CS 423
Operating System Design: Semaphores and Deadlocks
03/14

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

Next week: Spring break, enjoy!

Wednesday after that: wrap up concurrency/review

Friday: Midterm

Take home – you will have 24 hours to complete

Online submission (more details to follow)

AGENDA / LEARNING OUTCOMES

Semaphores

Today:

- Continue semaphores

RECAP

Producer/Consumer: Two CVs and WHILE

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        while (numfull == max) // p2
            Cond_wait(&empty, &m); // p3
        do_fill(i);  // p4
        Cond_signal(&fill); // p5
        Mutex_unlock(&m); //p6
    }
}
```

```
void *consumer(void *arg) {
    while (1) {
        Mutex_lock(&m);
        while (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

No concurrent access to shared state

Every time lock is acquired, assumptions are reevaluated

A consumer will get to run after every do_fill()

A producer will get to run after every do_get()

HOARE VS MESA SEMANTICS

- Mesa (used widely)
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor
  - Not necessarily the waiter runs next
- Hoare (almost no one uses)
  - Signal gives processor and lock to waiter
  - Waiter runs when woken up by signaler
  - When waiter finishes, processor/lock given back to signaler

Semaphores

Condition variables have no state (other than waiting queue)

- Programmer must track additional state

Semaphores have state: track integer value

- State cannot be directly accessed by user program, but state determines behavior of semaphore operations

Equivalence

Semaphores are equally powerful to Locks+CVs

- what does this mean?

One might be more convenient, but that's not relevant

Equivalence means each can be built from the other

SEMAPHORE OPERATIONS

Allocate and Initialize

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

User cannot read or write value directly after initialization

SEMAPHORE OPERATIONS

Wait or Test: sem_wait(sem_t*)
Decrements 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

Wait and Signal are atomic

BINARY Semaphore (LOCK)

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock_t->sem, 1);
}

void acquire(lock_t *lock) {
    sem_wait(&lock_t->sem);
}

void release(lock_t *lock) {
    sem_post(&lock_t->sem);
}
```

sem_init(sem_t*, int initial)
sem_wait(sem_t*): Decrement, wait if value < 0
sem_post(sem_t*): Increment value
then wake a single waiter

END RECAP

Reader/Writer Locks

Let multiple reader threads grab lock (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Let us see if we can understand code…

Reader/Writer Locks

```
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}
```

Reader/Writer Locks

```
void rwlock_acquire_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers++;
        if (rw->readers == 1)
            sem_wait(&rw->writelock);
        sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers--;
        if (rw->readers == 0)
            sem_post(&rw->writelock);
        sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
    // who runs?
T4: acquire_readlock()
    // what happens?
T5: acquire_readlock()
    // where blocked?
T3: release_writelock()
    // what happens next?

Reader/Writer Locks

```
void rwlock_acquire_readlock(rwlock_t *rw) {
       sem_wait(&rw->lock);
       rw->readers++;
       if (rw->readers == 1)
           sem_wait(&rw->writelock);
       sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
       sem_wait(&rw->lock);
       rw->readers--;
       if (rw->readers == 0)
           sem_post(&rw->writelock);
       sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T4: release_readlock()
    // what happens?
    // what's the problem?

Producer/Consumer: Semaphores #2

Single producer thread, single consumer thread
Shared buffer with **N elements** between producer and consumer
Use 2 semaphores

- emptyBuffer: Initialize to ___________
- fullBuffer: Initialize to ____________

```
Producer
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer[i]);
    i = (i+1)%N;
    sem_post(&fullBuffer);
}
```

```
Consumer
j = 0;
While (1) {
    sem_wait(&fullBuffer);
    Use(&buffer[j]);
    j = (j+1)%N;
    sem_post(&emptyBuffer);
}
```

Producer/Consumer: Semaphore #3

Final case:

- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element

Build Zemaphore!

```
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

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

```
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

Zemaphores	
Locks	CV's

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

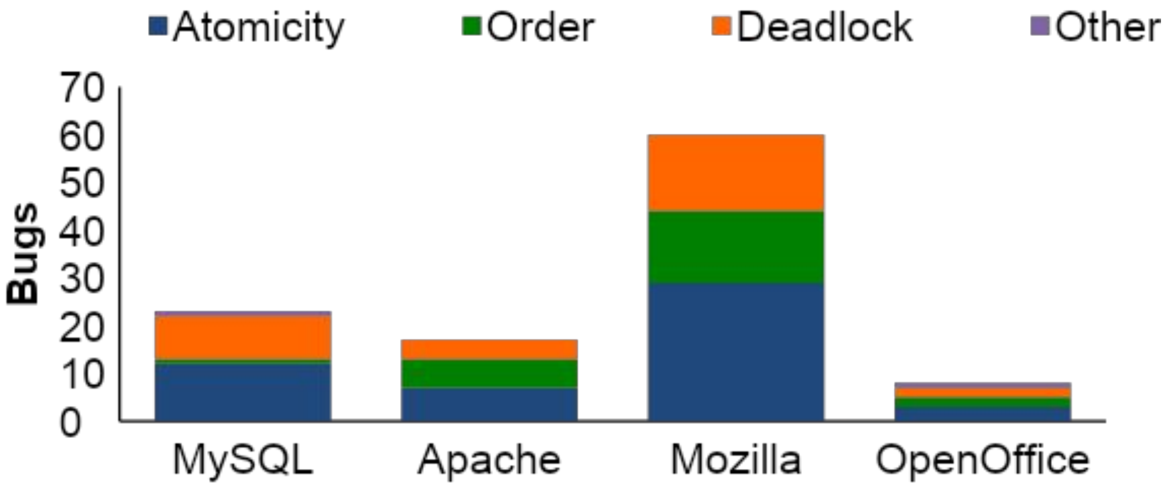
Sem_wait(): Decrement and then wait if < 0 (atomic)

Sem_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer and for reader/writer locks

CONCURRENCY BUGS

CONCURRENCY STUDY



Lu *etal.* [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:

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

Thread 2:

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

FIX ORDERING BUGS WITH CONDITION VARIABLES

Thread 1:

```
void init() {
    …

    mThread =
PR_CreateThread(mMain, …);


    …
}
```

Thread 2:

```
void mMain(…) {
  …

  mState = mThread->State;
  …
}
```

FIX ORDERING BUGS WITH CONDITION VARIABLES

Thread 1:

```
void init() {
    …

    mThread =
PR_CreateThread(mMain, …);

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

    …
}
```

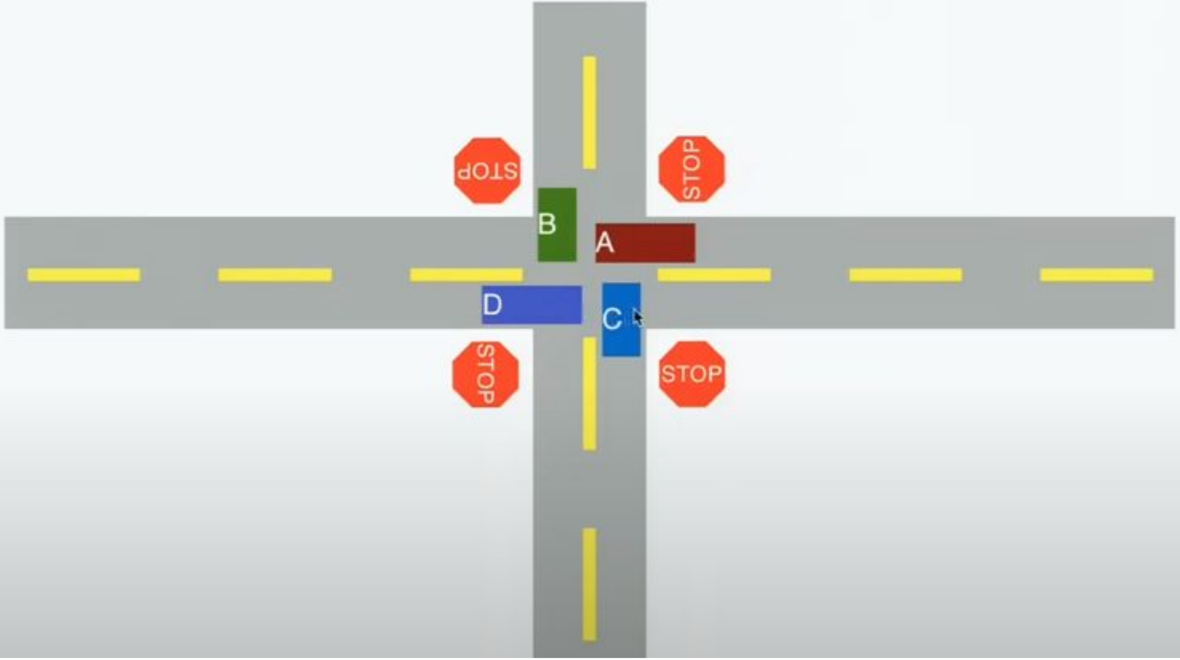
Thread 2:

```
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

4 cars move forward same time
Is this deadlocked?



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

CODE EXAMPLE

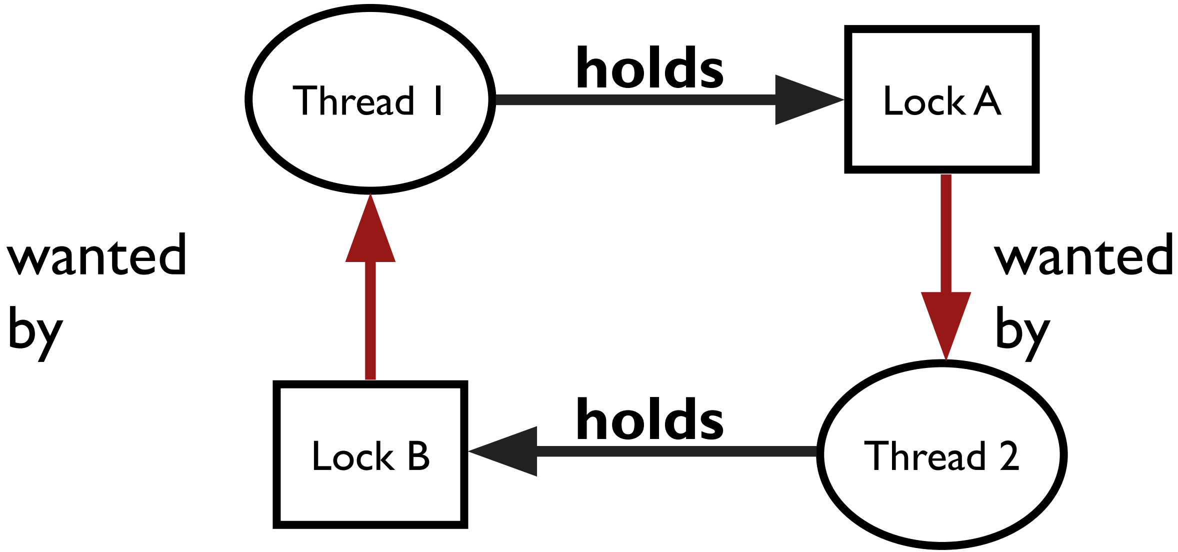
Thread 1:

```
lock(&A);
lock(&B);
```

Thread 2:

```
lock(&B);
lock(&A);
```

CIRCULAR DEPENDENCY



FIX DEADLOCKED CODE

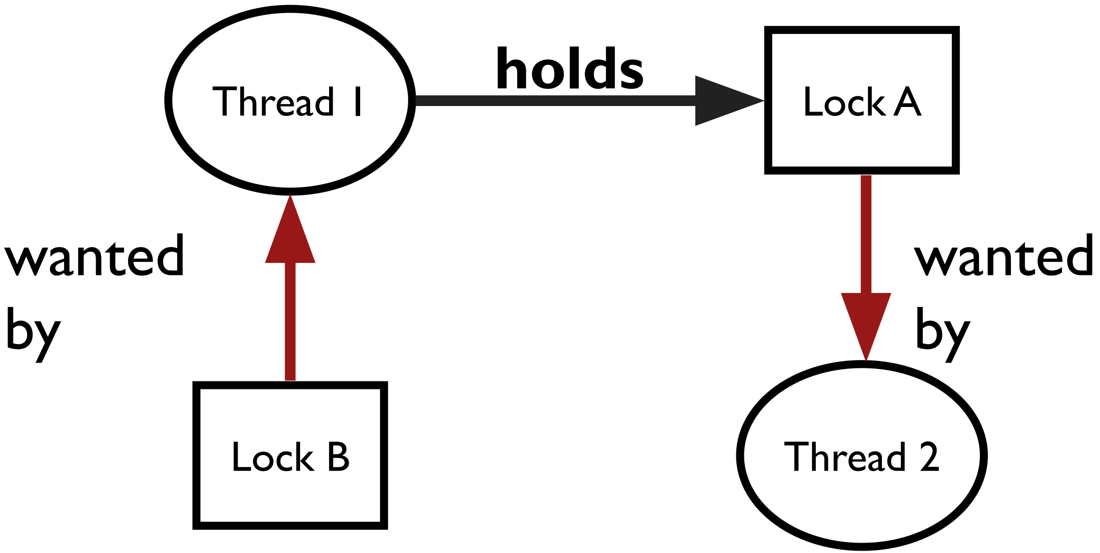
```
Thread 1:

lock(&A);
lock(&B);
```

```
Thread 2:

lock(&B);
lock(&A);
```

```
Thread 1
```

```
Thread 2
```

NON-CIRCULAR DEPENDENCY



```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);
}
```

Can there be a deadlock?

ENCAPSULATION

Modularity can make it harder to see deadlocks

Solution?

```
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?

1. Mutual Exclusion

Problem: Threads claim exclusive control of resources that they require

Strategy: Eliminate locks!

Try to replace locks with atomic HW primitive:

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

WAIT-FREE ADD

```
void add (int *val, int amt)
{
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}
```

```
void add (int *val, int amt) {
    do {
        int old = *val;
    } while(!CompAndSwap(val,      , old+amt);
}
```

WAIT-FREE ALGORITHM: LINKED LIST INSERT

```
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,
                          ___,____));
}
```

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:

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

3. NO PREEMPTION

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

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?

Solution?

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

Lock Ordering in XV6

Creating a file requires simultaneously holding:

- a lock on the directory,
- a lock on the new file's inode,
- a lock on a disk block buffer,
- idelock,
- ptable.lock

Always acquires locks in order listed

Linux has similar rules…

Summary

When in doubt about **correctness**, better to limit concurrency
(i.e., add unnecessary locks, one big lock)

Concurrency is hard, encapsulation makes it harder!

Have a strategy to avoid deadlock and stick to it

Choosing a lock order is probably most practical for reasonable performance

CONCURRENCY SUMMARY SO FAR

Motivation: Parallel programming patterns, multi-core machines

Abstractions, Mechanisms

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

Concurrency Bugs

NEXT STEPS

Next class: Deadlocks