CS 5600 Computer Systems

Spring 2026

Virtualization: Memory Paging
Lecture 5
February 10, 2026

Prof. Scott Valcourt

s.valcourt@northeastern.edu

603-380-2860 (cell)

Portland, ME

The Roux Institute
Northeastern University

Concept of Paging

- Paging **splits up** address space into **fixed-zed** unit called a **page**.
  - Segmentation: variable size of logical segments(code, stack, heap, etc.)

- With paging, **physical memory** is also **split** into some number of pages called a **page frame**.

- **Page table** per process is needed **to translate** the virtual address to physical address.

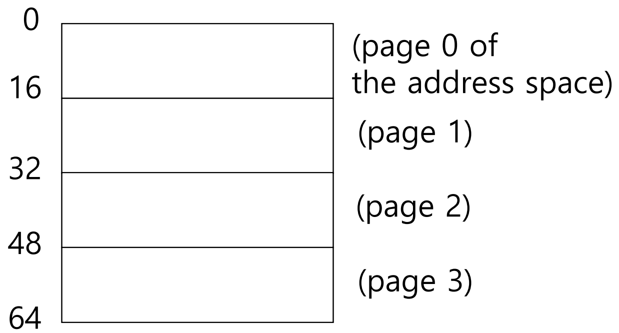
Advantages of Paging

- **Flexibility**: Supporting the abstraction of address space effectively
  - Don’t need assumption how heap and stack grow and are used.

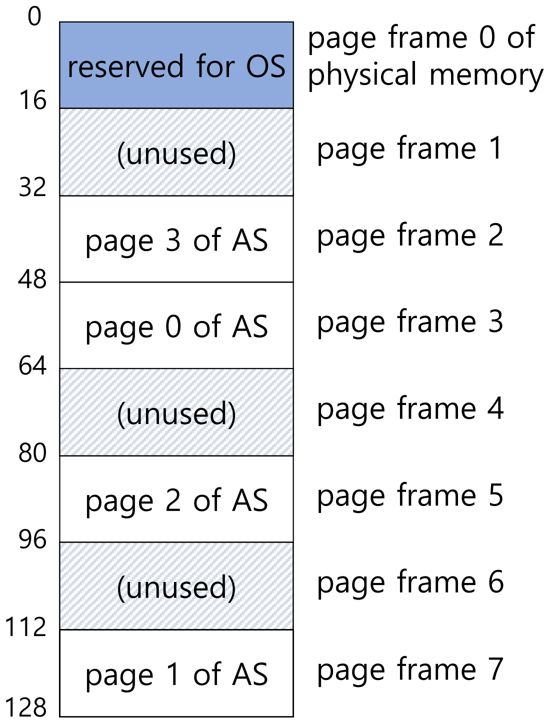
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

- 128-byte physical memory with 16-byte page frames
- 64-byte address space with 16-byte pages



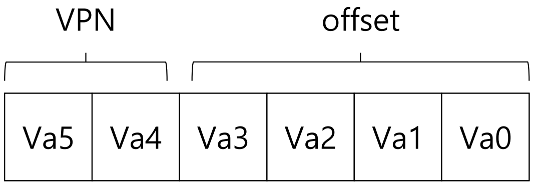
A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

Address Translation

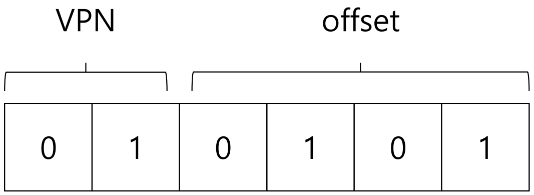
- Two components in the virtual address
  - VPN: virtual page number
  - Offset: offset within the page



VPN | offset

Va5	Va4	Va3	Va2	Va1	Va0

- Example: virtual address 21 in 64-byte address space

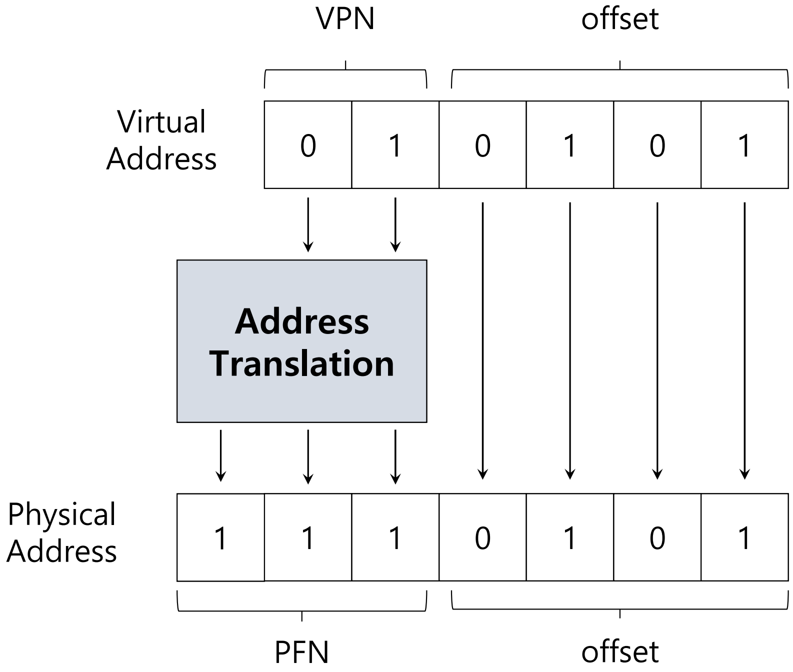


VPN | offset

0	1	0	1	0	1

Example: Address Translation

- The virtual address 21 in 64-byte address space

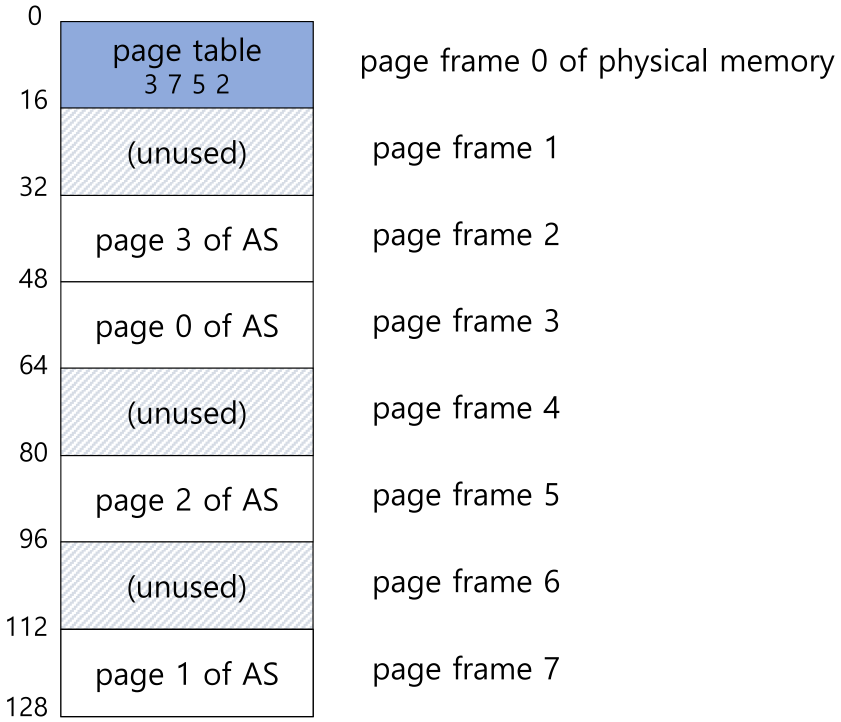


Where are Page Tables Stored?

- Page tables can get awfully large
  - 32-bit address space with 4-KB pages, 20 bits for VPN
    - $4MB = \ 2^{20}\ entries\ \ * 4\ Bytes\ per\ page\ table\ entry$

- Page tables for each process are stored in memory.

Example: Page Table in Kernel Physical Memory



0	page table 3 7 5 2	page frame 0 of physical memory
16	(unused)	page frame 1
32	page 3 of AS	page frame 2
48	page 0 of AS	page frame 3
64	(unused)	page frame 4
80	page 2 of AS	page frame 5
96	(unused)	page frame 6
112	page 1 of AS	page frame 7
128		

Physical Memory

What is in the Page Table?

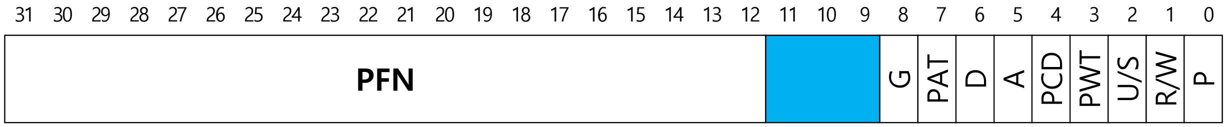
- The page table is just a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

- The OS **indexes** the array by VPN and looks up the page-table entry.

Common Flags of Page Table Entries

- **Valid Bit**: Indicating whether the particular translation is valid
- **Protection Bit**: Indicating whether the page could be read from, written to, or executed from
- **Present Bit**: Indicating whether this page is in physical memory or on disk (swapped out)
- **Dirty Bit**: Indicating whether the page has been modified since it was brought into memory
- **Reference Bit (Accessed Bit):** Indicating that a page has been accessed

Example: x86 Page Table Entry



An x86 Page Table Entry(PTE)

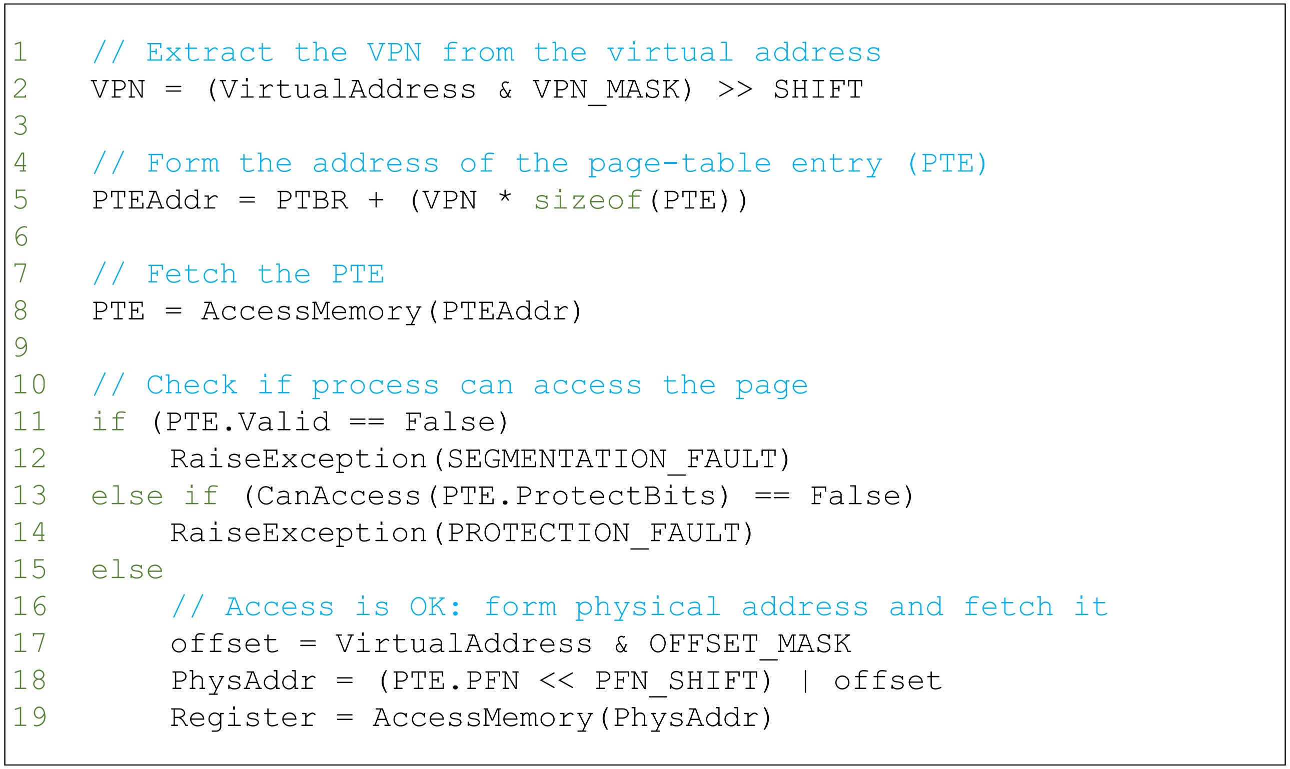
- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

Paging: Too Slow

- To find a location of the desired PTE, the **starting location** of the page table is **needed**.

- For every memory reference, paging requires the OS to perform one **extra memory reference**.

Accessing Memory With Paging



```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

A Memory Trace

- Example: A Simple Memory Access

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

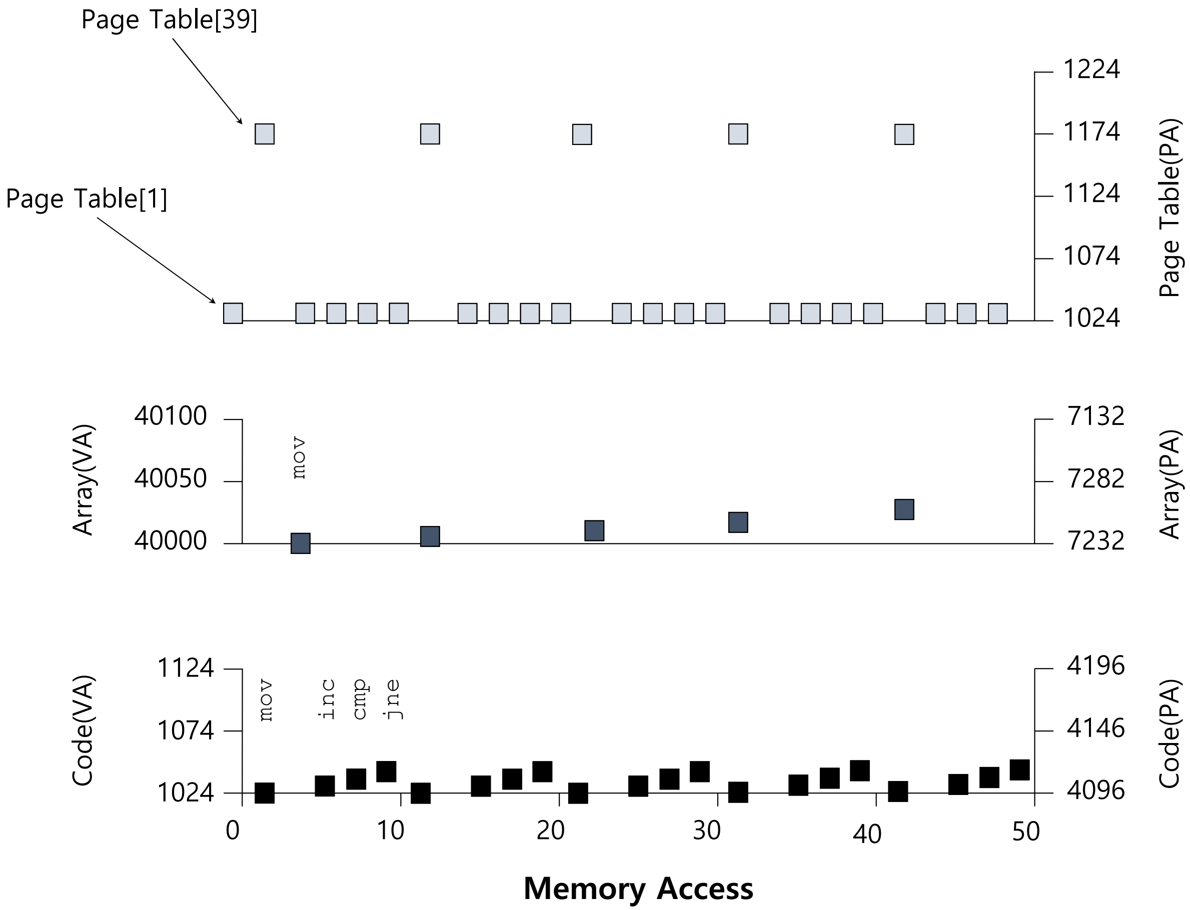
- Compile and execute

```
prompt> gcc -o array array.c -Wall -o
prompt>./array
```

- Resulting Assembly code

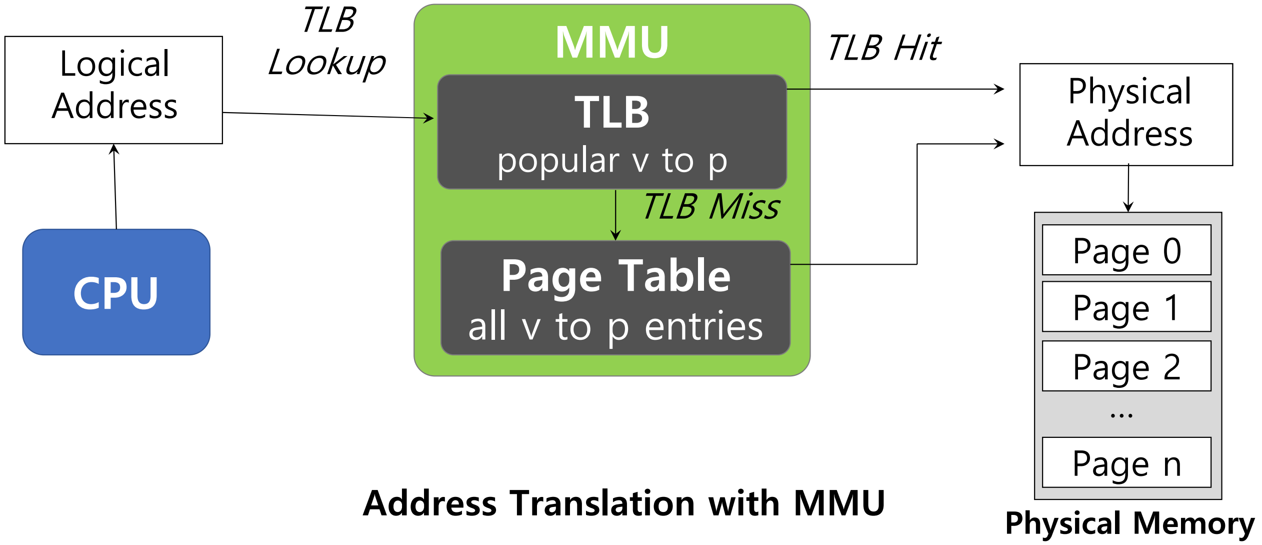
```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

A Virtual (and Physical) Memory Trace



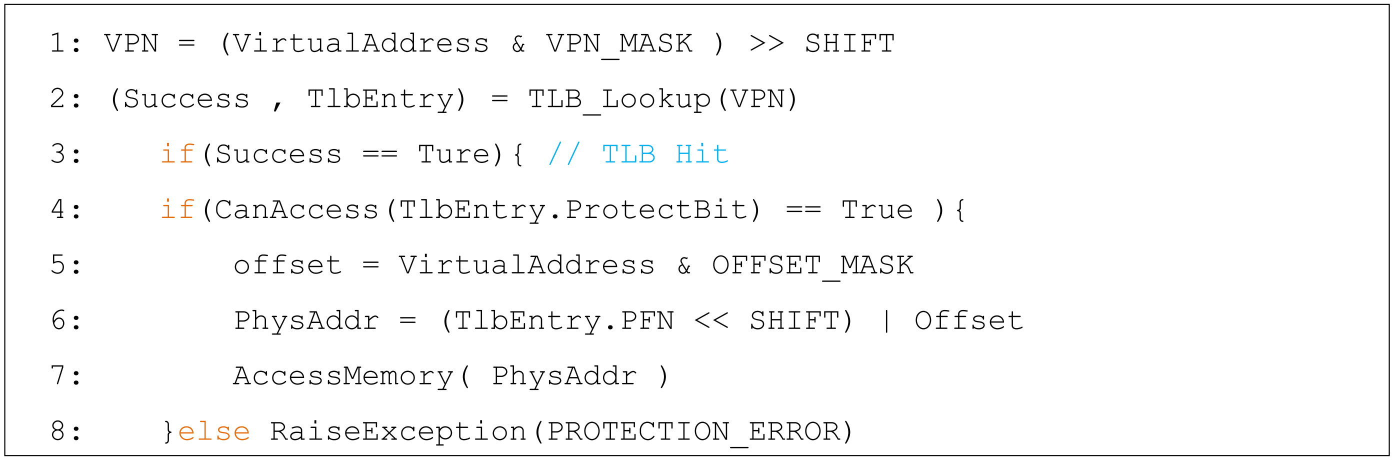
Translation Lookaside Buffer (TLB)

- Part of the chip’s memory-management unit(MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

TLB Basic Algorithms



```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == Ture){ // TLB Hit
4:    if(CanAccess(TlbEntry.ProtectBit) == True ){
5:        offset = VirtualAddress & OFFSET_MASK
6:        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:        AccessMemory( PhysAddr )
8:    }else RaiseException(PROTECTION_ERROR)
```

- (line 1) extract the virtual page number (VPN).
- (line 2) check if the TLB holds the translation for this VPN.
- (lines 5-8) extract the page frame number from the relevant TLB entry, and from the desired physical address and access memory.

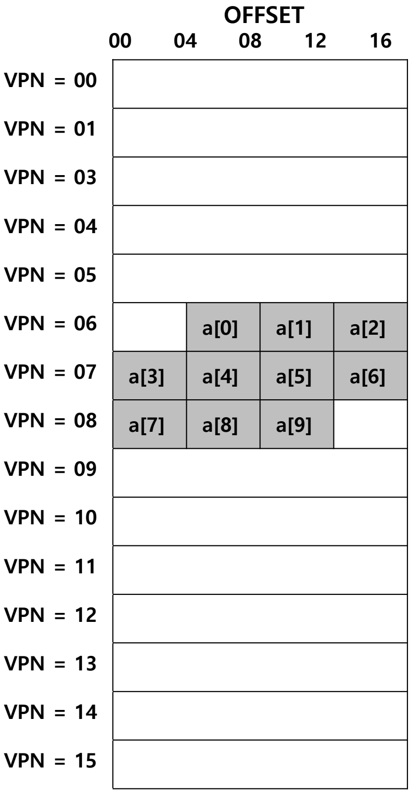
TLB Basic Algorithms (Cont.)

```
11:    }else{ //TLB Miss
12:        PTEAddr = PTBR + (VPN * sizeof(PTE))
13:        PTE = AccessMemory(PTEAddr)
14:        (…)
15:    }else{
16:        TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:        RetryInstruction()
18:    }
19:}
```

- (lines 11-12) The hardware accesses the page table to find the translation.
- (line 16) updates the TLB with the translation.

Example: Accessing An Array

- How a TLB can improve its performance



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					

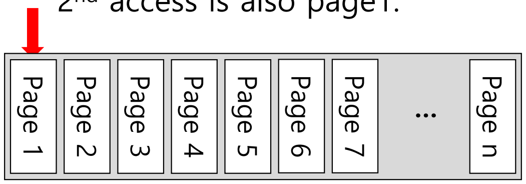
```
int sum = 0 ;
for( i=0; i<10; i++){
      sum+=a[i];
}
```

The TLB improves performance due to spatial locality

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

Locality

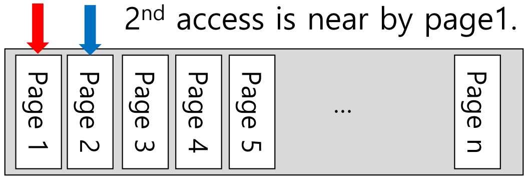
- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.



1st access is page1.
2nd access is also page1.

Virtual Memory

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



1st access is page1.
2nd access is near by page1.

Virtual Memory

Who Handles the TLB Miss?

- Hardware handle the TLB miss entirely on CISC.
  - The hardware has to know exactly where the page tables are located in memory.
  - The hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
  - **hardware-managed TLB.**

Who Handles the TLB Miss? (Cont.)

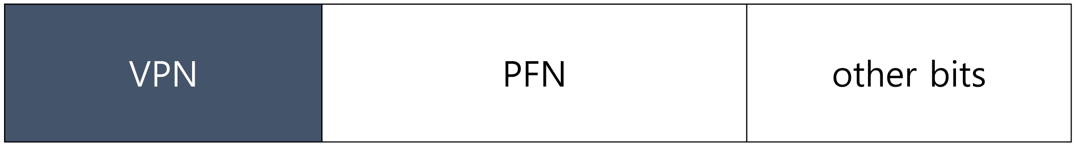
- RISC have what is known as a **software-managed TLB.**
  - On a TLB miss, the hardware raises exception(trap handler).
    - **Trap handler is code** within the OS that is written with the express purpose of handling TLB miss.

TLB Control Flow Algorithm (OS Handled)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
         Offset = VirtualAddress & OFFSET_MASK
         PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
         Register = AccessMemory(PhysAddr)
    else
         RaiseException(PROTECTION_FAULT)
else // TLB Miss
     RaiseException(TLB_MISS)
```

TLB Entry

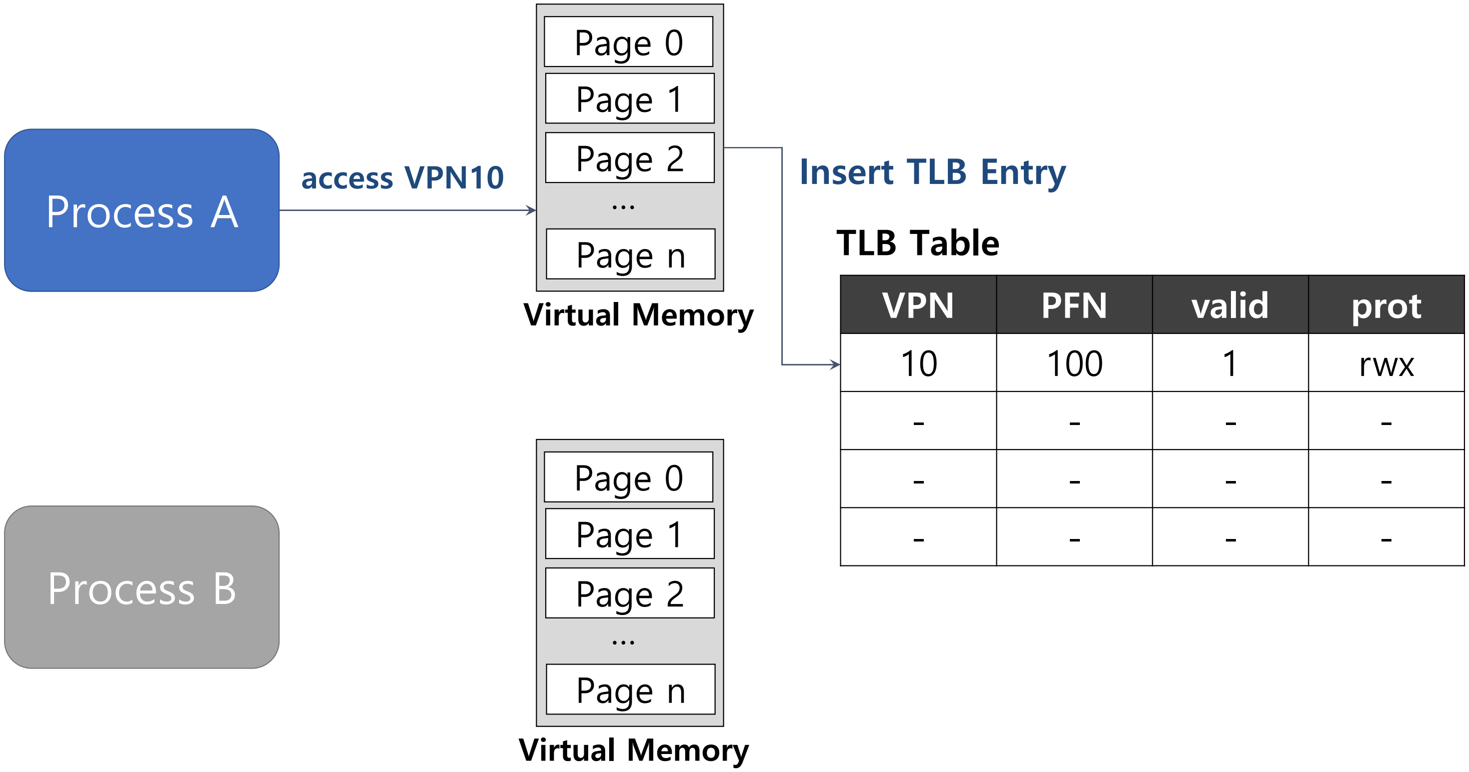
- TLB is managed by a **Full Associative** method.
  - A typical TLB might have 32, 64, or 128 entries.
  - Hardware search the entire TLB in parallel to find the desired translation.
  - other bits: valid bits, protection bits, address-space identifier, dirty bit



VPN	PFN	other bits

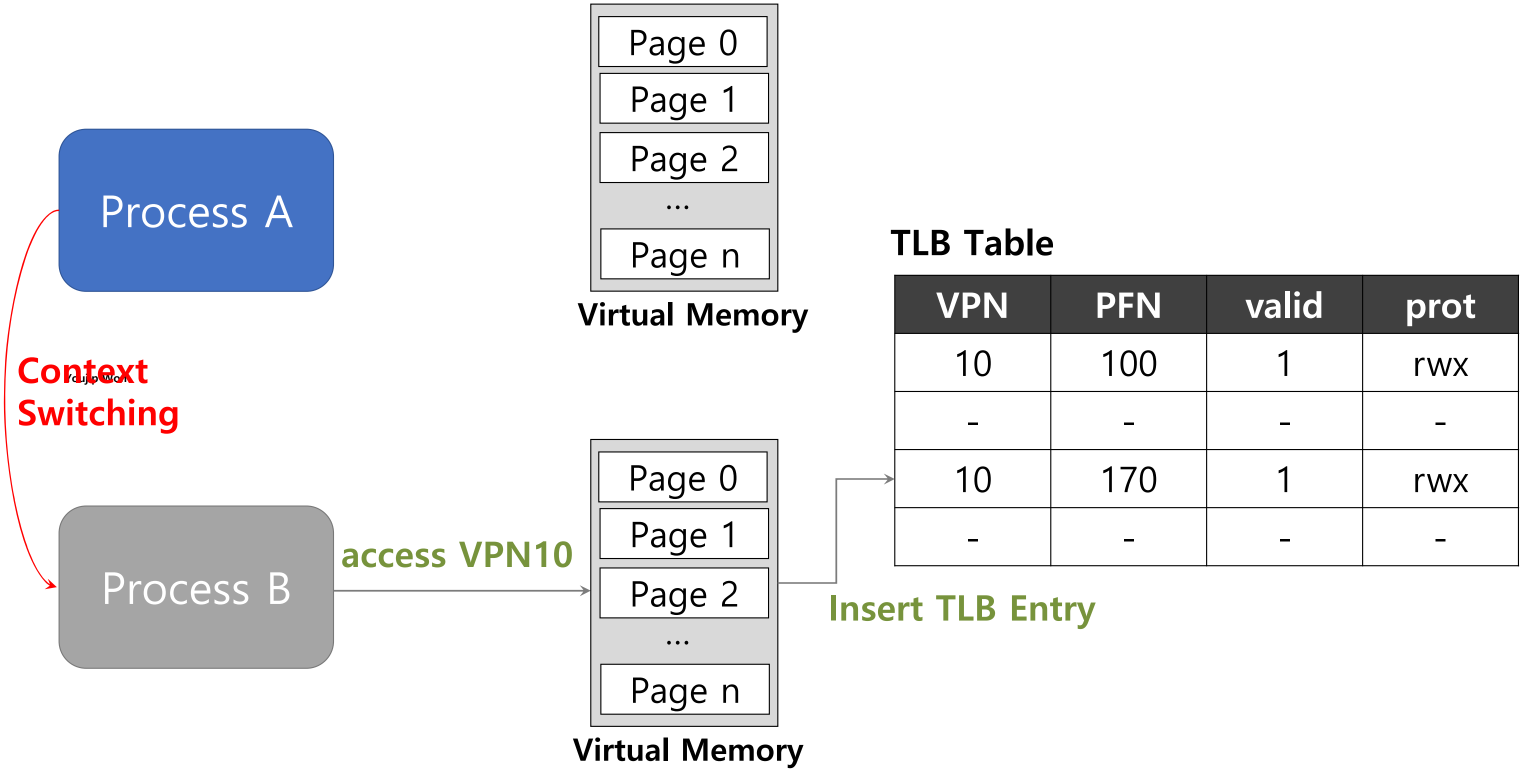
Typical TLB entry look like this

TLB Issue: Context Switching



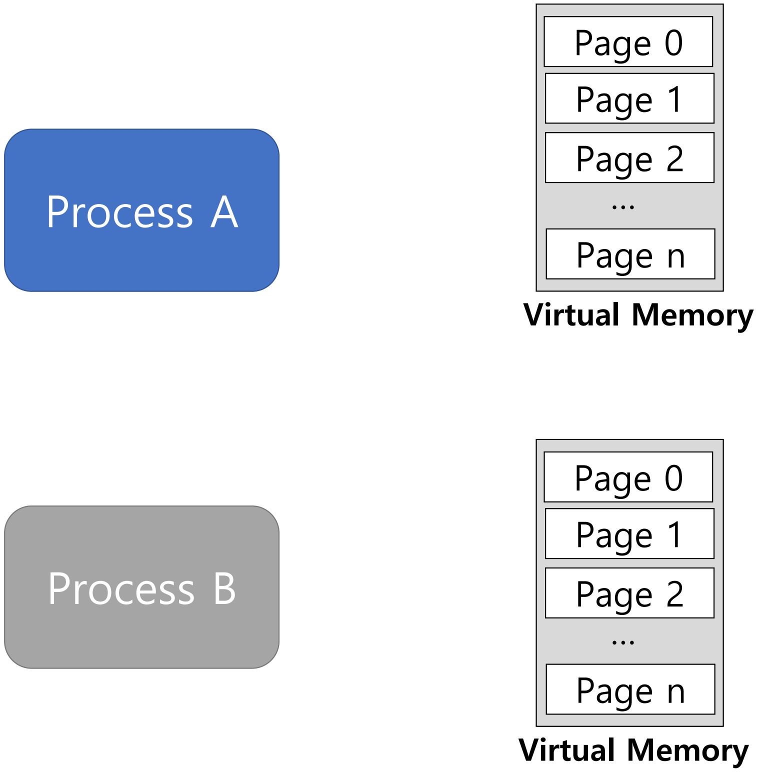
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
-	-	-	-
-	-	-	-

TLB Issue: Context Switching



VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

TLB Issue: Context Switching


Process A

Page 0

Page 1

Page 2

…

Page n

Virtual Memory

Process B

Page 0

Page 1

Page 2

…

Page n

Virtual Memory

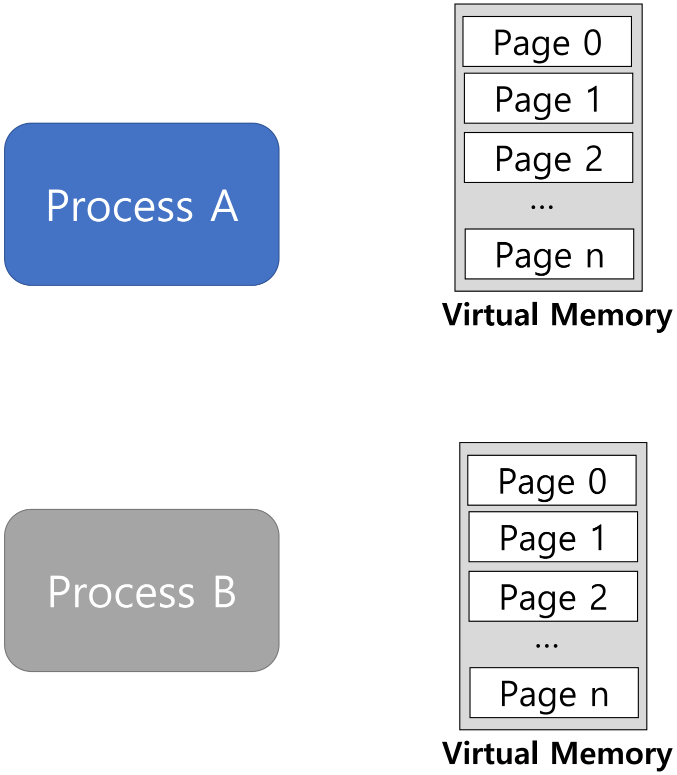
TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't distinguish which entry is meant for which process

To Solve the Problem

- Provide an address space identifier(ASID) field in the TLB.



TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Another Case

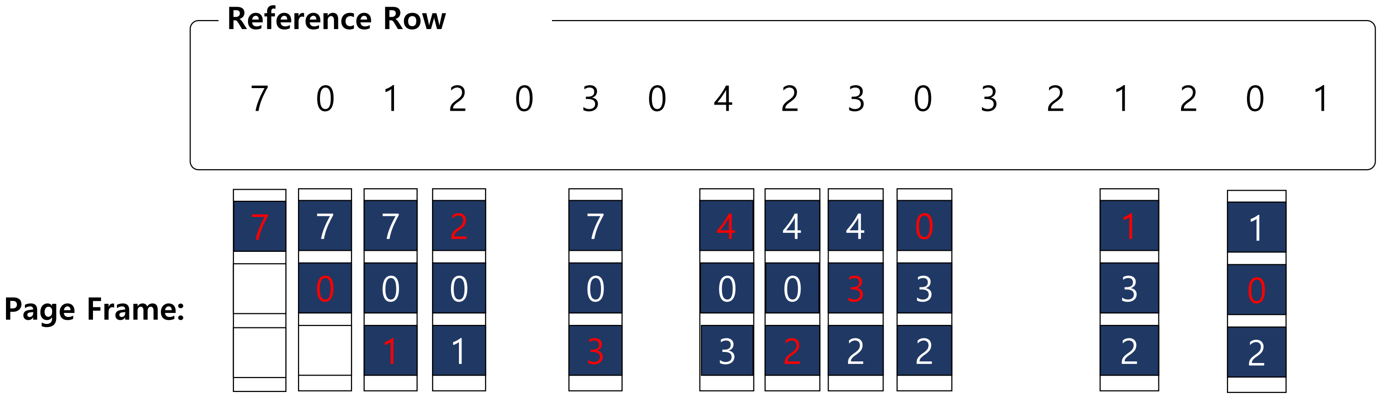
- Two processes share a page.
  - Process 1 is sharing physical page 101 with Process2.
  - P1 maps this page into the 10th page of its address space.
  - P2 maps this page to the 50th page of its address space.

VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Replacement Policy

- LRU (Least Recently Used)
  - Evict an entry that has not recently been used.
  - Take advantage of *locality* in the memory-reference stream.

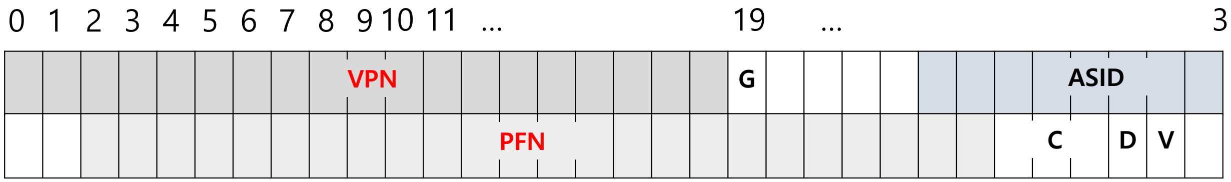


Reference Row	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
Page Frame:	7	7	7	2		7		4	4	4	0			1		1	
		0	0	0		0		0	0	3	3			3		0	
			1	1		3		3	2	2	2			2		2	

Total 11 TLB miss

A Real TLB Entry

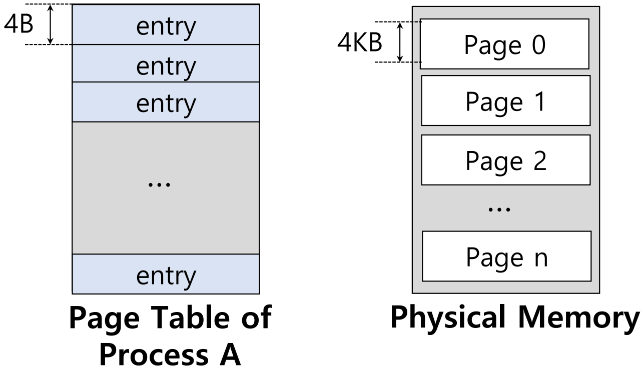
All 64 bits of this TLB entry(example of MIPS R4000)



Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory($2^{24} * 4KB$ pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

Paging: Linear Tables

- We usually have one page table for every process in the system.
  - Assume that 32-bit address space with 4KB pages and 4-byte page-table entry.

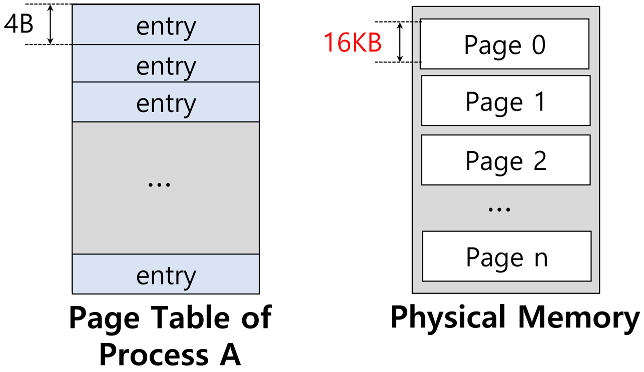


Page table size = $\frac{2^{32}}{2^{12}} * 4Byte = 4MByte$

Page tables are too big and thus consume too much memory.

Paging: Smaller Tables

- Page table are too big and thus consume too much memory.
  - Assume that 32-bit address space with 16KB pages and 4-byte page-table entry.

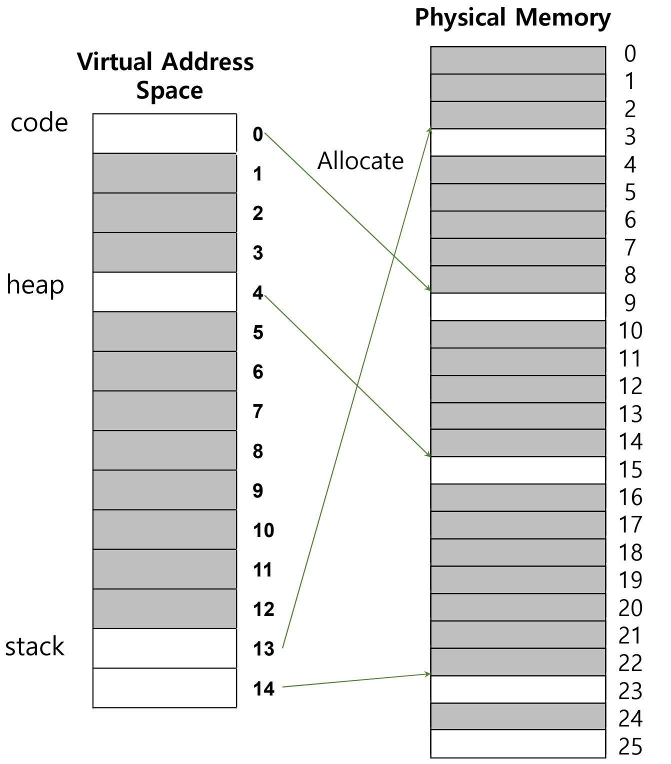


$\frac{2^{32}}{2^{16}} * 4 = 1MB$ per page table

Big pages lead to internal fragmentation.

Problem

- Single page table for the entries address space of process.



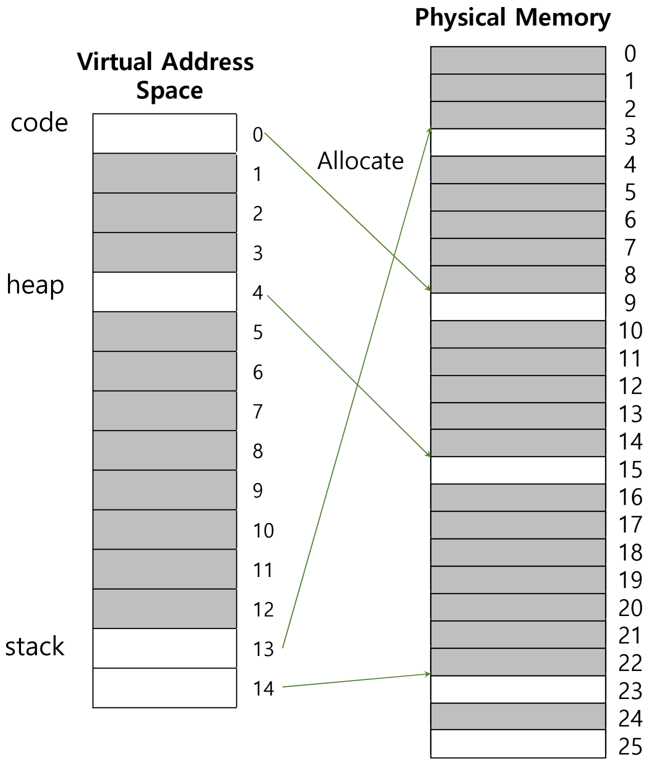
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

Problem

- Most of the page table is **unused**, full of invalid entries.



A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

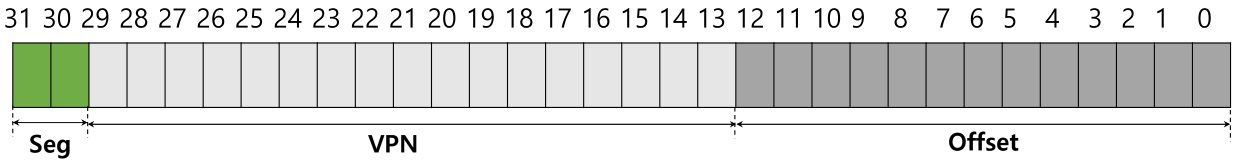
A Page Table For 16KB Address Space

Hybrid Approach: Paging and Segments

- In order to reduce the memory overhead of page tables.
  - Using base not to point to the segment itself but rather to hold the physical address of the page table of that segment.
  - The bounds register is used to indicate the end of the page table.

Simple Example of Hybrid Approach

- Each process has three page tables associated with it.
  - When process is running, the base register for each of these segments contains the physical address of a linear page table for that segment.



32-bit Virtual address space with 4KB pages

Seg value	Content
00	unused segment
01	code
10	heap
11	stack

TLB miss on Hybrid Approach

- The hardware get to **physical address** from **page table**.
  - The hardware uses the segment bits(SN) to determine which base and bounds pair to use.
  - The hardware then takes the physical address therein and combines it with the VPN as follows to form the address of the page table entry(PTE) .

```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

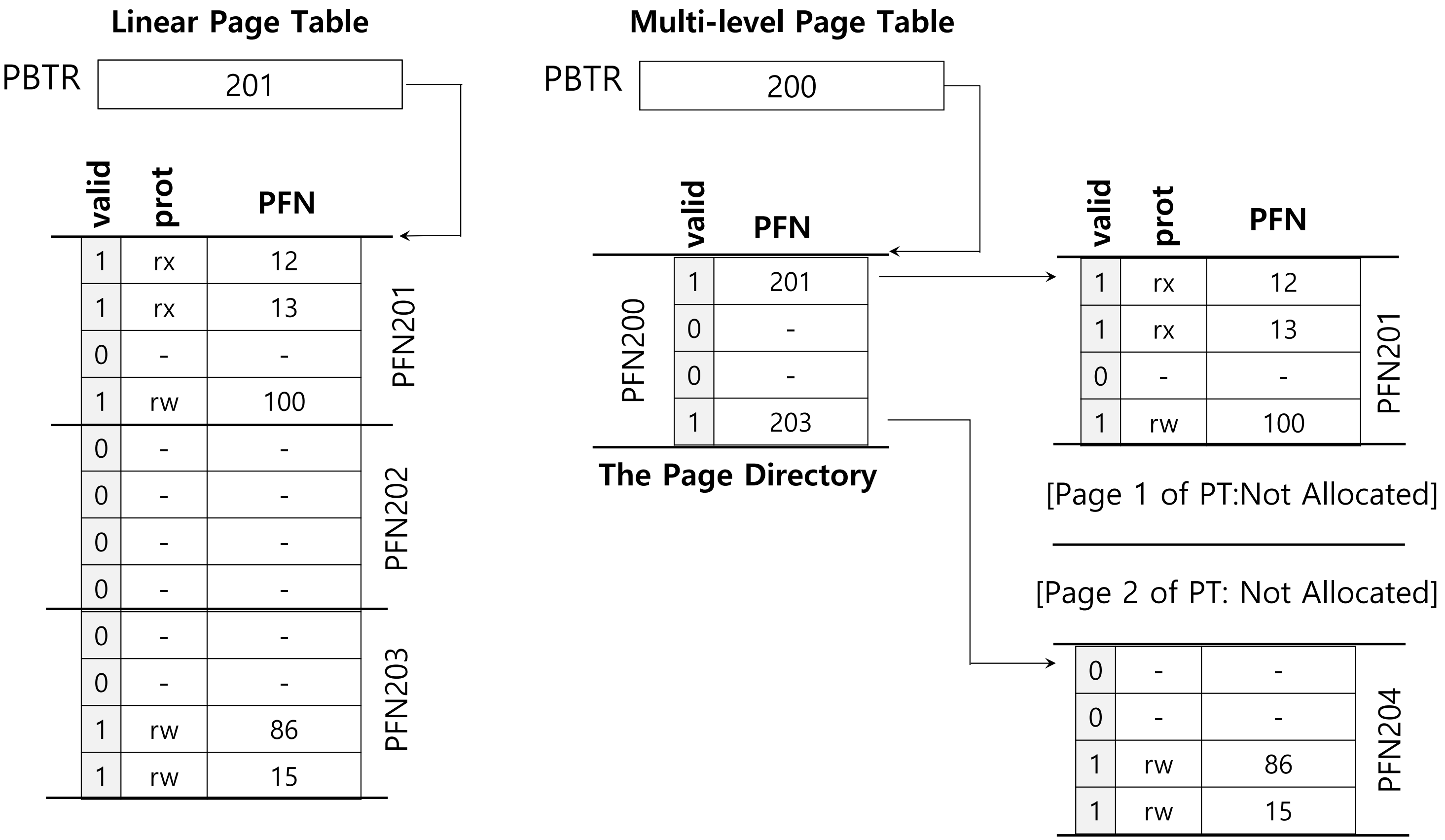
Problem of Hybrid Approach

- Hybrid Approach is not without problems.
  - If we have a large but sparsely-used heap, we can still end up with a lot of page table waste.
  - Causing external fragmentation to arise again.

Multi-level Page Tables

- Turns the linear page table into something like a tree.
  - 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 whether a page of the page table is valid, use a new structure, called page directory.

Multi-level Page Tables: Page Directory



Linear Page Table

PBTR 201

valid	prot	PFN	
1	rx	12	PFN201
1	rx	13	
0	-	-	
1	rw	100	
0	-	-	PFN202
0	-	-	
0	-	-	
0	-	-	
0	-	-	PFN203
0	-	-	
1	rw	86	
1	rw	15	

Multi-level Page Table

PBTR 200

	valid	PFN
PFN200	1	201
	0	-
	0	-
	1	203

The Page Directory

valid	prot	PFN	
1	rx	12	PFN201
1	rx	13	
0	-	-	
1	rw	100	

[Page 1 of PT:Not Allocated]

[Page 2 of PT: Not Allocated]

0	-	-	PFN204
0	-	-	
1	rw	86	
1	rw	15	

Linear (Left) And Multi-Level (Right) Page Tables

Multi-level Page Tables: Page Directory Entries

- The page directory contains one entry per page of the page table.
  - It consists of a number of page directory entries (PDE).
- PDE has a valid bit and page frame number (PFN).

Multi-level Page Tables: Advantages & Disadvantages

- Advantage
  - Only allocates page-table space in proportion to the amount of address space you are using.
  - The OS can grab the next free page when it needs to allocate or grow a page table.

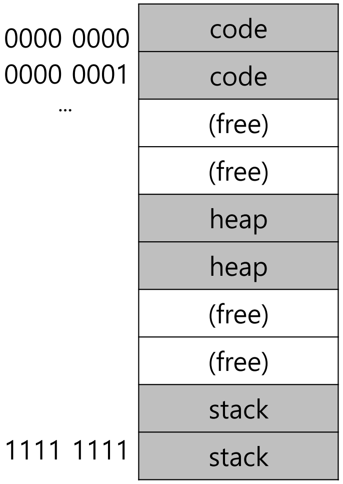
- Disadvantage
  - Multi-level table is a small example of a time-space trade-off.
  - Complexity.

Multi-level Page Table: Level of Indirection

- A multi-level structure can adjust level of indirection through use of the page directory.
  - Indirection place page-table pages wherever we would like in physical memory.

A Detailed Multi-Level Example

- To understand the idea behind multi-level page tables better, let’s do an example.



0000 0000 code

0000 0001 code

…

(free)

(free)

heap

heap

(free)

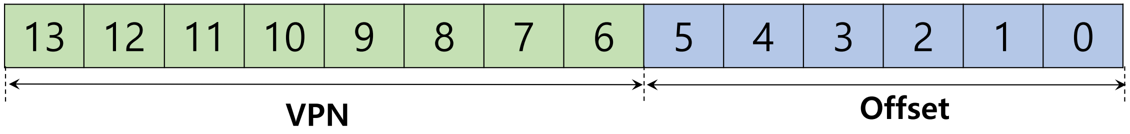
(free)

stack

1111 1111 stack

Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	2^8(256)

A 16-KB Address Space With 64-byte Pages



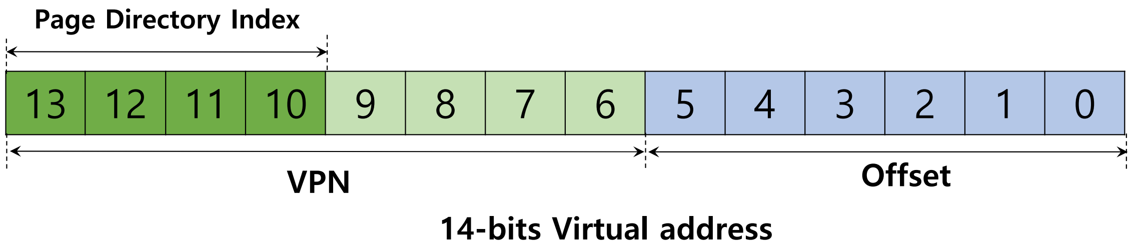
13 12 11 10 9 8 7 6 5 4 3 2 1 0

VPN

Offset

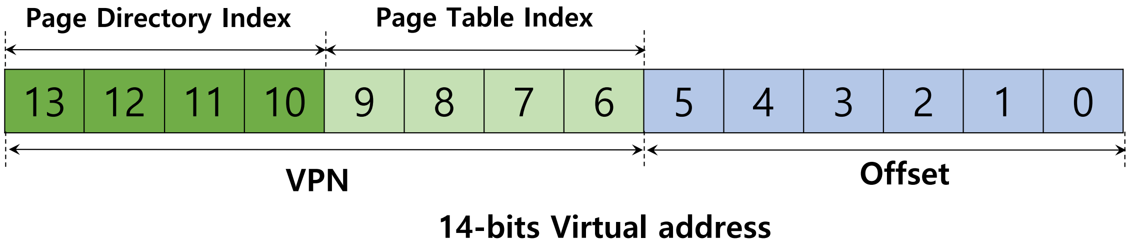
A Detailed Multi-Level Example: Page Directory Index

- The page directory needs one entry per page of the page table
  - it has 16 entries.
- The page-directory entry is invalid → Raise an exception (The access is invalid)



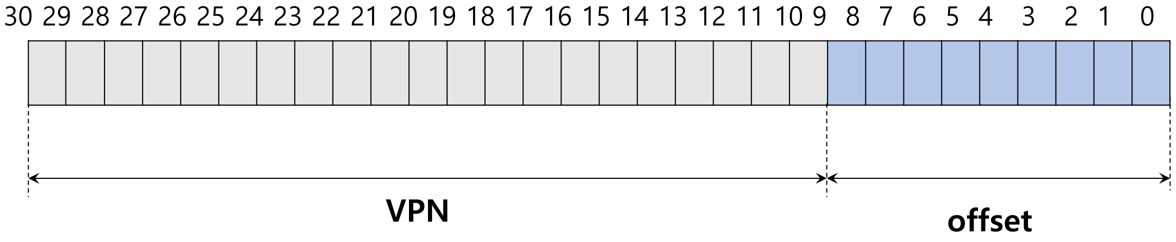
A Detailed Multi-Level Example: Page Table Index

- The PDE is valid, we have more work to do.
  - To fetch the page table entry (PTE) from the page of the page table pointed to by this page-directory entry.
- This page-table index can then be used to index into the page table itself.



More than Two Levels

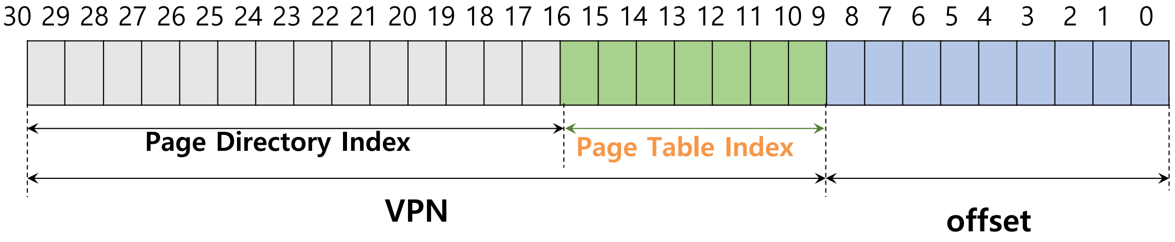
- In some cases, a deeper tree is possible.



Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit

More than Two Levels : Page Table Index

- In some cases, a deeper tree is possible.

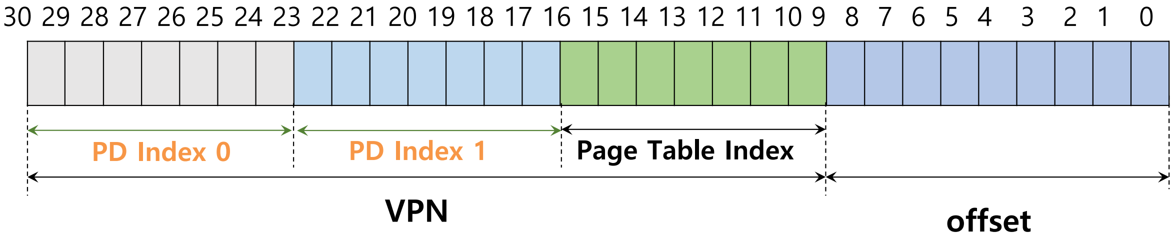


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

$$\log_2 128 = 7$$

More than Two Levels: Page Directory

- If our page directory has 2^{14}entries, it spans not one page but 128.
- To remedy this problem, we build a further level of the tree, by splitting the page directory itself into multiple pages of the page directory.



Multi-level Page Table Control Flow

```
01: VPN = (VirtualAddress & VPN_MASK) >> SHIFT
02: (Success,TlbEntry) = TLB_Lookup(VPN)
03: if(Success == True)   //TLB Hit
04:   if(CanAccess(TlbEntry.ProtectBits) == True)
05:     Offset = VirtualAddress & OFFSET_MASK
06:     PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
07:     Register = AccessMemory(PhysAddr)
08:   else RaiseException(PROTECTION_FAULT);
09: else // perform the full multi-level lookup
```

- (line 1) extract the virtual page number (VPN)
- (line 2) check if the TLB holds the transalation for this VPN
- (lines 5-8) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory

Multi-level Page Table Control Flow

```
11:  else
12:       PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13:       PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14:       PDE = AccessMemory(PDEAddr)
15:       if(PDE.Valid == False)
16:            RaiseException(SEGMENTATION_FAULT)
17:       else // PDE is Valid: now fetch PTE from PT
```

- (line 11) extract the Page Directory Index (PDIndex)
- (line 13) get Page Directory Entry (PDE)
- (lines 15-17) Check PDE valid flag. If valid flag is true, fetch Page Table entry from Page Table

The Translation Process: Remember the TLB

```
PTIndex = (VPN & PT_MASK) >> PT_SHIFT
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if(PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if(CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT);
else
    TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
    RetryInstruction()
```

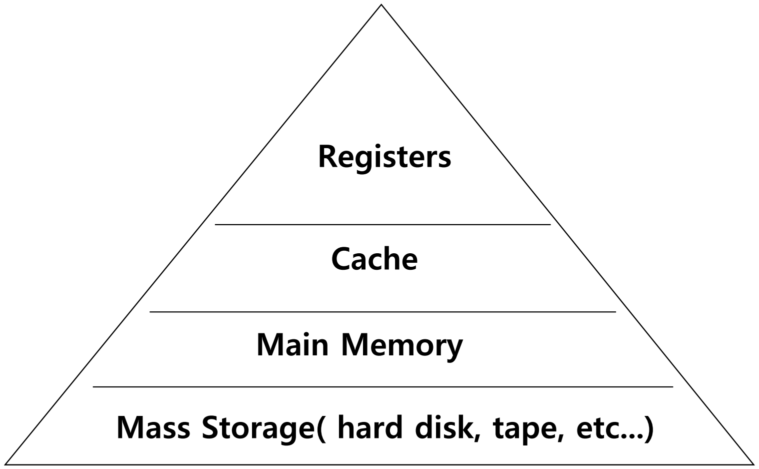
Inverted Page Tables

- Keeping a single page table that has an entry for each physical page of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

Youjip Won

Beyond Physical Memory: Mechanisms

- Require an additional level in the memory hierarchy.
  - OS need a place to stash away portions of address space that currently aren't in great demand.
  - In modern systems, this role is usually served by a hard disk drive



Memory Hierarchy in modern system

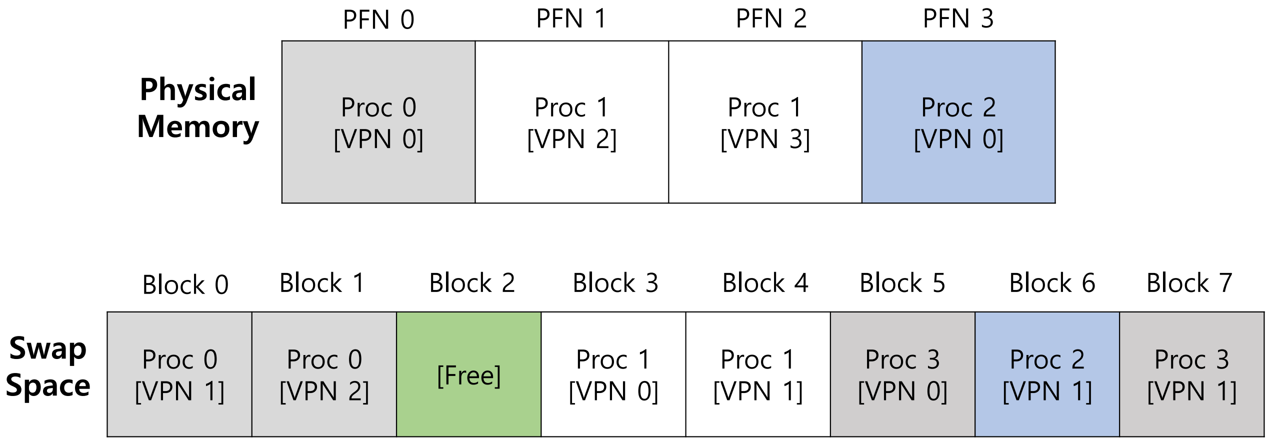
Single large address for a process

- Always need to first arrange for the code or data to be in memory when before calling a function or accessing data.

- To Beyond just a single process.
  - The addition of swap space allows the OS to support the illusion of a large virtual memory for multiple concurrently-running process

Swap Space

- Reserve some space on the disk for moving pages back and forth.
- OS need to remember to the swap space, in page-sized unit



	PFN 0	PFN 1	PFN 2	PFN 3
Physical Memory	Proc 0 [VPN 0]	Proc 1 [VPN 2]	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]

Physical Memory and Swap Space

Present Bit

- Add some machinery higher up in the system in order to support swapping pages to and from the disk.
  - When the hardware looks in the PTE, it may find that the page is not present in physical memory.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

What If Memory Is Full?

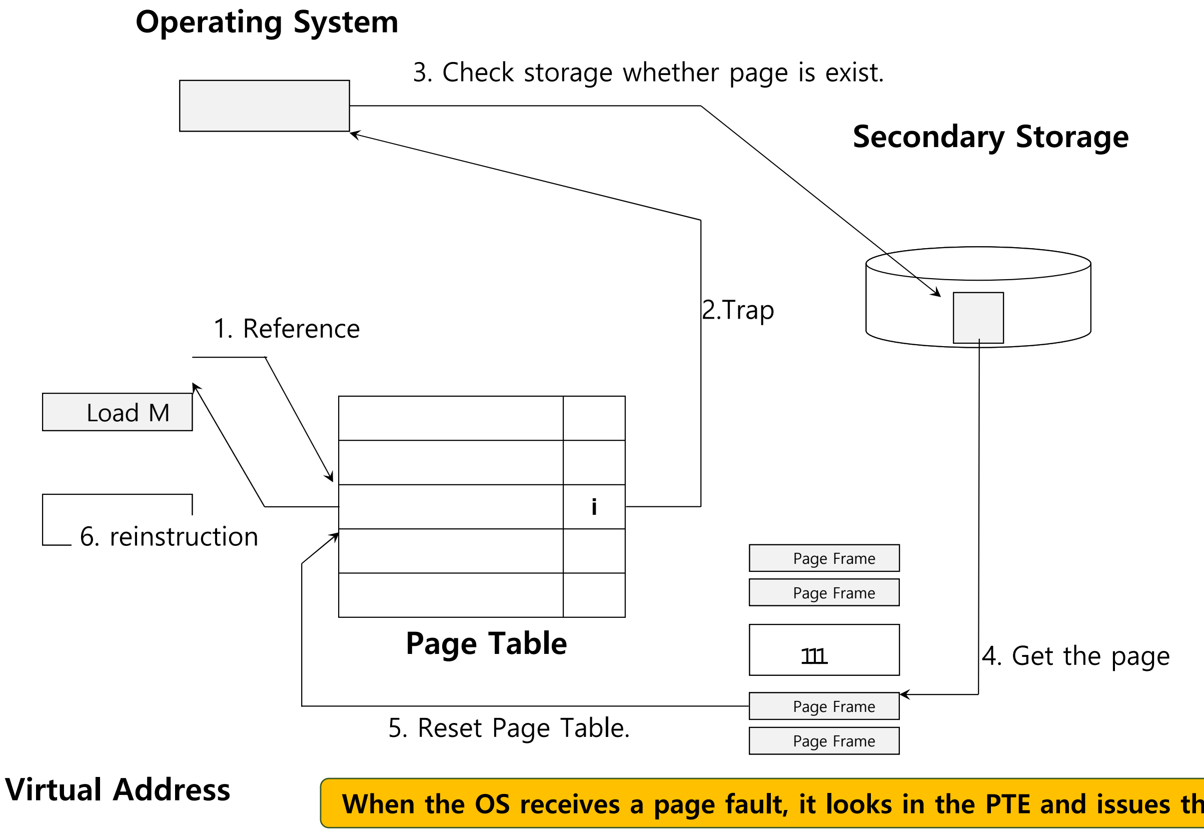
- The OS like to page out pages to make room for the new pages the OS is about to bring in.
  - The process of picking a page to kick out, or replace is known as page-replacement policy

The Page Fault

- Accessing a page that is not in physical memory.
  - If a page is not present and has been swapped disk, the OS need to swap the page into memory in order to service the page fault.

Page Fault Control Flow

- PTE used for data such as the PFN of the page for a disk address.



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

Page Fault Control Flow – Hardware

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
else RaiseException(PROTECTION_FAULT)
```

Page Fault Control Flow – Hardware

```
else // TLB Miss
PTEAddr = PTBR + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else
if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else if (PTE.Present == True)
// assuming hardware-managed TLB
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
else if (PTE.Present == False)
    RaiseException(PAGE_FAULT)
```

Page Fault Control Flow – Software

```
PFN = FindFreePhysicalPage()
if (PFN == -1) // no free page found
    PFN = EvictPage() // run replacement algorithm
    DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
    PTE.present = True // update page table with present
    PTE.PFN = PFN // bit and translation (PFN)
    RetryInstruction() // retry instruction
```

- The OS must find a physical frame for the soon-be-faulted-in page to reside within.
- If there is no such page, waiting for the replacement algorithm to run and kick some pages out of memory.

When Replacements Really Occur

- OS waits until memory is entirely full, and only then replaces a page to make room for some other page
  - This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.

- Swap Daemon, Page Daemon
  - There are fewer than LW pages available, a background thread that is responsible for freeing memory runs.
  - The thread evicts pages until there are HW pages available.

Beyond Physical Memory: Policies

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the *average memory access time(AMAT)*.

$$\boxed{AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)}$$

Arguement	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

The Optimal Replacement Policy

- Leads to the fewest number of misses overall
  - Replaces the page that will be accessed furthest in the future
  - Resulting in the fewest-possible cache misses
- Serve only as a comparison point, to know how close we are to perfect

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

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

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

A Simple Policy: FIFO

- Pages were placed in a queue when they enter the system.
- When a replacement occurs, the page on the tail of the queue(the "**First-in**" pages) is evicted.
  - It is simple to implement but can't determine the importance of blocks.

Tracing the FIFO Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

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		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

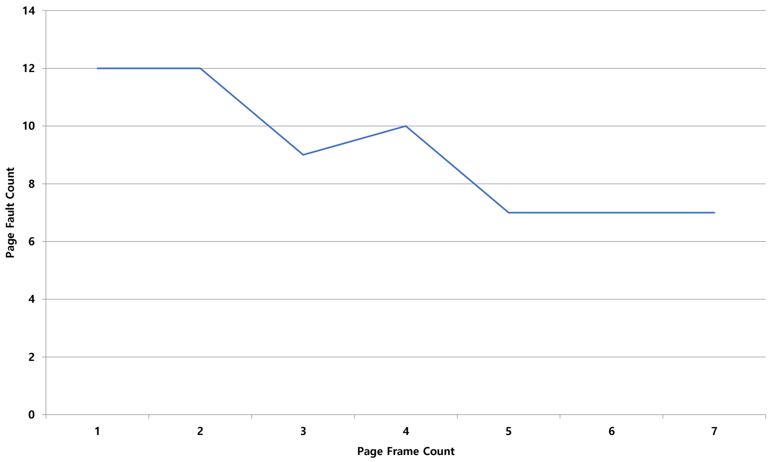
Even though page 0 had been accessed a number of times, FIFO still kicks it out.

BELADY’S ANOMALY

- We would expect the cache hit rate to increase when the cache gets larger. But in this case, with FIFO, it gets worse.

Reference Row

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



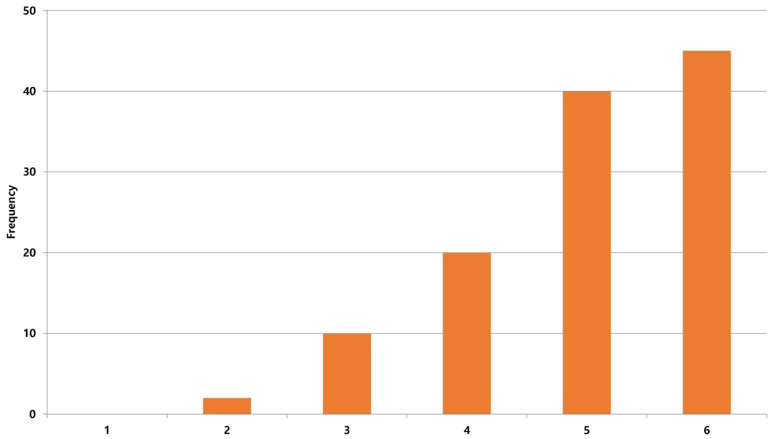
Another Simple Policy: Random

- Picks a random page to replace under memory pressure.
  - It doesn’t really try to be too intelligent in picking which blocks to evict.
  - Random does depend entirely upon how lucky Random gets in its choice.

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

Random Performance

- Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Random Performance over 10,000 Trials

Using History

- Lean on the past and use **history**.
  - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	The more recently a page has been accessed, the more likely it will be accessed again	LRU
frequency	If a page has been accessed many times, It should not be replcaed as it clearly has some value	LFU

Using History : LRU

- Replaces the least-recently-used page.

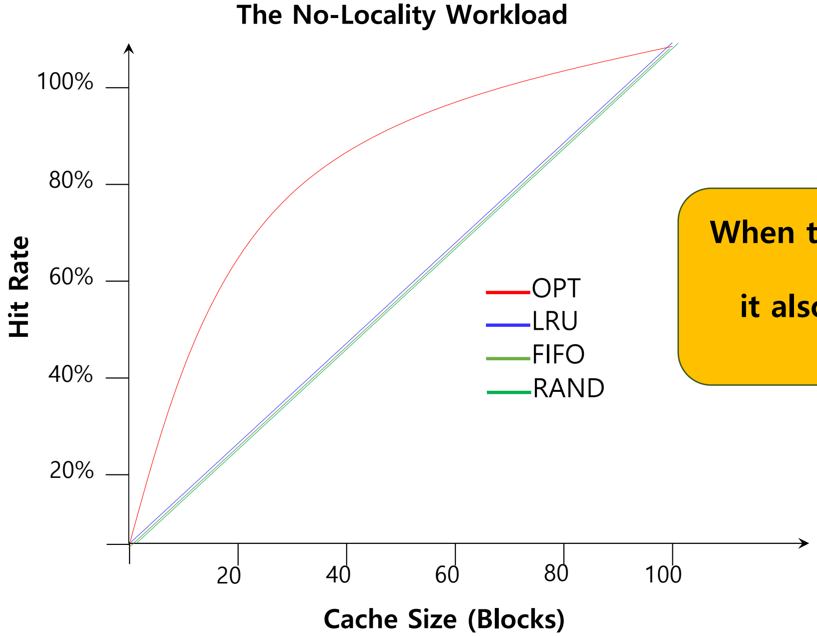
Reference Row

0 1 2 0 1 3 0 3 1 2 1

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

Workload Example : The No-Locality Workload

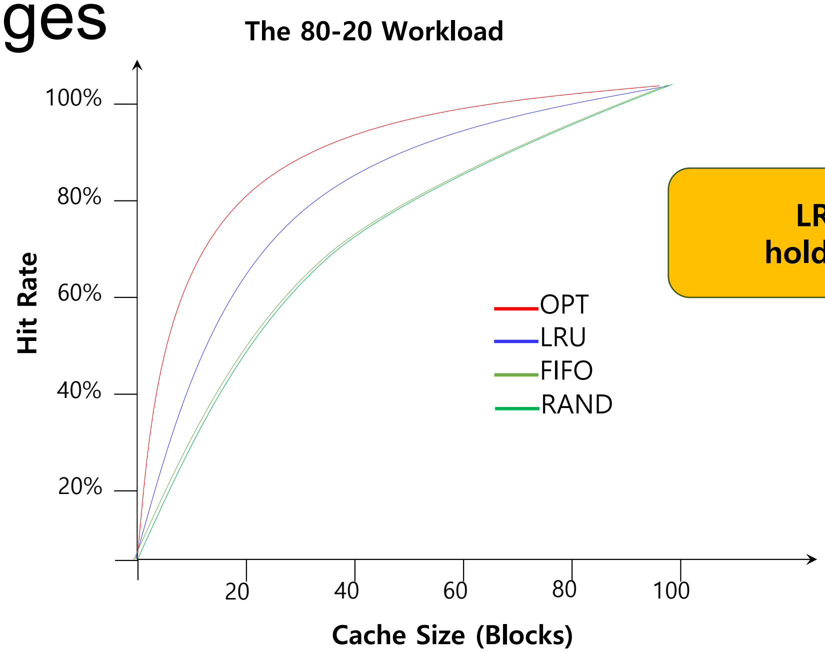
- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



When the cache is large enough to fit the entire workload, it also doesn't matter which policy you use.

Workload Example : The 80-20 Workload

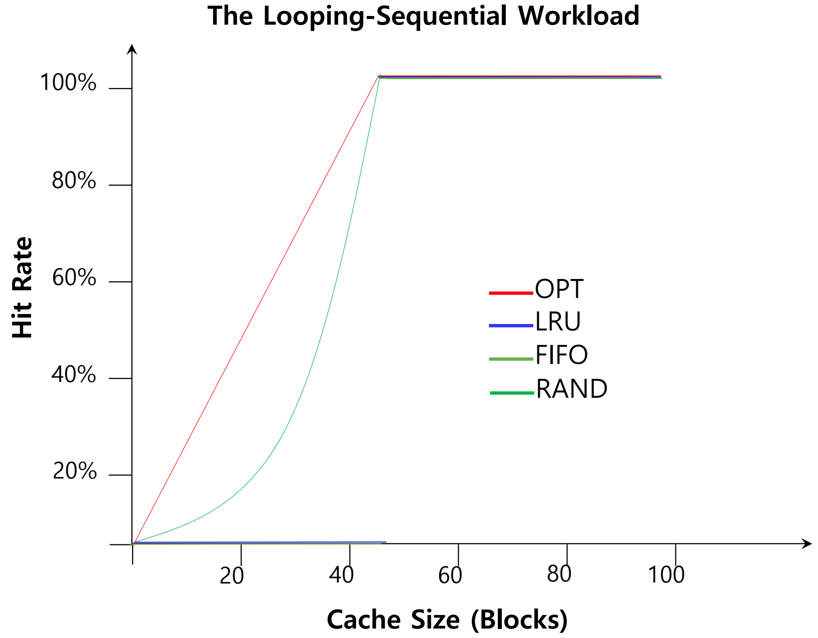
- Exhibits locality: 80% of the reference are made to 20% of the pages
- Remaining 20% of the reference are made to the remaining 80% of the pages



LRU is more likely to hold onto the hot pages.

Workload Example: The Looping Sequential

- Refer to 50 pages in sequence.
  - Starting at 0, then 1, … up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



Implementing Historical Algorithms

- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on **every memory reference.**
  - Add a little bit of hardware support.

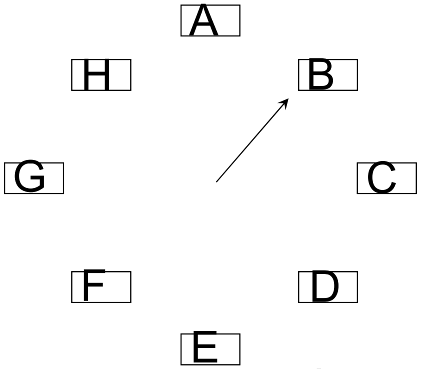
Approximating LRU

- Require some hardware support, in the form of a **use bit**
  - Whenever a page is referenced, the use bit is set by hardware to 1.
  - Hardware never clears the bit, though; that is the responsibility of the OS

- Clock Algorithm
  - All pages of the system arranges in a circular list.
  - A clock hand points to some particular page to begin with.

Clock Algorithm

- The algorithm continues until it finds a use bit that is set to 0.



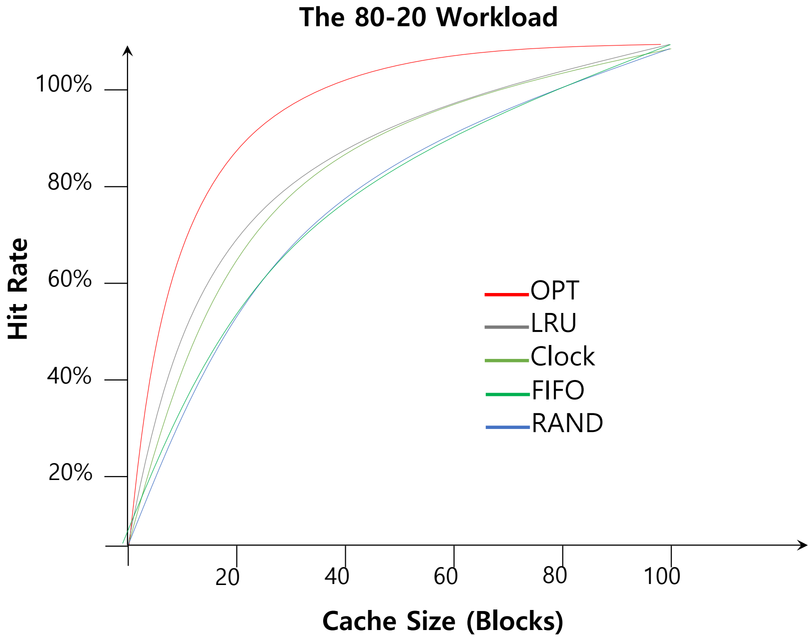
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

The Clock page replacement algorithm

When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the Use bit

Workload with Clock Algorithm

- Clock algorithm doesn’t do as well as perfect LRU, it does better then approach that don’t consider history at all.



Considering Dirty Pages

- The hardware include a **modified bit** (a.k.a **dirty bit**)
  - Page has been **modified** and is thus **dirty**, it must be written back to disk to evict it.
  - Page has not been modified, the eviction is free.

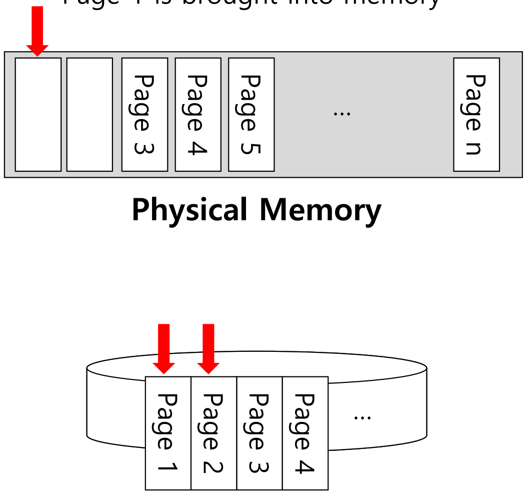
Page Selection Policy

- The OS has to decide when to bring a page into memory.
- Presents the OS with some different options.

Prefetching

- The OS guess that a page is about to be used, and thus bring it in ahead of time.

Page 1 is brought into memory



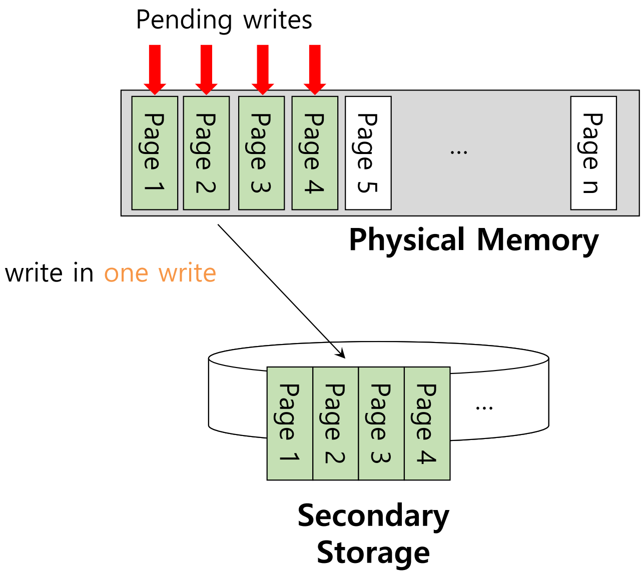
Physical Memory

Secondary Storage

Page 2 likely soon be accessed and
thus should be brought into memory too

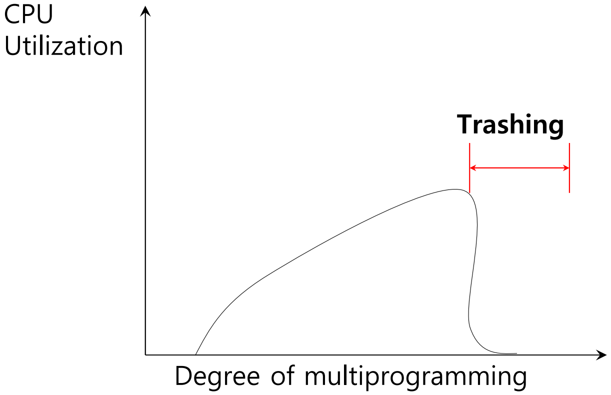
Clustering, Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
  - Perform a **single large write** more efficiently than **many small ones**.



Thrashing

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.
  - Decide not to run a subset of processes.
  - Reduced set of processes working sets fit in memory.



Project Launch

What's Next

- Homework Assignment 3
  - Due by Sunday, February 15 by 11:59pm ET

- Quiz 3
  - Due by Sunday, February 15 by 11:59pm ET

- Programming Assignment 2 (PA2)
  - Due by Friday, February 27, 2026 by 11:59pm ET

- Course Project – Task 0 and Task 1
  - Due by Sunday, March 8, 2026 by 11:59pm ET