

Week 7

Memory Management: Demand Paging

Max Kopinsky

February 17, 2025

Announcements

- Assignment 2 is out.
- Graded exercise sheet released end of this week
 - At the moment, not predicting a delay.
 - However, if it's delayed, due date will be extended.

Reality of Virtual Memory

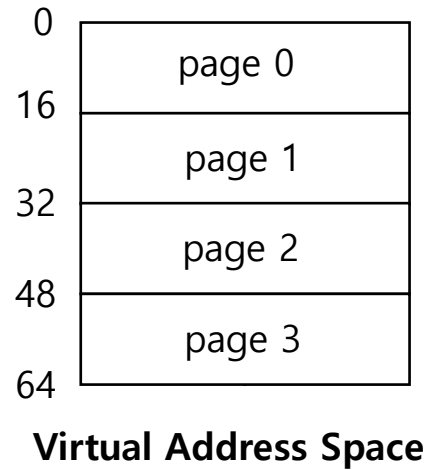
- Base-and-bounds only for niche
- Segmentation abandoned
 - High complexity for little gain
 - Effect approximated with paging + valid bits
- **Paging is now universal**
 - Paging creates some problems; let's see them and how we fix them.

Key Concepts

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

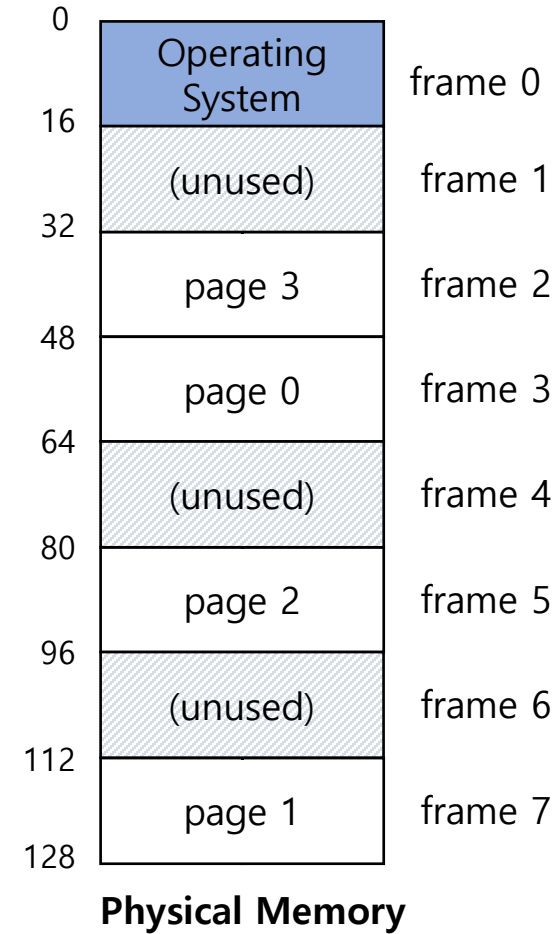
Part 1: Translation Lookaside Buffers (TLBs)

Paging

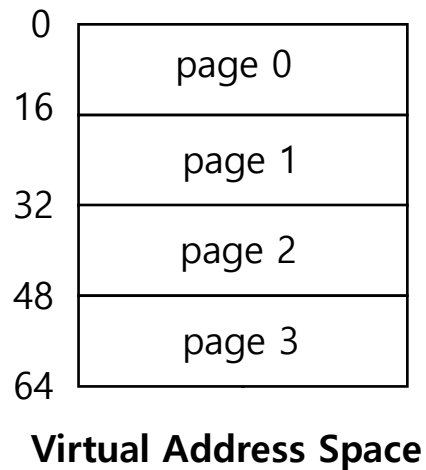


Page table

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

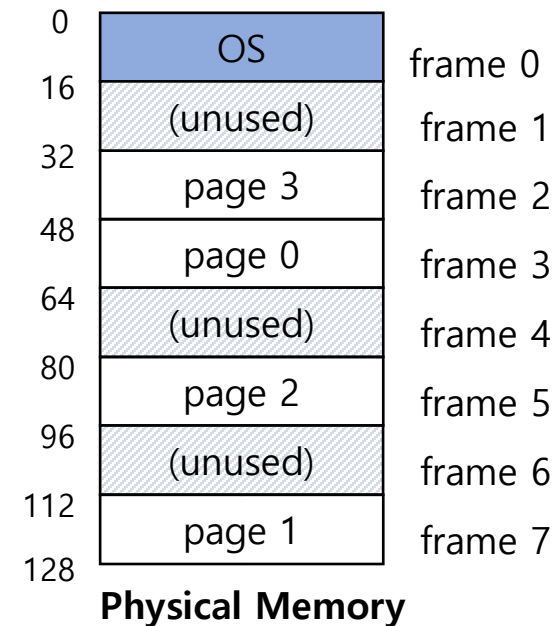


Problem: Paging Address Translation Performance

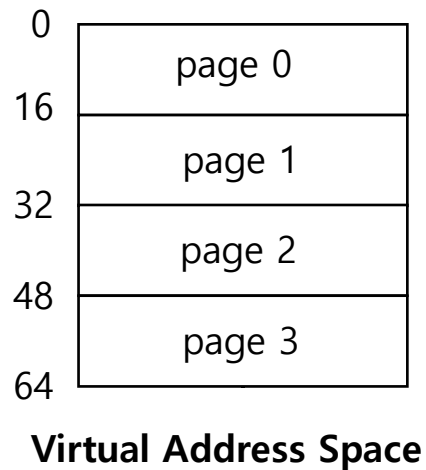


Page table

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



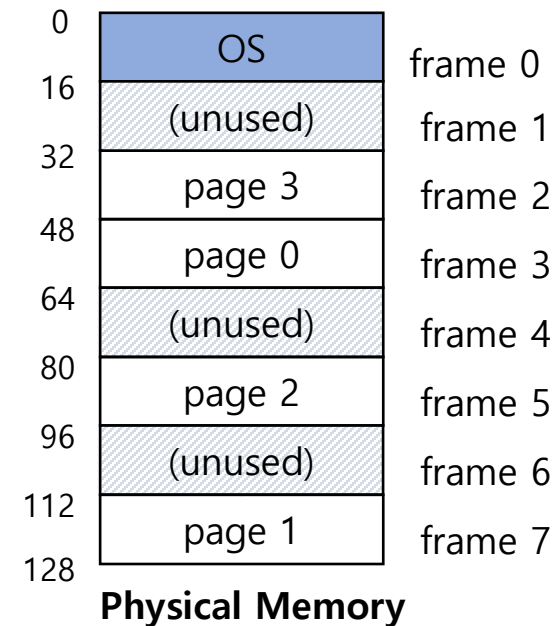
Problem: Paging Address Translation Performance



Page table

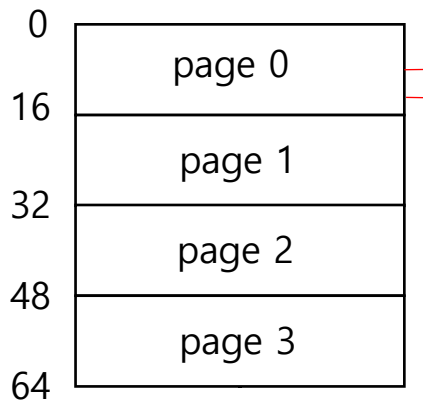
frame	valid/ invalid
3	v
7	v
5	v
2	v

Page table is in memory



Problem: Paging Address Translation Performance

1 virtual address →
2 physical memory accesses

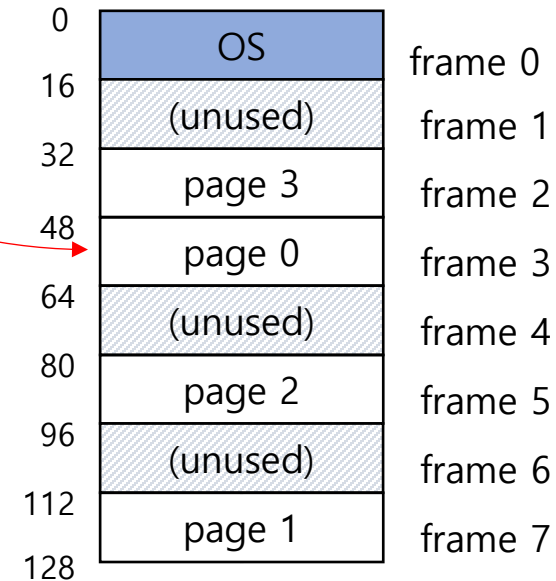


Virtual Address Space

Page table

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

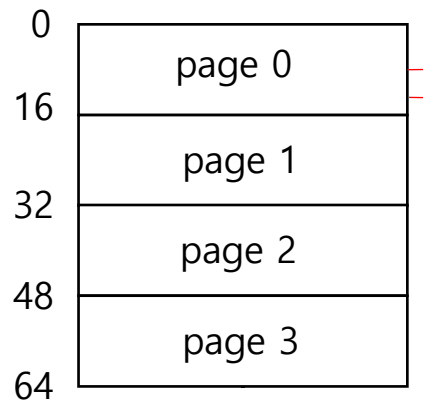
Page table is in memory



Physical Memory

Problem: Paging Address Translation Performance

1 virtual address →
2 physical memory accesses



Virtual Address Space

Would reduce performance by factor of 2! ☹️ ☹️
(memory speeds dominate most workloads)

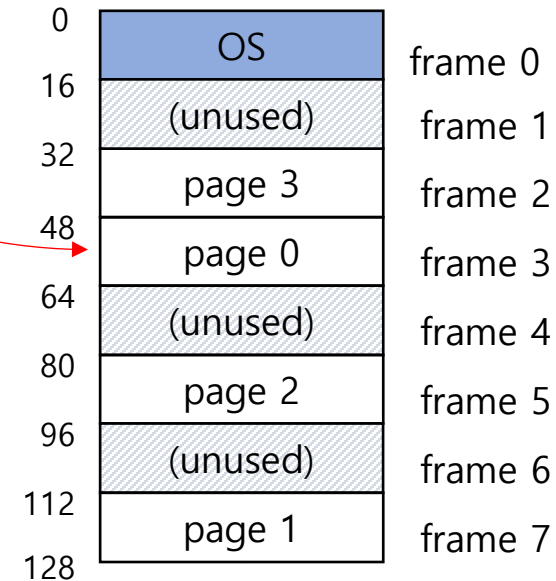
Page table

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

Page table is in memory

1

2

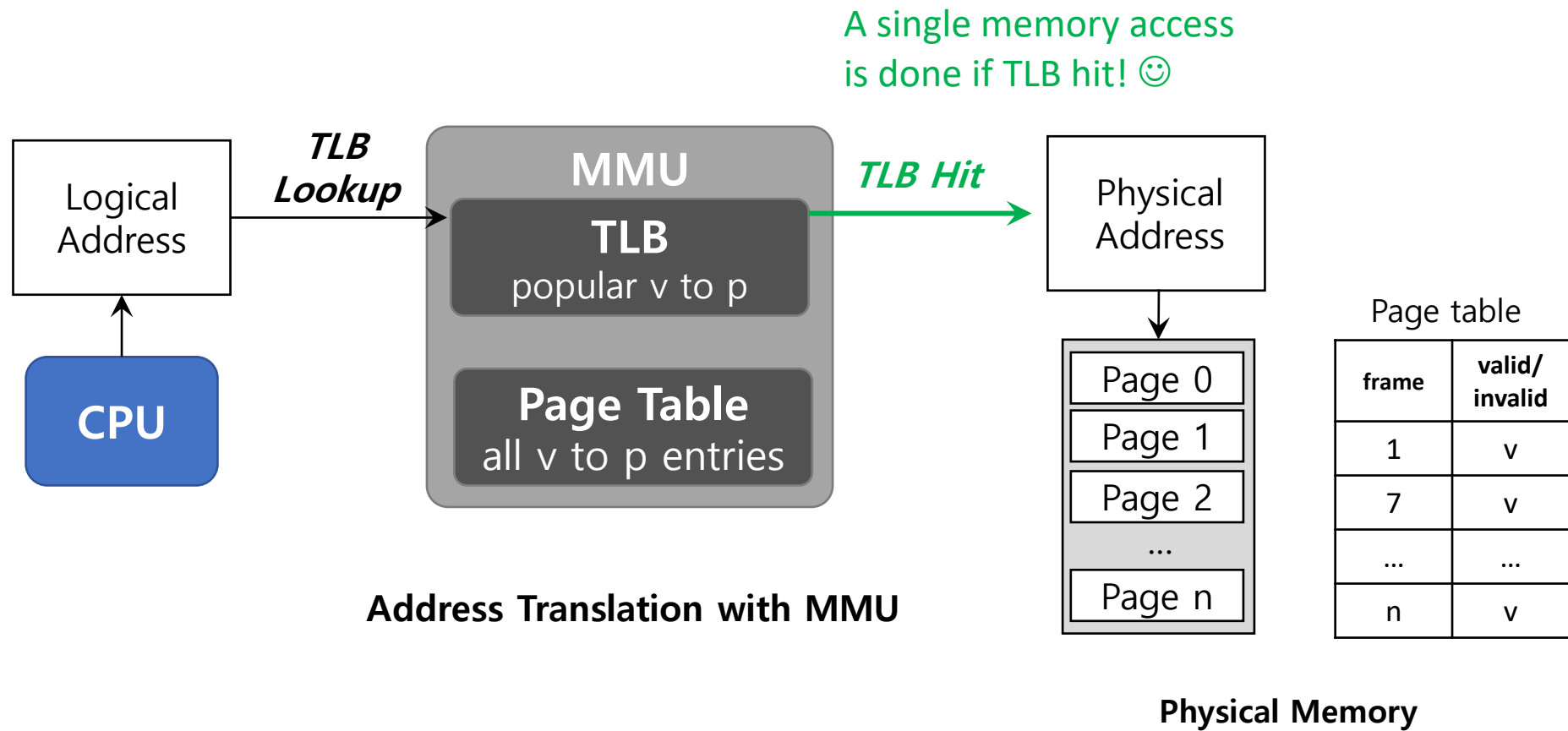


Physical Memory

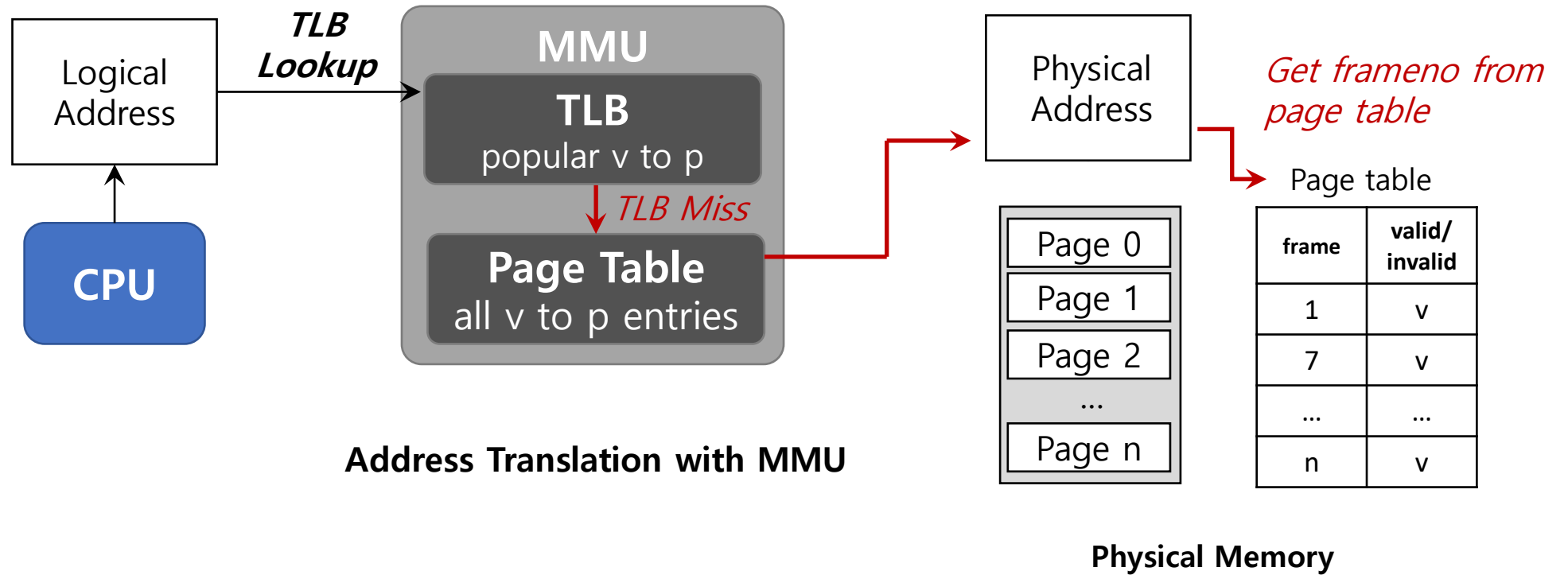
Solution: Translation Lookaside Buffer (TLB)

- 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
 - **Choice: Who performs the mapping?**
 - MMU in all major modern systems

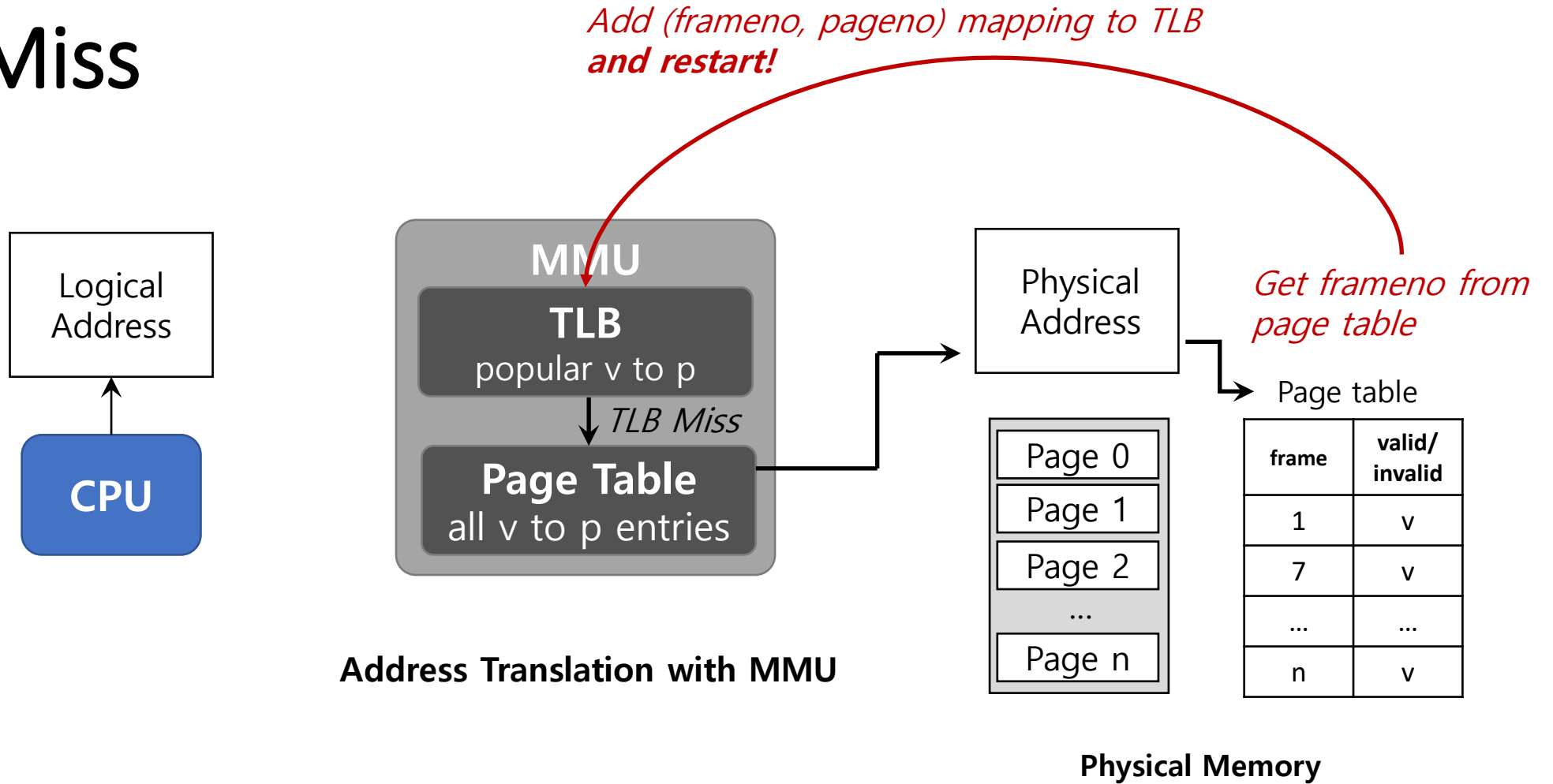
TLB Hit



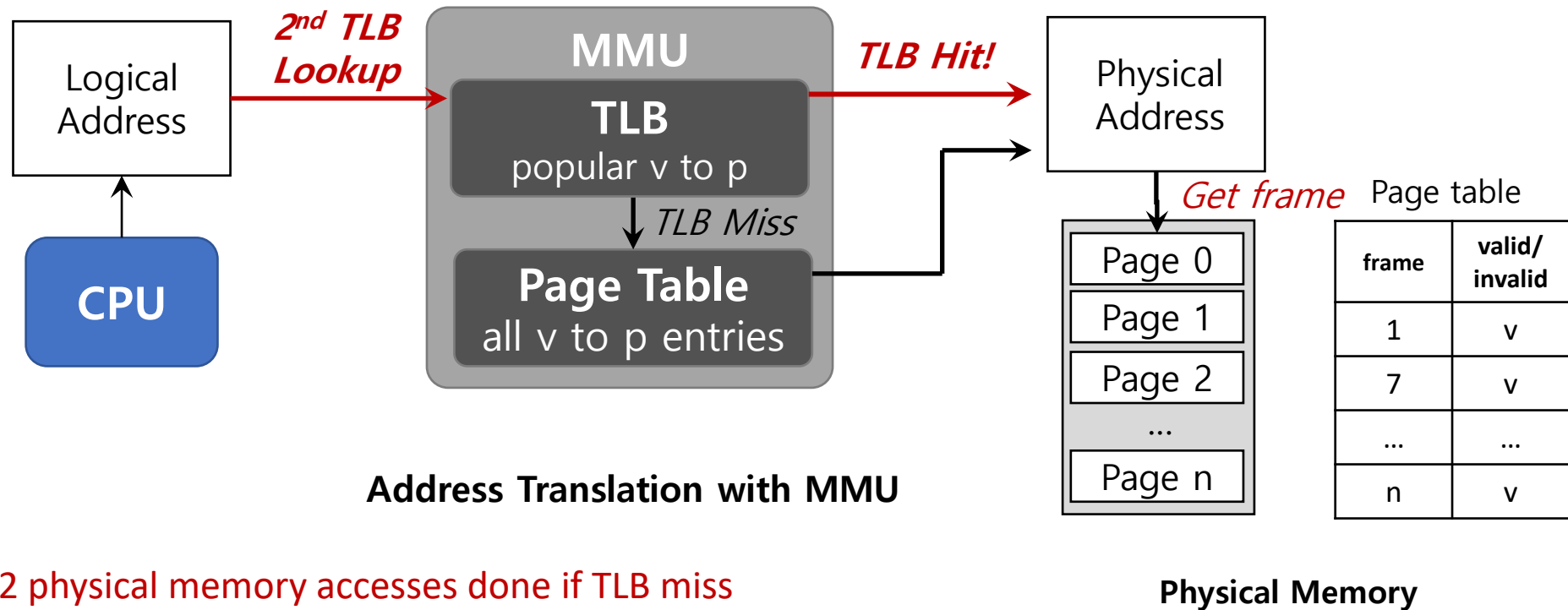
TLB Miss



TLB Miss



TLB Miss



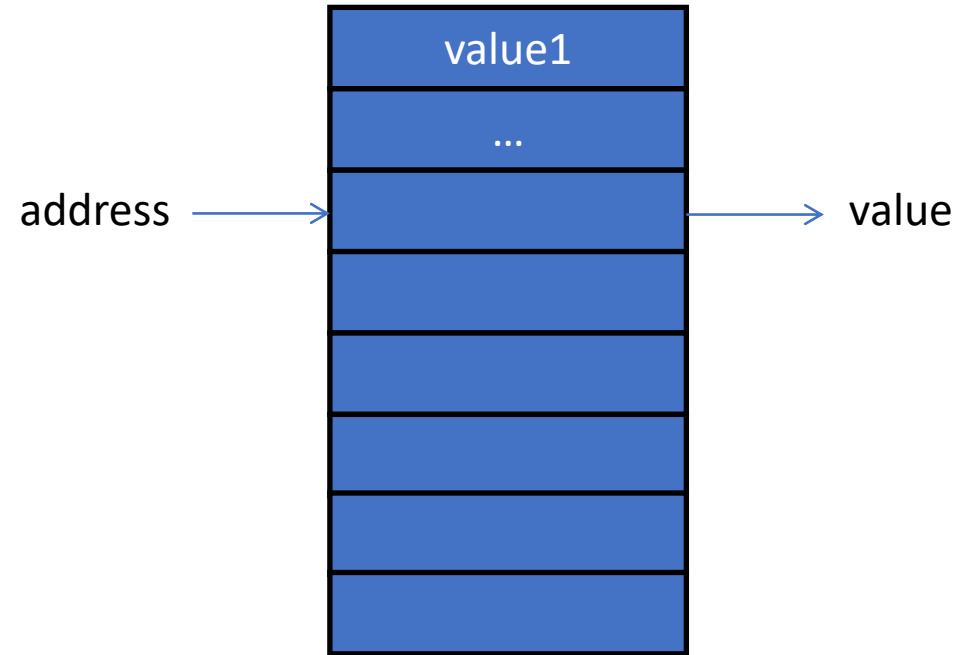
2 physical memory accesses done if TLB miss
→ Want high TLB hit rate!

How to make TLB fast?

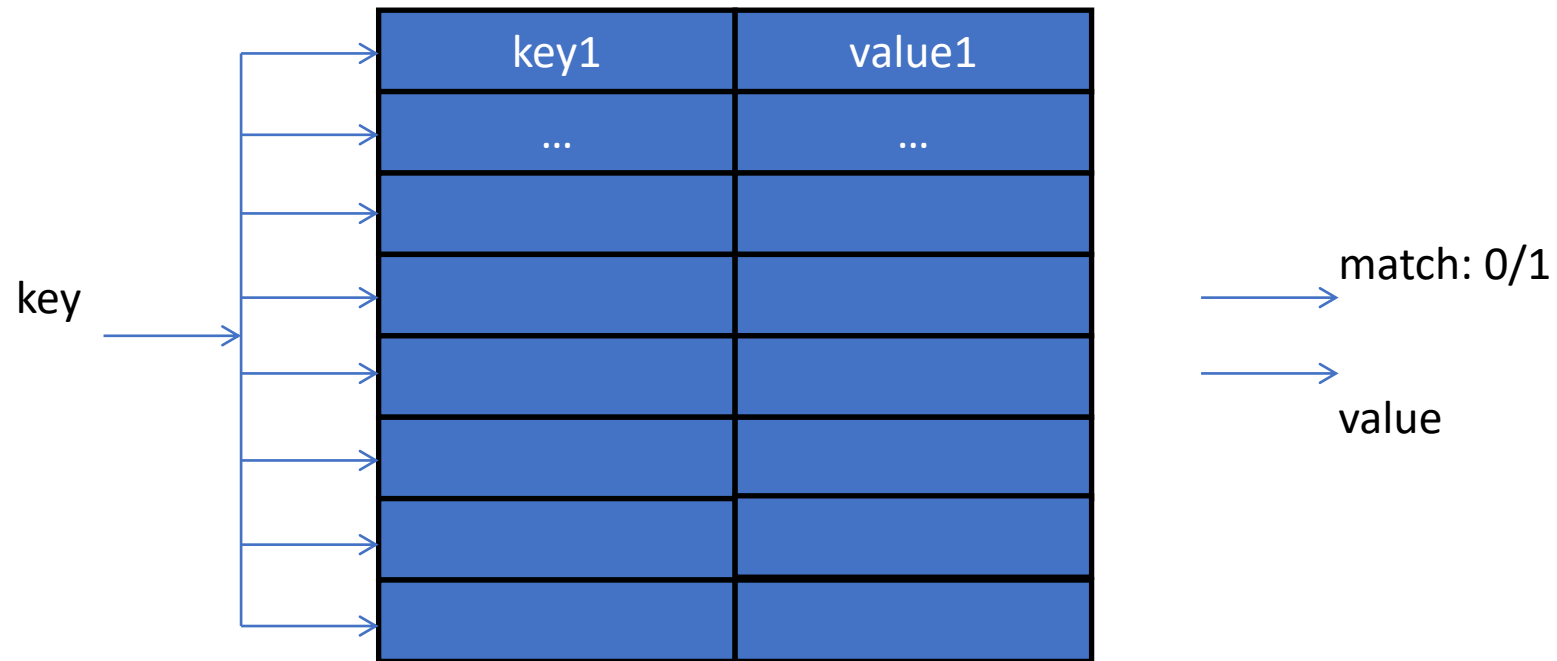
Use **associative memory** (special hardware)

- Regular memory
 Lookup by address
- Associative memory
 Lookup by contents
 Lookup in **parallel**
 Same concept as cache tags!

Regular Memory



Associative Memory



TLB



TLB Size

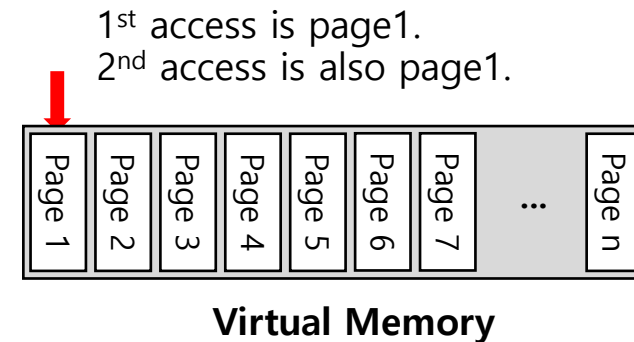
- Associative memory is very expensive
- Therefore, TLB small (64 – 1,024 entries)
- Want TLB hit-rate close to 100%
- If TLB full, **need to replace existing entry**. How?
 - Will discuss replacement policies later today.
 - Main idea: take advantage of locality.

TLB Hit Rate and Locality

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

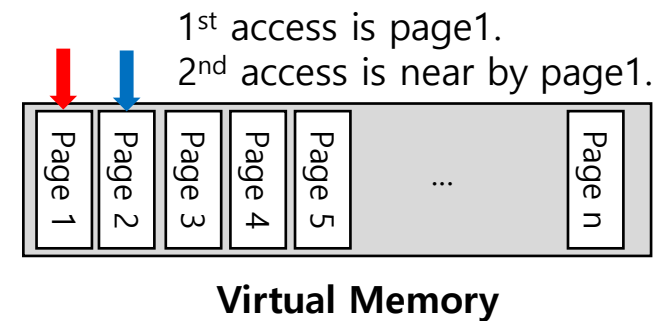
- **Temporal Locality**

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

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

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

Spatial Locality Example: Accessing an Array

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

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

TLB: miss

Spatial Locality Example: Accessing an Array

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

```
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

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

```
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

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

```
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

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

```
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

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

```
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

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

```

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

Temporal Locality Example: Accessing an Array

	OFFSET				
	00	04	08	12	16
VPN = 00					
VPN = 01					
VPN = 03					
VPN = 04					
VPN = 05					
VPN = 06		a[0]	a[1]	a[2]	
VPN = 07	a[3]	a[4]	a[5]	a[6]	
VPN = 08	a[7]	a[8]	a[9]		
VPN = 09					
VPN = 10					
VPN = 11					
VPN = 12					
VPN = 13					
VPN = 14					
VPN = 15					

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

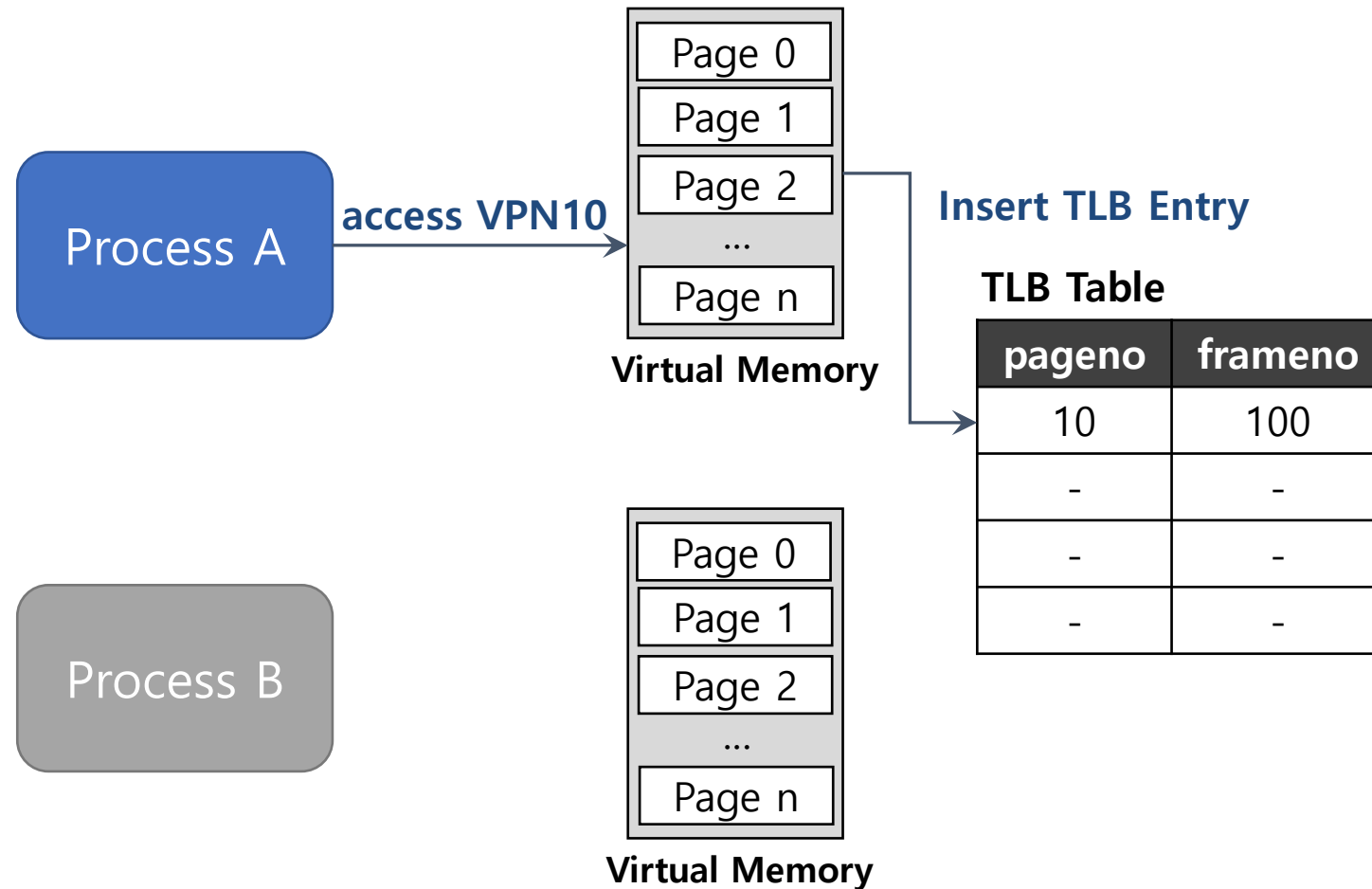
j=1 TLB: hit, hit, hit, hit, hit, hit, hit 0 misses and 10 hits.
hit, hit, hit Thus **TLB hit rate** is 100%.

The TLB improves performance
due to **temporal locality**

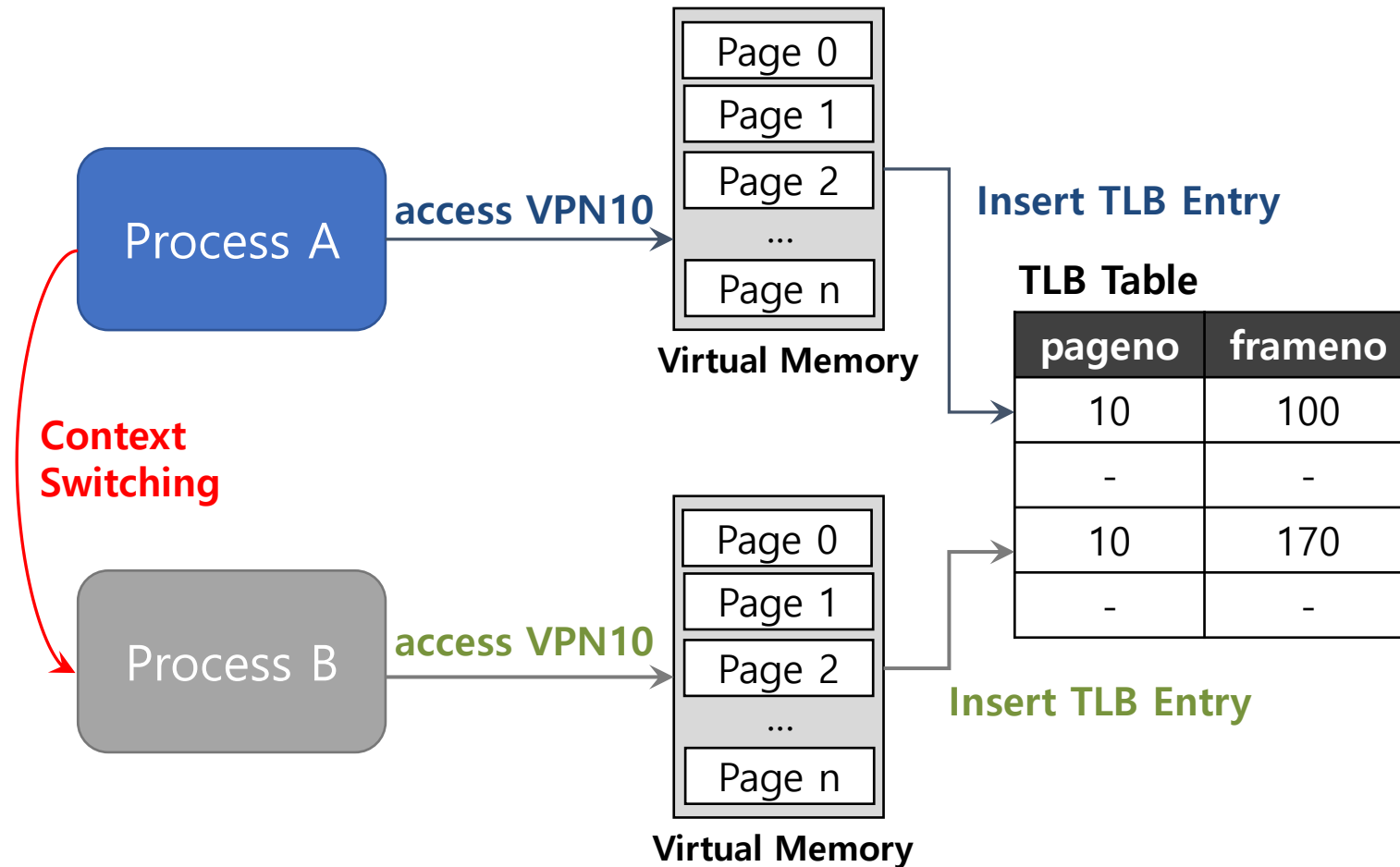
Revisiting Process Switching

- 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

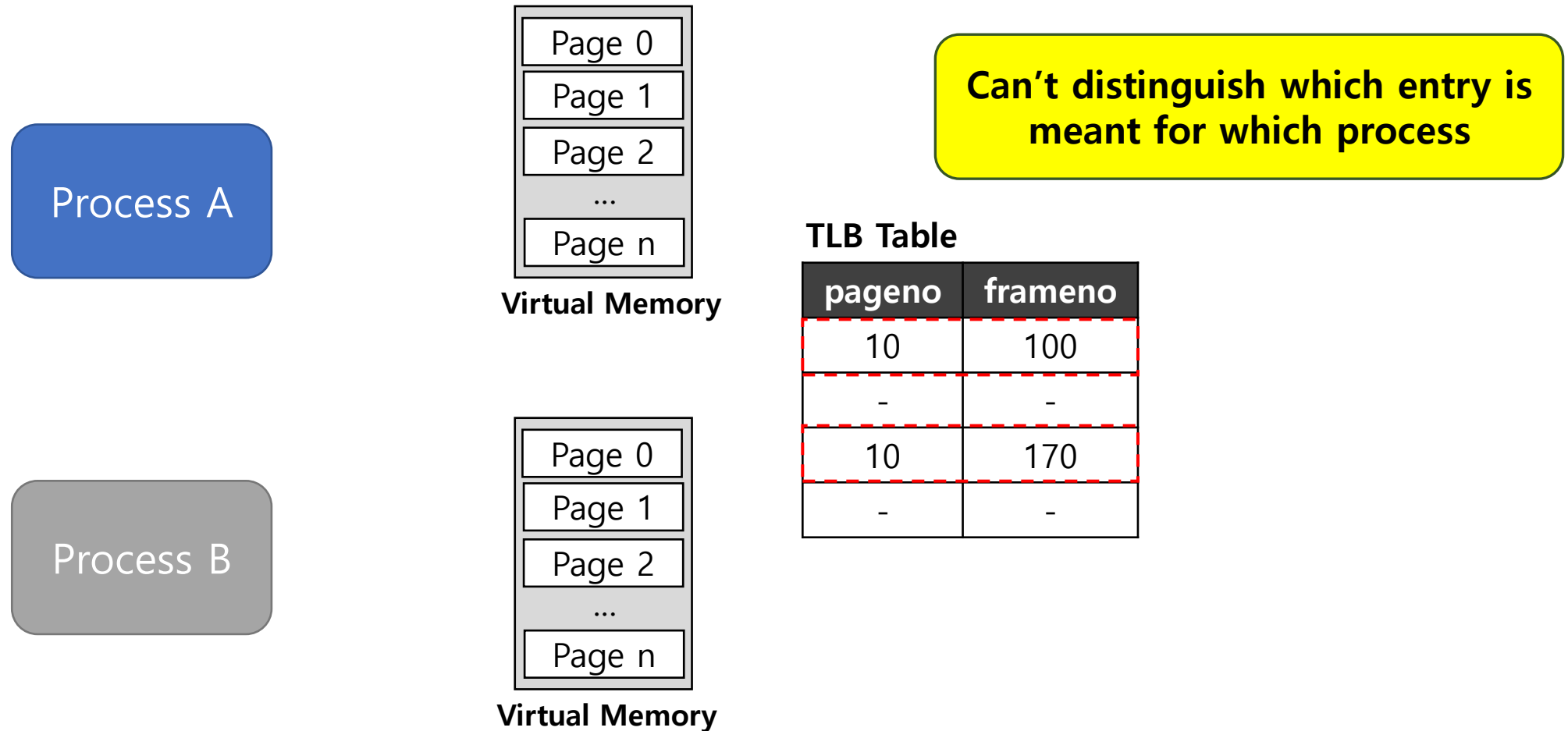
TLB Issue: Context Switching



TLB Issue: Context Switching



TLB Issue: Context Switching

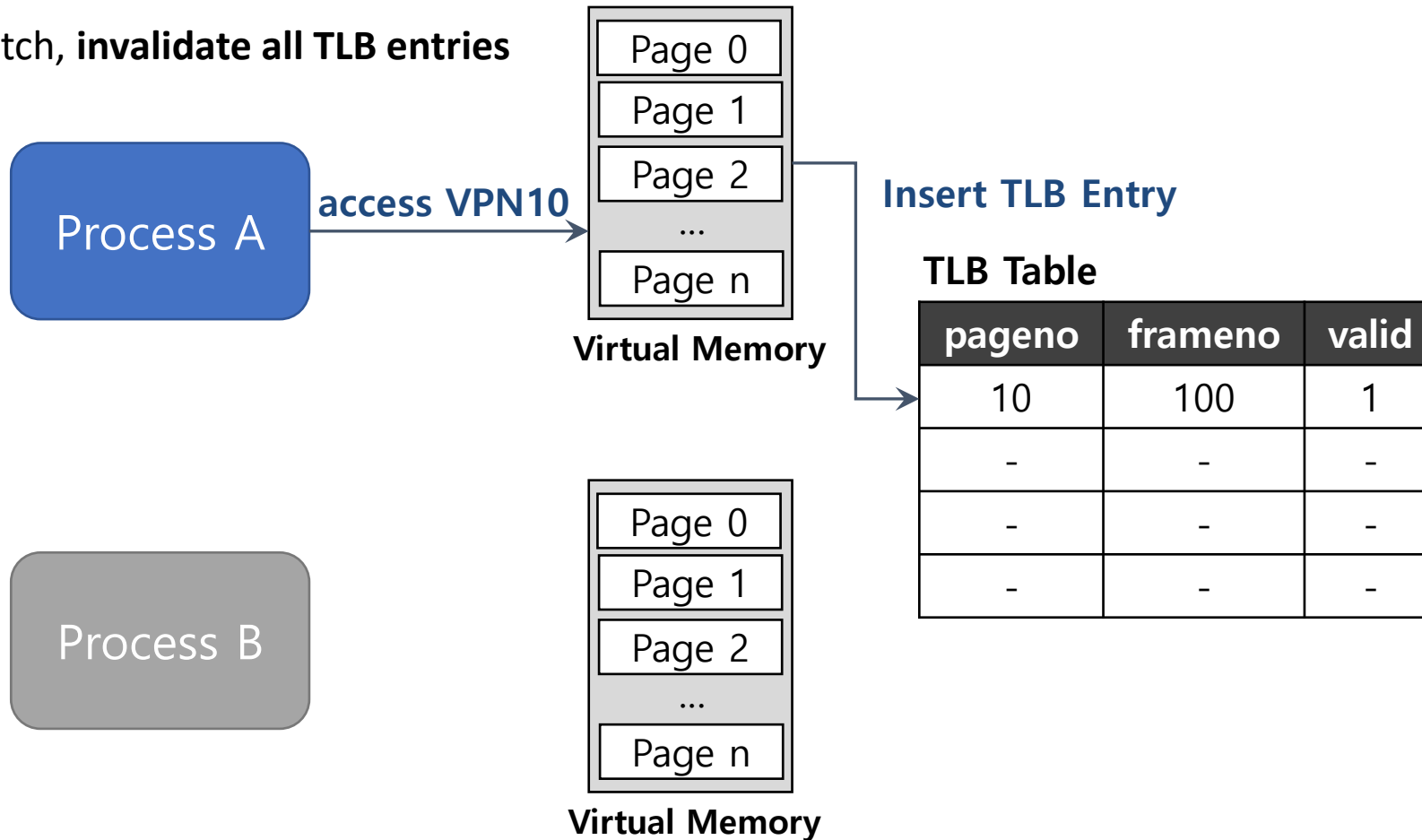


TLB Issue: Context Switching – Solution 1

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

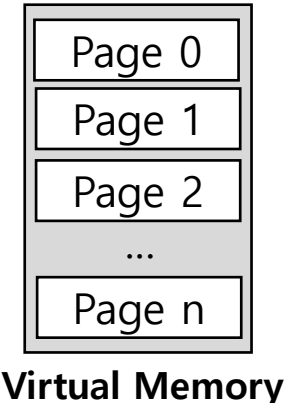
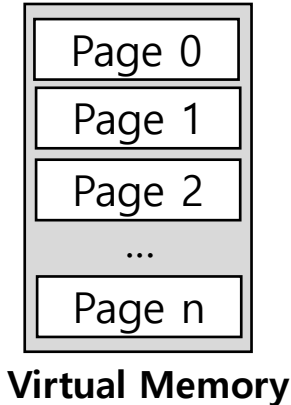
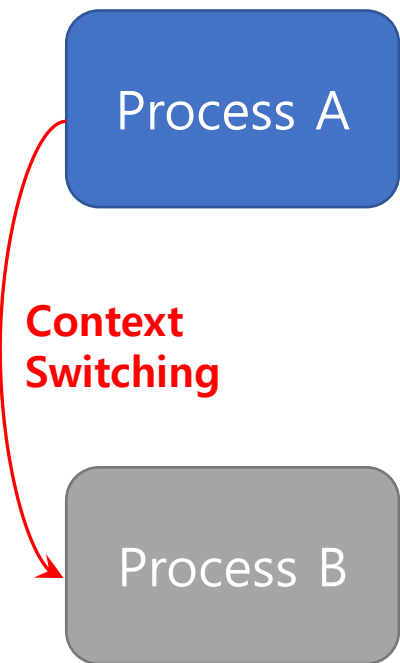
TLB Issue: Context Switching – Solution 1

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



TLB Issue: Context Switching – Solution 1

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



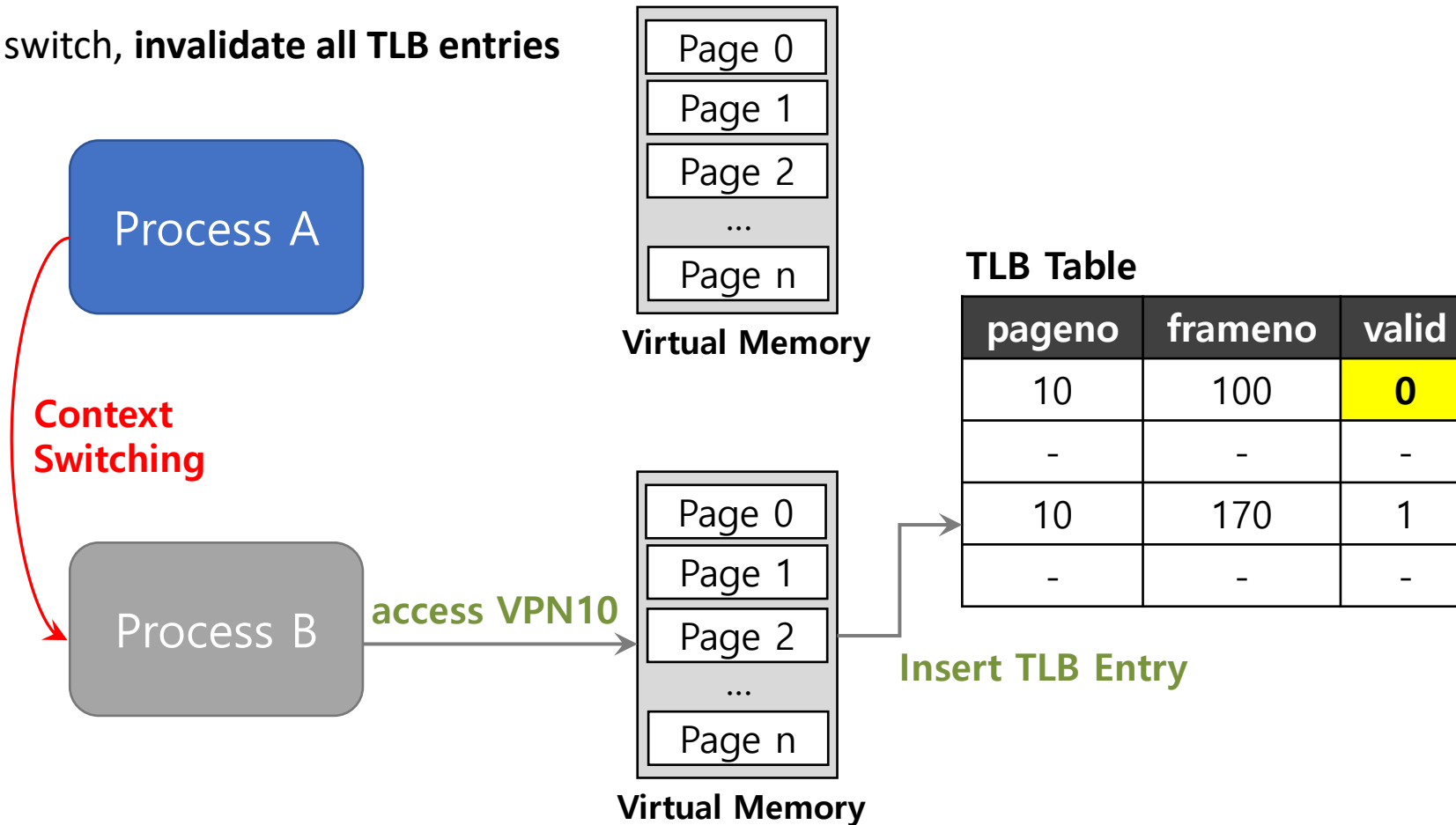
TLB Table

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

Setting valid bit to 0 effectively erases this entry even if the data hangs around a bit

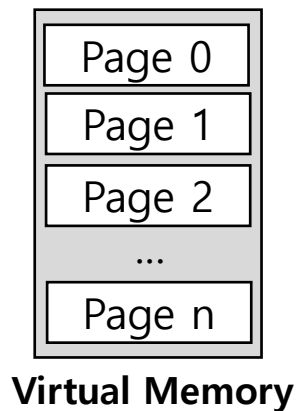
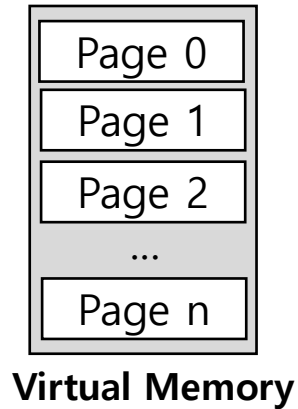
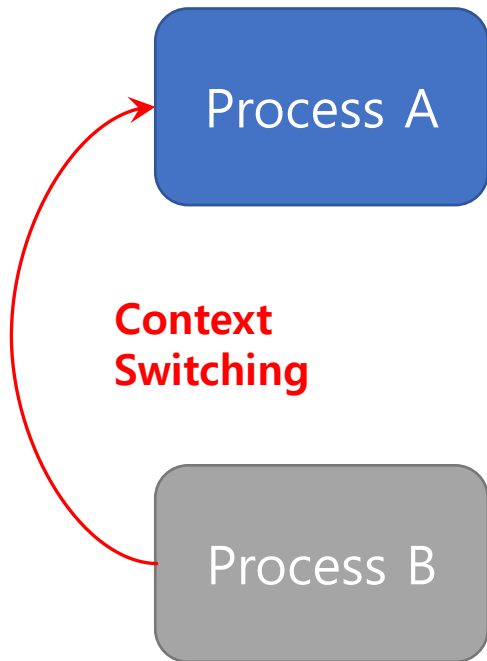
TLB Issue: Context Switching – Solution 1

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



TLB Issue: Context Switching – Solution 1

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



+ 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
-	-	-

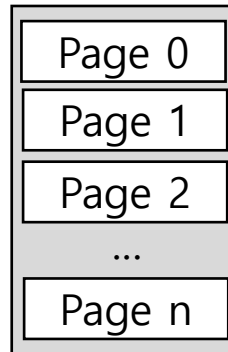
TLB Issue: Context Switching – Solution 2

- 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
- Almost all modern machines have this feature
 - But not all modern OSes use it (see: “TLB Shootdown”)

TLB Issue: Context Switching – Solution 2

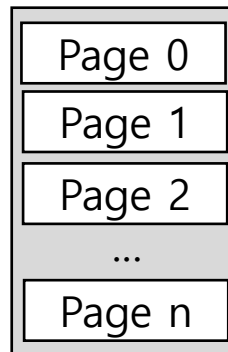
TLB match on pid AND match on pageno

Process A



Virtual Memory

Process B



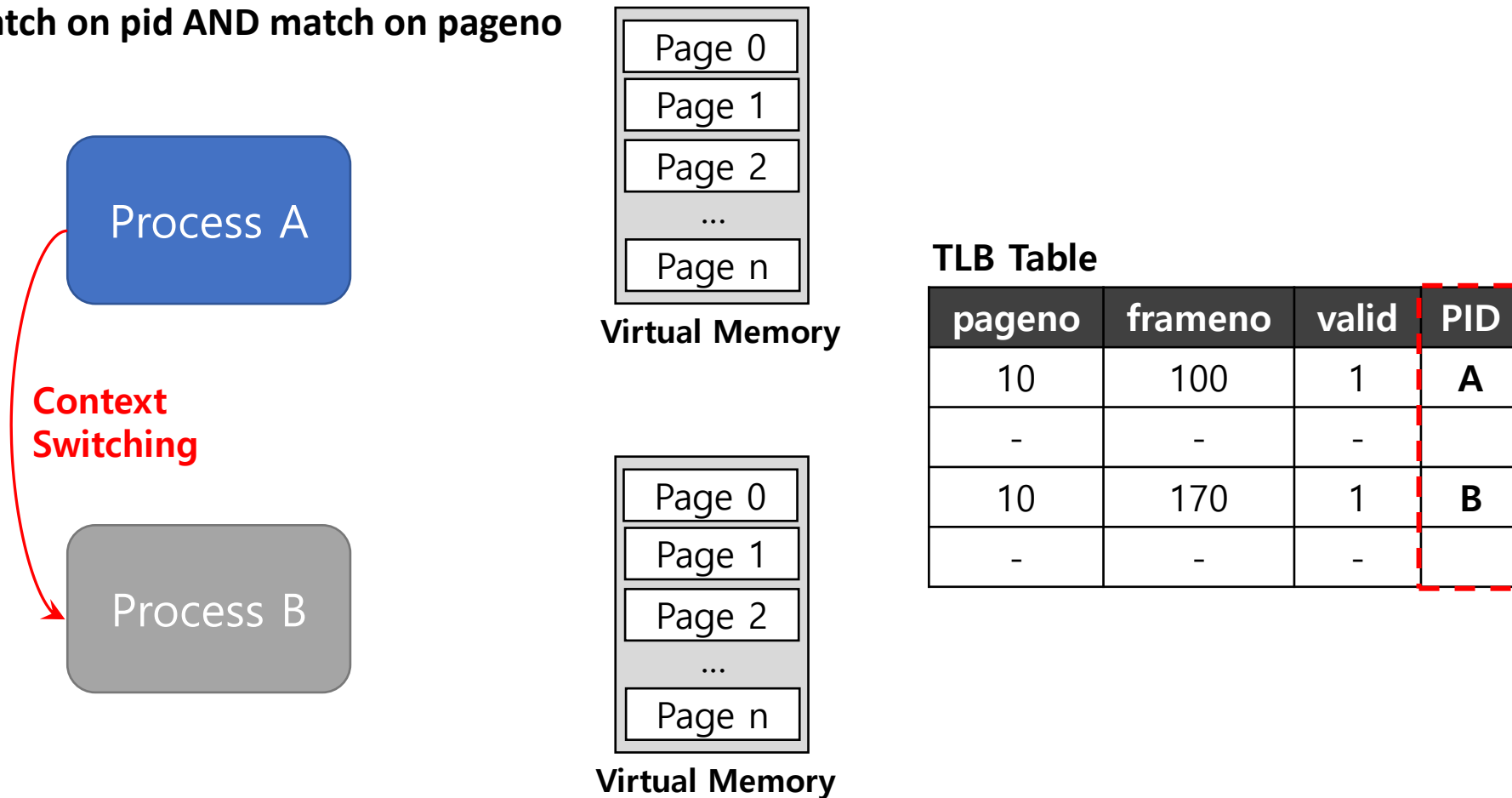
Virtual Memory

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



Part 2: Large Address Spaces

Dealing with Large Virtual Address Spaces

Dealing with Large Virtual Address Spaces

Example: 64-bit virtual address space (64-bit CPU instructions)

4kB pages \rightarrow 12-bit page offset

Leaves $64 - 12 = 52$ bits for pageno $\rightarrow 2^{52}$ page table entries

Let's say every page table entry 8B (true for x86 and aarch64)

\rightarrow Page table size for one process = $8\text{B} * 2^{52} \text{ PTEs} = 2^{55}\text{B}$

$= 2^5 \times 2^{50} = \mathbf{32 \text{ Petabytes (More than main memory!)}$

Dealing with Large Virtual Address Spaces

Example: 64-bit virtual address space (64-bit CPU instructions)

4kB pages \rightarrow 12-bit page offset

Leaves 64 – 12 = 52 bits for virtual page number

- Why 4kB pages ?
- How do we get to 12-bit page offset?

Let's say every page table entry is 8B

\rightarrow Page table size for one process = $8B * 2^{52} \text{ PTEs} = 2^{55}B$

$= 2^5 \times 2^{50} = 32 \text{ Petabytes (More than main memory!)}$

Dealing with Large Virtual Address Spaces

- Why 4kB pages ?

Typical value for page size; normally, this value is given as part of the problem statement, or you'd have enough information to deduce it.

Dealing with Large Virtual Address Spaces

- How do we get to 12-bit page offset?

Remember paging virtual address:

Virtual page number (bits)	Offset (bits)
----------------------------	---------------

Page size = 2^{Offset} Bytes . Why?

- *Every Byte needs to have an address, and*
- *We can represent 2^{Offset} addresses on Offset bits*

00

01

10

11

Page size = 4KB = $2^2 \times 2^{10}$ B = 2^{12} B \rightarrow Offset = 12.

Dealing with Large Virtual Address Spaces

Example: 64-bit virtual address space (64-bit CPU instructions)

~~4kB pages → 12-bit page offset~~

Leaves $64 - 12 = 52$ bits for pageno → 2^{52} page table entries

Let's say every page table entry 4B

→ Page table size for one process = $4B * 2^{52} \text{ PTEs} = 2^{54}B$

= $2^4 \times 2^{50} =$ **16 Petabytes (More than main memory!)**

Dealing with Large Virtual Address Spaces

Example: 64-bit virtual address space (64-bit CPU instructions)

~~4kB pages → 12-bit page offset~~

Leaves $64 - 12 = 52$ bits for pageno → 2^{52} page table entries

Let's say even

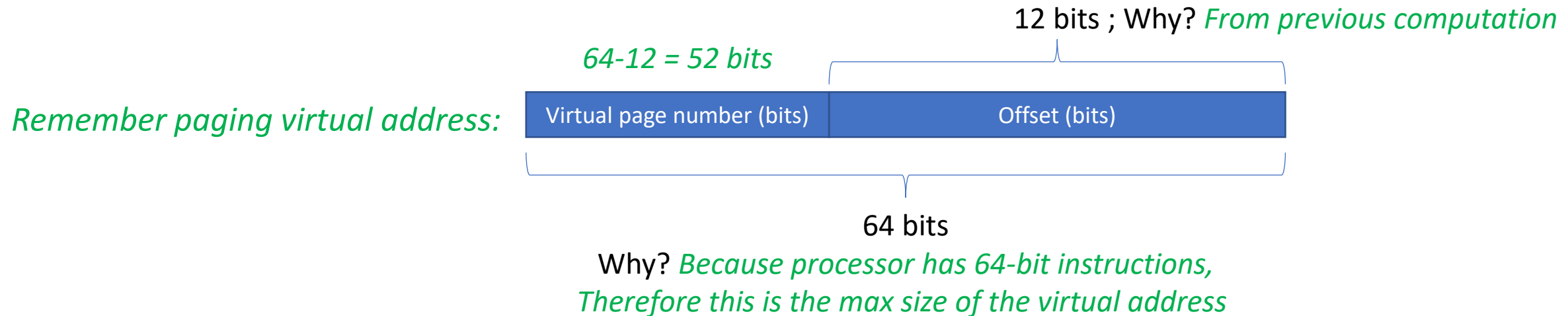
→ Page table

- Why 52 bits for pageno ?
- Why 2^{52} page table entries?

$= 2^4 \times 2^{50} = 16$ Petabytes (More than main memory!)

Dealing with Large Virtual Address Spaces

Why 52 bits for virt pageno?



Dealing with Large Virtual Address Spaces

Why 2^{52} Page table entries?

Remember paging virtual address:



If the virtual page number is represented on 52 bits, we have 2^{52} possible virtual page numbers.

The page table needs to keep track of all the virtual pages, therefore, it needs to be as big as the total number of virtual pages.

Dealing with Large Virtual Address Spaces

~~Example: 64-bit virtual address space (64-bit CPU instructions)~~

~~4kB pages \rightarrow 12-bit page offset~~

~~Leaves $64 - 12 = 52$ bits for pageno $\rightarrow 2^{52}$ page table entries~~

Let's say every page table entry 8B

\rightarrow Page table size for one process = $8\text{B} * 2^{52} \text{ PTEs} = 2^{55}\text{B}$

$= 2^5 \times 2^{50} = \mathbf{32 \text{ Petabytes (More than main memory!)}$

Dealing with Large Virtual Address Spaces

~~Example: 64-bit virtual address space (64-bit CPU instructions)~~

~~4kB pages \rightarrow 12-bit page offset~~

~~Leaves $64 - 12 = 52$ bits for pageno $\rightarrow 2^{52}$ page table entries~~

Let's say every page table entry 8B

\rightarrow Page table

$= 2^5 \times$

- **Why?** *This is an assumption*

The page table entry size is the amount of memory occupied by a page table entry. Depending on what metadata the page table entry stores, the size could range from a few Bytes to a few KB. On x86 and aarch64, it's the size of one physical pointer – 8B.

Dealing with Large Virtual Address Spaces

~~Example: 64-bit virtual address space (64-bit CPU instructions)~~


~~4kB pages → 12 bit page offset~~

~~Leaves $64 - 12 = 52$ bits for pageno → 2^{52} page table entries~~

~~Let's say every page table entry 8B~~

→ Page table size for one process = $8\text{B} * 2^{52} \text{ PTEs} = 2^{55}\text{B}$

= $2^5 \times 2^{50} \text{ B} = \mathbf{32 \text{ Petabytes (More than main memory!)}$

 $1\text{PB} = 2^{50} \text{ Bytes}$

Dealing with Large Virtual Address Spaces

Less extreme example: 32-bit virtual address space (32-bit CPU instructions)

4kB pages \rightarrow 12-bit page offset, 20 bits for pageno $\rightarrow 2^{20}$ page table entries

Again assuming every page table entry 8B

\rightarrow Page table size for one process = $2^{20} \times 8\text{B} =$

$$= 8 \times 2^{20} = 8\text{MB}$$

Let's say we run 100 processes in parallel (typical on a standard machine)

\rightarrow 800 MB of main memory used only for page tables.

QUIZ: How big are page Tables?

1. PTEs are **2 bytes**, and **32** possible virtual page numbers

$$32 * 2 \text{ bytes} = 64 \text{ bytes}$$

2. PTEs are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

$$2 \text{ bytes} * 2^{(24 - \log_2 16)} = 2^{21} \text{ bytes} = 2 * 2^{20} \text{ bytes (2 MB)}$$

3. PTEs are **4 bytes**, virtual addrs are **32 bits**, and pages are **2KB**

$$4 \text{ bytes} * 2^{(32 - \log_2 2K)} = 4 * 2^{21} \text{ bytes (8 MB)}$$

How big is each page table?

How to make Page Table smaller?

- Big pages
- Segmentation + Paging
- Multi-level page tables

Use Bigger Pages

32-bit virtual address space

~~4kB pages~~ **16kB pages**

- 14-bit page offset, 18 bits for pageno
- 2^{18} page table entries. Assuming 4B page table entry:
- Page table size = $4\text{B} \times 2^{18} = \mathbf{1\text{MB}}$

Factor of 4 reduction size compared to previous example.

Use Bigger Pages

- **Advantage:** Easy to implement.
- **Disadvantage:** Larger pages have **higher internal fragmentation**.
- Most systems use 4KB pages in common case. Sometimes 8KB.
 - But there is often support for “huge pages”: 2,4 MB or 1GB
 - Especially when page tables are multi-level

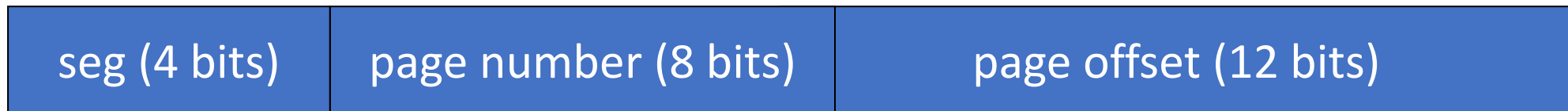
Segmentation + Paging

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages

Virtual address divided into three portions:



Segmentation + Paging

Implementation:

- Each segment has a page table
- Each segment track base (physical address) and bounds of **page table** for that segment

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read: **0x004070**
 0x202016 read: **0x003016**
 0x104c84 read: **error**
 0x010424 write: **error**
 0x210014 write: **error**
 0x203568 read: **0x02a568**

...
0x01f
0x011
0x003
0x02a
0x013
...
0x00c
0x007
0x004
0x00b
0x006
...

0x001000
0x001001

0x002000

Page Table
addresses not ordered,
but segments are contiguous

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x0	
2	0x001000	0x0f	1 1

within bounds

0x002070 read: 0x004070

0x202016 read: 0x003016

0x104c84 read: error

0x010424 write: error

0x210014 write: error

0x203568 read: 0x02a568

...		
0x01f		0x001000
0x011		
0x003		
0x02a		
0x013		
...		
0x00c	0x00	0x002000
0x007	0x01	
0x004	0x02	0x004000 +
0x00b		0x000070
0x006		
...		

Page Table

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

within bounds

0x002070 read: 0x004070

0x202016 read: 0x003016

0x104c84 read: error

0x010424 write: error

0x210014 write: error

0x203568 read: 0x02a568

...
0x01f 0x00
0x011 0x01
0x003 0x02
0x02a
0x013
...
0x00c
0x007
0x004
0x00b
0x006
...

0x001000

0x003000 +

0x000016

0x002000

Page Table

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read: 0x004070
 0x202016 read: 0x003016 No permission
 0x104c84 read: error
 0x010424 write: error
 0x210014 write: error
 0x203568 read: 0x02a568

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

Page Table

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read: 0x004070
 0x202016 read: 0x003016
 0x104c84 read: error No permission
 0x010424 write: error
 0x210014 write: error
 0x203568 read: 0x02a568

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

Page Table

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

Out of bounds

0x002070 read: 0x004070

0x202016 read: 0x003016

0x104c84 read: error

0x010424 write: error

0x210014 write: error

0x203568 read: 0x02a568

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

Page Table

Quiz: Segmentation + Paging

Every HEX digit represents
4 bits

seg (4 bits)	page number (8 bits)	page offset (12 bits)
--------------	----------------------	-----------------------

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

within bounds

0x002070 read: 0x004070

0x202016 read: 0x003016

0x104c84 read: error

0x010424 write: error

0x210014 write: error

0x203568 read: 0x02a568

...
0x01f 0x00
0x011 0x01
0x003 0x02
0x02a 0x03
0x013
...
0x00c
0x007
0x004
0x00b
0x006
...

0x001000

0x02a000 +

0x000568

0x002000

Page Table

Advantages of Segmentation + Paging

+ Supports sparse address spaces

- Decreases size of page tables
- If segment not used, not need for page table

+ Sharing

+ No external fragmentation

Disadvantages of Segmentation + Paging

- ☹ Potentially large page tables (for each segment)
- ☹ Must allocate each page table contiguously.
 - Can get tricky with large page table

Multi-level Page Tables

Turns the linear page table we've seen so far into a tree structure.

Multi-level Page Tables

Turns the linear page table we've seen so far into a tree structure.

2-level Page Table:

- Chop up the page table into page-sized units.
- If an entire page of page-table entries is invalid, don't allocate that page of the page table at all.
- To track if a page of page table is valid, use a **page directory** (new structure).

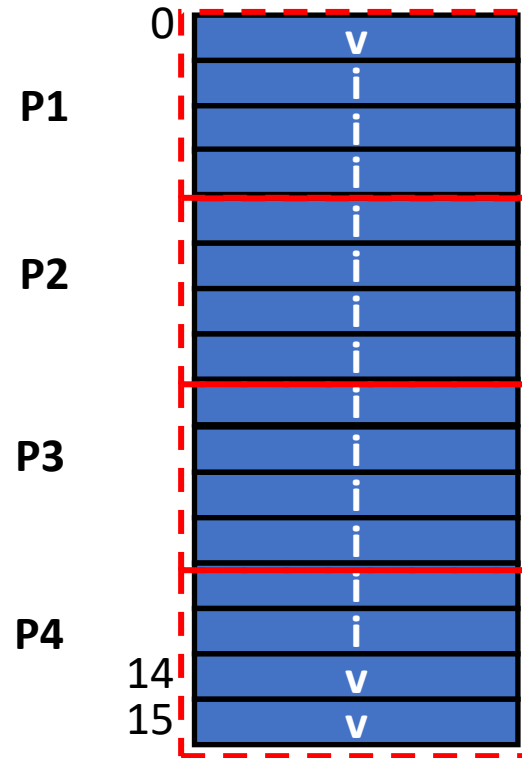
Flat (1-Level) Page Table

Page table size:
 $2^4 = 16$ entries

0	v
	i
	i
	i
	i
	i
	i
	i
	i
	i
	i
	i
	i
	i
	i
14	v
15	v

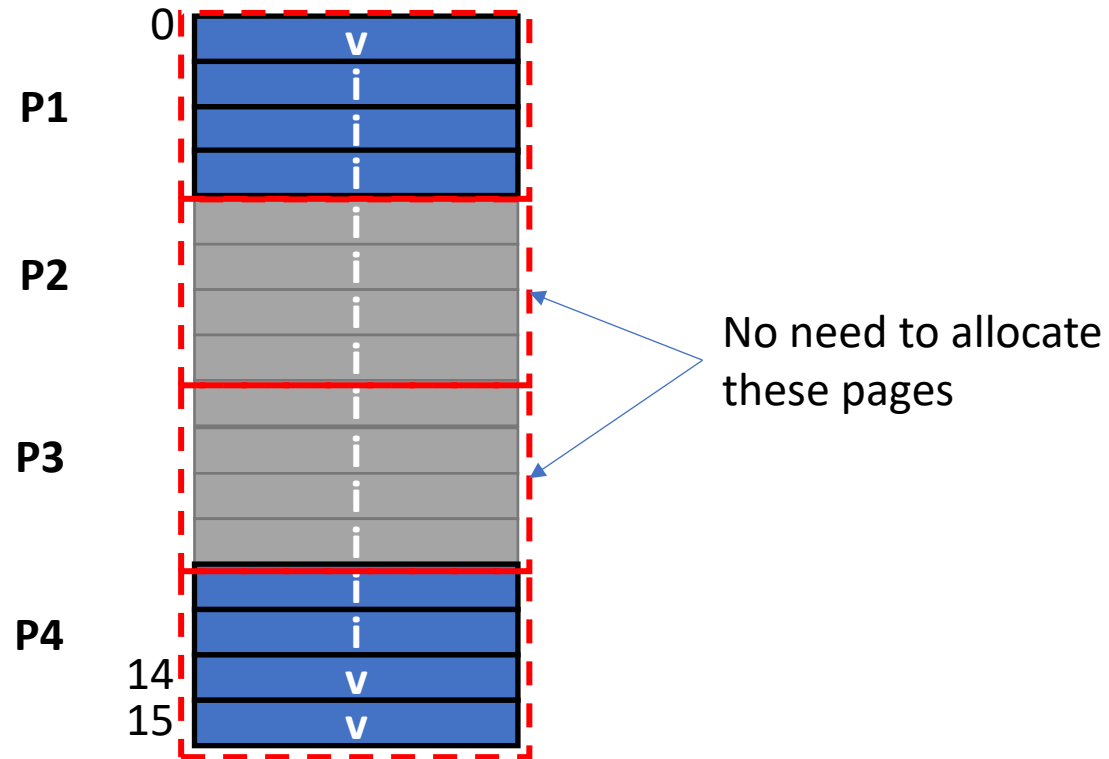
1-Level Page Table → 2-Level Page Table

Page table size:
 $2^4 = 16$ entries



1-Level Page Table → 2-Level Page Table

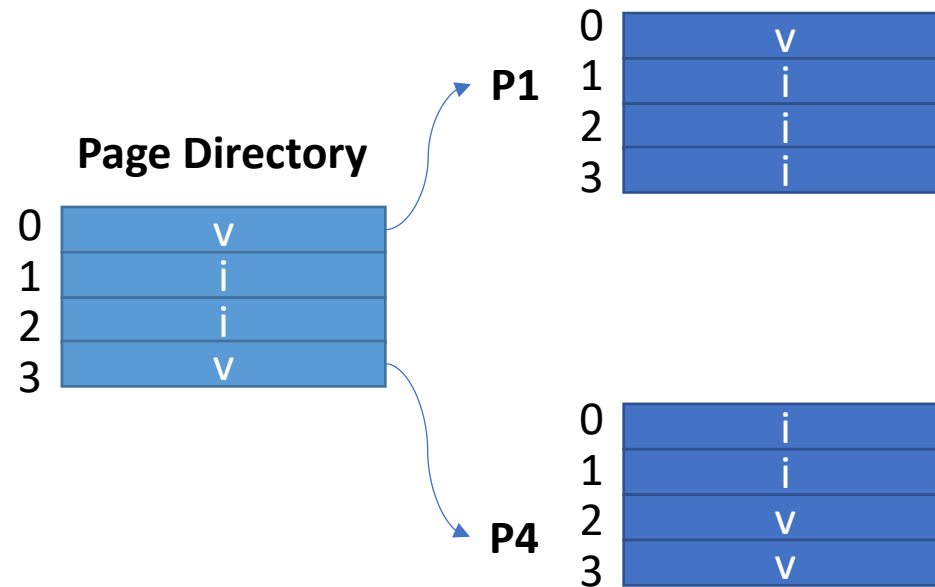
Page table size:
 $2^4 = 16$ entries



2-Level Page Table

Page table size (page directory + 2 page tables)

$2^2 + 2 \times 2^2 = 12$ entries (< 16)

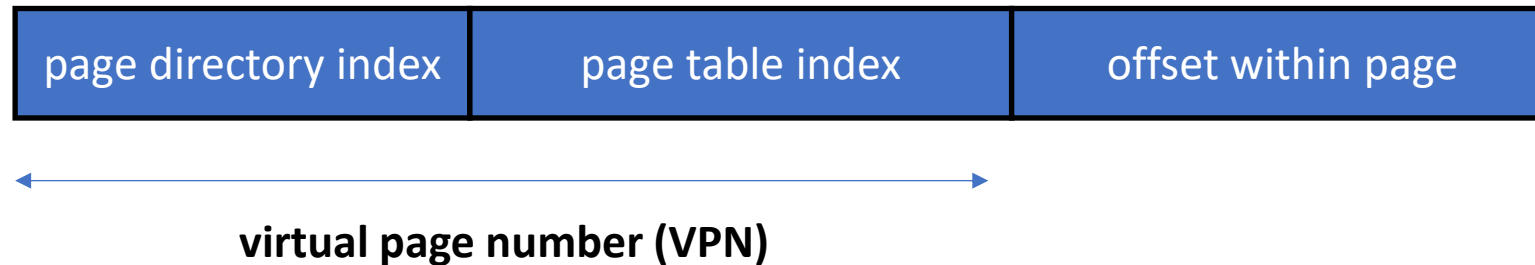


Virtual Address

- Single-level page table



- 2-level page table



Why Useful?

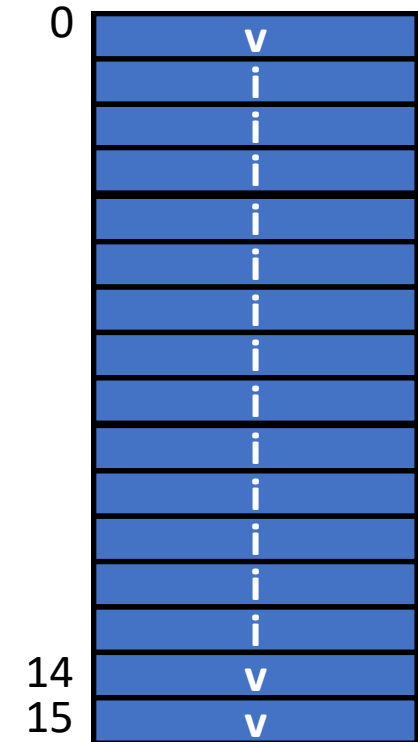
- **For sparse address spaces**
 - Most address spaces are sparsely populated

One-level page table:

- Need page table for entire address space

Two-level page table:

- Need top-level page table for entire address space
- Need only second-level page tables for populated parts of the address space



Let's practice!

- Virtual address – 32 bits
- Only low 20MB and upper 2MB valid
- Page size – 4k or offset = 12 bits, so VPN = 20 bits

What is the size of a 1-level page table (#PTEs)?

What is the size of a 2-level page table with 8-bit page directory indices and 12-bit level 2 page indices (#PTEs)?

Size of a 1-level page table (#PTEs)?

- Virtual address – 32 bits
- Only low 20MB and upper 2MB valid
- Page size – 4k or offset = 12 bits, so VPN = 20 bits

What is the size of a 1-level page table (#PTEs)?



For a 1-level page table, this is irrelevant because we need to represent all the virtual pages.

The VPN # of bits and the offset are computed like in the first example, with the exception that this time the CPU has 32-bit instructions, so the total number of bits in a virtual address is 32.

The #PTEs is computed like in the first example, by using the number of bits in the VPN:

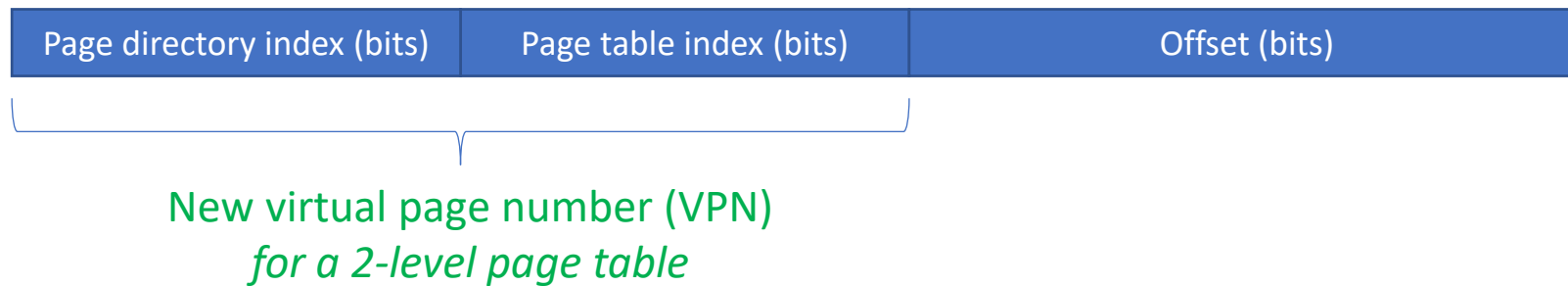
#PTEs = 2^{20} = 1,048,576 page table entries

Note that in this example we stop at the number of page table entries, without computing the size of the Page table. If we wanted to compute the size of the page table we would need to multiply 2^{20} by the Size of a page table entry (in the first example, we estimated it at 8B)

Size of a 2-level page table (#PTEs)?

What is the size of a 2-level page table with 8-bit page directory indices and 12-bit level 2 indices (#PTEs)?

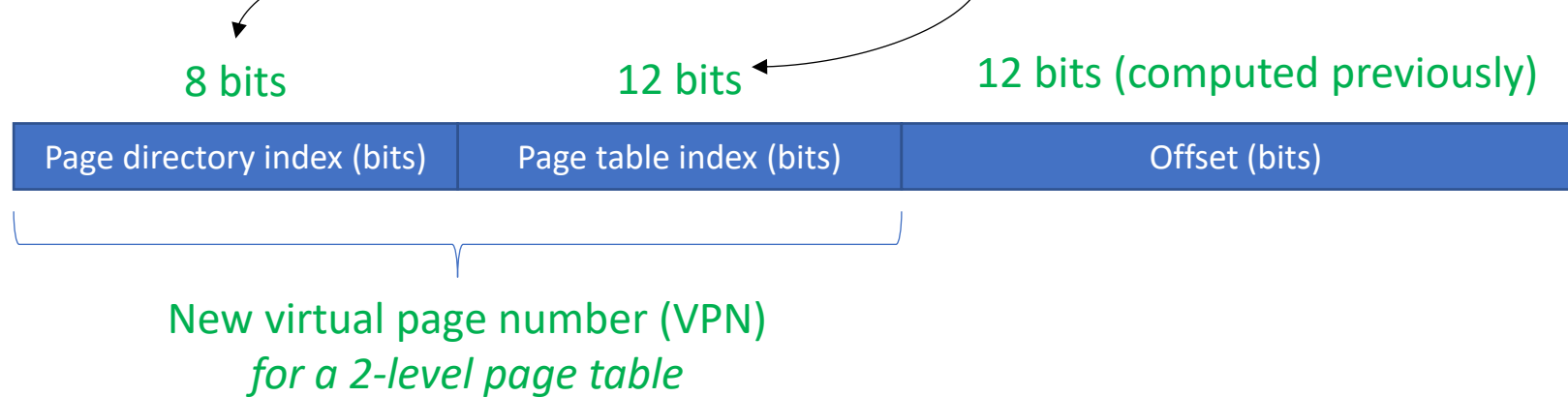
Virtual address structure changes with 2-level page table:



Size of a 2-level page table (#PTEs)?

What is the size of a 2-level table with 8-bit page directory indices and 12-bit level 2 page indices (#PTEs)?

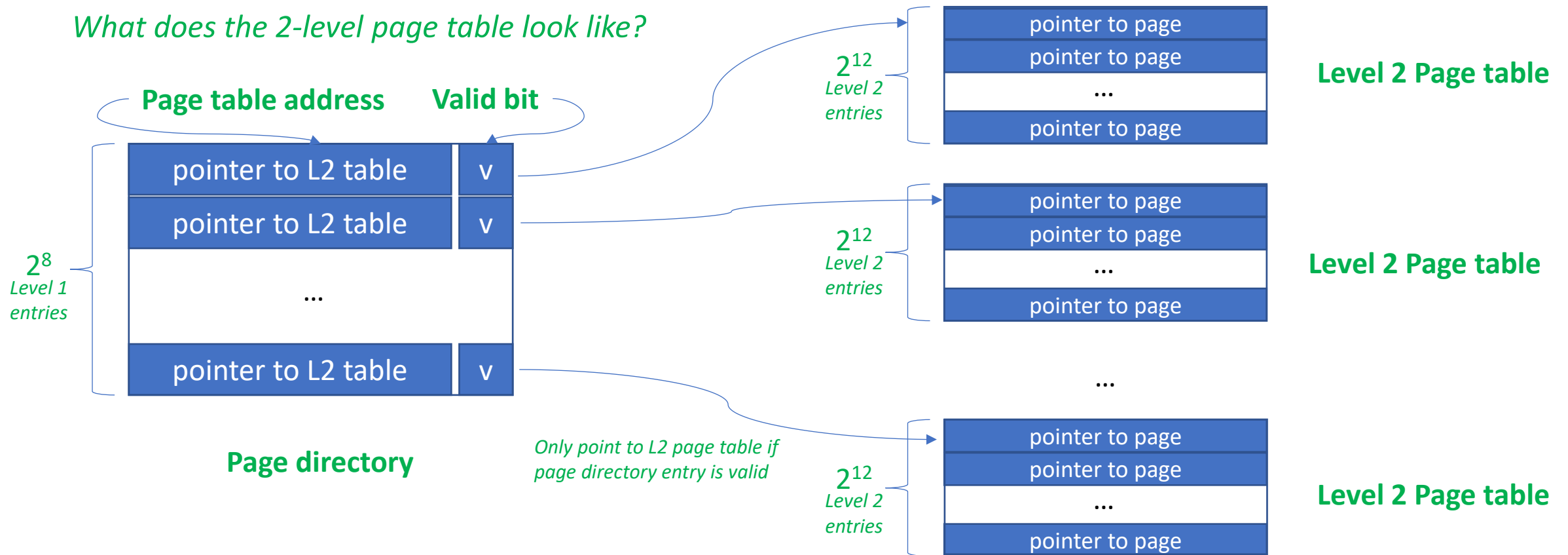
Virtual address structure changes with 2-level page table:



Size of a 2-level page table (#PTEs)?

What is the size of a 2-level page table with 8-bit page directory indices and 12-bit level 2 pages indices (#PTEs)?

What does the 2-level page table look like?



Size of a 2-level page table (#PTEs)?

Only low 20MB and upper 2MB valid

- *How many page directory entries are valid?*
- *How many level 2 page tables are allocated?*
- *How many total page table entries with a 2-level page table and a program with sparse address space?*

Size of a 2-level page table (#PTEs)?

- *How many level 2 page tables are allocated?*

Sparse address space → we will need at least one 2nd level page table for low 20MB and one 2nd level page table for upper 2MB

How large of an address space can we reference with a single level2 page table?

- *we know the size of one PTE index on level 2 is 12 bits → we can reference 2^{12} pages*
- *each page has 4KB → one level2 page table can reference $4KB * 2^{12} \rightarrow 2^2 * 2^{10} * 2^{12} B = 2^4 * 2^{20} B$
 $= 16MB$*

→ We need 2 level2 page tables to reference low 20MB + 1 level2 page table to reference upper 2MB.

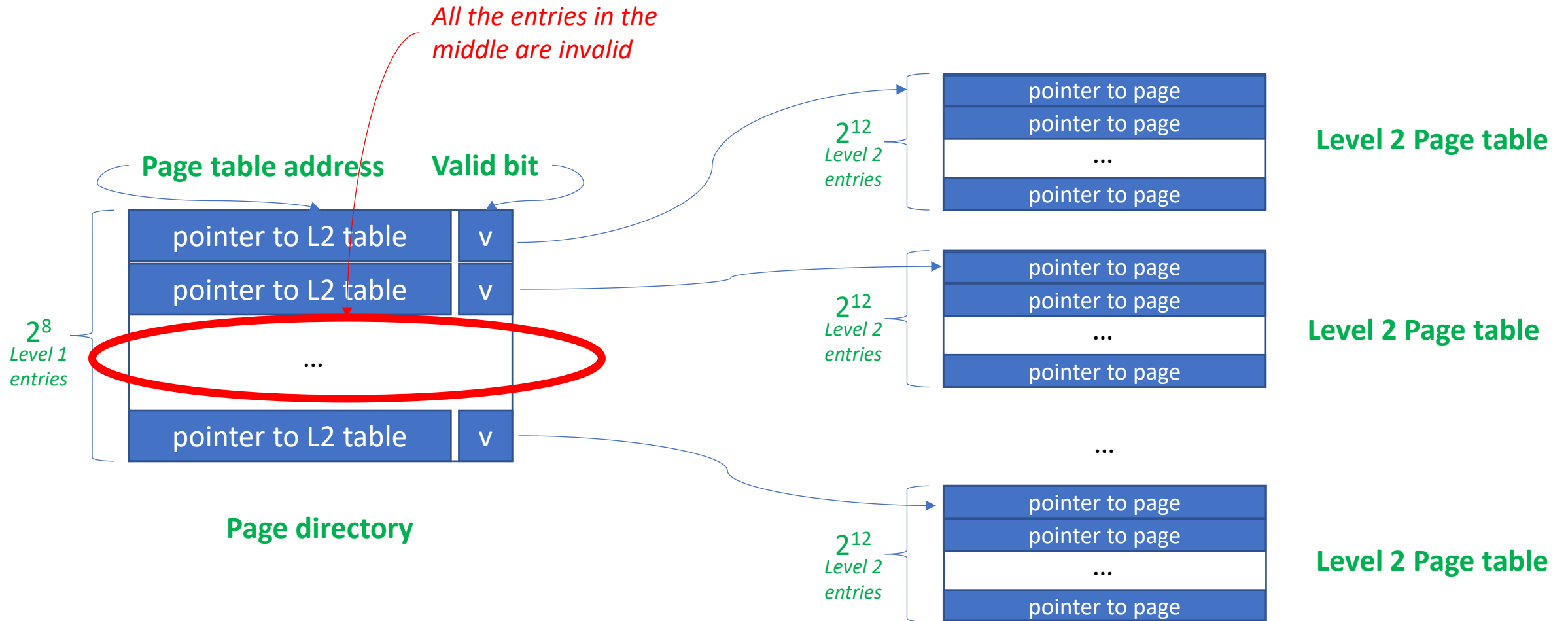
Size of a 2-level page table (#PTEs)?

- *How many page directory entries are valid?*

Only 3 page directory entries are valid.

Why? Because we need 2 level2 page tables to reference low 20MB + 1 level2 page table to reference upper 2MB.

Size of a 2-level page table (#PTEs)?



Size of a 2-level page table (#PTEs)?

- *How many total page table entries with a 2-level page table and a program with sparse address space?*
- 2^8 PTEs for 1st level
- $(2 + 1) \times 2^{12}$ PTEs for 2nd level;
- Total = $2^8 + 3 \times 2^{12} = 12,544$ PTEs

2-Level Page Tables for Dense Address Spaces?

- Not useful
- In fact, counter-productive (Why?)
- But most address spaces are sparse

Dense Address Space 1-Level Page Table

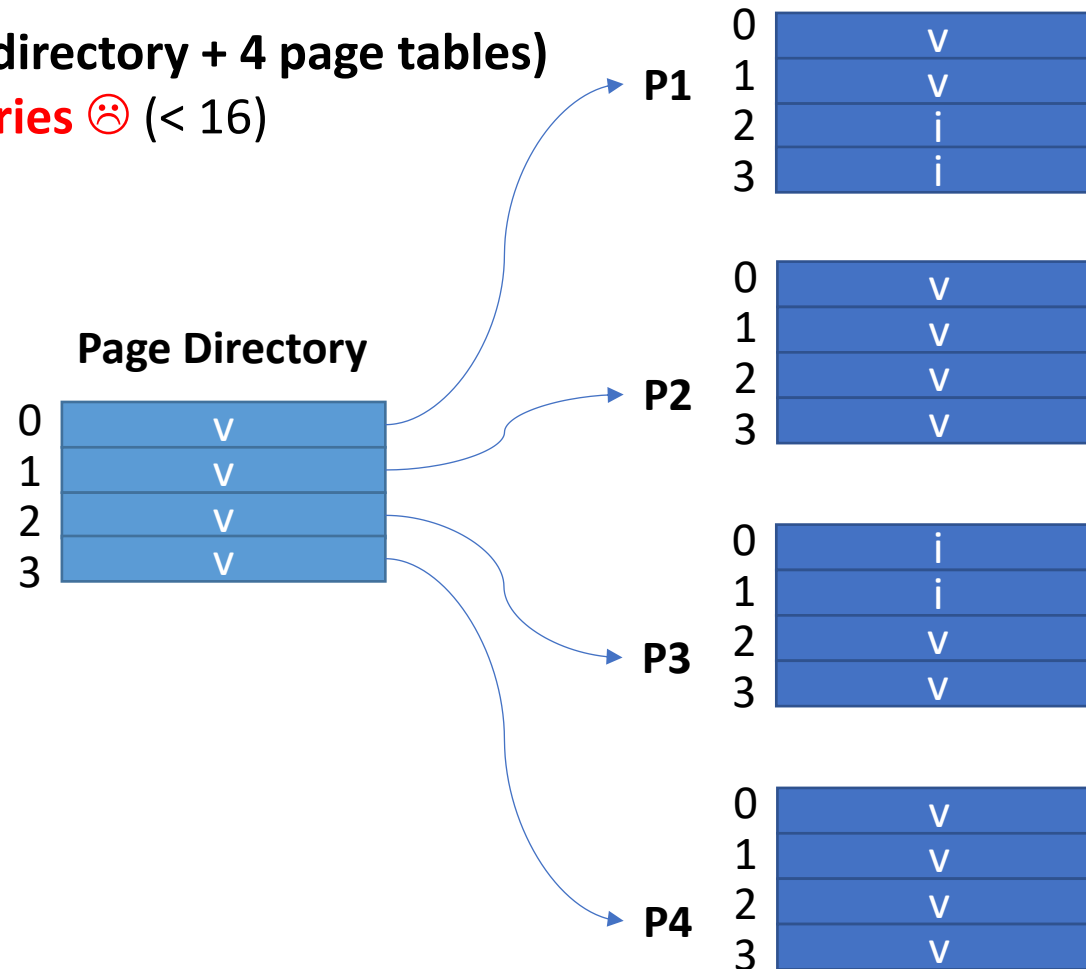
Page table size:
 $2^4 = 16$ entries

0	v
	v
	i
	i
	v
	v
	v
	v
	i
	i
	v
	v
	v
	v
14	v
15	v

Dense Address Space 2-Level Page Table

Page table size (page directory + 4 page tables)

$$2^2 + 4 \times 2^2 = \text{20 entries} \text{ 😞 } (< 16)$$



Are Two Levels Enough?

- Need top-level page table (page directory) for entire address space.
- Assume Size second-level page table == size of page
 - Why? Easy to allocate

Problem: Top-level can get too large if large address space (64 bits)

Solution (?): More levels.

More Levels: The Price to Be Paid

- Each level adds another memory access
- N-level page table
 - 1 memory access → N+1 memory accesses (N for page table + 1 for phys addr)
- But, TLB still works
 - If TLB hit, 1 memory access → 1 memory accesses
 - If miss, 1 memory access → N+1 memory accesses

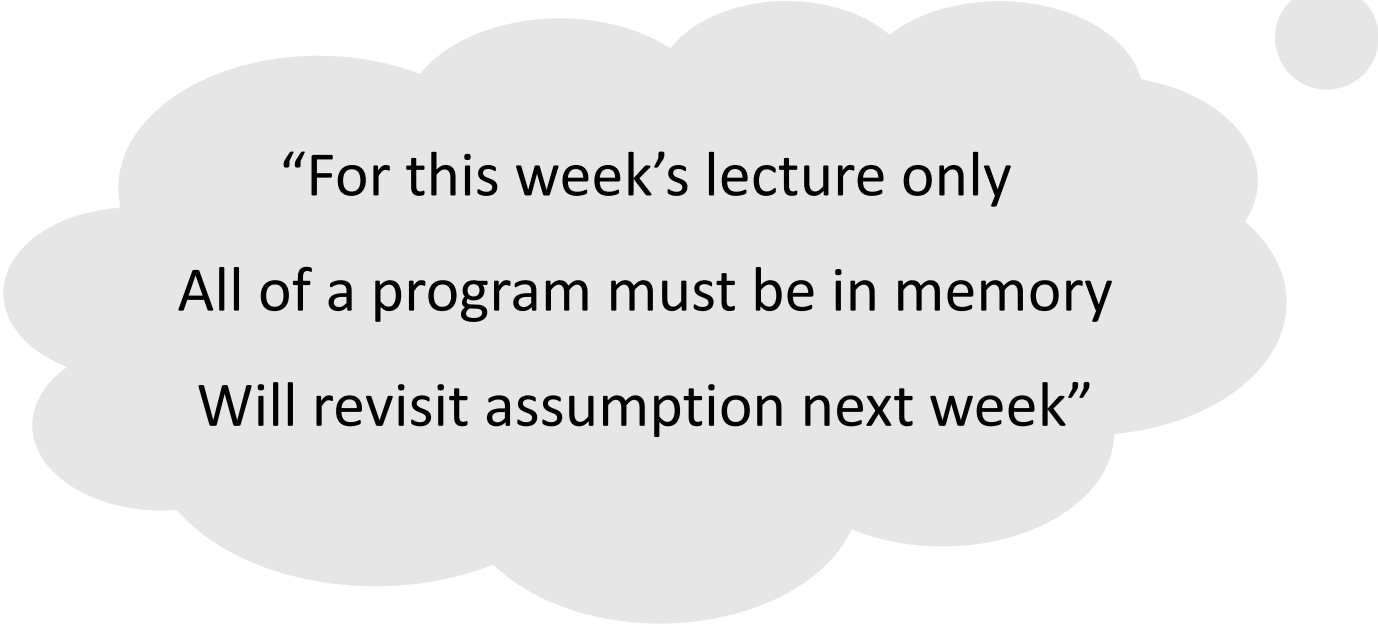
→ TLB hit rate must be very high (99+ %)

Part 3: Demand Paging

Demand Paging

Remember from Last 2 Weeks

Simplifying Assumption



“For this week’s lecture only
All of a program must be in memory
Will revisit assumption next week”

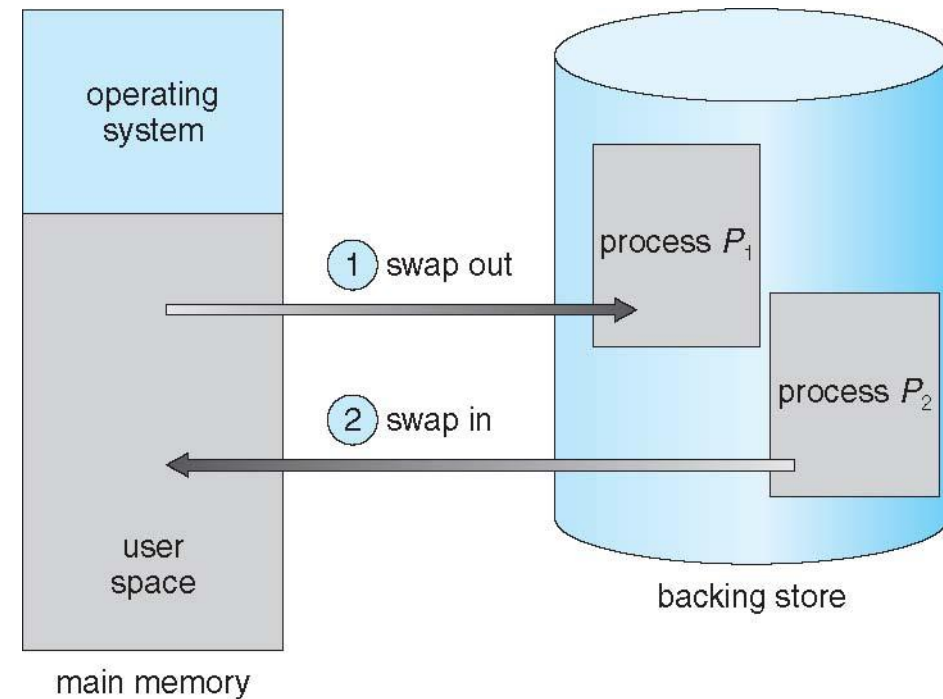
We are now going to drop this assumption

What if “out of memory”?

Need to get rid of one or more processes

Store them temporarily on disk

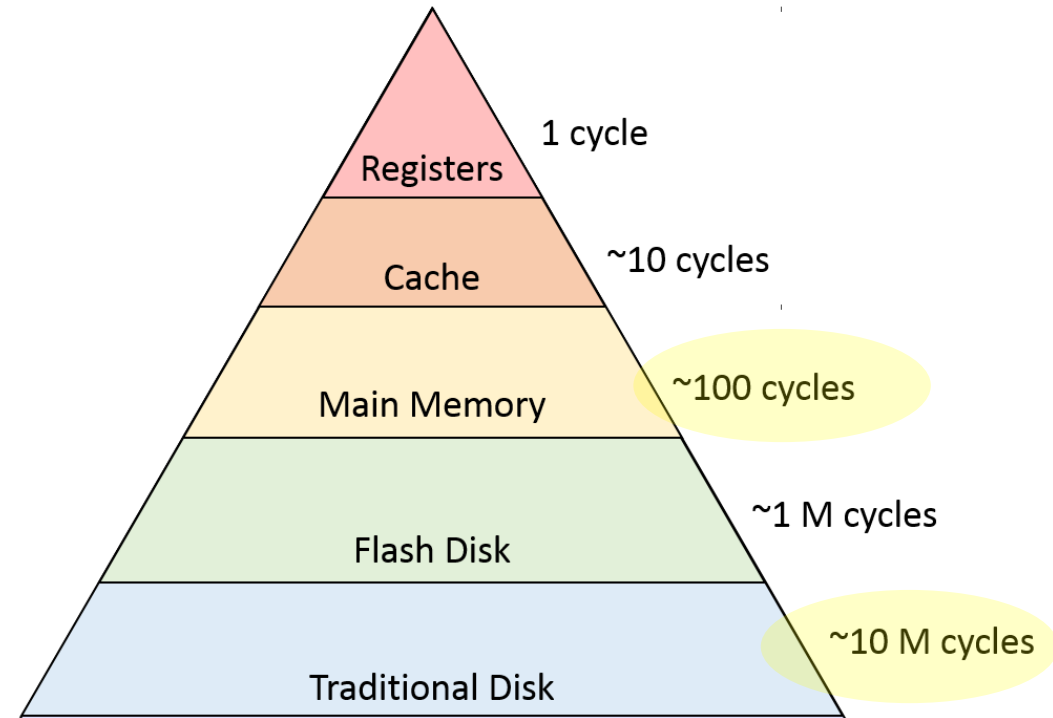
This is called **Swapping**



Process Switch to a Swapped Process?

Latency can be **very high**

Need to read image from disk



Process Switch to a Swapped Process?

A better solution:

- **Demand paging**
- **Not all of a process needs to be in memory**

Main Reason for Demand Paging

Virtual address spaces >> physical address space

- No machine has 2^{64} bytes (16 exabytes) of memory (yet).
- (Or 2^{57} , which we saw was the current limit of virtual addresses)

Why such large virtual address space?

- Convenient for programmer
- Don't have to worry about running out

Additional Benefits of Demand Paging

+ Shorter process startup latency

- Can start process without all of it in memory
- Even 1 page suffices

+ Better use of main memory

- Program often does not use certain parts
 - E.g., error handling routines
- Program often goes through different parts and doesn't come back
 - E.g., initialization, computation, termination

If the program is not in memory, then where is it?

Part of it is in memory

(Typically) all of it is on disk

- In a special partition called the **Backing Store**

If the program is not in memory, then where is it?

Part of it is in memory

(Typically) all of it is on disk

- In a special partition called the **Backing Store** or **Swap Memory**

Note the difference with swapping:

Swapping = *all* of program is **in memory** OR *all* of program is **on disk**

Demand paging = **part** of program is **in memory**

Despite this well-defined difference, backing store still often called swap.

If the program is not in memory, then where is it?

Part of it is in memory

(Typically) all of it is on disk

- In a special partition called the **Backing Store** or **Swap Memory**

Remember:

- CPU can only directly access memory
- CPU can only access data on disk through OS

Note the difference with swapping:

Swapping = *all* of program is **in memory** OR *all* of program is **on disk**

Demand paging = **part** of program is **in memory**

Despite this well-defined difference, backing store still often called swap.

Demand Paging Mechanism High Level

- What if program needs to access part only on disk?

Demand Paging Mechanism High Level

- What if program needs to access part only on disk?
- Program is suspended
- OS runs, gets page from disk
- Program is restarted

Demand Paging Mechanism High Level

- What if program needs to access part only on disk?

This is called a **page fault**

- Program is suspended
- OS runs, gets page from disk
- Program is restarted

This is called **page fault handling**

Demand Paging Issues

1. How to discover a page fault?
2. How to suspend process?
3. How to get a page from disk?
 - 3'. How to find a free frame in memory?
4. How to restart process?

1. Discover Page Fault

Idea: Use the valid bit in page table

- Without demand paging:
 - Valid bit = 0: page is invalid
 - Valid bit = 1: page is valid
- **With demand paging**
 - Valid bit = 0: page is invalid **OR page is on disk**
 - Valid bit = 1: page is valid **AND page is in memory**
 - OS needs additional table: invalid / on-disk?
 - (But the MMU doesn't care about this)

2. Suspending the Faulting Process

Idea: Trap into the OS

- Invalid bit access **generates trap**
- Save process information in PCB when trapping into the OS

3. Getting the Page from Disk

Idea: OS handles fetch from Disk

- Assume (for now) there is at least one free frame in memory
- Allocate a free frame to process
- Find page on disk
 - OS needs to remember the swap space in page-sized units.
 - Need an extra table for that in OS.
- Get disk to transfer page from disk to frame

3. Getting the Page from Disk

Idea: OS handles fetch from Disk

- Assume (for now) there is at least one free frame in memory
- Allocate a free frame to process
- Find page on disk
 - OS needs to remember the swap space in page-sized units.
 - Need an extra table for that in OS.
- Get disk to transfer page from disk to frame



3. While the Disk is Busy



- Shouldn't waste CPU cycles
- Invoke scheduler to run another process
- When **disk interrupt** arrives
 - Suspend running process
 - Get **back to page fault handling**

3. While the Disk is Busy



Why OK to handle page faults in OS (and not in hardware)?

- Disk is so slow that it masks the latency of handling this in OS.

3. Completing Page Fault Handling

- `Pagetable[pageno].framenno = new framenno`
- **`Pagetable[pageno].valid = 1`**
- Set process state to ready
- Invoke scheduler

4. When Process Runs Again

Idea: Restarts the previously faulting instruction

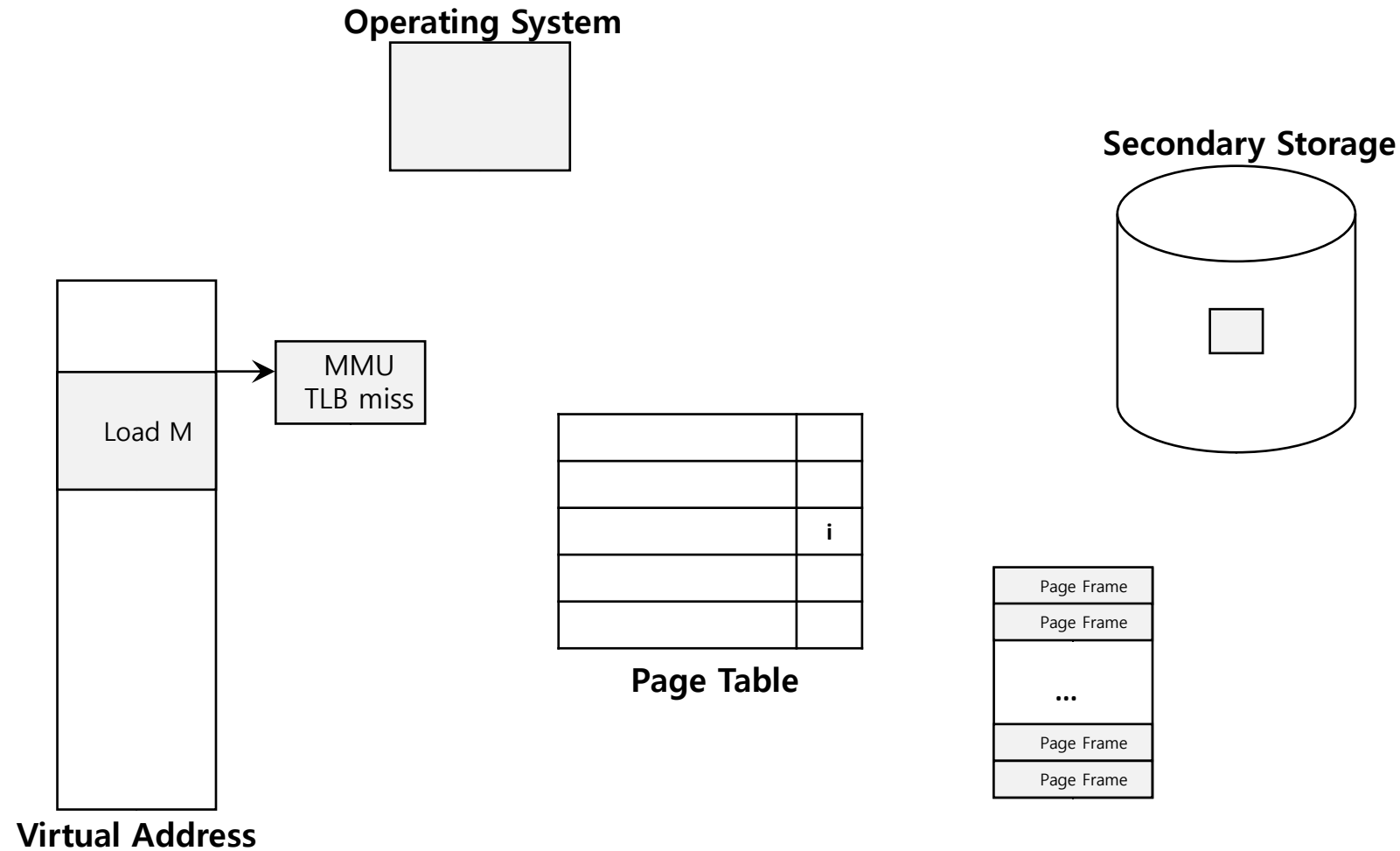
Process now finds

- Valid bit to be set to 1
- Page in corresponding frame in memory
- Note: **exceptions** like page faults are different from **interrupts** like the timer
 - After a context switch by timer, on next schedule, continues with the next instruction
 - After switch by exception, next time, continue by re-trying “exceptional” instruction

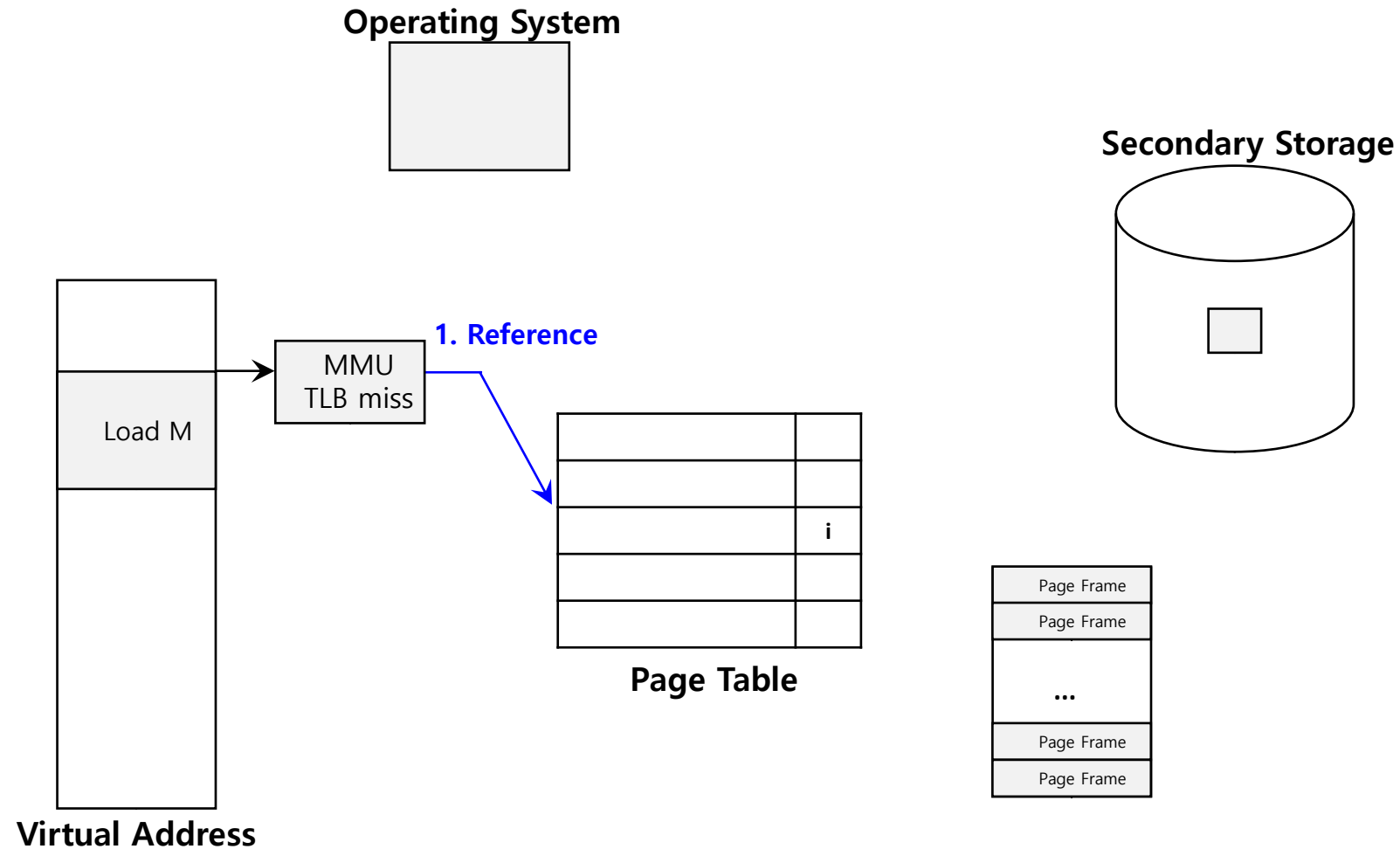
Demand Paging Summary

1. How to discover a page fault? ← Use valid bit in page table
2. How to suspend process? ← Invalid bit traps into OS
3. How to get a page from disk? ← OS handles fetch from disk
- 3'. How to find a free frame in memory?
4. How to restart process? ← Restart faulting instruction

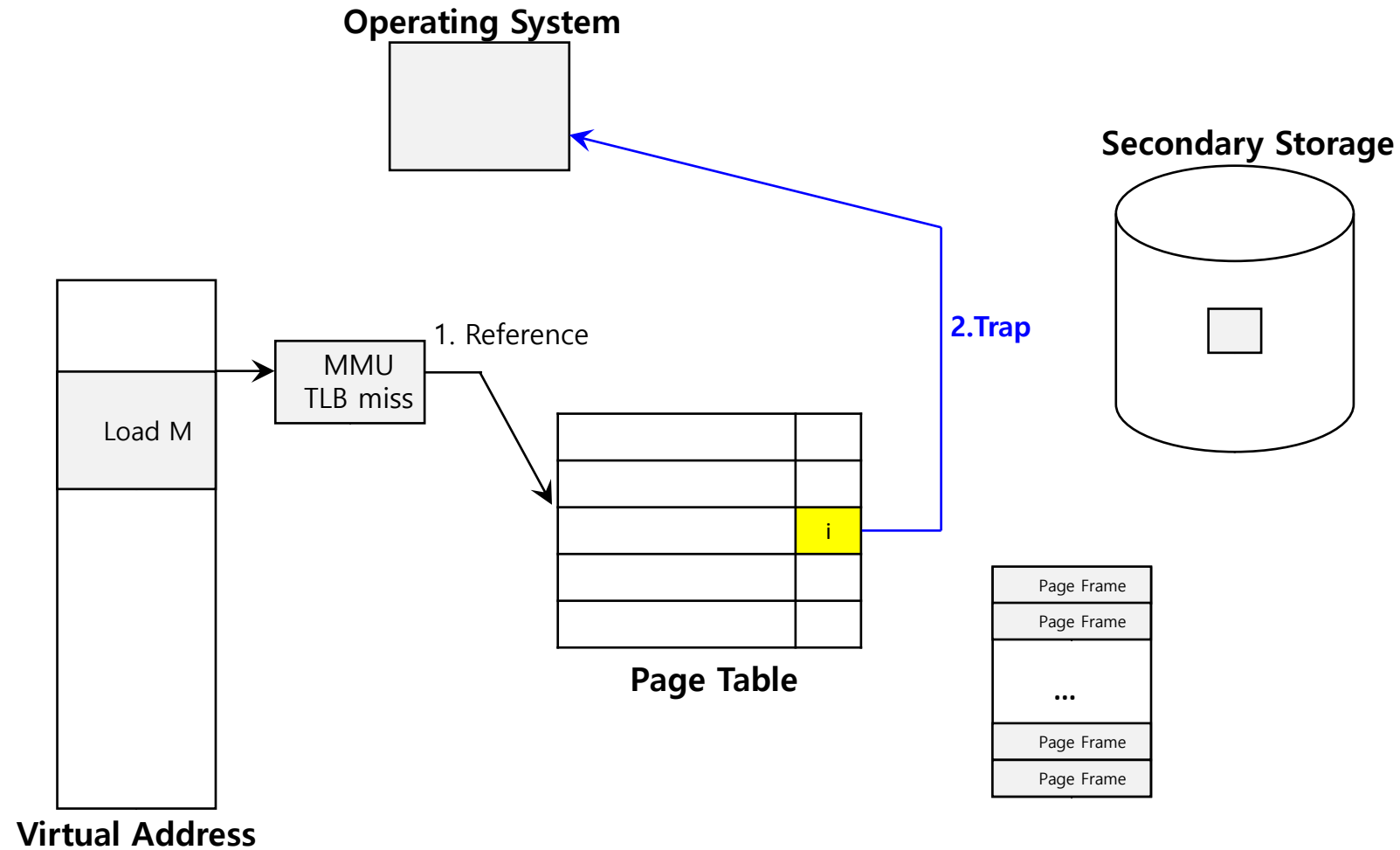
Demand Paging Summary



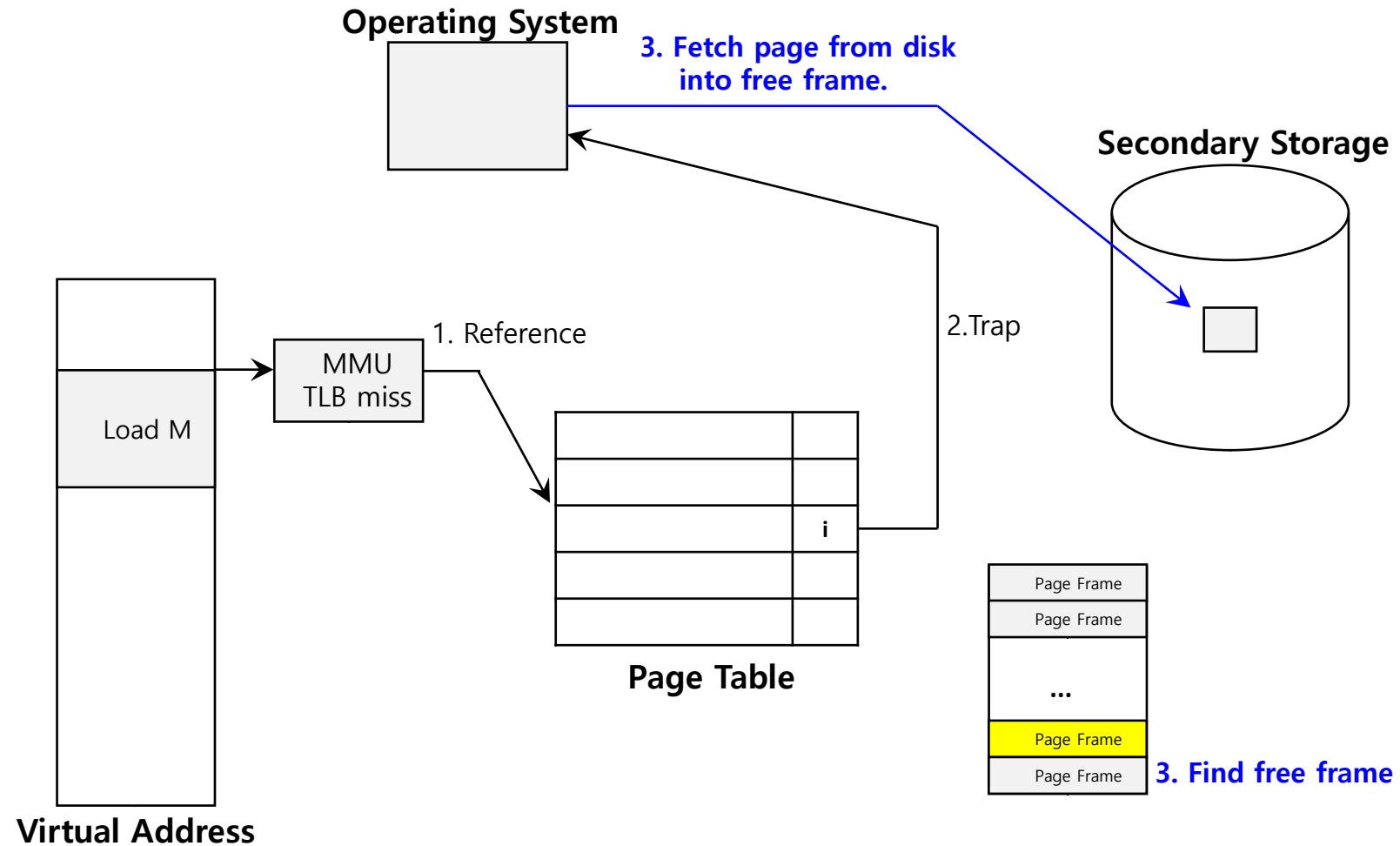
Demand Paging Summary



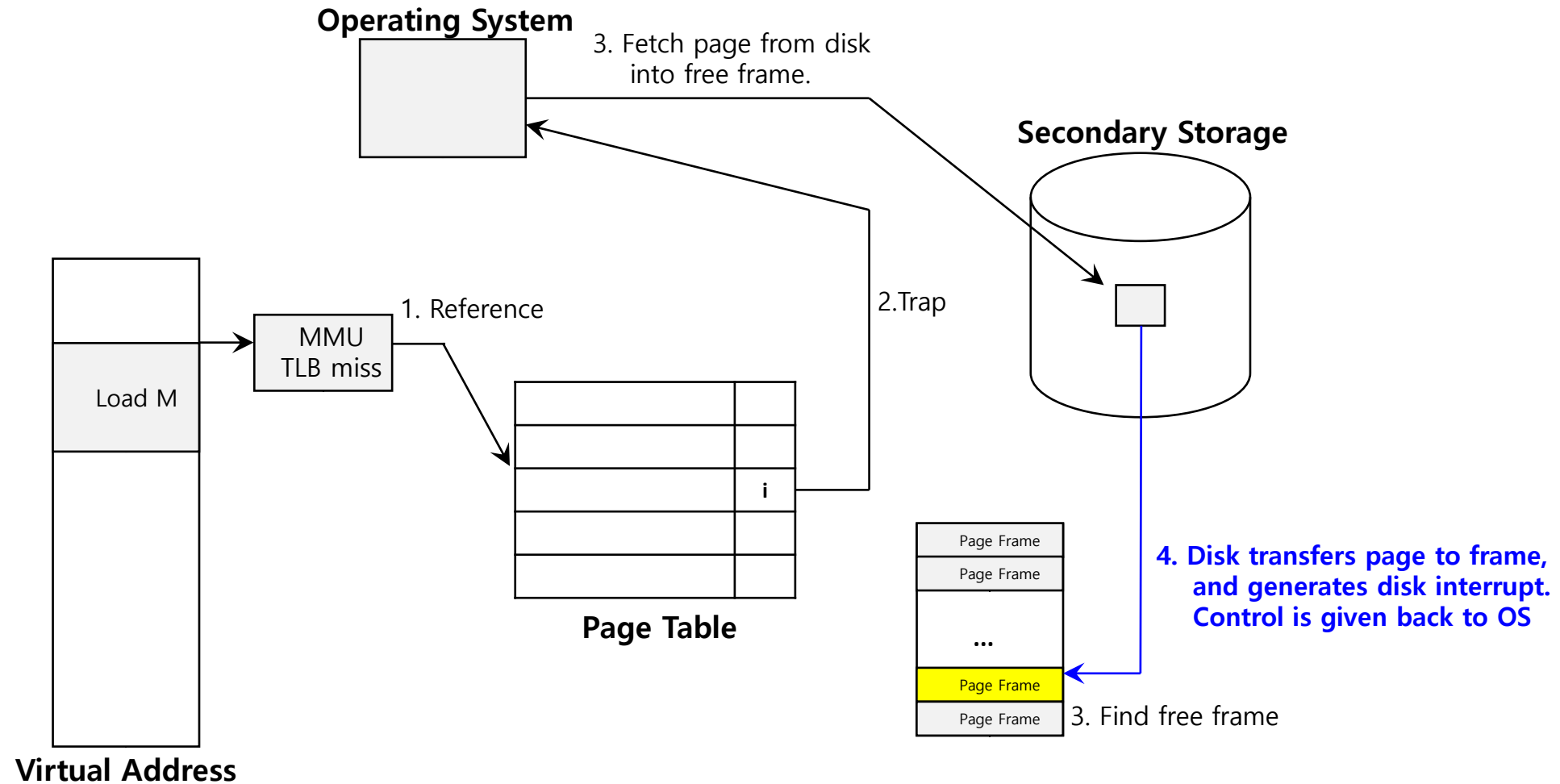
Demand Paging Summary



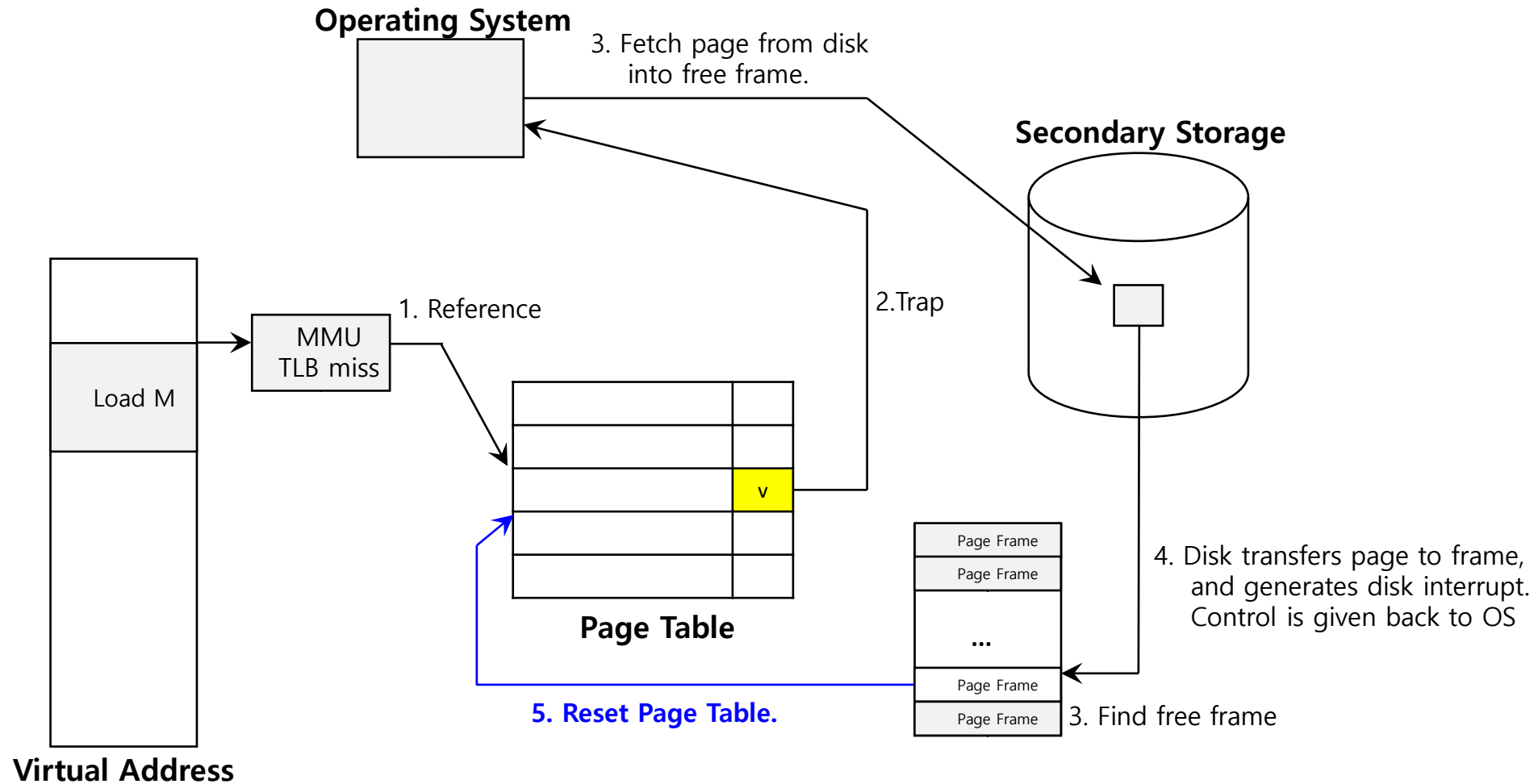
Demand Paging Summary



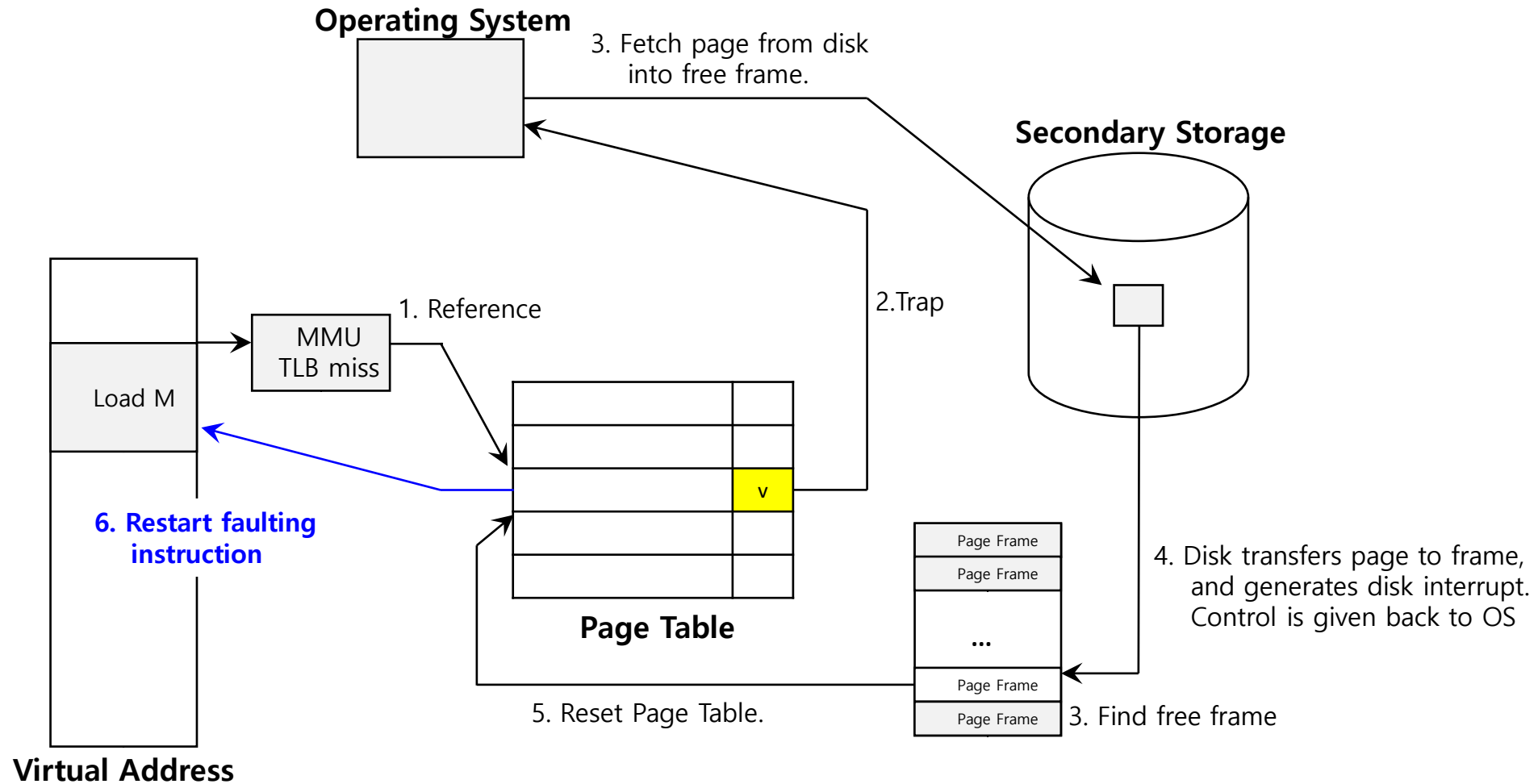
Demand Paging Summary



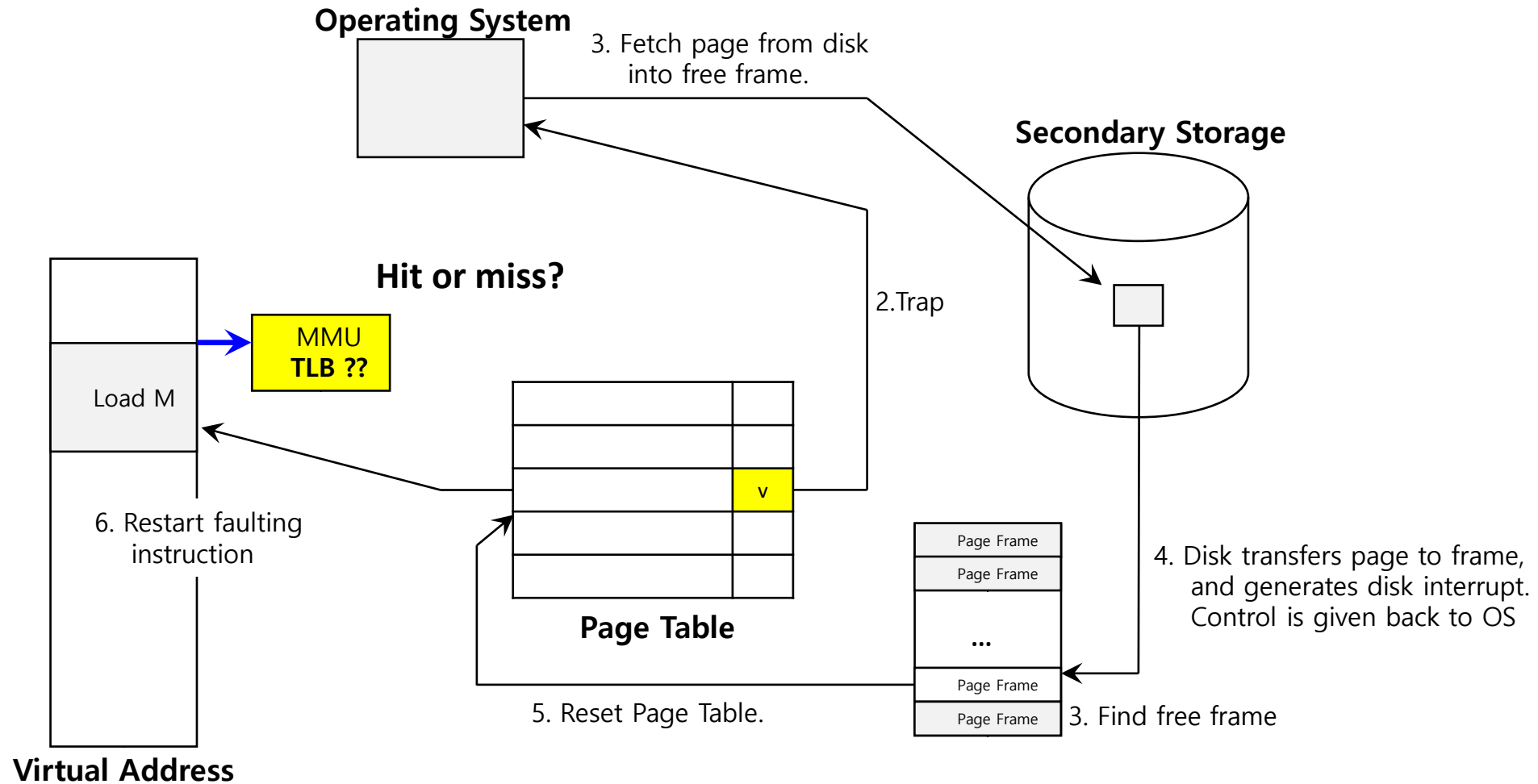
Demand Paging Summary



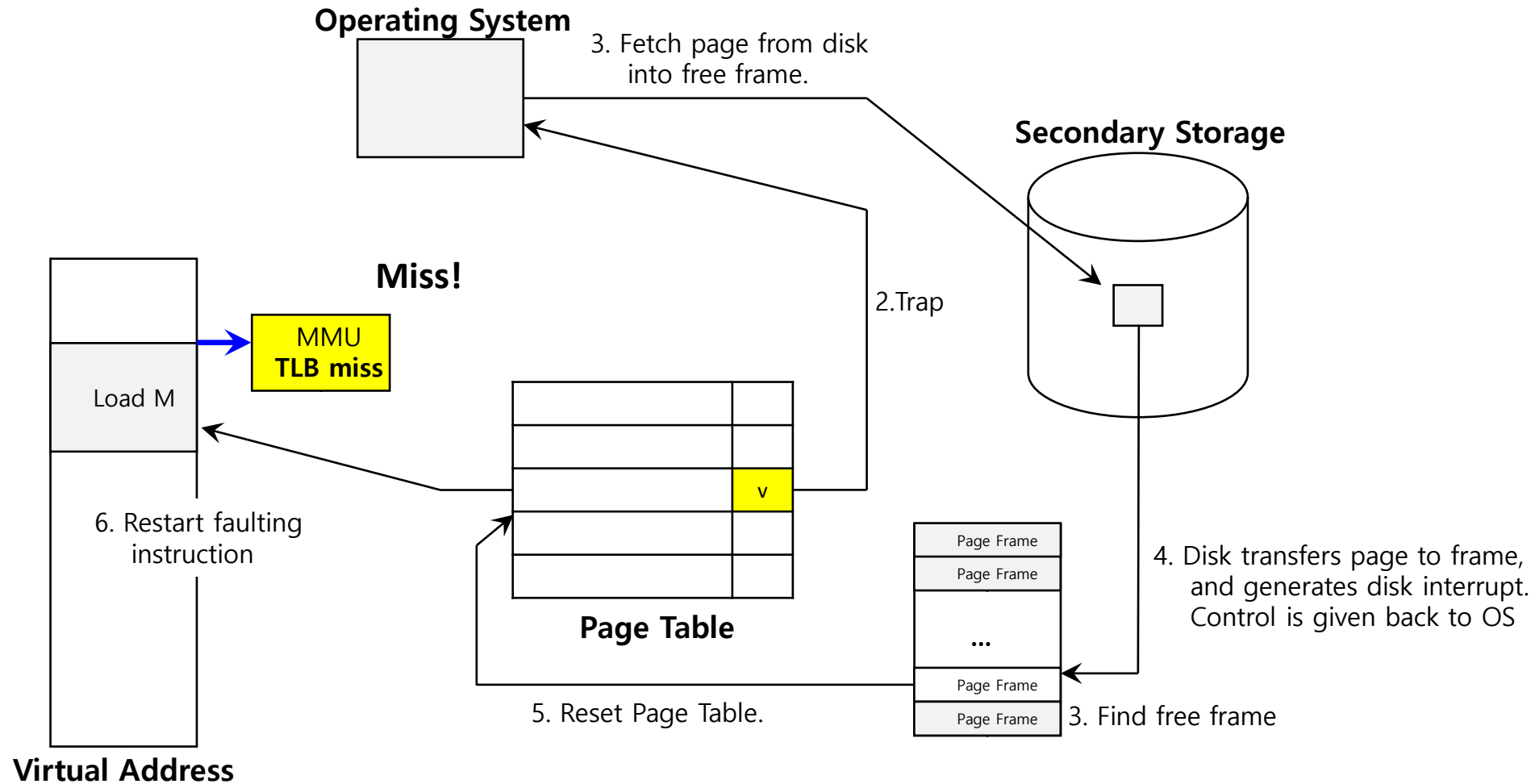
Demand Paging Summary



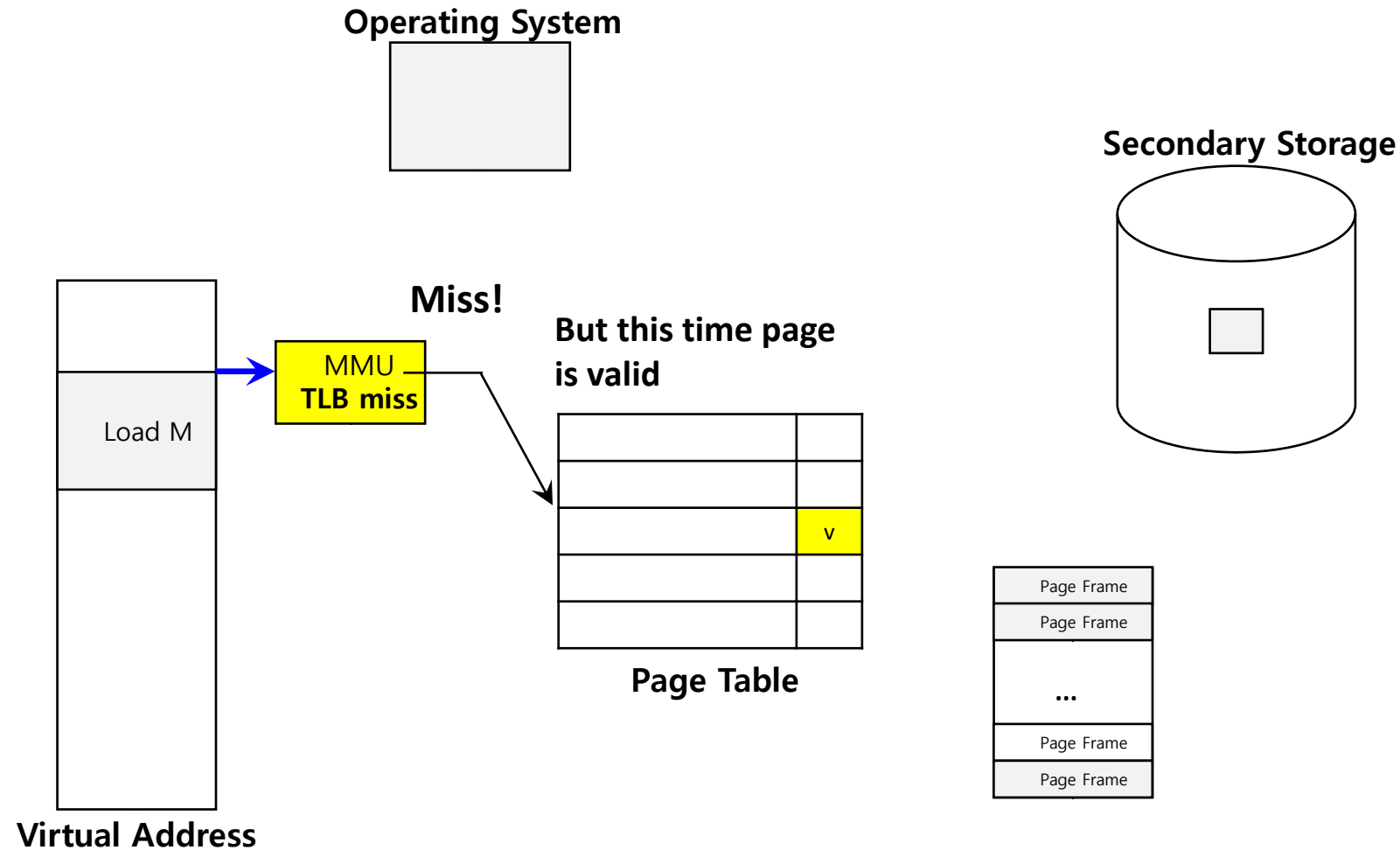
Demand Paging Summary



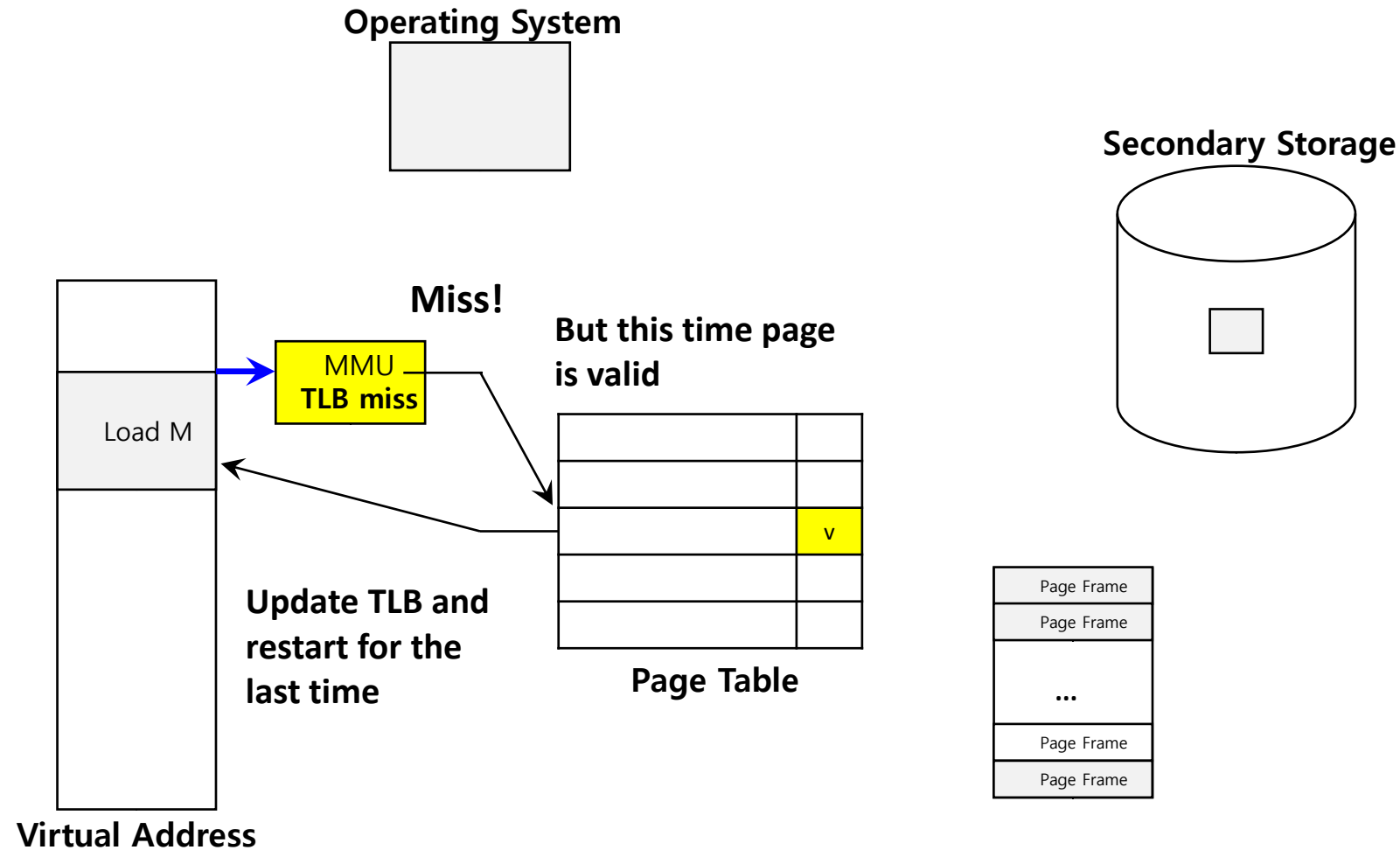
Demand Paging Summary



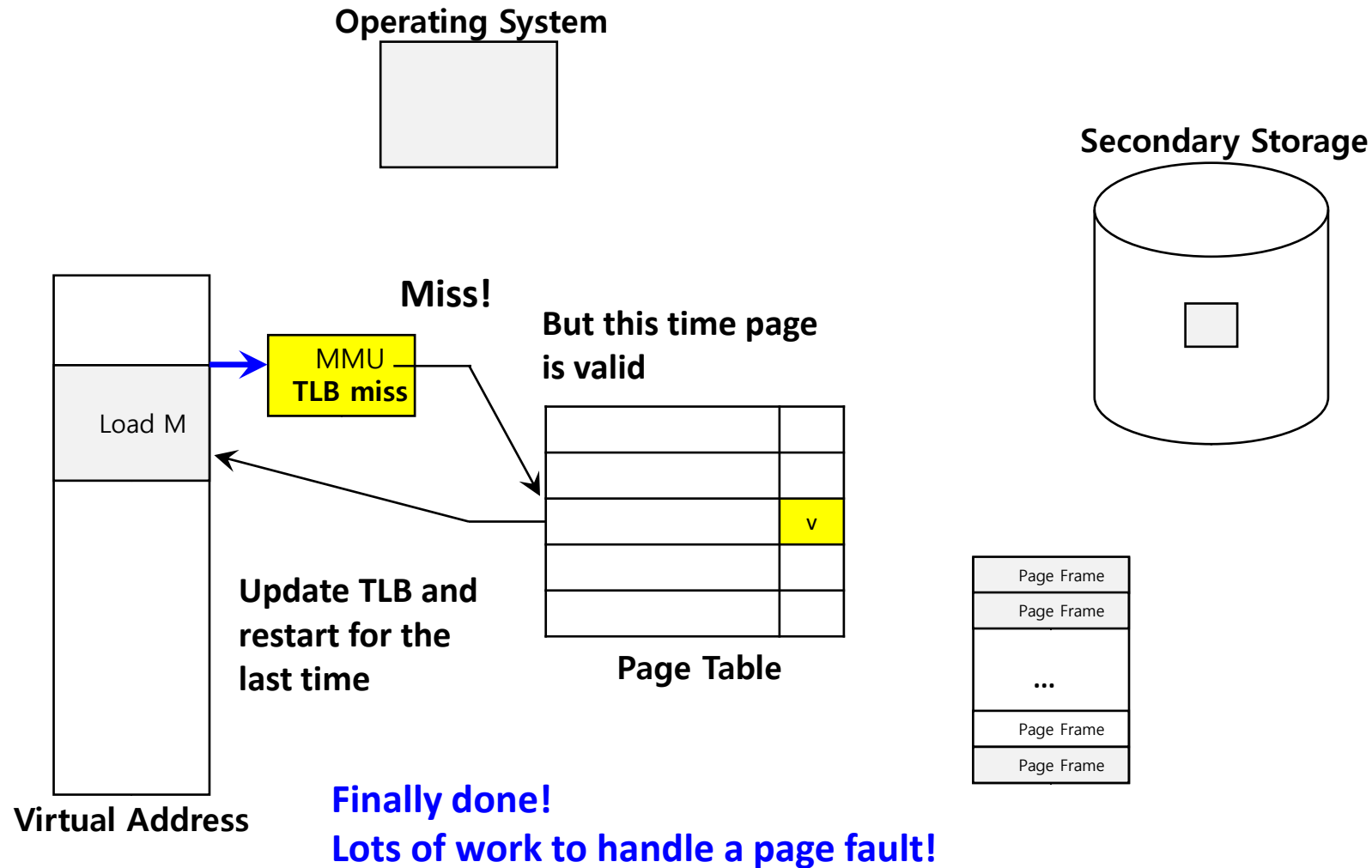
Demand Paging Summary



Demand Paging Summary



Demand Paging Summary



Remember Assumption: Getting the Page from Disk

“Assume (for now) there is at least one free frame in memory”

If no free frame available:

- Pick a frame to be replaced
- Invalidate its page table entry (and TLB entry)
- You may have to write that frame to disk

Remember Assumption: Getting the Page from Disk

“Assume (for now) there is at least one free frame in memory”

If no free frame available:

- Pick a frame to be replaced
- Invalidate any of its page table entries (and TLB entries)
 - OS needs a “frame table” to map frames to the page tables (entries) that use them
- You may have to write that frame to disk
 - Naively: always write
 - Better plan: remember if each frame has been modified with “modified bit”
 - If set, write out page to disk
 - If not, proceed with page fault handling

Remember Assumption: Getting the Page from Disk

“Assume (for now) there is at least one free frame in memory”

If no free frame available:

- Pick a frame to be replaced
 - **How?**
- Invalidate any of its page table entries (and TLB entries)
 - OS needs a “frame table” to map frames to the page tables (entries) that use them
- You may have to write that frame to disk
 - Naively: always write
 - Better plan: remember if each frame has been modified with “modified bit”
 - If set, write out page to disk
 - If not, proceed with page fault handling

How to pick with page/frame to replace?

Different page replacement policies:

- Random
- FIFO (First In, First Out)
- OPT
- LRU

Page Faults and Performance

- Normal memory access
 - ~ nanoseconds
- Faulting memory access
 - Disk I/O ~ 10 milliseconds
 - Factor of 1000000 or worse!
- Too many page faults -> **program very slow**
- **Hence, importance of good page replacement policy**

Page Replacement Policies

- Random
- FIFO (First In, First Out)
- OPT
- LRU

Page Replacement Policies

- Random
- FIFO (First In, First Out)
- OPT
- LRU

Plus, in general, prefer replacing clean over dirty

- 1 disk i/o instead of 2

Random

Random page is replaced

+ Easy to implement

⦿ Does not take advantage of spatial/temporal locality.

FIFO

Oldest page is replaced

- Age = Time since brought into memory

+ Easy to implement

- Keep a queue of pages
- Bring in a page: stick at the end of the queue
- Need replacement: pick head of queue

+ Fair

- All pages receive equal residency

☹ Does not take into account “hot” pages that may always be needed

Quiz: FIFO Page Replacement

reference string

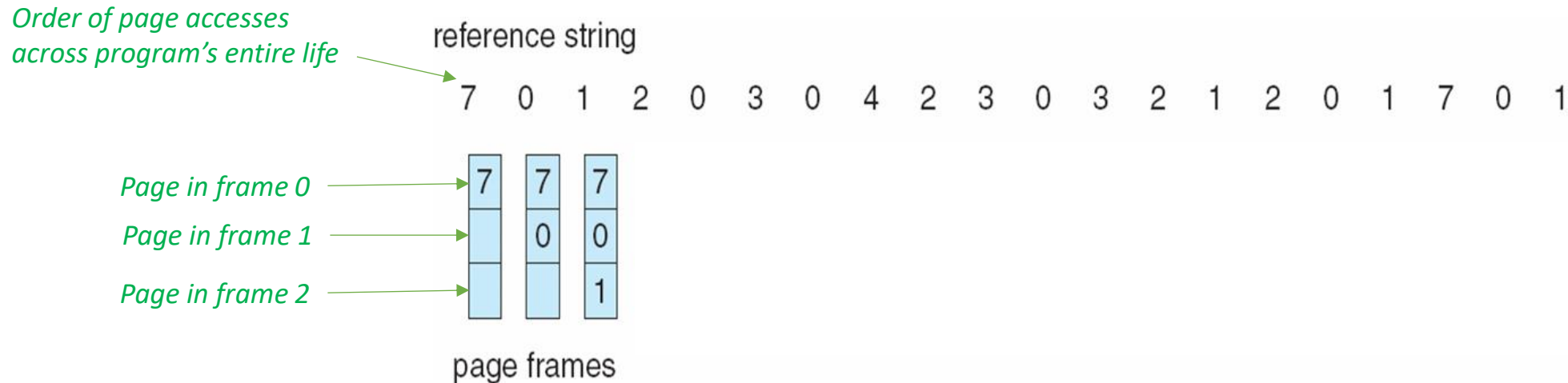
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7
	0	0
		1

page frames

?? page faults (not counting initial paging in)

Quiz: FIFO Page Replacement



?? page faults (not counting initial paging in)

Quiz: FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2																
	0	0	0	2	2	4	4	4	0										
		1	1	3	3	3	2	2	2										
				1	0	0	0	3	3										

page frames

12 page faults (not counting initial paging in)

OPT: An Optimal Algorithm

Replace the page that will be referenced the furthest in the future

+ Provably optimal

☹️ Can't implement (Can't predict the future)

(We mentioned previously that optimal algorithms often don't exist. It's because we can't predict the future of the workload. We have to do our best to guess the future based on the past!)

A basis of comparison for other algorithms



Quiz: Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7
	0	0
		1

page frames

?? page faults (not counting initial paging in)

Quiz: Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2	2		2		2							7
	0	0	0		0	4		0		0							0
		1	1		3	3		3		1							1

page frames

6 page faults (not counting initial paging in)

LRU: Least Recently Used

Cannot look into the future, but can try to predict future using past
Replace least recently accessed page

+ With locality, LRU approximates OPT

☹ Harder to implement, must track which pages have been
accessed **and when**

☹ Does not handle all workloads well

- Example: Large array scans that repeat. Popular in DBMS.

LRU Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7
	0	0
		1

page frames

?? page faults (not counting initial paging in)

LRU Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

page frames

9 page faults (not counting initial paging in)

LRU Implementation

- Too expensive to implement exactly
 - Need to timestamp every memory reference
- But can be (well) approximated

LRU Approximation with Hardware Support

Use a **reference bit**

- Bit in page table
- Hardware sets bit when page is referenced

Periodically

- Read out and store all reference bits
- Reset all reference bits to zero

Keep all reference bits for some time (newest bit = most significant)

- The more bits kept, the better approximation

Replacement

- **Page with smallest value of reference bit history**

Let's practice!

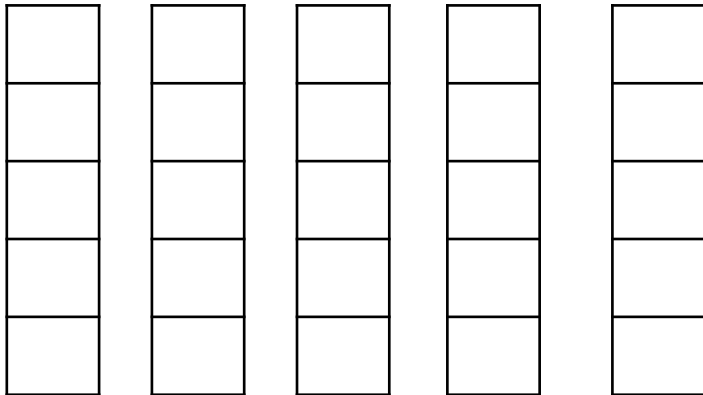
Page Replacement Policies

Consider a **memory with 5 pages**.

1. Generate worst-case address reference streams for FIFO and LRU. Worst-case reference streams cause the most misses possible.
2. Compare with OPT in all scenarios from point 1.
3. For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?

Page Replacement Policies

Worst case FIFO:



Page Replacement Policies

Worst case FIFO:

1 2 3 4 5 6 1 2 3 4 5 6

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

Page Replacement Policies

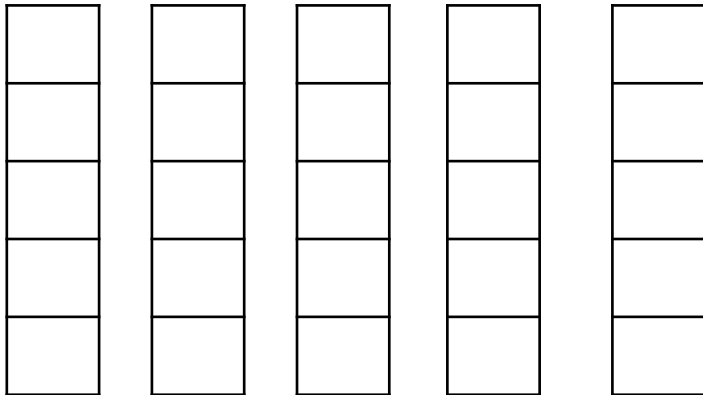
Worst case FIFO:

1 2 3 4 5 6 1 2 3 4 5 6

1	1	1	1	1	6	6	6	6	6	5	5
	2	2	2	2	2	1	1	1	1	1	6
		3	3	3	3	3	2	2	2	2	2
			4	4	4	4	4	3	3	3	3
				5	5	5	5	5	4	4	4

Page Replacement Policies

Worst case LRU:



Page Replacement Policies

Worst case LRU:

1 2 3 4 5 6 1 2 3 4 5 6

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

Page Replacement Policies

Worst case LRU:

1 2 3 4 5 6 1 2 3 4 5 6

1	1	1	1	1	6	6	6	6	6	5	5
	2	2	2	2	2	1	1	1	1	1	6
		3	3	3	3	3	2	2	2	2	2
			4	4	4	4	4	3	3	3	3
				5	5	5	5	5	4	4	4

Page Replacement Policies

OPT:

1 **2** **3** **4** **5** 6 1 2 3 4 5 6

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

Page Replacement Policies

OPT:

1 2 3 4 5 6 1 2 3 4 5 6

1

1
2

1
2
3

1
2
3
4

1
2
3
4
5

1
2
3
4
6

5
2
3
4
6

Page Replacement Policies

OPT:

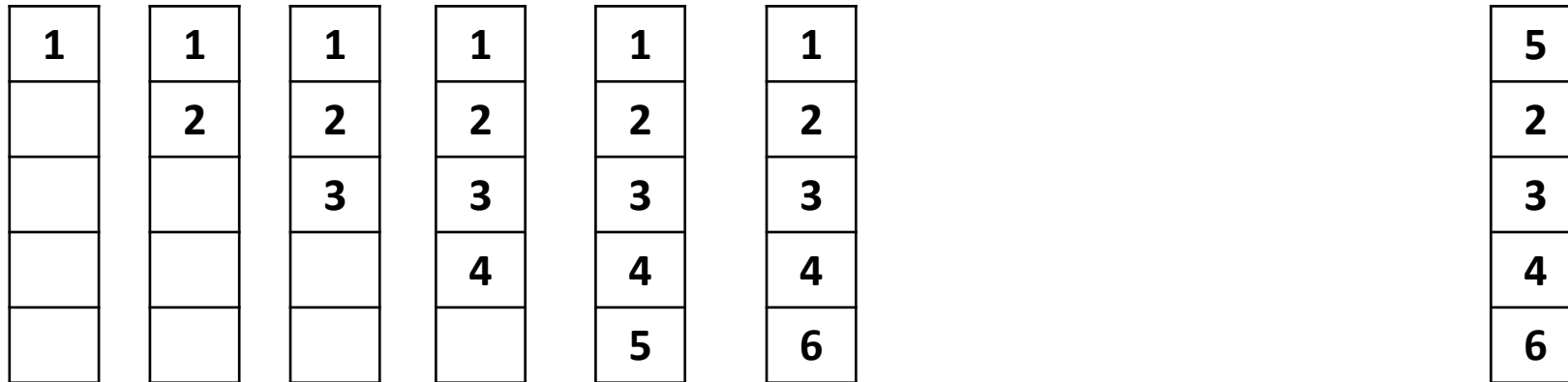
1 2 3 4 5 6 1 2 3 4 5 6

1	1	1	1	1	1							5
	2	2	2	2	2							2
		3	3	3	3							3
			4	4	4							4
				5	6							6

For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?

Page Replacement Policies

OPT:



For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?

→ Need a cache of 6

Page Replacement Policies

FIFO/LRU, cache of 6:

1 2 3 4 5 6 1 2 3 4 5 6

1	1	1	1	1	1
	2	2	2	2	2
		3	3	3	3
			4	4	4
				5	5
					6

Aside: Bélády's Paradox (aka Bélády's Anamoly)

- If #frames increases, is it guaranteed that #faults decreases?
 - ... for OPT?
 - ... for LRU?
 - ... for FIFO?

Aside: Bélády's Paradox (aka Bélády's Anamoly)

- If #frames increases, is it guaranteed that #faults decreases?
 - ... for OPT?
 - YES
 - ... for LRU?
 - YES
 - ... for FIFO?
 - NO

Summary – Key Concepts

- TLB
- Page Table for very large address spaces
- Demand paging
 - Page fault
- Page replacement policies:
 - Random
 - FIFO (First In, First Out)
 - OPT
 - LRU (Least Recently Used)

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).