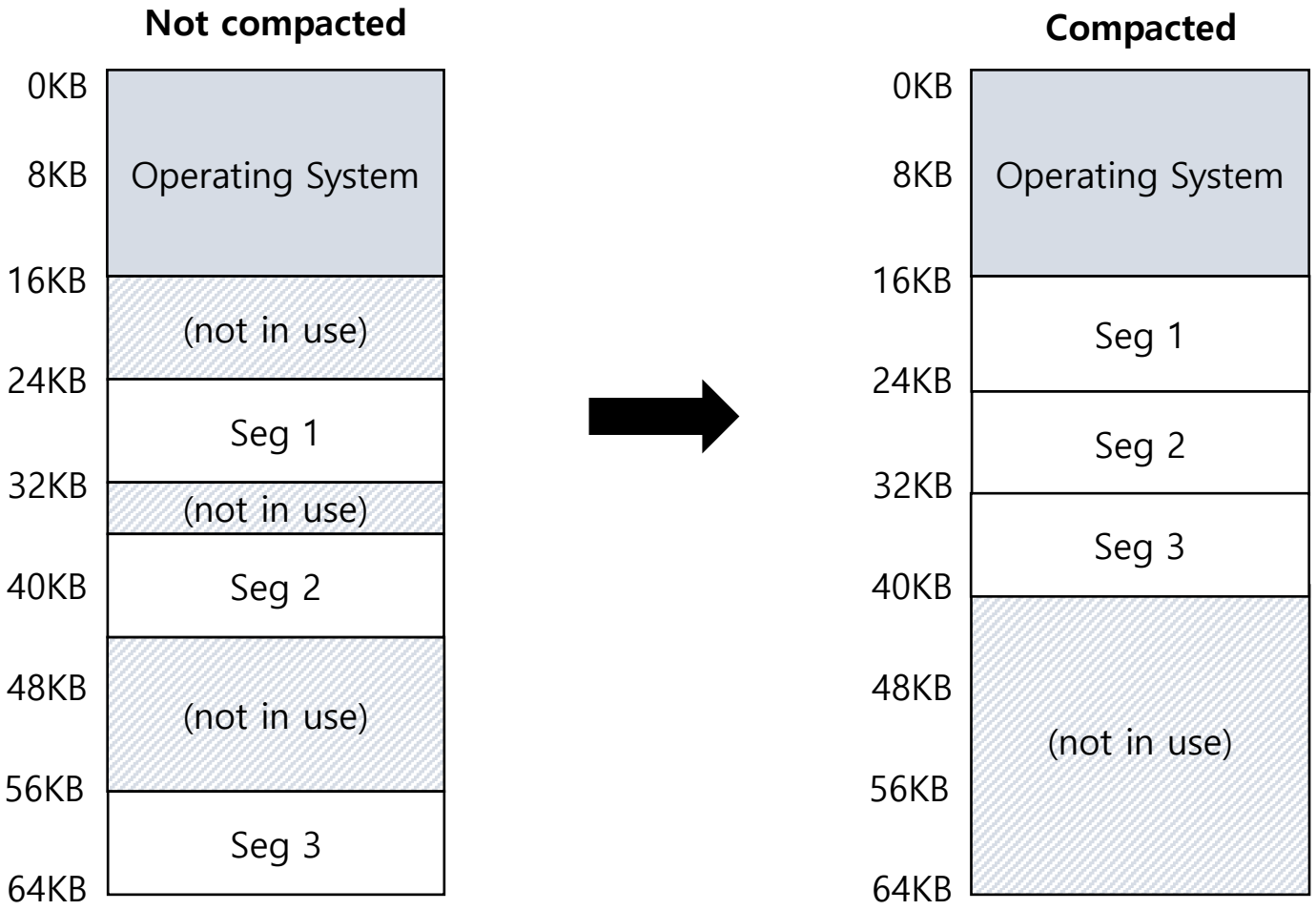
Main memory allocation with Segmentation

Compaction:

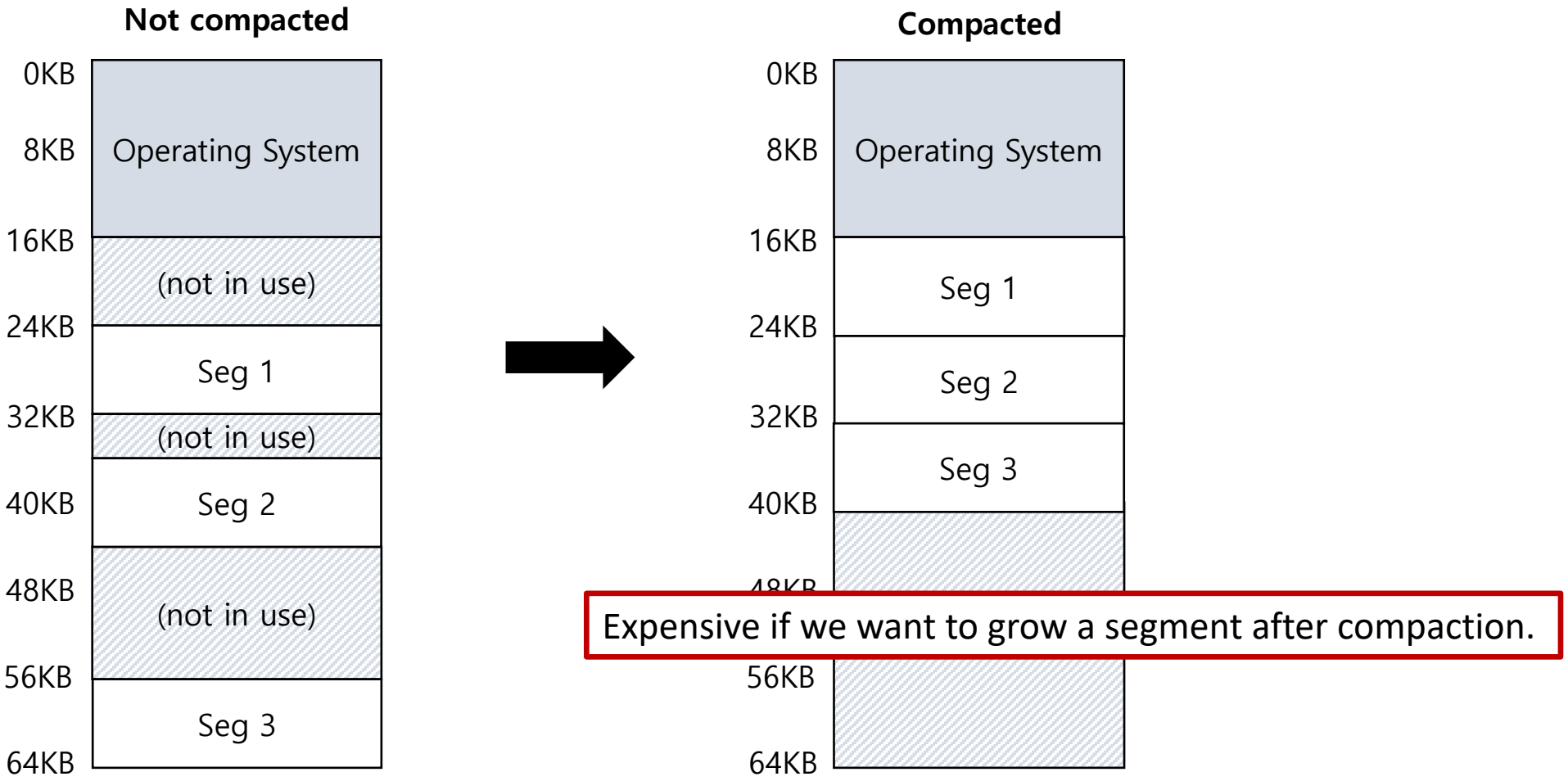
Rearrange segments in physical memory to get rid of “holes”.

- **Stop** running process. Inefficient! ☹
- **Copy** data to somewhere. Expensive! ☹☹
- **Change** segment register value.

Memory Compaction Example



Memory Compaction Example



Paging (simplified version)

Paging (simplified version)

- **Page:** fixed-size portion of virtual memory
- **Frame:** fixed-size portion of physical memory
- **Page size = frame size**
- Typical size: 4k – 8k (always power of 2)

Paging

Virtual Address Space

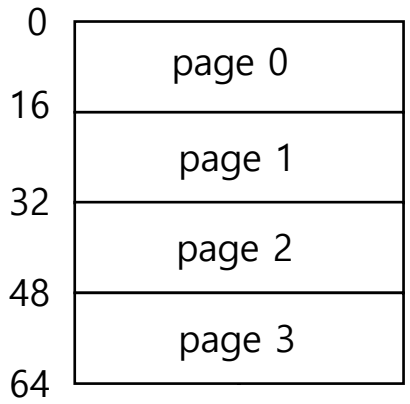
- Linear from 0 up to a multiple of page size

Physical Address Space

- Noncontiguous set of frames, one per page

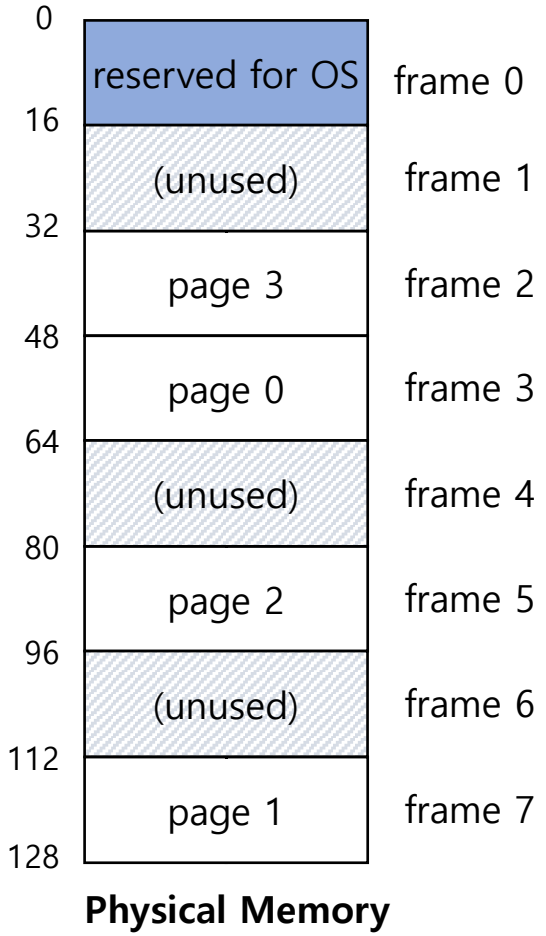
Paging: Example

Contiguous



Virtual Address Space

64B address space with four **16B pages**



Not Contiguous

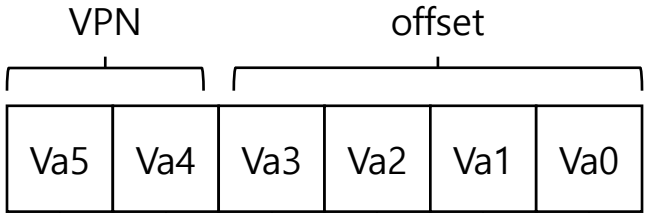
Physical Memory

128B physical memory with eight **16B frames**

Paging: Virtual Address

Two components in the virtual address

- VPN: virtual page number
- Offset: offset within the page; Page size = 2^{offset}

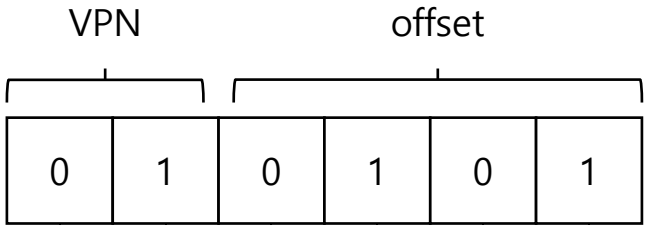


Paging: Virtual Address Example

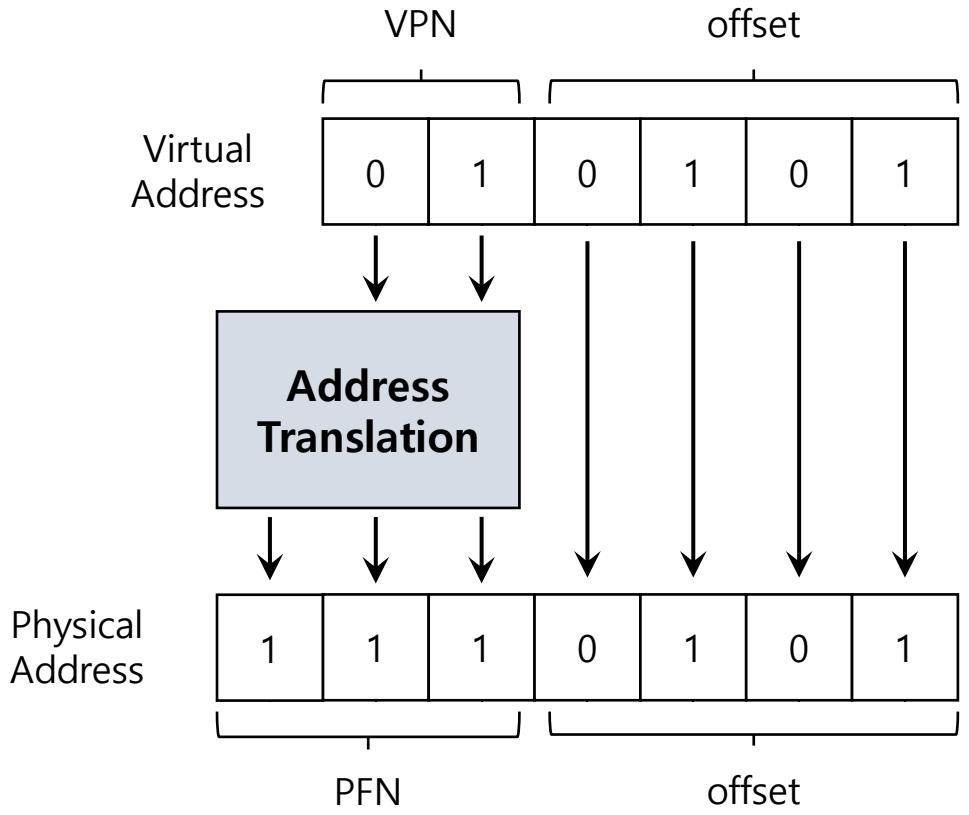
Virtual address 21 (binary 010101) in 64-byte address space

2 VPN bits → Number of pages = 2^2 = 4 pages

4 offset bits → Page size = 2^4 = 16B



Paging: Virtual to Physical Address Example



MMU for Paging

Page table

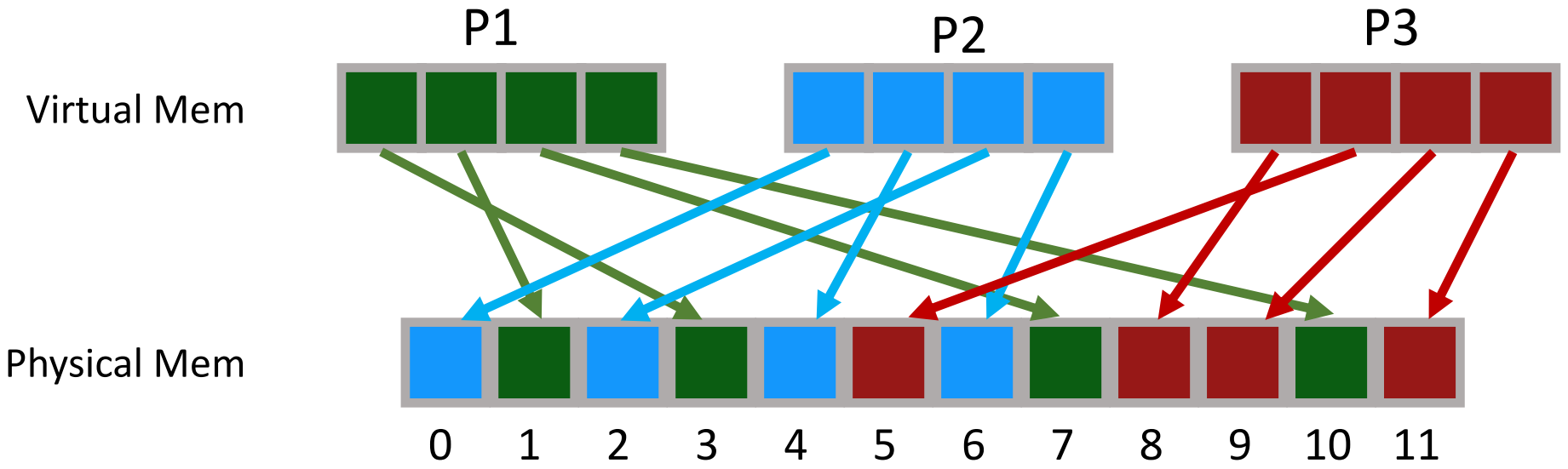
- **Data structure** used to map the virtual address to physical address
- Indexed by page number
- Contains frame number of page in memory
- **Each process has a page table**

Also need:

Pointer to page table in memory

Length of page table

Quiz: Fill in Page Table



Page Tables:

P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Page Tables can get large

32-bit address space with 4-KB pages, 20 bits for VPN

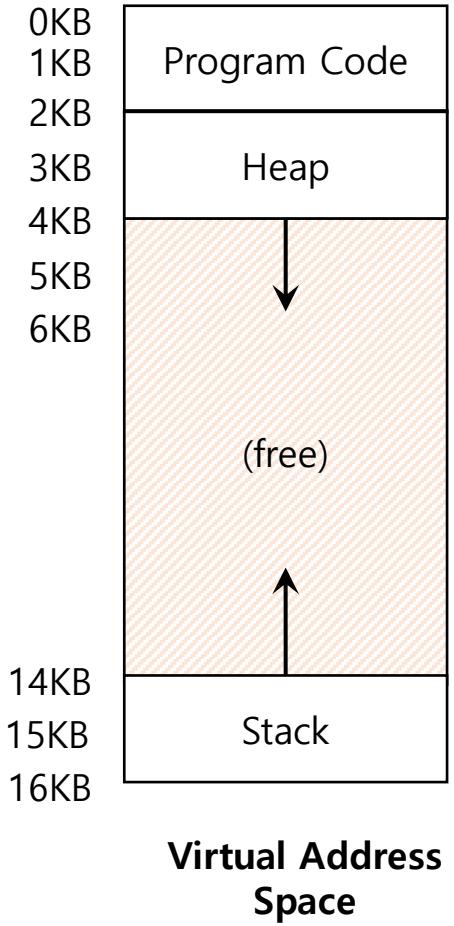
$$4MB = 2^{20}\ entries * 4B\ per\ page\ table\ entry$$

64-bit address space with 4-KB pages, 52 bits for VPN

$$2^{52}\ entries * 4B\ per\ page\ table\ entry = Petabyte - scale\ !$$

True, but address space is often sparsely used

Problem?



Address space **sparsely used**

Access to unused portion will appear valid

Would rather have an error

Solution: Valid/Invalid Bit

Page table has length **2^p**

→ Page table does not cover the entire possible virtual address space, only the pages that the process has allocated.

Have valid bit in each page table entry

- Set to valid for used portions of address space
- Invalid for unused portions

→ This is the common approach

Main Memory Allocation with Paging

Logical address space: fixed size pages

Physical address space: fixed size frames

New process

- Find frames for all of process's pages

Easier problem ☺

- **Fixed size**

(Internal) Fragmentation in Paging

With paging

- Address space = multiple of page size

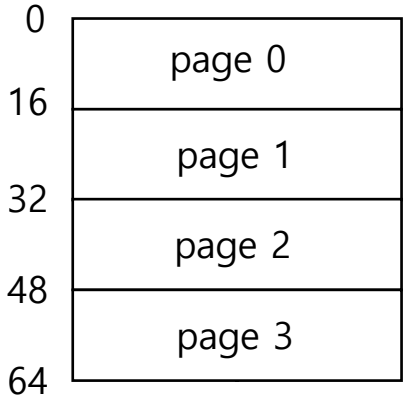
Part of last page may be unused

With reasonable page size, not a big problem

In Reality

- Base-and-bounds only for niche
- Segmentation abandoned
  - High complexity for little gain
  - Effect approximated with paging + valid bits
- **Paging is now universal**

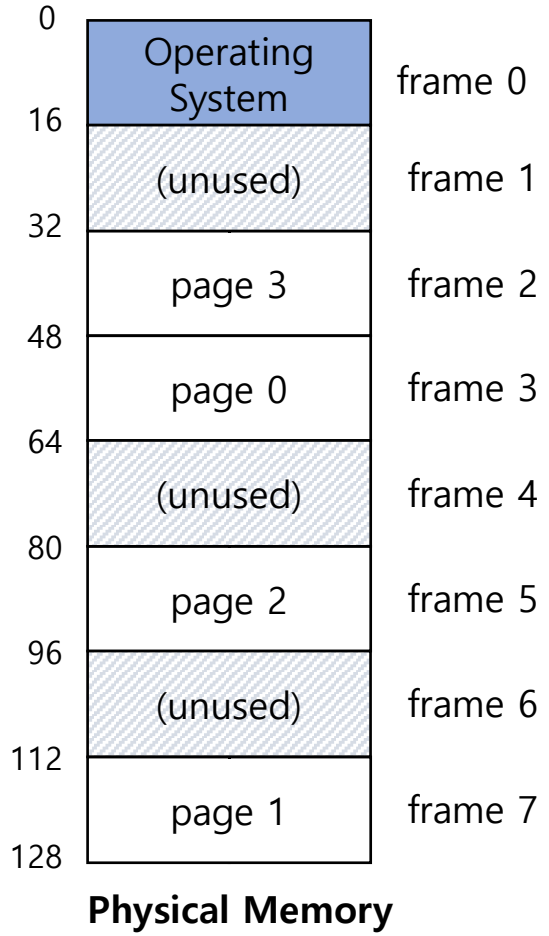
Paging



Virtual Address Space

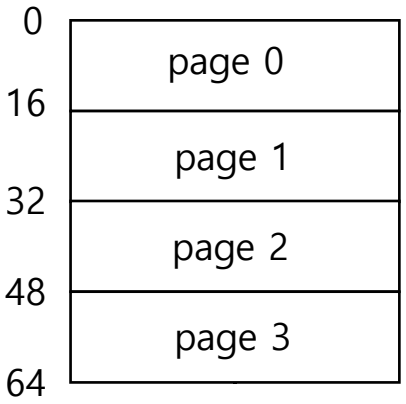
Page table

frame	valid/ invalid
3	v
7	v
5	v
2	v



Physical Memory

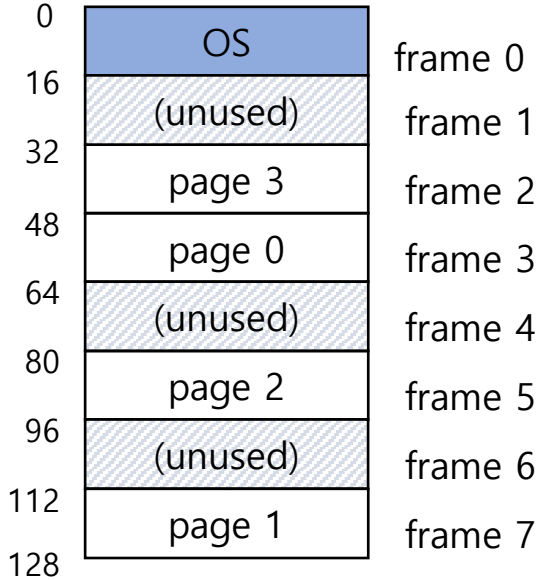
Problem: Paging Address Translation Performance



Virtual Address Space

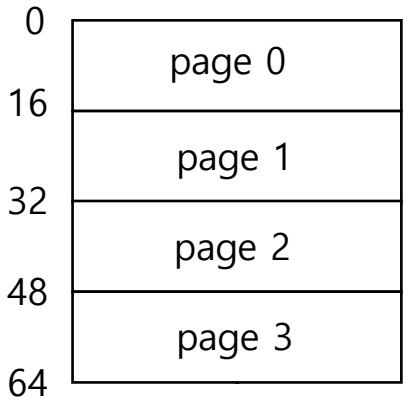
Page table

frame	valid/ invalid
3	v
7	v
5	v
2	v



Physical Memory

Problem: Paging Address Translation Performance

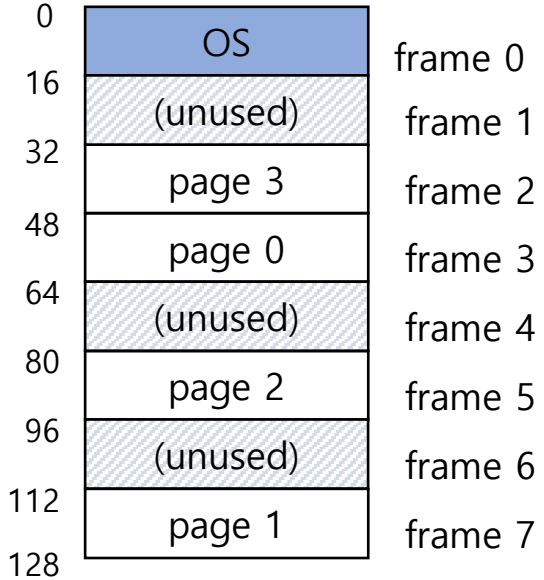


Virtual Address Space

Page table

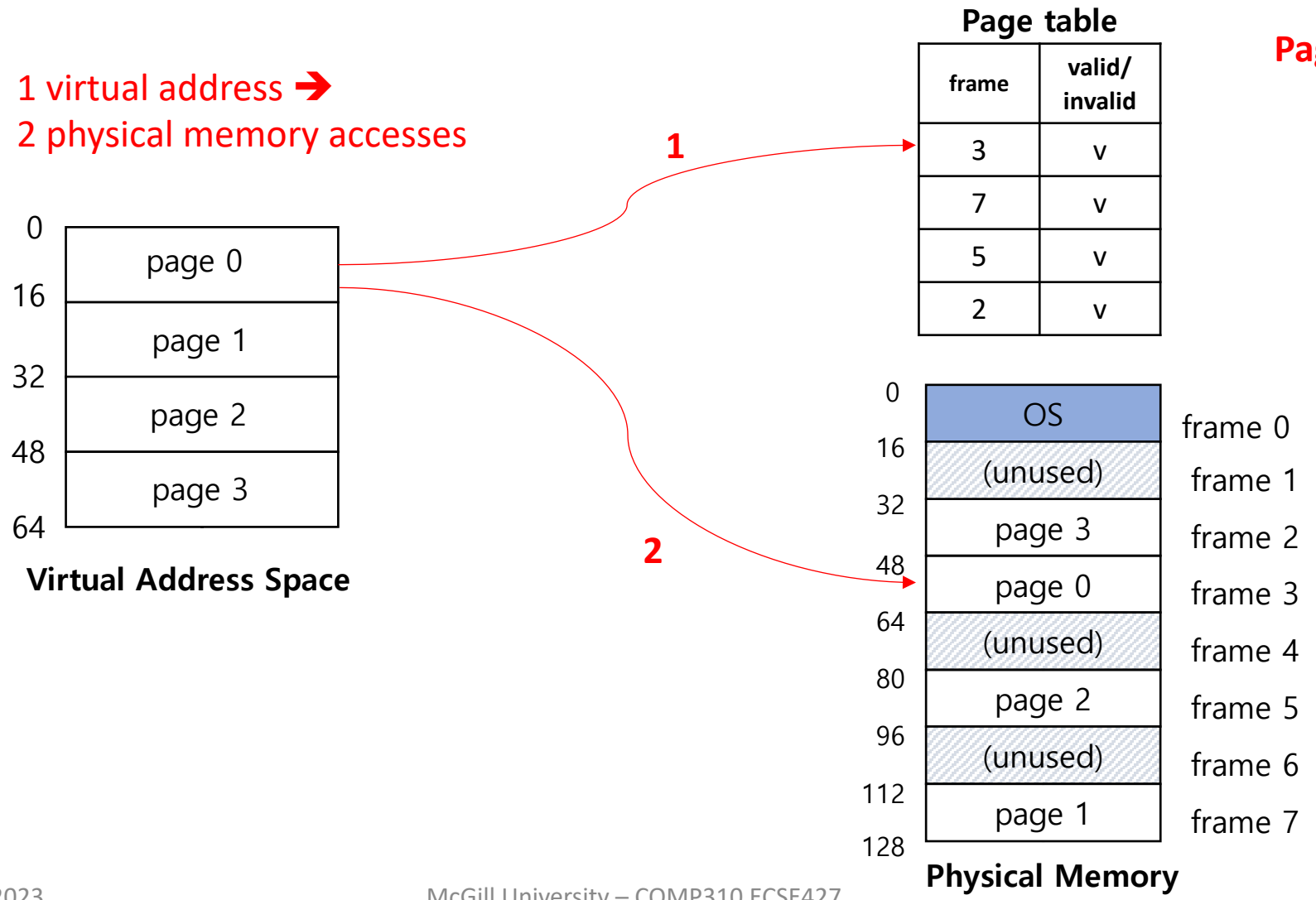
frame	valid/ invalid
3	v
7	v
5	v
2	v

Page table is in memory



Physical Memory

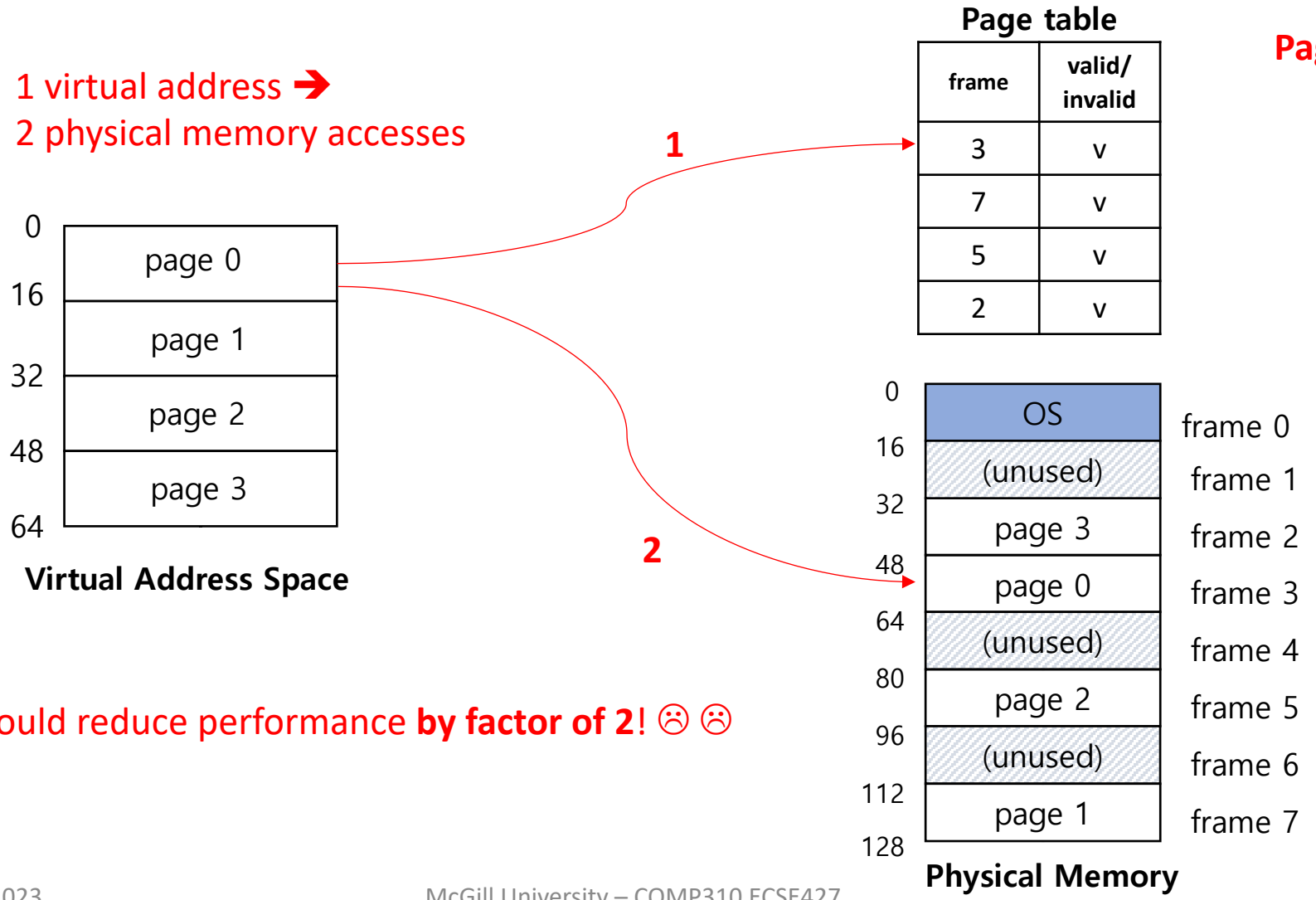
Problem: Paging Address Translation Performance



Page table is in memory

Problem: Paging Address Translation Performance

1 virtual address ➔
2 physical memory accesses



Page table is in memory

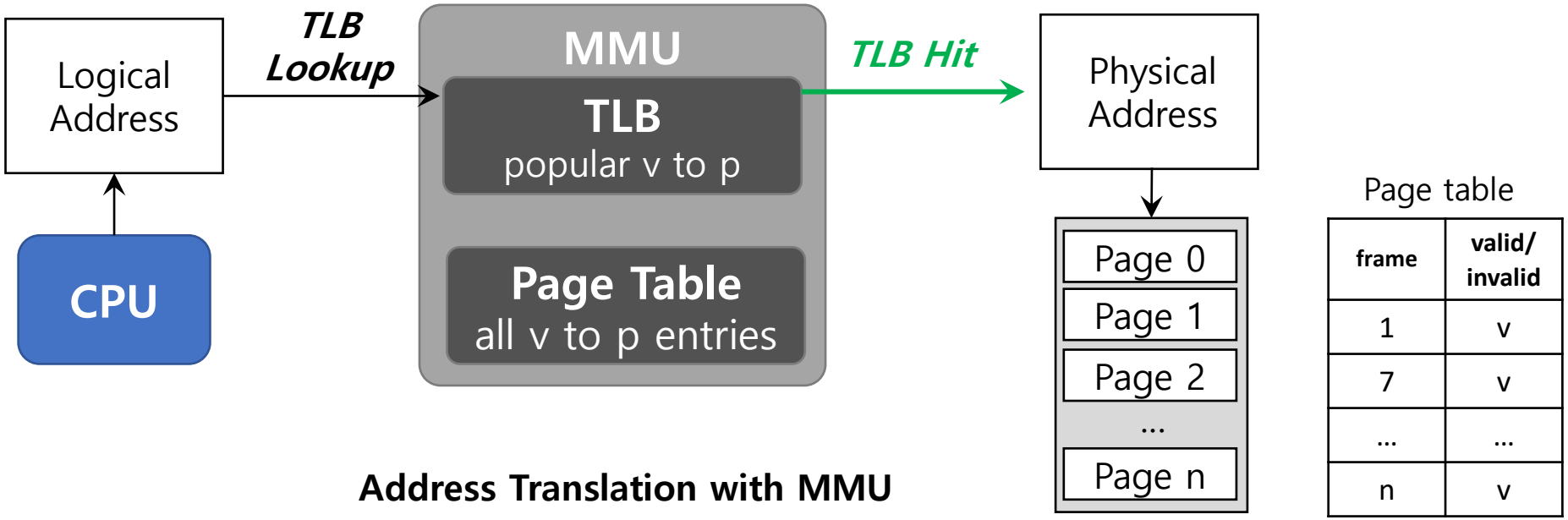
Would reduce performance **by factor of 2**! ☹ ☹

Solution: Translation Lookaside Buffer (TLB)

- Small fast **hardware cache** of **popular (pageno, frameno) maps**.
- **Part of MMU**
- If mapping for pageno found in TLB
  - Use frameno from TLB
  - Abort mapping using page table
- If not
  - Perform mapping using page table
  - Insert (pageno, frameno) in TLB

TLB Hit

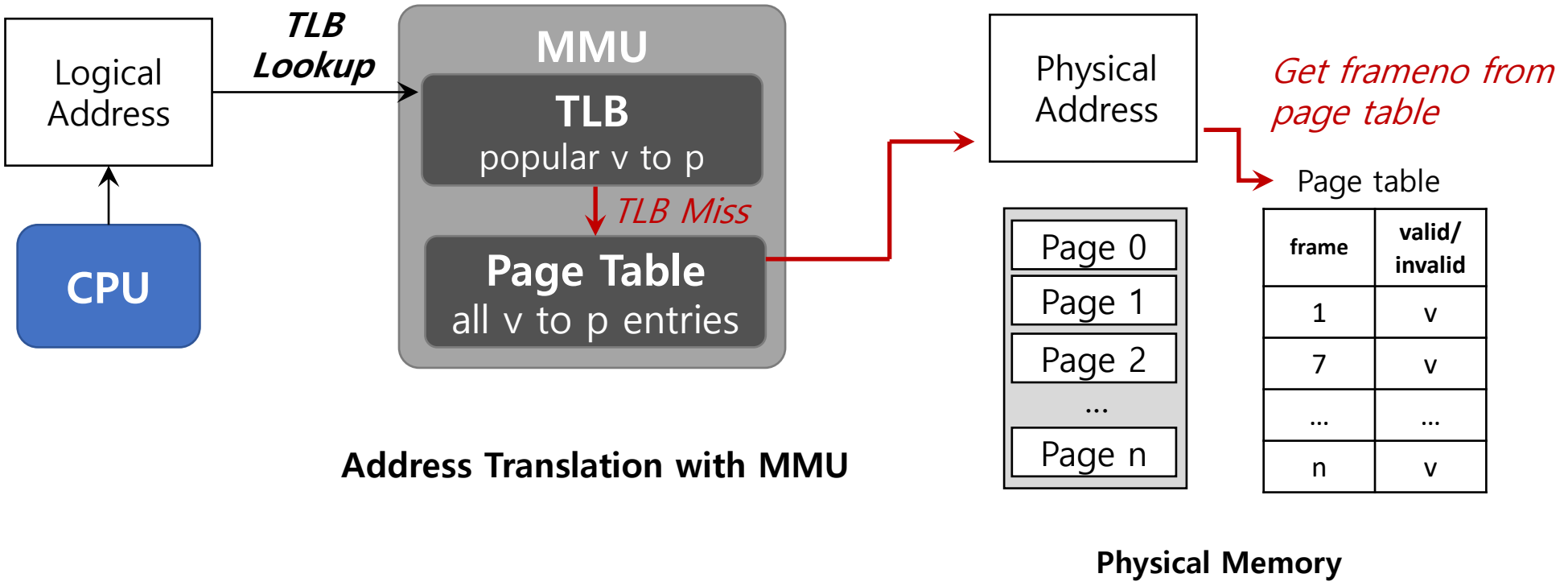
A single memory access is done if TLB hit! ☺



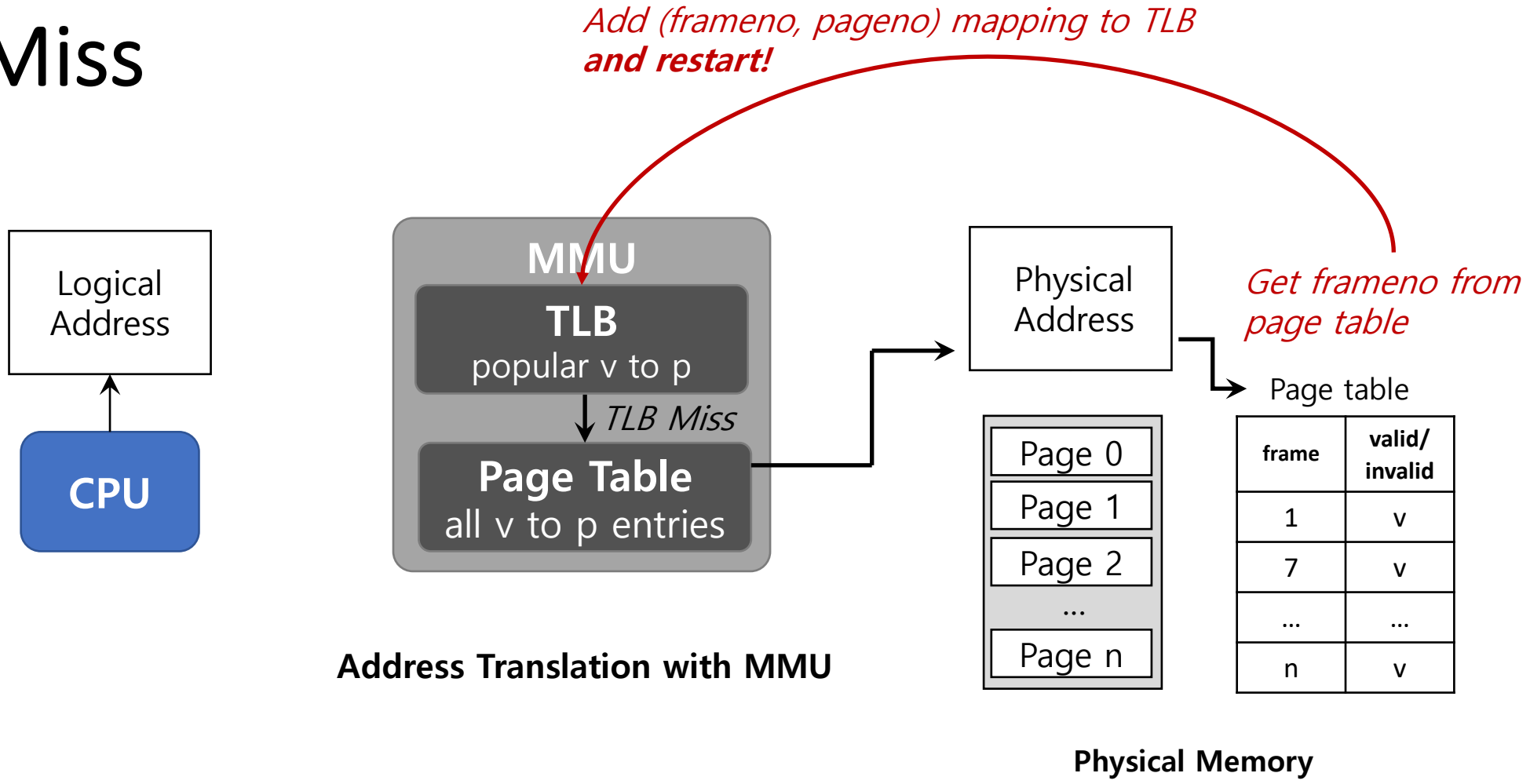
Address Translation with MMU

Physical Memory

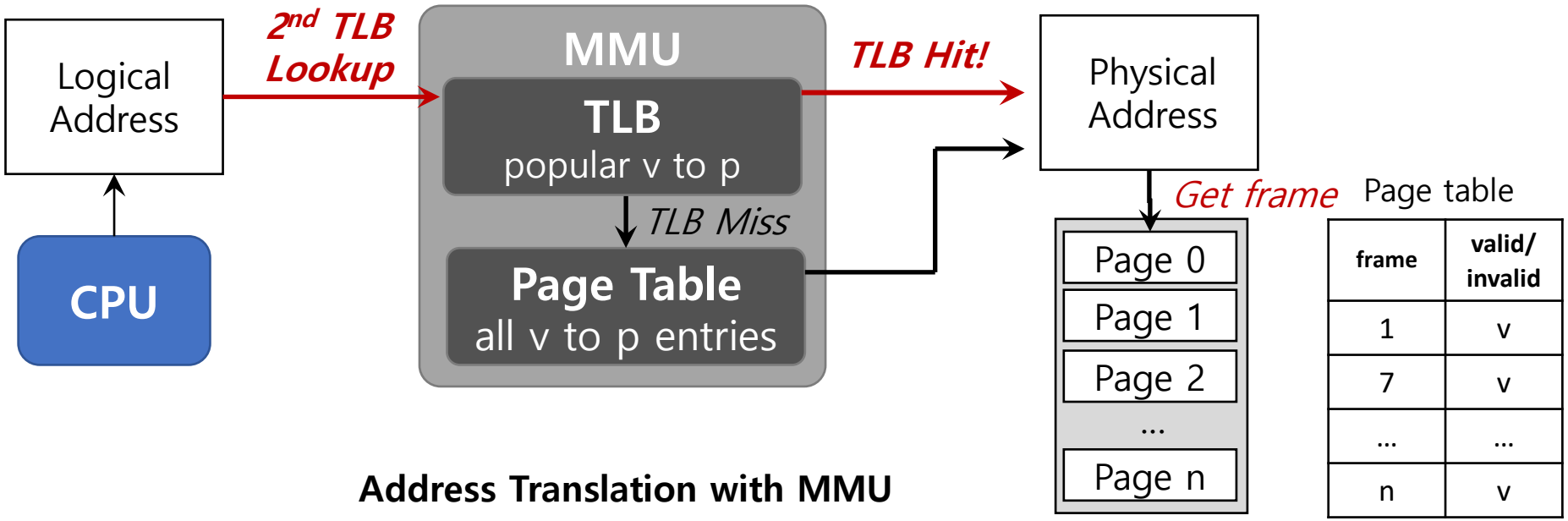
TLB Miss


Logical Address
CPU
TLB Lookup
MMU
TLB
popular v to p
TLB Miss
Page Table
all v to p entries
Physical Address
Get frameno from page table
Page table
frame
valid/ invalid
1
v
7
v
...
...
n
v
Page 0
Page 1
Page 2
...
Page n
Address Translation with MMU
Physical Memory

TLB Miss


Add (frameno, pageno) mapping to TLB
and restart!
Logical Address
CPU
MMU
TLB
popular v to p
TLB Miss
Page Table
all v to p entries
Physical Address
Get frameno from page table
Page table
frame
valid/ invalid
1
v
7
v
...
...
n
v
Page 0
Page 1
Page 2
...
Page n
Address Translation with MMU
Physical Memory

TLB Miss



Address Translation with MMU

Physical Memory

2 physical memory accesses done if TLB miss

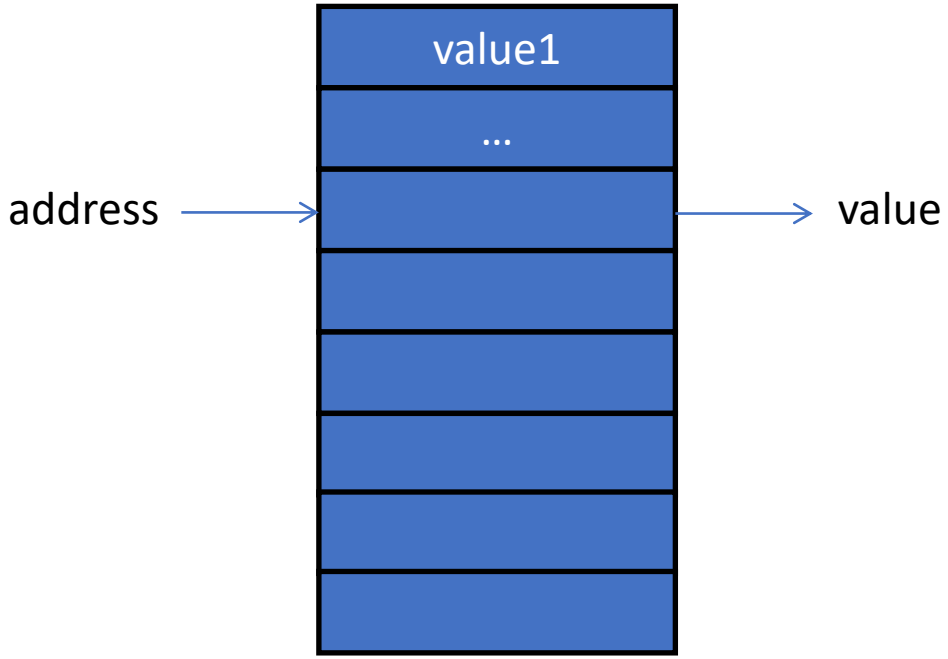
→ **Want high TLB hit rate!**

How to make TLB fast?

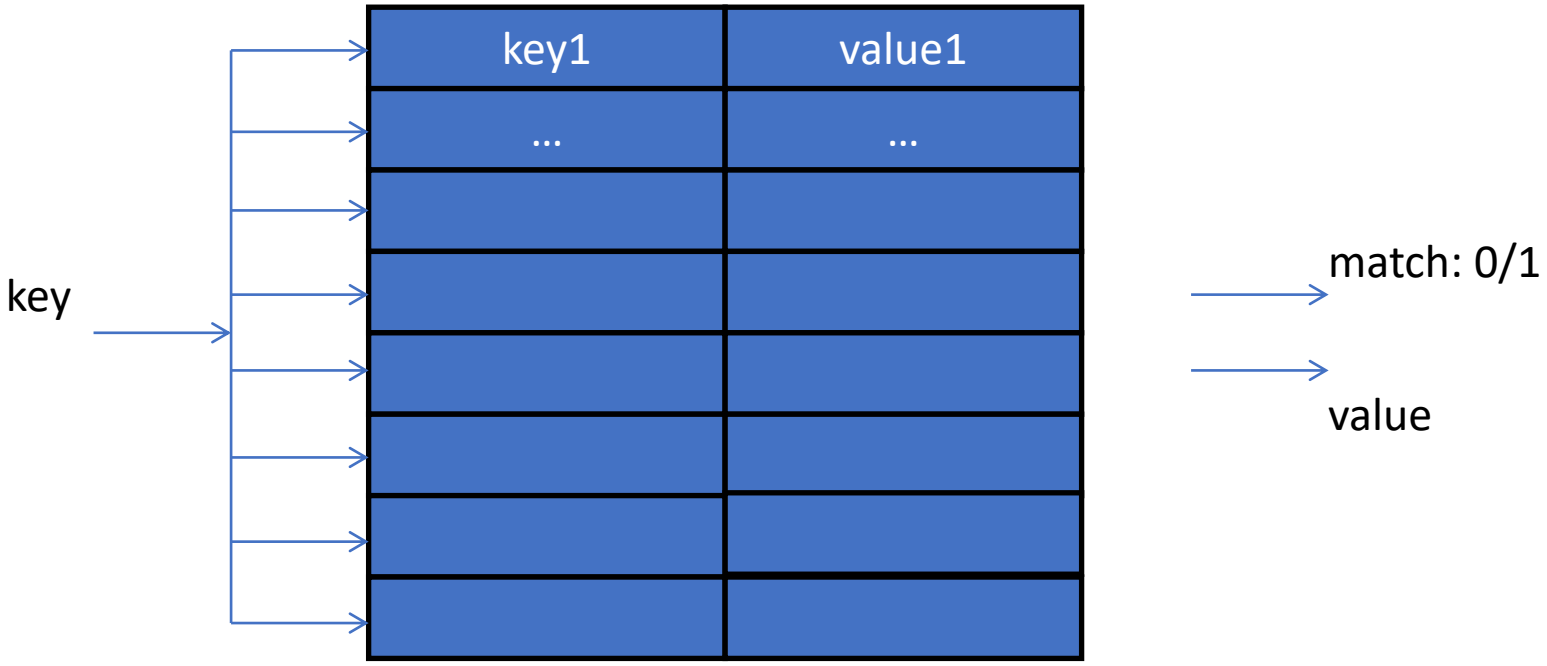
Use **associative memory** (special hardware)

- Regular memory
  Lookup by address
- Associative memory
  Lookup by contents
  Lookup in **parallel**

Regular Memory



Associative Memory



TLB

TLB Size

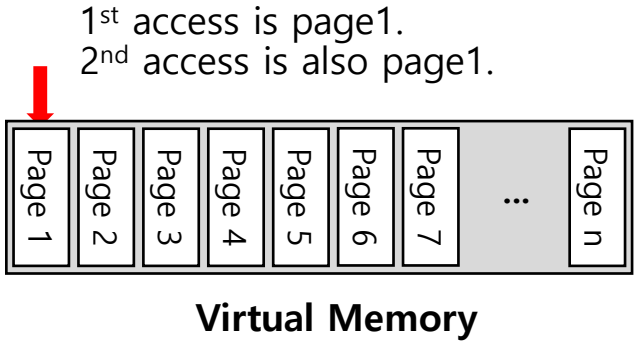
- Associative memory is very expensive
- Therefore, TLB small (64 – 1,024 entries)
- Want TLB hit-rate close to 100%
- If TLB full, **need to replace existing entry.** How?
  - Will discuss replacement policies later today.
  - Main idea: take advantage of locality.

TLB Hit Rate and Locality

Want TLB hit rate close to 100%. To do this, try to take advantage of locality:

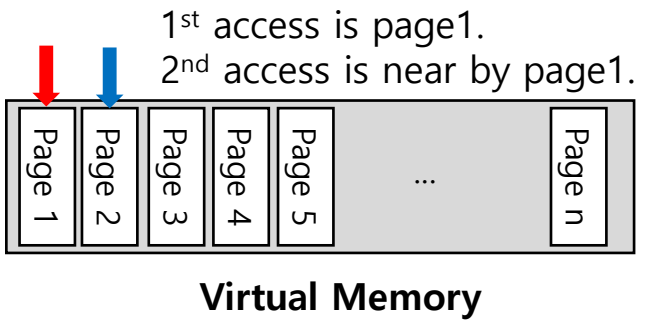
- **Temporal Locality**

  An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.

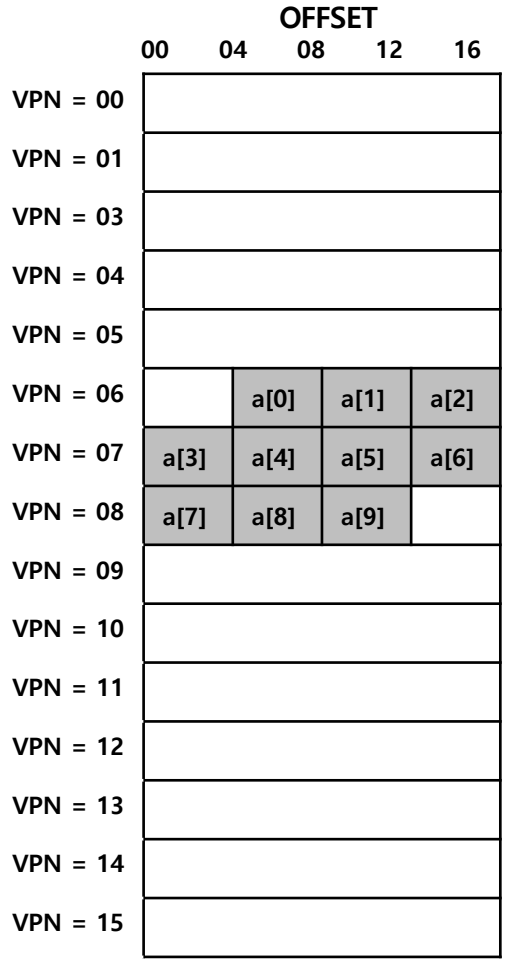


- **Spatial Locality**

  If a program accesses memory at address `x`, it will likely soon access memory near `x`.

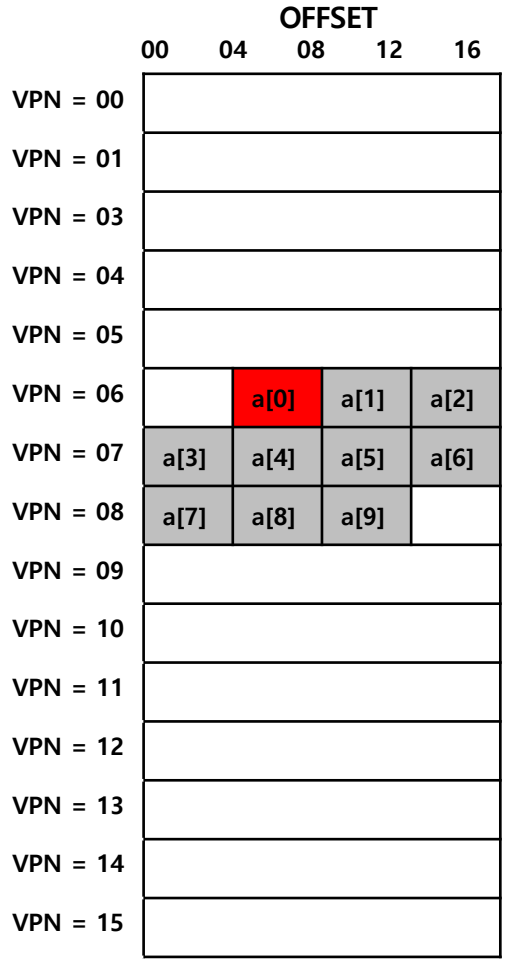


Spatial Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

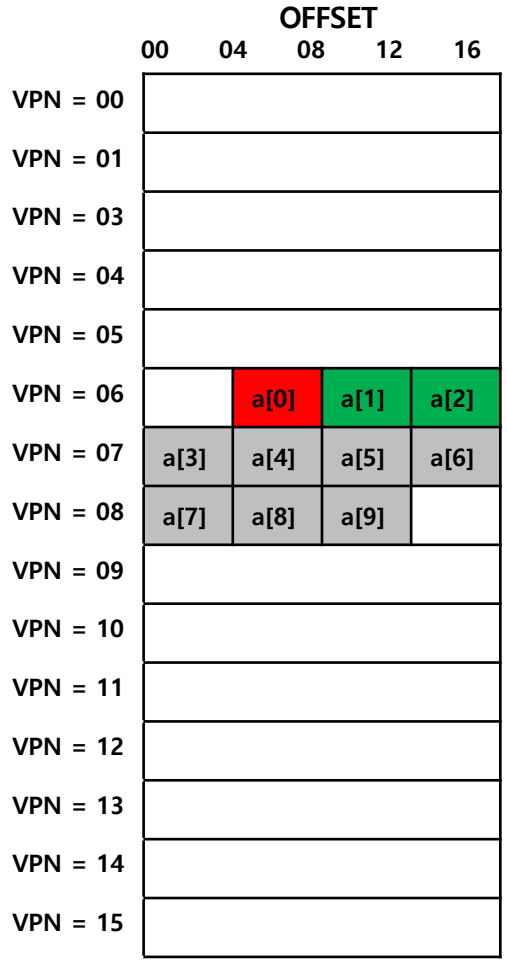
Spatial Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

TLB: miss

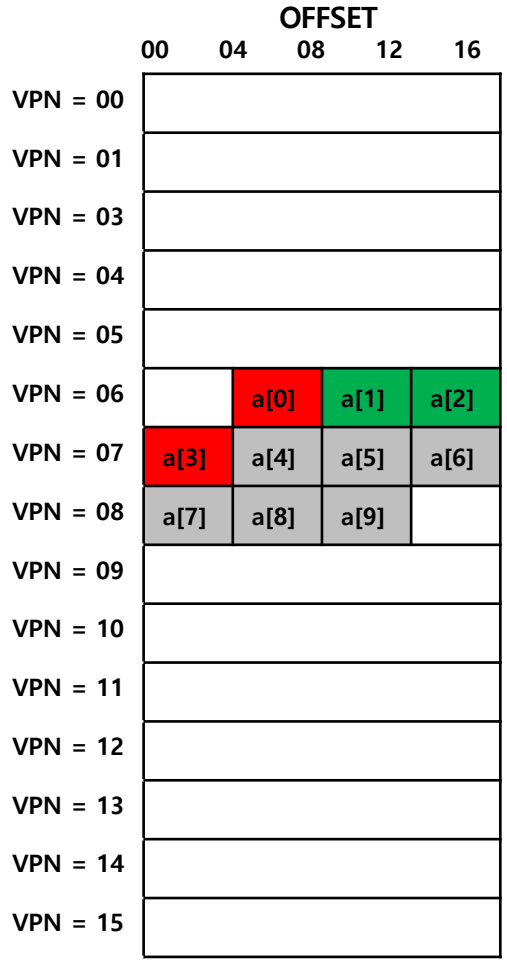
Spatial Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

TLB: miss, hit, hit

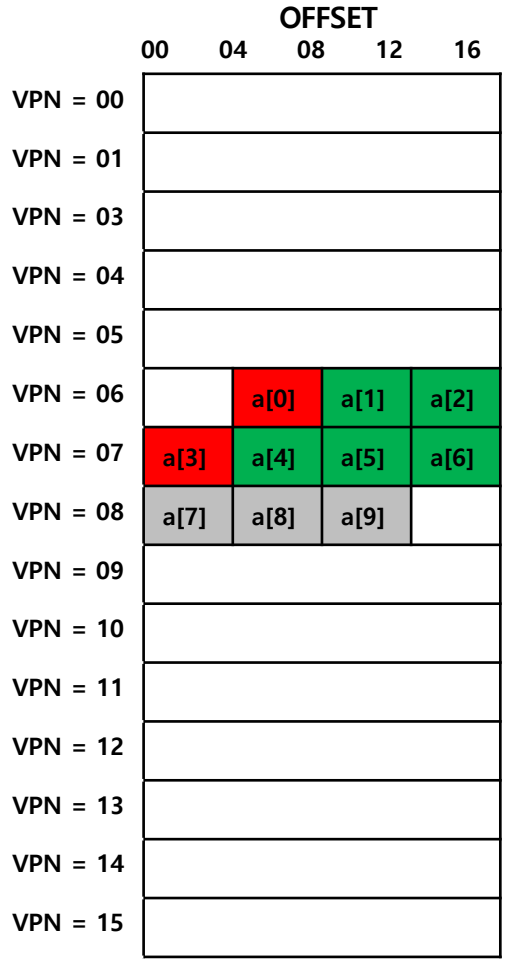
Spatial Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

TLB: miss, hit, hit, miss

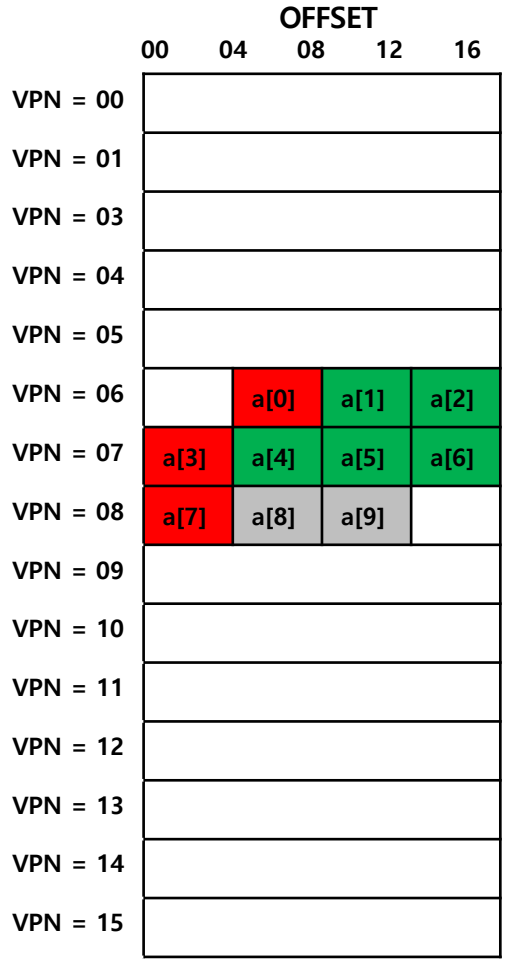
Spatial Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

TLB: miss, hit, hit, miss, hit, hit, hit

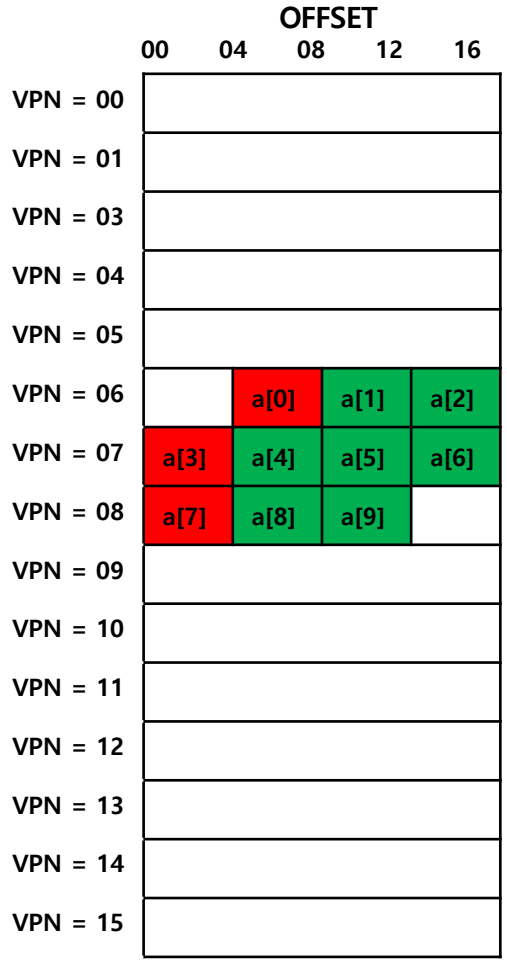
Spatial Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

TLB: miss, hit, hit, miss, hit, hit, hit
miss

Spatial Locality Example: Accessing an Array



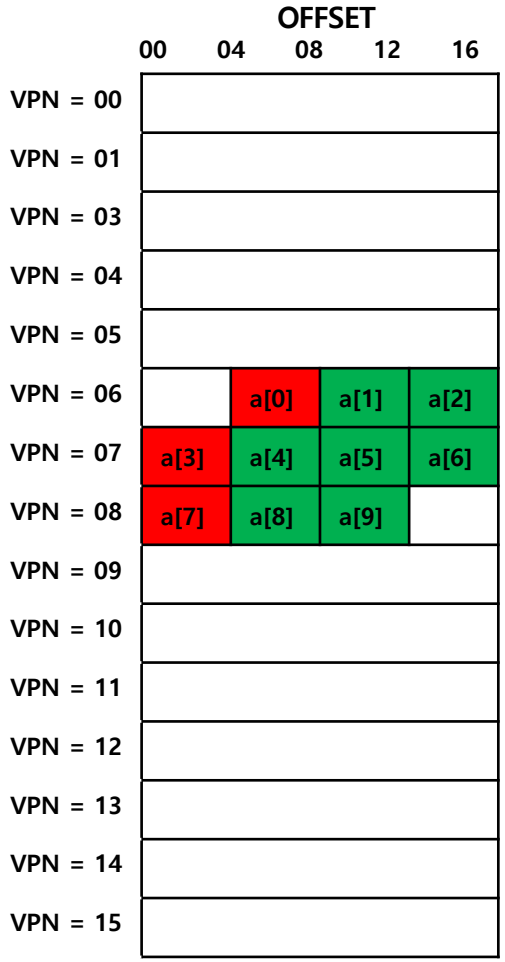

```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

TLB: miss, hit, hit, miss, hit, hit, hit
miss, hit, hit

3 misses and 7 hits.
Thus **TLB hit rate** is 70%.

The TLB improves performance due to spatial locality

Temporal Locality Example: Accessing an Array

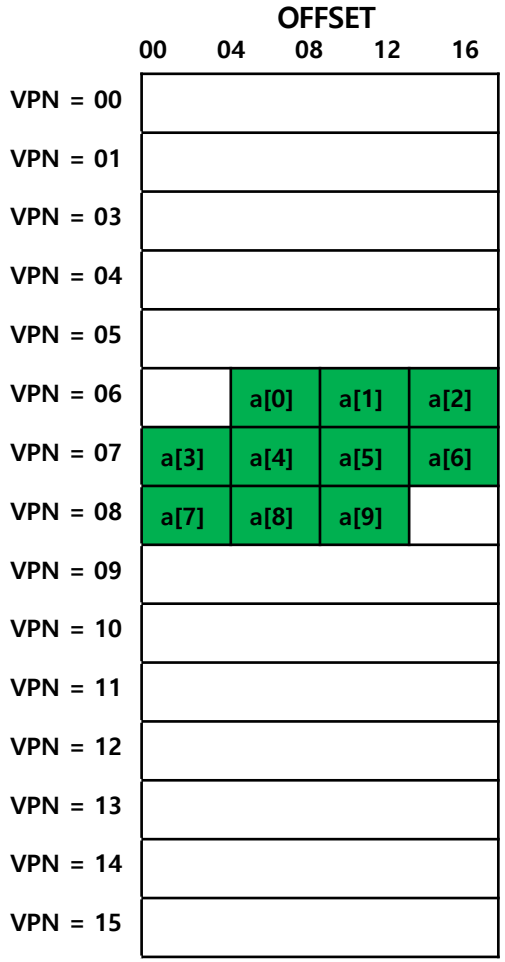



```
0:      int sum = 0 ;
1:      for( j=0; j<2; j++){
2:          for( i=0; i<10; i++)
3:              sum+=a[i];
4:      }
```

j=0 **TLB:** miss, hit, hit, miss, hit, hit, hit
miss, hit, hit

3 misses and 7 hits.
Thus **TLB hit rate** is 70%.

Temporal Locality Example: Accessing an Array




```
0:      int sum = 0 ;
1:      for( j=0; j<2; j++){
2:          for( i=0; i<10; i++)
3:              sum+=a[i];
4:      }
```

j=0 **TLB:** miss, hit, hit, miss, hit, hit, hit miss, hit, hit

3 misses and 7 hits.
Thus **TLB hit rate** is 70%.

j=1 **TLB:** hit, hit, hit, hit, hit, hit, hit hit, hit, hit

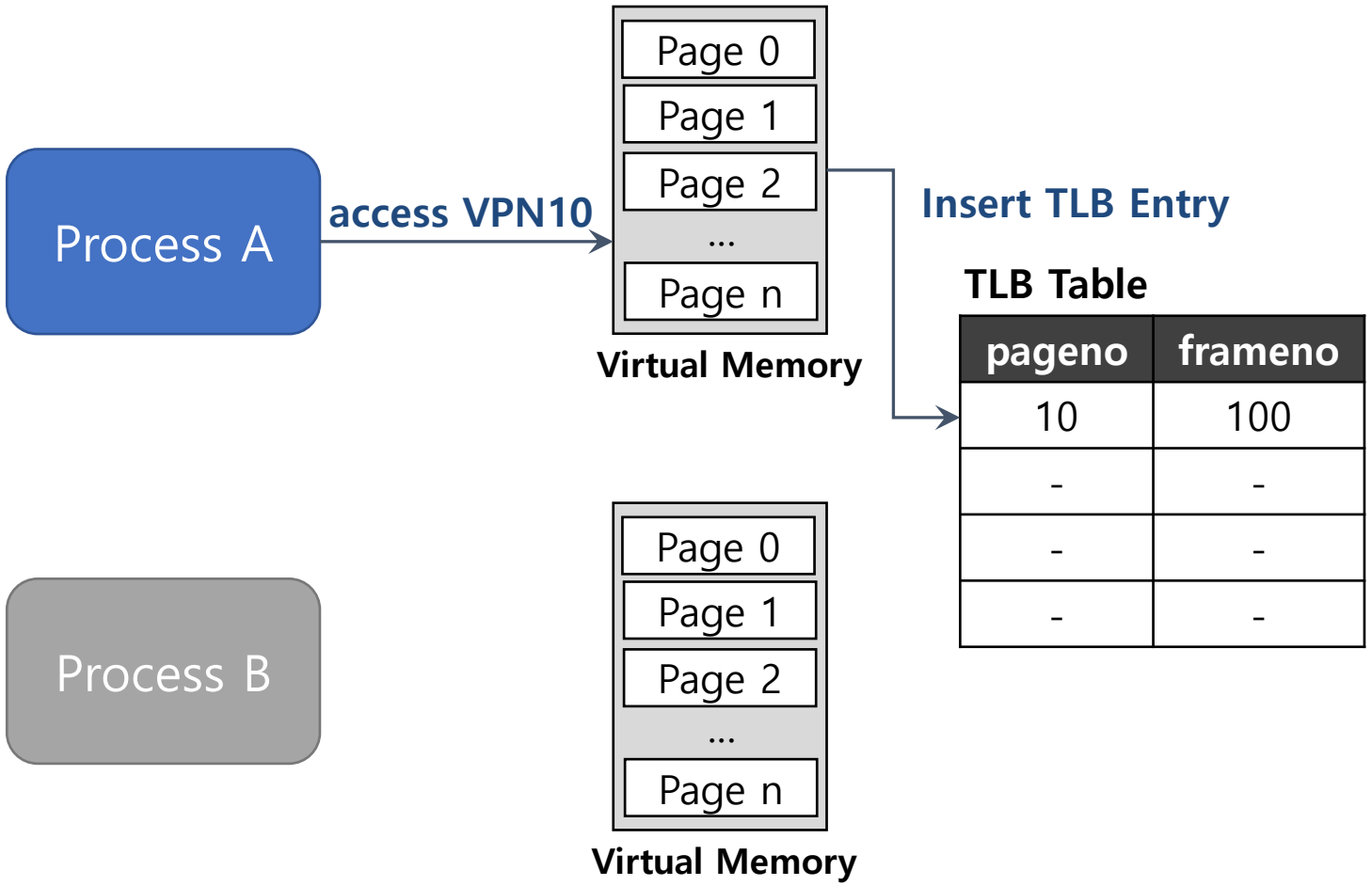
0 misses and 10 hits.
Thus **TLB hit rate** is **100%.**

The TLB improves performance due to temporal locality

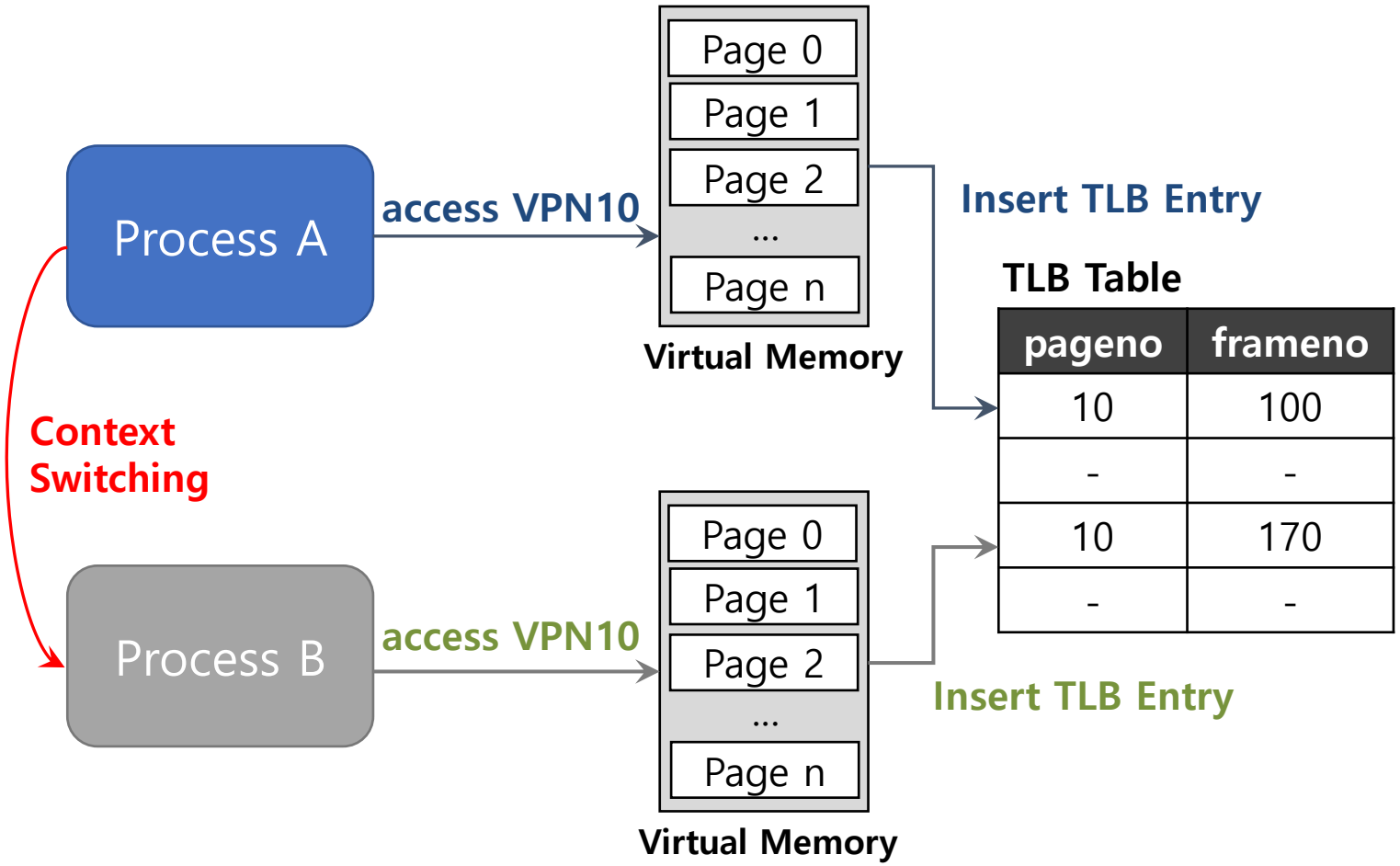
Revisiting Process Switching

- Suppose
  - Process P1 is running
  - Entry (**pageno**, frameno) in TLB
  - Switch from P1 to P2
  - P2 issues virtual address in page **pageno**
- P2 accesses P1's memory!
- Or unable to distinguish between P1 and P2 mapping

TLB Issue: Context Switching



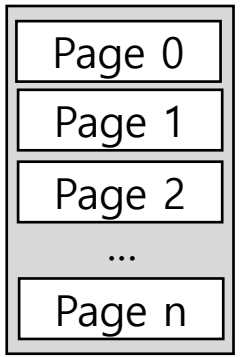
TLB Issue: Context Switching



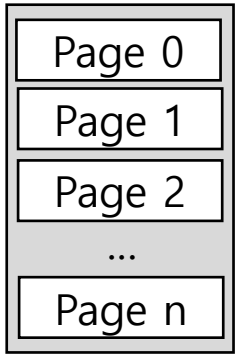
TLB Issue: Context Switching

Process A

Process B



Virtual Memory



Virtual Memory

Can't distinguish which entry is meant for which process

TLB Table

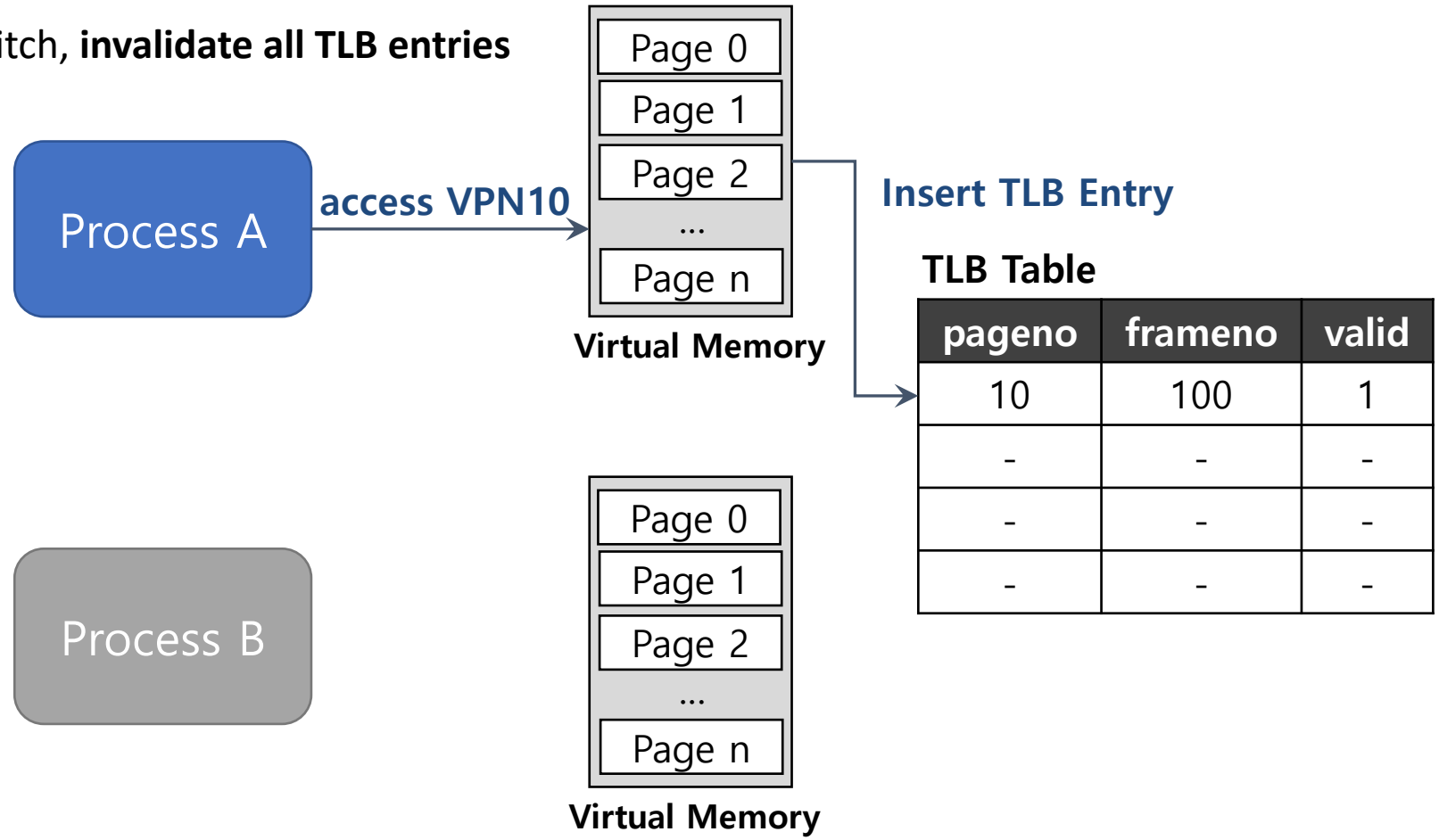
pageno	frameno
10	100
-	-
10	170
-	-

TLB Issue: Context Switching – Solution 1

On process switch, **invalidate all TLB entries**

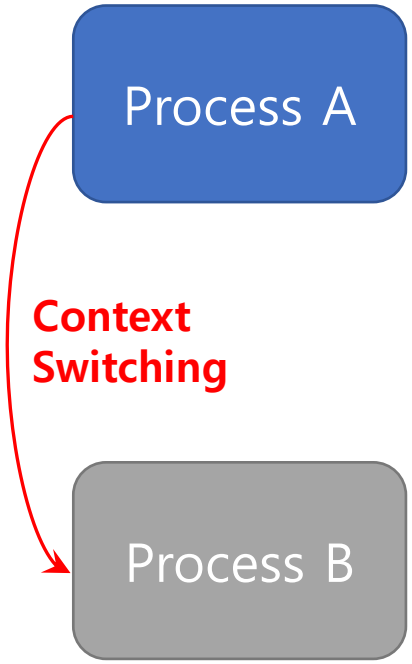
TLB Issue: Context Switching – Solution 1

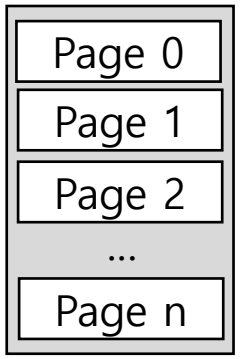
On process switch, **invalidate all TLB entries**



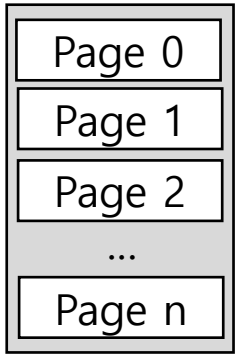
TLB Issue: Context Switching – Solution 1

On process switch, **invalidate all TLB entries**





Virtual Memory



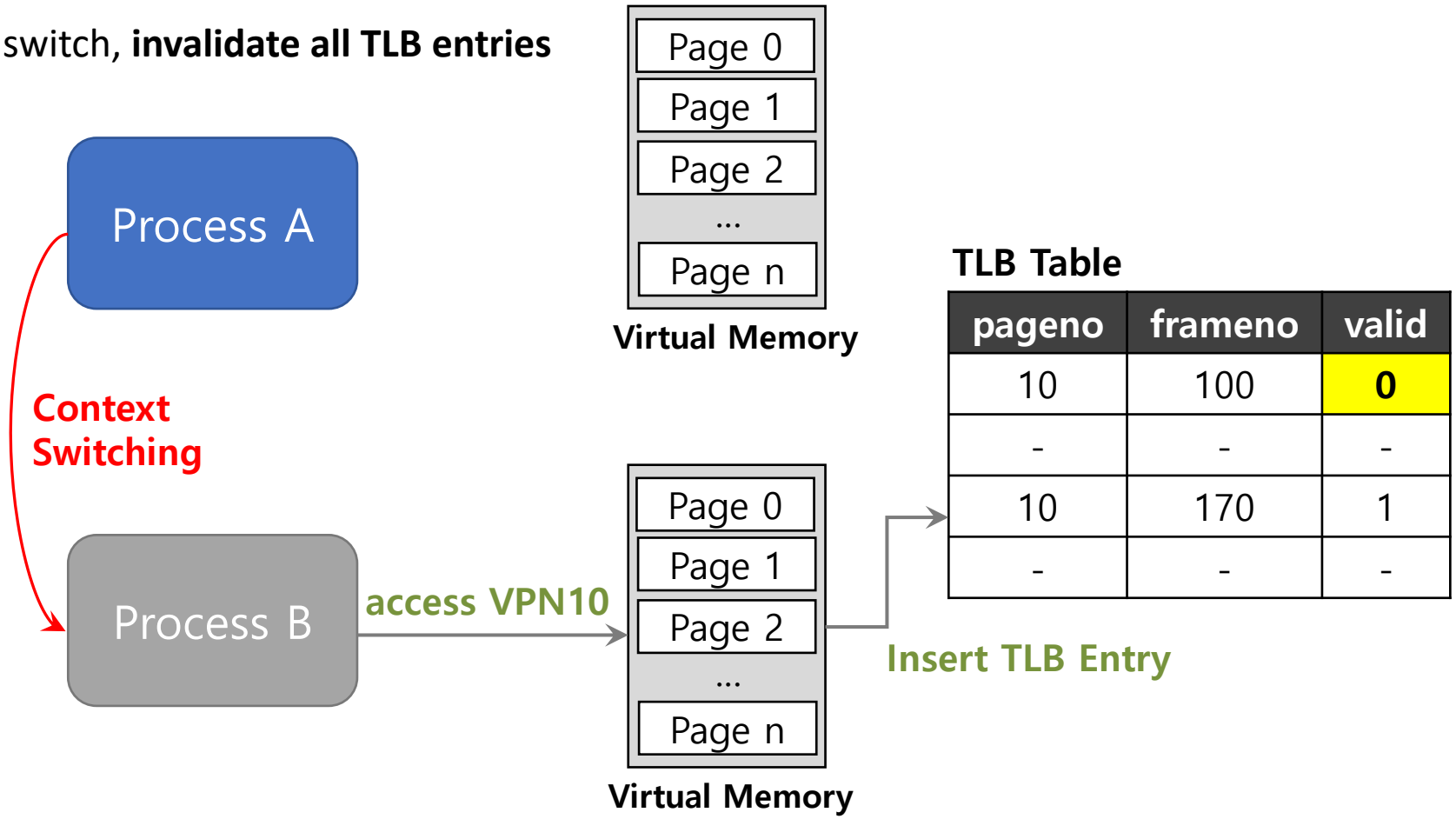
Virtual Memory

TLB Table

pageno	frameno	valid
10	100	**0**
-	-	-
-	-	-
-	-	-

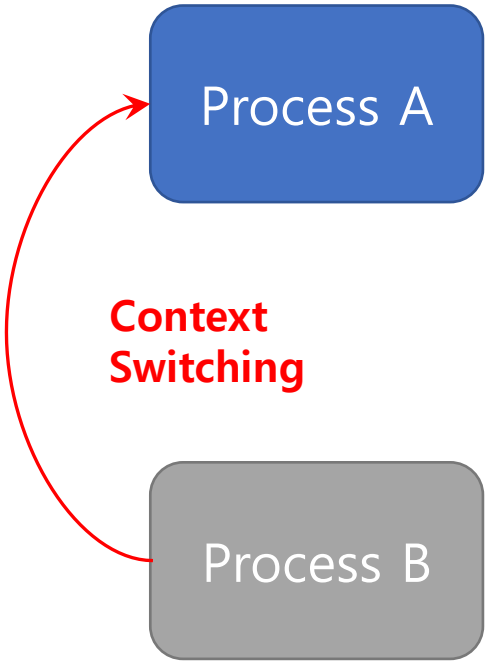
TLB Issue: Context Switching – Solution 1

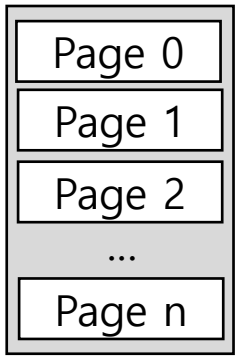
On process switch, **invalidate all TLB entries**



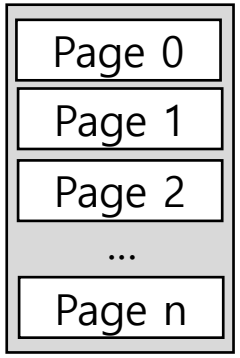
TLB Issue: Context Switching – Solution 1

On process switch, **invalidate all TLB entries**





Virtual Memory



Virtual Memory

+ Simply requires invalid bit in each TLB entry

☹ Makes process switch expensive

☹ ☹ New process initially incurs 100% TLB misses

TLB Table

pageno	frameno	valid
10	100	**0**
-	-	-
10	170	**0**
-	-	-

TLB Issue: Context Switching – Solution 2

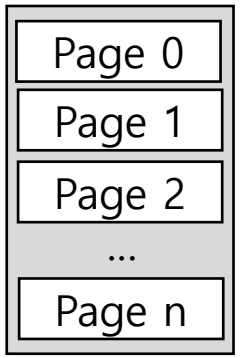
- Add **process identifier** to TLB entries
  - **Match = match on pid AND match on pageno**
  - ☹ Makes TLB more complicated and expensive
- Process switch
  - ☺ Nothing to do
  - ☺☺ Cheaper
- All modern machines have this feature

TLB Issue: Context Switching – Solution 2

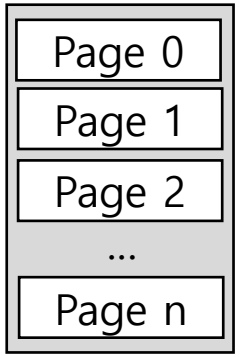
TLB **match on pid AND match on pageno**

Process A

Process B



Virtual Memory



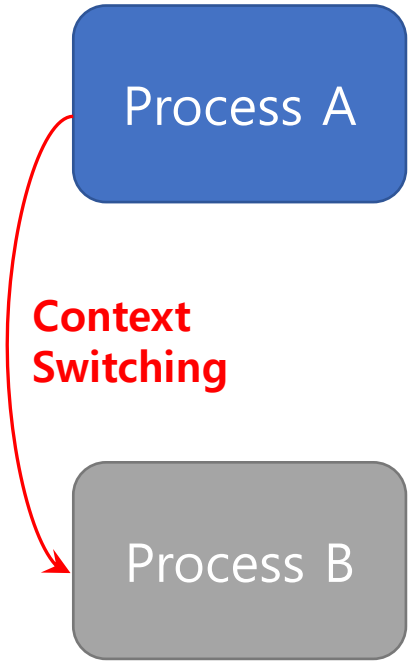
Virtual Memory

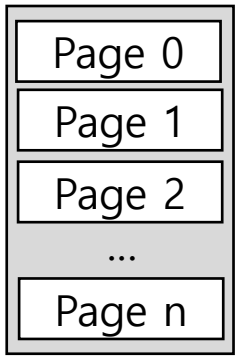
TLB Table

pageno	frameno	valid	PID
10	100	1	**A**
-	-	-	
10	170	1	**B**
-	-	-	

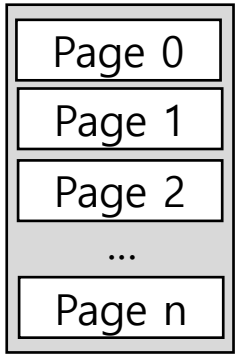
TLB Issue: Context Switching – Solution 2

TLB **match on pid AND match on pageno**





Virtual Memory



Virtual Memory

TLB Table

pageno	frameno	valid	PID
10	100	1	**A**
-	-	-	
10	170	1	**B**
-	-	-	

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 19–22

https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).