


<h1>TCSS 422: OPERATING SYSTEMS</h1>		
<h2>Memory Virtualization with Segments, Introduction to Paging</h2>		
<p>Wes J. Lloyd School of Engineering and Technology University of Washington - Tacoma</p>		
<p>May 14, 2024</p>	<p>TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington</p>	<p>Tacoma</p>

1

<h2>MIDTERM REVIEW SESSION</h2>		
<ul style="list-style-type: none">▪ Make-up midterm exams are scheduled and will be completed by the end of Friday this week▪ Midterm Review Session:<ul style="list-style-type: none">▪ Tuesday May 21, 6:00 pm (during office hour, from BHS106)▪ Via Zoom / Live Stream / Recording▪ Will discuss and review midterm exam problems and grading		
<p>May 14, 2024</p>	<p>TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma</p>	<p>L14.2</p>

2

OBJECTIVES – 5/14

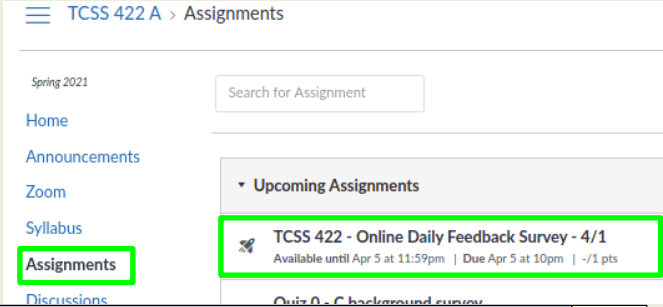
- **Questions from 5/9**
- Assignment 2 - May 31
- Quiz 3 – Class Activity-Synchronized Array - Thursday
- Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation
- Chapter 17: Free Space Management
- Chapter 18: Introduction to Paging

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.3
--------------	---	-------

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



The screenshot shows the Canvas LMS interface for 'TCSS 422 A > Assignments'. A sidebar on the left contains navigation links: Home, Announcements, Zoom, Syllabus, **Assignments** (highlighted), and Discussions. The main content area shows 'Upcoming Assignments' with a list item: 'TCSS 422 - Online Daily Feedback Survey - 4/1' (highlighted), which is 'Available until Apr 5 at 11:59pm | Due Apr 5 at 10pm | -/1 pts'.

May 14, 2024	TCSS422: Computer Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.4
--------------	--	-------

4

TCSS 422 - Online Daily Feedback Survey - 4/1

Quiz Instructions

Question 1 0.5 pts

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

1 2 3 4 5 6 7 8 9 10

Mostly Review To Me Equal New and Review Mostly New To Me

Question 2 0.5 pts

Please rate the pace of today's class:

1 2 3 4 5 6 7 8 9 10

Slow Just Right Fast

May 14, 2024 TCSS422: Computer Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L14.5

5

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (25 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average - 6.68 (↑ - previous 6.58)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average - 5.28 (↓ - previous 5.31)**

May 14, 2024 TCSS422: Computer Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma L14.6

6

FEEDBACK FROM 5/9

- **Is list insertion the only deadlock prevention for mutual exclusion?**
- List insertion is not a deadlock prevention technique in Chapter 32
- In lecture 13, the “mutual exclusion” cause for deadlock is when critical sections of code are protected with locks, and for some reason, the lock is never available
- The solution is to remove the use of locks where possible by replacing locks with an atomic implementation of the CompareAndSwap CPU instruction (**assembly language**)
- Atomic CompareAndSwap (assembly) can be used to eliminate the use locks as shown Chapter 32 examples:
 - Increment a counter variable atomically
 - Insert an item into a list

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.7
--------------	---	-------

7

FEEDBACK - 2

- **Is list insertion the only deadlock prevention for mutual exclusion?**
- **KEY TAKEHOME MESSAGE** from Chapter 32:
 - Protecting critical code sections with locks is the “Mutual Exclusion” cause for deadlock in Chapter 32
 - The solution is to eliminate locks to remove the requirement for mutual exclusion in high-level program code (C)
 - Locks can be replaced with atomic CPU instructions (CompareAndSwap) or atomic data types can be used
 - E.g. lock-free data structures in Java

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.8
--------------	---	-------

8

OBJECTIVES – 5/14		
<ul style="list-style-type: none">▪ Questions from 5/9▪ Assignment 2 - May 31▪ Quiz 3 – Activity-Synchronized Array - Thursday▪ Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24▪ 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		
May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.9

9

OBJECTIVES – 5/14		
<ul style="list-style-type: none">▪ Questions from 5/9▪ Assignment 2 - May 31▪ Quiz 3 – Activity-Synchronized Array - Thursday▪ Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24▪ 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		
May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.10

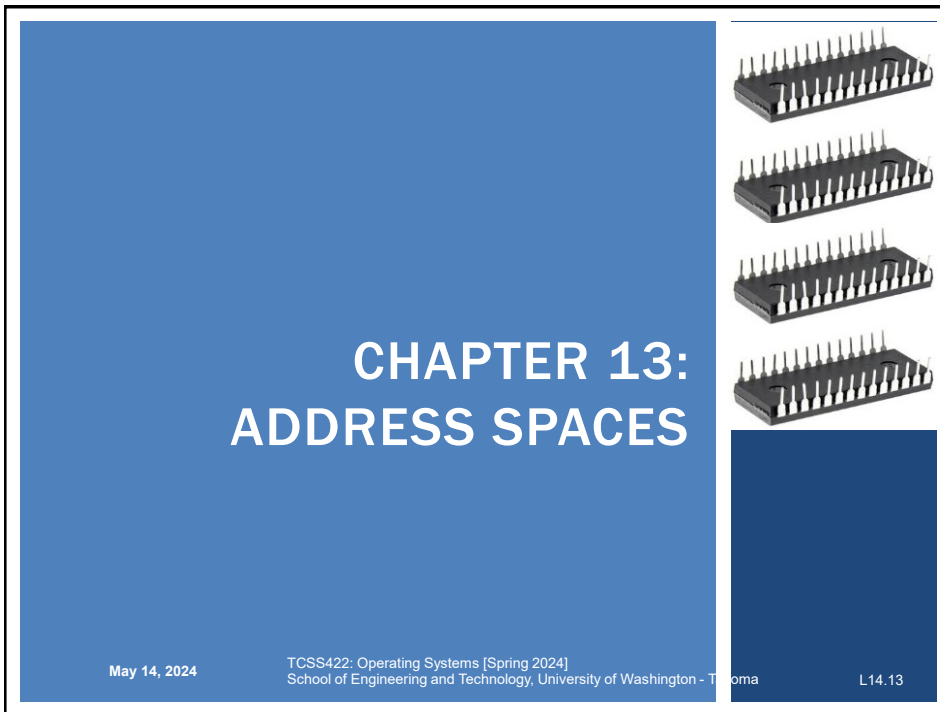
10

OBJECTIVES – 5/14		
<ul style="list-style-type: none">▪ Questions from 5/9▪ Assignment 2 - May 31▪ Quiz 3 – Activity-Synchronized Array - Thursday▪ Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24▪ 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		
May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.11

11

OBJECTIVES – 5/14		
<ul style="list-style-type: none">▪ Questions from 5/9▪ Assignment 2 - May 31▪ Quiz 3 – Activity-Synchronized Array - Thursday▪ Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24▪ 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		
May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.12

12



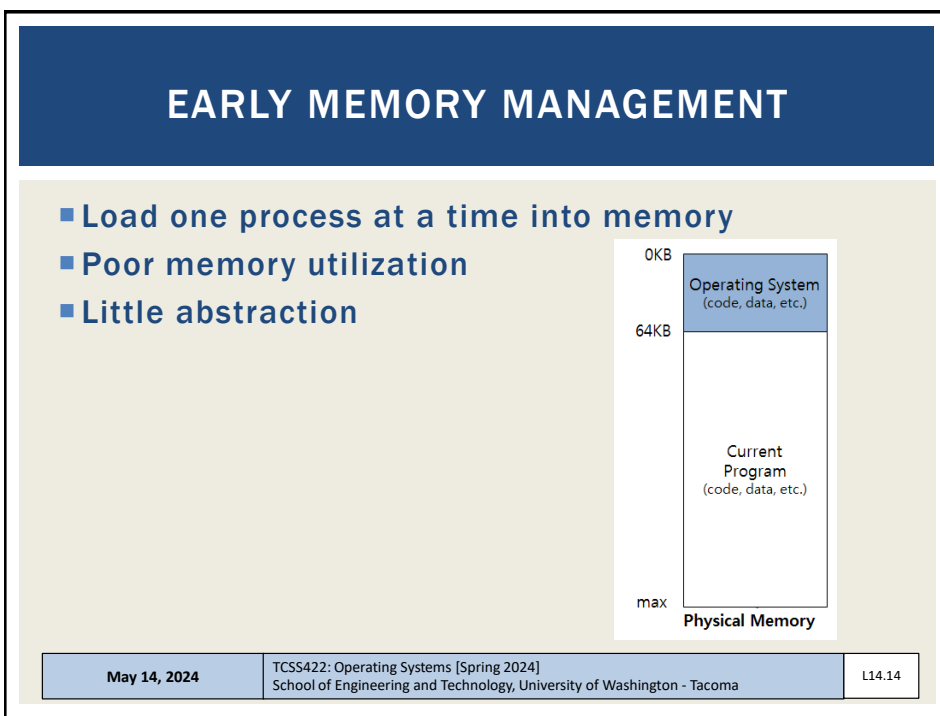
CHAPTER 13: ADDRESS SPACES

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

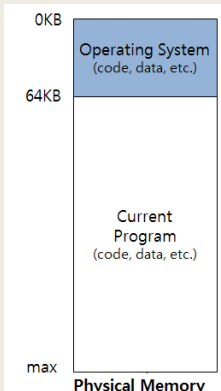
L14.13

13



EARLY MEMORY MANAGEMENT

- Load one process at a time into memory
- Poor memory utilization
- Little abstraction



0KB

64KB

max

Physical Memory

Operating System
(code, data, etc.)

Current Program
(code, data, etc.)

May 14, 2024

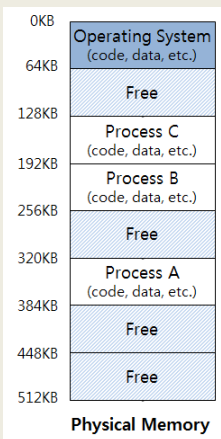
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.14

14

MULTIPROGRAMMING WITH SHARED MEMORY

- Later machines supported running multiple processes
- Swap out processes during I/O waits to increase system utilization and efficiency
- Swap entire memory of a process to disk for context switch
- Too slow, especially for large processes
- Solution →
 - Leave processes in memory
- Need to protect from errant memory accesses in a multiprocessing environment



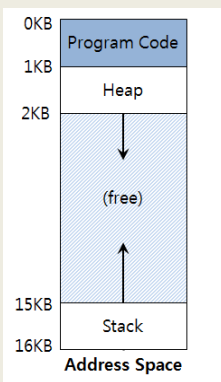
The diagram shows a vertical stack of memory blocks. From top to bottom: Operating System (code, data, etc.) from 0KB to 64KB; Free space from 64KB to 128KB; Process C (code, data, etc.) from 128KB to 192KB; Process B (code, data, etc.) from 192KB to 256KB; Free space from 256KB to 320KB; Process A (code, data, etc.) from 320KB to 384KB; Free space from 384KB to 448KB; Free space from 448KB to 512KB. The total is labeled 'Physical Memory'.

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.15
--------------	---	--------

15

ADDRESS SPACE

- Easy-to-use abstraction of physical memory for a process
- Main elements:
 - Program code
 - Stack
 - Heap
- Example: 16KB address space



The diagram shows a vertical stack of memory blocks. From top to bottom: Program Code from 0KB to 1KB; Heap from 1KB to 2KB; (free) space from 2KB to 15KB; Stack from 15KB to 16KB. Arrows indicate the direction of growth: the heap grows downwards and the stack grows upwards. The total is labeled 'Address Space'.

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.16
--------------	---	--------

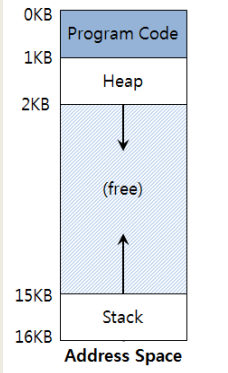
16

ADDRESS SPACE - 2

- **Code**
 - Program code

- **Stack**
 - Program counter (PC)
 - Local variables
 - Parameter variables
 - Return values (for functions)

- **Heap**
 - Dynamic storage
 - Malloc() new()



The diagram illustrates the address space layout. It is a vertical stack of memory regions. At the top is 'Program Code' from 0KB to 1KB. Below it is 'Heap' from 1KB to 2KB. A large shaded area labeled '(free)' spans from 2KB to 15KB. Below that is 'Stack' from 15KB to 16KB. A downward arrow is shown in the free space, and an upward arrow is shown in the stack area. The label 'Address Space' is at the bottom.

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.17
--------------	---	--------

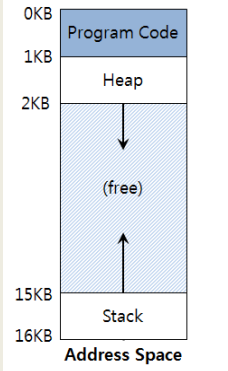
17

ADDRESS SPACE - 3

- **Program code**
 - Static size

- **Heap and stack**
 - Dynamic size
 - Grow and shrink during program execution
 - Placed at opposite ends

- **Addresses are virtual**
 - They must be physically mapped by the OS



The diagram illustrates the address space layout, identical to the previous slide. It shows 'Program Code' (0KB-1KB), 'Heap' (1KB-2KB), '(free)' space (2KB-15KB), and 'Stack' (15KB-16KB) with arrows indicating growth directions. The label 'Address Space' is at the bottom.

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.18
--------------	---	--------

18

VIRTUAL ADDRESSING

- Every address is virtual
 - OS translates virtual to physical addresses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code : %p\n", (void *) main);
    printf("location of heap : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

- **EXAMPLE: virtual.c**

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.19
--------------	---	--------

19

VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:

location of code: 0x400686
location of heap: 0x1129420
location of stack: 0x7ffe040d77e4

The diagram illustrates the virtual address space layout. It is divided into several segments: Code (Text) at 0x400000, Data at 0x401000, Heap at 0xc2000, a (free) region at 0xd13000, stack at 0x7ff9ca28000, and Stack at 0x7ff9ca49000. Arrows indicate the direction of memory growth: the heap grows downwards from 0xc2000, and the stack grows upwards from 0x7ff9ca49000.

Address Space

0x400000 Code (Text)

0x401000 Data

0xc2000 Heap

0xd13000

↓ heap

(free)

↑ stack

0x7ff9ca28000 Stack

0x7ff9ca49000

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.20
--------------	---	--------

20

GOALS OF OS MEMORY VIRTUALIZATION

- **Transparency**
 - Memory shouldn't appear virtualized to the program
 - OS multiplexes memory among different jobs behind the scenes

- **Protection**
 - Isolation among processes
 - OS itself must be isolated
 - One program should not be able to affect another (or the OS)

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.21
--------------	---	--------

21

GOALS - 2

- **Efficiency**
 - **Time**
 - Performance: virtualization must be fast

 - **Space**
 - Virtualization must not waste space
 - Consider data structures for organizing memory
 - Hardware support TLB: Translation Lookaside Buffer

- *Goals considered when evaluating memory virtualization schemes*

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.22
--------------	---	--------

22


OBJECTIVES – 5/14

- Questions from 5/9
- Assignment 2 - May 31
- Quiz 3 – Activity-Synchronized Array - Thursday
- Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24
- 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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.23
--------------	---	--------

23

CHAPTER 14: THE MEMORY API



May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.24
--------------	---	--------

24

OBJECTIVES – 5/18

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.25
--------------	---	--------

25

MALLOC

```
#include <stdlib.h>

void* malloc(size_t size)
```

- **Allocates memory on the heap**
- **size_t** unsigned integer (must be +)
- **size** size of memory allocation in bytes

- **Returns**
- **SUCCESS:** A void * to a memory address
- **FAIL:** NULL

- **sizeof()** often used to ask the system how large a given datatype or struct is

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.26
--------------	---	--------

26

sizeof()

- Not safe to assume data type sizes using different compilers, systems
- Dynamic array of 10 ints
- Static array of 10 ints

```
int *x = malloc(10 * sizeof(int));  
printf("%d\n", sizeof(x));
```

4

```
int x[10];  
printf("%d\n", sizeof(x));
```

40

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.27
--------------	---	--------

27

FREE()

```
#include <stdlib.h>  
  
void free(void* ptr)
```

- Free memory allocated with malloc()
- Provide: (void *) ptr to malloc'd memory
- Returns: nothing

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.28
--------------	---	--------

28

```
#include<stdio.h>

int * set_magic_number_a()
{
    int a =53247;
    return &a;
}

void set_magic_number_b()
{
    int b = 11111;
}

int main()
{
    int * x = NULL;
    x = set_magic_number_a();
    printf("The magic number is=%d\n",*x);
    set_magic_number_b();
    printf("The magic number is=%d\n",*x);
    return 0;
}
```

What will this code do?

29

29

```
#include<stdio.h>

int * set_magic_number_a()
{
    int a =53247;
    return &a;
}

void set_magic_number_b()
{
    int b = 11111;
}

int main()
{
    int * x = NULL;
    x = set_magic_number_a();
    printf("The magic number is=%d\n",*x);
    set_magic_number_b();
    printf("The magic number is=%d\n",*x);
    return 0;
}
```

What will this code do?

Output:
\$./pointer_error
The magic number is=53247
The magic number is=11111

We have not changed *x but
the value has changed!!

Why?

30

30

DANGLING POINTER (1/2)

- Dangling pointers arise when a variable referred (a) goes “out of scope”, and it’s memory is destroyed/overwritten (by b) without modifying the value of the pointer (*x).
- The pointer still points to the original memory location of the deallocated memory (a), which has now been reclaimed for (b).

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.31

31

DANGLING POINTER (2/2)

- Fortunately in the case, a compiler warning is generated:

```
$ g++ -o pointer_error -std=c++0x pointer_error.cpp
```

```
pointer_error.cpp: In function ‘int*  
set_magic_number_a()’:  
pointer_error.cpp:6:7: warning: address of local  
variable ‘a’ returned [enabled by default]
```

- This is a common mistake - - -
accidentally referring to addresses that have gone “out of scope”

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.32

32

CALLOC()

```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate “C”lear memory on the heap
- Calloc wipes memory in advance of use...
- `size_t num` : number of blocks to allocate
- `size_t size` : size of each block(in bytes)

- Calloc() prevents...

```
char *dest = malloc(20);
printf("dest string=%s\n", dest);

dest string=◆◆F
```

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.33
--------------	---	--------

33

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.34
--------------	---	--------

34

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.35
--------------	---	--------

35

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.36
--------------	---	--------

36

**WE WILL RETURN AT
5:07PM**



May 14, 2024 TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma L14.37


37

OBJECTIVES – 5/14

- Questions from 5/9
- Assignment 2 - May 31
- Quiz 3 – Activity-Synchronized Array - Thursday
- Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24
- 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

May 14, 2024 TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma L14.38

38



CHAPTER 15: ADDRESS TRANSLATION

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.39

39

OBJECTIVES - 5/18

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.40

40

ADDRESS TRANSLATION

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.41

41

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:

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.42

42

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma

L14.43

The diagram illustrates memory management. It shows a vertical axis from 0KB to 16KB. Program Code is located between 0KB and 1KB. The Heap is between 2KB and 4KB. A 'heap' pointer is shown with a downward arrow. A '(free)' region is between 4KB and 14KB. The 'stack' is between 14KB and 16KB, with an upward arrow. An 'Int x' variable is located at 15KB, with a value of 3000. The stack grows downwards from 16KB.

43

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)

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma

L14.44

44

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.

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma
L14.45

45

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

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
 School of Engineering and Technology, University of Washington - Tacoma
L14.46

46

OS: WHEN PROCESS STARTS RUNNING

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

The OS lookup the free list

Free list

↓

16KB

↓

48KB

0KB
16KB
32KB
48KB
64KB

Operating System
(not in use)
Code
Heap
(allocated but not in use)
Stack
(not in use)
Physical Memory

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.47

47

OS: WHEN PROCESS IS TERMINATED

- OS places memory back on the free list

0KB
16KB
32KB
48KB
64KB

Operating System
(not in use)
Process A
(not in use)
Physical Memory

Free list

↓

16KB

↓

32KB

↓

48KB

0KB
16KB
32KB
48KB
64KB

Operating System
(not in use)
(not in use)
(not in use)
Physical Memory

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.48

48

OS: WHEN CONTEXT SWITCH OCCURS

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

The diagram illustrates the state of physical memory during a context switch. On the left, labeled 'Physical Memory', the memory is divided into segments: Operating System (0KB-16KB), (not in use) (16KB-32KB), Process A Currently Running (32KB-48KB), and Process B (48KB-64KB). Process A's base register is at 32KB and its bounds register is at 48KB. An arrow labeled 'Context Switching' points to the right. On the right, the memory state is: Operating System (0KB-16KB), (not in use) (16KB-32KB), Process A (32KB-48KB), and Process B Currently Running (48KB-64KB). Process A's base register is now at 48KB and its bounds register is at 64KB. A callout box labeled 'Process A PCB' shows '... base : 32KB' and '... bounds : 48KB', indicating that the OS has saved the original registers of Process A into its PCB.

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.49
--------------	---	--------

49

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!**

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.50
--------------	---	--------

50

Consider a 64KB computer that 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 polllev.com/app

51


OBJECTIVES – 5/14

- Questions from 5/9
- Assignment 2 - May 31
- Quiz 3 – Activity-Synchronized Array - Thursday
- Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24
- 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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.52
--------------	---	--------

52

CHAPTER 16: SEGMENTATION



May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.53

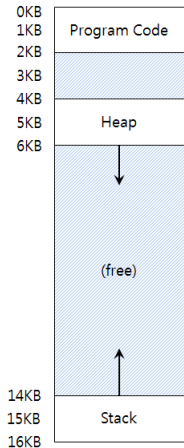
53

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.54

54

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)

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.55

55

SEGMENTS IN MEMORY

- Consider 3 segments:

Much smaller

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.56

56

ADDRESS TRANSLATION: CODE SEGMENT

$physical\ address = offset + base$

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

Segment	Base	Size
Code		16KB

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

or 32868
desired
address

Virtual Address Space
Physical Address Space

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.57

57

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

104 + 34K or 34920
is the desired
physical address

Address Space
Physical Memory

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.58

58

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 $4096 + 2 \text{ KB seg size} = 6144$
- $\text{Offset} = 7168 > 4096 + 2048 (6144)$

Address Space

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.59
--------------	---	--------

59

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.60
--------------	---	--------

60

SEGMENTATION DEREFERENCE

```

1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     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**

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.61
--------------	---	--------

61

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.62
--------------	---	--------

62

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

May 14, 2024TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - TacomaL14.63

63

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 polllev.com/app

64

SEGMENTATION GRANULARITY

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




May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.65
--------------	---	--------

65

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



May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.66
--------------	---	--------

66

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?

Not compacted

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.67

67

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)

Compacted

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.68


68

OBJECTIVES – 5/14

- Questions from 5/9
- Assignment 2 - May 31
- Quiz 3 – Activity-Synchronized Array - Thursday
- Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24
- 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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.69
--------------	---	--------

69



CHAPTER 17: FREE SPACE MANAGEMENT

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.70
--------------	---	--------

70

OBJECTIVES – 5/18

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.71
--------------	---	--------

71

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?

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.72
--------------	---	--------

72

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.73
--------------	---	--------

73

FRAGMENTATION

- Consider a 30-byte heap

30-byte heap: free used free

0	10	20
		30

- Request for 15-bytes

free list: head →

addr:0 len:10	→	addr:20 len:10	→	NULL
------------------	---	-------------------	---	------

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.74
--------------	---	--------

74

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.75
--------------	---	--------

75

ALLOCATION STRATEGY: SPLITTING

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

30-byte heap: free used free

0 10 20 30

free list: head → addr:0
len:10 → addr:20
len:10 → NULL

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

30-byte heap: free used free

0 10 20 21 30

free list: head → addr:0
len:10 → addr:21
len:9 → NULL

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.76
--------------	---	--------

76

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.77
--------------	---	--------

77

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.78
--------------	---	--------

78

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

May 14, 2024TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - TacomaL14.79

79

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);
}
```

May 14, 2024TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - TacomaL14.80

80

THE FREE LIST

- Simple free list struct

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

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

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.81
--------------	---	--------

81

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:

[virtual address: 16KB]
header: size field

size: 4088		
next: 0		header: next field(NULL is 0)
...		the rest of the 4KB chunk

head →

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.82
--------------	---	--------

82

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

A 4KB Heap With One Free Chunk

head →

size:	4088
next:	0
...	

the rest of the 4KB chunk

A Heap : After One Allocation

ptr →

size:	100
magic:	1234567
First block is used	
}	
the 100 bytes now allocated	
head →	
size:	3980
next:	0
...	
}	
the free 3980 byte chunk	

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.83

83

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

8 bytes header

size:	100
magic:	1234567
...	

100 bytes still allocated

ptr →

size:	100
magic:	1234567
Free this block	
}	
100 bytes still allocated (but about to be freed)	

head →

size:	100
magic:	1234567
...	
}	
100 bytes still allocated	
head →	
size:	3764
next:	0
...	
}	
The free 3764-byte chunk	

Free Space With Three Chunks Allocated

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.84

84

FREE LIST: FREE() CHUNK #2

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.85

85

FREE LIST- FREE ALL CHUNKS

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.86

86

GROWING THE HEAP

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

The diagram illustrates the process of growing the heap. It shows three stages of memory allocation:

- Initial State:** A small heap is shown in the address space, with a 'break' pointer at the top. The memory below the heap is labeled '(not in use)'. The physical memory below is also labeled '(not in use)'.
- Heap Growth:** The heap grows larger, and the 'break' pointer moves up. The physical memory below is still labeled '(not in use)'.
- Segmented Heap:** The heap is now segmented. The 'break' pointer is at the top, and the 'sbrk()' pointer is at the bottom. The physical memory below is labeled '(not in use)'. A blue box labeled 'Segmented heap' points to this stage.

At the bottom of the slide, there is a footer with the date 'May 14, 2024', the course information 'TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma', and the slide number 'L14.87'.

87




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

At the bottom of the slide, there is a footer with the date 'May 14, 2024', the course information 'TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma', and the slide number 'L14.88'.

88

EXAMPLES

- **Allocation request for 15 bytes**

- **Result of Best Fit**

- **Result of Worst Fit**


May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.89
--------------	---	--------

89

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.90
--------------	---	--------

90

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 polllev.com/app

91

SEGREGATED LISTS

- For popular sized requests
e.g. for kernel objects such as locks, inodes, etc.
- Manage as segregated free lists
- Provide object caches: stores pre-initialized objects
- How much memory should be dedicated for specialized requests (object caches)?
- If a given cache is low in memory, can request “*slabs*” of memory from the general allocator for caches.
- General allocator will reclaim slabs when not used

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.92
--------------	---	--------

92

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

The diagram illustrates the binary buddy allocation process. It starts with a single 64 KB block. This block is split into two 32 KB blocks. The left 32 KB block is further split into two 16 KB blocks. The left 16 KB block is split into two 8 KB blocks. The right 8 KB block is highlighted in a darker blue. Below the diagram, it states "64KB free space for 7KB request".

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.93
--------------	---	--------

93

BUDDY ALLOCATION - 2

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.94
--------------	---	--------

94

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 polllev.com/app

95

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 polllev.com/app

96


OBJECTIVES – 5/14

- Questions from 5/9
- Assignment 2 - May 31
- Quiz 3 – Activity-Synchronized Array - Thursday
- Tutorial 2 – Pthread/locks/conditions tutorial-Fri May 24
- 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**

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.97
--------------	---	--------

97

CHAPTER 18: INTRODUCTION TO PAGING



May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.98
--------------	---	--------

98

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.99
--------------	---	--------

99

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.100
--------------	---	---------

100

PAGING: EXAMPLE

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

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

A Simple 64-byte Address Space

0	
16	(page 0 of the address space) (page 1)
32	(page 2)
48	(page 3)
64	

64-Byte Address Space Placed In Physical Memory

0	reserved for OS	page frame 0 of physical memory
16	(unused)	page frame 1
32	page 3 of AS	page frame 2
48	page 0 of AS	page frame 3
64	(unused)	page frame 4
80	page 2 of AS	page frame 5
96	(unused)	page frame 6
112	page 1 of AS	page frame 7
128		

May 14, 2024
TCCS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.101

101

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*)

VPN			offset			
Va5	Va4	Va3	Va2	Va1	Va0	

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

VPN			offset			
0	1	0	1	0	1	

Here program can have just four pages...

May 14, 2024
TCCS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.102

102

EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte (2^6) program address space (4 pages $\rightarrow 2^2$)
- Stored in 128-byte (2^7) physical memory (8 frames $\rightarrow 2^3$)
- 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 \rightarrow PF3
 VP1 \rightarrow PF7
 VP2 \rightarrow PF5
 VP3 \rightarrow PF2

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.103

103

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?

May 14, 2024
TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma
L14.104

104

(1) WHERE ARE PAGE TABLES STORED?

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.105
--------------	---	---------

105

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

VPN ₀
VPN ₁
VPN ₂
...
...
VPN ₁₀₄₈₅₇₆

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.106
--------------	---	---------

106

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 $4\text{MB} \times 100 = 400\text{MB}$
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

$400\text{ MB} / 4000\text{ GB}$

- Is this efficient?

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.107
--------------	---	---------

107

(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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN												G	PAT	D	A	PCD	PWT	U/S	R/W	P											

An x86 Page Table Entry(PTE)

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.108
--------------	---	---------

108

PAGE TABLE ENTRY

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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																						G	PAT	D	A	PCD	PWT	U/S	R/W	P	

An x86 Page Table Entry(PTE)

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.109

109

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

May 14, 2024

TCSS422: Operating Systems [Spring 2024]
School of Engineering and Technology, University of Washington - Tacoma

L14.110

110

(3) HOW BIG ARE PAGE TABLES?

- Page tables are too big to store on the CPU
- Page tables are stored using physical memory
- Paging supports efficiently storing a sparsely populated address space
 - Reduced memory requirement
Compared to base and bounds, and segments

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.111
--------------	---	---------

111

(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

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

Stored in RAM →

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.112
--------------	---	---------

112

PAGING MEMORY ACCESS

```

1. // Extract the VPN from the virtual address
2. VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3.
4. // Form the address of the page-table entry (PTE)
5. PTEAddr = PTBR + (VPN * sizeof(PTE))
6.
7. // Fetch the PTE
8. PTE = AccessMemory(PTEAddr)
9.
10. // Check if process can access the page
11. if (PTE.Valid == False)
12.     RaiseException(SEGMENTATION_FAULT)
13. else if (CanAccess(PTE.ProtectBits) == False)
14.     RaiseException(PROTECTION_FAULT)
15. else
16.     // Access is OK: form physical address and fetch it
17.     offset = VirtualAddress & OFFSET_MASK
18.     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19.     Register = AccessMemory(PhysAddr)
    
```

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.113
--------------	---	---------

113

COUNTING MEMORY ACCESSES

■ Example: Use this Array initialization Code

```

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

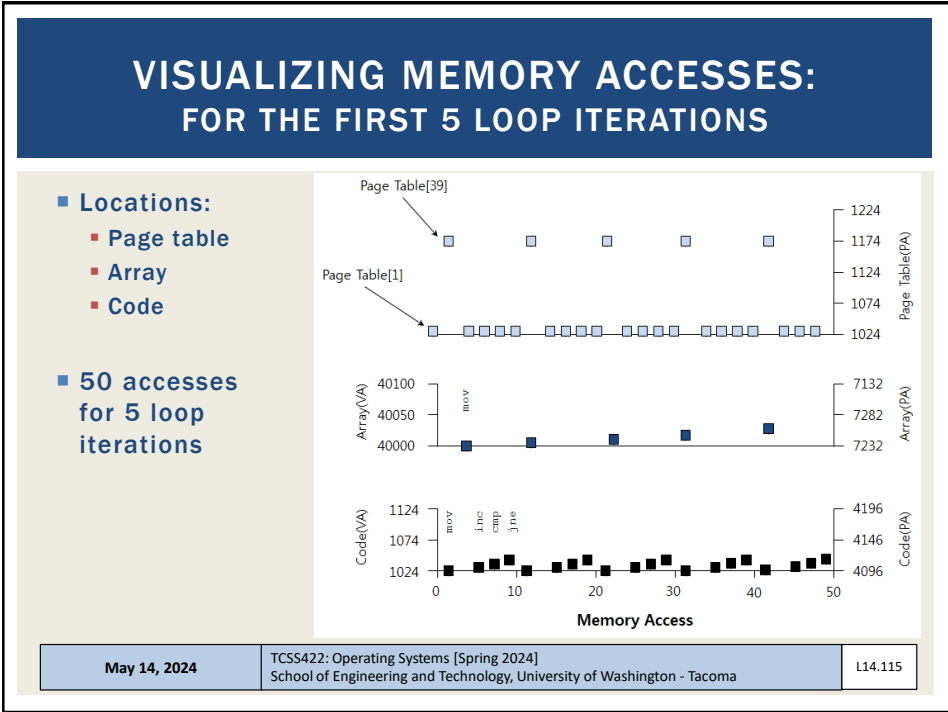
■ Assembly equivalent:

```

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.114
--------------	---	---------

114



115

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 polllev.com/app

116

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

May 14, 2024 TCSS422: Operating Systems (Spring 2024) L14.17

117

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

118

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

May 14, 2024 TCSS422: Operating Systems (Spring 2024) L14.19

119

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

$4\text{ GB} / 16\text{ KB} = 65,536$

$4\text{ GB} / 64\text{ MB} = 256$

$4\text{GB} / 256\text{ KB} = 16,384$

$4\text{GB} / 4\text{MB} = 1,024$

None of the above

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

120

PAGING SYSTEM EXAMPLE


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

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

May 14, 2024	TCSS422: Operating Systems [Spring 2024] School of Engineering and Technology, University of Washington - Tacoma	L14.121
--------------	---	---------

121

QUESTIONS



122