CONCURRENCY: DEADLOCK

Shivaram Venkataraman
CS 537, Spring 2019
Midterm is on Wednesday 3/13 at 5.15pm, details on Piazza
Venue: If your last name starts with A-L, go to VanVleck B102
    else (last name starts with M-Z), go to VanVleck B130

Bring your ID! Calculators allowed, no cheat sheet
Review session, Office hours at 5.30pm at Noland Hall, Room 132

Fill out mid semester course evaluation? https://aefis.wisc.edu/
Concurrency

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*
SUMMARY: CONDITION VARIABLES

**wait**(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

**signal**(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing
SUMMARY: SEMAPHORES

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

```plaintext
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
CONCURRENCE BUGS
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.
Fix Atomicity Bugs with Locks

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:
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    ...
}

Thread 2:
void mMain(...) {
    ...
    mutex_lock(&mtLock);
    while (mtInit == 0)  
        Cond_wait(&mtCond, &mtLock);
    Mutex_unlock(&mtLock);
    mState = mThread->State;
    ...
}
DEADLOCK

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 wanted by Thread 2

Lock A wanted by Thread 1

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

Thread 2:  
lock(&A);
lock(&B);
NON-CIRCULAR DEPENDENCY

Thread 1 holds Lock A

Lock B is wanted by Thread 2

Lock A is wanted by Thread 2
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?
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 add (int *val, int amt) {
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}

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

void add (int *val, int amt) {
    do {
        int old = *value;
    } while(!CompAndSwap(____, ____));
}
WAIT-FREE ALGORITHM: LINKED LIST INSERT

```c
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
}
```

```c
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;
    }
    ...

Disadvantages?
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

Concurrency Bugs
MIDTERM REVIEW