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

MIDTERM REVIEW SESSION

- Make-up midterm exams are completed and scores are posted
- Midterm exams are available for pick-up in class through May 30 (Lecture 19)

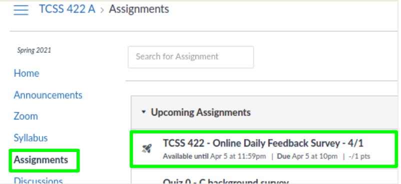
- **Midterm Review Session:**
- Tuesday May 21, 6:00 pm (during office hour, in BHS106)
- Via Zoom / Live Stream / Recording
- Will discuss and review midterm exam problems and grading

OBJECTIVES – 5/21

- **Questions from 5/16**
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

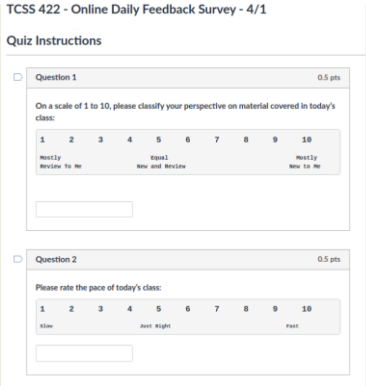
ONLINE DAILY FEEDBACK SURVEY

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



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

Quiz 0 - C background survey



TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 0.5 pts

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

```
1     2     3     4     5     6     7     8     9     10

Mostly                  Equal                          Mostly
Review To Me        New and Review                 New to Me
```

Question 2 0.5 pts

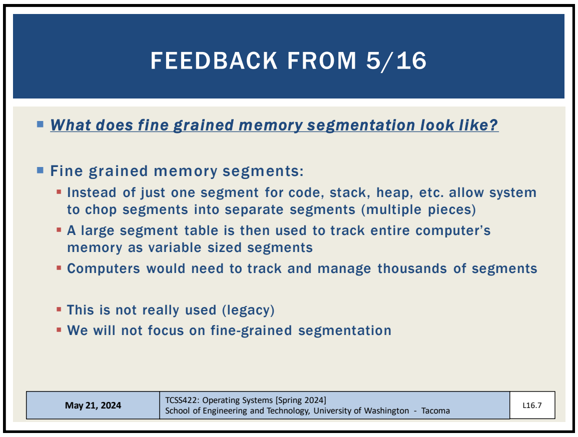
Please rate the pace of today's class:

```
1     2     3     4     5     6     7     8     9     10

Slow                Just Right                       Fast
```

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (27 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.00 (↑ - previous 5.96)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.19 (↓ - previous 5.42)**



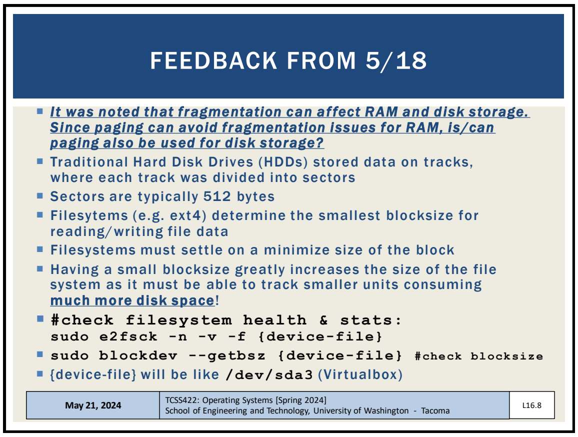
FEEDBACK FROM 5/16

- ***What does fine grained memory segmentation look like?***

- Fine grained memory segments:
  - Instead of just one segment for code, stack, heap, etc. allow system to chop segments into separate segments (multiple pieces)
  - A large segment table is then used to track entire computer's memory as variable sized segments
  - Computers would need to track and manage thousands of segments

  - This is not really used (legacy)
  - We will not focus on fine-grained segmentation

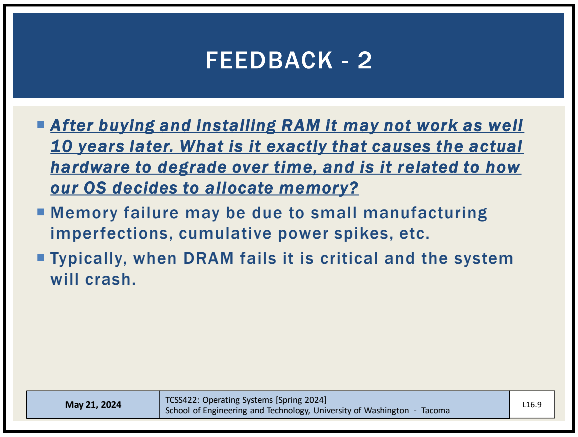
7



FEEDBACK FROM 5/18

- ***It was noted that fragmentation can affect RAM and disk storage. Since paging can avoid fragmentation issues for RAM, is/can paging also be used for disk storage?***
- Traditional Hard Disk Drives (HDDs) stored data on tracks, where each track was divided into sectors
- Sectors are typically 512 bytes
- Filesytems (e.g. ext4) determine the smallest blocksize for reading/writing file data
- Filesystems must settle on a minimize size of the block
- Having a small blocksize greatly increases the size of the file system as it must be able to track smaller units consuming ***much more disk space***!
- `#check filesystem health & stats:`
  `sudo e2fsck -n -v -f {device-file}`
- `sudo blockdev --getbsz {device-file} #check blocksize`
- {device-file} will be like `/dev/sda3` (Virtualbox)

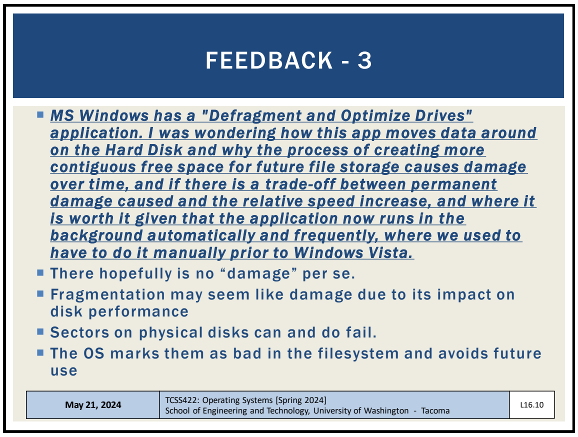
8



FEEDBACK - 2

- ***After buying and installing RAM it may not work as well 10 years later. What is it exactly that causes the actual hardware to degrade over time, and is it related to how our OS decides to allocate memory?***
- Memory failure may be due to small manufacturing imperfections, cumulative power spikes, etc.
- Typically, when DRAM fails it is critical and the system will crash.

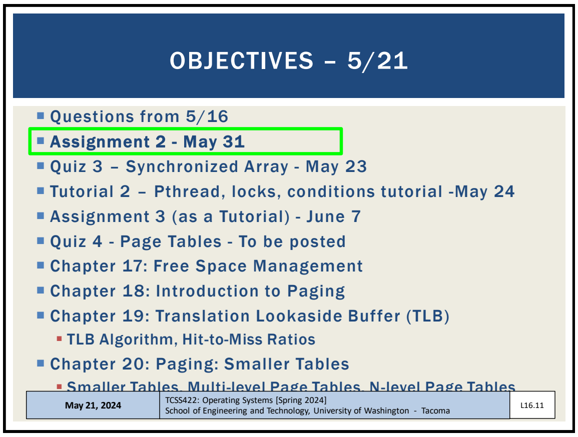
9



FEEDBACK - 3

- ***MS Windows has a "Defragment and Optimize Drives" application. I was wondering how this app moves data around on the Hard Disk and why the process of creating more contiguous free space for future file storage causes damage over time, and if there is a trade-off between permanent damage caused and the relative speed increase, and where it is worth it given that the application now runs in the background automatically and frequently, where we used to have to do it manually prior to Windows Vista.***
- There hopefully is no "damage" per se.
- Fragmentation may seem like damage due to its impact on disk performance
- Sectors on physical disks can and do fail.
- The OS marks them as bad in the filesystem and avoids future use

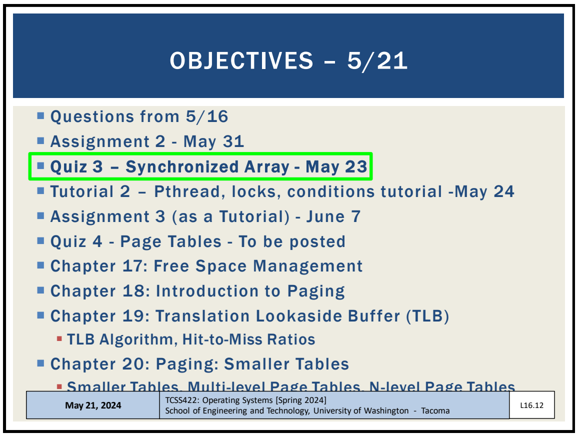
10



OBJECTIVES – 5/21

- Questions from 5/16
- **Assignment 2 - May 31**
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

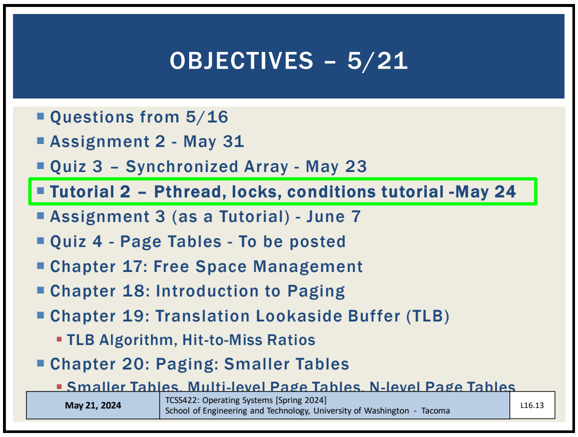
11



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- **Quiz 3 – Synchronized Array - May 23**
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

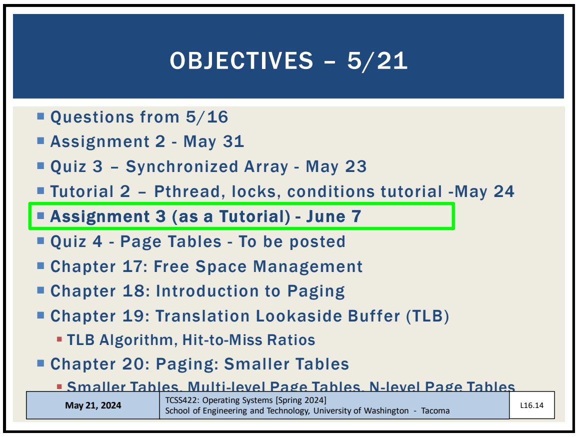
12



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- **Tutorial 2 – Pthread, locks, conditions tutorial -May 24**
- Assignment 3 (as a Tutorial) - June 7
- 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

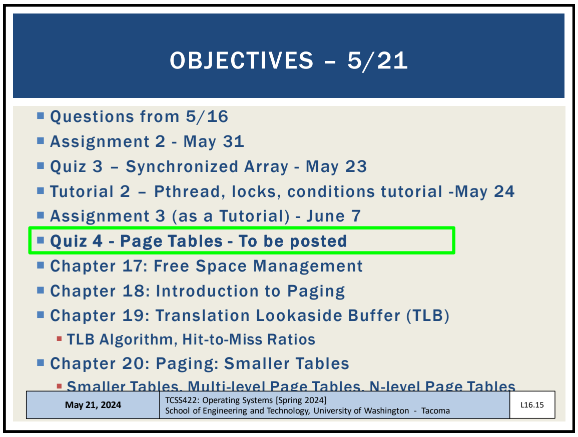
13



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- **Assignment 3 (as a Tutorial) - June 7**
- 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

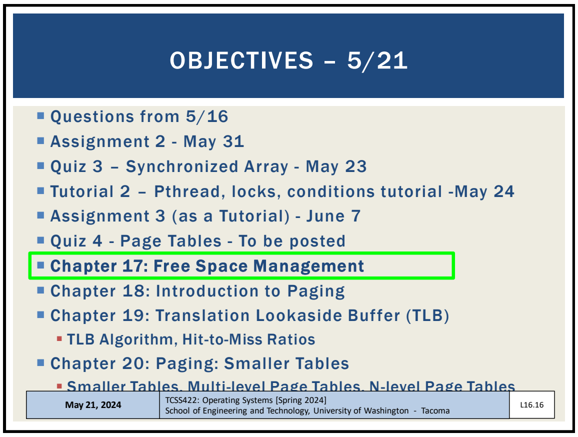
14



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- **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

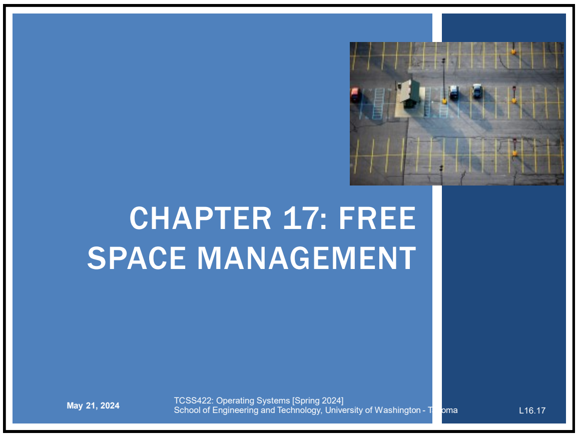
15



OBJECTIVES – 5/21

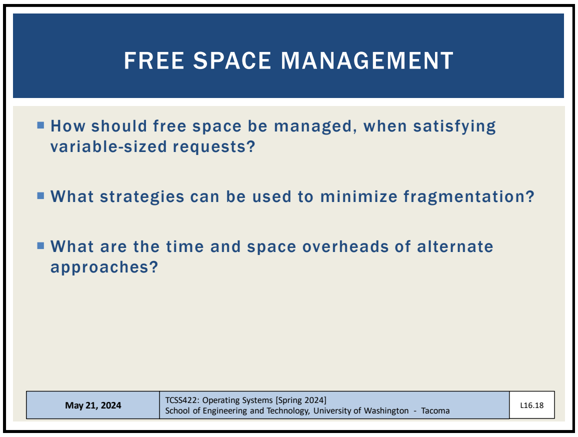
- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- Quiz 4 - Page Tables - To be posted
- **Chapter 17: Free Space Management**
- Chapter 18: Introduction to Paging
- 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

16



CHAPTER 17: FREE SPACE MANAGEMENT

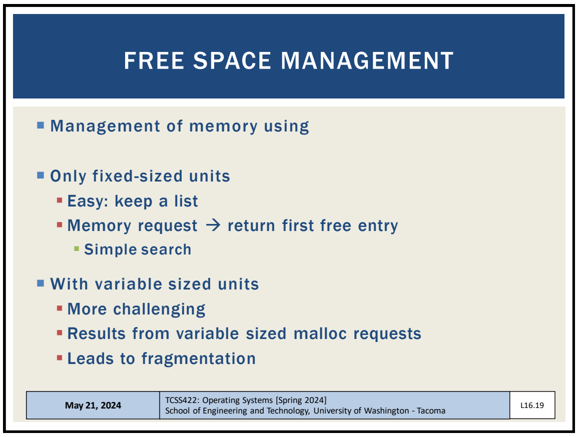
17



FREE SPACE MANAGEMENT

- How should free space be managed, when satisfying variable-sized requests?
- What strategies can be used to minimize fragmentation?
- What are the time and space overheads of alternate approaches?

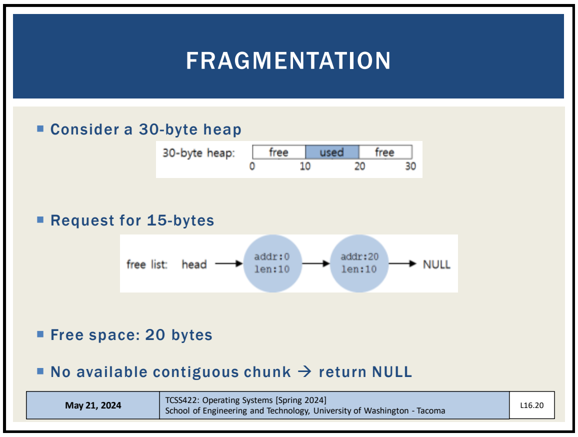
18



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

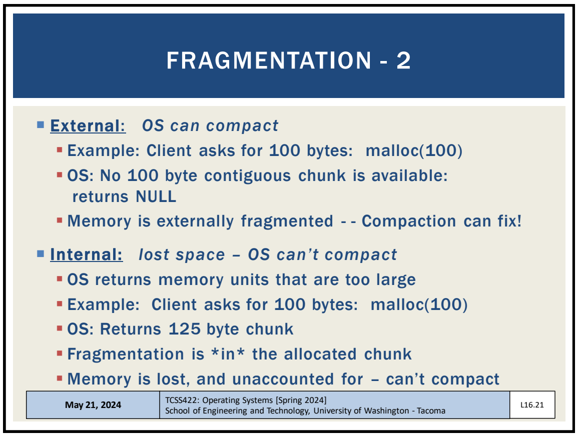


FRAGMENTATION

- Consider a 30-byte heap

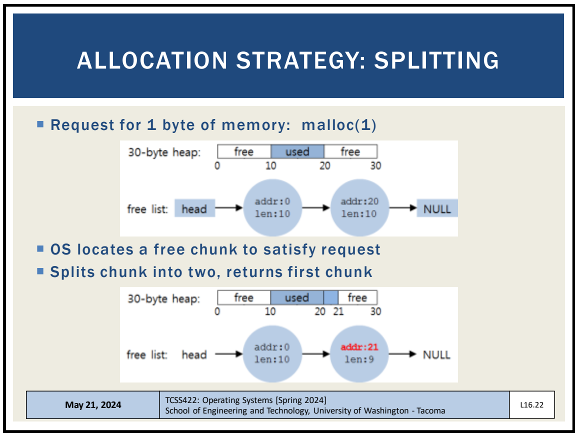
- Request for 15-bytes

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



FRAGMENTATION - 2

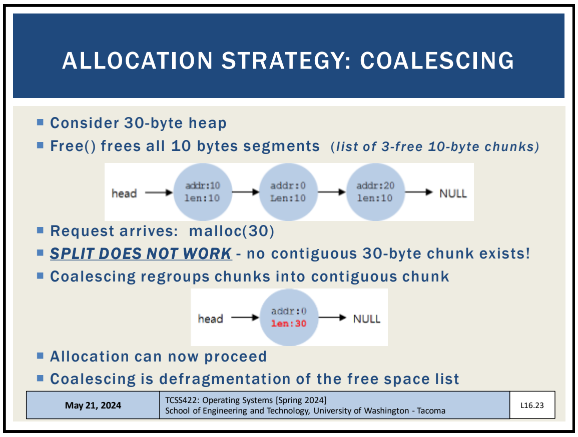
- **External:** *OS can compact*
  - Example: Client asks for 100 bytes: malloc(100)
  - OS: No 100 byte contiguous chunk is available: returns NULL
  - Memory is externally fragmented - - Compaction can fix!
- **Internal:** *lost space – OS can't compact*
  - OS returns memory units that are too large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



ALLOCATION STRATEGY: SPLITTING

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

- OS locates a free chunk to satisfy request
- Splits chunk into two, returns first chunk

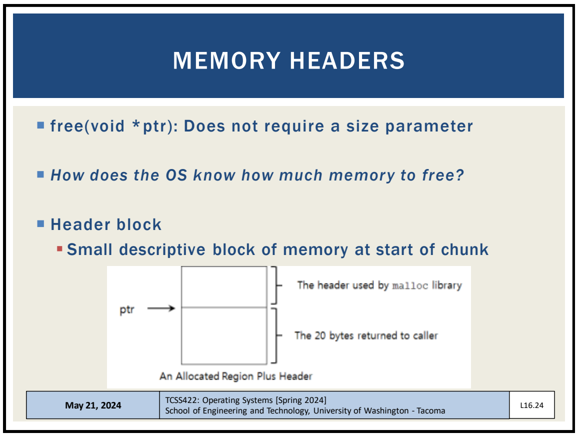


ALLOCATION STRATEGY: COALESCING

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

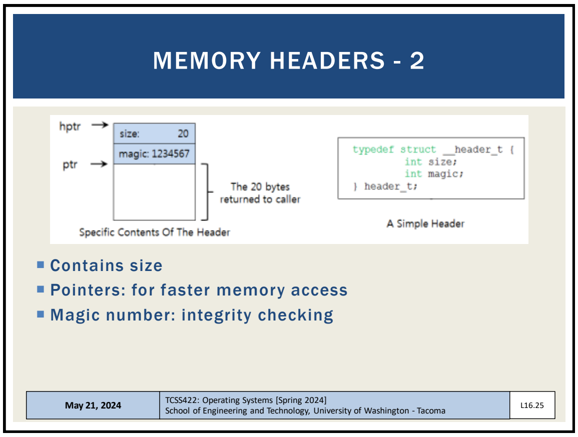
- Request arrives: malloc(30)
- ***SPLIT DOES NOT WORK*** - no contiguous 30-byte chunk exists!
- Coalescing regroups chunks into contiguous chunk

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



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



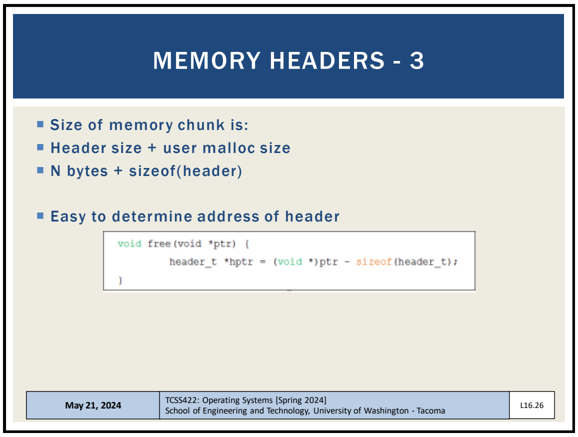
MEMORY HEADERS - 2

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

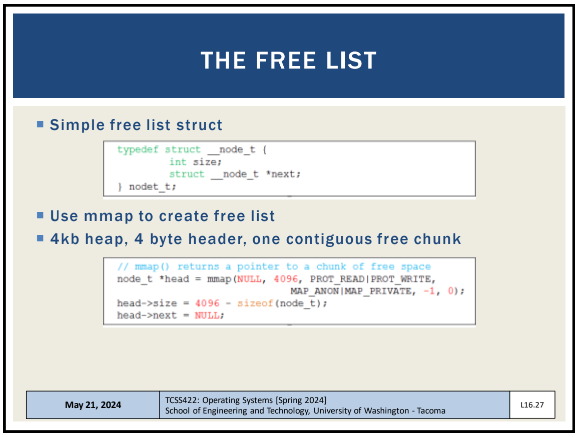


MEMORY HEADERS - 3

- 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);
}
```



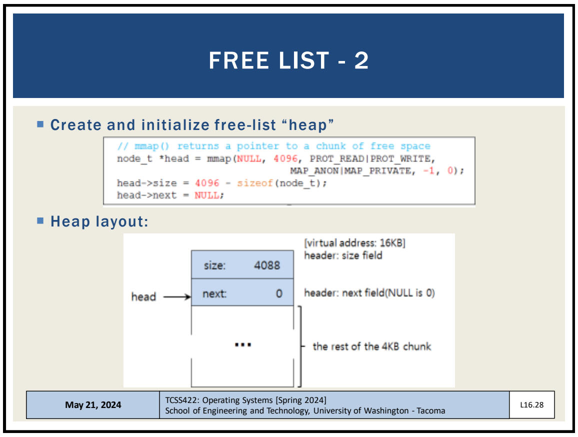
THE FREE LIST

- 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;
```

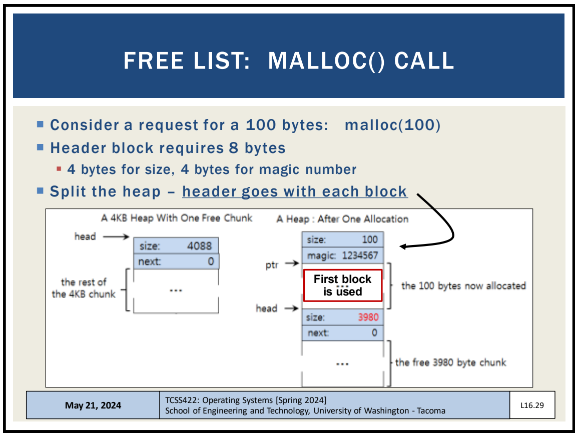


FREE LIST - 2

- Create and initialize free-list “heap”

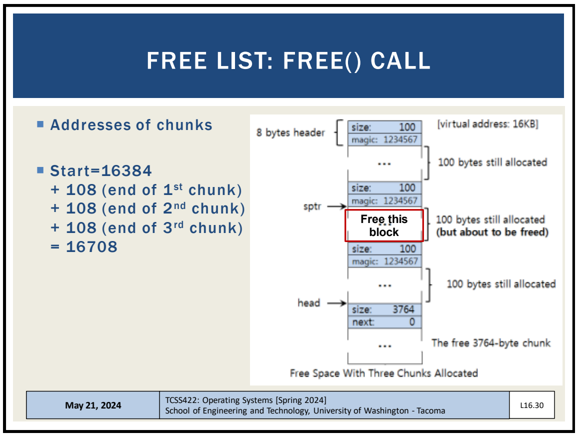
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                            MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```

- Heap layout:



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 – <u>header goes with each block</u>



FREE LIST: FREE() CALL

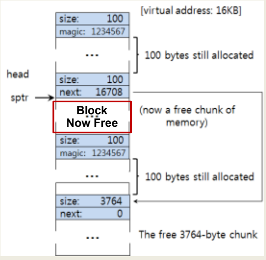
- Addresses of chunks

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

Free Space With Three Chunks Allocated

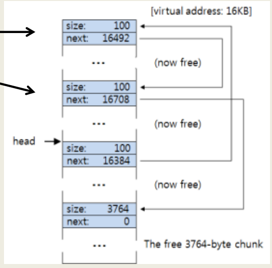
FREE LIST: FREE() CHUNK #2

- 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



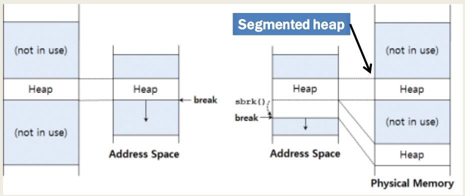
FREE LIST- FREE ALL CHUNKS

- 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



GROWING THE HEAP

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

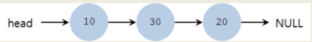


MEMORY ALLOCATION STRATEGIES

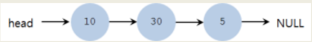
- **Best fit**
  - Traverse free list
  - Identify all candidate free chunks
  - Note which is smallest (has best fit)
  - When splitting, "leftover" pieces are small (and potentially less useful -- fragmented)
- **Worst fit**
  - Traverse free list
  - Identify largest free chunk
  - Split largest free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

EXAMPLES

- Allocation request for 15 bytes



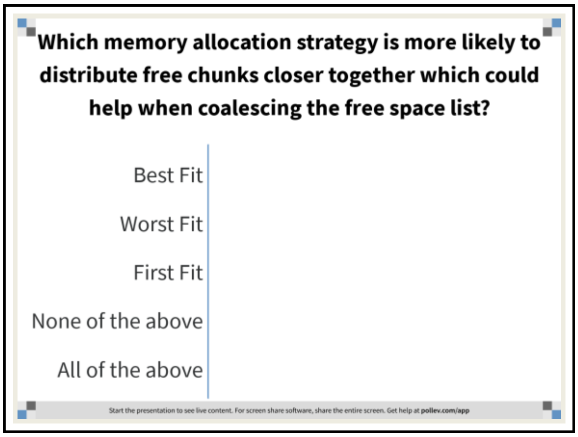
- Result of Best Fit



- Result of Worst Fit

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

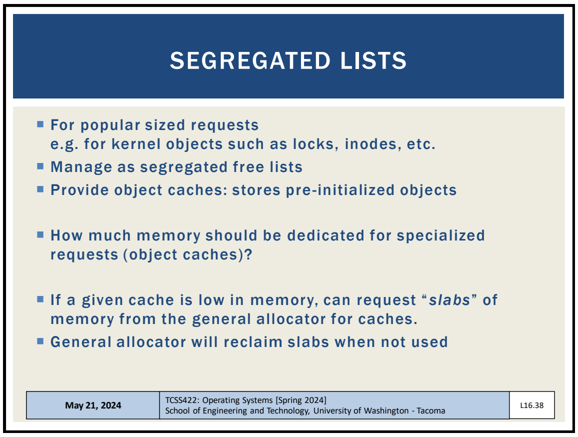


Which memory allocation strategy is more likely to distribute free chunks closer together which could help when coalescing the free space list?

- 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**

37



SEGREGATED LISTS

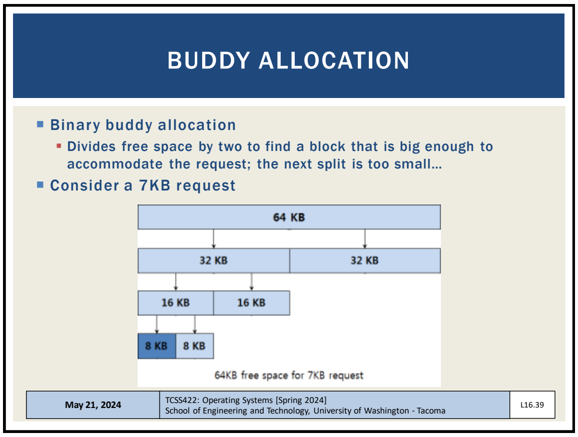
- For popular sized requests
  e.g. for kernel objects such as locks, inodes, etc.
- Manage as segregated free lists
- Provide object caches: stores pre-initialized objects

- 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 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.38

38



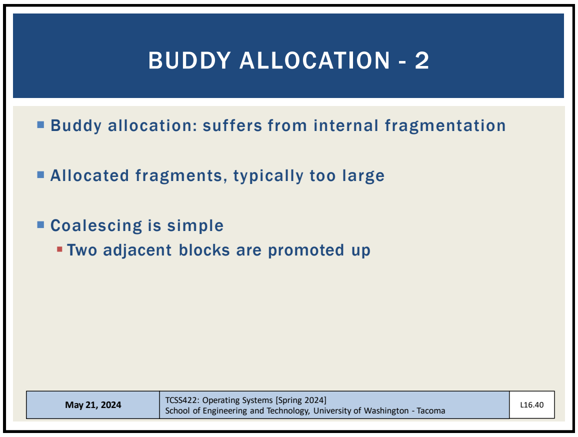
BUDDY ALLOCATION

- 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

64KB free space for 7KB request

May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.39

39



BUDDY ALLOCATION - 2

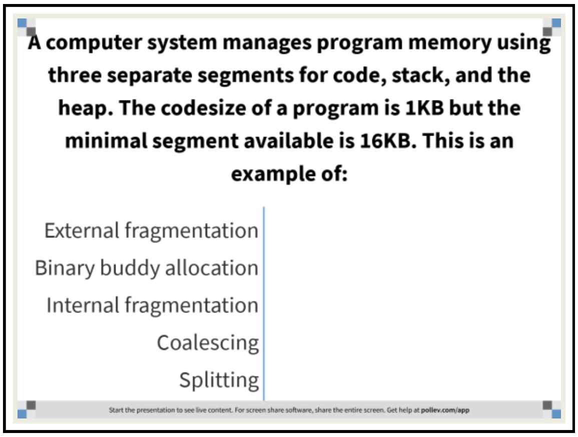
- Buddy allocation: suffers from internal fragmentation

- Allocated fragments, typically too large

- Coalescing is simple
  - Two adjacent blocks are promoted up

May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.40

40

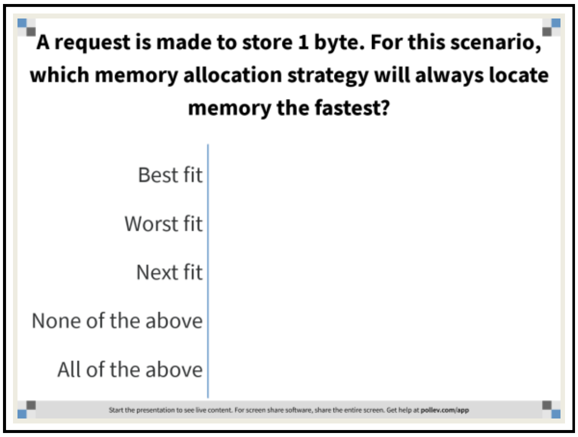


A computer system manages program memory using three separate segments for code, stack, and the heap. The codesize of a program is 1KB but the minimal segment available is 16KB. This is an example of:

- External fragmentation
- Binary buddy allocation
- Internal fragmentation
- Coalescing
- Splitting

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

41



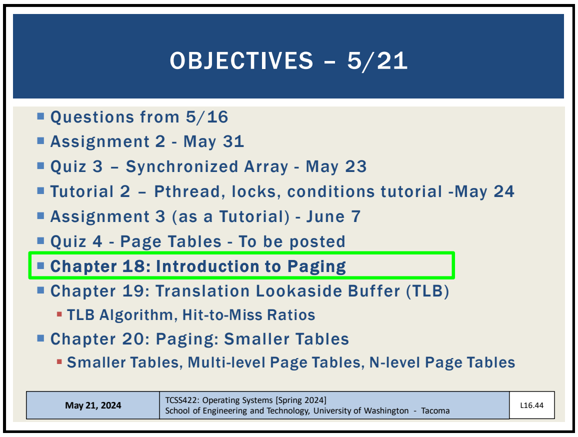
A request is made to store 1 byte. For this scenario, which memory allocation strategy will always locate memory the fastest?

- 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**

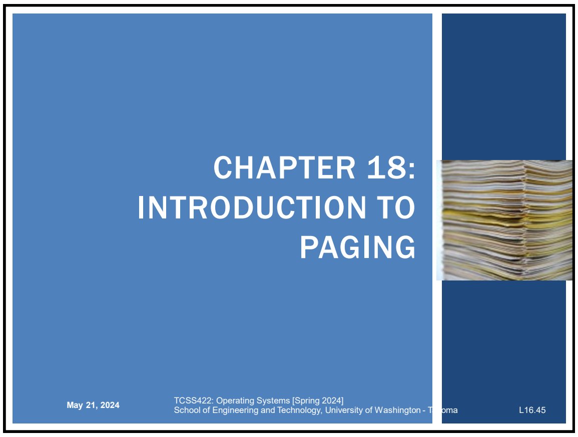
42

WE WILL RETURN AT 5:05PM

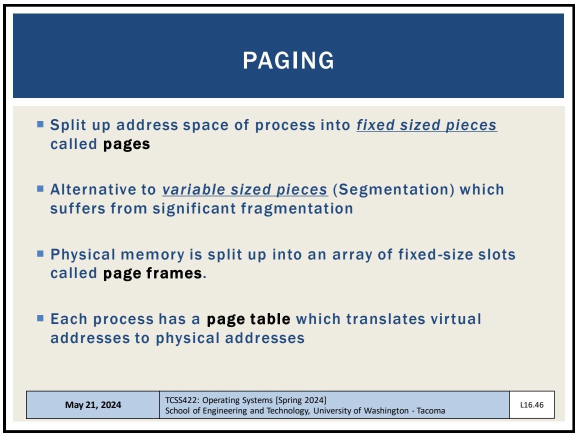


OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

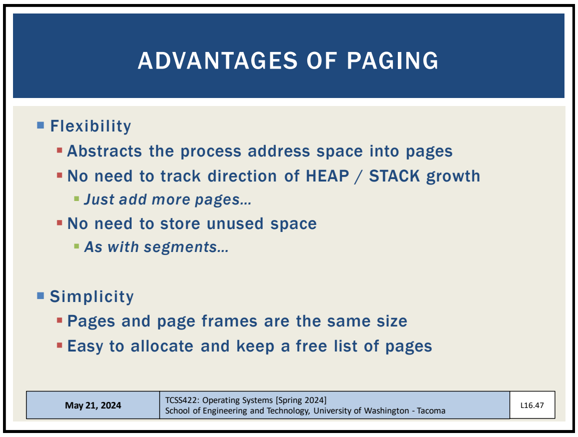


CHAPTER 18: INTRODUCTION TO PAGING



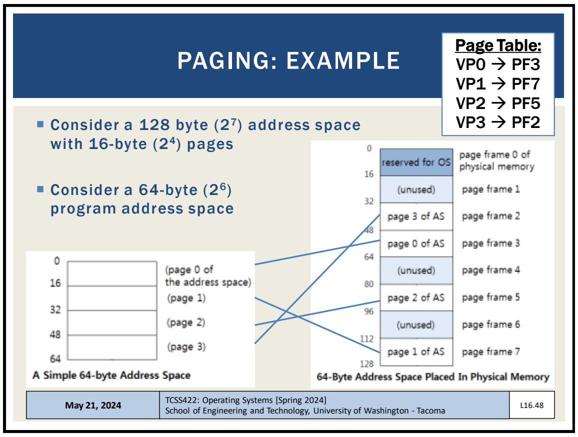
PAGING

- 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



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



PAGING: EXAMPLE

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

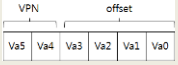
- Consider a 128 byte (2^7) address space with 16-byte (2^4) pages
- Consider a 64-byte (2^6) program address space

A Simple 64-byte Address Space

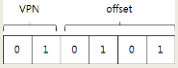
64-Byte Address Space Placed In Physical Memory

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)

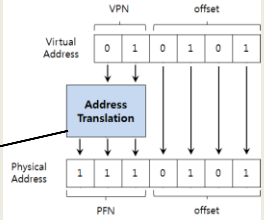


Here program can have just four pages...

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

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?

(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

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
...
...
$VPN_{1048576}$

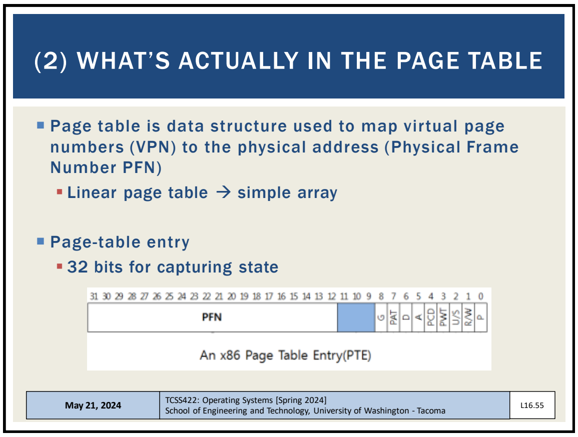
- How much memory is required to store the page table for 1 process?
  - Hint: # of entries x space per entry
  - 4,194,304 bytes (or 4MB) to index one process

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

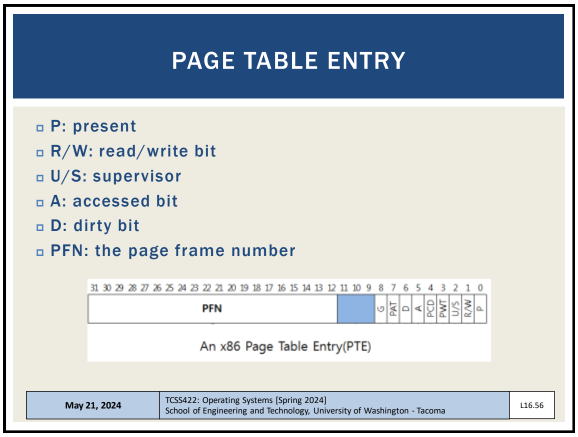
- **Is this efficient?**



(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

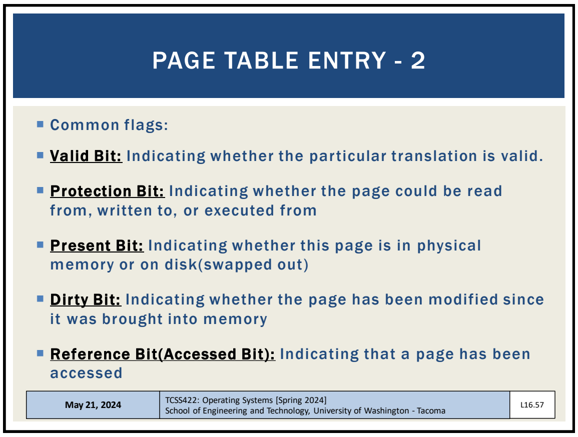
An x86 Page Table Entry(PTE)



PAGE TABLE ENTRY

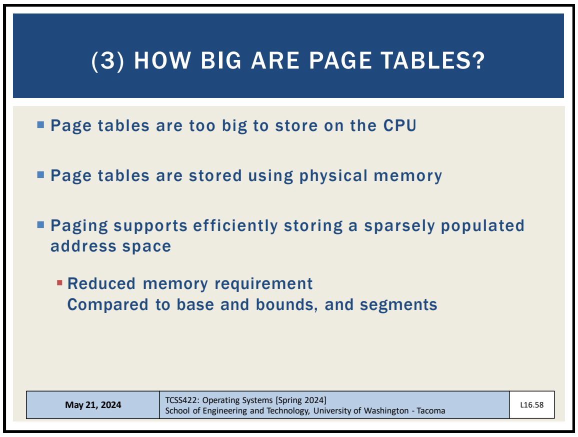
- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

An x86 Page Table Entry(PTE)



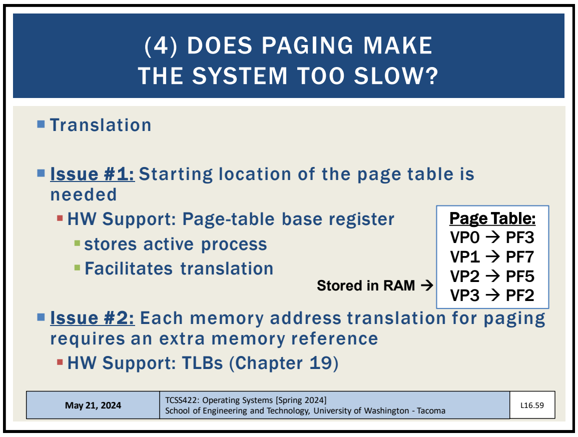
PAGE TABLE ENTRY - 2

- 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



(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

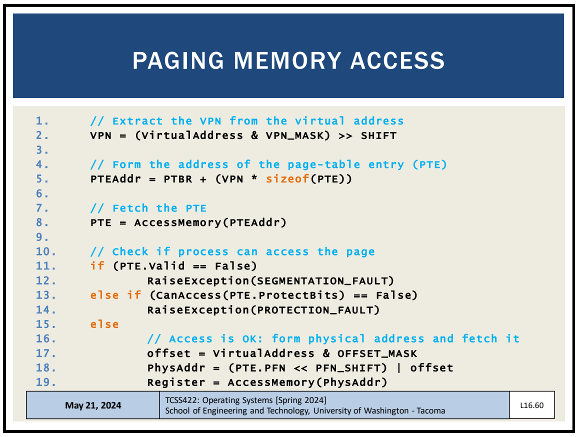


(4) DOES PAGING MAKE THE SYSTEM TOO SLOW?

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

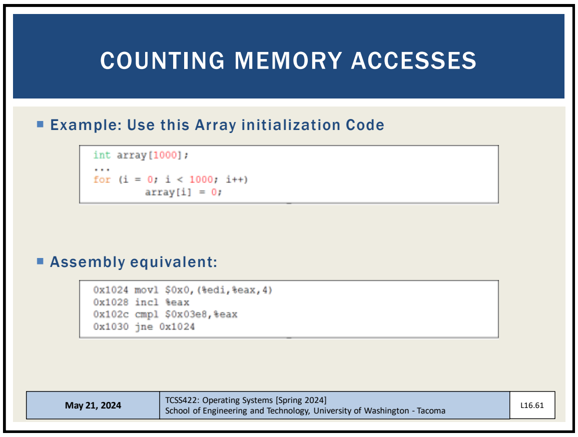
Stored in RAM →

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



PAGING MEMORY ACCESS

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT)
else
        // Access is OK: form physical address and fetch it
        offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
        Register = AccessMemory(PhysAddr)
```



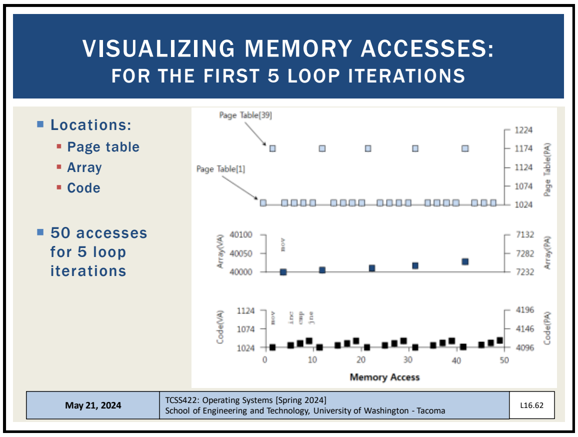
COUNTING MEMORY ACCESSES

- **Example: Use this Array initialization Code**

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

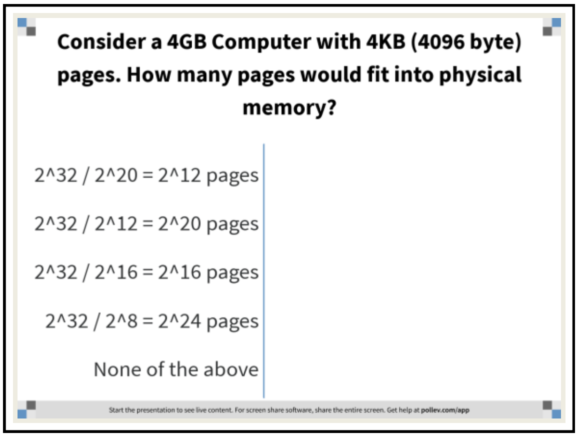
- **Assembly equivalent:**

```
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**



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

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

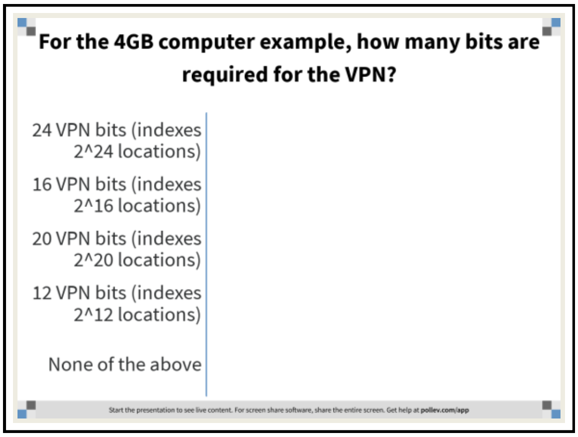
2^32 / 2^12 = 2^20 pages

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**



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

24 VPN bits (indexes 2^24 locations)

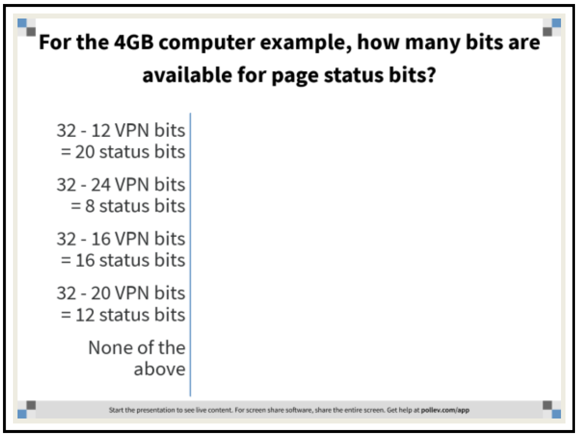
16 VPN bits (indexes 2^16 locations)

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**



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

32 - 12 VPN bits = 20 status bits

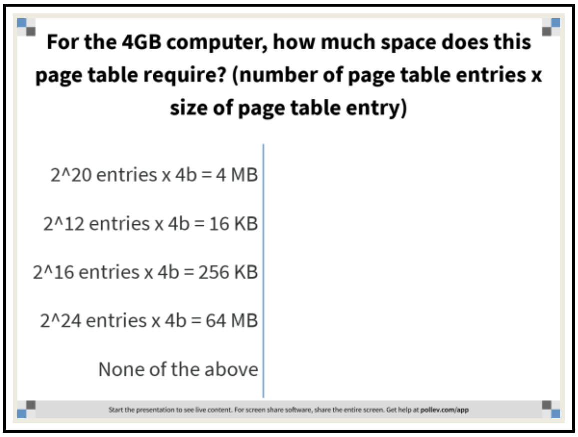
32 - 24 VPN bits = 8 status bits

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**



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

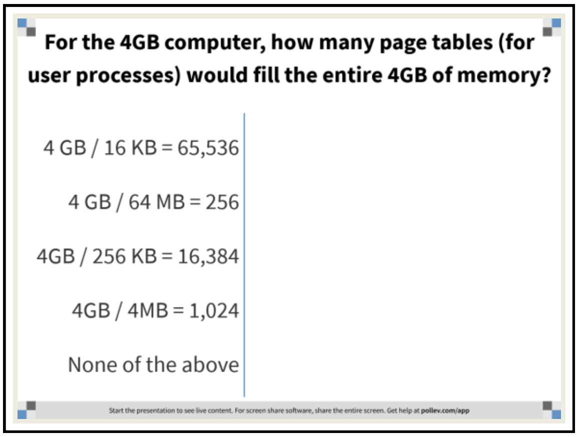
2^12 entries x 4b = 16 KB

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**



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

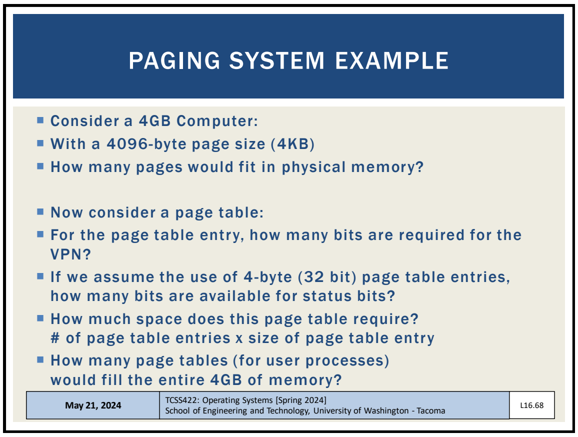
4GB / 256 KB = 16,384

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

67



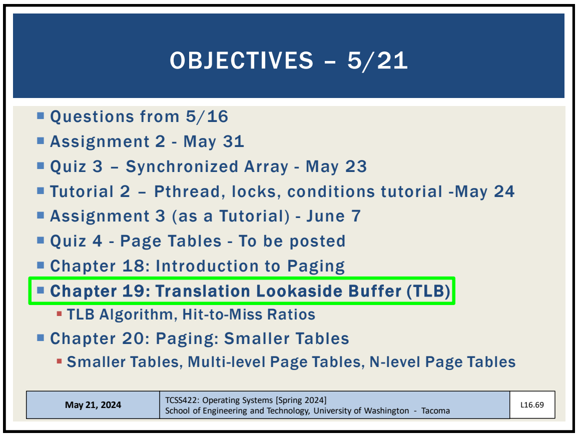
PAGING SYSTEM EXAMPLE

- 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?

May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.68

68

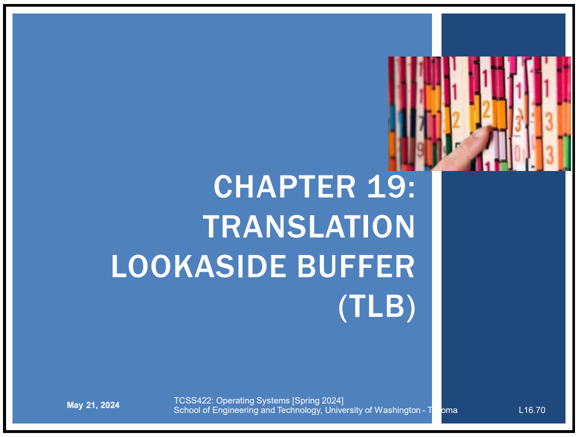


OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.69

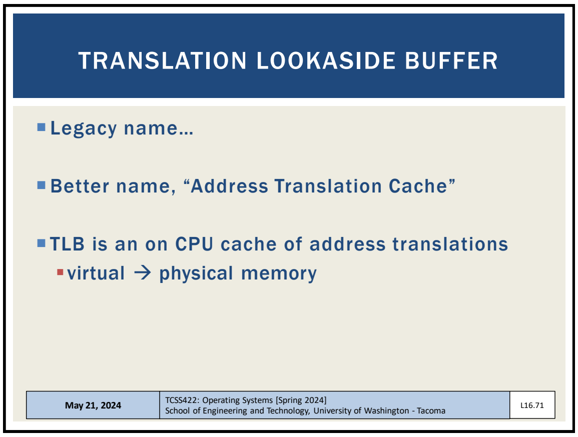
69



CHAPTER 19: TRANSLATION LOOKASIDE BUFFER (TLB)

May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - T oma | L16.70

70



TRANSLATION LOOKASIDE BUFFER

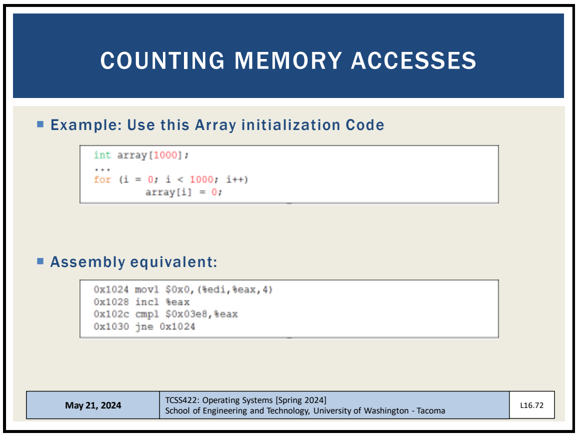
- Legacy name…

- Better name, "Address Translation Cache"

- TLB is an on CPU cache of address translations
  - virtual → physical memory

May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.71

71



COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

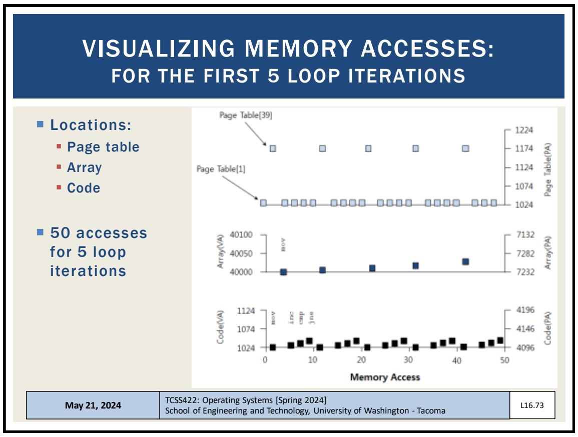
```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

- Assembly equivalent:

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

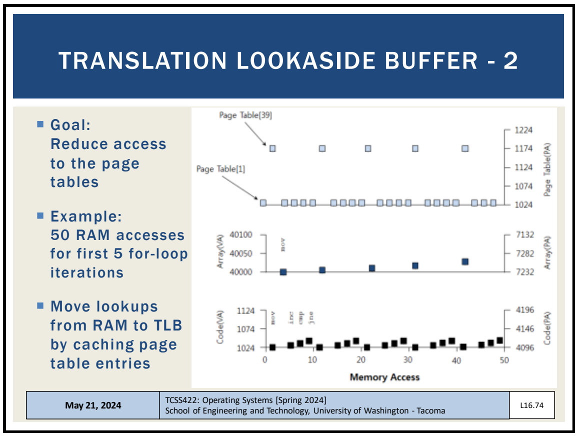
May 21, 2024 | TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma | L16.72

72



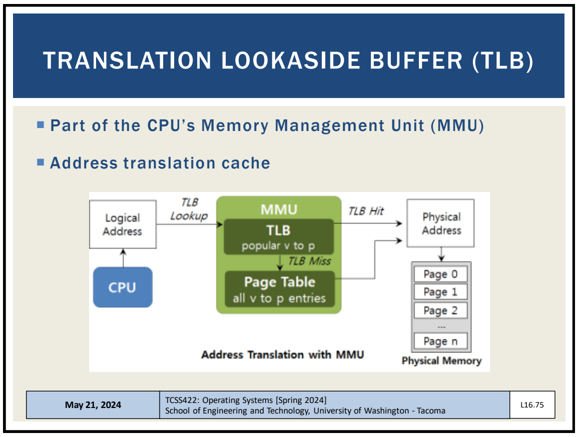
VISUALIZING MEMORY ACCESSES:
FOR THE FIRST 5 LOOP ITERATIONS

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



TRANSLATION LOOKASIDE BUFFER - 2

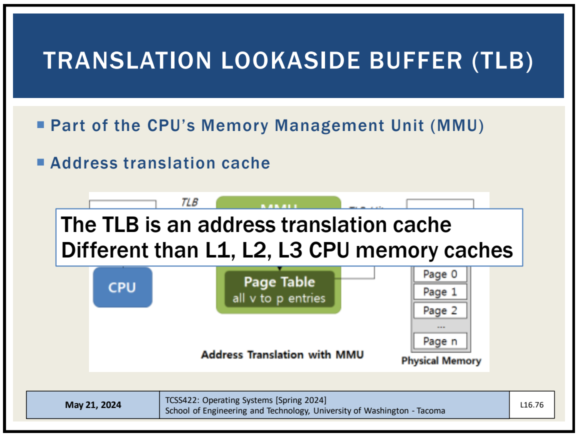
- Goal: Reduce access to the page tables
- Example: 50 RAM accesses for first 5 for-loop iterations
- Move lookups from RAM to TLB by caching page table entries



TRANSLATION LOOKASIDE BUFFER (TLB)

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

Address Translation with MMU

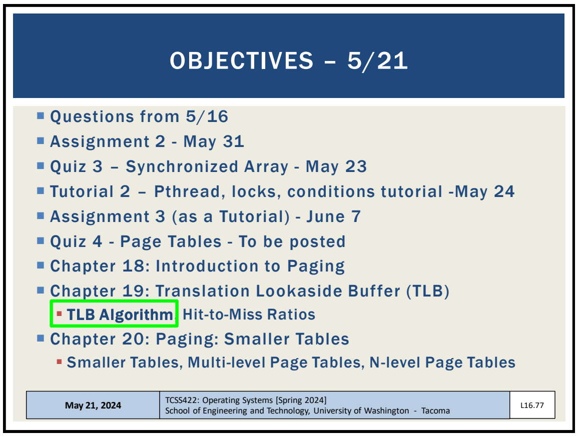


TRANSLATION LOOKASIDE BUFFER (TLB)

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

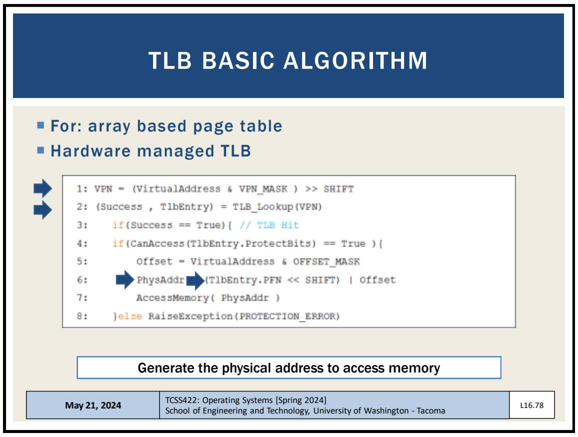
The TLB is an address translation cache
Different than L1, L2, L3 CPU memory caches

Address Translation with MMU



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

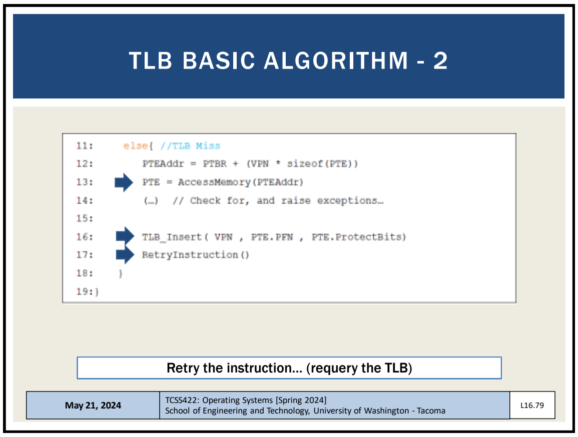


TLB BASIC ALGORITHM

- For: array based page table
- Hardware managed TLB

```
VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
(Success , TlbEntry) = TLB_Lookup(VPN)
   if(Success == True){ // TLB Hit
   if(CanAccess(TlbEntry.ProtectBits) == True ){
       Offset = VirtualAddress & OFFSET_MASK
       PhysAddr (TlbEntry.PFN << SHIFT) | Offset
       AccessMemory( PhysAddr )
   }else RaiseException(PROTECTION_ERROR)
```

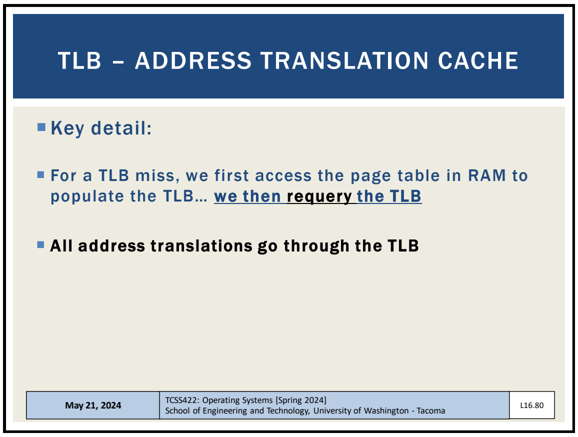
Generate the physical address to access memory



TLB BASIC ALGORITHM - 2

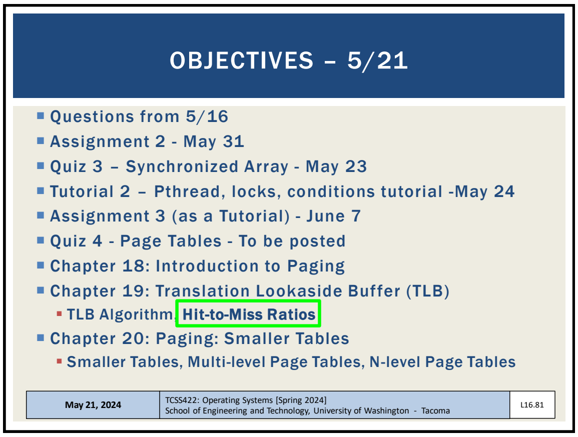
```
else{ //TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    (…)  // Check for, and raise exceptions…

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

Retry the instruction… (requery the TLB)



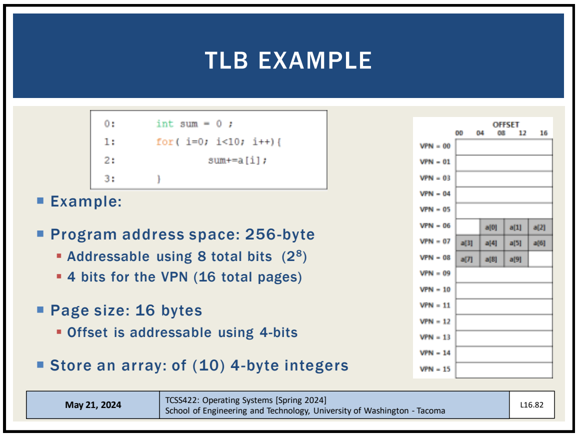
TLB – ADDRESS TRANSLATION CACHE

- 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**



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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



TLB EXAMPLE

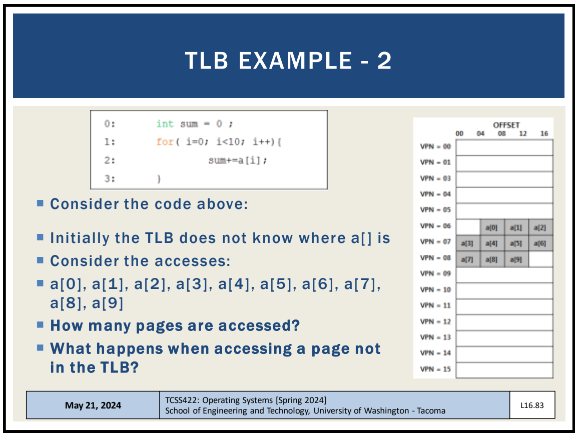
```
int sum = 0 ;
for( i=0; i<10; i++){
        sum+=a[i];
}
```

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

OFFSET

	00	04	08	12
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				

16



TLB EXAMPLE - 2

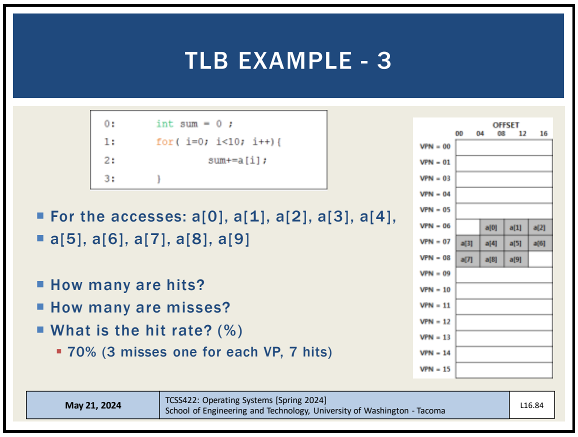
```
int sum = 0 ;
for( i=0; i<10; i++){
        sum+=a[i];
}
```

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

OFFSET

	00	04	08	12
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				

16



TLB EXAMPLE - 3

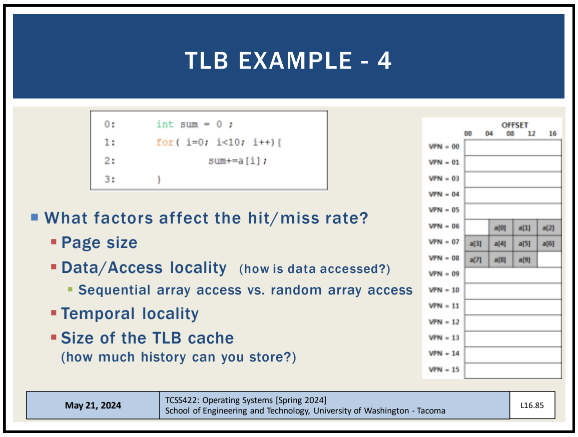
```
int sum = 0 ;
for( i=0; i<10; i++){
        sum+=a[i];
}
```

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

OFFSET

	00	04	08	12
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				

16



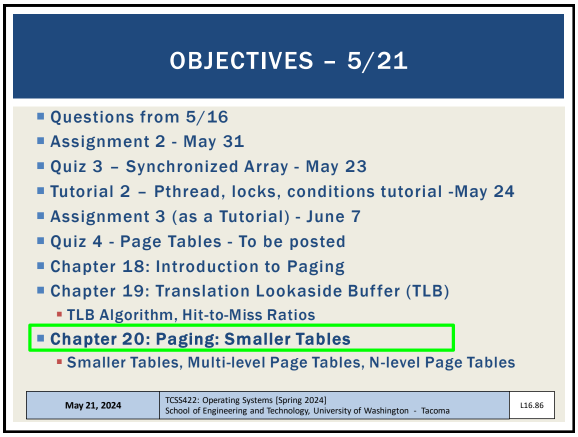
TLB EXAMPLE - 4

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

OFFSET
00 04 08 12 16

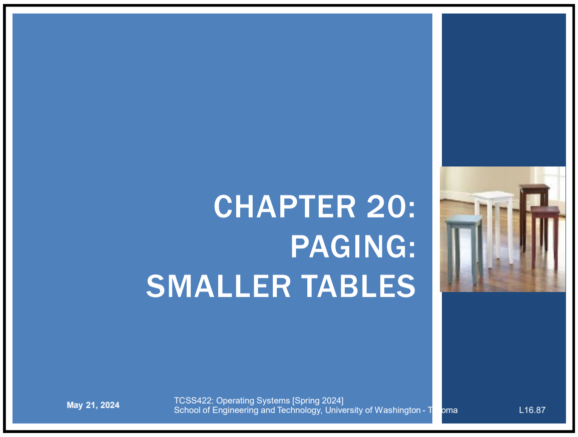
VPN = 00				
VPN = 01				
VPN = 03				
VPN = 04				
VPN = 05				
VPN = 06		a[0]	a[1]	a[2]
VPN = 07	a[3]	a[4]	a[5]	a[6]
VPN = 08	a[7]	a[8]	a[9]	
VPN = 09				
VPN = 10				
VPN = 11				
VPN = 12				
VPN = 13				
VPN = 14				
VPN = 15				

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

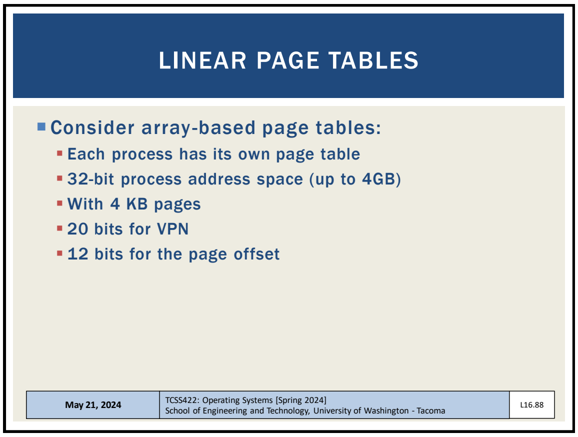


OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

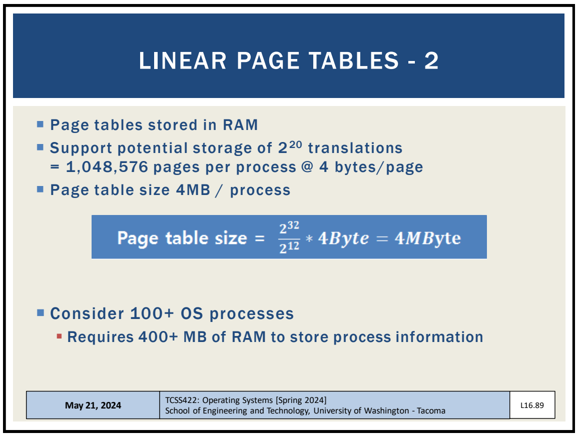


CHAPTER 20: PAGING: SMALLER TABLES



LINEAR PAGE TABLES

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

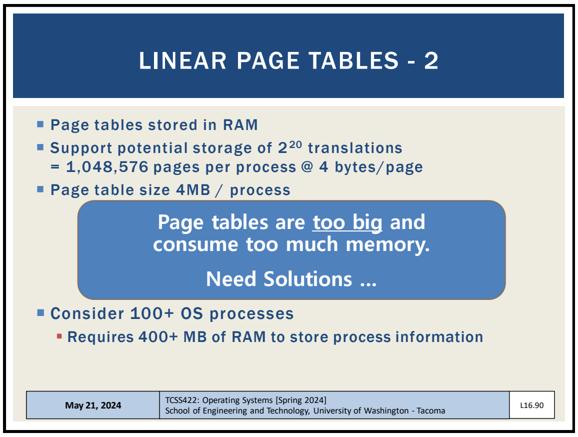


LINEAR PAGE TABLES - 2

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

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

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



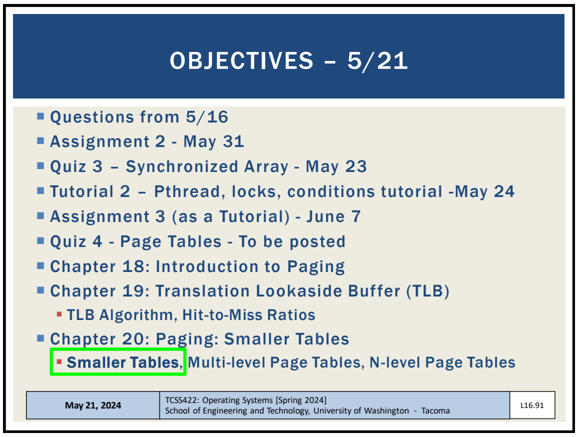
LINEAR PAGE TABLES - 2

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

Page tables are <u>too big</u> and consume too much memory.

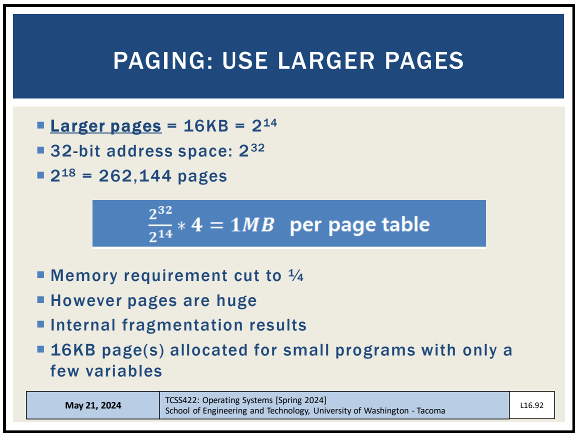
Need Solutions …

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



OBJECTIVES – 5/21

- Questions from 5/16
- Assignment 2 - May 31
- Quiz 3 – Synchronized Array - May 23
- Tutorial 2 – Pthread, locks, conditions tutorial -May 24
- Assignment 3 (as a Tutorial) - June 7
- 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

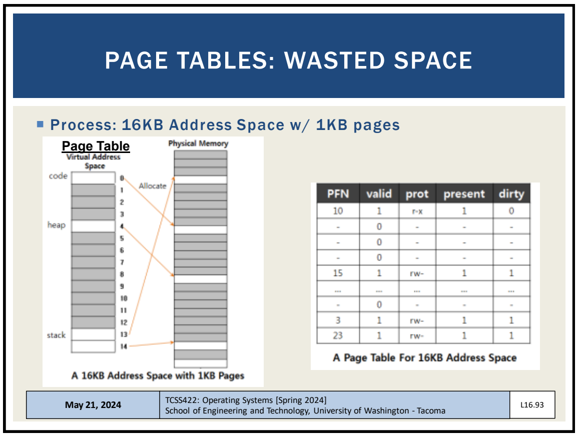


PAGING: USE LARGER PAGES

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



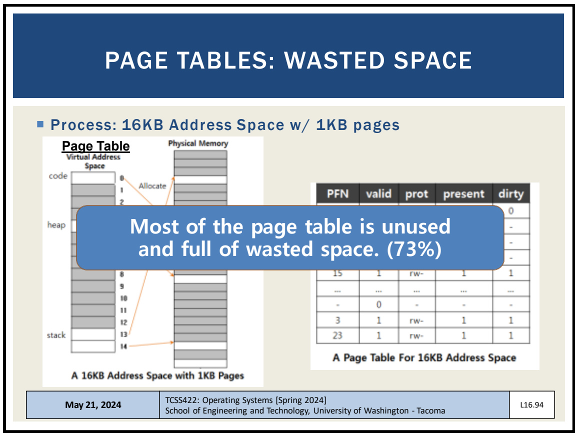
PAGE TABLES: WASTED SPACE

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

A 16KB Address Space with 1KB Pages

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

A Page Table For 16KB Address Space



PAGE TABLES: WASTED SPACE

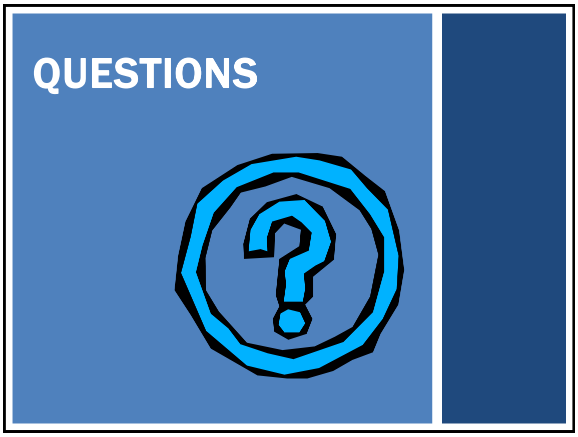
- Process: 16KB Address Space w/ 1KB pages

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

A 16KB Address Space with 1KB Pages



QUESTIONS