# TCSS 422: OPERATING SYSTEMS

***Memory Virtualization II:***
Memory Segments,
Free Space Management,
Introduction to Paging,
Translation Lookaside Buffer

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

1

## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

2

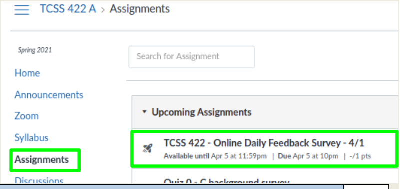
## QUIZ 2

- **Thursday March 5, 4:40 to 5:40pm, JOY 215**
- Last year average was just ~51 out of 70 (72.9%) !
- Coverage through today's lecture
- Primary chapters after midterm:
- Core concepts:
  - Multi-threading and thread-safe data structures, Bounded Buffer, Deadlock (Chapters 29, 30, 32)
  - Address Spaces, Memory API, Address Translation w/ Base & Bounds, Segments, Free Space Mgmt, Paging (Chapters 13, 14, 15, 16, 17, 18)
  - Translation Lookaside Buffer (Chapter 19)
- **Optional Quiz 2 Review Session (live stream & recorded)**
  - Wednesday March 4th 6pm - Zoom Only – Link to be shared

3

## ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p



TCSS 422 A › Assignments

Spring 2021

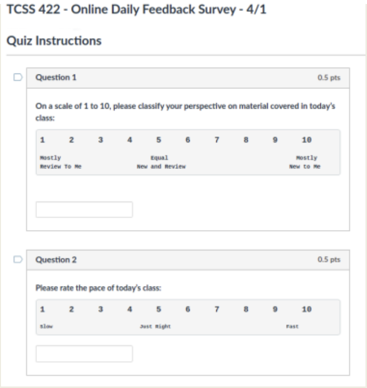
Search for Assignment

▾ Upcoming Assignments

TCSS 422 - Online Daily Feedback Survey - 4/1
Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts

Quiz 0 - C background survey

4



TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 0.5 pts

On a scale of 1 to 10, please classify your perspective on material covered in today's class:

```
1      2      3      4      5      6      7      8      9      10
Mostly                      Equal                           Mostly
Review To Me             New and Review                    New to Me
```

Question 2 0.5 pts

Please rate the pace of today's class:

```
1      2      3      4      5      6      7      8      9      10
Slow                     Just Right                        Fast
```

5

## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (31 of 46 respondents (7 online) – 67.39%):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.52 (↓ - previous 6.79)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.48 (↑ - previous 4.88)**

6

## FEEDBACK FROM 2/26

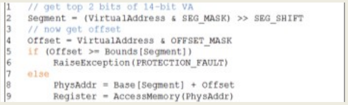
- ***What is the meaning of the checkmarks (√) on slide 14.7***
  - Each of the conditions must be present for deadlock to occur: mutual exclusion, hold and wait, no preemption, circular wait
- ***What happens if the physical address is out of bounds? Is something changed to fix that? Or does it just fail?***
  - If a virtual address translation maps to a physical address beyond the bounds register, then an error is generated (segmentation fault)
  - The user is presented from accessing beyond the bounds of the physical memory range reserved from their program
  - This could be memory for another program, and should never be readable !!

## FEEDBACK - 2

- ***How does the bitwise operations work for the address translation algorithm on slide L14.43 ?***
- We logically 'AND' addresses with bit MASKs
- The bit MASKs allow the bits we are interested in to pass-thru while zero-ing out bits we are not interested in
- SEG_MASK (segment mask) – allows segment ID to pass thru
- OFFSET_MASK (offset mask) – allows offset bits, the bits that index a memory address within the segment to pass thru

## FEEDBACK - 3

- ***How does virtual to physical address translation work with segments?***



```
1   // get top 2 bits of 14-bit VA
2   Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3   // now get offset
4   Offset = VirtualAddress & OFFSET_MASK
5   if (Offset >= Bounds[Segment])
6       RaiseException(PROTECTION_FAULT)
7   else
8       PhysAddr = Base[Segment] + Offset
9       Register = AccessMemory(PhysAddr)
```

- Consider address translation algorithm from slide 14.43
- Consider that we want to read the code (instruction) at address 100
- 64k computer – use 16 bits for physical addressing
- 16k address space – uses 12 bits to index every address in the range
  - Isolate 'offset' with offset mask, Isolate "segment' ID with the segment mask
  - Offset is "0000 0110 1000", Segment is the code segment (00)
  - The bounds register is 32k → 32,768 -> 1000 0000 0000 0000
  - We logically AND the base register with the offset: (line #8 above)
    1000 0000 0000 0000 (base reg)
    0000 0000 0110 1000 (offset)
    1000 0000 0110 1000 → 32,872 (physical address of code)

## BASE & BOUNDS EXAMPLE - 1

- Consider address translation
  - Base= 16,384 (16 KB), program loaded at 16 KB physical address
  - Bounds= 4,096 (4 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 100 | ? |
| 2000 | ? |
| ? | 20,384 |
| ? | out of bounds |

## BASE & BOUNDS EXAMPLE - 1

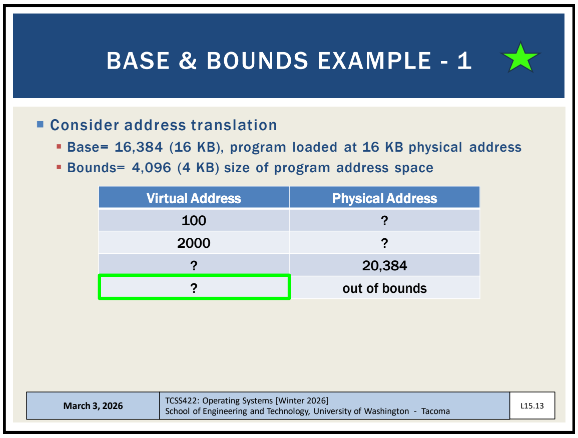
- Consider address translation
  - Base= 16,384 (16 KB), program loaded at 16 KB physical address
  - Bounds= 4,096 (4 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 100 | ? |
| 2000 | ? |
| ? | 20,384 |
| ? | out of bounds |

## BASE & BOUNDS EXAMPLE - 1

- Consider address translation
  - Base= 16,384 (16 KB), program loaded at 16 KB physical address
  - Bounds= 4,096 (4 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 100 | ? |
| 2000 | ? |
| ? | 20,384 |
| ? | out of bounds |

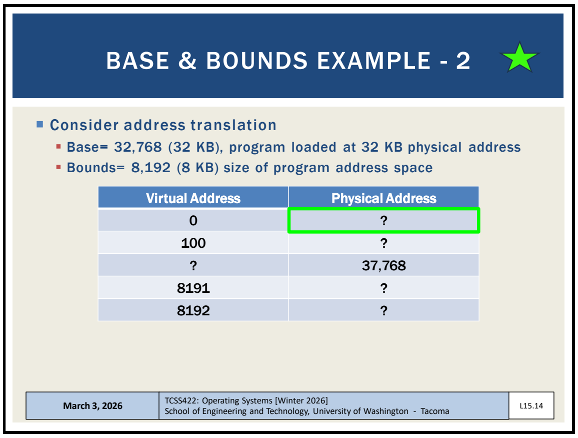


## BASE & BOUNDS EXAMPLE - 1

- Consider address translation
  - Base= 16,384 (16 KB), program loaded at 16 KB physical address
  - Bounds= 4,096 (4 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 100 | ? |
| 2000 | ? |
| ? | 20,384 |
| ? | out of bounds |

13

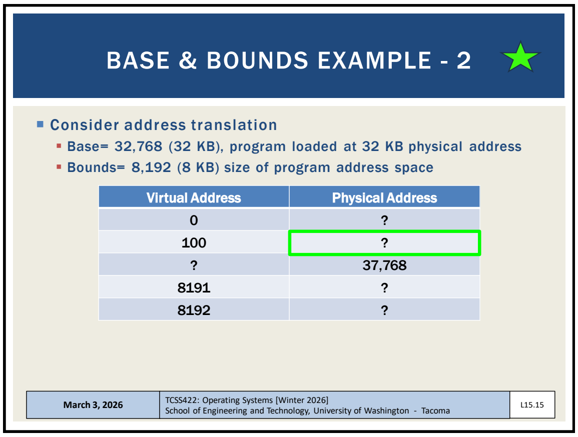


## BASE & BOUNDS EXAMPLE - 2

- Consider address translation
  - Base= 32,768 (32 KB), program loaded at 32 KB physical address
  - Bounds= 8,192 (8 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 0 | ? |
| 100 | ? |
| ? | 37,768 |
| 8191 | ? |
| 8192 | ? |

14

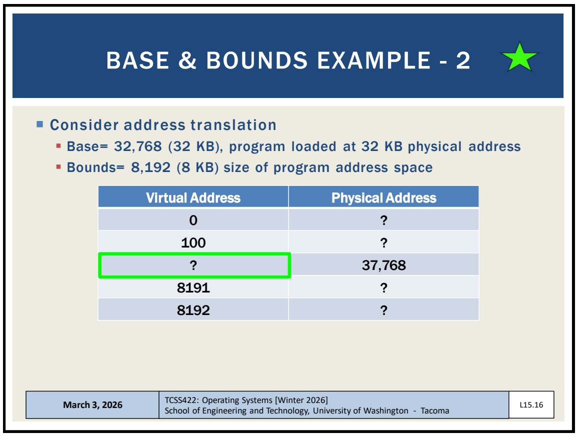


## BASE & BOUNDS EXAMPLE - 2

- Consider address translation
  - Base= 32,768 (32 KB), program loaded at 32 KB physical address
  - Bounds= 8,192 (8 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 0 | ? |
| 100 | ? |
| ? | 37,768 |
| 8191 | ? |
| 8192 | ? |

15

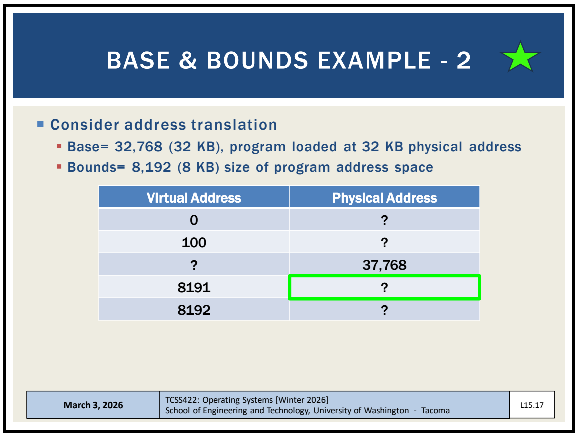


## BASE & BOUNDS EXAMPLE - 2

- Consider address translation
  - Base= 32,768 (32 KB), program loaded at 32 KB physical address
  - Bounds= 8,192 (8 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 0 | ? |
| 100 | ? |
| ? | 37,768 |
| 8191 | ? |
| 8192 | ? |

16

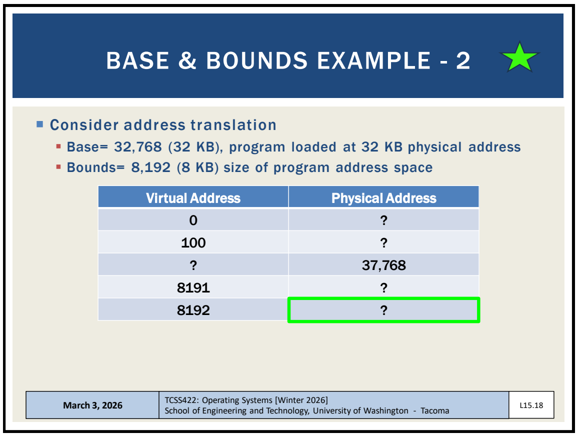


## BASE & BOUNDS EXAMPLE - 2

- Consider address translation
  - Base= 32,768 (32 KB), program loaded at 32 KB physical address
  - Bounds= 8,192 (8 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 0 | ? |
| 100 | ? |
| ? | 37,768 |
| 8191 | ? |
| 8192 | ? |

17

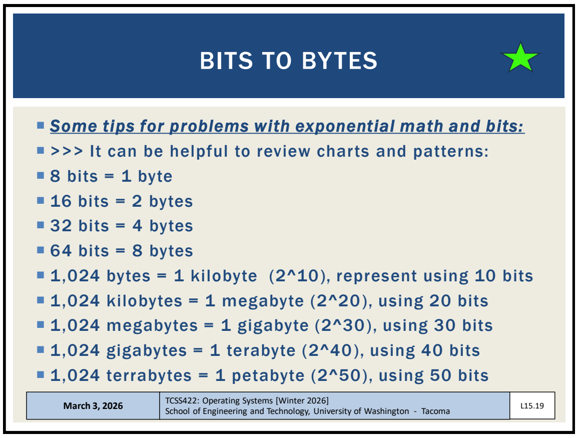


## BASE & BOUNDS EXAMPLE - 2

- Consider address translation
  - Base= 32,768 (32 KB), program loaded at 32 KB physical address
  - Bounds= 8,192 (8 KB) size of program address space

| Virtual Address | Physical Address |
|---|---|
| 0 | ? |
| 100 | ? |
| ? | 37,768 |
| 8191 | ? |
| 8192 | ? |

18

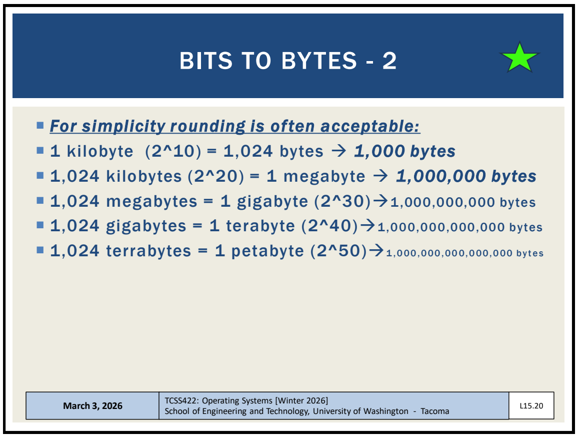


## BITS TO BYTES ★

- ***Some tips for problems with exponential math and bits:***
- >>> It can be helpful to review charts and patterns:
- 8 bits = 1 byte
- 16 bits = 2 bytes
- 32 bits = 4 bytes
- 64 bits = 8 bytes
- 1,024 bytes = 1 kilobyte (2^10), represent using 10 bits
- 1,024 kilobytes = 1 megabyte (2^20), using 20 bits
- 1,024 megabytes = 1 gigabyte (2^30), using 30 bits
- 1,024 gigabytes = 1 terabyte (2^40), using 40 bits
- 1,024 terrabytes = 1 petabyte (2^50), using 50 bits

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.19

19

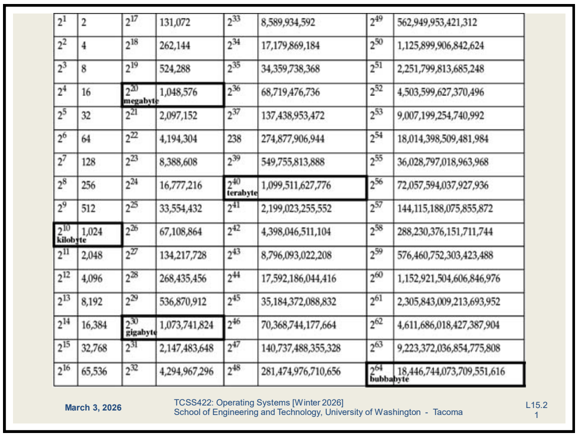


## BITS TO BYTES - 2 ★

- ***For simplicity rounding is often acceptable:***
- 1 kilobyte (2^10) = 1,024 bytes → ***1,000 bytes***
- 1,024 kilobytes (2^20) = 1 megabyte → ***1,000,000 bytes***
- 1,024 megabytes = 1 gigabyte (2^30)→1,000,000,000 bytes
- 1,024 gigabytes = 1 terabyte (2^40)→1,000,000,000,000 bytes
- 1,024 terrabytes = 1 petabyte (2^50)→1,000,000,000,000,000 bytes

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.20

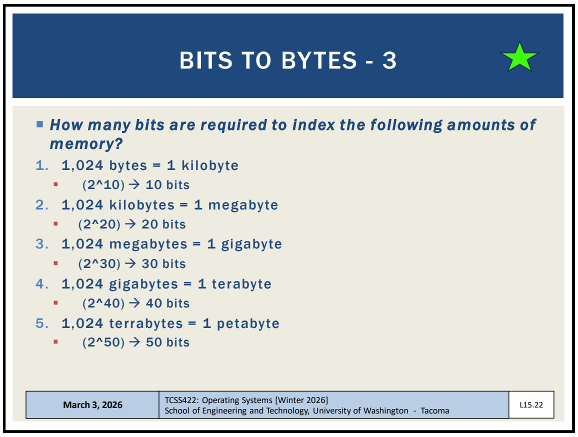
20



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $2^1$ | 2 | $2^{17}$ | 131,072 | $2^{33}$ | 8,589,934,592 | $2^{49}$ | 562,949,953,421,312 |
| $2^2$ | 4 | $2^{18}$ | 262,144 | $2^{34}$ | 17,179,869,184 | $2^{50}$ | 1,125,899,906,842,624 |
| $2^3$ | 8 | $2^{19}$ | 524,288 | $2^{35}$ | 34,359,738,368 | $2^{51}$ | 2,251,799,813,685,248 |
| $2^4$ | 16 | $2^{20}$ megabyte | 1,048,576 | $2^{36}$ | 68,719,476,736 | $2^{52}$ | 4,503,599,627,370,496 |
| $2^5$ | 32 | $2^{21}$ | 2,097,152 | $2^{37}$ | 137,438,953,472 | $2^{53}$ | 9,007,199,254,740,992 |
| $2^6$ | 64 | $2^{22}$ | 4,194,304 | 238 | 274,877,906,944 | $2^{54}$ | 18,014,398,509,481,984 |
| $2^7$ | 128 | $2^{23}$ | 8,388,608 | $2^{39}$ | 549,755,813,888 | $2^{55}$ | 36,028,797,018,963,968 |
| $2^8$ | 256 | $2^{24}$ | 16,777,216 | $2^{40}$ terabyte | 1,099,511,627,776 | $2^{56}$ | 72,057,594,037,927,936 |
| $2^9$ | 512 | $2^{25}$ | 33,554,432 | $2^{41}$ | 2,199,023,255,552 | $2^{57}$ | 144,115,188,075,855,872 |
| $2^{10}$ kilobyte | 1,024 | $2^{26}$ | 67,108,864 | $2^{42}$ | 4,398,046,511,104 | $2^{58}$ | 288,230,376,151,711,744 |
| $2^{11}$ | 2,048 | $2^{27}$ | 134,217,728 | $2^{43}$ | 8,796,093,022,208 | $2^{59}$ | 576,460,752,303,423,488 |
| $2^{12}$ | 4,096 | $2^{28}$ | 268,435,456 | $2^{44}$ | 17,592,186,044,416 | $2^{60}$ | 1,152,921,504,606,846,976 |
| $2^{13}$ | 8,192 | $2^{29}$ | 536,870,912 | $2^{45}$ | 35,184,372,088,832 | $2^{61}$ | 2,305,843,009,213,693,952 |
| $2^{14}$ | 16,384 | $2^{30}$ gigabyte | 1,073,741,824 | $2^{46}$ | 70,368,744,177,664 | $2^{62}$ | 4,611,686,018,427,387,904 |
| $2^{15}$ | 32,768 | $2^{31}$ | 2,147,483,648 | $2^{47}$ | 140,737,488,355,328 | $2^{63}$ | 9,223,372,036,854,775,808 |
| $2^{16}$ | 65,536 | $2^{32}$ | 4,294,967,296 | $2^{48}$ | 281,474,976,710,656 | $2^{64}$ bubbabyte | 18,446,744,073,709,551,616 |

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.2 1

21



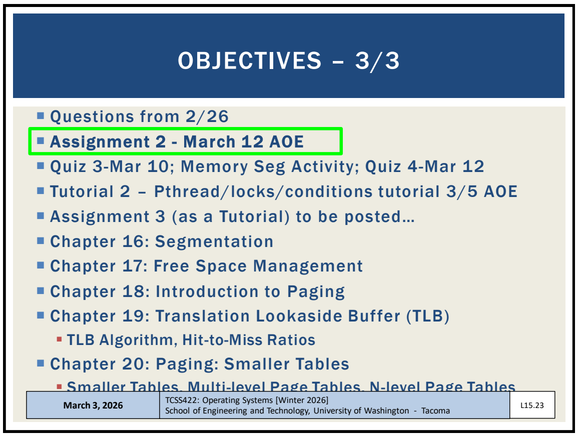
## BITS TO BYTES - 3 ★

- ***How many bits are required to index the following amounts of memory?***

1. 1,024 bytes = 1 kilobyte
   - (2^10) → 10 bits
2. 1,024 kilobytes = 1 megabyte
   - (2^20) → 20 bits
3. 1,024 megabytes = 1 gigabyte
   - (2^30) → 30 bits
4. 1,024 gigabytes = 1 terabyte
   - (2^40) → 40 bits
5. 1,024 terrabytes = 1 petabyte
   - (2^50) → 50 bits

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.22

22

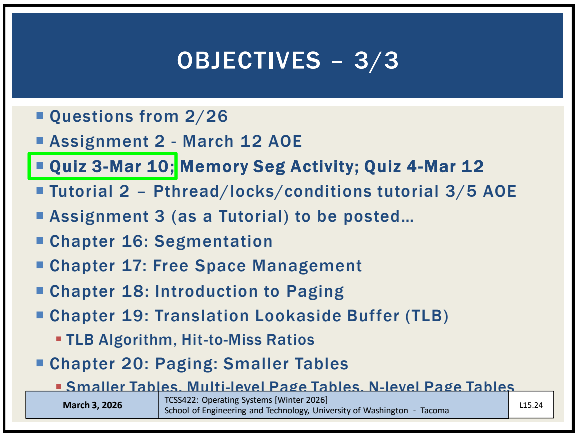


## OBJECTIVES – 3/3

- Questions from 2/26
- **Assignment 2 - March 12 AOE**
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.23

23

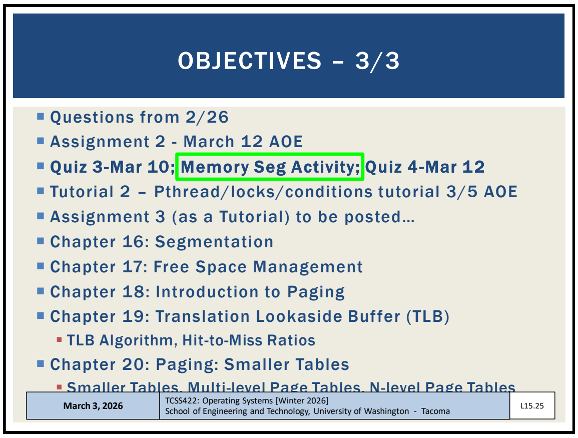


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- **Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12**
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.24

24

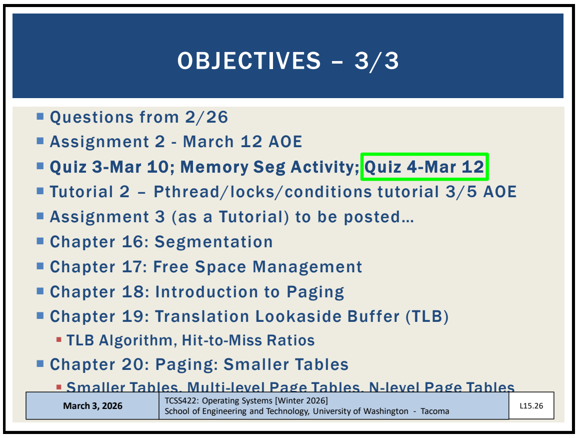


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- **Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12**
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.25

25

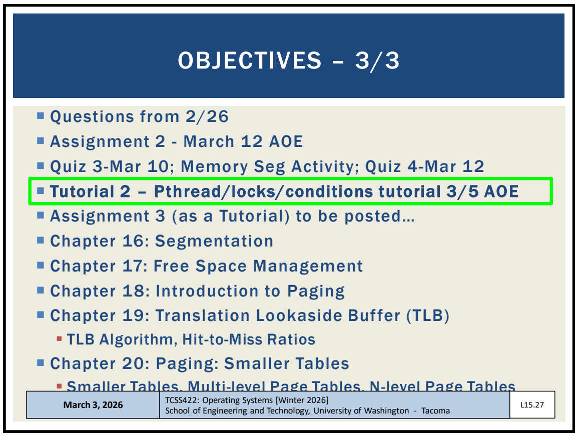


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- **Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12**
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.26

26

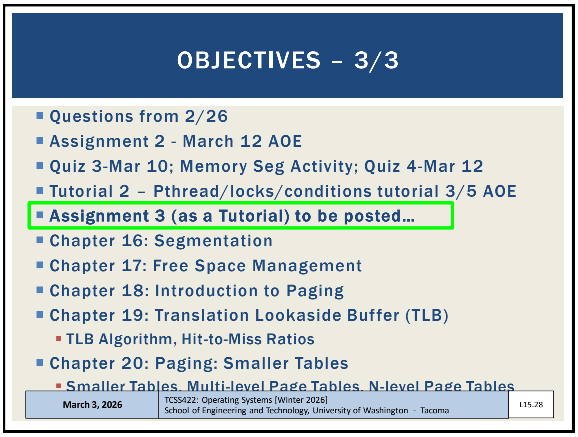


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- **Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE**
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.27

27

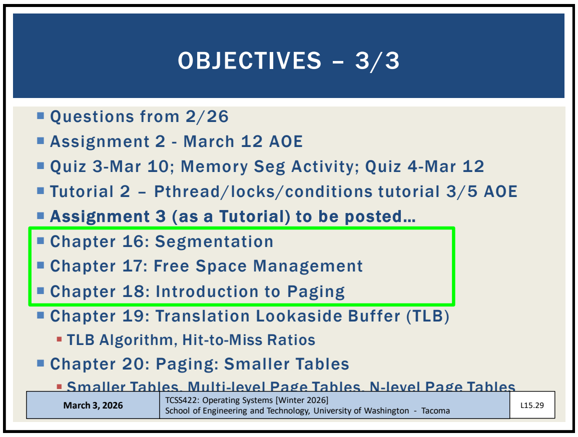


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- **Assignment 3 (as a Tutorial) to be posted…**
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.28

28

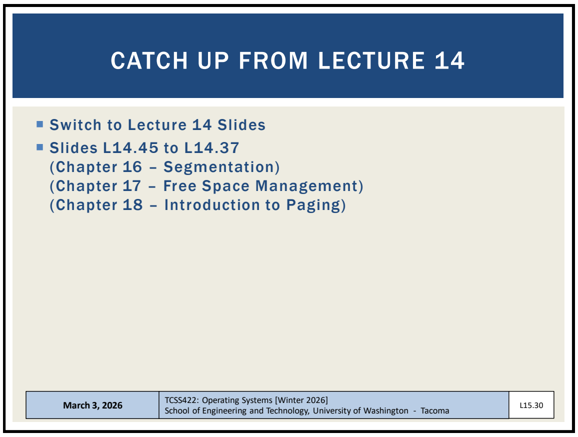


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- **Assignment 3 (as a Tutorial) to be posted…**
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.29

29



## CATCH UP FROM LECTURE 14

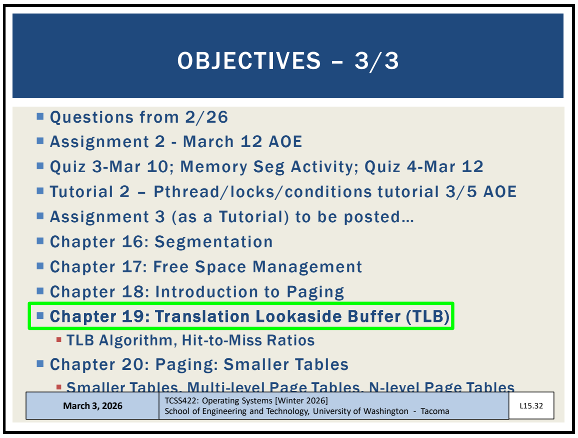
- Switch to Lecture 14 Slides
- Slides L14.45 to L14.37
  (Chapter 16 – Segmentation)
  (Chapter 17 – Free Space Management)
  (Chapter 18 – Introduction to Paging)

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.30

30

## WE WILL RETURN AT 4:55PM

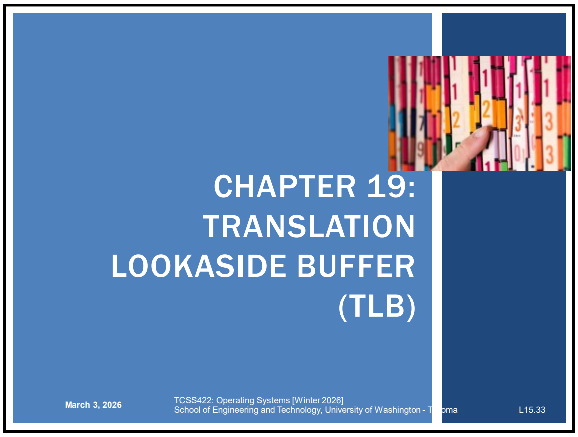
31



## OBJECTIVES – 3/3

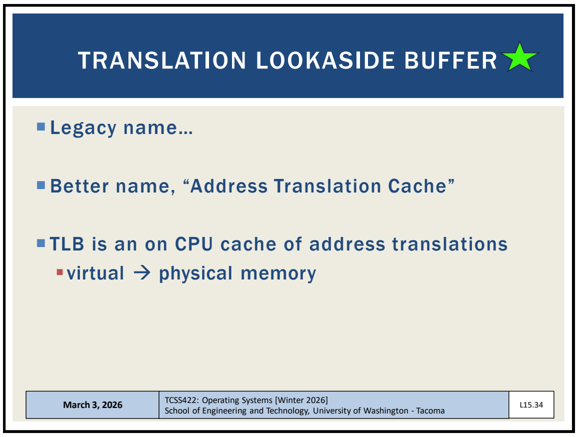
- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- **Chapter 19: Translation Lookaside Buffer (TLB)**
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

32



## CHAPTER 19: TRANSLATION LOOKASIDE BUFFER (TLB)

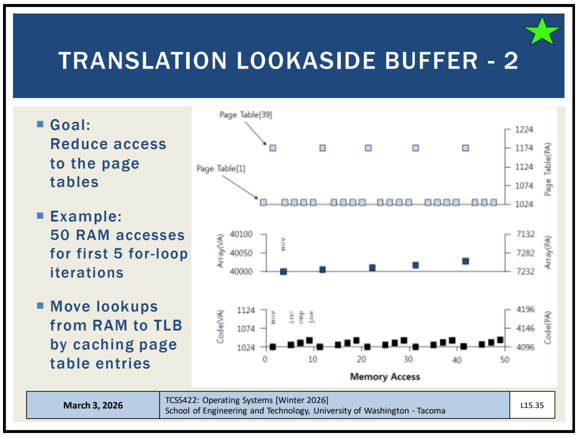
33



## TRANSLATION LOOKASIDE BUFFER ★

- Legacy name…
- Better name, "Address Translation Cache"
- TLB is an on CPU cache of address translations
  - virtual → physical memory

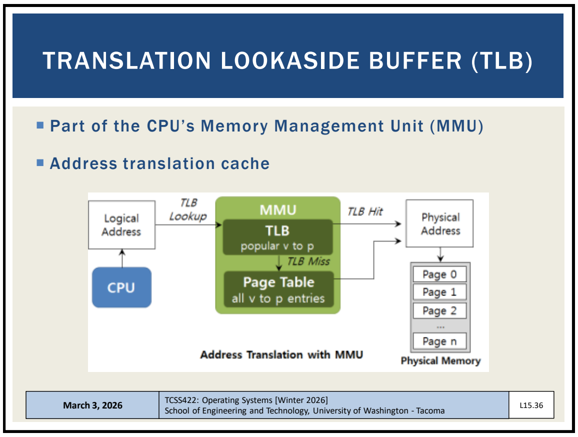
34



## TRANSLATION LOOKASIDE BUFFER - 2 ★

- Goal: Reduce access to the page tables
- Example: 50 RAM accesses for first 5 for-loop iterations
- Move lookups from RAM to TLB by caching page table entries

35

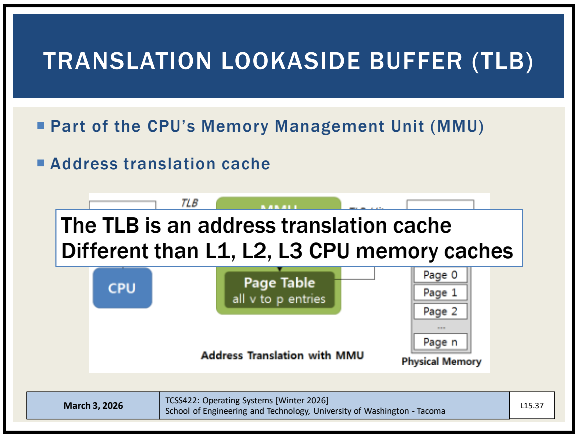


## TRANSLATION LOOKASIDE BUFFER (TLB)

- Part of the CPU's Memory Management Unit (MMU)
- Address translation cache

Address Translation with MMU

36

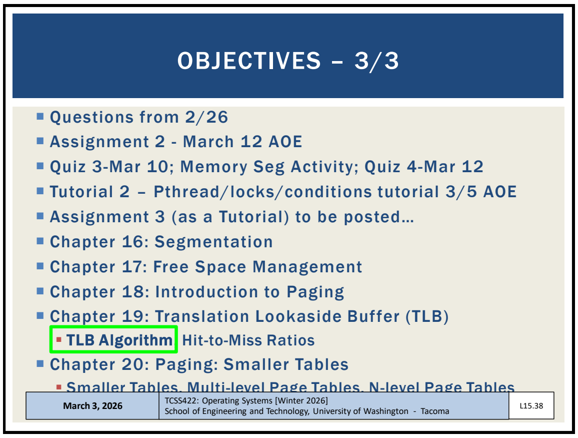


## TRANSLATION LOOKASIDE BUFFER (TLB)

- Part of the CPU's Memory Management Unit (MMU)
- Address translation cache

The TLB is an address translation cache
Different than L1, L2, L3 CPU memory caches

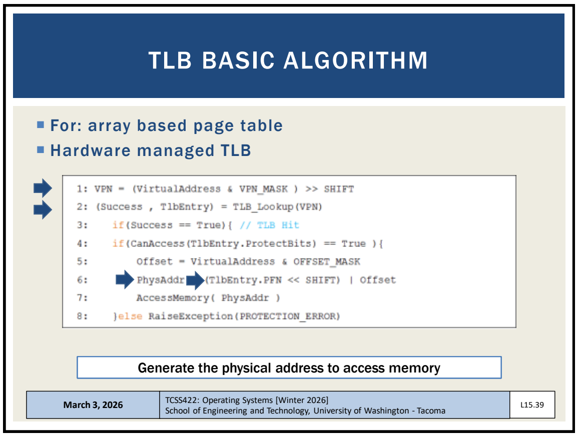
37



## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - **TLB Algorithm**, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

38



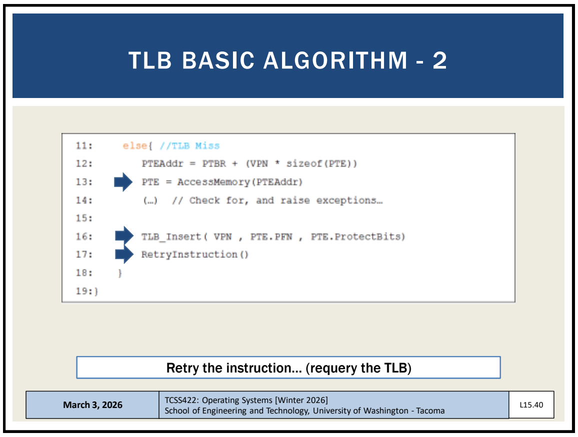
## TLB BASIC ALGORITHM

- For: array based page table
- Hardware managed TLB

```
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
        AccessMemory( PhysAddr )
    }else RaiseException(PROTECTION_ERROR)
```

Generate the physical address to access memory

39



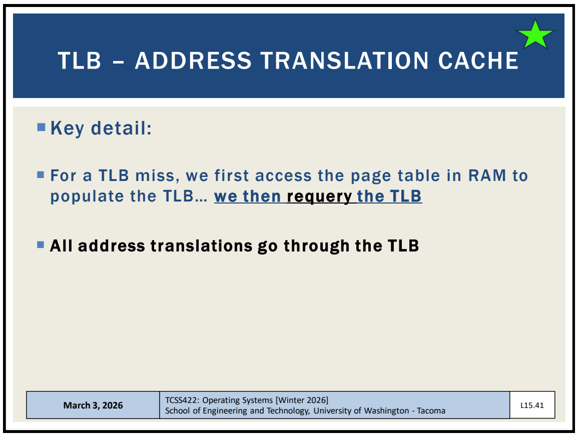
## TLB BASIC ALGORITHM - 2

```
    else{ //TLB Miss
        PTEAddr = PTBR + (VPN * sizeof(PTE))
        PTE = AccessMemory(PTEAddr)
        (…)  // Check for, and raise exceptions…

        TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
        RetryInstruction()
    }
}
```

Retry the instruction… (requery the TLB)

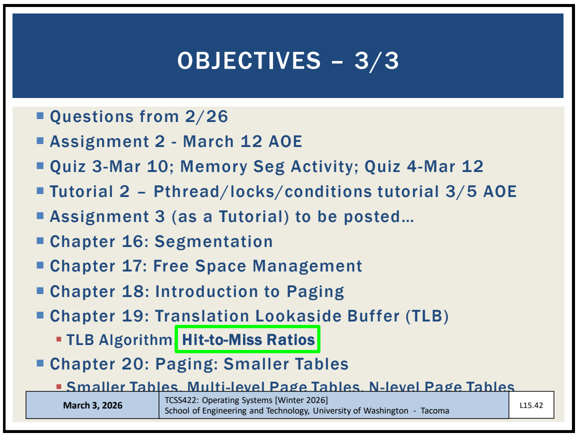
40



## TLB – ADDRESS TRANSLATION CACHE

- Key detail:
- For a TLB miss, we first access the page table in RAM to populate the TLB… **we then requery the TLB**
- **All address translations go through the TLB**

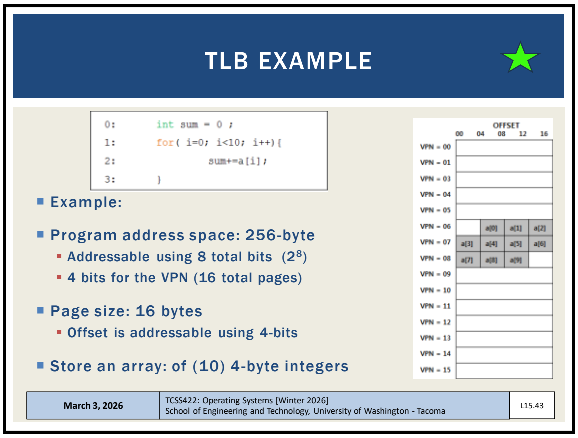
41



## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, **Hit-to-Miss Ratios**
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

42



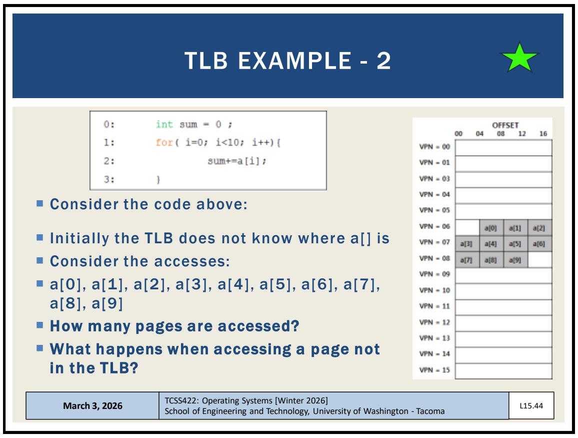
## TLB EXAMPLE

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

- Example:
- Program address space: 256-byte
  - Addressable using 8 total bits ($2^8$)
  - 4 bits for the VPN (16 total pages)
- Page size: 16 bytes
  - Offset is addressable using 4-bits
- Store an array: of (10) 4-byte integers

| | 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 | | | | | |

43



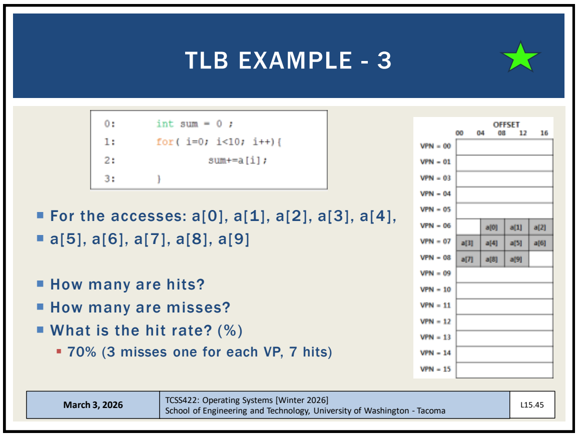
## TLB EXAMPLE - 2

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

- Consider the code above:
- Initially the TLB does not know where a[] is
- Consider the accesses:
- a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]
- **How many pages are accessed?**
- **What happens when accessing a page not in the TLB?**

| | 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 | | | | | |

44



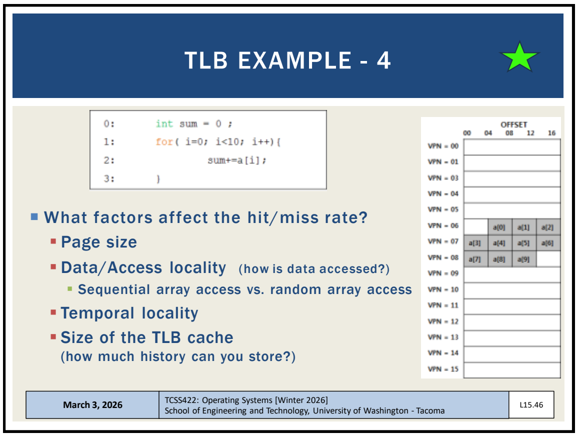
## TLB EXAMPLE - 3

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

- For the accesses: a[0], a[1], a[2], a[3], a[4],
- a[5], a[6], a[7], a[8], a[9]
- How many hits?
- How many misses?
- What is the hit rate? (%)
  - 70% (3 misses one for each VP, 7 hits)

| | 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 | | | | | |

45



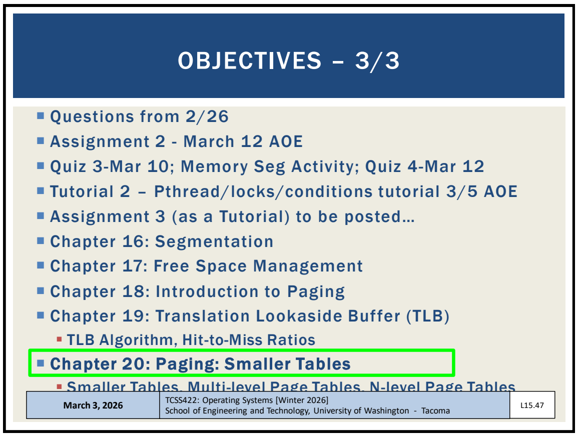
## TLB EXAMPLE - 4

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

- What factors affect the hit/miss rate?
  - Page size
  - Data/Access locality (how is data accessed?)
    - Sequential array access vs. random array access
  - Temporal locality
  - Size of the TLB cache
    (how much history can you store?)

| | 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 | | | | | |

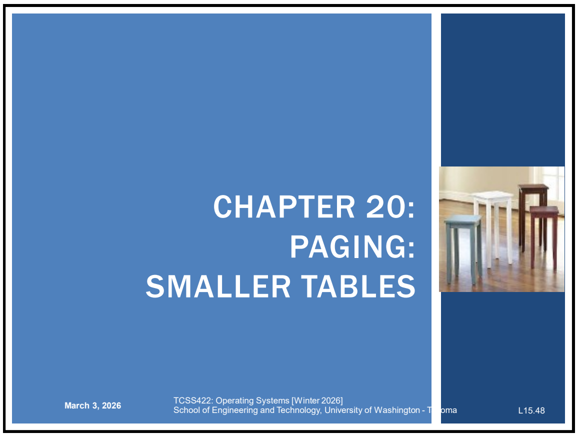
46



## OBJECTIVES – 3/3

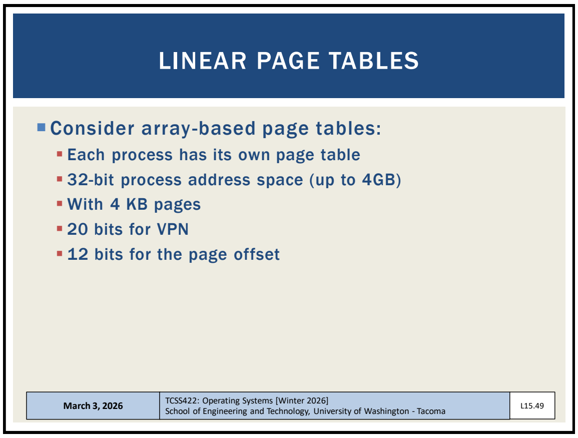
- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

47



## CHAPTER 20: PAGING: SMALLER TABLES

48

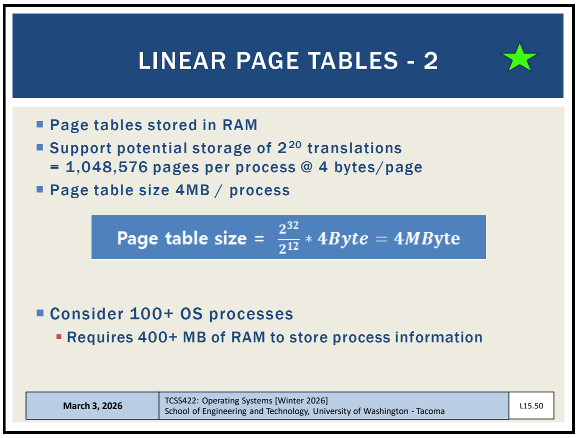


## LINEAR PAGE TABLES

- Consider array-based page tables:
  - Each process has its own page table
  - 32-bit process address space (up to 4GB)
  - With 4 KB pages
  - 20 bits for VPN
  - 12 bits for the page offset

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.49

49



## LINEAR PAGE TABLES - 2 ★

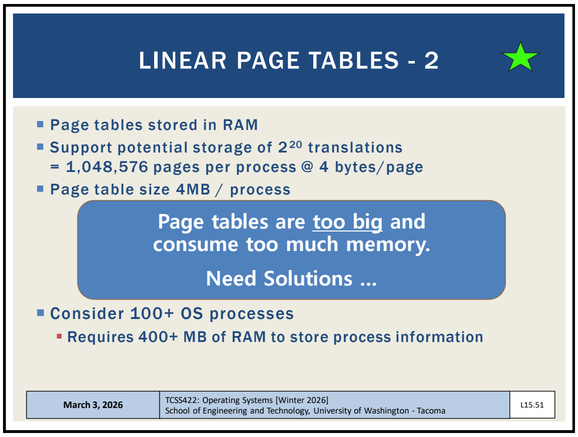
- Page tables stored in RAM
- Support potential storage of $2^{20}$ translations = 1,048,576 pages per process @ 4 bytes/page
- Page table size 4MB / process

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

- Consider 100+ OS processes
  - Requires 400+ MB of RAM to store process information

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.50

50



## LINEAR PAGE TABLES - 2 ★

- Page tables stored in RAM
- Support potential storage of $2^{20}$ translations = 1,048,576 pages per process @ 4 bytes/page
- Page table size 4MB / process

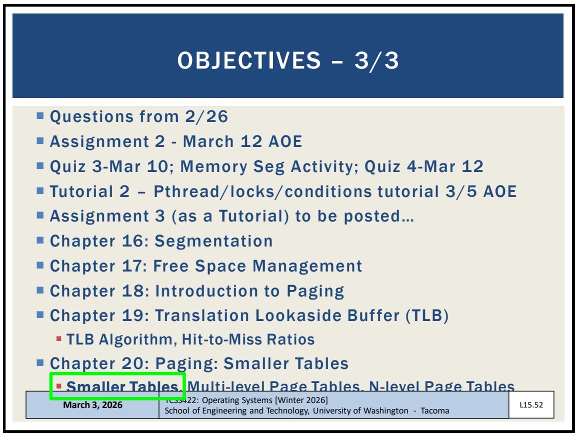
Page tables are too big and consume too much memory.

Need Solutions …

- Consider 100+ OS processes
  - Requires 400+ MB of RAM to store process information

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.51

51

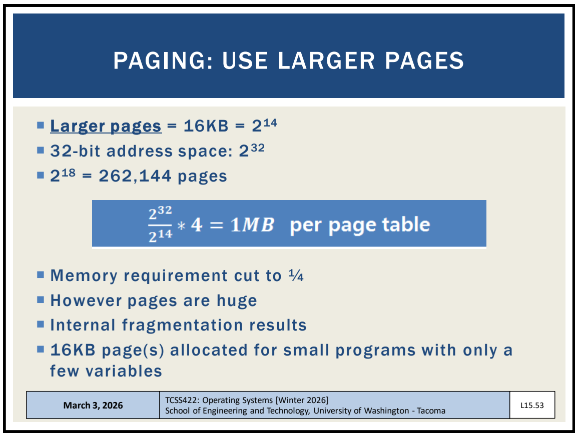


## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.52

52



## PAGING: USE LARGER PAGES

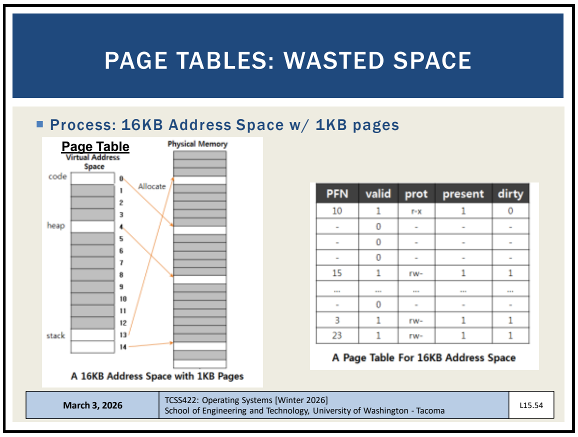
- Larger pages = 16KB = $2^{14}$
- 32-bit address space: $2^{32}$
- $2^{18}$ = 262,144 pages

$\frac{2^{32}}{2^{14}} * 4 = 1MB$ per page table

- Memory requirement cut to ¼
- However pages are huge
- Internal fragmentation results
- 16KB page(s) allocated for small programs with only a few variables

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.53

53



## PAGE TABLES: WASTED SPACE

- Process: 16KB Address Space w/ 1KB pages

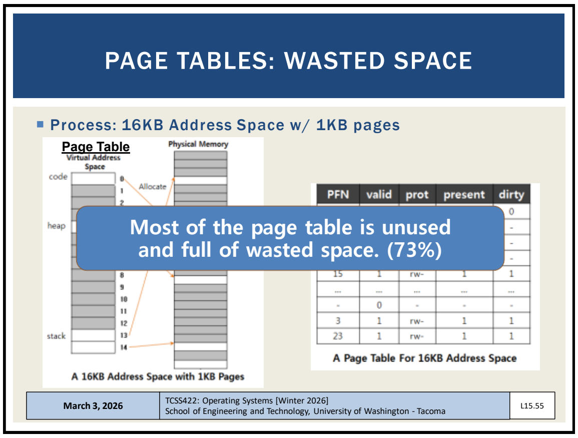
A 16KB Address Space with 1KB Pages

| 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

March 3, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L15.54

54



## PAGE TABLES: WASTED SPACE

- Process: 16KB Address Space w/ 1KB pages

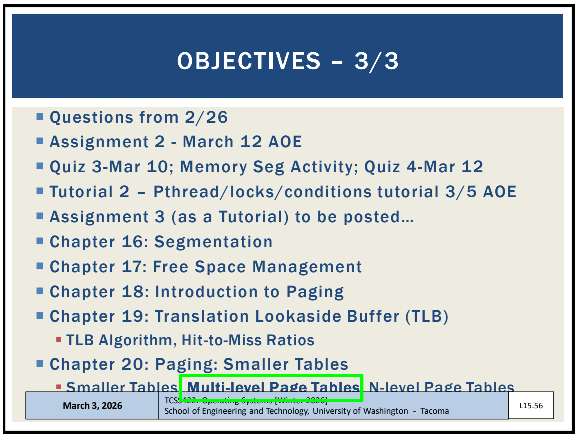
A 16KB Address Space with 1KB Pages

| PFN | valid | prot | present | dirty |
|---|---|---|---|---|
| | | | | 0 |
| | | | | - |
| | | | | - |
| | | | | - |
| 15 | 1 | rw- | 1 | 1 |
| ... | ... | ... | ... | ... |
| - | 0 | - | - | - |
| 3 | 1 | rw- | 1 | 1 |
| 23 | 1 | rw- | 1 | 1 |

A Page Table For 16KB Address Space

Most of the page table is unused and full of wasted space. (73%)

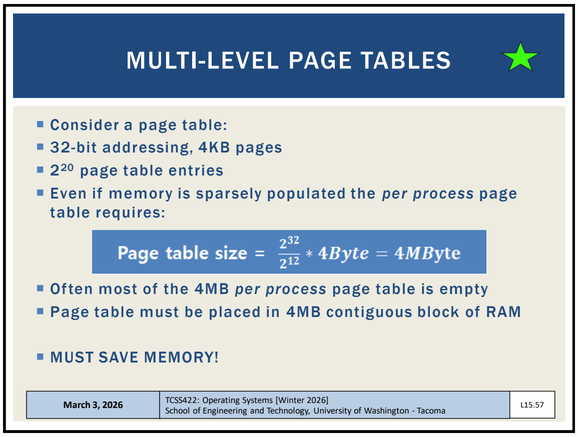
55



## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, **Multi-level Page Tables**, N-level Page Tables

56



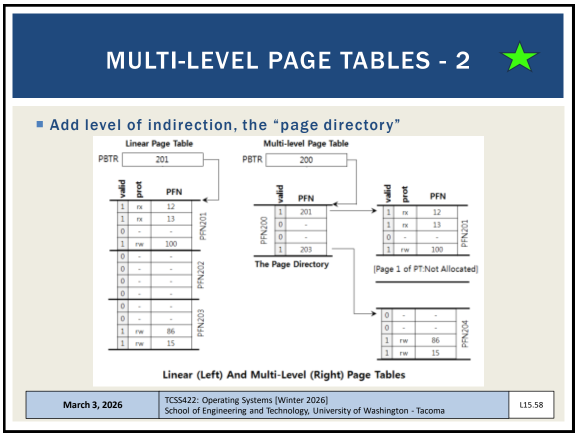
## MULTI-LEVEL PAGE TABLES ★

- Consider a page table:
- 32-bit addressing, 4KB pages
- $2^{20}$ page table entries
- Even if memory is sparsely populated the *per process* page table requires:

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

- Often most of the 4MB *per process* page table is empty
- Page table must be placed in 4MB contiguous block of RAM
- MUST SAVE MEMORY!

57



## MULTI-LEVEL PAGE TABLES - 2 ★

- Add level of indirection, the "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

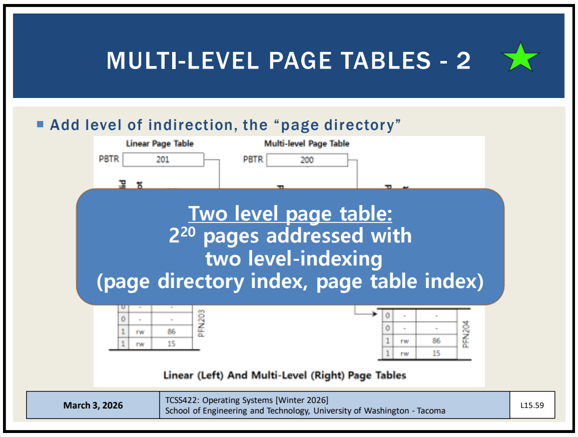
| valid | prot | PFN | |
|---|---|---|---|
| 1 | rx | 12 | PFN201 |
| 1 | rx | 13 | |
| 0 | - | - | |
| 1 | rw | 100 | |

[Page 1 of PT:Not Allocated]

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

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

58



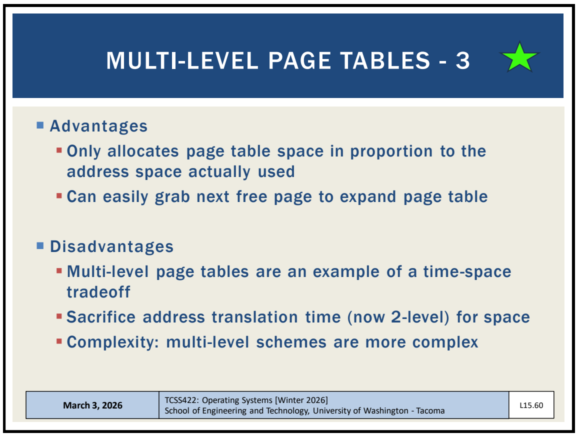
## MULTI-LEVEL PAGE TABLES - 2 ★

- Add level of indirection, the "page directory"

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

Two level page table:
$2^{20}$ pages addressed with two level-indexing
(page directory index, page table index)

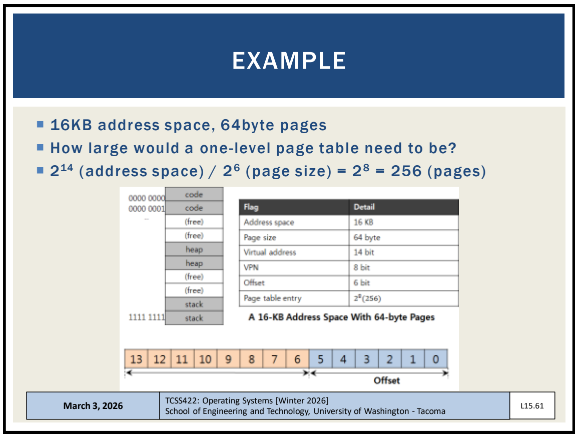
59



## MULTI-LEVEL PAGE TABLES - 3 ★

- Advantages
  - Only allocates page table space in proportion to the address space actually used
  - Can easily grab next free page to expand page table
- Disadvantages
  - Multi-level page tables are an example of a time-space tradeoff
  - Sacrifice address translation time (now 2-level) for space
  - Complexity: multi-level schemes are more complex

60



## EXAMPLE

- 16KB address space, 64byte pages
- How large would a one-level page table need to be?
- $2^{14}$ (address space) / $2^6$ (page size) = $2^8$ = 256 (pages)

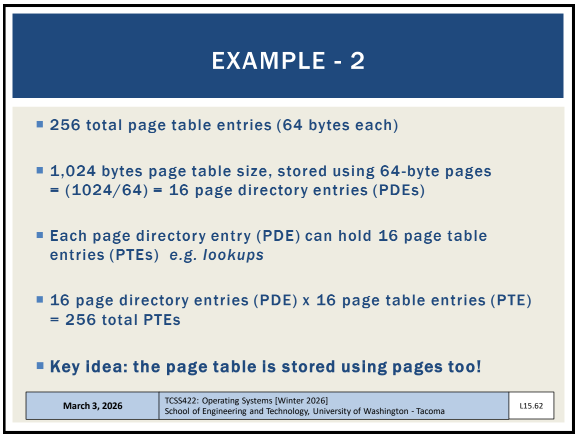
0000 0000
0000 0001
...
1111 1111
code
code
(free)
(free)
heap
heap
(free)
(free)
stack
stack

| 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
Offset

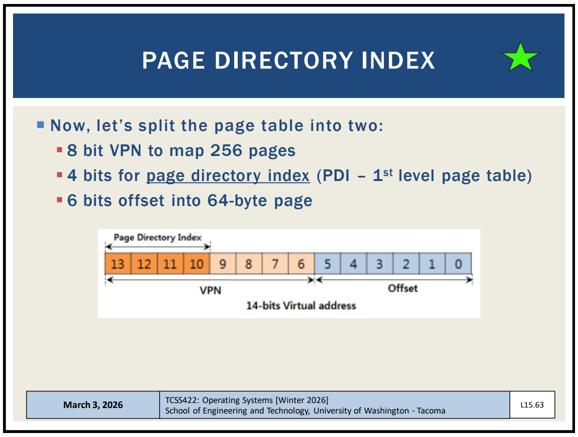
61



## EXAMPLE - 2

- 256 total page table entries (64 bytes each)
- 1,024 bytes page table size, stored using 64-byte pages = (1024/64) = 16 page directory entries (PDEs)
- Each page directory entry (PDE) can hold 16 page table entries (PTEs) *e.g. lookups*
- 16 page directory entries (PDE) x 16 page table entries (PTE) = 256 total PTEs
- **Key idea: the page table is stored using pages too!**

62

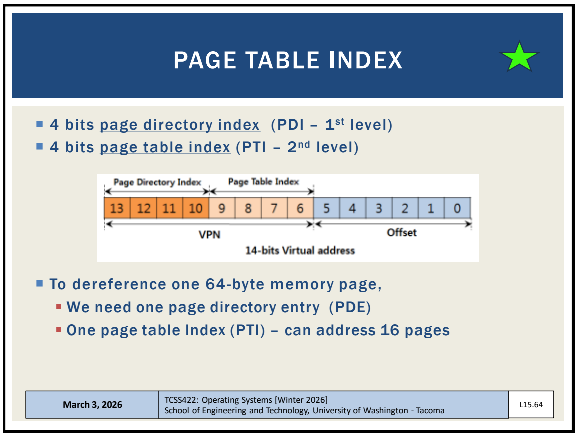


## PAGE DIRECTORY INDEX ★

- Now, let's split the page table into two:
  - 8 bit VPN to map 256 pages
  - 4 bits for page directory index (PDI – 1st level page table)
  - 6 bits offset into 64-byte page

Page Directory Index
13 12 11 10 9 8 7 6 5 4 3 2 1 0
VPN
Offset
14-bits Virtual address

63



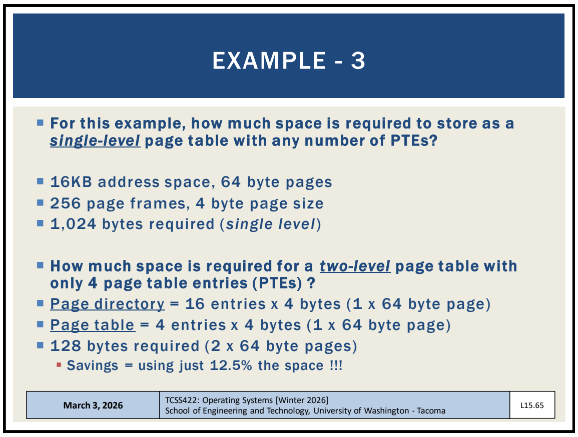
## PAGE TABLE INDEX ★

- 4 bits page directory index (PDI – 1st level)
- 4 bits page table index (PTI – 2nd level)

Page Directory Index
Page Table Index
13 12 11 10 9 8 7 6 5 4 3 2 1 0
VPN
Offset
14-bits Virtual address

- To dereference one 64-byte memory page,
  - We need one page directory entry (PDE)
  - One page table Index (PTI) – can address 16 pages

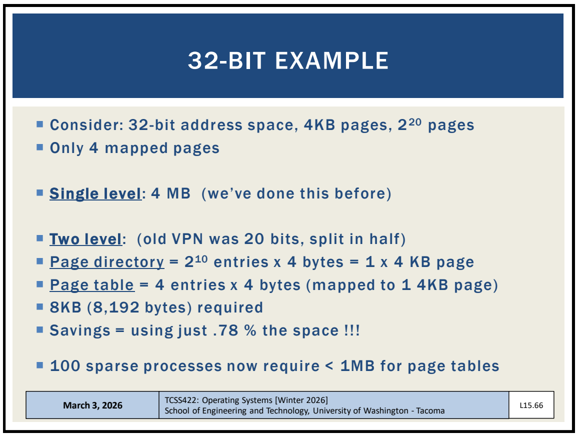
64



## EXAMPLE - 3

- **For this example, how much space is required to store as a *single-level* page table with any number of PTEs?**
- 16KB address space, 64 byte pages
- 256 page frames, 4 byte page size
- 1,024 bytes required (*single level*)
- **How much space is required for a *two-level* page table with only 4 page table entries (PTEs) ?**
- Page directory = 16 entries x 4 bytes (1 x 64 byte page)
- Page table = 4 entries x 4 bytes (1 x 64 byte page)
- 128 bytes required (2 x 64 byte pages)
  - Savings = using just 12.5% the space !!!

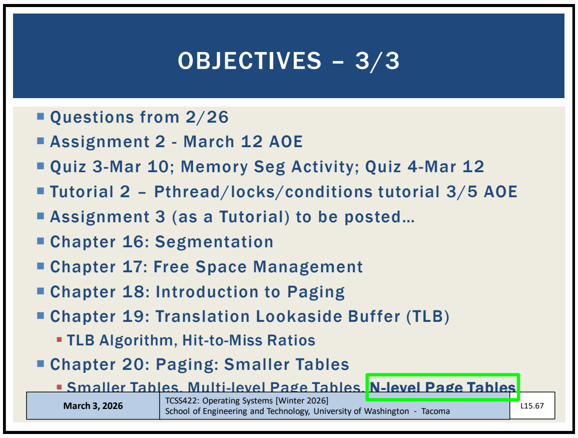
65



## 32-BIT EXAMPLE

- Consider: 32-bit address space, 4KB pages, $2^{20}$ pages
- Only 4 mapped pages
- **Single level**: 4 MB (we've done this before)
- **Two level**: (old VPN was 20 bits, split in half)
- Page directory = $2^{10}$ entries x 4 bytes = 1 x 4 KB page
- Page table = 4 entries x 4 bytes (mapped to 1 4KB page)
- 8KB (8,192 bytes) required
- Savings = using just .78 % the space !!!
- 100 sparse processes now require < 1MB for page tables

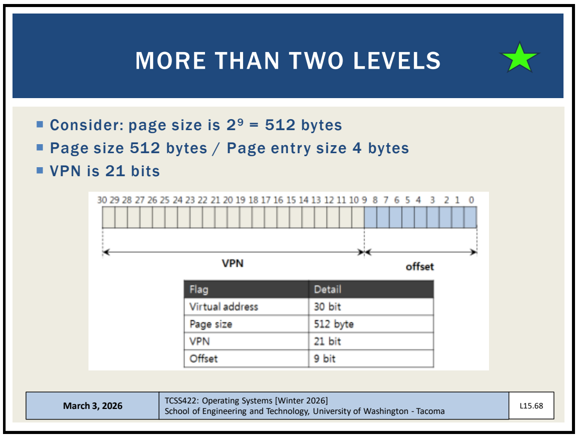
66



## OBJECTIVES – 3/3

- Questions from 2/26
- Assignment 2 - March 12 AOE
- Quiz 3-Mar 10; Memory Seg Activity; Quiz 4-Mar 12
- Tutorial 2 – Pthread/locks/conditions tutorial 3/5 AOE
- Assignment 3 (as a Tutorial) to be posted…
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

67

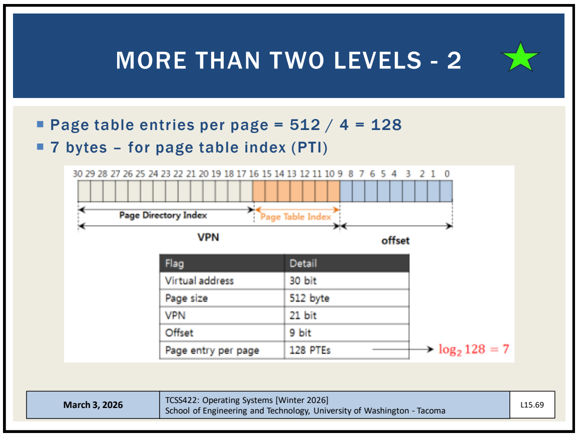


## MORE THAN TWO LEVELS

- Consider: page size is $2^9$ = 512 bytes
- Page size 512 bytes / Page entry size 4 bytes
- VPN is 21 bits

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

68



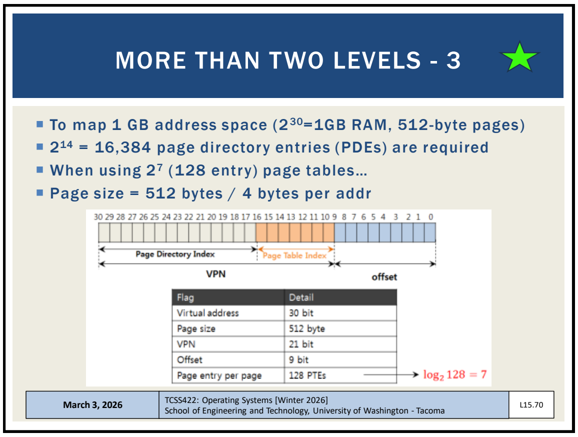
## MORE THAN TWO LEVELS - 2

- Page table entries per page = 512 / 4 = 128
- 7 bytes – for page table index (PTI)

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

$\rightarrow \log_2 128 = 7$

69



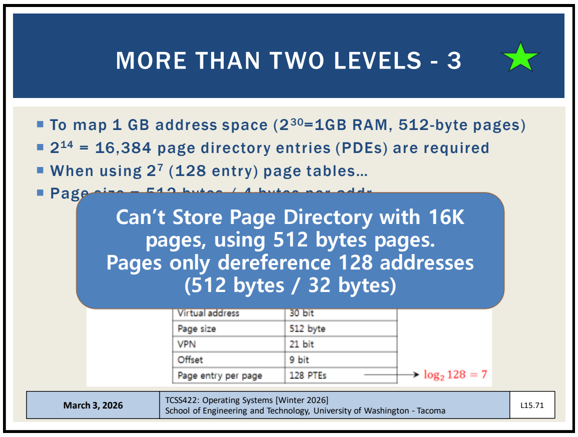
## MORE THAN TWO LEVELS - 3

- To map 1 GB address space ($2^{30}$=1GB RAM, 512-byte pages)
- $2^{14}$ = 16,384 page directory entries (PDEs) are required
- When using $2^7$ (128 entry) page tables…
- Page size = 512 bytes / 4 bytes per addr

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

$\rightarrow \log_2 128 = 7$

70



## MORE THAN TWO LEVELS - 3

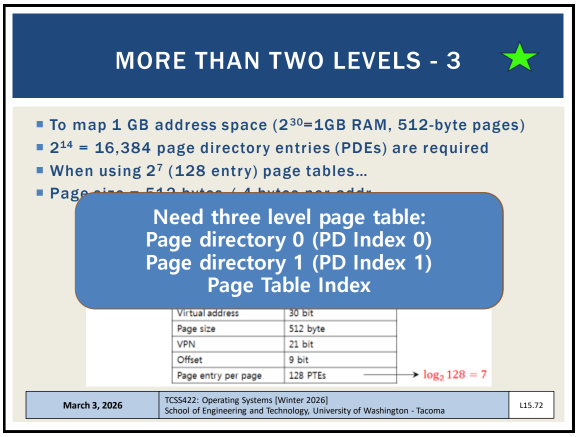
- To map 1 GB address space ($2^{30}$=1GB RAM, 512-byte pages)
- $2^{14}$ = 16,384 page directory entries (PDEs) are required
- When using $2^7$ (128 entry) page tables…
- Page size = 512 bytes / 4 bytes per addr

Can't Store Page Directory with 16K pages, using 512 bytes pages. Pages only dereference 128 addresses (512 bytes / 32 bytes)

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

$\rightarrow \log_2 128 = 7$

71



## MORE THAN TWO LEVELS - 3

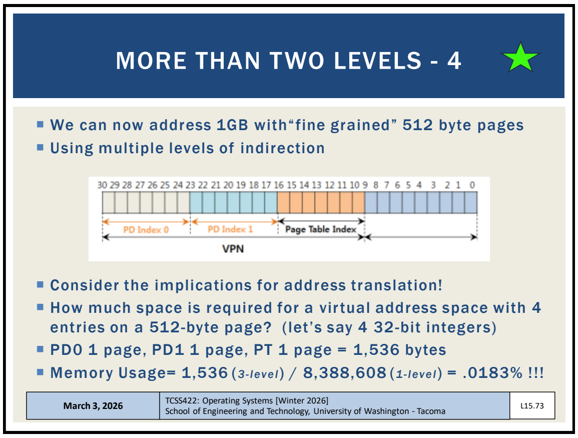
- To map 1 GB address space ($2^{30}$=1GB RAM, 512-byte pages)
- $2^{14}$ = 16,384 page directory entries (PDEs) are required
- When using $2^7$ (128 entry) page tables…
- Page size = 512 bytes / 4 bytes per addr

Need three level page table:
Page directory 0 (PD Index 0)
Page directory 1 (PD Index 1)
Page Table Index

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

$\rightarrow \log_2 128 = 7$

72

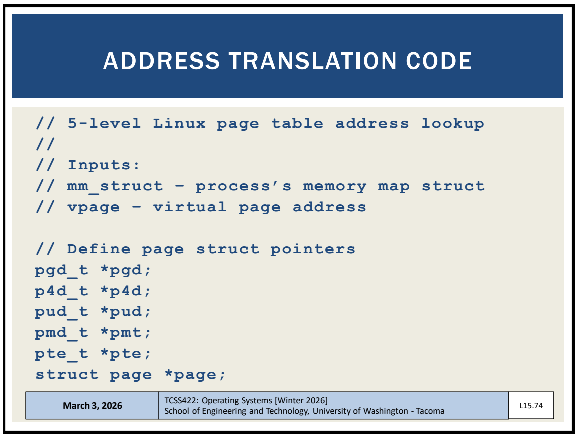


## MORE THAN TWO LEVELS - 4 ★

- We can now address 1GB with“fine grained” 512 byte pages
- Using multiple levels of indirection

- Consider the implications for address translation!
- How much space is required for a virtual address space with 4 entries on a 512-byte page? (let’s say 4 32-bit integers)
- PD0 1 page, PD1 1 page, PT 1 page = 1,536 bytes
- Memory Usage= 1,536 (*3-level*) / 8,388,608 (*1-level*) = .0183% !!!

73

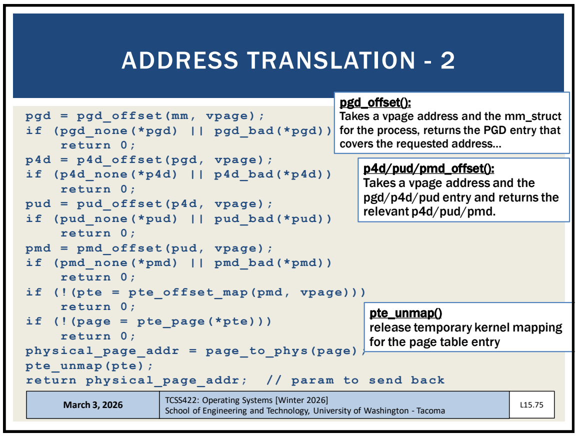


## ADDRESS TRANSLATION CODE

```
// 5-level Linux page table address lookup
//
// Inputs:
// mm_struct – process’s memory map struct
// vpage – virtual page address

// Define page struct pointers
pgd_t *pgd;
p4d_t *p4d;
pud_t *pud;
pmd_t *pmt;
pte_t *pte;
struct page *page;
```

74



## ADDRESS TRANSLATION - 2

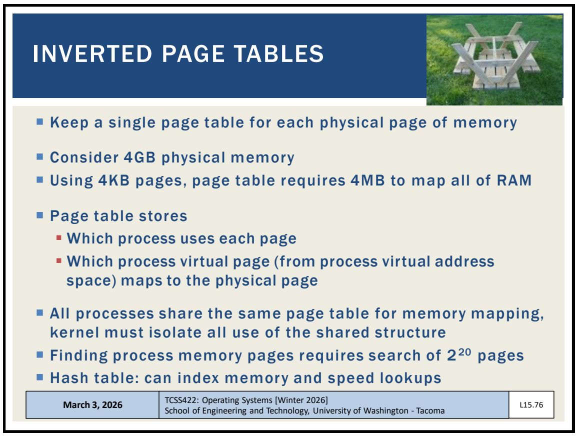
```
pgd = pgd_offset(mm, vpage);
if (pgd_none(*pgd) || pgd_bad(*pgd))
     return 0;
p4d = p4d_offset(pgd, vpage);
if (p4d_none(*p4d) || p4d_bad(*p4d))
     return 0;
pud = pud_offset(p4d, vpage);
if (pud_none(*pud) || pud_bad(*pud))
     return 0;
pmd = pmd_offset(pud, vpage);
if (pmd_none(*pmd) || pmd_bad(*pmd))
     return 0;
if (!(pte = pte_offset_map(pmd, vpage)))
     return 0;
if (!(page = pte_page(*pte)))
     return 0;
physical_page_addr = page_to_phys(page);
pte_unmap(pte);
return physical_page_addr;  // param to send back
```

**pgd_offset():**
Takes a vpage address and the mm_struct for the process, returns the PGD entry that covers the requested address...

**p4d/pud/pmd_offset():**
Takes a vpage address and the pgd/p4d/pud entry and returns the relevant p4d/pud/pmd.

**pte_unmap()**
release temporary kernel mapping for the page table entry

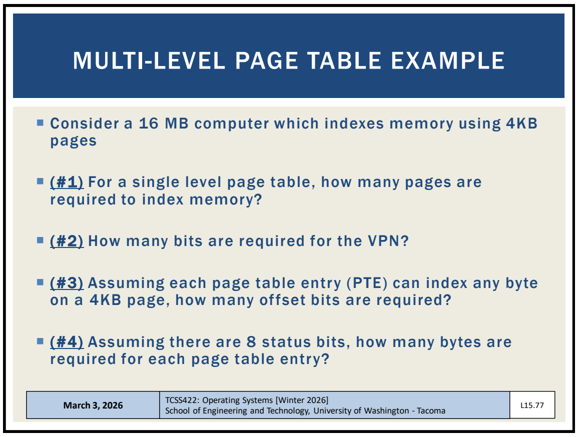
75



## INVERTED PAGE TABLES

- Keep a single page table for each physical page of memory
- Consider 4GB physical memory
- Using 4KB pages, page table requires 4MB to map all of RAM
- Page table stores
  - Which process uses each page
  - Which process virtual page (from process virtual address space) maps to the physical page
- All processes share the same page table for memory mapping, kernel must isolate all use of the shared structure
- Finding process memory pages requires search of $2^{20}$ pages
- Hash table: can index memory and speed lookups

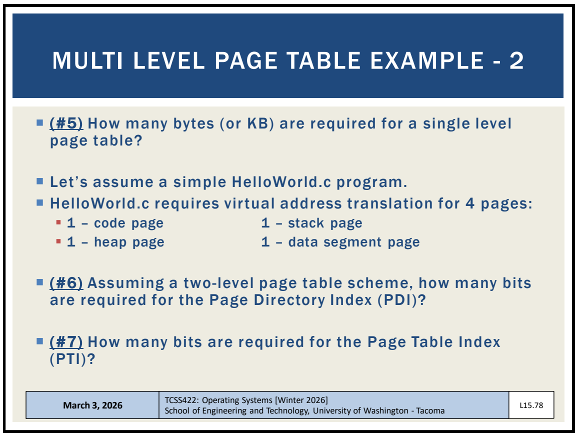
76



## MULTI-LEVEL PAGE TABLE EXAMPLE

- Consider a 16 MB computer which indexes memory using 4KB pages
- (#1) For a single level page table, how many pages are required to index memory?
- (#2) How many bits are required for the VPN?
- (#3) Assuming each page table entry (PTE) can index any byte on a 4KB page, how many offset bits are required?
- (#4) Assuming there are 8 status bits, how many bytes are required for each page table entry?

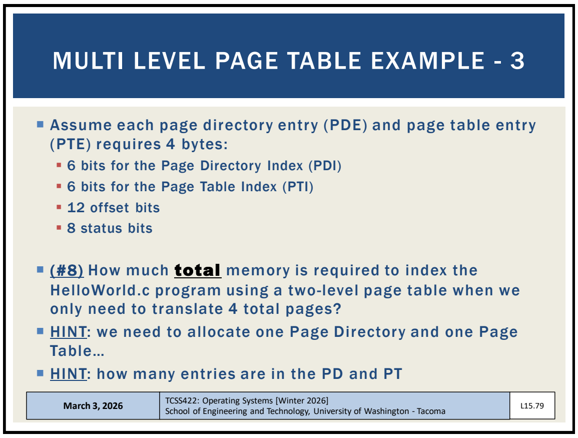
77



## MULTI LEVEL PAGE TABLE EXAMPLE - 2

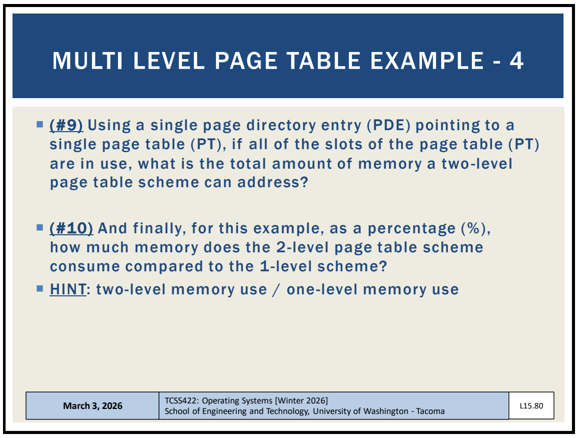
- (#5) How many bytes (or KB) are required for a single level page table?
- Let’s assume a simple HelloWorld.c program.
- HelloWorld.c requires virtual address translation for 4 pages:
  - 1 – code page
  - 1 – heap page
  - 1 – stack page
  - 1 – data segment page
- (#6) Assuming a two-level page table scheme, how many bits are required for the Page Directory Index (PDI)?
- (#7) How many bits are required for the Page Table Index (PTI)?

78



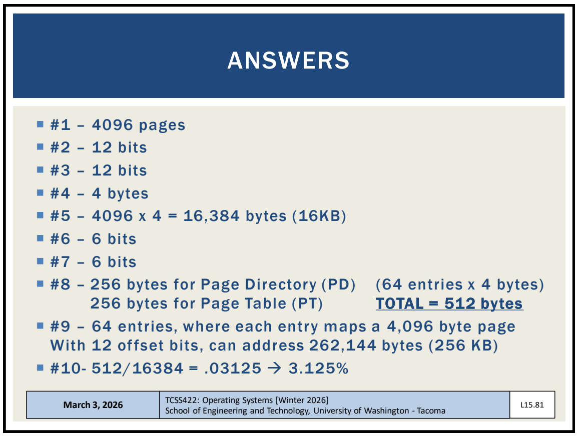
## MULTI LEVEL PAGE TABLE EXAMPLE - 3

- Assume each page directory entry (PDE) and page table entry (PTE) requires 4 bytes:
  - 6 bits for the Page Directory Index (PDI)
  - 6 bits for the Page Table Index (PTI)
  - 12 offset bits
  - 8 status bits
- (#8) How much **total** memory is required to index the HelloWorld.c program using a two-level page table when we only need to translate 4 total pages?
- HINT: we need to allocate one Page Directory and one Page Table…
- HINT: how many entries are in the PD and PT



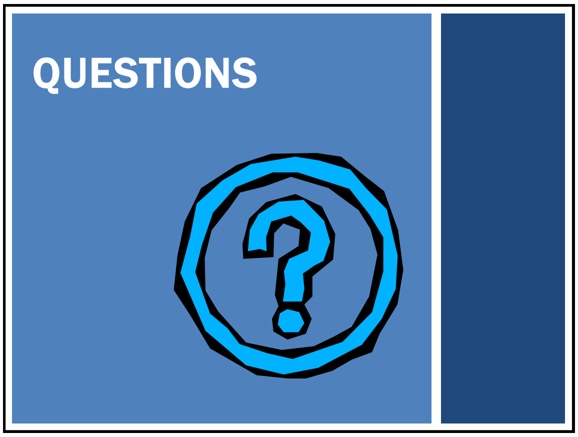
## MULTI LEVEL PAGE TABLE EXAMPLE - 4

- (#9) Using a single page directory entry (PDE) pointing to a single page table (PT), if all of the slots of the page table (PT) are in use, what is the total amount of memory a two-level page table scheme can address?
- (#10) And finally, for this example, as a percentage (%), how much memory does the 2-level page table scheme consume compared to the 1-level scheme?
- HINT: two-level memory use / one-level memory use



## ANSWERS

- #1 – 4096 pages
- #2 – 12 bits
- #3 – 12 bits
- #4 – 4 bytes
- #5 – 4096 x 4 = 16,384 bytes (16KB)
- #6 – 6 bits
- #7 – 6 bits
- #8 – 256 bytes for Page Directory (PD)     (64 entries x 4 bytes)
  256 bytes for Page Table (PT)     **TOTAL = 512 bytes**
- #9 – 64 entries, where each entry maps a 4,096 byte page
  With 12 offset bits, can address 262,144 bytes (256 KB)
- #10- 512/16384 = .03125 → 3.125%



## QUESTIONS