TCSS 422: OPERATING SYSTEMS

Condition Variables, Concurrency Problems

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

May 11, 2023 TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma

1

OFFICE HOURS – FRIDAY

- **SPECIAL TIME FRIDAY MAY 12**
  ***** THIS WEEK ONLY *****
  - Due to 2 faculty meetings in the afternoon
- **Friday Office Hours**
  - *11:30am to 12:30 pm – Zoom Only

April 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L5.2

2

FINAL EXAM SURVEY
* - NOW AVAILABLE IN CANVAS -*

- TCSS 422 Final is scheduled for:
  Thursday June 8th 3:40-5:40pm
- This is one of the last time slots of the final exams week.
- Please indicate your preference for scheduling of the TCSS 422 Final Exam for Spring 2023:
  - A. Thursday June 1, 3:40 to 5:40 pm
  - B. Thursday June 8, 3:40 to 5:40 pm
  - C. No Preference
- Regardless of the selected date, the content and coverage on the Final Exam will remain the same.
- *(please disregard scoring as the quiz is worth 0 points.)*

FINAL EXAM SURVEY - 2

- Moving the Final to the last class (June 1st) will result in one less lecture as a regular class session will be used for the exam.
- To make-up the missing class session, an additional class session will be required prior to June 1st.
- This session will be recorded. The session will be live-streamed and could be 100% online or hybrid depending on availability of physical classroom space.
- The make-up session could occur over a weekend to space the lecture out relative to others so as not to have lectures on back-to-back days, or the session may fall on a Monday, Wednesday, or Friday.

OBJECTIVES – 5/11

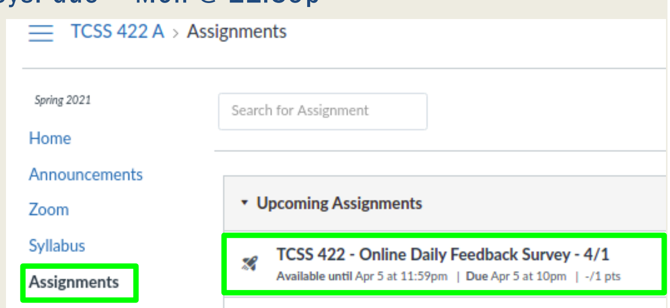
- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.5

5

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 11, 2023 | TCSS422: Computer Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.6

6

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

7

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (45 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.80 (↑ - previous 6.52)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.60 (↑ - previous 5.48)**

8

FEEDBACK

- ***Does the number of producers depend on the number of conditional variables?***
- If there is more than 1 producer thread, then it can help to have a separate producer condition variable.
- If 2 producers and 1 consumer share the same condition variable, then when 1 producer fires the signal to indicate the buffer is full, it may accidentally wake up the other producer and not the consumer (***program has only 1 condition variable***)
- The other producer, assuming a while statement is used, will reevaluate the state variable in the while, and will go back to sleep because there is no free space in the buffer to produce
- The problem is the second producer does not fire the signal to wake up the consumer, the buffer is never emptied, the consumer goes back to sleep, and the program DEADLOCKS

FEEDBACK - 2

- ***And how does the consumer depend on the producer?***
- With 2 producers, and 1 consumer, the consumer depends on the producer(s) adding data to the buffer, and firing a signal that only wakes up consumers
- If there is competition between the producers and consumers to receive the same signal DEADLOCK is possible
- The solution for multiple producer or multiple consumer is to have separate condition variables to enable signaling different events

FEEDBACK - 3

- ***In the final example with two condition variables, there is still a single lock that all producers and all consumers must use. Is it possible for this to be a bottleneck in a practical application with many producers and many consumers? If not, why? If so, is there a known solution?***
- The bounded buffer is still a synchronized data structure
- If two threads try to produce and add data to the buffer at the same time with coordination, data corruption is possible
- Synchronization also can address multiple consumers removing items at the same time
- Without sharing a lock, two consumers might try to consume the same item at the same time
- I have not tested this, but I would assume if we do not synchronize both producers and consumers sharing the buffer using the same lock corruption may occur when there is just 1 item in the buffer to remove/add

FEEDBACK - 4

- ***When having a producer consumer setup with multiple producers and multiple consumers why is this advantageous over a single consumer and a single producer?***
- Having multiple producers and/or multiple consumers enables more threads to work on the data processing problem at the same time
- Operations can be done in parallel, like creating a new data item/node or matrix without holding the lock
  - For example, generating a large 10000 x 10000 matrix is slow, we can just push the matrix pointer onto the bounded buffer, and have many producers can make matrices in parallel to improve throughput of the program
- We only need the lock to modify the buffer for a very short amount of time (changing the buffer must be synchronized)

OBJECTIVES – 5/11

- Questions from 5/9
- **Pthread Tutorial-May 26 / Assignment 2 posted next week**
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

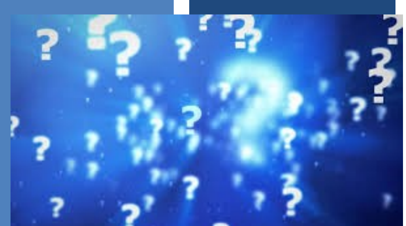
OBJECTIVES – 5/11

- Questions from 5/9
- **Pthread Tutorial-May 26 / Assignment 2 posted next week**
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- **Quiz 3 – Synchronized Array (class activity-next week)**
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

15



CHAPTER 30 – CONDITION VARIABLES

16

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - **Covering Conditions**
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

COVERING CONDITIONS

- A condition that covers ***all*** cases (conditions):
- Excellent use case for **pthread_cond_broadcast**
- Consider memory allocation:
  - When a program deals with huge memory allocation/deallocation on the heap
  - Access to the heap must be managed when memory is scarce

    PREVENT: Out of memory:
    - queue requests until memory is free
  - Which thread should be woken up?

COVERING CONDITIONS - 2

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;            // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);    //
    Pthread_mutex_unlock(&m);
}
```

Check available memory

Broadcast

19

COVER CONDITIONS - 3

- Broadcast awakens all blocked threads requesting memory
- Each thread evaluates if there's enough memory: (bytesLeft < size)
  - Reject: requests that cannot be fulfilled- go back to sleep
    - *Insufficient memory*
  - Run: requests which **can** be fulfilled
    - with newly available memory!
- **Another use case**: coordinate a group of busy threads to gracefully end, to EXIT the program
- **Overhead**
  - Many threads may be awoken which can't execute

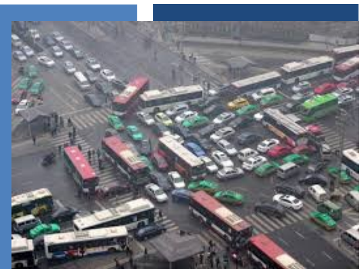
20

CHAPTER 31: SEMAPHORES

- Offers a combined C language construct that can assume the role of a lock or a condition variable depending on usage
  - Allows fewer concurrency related variables in your code
  - Potentially makes code more ambiguous
  - For this reason, with limited time in a 10-week quarter, we do not cover
- **Ch. 31.6 – Dining Philosophers Problem**
  - Classic computer science problem about sharing eating utensils
  - Each philosopher tries to obtain two forks in order to eat
  - Mimics deadlock as there are not enough forks
  - Solution is to have one left-handed philosopher that grabs forks in opposite order

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Producer/Consumer
  - Covering Conditions
- **Chapter 32: Concurrency Problems**
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API



CHAPTER 32 – CONCURRENCY PROBLEMS

May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.23

23

CONCURRENCY BUGS IN OPEN SOURCE SOFTWARE

- "Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics"
  - Shan Lu et al.
  - Architectural Support For Programming Languages and Operating Systems (ASPLOS 2008), Seattle WA

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		**74**	**31**

May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.24

24

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - **Non-deadlock concurrency bugs**
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

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 - MYSQL

- Two threads access the `proc_info` field in `struct thd`
- `NULL` is 0 in C
- Mutually exclusive access to shared memory among separate threads is not enforced (e.g. non-atomic)
- Simple example: ***proc_info deleted***

Programmer intended variable to be accessed atomically…

```
Thread1::
if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
    …
}

Thread2::
thd->proc_info = NULL;
```


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 BUGS

- Desired order between memory accesses is flipped
- E.g. something is checked before it is set
- Example:

```
Thread1::
void init(){
    mThread = PR_CreateThread(mMain, …);
}

Thread2::
void mMain(…){
    mState = mThread->State
}
```

- What if mThread is not initialized?

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;
      …
}
```


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)

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 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.33

33

WE WILL RETURN AT 4:50PM

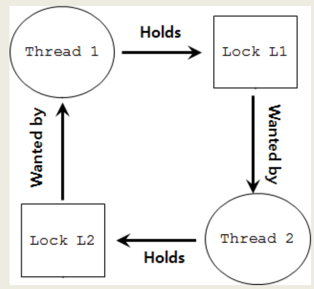
May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - coma | L13.34

34

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

May 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L13.35

35

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - **Deadlock causes**
  - Deadlock prevention
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

May 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L13.36

36

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

- <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

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - **Deadlock prevention**
- Chapter 13: Address Spaces
- Chapter 14: The Memory API

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;
}
```

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)

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

- **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 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.45

45

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…

May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.46

46

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

47

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

48

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

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

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**

53

DEADLOCK AVOIDANCE VIA INTELLIGENT SCHEDULING

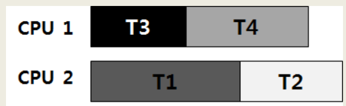
- Consider a smart scheduler
  - Scheduler knows which locks threads use
- Consider this scenario:
  - 4 Threads (T1, T2, T3, T4)
  - 2 Locks (L1, L2)
- Lock requirements of threads:

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

54

INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:



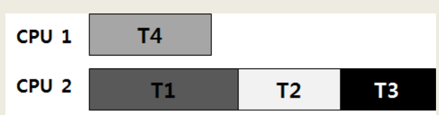
- No deadlock can occur
- Consider:

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

55

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

56

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 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L13.57

57

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- **Chapter 13: Address Spaces**
- Chapter 14: The Memory API

May 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L13.58

58

CHAPTER 13: ADDRESS SPACES

May 11, 2023 | TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L13.59

59

OBJECTIVES – 5/11

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

May 11, 2023 | TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L13.60

60

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

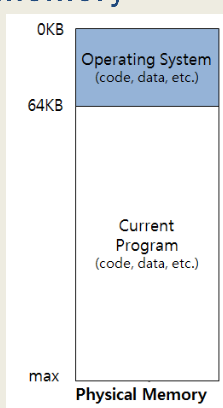
Process C

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)

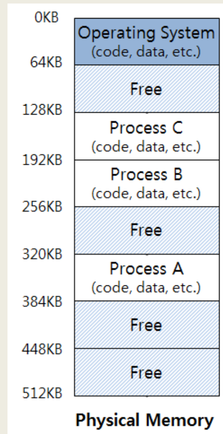
EARLY MEMORY MANAGEMENT

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



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

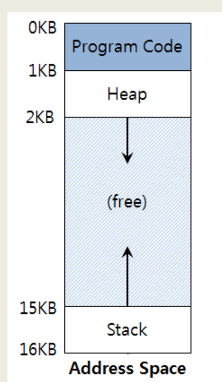


May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.65

65

ADDRESS SPACE

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

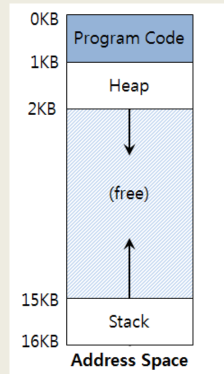


May 11, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L13.66

66

ADDRESS SPACE - 2

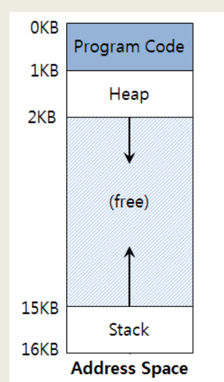
- Code
  - Program code
- Stack
  - Program counter (PC)
  - Local variables
  - Parameter variables
  - Return values (for functions)
- Heap
  - Dynamic storage
  - Malloc() new()



67

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



68

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

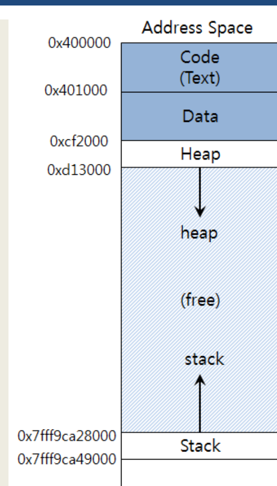
May 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L13.69

69

VIRTUAL ADDRESSING - 2

- Output from 64-bit Linux:

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



May 11, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L13.70

70

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)

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*

OBJECTIVES – 5/11

- Questions from 5/9
- Pthread Tutorial-May 26 / Assignment 2 posted next week
- Quiz 3 – Synchronized Array (class activity-next week)
- Chapter 30: Condition Variables
  - Covering Conditions
- Chapter 32: Concurrency Problems
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention
- Chapter 13: Address Spaces
- **Chapter 14: The Memory API**

73

CHAPTER 14: THE MEMORY API

74

OBJECTIVES – 5/11

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

75

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

76

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
```


77

FREE()

```
#include <stdlib.h>

void free(void* ptr)
```

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

78

```
#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?

79

79

```
#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?

80

80

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)
```

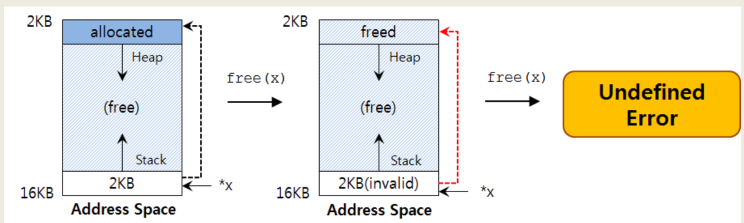
- Resize an existing memory allocation
- Returned pointer may be same address, or a new address
  - New if memory allocation must move
- `void *ptr`: Pointer to memory block allocated with malloc, calloc, or realloc
- `size_t size`: New size for the memory block(in bytes)

- 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



85

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

86

QUESTIONS

87