TCSS 422: OPERATING SYSTEMS

Memory Virtualization with Segments, Introduction to Paging

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

1

TCSS 422 – OFFICE HRS – SPRING 2025

- Office Hours plan for Winter:
- Tuesday 2:30 - 3:30 pm Instructor Wes, Zoom
- Tue/Thur 6:00 - 7:00 pm Instructor Wes, CP 229/Zoom
- Tue 7:00 – 8:00 pm GTA Robert, Zoom Only This Week
- Wed 7:00 – 8:00 pm GTA Robert, Zoom Only This Week

- Instructor is available after class at 6pm in CP 229 each day

2

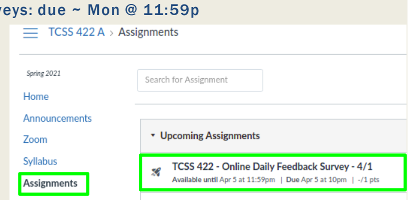
OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

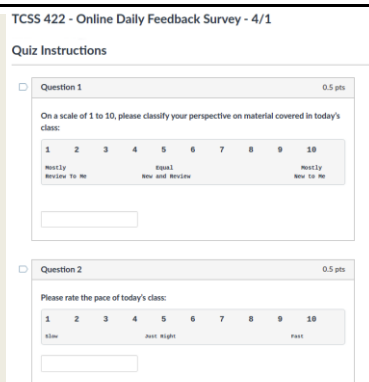
3

ONLINE DAILY FEEDBACK SURVEY

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



4



5

MATERIAL / PACE

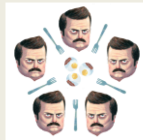
- Please classify your perspective on material covered in today's class (33 of 46 respondents (4 online) – 71.74%):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.79 (↓ - previous 6.92)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 4.88 (↓ - previous 5.08)

6

FEEDBACK FROM 2/24

- ***What dead locking problems does the dining philosopher's problem have?***
- For deadlock to occur, all four must be true:
- (√) **Mutual exclusion**: each fork can be held by only one philosopher at a time
- (√) **Hold and wait**: a philosopher picks up a fork and waits for another
- (√) **No preemption**: A fork can not be forcibly taken away
- (√) **Circular Wait**:
  - Philosopher 1 waits for philosopher 2
  - Philosopher 2 waits for philosopher 3



February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.7

7

FEEDBACK - 2

- ***Why would you not use a combination of deadlock solutions at the same time to prevent deadlock?***
- Methods we discussed:
  - **Mutal Exclusion:** Eliminate locks (use atomic data types & wait-free data structures)
  - **Hold and Wait:** Use non-blocking lock APIs to fix lack of ability to preempt threads that forever hold a lock
  - **No Preemption:** Introduce a guard lock to prevent hold and wait
  - **Circular Wait:** Use a total ordering of lock acquisition throughout the entire program to eliminate circular chains of events
- Fixing either of the first two clearly eliminates deadlock
  - Mutual exclusion: you remove locking
  - Hold and wait: you remove blocking lock calls
- Fixing either of the last two will also eliminate deadlock
  - No preemption: no longer acquire 1 lock, when 2 are needed
  - Circular wait: no longer acquire locks in wrong order

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.8

8

FEEDBACK - 3

- ***If all 4 conditions are required for deadlock to occur, then solving one of those conditions would prevent deadlock.***
- YES
- ***Would it be most optimal to only solve one problem or all 4?***
- It is less work to solve just 1 problem.
- Using non-blocking APIs (to fix hold and wait) seems the easiest:
  - You don't have to find and correct all circular wait and no preemption scenarios in your code (hard)
  - You don't have to use atomic variables or build wait-free data structures (hard)
  - You just have to deal with the live-lock problem, and introduce a random delay to solve it

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.9

9

FEEDBACK - 4

- ***Would intelligent schedulers (theoretically) build w/ AI cause less than 5 nines of uptime ?***
- 99.999% uptime is just 5.256 minutes of annual downtime
- It depends on the prompt engineer

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.10

10

OBJECTIVES – 2/26

- Questions from 2/24
- **Assignment 2 - March 12 AOE**
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.11

11

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- **Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4**
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.12

12

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- **Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4**
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

13

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- **Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4**
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

14

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- **Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE**
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

15

REVIEW FROM 5/15

- **Review: Conditions for deadlock to occur**
- **Mutual Exclusion:** threads claim exclusive control of resources
  > use atomic operations for data updates to eliminate mutual exclusion – ***prevents deadlock***
- **Hold-and-Wait:** threads hold resources while waiting for others
  > use guard locks when multiple resources are required
  – ***prevents deadlock***
- **No preemption:** Lock requested, but threads holding the resources can't be forcibly made to release them
  > use non-blocking lock instructions – ***prevents deadlock***
- **Circular Wait:** circle acquisition of locks - one thread holds what another needs and vice-versa
  >providing total ordering of lock acquisition throughout the code (e.g. L1, L2, L3) – ***prevents deadlock***

16

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- **Chapter 13: Address Spaces**
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

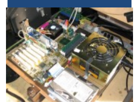
17

CATCH UP FROM LECTURE 13

- Switch to Lecture 12 Slides
- Slides L13.50 to L13.78
  (Chapter 13 –Address Spaces)
  (Chapter 14 – The Memory API)

18

WE WILL RETURN AT 4:49PM



February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.19

19

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- **Chapter 15: Address Translation**
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.20

20

CHAPTER 15: ADDRESS TRANSLATION

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.21

21

OBJECTIVES – 5/18

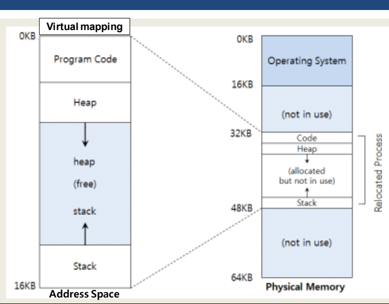
- **Chapter 15: Address translation**
  - Base and bounds
  - HW and OS Support

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.22

22

ADDRESS TRANSLATION

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



February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.23

23

BASE AND BOUNDS

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

$$physical\ address = virtual\ address + base$$

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

$$0 \leq virtual\ address < bounds$$

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.24

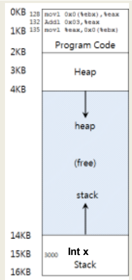
24

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

$physical\ address = virtual\ address + base$



25

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)

26

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.

27

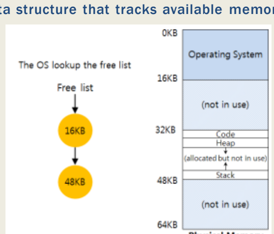
OS SUPPORT FOR MEMORY VIRTUALIZATION

- 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

28

OS: WHEN PROCESS STARTS RUNNING

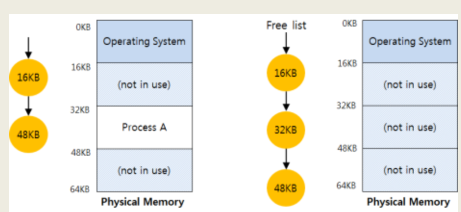
- OS searches for free space for new process
  - Free list: data structure that tracks available memory slots



29

OS: WHEN PROCESS IS TERMINATED

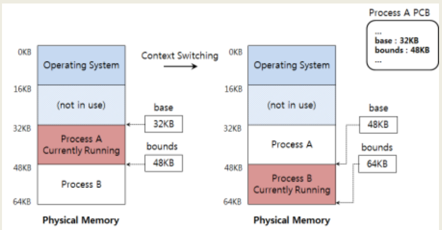
- OS places memory back on the free list



30

OS: WHEN CONTEXT SWITCH OCCURS

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



February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.31

31

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!**

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.32

32

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

33

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- **Chapter 16: Segmentation**
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.34

34

CHAPTER 16: SEGMENTATION

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.35

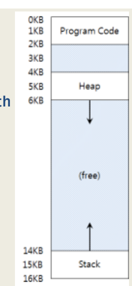
35

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?



February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.36

36

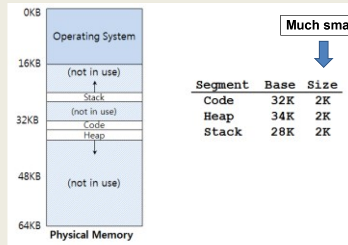
MULTIPLE SEGMENTS

- 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)

37

SEGMENTS IN MEMORY

- Consider 3 segments:



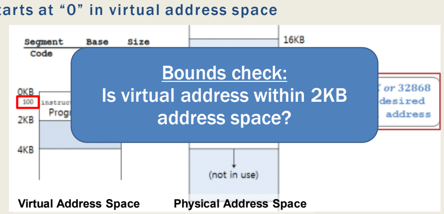
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

38

ADDRESS TRANSLATION: CODE SEGMENT

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

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

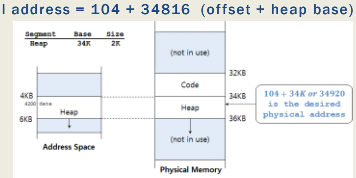


39

ADDRESS TRANSLATION: HEAP

Virtual address + base is not the correct physical address.

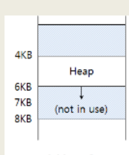
- Heap starts at virtual address 4096
- The data is at 4200
- Offset= 4200 – 4096 = 104 (virt addr – virt heap start)
- Physical address = 104 + 34816 (offset + heap base)



40

SEGMENTATION FAULT

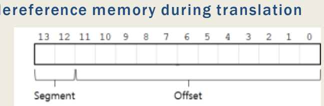
- Access beyond the address space
- Heap starts at virtual address: 4096
- Data pointer is to 7KB (7168)
- Is data pointer valid?
- Heap starts at 4096 + 2 KB seg size = 6144
- Offset= 7168 > 4096 + 2048 (6144)



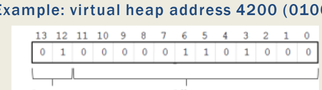
41

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

42

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)
```

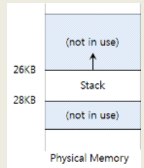
- VIRTUAL ADDRESS = 01000001101000 (on heap)
- SEG_MASK = 0x3000 (11000000000000)
- SEG_SHIFT = 01 → ***heap*** (mask gives us segment code)
- OFFSET_MASK = 0xFFF (00111111111111)
- OFFSET = 000001101000 = 104 (isolates segment offset)
- OFFSET < BOUNDS : 104 < 2048

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.43

43

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

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.44

44

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

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.45

45

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?

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

46

SEGMENTATION GRANULARITY

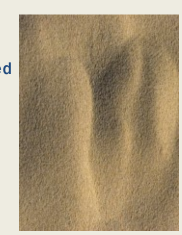
- Coarse-grained
- Manage memory as large purpose based segments:
  - Code segment
  - Heap segment
  - Stack segment

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.47

47

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

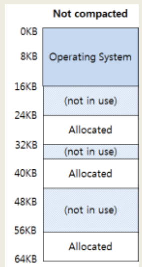


February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.48

48

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?

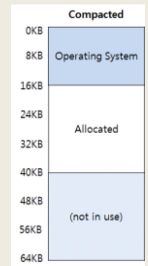


February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.49

49

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)



February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.50

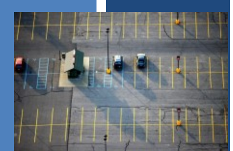
50

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- **Chapter 17: Free Space Management**
- Chapter 18: Introduction to Paging

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.51

51



CHAPTER 17: FREE SPACE MANAGEMENT

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.52

52

OBJECTIVES – 5/18

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

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.53

53

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?

February 26, 2026 | TCSS422: Operating Systems [Winter 2026] School of Engineering and Technology, University of Washington - Tacoma | L14.54

54

FREE SPACE MANAGEMENT

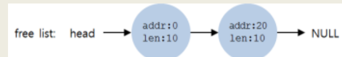
- Management of memory using
- Only fixed-sized units
  - Easy: keep a list
  - Memory request → return first free entry
    - Simple search
- With variable sized units
  - More challenging
  - Results from variable sized malloc requests
  - Leads to fragmentation

55

FRAGMENTATION

- Consider a 30-byte heap

- Request for 15-bytes



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

56

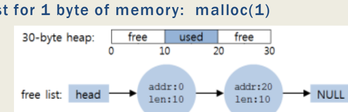
FRAGMENTATION - 2

- **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

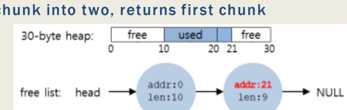
57

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



58

ALLOCATION STRATEGY: COALESCING

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

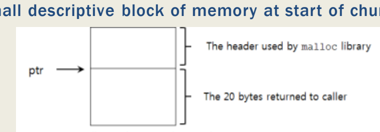
- Request arrives: malloc(30)
- ***SPLIT DOES NOT WORK*** - no contiguous 30-byte chunk exists!
- Coalescing regroups chunks into contiguous chunk

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

59

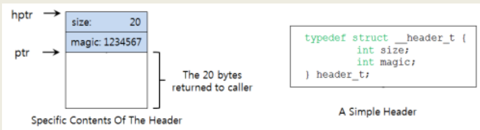
MEMORY HEADERS

- 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



60

MEMORY HEADERS - 2



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

61

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);
}
```


62

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;
```

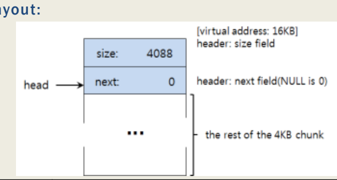

63

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;
```

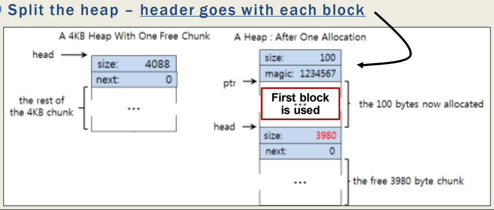
- Heap layout:



64

FREE LIST: MALLOC() CALL

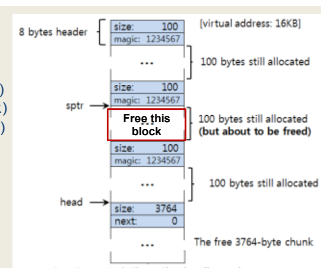
- Consider a request for a 100 bytes: malloc(100)
- Header block requires 8 bytes
  - 4 bytes for size, 4 bytes for magic number
- Split the heap – header goes with each block



65

FREE LIST: FREE() CALL

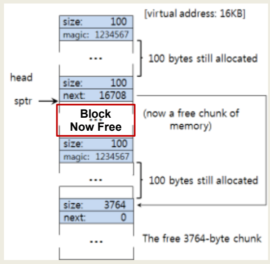
- Addresses of chunks
- Start=16384
  + 108 (end of 1st chunk)
  + 108 (end of 2nd chunk)
  + 108 (end of 3rd chunk)
  = 16708



66

FREE LIST: FREE() CHUNK #2

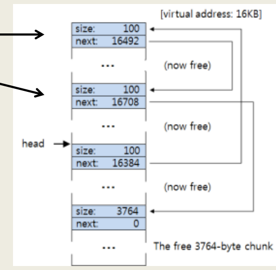
- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
  - addr – sizeof(node_t)
- Actual start of chunk #2
  - 16492



67

FREE LIST- FREE ALL CHUNKS

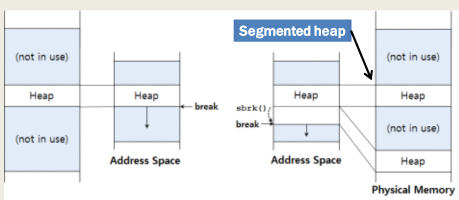
- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed



68

GROWING THE HEAP

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



69

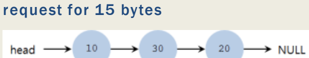
MEMORY ALLOCATION STRATEGIES

- **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

70

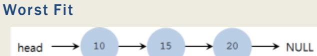
EXAMPLES

- Allocation request for 15 bytes



- Result of Best Fit

- Result of Worst Fit



71

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

72

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

73

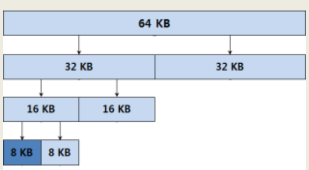
SEGREGATED LISTS

- 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

74

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

75

BUDDY ALLOCATION - 2

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

76

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

77

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

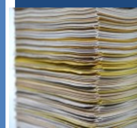
78

OBJECTIVES – 2/26

- Questions from 2/24
- Assignment 2 - March 12 AOE
- Quiz 3-Sync Array; Memory Segmentation Activity; Quiz 4
- Tutorial 2 – Pthread/locks/conditions tutorial-3/5 AOE
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- **Chapter 18: Introduction to Paging**

79

CHAPTER 18: INTRODUCTION TO PAGING



80

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

81

ADVANTAGES OF PAGING

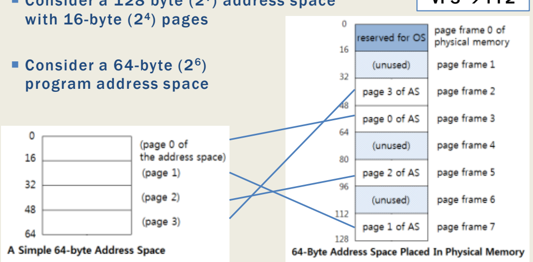
- Flexibility
  - Abstracts the process address space into pages
  - No need to track direction of HEAP / STACK growth
    - *Just add more pages...*
  - No need to store unused space
    - *As with segments...*
- Simplicity
  - Pages and page frames are the same size
  - Easy to allocate and keep a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

82

PAGING: EXAMPLE

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

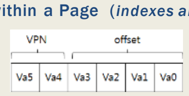
- Consider a 128 byte (2^7) address space with 16-byte (2^4) pages
- Consider a 64-byte (2^6) program address space



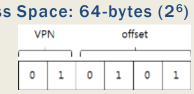
83

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)



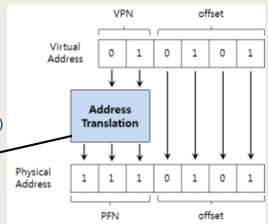
Here program can have just four pages...

84

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



85

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?

86

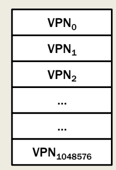
(1) WHERE ARE PAGE TABLES STORED?

- 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

87

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)



- 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

88

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

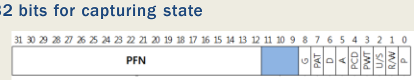
400 MB / 4000 GB

- **Is this efficient?**

89

(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

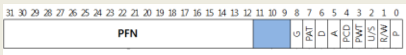


An x86 Page Table Entry(PTE)

90

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)

91

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

92

(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

93

(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)

94

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)
```


95

COUNTING MEMORY ACCESSES

- Example: Use this Array initialization Code

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

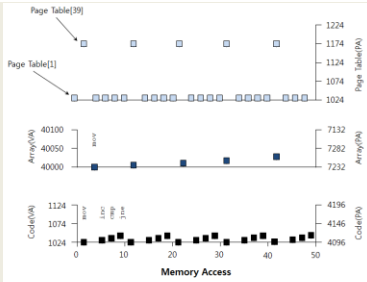
- Assembly equivalent:

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


96

VISUALIZING MEMORY ACCESSES: FOR THE FIRST 5 LOOP ITERATIONS

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



97

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

98

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

99

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

100

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

101

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

102

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?

103

QUESTIONS

104