Parallelism, Multicore, and Synchronization

Big Picture: Parallelism and Synchronization

How do I take advantage of *parallelism*?

How do I write (correct) parallel programs?

What primitives do I need to implement correct parallel programs?

Parallelism & Synchronization

Multicore → more cores!

Cache Coherency

- Processors cache *shared* data → they see different (incoherent) values for the *same* memory location

Threads

- Mechanism to take advantage of parallelism

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

Threads

Introducing: Thread-Level parallelism

Threads are separate tasks within the same process

Threads can run:

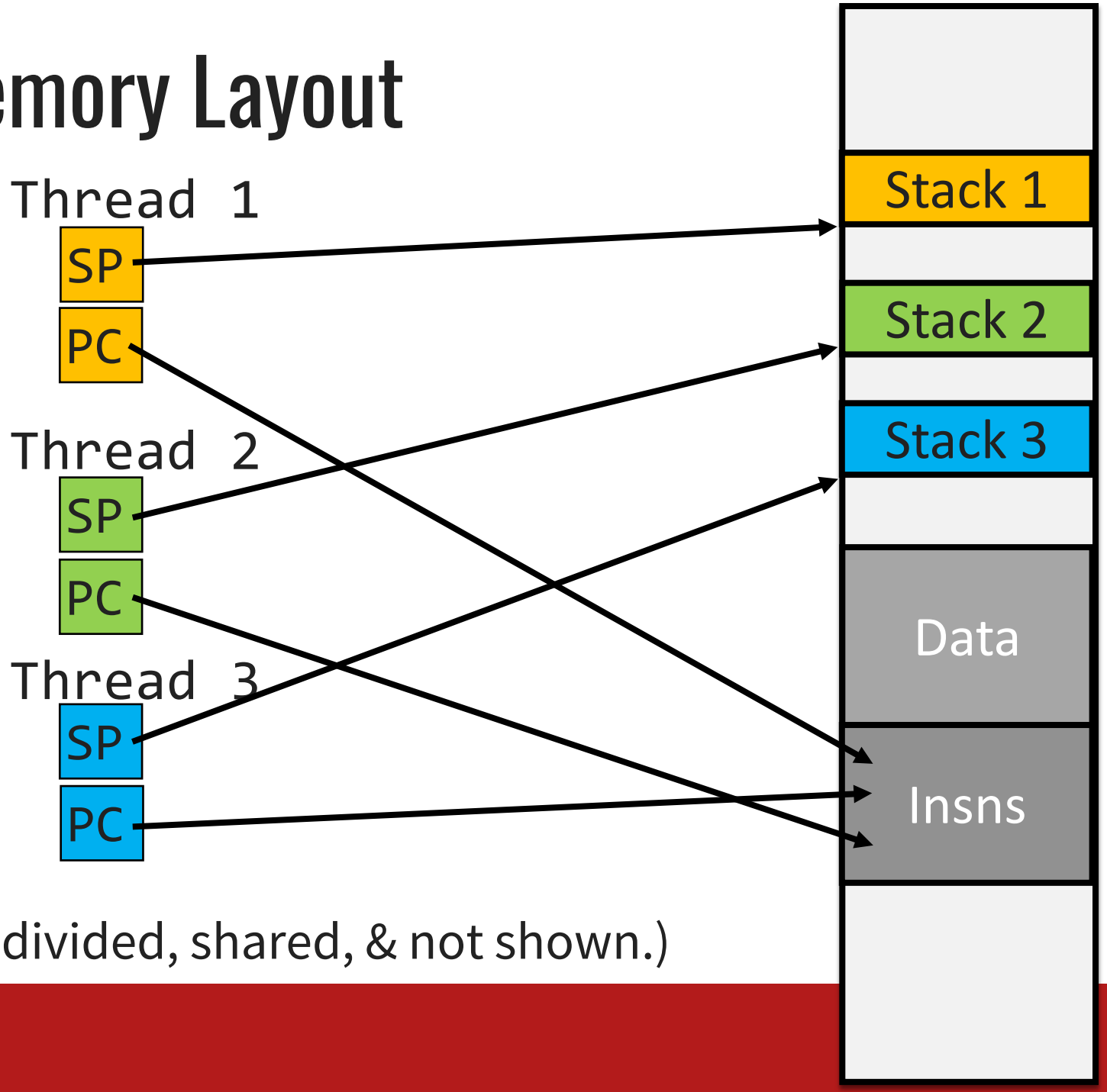
- On separate cores
- Taking turns on one core
- On one core at the same time (hyperthreading)

What is a thread?

Threads: concurrent computations that share the same address space

- Share: code, data, heap, files
- Do not share: registers or stack

Thread Memory Layout



(Heap subdivided, shared, & not shown.)

Why Threads?

Performance: exploiting multiple processors

Do threads make sense on a single core?

Encourages natural program structure

- Expressing logically concurrent tasks
- update screen, fetching data, receive user input

Responsiveness

- threads can do work in the background

Mask long latency of I/O devices

- do useful work while waiting

Problem

Thread 1	Thread 2
`for(i=0; i<5;i++){`	`for(i=0; i<5;i++) {`
`  x++`	`  x++`
`}`	`}`

Big Picture: Programming with threads

Within a thread: execution is sequential

Between threads?

- (Almost) no ordering or timing guarantees
- Might all execute on same core, or not!

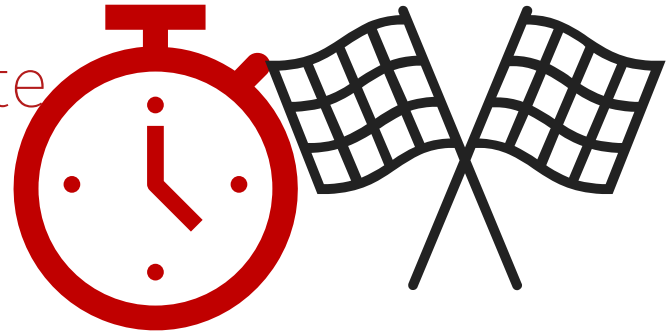
Problem: hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency is necessary but not sufficient…

- *Explicit* synchronization (i.e., software directed) is also necessary

Race conditions



Timing-dependent error involving access to shared state

Race conditions depend on how threads are scheduled

- i.e., who wins "the race" to update state

Challenges of Race Conditions

- Races are intermittent, may occur rarely or in certain scena
- They often appear to happen randomly

Program is correct *only* if *all possible* schedules are safe

Simplest Race Condition: 2 threads trying to increment `x`

Hardware Support for Synchronization

Atomic read & write memory operation

- Between read & write: *no writes to that address*

Many atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and swap (x86, ARM deprecated)
- load reserved / store conditional
  (RISC-V, MIPS, ARM, PowerPC, DEC Alpha, …)

Synchronization in RISC-V

Load Reserved: `LR.{W,D} rd, (rs1)*`

"I want the value at address X. Also, start monitoring any writes to this address."

Store Conditional: `SC.{W,D} rd, rs2, (rs1)*`

"If no one has changed the value at address X since the LR.{W,D}, perform this store and tell me it worked."

- Data at location has NOT been written to since the LR?
  - SUCCESS:
    - Performs the store (value in rs2 written to address in rs1)
    - Returns 0 in rd
- Data at location has been written to since the LR?
  - FAILURE:
    - Does not perform the store
    - Returns 1 in rd

* There are a few ways of writing the ASM; this is the notation we'll use

Using LR/SC to create Atomic Increment

- Load Reserved: `LR.{W,D} rd, (rs1)`
- Store Conditional: `SC.{W,D} rd, rs2, (rs1)`

i++

↓

```
LW t0, (s1)
ADDI t0,t0,1
SW t0, (s1)
```

atomic(i++)

↓

```
try:  LR.W t0, (s1)
      ADDI t0,t0,1
      SC.W t0, t0,(s1)
      BNEZ t0, try
```

Value in memory written between LR and SC ?

→ SC returns 1 in **t0** → go back & try again

Atomic Increment in Action

- Load Reserved: `LR.{W,D} rd, (rs1)`
- Store Conditional: `SC.{W,D} rd, rs2, (rs1)`

Time	Thread A	Thread B	Thread A t0	Thread B t0	Mem [s1]
0					0
1					
2					
3					
4					
5					
6					
7					
8					

Critical Sections

- Create atomic version of every instruction?
  - NO: Does not scale *or solve the problem*

- To eliminate races: identify *Critical Sections*
  - Places in code where shared state is read and written
  - Only 1 thread gets to execute at a time
  - Others *wait their turn*

Critical Section!

```
...
tmp = *cur_score;                                  //read shared var
*cur_score = update_score(tmp);                    //write shared var
if (tmp > *high_score) { *high_score = tmp; } //r/w shared var
...
```

Critical Sections

- Create atomic version of every instruction?
  - NO: Does not scale *or solve the problem*

- To eliminate races: identify *Critical Sections*
  - Places in code where shared state is read and written
  - Only 1 thread gets to execute at a time
  - Others *wait their turn*

```
cs_enter();
tmp = *cur_score;                                  //read shared var
*cur_score = update_score(tmp);                    //write shared var
if (tmp > *high_score) { *high_score = tmp; } //r/w shared var
cs_exit();
```

Mutual Exclusion Lock (Mutex)

- Implementation of *cs_enter()* and *cs_exit()*

```
lock = 0; // global variable
          // 0 means lock is free
          // 1 means lock is taken
```

```
cs_enter(lock);
tmp = *cur_score;
*cur_score = update_score(tmp);
if (tmp > *high_score) {
 *high_score = tmp;
}
cs_exit(lock);
...
```

Thread 0

Atomically:
Wait until lock is 0, then set to 1

I am the ONLY THREAD running this code!

Set lock to 0

Mutex from LR and SC

Start Here

```
lock = 0;  // global variable @ address 0x100
           // 0 => free; 1 => taken
mutex_lock(int *lockAddr) {




}

mutex_unlock(int *lockAddr) {
       ...
}
```

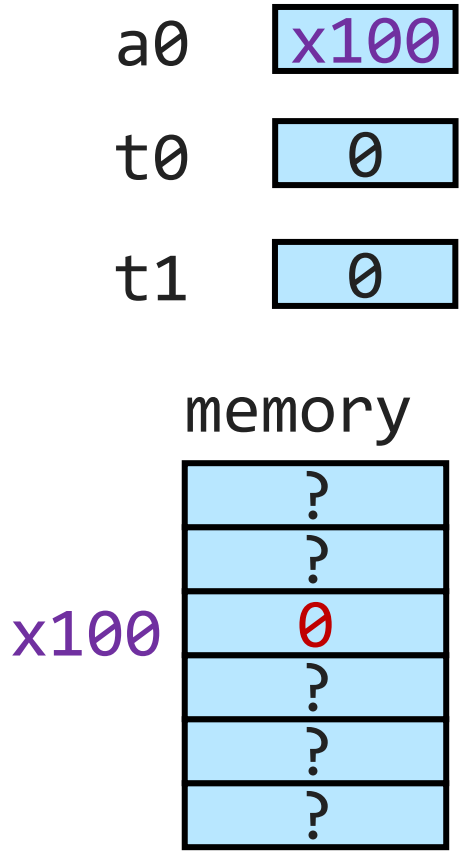
a0 x100

memory

	?
	?
x100	0
	?
	?
	?

Mutex from LR and SC

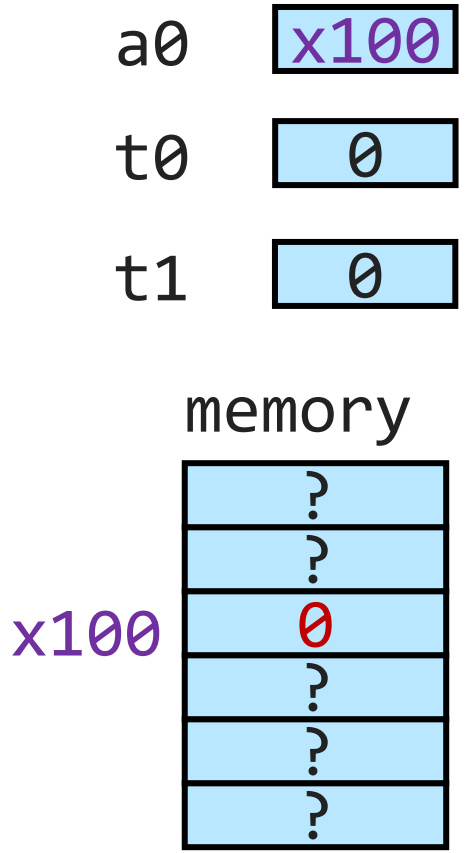
```
lock = 0;  // global variable @ address 0x100
           // 0 => free; 1 => taken
mutex_lock(int *lockAddr) { // a spin lock
 t_and_s: LI t0, 1
          LR.W t1, (a0)
          BNEZ t1, t_and_s // t1 != 0 => lock busy
          SC.W t0, t0, (a0)
          BNEZ t0, t_and_s // t0 != 0 => lost the race
}

mutex_unlock(int *lockAddr) {
       SW x0, (a0);
}
```



Mutex from LR and SC

```
lock = 0;  // global variable @ address 0x100
           // 0 => free; 1 => taken
mutex_lock(int *lockAddr) { // a spin lock
  while(t_and_s(lockAddr)){}
}
int t_and_s(int *lockAddr) {
  old = *lockAddr;
  *lockAddr = 1;
  return old;
}
mutex_unlock(int *lockAddr) {
      *lockAddr = 0;
}
```

LR.W
SC.W
Atomic



2 threads attempt to grab the lock

`mutex_lock(int *lockAddr)`

```
try: LI t0, 1
     LR.W t1, (a0)
     BNEZ t1, try
     SC.W t0, t0, (a0)
     BNEZ t0, try
```

Time	Thread A	Thread B	ThreadA		ThreadB		Mem
			t0	t1	t0	t1	M[a0]
0							**0**
1	try: LI t0, 1	try: LI t0, 1	**1**		**1**		0
2							
3							
4							
5							
6							
7							
8							

Problem Solved?

```
lock = 0;
x = 0;
```

Thread 1

```
for (i = 0; i < 5; i++) {
    mutex_lock(&lock);
    x = x + 1;
    mutex_unlock(&lock);
}
```

Thread 2

```
for (i = 0; i < 5; i++) {
    mutex_lock(&lock);
    x = x + 1;
    mutex_unlock(&lock);
}
```

Synchronization Variations

Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

Semaphores

- N threads can hold lock at the same time
- Used for "resource counting"

Monitors

- Concurrency-safe data structure with 1 mutex
- All operations on monitor acquire/release mutex
- One thread in the monitor at a time

Curious about these? *Take CS 4410!*

CS 3410 Takeaway: HW provides the primitives (e.g., LR/SC) to support thread-level synchronization operations.

Summary

Need parallel abstractions, especially for multicore

Writing correct programs is hard

- Need to prevent data races

Need critical sections to prevent data races

- Mutex, mutual exclusion, implements critical section
- Mutex often implemented using a lock abstraction

Hardware provides synchronization primitives such as LR and SC (load reserved and store conditional) instructions to efficiently implement locks