TCSS 422: OPERATING SYSTEMS


Condition Variables,
Producer/Consumer

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

OBJECTIVES

- Assignment 1
- Assignment 2
- Midterm

- **Parallel programming with P-threads cont'd**
- Chapter 30 – Condition Variables
- Chapter 32 – Concurrency Problems

- **Memory Virtualization**
- Chapter 13 – Address Spaces
- Chapter 14 – Memory API

CHAPTER 30 – CONDITION VARIABLES

CONDITION VARIABLES

- There are many cases where a thread wants to wait for another thread before proceeding with execution

- Consider when a precondition must be fulfilled before it is meaningful to proceed …

CONDITION VARIABLES - 2

- Support a signaling mechanism to alert threads when preconditions have been satisfied

- Eliminate busy waiting

- Alert one or more threads to "consume" a result, or respond to state changes in the application

- Threads are placed on an **explicit queue** (FIFO) to wait for signals

- **Signal**: wakes one thread
  broadcast wakes all (ordering by the OS)

CONDITION VARIABLES - 3

- Condition variable

```
pthread_cond_t c;
```

  - Requires initialization

- Condition API calls

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

- wait() accepts a mutex parameter
  - Releases lock, puts thread to sleep

- signal()
  - Wakes up thread, awakening thread acquires lock

CONDITION VARIABLES - QUESTIONS

- **Why would we want to put waiting threads on a queue… why not use a stack?**
  - Queue (FIFO), Stack (LIFO)
  - Using condition variables eliminates busy waiting by putting threads to "sleep" and yielding the CPU.
- **Why do we want to not busily wait for the lock to become available?**
- A program has 10-threads, where 9 threads are waiting. The working thread finishes and broadcasts that the lock is available. **What happens next?**

MATRIX GENERATOR

Matrix generation example

Chapter 30
signal.c

MATRIX GENERATOR

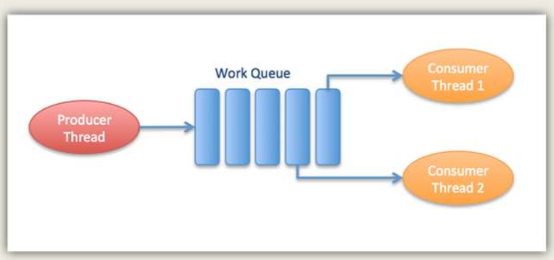
- The main thread, and worker thread (generates matrices) share a single matrix pointer.
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Let's try "nosignal.c"

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 **can** be fulfilled
    - with newly available memory!
- **Overhead**
  - Many threads may be awoken which can't execute

CHAPTER 32 – CONCURRENCY PROBLEMS

OBJECTIVES

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

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;
        …
}
```

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.38

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)

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.39

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

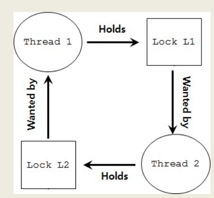
February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.40

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



February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.41

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

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.42

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

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.43

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

```c
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

```c
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

```c
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

```c
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

```c
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 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;
    }
```

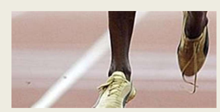
- Eliminates deadlocks

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

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

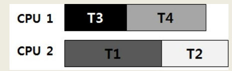
- Consider a smart scheduler
  - Scheduler knows which locks threads use
- Consider this scenario:
  - 4 Threads (T1, T2, T3, T4)
  - 2 Locks (L1, L2)
- Lock requirements of threads:

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

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.56

INTELLIGENT SCHEDULING - 2

- Scheduler produces schedule:



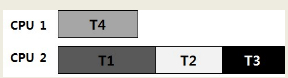
- No deadlock can occur
- Consider:

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

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.57

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

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.58

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.

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.59

CHAPTER 13: ADDRESS SPACES

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.60

OBJECTIVES – MEMORY VIRTUALIATION

- 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

February 25, 2019 | TCSS422: Operating Systems [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L11.61

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

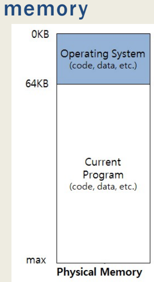
Process C

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)

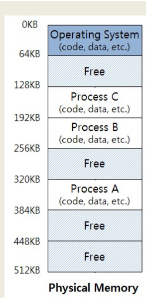
EARLY MEMORY MANAGEMENT

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



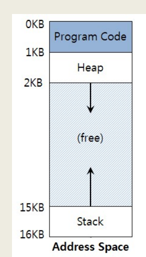
MULTIPROGRAMMING WITH SHARED MEMORY

- 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



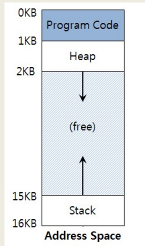
ADDRESS SPACE

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



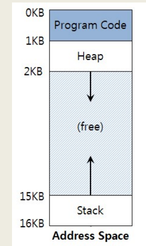
ADDRESS SPACE - 2

- 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

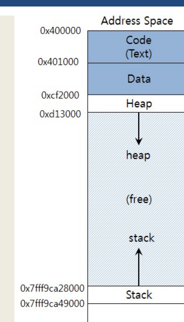
```
#include <stdio.h>
#include <stdlib.h>

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 evaluating memory virtualization schemes*

CHAPTER 14: THE MEMORY API

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

SIZEOF()

- Not safe to assume data type sizes using different compilers, systems

- Dynamic array of 10 ints

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

```
4
```

- Static array of 10 ints

```
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

What will this code do?

```
#include<stdio.h>

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

void set_magic_number_b()
{
  int b = 11111;
}

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


What will this code do?

```
#include<stdio.h>

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

void set_magic_number_b()
{
  int b = 11111;
}

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

Output:

```
$ ./pointer_error
The magic number is=53247
The magic number is=11111
```

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

Why?

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()

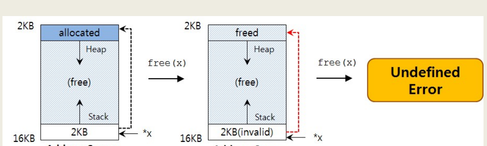
```
#include <stdlib.h>

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

QUESTIONS