


TCSS 422: OPERATING SYSTEMS



Intro to Paging, Translation Lookaside Buffer, Smaller Page Tables

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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

OBJECTIVES

- **Mon 3/11 (4pm):** Husky Alumni Visit from T-Mobile Q&A
CS work life after graduation-room 206C
Garrett Lahmann ('18), Vlad Kaganyuk ('17)
- **Wed 3/13:** Prof. Mohamed Ali- UWT CSS Grad Program
- **Active Reading Quiz Posted – Chapter 19**
- **Assignment 3**
- **Memory Virtualization**
- **Chapter 18 – Introduction to Paging**
- **Chapter 19 – Translation Lookaside Buffer (TLB)**
- **Chapter 20 – Smaller Page Tables**

March 6, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L14.2
---------------	---	-------

FEEDBACK FROM 3/4

- What is stored in data headers (for malloc) besides the size?
- See Malloc.c source code – Line 1044:
- <https://code.woboq.org/userspace/glibc/malloc/malloc.c.html>
- Also:
<https://reverseengineering.stackexchange.com/questions/15033/how-does-glibc-malloc-work>

```
1044 struct malloc_chunk {
1045
1046     INTERNAL_SIZE_T      mchunk_prev_size; /* Size of previous chunk (if free). */
1047     INTERNAL_SIZE_T      mchunk_size;      /* Size in bytes, including overhead. */
1048
1049     struct malloc_chunk* fd;                /* double links -- used only if free. */
1050     struct malloc_chunk* bk;
1051
1052     /* Only used for large blocks: pointer to next larger size. */
1053     struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
1054     struct malloc_chunk* bk_nextsize;
1055 };
```

FEEDBACK - 2

- Can internal fragments ever be recovered?
- No, not without changing how data chunks are provisioned from memory to the programmer (OS change)
- Internal fragmentation:
no tracking (data) of unused portion of a chunk
- OS provides programmer with chunks of memory that are too big
- Programmer receives memory chunk that is larger than the original request

FEEDBACK - 3

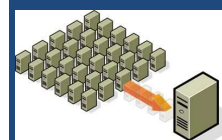
- Could you post solutions to the class activity
 - Happy to share answers after class, etc.
- How many notes for the final? Will it cover all material?
 - Final is comprehensive, 2 pages of notes, double-sided
- Do you have room for students interested in cloud computing for TCSS 499 Independent Study and TCSS 498 Directed Readings?
 - Yes, here's some quick background


March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.5



CLOUD AND DISTRIBUTED SYSTEMS RESEARCH








CLOUD AND DISTRIBUTED SYSTEMS LAB


WES LLOYD, WLLOYD@UW.EDU,
[HTTP://FACULTY.WASHINGTON.EDU/WLLOYD](http://FACULTY.WASHINGTON.EDU/WLLOYD)



- **Serverless Computing (FaaS):**
 - *How should cloud native applications be composed from microservices to optimize performance and cost? Code structure directly influences hosting costs.*
 - Service composition, performance and cost optimization/modeling/analytics, Application migration, Mitigation of Platform limitations, Influencing infrastructure, Lambda@Edge
- **Containerization (Docker):**
 - *How should containers and container platforms be leveraged and managed to optimize performance, reduce costs, and maximize server utilization?*
 - Containers, container orchestration frameworks, resource allocation, checkpointing
- **Infrastructure-as-a-Service (IaaS) Cloud:**
 - *How should applications and workloads be deployed to optimize performance and cost? There are many “knobs”, configuration options to consider.*
 - Application/workload deployment, performance and cost optimization/modeling/analytics, infrastructure management, resource contention detection/mitigation, HW heterogeneity



CHAPTER 18: INTRODUCTION TO PAGING



March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.8

PAGING

- Split up address space of process into fixed sized pieces called **pages**
- Alternative to variable sized pieces (Segmentation) which suffers from significant fragmentation
- Physical memory is split up into an array of fixed-size slots called **page frames**.
- Each process has a **page table** which translates virtual addresses to physical addresses

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.9

ADVANTAGES OF PAGING

- Flexibility
 - Abstracts the process address space into pages
 - No need to track direction of HEAP / STACK growth
 - *Just add more pages...*
 - No need to store unused space
 - *As with segments...*
- Simplicity
 - Pages and page frames are the same size
 - Easy to allocate and keep a free list of pages

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.10

PAGING: EXAMPLE

Page Table:
VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

- Consider a 128 byte address space with 16-byte pages
- Consider a 64-byte program address space

0		(page 0 of the address space)
16		(page 1)
32		(page 2)
48		(page 3)
64		

A Simple 64-byte Address Space

0	reserved for OS	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		

64-Byte Address Space Placed In Physical Memory

March 6, 2019

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.11

PAGING: ADDRESS TRANSLATION

- PAGE: Has two address components
 - VPN: Virtual Page Number
 - Offset: Offset within a Page

VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

- Example:
Page Size: 16-bytes, Address Space: 64-bytes

VPN		offset			
0	1	0	1	0	1

Here there are just four pages...

March 6, 2019

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

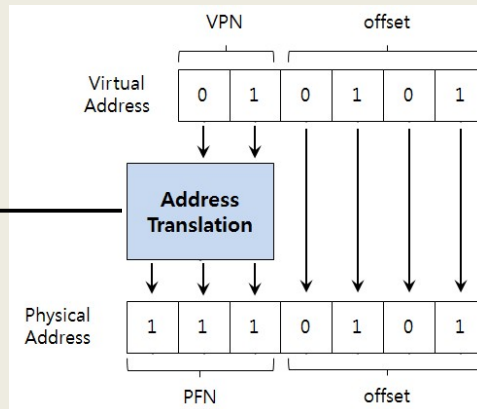
L14.12

EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte program address space (4 pages)
- Stored in 128-byte physical memory (8 frames)
- Offset is preserved
- VPN is looked up

Page Table:

VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2



March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.13

PAGING DESIGN QUESTIONS

- (1) Where are page tables stored?
- (2) What are the typical contents of the page table?
- (3) How big are page tables?
- (4) Does paging make the system too slow?

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.14

(1) 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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.15

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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.16

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?

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.17

(2) WHAT'S ACTUALLY IN THE PAGE TABLE

- Page table is data structure used to map virtual page numbers (VPN) to the physical address (Physical Frame Number PFN)
 - Linear page table → simple array
- Page-table entry
 - 32 bits for capturing state

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																						G		PAT	D	A	PCD	PWT	U/S	R/W	P

An x86 Page Table Entry(PTE)

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.18

PAGE TABLE ENTRY

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																						G	PAT	D	A	PCD	PWT	U/S	R/W	P	

An x86 Page Table Entry(PTE)

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.19

PAGE TABLE ENTRY - 2

- Common flags:
 - **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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.20

(3) HOW BIG ARE PAGE TABLES?

- Page tables are too big to store on the CPU
- Page tables are stored using physical memory
- Paging supports efficiently storing a sparsely populated address space
 - Reduced memory requirement
Compared to base and bounds, and segments

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.21

(4) DOES PAGING MAKE THE SYSTEM TOO SLOW?

- Translation
- **Issue #1:** Starting location of the page table is needed
 - HW Support: Page-table base register
 - stores active process
 - Facilitates translation
- **Issue #2:** Each memory address translation for paging requires an extra memory reference
 - HW Support: TLBs (Chapter 19)

Stored in RAM →

Page Table:

VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.22

PAGING MEMORY ACCESS

```
1. // Extract the VPN from the virtual address
2. VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3.
4. // Form the address of the page-table entry (PTE)
5. PTEAddr = PTBR + (VPN * sizeof(PTE))
6.
7. // Fetch the PTE
8. PTE = AccessMemory(PTEAddr)
9.
10. // Check if process can access the page
11. if (PTE.Valid == False)
12.     RaiseException(SEGMENTATION_FAULT)
13. else if (CanAccess(PTE.ProtectBits) == False)
14.     RaiseException(PROTECTION_FAULT)
15. else
16.     // Access is OK: form physical address and fetch it
17.     offset = VirtualAddress & OFFSET_MASK
18.     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19.     Register = AccessMemory(PhysAddr)
```

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.23

COUNTING MEMORY ACCESSSES

■ Example: Use this Array initialization Code

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

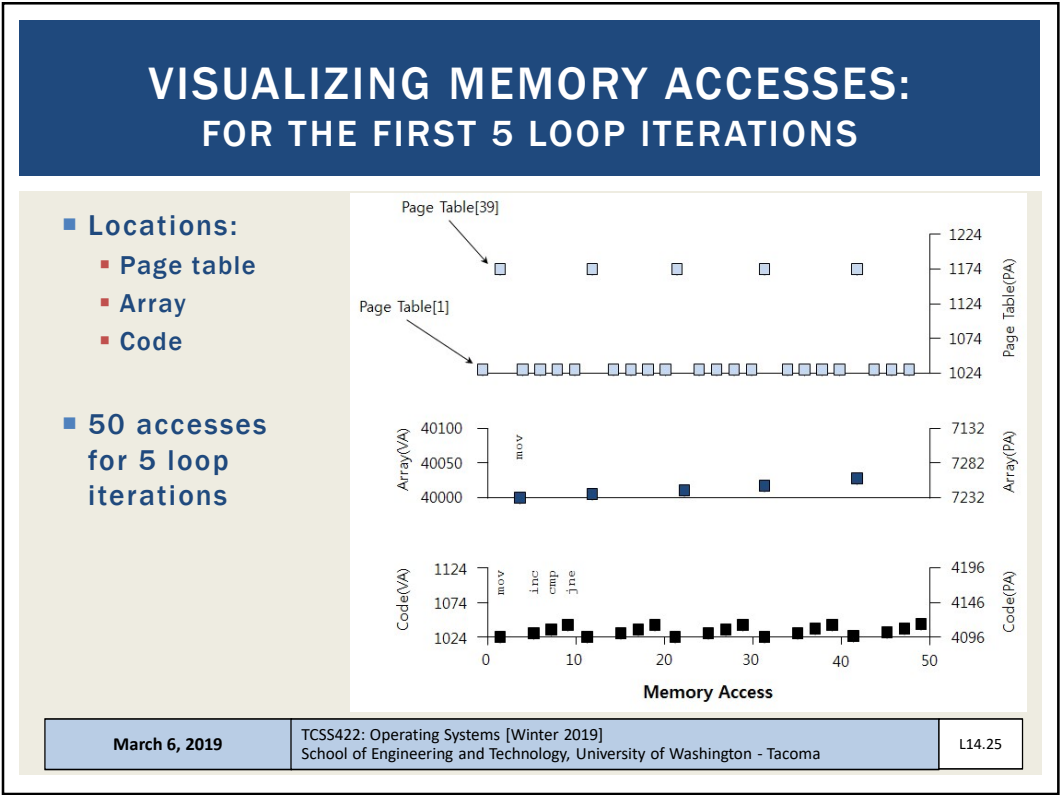
■ Assembly equivalent:


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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.24





CHAPTER 19: TRANSLATION LOOKASIDE BUFFER (TLB)

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.27

OBJECTIVES

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

March 6, 2019	TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L14.28
---------------	---	--------

TRANSLATION LOOKASIDE BUFFER

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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.29

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

The diagram illustrates memory access patterns over 50 memory accesses. It consists of three vertically stacked plots sharing a common x-axis labeled 'Memory Access' from 0 to 50.

- Page Table:** The top plot shows access to page table entries. The y-axis is labeled 'Page Table(PA)' with values 1024, 1074, 1124, 1174, and 1224. Arrows point to 'Page Table[1]' at access 0 and 'Page Table[39]' at access 40. The plot shows a sequence of accesses: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49.
- Array:** The middle plot shows access to array elements. The y-axis is labeled 'Array(VA)' with values 40000, 40050, and 40100. The plot shows a sequence of accesses: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49.
- Code:** The bottom plot shows access to code instructions. The y-axis is labeled 'Code(VA)' with values 1024, 1074, and 1124. The plot shows a sequence of accesses: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49.

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.30

TRANSLATION LOOKASIDE BUFFER (TLB)

■ Part of the CPU’s Memory Management Unit (MMU)

■ Address translation cache

The diagram illustrates the address translation process with an MMU. A CPU provides a Logical Address to the MMU's TLB. If there is a TLB Hit, the Physical Address is retrieved directly from the TLB. If there is a TLB Miss, the MMU consults the Page Table (which contains all v to p entries) to find the Physical Address. The Physical Memory is shown as a stack of pages (Page 0, Page 1, Page 2, ..., Page n).

Address Translation with MMU

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.31

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

The diagram illustrates the address translation process with an MMU. A CPU provides a Logical Address to the MMU's TLB. If there is a TLB Hit, the Physical Address is retrieved directly from the TLB. If there is a TLB Miss, the MMU consults the Page Table (which contains all v to p entries) to find the Physical Address. The Physical Memory is shown as a stack of pages (Page 0, Page 1, Page 2, ..., Page n).

Address Translation with MMU

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.32

TLB BASIC ALGORITHM

- For: array based page table
- Hardware managed TLB



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

Generate the physical address to access memory

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.33

TLB BASIC ALGORITHM - 2

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

Retry the instruction... (requery the TLB)

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.34

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

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.35

TLB EXAMPLE

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

- 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					

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.36

TLB EXAMPLE - 2

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

- 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					

March 6, 2019

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.37

TLB EXAMPLE - 3

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

- 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					

March 6, 2019

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.38

TLB EXAMPLE - 4

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

■ What factors affect the hit/miss rate?

■ Page size

■ Data locality

■ Temporal locality

	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					

March 6, 2019

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.39

CHAPTER 20:
PAGING:
SMALLER TABLES

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.40

OBJECTIVES

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.41

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.42

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

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

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.43

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.44

PAGING: USE LARGER PAGES

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

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

- Memory requirement cut to $\frac{1}{4}$
- However pages are huge
- Internal fragmentation results
- 16KB page(s) allocated for small programs with only a few variables

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.45

PAGE TABLES: WASTED SPACE

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

Page Table

Virtual Address Space

code

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

stack

Physical Memory

Allocate

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

A 16KB Address Space with 1KB Pages

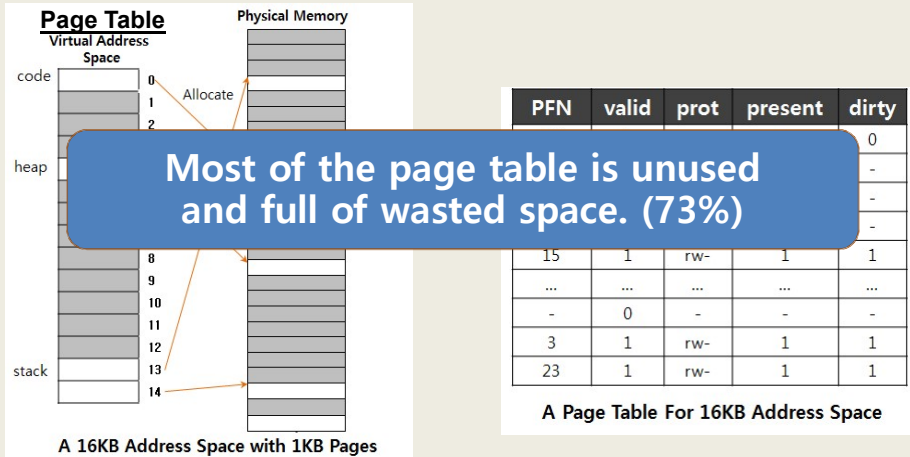
November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.46

PAGE TABLES: WASTED SPACE

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



November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.47

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:

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

- 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!**

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.48

MULTI-LEVEL PAGE TABLES - 2

■ Add level of indirection, the “page directory”

Linear Page Table

PBTR

201

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

PFN201
PFN202
PFN203

Multi-level Page Table

PBTR

200

valid	PFN
1	201
0	-
0	-
1	203

The Page Directory

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

PFN201

[Page 1 of PT:Not Allocated]

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

PFN204

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.49

MULTI-LEVEL PAGE TABLES - 2

■ Add level of indirection, the “page directory”

Linear Page Table

PBTR

201

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

PFN203

Multi-level Page Table

PBTR

200

valid	PFN
0	-
0	-
1	203

The Page Directory

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

PFN204

Two level page table:
2²⁰ pages addressed with
two level-indexing
(page directory index, page table index)

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.50

Slides by Wes J. Lloyd

L14.25

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.51

EXAMPLE

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

0000 0000

code

0000 0001

code

...

(free)

(free)

heap

heap

(free)

(free)

1111 1111

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.52

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!**

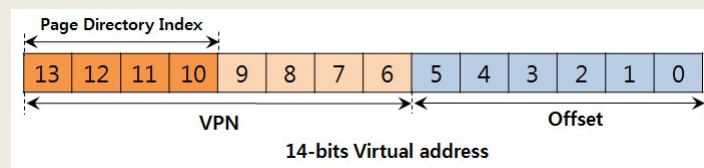
November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.53

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



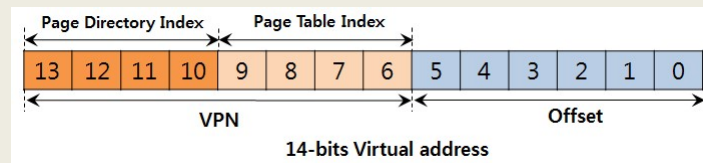
November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.54

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.55

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 !!!

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.56

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.57

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.58

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 → $\log_2 128 = 7$

November 26, 2018

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.59

MORE THAN TWO LEVELS - 3

- To map 1 GB address space ($2^{30}=1\text{GB}$ 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 → $\log_2 128 = 7$

November 26, 2018

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.60

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

→ $\log_2 128 = 7$

November 26, 2018

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.61

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

→ $\log_2 128 = 7$

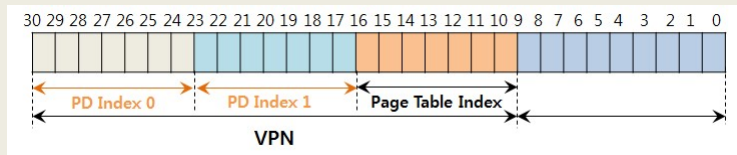
November 26, 2018

TCCS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.62

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 = $1,536 \text{ (3-level)} / 8,388,608 \text{ (1-level)} = .0183\% !!!$

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.63

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 *pmd;
pte_t *pte;
struct page *page;
```

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.64

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_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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.65

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.66

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?

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.67

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 – stack page
 - 1 – heap 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)?

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.68

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.69

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

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.70

ANSWERS

- #1 – 4096 pages
- #2 – 12 bits
- #3 – 12 bits
- #4 – 4 bytes
- #5 – $4096 \times 4 = 16,384$ bytes (16KB)
- #6 – 6 bits
- #7 – 6 bits
- #8 – 256 bytes for Page Directory (PD) (64 entries \times 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- $512/16384 = .03125 \rightarrow 3.125\%$

November 26, 2018

TCSS422: Operating Systems [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma

L14.71

QUESTIONS

