CONCURRENCY: DEADLOCK

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Midsemester grades

Upcoming

- Project 4 deadline
- Midterm 2

Shivaram travel

AGENDA / LEARNING OUTCOMES

- Concurrency
  - How do we build semaphores?
  - What are common pitfalls with concurrent execution?

RECAP

SEMAPHORES

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

Value of the semaphore, when negative = the number of waiting threads

Binary Semaphore (Lock)

```
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);
}
```

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

READER/WRITER LOCKS

```c
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);
}
```

Multiple reader threads can grab lock

Only one writer thread can grab lock

- No other readers then

READER/WRITER LOCKS

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()

```
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); }
```

READER/WRITER LOCKS

```c
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); }
```

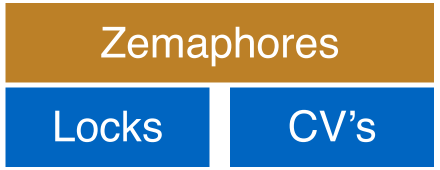
T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()
// what happens next?

BUILD ZEMAPHORE!

zem_wait(): Waits while value <= 0, Decrement

zem_post(): Increment value, then wake a single waiter

```
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);
}
```



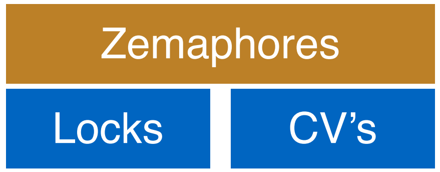
BUILD ZEMAPHORE FROM LOCKS AND CV

```
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**

- 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

QUIZ 13

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()

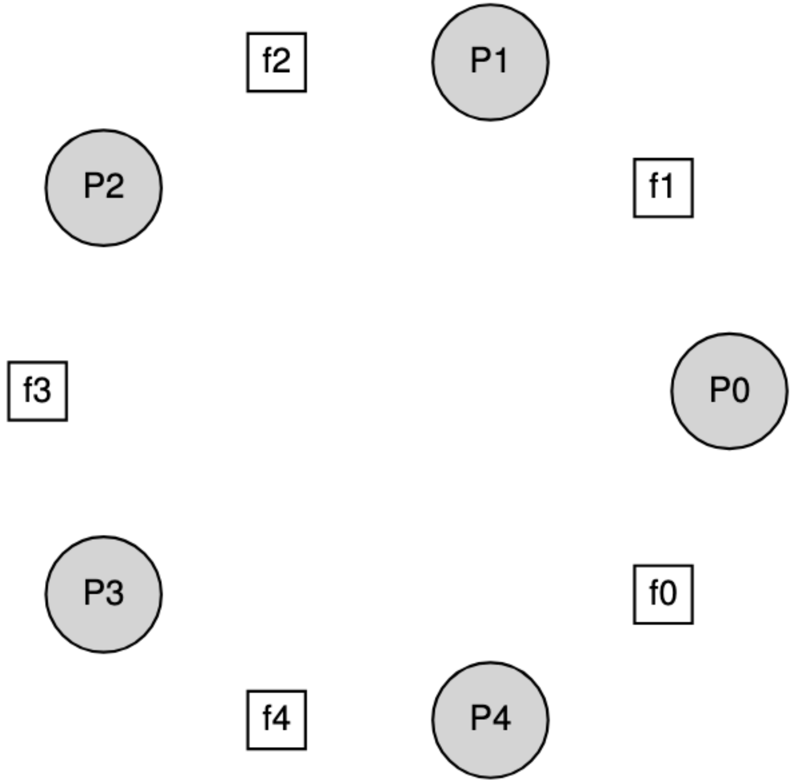
T4: acquire_writelock()
T5: acquire_writelock()
T6: acquire_readlock()

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

```
typedef struct __barrier_t {
        sem_t sem1,sem2;
        int at_barrier;
        int total_threads;
} barrier_t;

barrier_t b;

void init(barrier_t *b, int num_th) {
        b->total_threads = num_th;
        b->at_barrier = 0;
        sem_init(&b->sem1,0,X);
        sem_init(&b->sem2,0,Y);

}

void barrier(barrier_t *b) {
        sem_wait(&(b->sem1));
        b->at_barrier++;
        if (b->at_barrier < b->total_threads) {
                sem_post(&b->sem1);
                sem_wait(&b->sem2);
                sem_post(&b->sem2);
        } else {
                //finish code here
        }
}
```

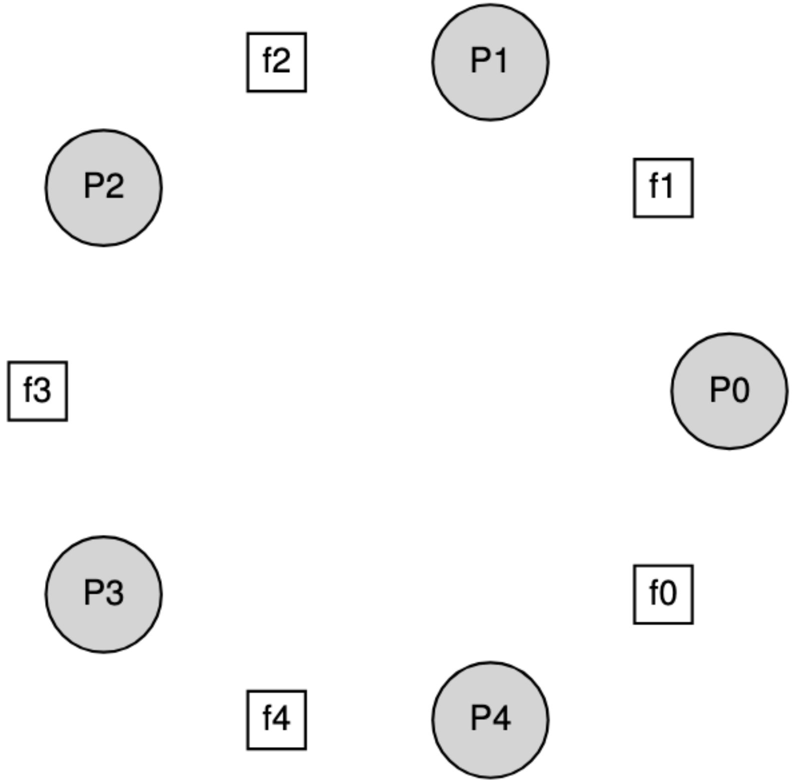
CONCURRENCY BUGS

DINING PHILOSOPHERS PROBLEM



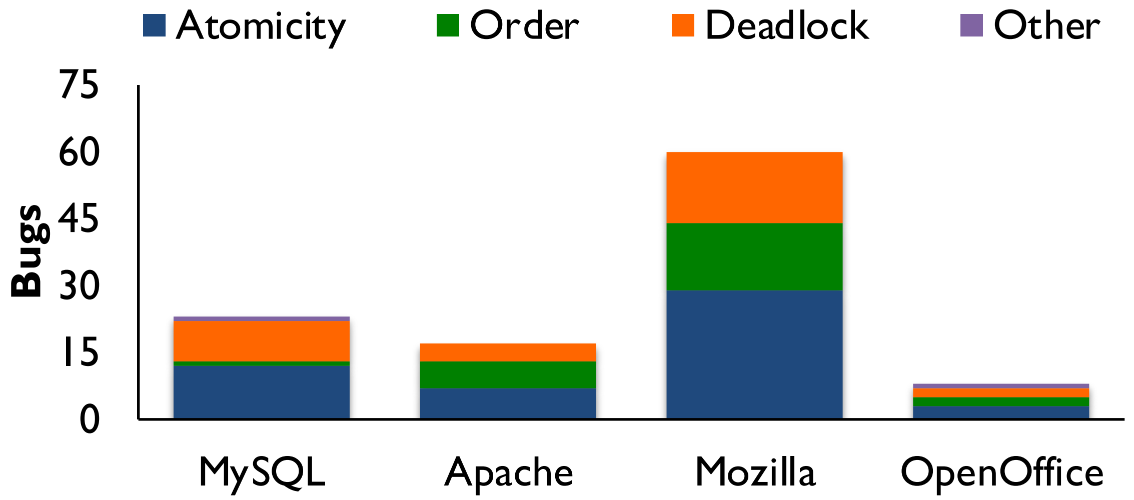
```
 while (1) {
     think();
     get_forks(p);
     eat();
     put_forks(p);
 }

void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

DINING PHILOSOPHERS PROBLEM



```
void get_forks(int p) {
  if (p == 4) {
      sem_wait(&forks[right(p)]);
      sem_wait(&forks[left(p)]);
  } else {
      sem_wait(&forks[left(p)]);
      sem_wait(&forks[right(p)]);
  }
}
```

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:

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

Thread 2:

```
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

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

CODE EXAMPLE

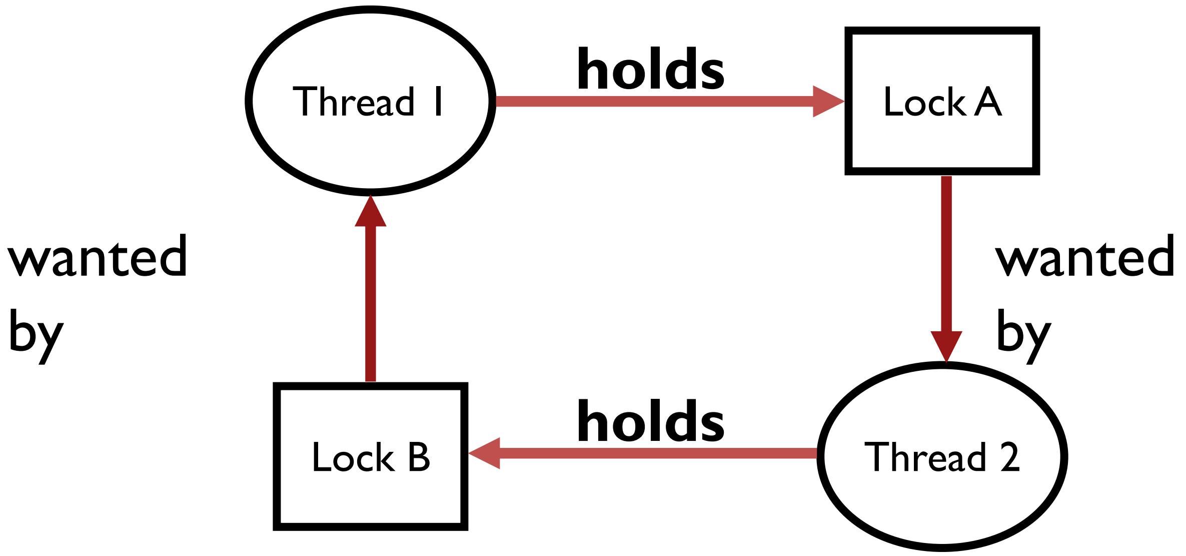
Thread 1:

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

Thread 2:

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

CIRCULAR DEPENDENCY



FIX DEADLOCKED CODE

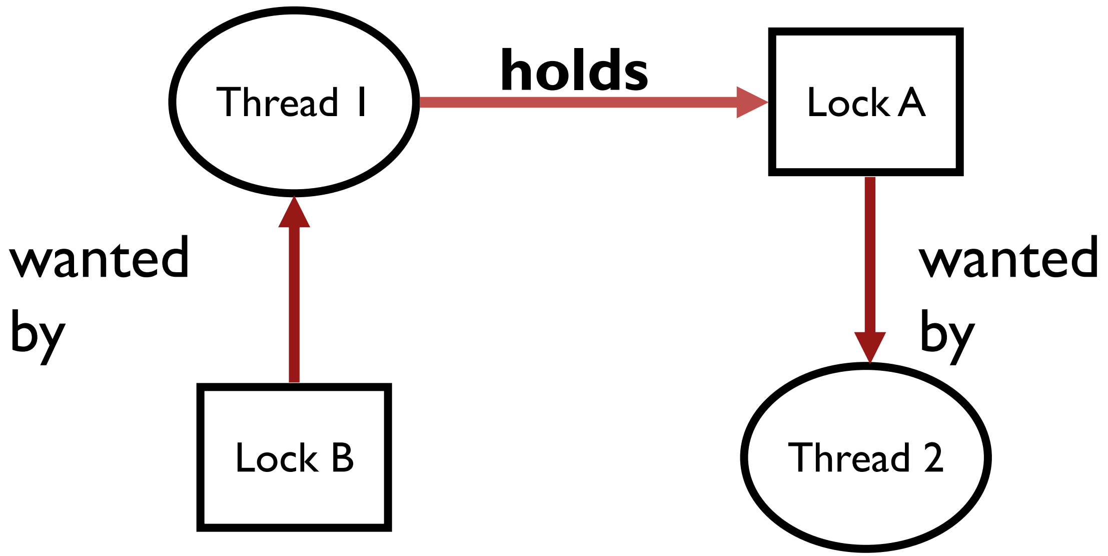
```
Thread 1:

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

```
Thread 2:

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

```
Thread 1
```

```
Thread 2
```

NON-CIRCULAR DEPENDENCY



```
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);
}
```

Modularity can make it harder to see deadlocks

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

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 e.g. xchg

```
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 holding them

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

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

LOOKING AHEAD

Midterm 2 review!