Concurrency: Threads

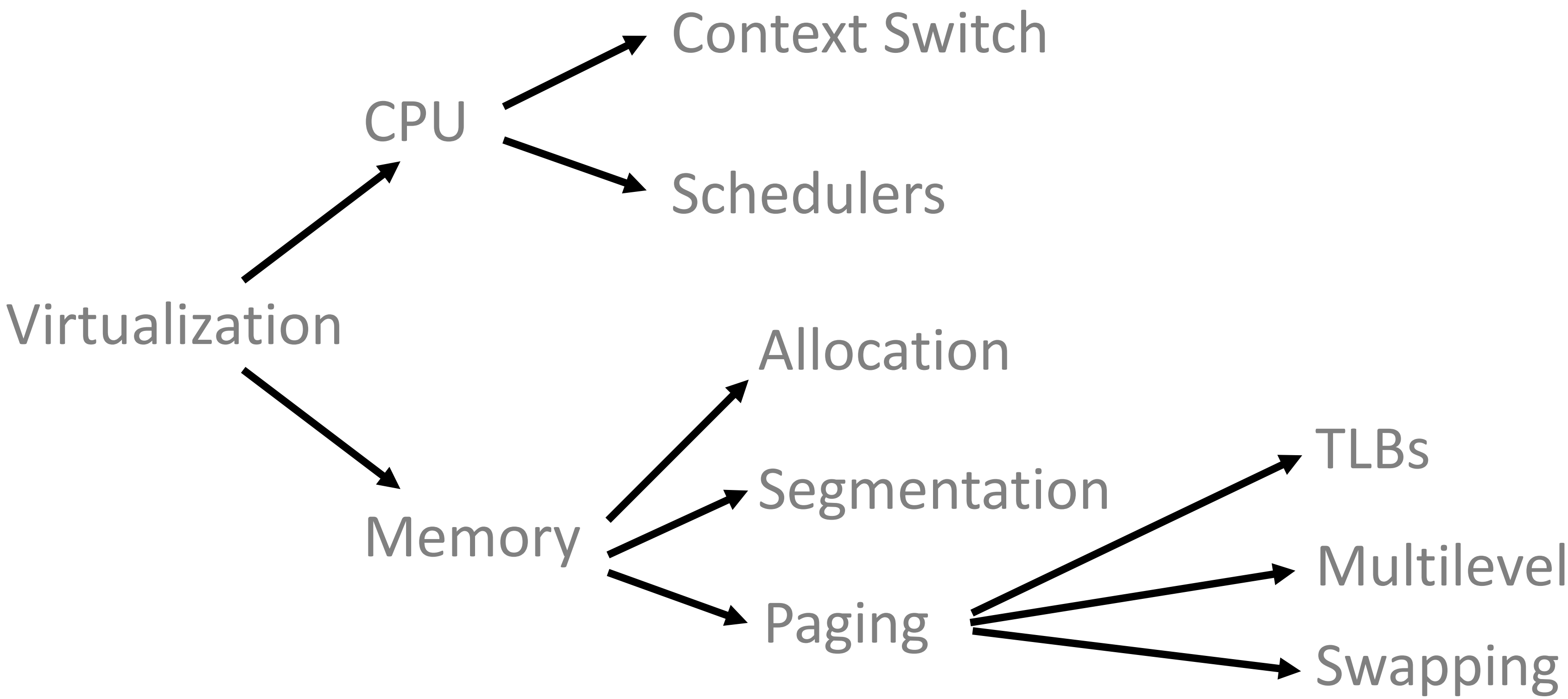
Questions answered in this lecture:

Why is concurrency useful?

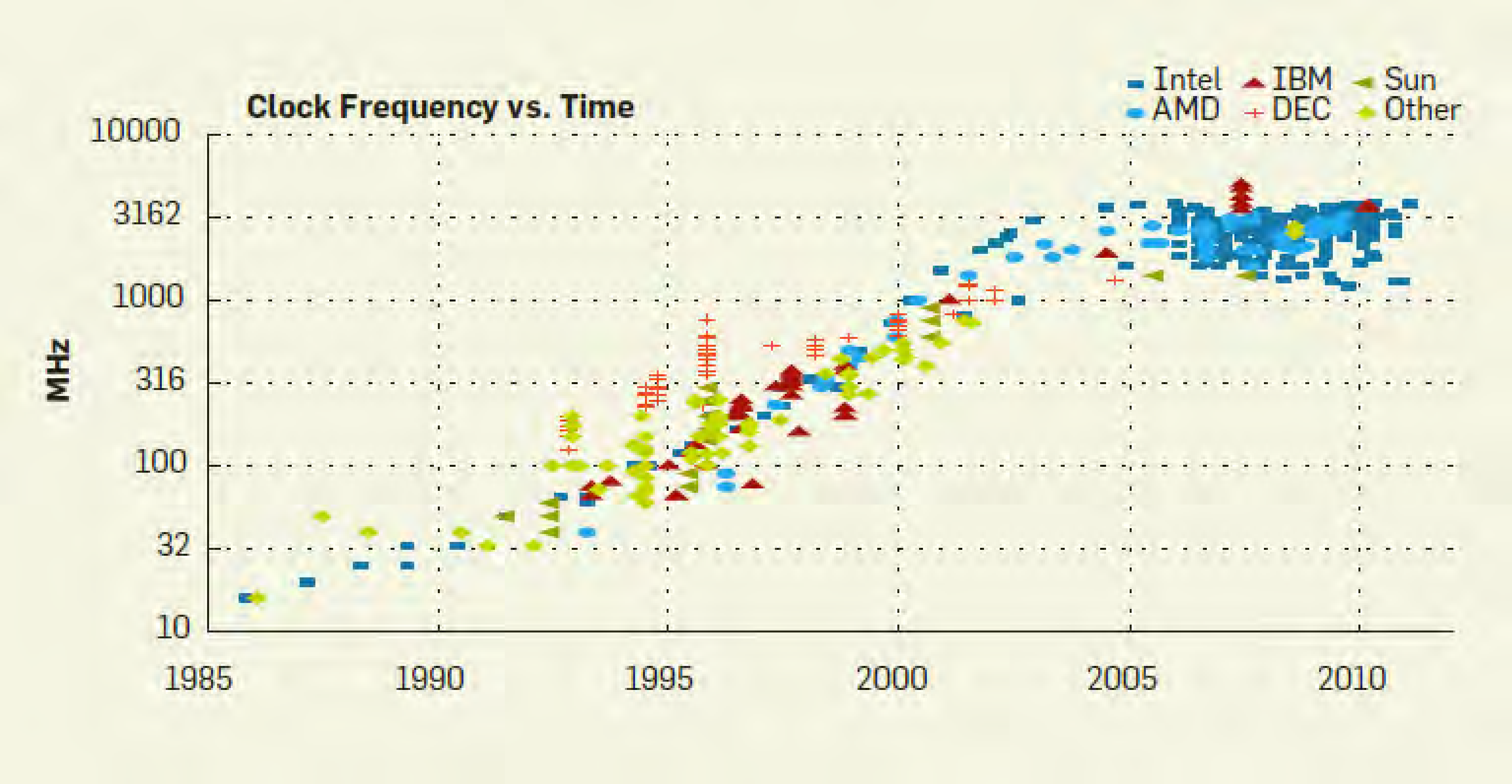
What is a thread and how does it differ from processes?

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

Review: Easy Piece 1

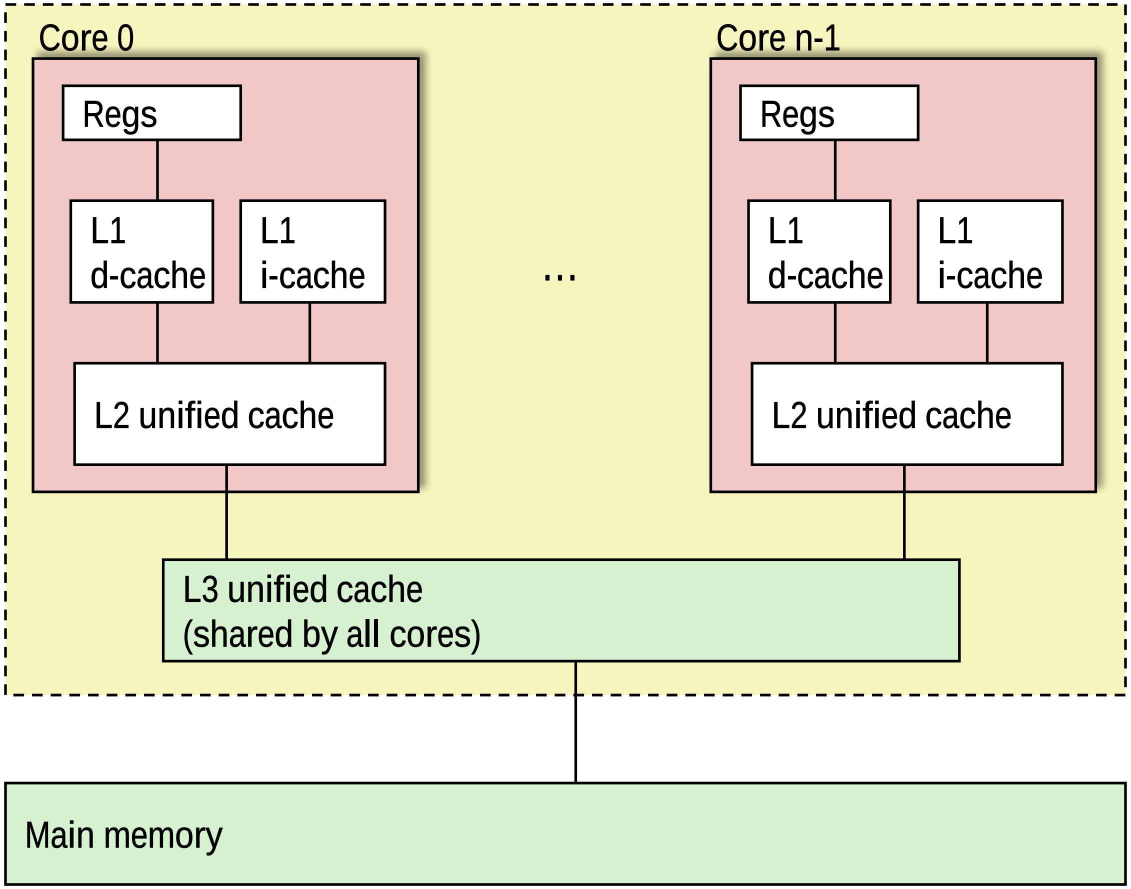


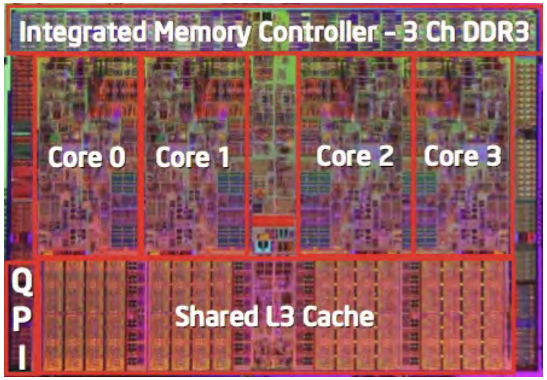
Motivation for Concurrency



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

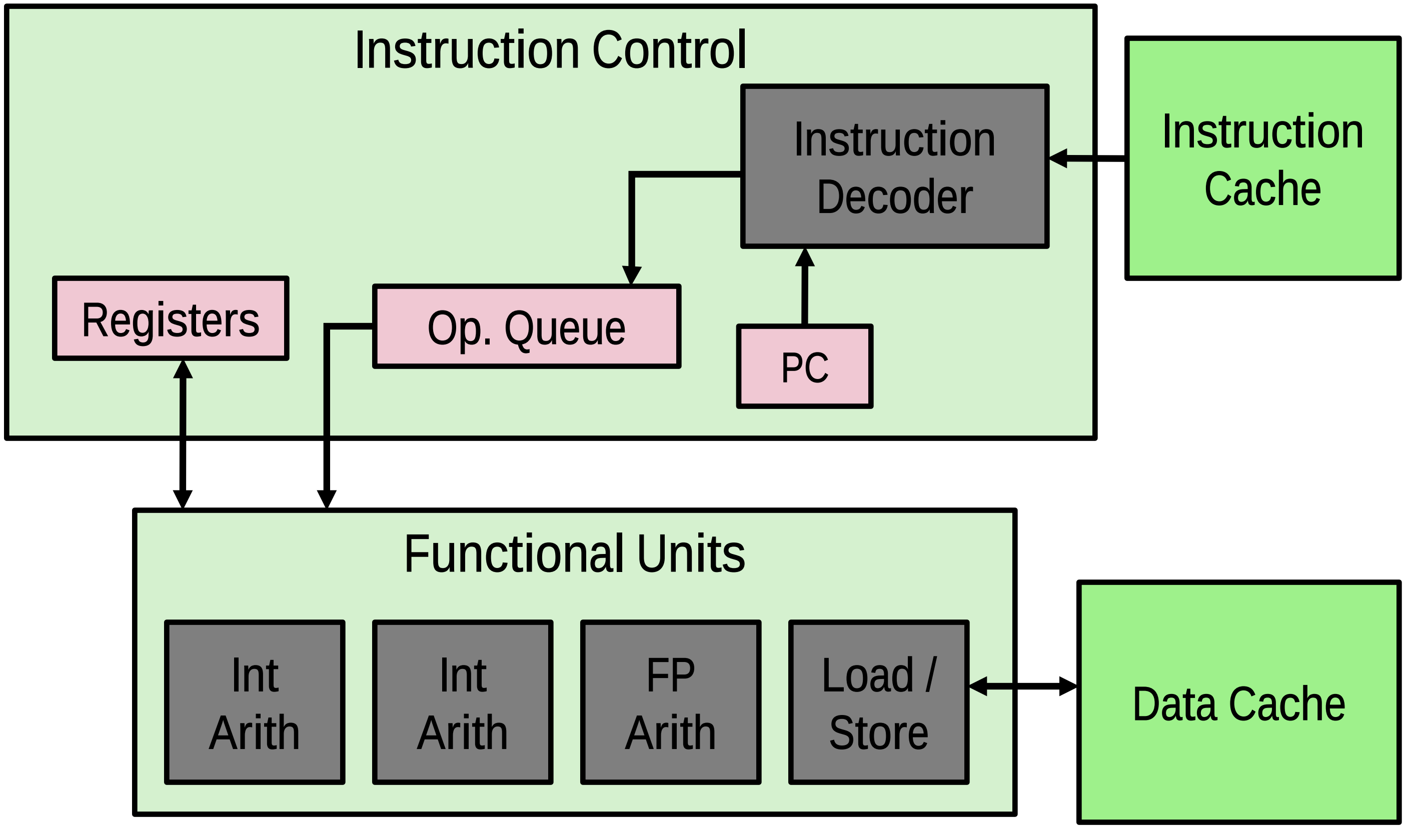
Typical Multicore Processor





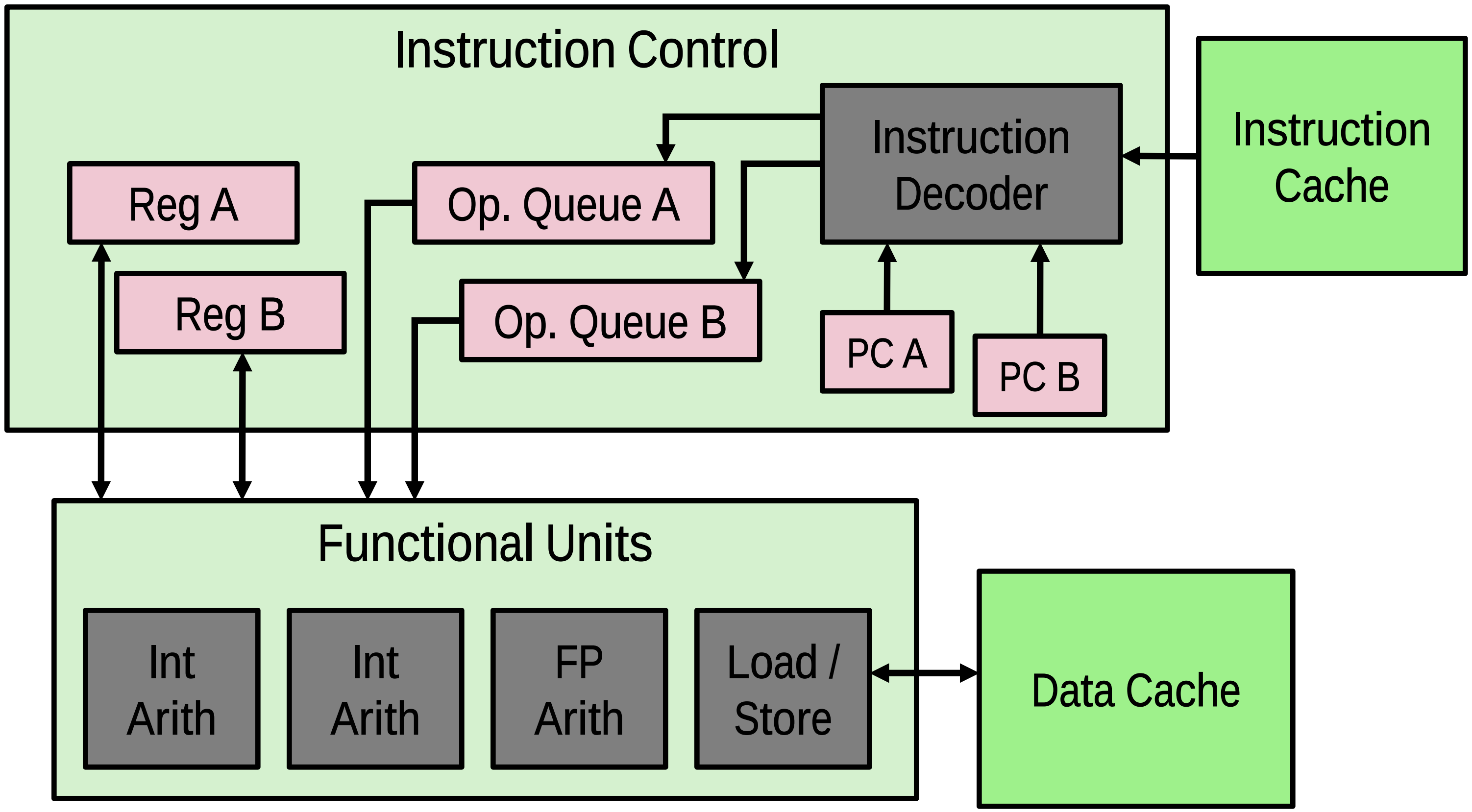
- **Multiple processors operating with coherent view of memory**

Out-of-Order Processor Structure



- Instruction control dynamically converts program into stream of operations
- Operations mapped onto functional units to execute in parallel

Hyperthreading Implementation



- Replicate instruction control to process K instruction streams
- K copies of all registers
- Share functional units

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

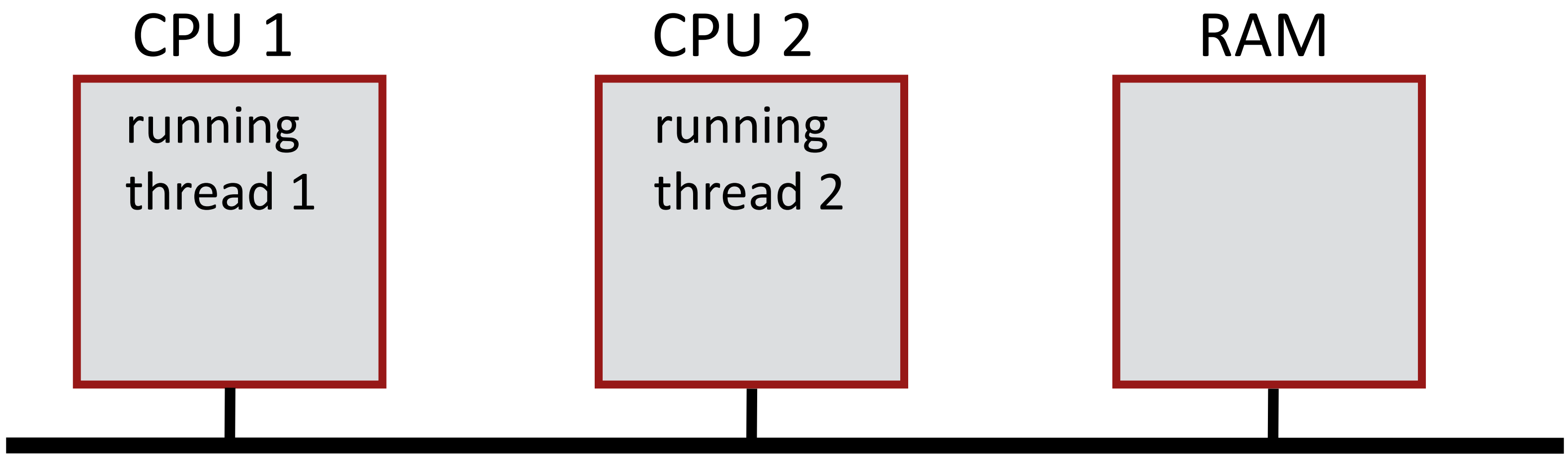
- New abstraction: thread

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

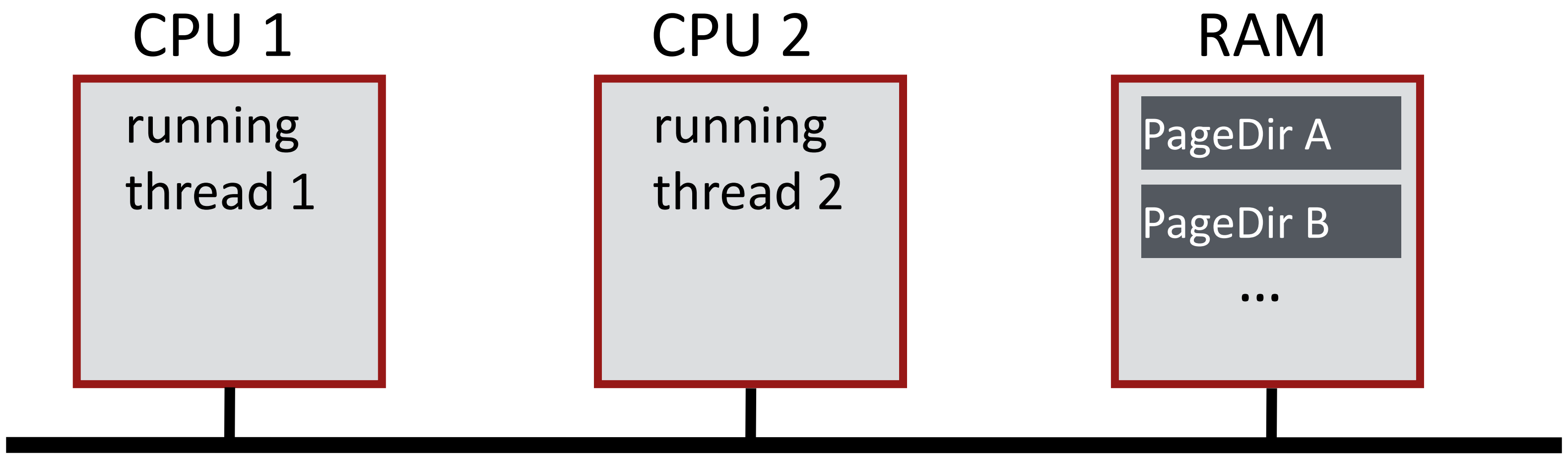
- Divide large task across several cooperative threads
- Communicate through shared address space

Common Programming Models

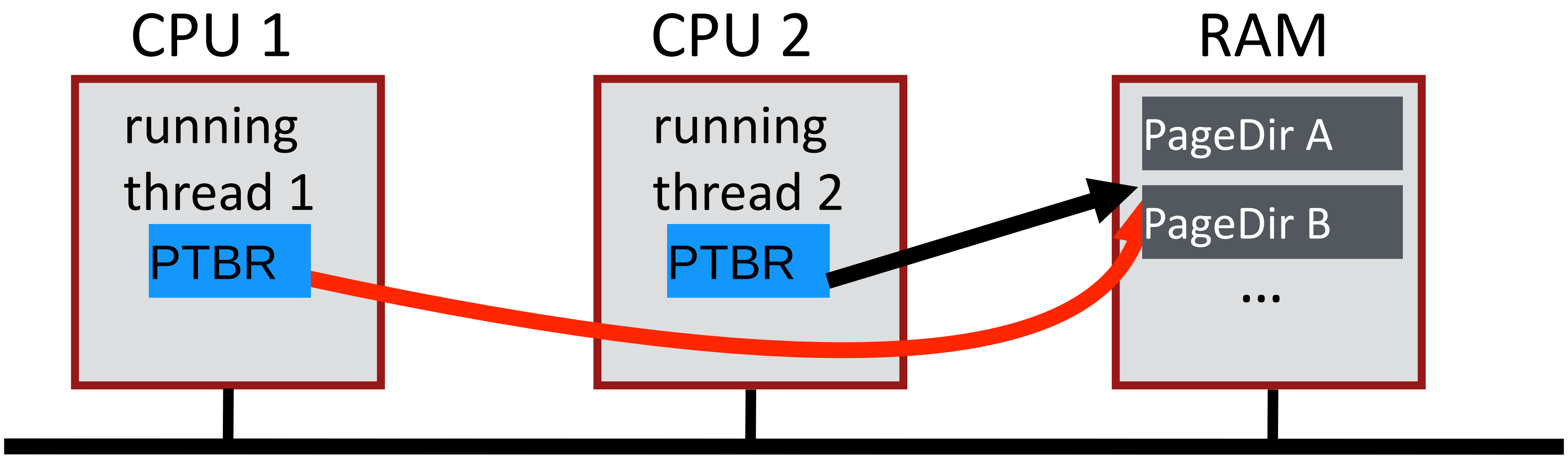
- **Multi-threaded programs tend to be structured as:**
  - **Producer/consumer**
    Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
  - **Pipeline**
    Task is divided into series of subtasks, each of which is handled in series by a different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



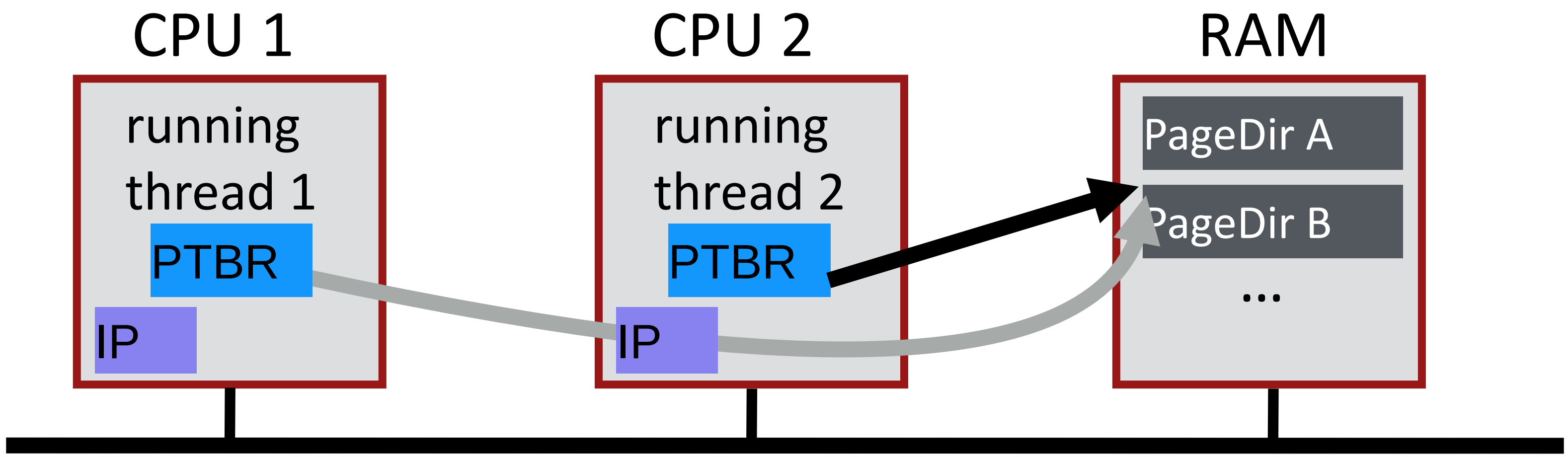
What state do threads share?



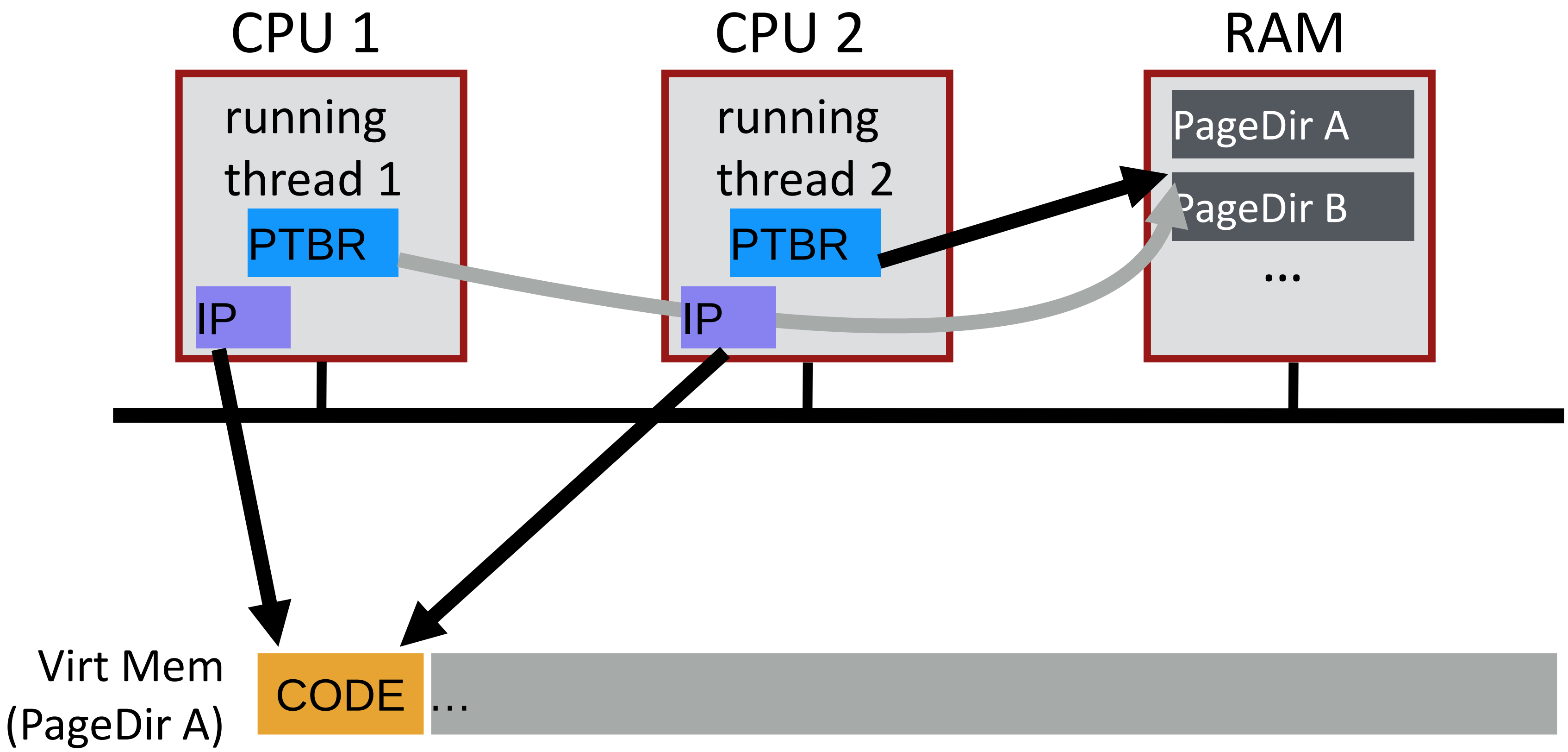
What threads share page directories?

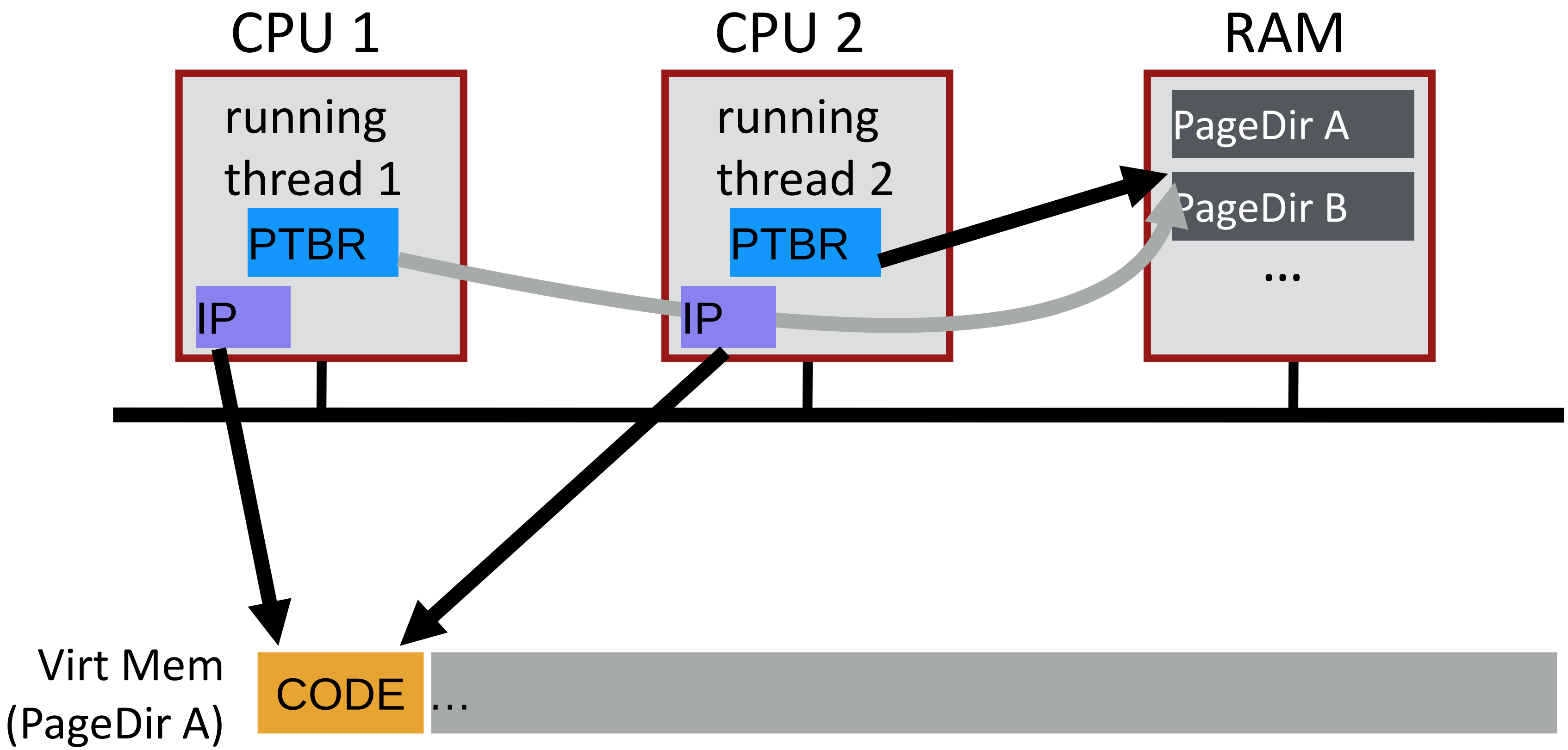


Threads belonging to the same process share page directory



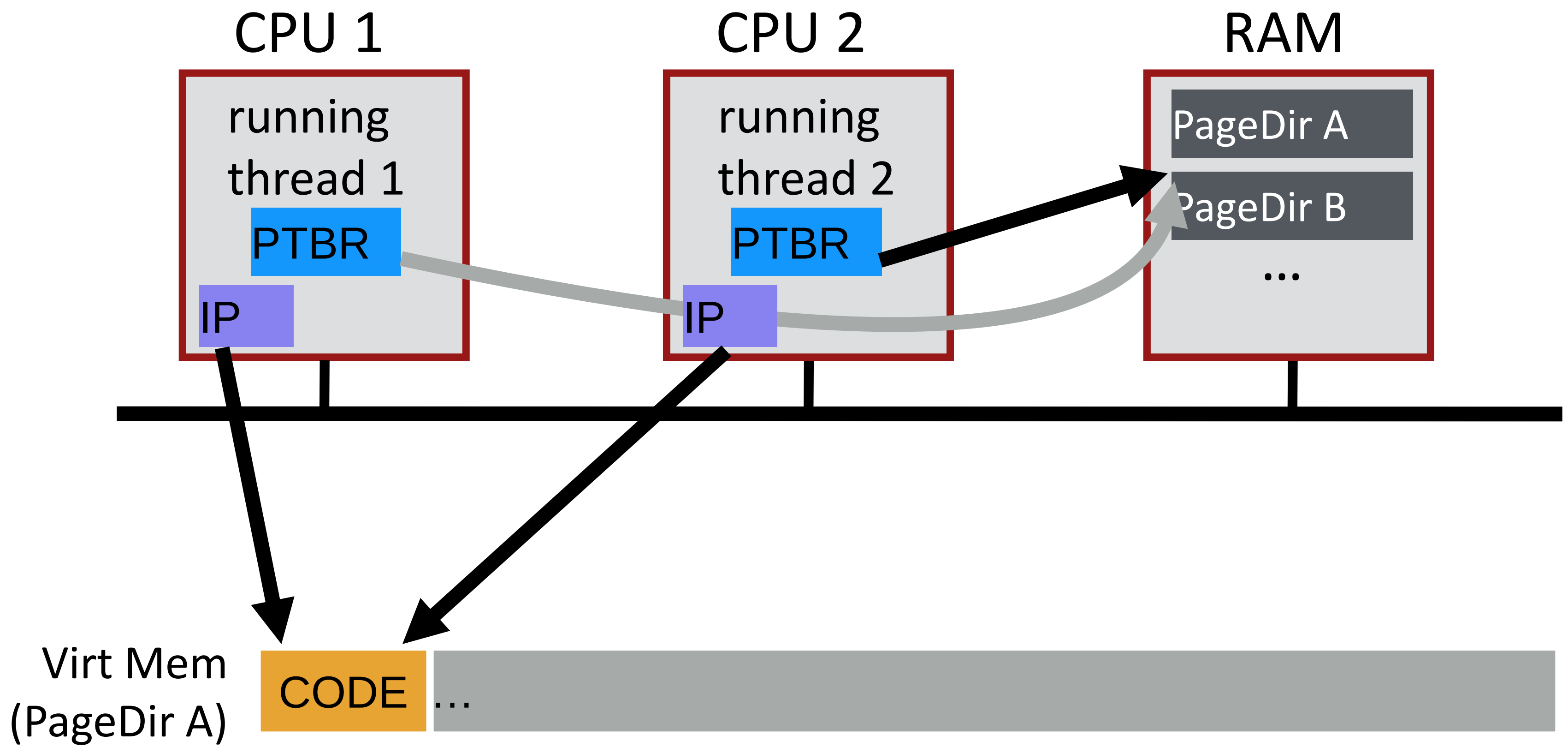
Do threads share Instruction Pointer?


CPU 1
running
thread 1
PTBR
IP
CPU 2
running
thread 2
PTBR
IP
RAM
PageDir A
PageDir B
...
Virt Mem
(PageDir A)
CODE
...

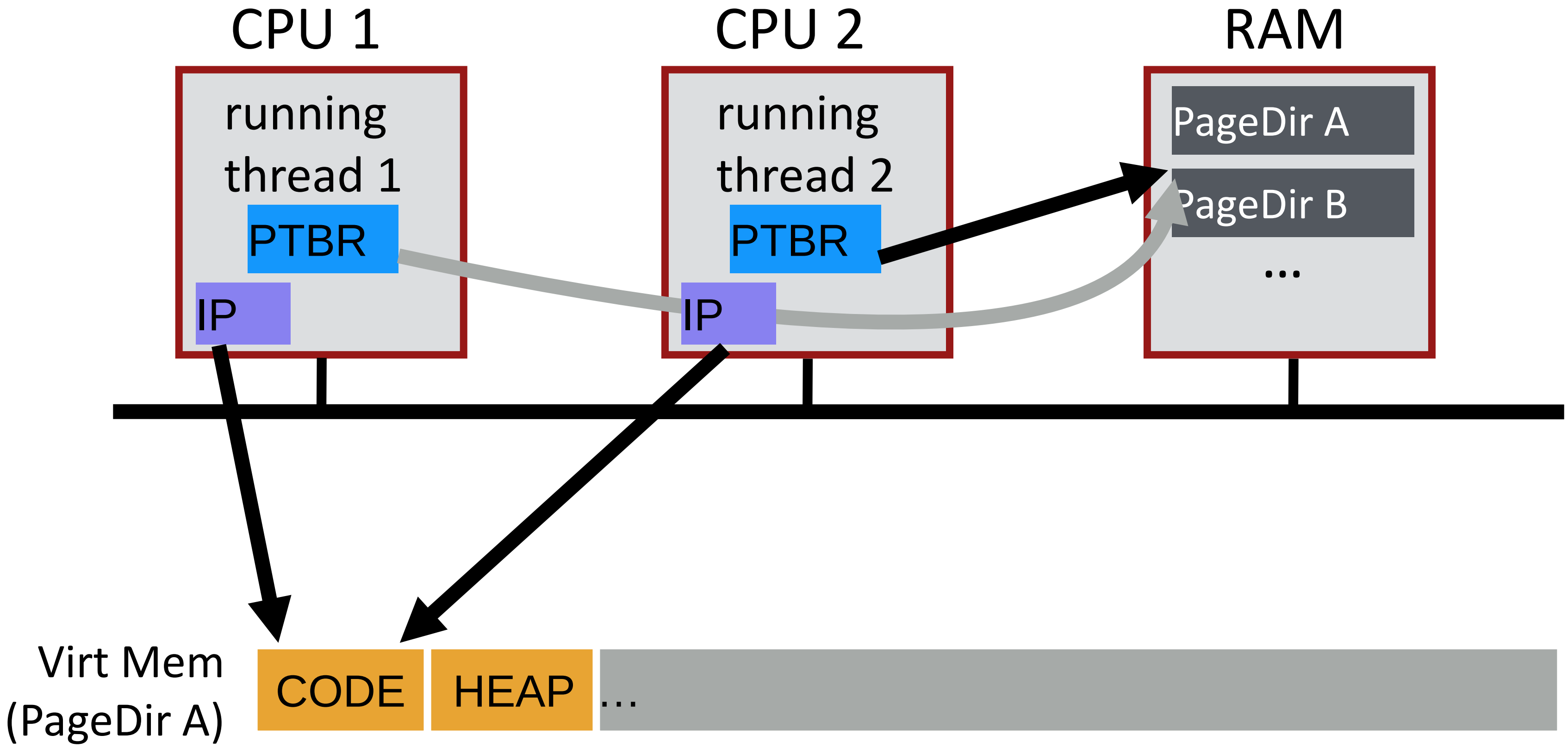


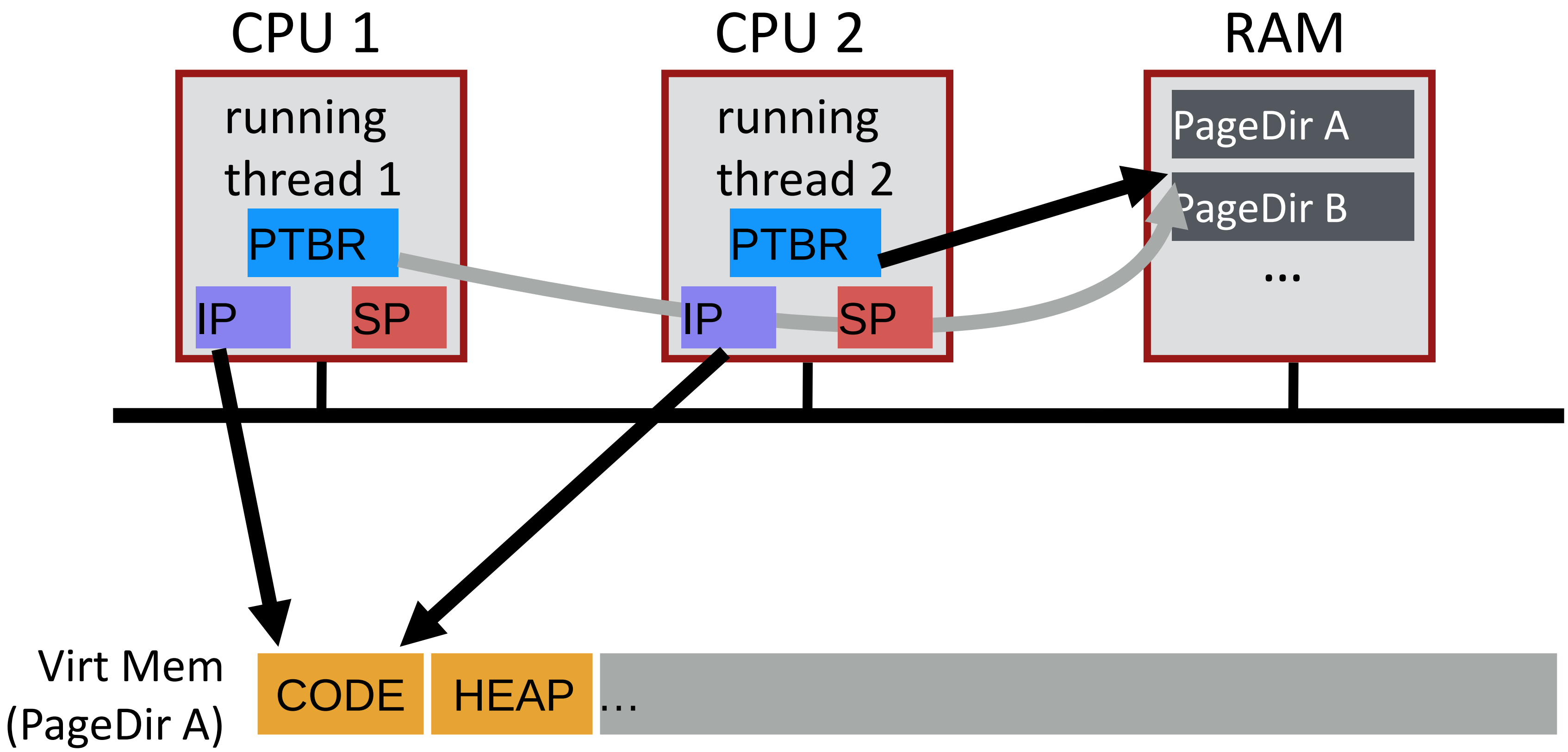
Share code, but each thread may be executing different code at the same time

→ Different Instruction Pointers

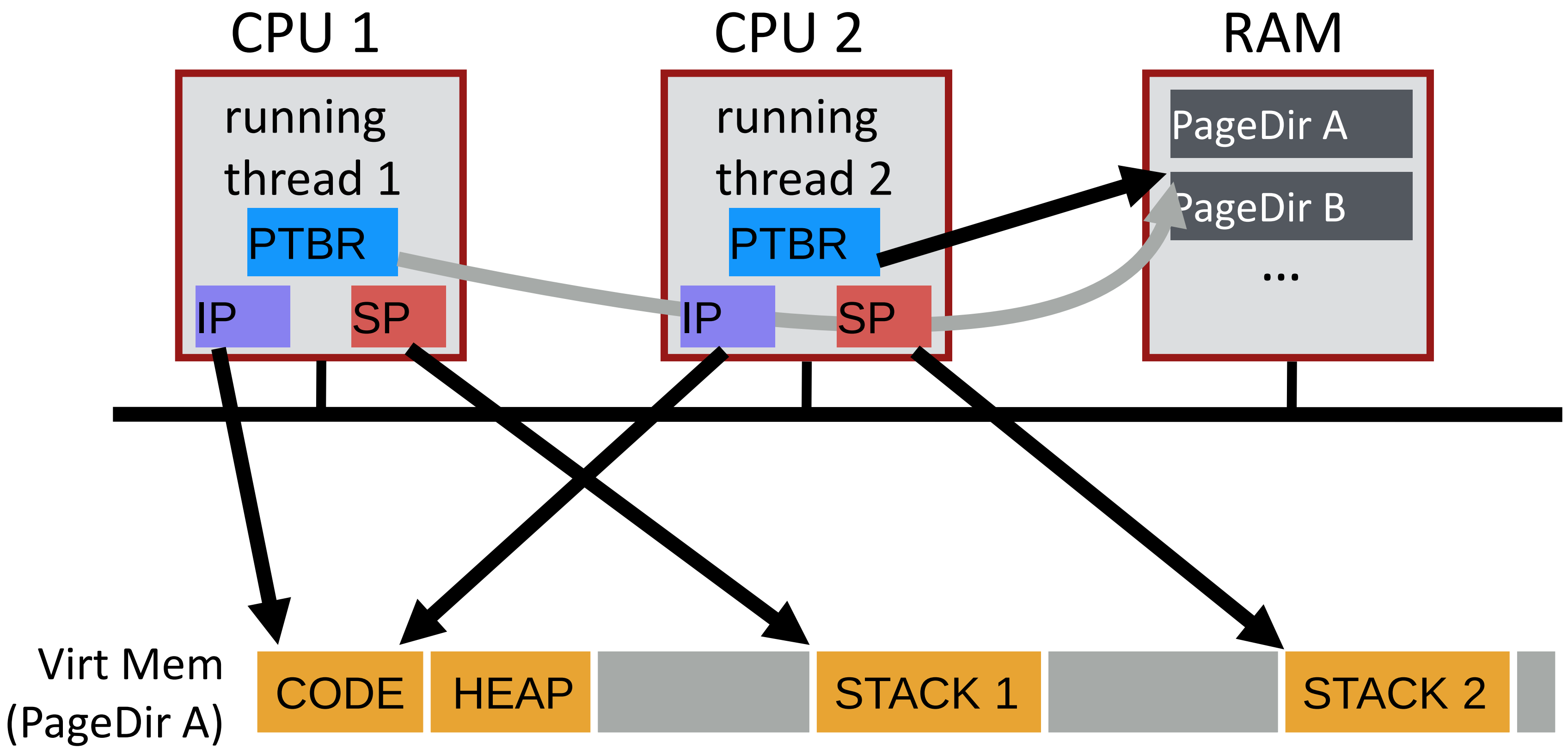


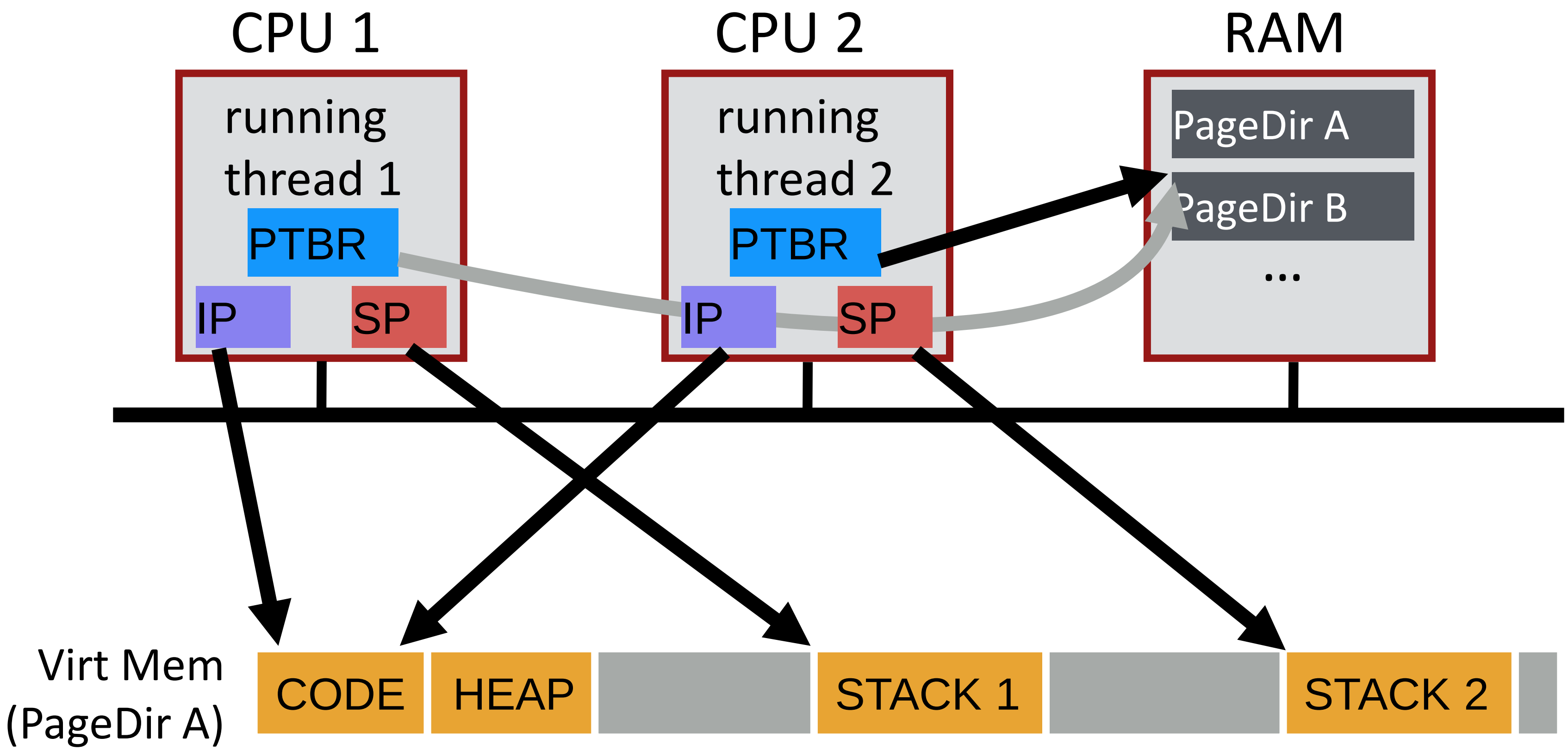
Do threads share heap?


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
...



Do threads share stack pointer?


CPU 1
running thread 1
PTBR
IP
SP
CPU 2
running thread 2
PTBR
IP
SP
RAM
PageDir A
PageDir B
...
Virt Mem
(PageDir A)
CODE
HEAP
STACK 1
STACK 2



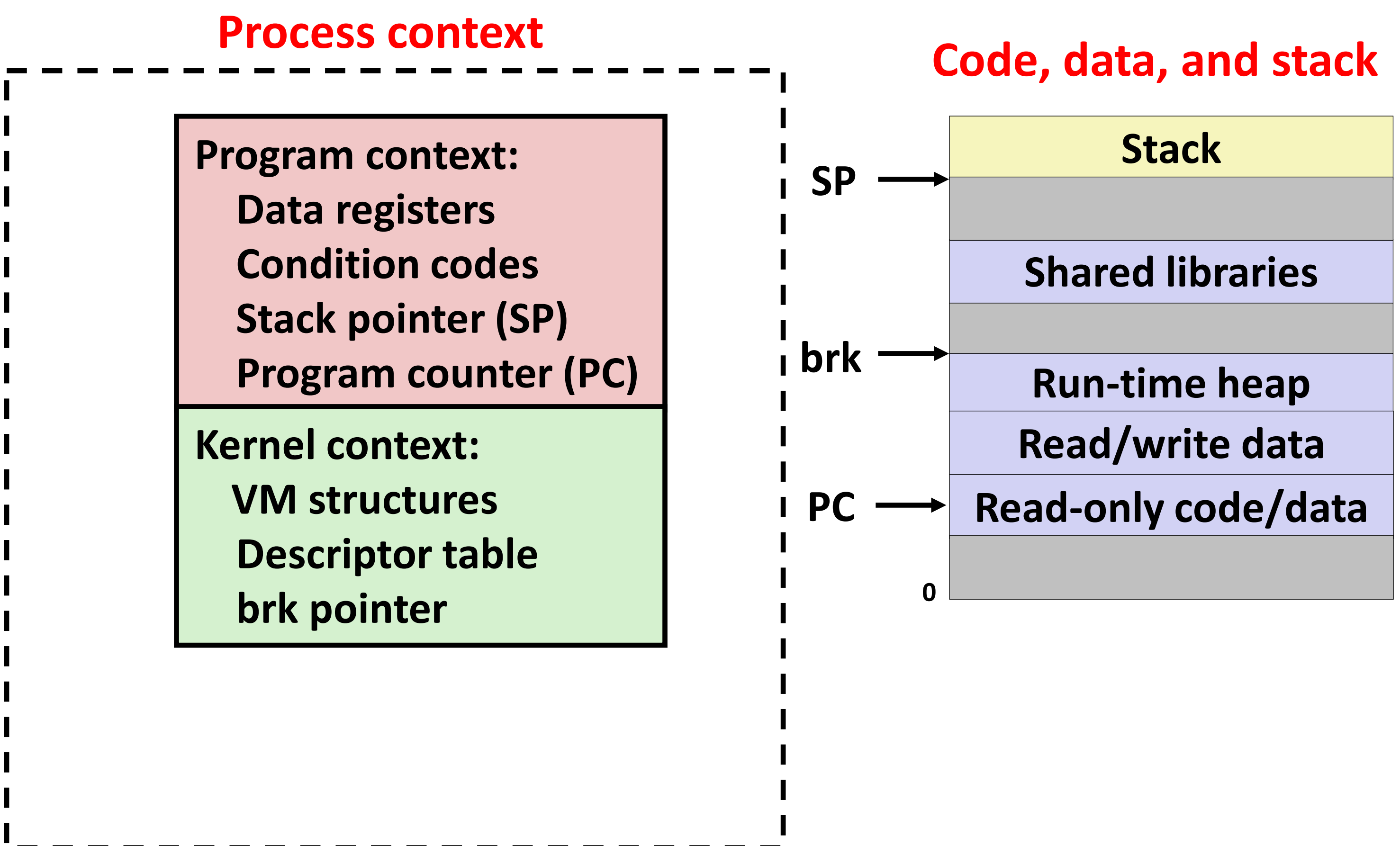
Threads executing different functions need different stacks

Processes vs. Threads

- **A process is different than a thread**
- **Thread: “Lightweight process” (LWP)**
  - An execution stream that shares an address space
  - Multiple threads within a single process
- **Example:**
  - Two **processes** examining same memory address 0xffe84264 see **different** values (I.e., different contents)
  - Two **threads** examining memory address 0xffe84264 see **same** value (I.e., same contents)

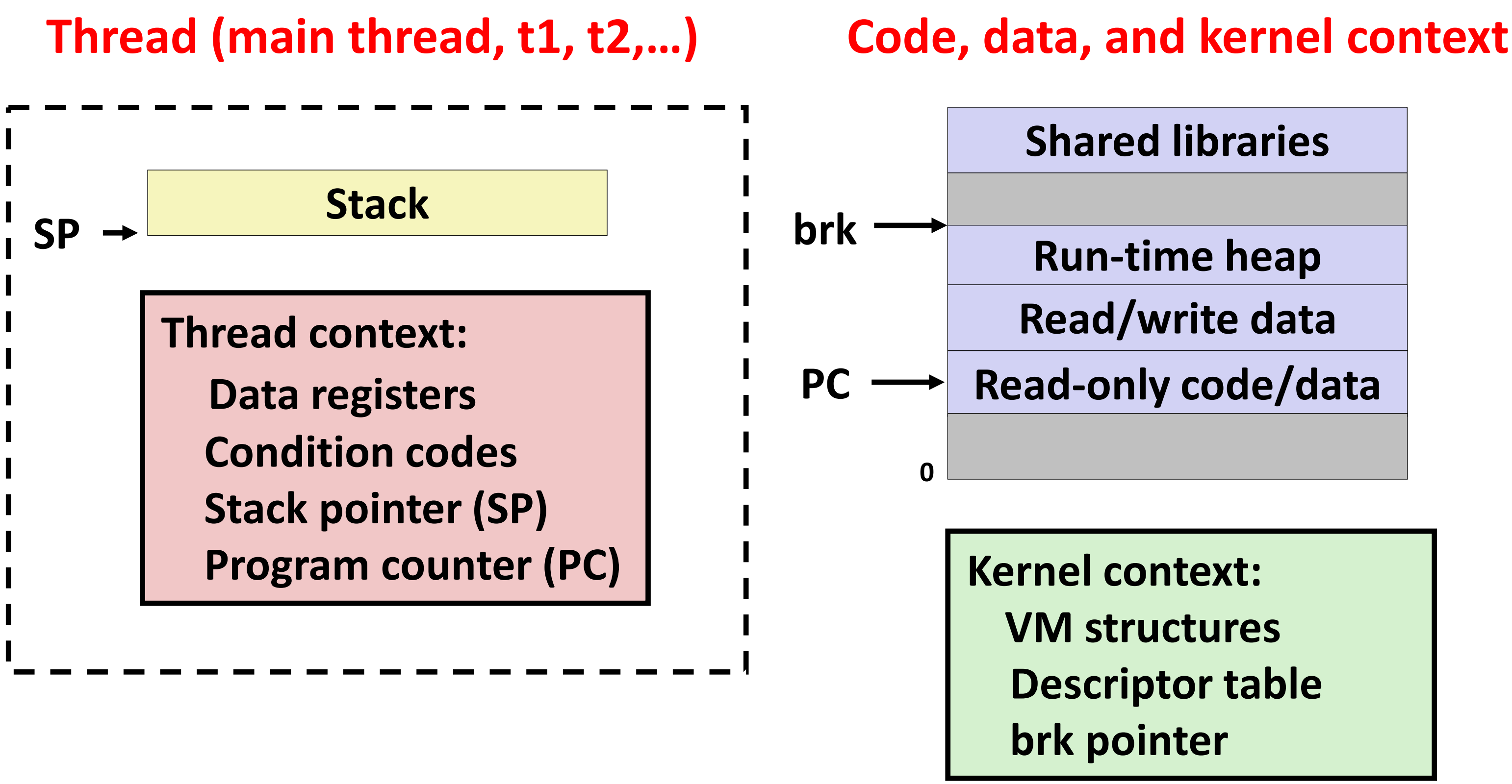
Traditional View of a Process

- Process = process context + code, data, and stack



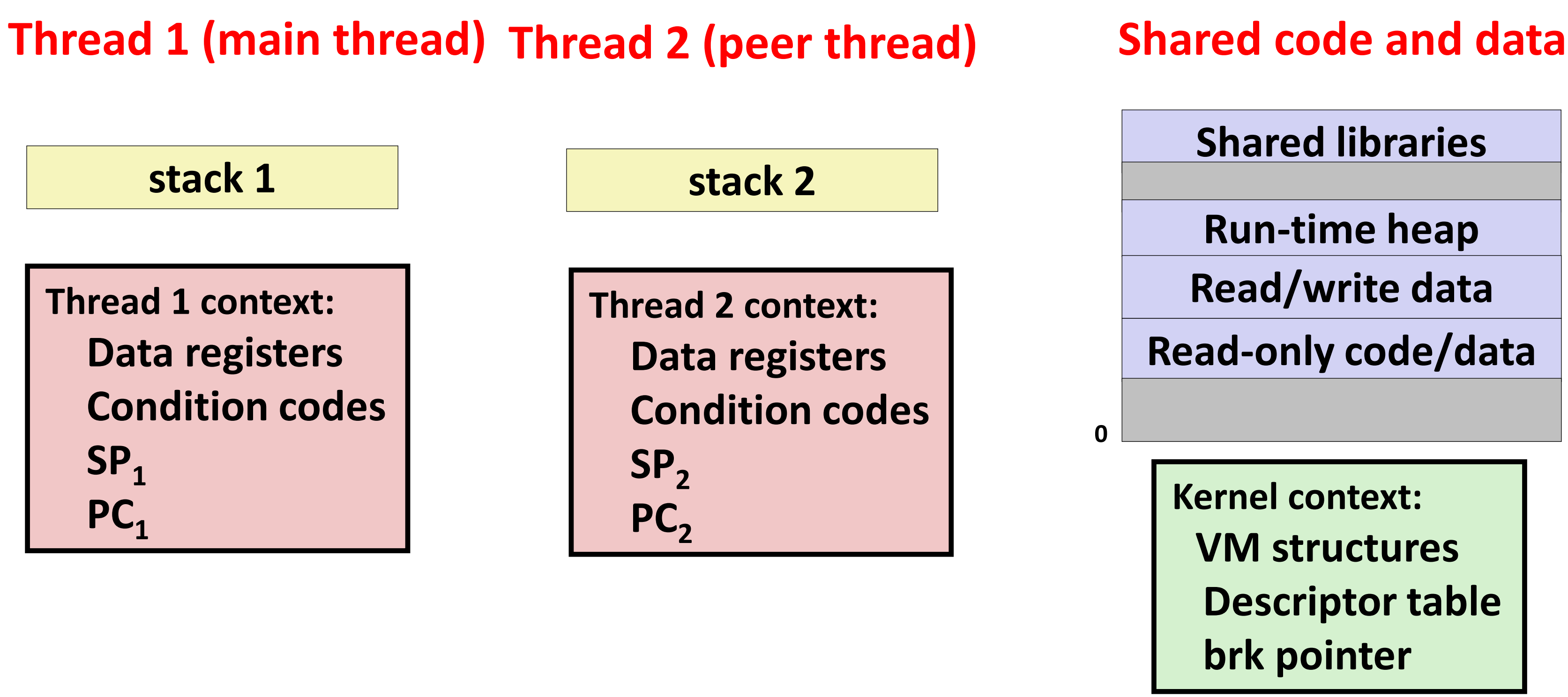
Alternate View of a Process

- Process = threads + code, data, and kernel context



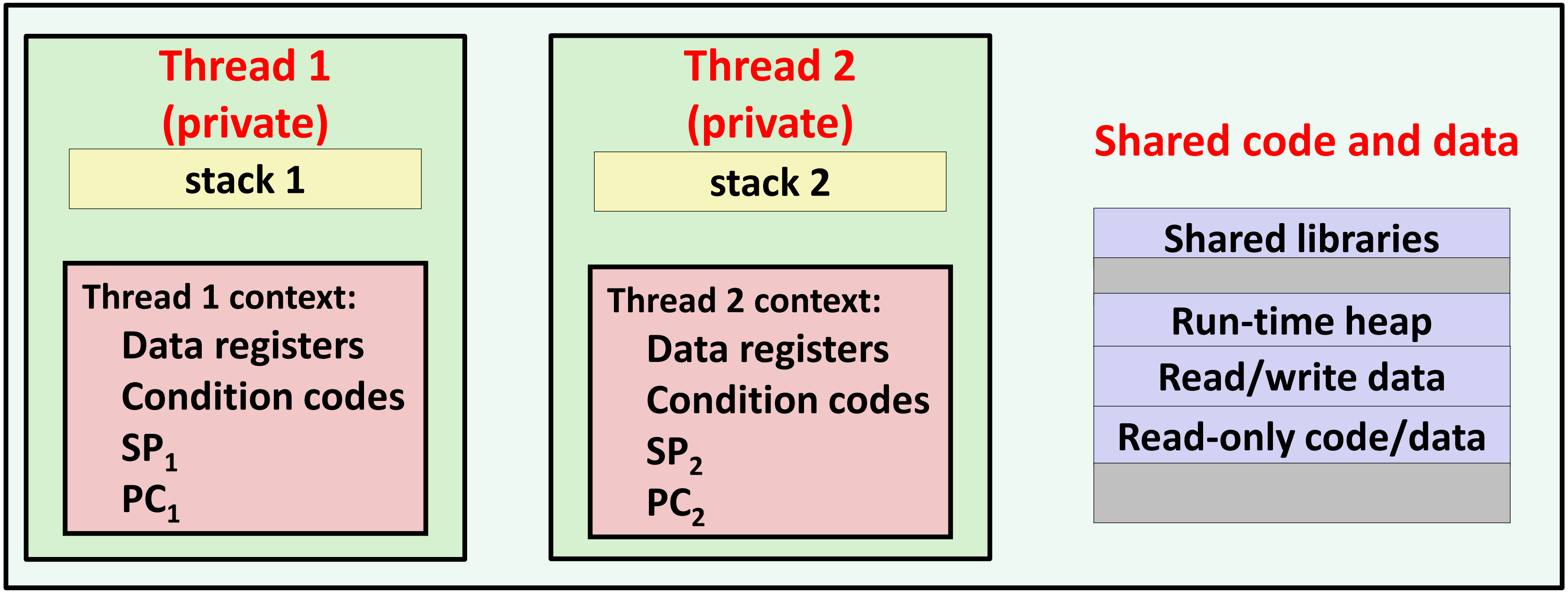
A Process With Multiple Threads

- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)



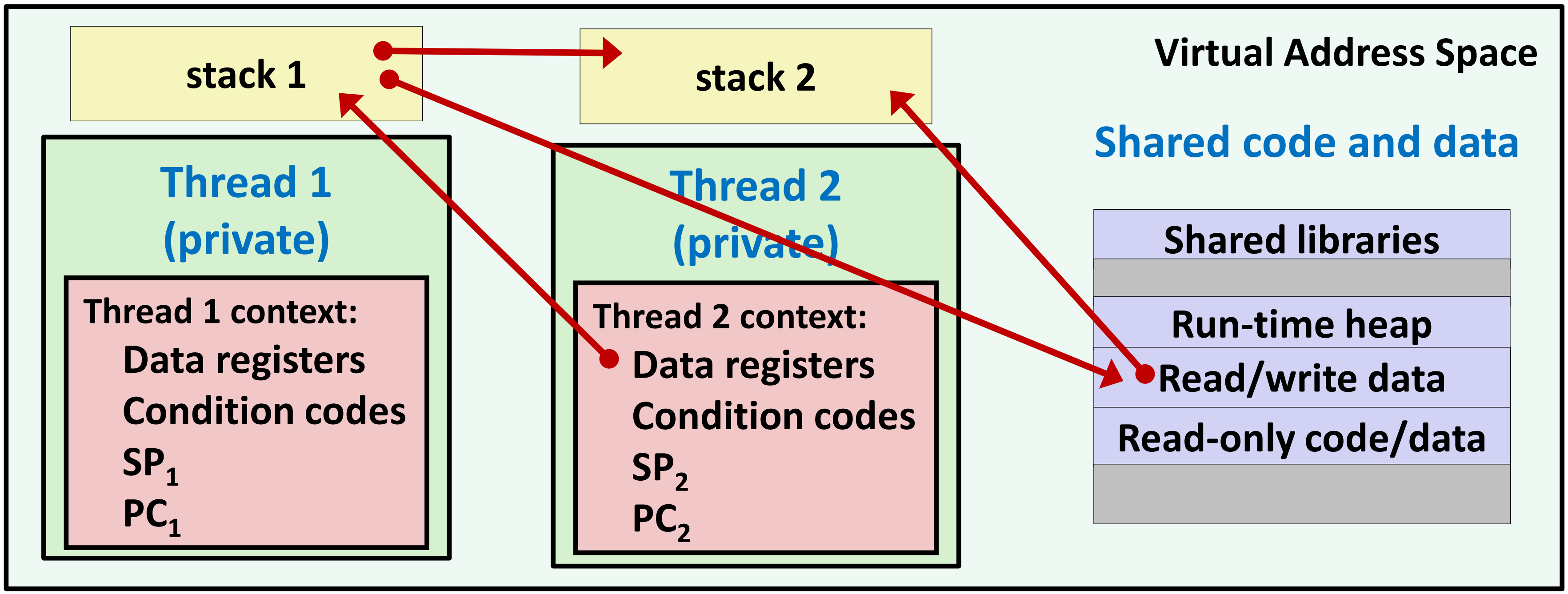
Threads Memory Model: Conceptual

- **Multiple threads run within the context of a single process**
- **Each thread has its own separate thread context**
  - Thread ID, stack, stack pointer, PC, condition codes(eflags), and General Purpose registers
- **All threads share the remaining process context**
  - Code, data, heap, and shared library segments of the process virtual address space
  - Open files and installed handlers



Threads Memory Model: Actual

- **Separation of data is not strictly enforced:**
  - Register values are truly separate and protected, but…
  - Any thread can read and write the stack of any other thread



The mismatch between the conceptual and operation model is a source of confusion and errors

Example Program to Illustrate Sharing

```
char **ptr;  /* global var */

int main(int argc, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread's stack indirectly through global ptr variable

A common, but inelegant way to pass a single argument to a thread routine

Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**

- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**

- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable**

Mapping Variable Instances to Memory

Global var: 1 instance (`ptr` [data])

Local vars: 1 instance (`i.m`, `msgs.m`)

```
char **ptr;  /* global var */

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

Local var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
    return NULL;
}
```

Local static var: 1 instance (`cnt` [data])

Shared Variable Analysis

- **Which variables are shared?**

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i (main)`	yes	no	no
`msgs (main)`	yes	yes	yes
`myid (t0)`	no	yes	no
`myid (t1)`	no	no	yes

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL, thread,(void *)i);
    Pthread_exit(NULL);}
```

```
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]:  %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```

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
  - 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)

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
  - 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

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: ?	%eax: ?
	%rip: 0x195	%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: ?	%eax: ?
%rip: 0x19a	%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	Thread 2
%eax: ? %rip: 0x19d	%eax: ? %rip: 0x195

- **0x195 mov (0x9cd4), %eax**
- **0x19a add $0x1, %eax**
- T1 → **0x19d mov %eax, (0x9cd4)**

Thread Schedule #1

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

process control blocks:

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

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

T1

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	%eax: ?
	%rip: 0x1a2	%rip: 0x19a

T2

- **0x195 mov (0x9cd4), %eax**
- **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: 0x19d

- **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: 0x1a2

- **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	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x1a2

- **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

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 #2

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

process control blocks:

	Thread 1	Thread 2
%eax:	?	?
%rip:	0x19a	0x195

T1 →

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

Thread Schedule #2

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

process control blocks:

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

- **0x195 mov (0x9cd4), %eax**
- **0x19a add $0x1, %eax**
- T1 → **0x19d mov %eax, (0x9cd4)**

Thread Context Switch

Thread Schedule #2

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

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

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x19a

T2

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

Thread Schedule #2

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

process control blocks:

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

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

T2 →

Thread Schedule #2

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

process control blocks:

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

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

T2

Thread Schedule #2

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

process control blocks:

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

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 Context Switch

Thread Schedule #2

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**
- **0x19d mov %eax, (0x9cd4)** ← T1

Thread Schedule #2

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

process 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

process 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

Thread 1

Thread 2

```
mov (0x123), %eax
add $0x1, %eax
mov %eax, (0x123)
```

```
mov (0x123), %eax
add $0x2, %eax
mov %eax, (0x123)
```

How much is added to shared variable?

3: correct!

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?

2: incorrect!

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?

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 (Thread 2)

add $0x2, %eax (Thread 2)

mov (0x123), %eax (Thread 1)

add $ 0x1, %eax (Thread 1)

mov %eax, (0x123) (Thread 1)

mov %eax, (0x123) (Thread 2)

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…**

badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
             *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

What went wrong?

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i **: Head**

L_i **: Load cnt**

U_i **: Update cnt**

S_i **: Store cnt**

T_i **: Tail**

Concurrent Execution

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i denotes that thread i executes instruction I
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Concurrent Execution

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i denotes that thread i executes instruction I
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

OK

Thread 1 critical section

Thread 2 critical section

Concurrent Execution (cont)

- **Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2**

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

Concurrent Execution (cont)

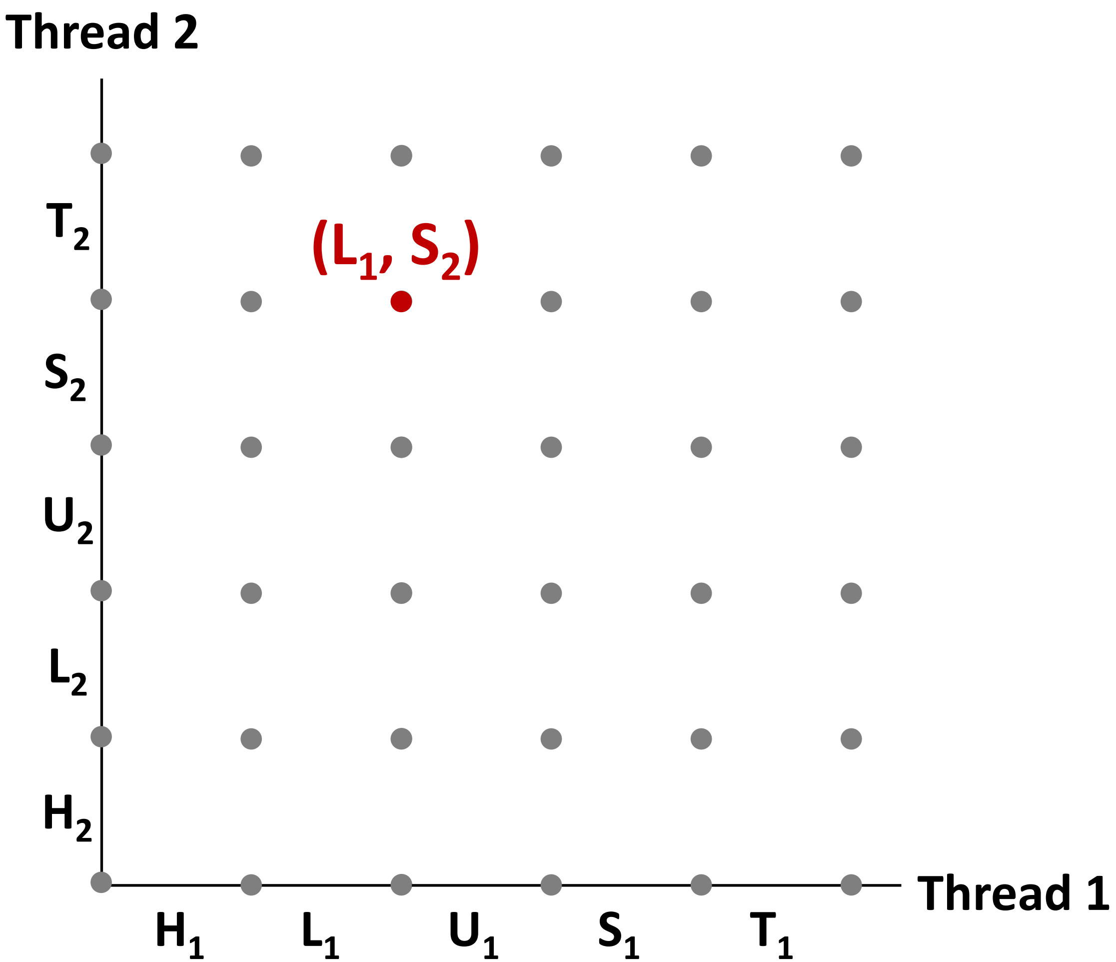
- **How about this ordering?**

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			1
2	T_2			1

Oops!

- **We can analyze the behavior using a *progress graph***

Progress Graphs



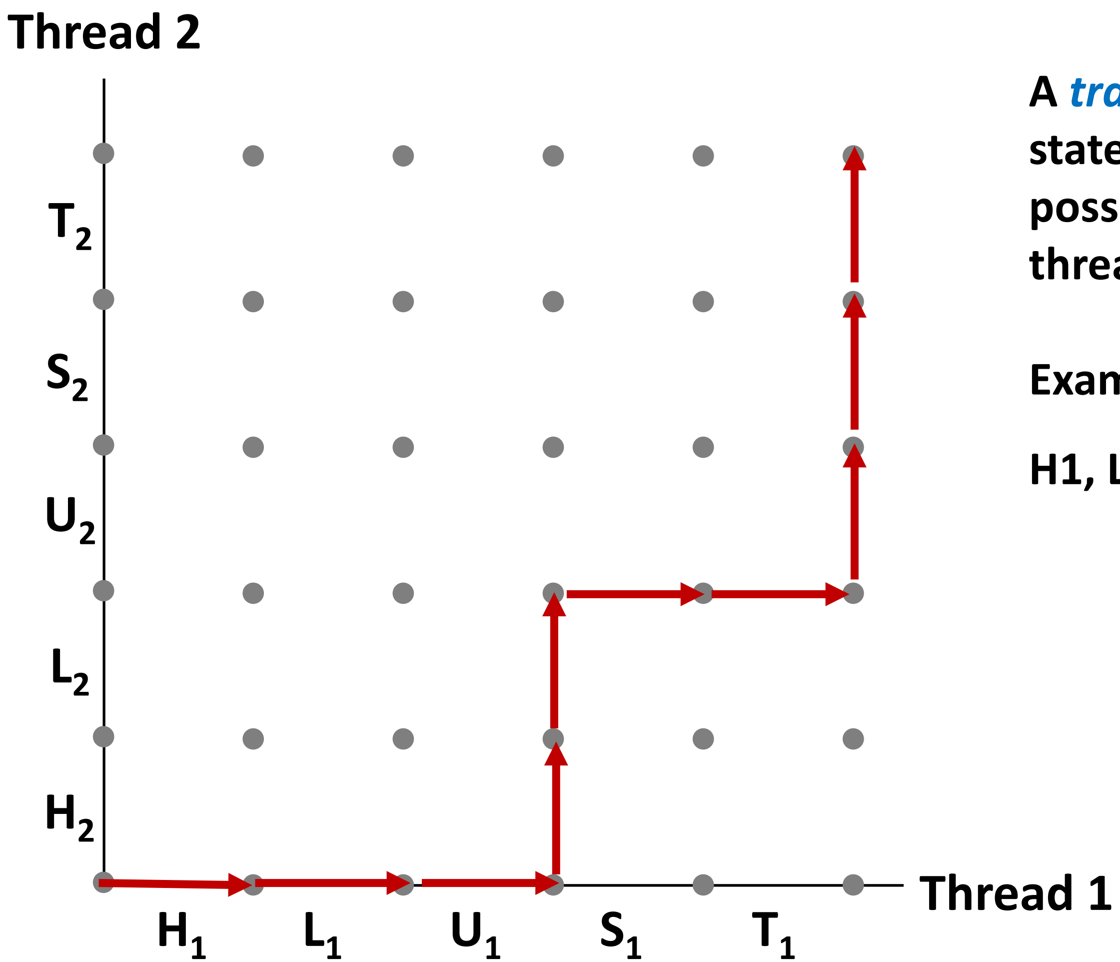
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible ***execution state*** $(Inst_1, Inst_2)$.

E.g., (L_1, S_2) denotes state where thread 1 has completed L_1 and thread 2 has completed S_2.

Trajectories in Progress Graphs

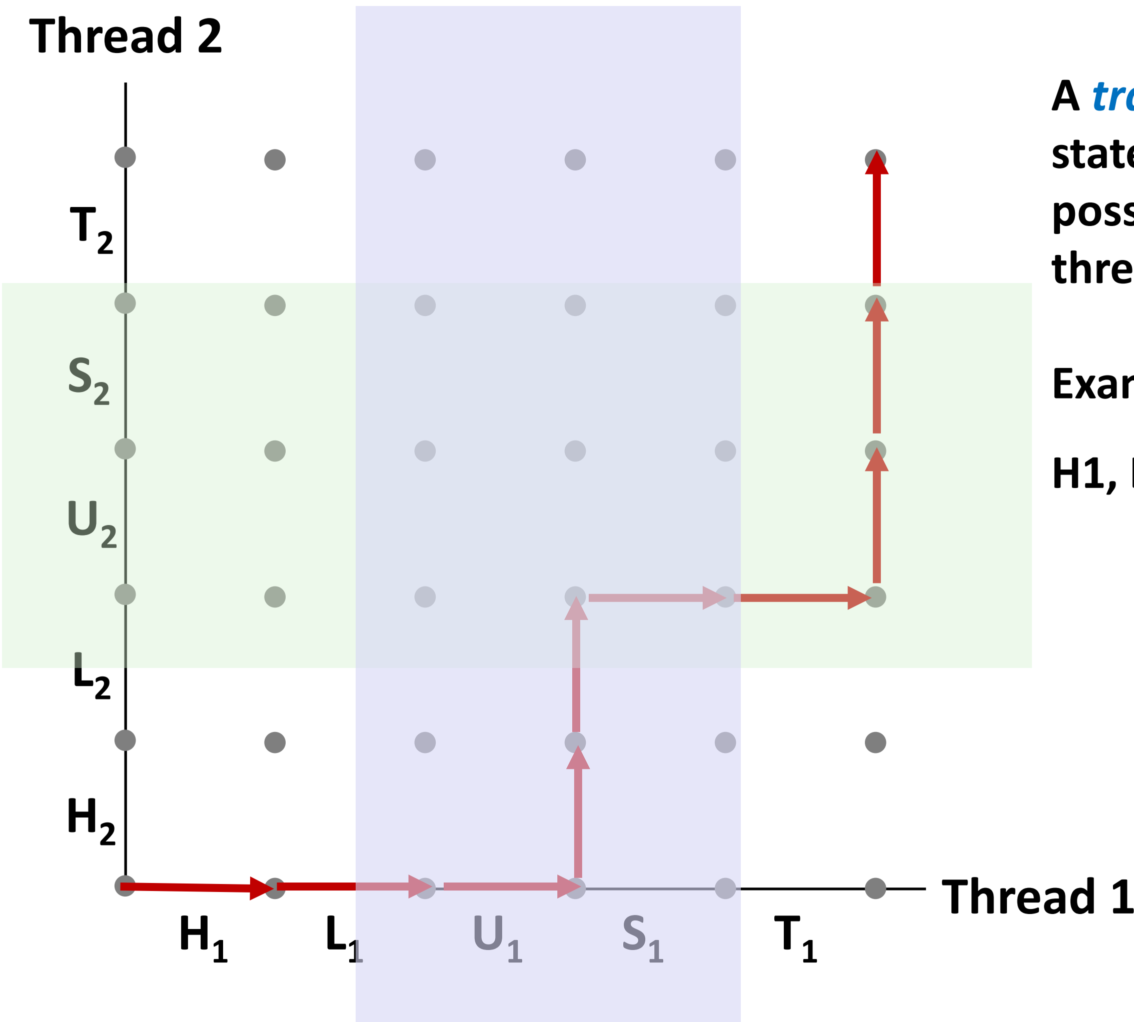


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Trajectories in Progress Graphs

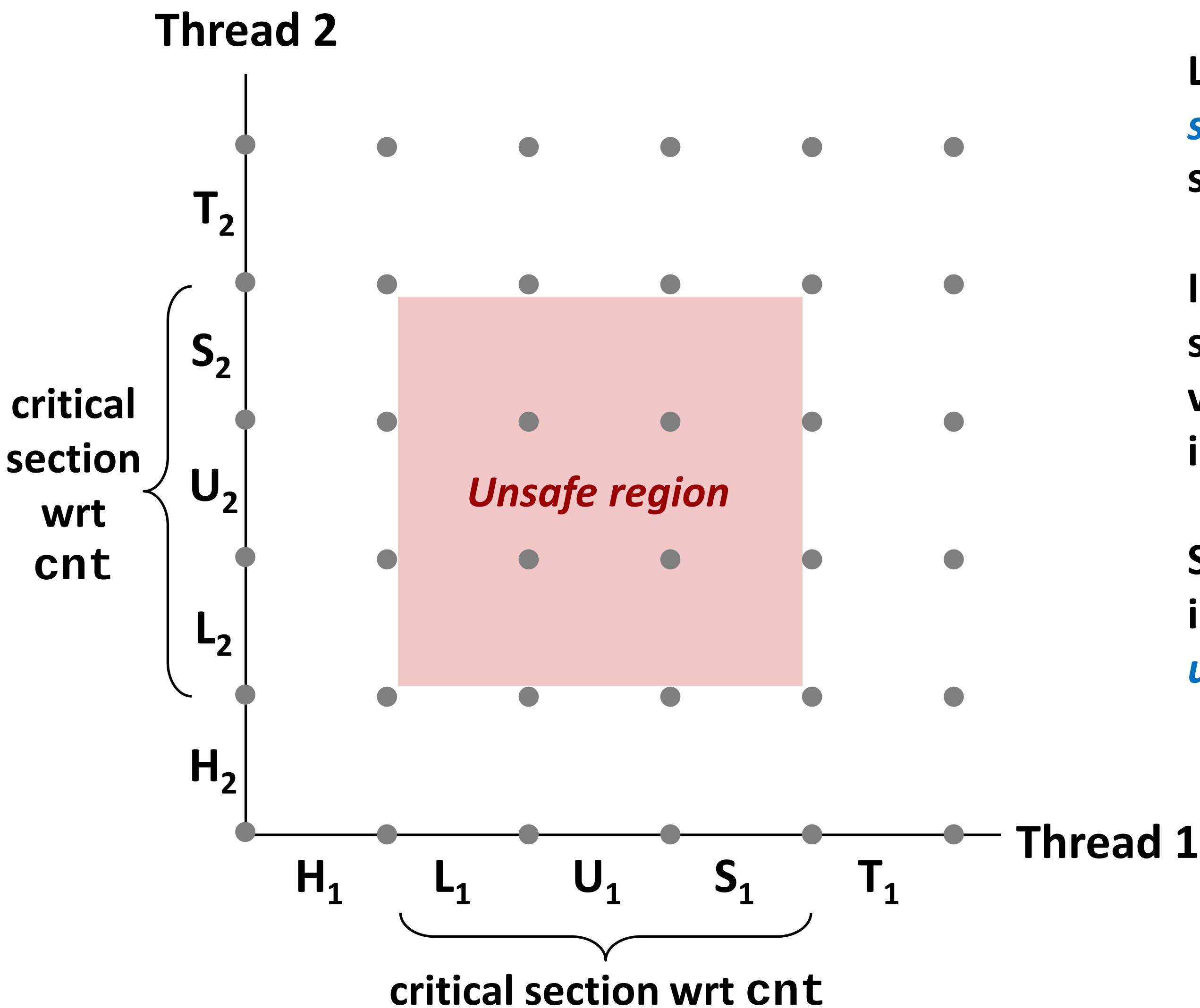


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

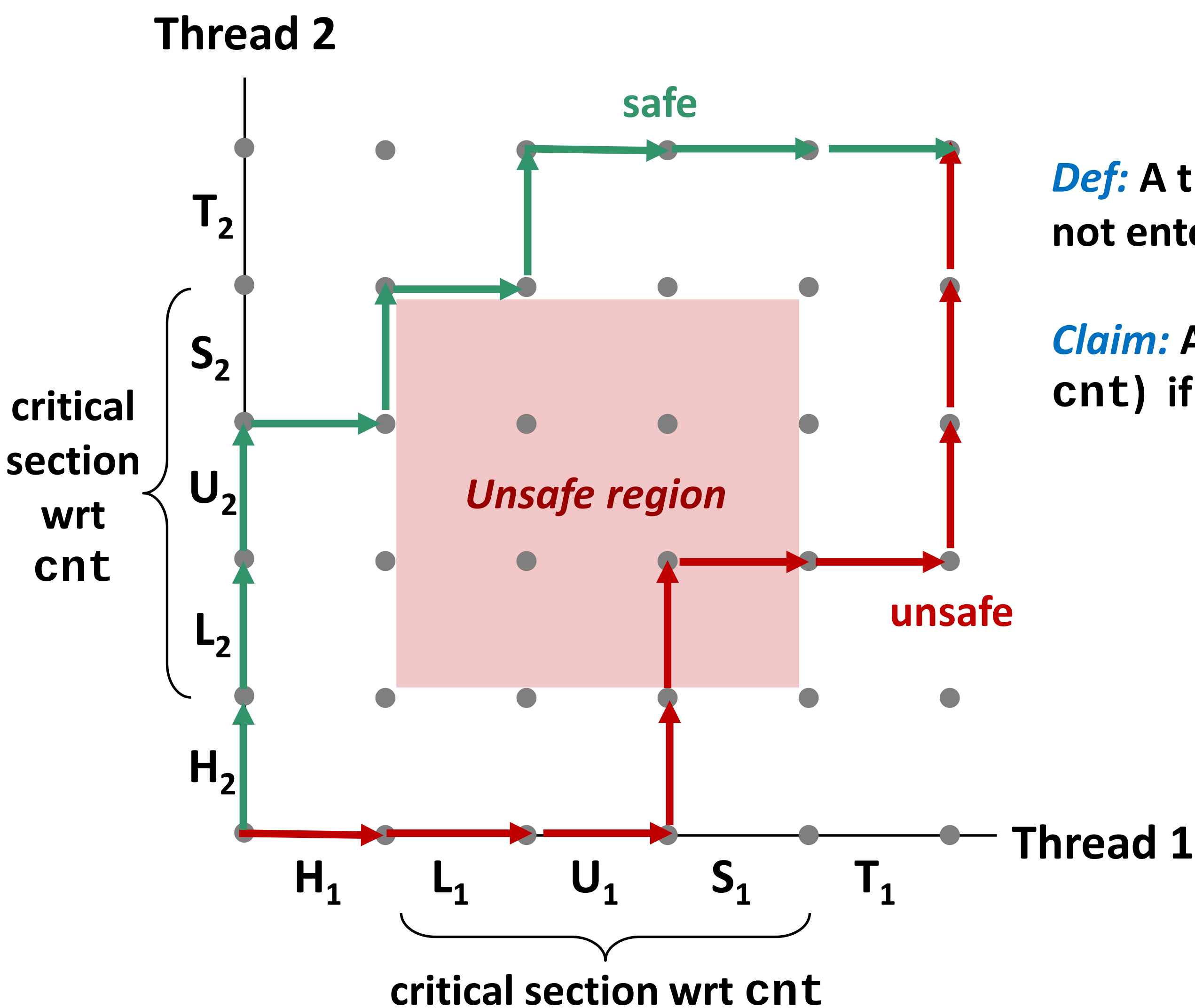


L, U, and S form a *critical section* with respect to the shared variable `cnt`

Instructions in critical sections (wrt some shared variable) should not be interleaved

Sets of states where such interleaving occurs form *unsafe regions*

Critical Sections and Unsafe Regions



Def: A trajectory is ***safe*** if it does not enter any unsafe region

Claim: A trajectory is correct (wrt `cnt`) if it is safe

badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
               *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

Variable	main	thread1	thread2
cnt	yes*	yes	yes
niters.m	yes	no	no
tid1.m	yes	no	no
i.1	no	yes	no
i.2	no	no	yes
niters.1	no	yes	no
niters.2	no	no	yes

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        — critical section
mov %eax, (0x123)
```

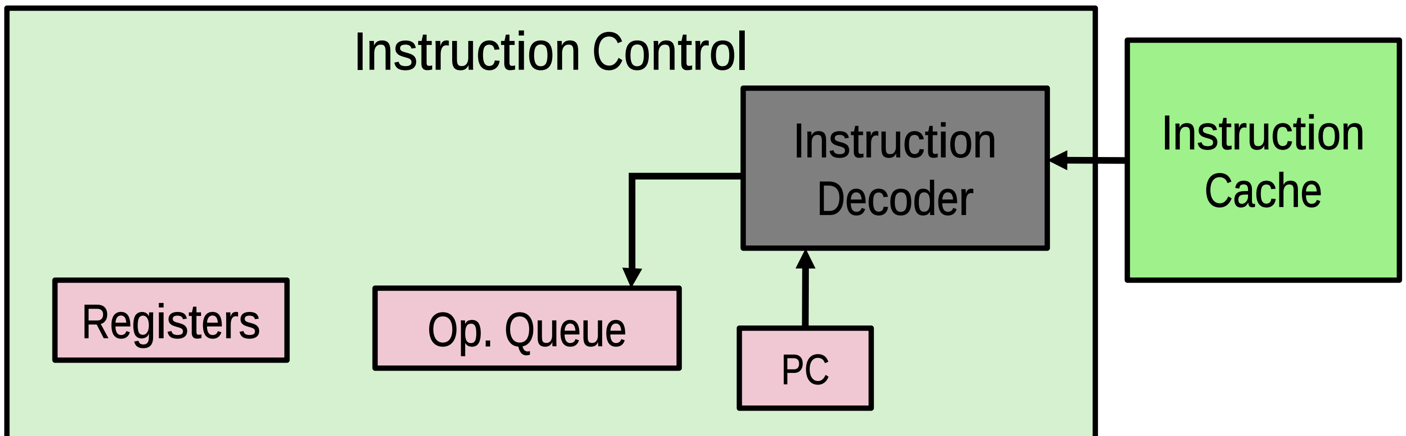
- More general: Need mutual exclusion for critical sections
- if process A is in critical section C, process B can't
    - (okay if other processes do unrelated work)

Single Instruction?

- **INC (0x123)**

Increase balance by one. (尽管只有一条汇编，但 CPU 可能分步执行，因此不能解决原子性问题)

- **Can inc instruction avoids races?**



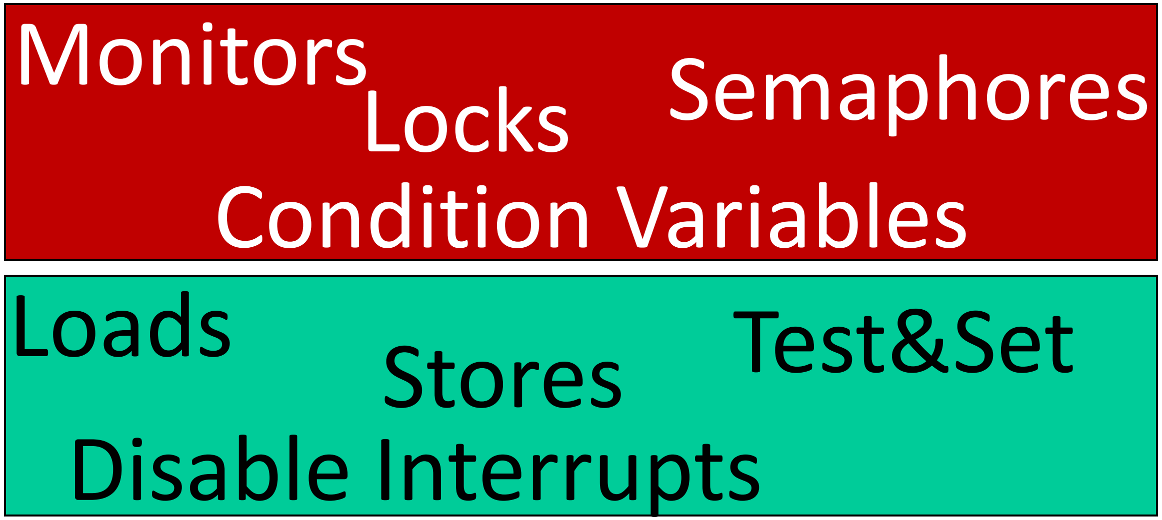
  - It depends.
  - Depends on if the CPU ISA treats single instruction atomic.
- **Lock prefix may help**
  - For x86, adding the lock prefix making sure the atomicity of the instruction.

Correct Concurrency is difficult

- **Compiler Optimization**
  - Compiler assumes nobody else is modifying memory
  - It may combine, split memory accesses to the same address
  - You may need "volatile" to direct the compiler to not think so
- **Compiler Re-ordering**
  - Compiler may reorder memory accesses to different addresses
  - You may need compiler barrier to prevent compiler reordering
    任何的重排序不会越过 barrier
- **CPU reordering**
  - In runtime, CPU may reorder memory accesses
  - You may need memory fence to restrict re-ordering
    赋值语句的重排序不会越过 fence

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**



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 to lock; 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