


TCCS 422: OPERATING SYSTEMS

**Concurrency Problems,
Memory Virtualization
with Segments**

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



May 18, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
Tacoma

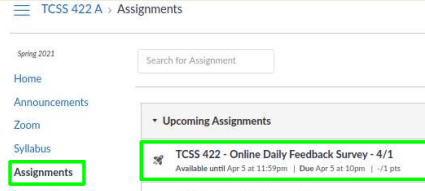
OBJECTIVES – 5/18

- **Questions from 5/13**
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- **Chapter 16: Segmentation**

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L14.2

ONLINE DAILY FEEDBACK SURVEY

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



May 18, 2021
TCCS422: Computer Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L14.3

TCCS 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 18, 2021
TCCS422: Computer Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L14.4

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (51 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.32 (↓ - previous 7.77)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.63 (↓ - previous 5.81)**

May 18, 2021
TCCS422: Computer Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L14.5

FEEDBACK

- **Could you review how execution trace table shows different results with while loop(condition)?**

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
 School of Engineering and Technology, University of Washington - Tacoma
L14.6

EXECUTION TRACE: ISSUE: NO WHILE 1 PRODUCER, 2 CONSUMERS, 1 CONDITION

■ Two threads

Legend
 c1/p1- lock
 c2/p2- check var
 c3/p3- wait
 c4- get()
 p4- put()
 c5/p5- signal
 c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running	Ready	Ready	Ready	Ready	0	
c2	Running	Ready	Ready	Ready	Ready	0	
c3	Sleep	Ready	Ready	Ready	Ready	0	Nothing to get
c3	Sleep	Ready	Ready	p1	Running	0	
c3	Sleep	Ready	Ready	p2	Running	0	
c3	Sleep	Ready	Ready	p4	Running	1	Buffer now full
c3	Sleep	Ready	Ready	p5	Running	1	T_{c1} awoken
c3	Sleep	Ready	Ready	p6	Running	1	
c3	Sleep	Ready	Ready	p1	Running	1	
c3	Sleep	Ready	Ready	p2	Running	1	
c3	Sleep	Ready	Ready	p3	Sleep	1	Buffer full; sleep
c3	Sleep	Ready	Ready	p1	Running	1	T_{c2} sneaks in ...
c3	Sleep	Ready	Ready	p2	Running	1	
c3	Sleep	Ready	Ready	p3	Sleep	1	... and grabs data
c3	Sleep	Ready	Ready	p4	Running	0	T_p awoken
c3	Sleep	Ready	Ready	p5	Running	0	
c3	Sleep	Ready	Ready	p6	Running	0	Oh oh! No data

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.7

EXECUTION TRACE: ISSUE: 1 CONDITION WHILE, 1 PRODUCER, 2 CONSUMERS

Legend
 c1/p1- lock
 c2/p2- check var
 c3/p3- wait
 c4- get()
 p4- put()
 c5/p5- signal
 c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running	Ready	Ready	Ready	Ready	0	
c2	Running	Ready	Ready	Ready	Ready	0	
c3	Sleep	Ready	Ready	Ready	Ready	0	Nothing to get
c3	Sleep	c1	Running	Ready	Ready	0	
c3	Sleep	c2	Running	Ready	Ready	0	
c3	Sleep	c3	Sleep	Ready	Ready	0	Nothing to get
c3	Sleep	p1	Running	Ready	Ready	0	
c3	Sleep	p2	Running	Ready	Ready	0	
c3	Sleep	p4	Running	Ready	Ready	1	Buffer now full
c3	Sleep	p5	Running	Ready	Ready	1	T_{c1} awoken
c3	Sleep	p6	Running	Ready	Ready	1	
c3	Sleep	p1	Running	Ready	Ready	1	
c3	Sleep	p2	Running	Ready	Ready	1	
c3	Sleep	p3	Sleep	Ready	Ready	1	Must sleep (full)
c3	Sleep	p4	Running	Ready	Ready	1	Recheck condition
c3	Sleep	p5	Running	Ready	Ready	0	T_{c1} grabs data
c3	Sleep	p6	Running	Ready	Ready	1	
c3	Sleep	p1	Running	Ready	Ready	1	
c3	Sleep	p2	Running	Ready	Ready	1	
c3	Sleep	p3	Sleep	Ready	Ready	1	
c3	Sleep	p4	Running	Ready	Ready	1	
c3	Sleep	p5	Running	Ready	Ready	0	Oops! Woke T_{c2}
c3	Sleep	p6	Running	Ready	Ready	0	

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.8

EXECUTION TRACE: ISSUE: 1 CONDITION - 2 WHILE, 1 PRODUCER, 2 CONSUMERS

■ T_{c2} runs, no data to consume

Legend
 c1/p1- lock
 c2/p2- check var
 c3/p3- wait
 c4- get()
 p4- put()
 c5/p5- signal
 c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
...	(cont)
c6	Running	Ready	Ready	Sleep	Sleep	0	
c1	Running	Ready	Ready	Sleep	Sleep	0	
c2	Running	Ready	Ready	Sleep	Sleep	0	
c3	Sleep	Ready	Ready	Sleep	Sleep	0	Nothing to get
c3	Sleep	c2	Running	Sleep	Sleep	0	
c3	Sleep	c3	Sleep	Sleep	Sleep	0	Everyone asleep ...

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.9

OBJECTIVES – 5/18

- Questions from 5/13
- **Assignment 2**
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.10

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- **Quiz 3 – Synchronized Array**
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.11

OBJECTIVES – 5/18


- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- **Tutorial 2 – Pthread, locks, conditions tutorial**
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.12

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - **Non-deadlock concurrency bugs**
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.13



CHAPTER 32 – CONCURRENCY PROBLEMS

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.14

NON-DEADLOCK BUGS - 1

- 97% of Non-Deadlock Bugs were
 - Atomicity
 - Order violations
- Consider what is involved in “spotting” these bugs in code
 - >> no use of locking constructs to search for
- Desire for automated tool support (IDE)

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.15

NON-DEADLOCK BUGS - 2

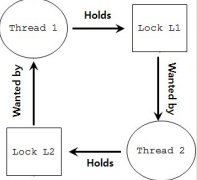
- Atomicity
 - How can we tell if a given variable is shared?
 - Can search the code for uses
 - How do we know if all instances of its use are shared?
 - Can some non-synchronized, non-atomic uses be legal?
 - Legal uses: before threads are created, after threads exit
 - Must verify the scope
- Order violation
 - Must consider all variable accesses
 - Must know desired order

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.16

DEADLOCK BUGS

- Presence of a cycle in code
- Thread 1 acquires lock L1, waits for lock L2
- Thread 2 acquires lock L2, waits for lock L1

Thread 1:	Thread 2:
lock (L1);	lock (L2);
lock (L2);	lock (L1);



```

        graph TD
            T1((Thread 1)) -- Holds --> L1[Lock L1]
            L1 -- "Wanted by" --> T2((Thread 2))
            T2 -- Holds --> L2[Lock L2]
            L2 -- "Wanted by" --> T1
    
```

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.17

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - **Deadlock causes**
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.18

REASONS FOR DEADLOCKS

- Complex code
 - Must avoid circular dependencies – can be hard to find...
- Encapsulation hides potential locking conflicts
 - Easy-to-use APIs embed locks inside
 - Programmer doesn't know they are there
 - Consider the Java Vector class:

```

1 Vector v1,v2;
2 v1.AddAll(v2);
    
```

- Vector is thread safe (synchronized) by design
- If there is a v2.AddAll(v1); call at nearly the same time deadlock could result

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.19

CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.20

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - **Deadlock prevention**
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.21

PREVENTION – MUTUAL EXCLUSION

- Build wait-free data structures
 - Eliminate locks altogether
 - Build structures using CompareAndSwap atomic CPU (HW) instruction
- C pseudo code for CompareAndSwap
- Hardware executes this code atomically

```

1 int CompareAndSwap(int *address, int expected, int new){
2     if(*address == expected){
3         *address = new;
4         return 1; // success
5     }
6     return 0;
7 }
    
```

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.22

PREVENTION – MUTUAL EXCLUSION - 2

- Recall atomic increment

```

1 void AtomicIncrement(int *value, int amount){
2     do{
3         int old = *value;
4     }while( CompareAndSwap(value, old, old+amount)!=0);
5 }
    
```

- Compare and Swap tries over and over until successful
- CompareAndSwap is guaranteed to be atomic
- When it runs it is **ALWAYS** atomic (at HW level)

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.23

MUTUAL EXCLUSION: LIST INSERTION

- Consider list insertion

```

1 void insert(int value){
2     node_t * n = malloc(sizeof(node_t));
3     assert( n != NULL );
4     n->value = value ;
5     n->next = head;
6     head = n;
7 }
    
```

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.24

MUTUAL EXCLUSION – LIST INSERTION - 2

- Lock based implementation

```

1 void insert(int value){
2     node_t * n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     lock(listlock); // begin critical section
6     n->next = head;
7     head = n;
8     unlock(listlock); //end critical section
9 }
```

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.25

MUTUAL EXCLUSION – LIST INSERTION - 3

- Wait free (no lock) implementation

```

1 void insert(int value) {
2     node_t *n = malloc(sizeof(node_t));
3     assert(n != NULL);
4     n->value = value;
5     do {
6         n->next = head;
7     } while (!CompareAndSwap(&head, n->next, n));
8 }
```

- Assign &head to n (new node ptr)
- Only when head = n->next

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.26

CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.27

PREVENTION LOCK – HOLD AND WAIT

- Problem: acquire all locks atomically
- Solution: use a "lock" "lock"... (like a *guard lock*)

```

1 lock(prevention);
2 lock(L1);
3 lock(L2);
4 ...
5 unlock(prevention);
```

- Effective solution – guarantees no race conditions while acquiring L1, L2, etc.
- Order doesn't matter for L1, L2
- Prevention (GLOBAL) lock decreases concurrency of code
 - Acts Lowers lock granularity
- Encapsulation: consider the Java Vector class...

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.28

CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.29

PREVENTION – (NO PREEMPTION)

- When acquiring locks, don't BLOCK forever if unavailable...
- pthread_mutex_trylock() - try once
- pthread_mutex_timedlock() - try and wait awhile

```

1 top:
2     lock(L1);
3     if( trylock(L2) == -1 ){
4         unlock(L1);
5         goto top;
6     }
```

- Eliminates deadlocks


May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.30

NO PREEMPTION – LIVELOCKS PROBLEM

- Can lead to livelock


```

            1 top:
            2   lock(L1);
            3   if( tryLock(L2) == -1 ){
            4     unlock(L1);
            5     goto top;
            6   }
```
- Two threads execute code in parallel → always fail to obtain both locks
- Fix: add random delay
 - Allows one thread to win the livelock race!



May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.31

CONDITIONS FOR DEADLOCK

- **Four conditions** are required for dead lock to occur

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
→ Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.32

PREVENTION – CIRCULAR WAIT

- Provide **total ordering** of lock acquisition throughout code
 - Always acquire locks in same order
 - L1, L2, L3, ...
 - Never mix: L2, L1, L3; L2, L3, L1; L3, L1, L2....
- Must carry out same ordering through entire program

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.33

CONDITIONS FOR DEADLOCK

- **If any of the following conditions DOES NOT EXSIST, describe why deadlock can not occur?**

Condition	Description
→ Mutual Exclusion	Threads claim exclusive control of resources that they require.
→ Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
→ No preemption	Resources cannot be forcibly removed from threads that are holding them.
→ Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.34

The dining philosophers problem where 5 philosophers compete for 5 forks, and where a philosopher must hold two forks to eat involves which deadlock condition(s)?

- Mutual Exclusion
- Hold-and-wait
- No preemption
- Circular wait
- All of the above

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

DEADLOCK AVOIDANCE VIA INTELLIGENT SCHEDULING

- Consider a **smart scheduler**
 - Scheduler knows which locks threads use
- Consider this scenario:
 - 4 Threads (T1, T2, T3, T4)
 - 2 Locks (L1, L2)
- Lock requirements of threads:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.36

INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:

CPU 1

T3

T4

CPU 2

T1

T2

- No deadlock can occur
- Consider:

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.37

INTELLIGENT SCHEDULING - 3

- Scheduler produces schedule

CPU 1

T4

CPU 2

T1

T2

T3

- Scheduler must be conservative and not take risks
- Slows down execution – many threads
- There has been limited use of these approaches given the difficulty having intimate lock knowledge about every thread

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.38

DETECT AND RECOVER

- Allow deadlock to occasionally occur and then take some action.
 - Example: When OS freezes, reboot...
- How often is this acceptable?
 - Once per year
 - Once per month
 - Once per day
 - Consider the effort tradeoff of finding every deadlock bug
- Many database systems employ deadlock detection and recovery techniques.

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.39

WE WILL RETURN AT 4:50PM




May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.40

OBJECTIVES - 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces**
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.41

CHAPTER 13: ADDRESS SPACES



May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.42

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 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.43

MEMORY VIRTUALIZATION


- What is memory virtualization?
- This is not “virtual” memory,
 - Classic use of disk space as additional RAM
 - When available RAM was low
 - Less common recently

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.44


MEMORY VIRTUALIZATION - 2

- Presentation of system memory to each process
- Appears as if each process can access the entire machine's address space
- Each process's view of memory is isolated from others
- Everyone has their own sandbox


Process A



Process B



Process C



May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.45

MOTIVATION FOR MEMORY VIRTUALIZATION

- Easier to program
 - Programs don't need to understand special memory models
- Abstraction enables sophisticated approaches to manage and share memory among processes
- Isolation
 - From other processes: easier to code
- Protection
 - From other processes
 - From programmer error (segmentation fault)

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.46

EARLY MEMORY MANAGEMENT

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

0KB

64KB

max

Operating System
(code, data, etc.)

Current Program
(code, data, etc.)

Physical Memory

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.47

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

0KB

64KB

128KB

192KB

256KB

320KB

384KB

448KB

512KB

Operating System
(code, data, etc.)

Free

Process C
(code, data, etc.)

Process B
(code, data, etc.)

Free

Process A
(code, data, etc.)

Free

Free

Physical Memory

May 18, 2021
TCSS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.48

ADDRESS SPACE

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

Address Space

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.49

ADDRESS SPACE - 2

- Code
 - Program code
- Stack
 - Program counter (PC)
 - Local variables
 - Parameter variables
 - Return values (for functions)
- Heap
 - Dynamic storage
 - Malloc() new()

Address Space

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.50

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

Address Space

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.51

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.52

VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:

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

Address Space

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.53

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.54

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.55

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- **Chapter 14: The Memory API**
- Chapter 15: Address Translation
- Chapter 16: Segmentation

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.56

CHAPTER 14: THE MEMORY API



May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.57

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.58

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.59

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.60

FREE()

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

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

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.61

What will this code do?

```
#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;
}
```

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.62

```
#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?

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.63

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.64

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.65

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.66

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 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.67

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 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.68

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 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.69

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation**
- Chapter 16: Segmentation

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.70

CHAPTER 15: ADDRESS TRANSLATION

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.71

OBJECTIVES – 5/18

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

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.72

ADDRESS TRANSLATION

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

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.73

BASE AND BOUNDS

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

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

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

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

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.74

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

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

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.75

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.76

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 to register exception handlers
Ability to raise exceptions	For out-of-bounds memory access, or attempts to access privileged instr.

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.77

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.78

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

Physical Memory

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.79

OS: WHEN PROCESS IS TERMINATED

- OS places memory back on the free list

Free list

Physical Memory

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.80

OS: WHEN CONTEXT SWITCH OCCURS

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

Context Switching

Process A PCB

base: 32KB
bounds: 48KB

Physical Memory

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.81

DYNAMIC RELOCATION

- OS can move process data when not running

1. OS deschedules 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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.82

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 poller.com/app


May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.84

OBJECTIVES – 5/18

- Questions from 5/13
- Assignment 2
- Quiz 3 – Synchronized Array
- Tutorial 2 – Pthread, locks, conditions tutorial
- Chapter 32: Concurrency Problems
 - Non-deadlock concurrency bugs
 - Deadlock causes
 - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API
- Chapter 15: Address Translation
- **Chapter 16: Segmentation**

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.84

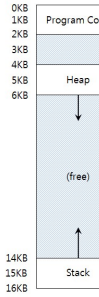
CHAPTER 16: SEGMENTATION



May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.85

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?



May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.86

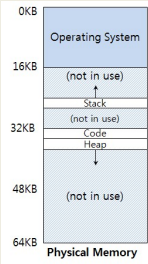
MULTIPLE SEGMENTS

- 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 be placed separately
- Track base and bounds for each segment (registers)

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.87

SEGMENTS IN MEMORY

■ Consider 3 segments:



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

↓
Much smaller

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.88

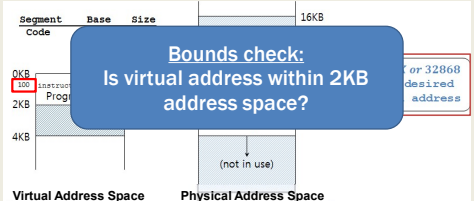
ADDRESS TRANSLATION: CODE SEGMENT

$physical\ address = offset + base$

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

Segment	Base	Size
Code	32KB	2KB

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



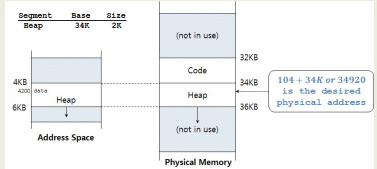
May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.89

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



May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.90

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

Address Space

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.91

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 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.92

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 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.93

STACK SEGMENT

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

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

May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.94

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

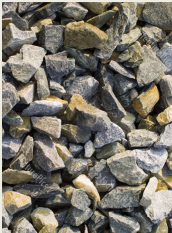
May 18, 2021 | TCCS422: Operating Systems [Spring 2021] | School of Engineering and Technology, University of Washington - Tacoma | L14.95

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 poller.com/app

SEGMENTATION GRANULARITY


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



May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.97

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 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.98

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	
0KB	
8KB	Operating System
16KB	(not in use)
24KB	Allocated
32KB	(not in use)
40KB	Allocated
48KB	(not in use)
56KB	Allocated
64KB	

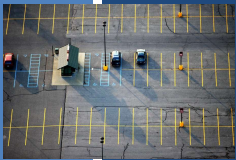
May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.99

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	
0KB	
8KB	Operating System
16KB	
24KB	Allocated
32KB	
40KB	
48KB	(not in use)
56KB	
64KB	

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.100



CHAPTER 17: FREE SPACE MANAGEMENT

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.101

OBJECTIVES – 5/18

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

May 18, 2021
TCCS422: Operating Systems [Spring 2021]
School of Engineering and Technology, University of Washington - Tacoma
L14.102

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.103

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.104

FRAGMENTATION

- Consider a 30-byte heap
- Request for 15-bytes
- Free space: 20 bytes
- No available contiguous chunk → return NULL

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.105

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.106

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

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.107

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.108

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

An Allocated Region Plus Header

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.109

MEMORY HEADERS - 2

Specific Contents Of The Header

A Simple Header

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

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.110

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.111

THE FREE LIST

- Simple free list struct

```
typedef struct __node_t {
    int size;
    struct __node_t *next;
} node_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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.112

FREE LIST - 2

- Create and initialize free-list "heap"

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

- Heap layout:

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.113

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

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.114

FREE LIST: FREE() CALL

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

Free Space With Three Chunks Allocated

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.115

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.116

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.117

GROWING THE HEAP

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

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.118

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

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.119

EXAMPLES

- Allocation request for 15 bytes
 - head → 10 → 30 → 20 → NULL
- Result of Best Fit
 - head → 10 → 30 → 5 → NULL
- Result of Worst Fit
 - head → 10 → 15 → 20 → NULL

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.120

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.121

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 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.122

BUDDY ALLOCATION

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

64KB free space for 7KB request

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.123

BUDDY ALLOCATION - 2

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

May 18, 2021 TCCS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma L14.124

QUESTIONS