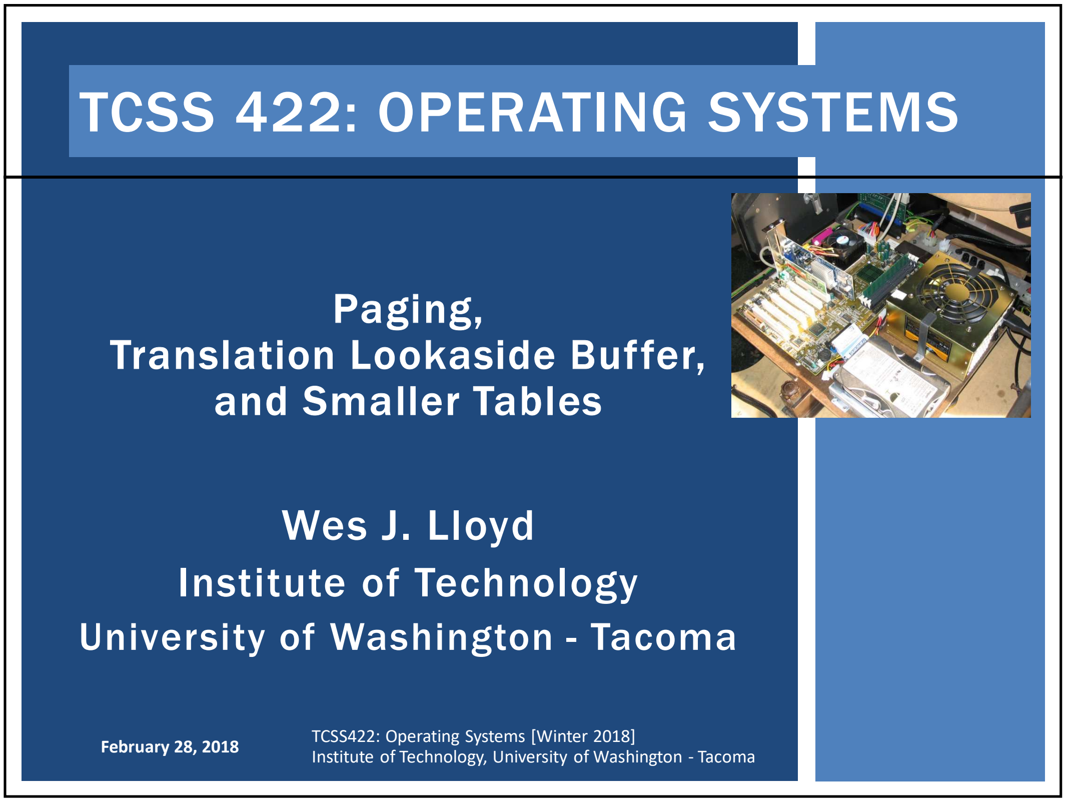
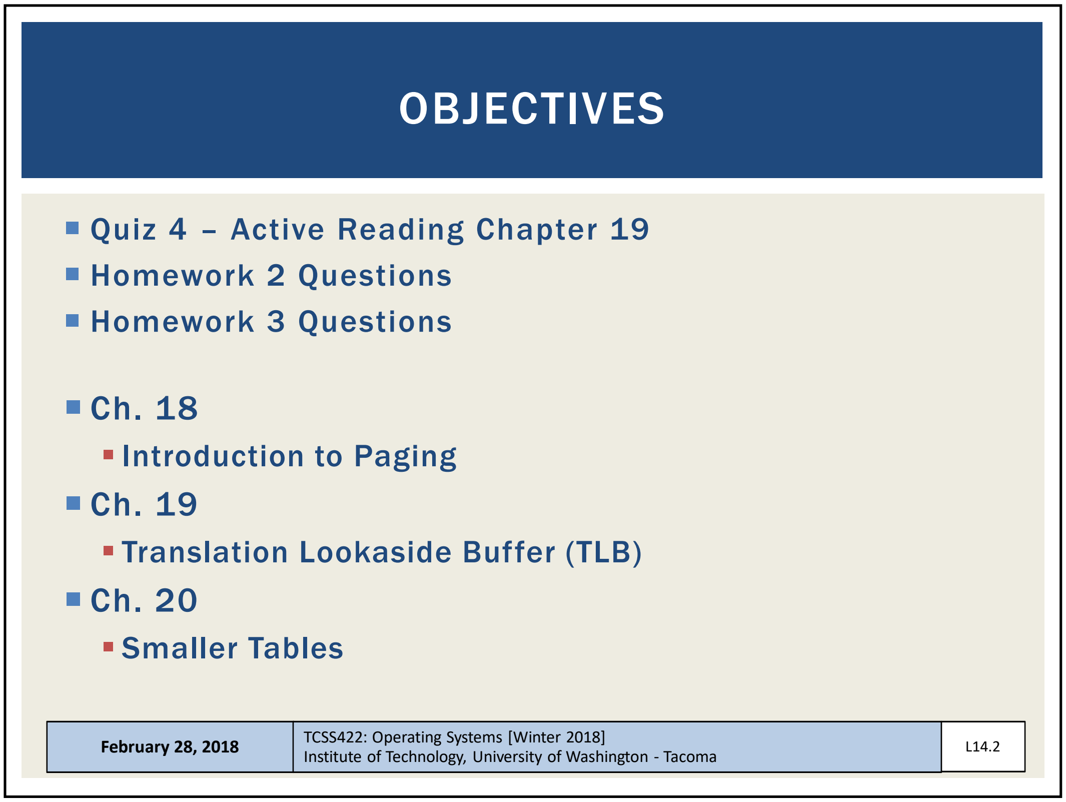


TCSS 422: OPERATING SYSTEMS

Paging,
Translation Lookaside Buffer,
and Smaller Tables

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

February 28, 2018 TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

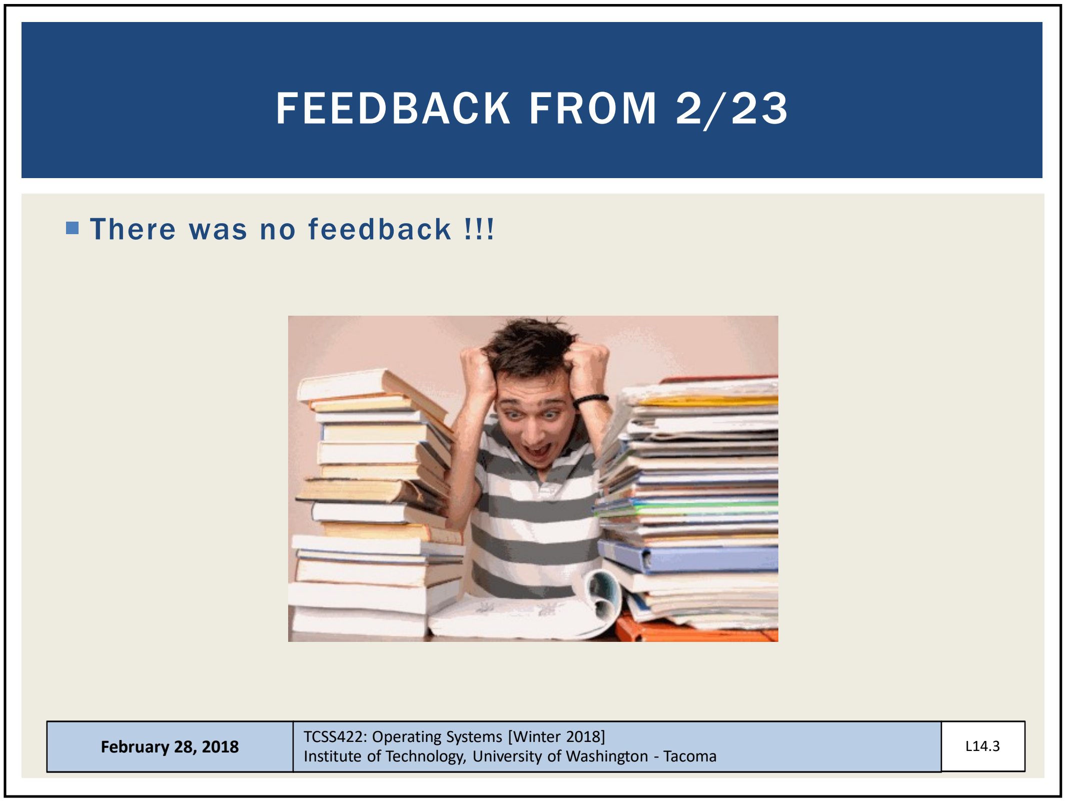


OBJECTIVES

- Quiz 4 – Active Reading Chapter 19
- Homework 2 Questions
- Homework 3 Questions

- Ch. 18
  - Introduction to Paging
- Ch. 19
  - Translation Lookaside Buffer (TLB)
- Ch. 20
  - Smaller Tables

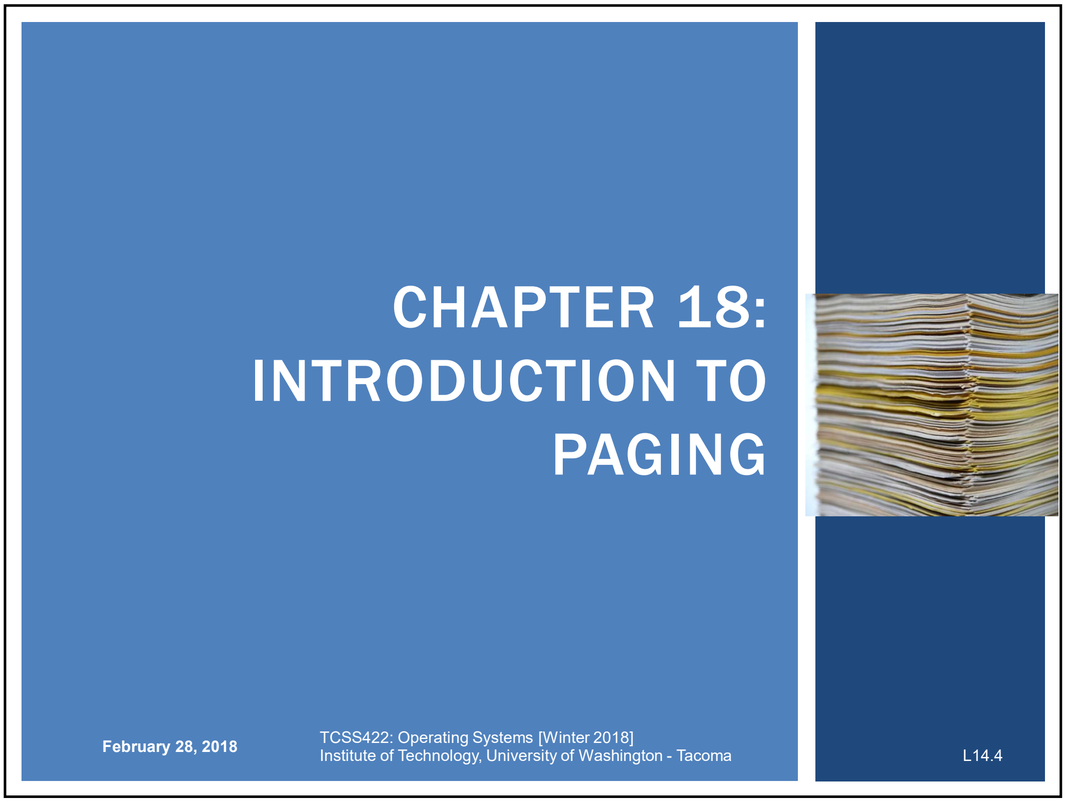
February 28, 2018	TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma	L14.2



FEEDBACK FROM 2/23

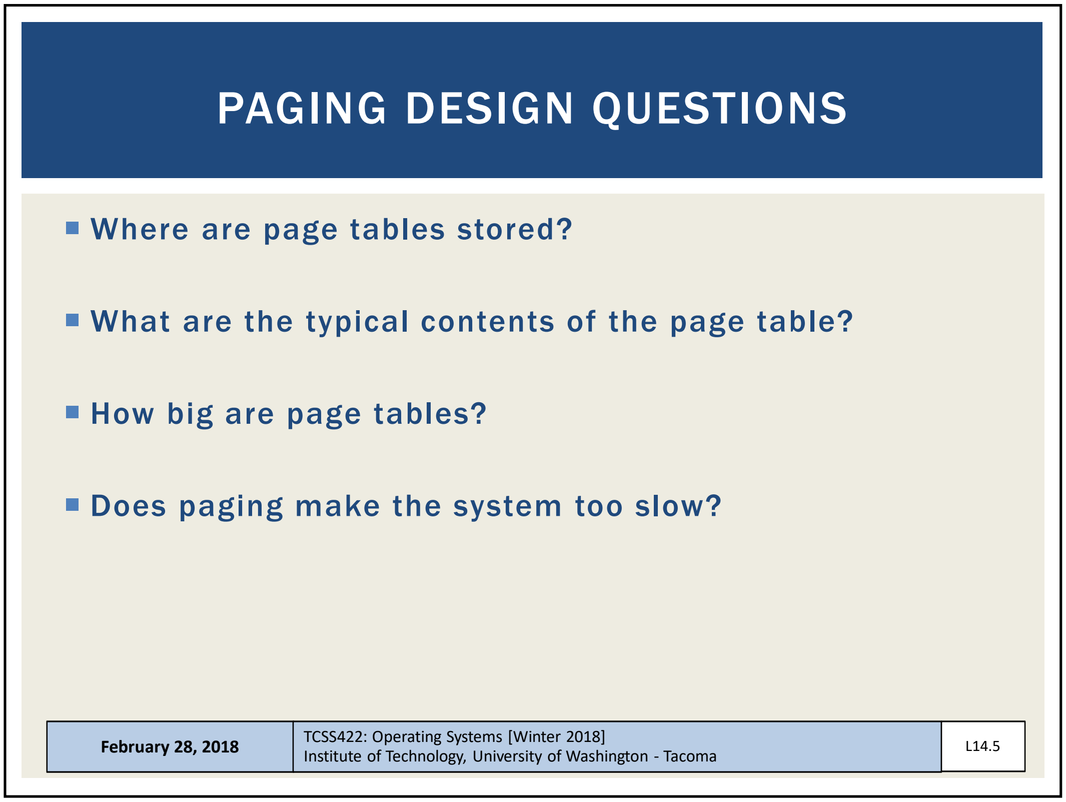
- There was no feedback !!!

February 28, 2018 | TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma | L14.3



CHAPTER 18: INTRODUCTION TO PAGING

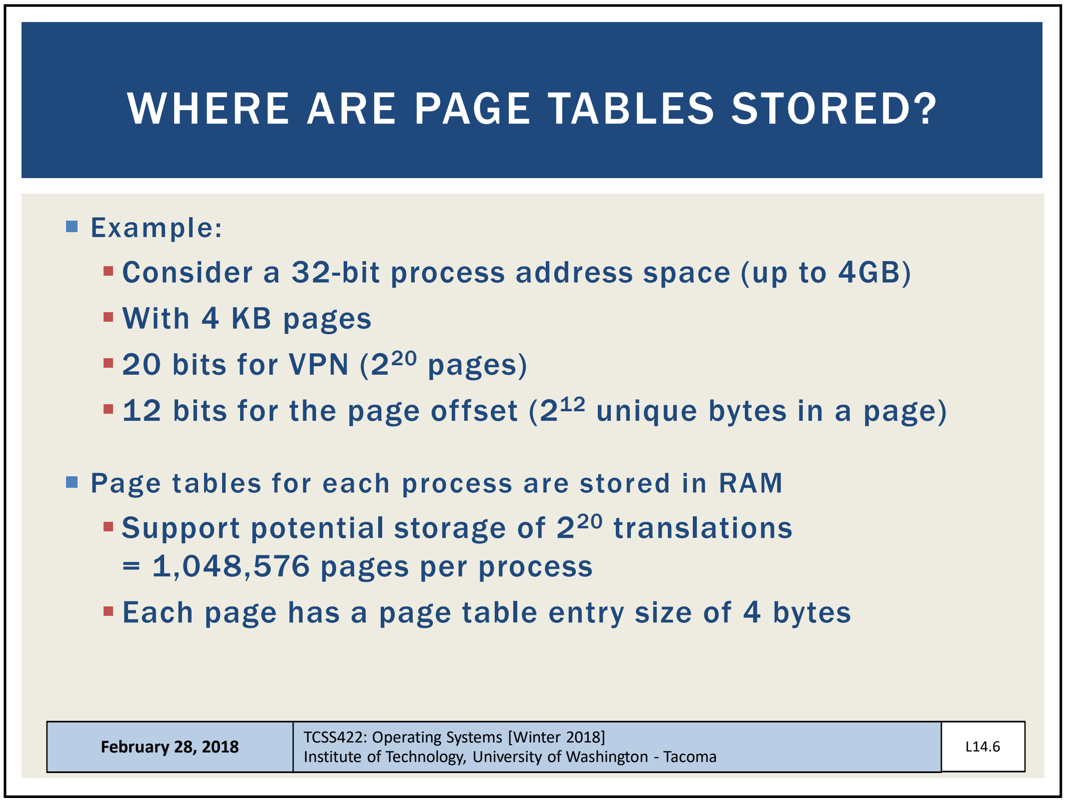
February 28, 2018 | TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma | L14.4



PAGING DESIGN QUESTIONS

- Where are page tables stored?
- What are the typical contents of the page table?
- How big are page tables?
- Does paging make the system too slow?

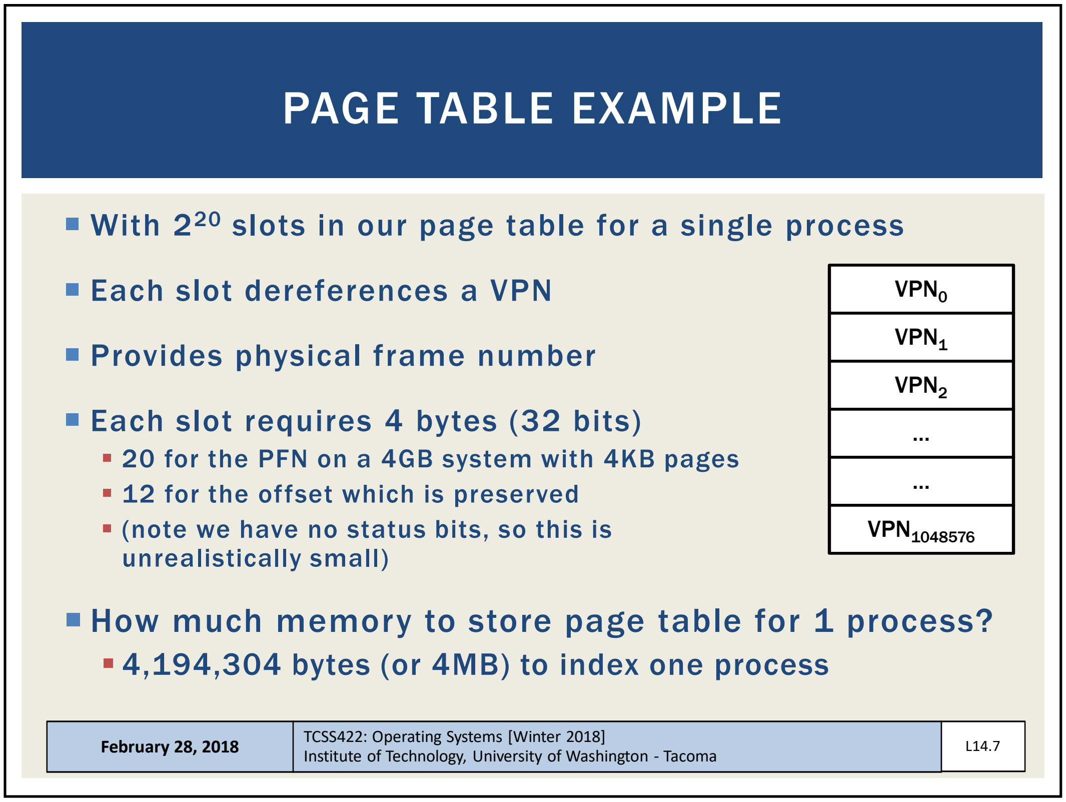
February 28, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L14.5



WHERE ARE PAGE TABLES STORED?

- Example:
  - Consider a 32-bit process address space (up to 4GB)
  - With 4 KB pages
  - 20 bits for VPN (2^{20} pages)
  - 12 bits for the page offset (2^{12} unique bytes in a page)
- Page tables for each process are stored in RAM
  - Support potential storage of 2^{20} translations = 1,048,576 pages per process
  - Each page has a page table entry size of 4 bytes

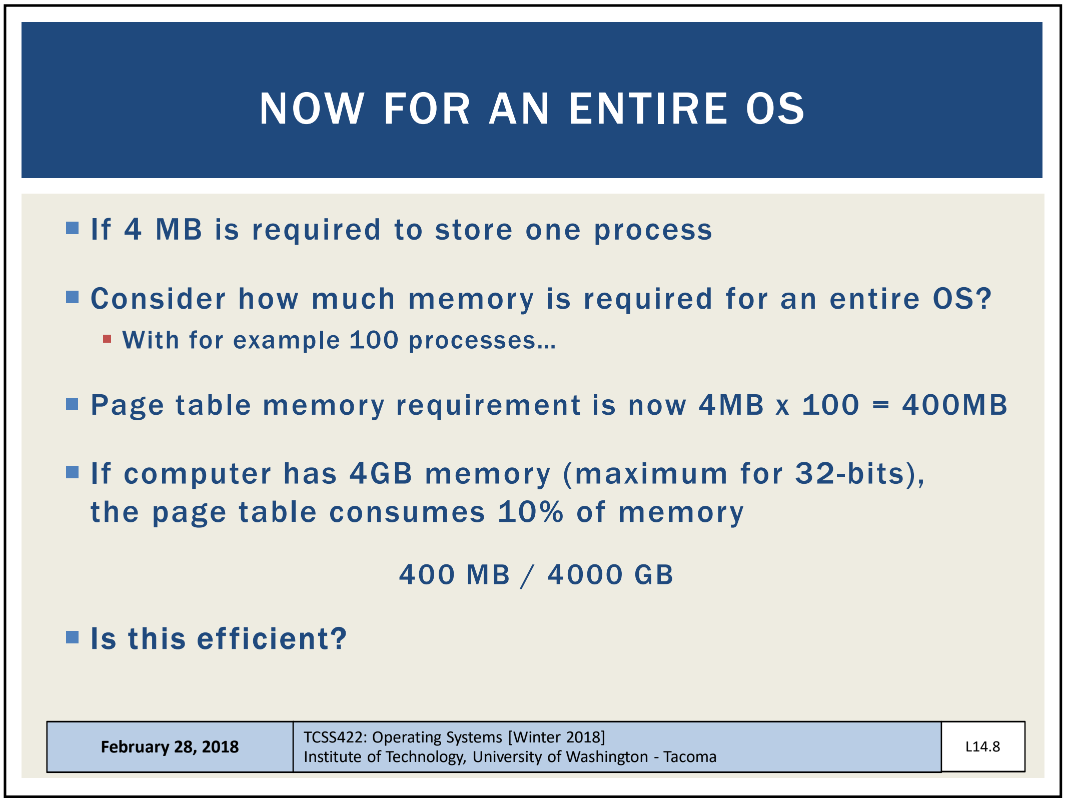
February 28, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L14.6



PAGE TABLE EXAMPLE

- With 2^{20} slots in our page table for a single process
- Each slot dereferences a VPN
- Provides physical frame number
- Each slot requires 4 bytes (32 bits)
  - 20 for the PFN on a 4GB system with 4KB pages
  - 12 for the offset which is preserved
  - (note we have no status bits, so this is unrealistically small)
- How much memory to store page table for 1 process?
  - 4,194,304 bytes (or 4MB) to index one process

VPN_0
VPN_1
VPN_2
…
…
$VPN_{1048576}$

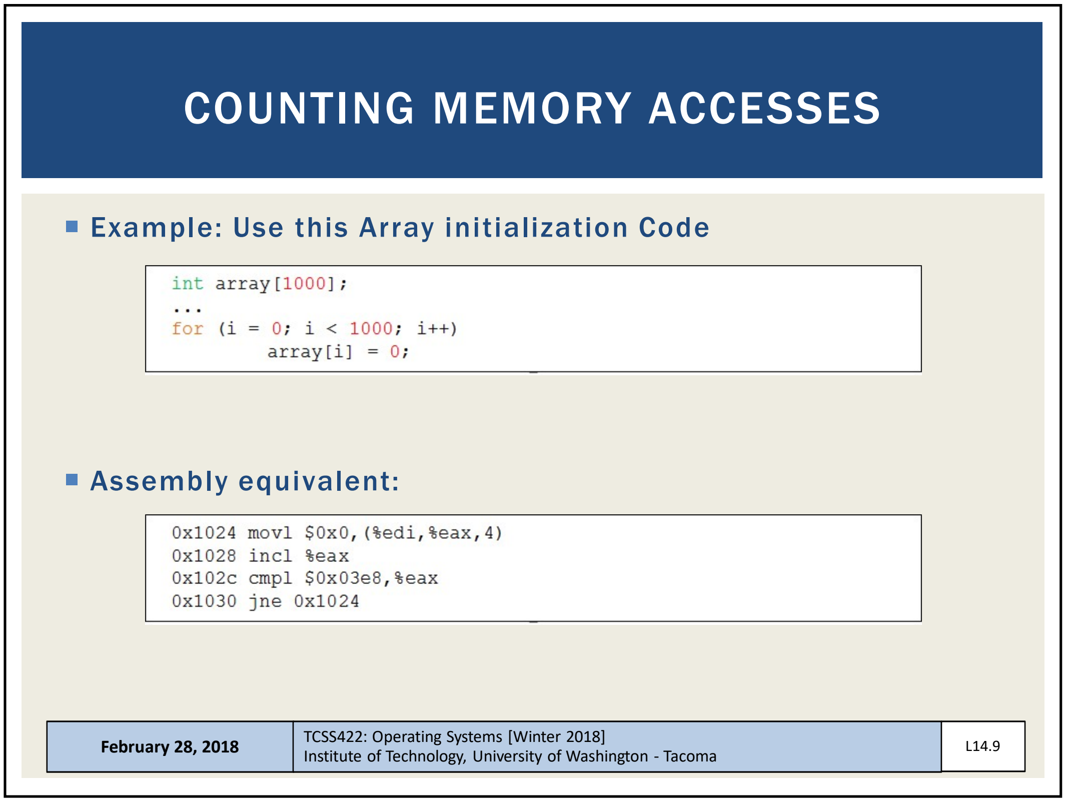


NOW FOR AN ENTIRE OS

- If 4 MB is required to store one process
- Consider how much memory is required for an entire OS?
  - With for example 100 processes…
- Page table memory requirement is now 4MB x 100 = 400MB
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

400 MB / 4000 GB

- Is this efficient?



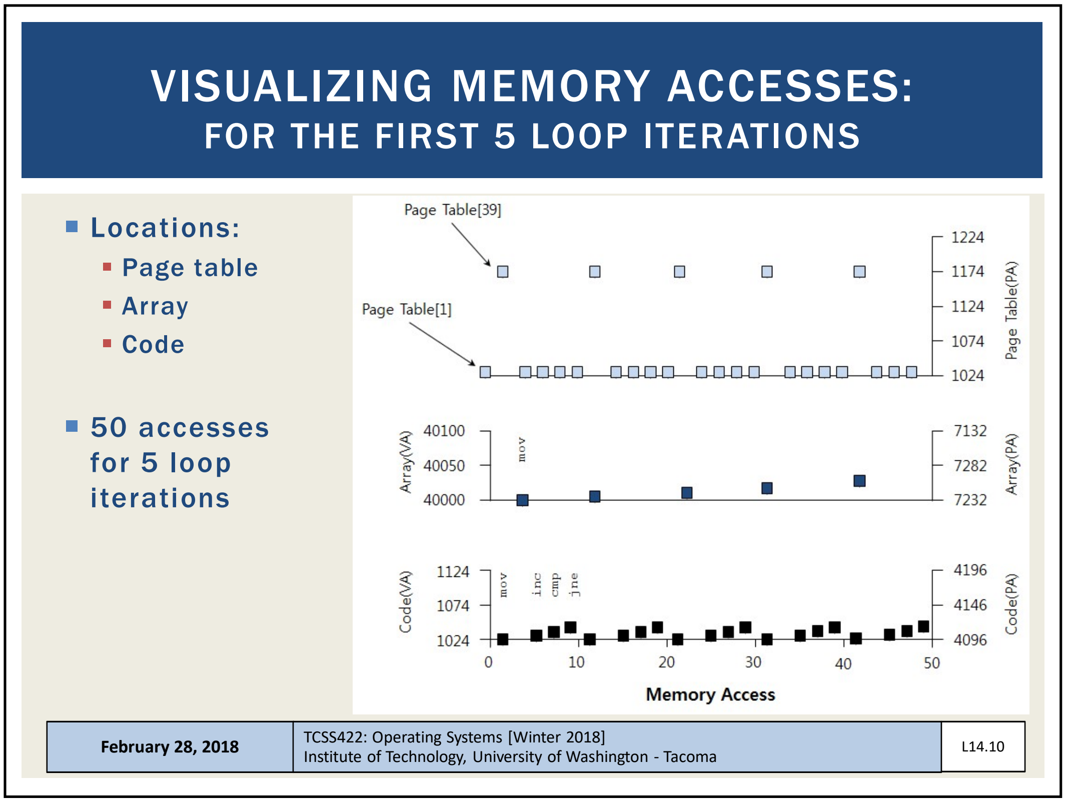
COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

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

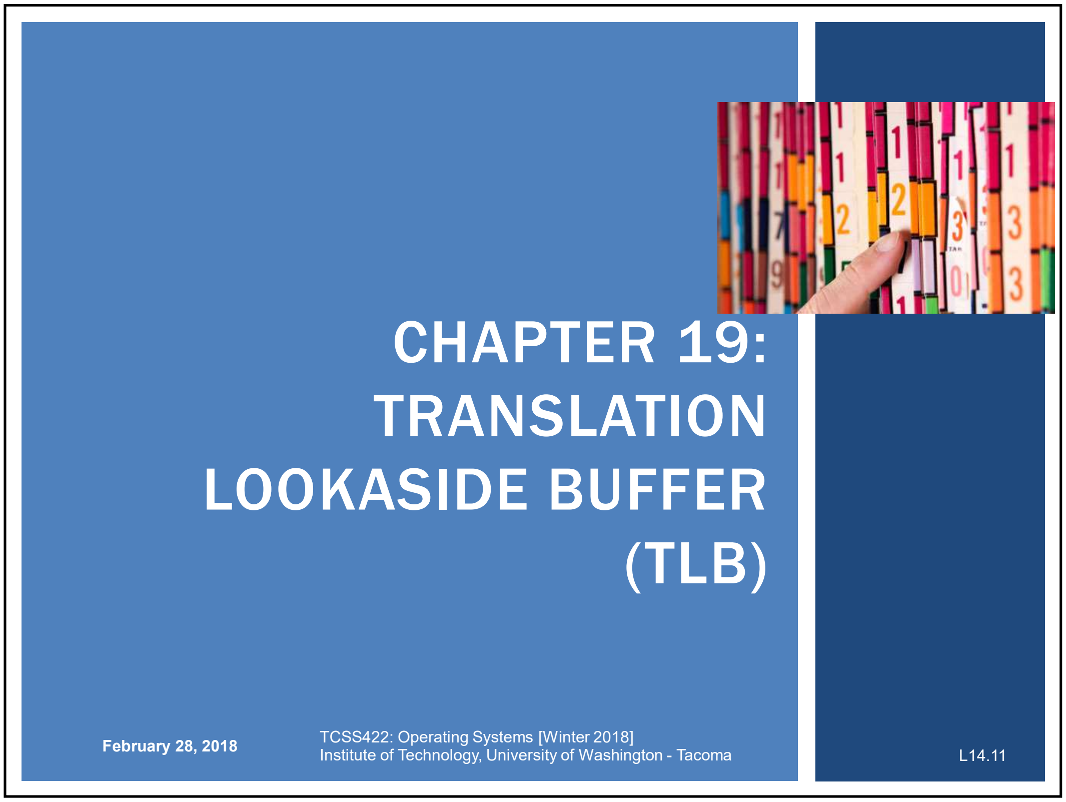
- Assembly equivalent:

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



VISUALIZING MEMORY ACCESSES: FOR THE FIRST 5 LOOP ITERATIONS

- Locations:
  - Page table
  - Array
  - Code
- 50 accesses for 5 loop iterations

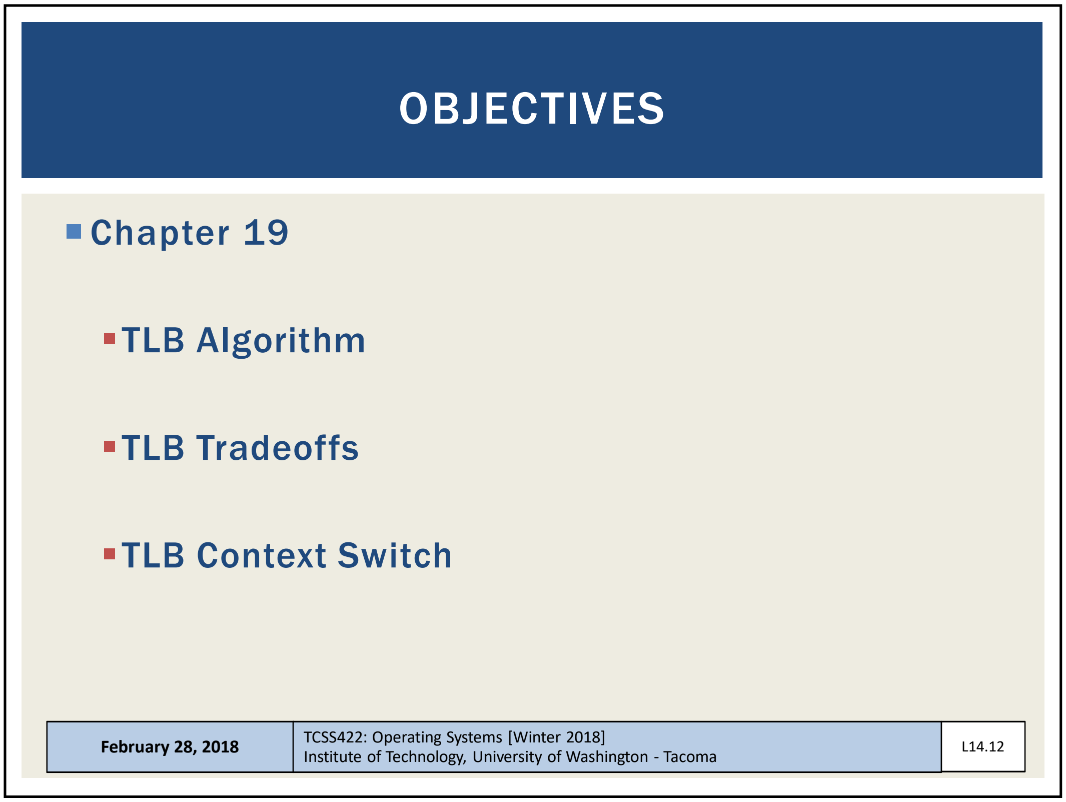


CHAPTER 19: TRANSLATION LOOKASIDE BUFFER (TLB)

February 28, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

L14.11



OBJECTIVES

- Chapter 19
  - TLB Algorithm
  - TLB Tradeoffs
  - TLB Context Switch

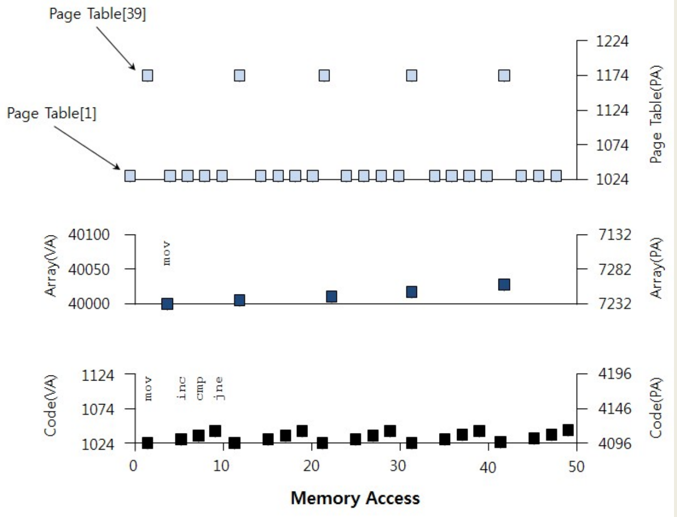
February 28, 2018	TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma	L14.12

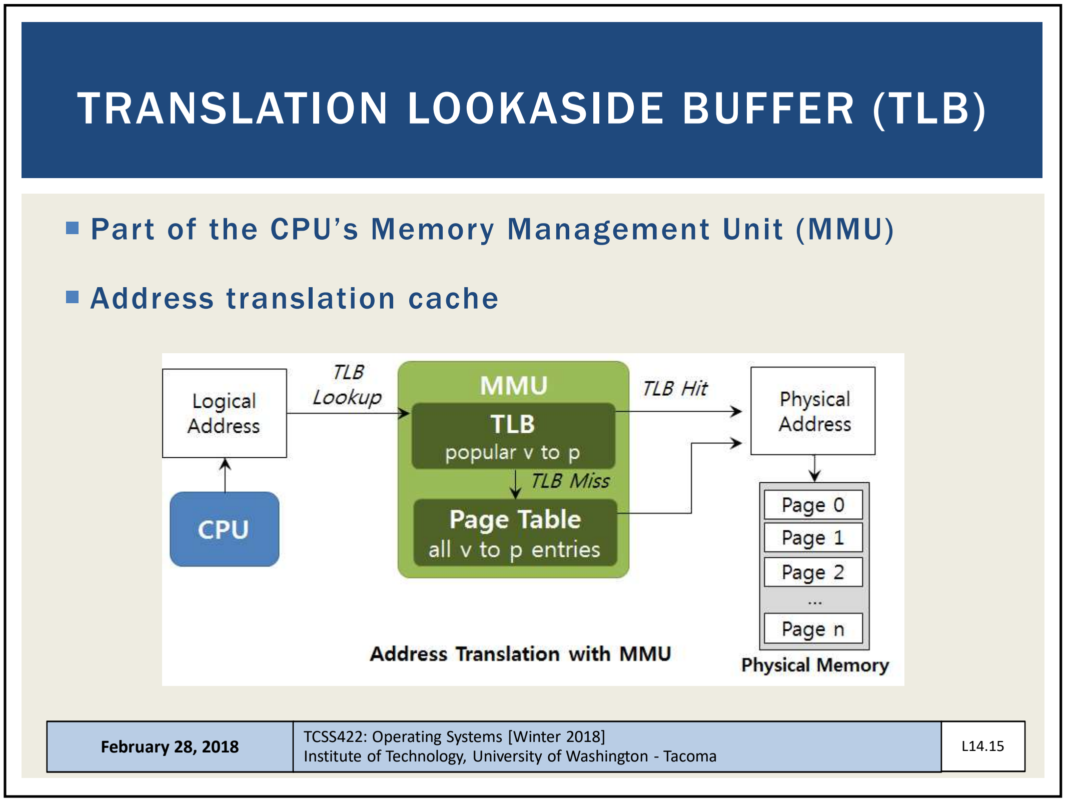
TRANSLATION LOOKASIDE BUFFER

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

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



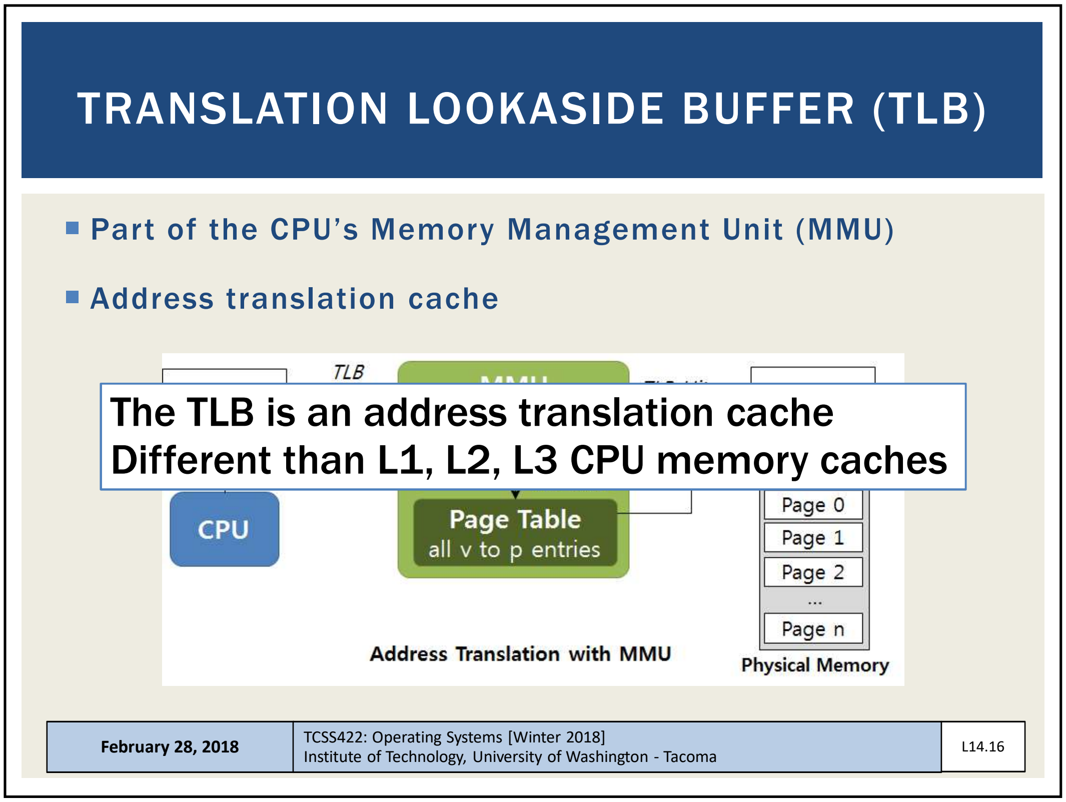


TRANSLATION LOOKASIDE BUFFER (TLB)

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

Logical Address
CPU
TLB Lookup
MMU
TLB
popular v to p
TLB Miss
Page Table
all v to p entries
TLB Hit
Physical Address
Page 0
Page 1
Page 2
...
Page n
Physical Memory

Address Translation with MMU



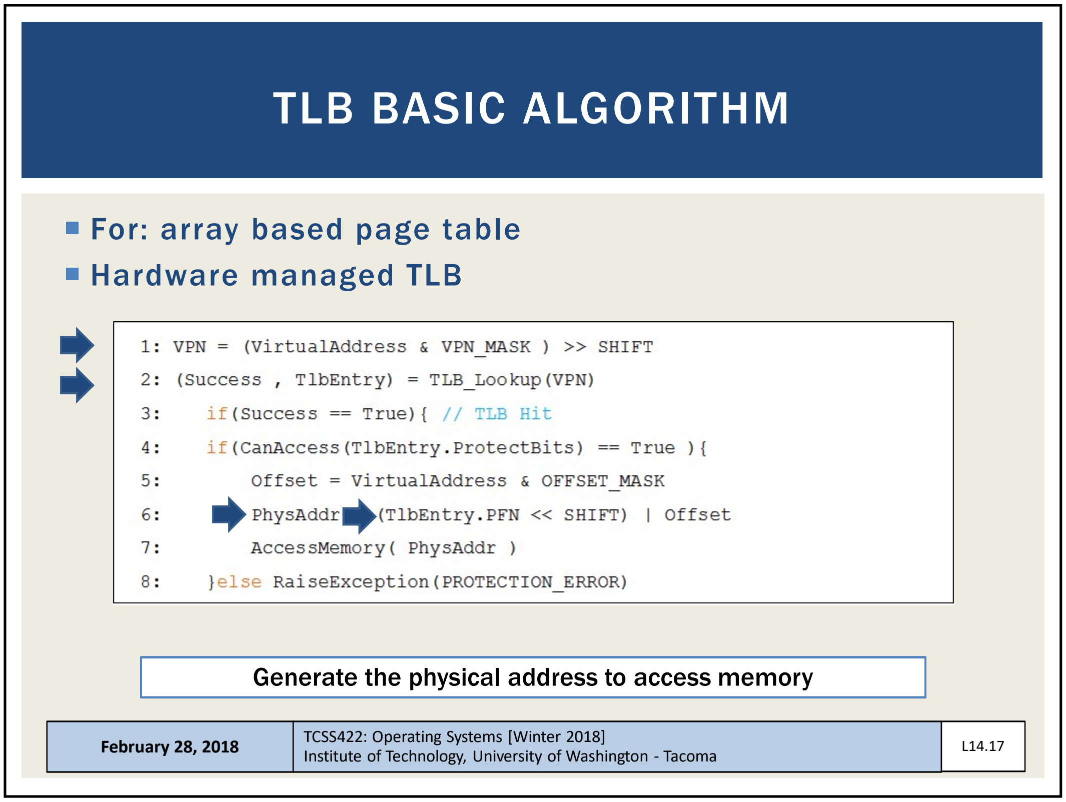
TRANSLATION LOOKASIDE BUFFER (TLB)

- 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

TLB
CPU
Page Table
all v to p entries
Page 0
Page 1
Page 2
...
Page n
Physical Memory

Address Translation with MMU



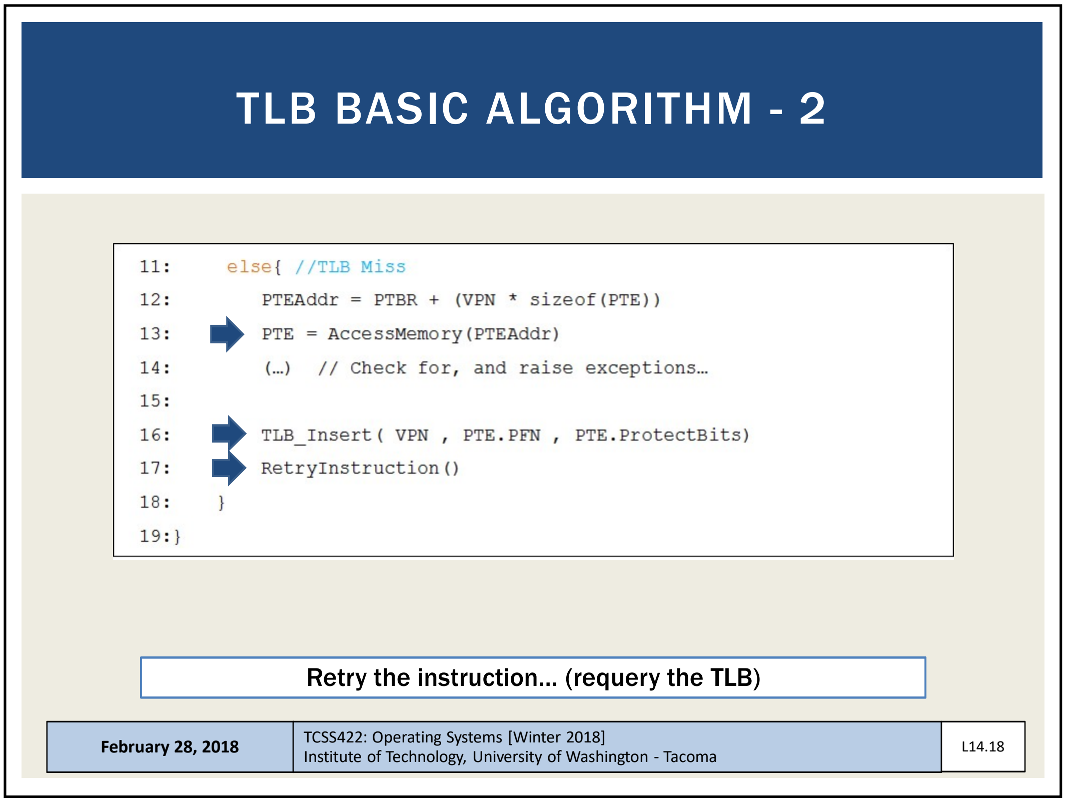
TLB BASIC ALGORITHM

- 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

February 28, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L14.17



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)

February 28, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L14.18

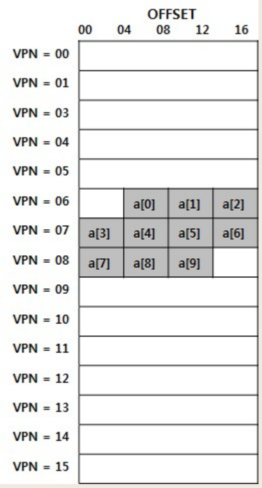
TLB – ADDRESS TRANSLATION CACHE

- 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**

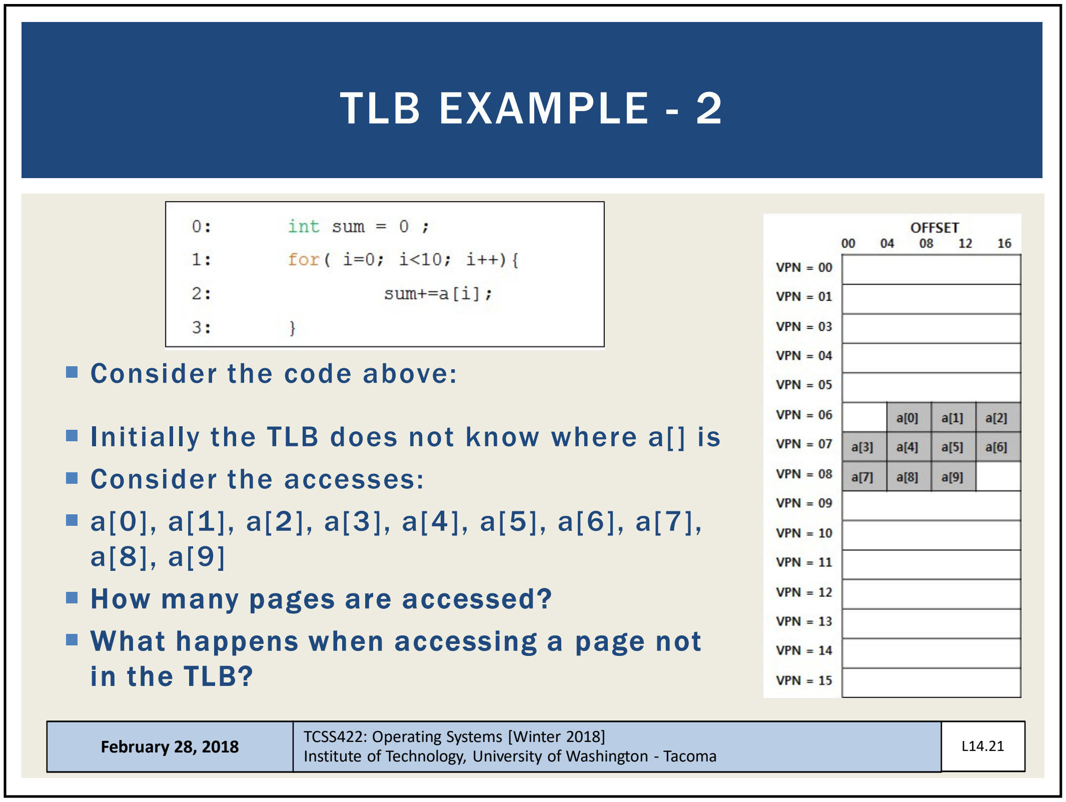
TLB EXAMPLE

```
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					



TLB EXAMPLE - 2

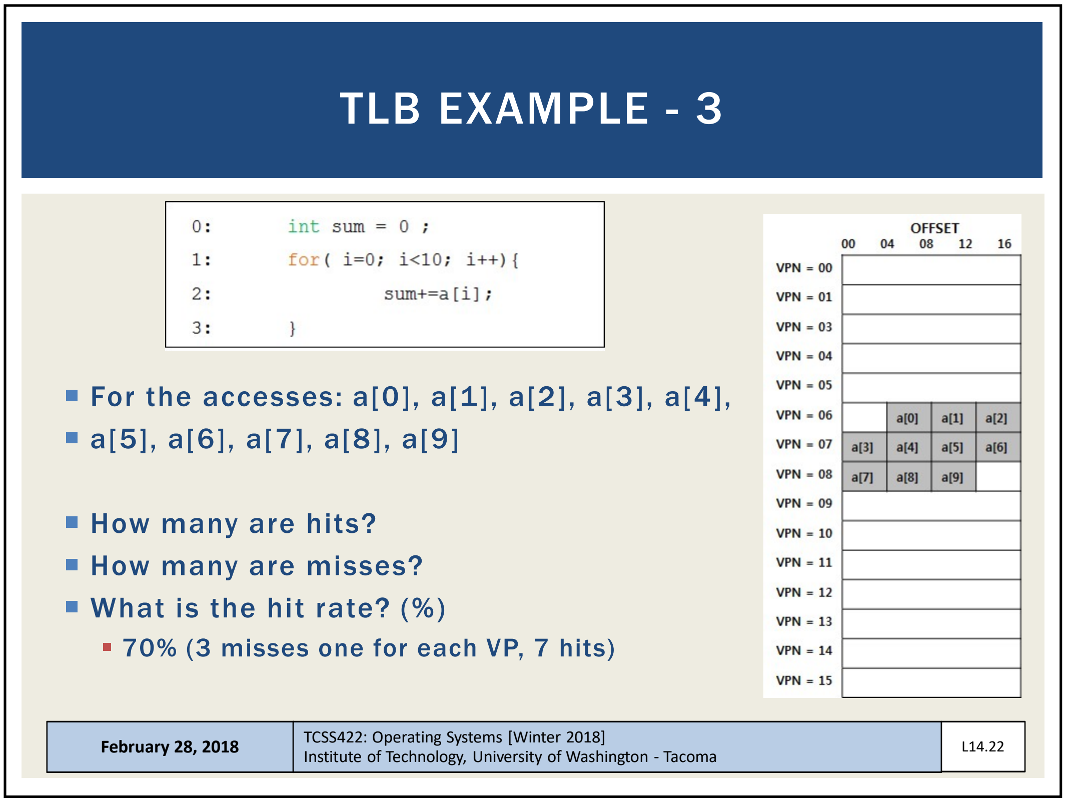
```
        int sum = 0 ;
        for( i=0; i<10; i++){
                sum+=a[i];
        }
```

- 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				

February 28, 2018 | TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma | L14.21



TLB EXAMPLE - 3

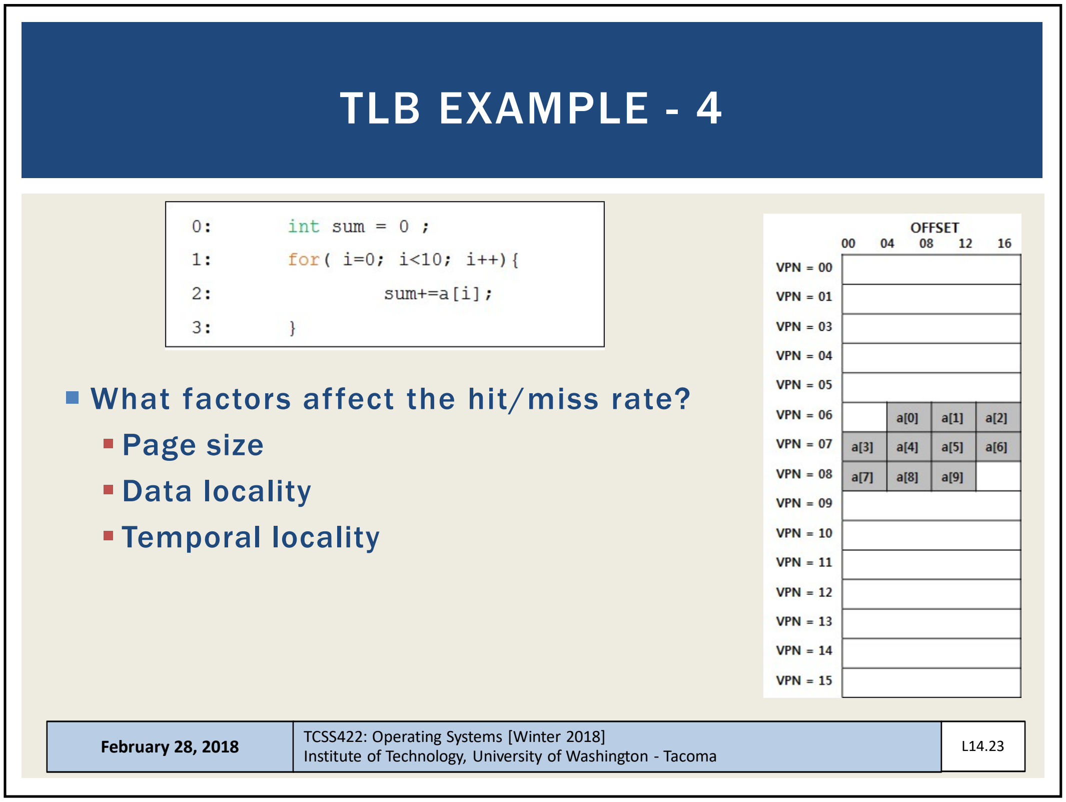
```
        int sum = 0 ;
        for( i=0; i<10; i++){
                sum+=a[i];
        }
```

- 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 are hits?
- How many are 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				

February 28, 2018 | TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma | L14.22



TLB EXAMPLE - 4

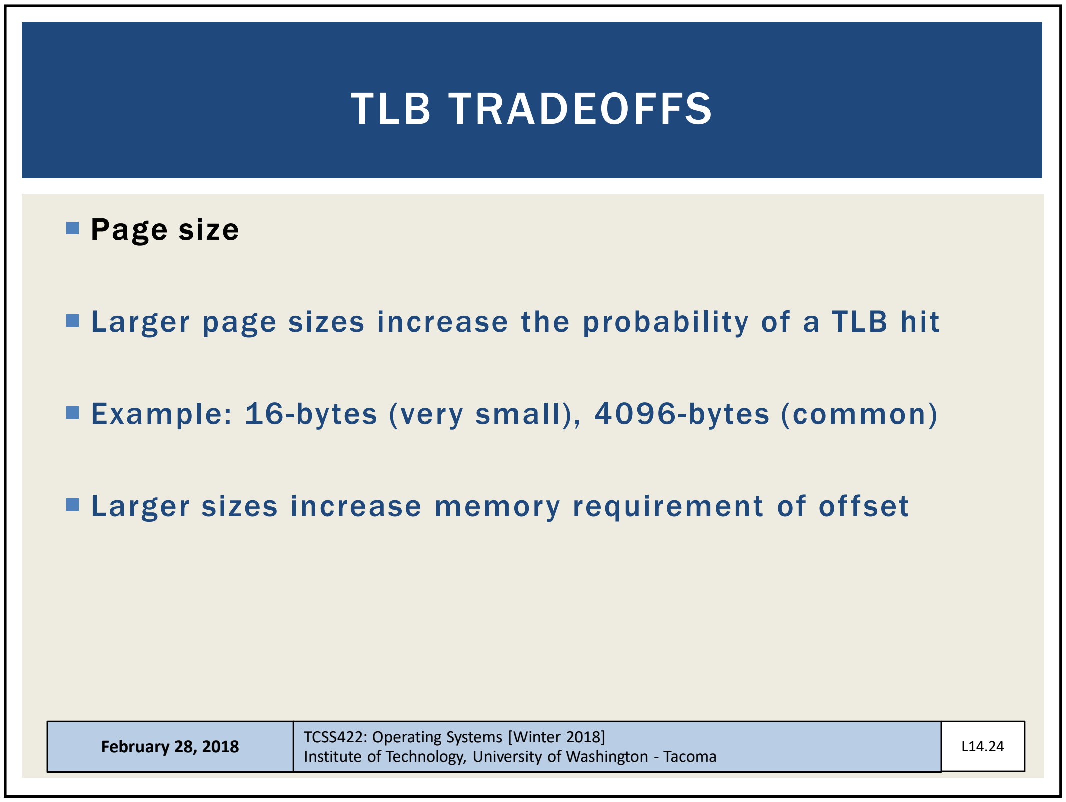
```
int sum = 0 ;
for( i=0; i<10; i++){
        sum+=a[i];
}
```

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				

- What factors affect the hit/miss rate?
  - Page size
  - Data locality
  - Temporal locality

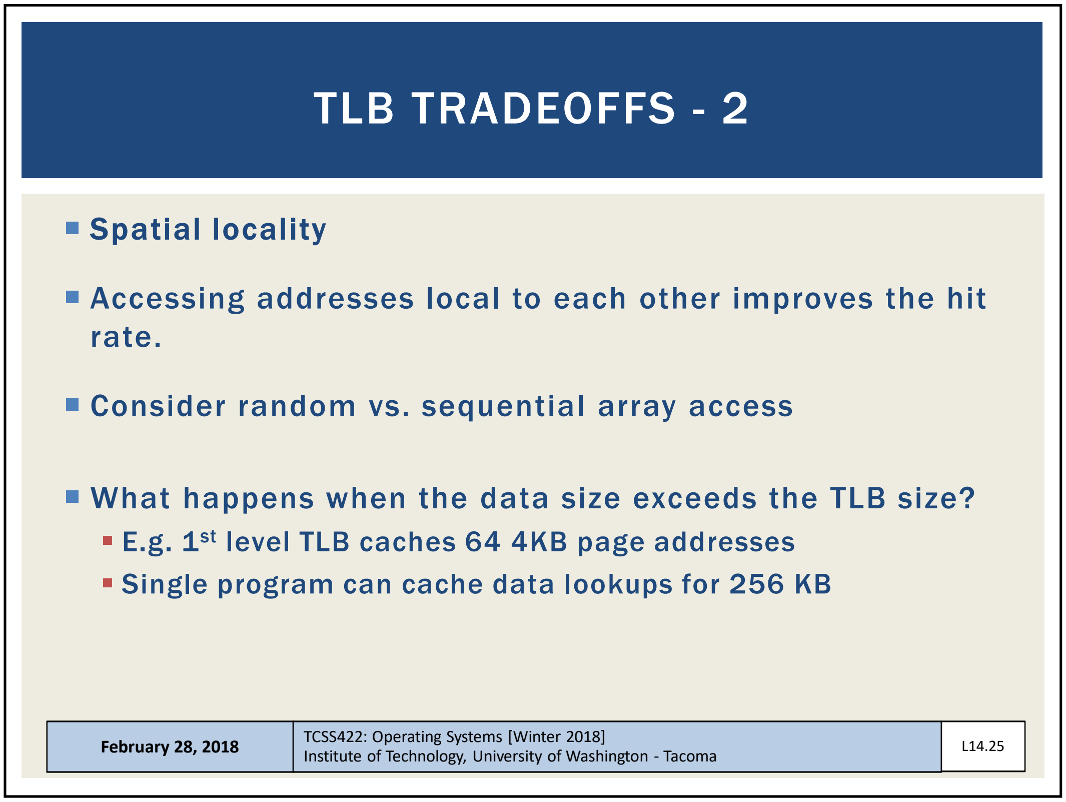
February 28, 2018 | TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma | L14.23



TLB TRADEOFFS

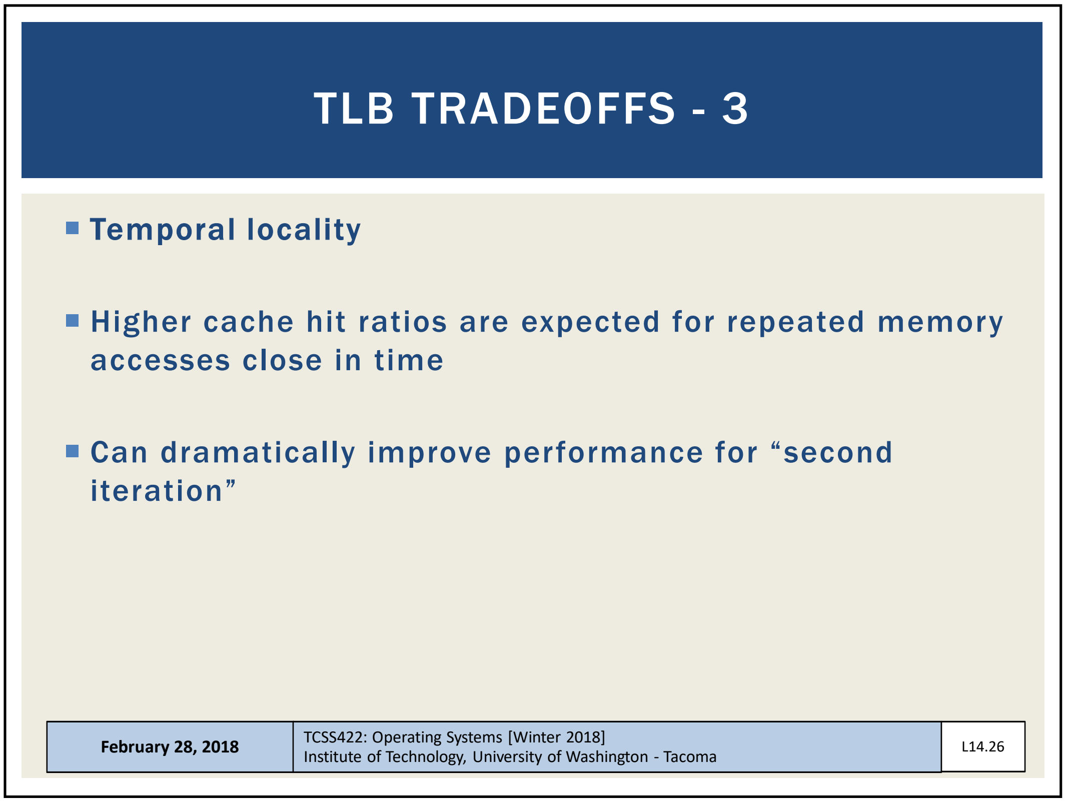
- **Page size**
- Larger page sizes increase the probability of a TLB hit
- Example: 16-bytes (very small), 4096-bytes (common)
- Larger sizes increase memory requirement of offset

February 28, 2018 | TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma | L14.24



TLB TRADEOFFS - 2

- Spatial locality
- Accessing addresses local to each other improves the hit rate.
- Consider random vs. sequential array access
- What happens when the data size exceeds the TLB size?
  - E.g. 1st level TLB caches 64 4KB page addresses
  - Single program can cache data lookups for 256 KB



TLB TRADEOFFS - 3

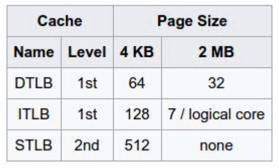
- Temporal locality
- Higher cache hit ratios are expected for repeated memory accesses close in time
- Can dramatically improve performance for “second iteration”

EXAMPLE: LARGE ARRAY ACCESS

- Example: Consider an array of a custom struct where each struct is 64-bytes. Consider sequential access for an array of 8,192 elements stored contiguously in memory:
- 64 structs per 4KB page
- 128 total pages
- TLB caches stores a maximum of 64 - 4KB page lookups
- How many hits vs. misses for sequential array iteration?
  - 1 miss for every 64 array accesses, 63 hits
  - Complete traversal: 128 total misses, 8,064 hits (98.4% hit ratio)

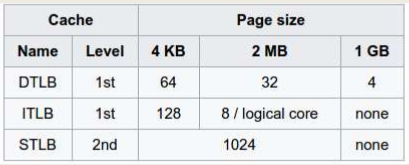
TLB EXAMPLE IMPLEMENTATIONS

- Intel Nehalem microarchitecture 2008 – multi level TLBs
  - First level TLB:
    separate cache for data (DTLB) and code (ITLB)
  - Second level TLB:
    shared TLB (STLB) for data and code
  - Multiple page sizes (4KB, 2MB)
  - Page Size Extension (PSE) CPU flag
    for larger page sizes



Cache		Page Size	
Name	Level	4 KB	2 MB
DTLB	1st	64	32
ITLB	1st	128	7 / logical core
STLB	2nd	512	none

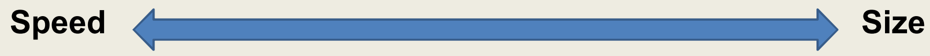
- Intel Haswell microarchitecture 22nm 2013
  - Two level TLB
  - Three page sizes (4KB, 2MB, 1GB)
- Without large page sizes consider the # of TLB entries to address 1.9 MB of memory…



Cache		Page size		
Name	Level	4 KB	2 MB	1 GB
DTLB	1st	64	32	4
ITLB	1st	128	8 / logical core	none
STLB	2nd	1024		none

HW CACHE TRADEOFF

- Speed vs. size



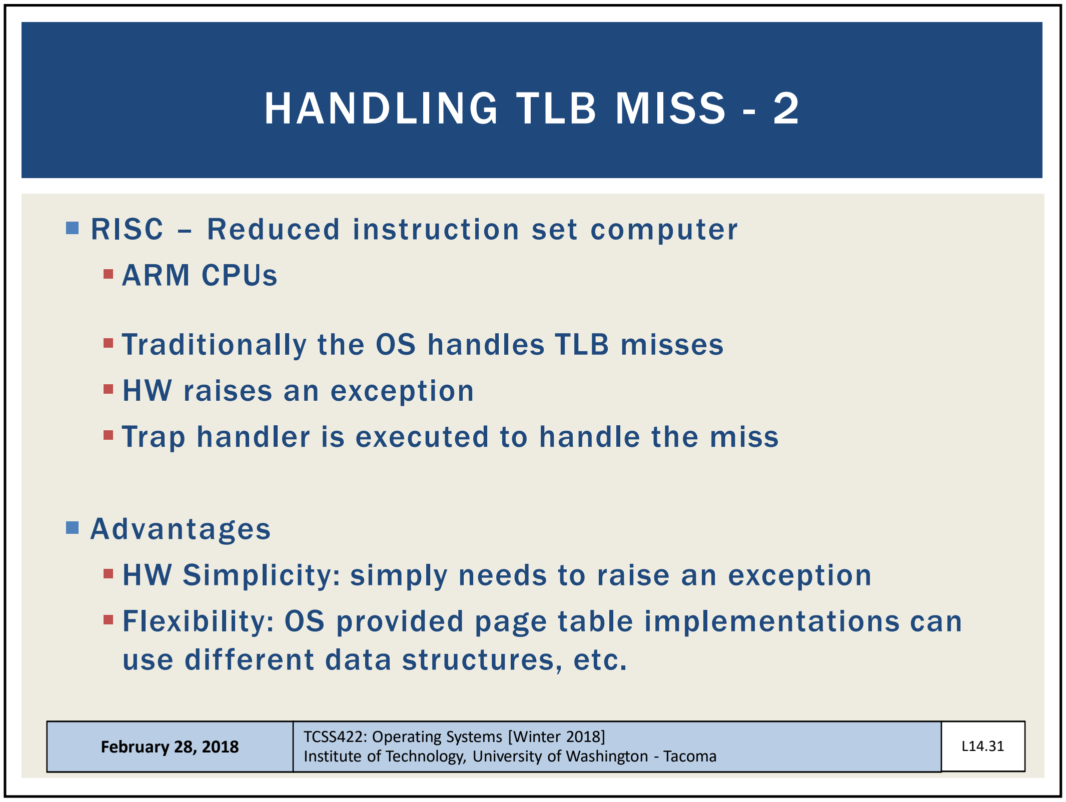
- In order to be fast, caches must be small
- Too large of a cache will mimic physical memory
- Limitations for on chip memory

"YOU CAN'T HAVE YOUR CAKE AND EAT IT TOO"
FALSE. YOU CAN HAVE A CAKE AND EAT IT. YOU JUST WON'T HAVE IT ANYMORE.

Dwight on "tradeoffs"

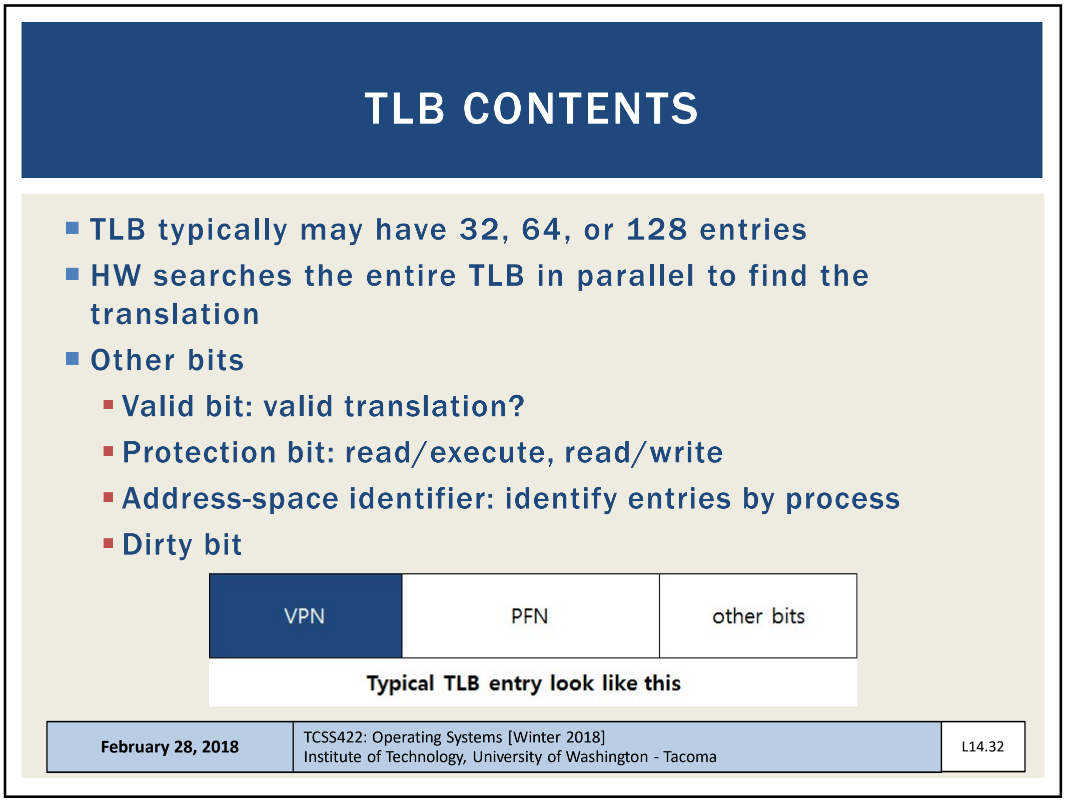
HANDLING TLB MISS

- Historical view
- CISC – Complex instruction set computer
  - Intel x86 CPUs
  - Traditionally have provided on CPU HW instructions and handling of TLB misses
  - HW has a page table register to store location of page table



HANDLING TLB MISS - 2

- RISC – Reduced instruction set computer
  - ARM CPUs
  - Traditionally the OS handles TLB misses
  - HW raises an exception
  - Trap handler is executed to handle the miss
- Advantages
  - HW Simplicity: simply needs to raise an exception
  - Flexibility: OS provided page table implementations can use different data structures, etc.



TLB CONTENTS

- TLB typically may have 32, 64, or 128 entries
- HW searches the entire TLB in parallel to find the translation
- Other bits
  - Valid bit: valid translation?
  - Protection bit: read/execute, read/write
  - Address-space identifier: identify entries by process
  - Dirty bit

VPN	PFN	other bits

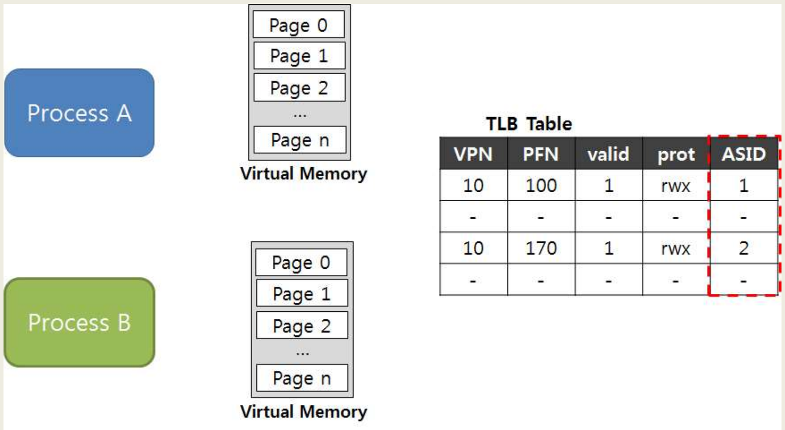
Typical TLB entry look like this

TLB: ON CONTEXT SWITCH

- TLB stores address translations for current running process
- A context/switch to a new process invalidates the TLB
- Must "switch" out the TLB
- <u>TLB flush</u>
  - Flush TLB on context switches, set all entries to 0
  - Requires time to flush
  - TLB must be reloaded for each C/S
  - If process not in CPU for long, the TLB may not get reloaded
- <u>Alternative</u>: be lazy…
  - Don't flush TLB on C/S
  - Share TLB across processes during C/S
  - Use address space identifier (ASID) to tag TLB entries by process

TLB: CONTEXT SWITCH - 2

- Address space identifier (ASID): enables TLB data to persist during context switches – also can support virtual machines



TLB Table

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

SHARED MEMORY SPACE

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

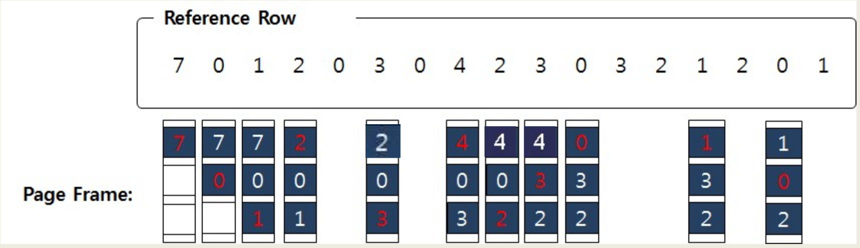
- When processes share a code page
  - Shared libraries ok
  - Code pages typically are RX, not RWX

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

CACHE REPLACEMENT POLICIES

- When TLB cache is full, how add a new address translation to the TLB?
- Observe how the TLB is loaded / unloaded…
- Goal minimize miss rate, increase hit rate
- **Least Recently Used (LRU)**
  - Evict the oldest entry
- **Random policy**
  - Pick a candidate at random to free-up space in the TLB

LEAST RECENTLY USED



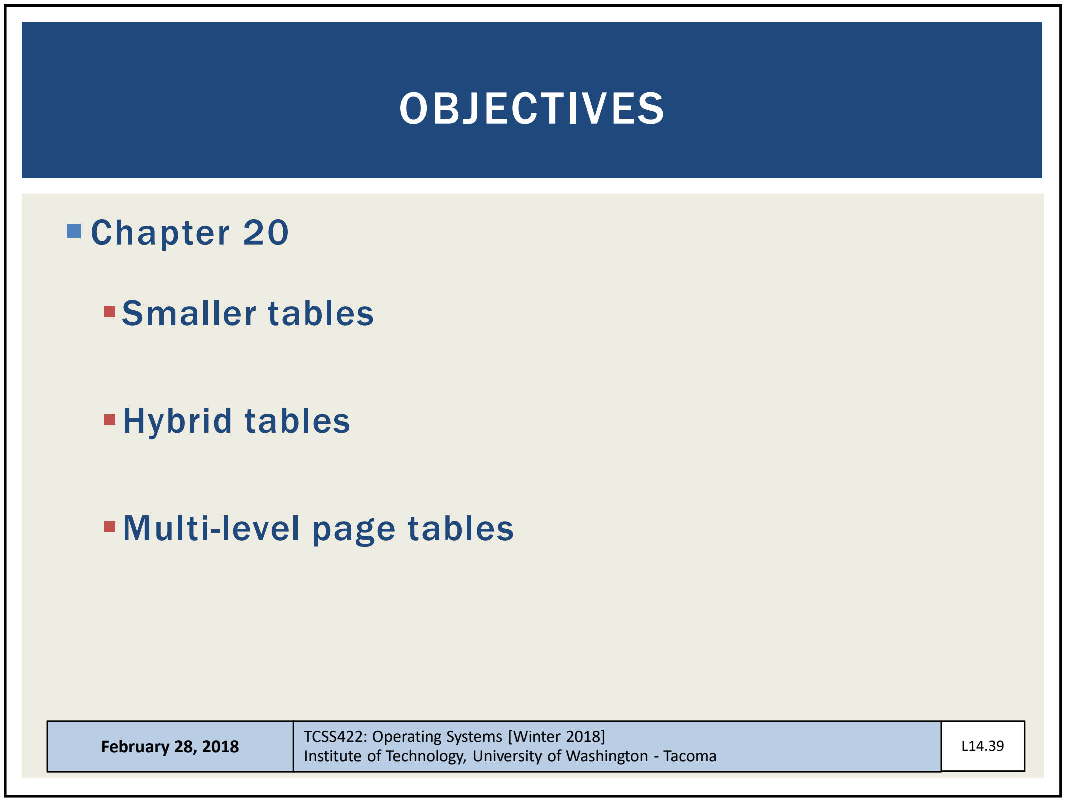
- RED – miss
- WHITE – hit
- For 3-page TLB, observe replacement

11 TLB miss, 5 TLB hit

February 28, 2018	TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma	L14.37

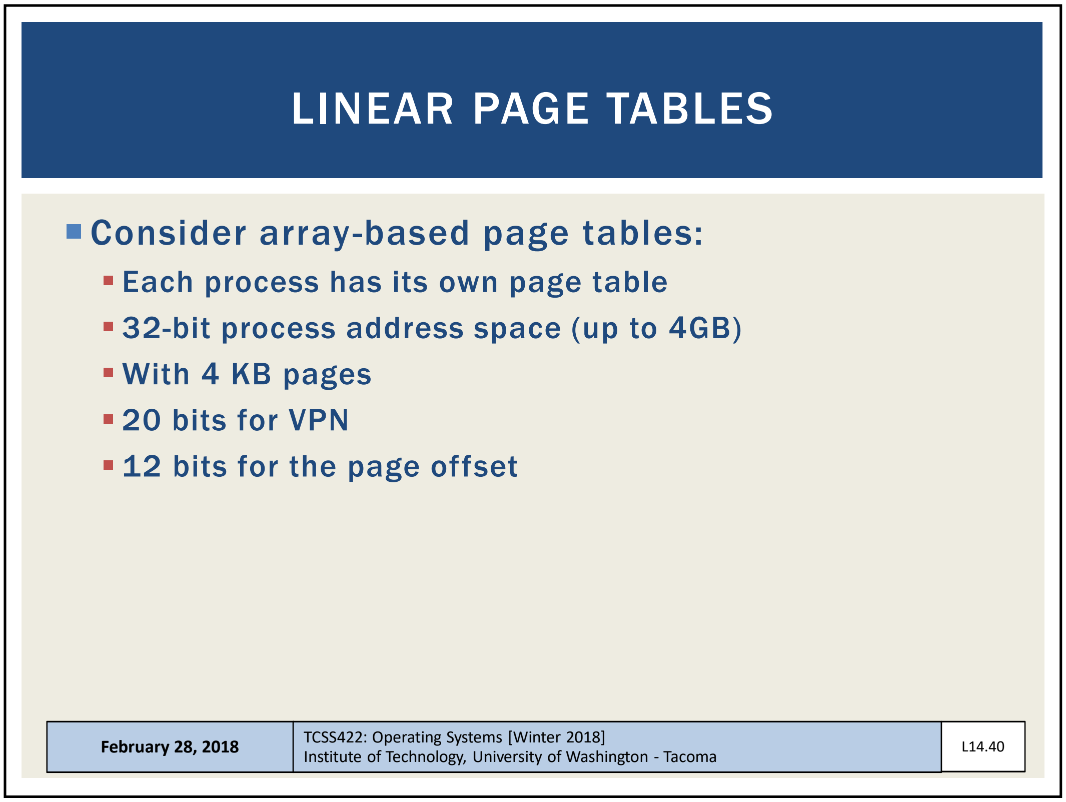
CHAPTER 20: PAGING: SMALLER TABLES

February 28, 2018 TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma L14.38



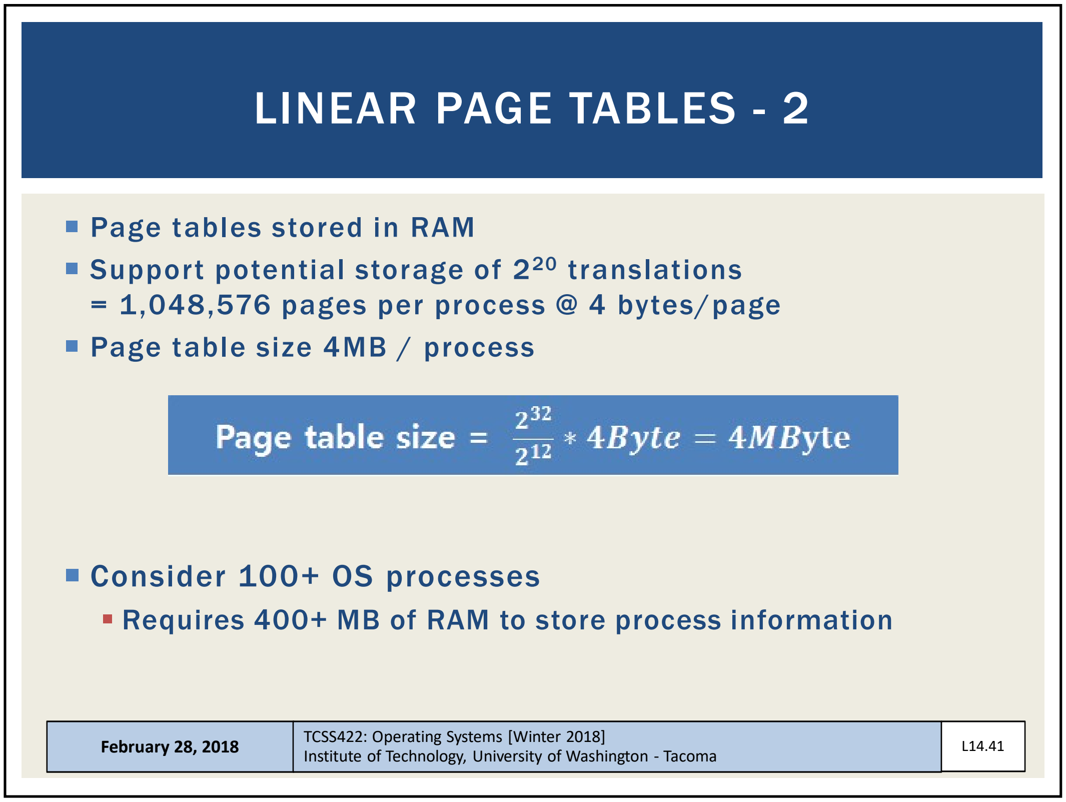
OBJECTIVES

- Chapter 20
  - Smaller tables
  - Hybrid tables
  - Multi-level page tables



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

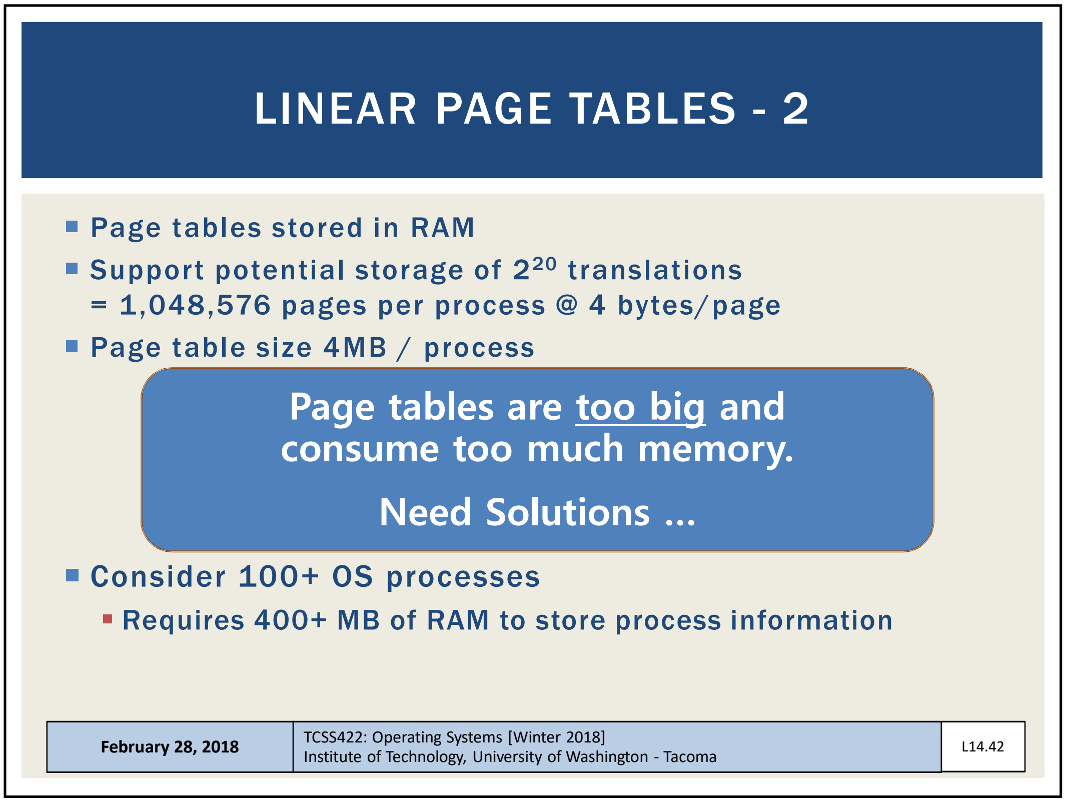


LINEAR PAGE TABLES - 2

- 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

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

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



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

PAGING: USE LARGER PAGES

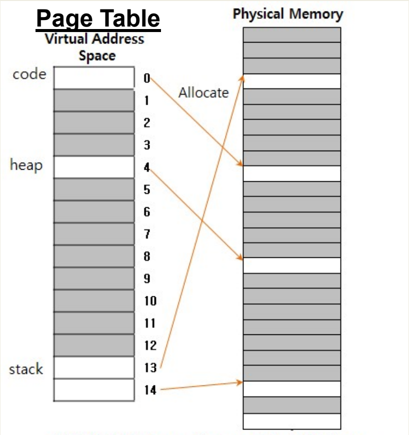
- Larger pages = 16KB = 2^{14}
- 32-bit address space: 2^{32}
- 2^{18} = 262,144 pages

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

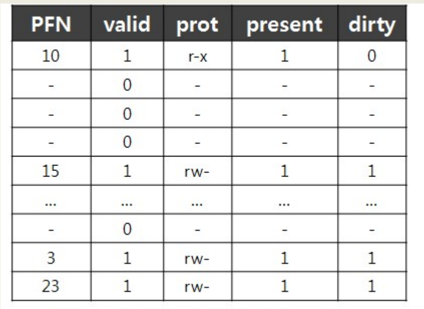
- Memory requirement cut to ¼
- However pages are huge
- Internal fragmentation results
- 16KB page(s) allocated for small programs with only a few variables

PAGE TABLES: WASTED SPACE

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



A 16KB Address Space with 1KB Pages

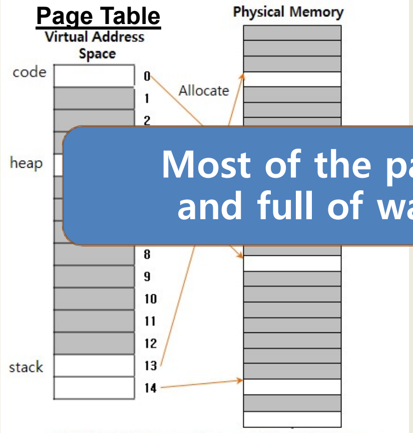


PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
…	…	…	…	…
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

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

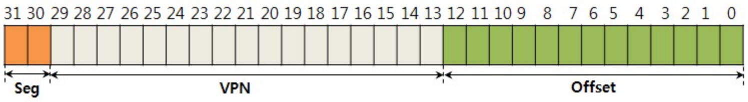
Most of the page table is unused and full of wasted space. (73%)

HYBRID TABLES

- Combine segments and page tables
- Use stack, heap, code segment base/bound registers
- Base register: point to page table
- Bounds register: store end of page table

HYBRID TABLES - 2

- Each process has (3) page tables
- 1 each for code, stack, heap segments
- Base register stores address of start of table
- 2^{16} bits for VPN, can only address 65,536 pages/segment



32-bit Virtual address space with 4KB pages

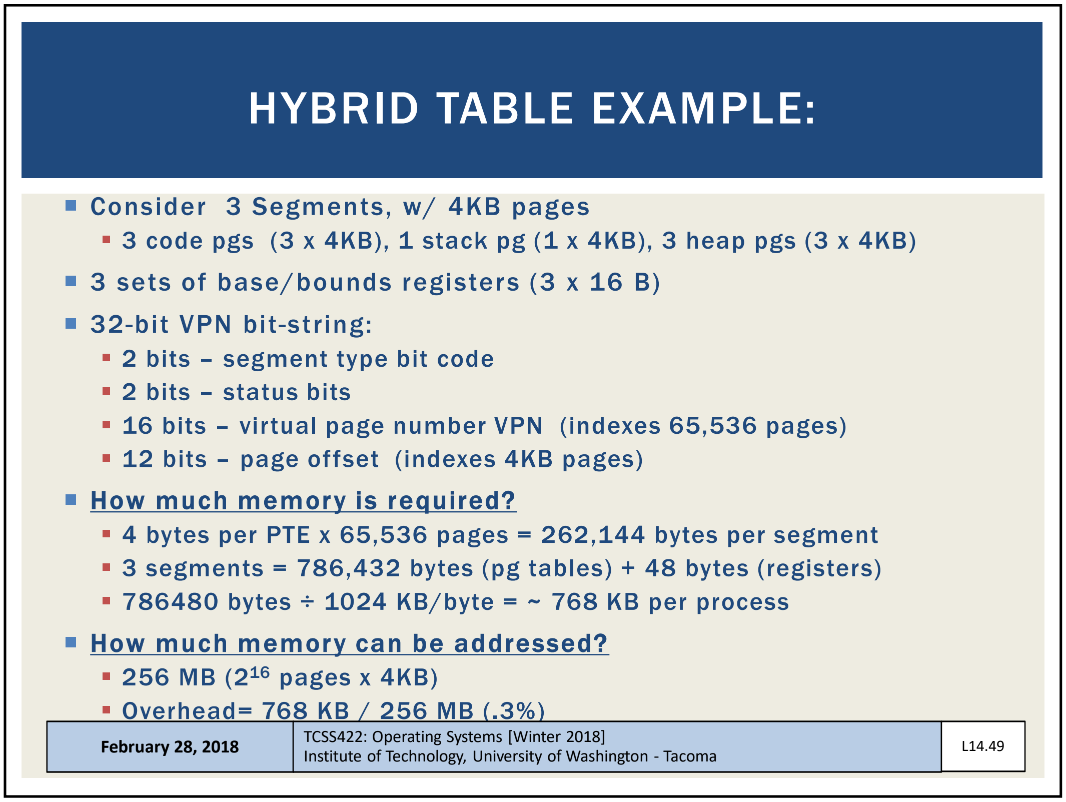
Seg value	Content
00	unused segment
01	code
10	heap
11	stack

HYBRID TABLES: COMPUTING PAGE TABLE ADDRESS

- HW must look up page table ADDR on TLB miss
- Segment (SN) bits: indicate which base/bound registers to use

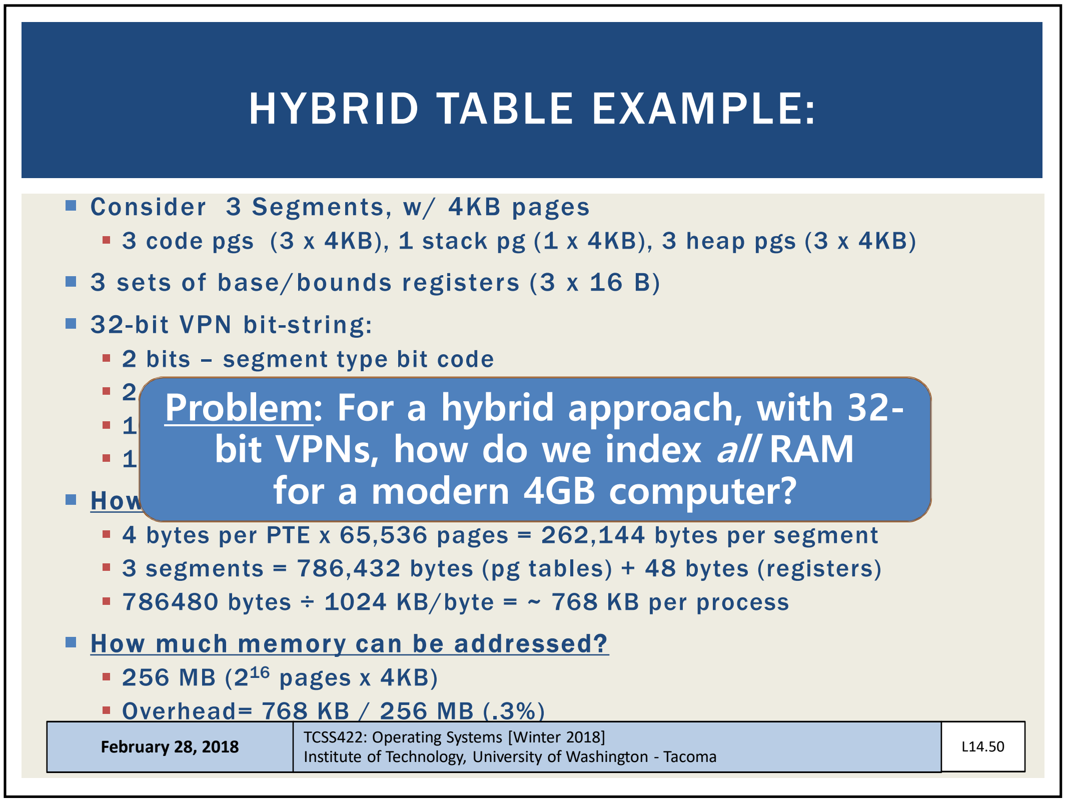
```
SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

- SEG_MASK = 1100 0000 0000 0000 0000 0000 0000 0000
- SN_SHIFT = 30 bits (shift 30 bits right)
  - *The SN will just be 2 bits…*
- VPN_MASK = 0011 1111 1111 1111 1111 0000 0000 0000
- VPN_SHIFT = 12 bits (shift 12 bits right)
  - *The VPN will just be 18 bits…*
- PTE ADDR = Base of table + VPN * size of a page table entry
  - *PTE=4 (or “10” in binary), will shift VPN 2 bits left* ←



HYBRID TABLE EXAMPLE:

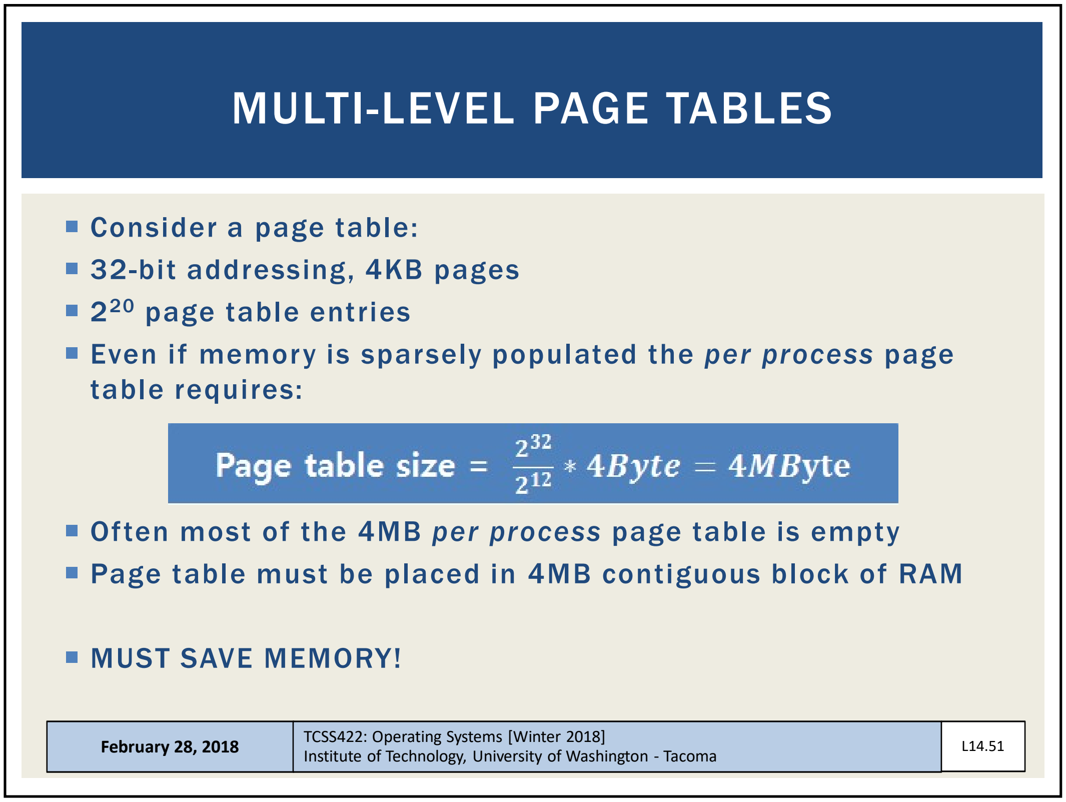
- Consider 3 Segments, w/ 4KB pages
  - 3 code pgs (3 x 4KB), 1 stack pg (1 x 4KB), 3 heap pgs (3 x 4KB)
- 3 sets of base/bounds registers (3 x 16 B)
- 32-bit VPN bit-string:
  - 2 bits – segment type bit code
  - 2 bits – status bits
  - 16 bits – virtual page number VPN (indexes 65,536 pages)
  - 12 bits – page offset (indexes 4KB pages)
- <u>How much memory is required?</u>
  - 4 bytes per PTE x 65,536 pages = 262,144 bytes per segment
  - 3 segments = 786,432 bytes (pg tables) + 48 bytes (registers)
  - 786480 bytes ÷ 1024 KB/byte = ~ 768 KB per process
- <u>How much memory can be addressed?</u>
  - 256 MB (2^{16} pages x 4KB)
  - Overhead= 768 KB / 256 MB (.3%)



HYBRID TABLE EXAMPLE:

- Consider 3 Segments, w/ 4KB pages
  - 3 code pgs (3 x 4KB), 1 stack pg (1 x 4KB), 3 heap pgs (3 x 4KB)
- 3 sets of base/bounds registers (3 x 16 B)
- 32-bit VPN bit-string:
  - 2 bits – segment type bit code
  - 2
  - 1
  - 1
- How
  - 4 bytes per PTE x 65,536 pages = 262,144 bytes per segment
  - 3 segments = 786,432 bytes (pg tables) + 48 bytes (registers)
  - 786480 bytes ÷ 1024 KB/byte = ~ 768 KB per process
- <u>How much memory can be addressed?</u>
  - 256 MB (2^{16} pages x 4KB)
  - Overhead= 768 KB / 256 MB (.3%)

<u>Problem</u>: For a hybrid approach, with 32-bit VPNs, how do we index *all* RAM for a modern 4GB computer?



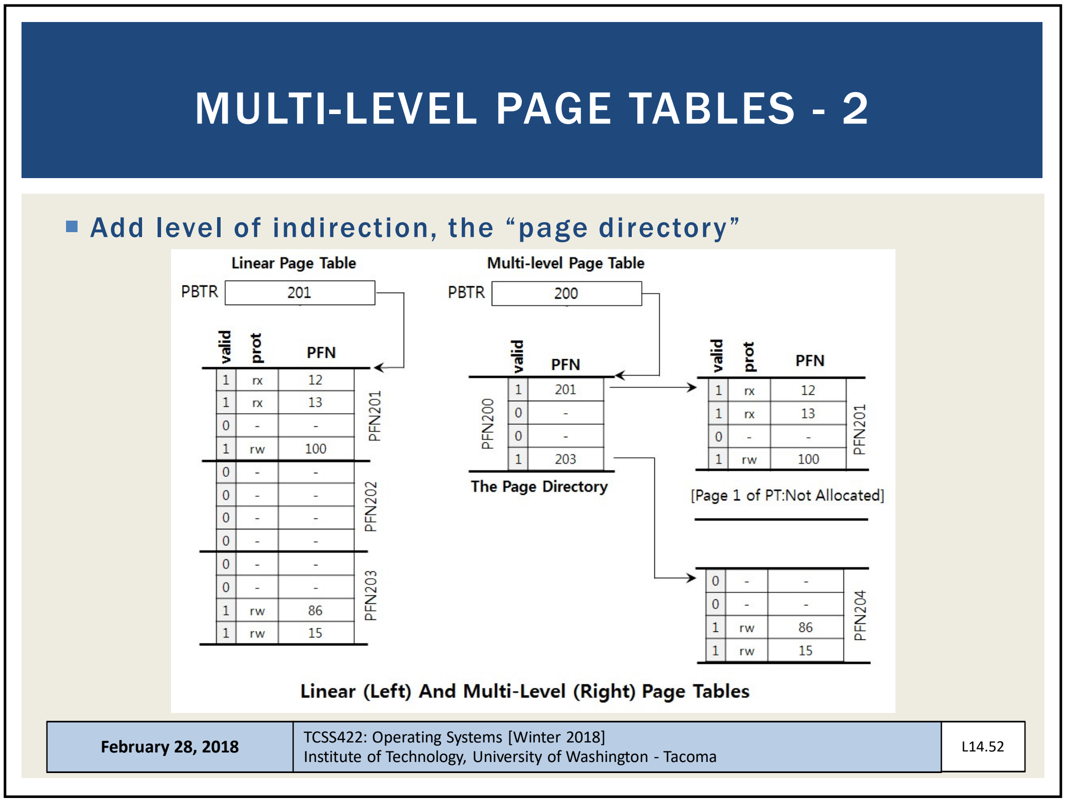
MULTI-LEVEL PAGE TABLES

- 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!

February 28, 2018
TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma
L14.51



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	
1	201	PFN200
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]

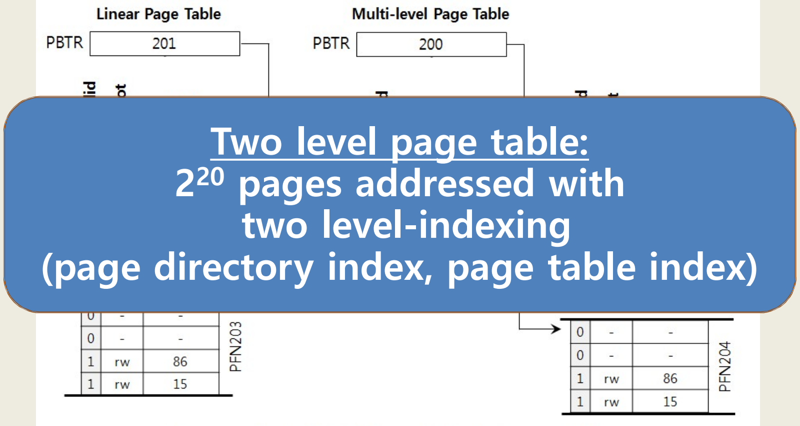
valid	prot	PFN	
0	-	-	PFN204
0	-	-	
1	rw	86	
1	rw	15	

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

February 28, 2018
TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma
L14.52

MULTI-LEVEL PAGE TABLES - 2

- Add level of indirection, the "page directory"



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

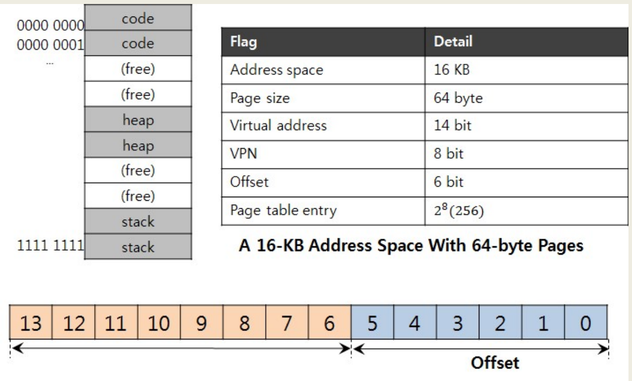
Linear (Left) And Multi-Level (Right) Page Tables

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

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)



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)

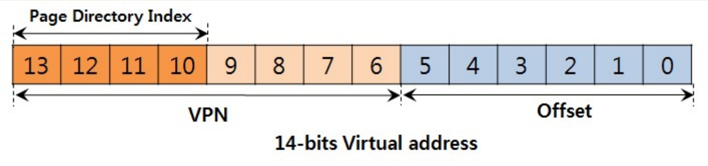
A 16-KB Address Space With 64-byte Pages

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!**

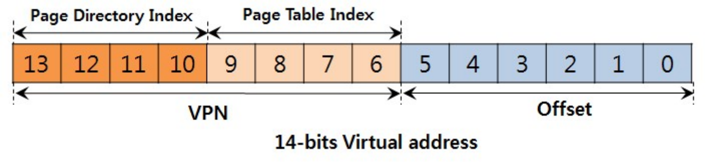
PAGE DIRECTORY INDEX

- 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

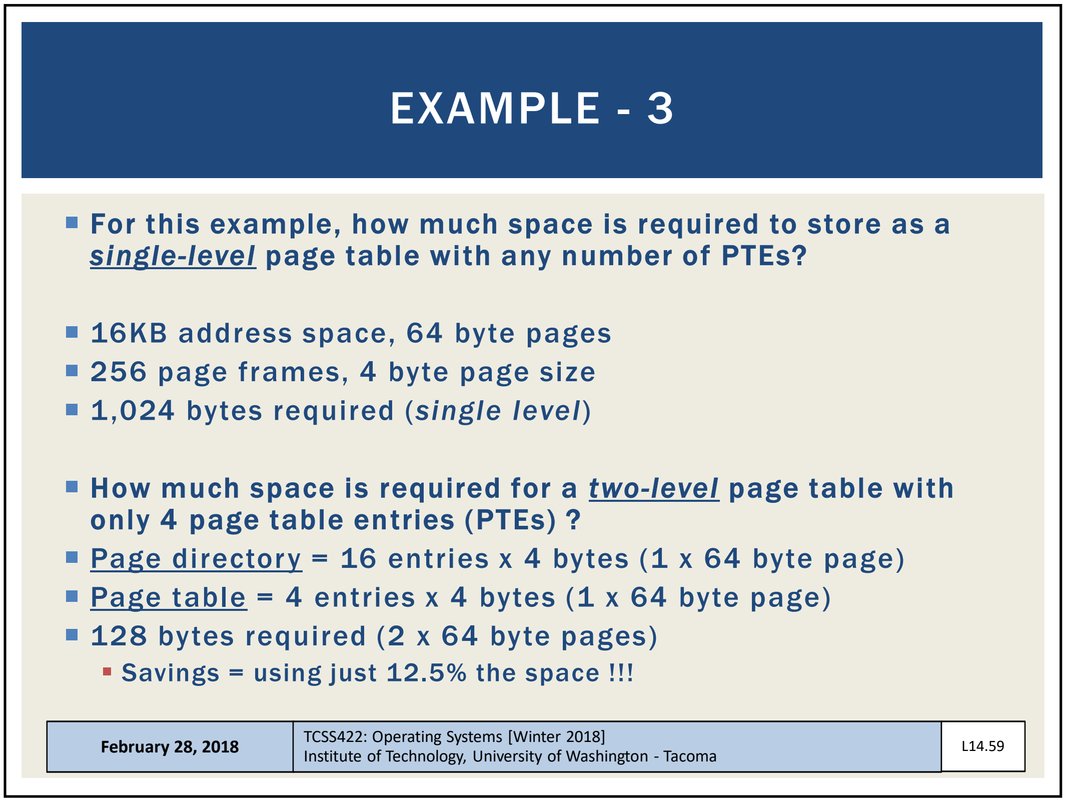


PAGE TABLE INDEX

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



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

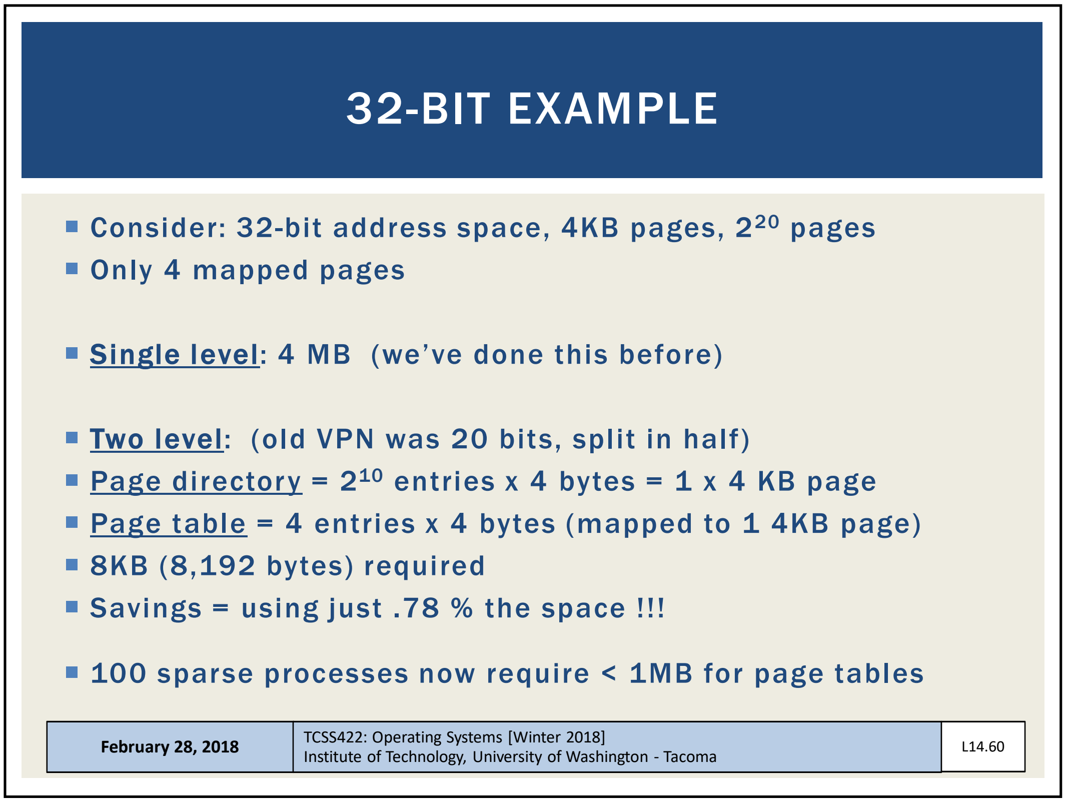


EXAMPLE - 3

- 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 !!!



32-BIT EXAMPLE

- Consider: 32-bit address space, 4KB pages, 2^{20} pages
- Only 4 mapped pages

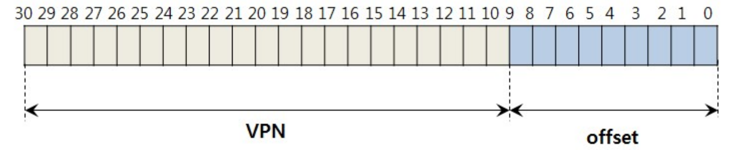
- Single level: 4 MB (we've done this before)

- 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

MORE THAN TWO LEVELS

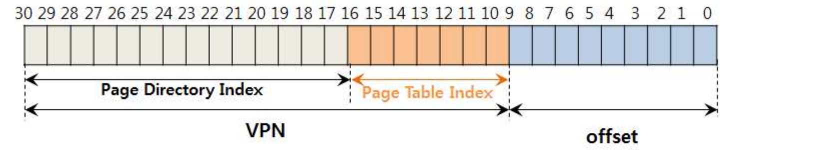
- Consider: page size is 2^9 = 512 bytes
- Page size 512 bytes / Page entry size 4 bytes
- VPN is 21 bits



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

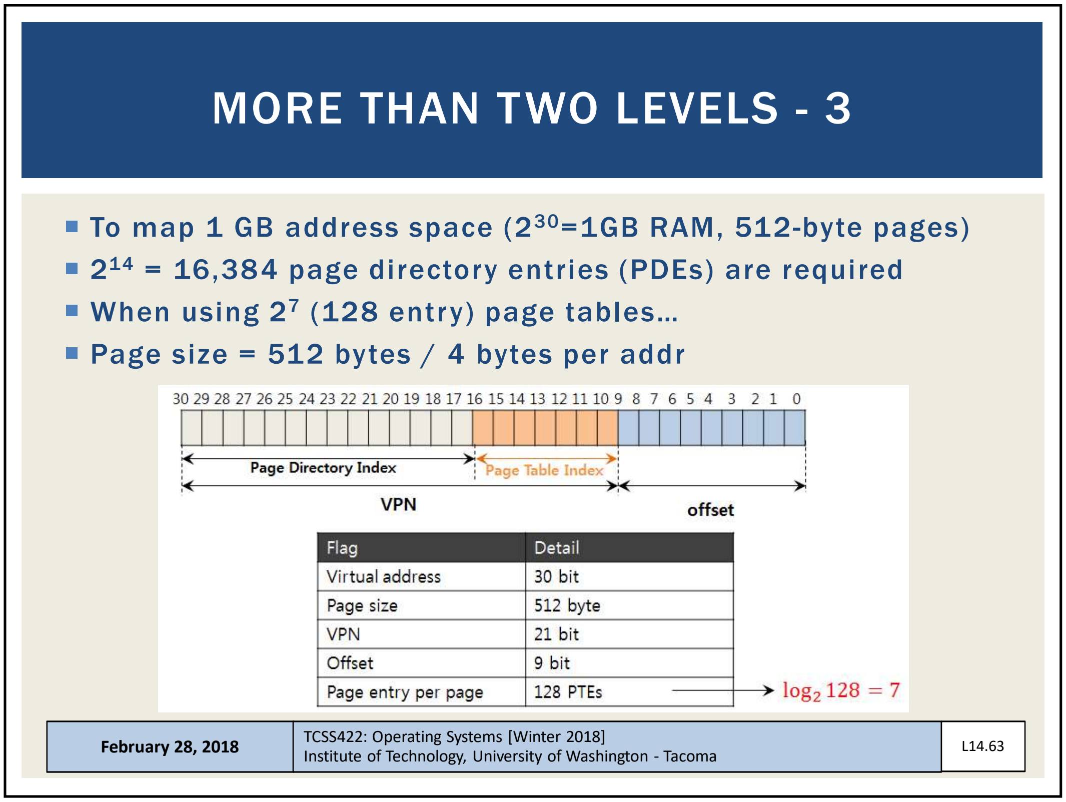
MORE THAN TWO LEVELS - 2

- 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

$\rightarrow \log_2 128 = 7$

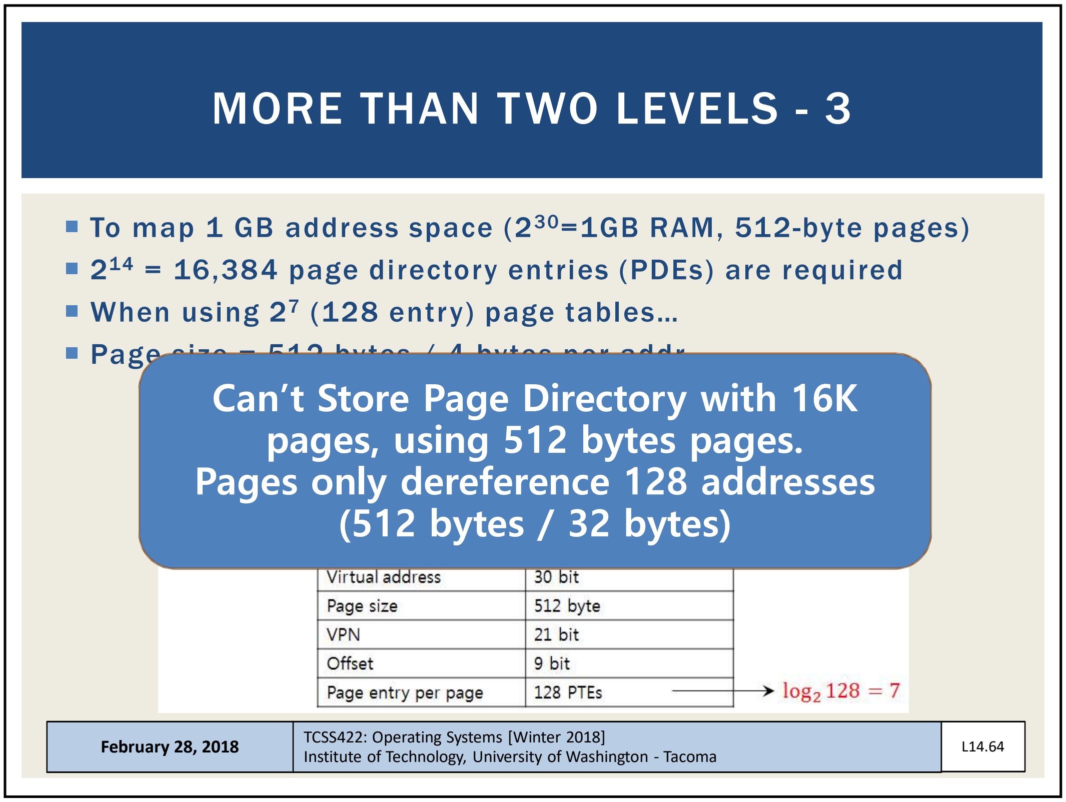


MORE THAN TWO LEVELS - 3

- 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

$\rightarrow \log_2 128 = 7$



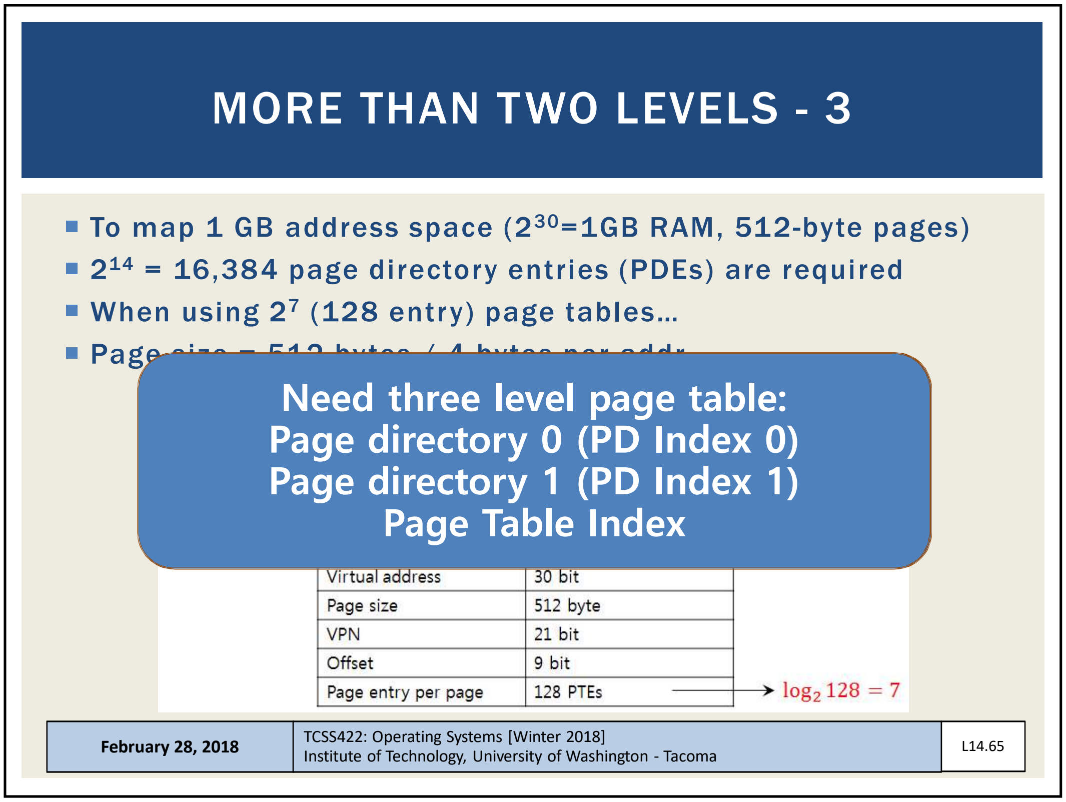
MORE THAN TWO LEVELS - 3

- 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

$\rightarrow \log_2 128 = 7$



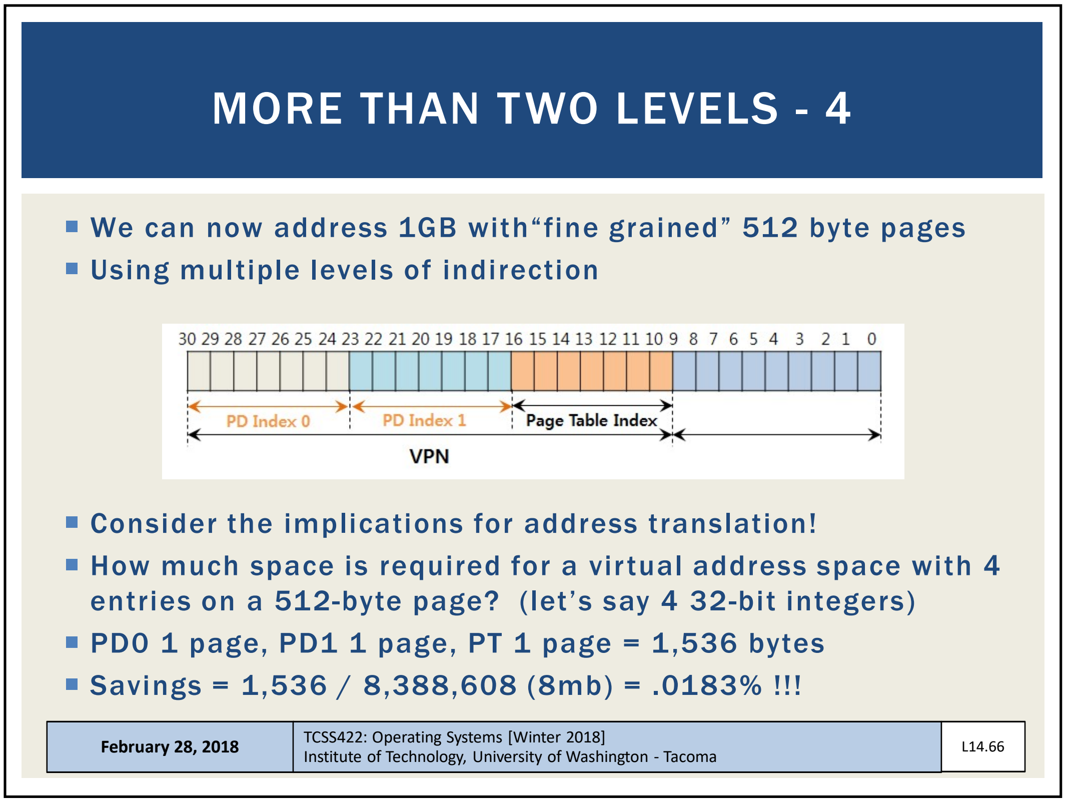
MORE THAN TWO LEVELS - 3

- 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

→ $\log_2 128 = 7$



MORE THAN TWO LEVELS - 4

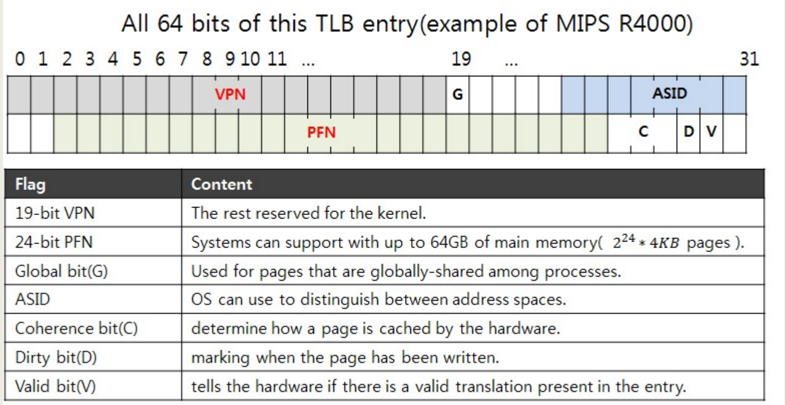
- We can now address 1GB with“fine grained” 512 byte pages
- Using multiple levels of indirection

- 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
- Savings = 1,536 / 8,388,608 (8mb) = .0183% !!!

QUESTIONS

EXAMPLE TLB ENTRY – MIPS R4000

- **Early 64-bit RISC processor**



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.