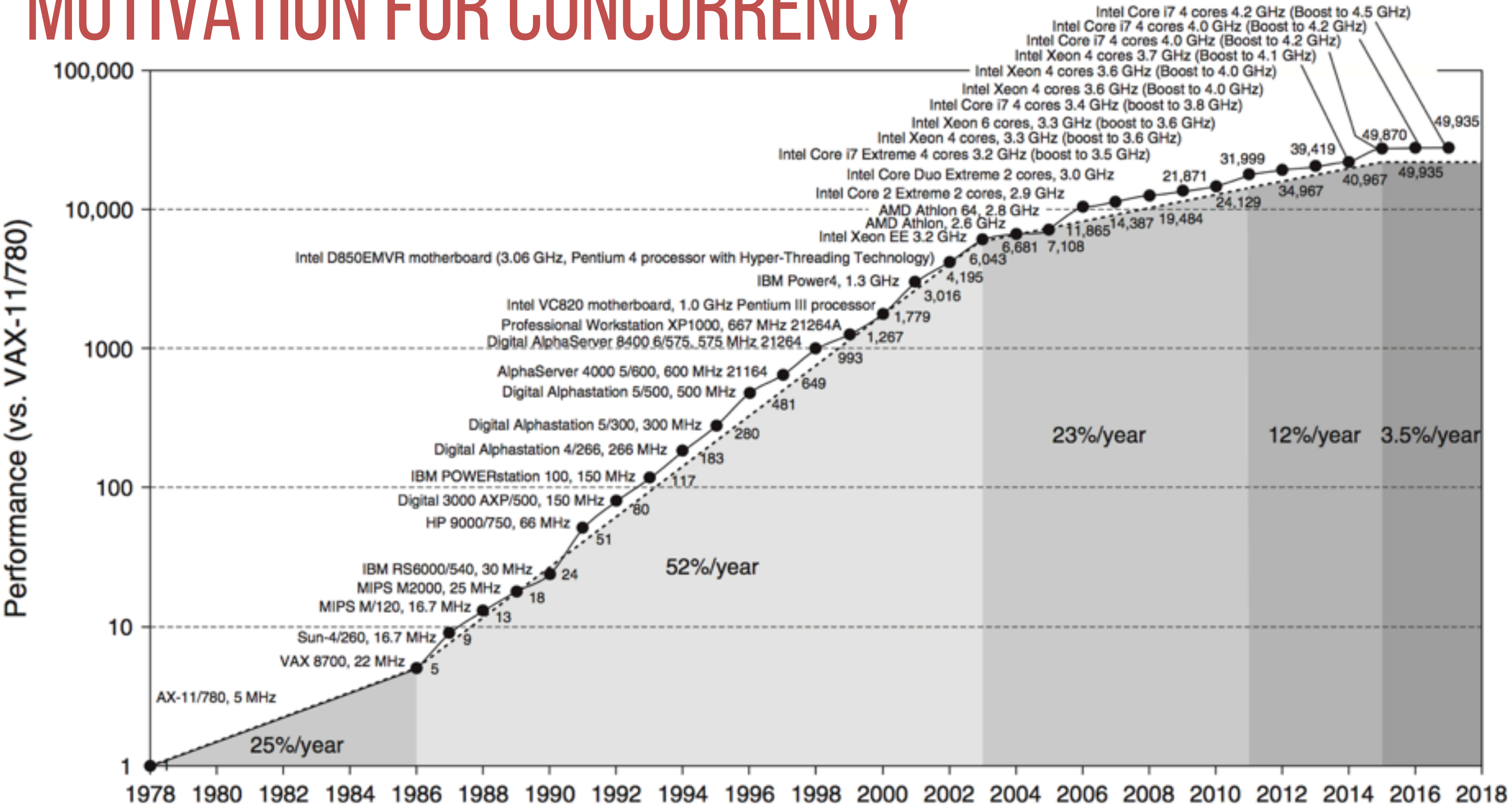
CONCURRENCY: INTRODUCTION

Shivaram Venkataraman

CS 537, Spring 2020

CONCURRENCY

MOTIVATION FOR CONCURRENCY



MOTIVATION

CPU Trend: Same speed, but multiple cores

Goal: Write applications that fully utilize many cores

Option 1: Build apps from many communicating **processes**

- Example: Chrome (process per tab)
- Communicate via pipe() or similar

Pros?

- Don't need new abstractions; good for security

Cons?

- Cumbersome programming
- High communication overheads
- Expensive context switching (why expensive?)

CONCURRENCY: OPTION 2

New abstraction: thread

Threads are like processes, except:

multiple threads of same process share an address space

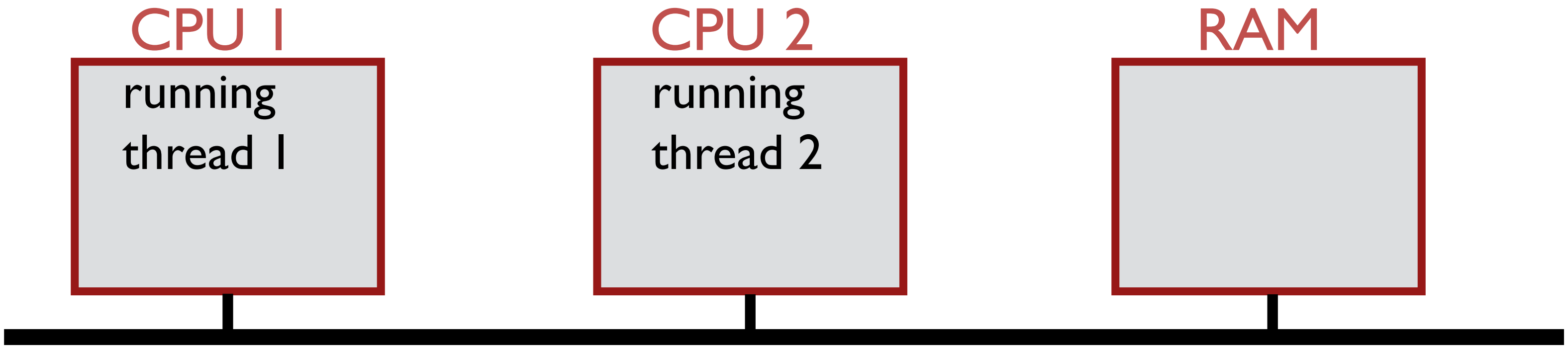
Divide large task across several cooperative threads

Communicate through shared address space

COMMON PROGRAMMING MODELS

Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread**
  One thread performs non-critical work in the background (when CPU idle)



What state do threads share?

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)

OS SUPPORT: APPROACH 1

User-level threads: Many-to-one thread mapping

- Implemented by user-level runtime libraries

  Create, schedule, synchronize threads at user-level

- OS is not aware of user-level threads

  OS thinks each process contains only a single thread of control

Advantages

- Does not require OS support; Portable
- Lower overhead thread operations since no system call

Disadvantages?

- Cannot leverage multiprocessors
- Entire process blocks when one thread blocks

OS SUPPORT: APPROACH 2

Kernel-level threads: One-to-one thread mapping

- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages

- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages

- Higher overhead for thread operations
- OS must scale well with increasing number of threads

THREAD SCHEDULE

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

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
      balance++;
    }
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

```
» ./threads 100000
Initial value : 0
Final value   : 162901
```

TIMELINE VIEW

Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

Thread 1	Thread 2
mov 0x123, %eax	
add %0x1, %eax	
	mov 0x123, %eax
mov %eax, 0x123	
	add %0x2, %eax
	mov %eax, 0x123

Thread 1	Thread 2
	mov 0x123, %eax
mov 0x123, %eax	
	add %0x2, %eax
add %0x1, %eax	
	mov %eax, 0x123
mov %eax, 0x123	

Thread 1	Thread 2
	mov 0x123, %eax
	add %0x2, %eax
	mov %eax, 0x123
mov 0x123, %eax	
add %0x1, %eax	
mov %eax, 0x123	

NON-DETERMINISM

Concurrency leads to non-deterministic results

- Different results even with same inputs
- race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!

WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group

That is, we want them to be atomic

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

More general: Need mutual exclusion for critical sections

    if thread A is in critical section C, thread B isn't
    (okay if other threads do unrelated work)

SYNCHRONIZATION

Build higher-level synchronization primitives in OS

Operations that ensure correct ordering of instructions across threads

Use help from hardware

Motivation: Build them once and get them right

Monitors

Locks

Semaphores

Condition Variables

Loads

Test&Set

Stores

Disable Interrupts

LOCKS

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);

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion*

  Only one thread in critical section at a time
- *Progress* (deadlock-free)

  If several simultaneous requests, must allow one to proceed
- *Bounded* (starvation-free)

  Must eventually allow each waiting thread to enter

Fairness: Each thread waits for same amount of time

Performance: CPU is not used unnecessarily

IMPLEMENTING SYNCHRONIZATION

Atomic operation: No other instructions can be interleaved

Approaches

- Disable interrupts
- Locks using loads/stores
- Using special hardware instructions

IMPLEMENTING LOCKS: W/ INTERRUPTS

Turn off interrupts for critical sections

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
void acquire(lockT *l) {
    disableInterrupts();
}
```

```
void release(lockT *l) {
    enableInterrupts();
}
```

Disadvantages?

Only works on uniprocessors

Process can keep control of CPU for arbitrary length

Cannot perform other necessary work

IMPLEMENTING LOCKS: W/ LOAD+STORE

Code uses a single **shared** lock variable

```
// shared variable
boolean lock = false;
void acquire(Boolean *lock) {
    while (*lock) /* wait */ ;
    *lock = true;
}
```

```
void release(Boolean *lock) {
    *lock = false;
}
```

Does this work? What situation can cause this to not work?

RACE CONDITION WITH LOAD AND STORE

```
*lock == 0 initially

Thread 1                    Thread 2
while(*lock == 1)
                               while(*lock == 1)
                               *lock = 1
*lock = 1
```

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic

XCHG: ATOMIC EXCHANGE OR TEST-AND-SET

How do we solve this ? Get help from the hardware!

```
// xchg(int *addr, int newval)
// 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;
}
```

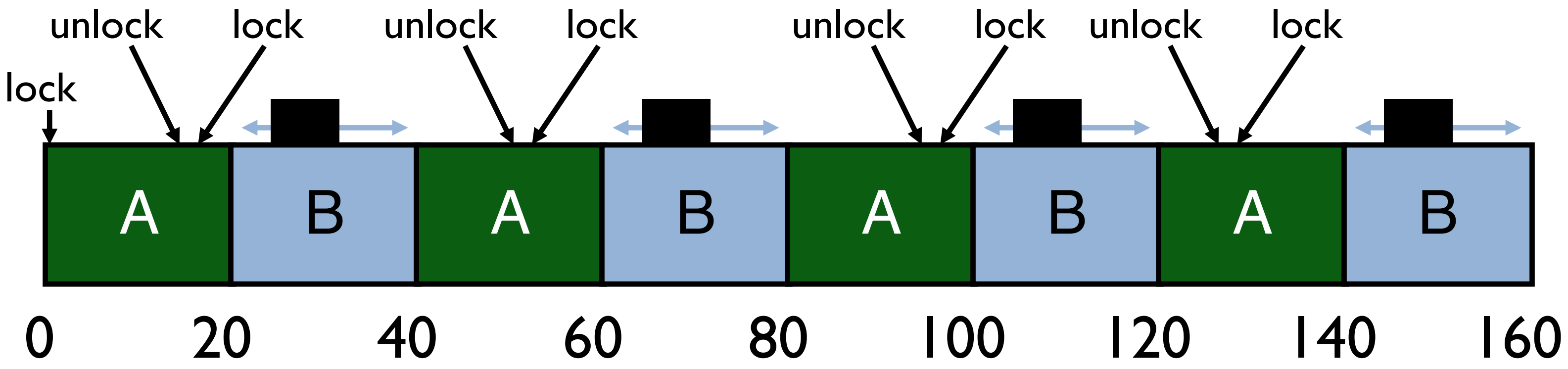
```
movl 4(%esp), %edx
movl 8(%esp), %eax
xchgl (%edx), %eax
ret
```

LOCK IMPLEMENTATION WITH XCHG

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    lock->flag = ??;
}

void acquire(lock_t *lock) {
    ????;
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = ??;
}
```

```
int xchg(int *addr, int newval)
```

BASIC SPINLOCKS ARE UNFAIR



Scheduler is unaware of locks/unlocks!

FAIRNESS: TICKET LOCKS

Idea: reserve each thread's turn to use a lock.

Each thread spins until their turn.

Use new atomic primitive, fetch-and-add

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Acquire: Grab ticket; Spin while not thread's ticket != turn

Release: Advance to next turn

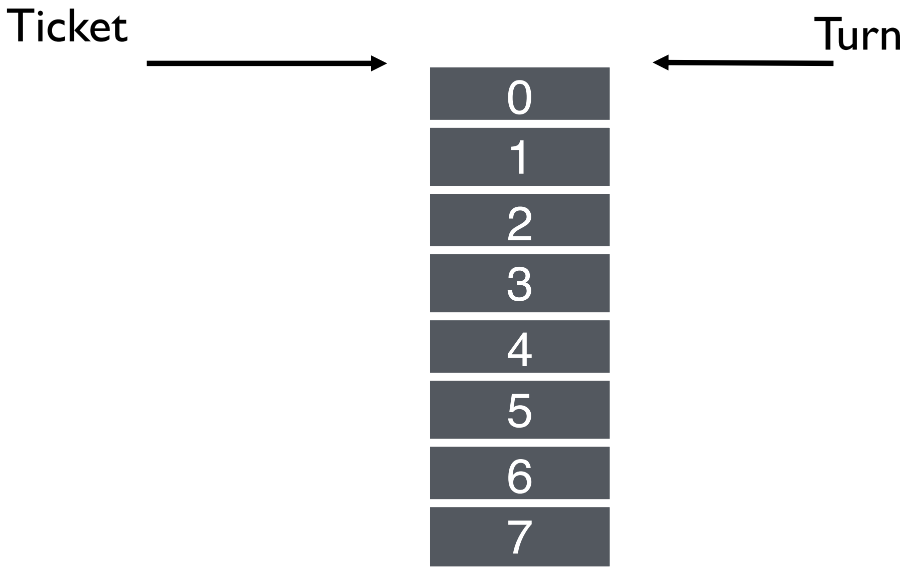
Ticket Lock Example

A lock():
B lock():
C lock():

A unlock():

A lock():
B unlock():

C unlock():
A unlock():



TICKET LOCK IMPLEMENTATION

```
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);
}
```

SPINLOCK PERFORMANCE

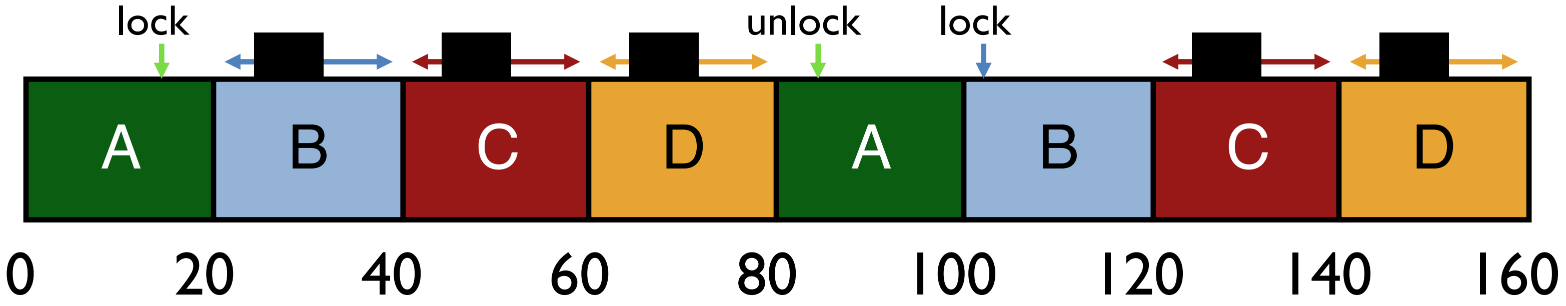
Fast when…

- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when…

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

CPU SCHEDULER IS IGNORANT



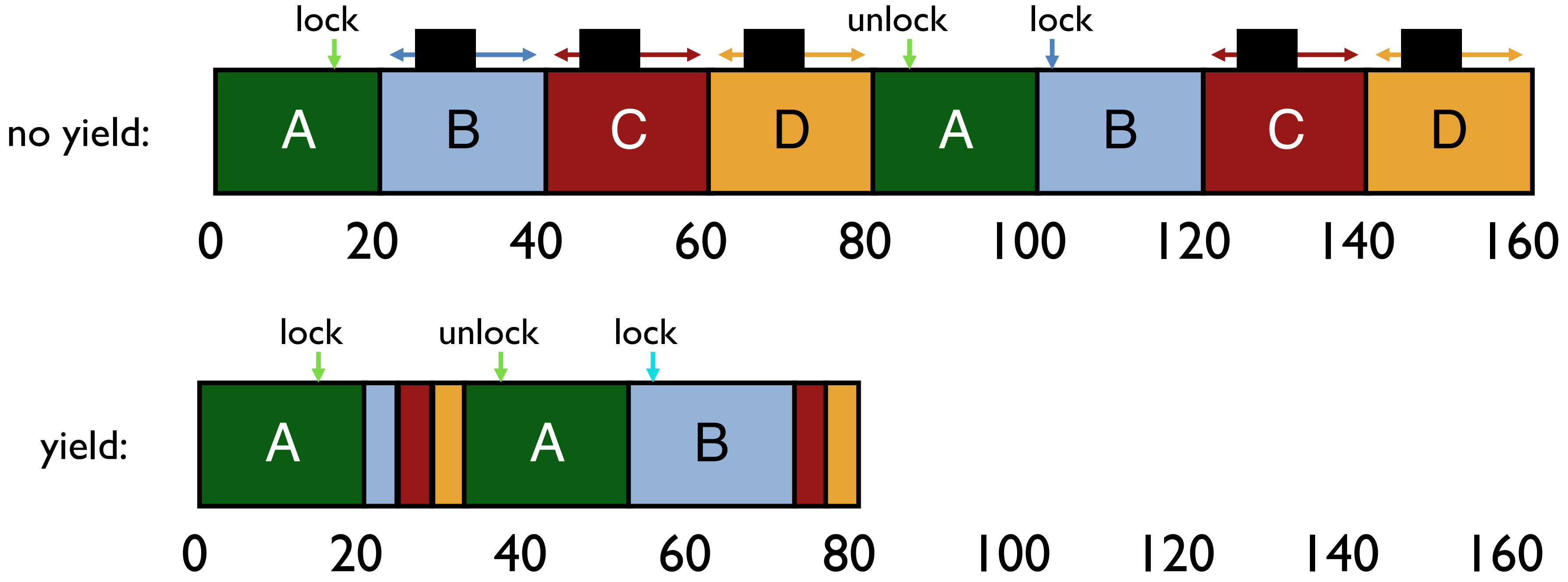
CPU scheduler may run **B, C, D** instead of **A**
even though **B, C, D** are waiting for **A**

TICKET LOCK WITH YIELD

```
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);
    while (lock->turn != myturn)
        yield();
}

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

YIELD INSTEAD OF SPIN



QUIZ 16

https://tinyurl.com/cs537-sp20-quiz16

```
a = 1
int b = xchg(&a, 2)
int c = CAS(&b, 2, 3)
int d = CAS(&b, 1, 3)
```

Final values

Assuming round-robin scheduling, 10ms time slice. Processes A, B, C, D, E, F, G, H in the system

Timeline

A: lock() … compute … unlock()

B: lock() … compute … unlock()

C: lock()

SPINLOCK PERFORMANCE

Waste of CPU cycles?

Without yield: O(threads * **time_slice**)

With yield: O(threads * **context_switch**)

Even with yield, spinning is slow with high thread contention

Next improvement: Block and put thread on waiting queue instead of spinning

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

Remove waiting threads from scheduler runnable queue
(e.g., park() and unpark(threadID))

Scheduler runs any thread that is **runnable**

A B D contend for lock, C is not contending

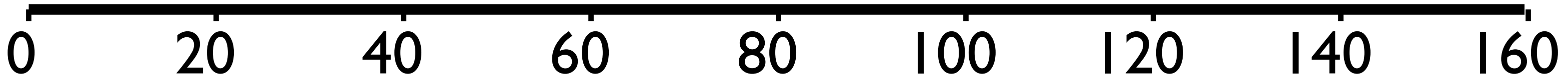
A has 60 ms worth of work

20ms is the timeslice

RUNNABLE: A, B, C, D

RUNNING:

WAITING:



LOCK IMPLEMENTATION: BLOCK WHEN WAITING

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

```
void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        park();     // blocked
    } else {
        l->lock = true;
        l->guard = false;
    }
}

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

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

(a) Why is **guard** used?

(b) Why okay to **spin** on guard?

(c) In release(), why not set lock=false when unpark?

(d) Is there a race condition?

```
void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
            qadd(l->q, tid);
            l->guard = false;
            park();      // blocked
    } else {
            l->lock = true;
            l->guard = false;
    }
}

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

RACE CONDITION

Thread 1 (in lock)

```
if (l->lock) {
    qadd(l->q, tid);
    l->guard = false;

    park();    // block
```

Thread 2 (in unlock)

```
while (TAS(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
```

BLOCK WHEN WAITING: FINAL CORRECT LOCK

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

setpark() fixes race condition

```
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;
}
```

SPIN-WAITING VS BLOCKING

Each approach is better under different circumstances

Uniprocessor

- Waiting process is scheduled → Process holding lock isn't
- Waiting process should always relinquish processor
- Associate queue of waiters with each lock (as in previous implementation)

Multiprocessor

- Waiting process is scheduled → Process holding lock might be
- Spin or block depends on how long, t, before lock is released
  - Lock released quickly → Spin-wait
  - Lock released slowly → Block
  - Quick and slow are relative to context-switch cost, C

WHEN TO SPIN-WAIT? WHEN TO BLOCK?

If know how long**, t,** before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

t

How much wasted when blocking?

What is the best action when t<C?

When t>C?

Problem:

Requires knowledge of future; too much overhead to do any special prediction

TWO-PHASE WAITING

Theory: Bound worst-case performance; ratio of actual/optimal

When does worst-possible performance occur?

Spin for very long time t >> C
Ratio: t/C (unbounded)

Algorithm: Spin-wait for C then block → Factor of 2 of optimal

Two cases:

t < C: optimal spin-waits for t; we spin-wait t too

t > C: optimal blocks immediately (cost of C);

we pay spin C then block (cost of 2 C);

2C / C → 2-competitive algorithm

CONCURRENCY: QUEUE LOCKS, CONDITION VARIABLES

Shivaram Venkataraman

CS 537, Spring 2019

CONDITION VARIABLES

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*

ORDERING EXAMPLE: JOIN

```
pthread_t p1, p2;
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done\n [balance: %d]\n [should: %d]\n",
    balance, max*2);
return 0;
```

how to implement join()?

CONDITION VARIABLES

Condition Variable: queue of waiting threads

B waits for a signal on CV before running

- wait(CV, …)

A sends signal to CV when time for ***B*** to run

- signal(CV, …)

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

JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);      // x
        Cond_wait(&c, &m);   // y
        Mutex_unlock(&m);    // z
}
```

Child:

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

Example schedule:

Parent:	x	y				z
Child:			a	b	c	

JOIN IMPLEMENTATION: ATTEMPT 1

Parent:

```
void thread_join() {
        Mutex_lock(&m);      // x
        Cond_wait(&c, &m);   // y
        Mutex_unlock(&m);    // z
}
```

Child:

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

Example broken schedule:

Parent:				x	y
Child:	a	b	c		

RULE OF THUMB 1

Keep state in addition to CV's!

CV's are used to signal threads when state changes

If state is already as needed, thread doesn't wait for a signal!

JOIN IMPLEMENTATION: ATTEMPT 2

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() {
        done = 1;            // a
        Cond_signal(&c); // b
}
```

Parent:			w	x	y	z
Child:	a	b				

JOIN IMPLEMENTATION: ATTEMPT 2

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() {
        done = 1;          // a
        Cond_signal(&c);// b
}
```

Parent:	w	x			y
Child:			a	b	

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 PROBLEM

EXAMPLE: UNIX PIPES

A pipe may have many writers and readers

Internally, there is a finite-sized buffer

Writers add data to the buffer

- Writers have to wait if buffer is full

Readers remove data from the buffer

- Readers have to wait if buffer is empty

EXAMPLE: UNIX PIPES

start

Buf:

end

EXAMPLE: UNIX PIPES

Implementation:

- reads/writes to buffer require locking
- when buffers are full, writers must wait
- when buffers are empty, readers must wait

PRODUCER/CONSUMER PROBLEM

Producers generate data (like pipe writers)

Consumers grab data and process it (like pipe readers)

Producer/consumer problems are frequent in systems (e.g. web servers)

General strategy use condition variables to:

 make producers wait when buffers are full

 make consumers wait when there is nothing to consume

PRODUCE/CONSUMER EXAMPLE

Start with easy case:

- 1 producer thread
- 1 consumer thread
- 1 shared buffer to fill/consume (max = 1)

Numfull = number of buffers currently filled

numfull

Thread 1 state:

Thread 2 state:

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);
        if(numfull == max)
            Cond_wait(&cond, &m);
        do_fill(i);
        Cond_signal(&cond);
        Mutex_unlock(&m);
    }
}
```

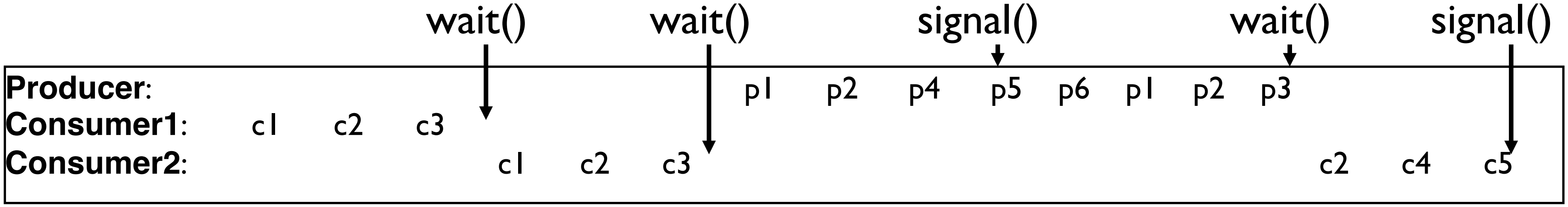
```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);
        if(numfull == 0)
            Cond_wait(&cond, &m);
        int tmp = do_get();
        Cond_signal(&cond);
        Mutex_unlock(&m);
        printf("%d\n", tmp);
    }
}
```

WHAT ABOUT 2 CONSUMERS?

Can you find a problematic timeline with 2 consumers (still 1 producer)?

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

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);  // c1
        if(numfull == 0) // c2
            Cond_wait(&cond, &m); // c3
        int tmp = do_get(); // c4
        Cond_signal(&cond); // c5
        Mutex_unlock(&m); // c6
        printf("%d\n", tmp); // c7
    }
}
```



HOW TO WAKE THE RIGHT THREAD?

Wake all the threads!?

Better solution (usually): use two condition variables

PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (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);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

PRODUCER/CONSUMER: TWO CVS

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        Mutex_lock(&m); // p1
        if (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);
        if (numfull == 0)
            Cond_wait(&fill, &m);
        int tmp = do_get();
        Cond_signal(&empty);
        Mutex_unlock(&m);
    }
}
```

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

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()

GOOD RULE OF THUMB 3

Whenever a lock is acquired, recheck assumptions about state!

Another thread could grab lock in between signal and wakeup from wait

Note that some libraries also have "spurious wakeups" (may wake multiple waiting threads at signal or at any time)

SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV's

2. Always do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state

WAKING ALL WAITING THREADS

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

broadcast(cond_t *cv)

- wake **all** waiting threads (if >= 1 thread is waiting)
- if there are no waiting thread, just return, doing nothing

WHEN TO SPIN-WAIT? WHEN TO BLOCK?

If know how long**, t,** before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

How much wasted when block?

What is the best action when t<C?

When t>C?

CONCURRENCY: SEMAPHORES

Shivaram Venkataraman

CS 537, Spring 2020

AGENDA / LEARNING OUTCOMES

Concurrency abstractions

- How can semaphores help with producer-consumer?
- How to implement semaphores?

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 (with state)*

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

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()

SUMMARY: RULES OF THUMB FOR CVS

1. Keep state in addition to CV's

2. Always do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state

INTRODUCING 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

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

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) {

}

void acquire(lock_t *lock) {

}

void release(lock_t *lock) {

}
```

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

JOIN WITH CV VS SEMAPHORES

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

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

```
sem_t s;
sem_init(&s, ___-);
```

sem_wait(): Decrement, wait if value < 0

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

```
void thread_join() {
        sem_wait(&s);
}
```

```
void thread_exit() {
        sem_post(&s)
}
```

PRODUCER/CONSUMER: SEMAPHORES #1

Single producer thread, single consumer thread

Single shared buffer between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to _________
- fullBuffer: Initialize to ___________

Producer

```
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer);

    sem_post(&fullBuffer);
}
```

Consumer

```
while (1) {
    sem_wait(&fullBuffer);
    Use(&buffer);

    sem_post(&emptyBuffer);
}
```

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

PRODUCER/CONSUMER: MULTIPLE THREADS

Producer

```
while (1) {
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_post(&fullBuffer);
}
```

Consumer

```
while (1) {
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_post(&emptyBuffer);
}
```

Are `my_i` and `my_j` private or shared? Where is mutual exclusion needed???

PRODUCER/CONSUMER: MULTIPLE THREADS

Consider three possible locations for mutual exclusion

Which work??? Which is best???

```
Producer #1
    sem_wait(&mutex);
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_post(&fullBuffer);
    sem_post(&mutex);
```

```
Consumer #1
    sem_wait(&mutex);
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_post(&emptyBuffer);
    sem_post(&mutex);
```

PRODUCER/CONSUMER: MULTIPLE THREADS

```
Producer #2
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    Fill(&buffer[myi]);
    sem_post(&mutex);
    sem_post(&fullBuffer);
```

```
Consumer #2
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    sem_post(&mutex);
    sem_post(&emptyBuffer);
```

Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers

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

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()
T4: acquire_readlock()
T5: acquire_readlock()
T3: release_writelock()
// what happens next?

QUIZ 18

https://tinyurl.com/cs537-sp20-quiz18

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()

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

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

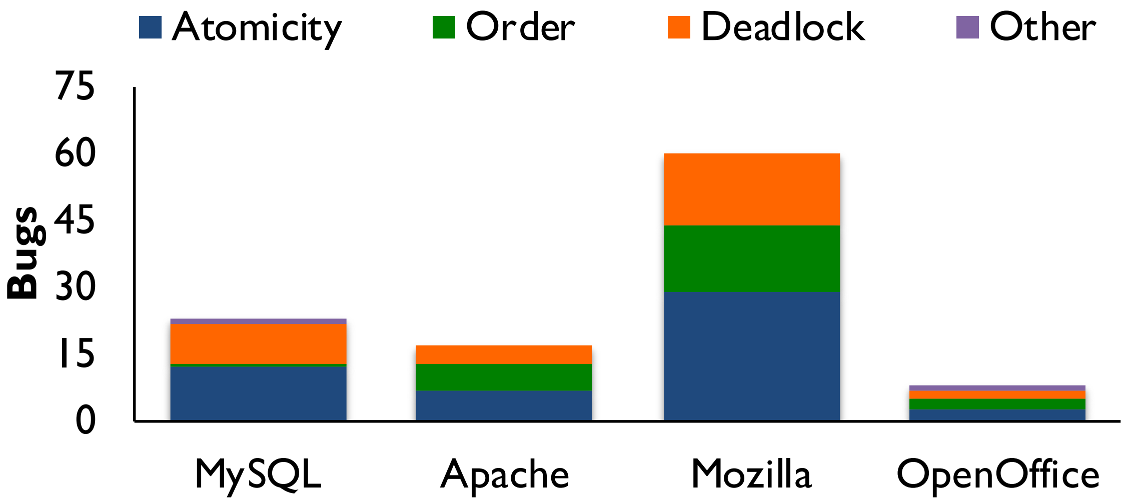
sem_wait(): Decrement and waits if value < 0

sem_post() or sem_signal(): Increment value, then wake a single waiter (atomic)

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

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:

```
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

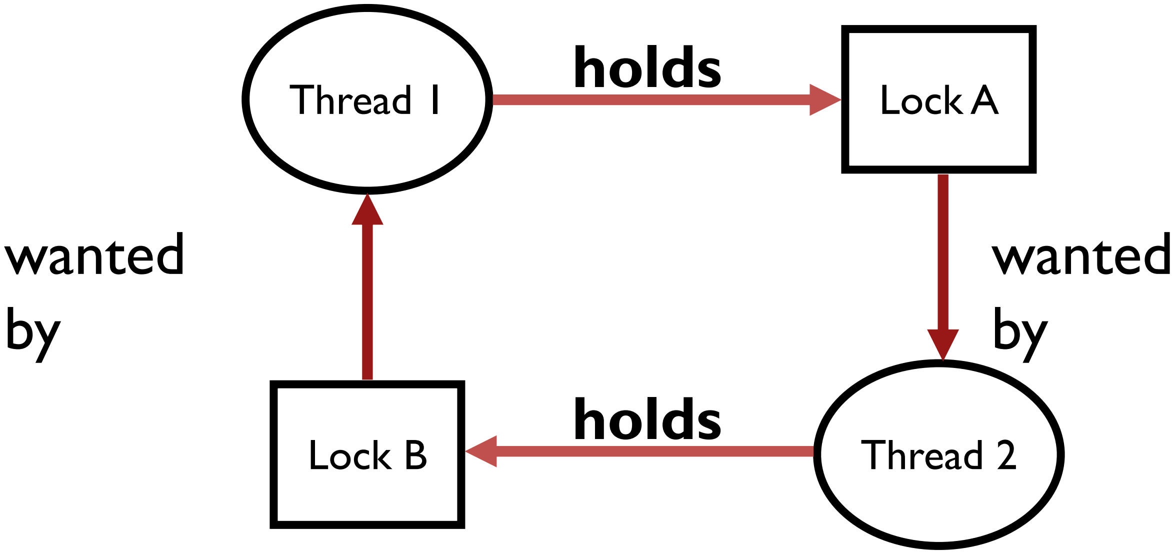
Thread 1:

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

Thread 2:

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

CIRCULAR DEPENDENCY



FIX DEADLOCKED CODE

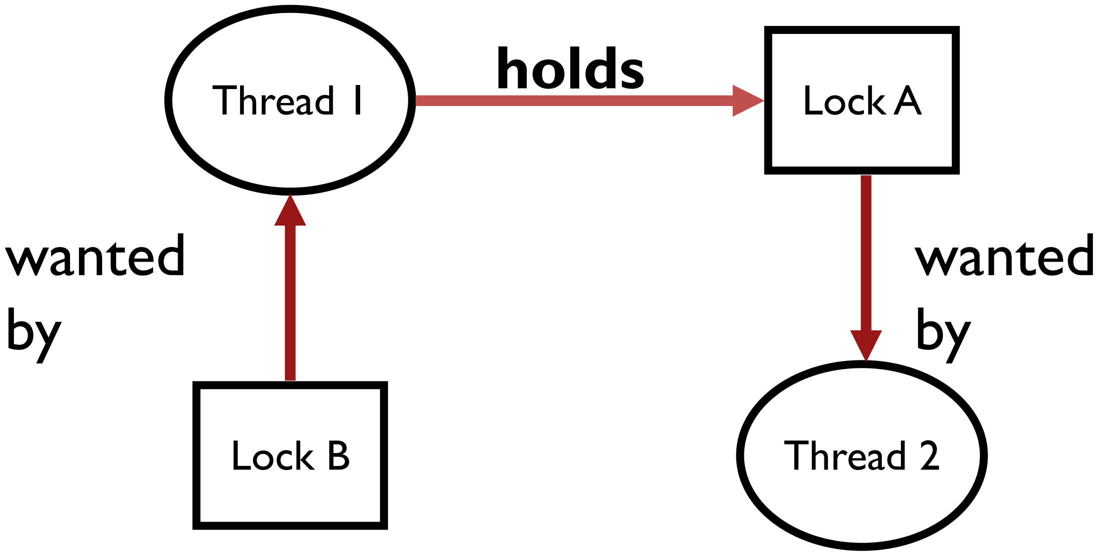
```
Thread 1:

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

```
Thread 2:

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

```
Thread 1
```

```
Thread 2
```

NON-CIRCULAR DEPENDENCY



```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);
}
```

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

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

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?

QUIZ 19

https://tinyurl.com/cs537-sp20-quiz19

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

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:

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

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,
                    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