CONCURRENCY: LOCKS

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

- Midterm 1: Today!
  - Last name on Canvas starts with **A-K**: **Van Vleck B102**
  - Last name on Canvas starts with **L-Z: Ingraham B10**
- Project 2, 3 grading → Progress end of this week
- Code review? → Signed up ?!

AGENDA / LEARNING OUTCOMES

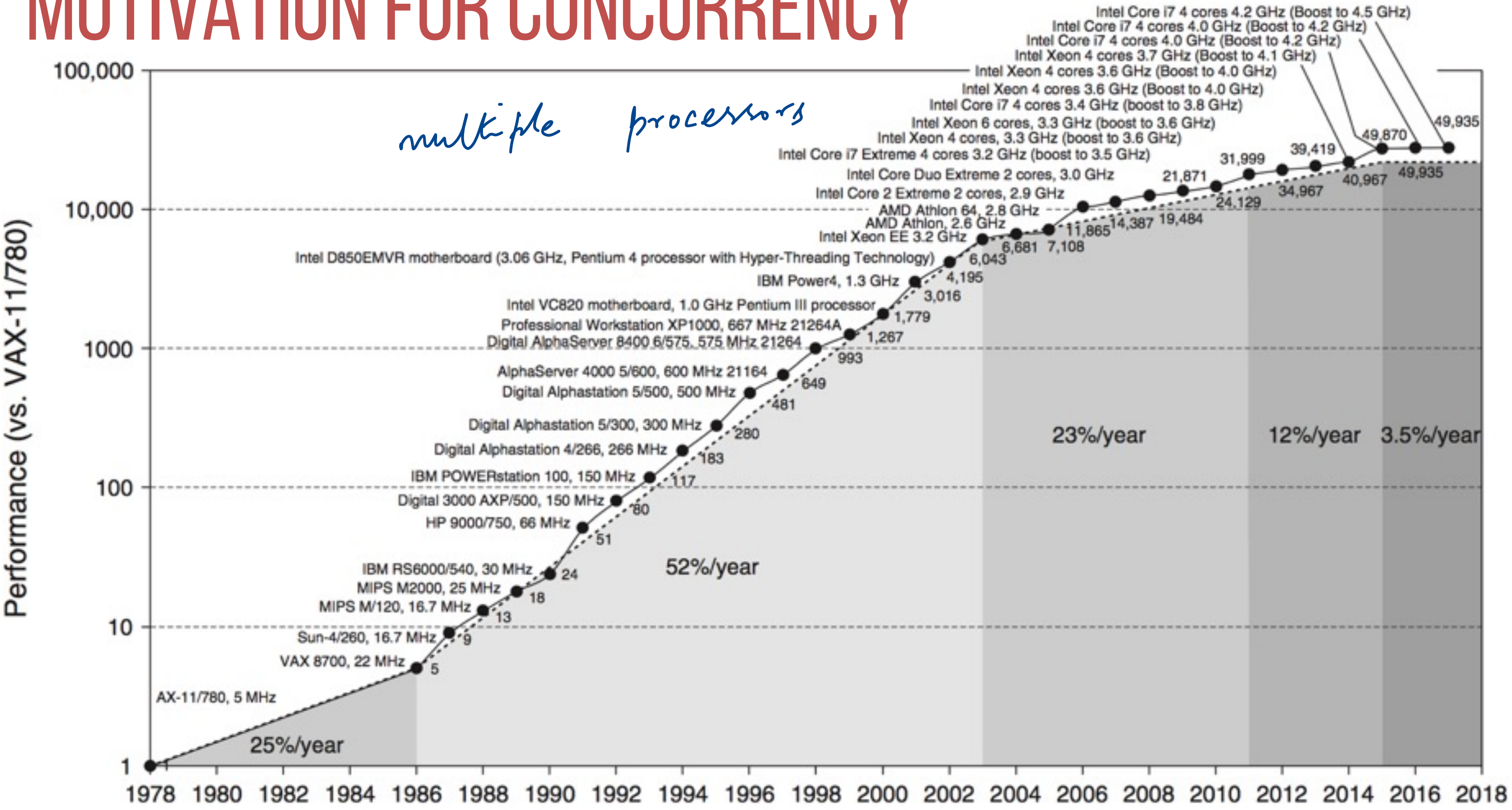
Concurrency

- What are some of the challenges in concurrent execution?
- How do we design locks to address this?
  - ↳ Trade-offs

RECAP

MOTIVATION FOR CONCURRENCY

multiple processors



TIMELINE VIEW

share code and heap

Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

interleaved executions → wrong results possible based on schedule

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
```

→ only 1 thread can be active

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)

LOCKS

block if some other thread has the lock

thread 1

mylock -> acquire();

critical section

mylock -> release();

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

  → if N threads try to acquire, 1 thread

- *Bounded* (starvation-free)

  Must eventually allow each waiting thread to enter

  ↳ not stuck forever

Fairness: Each thread waits for same amount of time

Performance: CPU is not used unnecessarily

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

int variable → 0 unlocked, 1 locked

interleaving

acquire

read → spinning

waiting for lock to be zero

update → acquires the lock

acquire() — disable Interrupts()

release — enable interrupts()

↳ Process could keep running

XCHG: ATOMIC EXCHANGE OR TEST-AND-SET

→ Atomic Instructions

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

return old value → memory → value we want to set

```
// 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;
}
```

i = 0;
→ xchg (&i, 1) = 0
xchg (&i, 1) = 1

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

→ explain this instruction

SPIN LOCK WITH XCHG

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

void init(lock_t *lock) {
    lock->flag = ??;  0
}

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

void release(lock_t *lock) {
    lock->flag = ??;  0
}
```

ensures that two threads cannot acquire at same time

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

```
while ( xchg (&lock->flag, 1) == 1 )
```

↳ if old value was 1 repeat this inst.

OTHER ATOMIC HW INSTRUCTIONS

```
int CompareAndSwap(int *addr, int expected, int new) {
  int actual = *addr;
  if (actual == expected)
    *addr = new;
  return actual;
}
```

if old value matches expected then set addr new value

return old value

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

we didn't get the lock!

expect that nobody has lock

acquire this lock

flag 0 → unlocked

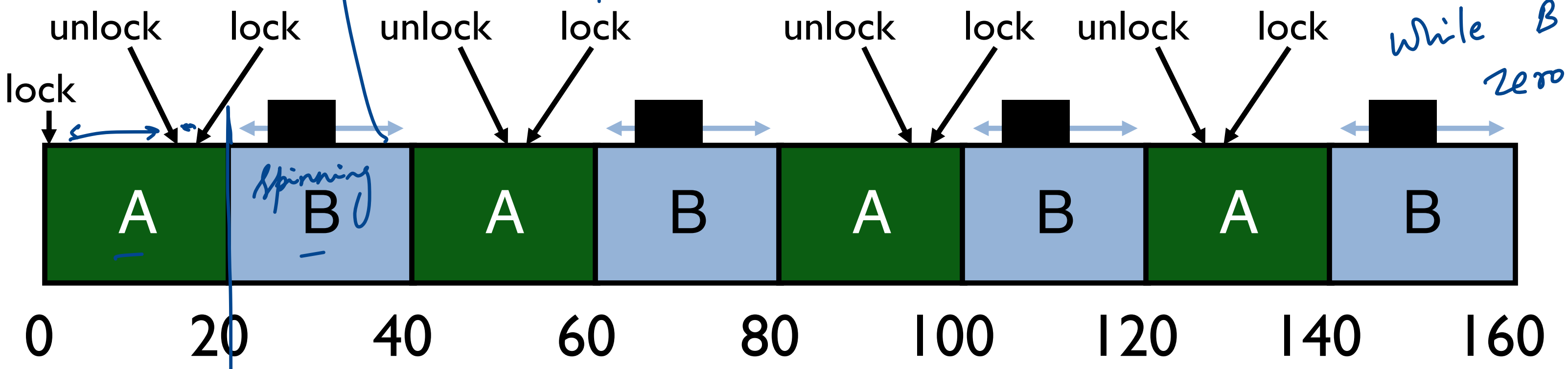
flag 1 → locked

BASIC SPINLOCKS ARE UNFAIR

CPU utilization suffers ← no useful work

Round robin scheduling

A gets to lock three times while B has zero locks



A holds the lock

Scheduler is unaware of locks/unlocks!

FAIRNESS: TICKET LOCKS

Spin lock → flag

FAIRNESS: TICKET LOCKS → ticket, turn

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

Each thread spins until their turn.

Use new atomic primitive, fetch-and-add

ticket → order in which threads will acquire lock

turn → who gets the lock now

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

old value of shared ← int; shared var ← int *ptr; increments ← *ptr = old + 1;

shared — value of i

i = 0

FAA(&i) = 0   1

FAA(&i) = 1   2

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

Release: Advance to next turn

TICKET LOCK EXAMPLE

A lock(): ticket = 0 . A gets lock

B lock(): ticket = 1 . waits

C lock(): ticket = 2

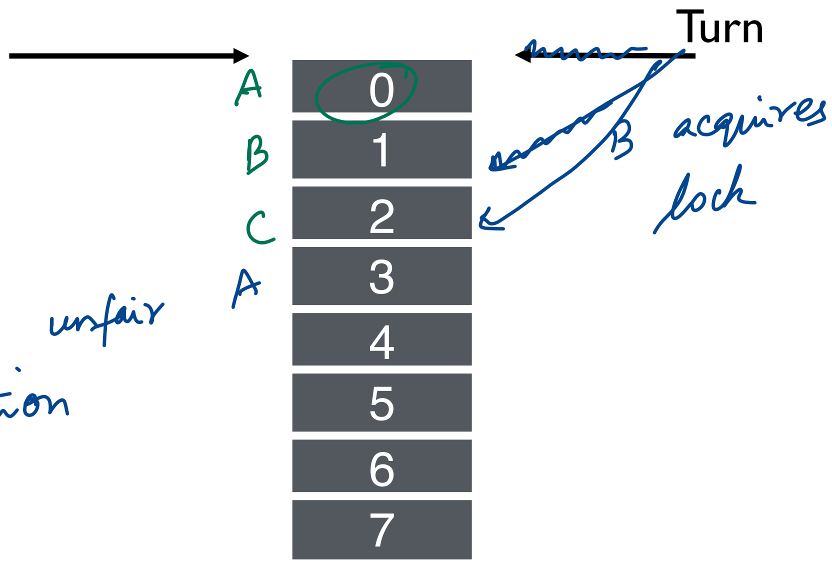
A unlock(): inc turn value

→ A lock(): ticket = 3 ; wait → avoid unfair execution

B unlock(): C gets lock

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);
}
```

increments

old value

wait turn equals your ticket

increment turn value ⇒ next thread can acquire it

SPINLOCK PERFORMANCE

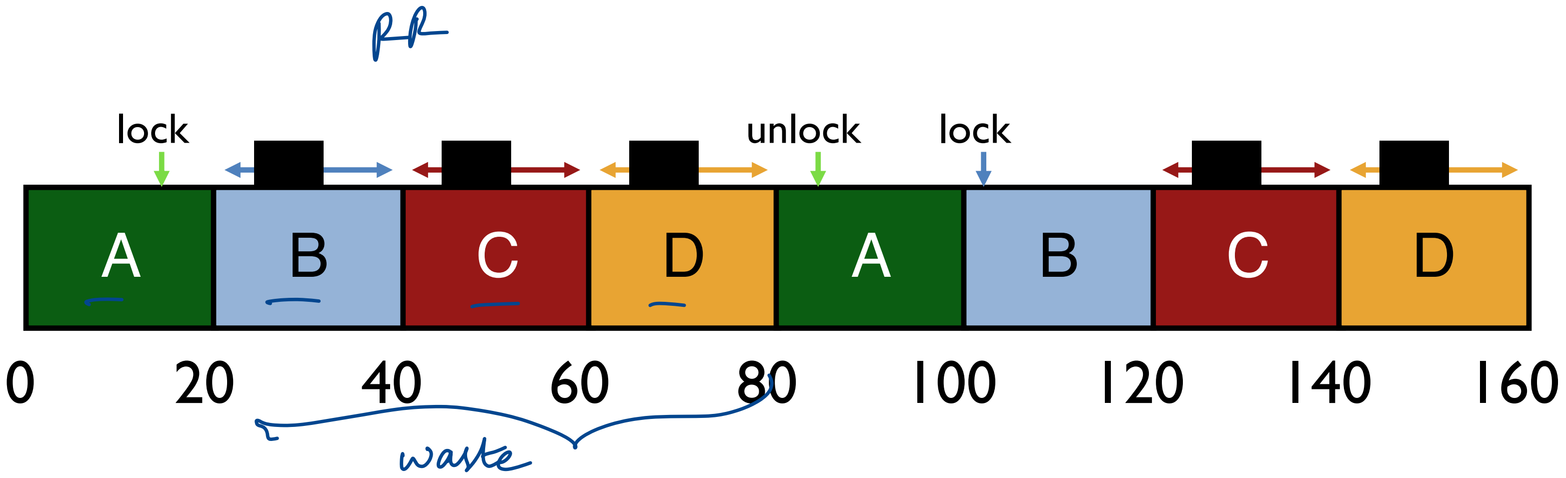
Fast when…

- 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**

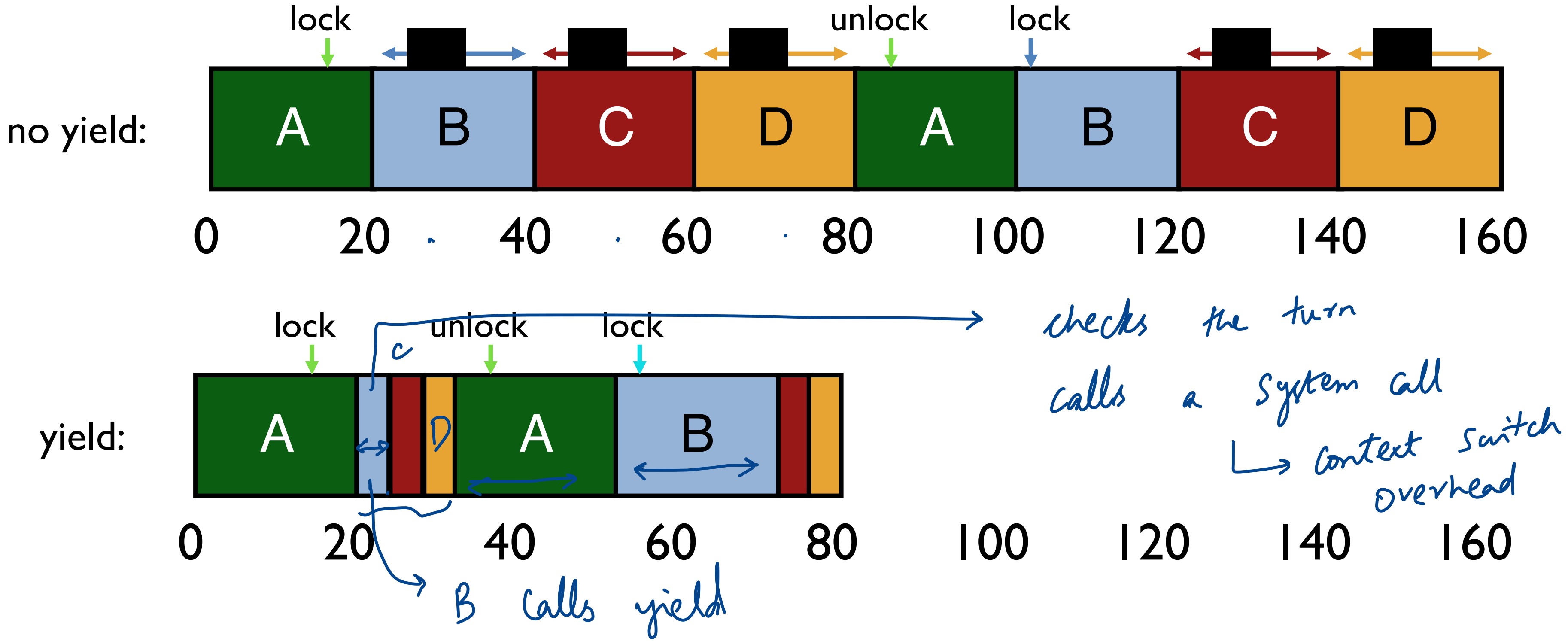
TICKET LOCK WITH YIELD

```
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);
}
```

→ telling the OS I yield my time slice

YIELD INSTEAD OF SPIN



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

need some help from OS

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

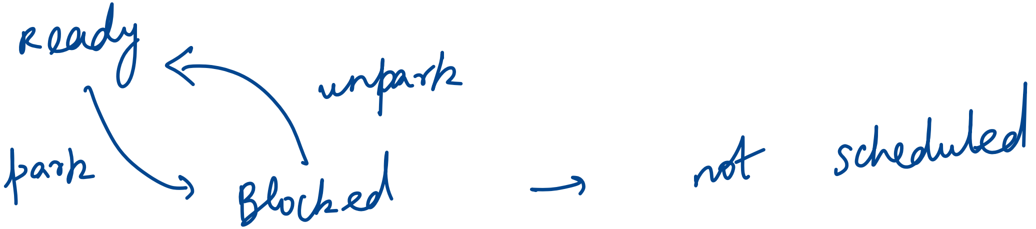
lets one thread wakeup another thread

Remove waiting threads from scheduler runnable queue

(e.g., park() and unpark(threadID)) → Solaris

↳ mark thread is blocked

Scheduler runs any thread that is **runnable**



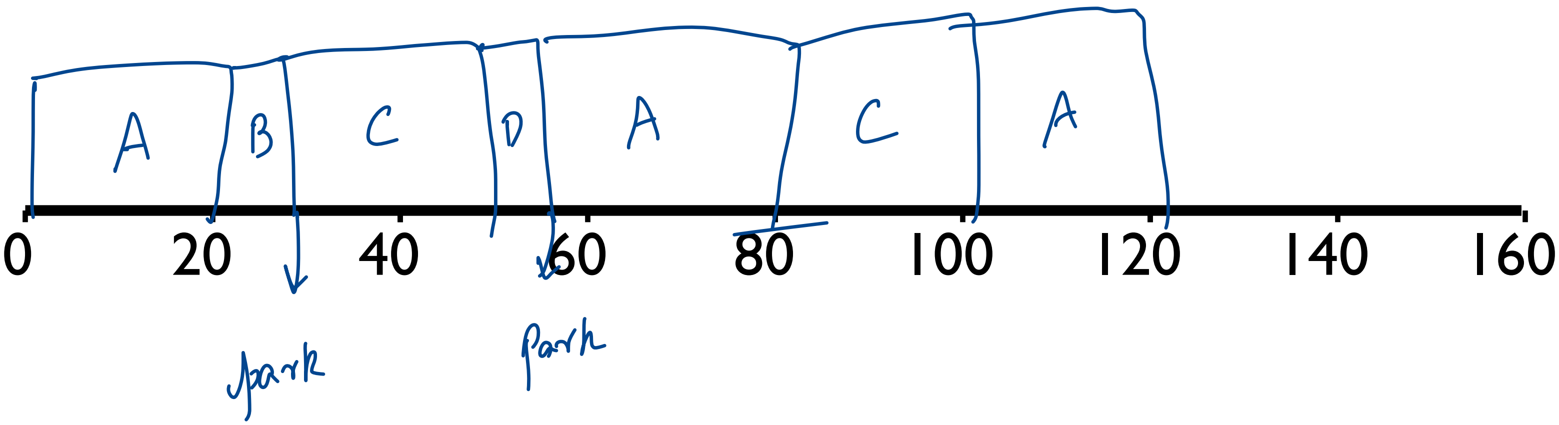
A B D contend for lock, C is not contending

A has 60 ms worth of work
20ms is the timeslice

RUNNABLE: A, B, C, D

RUNNING:

WAITING: B, D



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;
}
```

protects ops to the queue

grab a spin lock

some one holds lock

enqueue

indicate to OS now blocked

acquire

T1 releasing wake up T2

release lock

wakes up T2

head of queue

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

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

↳ Atomically update the queue

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

because critical section is small

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

passing the lock from one thread to another
lock = true

(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)); → exactly 1 thread woken up.
    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 (t << C)
    - Lock released slowly → Block (t >= C)
    - Quick and slow are relative to context-switch cost, C

NEXT STEPS

Midterm1 Today!