CONCURRENCY: DEADLOCK

Shivaram Venkataraman
CS 537, Spring 2023
ADMINISTRIVIA

Grades
Project 3, Project 4 – Check Piazza
Midterm 1 – Check Canvas post

Upcoming
Project 5 – Out today! Check your groups on Canvas!
Midterm 2 – Conflict form on Piazza

Practice exams – this week!
April 4th
5:45 pm – 7:15 pm
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* → xchg

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

Parent thread

child

join() ←

lock() →

only want 1 thread to be active

unlock()
**SEMAPHORES**

**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 __lock_t {
    sem_t sem;
} lock_t;

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

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

void release(lock_t *lock) {
    sem_post(&lock->sem);
}
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…
typedef struct _rwlock_t {
    sem_t lock;  // two semaphores
    sem_t writelock;  // number of active reader threads
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);  // Initialized similar to binary lock
    sem_init(&rw->writelock, 1);
}
READER/WRITER LOCKS

13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1) {
17         sem_wait(&rw->writelock);
18         sem_post(&rw->lock);
19     }
20 }
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0) {
25         sem_post(&rw->writelock);
26         sem_post(&rw->lock);
27     }
28 }
29 rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
31 rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }

// what happens next?

T1: acquire_readlock() 1 0 1
T2: acquire_readlock() 2 0 1
T3: acquire_writelock() -1
T2: release_readlock() 1 -1
T1: release_readlock() 0 0
T4: acquire_readlock() 0 0 0
T5: acquire_readlock() 0 0 0
T3: release_writelock() 0 0 0

Not fair!

ensure no writers are present
protect readers from racing grab write lock.

present - from racing grab write lock.

// what happens next?
T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()

→ Running. We allow multiple reader threads

T4: acquire_writelock()
T5: acquire_writelock()
T6: acquire_readlock()

→ Waiting for write lock

T8: acquire_writelock()
T7: acquire_readlock()
T9: acquire_readlock()

→ Waiting for read lock
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
SUMMARY: SEMAPHORES

Semaphores are equivalent to locks + condition variables
  – Can be used for both mutual exclusion and ordering

Semaphores contain **state** — good for programmer
  – 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

Can use semaphores in producer/consumer and for reader/writer locks
CONCURRENCY 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.
Thread 1:
```
pthread_mutex_lock(&lock);
if (thd->proc_info) {
    ...
    fputs(thd->proc_info, ...);
    ...
}
pthread_mutex_unlock(&lock);
```

Thread 2:
```
pthread_mutex_lock(&lock);
if (thd->proc_info) {
    thread->proc_info = NULL;
}
```
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;
    ...}
```
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);

Possible Interleaving:

T1: lock(&A) → acquires
T2: lock(&B) → acquires
T1: lock(&B) → blocked
T2: lock(&A) → blocked

Order of locks:
No other coordination
No way to release or preempt

Deadlock
CIRCULAR DEPENDENCY

Thread 1 holds Lock A
Lock A holds Lock B
Lock B holds Thread 2

wanted by

edge from thread to lock it holds

edge from lock to thread that wants lock
Thread 1:
lock(&A);
lock(&B);

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

Thread 1
lock(&A); ✓
lock(&B); ✓

Thread 2
→ lock(&A); X blocked
   lock(&B);
NON-CIRCULAR DEPENDENCY

Thread 1 holds Lock A

Lock B is wanted by Lock A

Lock A is wanted by Thread 2

Thread 1 holds Lock A

removed cycle from this graph
```
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);
```

Modularity can make it harder to see deadlocks — always lock s1 first before s2

*set_intersection* is a function used in multiple threads.
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)
T2 foo(b,c,a)
T3 foo(c,a,b)

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

T1: A, B
T2: B, C
T3: C, A

No!
Same order

Deadlock!
Deadlock Theory

Deadlocks can only happen with these four conditions:

1. mutual exclusion
2. hold-and-wait → thread grab some locks & wait for others
3. no preemption → no way to ask a thread to release lock
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 e.g. xchg

```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:

\[ \text{lock (A)} \rightarrow \text{holding lock A} \]
\[ \vdots \]
\[ \text{lock (B)} \rightarrow \text{wait} \]
\[ \vdots \]
\[ \text{unlock (B)} \rightarrow \text{wait} \]
\[ \text{lock (C)} \rightarrow \text{wait} \]

Disadvantages?

get all of locks or none of them

limits concurrency
3. NO PREEMPTION

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

Strategy: If a thread can’t get what it wants, release what it holds.

```
top:
lock(A);
if (trylock(B) == -1) {
  unlock(A);
  sleep(??)
  goto top;
}
...
```

Disadvantages?

Unfairness:

"Live lock":

1. One thread
2. Back off
3. Random
4. Exponential back off
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
LOOKING AHEAD

Project 5 out!

Midterm 2 on concurrency

Next: New Module on Persistence