OSTEP
Concurrency:
Threads

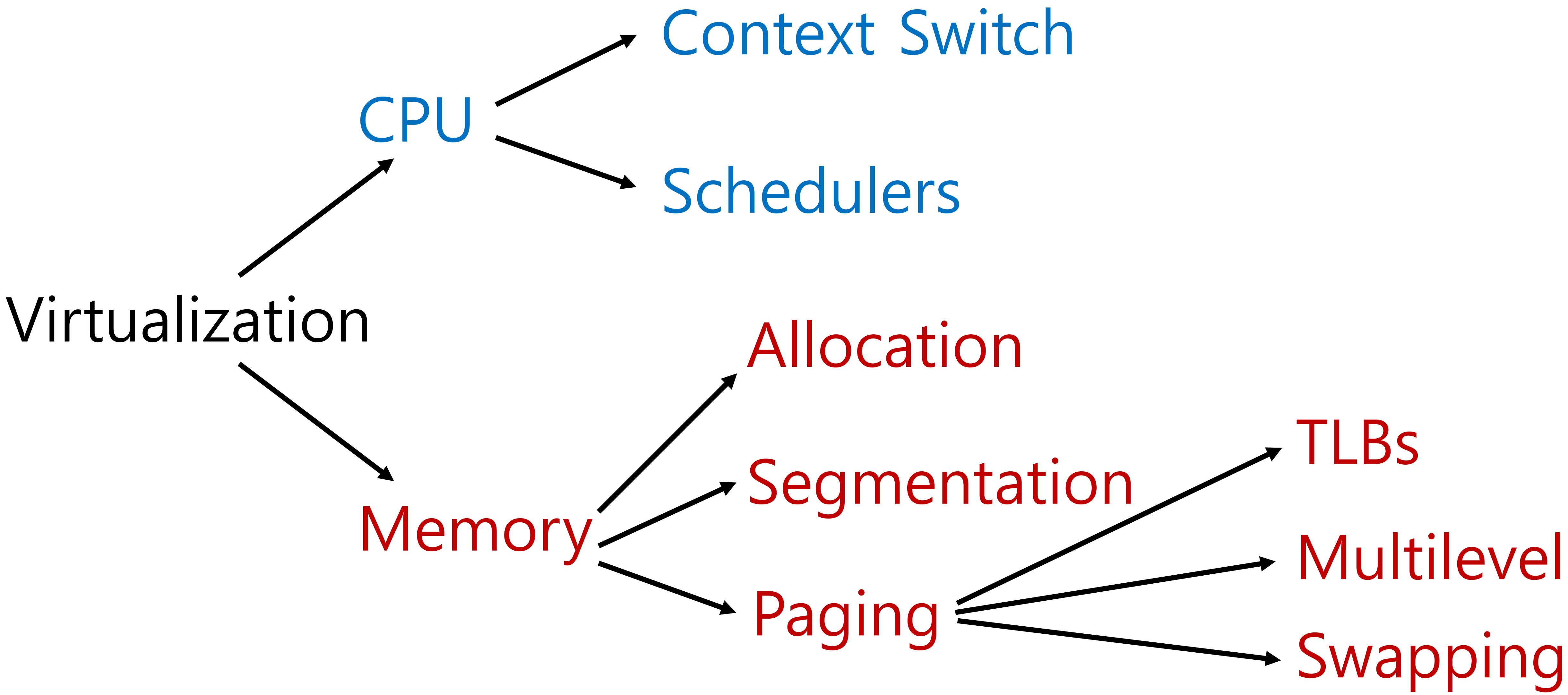
Questions answered in this lecture:

Why is concurrency useful?

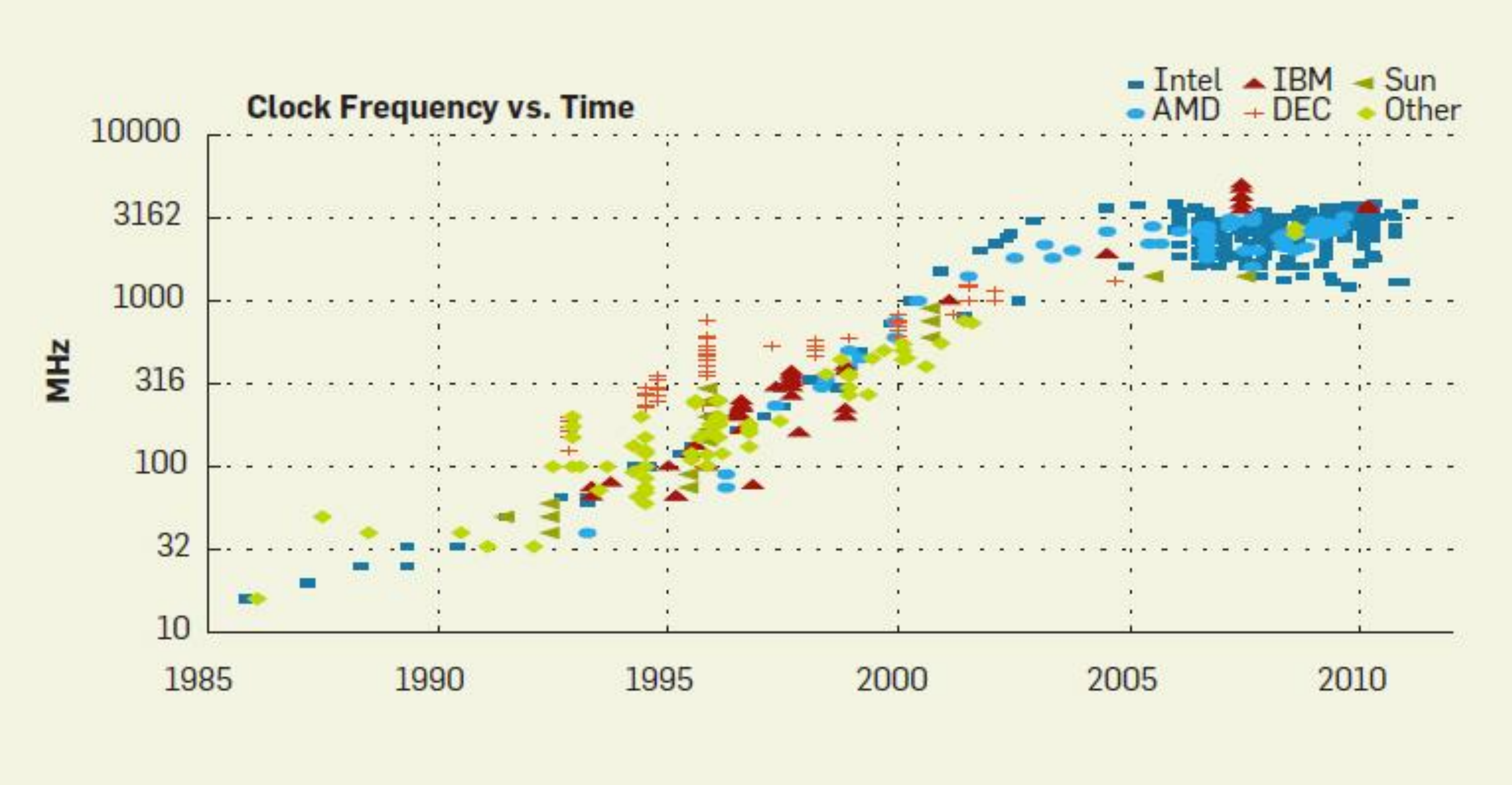
What is a thread and how does it differ from processes?

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

Review: Easy Piece 1



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: 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

Thread is a new abstraction for a single running process:

Threads are like processes, except:
multiple threads of same process share an address space (e.g., using same PT).

Divide large task across several cooperative threads

Communicate through shared address space

Context switch between threads

Each thread has its own <u>program counter</u> and <u>set of registers</u>.

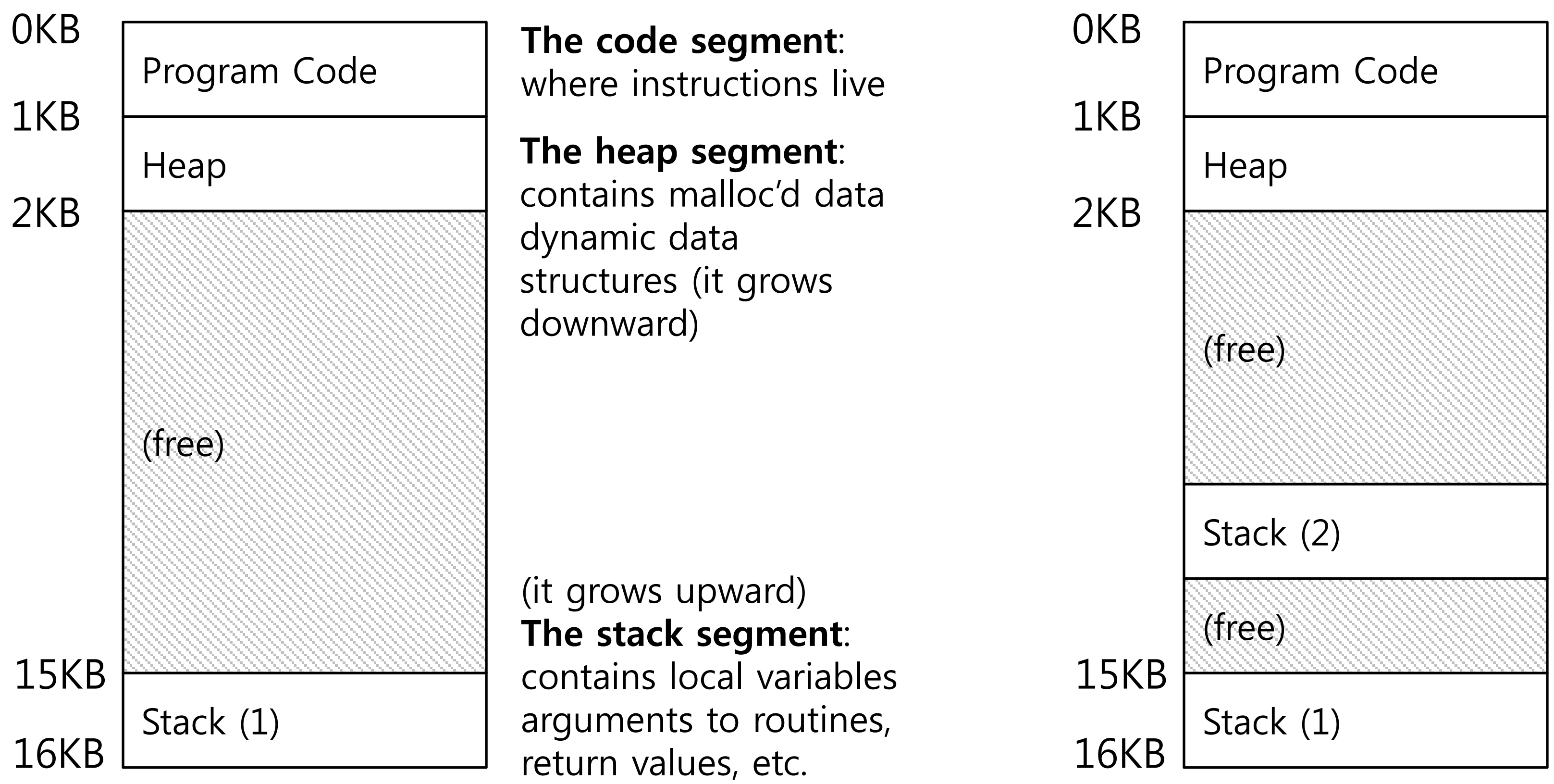
- One or more **thread control blocks(TCBs)** are needed to store the state of each thread.

When switching from running one (T1) to running the other (T2),

- The register state of T1 be saved.
- The register state of T2 restored.
- The address space remains the same.

The stack of the relevant thread

There will be one stack per thread.



A Single-Threaded Address Space

Two threaded Address Space

Race condition

Example with two threads

- counter = counter + 1 (default is 50)
- We expect the result is 52. However,

OS	Thread1	Thread2	(after instruction) PC	%eax	counter
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

Critical section

A piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.

- Multiple threads executing critical section can result in a race condition.
- Need to support **atomicity** for critical sections (**mutual exclusion**)

Locks

Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

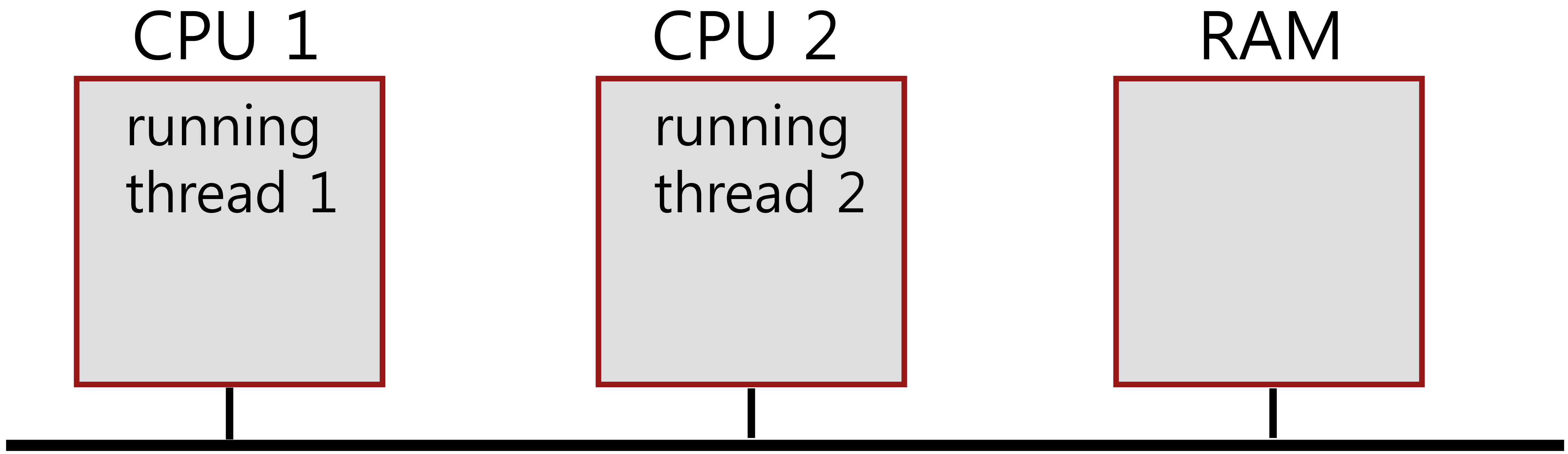
Common Programming Models

Multi-threaded programs tend to be structured as:

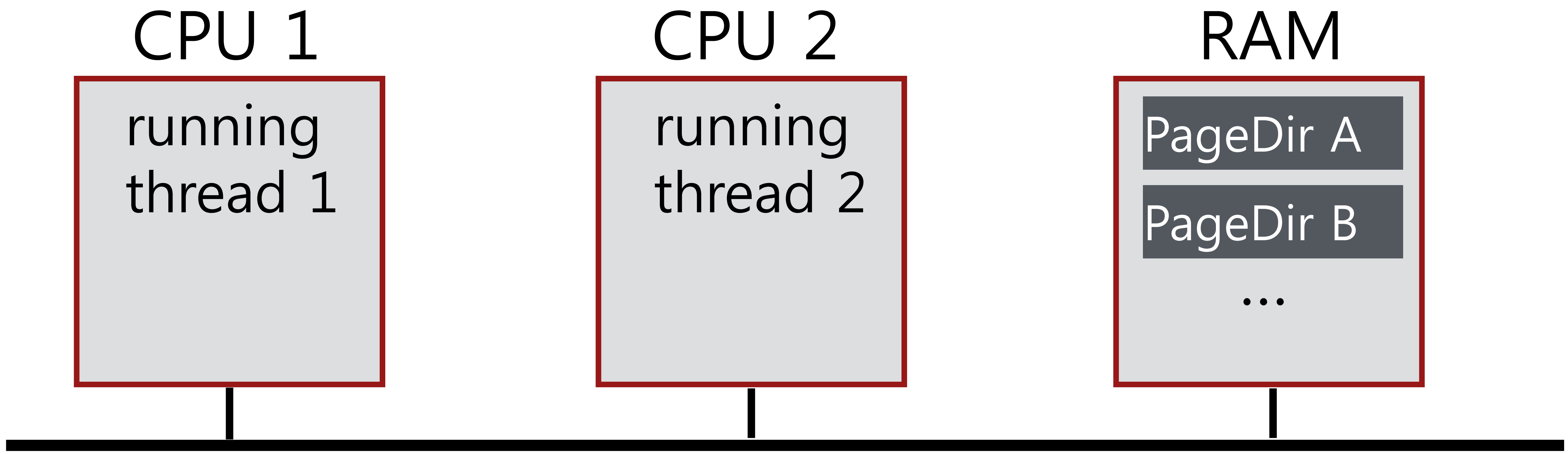
- **Producer/consumer**
  Multiple producer threads create data (or work) t hat is handled by one of the multiple consumer t hreads

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

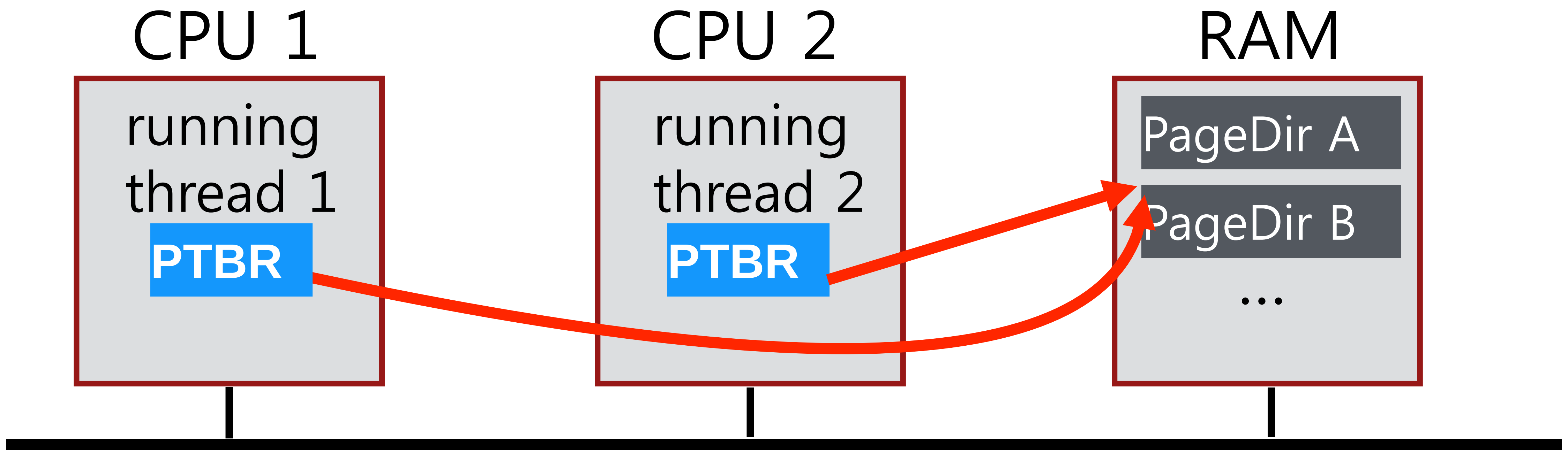
- **Defer work with background thread**
  One thread performs non-critical work in the bac kground (when CPU idle)

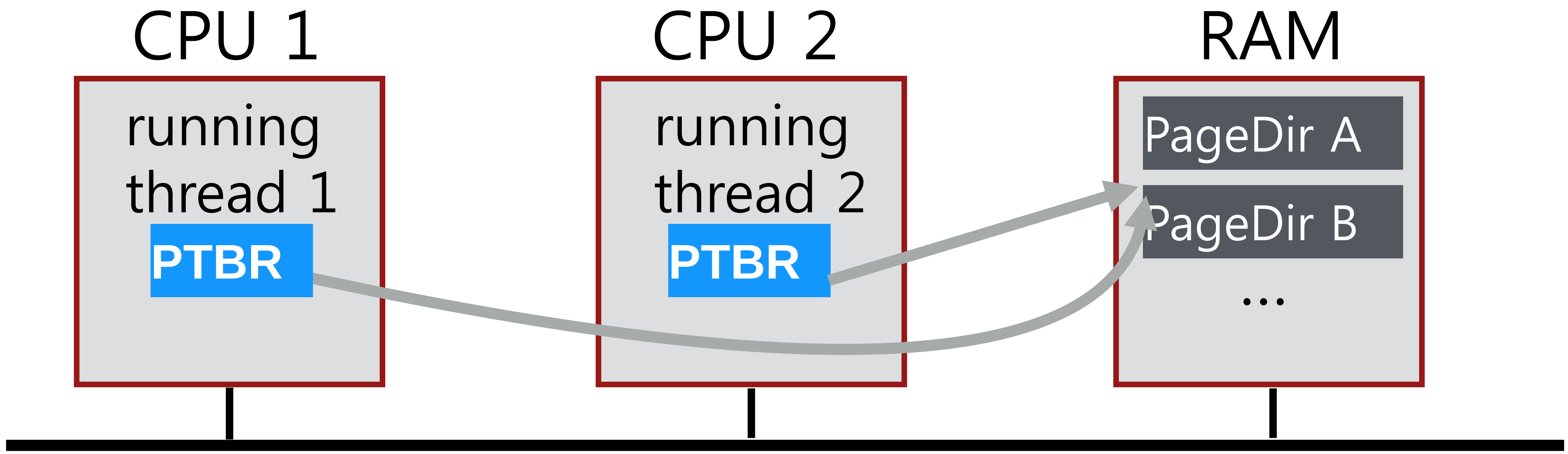


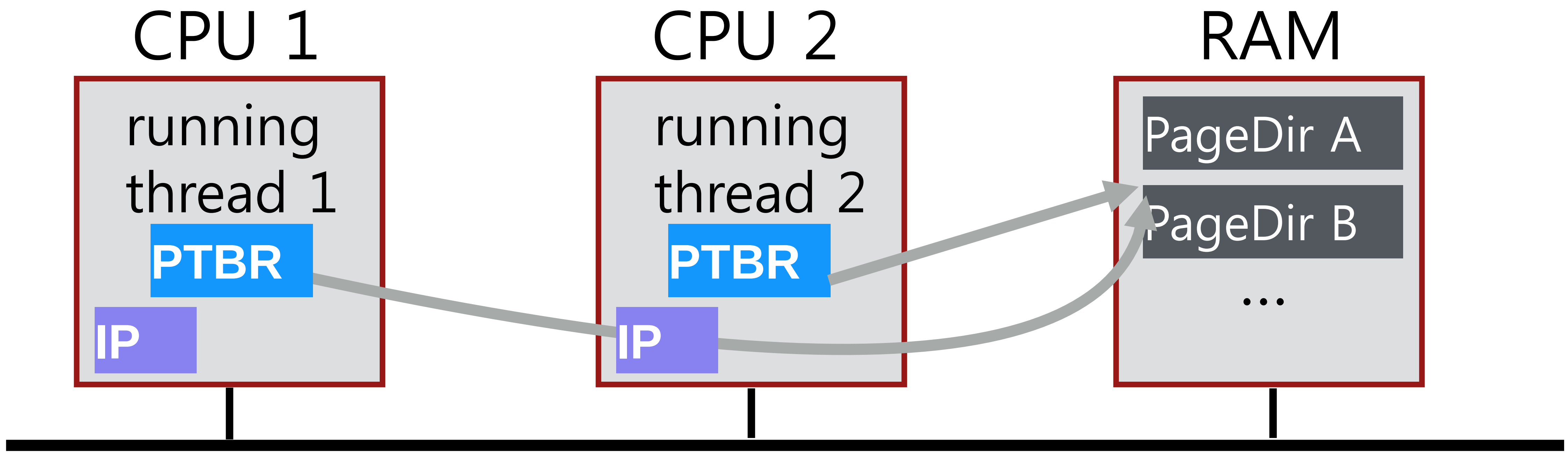
What state do threads share?



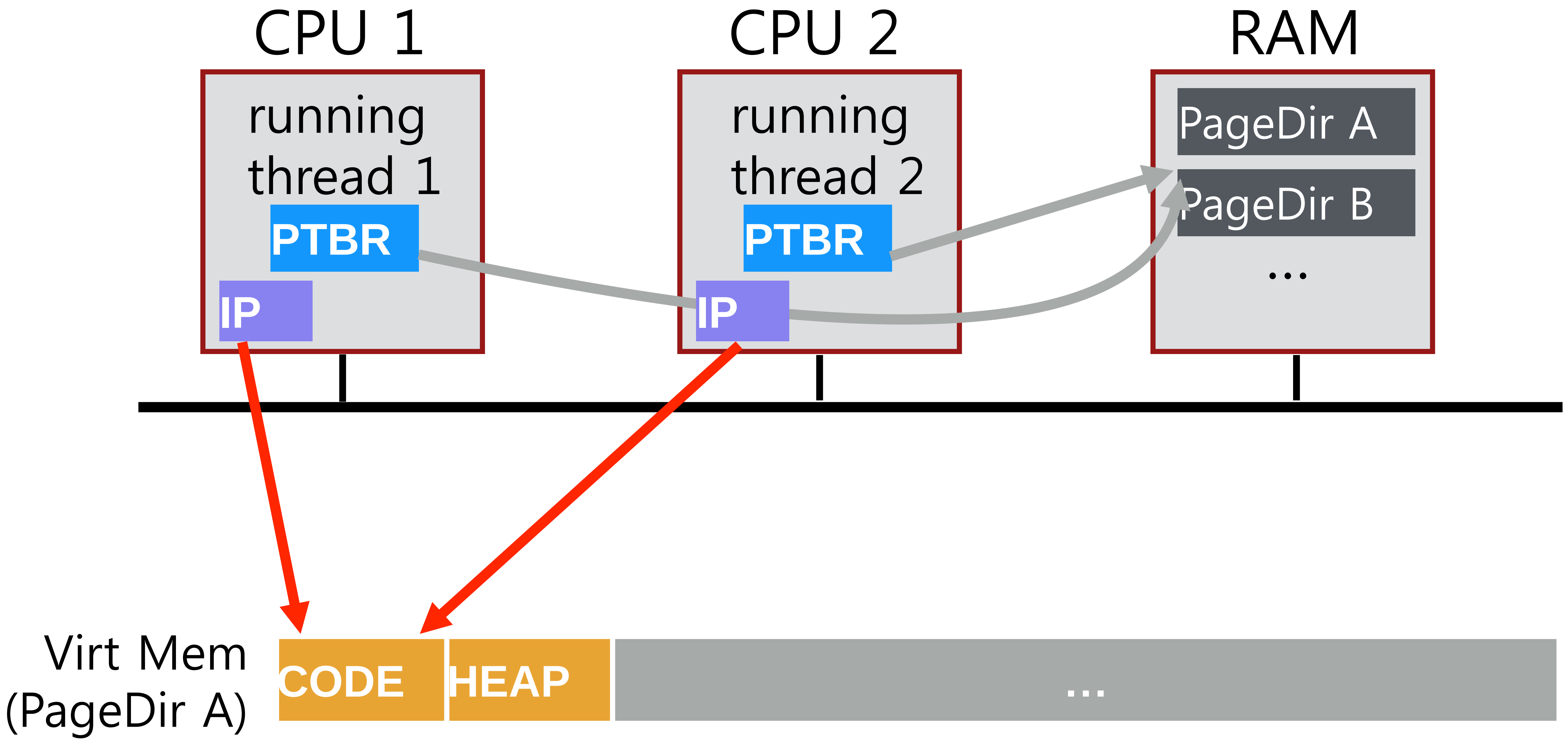
What threads share page directories?

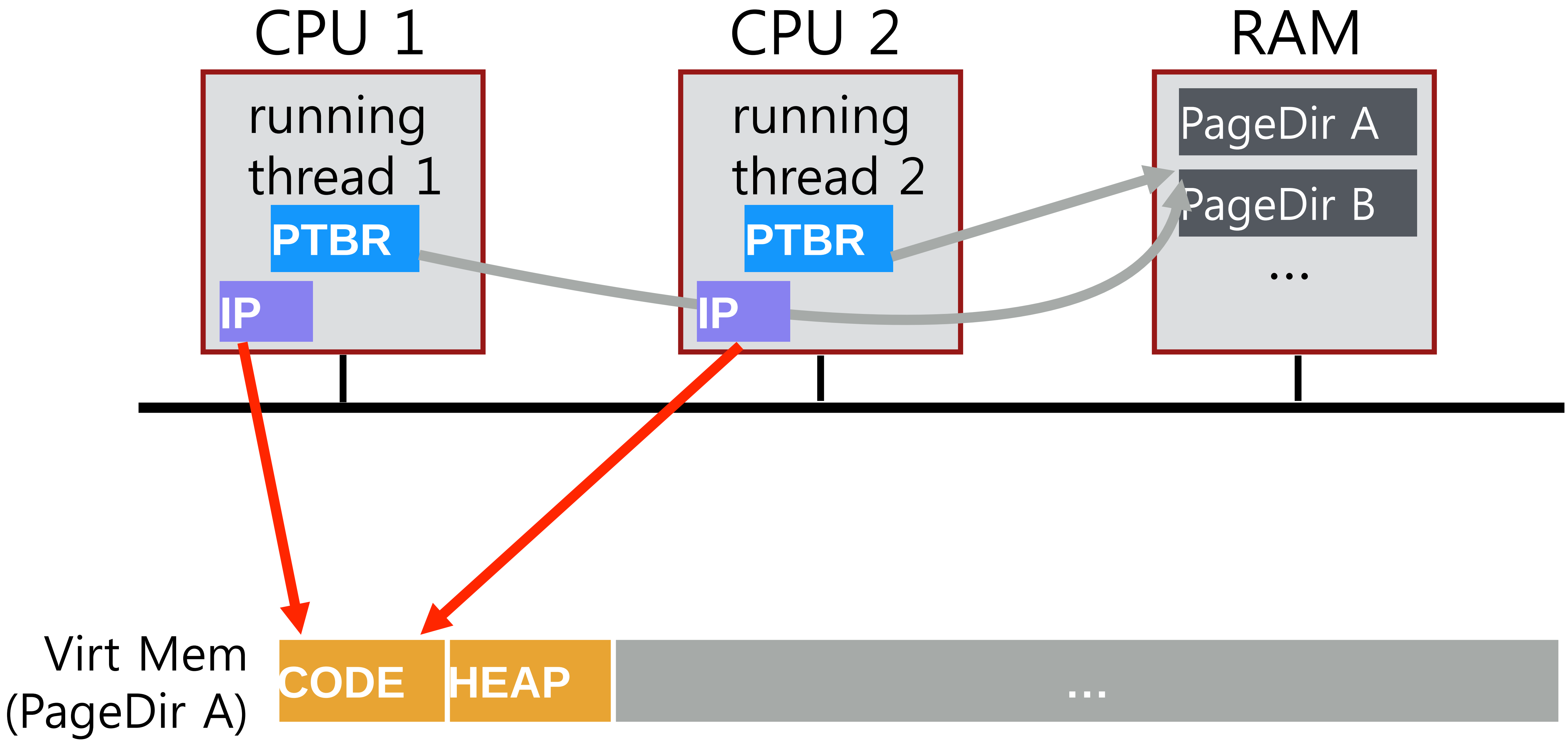

CPU 1
running thread 1
PTBR
CPU 2
running thread 2
PTBR
RAM
PageDir A
PageDir B
...


CPU 1
running thread 1
PTBR
CPU 2
running thread 2
PTBR
RAM
PageDir A
PageDir B
...



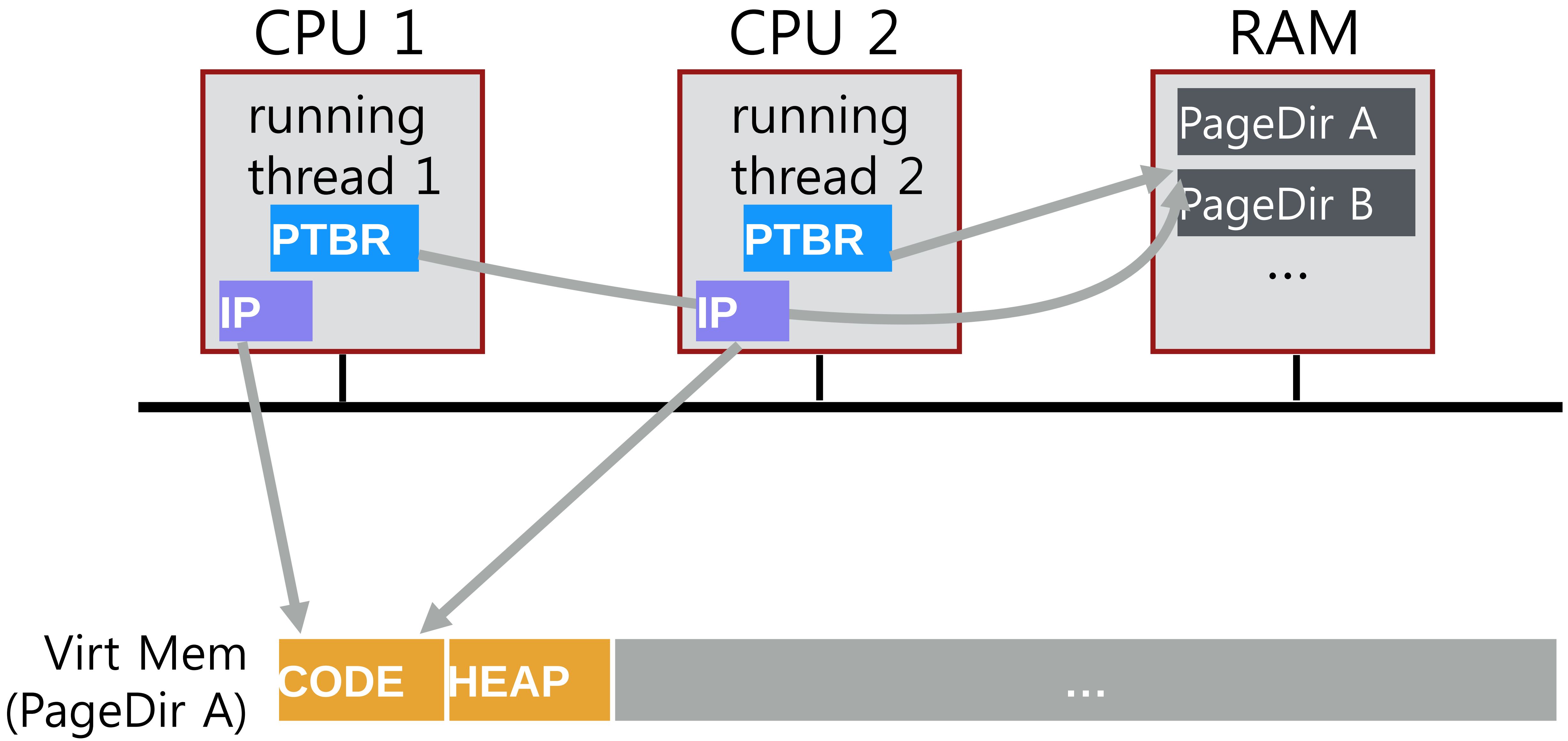
Do threads share Instruction Pointer?

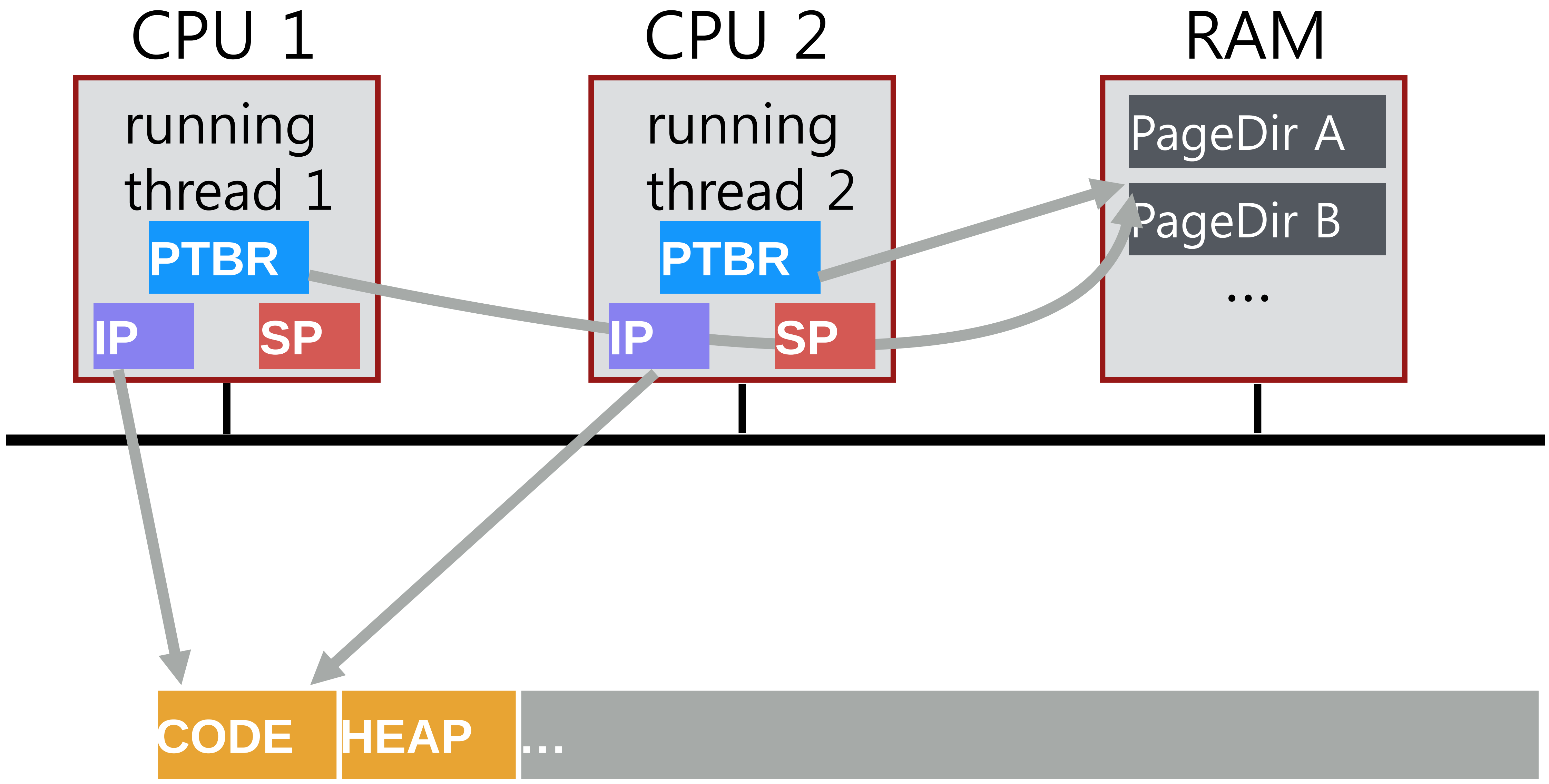

CPU 1
running thread 1
PTBR
IP
CPU 2
running thread 2
PTBR
IP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir A)
CODE
HEAP
...



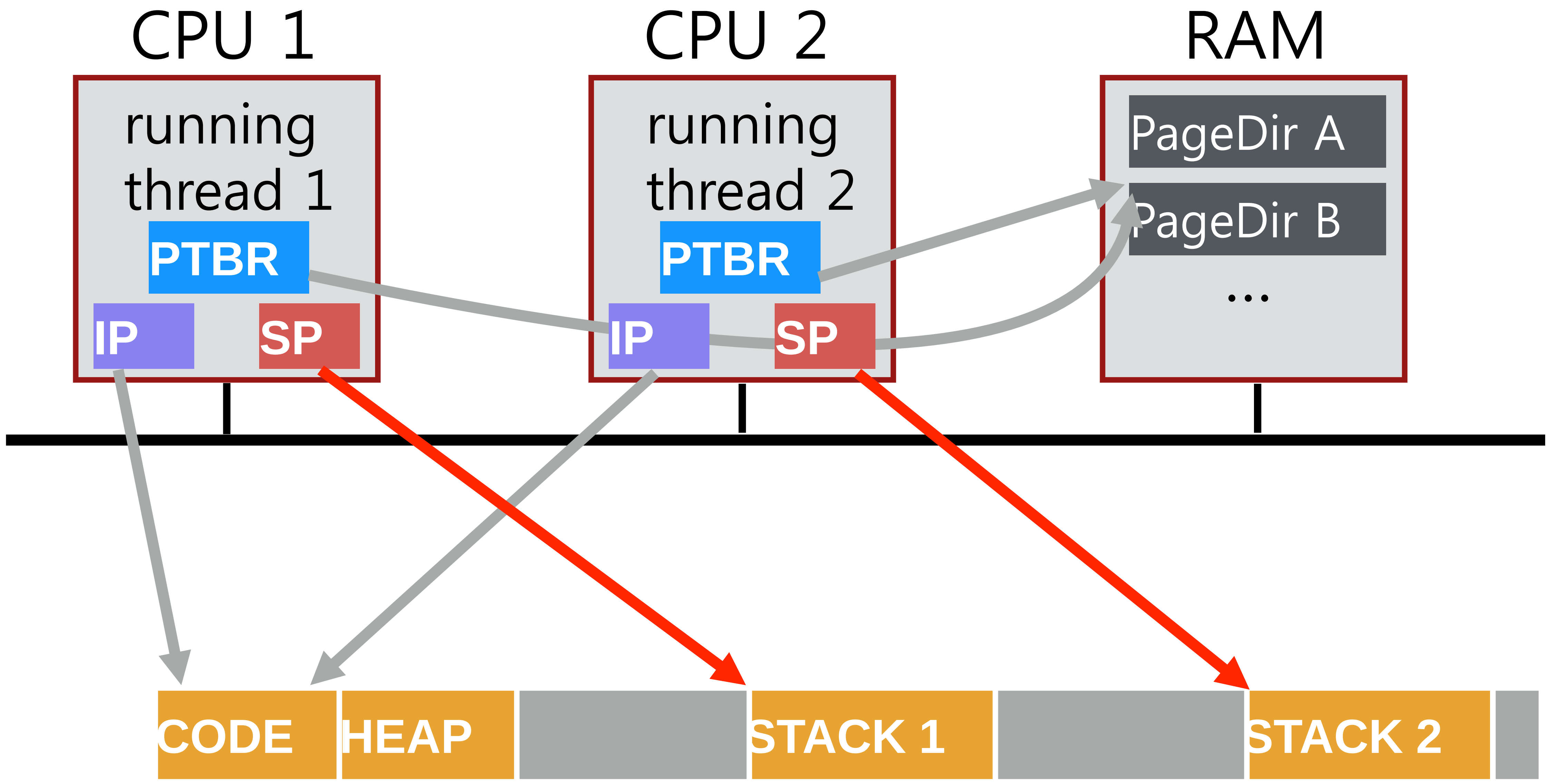
Share code, but each thread may be executing different code at the same time

→ Different Instruction Pointers

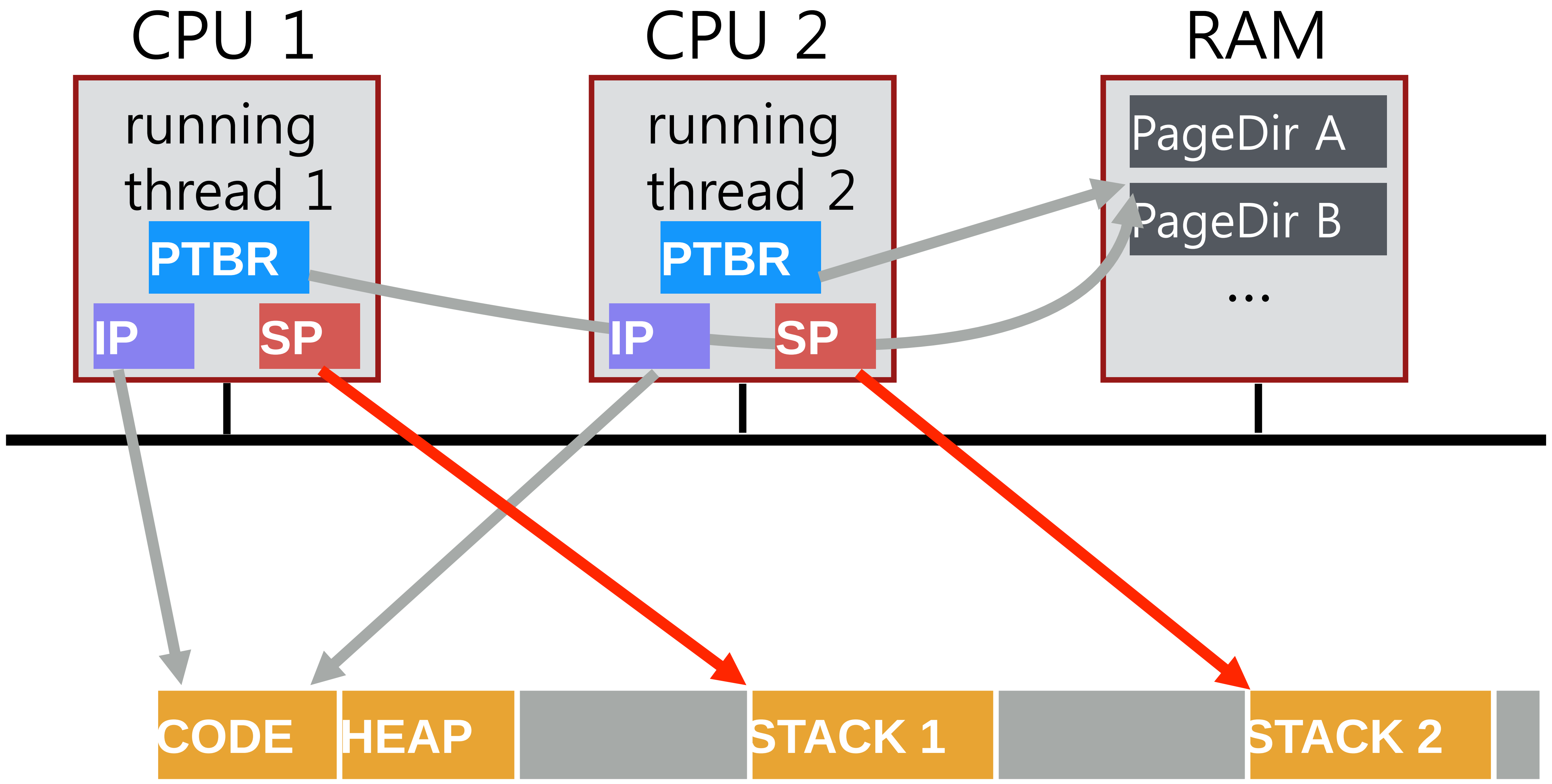

CPU 1
running thread 1
PTBR
IP
CPU 2
running thread 2
PTBR
IP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir A)
CODE
HEAP
...



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
...
CODE
HEAP
STACK 1
STACK 2



threads executing different functions need different stacks

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)

Thread Creation

How to create and control threads?

```
#include <pthread.h>

int
pthread_create(      pthread_t*       thread,
               const pthread_attr_t* attr,
                     void*            (*start_routine)(void*),
                     void*            arg);
```

- `thread`: Used to interact with this thread.
- `attr`: Used to specify any attributes this thread might have.
  - Stack size, Scheduling priority, …
- `start_routine`: the function this thread start running in.
- `arg`: the argument to be passed to the function (`start routine`)
  - *a void pointer* allows us to pass in *any type of* argument.

Thread Creation (Cont.)

If `start_routine` instead required another type argument, the declaration would look like this:

- An integer argument:

```
int
pthread_create(…, // first two args are the same
                  void*   (*start_routine)(int),
                  int     arg);
```

- Return an integer:

```
int
pthread_create(…, // first two args are the same
                  int   (*start_routine)(void*),
                  void*    arg);
```

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
       int a;
       int b;
} myarg_t;

void *mythread(void *arg) {
       myarg_t *m = (myarg_t *) arg;
       printf(“%d %d\n”, m->a, m->b);
       return NULL;
}

int main(int argc, char *argv[]) {
       pthread_t p;
       int rc;

       myarg_t args;
       args.a = 10;
       args.b = 20;
       rc = pthread_create(&p, NULL, mythread, &args);
       …
}
```

Wait for a thread to complete

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- `thread`: Specify which thread *to wait for*
- `value_ptr`: A pointer to the return value
  - Because `pthread_join()` routine changes the value, you need to pass in a pointer to that value.

Example: Waiting for Thread Completion

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example: Waiting for Thread Completion (Cont.)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                                     // waiting inside of the
                               // pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Example: Dangerous code

Be careful with how values are returned from a thread.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

- When the variable `r` returns, it is automatically de-allocated.

Example: Simpler Argument Passing to a Thread

Just passing in a single value

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

Locks

Provide mutual exclusion to a critical section

- Interface

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

- No other thread holds the lock → the thread will acquire the lock and enter the critical section.
- If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

All locks must be properly initialized.

- One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks (Cont.)

Check errors code when calling lock and unlock

- An example wrapper

```
•// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

These two calls are used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- trylock: return failure if the lock is already held
- timelock: return after a timeout

Locks (Cont.)

These two calls are also used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

- `trylock`: return failure if the lock is already held
- `timelock`: return after a timeout or after acquiring the lock

Condition Variables

Condition variables are useful when some kind of signaling must take place between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait`:
  - Put the calling thread to sleep.
  - Wait for some other thread to signal it.
- `pthread_cond_signal`:
  - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
      pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- The wait call releases the lock when putting said caller to sleep.
- Before returning after being woken, the wait call re-acquire the lock.

A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
      pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.

Condition Variables (Cont.)

Don't ever to this.

- A thread calling wait routine:

```
while(initialized == 0)
        ; // spin
```

- A thread calling signal routine:

```
initialized = 1;
```

- It performs poorly in many cases. → just wastes CPU cycles.
- It is error prone.

Compiling and Running

To compile them, you must include the header `pthread.h`

- Explicitly link with the pthreads library, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

- For more information,

```
man -k pthread
```

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 schedu led

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

T1 →

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

Thread SchedulE #1

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

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

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

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

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

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 Context Switch

Thread SchedulE #1

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

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

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

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

process control blocks:

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

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: 0x195

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

T2

Thread Schedule #2

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

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

T2

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

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
- 0x19d mov %eax, 0x9cd4

T1

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	101
%rip	0x1a2	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	**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

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?

1: incorrect!

Thread 1

mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123

Thread 2

mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

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…

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
```

critical section

More general:
Need mutual exclusion for critical sections

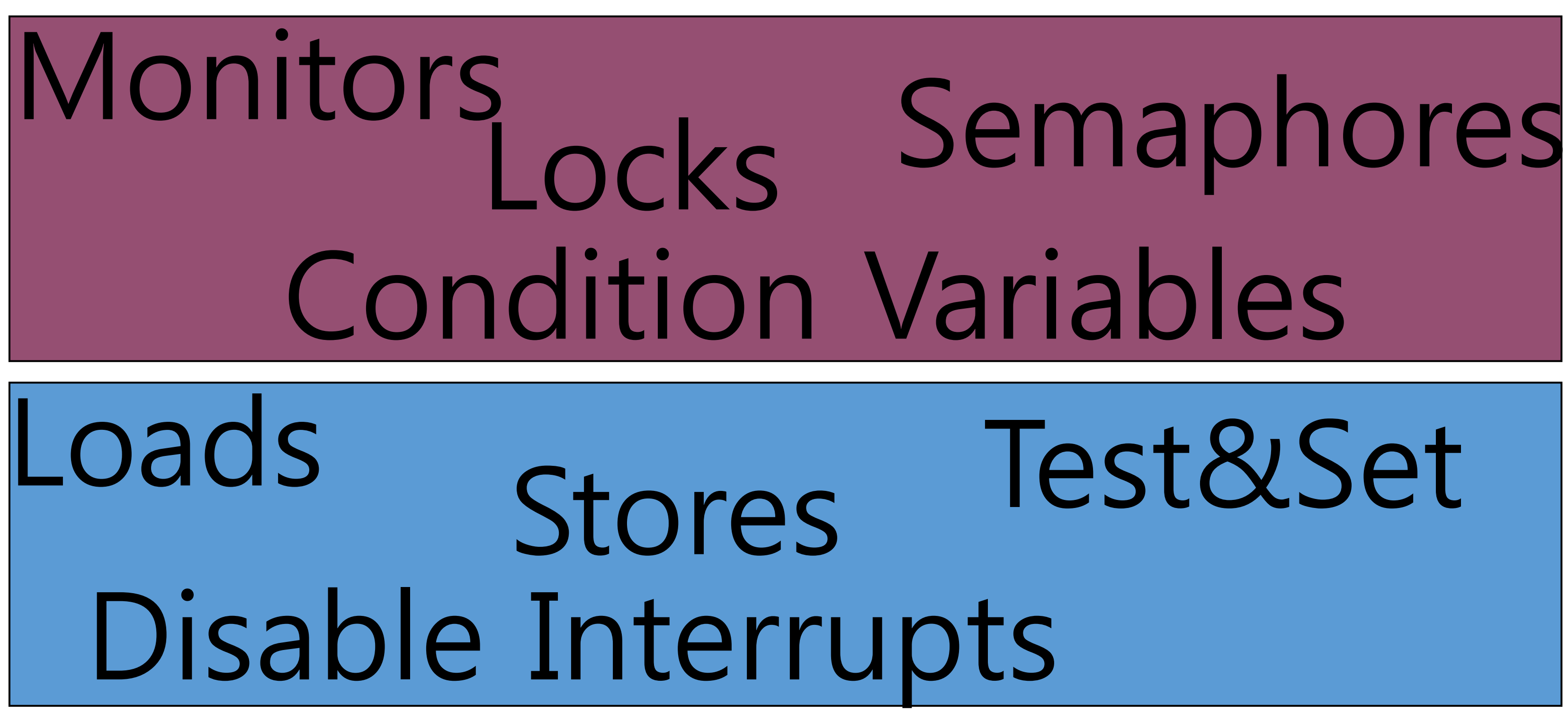
- if process A is in critical section C, process B can't
  (okay if other processes do unrelated work)

Synchronization

Build higher-level synchronization primitives in OS

- Operations that ensure correct ordering of instructions acr oss 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 se ction
  - Pthread_mutex_unlock(&mylock);

More Demos

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 a nd stack)

Context switches within a critical section can lead to non-deter ministic bugs (race conditions)

Use locks to provide mutual exclusion

Implementing Synchronization

To implement, need atomic operations

Atomic operation: No other instructions can be interleaved

Examples of atomic operations

- Code between interrupts on uniprocessors
  - Disable timer interrupts, don’t do any I/O
- Loads and stores of words
  - Load r1, B
  - Store r1, A
- **Special hw instructions**
  - **Test&Set**
  - **Compare&Swap**

Implementing Locks: Attempt #1

Turn off interrupts for critical sections

- Prevent dispatcher from running another thread
- Code executes atomically

```
Void acquire(lockT *l) {
	disableInterrupts();
}
Void release(lockT *l) {
	enableInterrupts();
}
```

Disadvantages??

Implementing LOCKS: Attempt #2

Code uses a single shared lock variable

```
Boolean lock = false; // shared variable
Void acquire() {
        while (lock) /* wait */ ;
        lock = true;
}

Void release() {
        lock = false;
}
```

Why doesn’t this work?