CS5460: Operating Systems

Lecture 14: Threads & Locks

(Chapters 26, 27, 28)

Slide Credit: Andrea Arpaci-Dusseau

Assignments

- Assignment 3
  - xv6 Lottery Scheduler
  - Similar to getticks() but many more components
  - Due Thu Mar 18
  - **Note Thu deadline (since the exam is Tue Mar 16)**

- Homework 1
  - Due Mon Mar 15
  - Unlimited attempts; good exam practice

- Midterm
  - Tue Mar 16

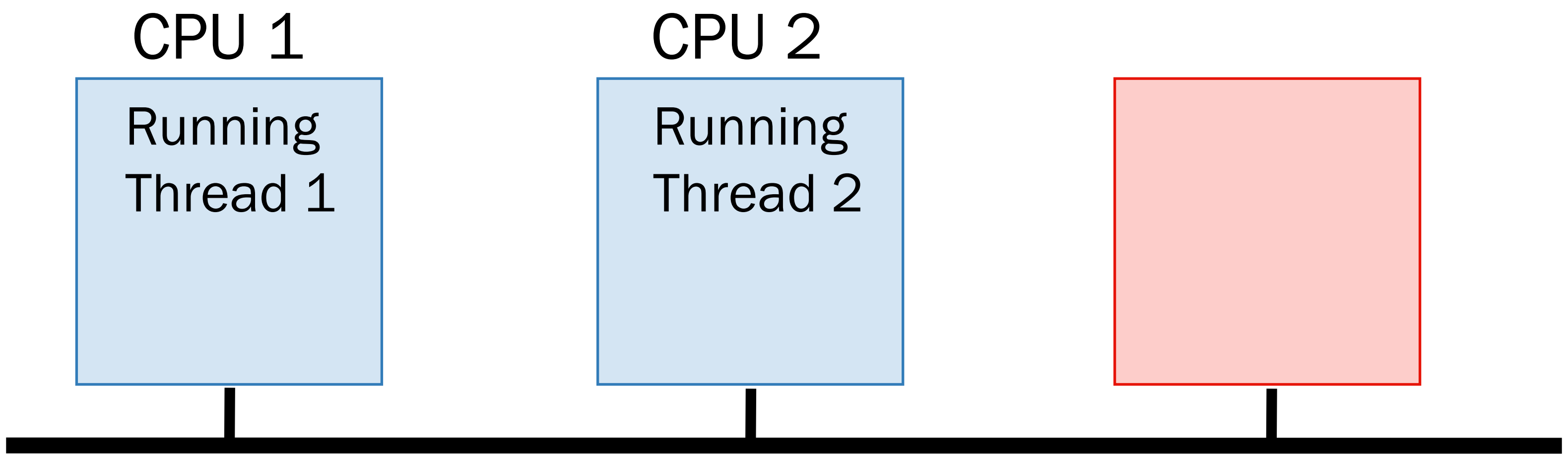
Motivation

- CPU Trend: Same speed, but multiple cores
- Goal: Write applications that fully utilize many cores
- **Option 1**: Use 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

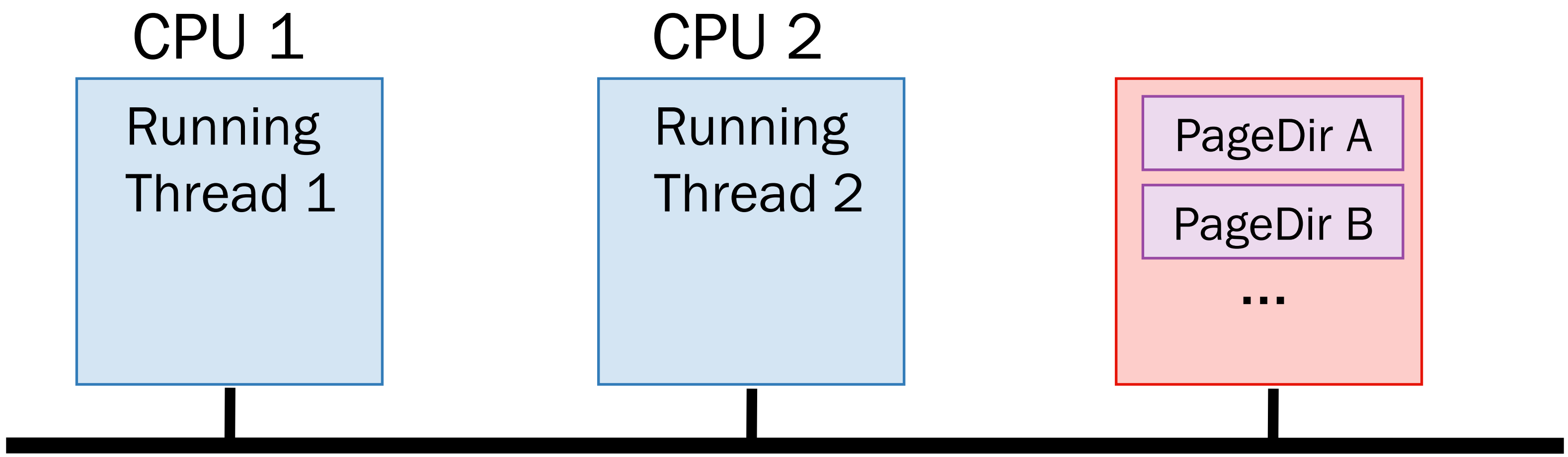
Option 2: Threads

- Threads: virtualize CPU like processes, but threads of same process share address space

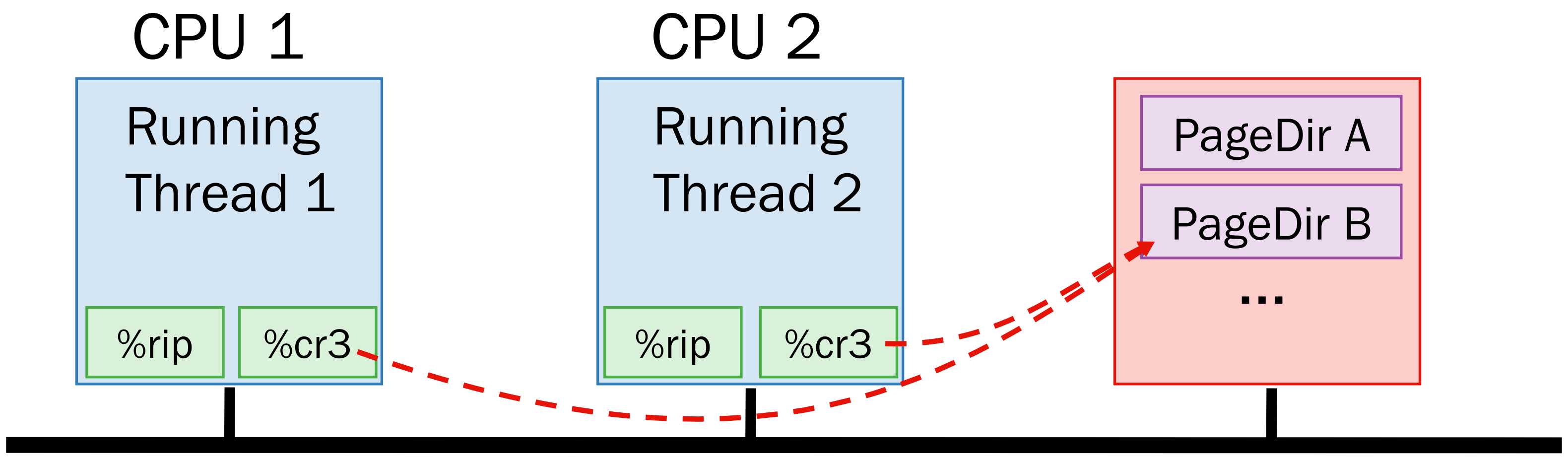
- Divide
  - large task across several cooperative threads
  - many small concurrent tasks across threads
- Communicate through shared address space



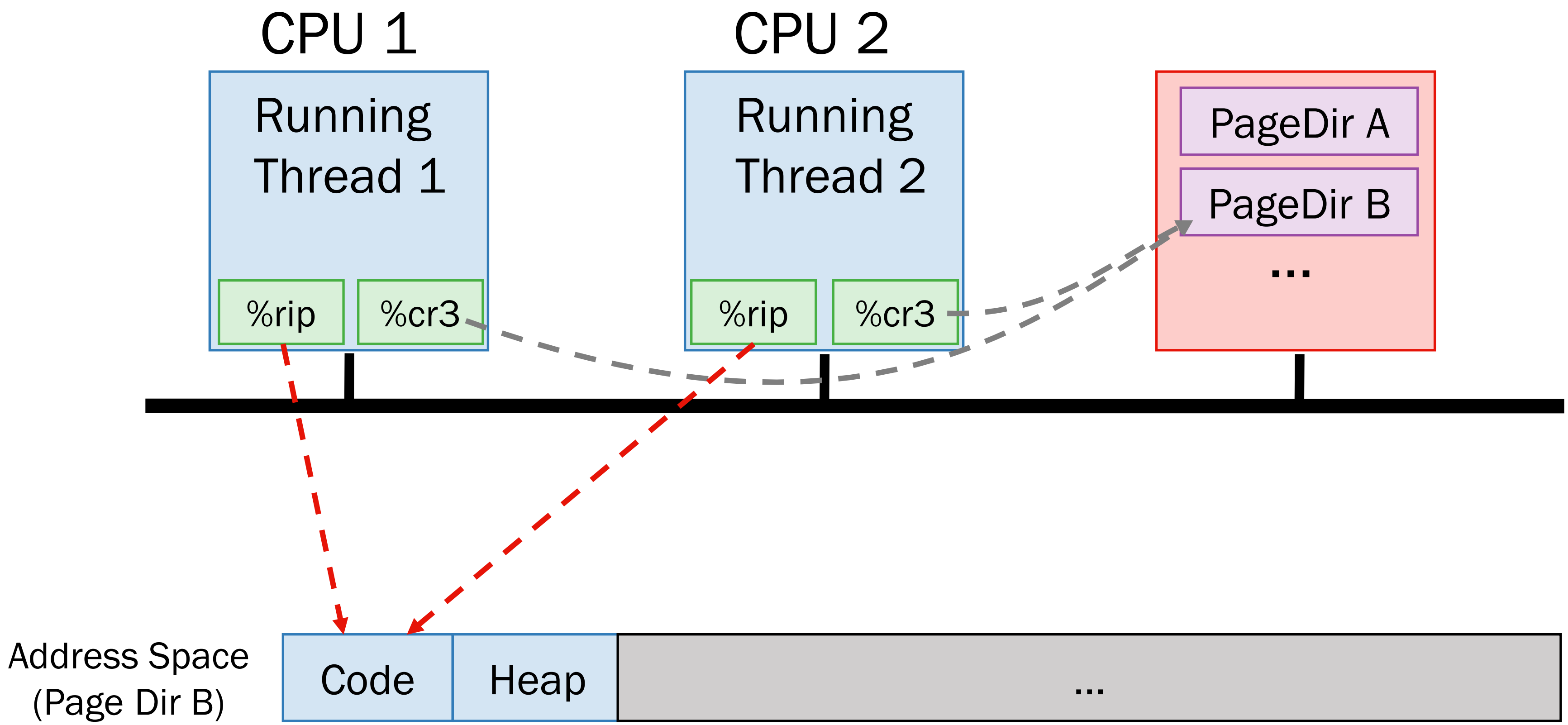
What state do threads share?



What threads share page directories?

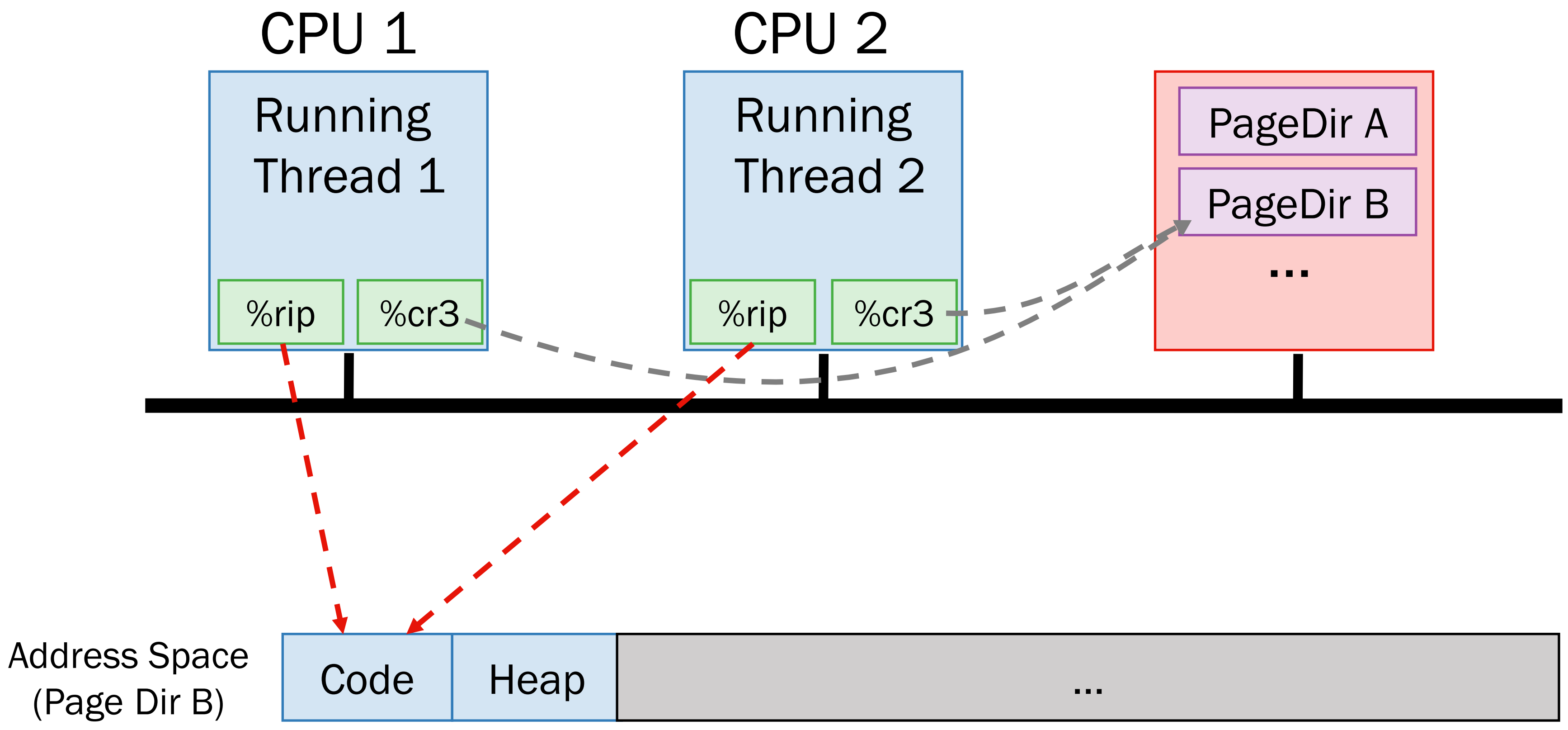


Do threads share Instruction Pointer?

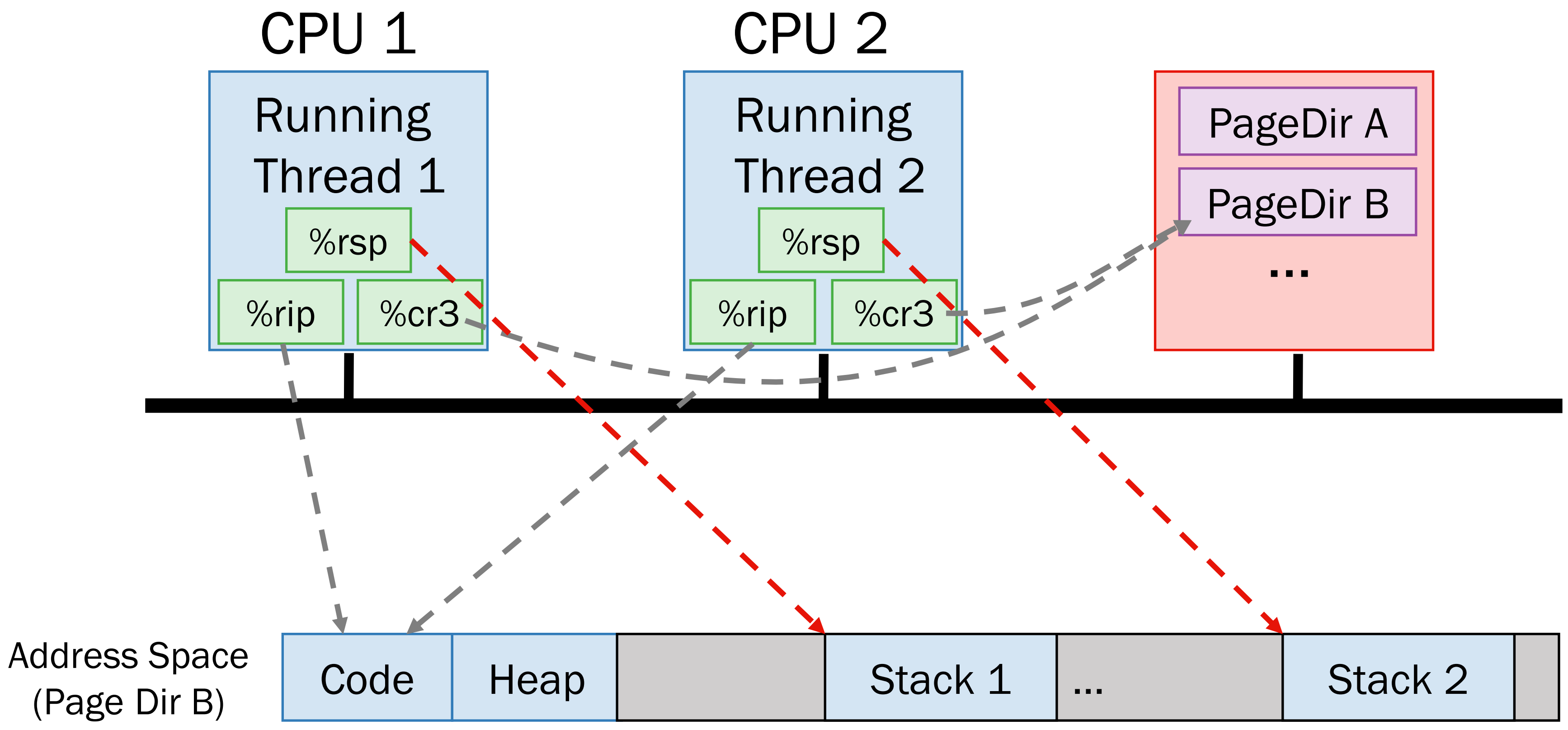


Share code, but each thread may be executing different code at the same time

→ Different Instruction Pointers



Do threads share stack pointer?



Threads executing different functions need different stacks

Threads versus 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)

Can threads access and modify each other's stacks?

Thread API

- Variety of thread systems exist
  - POSIX pthreads
- Common thread operations
  - Create
  - Exit
  - Join (like wait() for processes)

```
int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg);

void pthread_exit(void *retval);

int pthread_join(
        pthread_t thread,
        void **retval);
```

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
  - Kernel is not aware of user-level threads
    - Thinks each process contains only a single thread of control
- Advantages
  - Does not require kernel support; portable
  - Can tune scheduling policy to meet application demands
  - 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 kernel
- 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
  - Kernel must scale well with increasing number of threads

Managing Concurrency

```
int i = 0;

void* run(void* _) {
    for (int j = 0; j < 1000000; j++) i++;
}

void main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, run, NULL);
    pthread_create(&t2, NULL, run, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", i);
}
```

☠️
Please don't write code like this

```
$ ./inc
1041048
$ ./inc
1087180
```

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

Process Control Blocks

	Thread 1	Thread 2
%eax	?	?
%rip	0x195	0x195

T1 →

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

Process Control Blocks

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

```
    0x195  mov 0x9cd4, %eax
T1  0x19a  add $0x1, %eax
    0x19d  mov %eax, 0x9cd4
```

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

Process Control Blocks	Thread 1	Thread 2
	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

```
    0x195  mov 0x9cd4, %eax
    0x19a  add $0x1, %eax
T1  0x19d  mov %eax, 0x9cd4
```

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

	Thread 1	Thread 2
Process Control Blocks	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T1

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

Process Control Blocks

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4

T1

Context Switch

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: ?
%rip = 0x195

Process Control Blocks	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2 →

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x19a

Process Control Blocks

Thread 1	Thread 2
%eax: 101	%eax: ?
%rip: 0x1a2	%rip: 0x195

0x195 mov 0x9cd4, %eax
T2 → 0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 102
%rip = 0x19d

Process Control Blocks

Thread 1	Thread 2
%eax: 101	%eax: ?
%rip: 0x1a2	%rip: 0x195

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
T2 → 0x19d mov %eax, 0x9cd4

Thread Schedule #1

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 102
%eax: 102
%rip = 0x1a2

Process Control Blocks

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

Desired result!

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4

T2

Another schedule

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

Process Control Blocks

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

T1 ➡

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

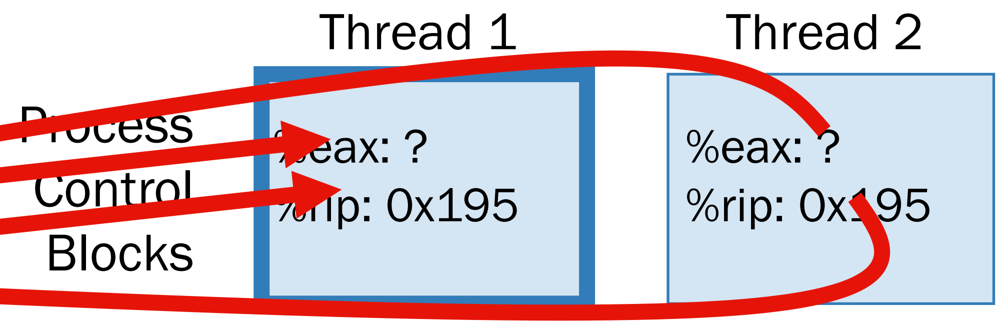
Process Control Blocks	Thread 1	Thread 2
	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

```
     0x195  mov 0x9cd4, %eax
T1 → 0x19a  add $0x1, %eax
     0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d



0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
T1 → 0x19d mov %eax, 0x9cd4

Context Switch

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

Process Control Blocks	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

T2 →

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

Process Control Blocks	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
    0x195  mov 0x9cd4, %eax
T2 → 0x19a  add $0x1, %eax
    0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

Process Control Blocks	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
    0x195  mov 0x9cd4, %eax
    0x19a  add $0x1, %eax
T2  0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

Process Control Blocks

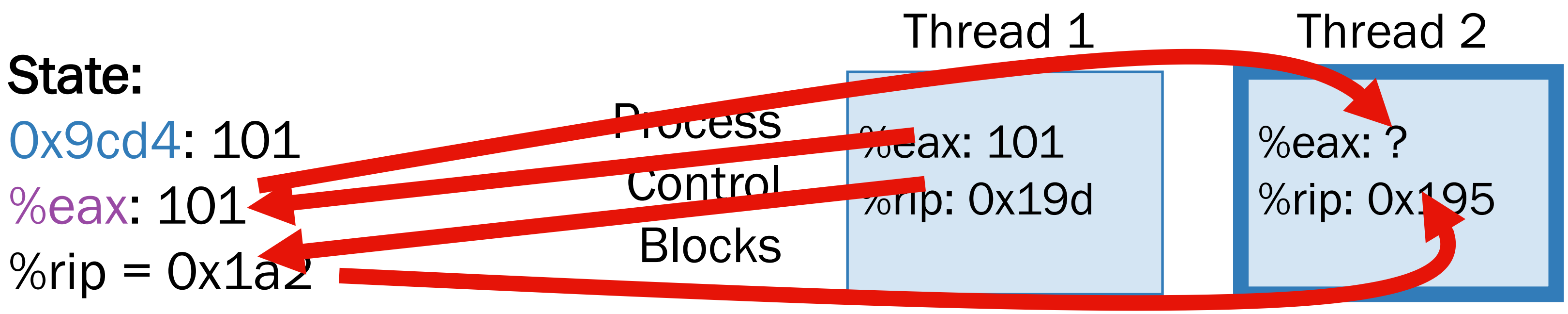
Thread 1	Thread 2
%eax: 101	%eax: ?
%rip: 0x19d	%rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4
```

T2

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

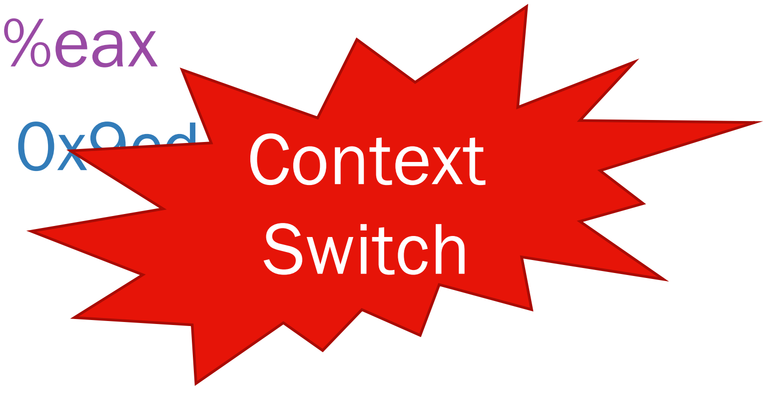


0x195 mov 0x9cd4, %eax

0x19a add $0x1, %eax

0x19d mov %eax, 0x9cd

T2



Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x19d

Process Control Blocks

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

```
    0x195  mov 0x9cd4, %eax
    0x19a  add $0x1, %eax
T1  0x19d  mov %eax, 0x9cd4
```

Thread Schedule #2

balance = balance + 1; balance at 0x9cd4

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

Process Control Blocks	Thread 1	Thread 2
	%eax: 101 %rip: 0x19d	%eax: 101 %rip: 0x1a2

Unexpected result!

0x195 mov 0x9cd4, %eax
0x19a add $0x1, %eax
0x19d mov %eax, 0x9cd4

T1

Timeline View

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

How much is added to shared variable?

Timeline View

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

How much is added to shared variable?

Timeline View

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

How much is added to shared variable?

Timeline View

Thread 1

Thread 2

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

How much is added to shared variable?

Timeline View

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

How much is added to shared variable?

Non-Determinism

- Concurrency leads to non-deterministic results
  - Race condition: non-deterministic result depending on timing of execution; different results even with same inputs
- Whether bug manifests depends on CPU schedule!
- Passing tests means little
- How do we reason about this: imagine scheduler is malicious
  - Assume scheduler will pick bad interleaving at some point…

What do we want?

Want 3 instructions to execute as an uninterruptable group

That is, we want them to appear to be atomic

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

critical section

More generally:
Need mutual exclusion for critical sections

- if process A is in critical section C, process B can't be
  (okay if other processes do unrelated work)

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions across threads

Motivation: Build them once and get them right

Monitors Locks Semaphores

Condition Variables

Loads Stores Test&Set

Disable Interrupts

Threads Conclusions

- Concurrency is needed to obtain high performance by utilizing multiple cores
- Threads are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)
- Context switches within a critical section can lead to non-deterministic bugs (race conditions)
- Use locks to provide mutual exclusion

Locks

Goal: Provide mutual exclusion (mutex)

Three common operations:

- 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; let another process enter critical section
  - **`pthread_mutex_unlock(&mylock);`**

Other Examples

- Consider multi-threaded applications that do more than increment shared balance

- Multi-threaded application with shared linked-list
  - All concurrent:
    - Thread A inserting element a
    - Thread B inserting element b
    - Thread C looking up element c

Shared Linked List

```
void insert(list_t *L, int key) {
  node_t *n = malloc(sizeof(node_t));
  assert(n);
  n->key = key;
  n->next = L->head;
  L->head = n;
}

int lookup(list_t *L, int key) {
  node_t *tmp = L->head;
  while (tmp) {
    if (tmp->key == key)
      return 1;
    tmp = tmp->next;
  }
  return 0;
}
```

```
typedef struct n {
  int key;
  struct n *next;
} node_t;

typedef struct {
  node_t *head;
} list_t;

void init(list_t *L) {
  L->head = NULL;
}
```

Linked-List Race

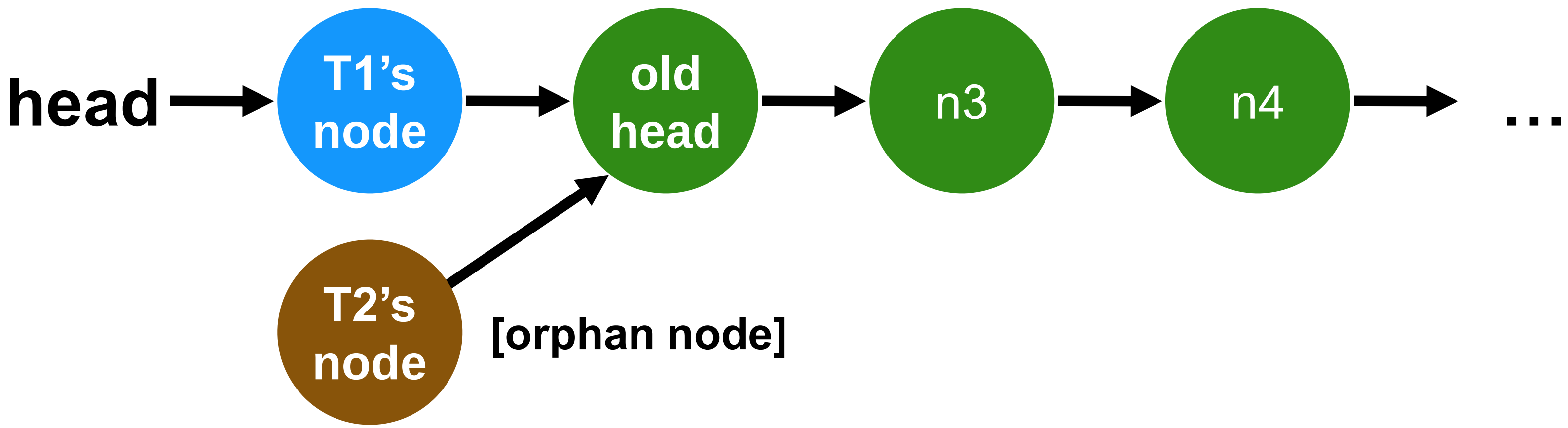
Thread 1	Thread 2
n->key = key	
n->next = L->head	
	n->key = key
	n->next = L->head
	L->head = n
L->head = n	

Both entries point to old head

Only one entry (which one?) can be the new head.

Actually, worse than this: any data race in C results in undefined behavior.

Resulting Linked List



Locking Linked Lists

```
typedef struct {
  int key;
  struct __node_t *next;
} node_t;

typedef struct {
  node_t *head;
  pthread_mutex_t lock;
} list_t;

void init(list_t *L) {
  L->head = NULL;
  pthread_mutex_init(
       &L->lock, NULL);
}
```

Locking: Approach #1

```
void insert(list_t *L, int key) {
  node_t *n = malloc(sizeof(node_t));     pthread_mutex_lock(&L->lock);
  assert(n);
  n->key = key;
  n->next = L->head;
  L->head = n;
}                                         pthread_mutex_unlock(&L->lock);

int lookup(list_t *L, int key) {
  node_t *tmp = L->head;                  pthread_mutex_lock(&L->lock);
  while (tmp) {
    if (tmp->key == key)
      return 1;                           pthread_mutex_unlock(&L->lock);
    tmp = tmp->next;
  }
  return 0;                               pthread_mutex_unlock(&L->lock);
}
```

Locking: Approach #2

```
void insert(list_t *L, int key) {
  node_t *n = malloc(sizeof(node_t));
  assert(n);
  n->key = key;
  n->next = L->head;
  L->head = n;
}

int lookup(list_t *L, int key) {
  node_t *tmp = L->head;
  while (tmp) {
    if (tmp->key == key)
      return 1;
    tmp = tmp->next;
  }
  return 0;
}
```

```
pthread_mutex_lock(&L->lock);
pthread_mutex_unlock(&L->lock);
```

```
pthread_mutex_lock(&L->lock);
pthread_mutex_unlock(&L->lock);
```

This tweak to lookup only works if list has no remove() operation

Data Races

- Race conditions can be because there were concurrent conflicting accesses to a resource or a "data race"

- Data race: when there are two memory accesses in a program where both:
  - target the same location
  - are performed concurrently by two threads
  - are not **both** reads
  - are not synchronization operations (e.g. atomics)
- Data races are always bad news and cause undefined behavior in C
- Data-race-freedom does not imply no race conditions

Reinforcing Terms

- **Race condition**: processes/threads run, and result depends on timing of their execution
- **Data race**: unsynchronized non-read-only accesses to shared data
- **Synchronization**:
  - Using atomic operations to eliminate race conditions
- **Critical section**: code that must run atomically
- **Mutual exclusion**: Ensure at most one thread at a time
- **Lock**: Sync mechanism that enforces atomicity via mutual excl.
  - Lock(L): If L is not currently locked → atomically lock it
If L is currently locked → block until it becomes free
  - Unlock(L): Release control of L
  - Lock "protects" data: Lock(L) before accessing, Unlock(L) when done

Lock Requirements and 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 Waiting (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 (e.g., spinning)

Implementing Synchronization

To implement, need atomic operations

Atomic operation: No other instructions can be interleaved

Examples of atomic operations

- Code between interrupts on uniprocessors
  - Disable timer interrupts, don't do any I/O
- Loads and stores of words
  - Load r1, B
  - Store r1, A
- Special "Atomic" Hardware instructions
  - Compare&Swap(&loc, a, b)
    - Atomically: tmp = *loc; if (*loc == a) { *loc = b; }; return tmp
  - Fetch&Add(&loc, a, b)
    - Atomically: tmp = *loc; (*loc)++; return tmp
  - Test&Set(&loc, a)
    - Atomically: tmp = *loc; *loc = a; return tmp

Implementing Locks: Attempt 1

Turn off interrupts for critical sections

- Prevent dispatcher from running another thread
- Code executes atomically

```
void acquire(lock *l) {
      disableInterrupts();
}
void release(lock *l) {
      enableInterrupts();
}
```

Disadvantages?

Implementing Locks: Attempt 2

Code uses a single shared lock variable

```
atomic_int lock = 0; // shared variable
void acquire() {
      while (atomic_load(&lock)) /* wait */ ;
      atomic_store(&lock, 1);
}

void release() {
      atomic_store(&lock, 0);
}
```

Why doesn’t this work?

Race Condition with Load/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

Compare&Swap

- cas(loc, a, b)
  - Atomically tests if loc contains a; if so, stores b into loc
  - Returns old value from loc

- Acquire()
  - If free, what happens?
  - If locked, what happens?
  - If more than one at a time trying to acquire, what happens?

```
atomic_int locked = 0;

void acquire() {
  while (cas(&locked, 0, 1));
}

void release() {
  atomic_store(&locked, 0);
}
```

Lock Requirements and 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

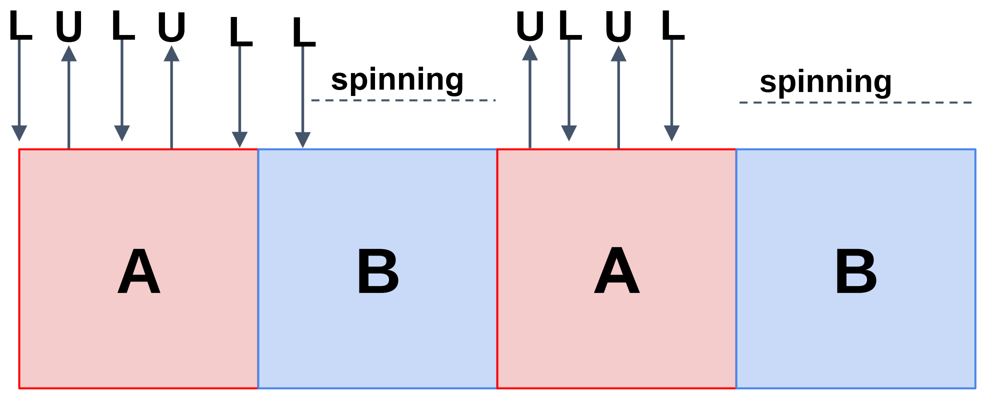
Each thread waits for same amount of time

Performance

CPU is not used unnecessarily (e.g., spinning)

Performance and Unfairness

Scheduler doesn’t know about spin-locks, so it makes bad choices



Performance when busy-waiters > # cores

A general problem with busy waiting

Could even violate bounded waiting property

Fetch&Add Ticket Locks

- faa(loc, val)
  - Atomically reads loc, adds val to it, and writes new value back
- Busy waiting still but first-come-first-served ordering of threads provides fairness

```
atomic_int counter = 0;
atomic_int turn = 0;

void acquire() {
  int me;
  me = faa(&counter, 1);
  while (me != atomic_load(&turn));
}

void release() {
   atomic_store(&turn,
                atomic_load(&turn)+1);
}
```

Lock Requirements and 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 (e.g., spinning)

Fetch&Add Ticket Locks

- faa(loc, val)
  - Atomically reads loc, adds val to it, and writes new value back
- Busy waiting still but first-come-first-served ordering of threads provides fairness
- Try to yield on contention

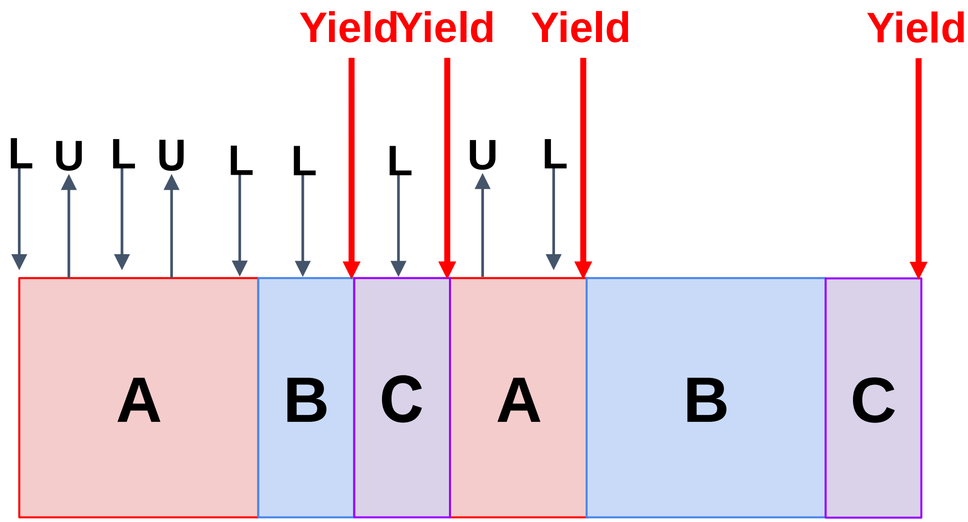
```
atomic_int counter = 0;
atomic_int turn = 0;

void acquire() {
  int me;
  me = faa(&counter, 1);
  while (me != atomic_load(&turn))
    yield();
}

void release() {
   atomic_store(&turn,
                atomic_load(&turn)+1);
}
```

Impact of Yield

Tickets improve **fairness**, yield improves **performance**



Still, wasting resources scheduling processes that may not be able to run anyway.

e.g. A has lock; B, C try to acquire;
RR schedules C multiple times before lock can be acquired

Spinlock Performance

CPU waste…

- Without yield: O(threads * **time_slice**)
- With yield: O(threads * **context_switch**)

So even with yield, spinning is slow with high thread contention

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