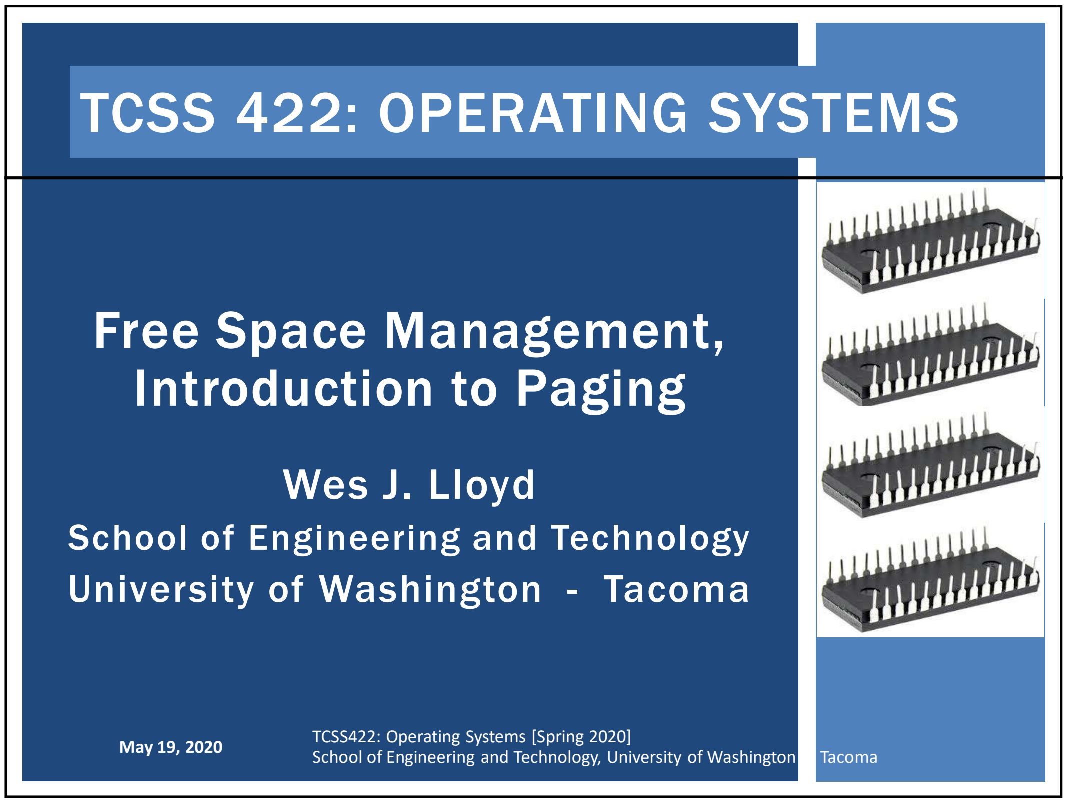
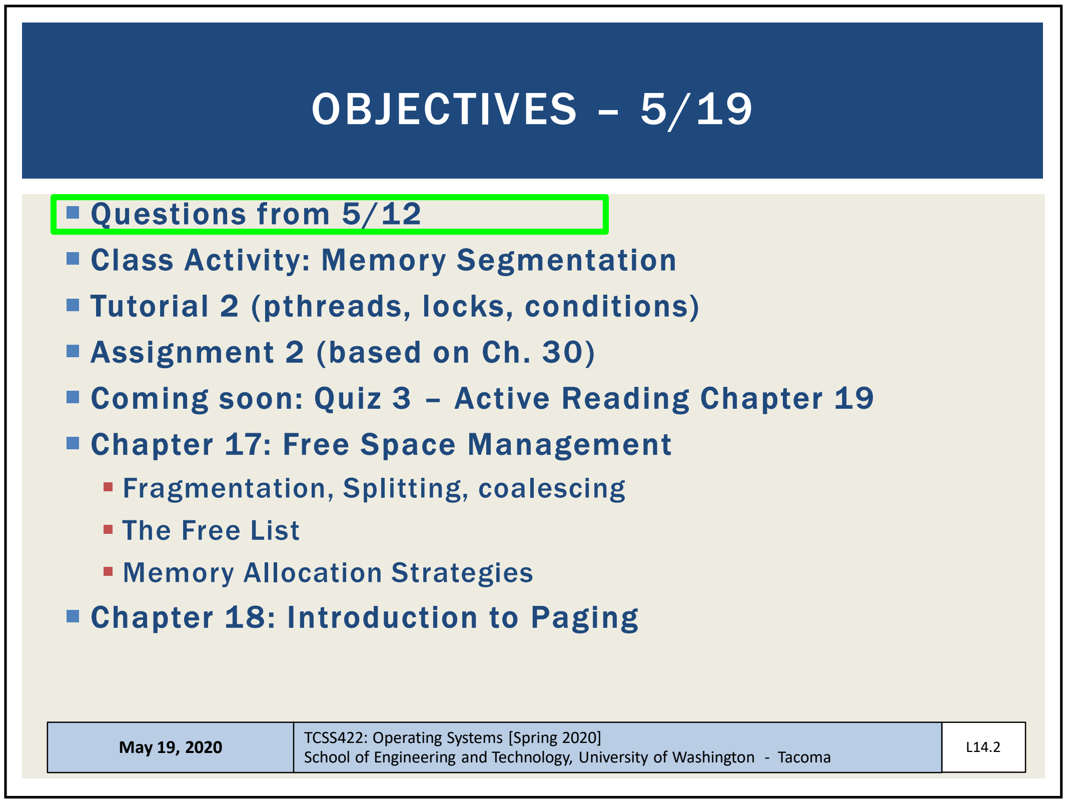


TCSS 422: OPERATING SYSTEMS

Free Space Management, Introduction to Paging

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

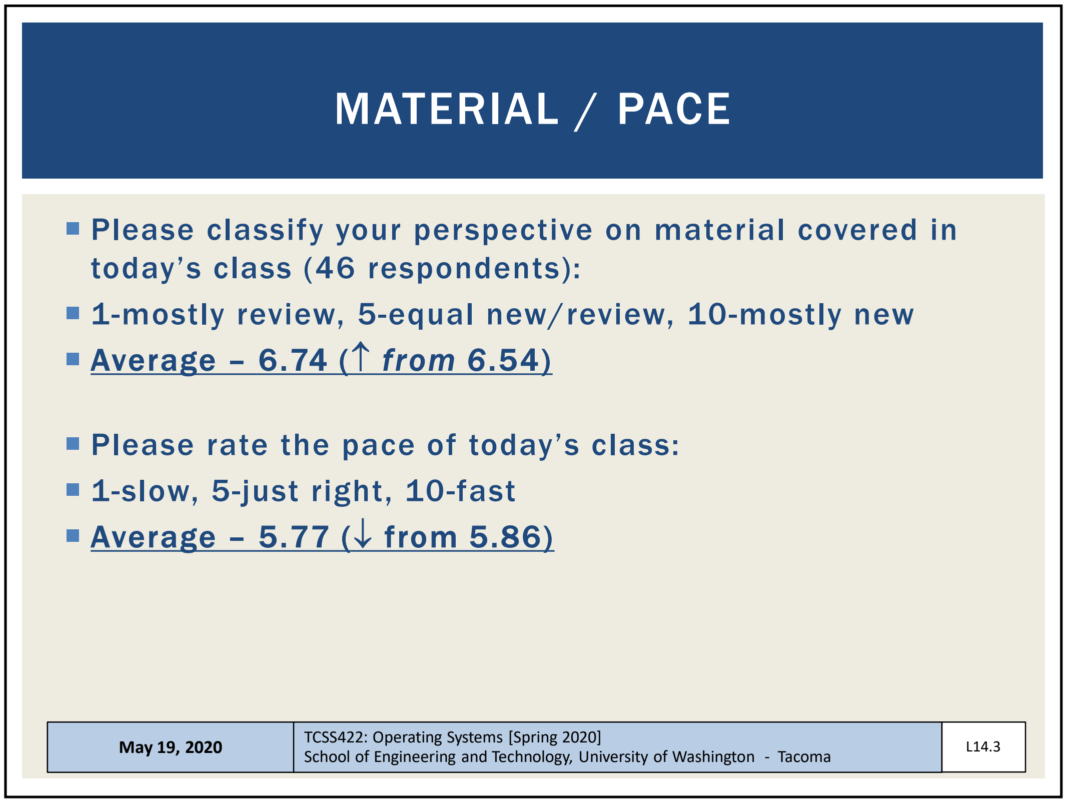
May 19, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma



OBJECTIVES – 5/19

- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- Chapter 18: Introduction to Paging

May 19, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L14.2

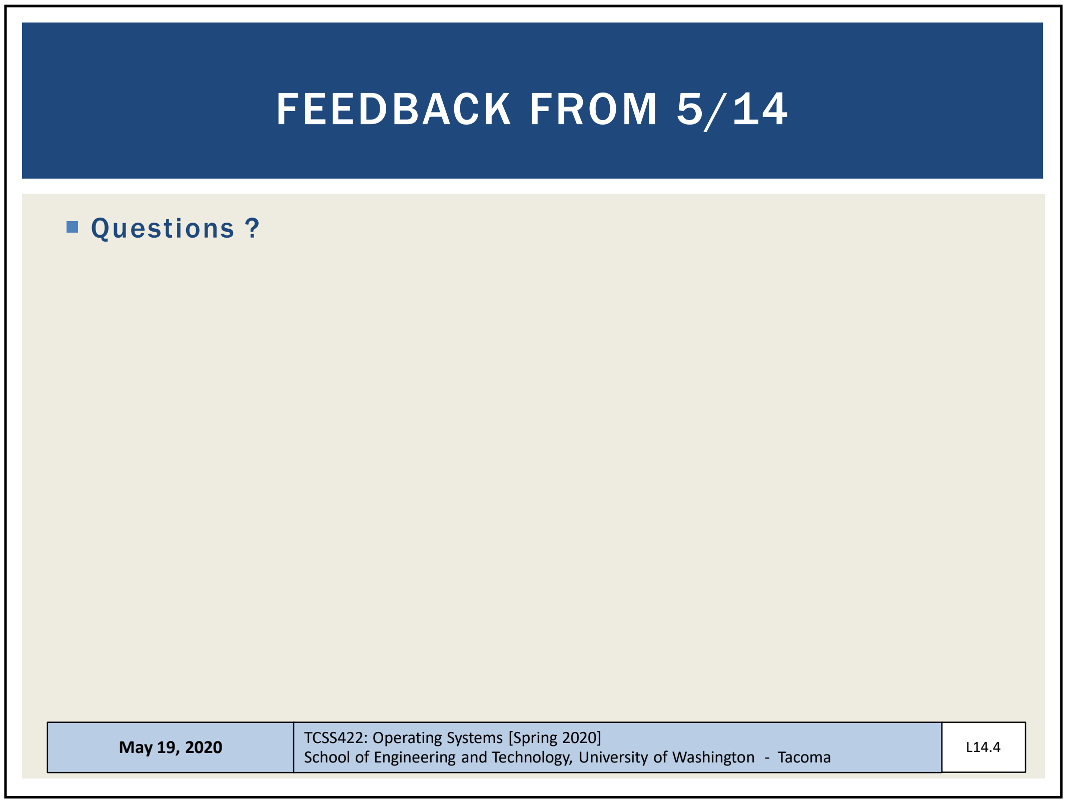


MATERIAL / PACE

- Please classify your perspective on material covered in today's class (46 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.74 (↑ *from* 6.54)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.77 (↓ from 5.86)

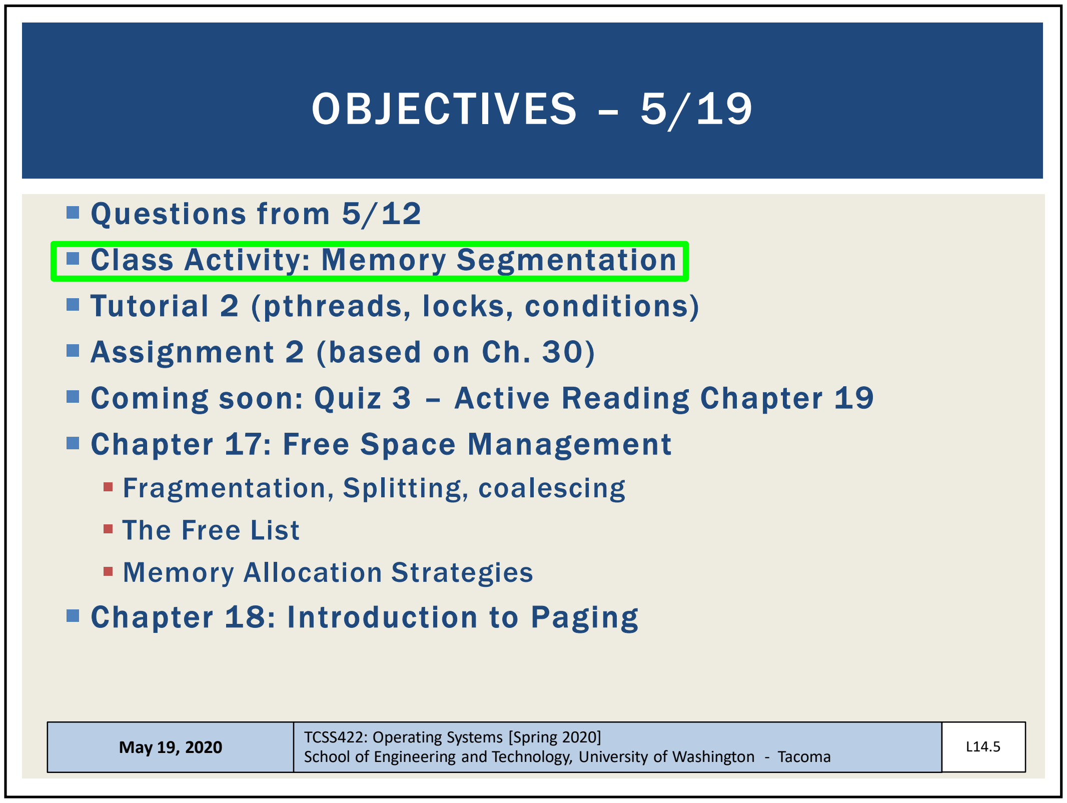
May 19, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L14.3



FEEDBACK FROM 5/14

- Questions ?

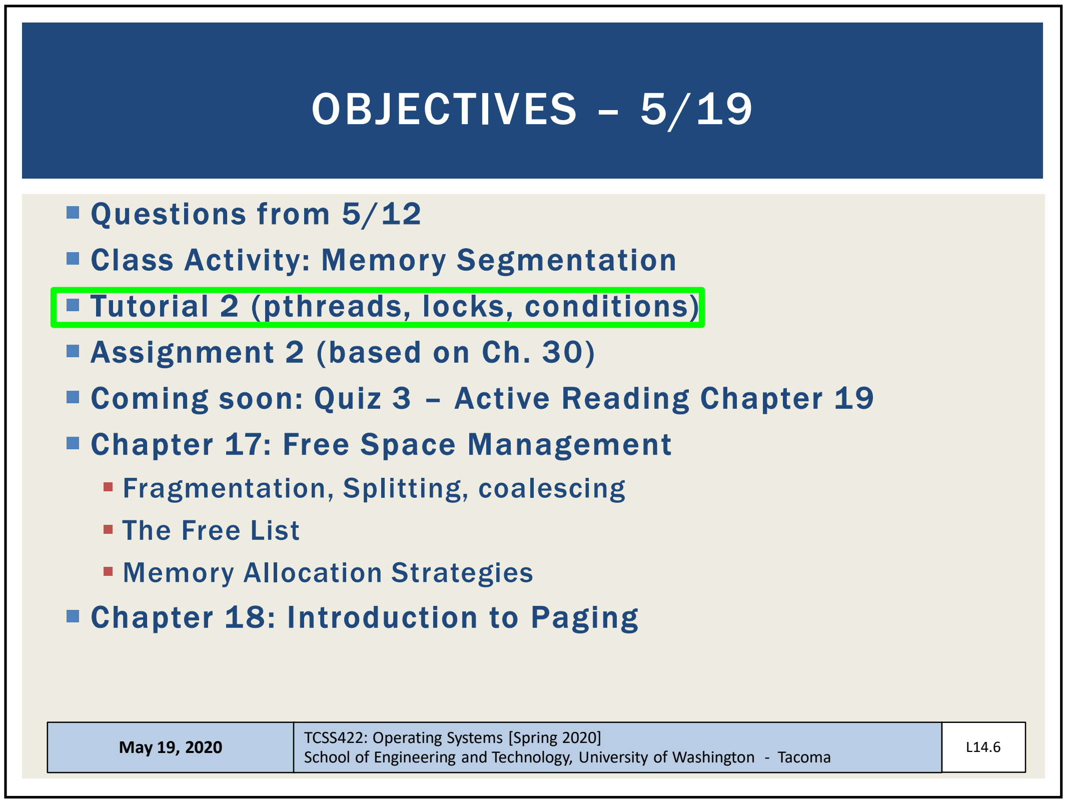
May 19, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L14.4



OBJECTIVES – 5/19

- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- Chapter 18: Introduction to Paging

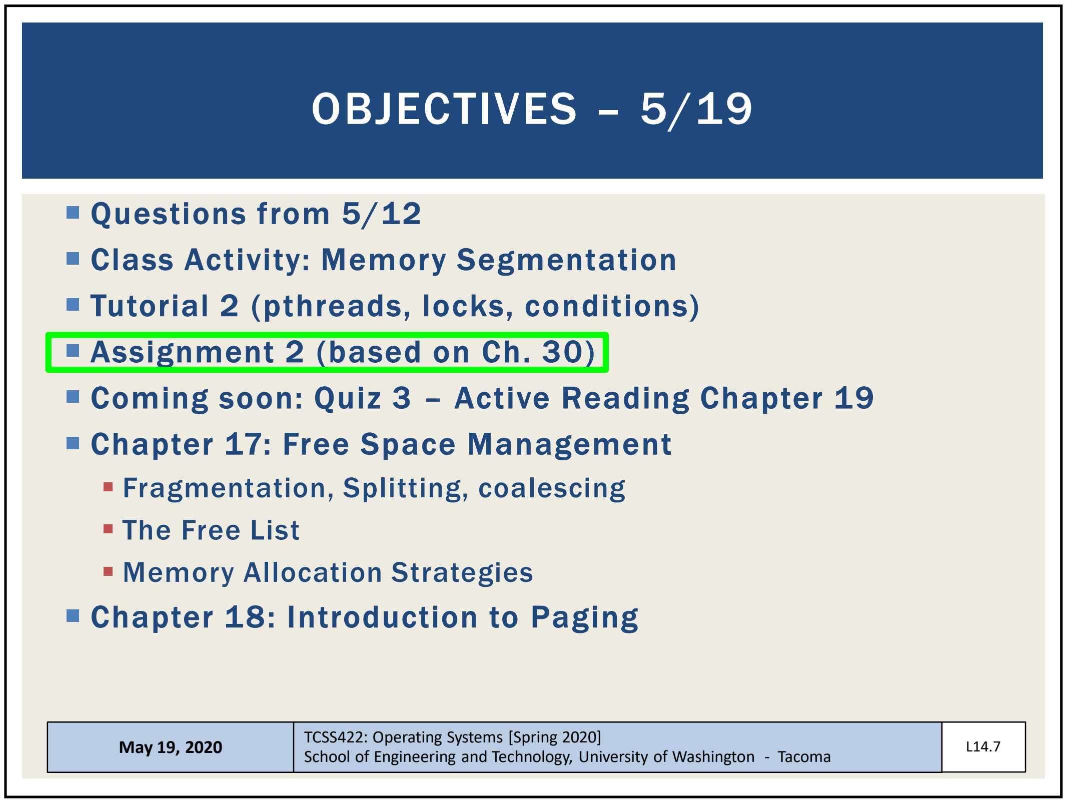
May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.5



OBJECTIVES – 5/19

- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- Chapter 18: Introduction to Paging

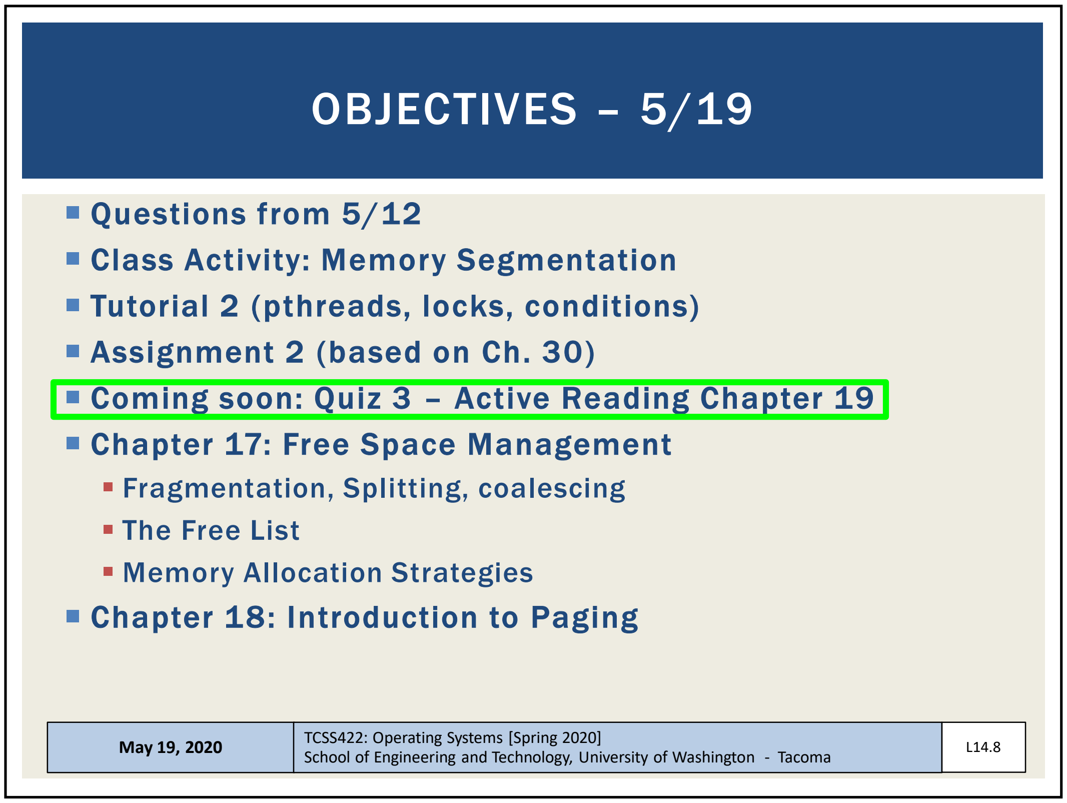
May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.6



OBJECTIVES – 5/19

- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- Chapter 18: Introduction to Paging

May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.7

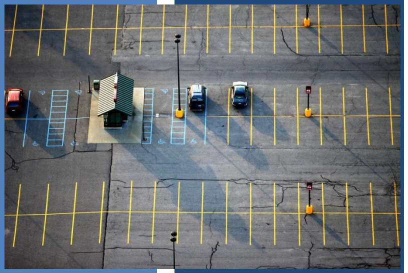


OBJECTIVES – 5/19

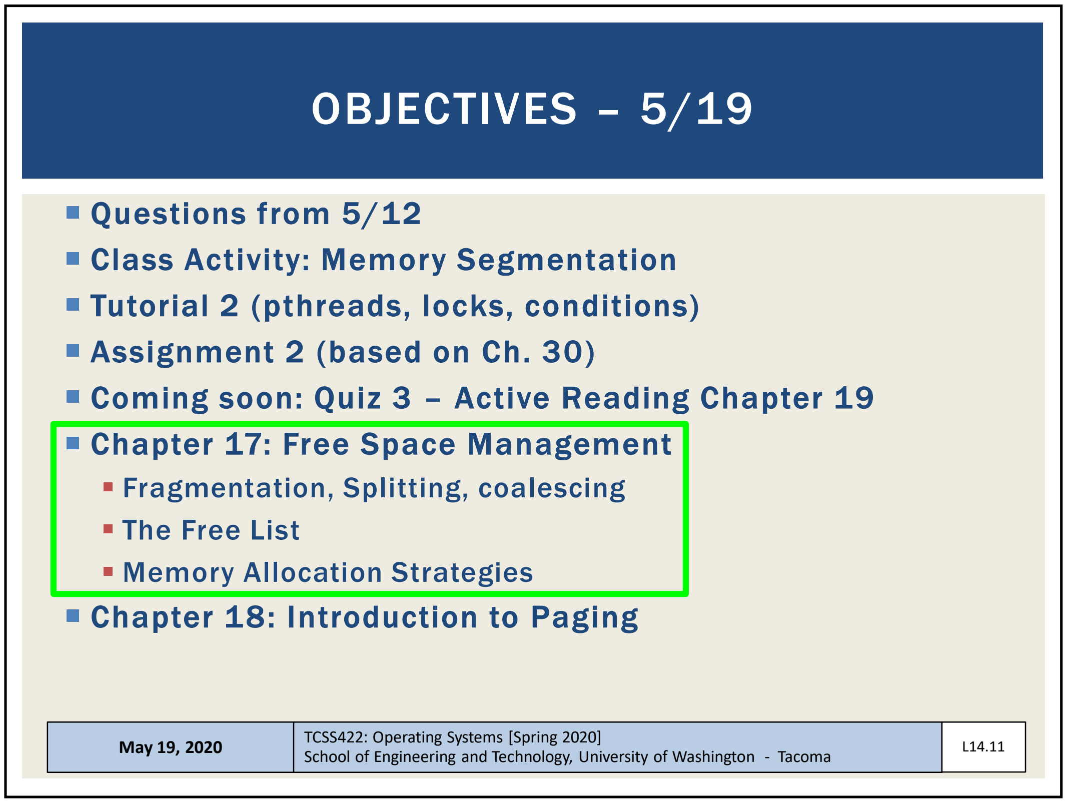
- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- Chapter 18: Introduction to Paging

May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.8

TCSS 422 WILL RETURN AT ~2:40PM

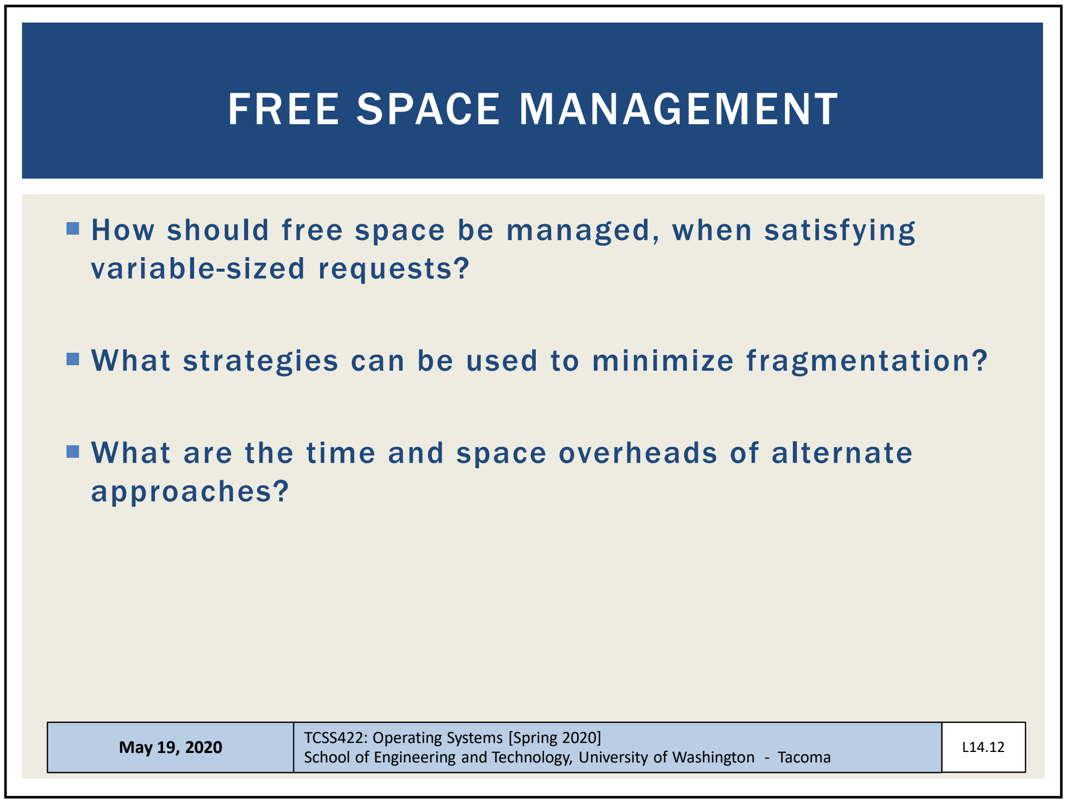


CHAPTER 17: FREE SPACE MANAGEMENT



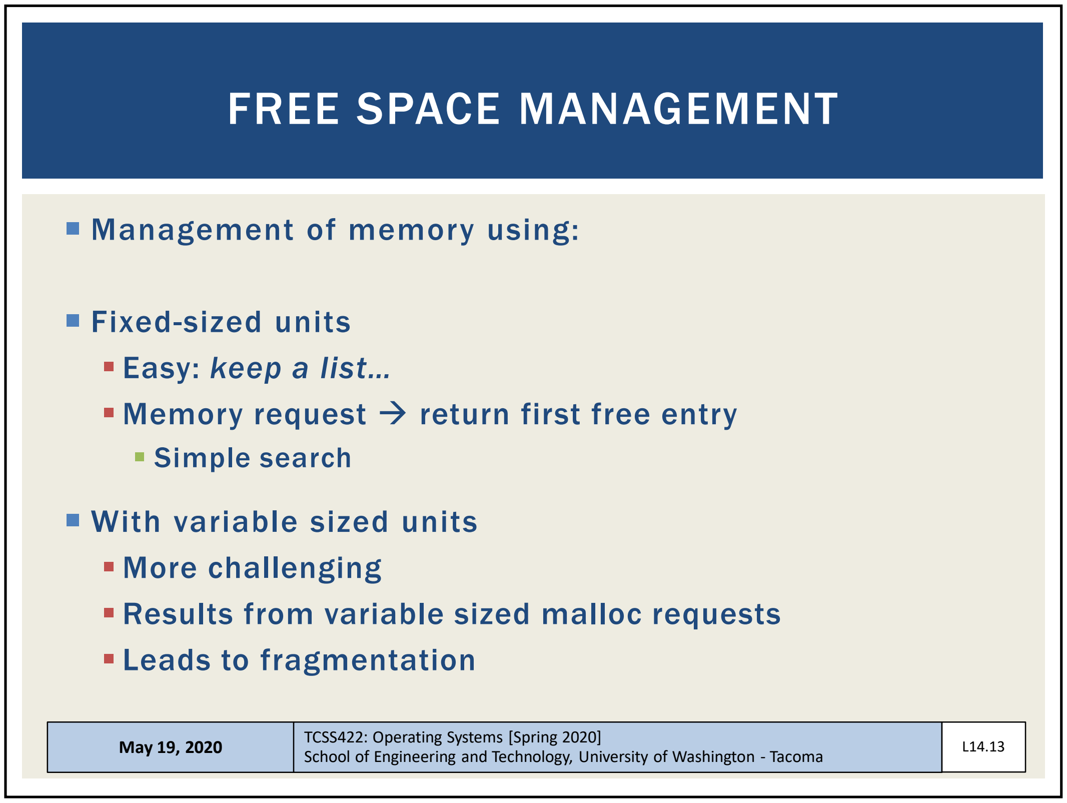
OBJECTIVES – 5/19

- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- Chapter 18: Introduction to Paging



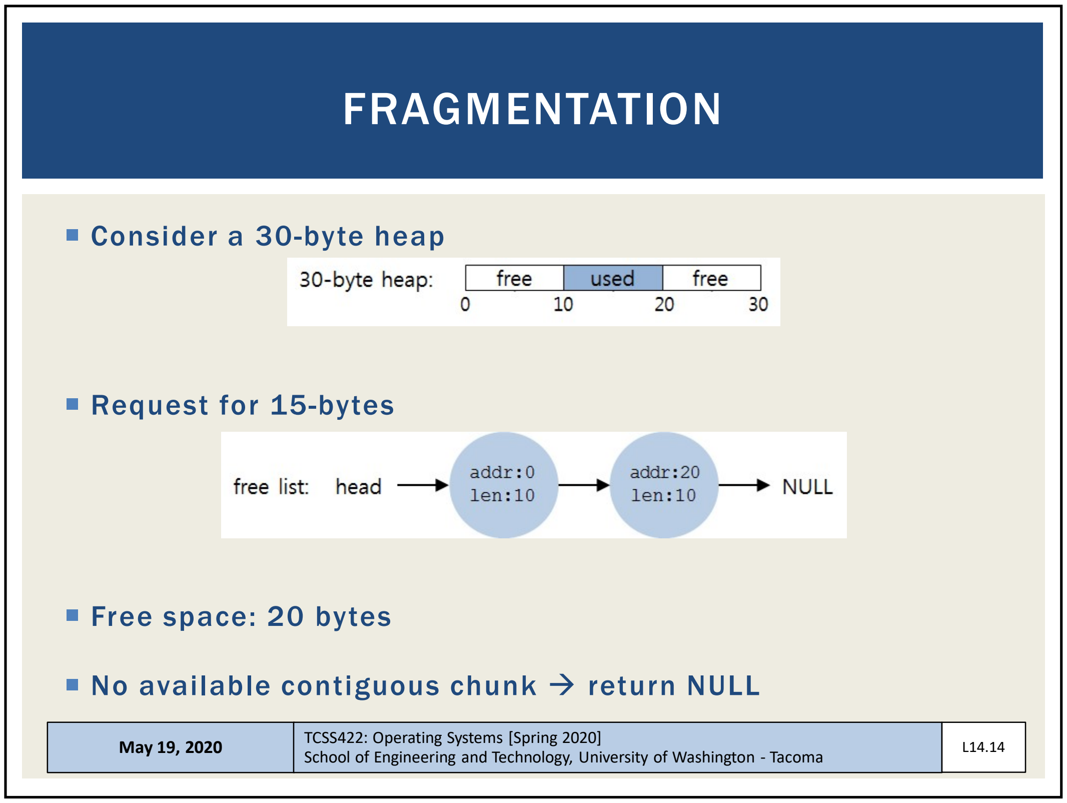
FREE SPACE MANAGEMENT

- 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?



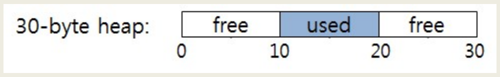
FREE SPACE MANAGEMENT

- Management of memory using:
- 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

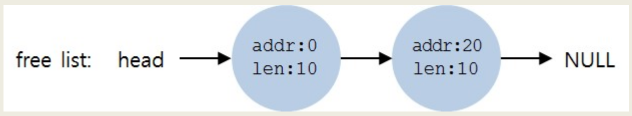


FRAGMENTATION

- Consider a 30-byte heap



- Request for 15-bytes



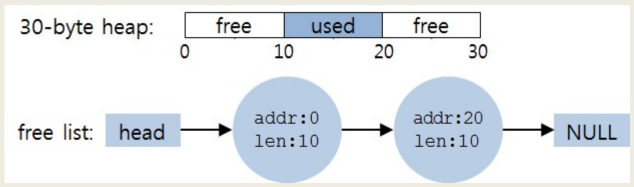
- Free space: 20 bytes
- No available contiguous chunk → return NULL

FRAGMENTATION - 2

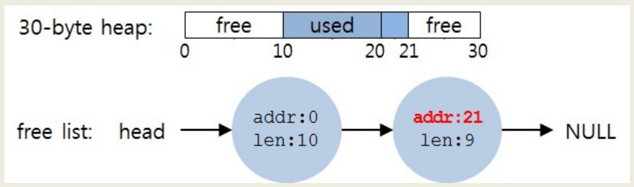
- **External:** *OS can compact*
  - Example: Client asks for 100 bytes: malloc(100)
  - OS: No 100 byte contiguous chunk is available: returns NULL
  - Memory is externally fragmented - - Compaction can fix!
- **Internal:** *lost space – OS can't compact*
  - OS returns memory units that are too large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

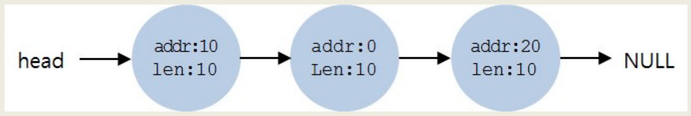


- OS locates a free chunk to satisfy request
- Splits chunk into two, returns first chunk

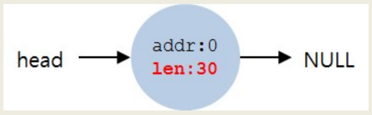


ALLOCATION STRATEGY: COALESCING

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



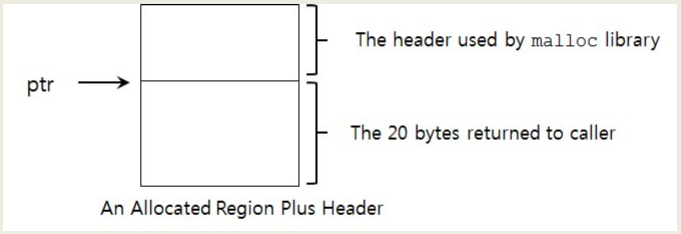
- Request arrives: malloc(30)
- ***SPLIT DOES NOT WORK*** - no contiguous 30-byte chunk exists!
- Coalescing regroups chunks into contiguous chunk



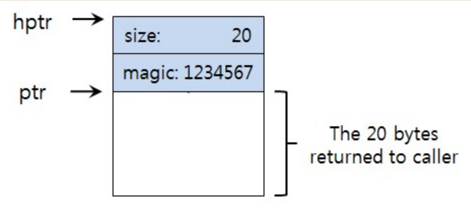
- Allocation can now proceed
- Coalescing is defragmentation of the free space list

MEMORY HEADERS

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


ptr
The header used by malloc library
The 20 bytes returned to caller
An Allocated Region Plus Header

MEMORY HEADERS - 2



Specific Contents Of The Header

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

A Simple Header

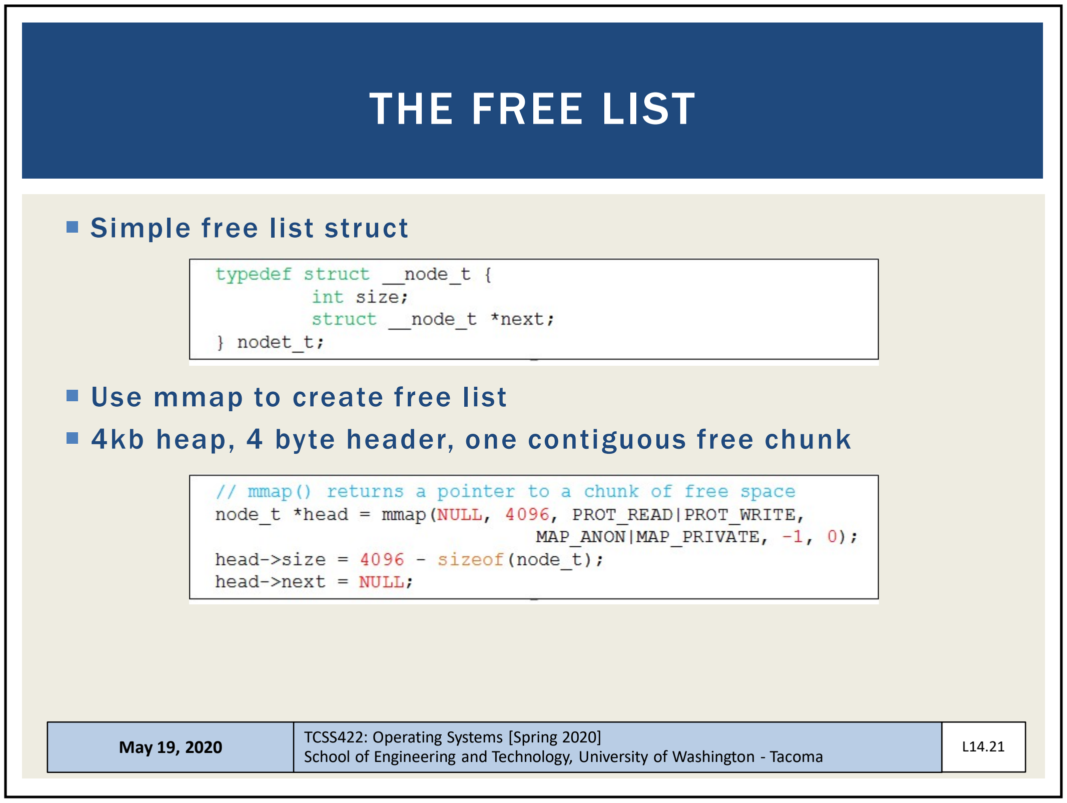
- Contains size
- Pointers: for faster memory access
- Magic number: integrity checking

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

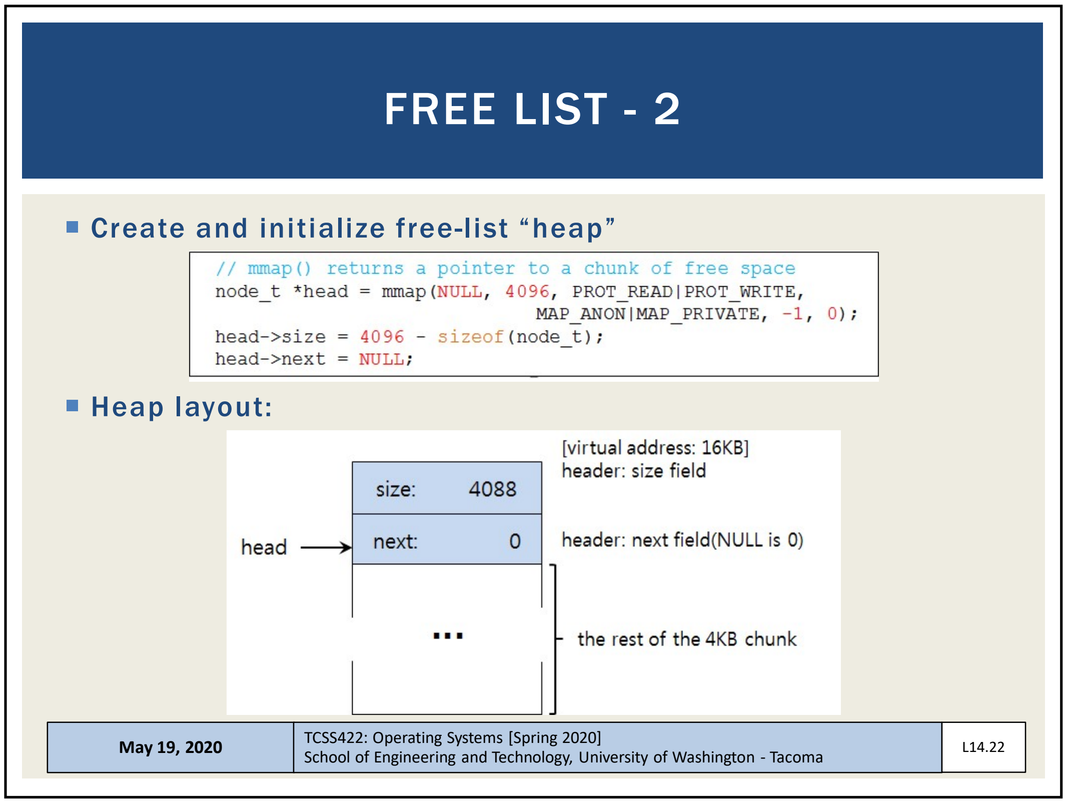
- Simple free list struct

```
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;
```

May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.21

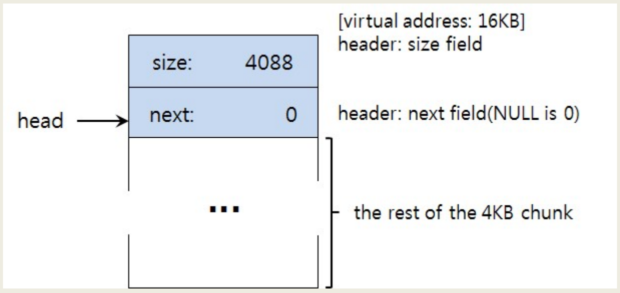


FREE LIST - 2

- 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;
```

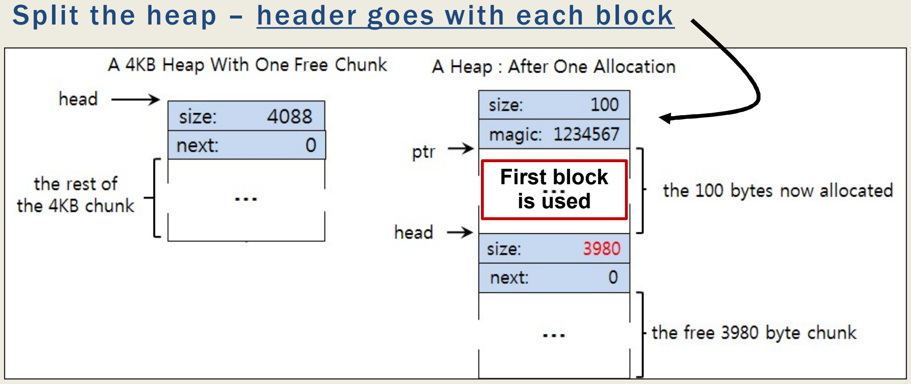
- Heap layout:



May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.22

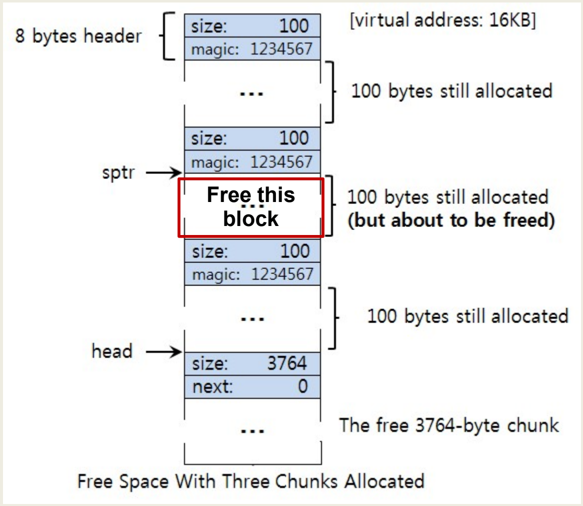
FREE LIST: MALLOC() CALL

- 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



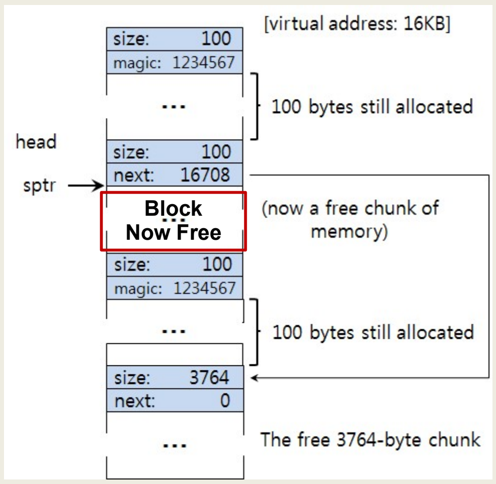
FREE LIST: FREE() CALL

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



FREE LIST: FREE() CHUNK #2

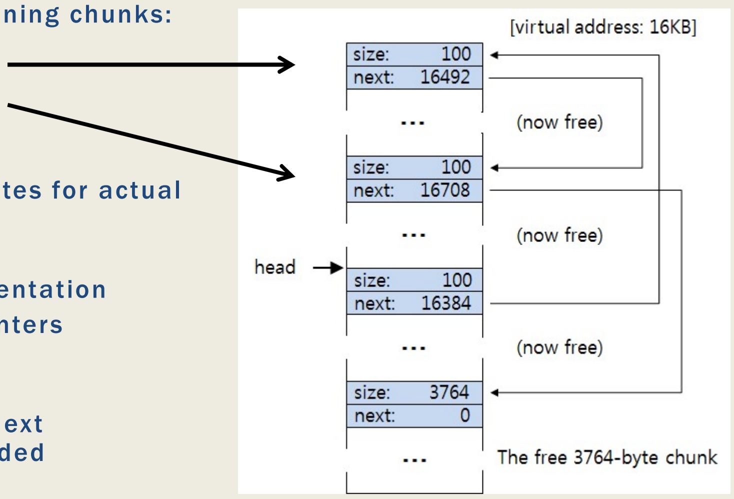
- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
  - addr – sizeof(node_t)
- Actual start of chunk #2
  - 16492



May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.25

FREE LIST- FREE ALL CHUNKS

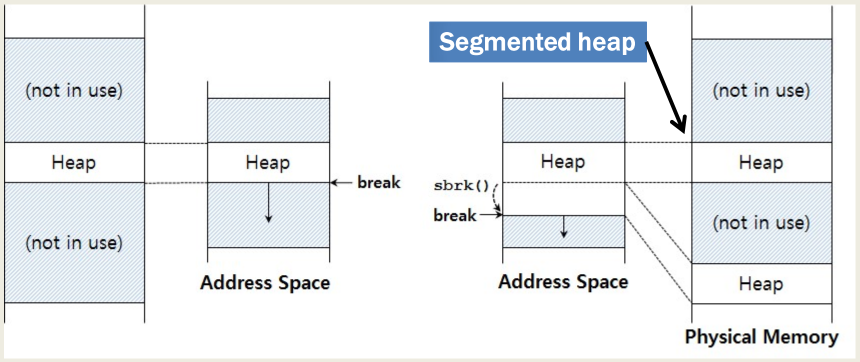
- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed



May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.26

GROWING THE HEAP

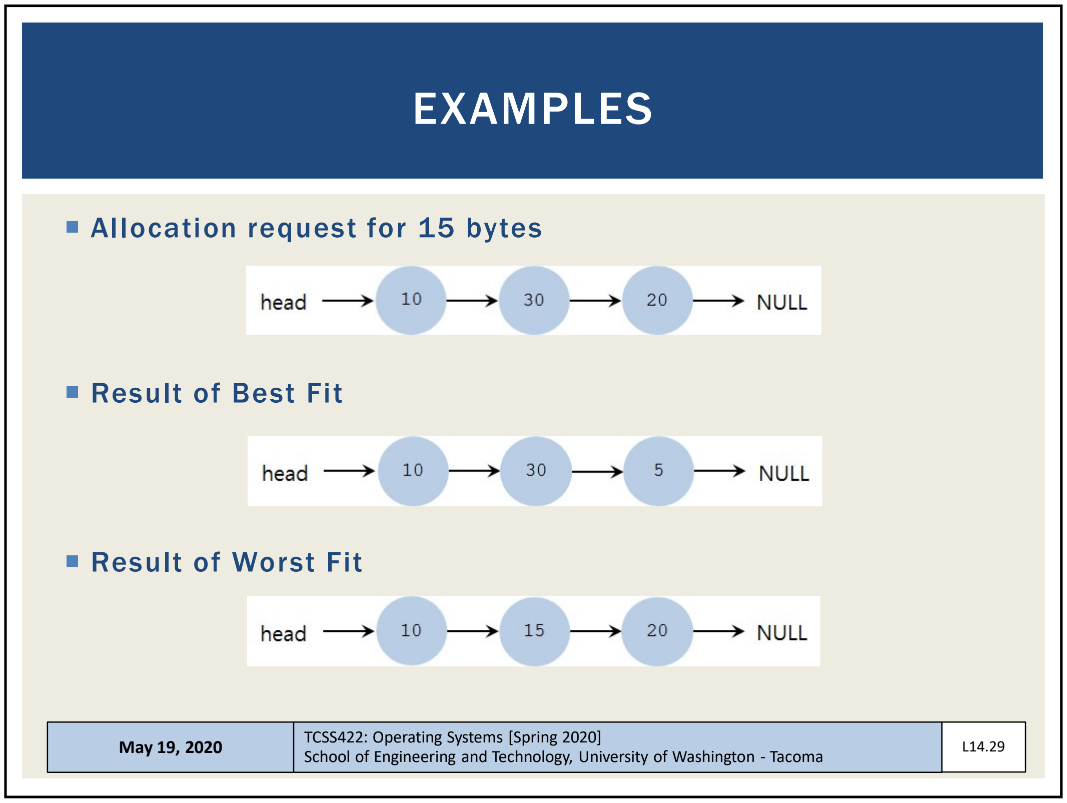
- Start with small sized heap
- Request more memory when full
- sbrk(), brk()



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

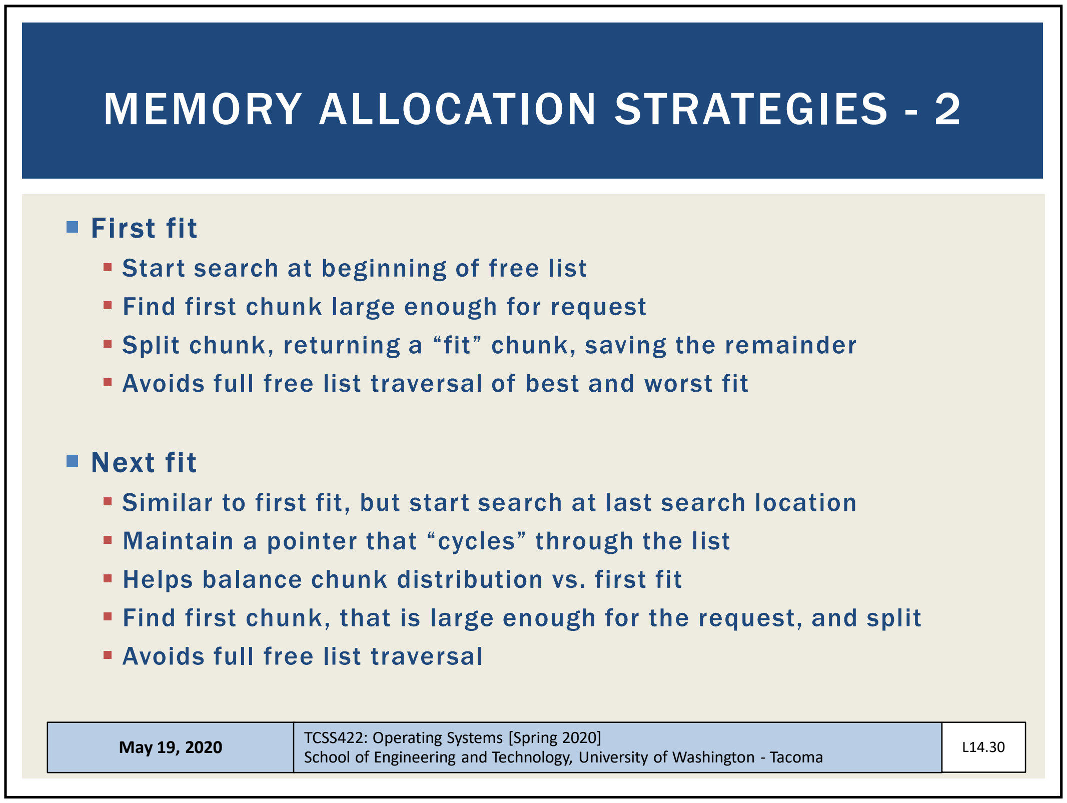
$$\begin{tikzcd} \text{head} \arrow[r] & 10 \arrow[r] & 30 \arrow[r] & 20 \arrow[r] & \text{NULL} \end{tikzcd}$$

- Result of Best Fit

$$\begin{tikzcd} \text{head} \arrow[r] & 10 \arrow[r] & 30 \arrow[r] & 5 \arrow[r] & \text{NULL} \end{tikzcd}$$

- Result of Worst Fit

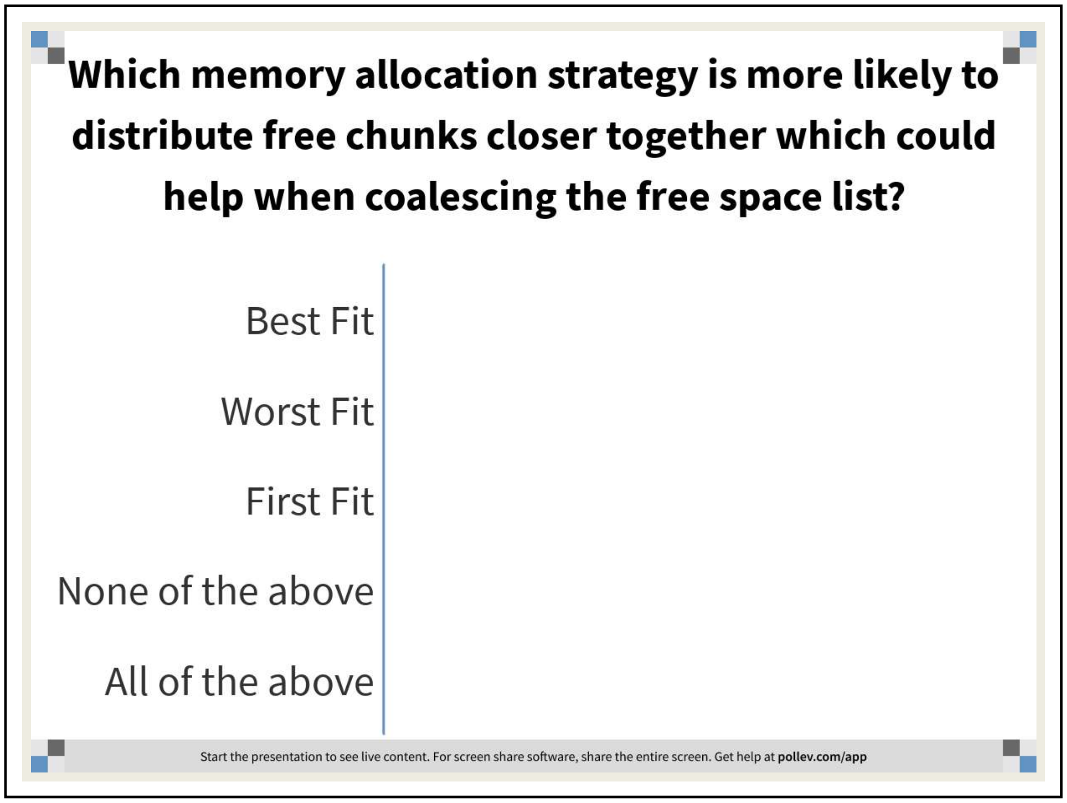
$$\begin{tikzcd} \text{head} \arrow[r] & 10 \arrow[r] & 15 \arrow[r] & 20 \arrow[r] & \text{NULL} \end{tikzcd}$$



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

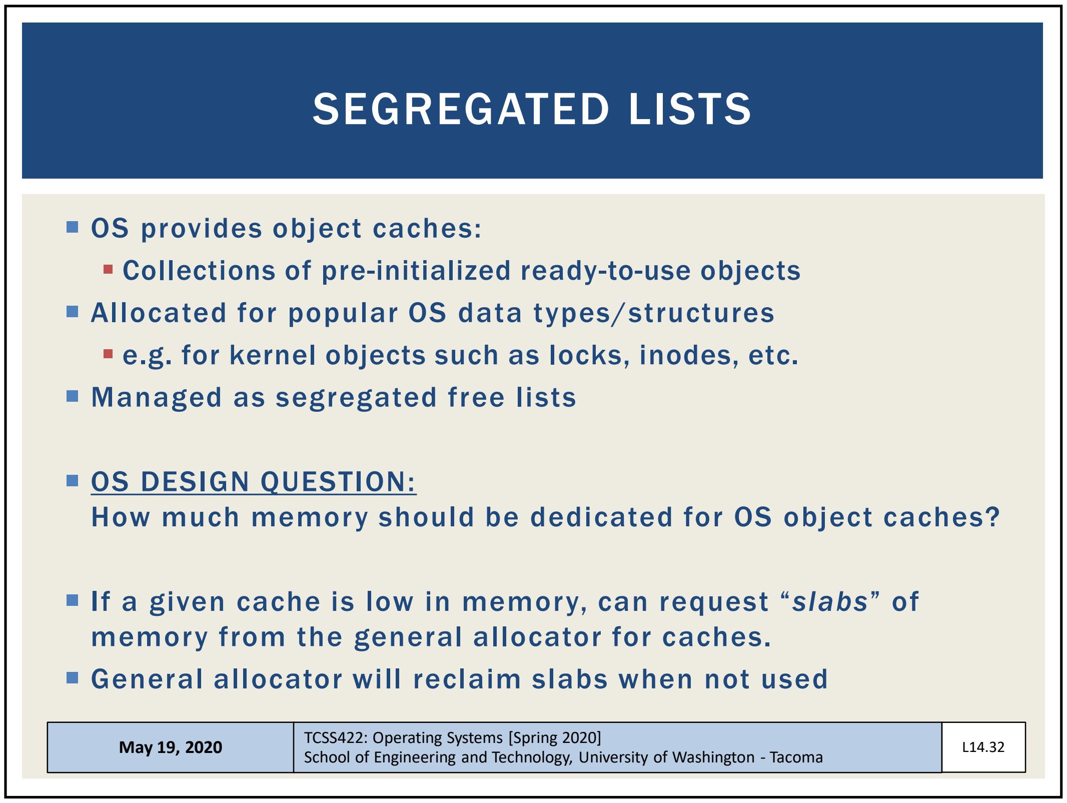
- Next fit
  - Similar to first fit, but start search at last search location
  - Maintain a pointer that "cycles" through the list
  - Helps balance chunk distribution vs. first fit
  - Find first chunk, that is large enough for the request, and split
  - Avoids full free list traversal



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**



SEGREGATED LISTS

- OS provides object caches:
  - Collections of pre-initialized ready-to-use objects
- Allocated for popular OS data types/structures
  - e.g. for kernel objects such as locks, inodes, etc.
- Managed as segregated free lists

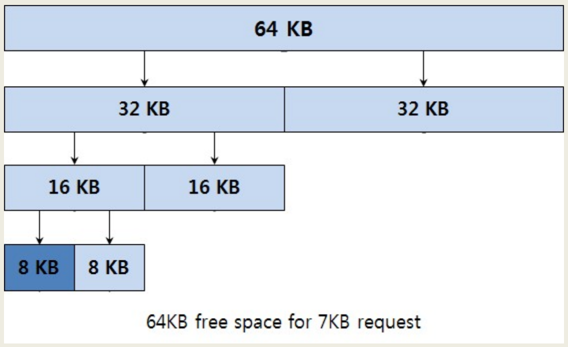
- OS DESIGN QUESTION:
  How much memory should be dedicated for OS object caches?

- 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 19, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L14.32

BUDDY ALLOCATION

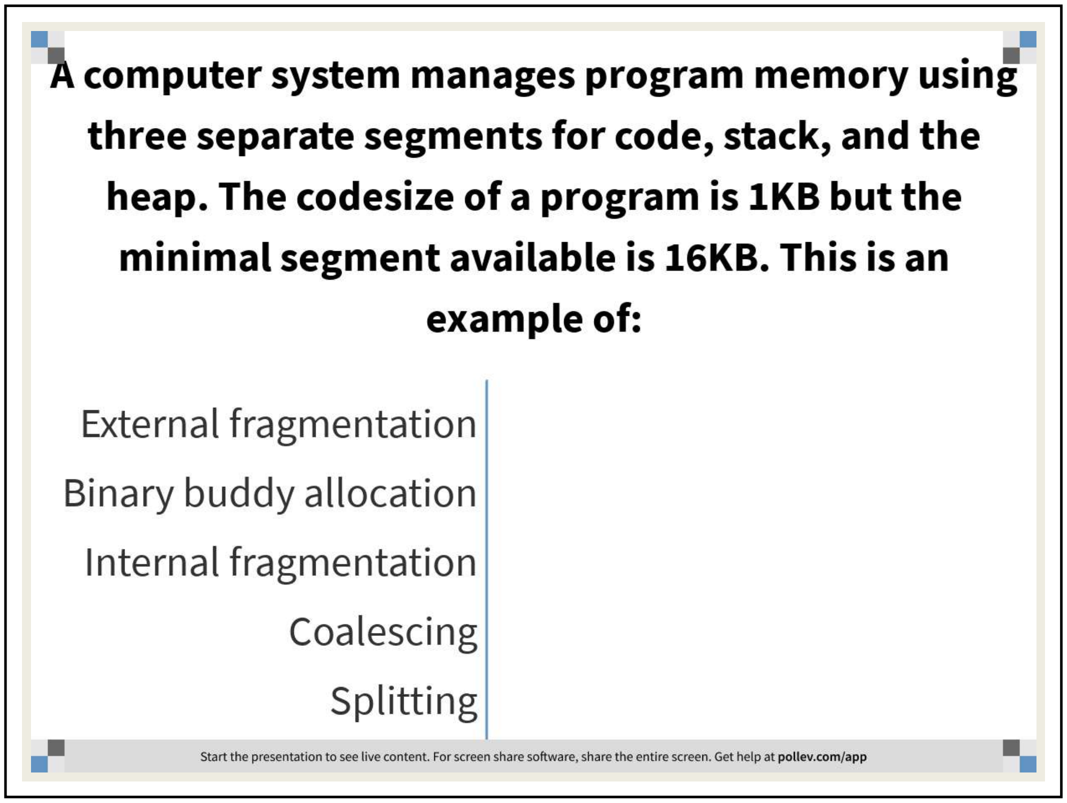
- Binary buddy allocation
  - Divides free space by two to find a block that is big enough to accommodate the request; the next split is too small…
- Consider a 7KB request



64KB free space for 7KB request

BUDDY ALLOCATION - 2

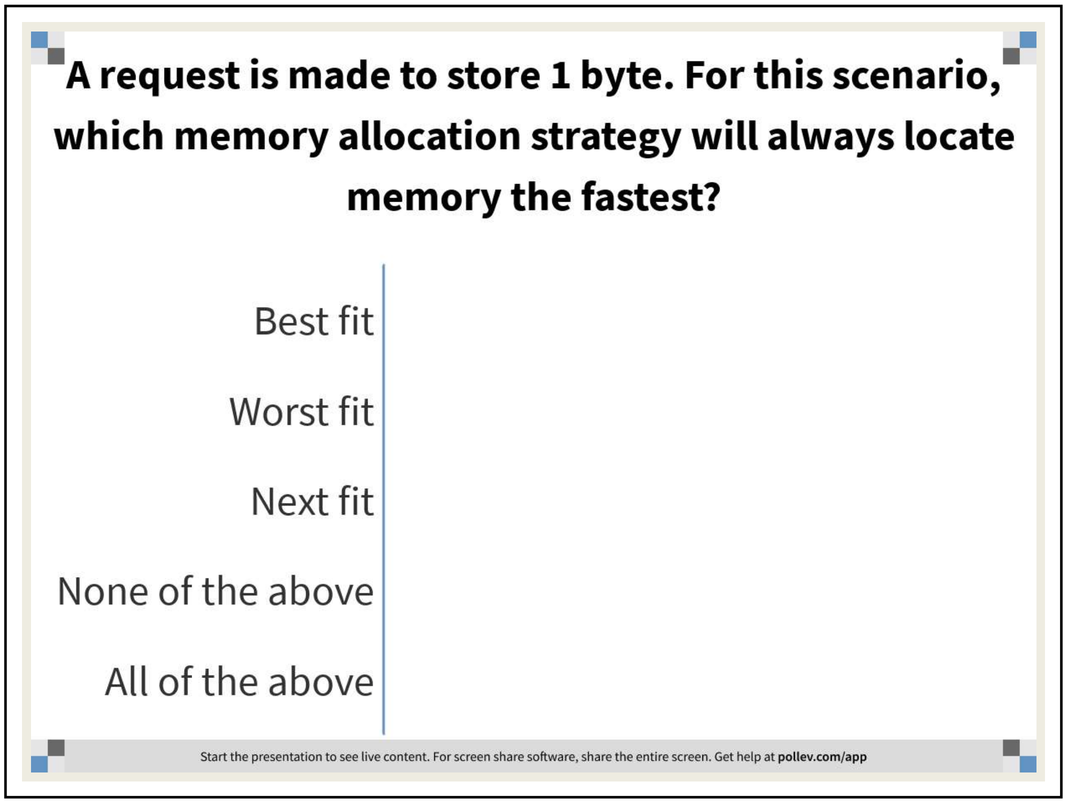
- Buddy allocation: suffers from internal fragmentation
- Allocated fragments, typically too large
- Coalescing is simple
  - Two adjacent blocks are promoted up



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**



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

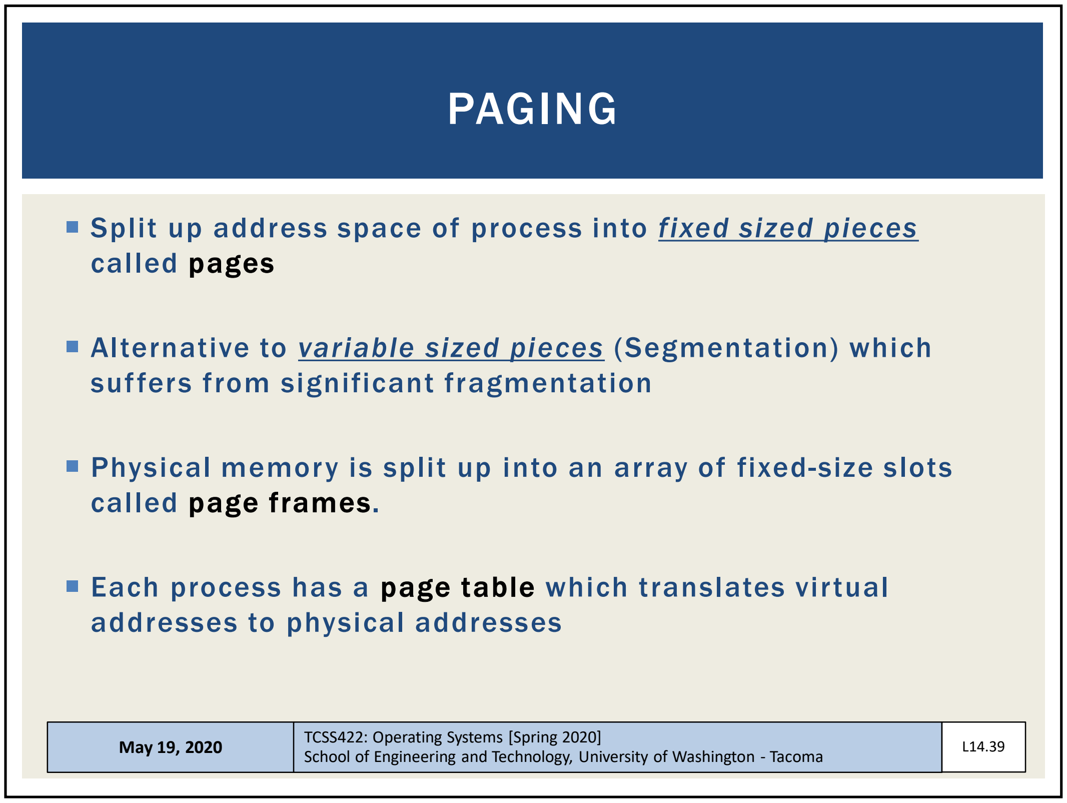
Best fit
Worst fit
Next fit
None of the above
All of the above

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

CHAPTER 18: INTRODUCTION TO PAGING

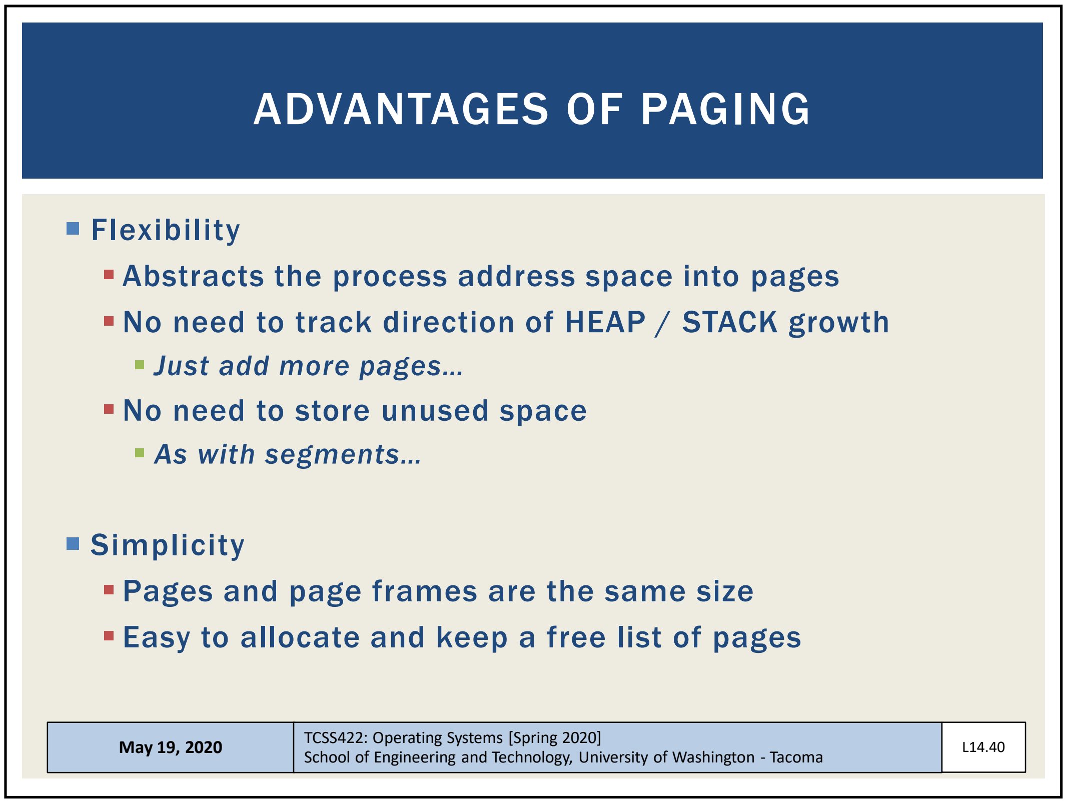
OBJECTIVES – 5/19

- Questions from 5/12
- Class Activity: Memory Segmentation
- Tutorial 2 (pthreads, locks, conditions)
- Assignment 2 (based on Ch. 30)
- Coming soon: Quiz 3 – Active Reading Chapter 19
- Chapter 17: Free Space Management
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies
- 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



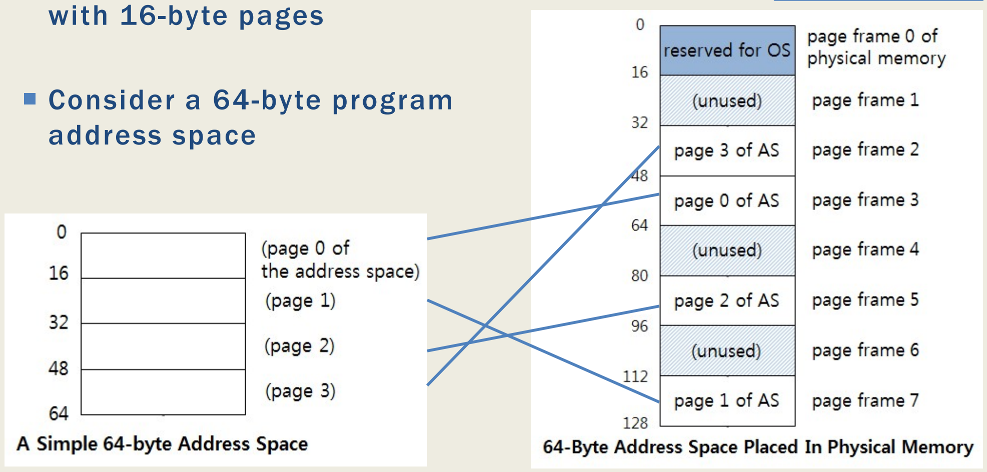
ADVANTAGES OF PAGING

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

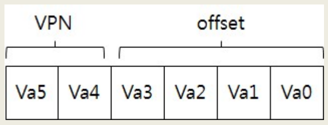


A Simple 64-byte Address Space

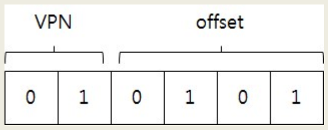
64-Byte Address Space Placed In Physical Memory

PAGING: ADDRESS TRANSLATION

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



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



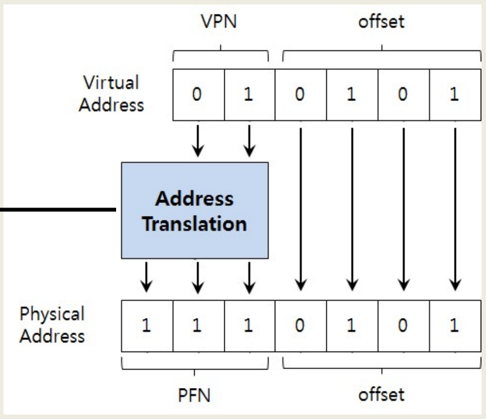
Here there are just four pages...

EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte program address space (4 pages)
- Stored in 128-byte physical memory (8 frames)

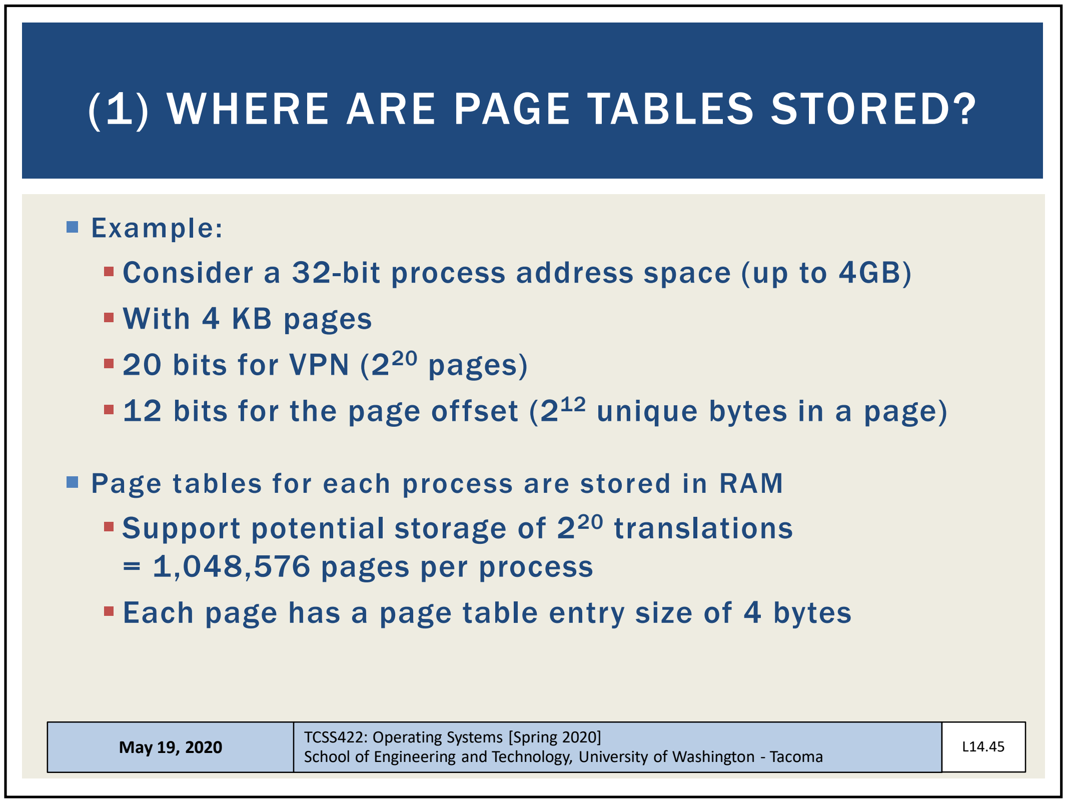
- Offset is preserved
- VPN is looked up

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



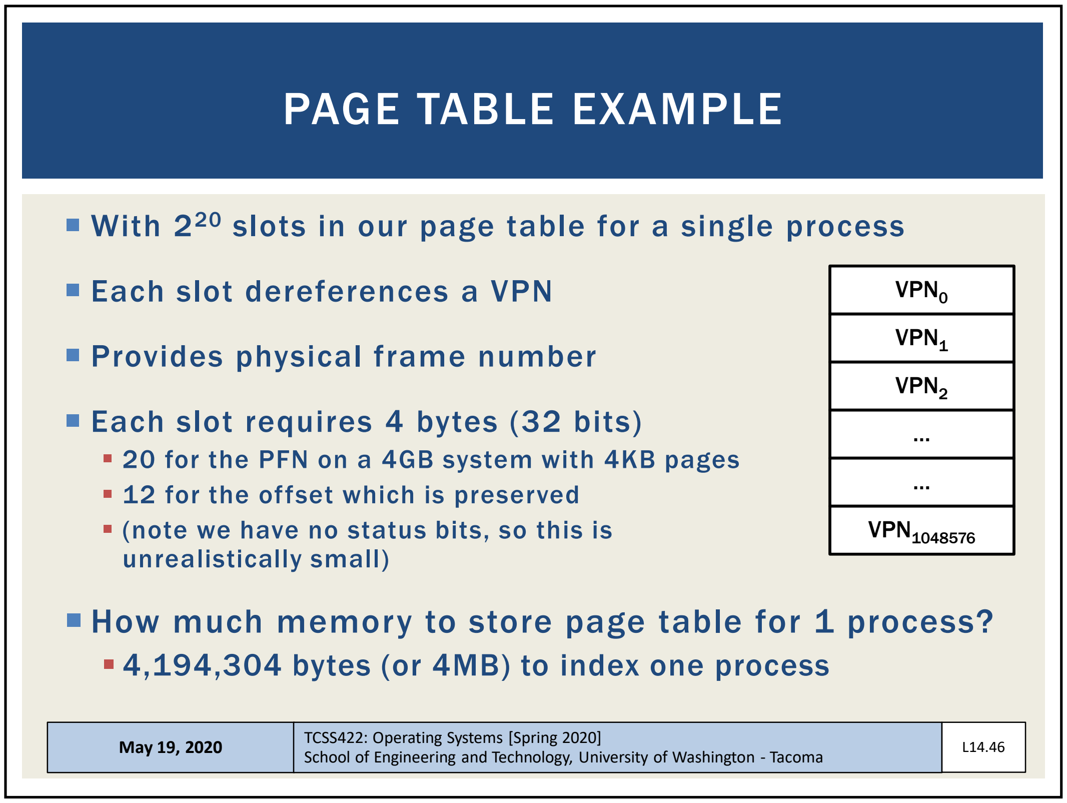
PAGING DESIGN QUESTIONS

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



PAGE TABLE EXAMPLE

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

VPN_0
VPN_1
VPN_2
…
…
$VPN_{1048576}$

- How much memory to store page table for 1 process?
  - 4,194,304 bytes (or 4MB) to index one process

NOW FOR AN ENTIRE OS

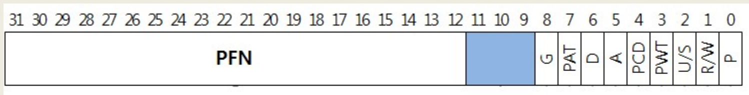
- If 4 MB is required to store one process
- Consider how much memory is required for an entire OS?
  - With for example 100 processes…
- Page table memory requirement is now 4MB x 100 = 400MB
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

400 MB / 4000 GB

- Is this efficient?

(2) WHAT'S ACTUALLY IN THE PAGE TABLE

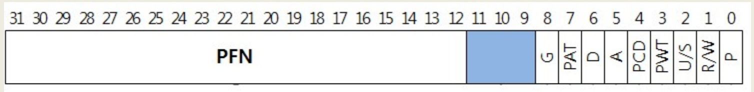
- Page table is data structure used to map virtual page numbers (VPN) to the physical address (Physical Frame Number PFN)
  - Linear page table → simple array
- Page-table entry
  - 32 bits for capturing state



An x86 Page Table Entry(PTE)

PAGE TABLE ENTRY

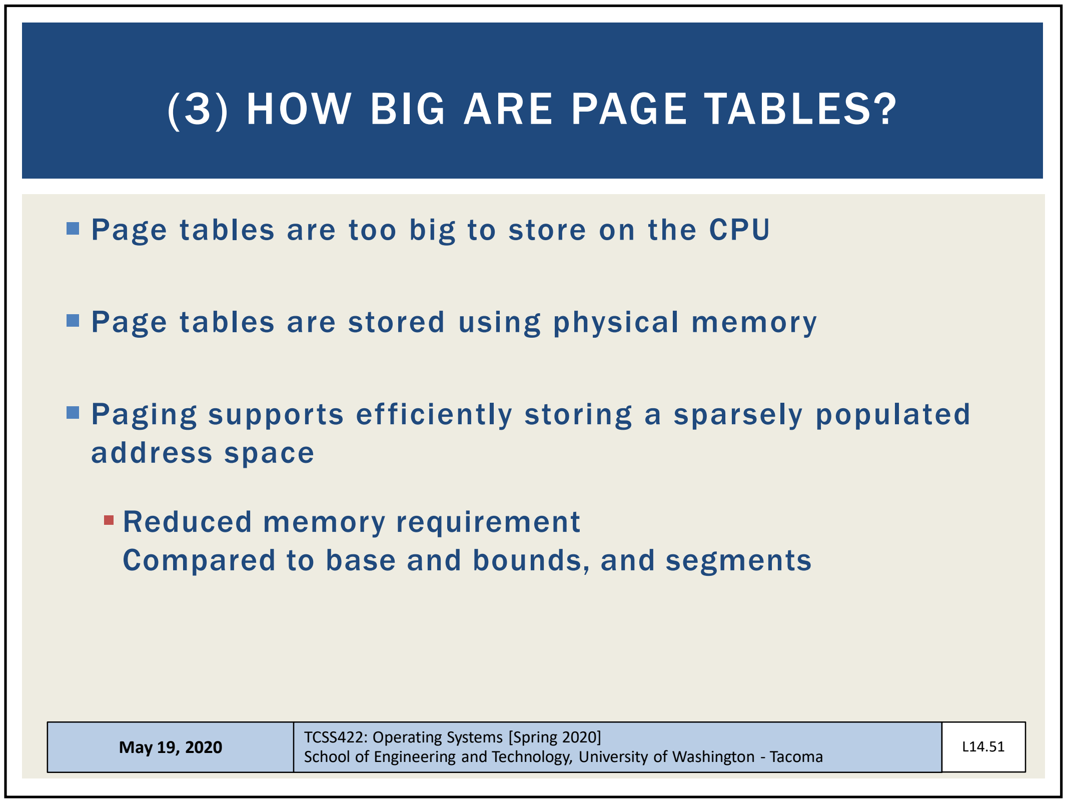
- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number



An x86 Page Table Entry(PTE)

PAGE TABLE ENTRY - 2

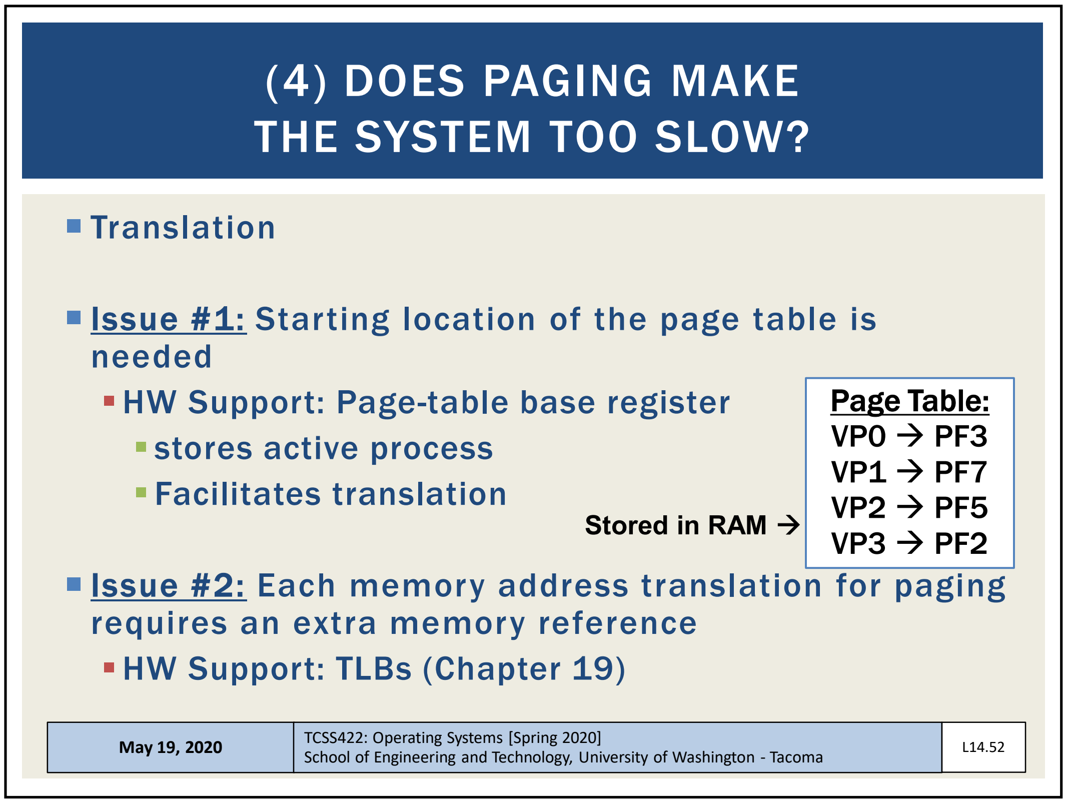
- Common flags:
- **Valid Bit:** Indicating whether the particular translation is valid.
- **Protection Bit:** Indicating whether the page could be read from, written to, or executed from
- **Present Bit:** Indicating whether this page is in physical memory or on disk(swapped out)
- **Dirty Bit:** Indicating 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

May 19, 2020 | TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma | L14.51



(4) DOES PAGING MAKE THE SYSTEM TOO SLOW?

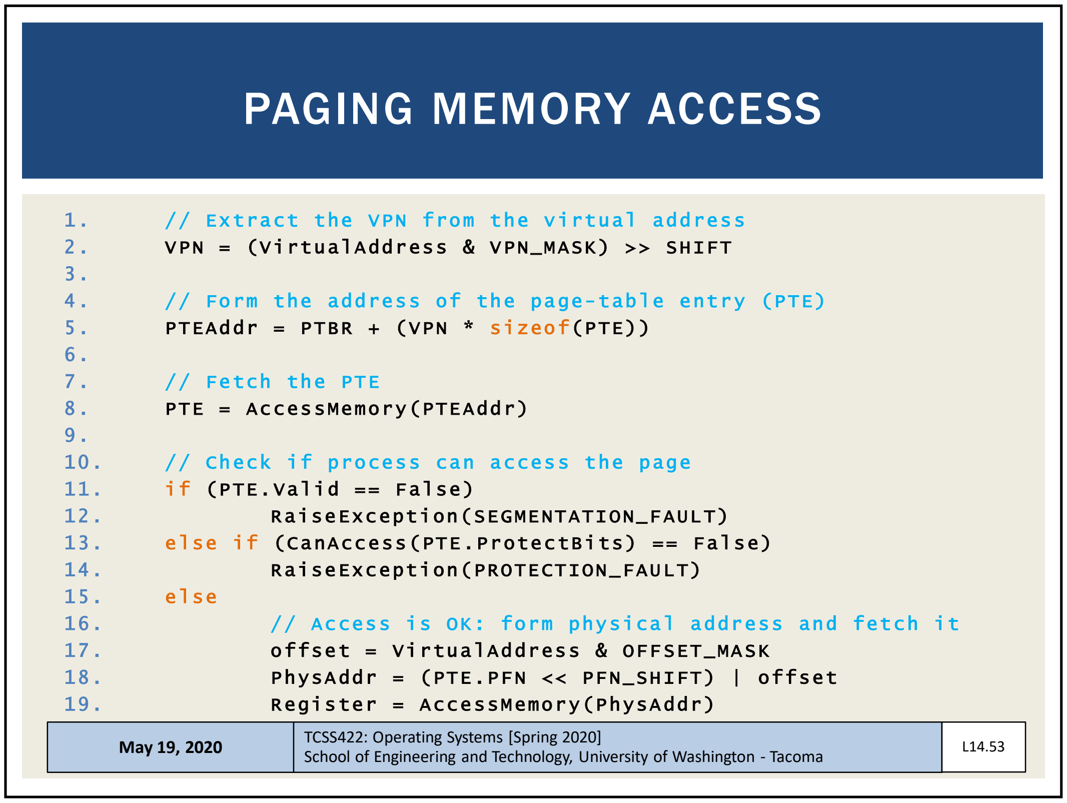
- Translation
- __Issue #1:__ Starting location of the page table is needed
  - HW Support: Page-table base register
    - stores active process
    - Facilitates translation

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)

May 19, 2020 | TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma | L14.52



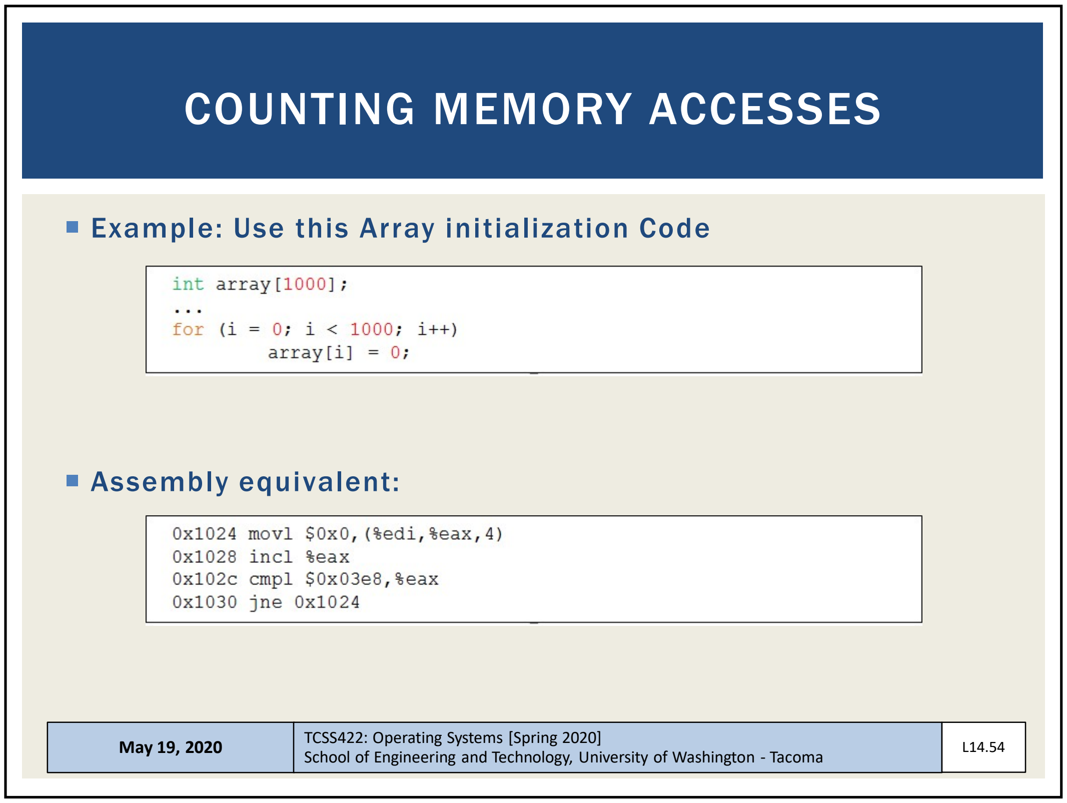
PAGING MEMORY ACCESS

```
// 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

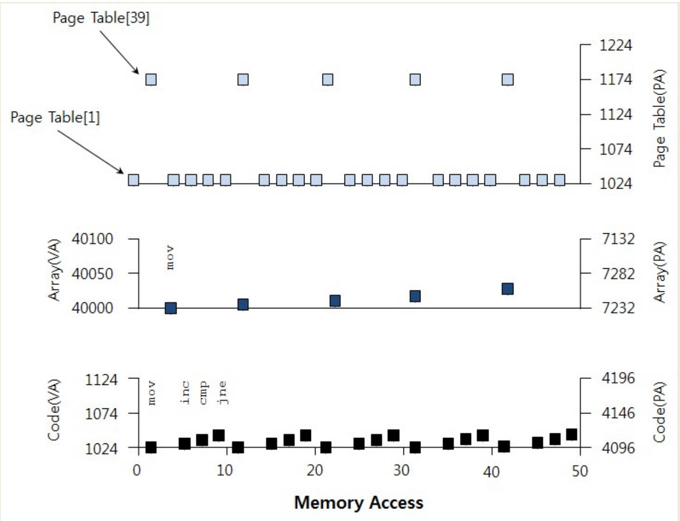
```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

- 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



May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.55

PAGING SYSTEM EXAMPLE

- Consider a 4GB Computer:
- With a 4096-byte page size (4KB)
- How many pages would fit in physical memory?
- Now consider a page table:
- For the page table entry, how many bits are required for the VPN?
- If we assume the use of 4-byte (32 bit) page table entries, how many bits are available for status bits?
- How much space does this page table require? Page Table Entries x Number of pages
- How many page tables (for user processes) would fill the entire 4GB of memory?

May 19, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.56

QUESTIONS

WILL RETURN IN A FEW MINUTES