CONCURRENCY: INTRODUCTION

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Project 3 Due Tonight

- Turn in Makefile and all src code (.c and .h) to make mysh

Plan Project 4 to be available Tuesday

Office hours instead of Lab Hours from Friday to Monday?

Midterm 1 on Thursday 7:30pm - 9:30pm

- Covers material including today's lecture
- Read textbook if you haven't
- Complete Canvas homeworks
- Take sample exams and check what you don't understand
- Look over lecture notes; can you answer all questions?
- Post questions for Tuesday lecture review

AGENDA / LEARNING OUTCOMES

Virtual memory: Summary

Concurrency

- What is the motivation for concurrent execution?
- What are some of the challenges?

RECAP

SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to make large disk seem like memory

- Same behavior as if all of address space in main memory

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

Else TLB miss...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory
  (i.e., present bit is cleared)

Else Page fault…

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace (look at use bits for clock)
  - Write victim page out to disk if modified (add dirty bit to PTE)
- OS reads referenced page from disk into memory (faulting process is blocked)
- Page table is updated, present bit is set
- Process continues execution (process moved to ready state)

CHAT: PERFORMANCE IMPACT?

Assume workload of multiple processes

- Processes each alternate between CPU and I/O
- Each access some amount of memory

What happens to system performance (throughput = jobs/sec) as we increase the number of processes?

- If the sum of the working sets > physical memory?

BONUS: WHAT IF NO HARDWARE SUPPORT?

What can the OS do if hardware does not have `use` bit (or `dirty` bit) (and hardware-filled TLB)?

- Can the OS "emulate" these bits?

Leading question:

- How can the OS get control (i.e., generate a trap) every time `use` bit should be set? (i.e., when a page is accessed?)

SUMMARY: VIRTUAL MEMORY

Abstraction: Virtual address space with code, heap, stack

Address translation

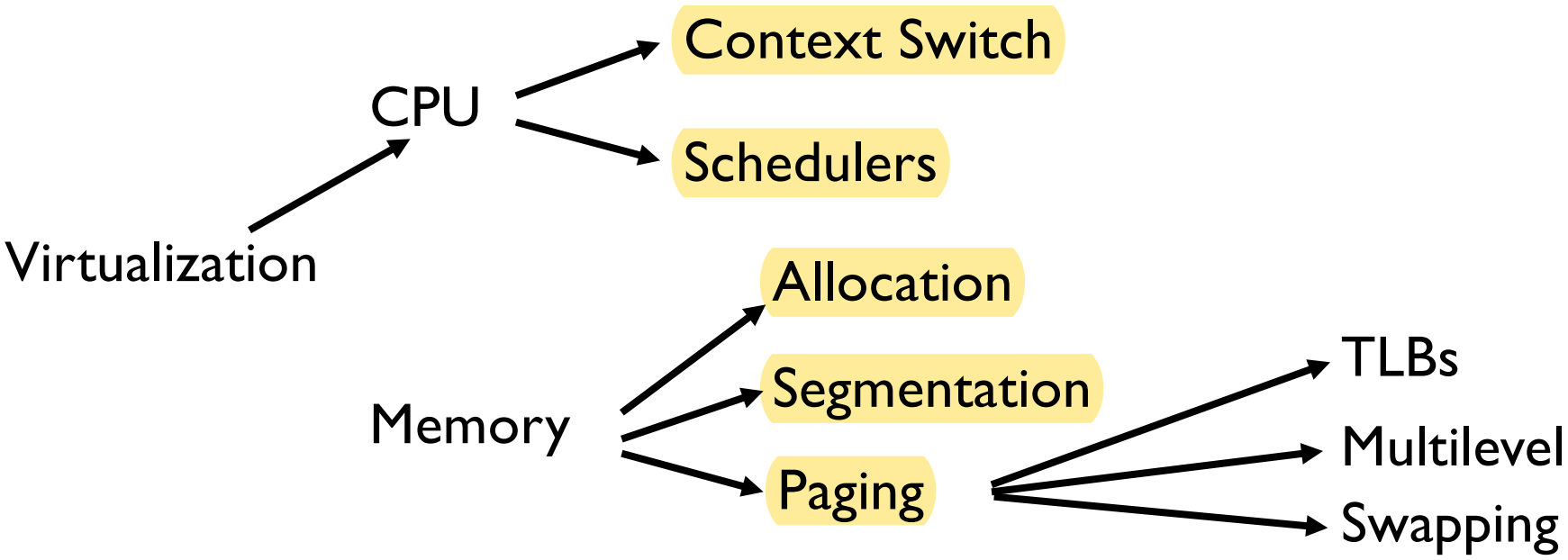
- Contiguous memory: base, bounds, segmentation
- Using fixed sizes pages with page tables

Challenges with paging

- Extra memory references: avoid with TLB
- Page table size: avoid with multi-level paging

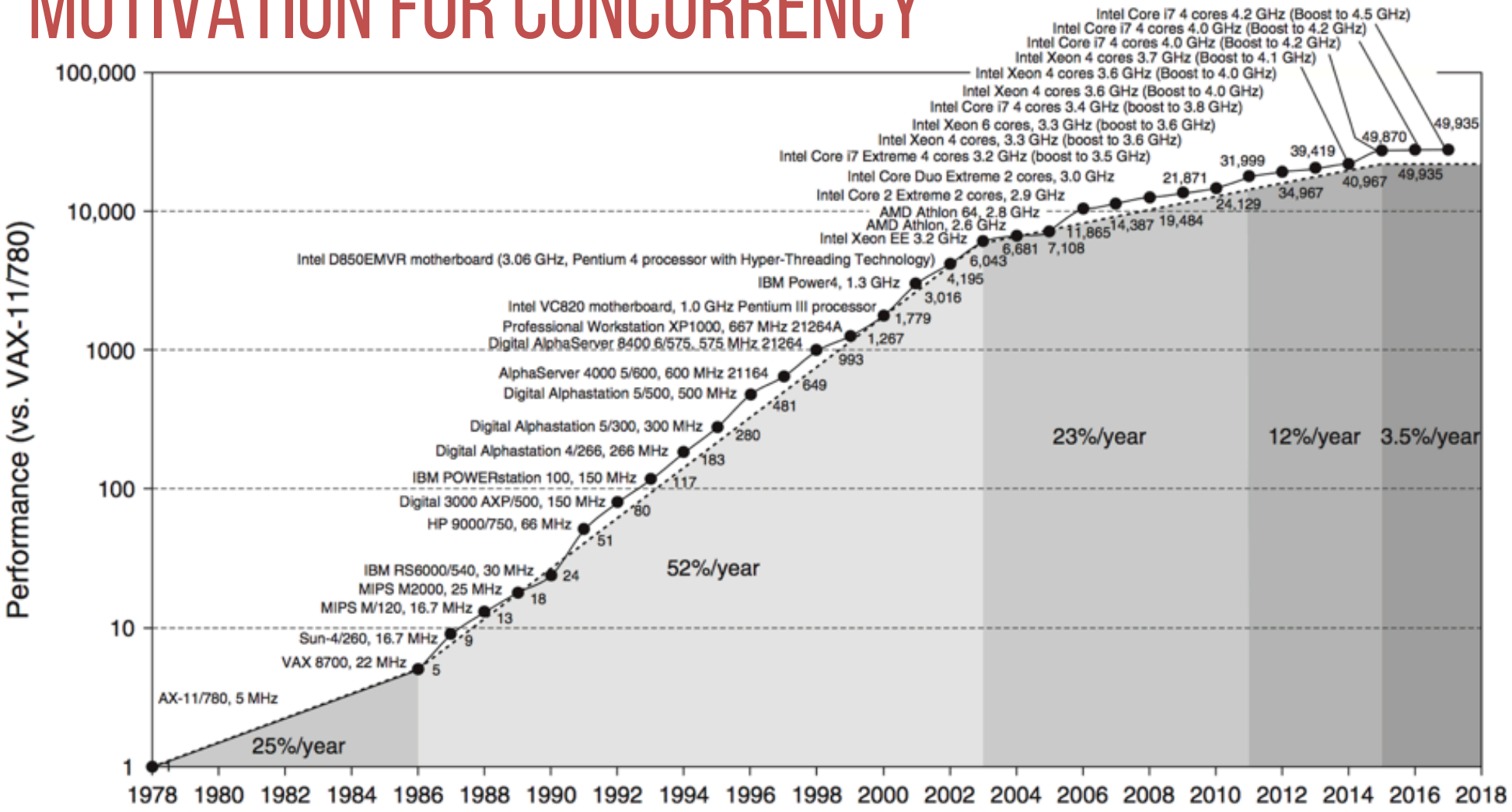
Larger address spaces: Swapping mechanisms, policies (LRU, Clock)

REVIEW: EASY PIECE 1



CONCURRENCY

MOTIVATION FOR CONCURRENCY



MOTIVATION

CPU Trend: Same speed, but multiple cores

Option 0: Run many different applications on one machine

- OS scheduler ensures cores are used efficiently

Our Goal: Write applications that fully utilize many cores

OPTION 1: USE MULTIPLE PROCESSES

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

- Example: Chrome (process per tab)
- Communicate via pipe() or other message passing primitives

Pros?

- Don't need new abstractions; good for security

Cons?

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

OPTION 2: DIVIDE PROCESS INTO THREADS

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

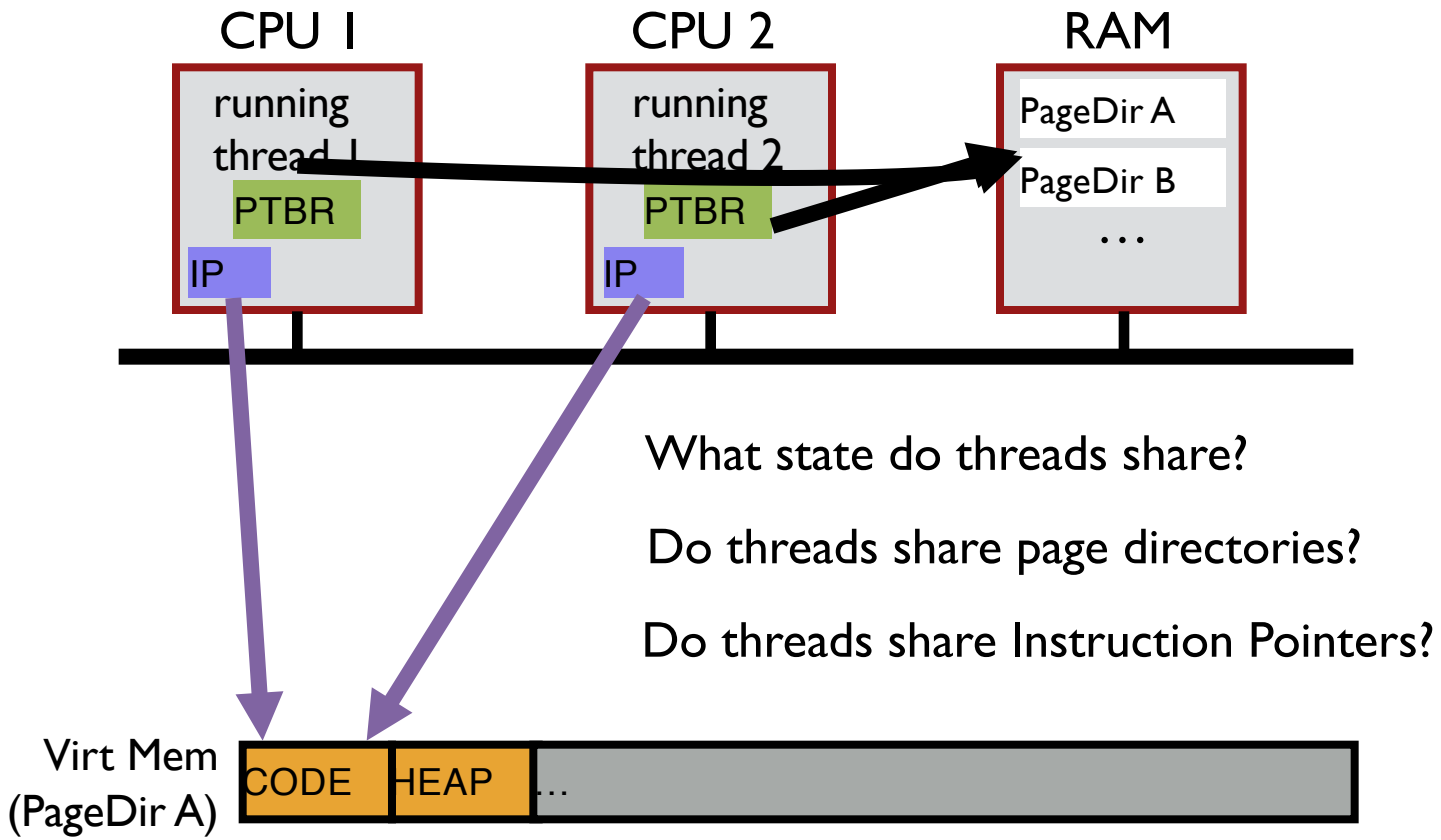
COMMON PROGRAMMING MODELS

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

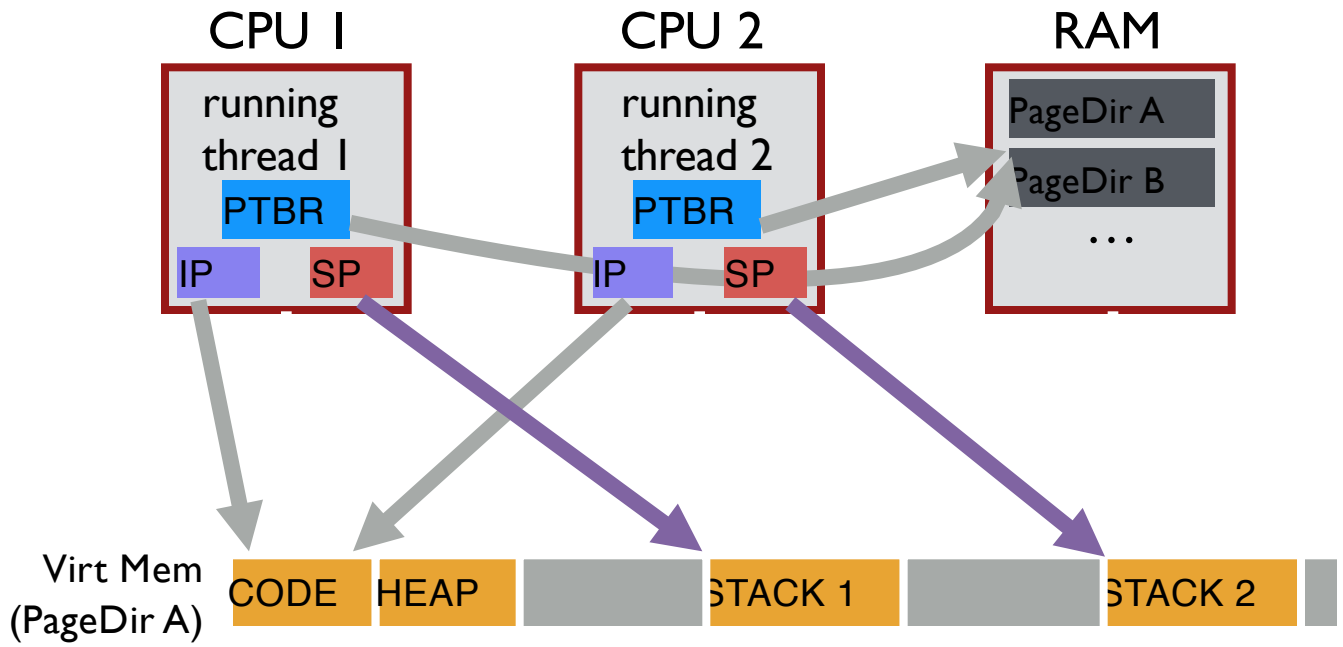
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread

- **Defer work with background thread**
  One thread performs non-critical work in the background
  (e.g., when CPU idle)



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

(But, stacks are in same address space, so trusted to be cooperative)

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 (environment variables)
- User and group id (permissions, limits)

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)

THREAD API

Variety of thread systems exist

- POSIX Pthreads

Common thread operations

- Create
- Exit
- Join (instead of wait() for processes)

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
- Can tune scheduling policy to meet application demands (simplify)
- 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

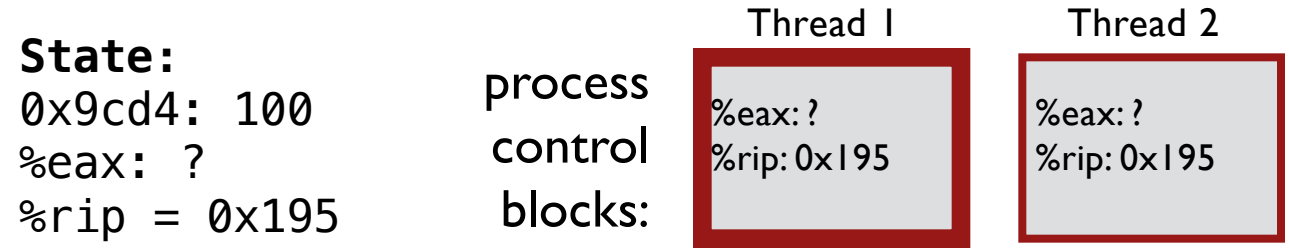
THREADS DEMO

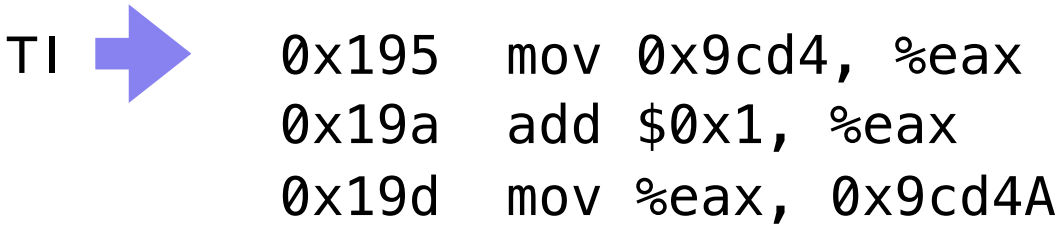
THREAD SCHEDULE #1

```
balance = balance + 1; balance at 0x9cd4
```

Registers are virtualized by OS;
Each thread thinks it has own

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



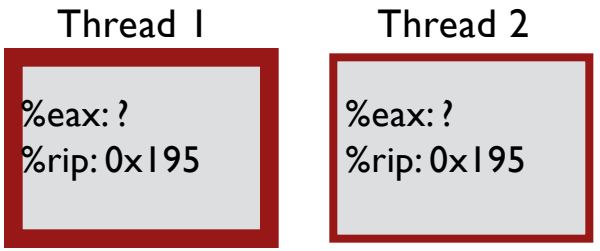


What is state after instruction 0x195 completes?

THREAD SCHEDULE #1

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

process control blocks:



T1 

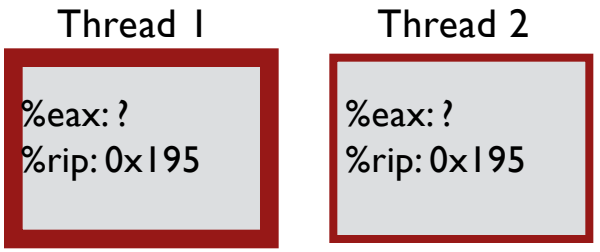
```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

What is state after instruction 0x19a completes?

THREAD SCHEDULE #1

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

process control blocks:



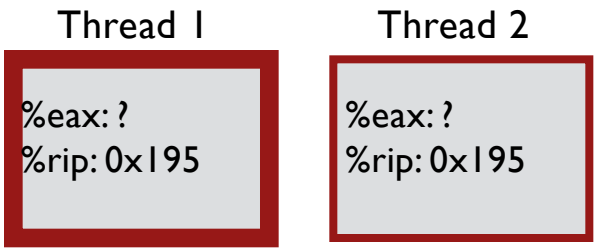

```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
TI      0x19d   mov %eax, 0x9cd4A
```

What is state after instruction 0x19d completes?

THREAD SCHEDULE #1

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

process control blocks:




```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

TI

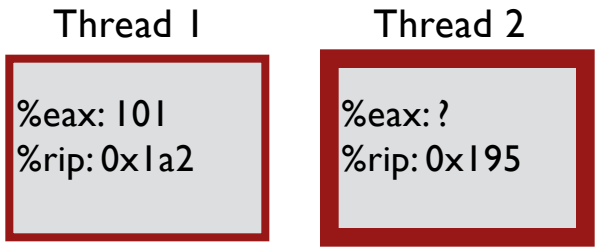
Thread Context Switch

New contents of PCB and %eax and %rip?

THREAD SCHEDULE #1

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

process control blocks:



T2

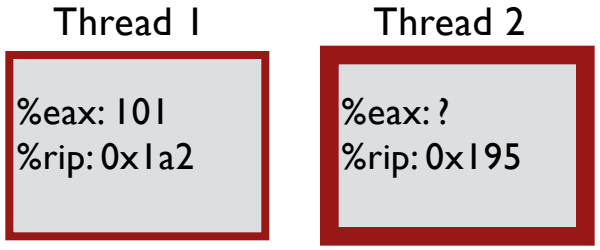
```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

What is state after instruction 0x195 completes?

THREAD SCHEDULE #1

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

process control blocks:



T2 

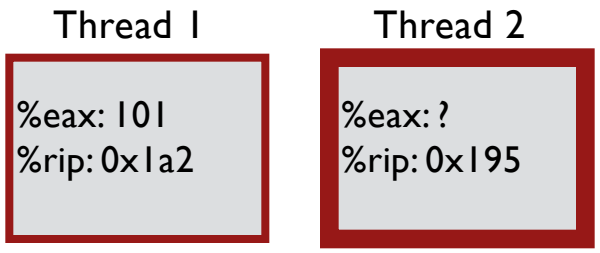
```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

What is state after instruction 0x19a completes?

THREAD SCHEDULE #1

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

process control blocks:



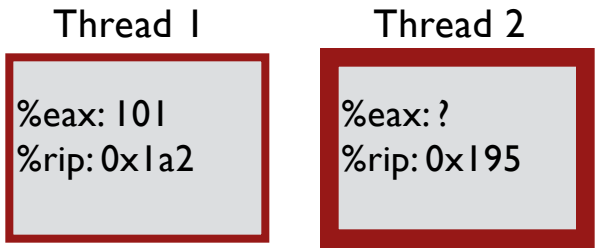

```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T2      0x19d   mov %eax, 0x9cd4A
```

What is state after instruction 0x19d completes?

THREAD SCHEDULE #1

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

process control blocks:




```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

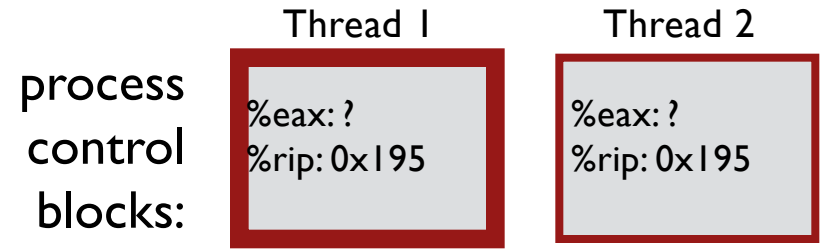
T2 ➡

Desired Result!

ANOTHER SCHEDULE

THREAD SCHEDULE #2

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



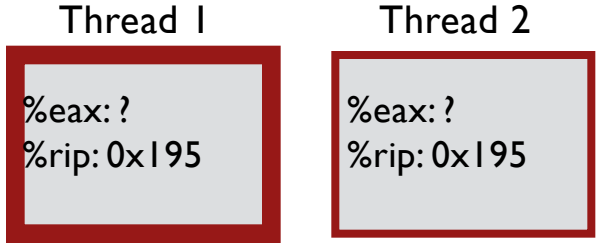
T1

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

THREAD SCHEDULE #2

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

process control blocks:



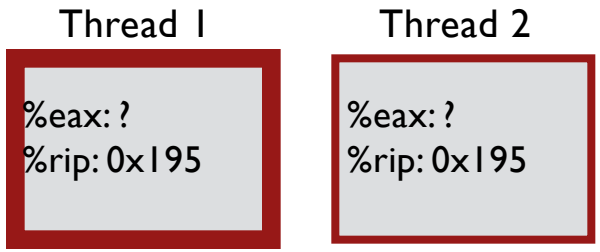
TI

```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

THREAD SCHEDULE #2

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

process control blocks:




```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T1      0x19d   mov %eax, 0x9cd4A
```

Thread Context Switch before T1 executes 0x19d

THREAD SCHEDULE #2

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

process control blocks:

Thread 1

%eax: 101
%rip: 0x19d

Thread 2

%eax: ?
%rip: 0x195

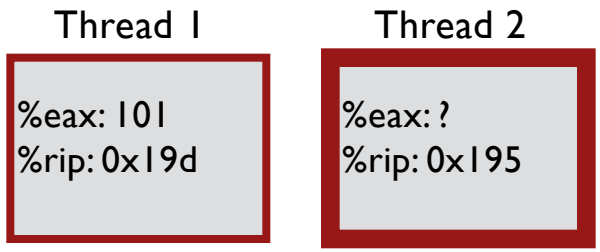
T2 →

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

THREAD SCHEDULE #2

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

process control blocks:



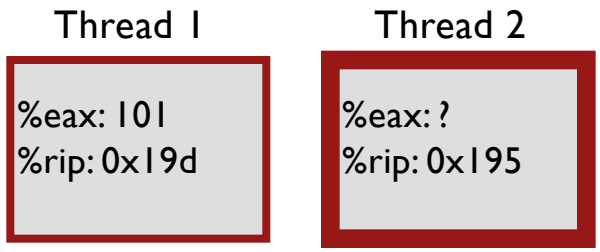
T2 

```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

THREAD SCHEDULE #2

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

process control blocks:




```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T2      0x19d   mov %eax, 0x9cd4A
```

THREAD SCHEDULE #2

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

process control blocks:

Thread 1
%eax: 101
%rip: 0x19d

Thread 2
%eax: ?
%rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

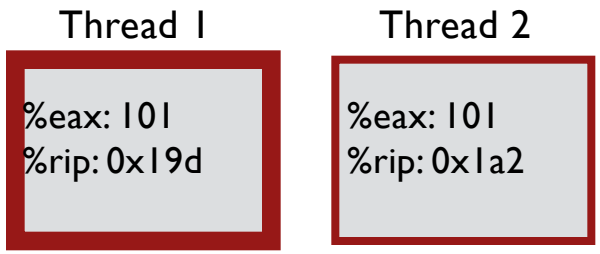
T2 

Thread Context Switch back to T1

THREAD SCHEDULE #2

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

process control blocks:

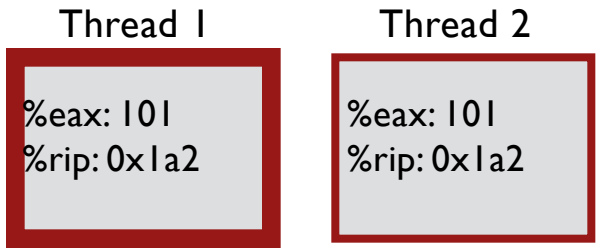



```
        0x195   mov 0x9cd4, %eax
        0x19a   add $0x1, %eax
T1      0x19d   mov %eax, 0x9cd4A
```

THREAD SCHEDULE #2

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

process control blocks:



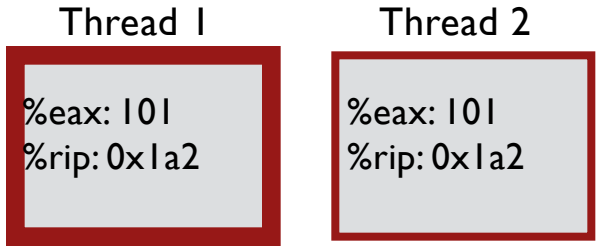

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T1 

THREAD SCHEDULE #2

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

process control blocks:




```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

T1

WRONG Result! Final value of balance is 101

TIMELINE VIEW

Thread 1	**Thread 2**	Balance	T1-eax	T2-eax
mov 0x123, %eax		0	0	
add %0x1, %eax			1	
mov %eax, 0x123		1		
	mov 0x123, %eax			1
	add %0x2, %eax			3
	mov %eax, 0x123	3		

How much is added to shared variable? 3: correct!

TIMELINE VIEW

Thread 1	**Thread 2**	Balance	T1-eax	T2-eax
		0	0	
mov 0x123, %eax			1	
add %0x1, %eax				
	mov 0x123, %eax	1		0
mov %eax, 0x123				
	add %0x2, %eax			2
	mov %eax, 0x123	2		

How much is added?

2: incorrect!

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1-eax	T2-eax
		0		0
mov 0x123, %eax			0	
	mov 0x123, %eax			0
add %0x1, %eax			1	
	add %0x2, %eax			2
mov %eax, 0x123		1		
	mov %eax, 0x123	2		

How much is added?

1: incorrect!

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1-eax	T2-eax
		0		0
	mov 0x123, %eax			
	add %0x2, %eax			2
	mov %eax, 0x123	2		
mov 0x123, %eax			2	
add %0x1, %eax			3	
mov %eax, 0x123		3		

How much is added? 3: correct!

TIMELINE VIEW

Thread 1	Thread 2	Balance	T1-eax	T2-eax
	mov 0x123, %eax	0		0
	add %0x2, %eax			2
mov 0x123, %eax			0	
add %0x1, %eax			1	
mov %eax, 0x123		1		
	mov %eax, 0x123	2		

How much is added? 2: incorrect!

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)

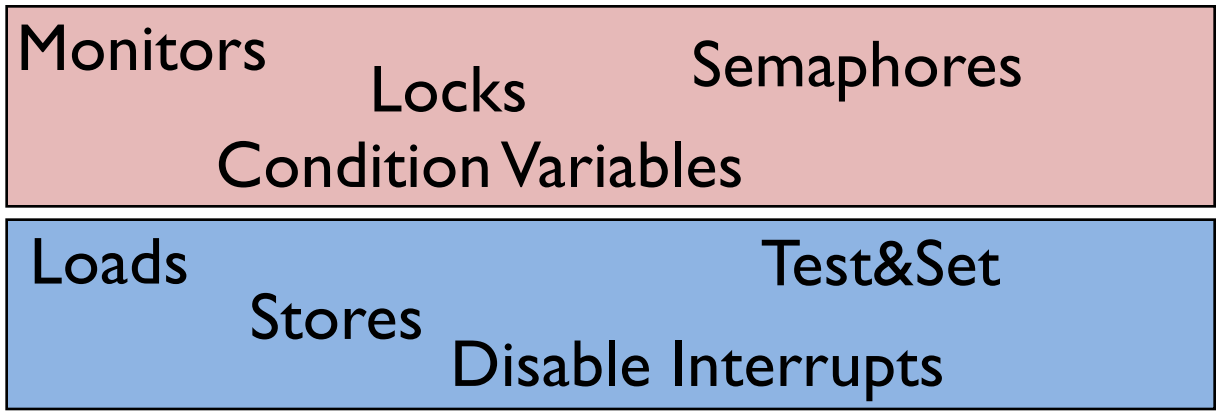
SYNCHRONIZATION

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



CONCURRENCY SUMMARY

Concurrency is needed for high performance when using 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

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);

THREADS DEMO2