TCSS 422: OPERATING SYSTEMS

Memory Virtualization with Segments, Introduction to Paging

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

1

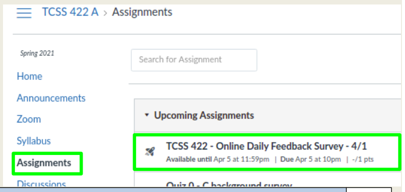
OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

2

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 A > Assignments

Spring 2021

Home
Announcements
Zoom
Syllabus
Assignments
Discussions

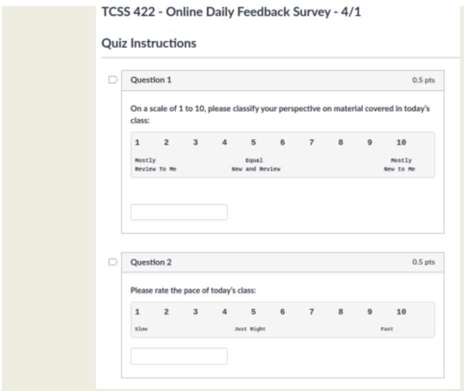
Search for Assignment

Upcoming Assignments

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

3



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

4

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (29 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- <u>**Average – 5.98 (↓ - previous 6.08)**</u>

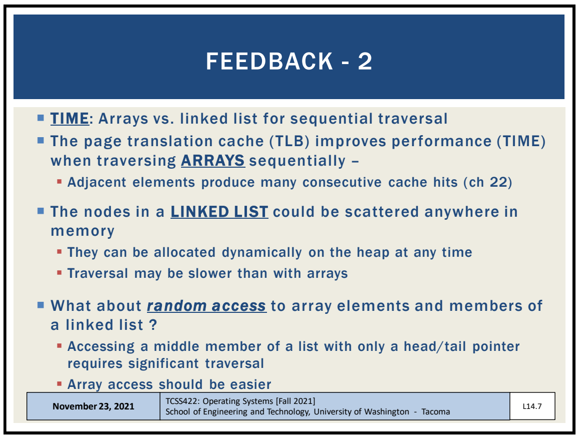
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- <u>**Average – 5.41 (↑ - previous 5.20)**</u>

5

FEEDBACK

- ***What data structures lend themselves to our goals for address spaces?***
- Goals for OS Memory Virtualization:
  - Transparency
  - Protection (Security/Isolation)
  - Efficiency (Time & Space)
- Some data structures may be more compact (space) with lower memory virtualization overhead (time)
  - SPACE: consider a statically declared multi-dimensional array vs. a linked list
  - One has a huge up-front cost, while the other grows incrementally

6

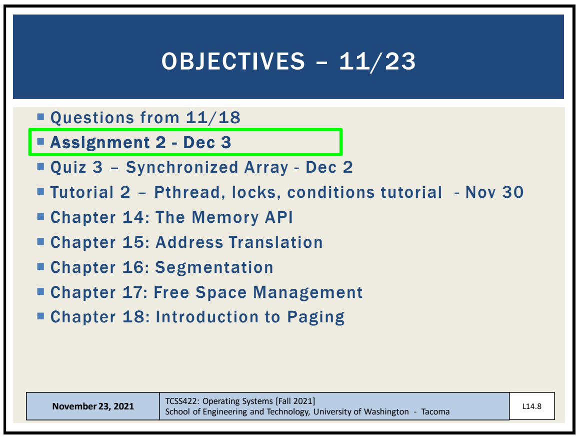


FEEDBACK - 2

- **TIME**: Arrays vs. linked list for sequential traversal
- The page translation cache (TLB) improves performance (TIME) when traversing **ARRAYS** sequentially –
  - Adjacent elements produce many consecutive cache hits (ch 22)
- The nodes in a **LINKED LIST** could be scattered anywhere in memory
  - They can be allocated dynamically on the heap at any time
  - Traversal may be slower than with arrays
- What about ***random access*** to array elements and members of a linked list ?
  - Accessing a middle member of a list with only a head/tail pointer requires significant traversal
  - Array access should be easier

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.7

7

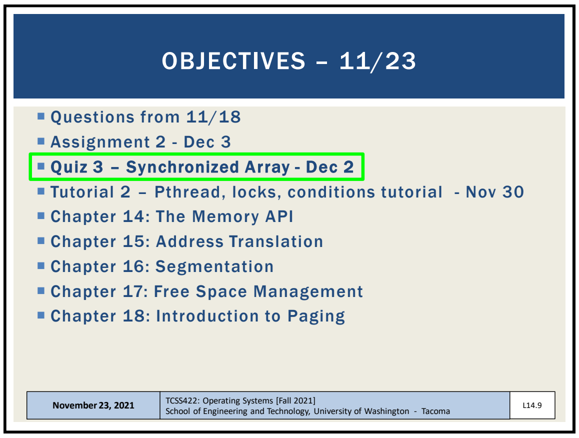


OBJECTIVES – 11/23

- Questions from 11/18
- **Assignment 2 - Dec 3**
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.8

8

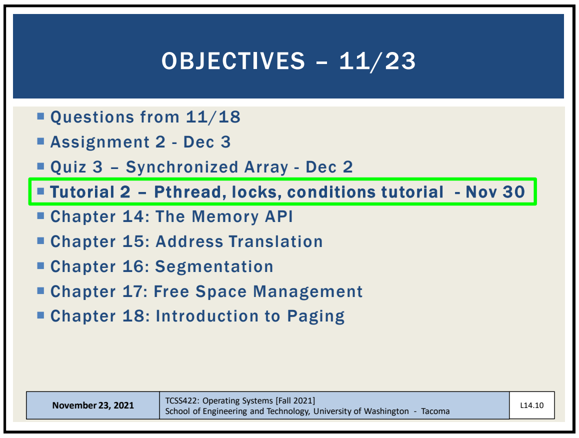


OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- **Quiz 3 – Synchronized Array - Dec 2**
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.9

9

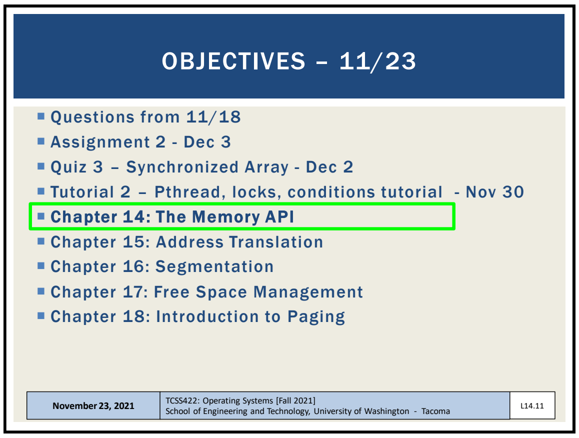


OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- **Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30**
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.10

10

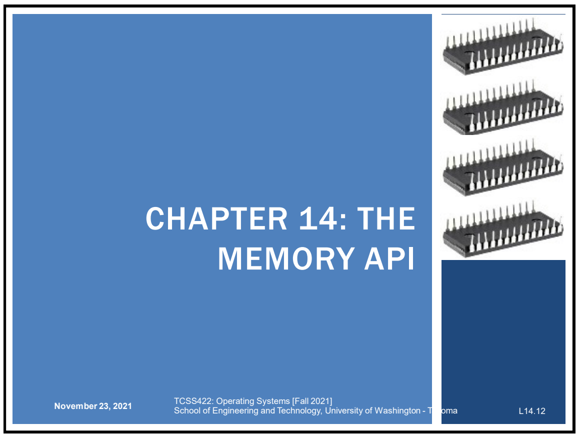


OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- **Chapter 14: The Memory API**
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.11

11



CHAPTER 14: THE MEMORY API

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.12

12

OBJECTIVES – 5/18

- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.13

13

REALLOC()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Resize an existing memory allocation
- Returned pointer may be same address, or a new address
  - New if memory allocation must move
- `void *ptr`: Pointer to memory block allocated with malloc, calloc, or realloc
- `size_t size`: New size for the memory block(in bytes)
- EXAMPLE: realloc.c
- EXAMPLE: nom.c

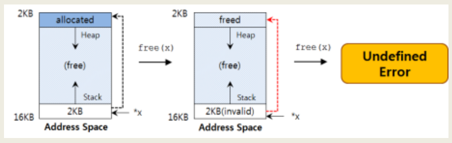
November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.14

14

DOUBLE FREE

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```

- Can't deallocate twice
- Second call core dumps



November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.15

15

SYSTEM CALLS

- brk(), sbrk()
- Used to change data segment size (the end of the heap)
- Don't use these
- Mmap(), munmap()
- Can be used to create an extra independent "heap" of memory for a user program
- See man page

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.16

16

OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.17

17

CHAPTER 15: ADDRESS TRANSLATION

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.18

18

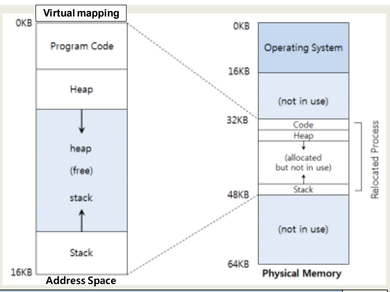
OBJECTIVES – 5/18

- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

19

ADDRESS TRANSLATION

- 64KB Address space example
- Translation: mapping virtual to physical



20

BASE AND BOUNDS

- Dynamic relocation
- Two registers base & bounds: **on the CPU**
- OS places program in memory
- Sets base register

$$\textit{physical address} = \textit{virtual address} + \textit{base}$$

- Bounds register
  - Stores size of program address space (16KB)
- OS verifies that every address:

$$0 \leq \textit{virtual address} < \textit{bounds}$$

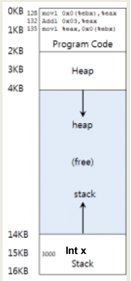
21

INSTRUCTION EXAMPLE

```
128 : movl 0x0(%ebx), %eax
```

- Base = 32768
- Bounds =16384
- Fetch instruction at 128 (virt addr) ↑
  - Phy addr = virt addr + base reg
  - 32896 = 128 + 32768 (base)
- Execute instruction
  - Load from address (var x is @ 15kb=15360)
  - 48128 = 15360 + 32768 (base) -- found x…
- Bounds register: terminate process if
  - ACCESS VIOLATION: Virtual address > bounds reg

$$\textit{physical address} = \textit{virtual address} + \textit{base}$$



22

MEMORY MANAGEMENT UNIT

- MMU
  - Portion of the CPU dedicated to address translation
  - Contains base & bounds registers
- Base & Bounds Example:
  - Consider address translation
  - 4 KB (4096 bytes) address space, loaded at 16 KB physical location

	Virtual Address	Physical Address
	0	16384
	1024	17408
	3000	19384
FAULT	4400	20784 (out of bounds)

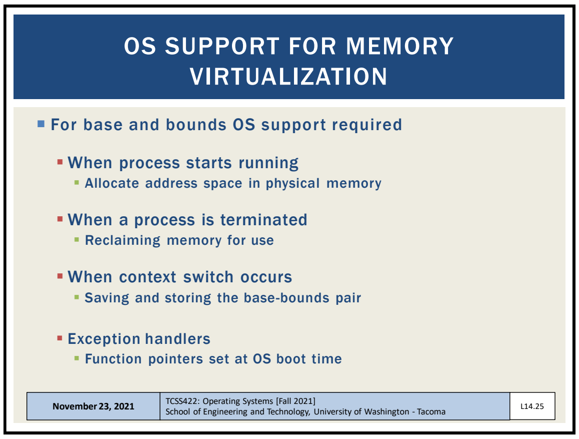
23

DYNAMIC RELOCATION OF PROGRAMS

- Hardware requirements:

Requirements	HW support
Privileged mode	CPU modes: kernel, user
Base / bounds registers	Registers to support address translation
Translate virtual addr; check if in bounds	Translation circuitry, check limits
Privileged instruction(s) to update base / bounds regs	Instructions for modifying base/bound registers
Privileged instruction(s) to register exception handlers	Set code pointers to OS code to handle faults
Ability to raise exceptions	For out-of-bounds memory access, or attempts to access privileged instr.

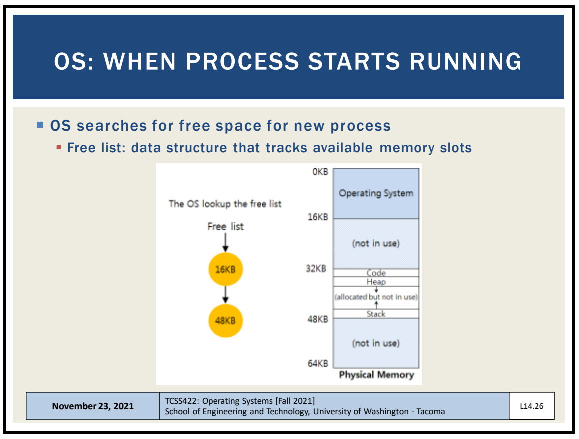
24



OS SUPPORT FOR MEMORY VIRTUALIZATION

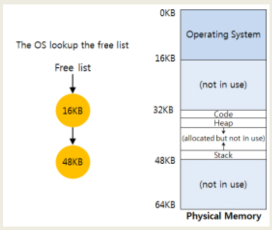
- For base and bounds OS support required
  - When process starts running
    - Allocate address space in physical memory
  - When a process is terminated
    - Reclaiming memory for use
  - When context switch occurs
    - Saving and storing the base-bounds pair
  - Exception handlers
    - Function pointers set at OS boot time

25

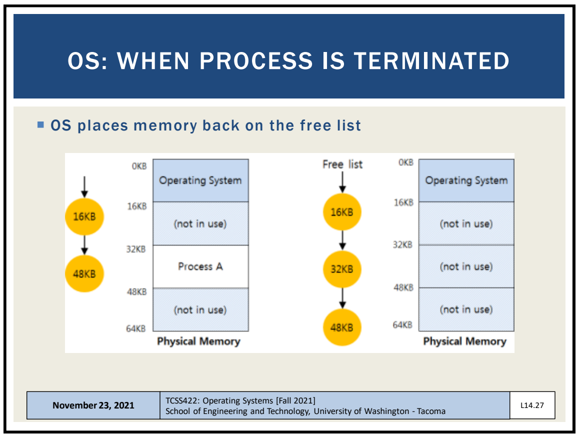


OS: WHEN PROCESS STARTS RUNNING

- OS searches for free space for new process
  - Free list: data structure that tracks available memory slots

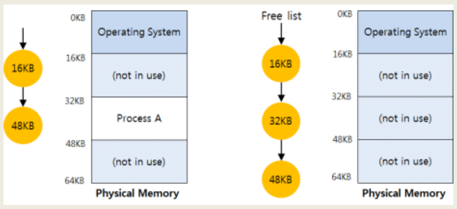


26

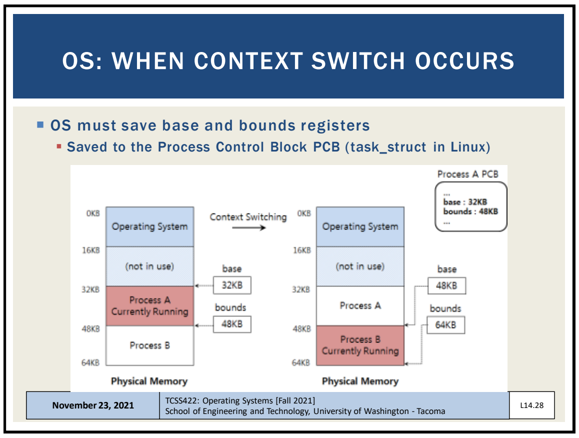


OS: WHEN PROCESS IS TERMINATED

- OS places memory back on the free list

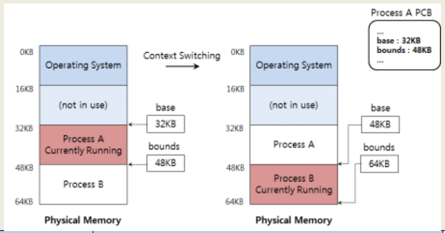


27

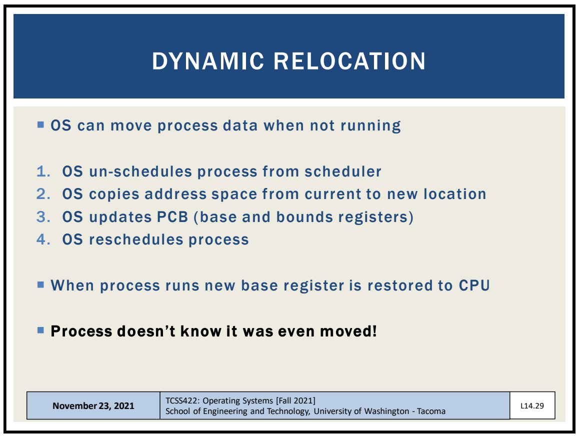


OS: WHEN CONTEXT SWITCH OCCURS

- OS must save base and bounds registers
  - Saved to the Process Control Block PCB (task_struct in Linux)



28



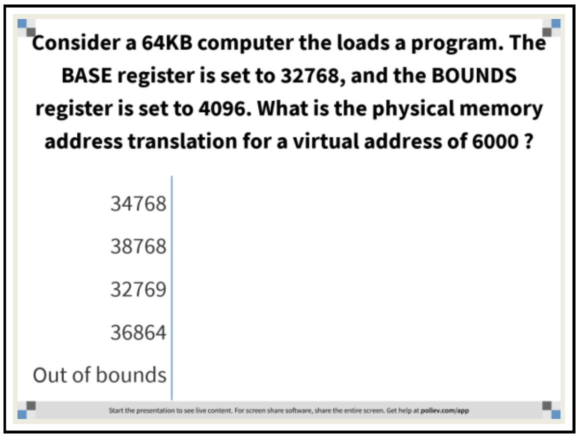
DYNAMIC RELOCATION

- OS can move process data when not running

1. OS un-schedules process from scheduler
2. OS copies address space from current to new location
3. OS updates PCB (base and bounds registers)
4. OS reschedules process

- When process runs new base register is restored to CPU
- **Process doesn't know it was even moved!**

29



Consider a 64KB computer the loads a program. The BASE register is set to 32768, and the BOUNDS register is set to 4096. What is the physical memory address translation for a virtual address of 6000 ?

34768

38768

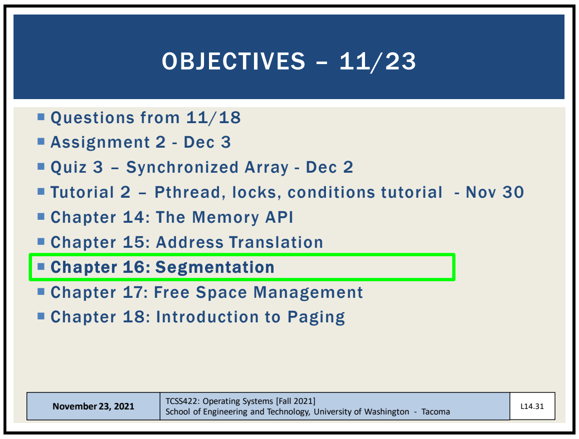
32769

36864

Out of bounds

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

30

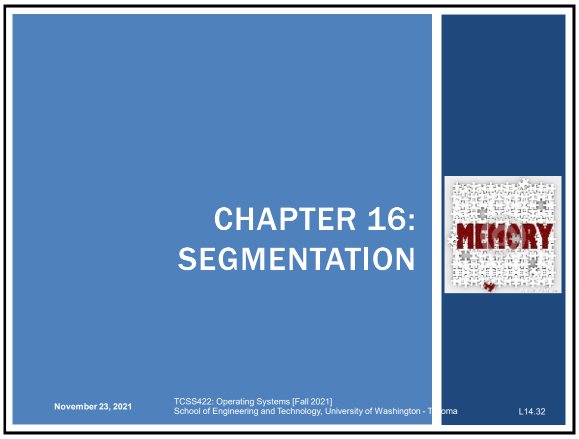


OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- **Chapter 16: Segmentation**
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.31

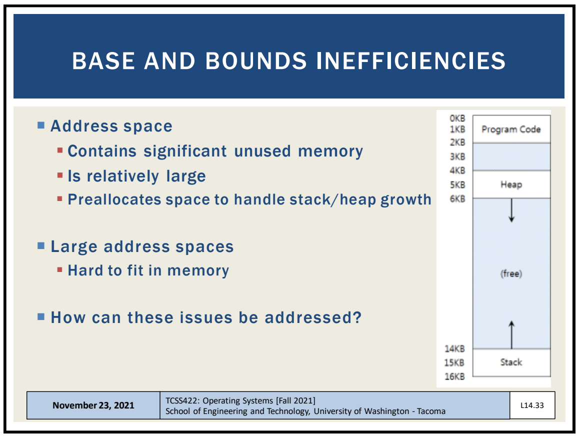
31



CHAPTER 16: SEGMENTATION

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.32

32



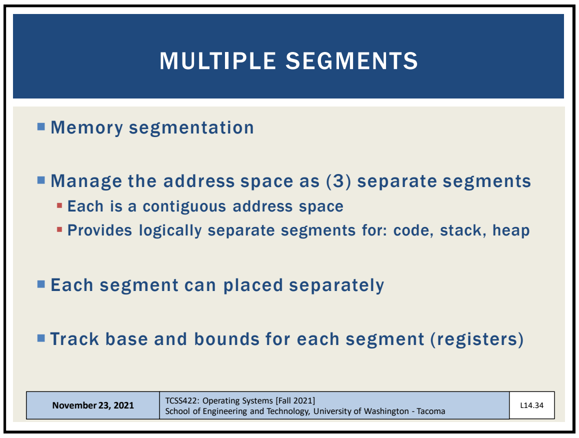
BASE AND BOUNDS INEFFICIENCIES

- Address space
  - Contains significant unused memory
  - Is relatively large
  - Preallocates space to handle stack/heap growth
- Large address spaces
  - Hard to fit in memory
- How can these issues be addressed?

0KB
1KB
2KB
3KB
4KB
5KB
6KB
14KB
15KB
16KB
Program Code
Heap
(free)
Stack

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.33

33

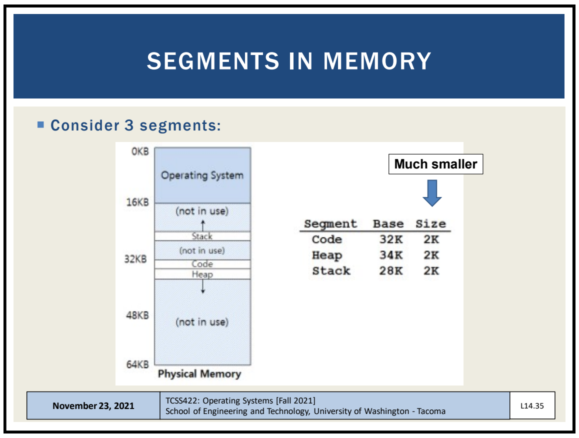


MULTIPLE SEGMENTS

- Memory segmentation
- Manage the address space as (3) separate segments
  - Each is a contiguous address space
  - Provides logically separate segments for: code, stack, heap
- Each segment can placed separately
- Track base and bounds for each segment (registers)

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.34

34



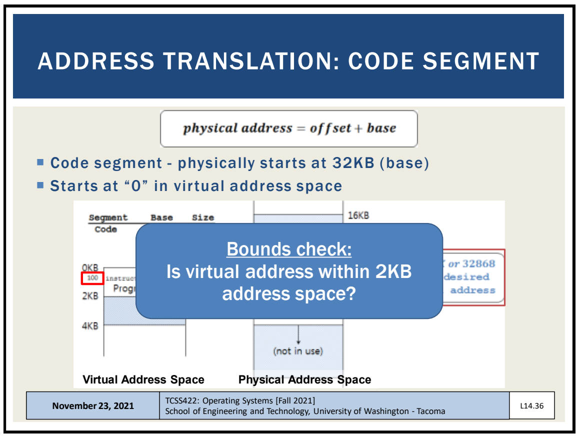
SEGMENTS IN MEMORY

- Consider 3 segments:

Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.35

35



ADDRESS TRANSLATION: CODE SEGMENT

$$physical\ address = offset + base$$

- Code segment - physically starts at 32KB (base)
- Starts at "0" in virtual address space

Bounds check:
Is virtual address within 2KB address space?

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.36

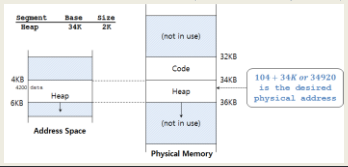
36

ADDRESS TRANSLATION: HEAP

Virtual address + base is not the correct physical address.

- Heap starts at virtual address 4096
- The data is at 4200
- Offset= 4200 – 4096 = 104 (virt addr – virt heap start)
- Physical address = 104 + 34816 (offset + heap base)

Segment	Base	Size
Heap	34K	2K

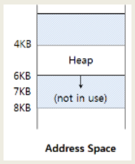


37

SEGMENTATION FAULT

- Access beyond the address space
- Heap starts at virtual address: 4096
- Data pointer is to 7KB (7168)
- Is data pointer valid?

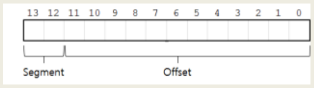
- Heap starts at 4096 + 2 KB seg size = 6144
- Offset= 7168 > 4096 + 2048 (6144)



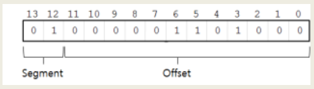
38

SEGMENT REGISTERS

- Used to dereference memory during translation



- First two bits identify segment type
- Remaining bits identify memory offset
- Example: virtual heap address 4200 (01000001101000)



Segment	bits
Code	00
Heap	01
Stack	10
-	11

39

SEGMENTATION DEREFERENCE

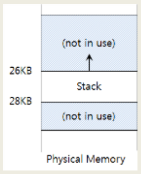
```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
     RaiseException(PROTECTION_FAULT)
else
     PhysAddr = Base[Segment] + Offset
     Register = AccessMemory(PhysAddr)
```

- VIRTUAL ADDRESS = 01000001101000 (on heap)
- SEG_MASK = 0x3000 (11000000000000)
- SEG_SHIFT = 01 → ***heap*** (mask gives us segment code)
- OFFSET_MASK = 0xFFF (00111111111111)
- OFFSET = 000001101000 = 104 (isolates segment offset)
- OFFSET < BOUNDS : 104 < 2048

40

STACK SEGMENT

- Stack grows backwards (FILO)
- Requires hardware support:
- Direction bit: tracks direction segment grows



Segment Register(with Negative-Growth Support)

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

41

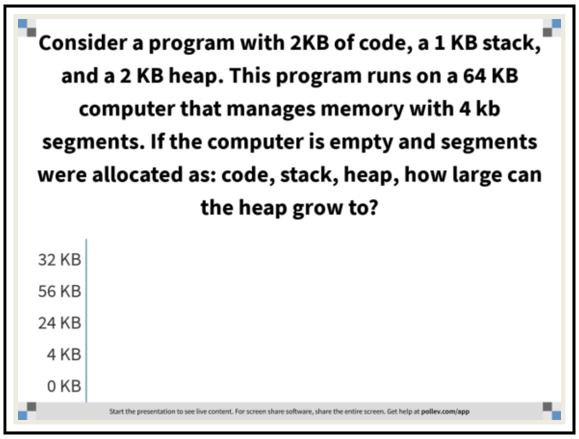
SHARED CODE SEGMENTS

- Code sharing: enabled with HW support
- Supports storing shared libraries in memory only once
- DLL: dynamic linked library
- .so (linux): shared object in Linux (under /usr/lib)
- Many programs can access them
- Protection bits: track permissions to segment

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

42

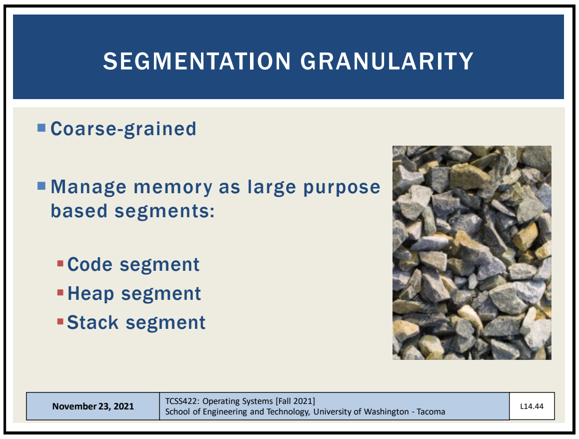


Consider a program with 2KB of code, a 1 KB stack, and a 2 KB heap. This program runs on a 64 KB computer that manages memory with 4 kb segments. If the computer is empty and segments were allocated as: code, stack, heap, how large can the heap grow to?

- 32 KB
- 56 KB
- 24 KB
- 4 KB
- 0 KB

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

43

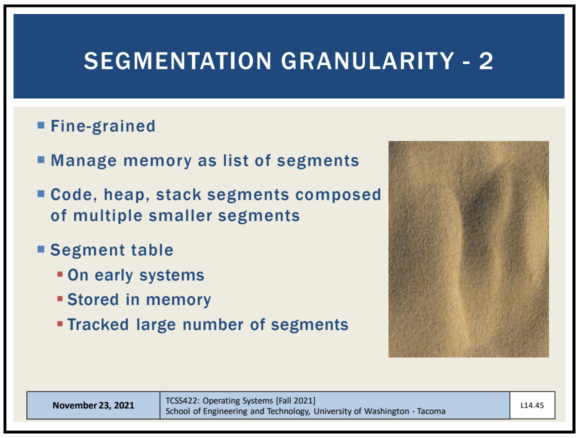


SEGMENTATION GRANULARITY

- Coarse-grained
- Manage memory as large purpose based segments:
  - Code segment
  - Heap segment
  - Stack segment

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.44

44

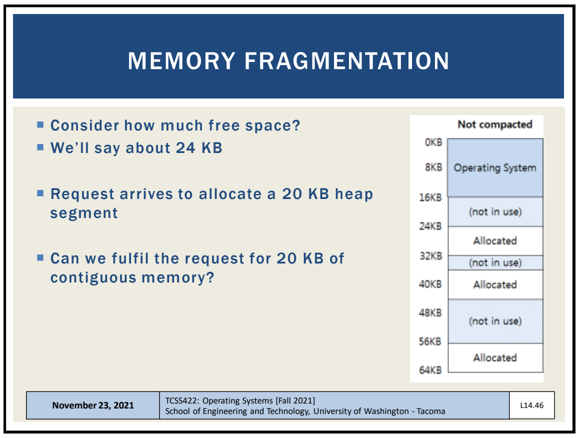


SEGMENTATION GRANULARITY - 2

- Fine-grained
- Manage memory as list of segments
- Code, heap, stack segments composed of multiple smaller segments
- Segment table
  - On early systems
  - Stored in memory
  - Tracked large number of segments

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.45

45

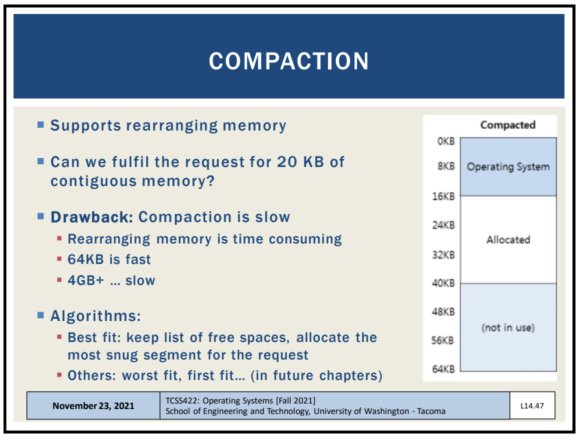


MEMORY FRAGMENTATION

- Consider how much free space?
- We'll say about 24 KB
- Request arrives to allocate a 20 KB heap segment
- Can we fulfil the request for 20 KB of contiguous memory?

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.46

46



COMPACTION

- Supports rearranging memory
- Can we fulfil the request for 20 KB of contiguous memory?
- **Drawback:** Compaction is slow
  - Rearranging memory is time consuming
  - 64KB is fast
  - 4GB+ … slow
- Algorithms:
  - Best fit: keep list of free spaces, allocate the most snug segment for the request
  - Others: worst fit, first fit… (in future chapters)

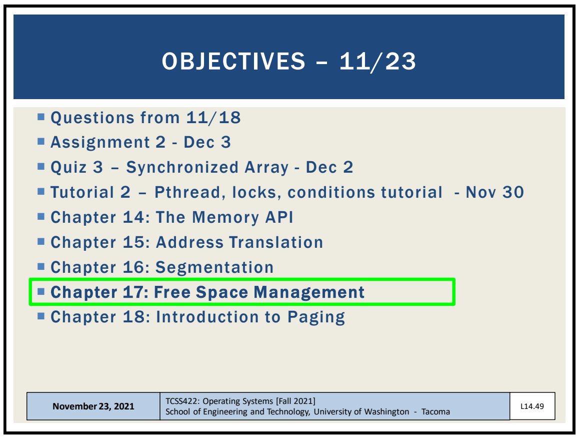
November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.47

47

WE WILL RETURN AT 4:50PM

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - coma | L14.48

48

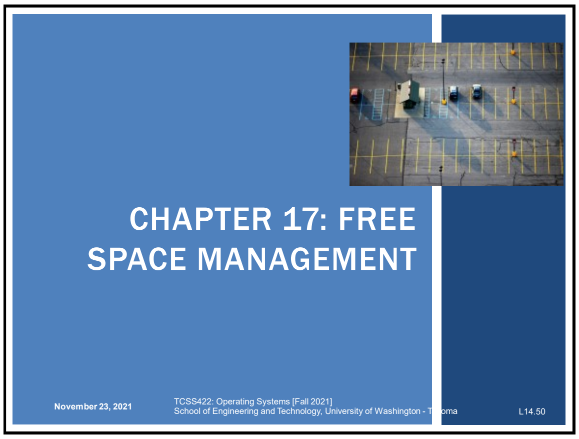


OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- **Chapter 17: Free Space Management**
- Chapter 18: Introduction to Paging

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.49

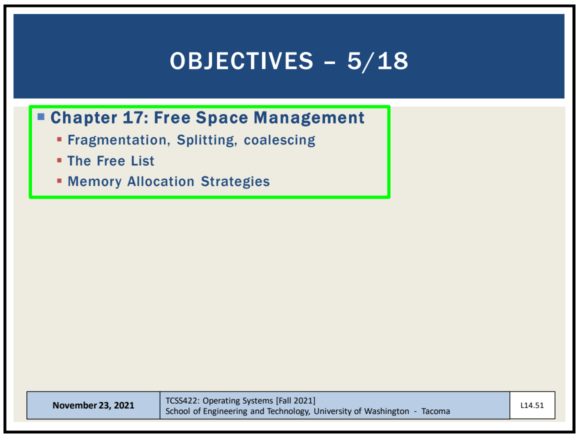
49



CHAPTER 17: FREE SPACE MANAGEMENT

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.50

50

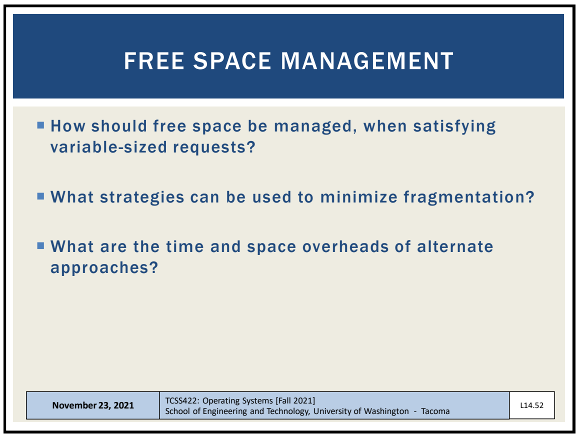


OBJECTIVES – 5/18

- **Chapter 17: Free Space Management**
  - Fragmentation, Splitting, coalescing
  - The Free List
  - Memory Allocation Strategies

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.51

51

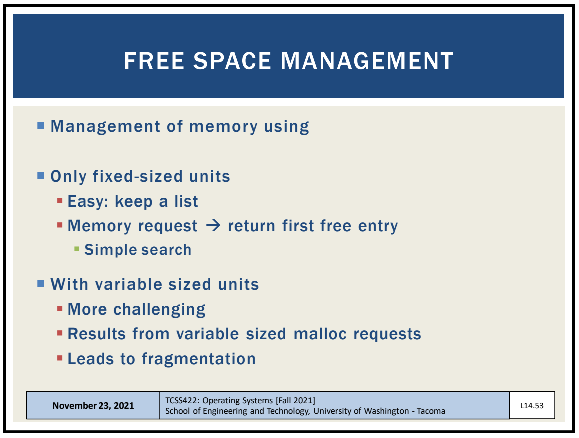


FREE SPACE MANAGEMENT

- 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?

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.52

52

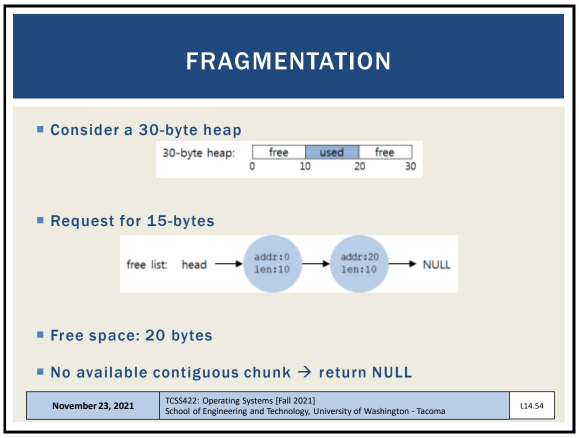


FREE SPACE MANAGEMENT

- 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

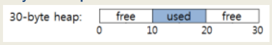
November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.53

53

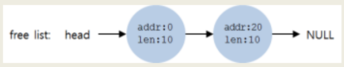


FRAGMENTATION

- Consider a 30-byte heap



- Request for 15-bytes



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

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.54

54

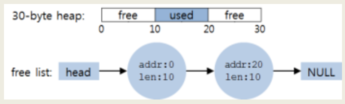
FRAGMENTATION - 2

- **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

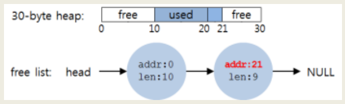
55

ALLOCATION STRATEGY: SPLITTING

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



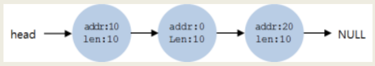
- OS locates a free chunk to satisfy request
- Splits chunk into two, returns first chunk



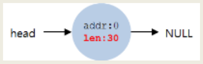
56

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

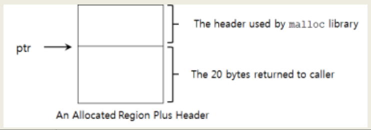


- Allocation can now proceed
- Coalescing is defragmentation of the free space list

57

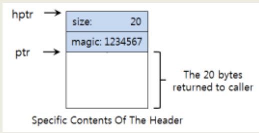
MEMORY HEADERS

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

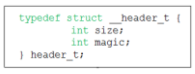

ptr
The header used by malloc library
The 20 bytes returned to caller
An Allocated Region Plus Header

58

MEMORY HEADERS - 2



Specific Contents Of The Header



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

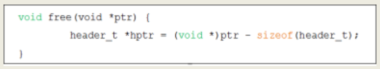
A Simple Header

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

59

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);
}
```

60

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;
```

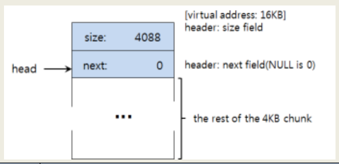
61

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;
```

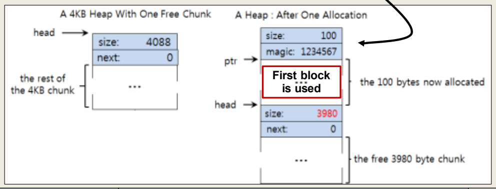
- Heap layout:



62

FREE LIST: MALLOC() CALL

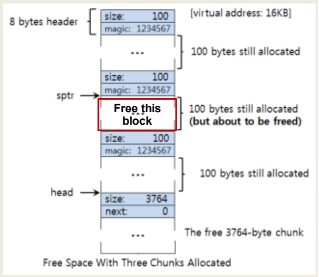
- Consider a request for a 100 bytes: malloc(100)
- Header block requires 8 bytes
  - 4 bytes for size, 4 bytes for magic number
- Split the heap – <u>header goes with each block</u>



63

FREE LIST: FREE() CALL

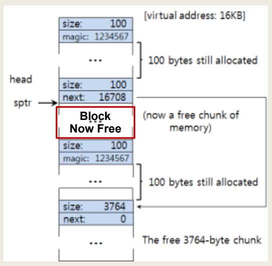
- Addresses of chunks
- Start=16384
  + 108 (end of 1st chunk)
  + 108 (end of 2nd chunk)
  + 108 (end of 3rd chunk)
  = 16708



64

FREE LIST: FREE() CHUNK #2

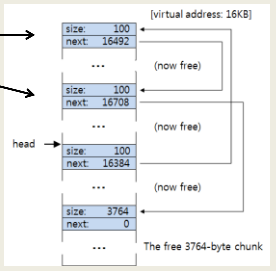
- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
  - addr – sizeof(node_t)
- Actual start of chunk #2
  - 16492



65

FREE LIST- FREE ALL CHUNKS

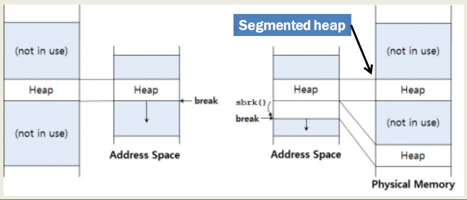
- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed



66

GROWING THE HEAP

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



67

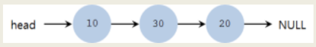
MEMORY ALLOCATION STRATEGIES

- **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

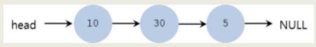
68

EXAMPLES

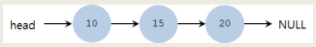
- Allocation request for 15 bytes



- Result of Best Fit



- Result of Worst Fit



69

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

70

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

71

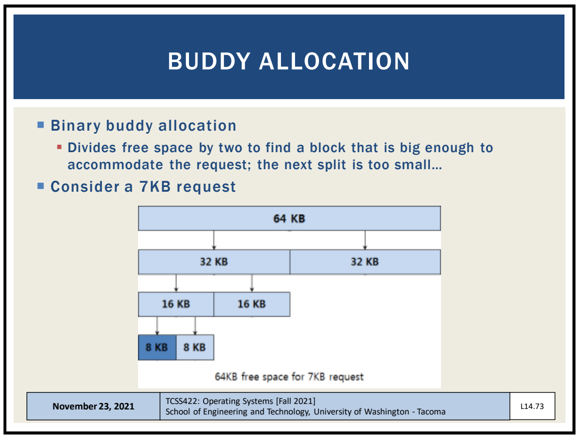
SEGREGATED LISTS

- 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)?

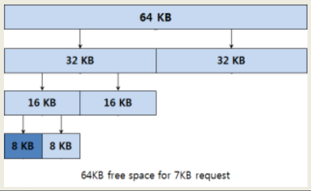
- If a given cache is low in memory, can request "*slabs*" of memory from the general allocator for caches.
- General allocator will reclaim slabs when not used

72



BUDDY ALLOCATION

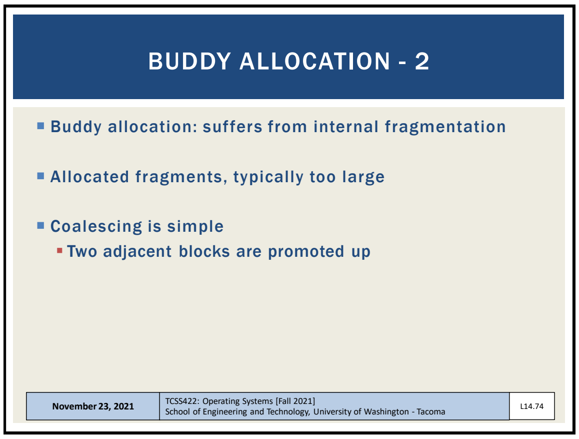
- Binary buddy allocation
  - Divides free space by two to find a block that is big enough to accommodate the request; the next split is too small…
- Consider a 7KB request



64KB free space for 7KB request

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.73

73

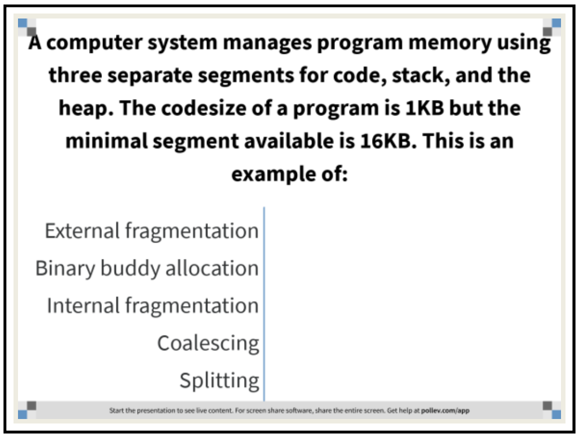


BUDDY ALLOCATION - 2

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

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.74

74

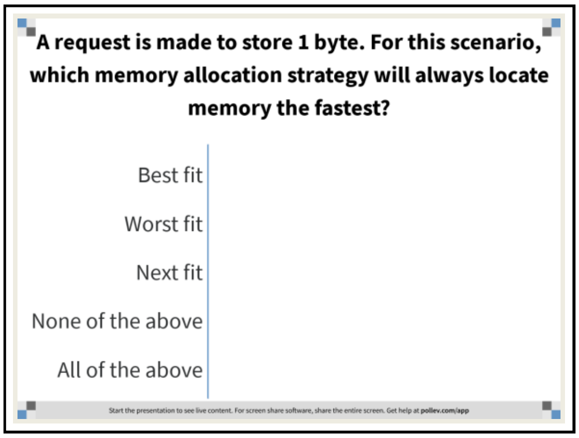


A computer system manages program memory using three separate segments for code, stack, and the heap. The codesize of a program is 1KB but the minimal segment available is 16KB. This is an example of:

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

75

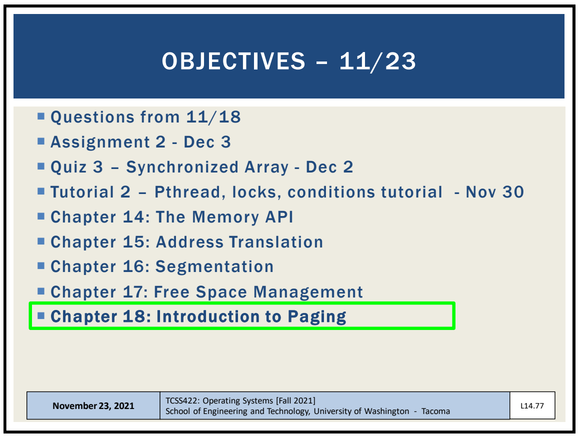


A request is made to store 1 byte. For this scenario, which memory allocation strategy will always locate memory the fastest?

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

76

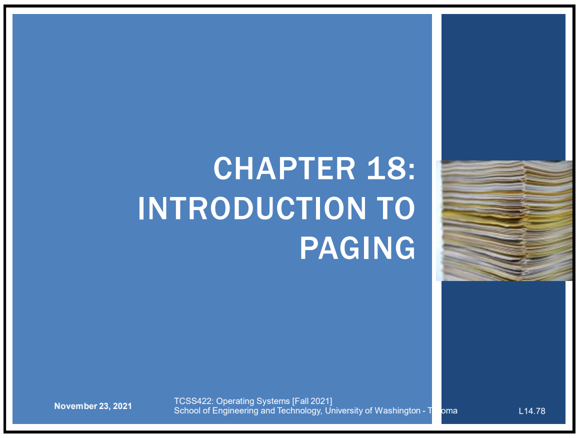


OBJECTIVES – 11/23

- Questions from 11/18
- Assignment 2 - Dec 3
- Quiz 3 – Synchronized Array - Dec 2
- Tutorial 2 – Pthread, locks, conditions tutorial - Nov 30
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- **Chapter 18: Introduction to Paging**

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.77

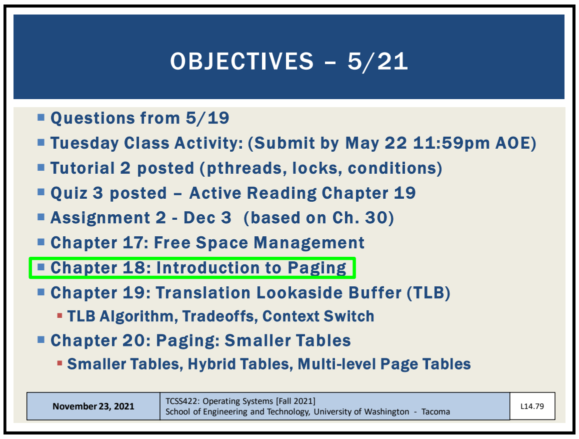
77



CHAPTER 18: INTRODUCTION TO PAGING

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.78

78

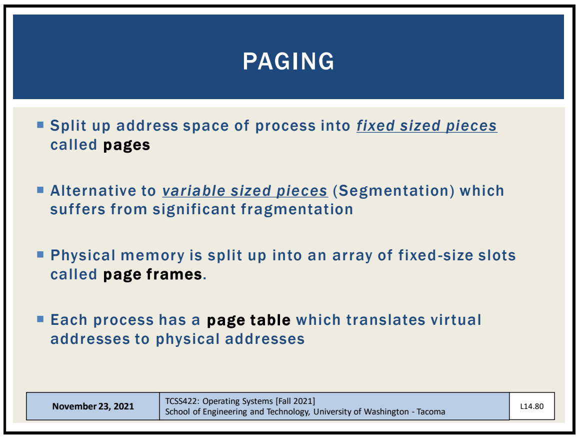


OBJECTIVES – 5/21

- Questions from 5/19
- Tuesday Class Activity: (Submit by May 22 11:59pm AOE)
- Tutorial 2 posted (pthreads, locks, conditions)
- Quiz 3 posted – Active Reading Chapter 19
- Assignment 2 - Dec 3 (based on Ch. 30)
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging
- Chapter 19: Translation Lookaside Buffer (TLB)
  - TLB Algorithm, Tradeoffs, Context Switch
- Chapter 20: Paging: Smaller Tables
  - Smaller Tables, Hybrid Tables, Multi-level Page Tables

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.79

79

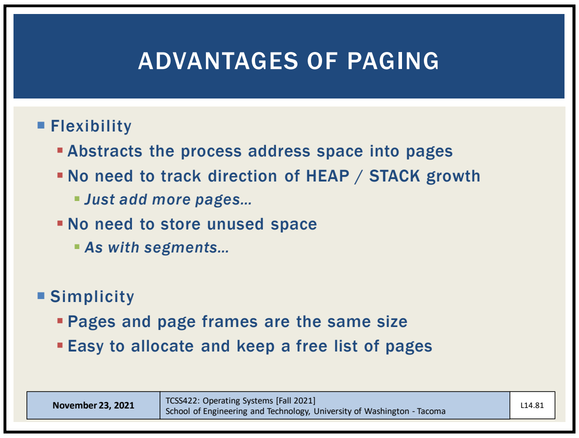


PAGING

- 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

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.80

80

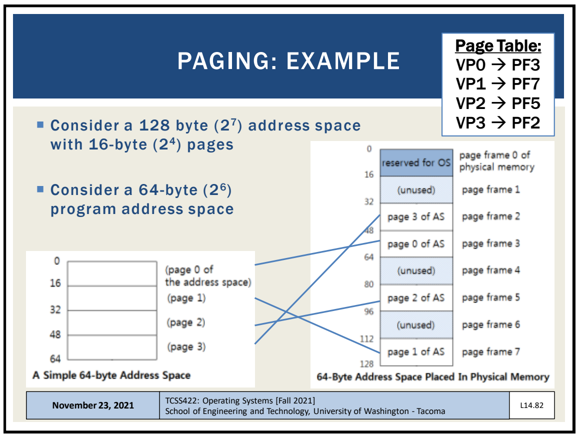


ADVANTAGES OF PAGING

- 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

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.81

81



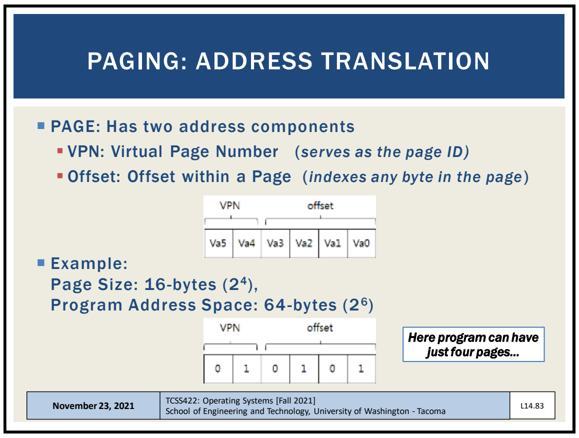
PAGING: EXAMPLE

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

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.82

82



PAGING: ADDRESS TRANSLATION

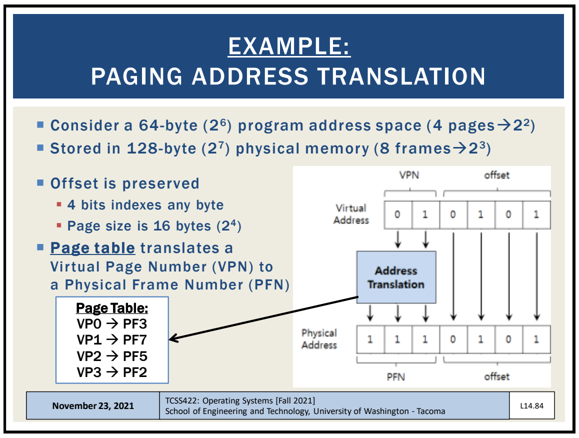
- PAGE: Has two address components
  - VPN: Virtual Page Number (*serves as the page ID)*
  - Offset: Offset within a Page (*indexes any byte in the page*)

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

Here program can have just four pages...

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.83

83



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

November 23, 2021 | TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma | L14.84

84

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?

85

(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

86

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

87

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

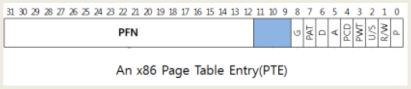
400 MB / 4000 GB

- **Is this efficient?**

88

(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

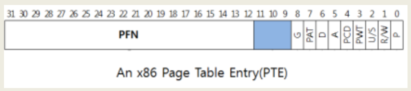


An x86 Page Table Entry(PTE)

89

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)

90

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

91

(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

92

(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)

93

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)
```

94

COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

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

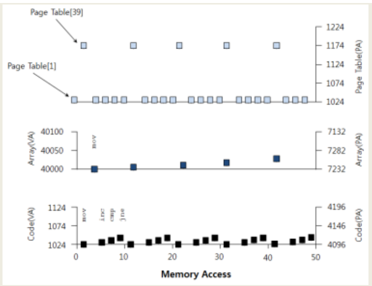
- Assembly equivalent:

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

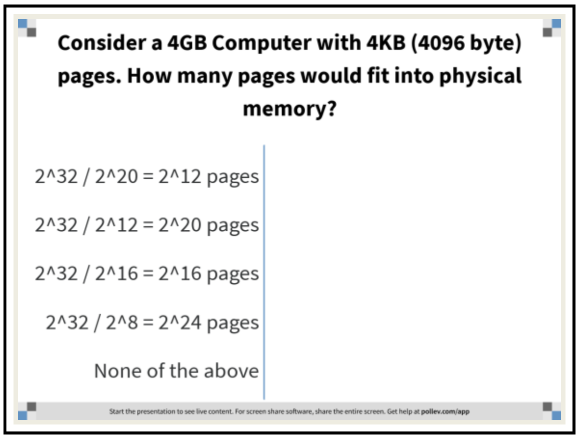
95

VISUALIZING MEMORY ACCESSES: FOR THE FIRST 5 LOOP ITERATIONS

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



96



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

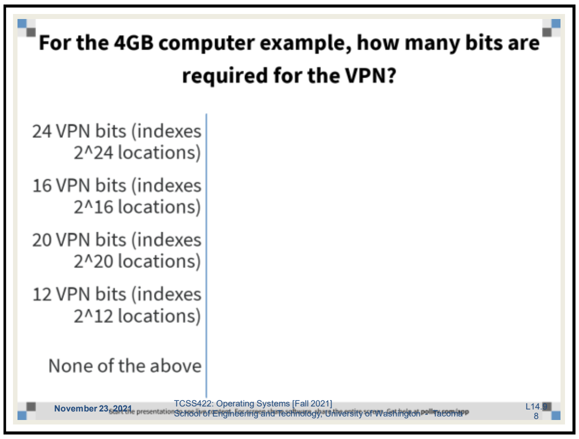
2^32 / 2^16 = 2^16 pages

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

97



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)

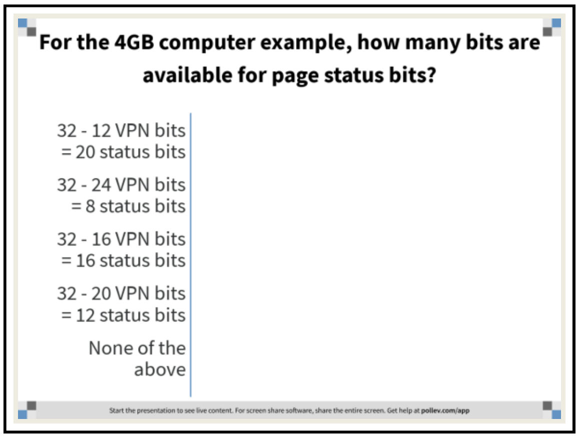
20 VPN bits (indexes 2^20 locations)

12 VPN bits (indexes 2^12 locations)

None of the above

November 23, 2021 — TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma — L14.98

98



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

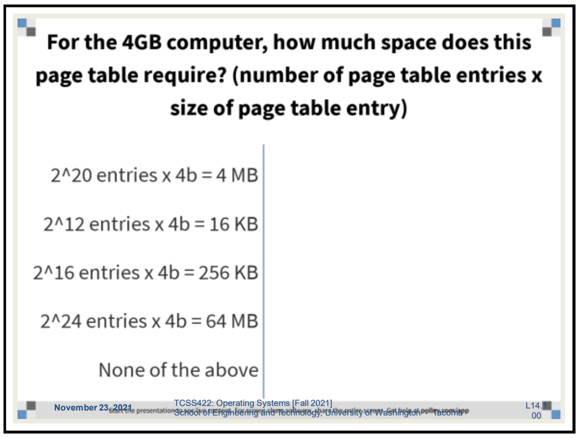
32 - 16 VPN bits = 16 status bits

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

99



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

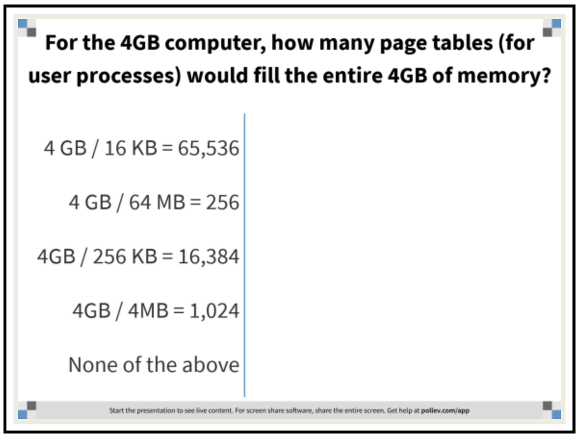
2^16 entries x 4b = 256 KB

2^24 entries x 4b = 64 MB

None of the above

November 23, 2021 — TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma — L14.100

100



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

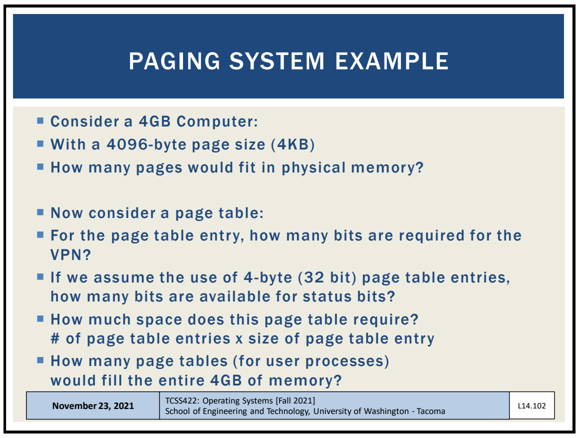
4GB / 256 KB = 16,384

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

101



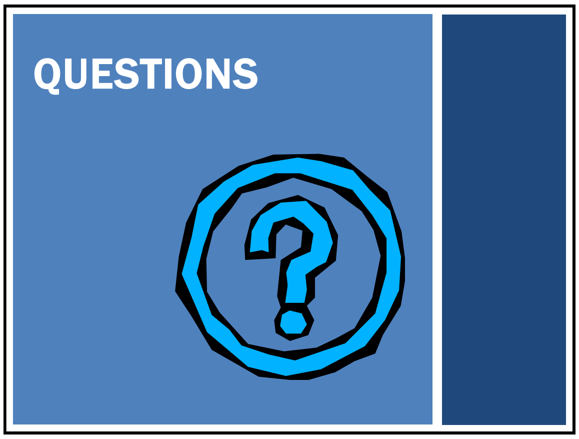
PAGING SYSTEM EXAMPLE

- 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?

November 23, 2021 — TCSS422: Operating Systems [Fall 2021] School of Engineering and Technology, University of Washington - Tacoma — L14.102

102



QUESTIONS

103