TCSS 422: OPERATING SYSTEMS

Concurrency Problems, Introduction to memory virtualization

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

May 12, 2020
TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.2

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (44 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 6.87 (↑ *from* 6.27)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.81 (↑ from 5.77)

FEEDBACK FROM 5/7

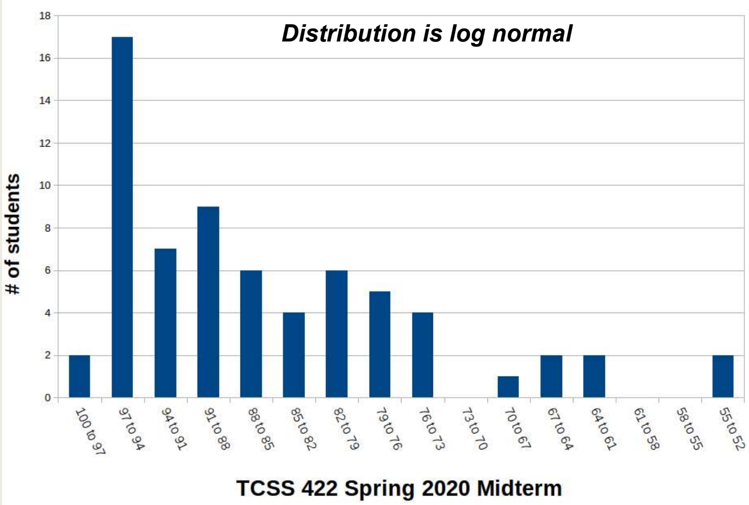
OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.5

MIDTERM RESULTS

Average: 85.41
Mode: 95.5
Median: 88.3
1st quartile: 0 to 79.5
2nd quartile: 79.5 to 88.3
3rd quartile: 88.3 to 94.5
4th quartile: 94.5 to 99.5
Std. dev: 10.96



May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.6

NOTES ON SCORING

- **Questions 5, 6, 10, 11:**
- When I transferred this question to Canvas, I changed job D's arrival time from T=20 to T=10
- My solution accidentally did not capture this change
- Initial answers in Canvas incorrectly scored these questions
- Scores have been updated, please verify scoring:
- **Question 5:** 25 (turnaround time job D) +1
- **Question 6:** 28.8 (average turnaround time all jobs) +1
- **Question 10:** 20 (response time job D) +1
- **Question 11:** 16.3 (average response time all jobs) +1

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.7

NOTES ON SCORING - 2

- **Question 34:**
- MLFQ Scheduler Question
- How many jobs execute in the medium priority queue?
- This question was simply asking of jobs A, B, C, how many of them ran at some point in the medium priority queue

  (this question can be answered using only the starter graph)
- Answer: 3
- I have also accepted as an answer to the questions, the total number of timer units jobs ran in the medium priority queue
- Second answer: 24 (ok)

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.8

NOTES ON SCORING - 3

- **Question 48:**
- Classify the API:
- **pthread_mutex_timelock()**
- This API is both BLOCKING (for a while) and NON-BLOCKING, so the best answer was "all of the above"
- Partial credit for BLOCKING or NON-BLOCKING

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.9

NOTES ON SCORING - 4

- All other questions have been reviewed for partial credit, and Canvas grading interpretation
- Canvas autograder is sensitive and wants exact answers
- Where answers were not recognized, I have manually added points

- Please review scoring for correctness, and notify of any issues / questions

- Midterm questions ?
- Special session - midterm review: recording LIVE on Wed May 13 @ 5:50p

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.10

OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.11

CHAPTER 32 – CONCURRENCY PROBLEMS

May 12, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L12.12

OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

NON-DEADLOCK BUGS

- Majority of concurrency bugs
- Most common:
  - Atomicity violation: forget to use locks
  - Order violation: failure to initialize lock/condition before use

ATOMICITY VIOLATION - SOLUTION

- Add locks for all uses of: `thd->proc_info`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1::
pthread_mutex_lock(&lock);
if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
    …
}
pthread_mutex_unlock(&lock);

Thread2::
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```


ORDER VIOLATION - SOLUTION

- Use condition & signal to enforce order

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;

Thread 1::
void init(){
    …
    mThread = PR_CreateThread(mMain,…);

    // signal that the thread has been created.
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    …
}

Thread2::
void mMain(…){
    …
```

ORDER VIOLATION – SOLUTION - 2

- Use condition & signal to enforce order

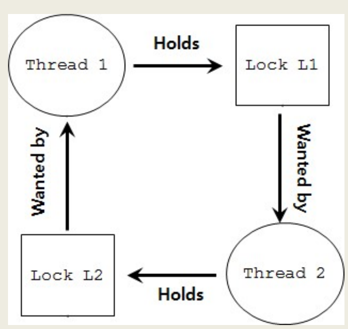
```
        // wait for the thread to be initialized …
        pthread_mutex_lock(&mtLock);
        while(mtInit == 0)
                pthread_cond_wait(&mtCond, &mtLock);
        pthread_mutex_unlock(&mtLock);

        mState = mThread->State;
        …
}
```


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);
```

- Both threads can block, unless one manages to acquire both locks



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:

```
Vector v1,v2;
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

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

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

```
int CompareAndSwap(int *address, int expected, int new){
    if(*address == expected){
            *address = new;
            return 1; // success
    }
    return 0;
}
```

May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.21

PREVENTION – MUTUAL EXCLUSION - 2

- Recall atomic increment

```
void AtomicIncrement(int *value, int amount){
    do{
            int old = *value;
    }while( CompareAndSwap(value, old, old+amount)==0);
}
```

- 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 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.22

MUTUAL EXCLUSION: LIST INSERTION

- Consider list insertion

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    n->next  = head;
    head     = n;
}
```


MUTUAL EXCLUSION – LIST INSERTION - 2

- Lock based implementation

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    lock(listlock); // begin critical section
    n->next  = head;
    head     = n;
    unlock(listlock) ;  //end critical section
}
```

MUTUAL EXCLUSION – LIST INSERTION - 3

- Wait free (no lock) implementation

```
void insert(int value) {
     node_t *n = malloc(sizeof(node_t));
     assert(n != NULL);
     n->value = value;
     do {
             n->next = head;
     } while (CompareAndSwap(&head, n->next, n));
}
```

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

CONDITIONS FOR DEADLOCK

- <u>Four conditions</u> 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

PREVENTION LOCK – HOLD AND WAIT

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

```
lock(prevention);
lock(L1);
lock(L2);
…
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…

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

PREVENTION – NO PREEMPTION

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

```
top:
    lock(L1);
    if( tryLock(L2) == -1 ){
        unlock(L1);
        goto top;
    }
```

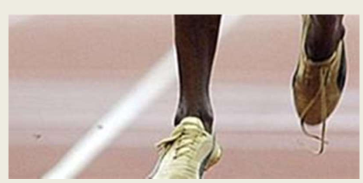
- Eliminates deadlocks

NO PREEMPTION – LIVELOCKS PROBLEM

- Can lead to livelock

```
top:
    lock(L1);
    if( tryLock(L2) == -1 ){
        unlock(L1);
        goto top;
    }
```

- Two threads execute code in parallel → always fail to obtain both locks
- Fix: add random delay
  - Allows one thread to win the livelock race!



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 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.31

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 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.32

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

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 **pollev.com/app**

DEADLOCK AVOIDANCE VIA INTELLIGENT SCHEDULING

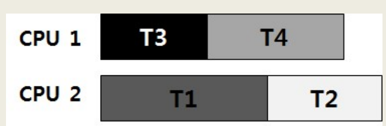
- Consider a smart scheduler
  - Scheduler knows which locks threads use
- Consider this scenario:
  - 4 Threads (T1, T2, T3, T4)
  - 2 Locks (L1, L2)
- Lock requirements of threads:

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

May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.35

INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:



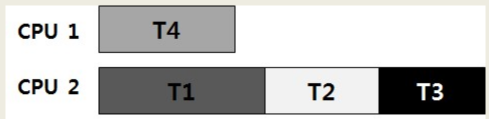
- No deadlock can occur
- Consider:

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

May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.36

INTELLIGENT SCHEDULING - 3

- Scheduler produces schedule



- 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

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.

TCSS 422 WILL RETURN AT ~2:48PM

May 12, 2020 | TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma | L12.3
9

CHAPTER 13: ADDRESS SPACES

May 12, 2020 | TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma | L12.40

OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

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

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

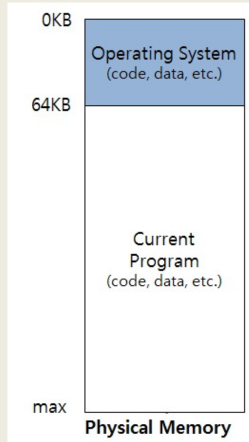
Process C

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)

EARLY MEMORY MANAGEMENT

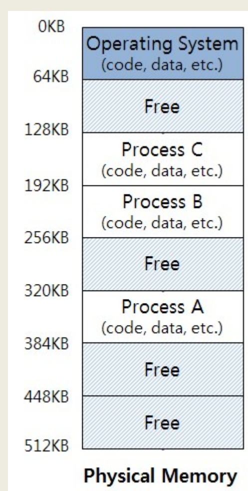
- Load one process at a time into memory
- Poor memory utilization
- Little abstraction



May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.45

MULTIPROGRAMMING WITH SHARED MEMORY

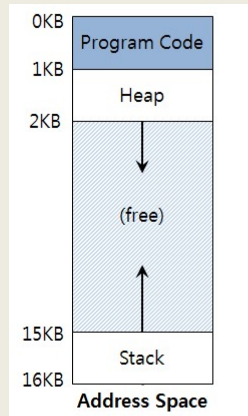
- Later machines supported running multiple processes
- Swap out processes during I/O waits to increase system utilization and efficiency
- Swap entire memory of a process to disk for context switch
- Too slow, especially for large processes
- Solution→
  - Leave processes in memory
- Need to protect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



May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.46

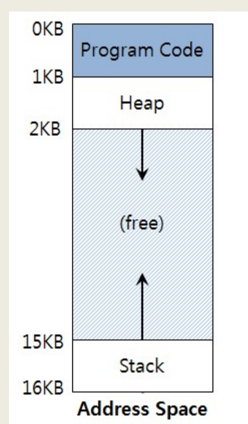
ADDRESS SPACE

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



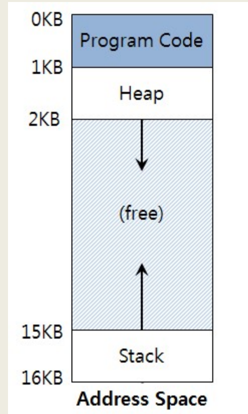
ADDRESS SPACE - 2

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



ADDRESS SPACE - 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



VIRTUAL ADDRESSING

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

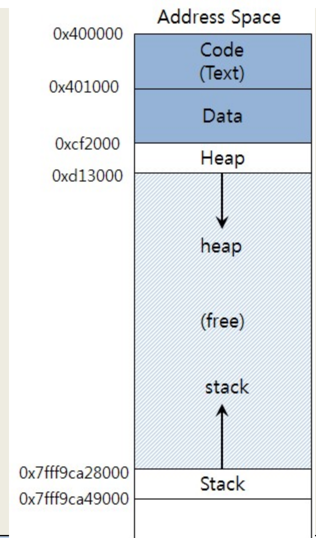
```
#include <stdio.h>
#include <stdlib.h>

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

VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:

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



May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.51

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 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.52

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*

CHAPTER 14: THE MEMORY API

OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.55

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 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.56

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 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.57

FREE()

```
#include <stdlib.h>

void free(void* ptr)
```

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

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.58

```
#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?

59

```
#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?

60

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).

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"

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
```


REALLOC()

```
#include <stdlib.h>

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

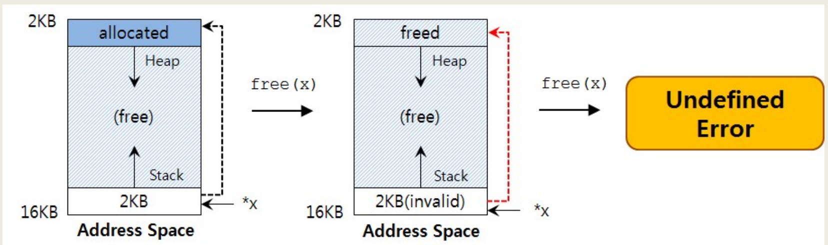
- Resize an existing memory allocation
- Returned pointer may be same address, or a new address
  - New if memory allocation must move
- `void *ptr`: Pointer to memory block allocated with malloc, calloc, or realloc
- `size_t size`: New size for the memory block(in bytes)

- EXAMPLE: realloc.c
- EXAMPLE: nom.c

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



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

CHAPTER 15: ADDRESS TRANSLATION

May 12, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L12.67

OBJECTIVES – 5/12

- Questions from 5/7
- Midterm review
- Assignment 2 (based on Ch. 30)
- Chapter 32: Concurrency Problems
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Introduction to memory virtualization
  - The address space
  - Goals of OS memory virtualization
- Chapter 14: Memory API
  - Common memory errors
- Chapter 15: Address translation
  - Base and bounds
  - HW and OS Support

May 12, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma L12.68

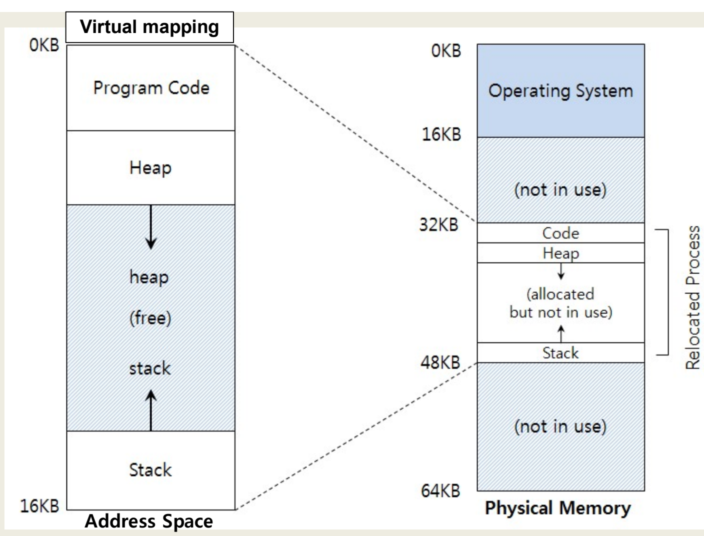
CH. 15: OBJECTIVES

- Address translation
- Base and bounds
- HW and OS Support
- Memory segments
- Memory fragmentation

May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.69

ADDRESS TRANSLATION

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



May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.70

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:

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

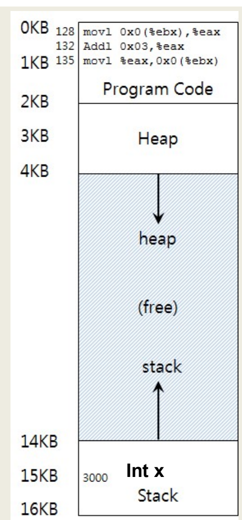
May 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.71

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 12, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.72

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)

DYNAMIC RELOCATION OF PROGRAMS

- Hardware requirements:

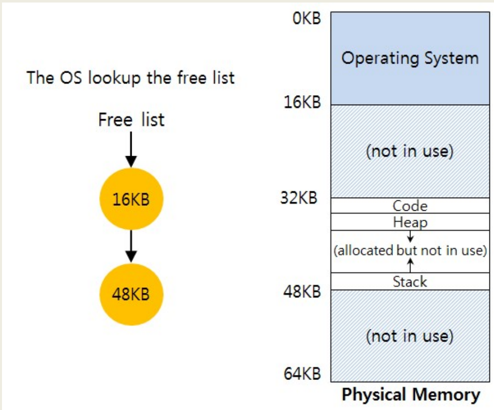
Requirements	HW support
Privileged mode	CPU modes: kernel, user
Base / bounds registers	Registers to support address translation
Translate virtual addr; check if in bounds	Translation circuitry, check limits
Privileged instruction(s) to update base / bounds regs	Instructions for modifying base/bound 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.

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

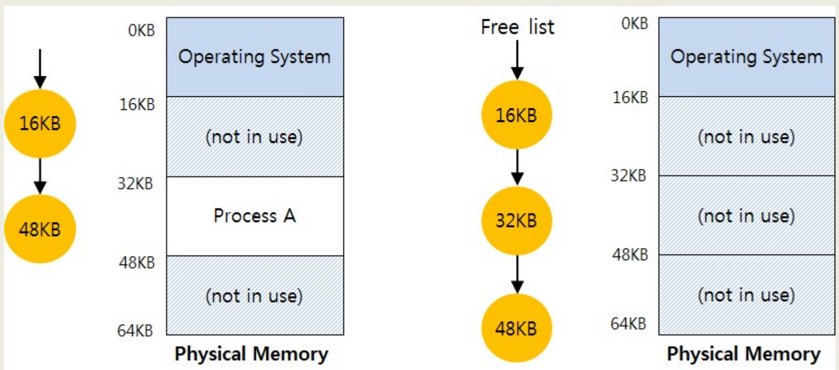
OS: WHEN PROCESS STARTS RUNNING

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



OS: WHEN PROCESS IS TERMINATED

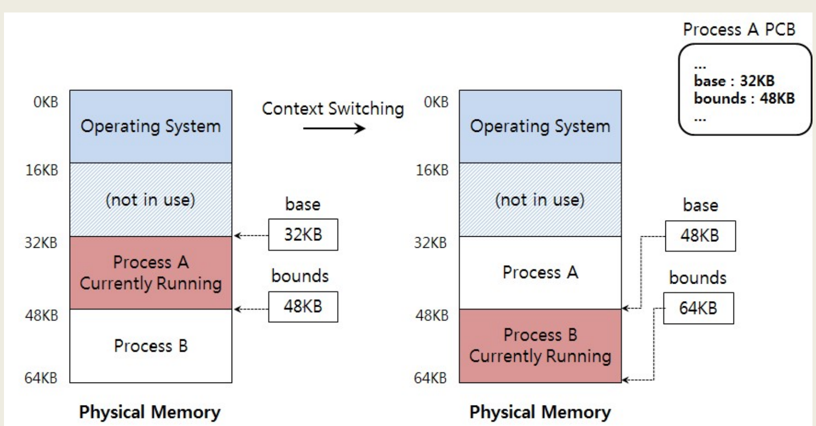
- OS places memory back on the free list



May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.77

OS: WHEN CONTEXT SWITCH OCCURS

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



May 12, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.78

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!**

QUESTIONS

TCSS 422
OFFICE HOURS

WILL RETURN IN A FEW MINUTES