CONCURRENCY: DEADLOCK

Shivaram Venkataraman
CS 537, Spring 2020
Midterm is on Wednesday 3/12 at 5.30pm-7pm, details on Piazza

Venue: If your last name starts with A-R, go to Humanities 3650
else (last name starts with S-Z), go to Psych 113

Bring your ID!
Calculators allowed
No cheat sheet
AEFIS SURVEY RESULTS

Thank you for the responses!

Points to improve

1. Upload slides earlier
2. Go slower
3. More details on project

31%
Response Rate
92 of 300
Concurrency

How do we build semaphores?

What are common pitfalls with concurrent execution?
RECAP
CONCURRENCY OBJECTIVES

**Mutual exclusion** (e.g., A and B don’t run at same time)
solved with *locks*

**Ordering** (e.g., B runs after A does something)
solved with *condition variables* and *semaphores*
**SEMAPHORE OPERATIONS**

**Wait or Test:** `sem_wait(sem_t*)`  
Decrement sem value by 1, Waits if value of sem is negative (< 0)

**Signal or Post:** `sem_post(sem_t*)`  
 Increment sem value by 1, then wake a single waiter if exists

Value of the semaphore, when negative = the number of waiting threads
Typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} zem_t;

void zem_init(zem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}

zem_wait(): Waits while value <= 0, Decrement
zem_post(): Increment value, then wake a single waiter
Build Zemaphore from Locks and CV

```c
zem_wait(zem_t *s) {
    lock_acquire(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond);
    s->value--;
    lock_release(&s->lock);
}
```

```c
zem_post(zem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```

zem_wait(): Waits while value <= 0, Decrement
zem_post(): Increment value, then wake a single waiter
Semaphores are equivalent to locks + condition variables
  – Can be used for both mutual exclusion and ordering

Semaphores contain state
  – How they are initialized depends on how they will be used
  – Init to 0: Join (1 thread must arrive first, then other)
  – Init to N: Number of available resources

sem_wait(): Decrement and waits if value < 0
sem_post() or sem_signal(): Increment value, then wake a single waiter (atomic)
Can use semaphores in producer/consumer and for reader/writer locks
CONCURRENCY BUGS
CONCURRENCY STUDY

Lu et al. [ASPLOS 2008]:
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.
Thread 1:

```c
pthread_mutex_lock(&lock);
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}
pthread_mutex_unlock(&lock);
```

Thread 2:

```c
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```
Thread 1:

```c
void init() {

    mThread = PR_CreateThread(mMain, ...);

    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);

    ... 
}
```

Thread 2:

```c
void mMain(...) {

    mutex_lock(&mtLock);
    while (mtInit == 0)
        Cond_wait(&mtCond, &mtLock);
    Mutex_unlock(&mtLock);

    mState = mThread->State;

    ...
}
```
No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does
Thread 1:
lock(&A);
lock(&B);

Thread 2:
lock(&B);
lock(&A);
Circular Dependency

Thread 1 holds Lock A

Lock B holds Lock A

Thread 2 holds Lock B

Lock A wanted by Thread 2

Lock B wanted by Thread 1

Thread 1 wanted by Lock B

Lock A wanted by Thread 1

Thread 2 wanted by Lock B
Thread 1:
lock(&A);
lock(&B);

Thread 2:
lock(&B);
lock(&A);

Thread 1:  Thread 2:
NON-CIRCULAR DEPENDENCY

Thread 1 holds Lock A

Lock A wanted by Thread 2

Lock B wanted by Thread 1
```c
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = malloc(sizeof(*rv));
    mutex_lock(&s1->lock);
    mutex_lock(&s2->lock);
    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    }
    mutex_unlock(&s2->lock);
    mutex_unlock(&s1->lock);
}

Thread 1: rv = set_intersection(setA, setB);
Thread 2: rv = set_intersection(setB, setA);
```
ENCAPSULATION

Modularity can make it harder to see deadlocks

Solution?

```c
if (m1 > m2) {
    // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

Any other problems?
void foo(pthread_mutex_t *t1, pthread_mutex_t *t2, , pthread_mutex_t *t3) {
    pthread_mutex_lock(t1);
    pthread_mutex_lock(t2);
    pthread_mutex_lock(t3);
    do_stuffs();
    pthread_mutex_unlock(t1);
    pthread_mutex_unlock(t2);
    pthread_mutex_unlock(t3);
}

T1 foo(a,b,c)  T1 foo(a,b,c)  T1 foo(a,b,c)
T2 foo(b,c,a)  T2 foo(a,b,c)  T2 foo(b,c,e)
T3 foo(c,a,b)  T3 foo(a,b,c)  T3 foo(f,e,a)
Deadlock Theory

Deadlocks can only happen with these four conditions:

1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Can eliminate deadlock by eliminating any one condition
1. MUTUAL EXCLUSION

Problem: Threads claim exclusive control of resources that they require.
Strategy: Eliminate locks!
Try to replace locks with atomic primitive:

```c
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1; // success
    }
    return 0; // failure
}
```
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
}

void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    do {
        n->next = head;
    } while (!CompAndSwap(&head, n->next, n));
}
2. HOLD-AND-WAIT

Problem: Threads hold resources allocated to them while waiting for additional resources

Strategy: Acquire all locks atomically **once**. Can release locks over time, but cannot acquire again until all have been released

How to do this? Use a meta lock:

Disadvantages?
3. NO PREEMPTION

Problem: Resources (e.g., locks) cannot be forcibly removed from threads that are

Strategy: if thread can’t get what it wants, release what it holds

top:

    lock(A);
    if (trylock(B) == -1) {
        unlock(A);
        goto top;
    }

...
4. CIRCULAR WAIT

Circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.

Strategy:
- decide which locks should be acquired before others
- if A before B, never acquire A if B is already held!
- document this, and write code accordingly

Works well if system has distinct layers
CONCURRENCY SUMMARY SO FAR

Motivation: Parallel programming patterns, multi-core machines

Abstractions, Mechanisms
- Spin Locks, Ticket locks
- Queue locks
- Condition variables
- Semaphores

Concurrent Bugs
LOOKING AHEAD

Midterm on Thursday!

Thursday class:
  Summary,
  More quizzes?
  In-class OH?