CS5460: Operating Systems

Lecture 13: Threads

(Chapters 26, 27)

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

Thrashing

- **Working set**: collection of memory currently being used by a process
- If all working sets do not fit in memory → thrashing
  - One “hot” page replaces another
  - Percentage of accesses that generate page faults skyrockets
- Typical solution: “swap out” entire processes
  - Scheduler needs to get involved
  - Two-level scheduling policy → runnable vs memory-available
  - Need to be fair
  - Invoked when page fault rate exceeds some bound
- When swap devices are full,
  Linux invokes the “OOM killer”

Frame Allocation

- Who should we compete against for memory?
- **Global replacement**:
  - All pages for all processes come from single shared pool
  - Advantage: very flexible → can globally "optimize" memory usage
  - Disadvantages: thrashing more likely, can often do just the wrong thing (e.g., replace the pages of a process about to be scheduled)
  - Many OSes, including Linux, do this
- **Per-process replacement**:
  - Each process has private pool of pages → competes with itself
  - Alleviates inter-process problems, but not every process equal
  - Need to know working set size for each process
  - Windows has calls to set process's min/max working set sizes

fork(), Copy-on-Write, & Laziness

- Copy-on-write: initially use shared pages for parent and child to share memory
  - On fork, child gets a copy of parent's page tables
  - (Re-)mark all pages read-only even if child/parent has write permissions
  - On write, trap, copy the page, record new location in page table, restart operation
- Parent/child share memory, unless one of them modifies memory contents after fork()
- Insight: much of parent/child address space remains unchanged after fork()
  - Saves space and work

Demand Zeroing

- Page frames cannot be reused directly
  - May contain sensitive data!
- OS zeroes pages before (re-)mapping them
- Can be lazy
  - Only zero a page frame when process accesses the memory
  - Even lazier: map same read-only zero page and use COW

mmap()

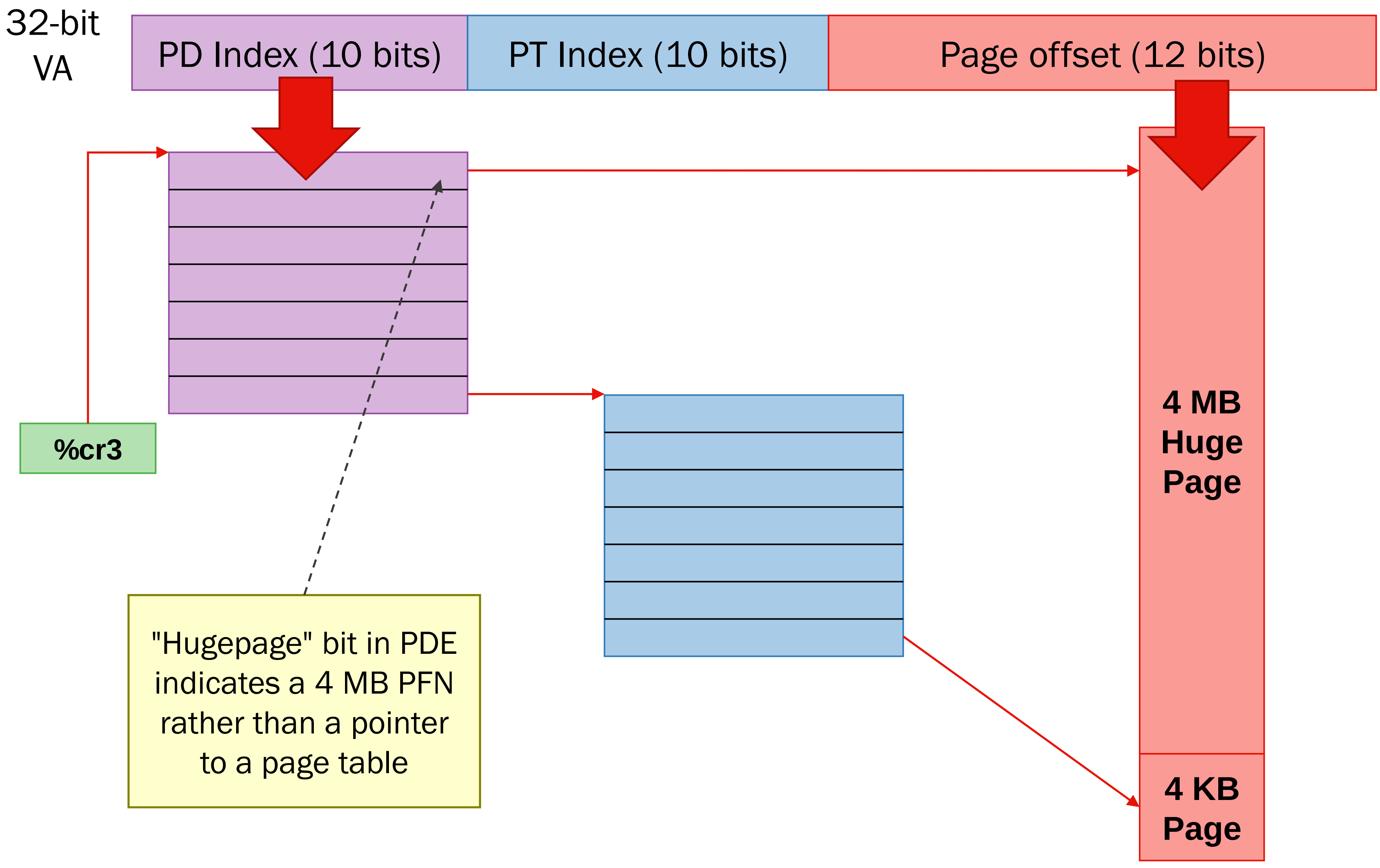
- System call to manipulate address space
- Map a file for demand paging
  - Can treat file as a big byte array
  - Other processes can map too to share state
- Map anonymous pages to add heap space
  - Can map regions larger than memory (how?)
  - Modern malloc() uses this instead of sbrk()
- Map pages that can be shared with children
  - On fork(), mappings copied without COW protection

Hugepages/Superpages

- **Problem**: TLB reach shrinking as % of memory size
- **Solution**: Hugepages
  - Permit (some) larger pages
  - For simplicity, restrict generality:
    - Same "coverage" as higher levels of multi-level page tables
    - Aligned to huge page size (e.g., 2 MB page aligned on 2 MB bdy)
    - Contiguous

- **Problem**: Restrictions limit applicability. How?

Example: Hugepage Usage



Hugepage Discussion

- What are good candidates for hugepages?
  - Kernel – or at least the portions of kernel that are not “paged”
  - Frame buffer
  - Large “wired” data structures
    - Scientific applications being run in “batch” mode
    - In-core databases
- How might OS exploit hugepages?
  - **Simple**: Few hardwired regions (e.g., kernel and frame buffer)
  - **Improved**: Provide system calls so applications can request it
  - **Holy grail**: OS watches page access behavior and determines which pages are “hot” enough to warrant hugepages
- Why might you **not** want to use hugepages?
- 32-bit Intel: 4 KB pages with 4 MB hugepages
- 64-bit Intel: 4 KB pages with 2 MB and 1 GB hugepages

Conclusions

Illusion of virtual memory:
Processes can run when sum of virtual address spaces is more than amount of physical memory

Mechanism:

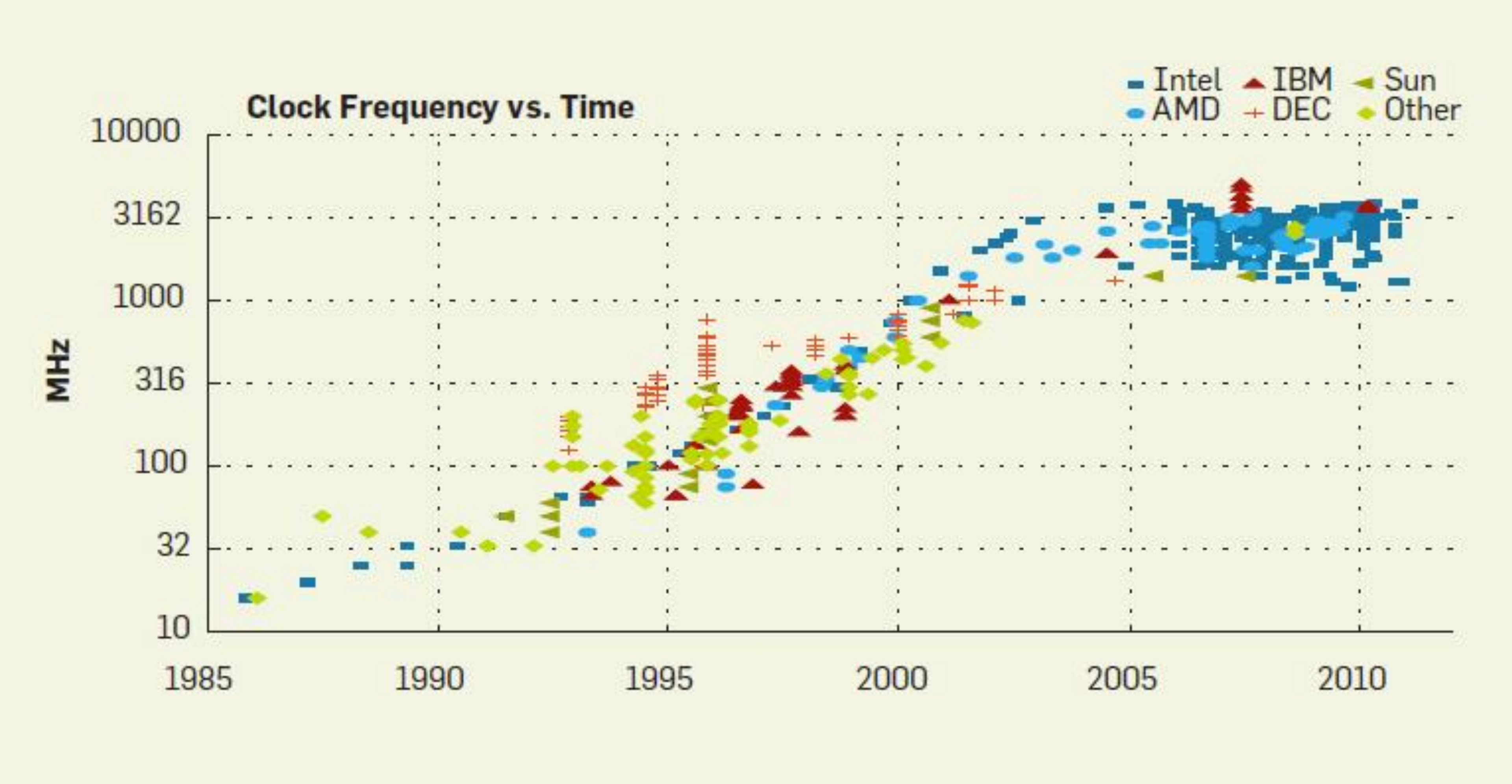
- Use page table "present" bit
- OS handles page faults (or page misses) by reading in desired page from disk

Policy:

- Page selection – demand paging, prefetching, hints
- Page replacement – OPT, FIFO, LRU, others

Implementations (clock) perform approximation of LRU

Motivation for Concurrency



http://cacm.acm.org/magazines/2012/4/147359-cpu-db-recording-microprocessor-history/fulltext

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

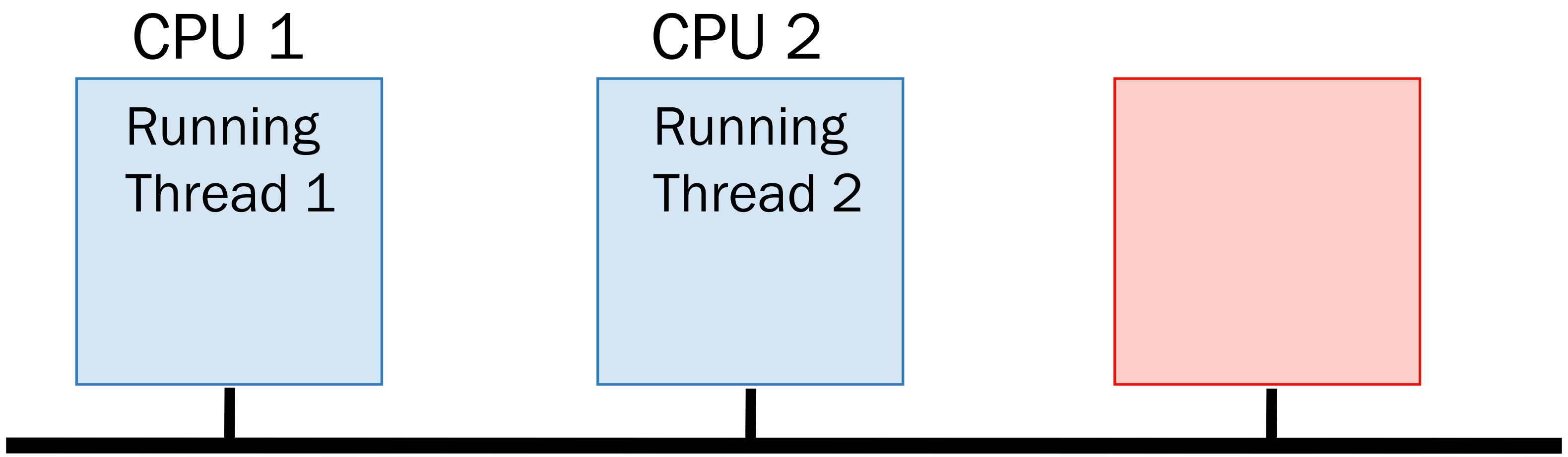
Option 2: Threads

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

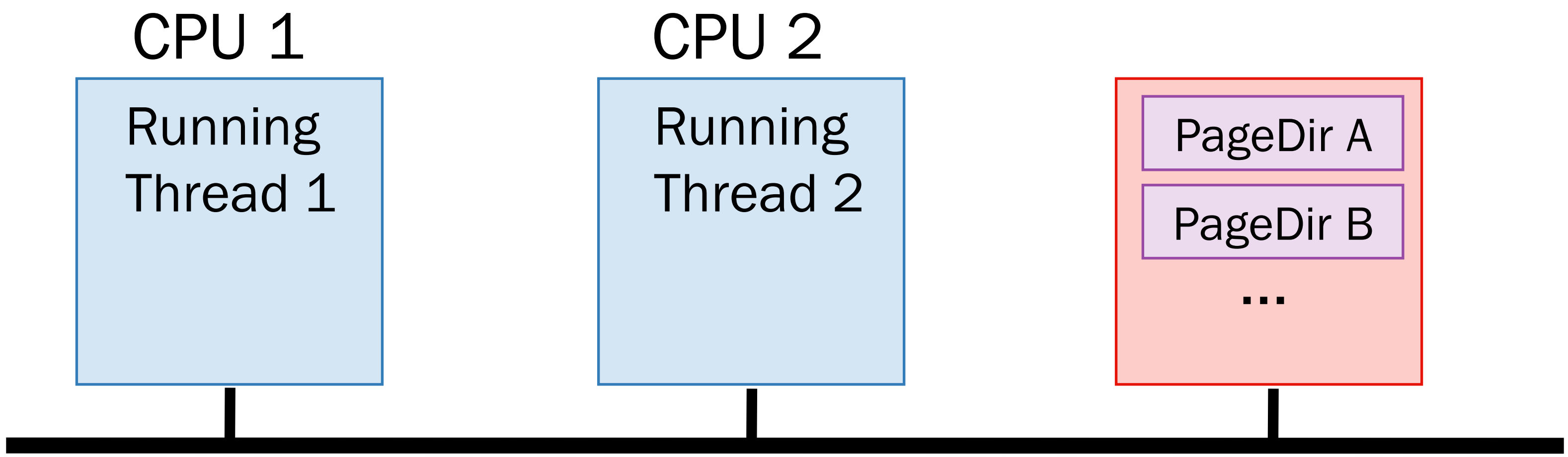
- Divide
  - large task across several cooperative threads
  - many small concurrent tasks across threads
- Communicate through shared address space

Common Programming Models

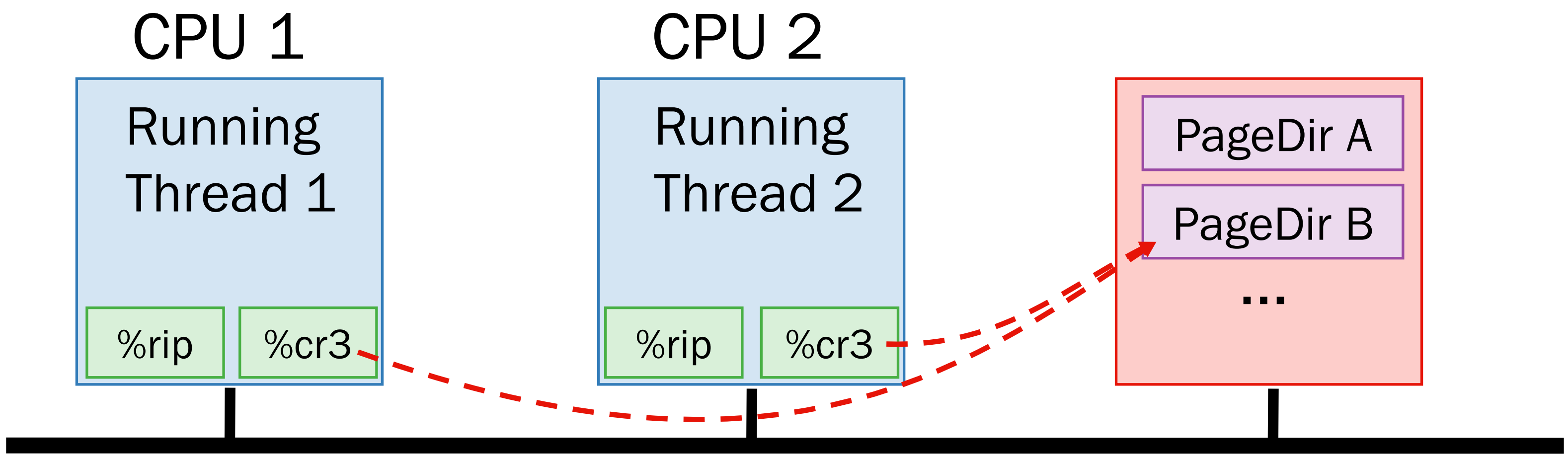
- Multi-threaded programs structured as:
  - **Producer/consumer**
    Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
  - **Pipeline**
    Task is divided into series of subtasks, each of which is handled in series by a different thread
  - **Defer work with background thread**
    One thread performs non-critical work in the background (when CPU idle)



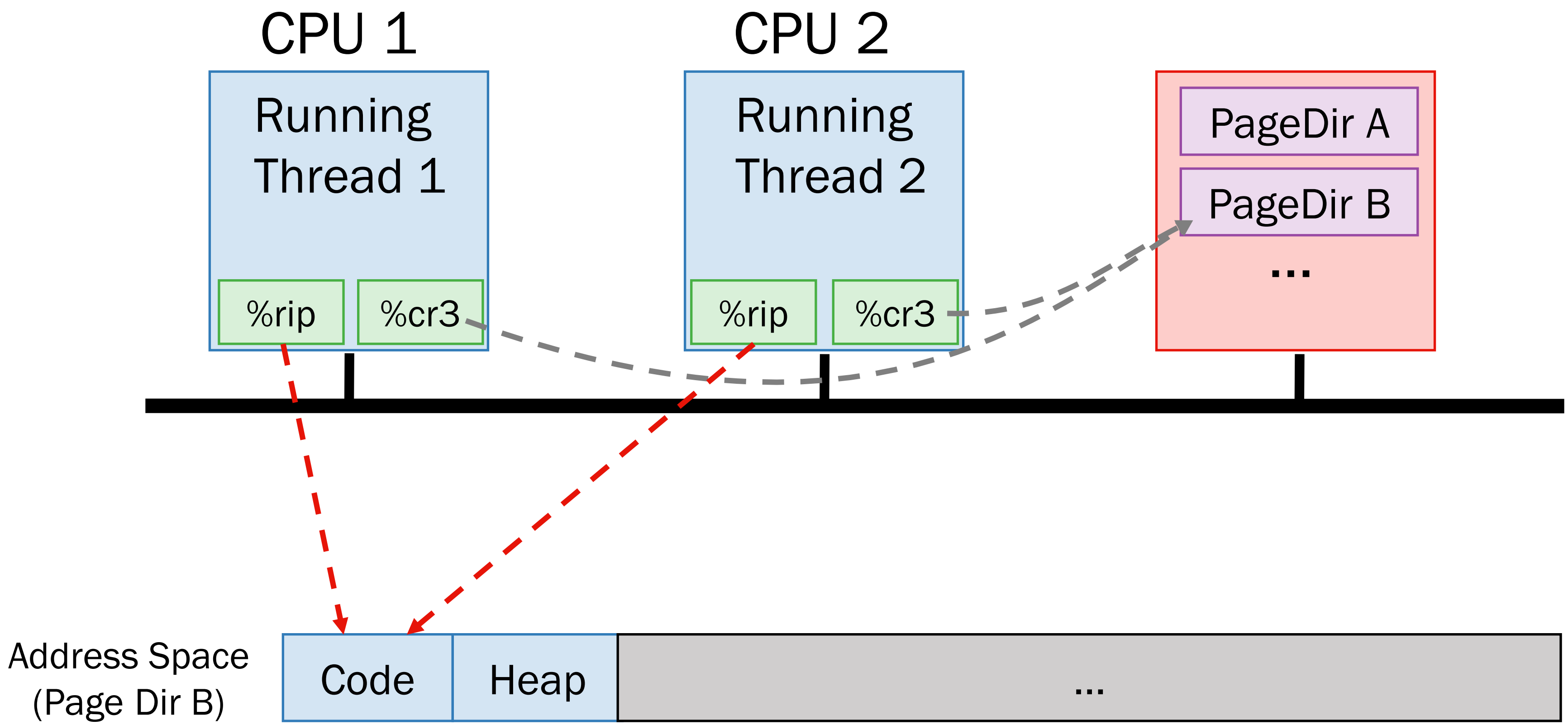
What state do threads share?



What threads share page directories?

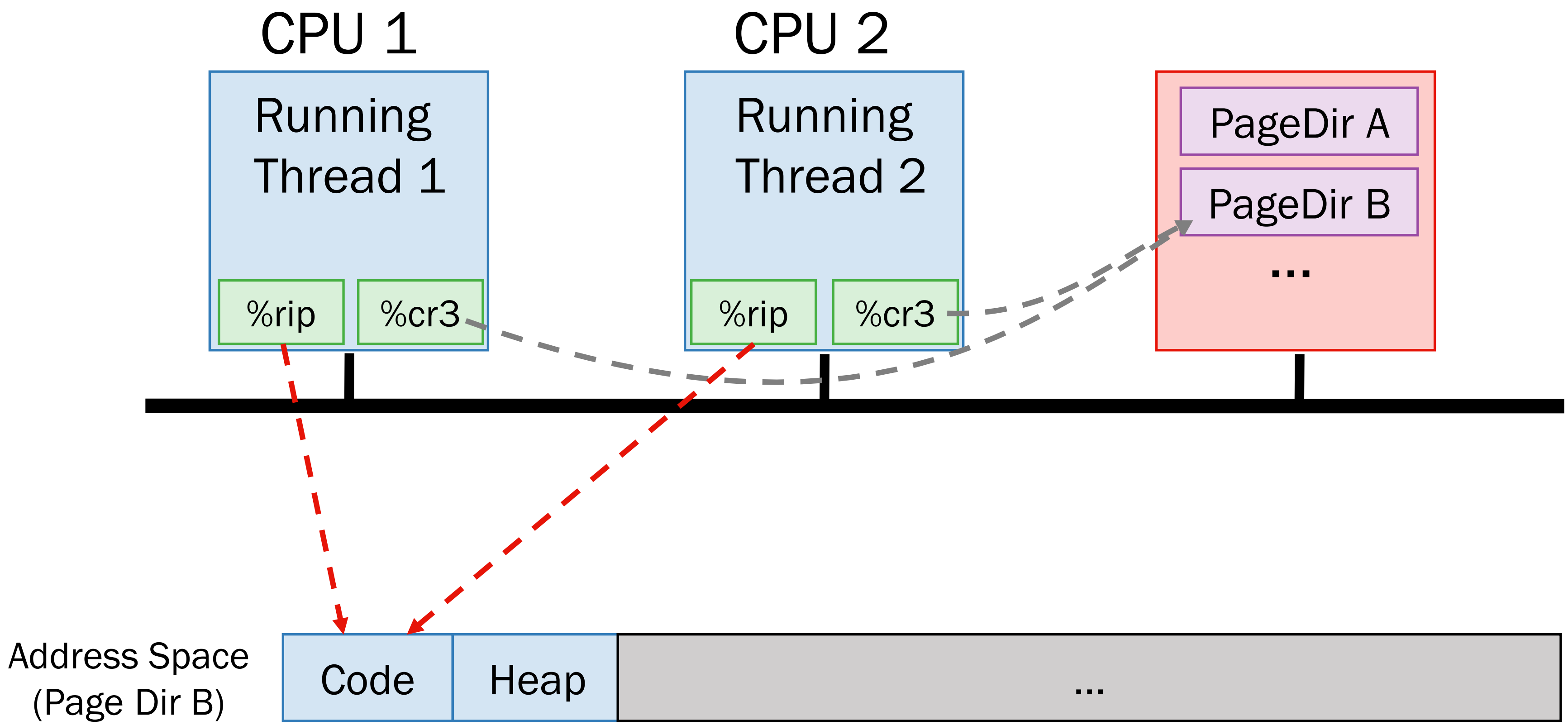


Do threads share Instruction Pointer?

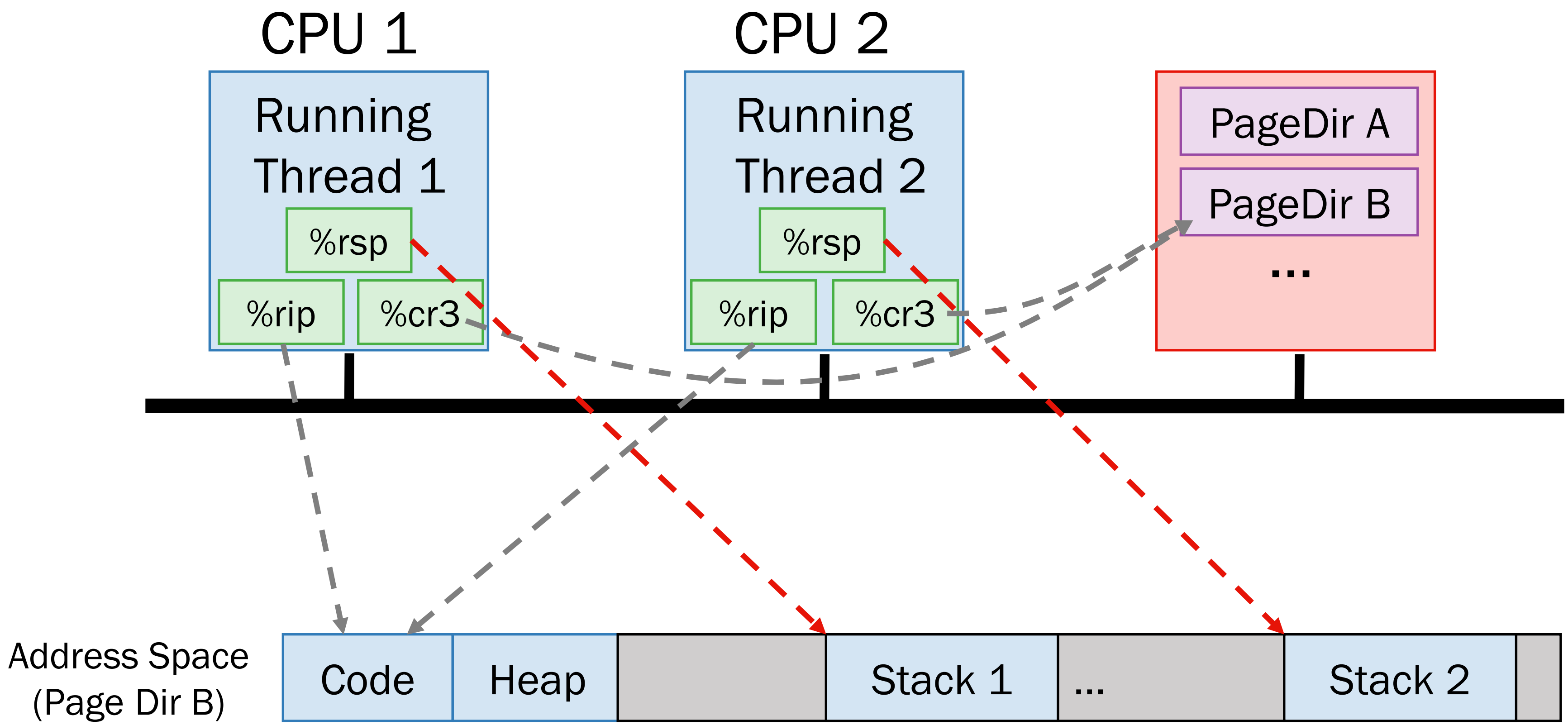


Share code, but each thread may be executing different code at the same time

→ 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

☠
Please don't write code like this

```
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);
}
```

```
$ ./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	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

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

Thread 2
%eax: ?
%rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9c
```

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	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	?
%rip	0x1a2	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 %rip: 0x1a2	%eax: ? %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	%eax: ?
%rip: 0x1a2	%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

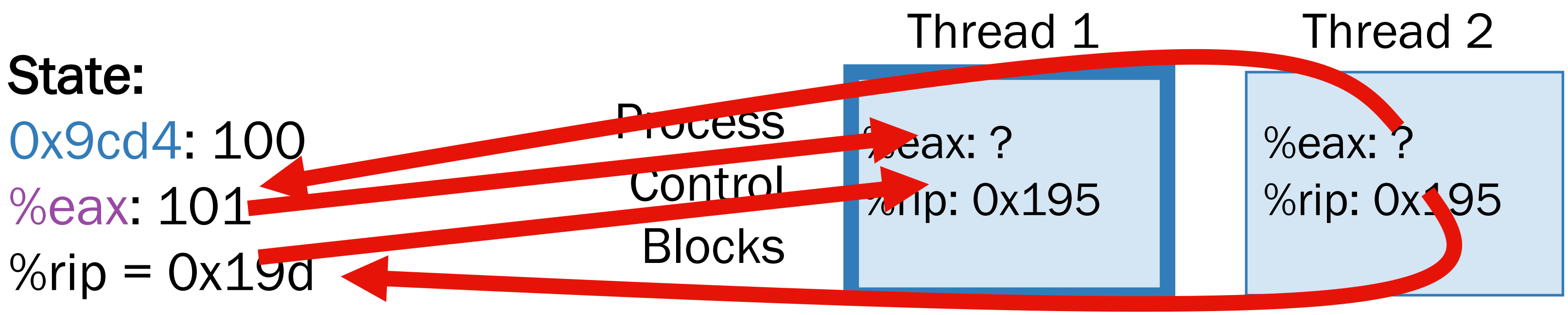
State:
0x9cd4: 100
%eax: 100
%rip = 0x19a

	Thread 1	Thread 2
Process Control Blocks	%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



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	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 %rip: 0x19d	%eax: ? %rip: 0x195

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

T2

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, 0x9cd

T2

Context Switch

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	%eax: 101
%rip: 0x19d	%rip: 0x1a2

Unexpected result!

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

T1

Timeline View

Thread 1

mov 0x123, %eax

add %0x1, %eax

mov %eax, 0x123

Thread 2

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

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);`**

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