


TCSS 422: OPERATING SYSTEMS

Three Easy Pieces:
Free Space Management,
Introduction to Paging



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

May 14, 2018

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

OBJECTIVES

- Assignment 2 – Matrix Task Processor
- Assignment 3 – Posted Tuesday...
- Active reading Quiz #4 – Chapter 19
- “Group” Quiz #5 – Wednesday in class
- Memory Virtualization
- Free Space Management – Ch. 17
- Introduction to Paging – Ch. 18
- Translation Lookaside Buffer – Ch. 19

May 14, 2018

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

L12.2

FEEDBACK – 5/9

- Assignment 2 questions...
- “s MAT1 20 20 2”
- Does not print sum to console or a .sum file
- “d” command prints matrix to stdout
- “s” command creates only a sum file which is the sum of all matrix elements. In the process a matrix is created (but not saved anywhere)
- “x”
- Does not stop program, but should

May 14, 2018

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

L12.3

FEEDBACK - 2

- What is the math formula to where the stack is loaded in memory after the program is split into 3 segments in memory?

May 14, 2018

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

L12.4

FEEDBACK - 3

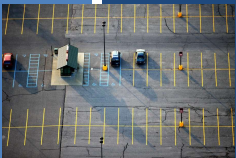
- Segment registers – first two bits identify segment type
- Stack bits are “10”
- Consider virtual address: 4200
- VIRTUAL ADDRESS = 1000001101000 (on stack)
- SEG_MASK=0x3000 (10000000000000) *LOGICAL AND THE MASK*
- **ZEROES OUT everything but the segment bits to learn the segment**
- SEG_SHIFT = 10 → stack (mask gives us segment code)
- OFFSET_MASK=0xFFFF (00111111111111) * LOGICAL AND THE MASK *
- **ZEROES OUT segment bits to reveal the offset address**
- OFFSET = 000001101000 = 104 (isolates segment offset)
- OFFSET < BOUNDS : 104 < 2048

May 14, 2018

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

L12.5

CHAPTER 17: FREE SPACE MANAGEMENT



May 14, 2018

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

L12.6

Slides by Wes J. Lloyd

L12.1

FREE SPACE MANAGEMENT

- Management of memory using
 - Only fixed-sized units
 - Easy: keep a list
 - Memory request → return first free entry
 - Simple search
 - With variable sized units
 - More challenging
 - Results from variable sized malloc requests
 - Leads to fragmentation

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.7

FRAGMENTATION

- Consider a 30-byte heap

30-byte heap:

free	used	free	
0	10	20	30
- Request for 15-bytes

free list: head →

addr:0	len:10
--------	--------

 →

addr:20	len:10
---------	--------

 → NULL
- Free space: 20 bytes
- No available contiguous chunk → return NULL

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.8

FRAGMENTATION - 2

- External:** OS can compact
 - Example: Client asks for 100 bytes: malloc(100)
 - OS: No 100 byte contiguous chunk is available: returns NULL
 - Memory is externally fragmented - - Compaction can fix!
- Internal:** lost space – OS can't compact
 - OS returns memory units that are too large
 - Example: Client asks for 100 bytes: malloc(100)
 - OS: Returns 125 byte chunk
 - Fragmentation is *in* the allocated chunk
 - Memory is lost, and unaccounted for – can't compact

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.9

ALLOCATION STRATEGY: SPLITTING

- Request for 1 byte of memory: malloc(1)

30-byte heap:

free	used	free	
0	10	20	30
- OS locates a free chunk to satisfy request
- Splits chunk into two, returns first chunk

30-byte heap:

free	used	free	
0	10	21	30

free list: head →

addr:0	len:10
--------	--------

 →

addr:21	len:9
---------	-------

 → NULL

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.10

ALLOCATION STRATEGY: COALESCING

- Consider 30-byte heap
- Free() frees all 10 bytes segments (list of 3-free 10-byte chunks)

head →

addr:0	len:10
--------	--------

 →

addr:10	len:10
---------	--------

 →

addr:20	len:10
---------	--------

 → NULL
- Request arrives: malloc(30)
- SPLIT DOES NOT WORK** - no contiguous 30-byte chunk exists!
- Coalescing regroups chunks into contiguous chunk

head →

addr:0	len:30
--------	--------

 → NULL
- Allocation can now proceed
- Coalescing is defragmentation of the free space list

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.11

MEMORY HEADERS

- free(void *ptr): Does not require a size parameter
- How does the OS know how much memory to free?
- Header block
 - Small descriptive block of memory at start of chunk

ptr

	The header used by malloc library
	The 20 bytes returned to caller

An Allocated Region Plus Header

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.12

MEMORY HEADERS - 2

hptr →

size: 20

magic: 1234567

ptr →

The 20 bytes returned to caller

Specific Contents Of The Header

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

A Simple Header

- Contains size
- Pointers: for faster memory access
- Magic number: integrity checking

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.13

MEMORY HEADERS - 3

- Size of memory chunk is:
 - Header size + user malloc size
 - N bytes + sizeof(header)
- Easy to determine address of header

```
void free(void *ptr) {
    header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.14

THE FREE LIST

- Simple free list struct

```
typedef struct __node_t {
    int size;
    struct __node_t *next;
} node_t;
```

- Use mmap to create free list
- 4kb heap, 4 byte header, one contiguous free chunk

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.15

FREE LIST - 2

- Create and initialize free-list "heap"

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

- Heap layout:

head →

size: 4088

next: 0

...

(virtual address: 16KB)
header: size field
header: next field(NULL is 0)
the rest of the 4KB chunk

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.16

FREE LIST: MALLOC() CALL

- Consider a request for a 100 bytes: malloc(100)
- Header block requires 8 bytes
 - 4 bytes for size, 4 bytes for magic number
- Split the heap – header goes with each block

A 4KB Heap With One Free Chunk

A Heap : After One Allocation

ptr →

First block is used

the 100 bytes now allocated

the free 3980 byte chunk

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.17

FREE LIST: FREE() CALL

- Addresses of chunks
- Start=16384
 - + 108 (end of 1st chunk)
 - + 108 (end of 2nd chunk)
 - + 108 (end of 3rd chunk)
 - = 16708

8 bytes header

Free this block

Free Space With Three Chunks Allocated

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.18

FREE LIST:
FREE() CHUNK #2

- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
 - addr - sizeof(node_t)
- Actual start of chunk #2
 - 16492

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.19

FREE LIST- FREE ALL CHUNKS

- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.20

GROWING THE HEAP

- Start with small sized heap
- Request more memory when full
- sbrk(), brk()

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.21

MEMORY ALLOCATION STRATEGIES

- Best fit
 - Traverse free list
 - Identify all candidate free chunks
 - Note which is smallest (has best fit)
 - When splitting, "leftover" pieces are small (and potentially less useful -- fragmented)
- Worst fit
 - Traverse free list
 - Identify largest free chunk
 - Split largest free chunk, leaving a still large free chunk

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.22

EXAMPLES

- Allocation request for 15 bytes

- Result of Best Fit

- Result of Worst Fit

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.23

MEMORY ALLOCATION STRATEGIES - 2

- First fit
 - Start search at beginning of free list
 - Find first chunk large enough for request
 - Split chunk, returning a "fit" chunk, saving the remainder
 - Avoids full free list traversal of best and worst fit
- Next fit
 - Similar to first fit, but start search at last search location
 - Maintain a pointer that "cycles" through the list
 - Helps balance chunk distribution vs. first fit
 - Find first chunk, that is large enough for the request, and split
 - Avoids full free list traversal

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.24

SEGREGATED LISTS

- For popular sized requests
e.g. for kernel objects such as locks, inodes, etc.
- Manage as segregated free lists
- Provide object caches: stores pre-initialized objects
- How much memory should be dedicated for specialized requests (object caches)?
- If a given cache is low in memory, can request “slabs” of memory from the general allocator for caches.
- General allocator will reclaim slabs when not used

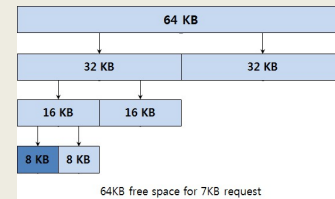
May 14, 2018

TCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.25

BUDDY ALLOCATION

- Binary buddy allocation
 - Divides free space by two to find a block that is big enough to accommodate the request; the next split is too small...
- Consider a 7KB request



May 14, 2018

TCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.26

BUDDY ALLOCATION - 2

- Buddy allocation: suffers from internal fragmentation
- Allocated fragments, typically too large
- Coalescing is simple
 - Two adjacent blocks are promoted up

May 14, 2018

TCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.27

CHAPTER 18: INTRODUCTION TO PAGING



May 14, 2018

TCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.28

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

May 14, 2018

TCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.29

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

May 14, 2018

TCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.30

PAGING: EXAMPLE

■ Consider a 128 byte address space with 16-byte pages

■ Consider a 64-byte program address space

0
16
32
48
64
80
96
112
128

(page 0 of the address space)
(page 1)
(page 2)
(page 3)

0
16
32
48
64
80
96
112
128

reserved for OS
(unused)
page 3 of AS
page 0 of AS
(unused)
page 2 of AS
(unused)
page 1 of AS

page frame 0 of physical memory
page frame 1
page frame 2
page frame 3
page frame 4
page frame 5
page frame 6
page frame 7

A Simple 64-byte Address Space

64-Byte Address Space Placed In Physical Memory

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.31

PAGING: ADDRESS TRANSLATION

■ PAGE: Has two address components

■ VPN: Virtual Page Number

■ Offset: Offset within a Page

VPNoffset

Va5Va4Va3Va2Va1Va0

■ Example:

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

VPNoffset

010101

Here there are just four pages...

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.32

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

Virtual Address

VPNoffset

010101

Address Translation

Physical Address

1110101

PFNoffset

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.33

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?

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.34

(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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.35

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

VPN₀

VPN₁

VPN₂

...

...

VPN₁₀₄₈₅₇₆

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.36

Slides by Wes J. Lloyd

L12.6

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 $4\text{MB} \times 100 = 400\text{MB}$
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

400 MB / 4000 GB
- Is this efficient?

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.37

(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		R/W		D		A		P		U/S		P		A															

An x86 Page Table Entry(PTE)

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.38

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		R/W		D		A		P		U/S		P		A													

An x86 Page Table Entry(PTE)

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.39

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.40

(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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.41

(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)

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.42

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

Stored in RAM →

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)
```

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.43

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
```

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.44

VISUALIZING MEMORY ACCESSES:
FOR THE FIRST 5 LOOP ITERATIONS

■ Locations:

■ Page table

■ Array

■ Code

■ 50 accesses for 5 loop iterations

Iteration	Access Address (PA)
1	1024
2	1024
3	1024
4	1024
5	1024
6	1024
7	1024
8	1024
9	1024
10	1024
11	1024
12	1024
13	1024
14	1024
15	1024
16	1024
17	1024
18	1024
19	1024
20	1024
21	1024
22	1024
23	1024
24	1024
25	1024
26	1024
27	1024
28	1024
29	1024
30	1024
31	1024
32	1024
33	1024
34	1024
35	1024
36	1024
37	1024
38	1024
39	1024
40	1024
41	1024
42	1024
43	1024
44	1024
45	1024
46	1024
47	1024
48	1024
49	1024
50	1024

Iteration	Access Address (PA)
1	40100
2	40100
3	40100
4	40100
5	40100
6	40100
7	40100
8	40100
9	40100
10	40100
11	40100
12	40100
13	40100
14	40100
15	40100
16	40100
17	40100
18	40100
19	40100
20	40100
21	40100
22	40100
23	40100
24	40100
25	40100
26	40100
27	40100
28	40100
29	40100
30	40100
31	40100
32	40100
33	40100
34	40100
35	40100
36	40100
37	40100
38	40100
39	40100
40	40100
41	40100
42	40100
43	40100
44	40100
45	40100
46	40100
47	40100
48	40100
49	40100
50	40100

Iteration	Access Address (PA)
1	1124
2	1124
3	1124
4	1124
5	1124
6	1124
7	1124
8	1124
9	1124
10	1124
11	1124
12	1124
13	1124
14	1124
15	1124
16	1124
17	1124
18	1124
19	1124
20	1124
21	1124
22	1124
23	1124
24	1124
25	1124
26	1124
27	1124
28	1124
29	1124
30	1124
31	1124
32	1124
33	1124
34	1124
35	1124
36	1124
37	1124
38	1124
39	1124
40	1124
41	1124
42	1124
43	1124
44	1124
45	1124
46	1124
47	1124
48	1124
49	1124
50	1124

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.45

PAGING SYSTEM EXAMPLE

■ Consider a 4GB Computer:

■ With a 4096-byte page size (4KB)

■ How many pages would fit in physical memory?

■ Now consider a page table:

■ For the page table entry, how many bits are required for the VPN?

■ If we assume the use of 4-byte (32 bit) page table entries, how many bits are available for status bits?

■ How much space does this page table require?
Page Table Entries x Number of pages

■ How many page tables (for user processes) would fill the entire 4GB of memory?

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.46

CHAPTER 19:
TRANSLATION
LOOKASIDE BUFFER
(TLB)

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.47

OBJECTIVES

■ Chapter 19

■ TLB Algorithm

■ TLB Tradeoffs

■ TLB Context Switch

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.48

TRANSLATION LOOKASIDE BUFFER

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.49

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.50

TRANSLATION LOOKASIDE BUFFER (TLB)

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.51

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.52

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.53

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)

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.54

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.55

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

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.56

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?

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.57

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)

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.58

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

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

May 14, 2018

TCCS422: Operating Systems [Spring 2018]
Institute of Technology, University of Washington - Tacoma

L12.59

QUESTIONS

