

# MIDTERM 2 REVIEW

Shivaram Venkataraman

CS 537, Fall 2024

# ADMINISTRIVIA

Midsemester grades

Upcoming

Midterm 2

Project 4

Shivaram travel

# AGENDA / LEARNING OUTCOMES

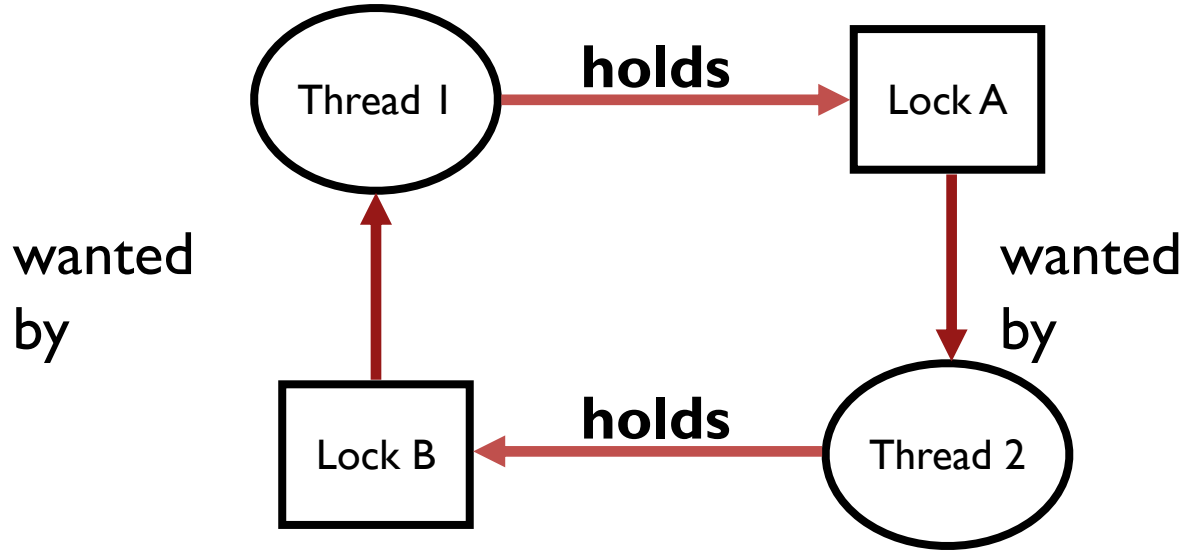
## Concurrency

What are common pitfalls with concurrent execution?

## Summary of Concurrency

**RECAP**

# DEADLOCK: CIRCULAR DEPENDENCY



No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

# 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

# QUIZ 14

```
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(a,b,c)

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



# CONCURRENCY REVIEW

# THREAD VS. PROCESS

Multiple threads within a single process share:

- Process ID (PID)
- Address space: Code (instructions), Most data (heap)
- Open file descriptors
- Current working directory
- User and group id

Each thread has its own

- Thread ID (TID)
- Set of registers, including Program counter and Stack pointer
- Stack for local variables and return addresses  
(in same address space)

# THREAD SCHEDULES

## Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

## Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

# LOCKS

Goal: Provide mutual exclusion (**mutex**)

Allocate and Initialize

- **Pthread\_mutex\_t** mylock = PTHREAD\_MUTEX\_INITIALIZER;

Acquire

- Acquire exclusion access to lock;
- Wait if lock is not available (some other process in critical section)
- Spin or block (relinquish CPU) while waiting
- **Pthread\_mutex\_lock**(&mylock);

Release

- Release exclusive access to lock; let another process enter critical section
- **Pthread\_mutex\_unlock**(&mylock);

# LOCKS USING ATOMIC INSTRUCTIONS

```
// return what was pointed to by addr
// at the same time, store newval into addr
int xchg(int *addr, int newval) {
    int old = *addr;
    *addr = newval;
    return old;
}

int CompareAndSwap(int *addr, int ex, int n) {
    int actual = *addr;
    if (actual == ex)
        *addr = n;
    return actual;
}

void acquire(lock_t *lock) {
    while(CompareAndSwap(&lock->flag, 0, 1) == 1 ) ;
    // spin-wait (do nothing)
}
```



# FAIRNESS USING TICKET LOCKS

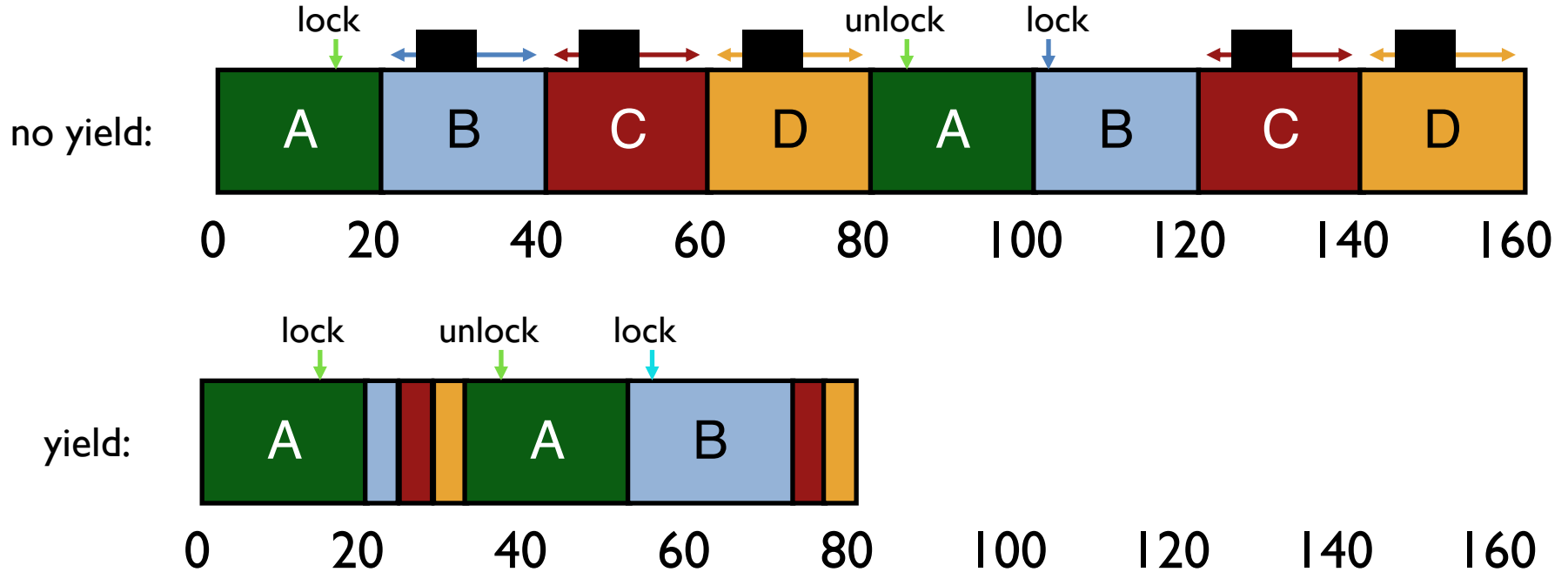
```
typedef struct __lock_t {
    int ticket;
    int turn;
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    // spin
    while (lock->turn != myturn);
}

void release(lock_t *lock) {
    FAA(&lock->turn);
}
```

# YIELD INSTEAD OF SPIN



# BLOCK WHEN WAITING: FINAL CORRECT LOCK

```
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;
```

```
void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        setpark(); // notify of plan
        l->guard = false;
        park(); // unless unpark()
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(qremove(l->q));
    l->guard = false;
}
```

# CONCURRENT DATA STRUCTURES

Simple approach: Add a lock to each method?!

Check for scalability – weak scaling, strong scaling

Avoid cross-thread, cross-core traffic

- Per-core counter

- Buckets in hashtable

- Keep critical sections small!

# PRACTICE QUESTIONS

```
volatile int balance = 0;
```

```
void *mythread(void *arg) {  
    int result = 0;  
    result = result + 200;  
    balance = balance + 200;  
    printf("Result is %d\n", result);  
    printf("Balance is %d\n", balance);  
    return NULL;  
}
```

```
int main() {  
    pthread_t p1, p2;  
    pthread_create(&p1, NULL, mythread, "A");  
    pthread_create(&p2, NULL, mythread, "B");  
    pthread_join(p1, NULL);  
    pthread_join(p2, NULL);  
    printf("Final Balance is %d\n", balance);  
}
```

Thread p1 prints "Result is %d\n", what value?

- A. Due to race conditions, "result" may have different values on different runs.
- B. 0
- C. 200
- D. 400
- E. A constant value, but none of the above

Thread p1 prints "Balance is %d\n", what value?

- A. Due to race conditions, "result" may have different values on different runs.
- B. 0
- C. 200
- D. 400
- E. A constant value, but none of the above

```

Thread 0                               Thread 1
1000 mov 2000, %ax
1001 add $1, %ax
----- Interrupt ----- Interrupt -----
                               1000 mov 2000, %ax
                               1001 add $1, %ax
----- Interrupt ----- Interrupt -----
1002 mov %ax, 2000 37.
----- Interrupt ----- Interrupt -----
                               1002 mov %ax, 2000 38.
----- Interrupt ----- Interrupt -----
1003 sub $1, %bx
1004 test $0, %bx
----- Interrupt ----- Interrupt -----
                               1003 sub $1, %bx
----- Interrupt ----- Interrupt -----
1005 jgt .top

```

Incrementing a variable many times in a loop. Assume that the %bx register begins with the value 3, (each thread performs the loop 3 times).

Assume the code is loaded at address 1000 and that the memory address 2000 originally contains the value 0.

Determine the contents of the memory address 2000 AFTER that assembly instruction

37

38

A. 1

B. 2

C. 3

D. 4

E. None of the above

```
typedef struct __lock_t {
    int flag;
} lock_t;
void init(lock_t *lock) {
    lock->flag = abc;
}

void acquire(lock_t *lock) {
    while(xchg(&lock->flag, xyz) == xyz)
        ; //spin-wait
}
void release(lock_t *lock) {
    lock->flag = 1;
}
```

What should be the value of abc?

- A. 0
- B. 1
- C. 2
- D. 0 or 2
- E. 0 or 1

What should be the value of xyz?

- A. 0
- B. 1
- C. 2
- D. 0 or 2
- E. 0 or 1



# 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  $\geq 1$  thread is waiting)
- if there is no waiting thread, just return, doing nothing

# JOIN IMPLEMENTATION: CORRECT

Parent:

```
void thread_join() {  
    Mutex_lock(&m);           // w  
    if (done == 0)           // x  
        Cond_wait(&c, &m); // y  
    Mutex_unlock(&m);        // z  
}
```

Child:

```
void thread_exit() {  
    Mutex_lock(&m);           // a  
    done = 1;                // b  
    Cond_signal(&c);         // c  
    Mutex_unlock(&m);        // d  
}
```

Parent: w	x	y			z
Child:			a	b	c

Use mutex to ensure no race between interacting with state and wait/signal

# 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);
    }
}
```

1. Keep state in addition to CV's
2. Always do wait/signal with lock held
3. Whenever thread wakes from waiting, recheck state

# 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

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

# PRODUCER/CONSUMER: MULTIPLE THREADS

Producer #3

```
sem_wait(&emptyBuffer);  
sem_wait(&mutex);  
myi = findempty(&buffer);  
sem_post(&mutex);  
Fill(&buffer[myi]);  
sem_post(&fullBuffer);
```

Consumer #3

```
sem_wait(&fullBuffer);  
sem_wait(&mutex);  
myj = findfull(&buffer);  
sem_post(&mutex);  
Use(&buffer[myj]);  
sem_post(&emptyBuffer);
```

Works and increases concurrency; only finding a buffer is protected by mutex;  
Filling or Using different buffers can proceed concurrently

# 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 }
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 }
29 rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
31 rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

# 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  $\leq 0$ , Decrement

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

Zemaphores

Locks

CV's

# PRACTICE QUESTIONS



```

Acquire_readlock() {
    Sem_wait(&mutex); // AR1
    If (ActiveWriters + // AR2
        WaitingWriters==0) { // AR3
        sem_post(OKToRead); // AR4
        ActiveReaders++; // AR5
    } else WaitingReaders++; // AR6
    Sem_post(&mutex); // AR7
    Sem_wait(OKToRead); // AR8
}

Release_readlock() {
    Sem_wait(&mutex); // RR1
    ActiveReaders--; // RR2
    If (ActiveReaders==0 && // RR3
        WaitingWriters > 0) { // RR4
        ActiveWriters++; // RR5
        WaitingWriters--; // RR6
        Sem_post(OKToWrite); // RR7
    }
    Sem_post(&mutex); // RR8
}

```

```

Acquire_writelock() {
    Sem_wait(&mutex); // AW1
    If (ActiveWriters + ActiveReaders +
        WaitingWriters==0) { // AW2
        ActiveWriters++; // AW3
        sem_post(OKToWrite); // AW4
    } else WaitingWriters++; // AW5
    Sem_post(&mutex); // AW6
    Sem_wait(OKToWrite); // AW7
}

Release_writelock() {
    Sem_wait(&mutex); // RW1
    ActiveWriters--; // RW2
    If (WaitingWriters > 0) { // RW3
        ActiveWriters++; // RW4
        WaitingWriters--; // RW5
        Sem_post(OKToWrite); // RW6
    } else while(WaitingReaders>0) { // RW7
        ActiveReaders++; // RW8
        WaitingReaders--; // RW9
        sem_post(OKToRead); // RW10
    }
    Sem_post(&mutex); // RW11
}

```

More such questions?

Little Book of Semaphores - Free online book!

<https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf>

# CONCURRENCY SUMMARY

Motivation: Parallel programming patterns, multi-core machines

Abstractions, Mechanisms

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

Concurrency Bugs

# LOOKING AHEAD

Midterm 2!

New module on Persistence