CPSC 410

Keith Perkins

Content from OSTEP

Concurrency: Threads

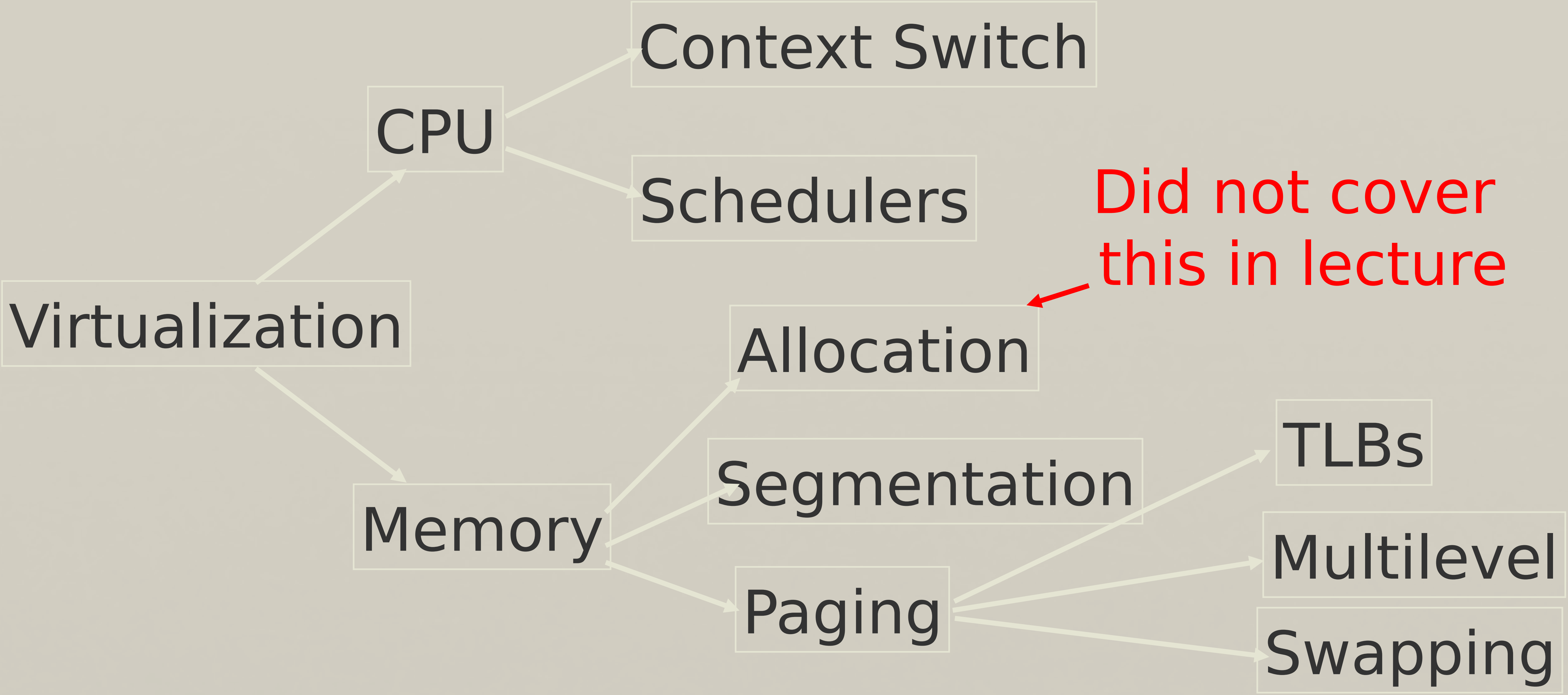
Questions answered in this lecture:

Why is concurrency useful?

What is a thread and how does it differ from processes?

What can go wrong if scheduling of critical sections is not atomic?

First Part of Course



Virtualization

CPU

Context Switch

Schedulers

Memory

Allocation

Segmentation

Paging

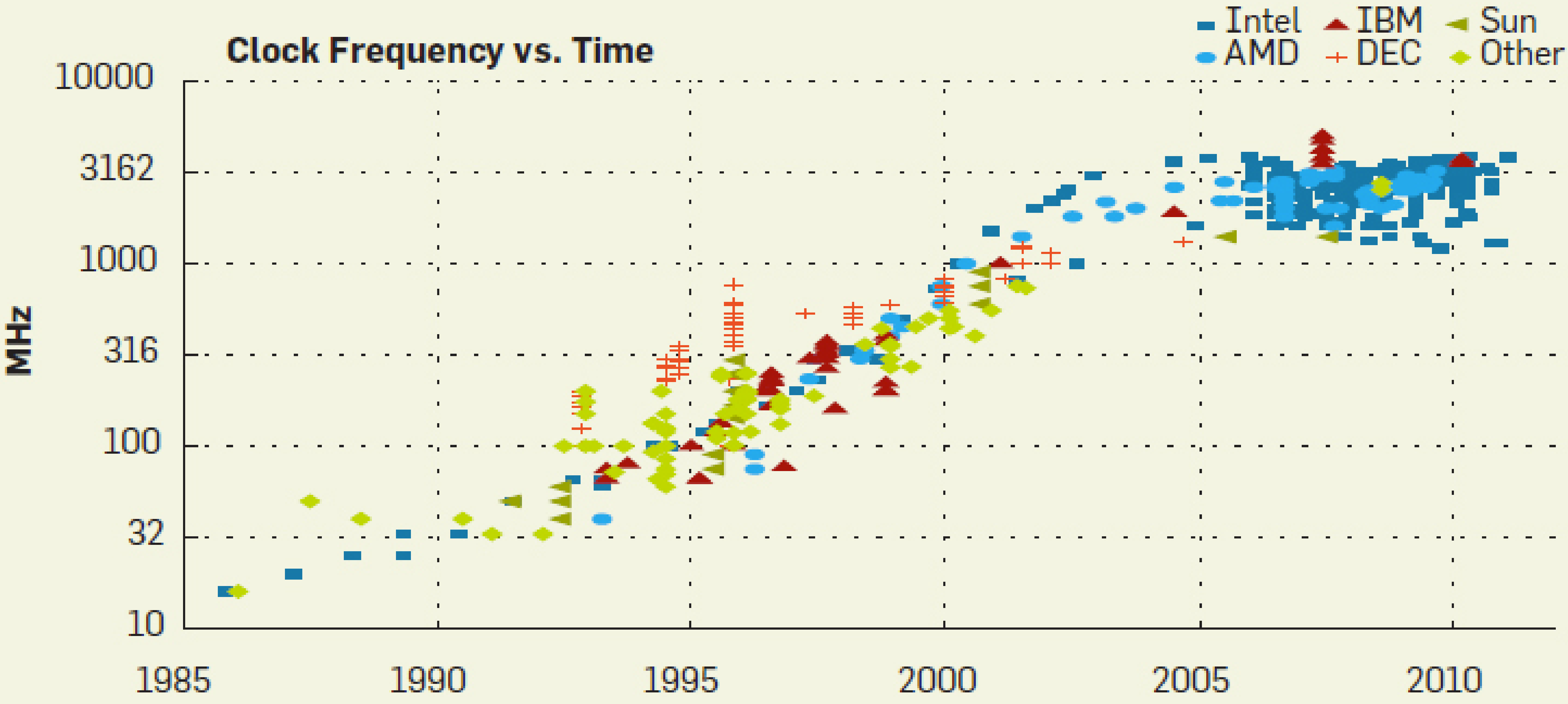
TLBs

Multilevel

Swapping

Did not cover this in lecture

Motivation for Concurrency



Motivation

CPU Trend: Same speed, but multiple cores

Goal: Write applications that fully utilize many cores

Option 1: Build apps from many 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 (why expensive?)

CONCURRENCY: Option 2

New abstraction: **thread**

Threads are like processes, except:
multiple threads of same process share an address space

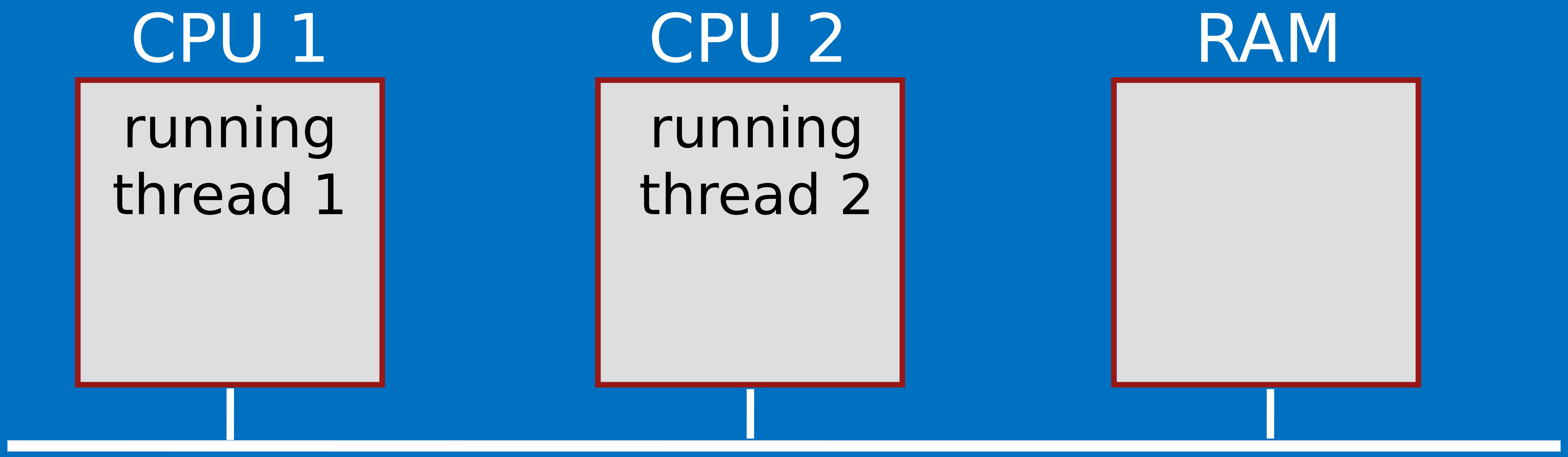
Divide large task across several cooperative threads

Communicate through shared address space

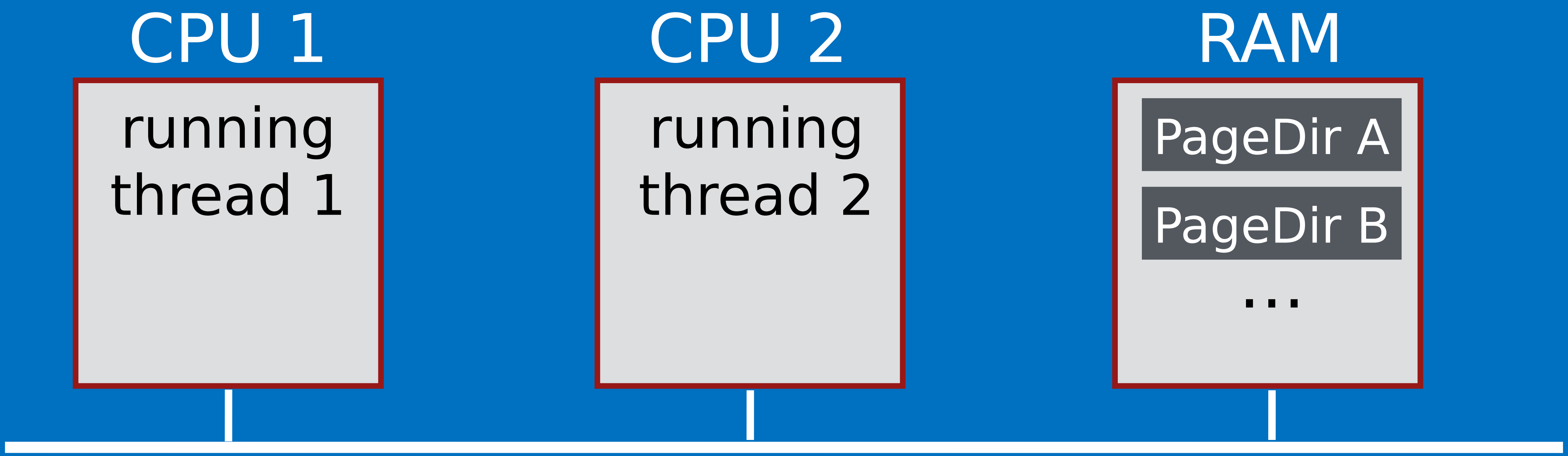
Common Programming Models

Multi-threaded programs tend to be structured as:

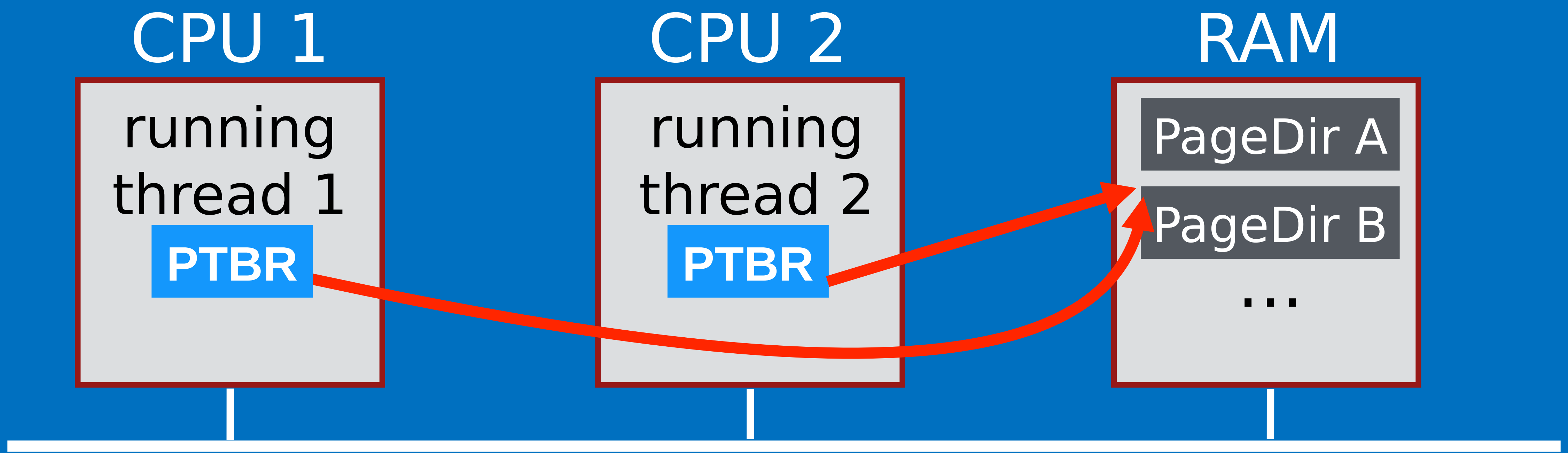
- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread**
  One thread performs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



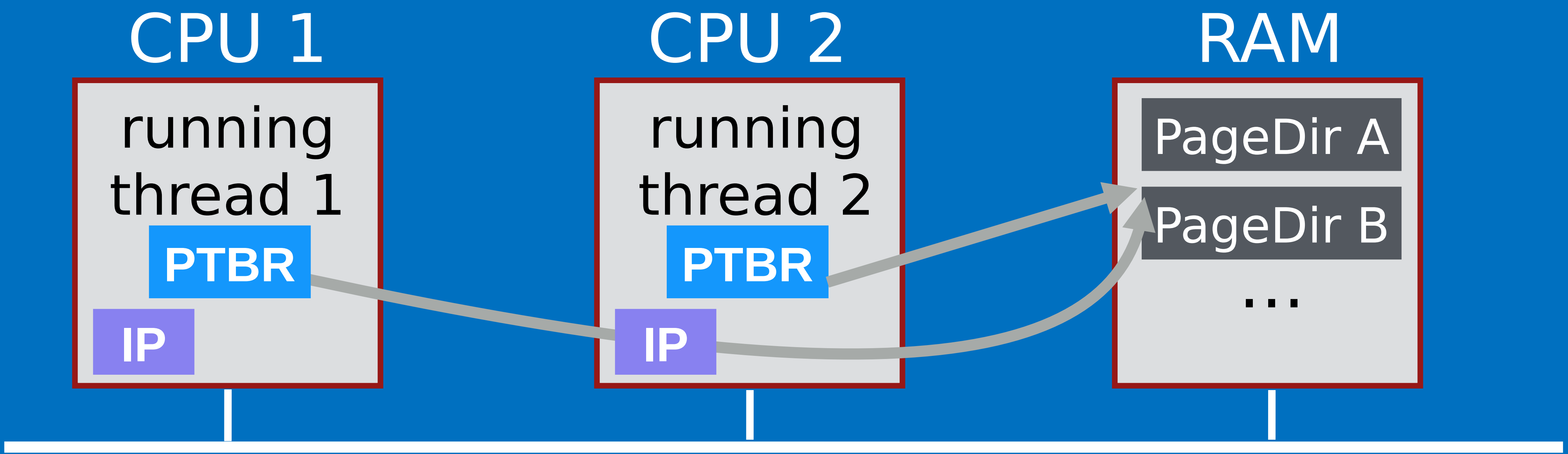
What state do threads share?



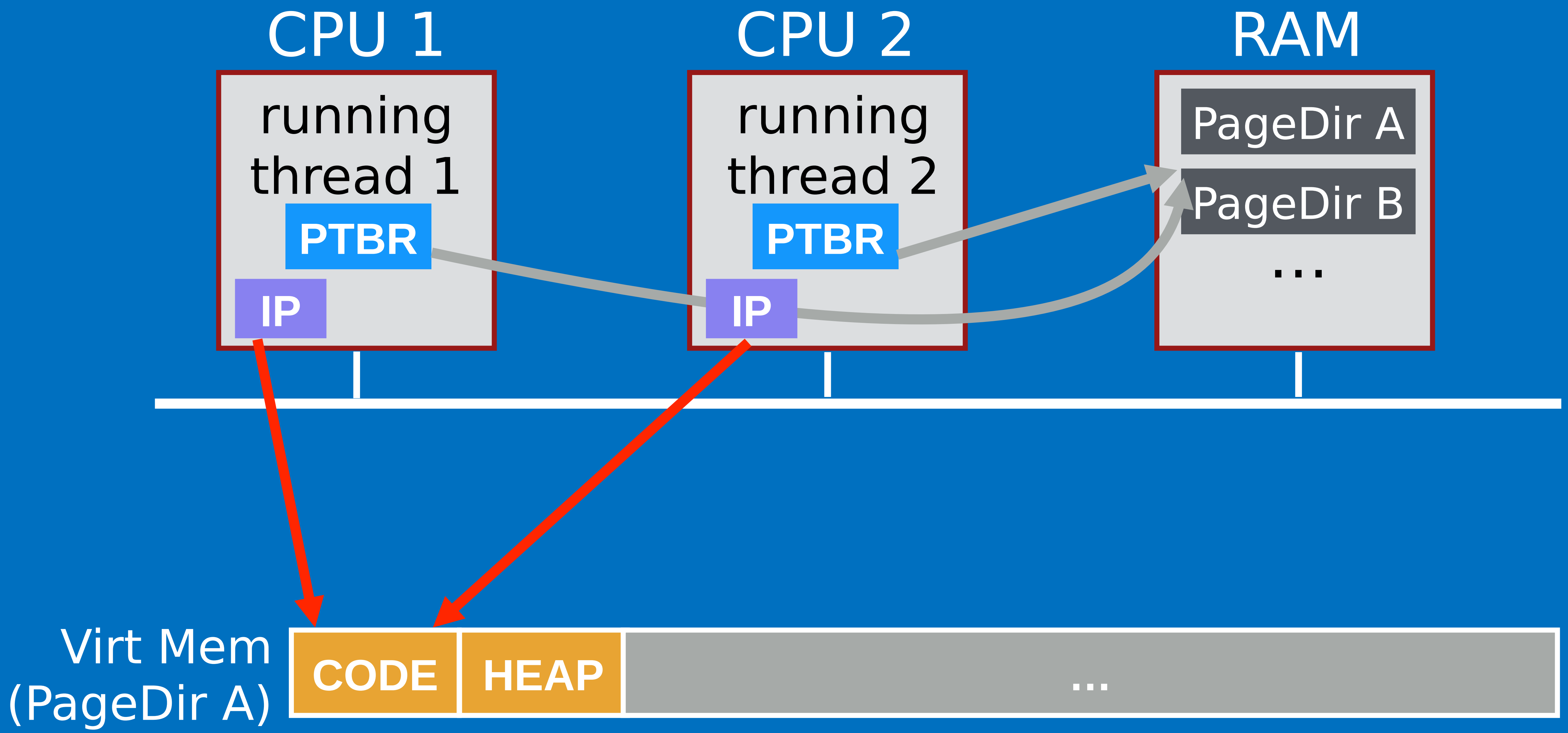
What state do threads share?



They share the page table of their host process



Do threads share Instruction Pointer?



CPU 1

running thread 1

PTBR

IP

CPU 2

running thread 2

PTBR

IP

RAM

PageDir A

PageDir B

...

Virt Mem (PageDir A)

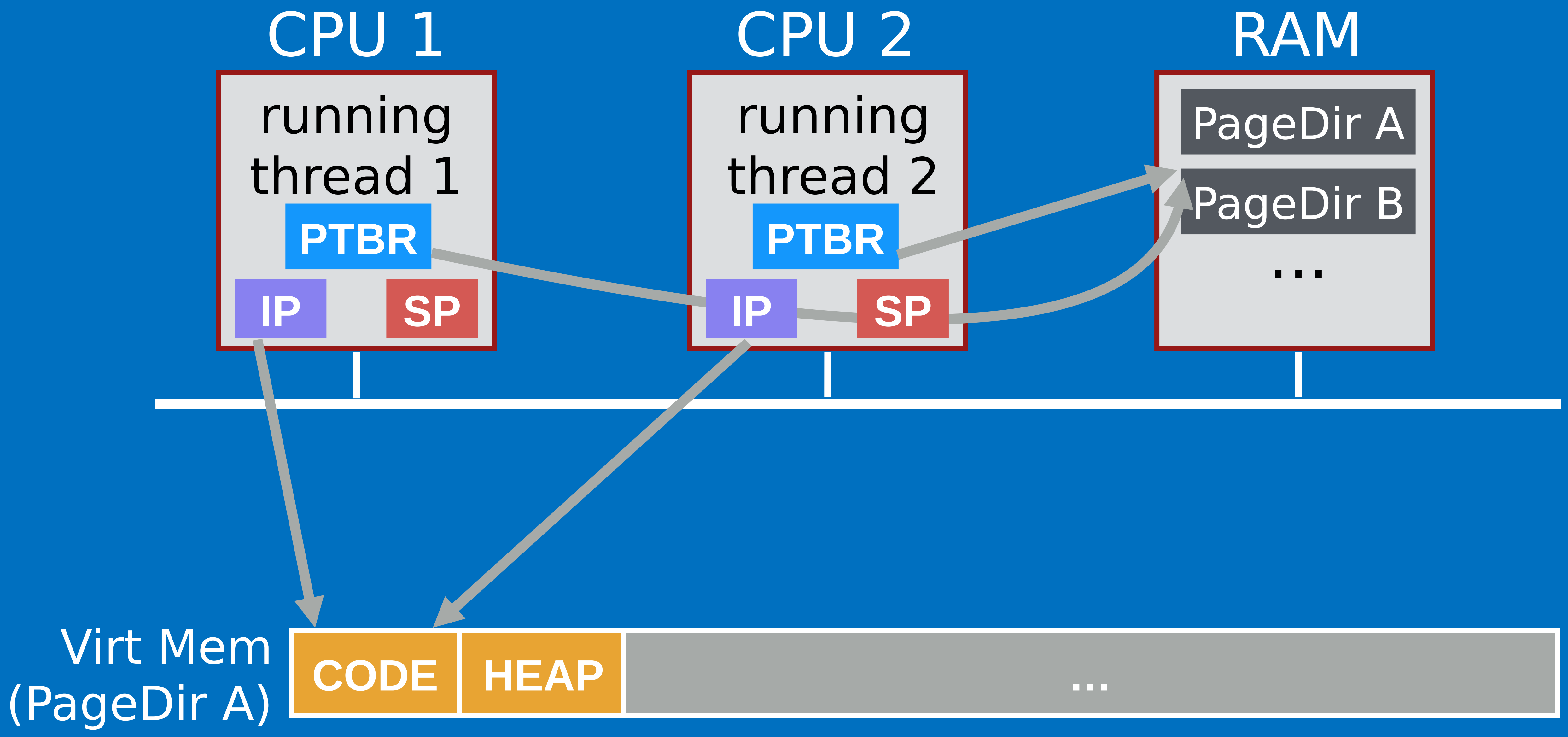
CODE

HEAP

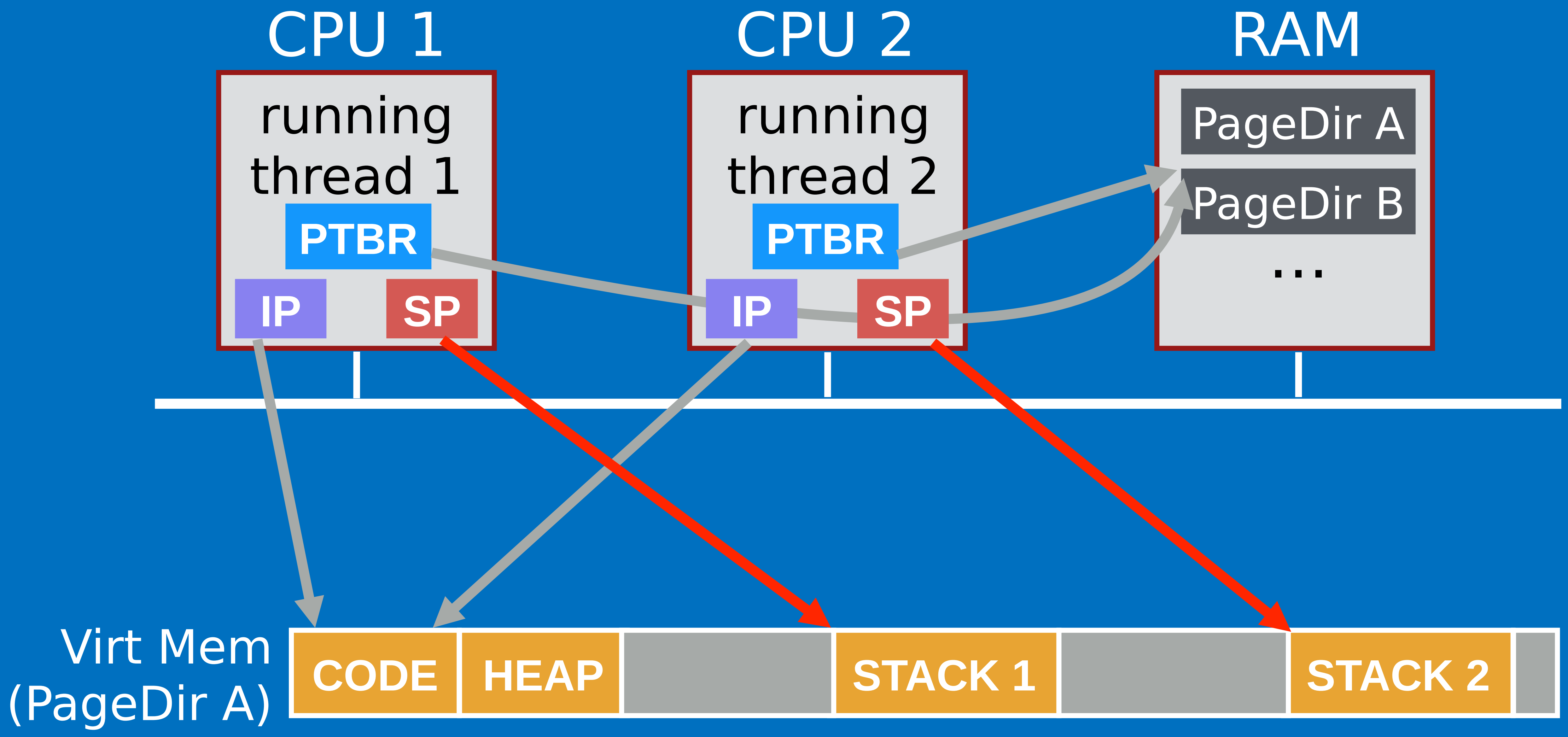
...

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

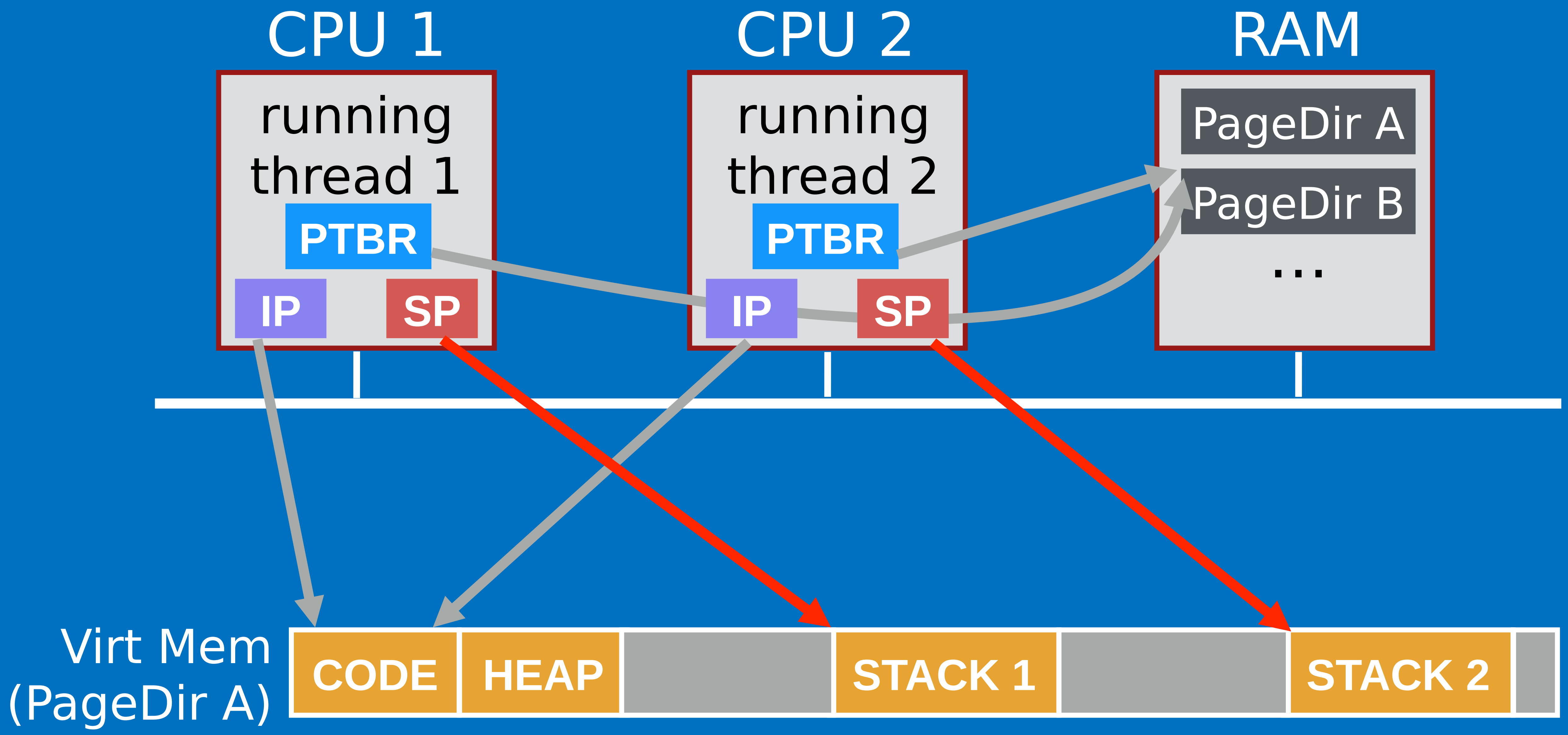
So...Different Instruction Pointers



Do threads share stack pointer?



Do threads share stack pointer?



Do threads share stack pointer?

No..threads executing different functions need different stacks

THREAD VS. Process

Multiple threads within a single process share:

- Thread Group ID (TGID)
- Address space
  - Code (instructions)
  - Most data (heap)
- Open file descriptors
- Current working directory
- User and group id

Each thread has its own

- Process ID (PID)
- 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
- Lower overhead thread operations since no system call

Disadvantages?

- Cannot leverage multiprocessors
- If one thread blocks they all block

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

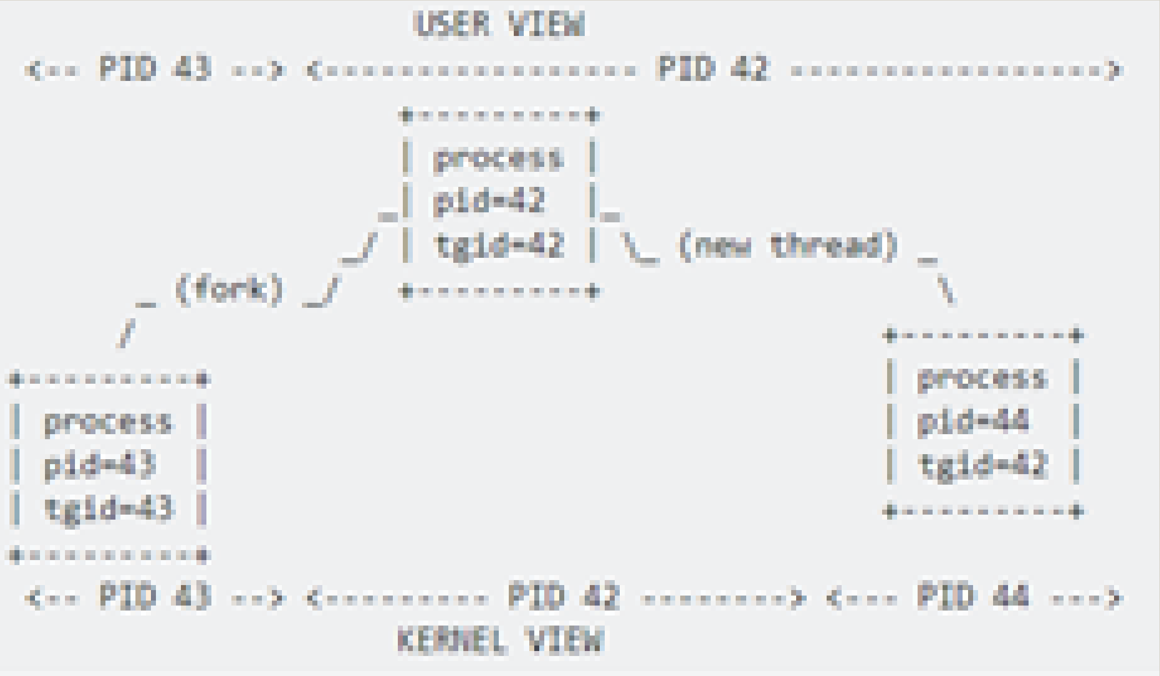
Advantages

- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Aside: Linux Processes and Threads

Linux treats threads as processes

- Parent process pid==tgid
- Launched threads have new pid and **Parents** tgid
- So scheduler sees different pids (for scheduling)
- But same tgid means don't swap memory if swapping to same tgid



```
                         USER VIEW
 <-- PID 43 --> <----------------- PID 42 ----------------->
                     +---------+
                     | process |
                    _| pid=42  |_
                  _/ | tgid=42 | \_ (new thread) _
       _ (fork) _/   +---------+                  \
      /                                        +---------+
+---------+                                    | process |
| process |                                    | pid=44  |
| pid=43  |                                    | tgid=42 |
| tgid=43 |                                    +---------+
+---------+
 <-- PID 43 --> <--------- PID 42 --------> <--- PID 44 --->
                     KERNEL VIEW
```

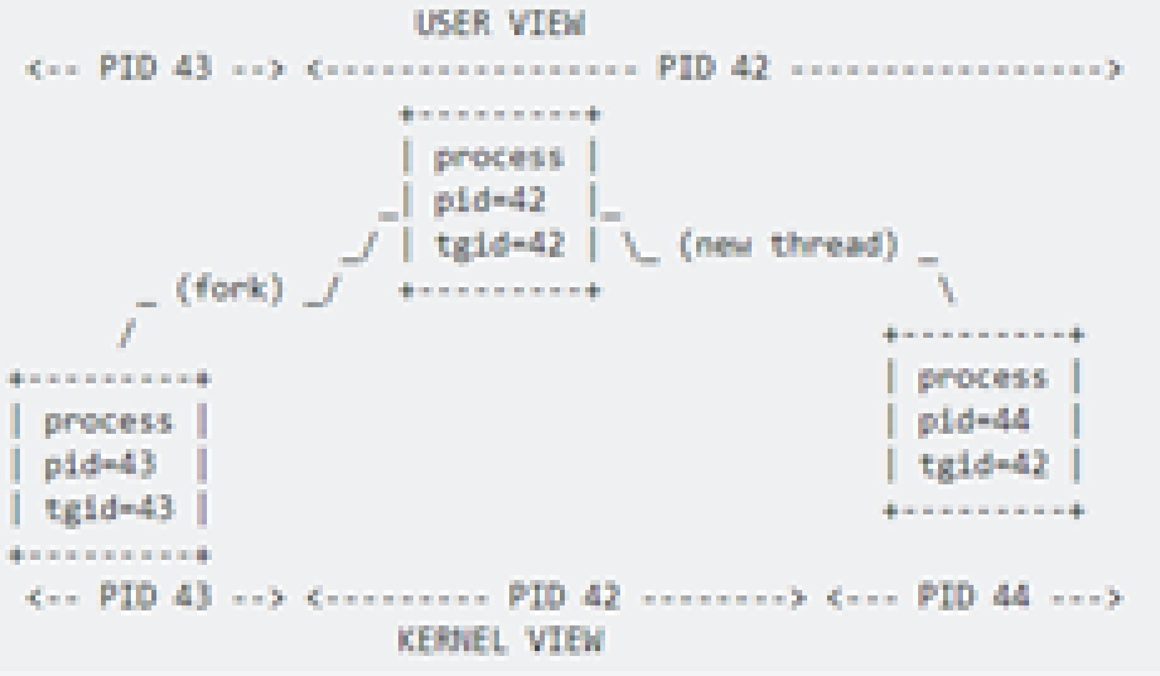
Graphic from https://stackoverflow.com/questions/9305992/if-threads-share-the-same-pid-how-can-they-be-identified

Aside: HTOP

Also means you can just show processes (not their internal threads) by displaying tgid only

Show how HTOP tracks processes and threads

- F2 to setup columns (PID, TGID)
- Show tree view (Display options->to see parent process



```
                         USER VIEW
 <-- PID 43 --> <----------------- PID 42 ----------------->
                     +---------+
                     | process |
                    _| pid=42  |_
                  _/ | tgid=42 | \_ (new thread) _
       _ (fork) _/   +---------+                  \
      /                                        +---------+
+---------+                                    | process |
| process |                                    | pid=44  |
| pid=43  |                                    | tgid=42 |
| tgid=43 |                                    +---------+
+---------+
 <-- PID 43 --> <--------- PID 42 --------> <--- PID 44 --->
                        KERNEL VIEW
```

Graphic from https://stackoverflow.com/questions/9305992/if-threads-share-the-same-pid-how-can-they-be-identified

Demo: basic threads

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	%eax: ? %rip: 0x195

T1 →

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

Thread SchedulE #1

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

process control blocks:

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

T1

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

Thread SchedulE #1

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

process control blocks:

Thread 1
%eax: ?
%rip: 0x195

Thread 2
%eax: ?
%rip: 0x195

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

T1

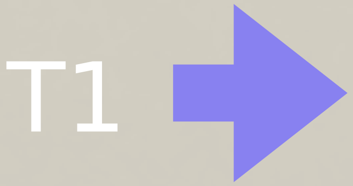
Thread SchedulE #1

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



Thread SchedulE #1

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
- T1 ➡ 0x19d mov %eax, 0x9cd4

Thread Context Switch

Thread SchedulE #1

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

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

process control blocks:

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

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

Thread SchedulE #1

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

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

process control blocks:

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

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

T2 ➡

Thread SchedulE #1

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

process control blocks:

Thread 1
%eax: 101
%rip: 0x1a2

Thread 2
%eax: ?
%rip: 0x195

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

T2 ➡

Desired Result!

Another schedule

Thread SchedulE #2

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

	Thread 1	Thread 2
thread control blocks:	%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1 ➡

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

Thread SchedulE #2

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

thread 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

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

thread 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 Context Switch

Thread SchedulE #2

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

thread 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

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

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

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

Thread SchedulE #2

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

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

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

T2 ➡

- 0x19d mov %eax, 0x9cd4

Thread SchedulE #2

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

thread control blocks:

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

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

T2 ➡

Thread SchedulE #2

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

thread 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 Context Switch

Thread SchedulE #2

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

thread 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 Context Switch

Thread SchedulE #2

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

thread control blocks:

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

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

T1 ➡

Thread SchedulE #2

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

thread control blocks:

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

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

T1

Thread SchedulE #2

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

thread control blocks:	Thread 1	Thread 2
	%eax: 101 %rip: 0x1a2	%eax: 101 %rip: 0x1a2

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

T1 ➡

WRONG Result! Final value of balance is 101

Timeline View

Thread1

```
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? 3: correct!

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?

2: incorrect!

Timeline View

Thread1	**Thread 2**
	mov 0x123, %eax
mov 0x123, %eax	
	add %0x2, %eax
add %0x1, %eax	
	mov %eax, 0x123
mov %eax, 0x123	

How much is added? 1: incorrect!

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? 3: correct!

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? 2: incorrect!

Non-Determinism

Concurrency leads to non-deterministic results

- Not deterministic result: different results even with same inputs
- race conditions

Whether bug manifests depends on CPU schedule!

Passing tests means little

How to program: imagine scheduler is malicious

Assume scheduler will pick bad ordering at some point...

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)