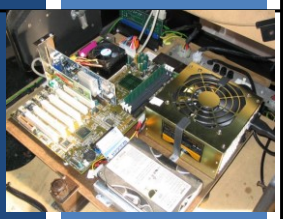
TCSS 422: OPERATING SYSTEMS



Memory Virtualization III: Free Space Management, Introduction to Paging, Translation Lookaside Buffer (TLB), Smaller Tables

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

1

TCSS 422 – OFFICE HRS – SPRING 2025

- **Office Hours plan for Spring (by Zoom):**
- **Monday 11:30am - 12:30p GTA Xinghan**
- **Tuesday 11:30am - 12:30p GTA Xinghan**
- **Wednesday 11:00am - 12:00p Instructor Wes**
- **Friday 12:00pm - 1:00p Instructor Wes or GTA Xinghan**
  - **THIS FRIDAY: Wes**
- Instructor is available immediately after class today via Zoom for questions

2

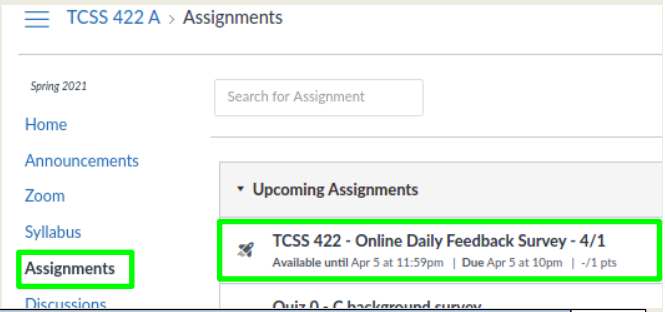
OBJECTIVES – 5/27

- **Questions from 5/22**
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables – (ch 18/20) posted
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

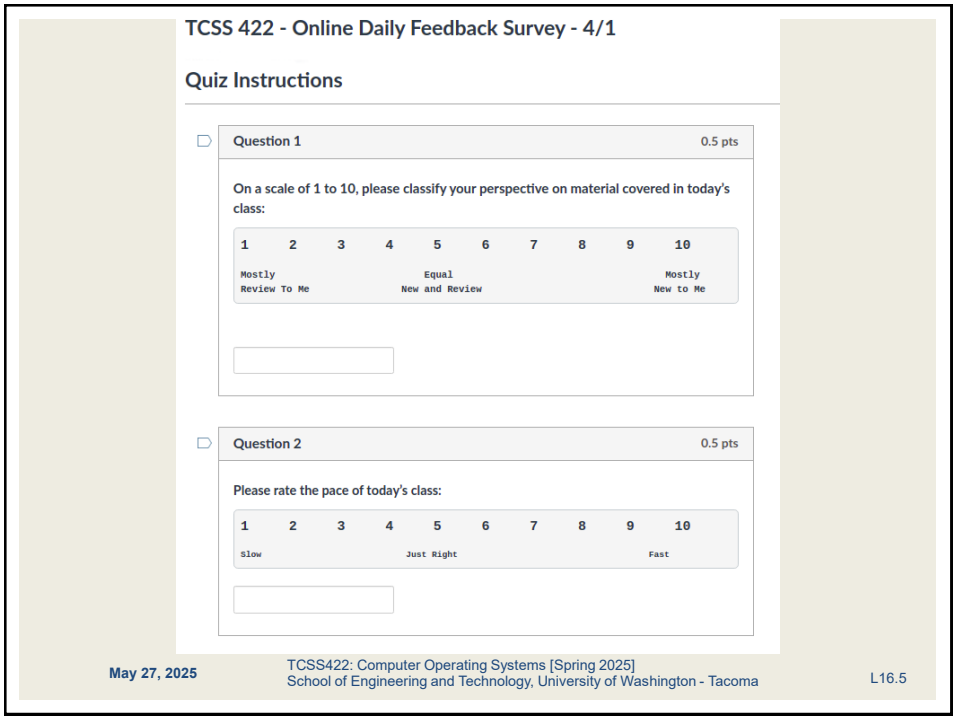
3

ONLINE DAILY FEEDBACK SURVEY

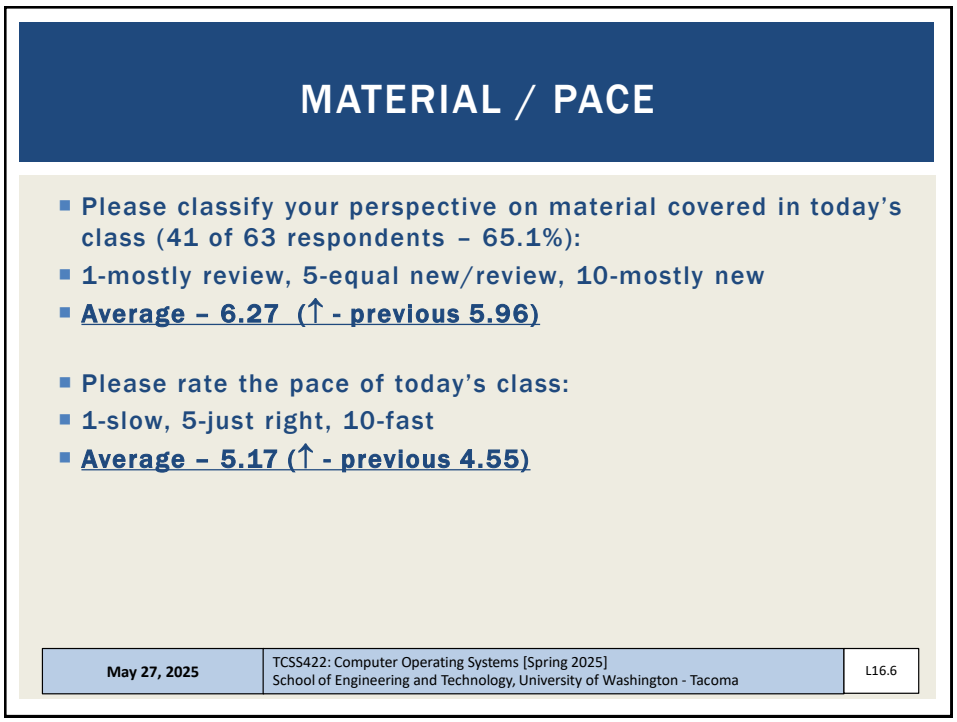
- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p



4


TCSS 422 - Online Daily Feedback Survey - 4/1
Quiz Instructions
Question 1
0.5 pts
On a scale of 1 to 10, please classify your perspective on material covered in today's class:
1 2 3 4 5 6 7 8 9 10
Mostly Review To Me
Equal New and Review
Mostly New to Me
Question 2
0.5 pts
Please rate the pace of today's class:
1 2 3 4 5 6 7 8 9 10
Slow
Just Right
Fast
May 27, 2025
TCSS422: Computer Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L16.5

5


MATERIAL / PACE
Please classify your perspective on material covered in today's class (41 of 63 respondents – 65.1%):
1-mostly review, 5-equal new/review, 10-mostly new
Average – 6.27 (↑ - previous 5.96)
Please rate the pace of today's class:
1-slow, 5-just right, 10-fast
Average – 5.17 (↑ - previous 4.55)
May 27, 2025
TCSS422: Computer Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L16.6

6

FEEDBACK FROM 5/22

- ***How do segmentation faults happen?***
  - See example on slides 9.22 and 9.23
  - Data on stack segment for child process returned, where the child process has ended – (this segment no longer exists)
- ***If they occur when trying to access memory outside of its own address space, how does say a off by one go out of program memory wouldn't it still be in the stack/heap space just accessing junk?***
- Correct – it accesses 'junk'
  - For "off by one", if the memory address accessed goes beyond the memory address allocated to store a variable, this likely is not a segmentation fault. The memory access is still within the heap or stack segment, the access simply reads from the wrong location.
- ***Also I thought the point of virtual address spaces was to improve isolation.***
  - The memory access is isolated – the program only reads from it's segments, it's just that the programmer made a pointer error

7

OBJECTIVES – 5/27

- Questions from 5/22
- **Assignment 2 - June 5 AOE**
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

8

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- **Quiz 3 – Synchronized Array - May 26 AOE**
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

9

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- **Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE**
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

10

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- **Assignment 3 (as a Tutorial) - June 10 AOE**
- Quiz 4 - Page Tables - To be posted
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

11

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- **Quiz 4 - Page Tables - To be posted**
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

12

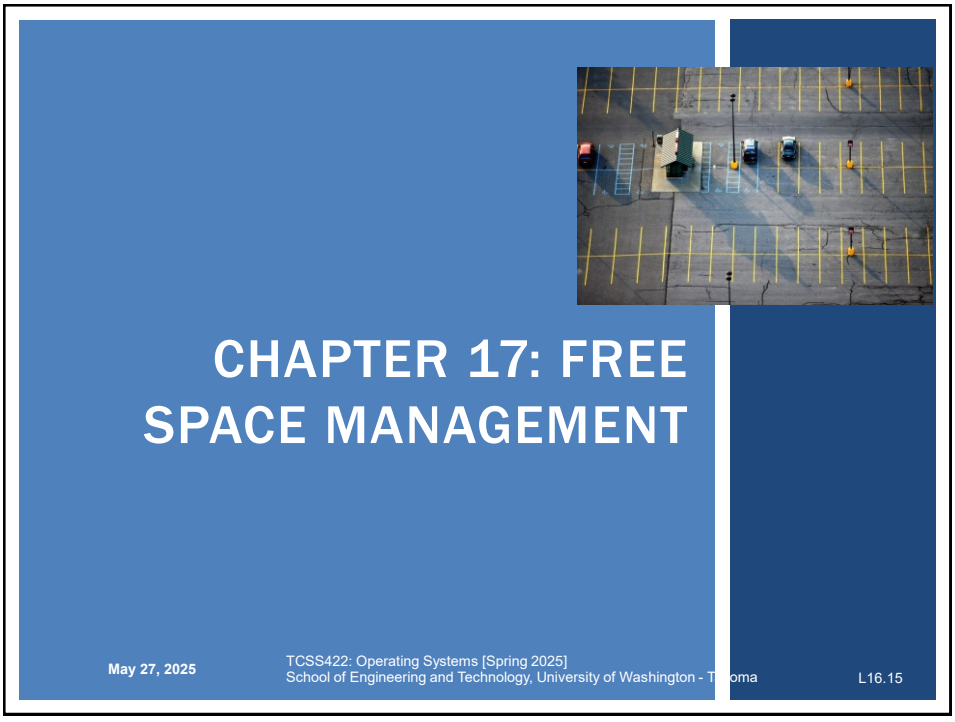
OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- **Chapter 17: Free Space Management**
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

13

WE ARE CAUGHT UP FROM LECTURE 15

14



15

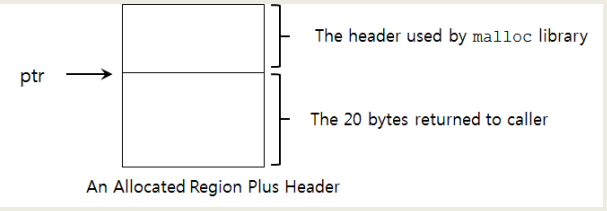
CHAPTER 17

- Free space list memory allocation:
  - Splitting – split chunk into 2, use one chunk, leave behind another
    - Fast
  - Coalescing – defragment the free space list
    - Time consuming, only perform when needed
- Next up
  - Special struct for free space list malloc allocations
- Free space list memory allocation strategies
  - Best fit, worst fit, first fit, next fit
- Another approach: binary buddy allocation

16

MEMORY HEADERS

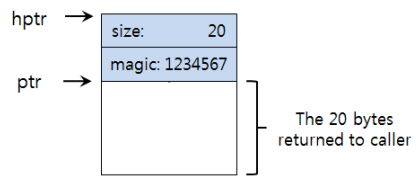
- free(void *ptr): Does not require a size parameter
- *How does the OS know how much memory to free?*
- Header block
  - Small descriptive block of memory at start of chunk



An Allocated Region Plus Header

17

MEMORY HEADERS - 2



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

18

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);
}
```


THE FREE LIST

- Simple free list struct

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

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

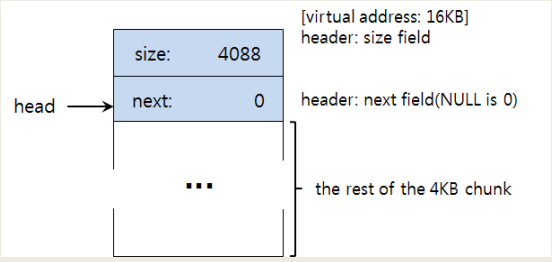
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

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;
```

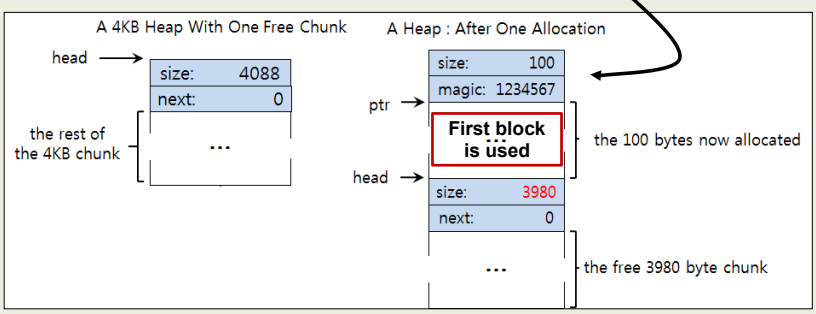
- Heap layout:



21

FREE LIST: MALLOC() CALL

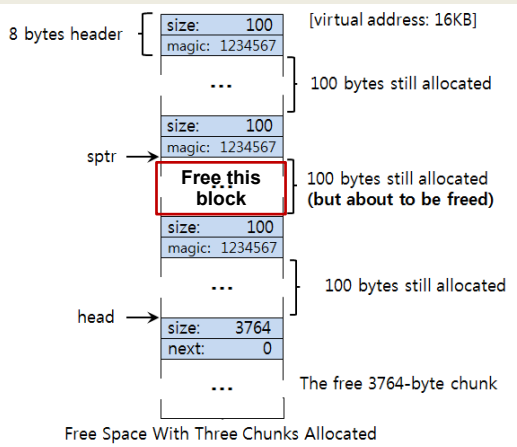
- Consider a request for a 100 bytes: malloc(100)
- Header block requires 8 bytes
  - 4 bytes for size, 4 bytes for magic number
- Split the heap – header goes with each block



22

FREE LIST: FREE() CALL

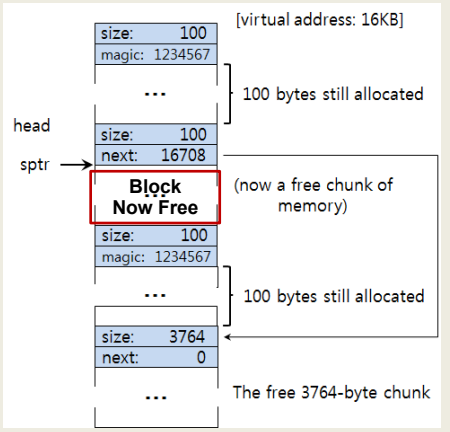
- Addresses of chunks
- Start=16384
  + 108 (end of 1st chunk)
  + 108 (end of 2nd chunk)
  + 108 (end of 3rd chunk)
  = 16708



23

FREE LIST: FREE() CHUNK #2

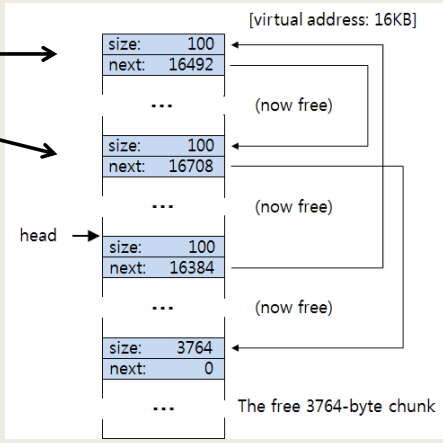
- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
  - addr – sizeof(node_t)
- Actual start of chunk #2
  - 16492



24

FREE LIST- FREE ALL CHUNKS

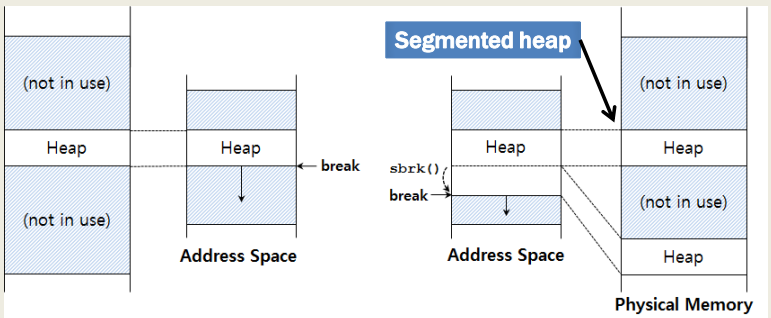
- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed



25

GROWING THE HEAP

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



26

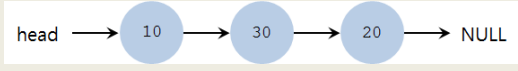
MEMORY ALLOCATION STRATEGIES

- **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

27

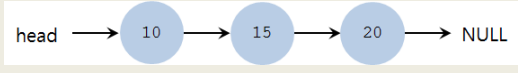
EXAMPLES

- **Allocation request for 15 bytes**



- **Result of Best Fit**

- **Result of Worst Fit**



28

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

29

Which memory allocation strategy is more likely to distribute free chunks closer together which could help when coalescing the free space list?

- Best Fit
- Worst Fit
- First Fit
- None of the above
- All of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

30

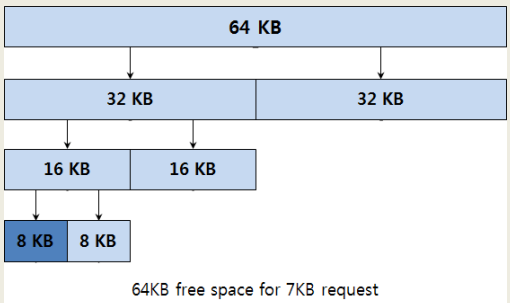
SEGREGATED LISTS

- 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

31

BINARY 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



32

BINARY BUDDY ALLOCATION - 2

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

33

A computer system manages program memory using three separate segments for code, stack, and the heap. The codesize of a program is 1KB but the minimal segment available is 16KB. This is an example of:

External fragmentation
Binary buddy allocation
Internal fragmentation
Coalescing
Splitting

34

A request is made to store 1 byte. For this scenario, which memory allocation strategy will always locate memory the fastest?

- Best fit
- Worst fit
- Next fit
- None of the above
- All of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

35

WE WILL RETURN AT 4:56PM

36

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- **Chapter 18: Introduction to Paging**
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

37

CHAPTER 18: INTRODUCTION TO PAGING

38

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

39

ADVANTAGES OF PAGING

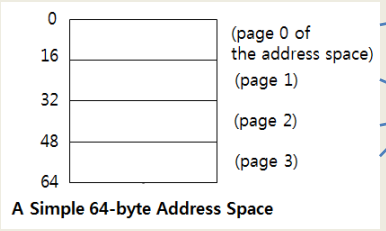
- Flexibility
  - Abstracts the process address space into pages
  - No need to track direction of HEAP / STACK growth
    - *Just add more pages…*
  - No need to store unused space
    - *As with segments…*
- Simplicity
  - Pages and page frames are the same size
  - Easy to allocate and keep a free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

40

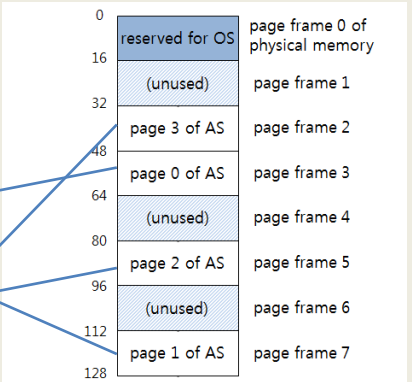
PAGING: EXAMPLE

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

- Consider a 128 byte (2^7) address space with 16-byte (2^4) pages
- Consider a 64-byte (2^6) program address space



A Simple 64-byte Address Space

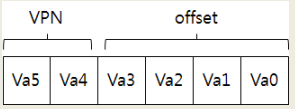


64-Byte Address Space Placed In Physical Memory

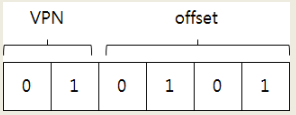
41

PAGING: ADDRESS TRANSLATION

- PAGE: Has two address components
  - VPN: Virtual Page Number (*serves as the page ID*)
  - Offset: Offset within a Page (*indexes any byte in the page*)



- Example:
  Page Size: 16-bytes (2^4),
  Program Address Space: 64-bytes (2^6)



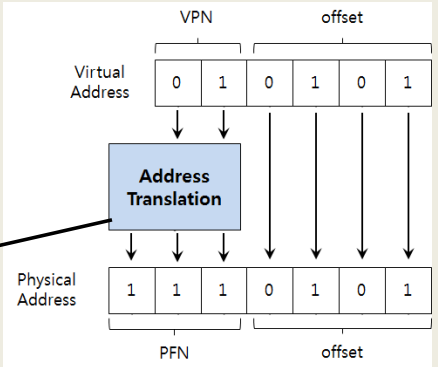
Here program can have just four pages…

42

EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte (2^6) program address space (4 pages→2^2)
- Stored in 128-byte (2^7) physical memory (8 frames→2^3)
- Offset is preserved
  - 4 bits indexes any byte
  - Page size is 16 bytes (2^4)
- **Page table** translates a Virtual Page Number (VPN) to a Physical Frame Number (PFN)

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



43

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?

44

(1) WHERE ARE PAGE TABLES STORED?

- Example:
  - Consider a 32-bit process address space (4GB=2^{32} bytes)
  - With 4 KB pages (4KB=2^{12} bytes)
  - 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

45

PAGE TABLE EXAMPLE

- With 2^{20} slots in our page table for a single process
- Each slot (i.e. entry) dereferences a VPN
- Each entry provides a physical frame number
- Each entry 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)

VPN_0
VPN_1
VPN_2
...
...
$VPN_{1048576}$

- How much memory is required to store the page table for 1 process?
  - Hint: # of entries x space per entry
  - 4,194,304 bytes (or 4MB) to index one process

46

NOW FOR AN ENTIRE OS

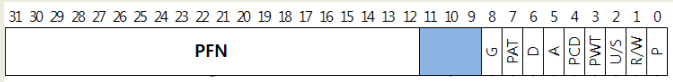
- If 4 MB is required to store one process
- Consider how much memory is required for an entire OS?
  - With for example 100 processes…
- Page table memory requirement is now 4MB x 100 = 400MB
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

  400 MB / 4000 GB
- **Is this efficient?**

47

(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

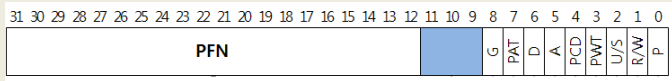


An x86 Page Table Entry(PTE)

48

PAGE TABLE ENTRY

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



An x86 Page Table Entry(PTE)

49

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

50

(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

(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

Stored in RAM →

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

- **Issue #2:** Each memory address translation for paging requires an extra memory reference
  - HW Support: TLBs (Chapter 19)

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)
```


COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

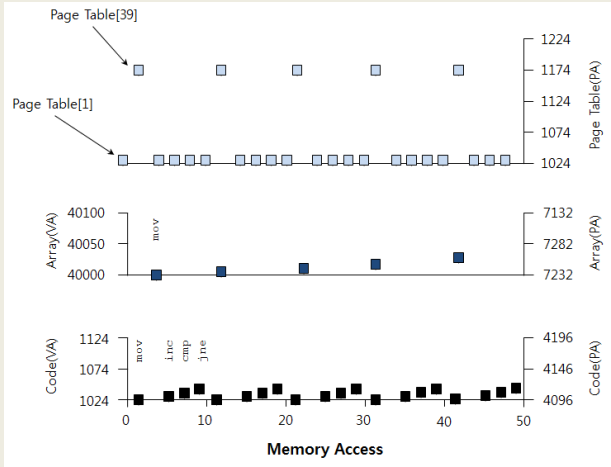
```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

- 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



55

Consider a 4GB Computer with 4KB (4096 byte) pages. How many pages would fit into physical memory?

2^32 / 2^20 = 2^12 pages

2^32 / 2^12 = 2^20 pages

2^32 / 2^16 = 2^16 pages

2^32 / 2^8 = 2^24 pages

None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

56

For the 4GB computer example, how many bits are required for the VPN?

- 24 VPN bits (indexes 2^24 locations)
- 16 VPN bits (indexes 2^16 locations)
- 20 VPN bits (indexes 2^20 locations)
- 12 VPN bits (indexes 2^12 locations)
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

57

For the 4GB computer example, how many bits are available for page status bits?

- 32 - 12 VPN bits = 20 status bits
- 32 - 24 VPN bits = 8 status bits
- 32 - 16 VPN bits = 16 status bits
- 32 - 20 VPN bits = 12 status bits
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

58

For the 4GB computer, how much space does this page table require? (number of page table entries x size of page table entry)

- 2^20 entries x 4b = 4 MB
- 2^12 entries x 4b = 16 KB
- 2^16 entries x 4b = 256 KB
- 2^24 entries x 4b = 64 MB
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

59

For the 4GB computer, how many page tables (for user processes) would fill the entire 4GB of memory?

- 4 GB / 16 KB = 65,536
- 4 GB / 64 MB = 256
- 4GB / 256 KB = 16,384
- 4GB / 4MB = 1,024
- None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

60

PAGING SYSTEM EXAMPLE

- Consider a 4GB Computer:
- With a 4096-byte page size (4KB)
- How many pages would fit in physical memory?

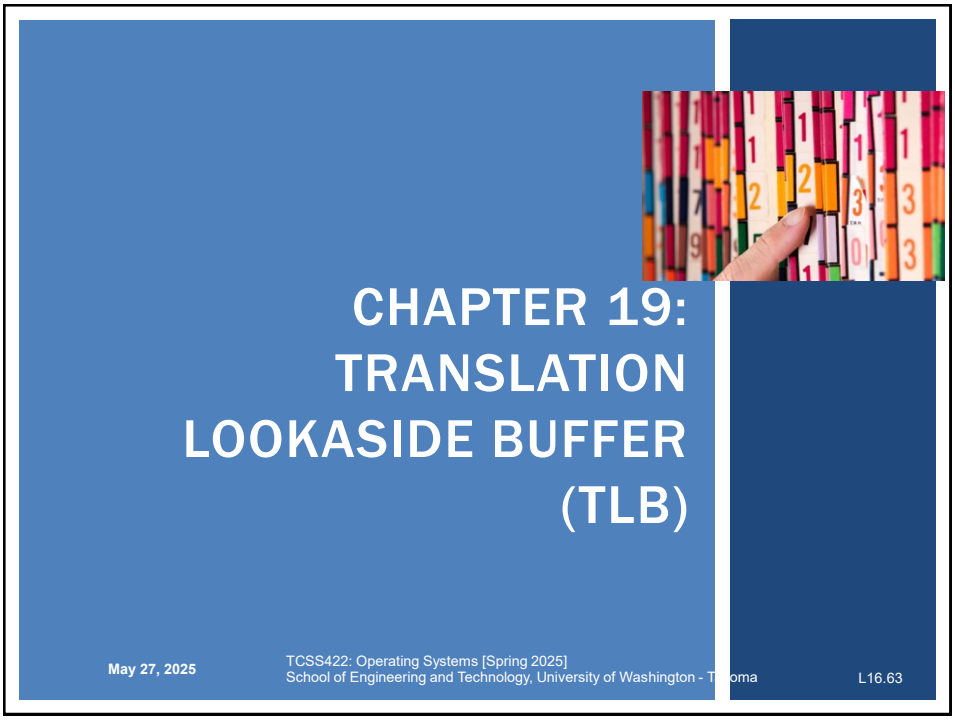
- Now consider a page table:
- For the page table entry, how many bits are required for the VPN?
- If we assume the use of 4-byte (32 bit) page table entries, how many bits are available for status bits?
- How much space does this page table require?
  # of page table entries x size of page table entry
- How many page tables (for user processes) would fill the entire 4GB of memory?

61

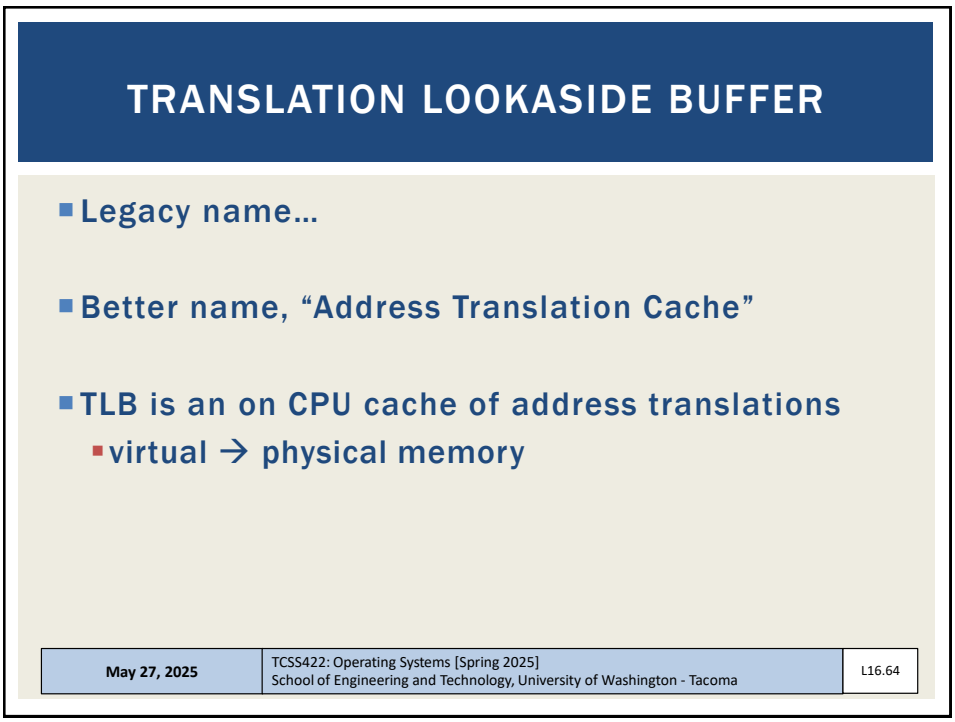
OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- **Chapter 19: Translation Lookaside Buffer (TLB)**
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

62


CHAPTER 19:
TRANSLATION
LOOKASIDE BUFFER
(TLB)
May 27, 2025
TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L16.63

63


TRANSLATION LOOKASIDE BUFFER
Legacy name…
Better name, "Address Translation Cache"
TLB is an on CPU cache of address translations
virtual → physical memory
May 27, 2025
TCSS422: Operating Systems [Spring 2025]
School of Engineering and Technology, University of Washington - Tacoma
L16.64

64

COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

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

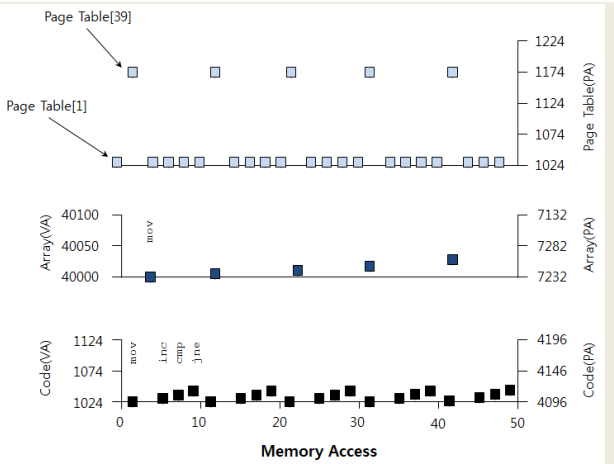
- Assembly equivalent:

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


65

VISUALIZING MEMORY ACCESSES: FOR THE FIRST 5 LOOP ITERATIONS

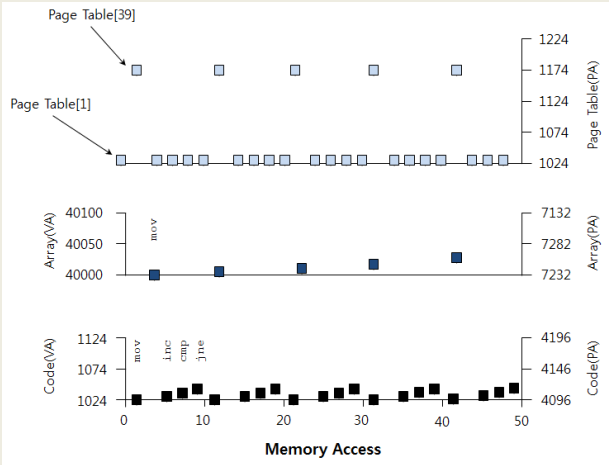
- Locations:
  - Page table
  - Array
  - Code
- 50 accesses for 5 loop iterations



66

TRANSLATION LOOKASIDE BUFFER - 2

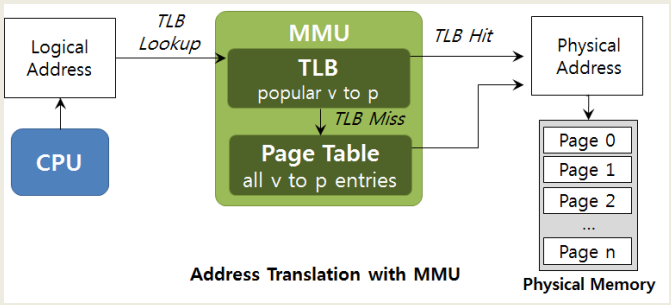
- Goal: Reduce access to the page tables
- Example: 50 RAM accesses for first 5 for-loop iterations
- Move lookups from RAM to TLB by caching page table entries



67

TRANSLATION LOOKASIDE BUFFER (TLB)

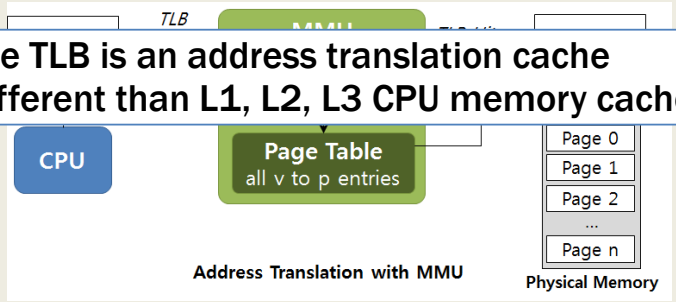
- Part of the CPU's Memory Management Unit (MMU)
- Address translation cache



68

TRANSLATION LOOKASIDE BUFFER (TLB)

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



69

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - **TLB Algorithm** Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

70

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

71

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)

72

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**

73

OBJECTIVES – 5/27

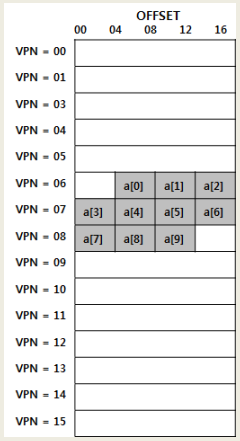
- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, **Hit-to-Miss Ratios**
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

74

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

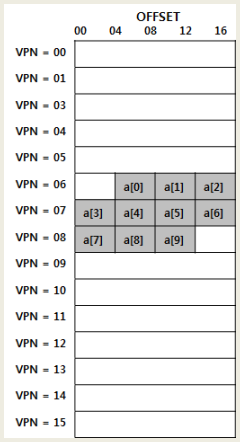


75

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?**



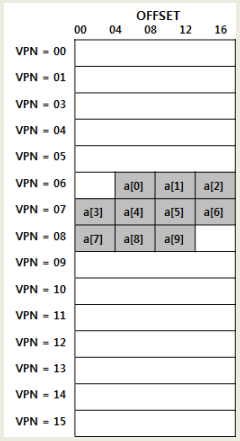
76

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)

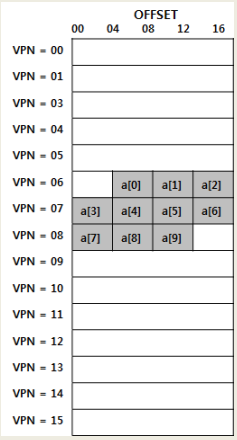


77

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/Access locality (how is data accessed?)
    - Sequential array access vs. random array access
  - Temporal locality
  - Size of the TLB cache
    (how much history can you store?)



78

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- **Chapter 20: Paging: Smaller Tables**
  - Smaller Tables, Multi-level Page Tables, N-level Page Tables

79

CHAPTER 20: PAGING: SMALLER TABLES

80

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

81

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

82

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

83

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - **Smaller Tables**, Multi-level Page Tables, N-level Page Tables

84

PAGING: USE LARGER PAGES

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

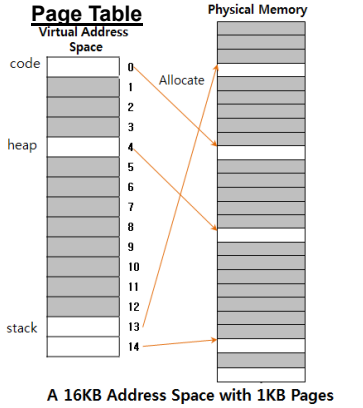
$$\frac{2^{32}}{2^{14}} * 4 = 1MB \quad \text{per page table}$$

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

85

PAGE TABLES: WASTED SPACE

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



A 16KB Address Space with 1KB Pages

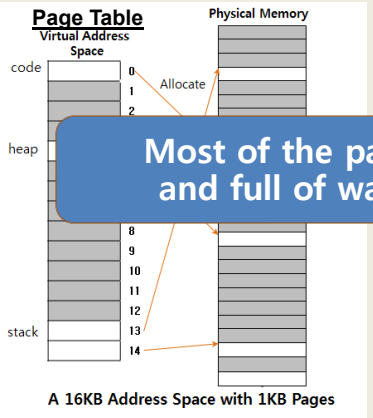
PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

86

PAGE TABLES: WASTED SPACE

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



A 16KB Address Space with 1KB Pages

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

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

87

OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, **Multi-level Page Tables** N-level Page Tables

88

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:

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

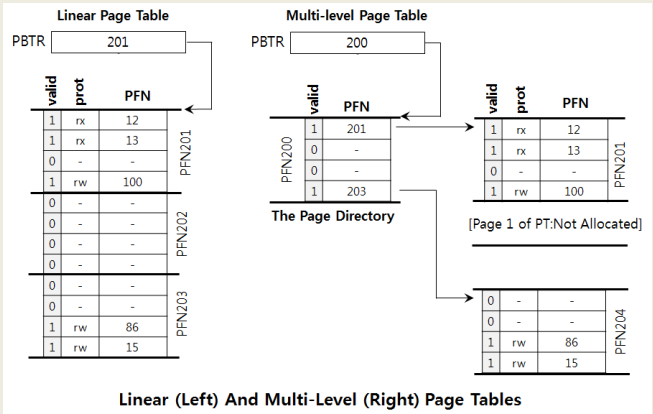
- Often most of the 4MB *per process* page table is empty
- Page table must be placed in 4MB contiguous block of RAM

- MUST SAVE MEMORY!

89

MULTI-LEVEL PAGE TABLES - 2

- Add level of indirection, the "page directory"

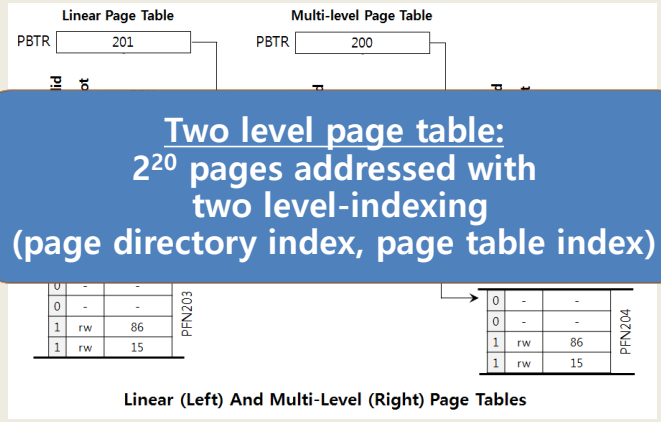


Linear (Left) And Multi-Level (Right) Page Tables

90

MULTI-LEVEL PAGE TABLES - 2

- Add level of indirection, the "page directory"



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

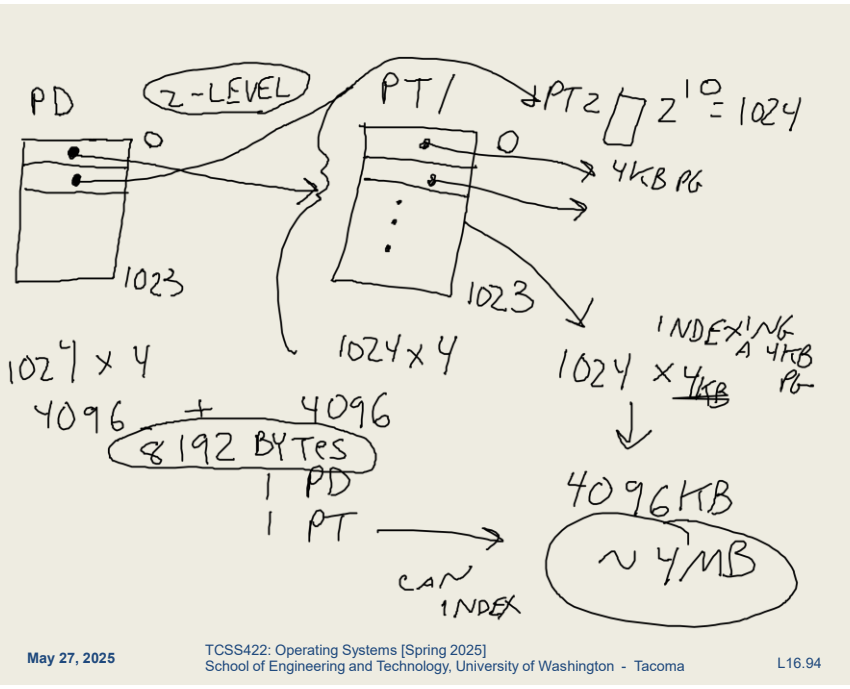
91

4 GB computer (2^32) and 4KB pages (2^12)

1. How much space is required for a 2-level page table with one page directory (PD) and one page table (PT)?
2. How much memory can a single PD pointing to a single PT address?

92

93


PD
2-LEVEL
PT1
PT2
$2^{10} = 1024$
0
0
4KB PG
1023
1023
1024 x 4
1024 x 4
1024 x 4KB
INDEXING A 4KB PG
4096 + 4096
8192 BYTES
1 PD
1 PT
4096KB
~ 4MB
CAN INDEX

94

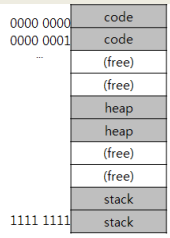
MULTI-LEVEL PAGE TABLES - 3

- 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

95

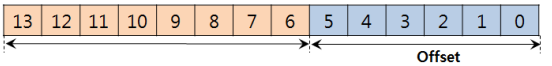
EXAMPLE

- 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



96

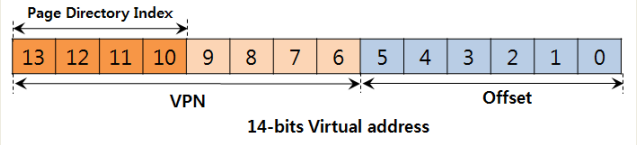
EXAMPLE - 2

- 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!**

97

PAGE DIRECTORY INDEX

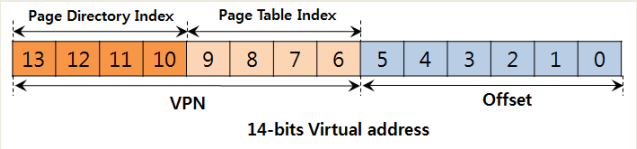
- Now, let's split the page table into two:
  - 8 bit VPN to map 256 pages
  - 4 bits for page directory index (PDI – 1st level page table)
  - 6 bits offset into 64-byte page



98

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

101

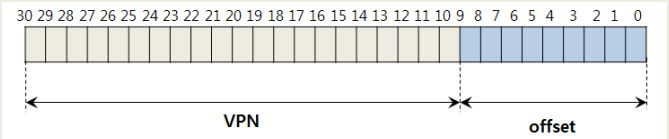
OBJECTIVES – 5/27

- Questions from 5/22
- Assignment 2 - June 5 AOE
- Quiz 3 – Synchronized Array - May 26 AOE
- Tutorial 2 – Pthread/locks/conditions tutorial 5/29 AOE
- Assignment 3 (as a Tutorial) - June 10 AOE
- Quiz 4 - Page Tables - To be posted
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Hit-to-Miss Ratios
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Multi-level Page Tables, **N-level Page Tables**

102

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

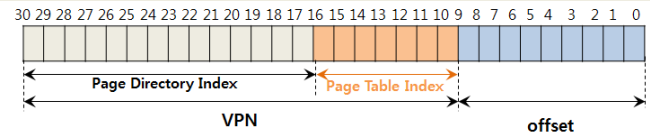


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit

103

MORE THAN TWO LEVELS - 2

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

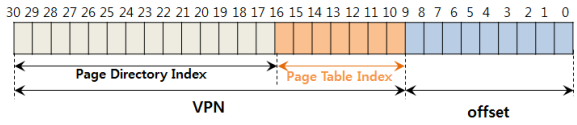


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs → $\log_2 128 = 7$

104

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 → $\log_2 128 = 7$

105

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 → $\log_2 128 = 7$

106

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

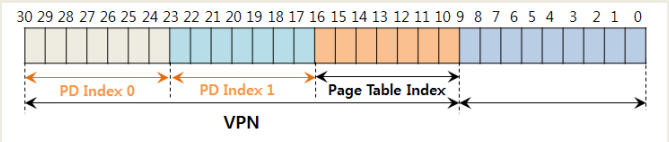
Need three level page table:
Page directory 0 (PD Index 0)
Page directory 1 (PD Index 1)
Page Table Index

Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs → $\log_2 128 = 7$

107

MORE THAN TWO LEVELS - 4

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



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

108

ADDRESS TRANSLATION CODE

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

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


109

ADDRESS TRANSLATION - 2

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

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

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

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

110

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

111

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?

112

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)?

113

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

114

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

115

ANSWERS

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

116

QUESTIONS

117