Lecture 04: Data Races & Locks

CS343 – Operating Systems

Branden Ghena – Fall 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS61C and CS162

Northwestern

Administrivia

- Getting Started Lab due today
    - ~20 people haven't completed it yet

- Producer-Consumer Lab will be up later today
    - Solve concurrency problems three ways
        - First two are ready after today's lecture
    - Fill out partnership survey if you want to be grouped up
        - I'll be starting to match people on that tonight

    - **Start early!**
        - Getting Started Lab is NOT an accurate representation of how long labs take

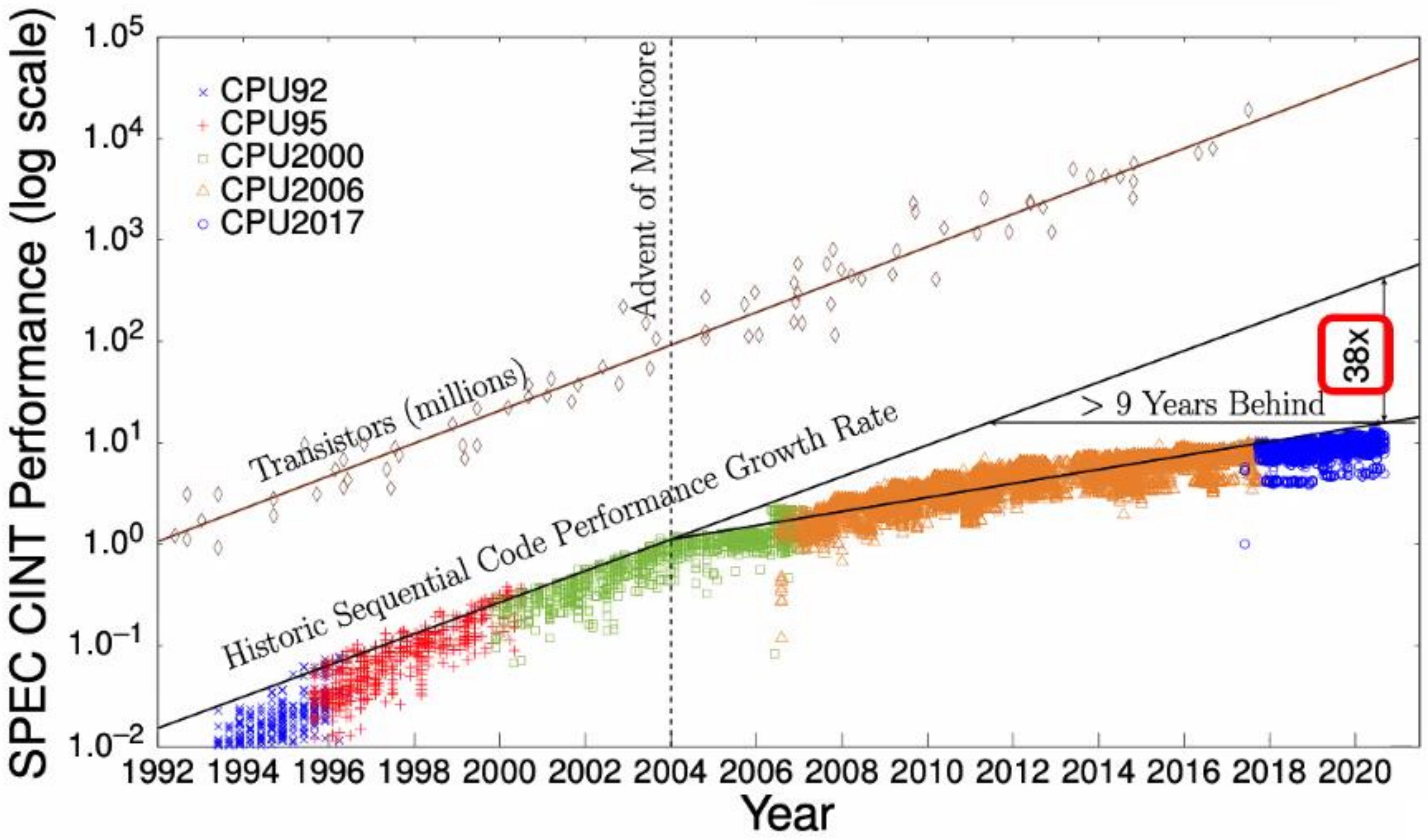
Today’s Goals

- Explore problems with concurrently shared memory.

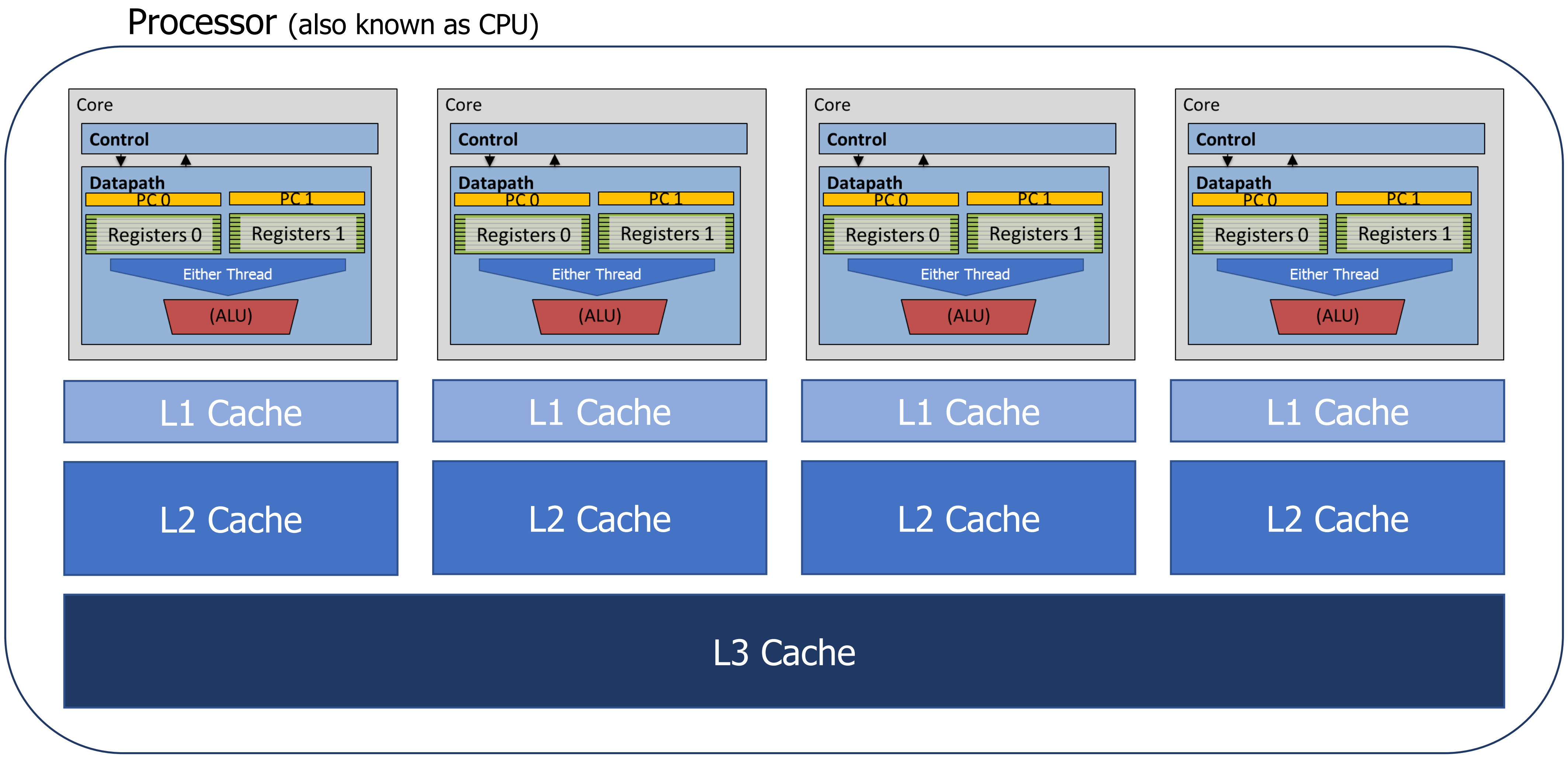
- Introduce locks as a simple solution for correctness.
    - Design of locks
    - Implementation of locks

- Optimize locks to enforce fairness and increase performance.

Performance for sequential code is falling behind



Review: modern hardware capabilities



Outline

- **Race Conditions**
- Critical Sections
- Lock Design
  - Overview
  - Basic Lock Implementation
  - Lock Optimizations

Challenges to concurrency

Concurrency is great! We can do so many things!!

But what’s the downside…?

1. How much speedup can we get from it?
2. **How hard is it to write parallel programs?**

Concurrency problem: data races

Consider two threads with a shared global variable: `int count = 0`

Thread 1:

```
void main(){
  count += 1;
}
```

Thread 2:

```
void main(){
  count += 1;
}
```

`count` could end up with a final value of 1 or 2. How?

Concurrency problem: data races

Consider two threads with a shared global variable: `int count = 0`

Assuming "count" is in memory location 0x8049a1c

Thread 1:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

Thread 2:

```
void thread_fn(){
  mov $0x8049a1c, %edi
  mov (%edi), %eax
  add $0x1, %eax
  mov %eax, (%edi)
}
```

`count` could end up with a final value of 1 or 2. How?

These instructions could be interleaved in any way.

Data race example – Count = 2

Before this code starts

Time

Thread 1	**Thread 2**
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	**Value**
`%eax`	???

Thread 2

Register	**Value**
`%eax`	???

Memory

Variable	**Value**
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 2

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 2

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	2

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 2

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	2

Memory

Variable	Value
`count`	2

Assuming "count" is in memory location pointed to by `%edi`

Theads do not have guaranteed ordering

BUT, there’s no guarantee that the instructions occur in that order!

Since the two threads are running in parallel, the instructions could be interleaved in any way
(both threads are really running simultaneously)

Data race example – Count = 1

Remember, each thread has its own separate registers!

Before this code starts

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	???

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	???

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	0

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 1

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	0

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 1

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	0

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by `%edi`

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race example – Count = 1

Time ↓

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Thread 1

Register	Value
`%eax`	1

Thread 2

Register	Value
`%eax`	1

Memory

Variable	Value
`count`	1

Assuming "count" is in memory location pointed to by **`%edi`**

Data race comparison

Assuming "count" is in memory location pointed to by **`%edi`**

Time

Thread 1	Thread 2
`mov (%edi), %eax`	
`add $0x1, %eax`	
`mov %eax, ($edi)`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`

Final value of count: 2

Thread 1	Thread 2
`mov (%edi), %eax`	
	`mov (%edi), %eax`
	`add $0x1, %eax`
	`mov %eax, (%edi)`
`add $0x1, %eax`	
`mov %eax, (%edi)`	

Final value of count: 1

Data race explanation

- Thread scheduling is **non-deterministic**
  - There is no guarantee that any thread will go first or last or not be interrupted at any point

- If different threads write to the same variable
  - The final value of the variable is also non-deterministic
  - This is a *data race*

Concurrency example: initialization code

```
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  printf("main: begin (counter = %d)\n", counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d, goal was %d)\n",
         counter, 2*LOOPS);
  return 0;
}
```

Start two threads

Wait until done

Concurrency example: threaded code

```
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    counter++;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

Each thread runs this function

Add to global variable

Concurrency example: threaded code

```
#include <stdio.h>
#include <pthread.h>

static volatile int counter = 0;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    counter++;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

`volatile` marks memory that the compiler shouldn't try to optimize

i.e., something tricky is going on here!

Live example – data race

zip with code linked on Canvas

- Compile with “gcc -pthread -o race data_race.c”

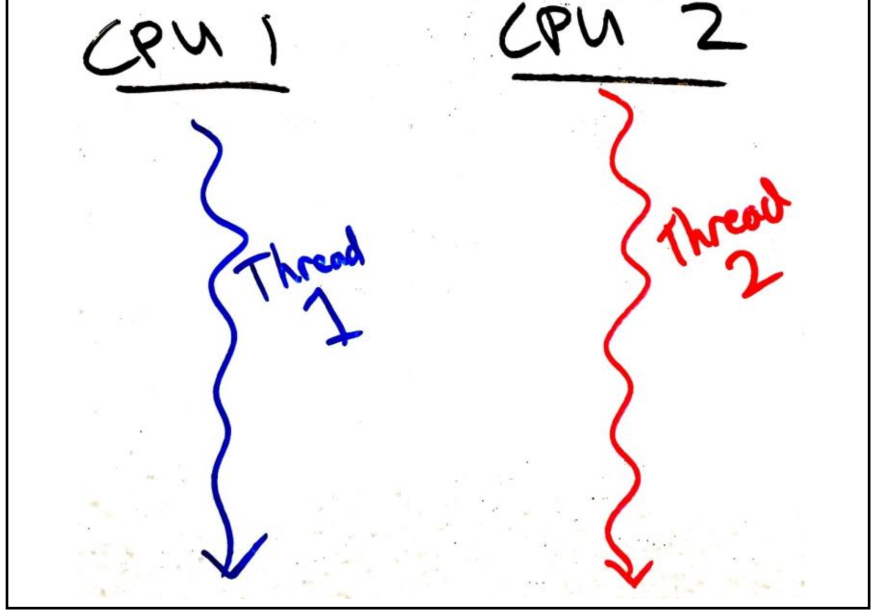
```
[brghena@ubuntu race_condition] $ ./race
main: begin (counter = 0)
B: begin
A: begin
A: done
B: done
main: done with both (counter = 12161815, goal was 20000000)
```

- Different results each time you run it

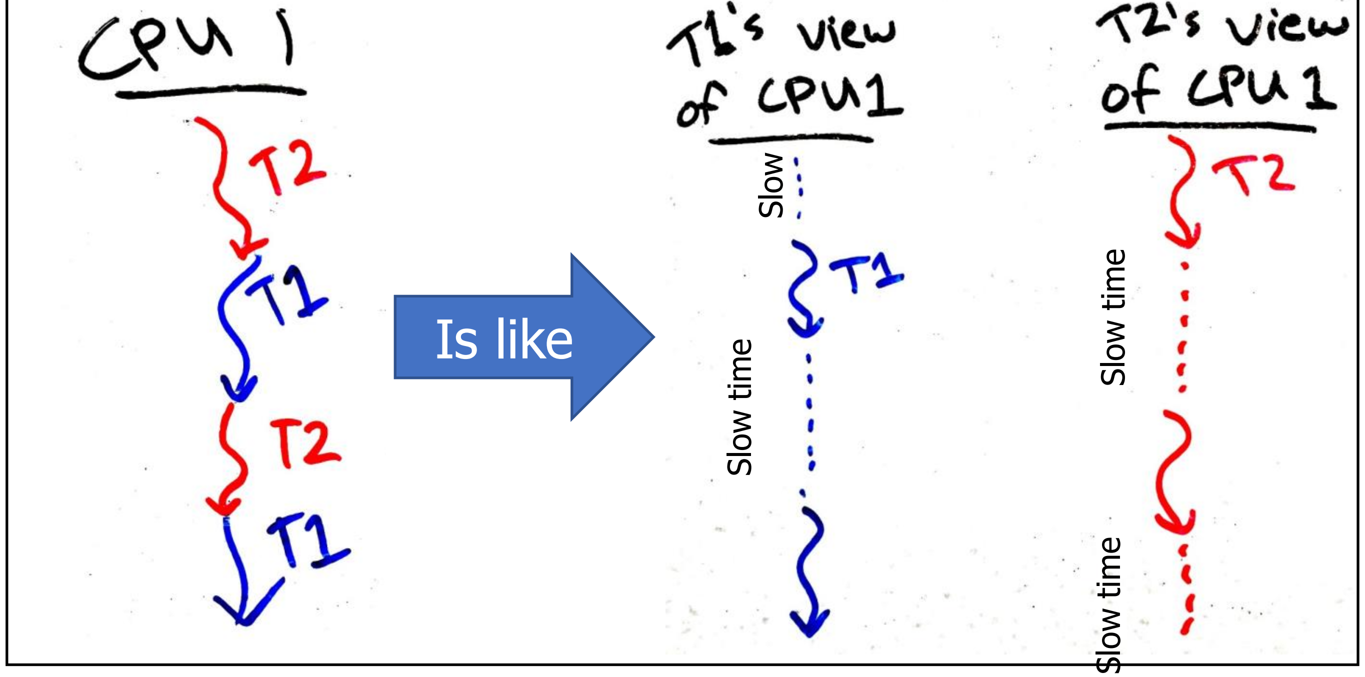
The process scheduler creates concurrency

- Even if only one CPU is present, threads operate "concurrently" because they are taking turns using the CPU.
- Each process thinks it has its own CPU that is sometimes very, very slow…

Obvious concurrency on two CPU cores:



Simulated concurrency on one CPU core:



Assume the scheduler is evil

- Remember that processes have no control over the scheduler.
- So, to protect against concurrency bugs, we must assume that the scheduler can interrupt us at any time and schedule any other process.
- In other words, assume that the scheduler is ***adversarial***, and will do the worst possible scheduling.
- To prevent weird and rare concurrency bugs, your code **must** work correctly even when faced with an *evil scheduler.*

Live example – data races when executing for less time

- What happens if we modify the loop duration?

```
[brghena@ubuntu race_condition] $ ./race
main: begin (counter = 0)
B: begin
B: done
A: begin
A: done
main: done with both (counter = 200, goal was 200)
```

- Thread is now completing its work before being re-scheduled
  - The problem is not solved, it will just occur rarely (and be harder to debug)

Race Condition

- Two or more things are happening at the same time
- It’s not clear which will run when
- The result will be different depending on execution order
- Result becomes indeterminate (non-deterministic)

- **Data race**:
  - Two or more threads access shared memory at the same time and at least one modifies it
  - (Race Conditions are more broad)

Data race fix for single-core machines: disable interrupts

```
void lock() {
  disable_interrupts();
}

void unlock() {
  enable_interrupts();
}
```

- Disable interrupts to prevent preemption during critical section
  - Scheduler can't run if the OS never takes control
  - Also stops data races in interrupt handlers
- Problems
  - Doesn't work on multicore machines
  - Bad Idea™ to let processes disable the OS
    - Process could freeze the entire computer
  - Might screw up timing for interrupt handling

Outline

- Race Conditions
- **Critical Sections**
- Lock Design
    - Overview
    - Basic Lock Implementation
    - Lock Optimizations

Critical Section

- Code that interacts with a shared resource must not be executed concurrently

- The code that accesses a shared resource is a **Critical Section**
    - In other words, code that would lead to a data race
    - May be multiple, unrelated critical sections for multiple shared resources

- Critical sections need to be addressed for correctness
    - Races can be avoided by never overlapping multiple critical sections
    - We must execute critical sections “atomically” (all or none)

Critical section occurs when shared memory is accessed

```
static volatile int counter = 0;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  printf("%s: begin\n“,(char*)arg);
  for (int i=0; i<LOOPS; i++) {
    counter++;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

Initialization code

```
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  printf("main: begin (counter = %d)\n",
counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d,
goal was %d)\n", counter, 2*LOOPS);
  return 0;
}
```

When do critical sections occur?

- Critical sections often involve modification of multiple related data
  - While the modifications are happening there is some inconsistency
  - The inconsistency is eventually resolved before leaving the critical section
- For example:
  - Inserting an element in the middle of a linked list
    - Two pointers must change. List is broken if just one is changed.
  - Reading a value and then modifying it
- Don’t have to worry about critical sections if:
  - Program is single-threaded, OR
  - The particular data is not shared among threads and modified
  - (or *maybe* if it’s just one assembly instruction, depends)

Check your understanding. Where is the critical section?

```c
static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

Initialization code

```c
int main(int argc, char* argv[]) {
 pthread_t p1, p2;
 person1 = "Jack";
 person2 = "Jill";
 printf("main: begin (%s, %s)\n",
        person1, person2);
 pthread_create(&p1,NULL,mythread,"A");
 pthread_create(&p2,NULL,mythread,"B");

 // wait for threads to finish
pthread_join(p1, NULL);
 pthread_join(p2, NULL);
 printf("main: end (%s, %s)\n",
        person1, person2);
}
```

Buggy concurrent swap. What can go wrong?

```c
static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

Initialization code

```c
int main(int argc, char* argv[]) {
 pthread_t p1, p2;
 person1 = "Jack";
 person2 = "Jill";
 printf("main: begin (%s, %s)\n",
        person1, person2);
 pthread_create(&p1,NULL,mythread,"A");
 pthread_create(&p2,NULL,mythread,"B");

 // wait for threads to finish
pthread_join(p1, NULL);
 pthread_join(p2, NULL);
 printf("main: end (%s, %s)\n",
        person1, person2);
}
```

Buggy concurrent swap. What can go wrong?

```c
static volatile char* person1;
static volatile char* person2;
static const int LOOPS = 1e4;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  int i;
  for (i=0; i<LOOPS; i++) {
    // swap
    volatile char* tmp = person1;
    person1 = person2;
    person2 = tmp;
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

Initialization code

```c
int main(int argc, char* argv[]) {
 pthread_t p1, p2;
 person1 = "Jack";
 person2 = "Jill";

 p
 pthread_join(p2, NULL);
 printf("main: end (%s, %s)\n",
        person1, person2);
}
```

For a brief period in time:

person1: “Jill”
person2: “Jill”

Break + Check your understanding. Is there a problem here?

```c
static volatile int sum_amount = 2;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  int counter = 0;
  printf("%s: begin\n", (char*)arg);

  for (int i=0; i<LOOPS; i++) {
    counter += sum_amount;
  }

  printf("%s: done %d\n",(char*)arg,
                        counter);
  return NULL;
}
```

Initialization code

```c
int main(int argc, char* argv[]) {
 pthread_t p1, p2;
 printf("main: begin\n");
 pthread_create(&p1, NULL, mythread, "A");
 pthread_create(&p2, NULL, mythread, "B");

 // wait for threads to finish
 pthread_join(p1, NULL);
 pthread_join(p2, NULL);
 printf("main: done\n");
 return 0;
}
```

Break + Check your understanding. Is there a problem here?

```
static volatile int sum_amount = 2;
static const int LOOPS = 1e7;

void* mythread(void* arg) {
  int counter = 0;
  printf("%s: begin\n", (char*)arg);

  for (int i=0; i<LOOPS; i++) {
    counter += sum_amount;
  }

  printf("%s: done %d\n",(char*)arg,
                          counter);

  return NULL;
}
```

This code will work!

All threads only *read* from shared memory.

If at least one *wrote* to shared memory, it would be a problem.

```
pthread_join(p2, NULL);
printf("main: done\n");
return 0;
}
```

Outline

- Race Conditions
- Critical Sections
- **Lock Design**
  - **Overview**
  - Basic Lock Implementation
  - Lock Optimizations

Solution Requirements

We **MUST** stop data races from occurring in our programs.

1. No two threads may simultaneously be in their critical sections.

2. Threads outside of critical sections should have no impact.

3. No assumptions should be made about number of cores, speed of cores, or scheduler choices.

Locks (also known as a mutex)

- Locks are the simplest mutual exclusion primitive
  - Represent a resource that can be reserved and freed

- ***Acquire/lock***:
  - Used before a critical section to **reserve** the resource
  - If the lock is free (unlocked), then lock it and proceed.
  - If the lock is already taken (someone else called *acquire/lock*), then **wait until it's free** before proceeding.
- ***Release/unlock***:
  - Used at the end of a critical section to **free** the resource
  - Only the thread holding the lock can release it
  - Allows one waiting (or future) thread to acquire the lock

Two different metaphors & etymology

Lock

- Think about locking a bathroom door
- Our virtual lock works as follows:
  - Anyone can **lock** or **unlock** (there is no "key").
  - Trying to enter (**lock**) if the lock is already-locked will cause you to wait until it's unlocked.

Token

- Holding the token gives you permission to do something.
- There is only one token.
- Thus, you:
  1. Try to **acquire** the token ("lock"). You have to wait your turn if someone else is holding it.
  2. When done, **release** the token/lock.
- The token represents exclusive access to a shared resource or a critical section.

Locks prevent data races

```c
static volatile int counter = 0;
static const int LOOPS = 1e7;
static pthread_mutex_t lock;

void* mythread(void* arg) {
  printf("%s: begin\n", (char*)arg);
  for (int i=0; i<LOOPS; i++) {
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
  }
  printf("%s: done\n", (char*)arg);
  return NULL;
}
```

```c
int main(int argc, char* argv[]) {
  pthread_t p1, p2;
  pthread_mutex_init(&lock, 0);
  printf("main: begin (counter = %d)\n",
counter);
  pthread_create(&p1, NULL, mythread, "A");
  pthread_create(&p2, NULL, mythread, "B");

  // wait for threads to finish
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  printf("main: done with both (counter = %d,
goal was %d)\n", counter, 2*LOOPS);
  return 0;
}
```

Guidelines for implementing locks

Requirements for 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 Wait (starvation-free):
  - Must eventually allow every waiting thread to proceed

Additional goals

- Fairness – each thread waits for the same amount of time
- Performance – do the above in minimal execution time

Outline

- Race Conditions

- Critical Sections

- **Lock Design**
  - Overview
  - **Basic Lock Implementation**
  - Lock Optimizations

Algorithmic approach: Peterson’s Solution

- There are indeed several algorithmic approaches to create a lock!
- See textbook (or other sources) for Peterson’s Solution for two threads
- Advantages:
  - Algorithm, so it works on any platform no matter the hardware
- Disadvantages:
  - Solution for N threads gets *complicated*
  - Performance is slow

Hardware approach: atomic instructions

- **Atomic** instructions perform operations on memory in one uninterruptable instruction
  - Guarantees that all parts of the instruction occur before the next instruction
  - In multicore, guarantees that entire access to memory is serialized

- Commonly read, modify, and write in a single instruction

Atomic Instruction: Exchange

- Example atomic_exchange

pseudocode for the instruction: remember, this is actually in hardware NOT C

```
int atomic_exchange(int* pointer, int new_value) {
    int old_value = *pointer; // fetch old value from memory
    *pointer = new_value;     // write new value to memory
    return old_value;         // return old value
}
```

- atomic_exchange(destptr, newval)
  - Write a new value to memory, and return the old one
  - Also known as test-and-set when operating on boolean data
  - x86-64 instruction: `lock; xchg`

Atomic Instruction: Compare And Swap

- Example atomic_compare_and_swap (remember, this is pseudocode for hardware)

```
bool atomic_compare_and_swap(int* pointer, int expected_value, int new_value) {
    int actual_value = *pointer;
    if (actual_value == expected_value) {
        *pointer = new_value;
        return true;
    }
    return false;
}
```

atomic_compare_and_swap(destptr, oldval, newval)

x86-64 instruction: `lock; cmpxchg`

Generalization of exchange

Other atomics: sequential memory consistency

- Memory barrier
  - Guarantees that all load/stores **before** this line of code are completed before any load/stores **after** this line of code are started
  - Comes in software (compiler orders things) and hardware (processor orders things) forms
    - Both are necessary for correct execution!
  - C wrappers for atomics allow you to specify a memory barrier

- Atomic Load/Store C-wrappers
  - Guarantee sequential consistency (will happen in order)
  - Remember:
    - Normally, accesses could be reordered by compiler or processor!

Atomic instructions can be used to build locks

- Spinlocks
  - Simple lock mechanism built on top of atomic instructions
  - Thread “spins” in a loop until the lock is available
- Notably: spinlock implementation doesn’t interact with the OS kernel
  - No system calls
  - Upside: Very cheap – no context switch
  - Downside: Scheduler isn’t aware that the thread is “blocked”

Spinlock implementation

```
typedef struct {
    int flag; // 0 indicates that mutex is available, 1 that it is held
} lock_t;

void mutex_init(lock_t* mutex) {
    mutex->flag = 0; // lock starts available
}
void mutex_acquire(lock_t* mutex) {
    while (atomic_exchange(&(mutex->flag), 1) == 1); // spin-wait until available
}
void mutex_release(lock_t* mutex) {
    atomic_store(&(mutex->flag), 0); // make lock available
}
```

Break + Question: did we need atomics?

Initialization: `bool lock = false;`

Is this code sufficient?

```
// wait for lock released
while (lock != 0);
// lock == 0 now (unlocked)

// set lock
lock = 1;

     // access shared resource ...

// release lock
lock = 0;
```

Break + Question: did we need atomics?

Initialization: `bool lock = false;`

```
// wait for lock released
while (lock != 0);
// lock == 0 now (unlocked)

// set lock
lock = 1;

     // access shared resource ...

// release lock
lock = 0;
```

Is this code sufficient?

No! **lock** is a shared resource and reading then writing it is not atomic

Outline

- Race Conditions
- Critical Sections
- **Lock Design**
  - Overview
  - Basic Lock Implementation
  - **Lock Optimizations**

Evaluating a lock

Requirements for correctness

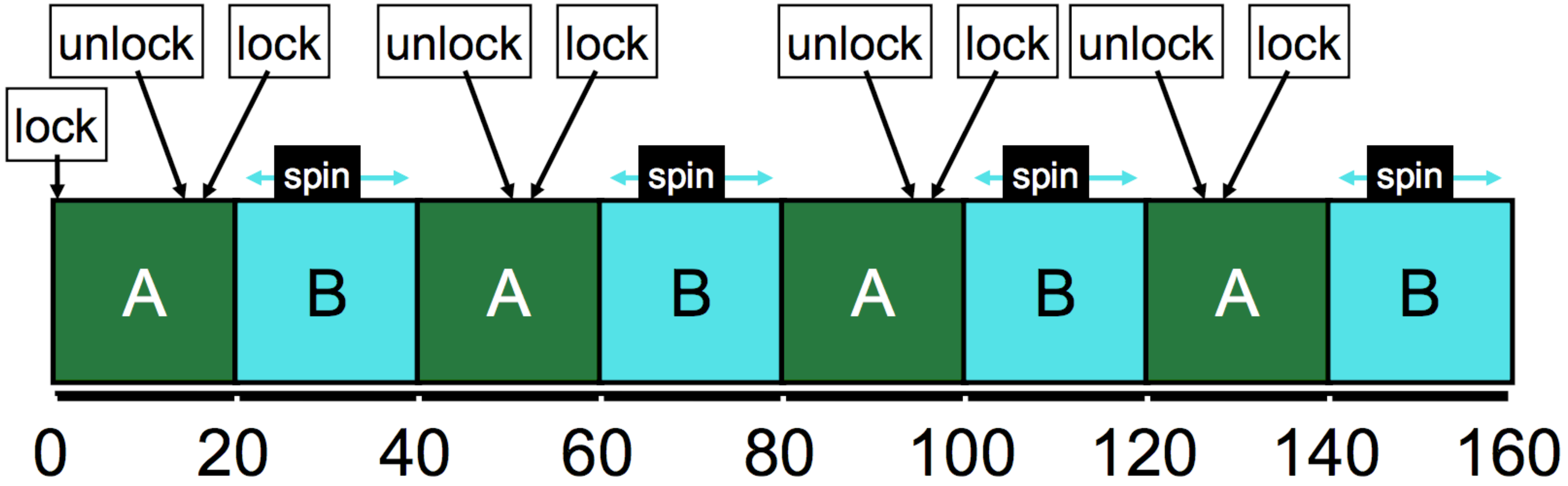
- Mutual Exclusion:
    - Only one thread in critical section at a time
- Progress (deadlock-free):
    - If several simultaneous requests, must allow one to proceed
- Bounded Wait (starvation-free):
    - Must eventually allow every waiting thread to proceed

Additional goals

- Fairness – each thread waits for the same amount of time
- Performance – do the above in minimal execution time

Spinlock evaluation - Correctness

- Mutual Exclusion and Progress **Yes**
- Bounded Wait **No**
  - No guarantee that a thread will eventually get its turn (assume an infinite system)



Spinlock evaluation – Goals

- Fairness
  - Doesn’t even guarantee no starvation
  - No control at all over whether each thread waits an even amount

- Performance (uniprocessor)
  - Process “spins”, repeatedly checking a variable that will not change
  - Timeslice must expire before another thread is given a chance to unlock
  - If N threads want the lock, then N timeslices are wasted spinning

- Performance (multiprocessor)
  - Doesn’t waste entire timeslice anymore
  - No calls to OS means process gets the lock as soon as it is free. So fast!

Addressing the bounded wait problem

- Need some way to track "whose turn it is" to take the lock
- You can have the lock when not held AND it's no one else's turn

- Idea: hand out numbered tickets

Atomic Instruction: Fetch and Add

- Example atomic_fetch_and_add (remember, in hardware **not C**)

```
int atomic_fetch_and_add(int* pointer, int increment) {
    int old_value = *pointer;
    *pointer = old_value + increment;
    return old_value;
}
```

- atomic_fetch_and_add(destptr, incr)
  - Add a new value to the current value in memory, and return the old one
  - x86-64 instruction: `lock; xadd`

- List of C wrappers available here: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

Ticket lock implementation

```
typedef struct {
 unsigned int ticket; // current available ticket
 unsigned int turn;   // which ticket gets to proceed
} lock_t;

void mutex_init(lock_t* mutex) {
 mutex->ticket = 0; mutex->turn = 0;
}

void mutex_lock(lock_t* mutex) {
 int myturn = atomic_fetch_and_add(&(mutex->ticket), 1); // take a ticket
 while (mutex->turn != myturn); // spin-wait until available
}

void mutex_unlock(lock_t* mutex) {
 atomic_fetch_and_add(&(mutex->turn), 1); // next turn
}
```

- Each thread atomically reserves its turn
- Unique turn numbers prevent a data race
  - Fails with 2^32 threads!
- When finished, set to next turn

Prevents starvation with FIFO ordering of access!

Ticket Lock Example

A lock(): Ticket 0, Turn 0 // enter critical section!

B lock(): Ticket 1, Turn 0 // wait

C lock(): Ticket 2, Turn 0 // wait

Ticket Lock Example

A lock(): Ticket 0, Turn 0

B lock(): Ticket 1, Turn 0

C lock(): Ticket 2, Turn 0

A unlock(): Turn 1 // B gets to go

Ticket Lock Example

A lock(): Ticket 0, Turn 0

B lock(): Ticket 1, Turn 0

C lock(): Ticket 2, Turn 0

A unlock(): Turn 1

A lock(): Ticket 3, Turn 1

B unlock(): Turn 2 // C gets to go

Ticket Lock Example

A lock(): Ticket 0, Turn 0

B lock(): Ticket 1, Turn 0

C lock(): Ticket 2, Turn 0

A unlock(): Turn 1

A lock(): Ticket 3, Turn 1

B unlock(): Turn 2

C unlock(): Turn 3 // A gets to go again

A unlock(): Turn 4 // Available ticket is turn 4 too, so next request goes immediately

Ticket Lock Evaluation

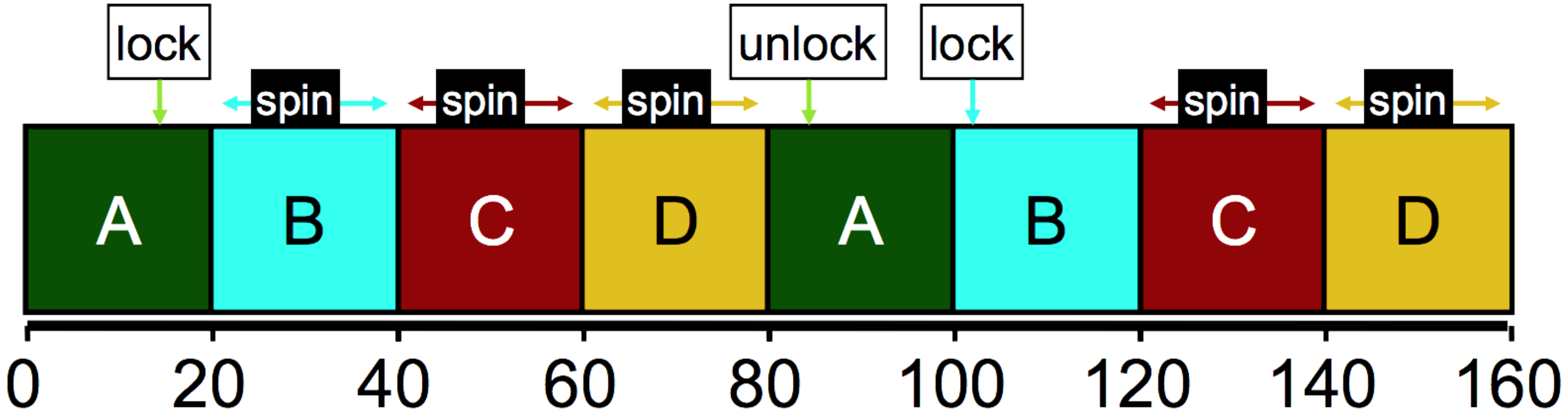
Correctness: Mutual Exclusion, Progress, Bounded Wait **Yes**

Goals

- Fairness **Yes**
  - FIFO ordering of threads
- Performance
  - Similar positives and negatives as original spinlock
  - One downside: on a **release**() all threads must check if it is their turn

Ticket lock still wastes time spinning

- B, C, and D are "busy waiting"
  - Might be occupying an entire core in multicore
- Scheduler is fairly scheduling all threads, but ignorant of locks
- Idea: can we skip threads that are waiting on a lock?

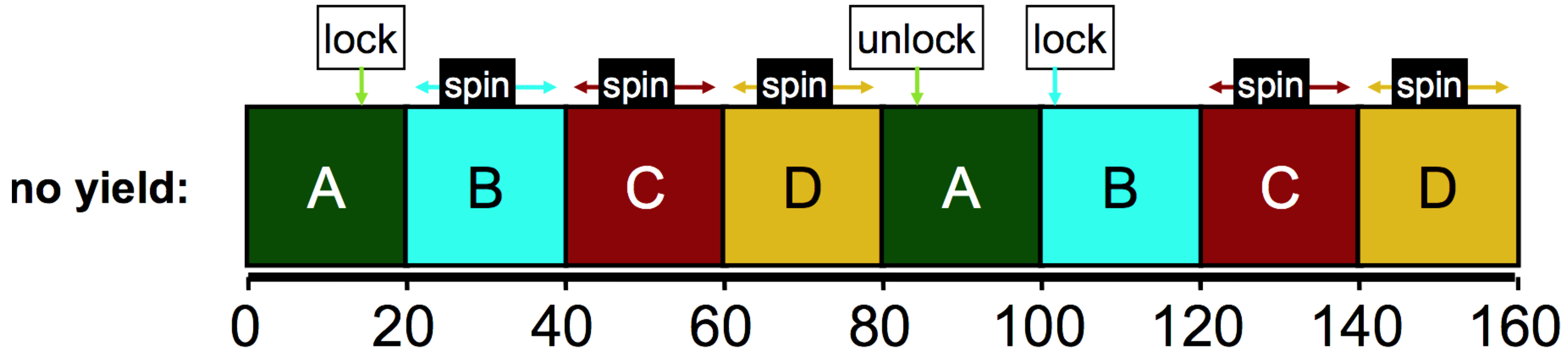


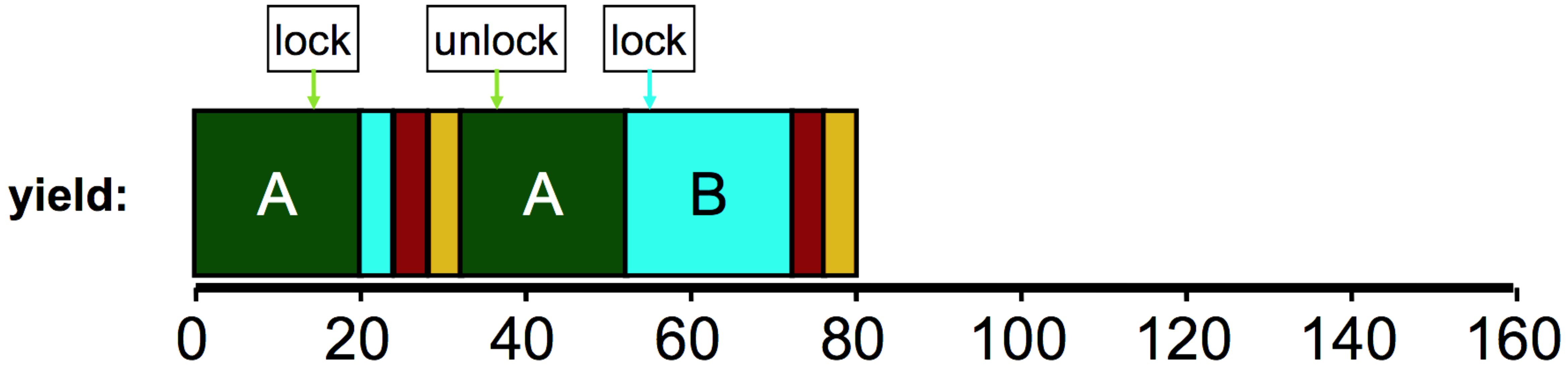
Yield timeslice when not yet ready

- Yield syscall unschedules the current thread
  - sched_yield() in POSIX API
  - Gives the user process *just a little* control over the scheduler
- In acquire(), yield after checking condition
- Might delay thread response time in multicore scenario

```
void mutex_lock(lock_t* mutex) {
    int myturn = atomic_fetch_and_add(&(mutex->ticket), 1); // take a ticket
    while (mutex->turn != myturn) {
        sched_yield(); // not ready yet
    }
}
```

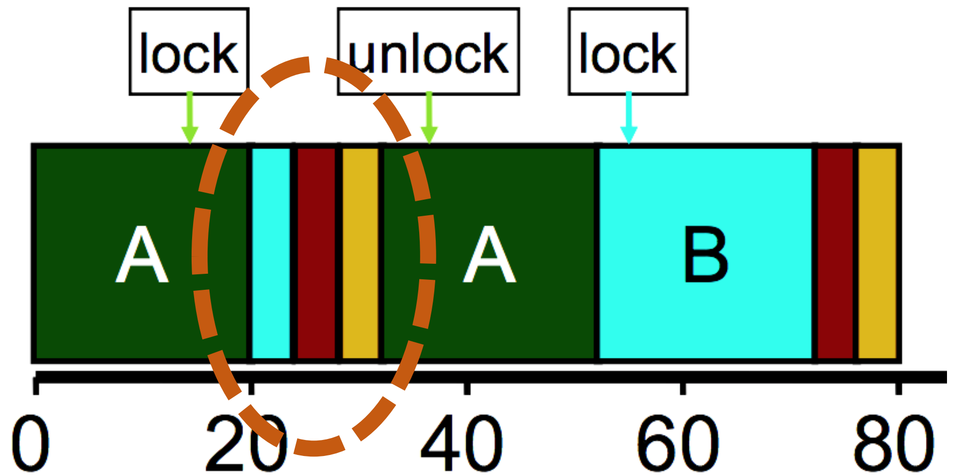
Yielding reduces busy-waiting





How much does yielding improve things?

- Performance better with yield(), but still doing a lot of unnecessary context switches



- Wasted CPU cycles
  - Without yield(): O(threads*timeslice)
  - With yield(): O(threads*context_switch)
  - Timeslice ~1 ms, Context switch: ~1 µs

- Still expensive if we expect many threads to be contending over the lock

Building a blocking lock

- A more performant solution requires cooperation between thread's locks and the OS scheduler to block threads

- Some OSes (Solaris) have system calls to do so
  - park() – blocks the current thread
  - unpark(thread_id) – unblocks another thread, specified by thread ID

- Building locks on park/unpark
  - If lock **acquire** fails, add own thread ID to waiting thread queue and park()
  - **Release** dequeues the next waiting thread ID and calls unpark() on it
  - Fairness: unlocking thread effectively decides which thread goes next

Linux Futex (fast userspace mutex) syscalls

- Similar to park/unpark, but the queue is in the kernel
- Key idea: only makes the kernel calls when you actually need to wait or wake a sleeping thread

- futex_wait(int* pointer, int expected)
  - Put thread to sleep if the value at address equals "expected"
  - Used to build **acquire**()

- futex_wake(int* pointer)
  - Unblock one thread waiting on "pointer"
  - Used to build **release**()

- See https://eli.thegreenplace.net/2018/basics-of-futexes/

Spinning versus Blocking

- Each approach is better under different circumstances

- **Single core systems**
  - If waiting process is scheduled, then process holding lock is not
  - Waiting process should *always* yield its time

- **Multicore systems**
  - If waiting process is scheduled, then process holding lock could also be
  - Spin or block depends how long until the lock is released
    - If the lock is released quickly, spin wait
    - If the lock is released slowly, block
    - Where quick and slow are relative to context-switch cost

Two-phase waiting

- Problem: we can't always know how long the wait will be
  - Programmer might know…
  - Library definitely can't know

- Idea:
  - Spin lock for a little while, and then give up and block
  - Example: Linux Native POSIX Thread Library (NPTL)
    - Check the lock at least *three* times before blocking with Futex

Summary on lock implementations

- Spinlocks
- Ticket locks
- Yielding locks
- Queueing locks
  - Futex on Linux

- Sophisticated locks are more fair and do not waste processor time "busy waiting"
- But also have unnecessary context-switch overhead if the lock is only briefly and rarely held

Outline

- Race Conditions
- Critical Sections
- Lock Design
    - Overview
    - Basic Lock Implementation
    - Lock Optimizations