TCSS 422: OPERATING SYSTEMS

Condition Variables, Concurrency Problems, Intro to Memory Virtualization

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

February 14, 2018
TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Homework 2 Questions
- Ch. 30
  - Condition Variables
- Ch. 32
  - Concurrency Problems
- Ch. 13
  - Address Spaces
- Ch. 14
  - Memory API

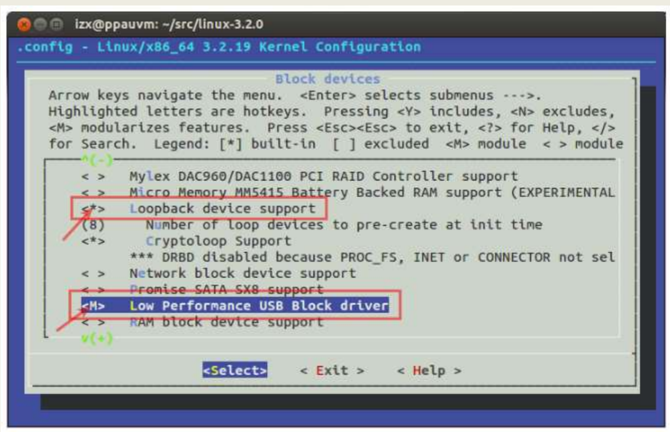
February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.2

FEEDBACK FROM 2/12

- **Are device drivers implemented as kernel modules?**
- Device drivers are frequently built as kernel modules:
- See: http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/
- Required drivers can loaded on demand to support running on various hardware configurations.
- To see list of device drivers:
  - cd /lib/modules/$(uname –r)
  - find . | grep ko
- To count the number of drivers:
  - find . | grep ko | wc –l

FEEDBACK - 2

- Alternatively, device drivers can be integrated into the Linux kernel.
- Some drivers are provided natively (e.g. loopback device driver)
- Linux kernel can be rebuilt to automatically include device drivers:



FEEDBACK - 3

- **Are kernel modules only run when accessed via the procfile interface, or do they also have a main/background thread/process?**
- It depends on the kernel module.
- A basic kernel module with only a read or write callback function that doesn't actively perform other tasks will sleep until a user performs I/O on the /proc file
- When I/O is performed a [kworker] process traps the event and calls the callback function
- See:
- https://www.ibm.com/developerworks/library/l-proc/index.html
- Google search: "linux proc read callback"

FEEDBACK - 4

- If the bounded buffer example (signal.c) used a data structure for multiple matrices (queue/stack), would you use that data structure's size as a "ready" variable?
- A bounded buffer is similar to a queue
- Elements are added at the front, and retrieved from the tail.
- If the data structure is "bounded" (i.e. has a fixed size) then:
  - We can only **ADD** items to *QUEUE* if there is free capacity
  - Can only **REMOVE** items from *QUEUE* if there's content to "consume"

CHAPTER 30 – CONDITION VARIABLES

February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.7

MATRIX GENERATOR

Matrix generation example

Chapter 30
signal.c

February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.8

MATRIX GENERATOR

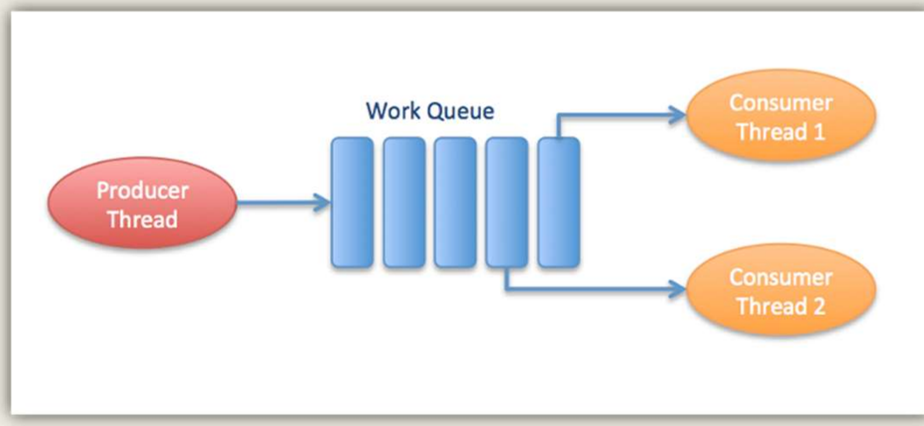
- The main thread, and worker thread (generates matrices) share a single matrix pointer.
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Let's try "nosignal.c"

SUBTLE RACE CONDITION: WITHOUT A WHILE

```
void thr_exit() {
        done = 1;
        Pthread_cond_signal(&c);
}

void thr_join() {
        if (done == 0)
                Pthread_cond_wait(&c);
}
```

- Parent thread calls thr_join() and executes the comparison
- The context switches to the child
- The child runs thr_exit() and signals the parent, but the parent is not waiting yet.
- **The signal is lost**
- The parent deadlocks

PRODUCER / CONSUMER



PRODUCER / CONSUMER

- Producer
  - Produces items – consider the child matrix maker
  - Places them in a buffer
    - Example: the buffer is only 1 element (single array pointer)
- Consumer
  - Grabs data out of the buffer
  - Our example: parent thread receives dynamically generated matrices and performs an operation on them
    - Example: calculates average value of every element (integer)
- Multithreaded web server example
  - Http requests placed into work queue; threads process

PRODUCER / CONSUMER - 2

- Producer / Consumer is also known as **Bounded Buffer**
- Bounded buffer
  - Similar to piping output from one Linux process to another
  - grep pthread signal.c | wc –l
  - Synchronized access:
    sends output from grep → wc as it is produced
  - File stream

PUT/GET ROUTINES

- Buffer is a one element shared data structure (int)
- Producer "puts" data
- Consumer "gets" data
- Shared data structure requires synchronization

```
int buffer;
int count = 0;    // initially, empty

void put(int value) {
        assert(count == 0);
        count = 1;
        buffer = value;
}

int get() {
        assert(count == 1);
        count = 0;
        return buffer;
}
```

PRODUCER / CONSUMER - 3

- Producer adds data
- Consumer removes data (busy waiting)
- **Will this code work (spin locks) with 2-threads?**
  1. Producer 2. Consumer

```
void *producer(void *arg) {
        int i;
        int loops = (int) arg;
        for (i = 0; i < loops; i++) {
                put(i);
        }
}

void *consumer(void *arg) {
        int i;
        while (1) {
                int tmp = get();
                printf("%d\n", tmp);
        }
}
```


PRODUCER / CONSUMER - 3

- The shared data structure needs synchronization!

Producer

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                // p1
        if (count == 1)                            // p2
            Pthread_cond_wait(&cond, &mutex);      // p3
        put(i);                                    // p4
        Pthread_cond_signal(&cond);                // p5
        Pthread_mutex_unlock(&mutex);              // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                // c1
```

PRODUCER/CONSUMER - 4

```
            if (count == 0)                              // c2
                Pthread_cond_wait(&cond, &mutex);        // c3
            int tmp = get();                             // c4
            Pthread_cond_signal(&cond);                  // c5
            Pthread_mutex_unlock(&mutex);                // c6
            printf("%d\n", tmp);
        }
    }
```

Consumer

- This code as-is works with just:
  - (1) Producer
  - (1) Consumer
- If we scale to (2+) consumer's it fails
  - How can it be fixed ?

EXECUTION TRACE:
NO WHILE, 1 PRODUCER, 2 CONSUMERS

- Two threads

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

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

PRODUCER/CONSUMER SYNCHRONIZATION

- When producer threads awake, they do not check if there is any data in the buffer…
  - Need while, not if
- What if T_p puts a value, wakes T_{c1} whom consumes the value
- Then T_p has a value to put, but T_{c1}'s signal on &cond wakes T_{c2}
- There is nothing for T_{c2} consume, so T_{c2} sleeps
- T_{c1}, T_{c2}, and T_p all sleep forever
- T_{c1} needs to wake T_p to T_{c2}

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

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

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}

EXECUTION TRACE – 2

WHILE, 1 CONDITION, 1 PRODUCER, 2 CONSUMERS

- T_{c2} runs, no data to consume

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

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

TWO CONDITIONS

- Use two condition variables: empty & full
  - One condition handles the producer
  - the other the consumer

```
cond_t empty, full;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal( &full);
        Pthread_mutex_unlock(&mutex);
    }
}
```

FINAL PRODUCER/CONSUMER

- Change buffer from int, to int buffer[MAX]
- Add indexing variables

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```


FINAL P/C - 2

```
cond_t empty, full
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == MAX)                   // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal (&full);           // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait( &full, &mutex); // c3
        int tmp = get();                       // c4
```

FINAL P/C - 3

```
(Cont.)
            Pthread_cond_signal(&empty);              // c5
            Pthread_mutex_unlock(&mutex);             // c6
            printf("%d\n", tmp);
        }
    }
```

- Producer: only sleeps when buffer is full
- Consumer: only sleeps if buffers are empty

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

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 <u>can</u> be fulfilled
    - with newly available memory!
- <u>Overhead</u>
  - Many threads may be awoken which can't execute

CHAPTER 32 – CONCURRENCY PROBLEMS

February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.29

OBJECTIVES

- Chapter 32:
  - Non-deadlock concurrency bugs
  - Deadlock causes
  - Deadlock prevention

February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.30

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**

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
- Serialized access to shared memory among separate threads is not enforced (e.g. non-atomic)
- Simple example:

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 variable 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

```
    // 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
- Desire for automated tool support (IDE)

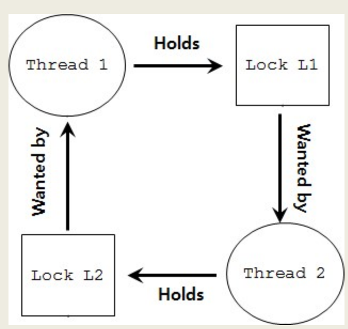
NON-DEADLOCK BUGS - 2

- 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?
    - Before threads are created, after threads exit
    - Must verify the scope
- Order violation
  - Must consider all variable accesses
  - Must known desired order

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;
}
```


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

PREVENTION – 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

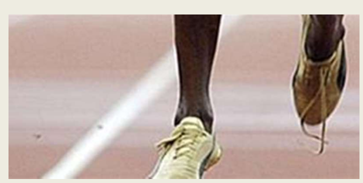
February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.51

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
- Add random delay
  - Allows one thread to win livelock race!



February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.52

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

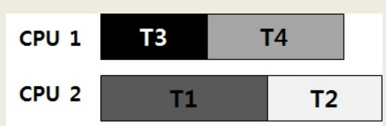
DEADLOCK AVOIDANCE VIA INTELLIGENT SCHEDULING

- 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

INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:

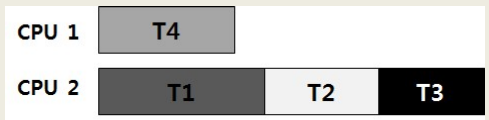


- No deadlock can occur
- Consider:

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

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?
- Many database systems employ deadlock detection and recovery techniques.

CHAPTER 13: ADDRESS SPACES

OBJECTIVES

- 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
- Chapter 16
  - Memory segments, fragmentation

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

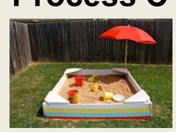
MEMORY VIRTUALIZATION - 2

- 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

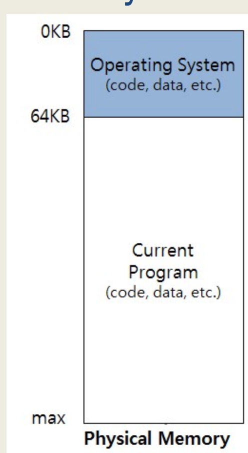


MOTIVATION FOR MEMORY VIRTUALIZATION

- 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)

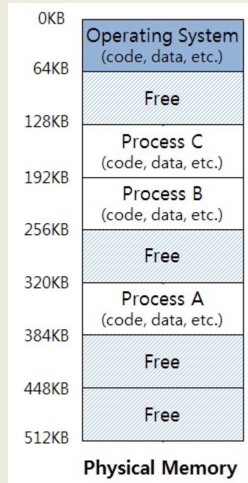
EARLY MEMORY MANAGEMENT

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



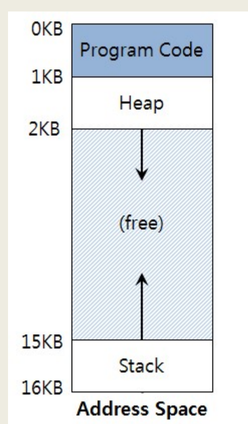
MULTIPROGRAMMING WITH SHARED MEMORY

- 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



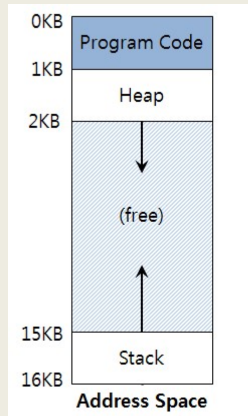
ADDRESS SPACE

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



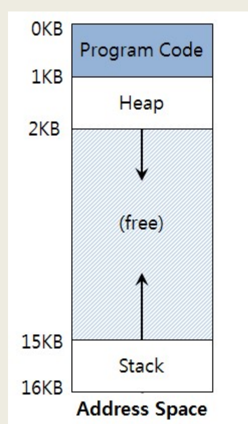
ADDRESS SPACE - 2

- 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

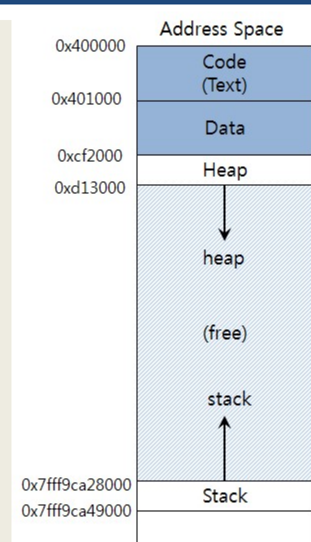
```
#include <stdio.h>
#include <stdlib.h>

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



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 evaluation memory virtualization schemes

CHAPTER 14: THE MEMORY API

February 14, 2018 TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma L11.73

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

February 14, 2018 TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma L11.74

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
```


FREE()

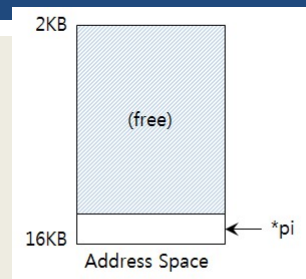
```
#include <stdlib.h>

void free(void* ptr)
```

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

VIRTUAL ADDRESS SPACE

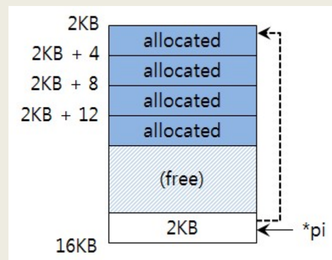
```
int *pi; // local variable
```

- Pointer is a local variable on the stack



- Malloc returns space on the heap

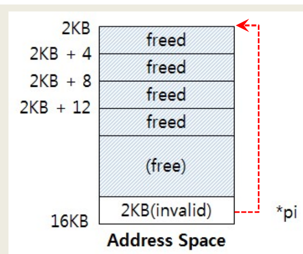
```
pi = (int *)malloc(sizeof(int)* 4);
```

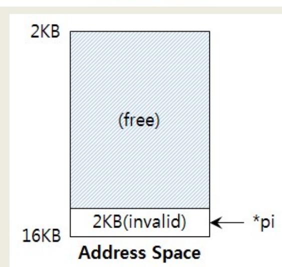


VIRTUAL ADDRESS SPACE - 2

- Releases heap space pointed to by the pointer on the stack

```
free(pi);
```





COMMON MEMORY ERRORS

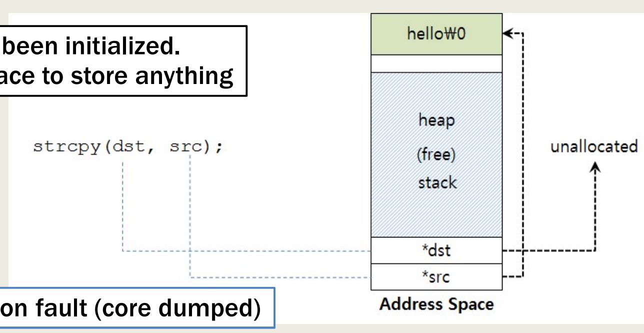
- Forgetting to malloc memory
- Unterminated string
- Uninitialized memory
- Memory leak
- Dangling pointer

FORGETTING TO MALLOC

- C is not Java
- When forgetting to maloc:

```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```

dst has not been initialized.
It has no place to store anything

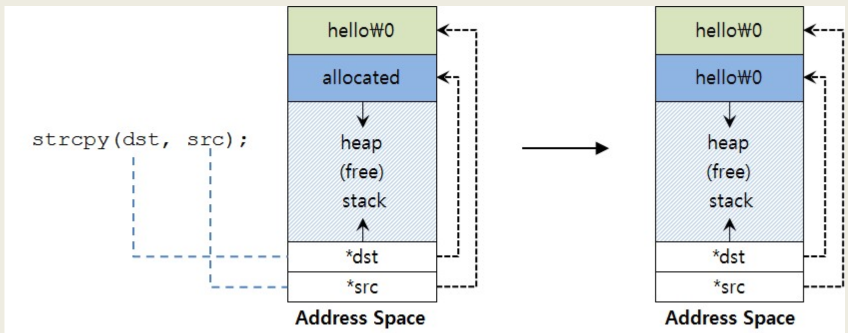


Segmentation fault (core dumped)

CORRECTION

```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```

- Why do we malloc length + 1 ?

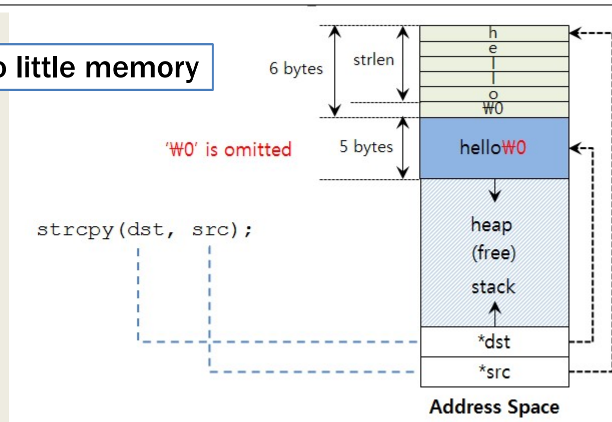


February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.81

UNTERMINATED STRING

```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```

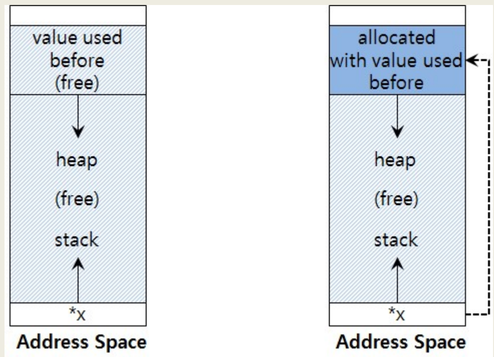
Malloc too little memory



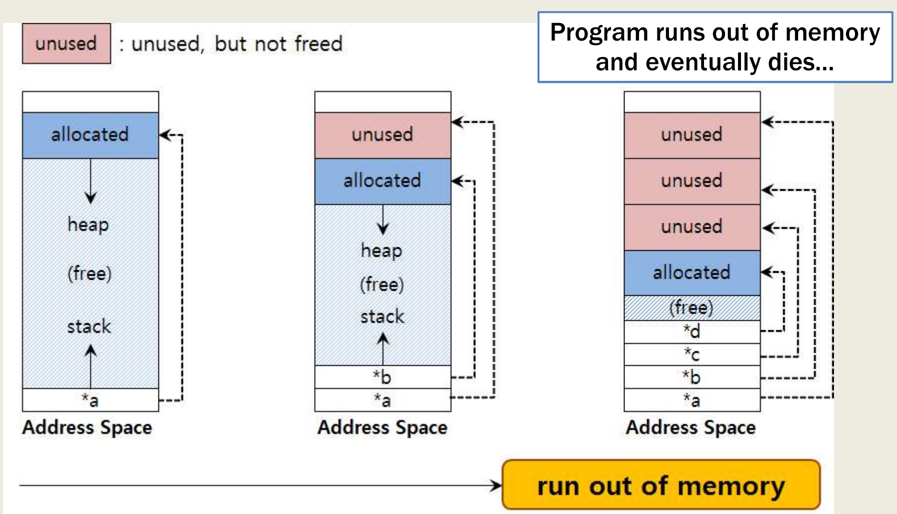
February 14, 2018 | TCSS422: Operating Systems [Winter 2018] Institute of Technology, University of Washington - Tacoma | L11.82

FORGETTING TO INITIALIZE

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```



MEMORY LEAK



```
#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?

85


```
#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?

86

QUESTIONS