CS 471 Operating Systems

Yue Cheng

George Mason University

Fall 2019

Threads

Why Thread Abstraction?

Process Abstraction: Challenge 1

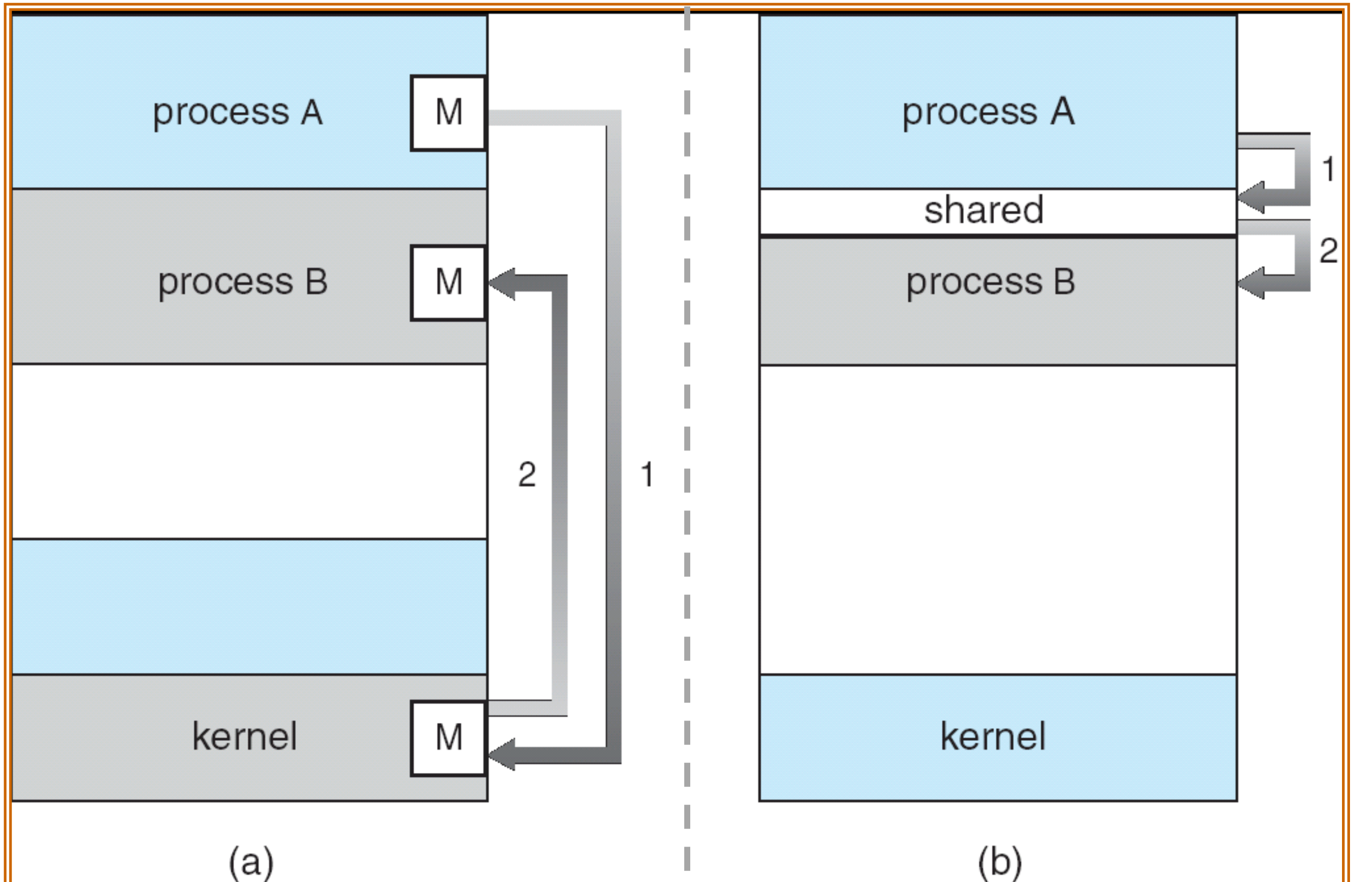
- Inter-process communication (IPC)

Inter-Process Communication

- Mechanism for processes to communicate and to synchronize their actions.

- Two models
  - Communication through a shared memory region
  - Communication through message passing

Communication Models



Message Passing

Shared Memory

Previously, in a distributed system, message-passing was the only possible communication model. However, remote direct memory access (RDMA) technique bridges this gap by providing remote memory access through network.

Communication through Message Passing

- Message system – processes communicate with each other without resorting to shared variables

- A message-passing facility must provide at least two operations:
  - `send(message, recipient)`
  - `receive(message, recipient)`

- With **indirect** communication, the messages are sent to and received from mailboxes (or, ports)
  - `send(A, message) /* A is a mailbox */`
  - `receive(A, message)`

Communication through Message Passing

- Message passing can be either **blocking** (synchronous) or **non-blocking** (asynchronous)
  - Blocking Send: The sending process is blocked until the message is received by the receiving process or by the mailbox
  - Non-blocking Send: The sending process resumes the operation as soon as the message is received by the kernel
  - Blocking Receive: The receiver blocks until the message is available
  - Non-blocking Receive: "Receive" operation does not block; it either returns a valid message or a default value (null) to indicate a non-existing message

Communication through Shared Memory

- The memory region to be shared must be explicitly defined
- System calls (Linux):
  - `shmget` creates a shared memory block
  - `shmat` maps/attaches an existing shared memory block into a process's address space
  - `shmdt` removes ("`unmaps`") a shared memory block from the process's address space
  - `shmctl` is a general-purpose function allowing various operations on the shared block (receive information about the block, set the permissions, lock in memory, …)
- Problems with simultaneous access to the shared variables
- Compilers for concurrent programming languages can provide direct support when declaring variables (e.g., "`shared int buffer`")

Process Abstraction: Challenge 1

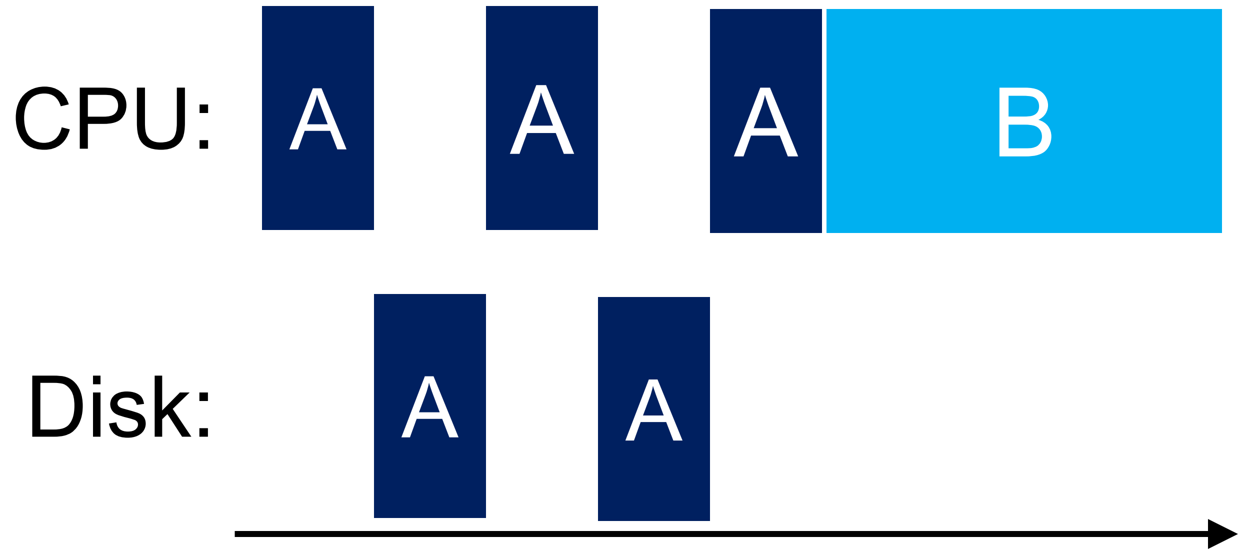
- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)

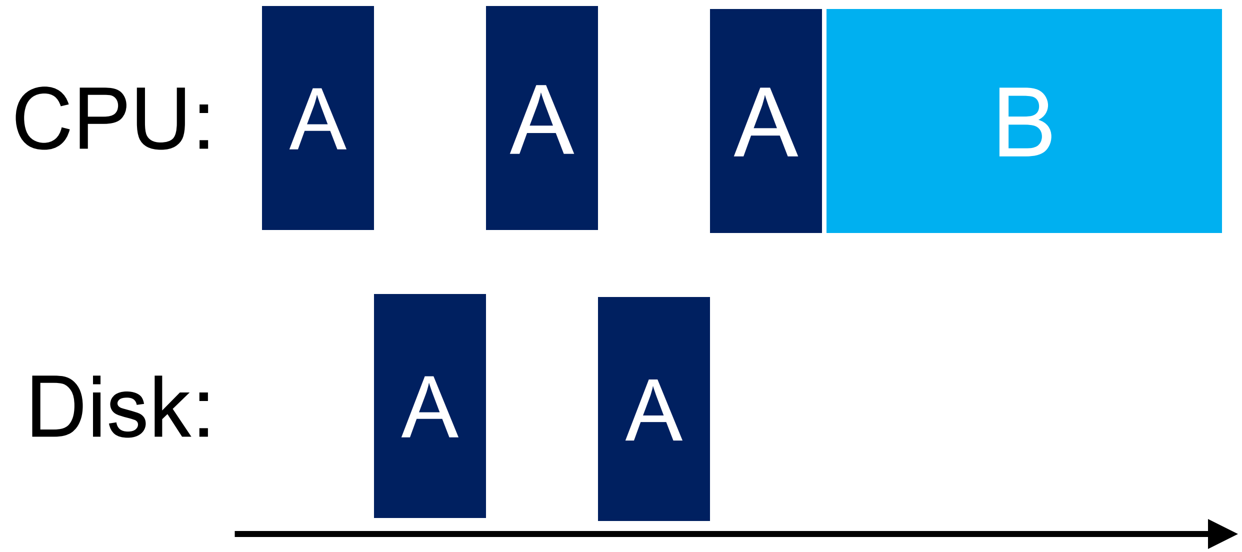
Process Abstraction: Challenge 2

- Inter-process communication (IPC)
  - Cumbersome programming!
  - Copying overheads (inefficient communication)
  - Expensive context switching (why expensive?)

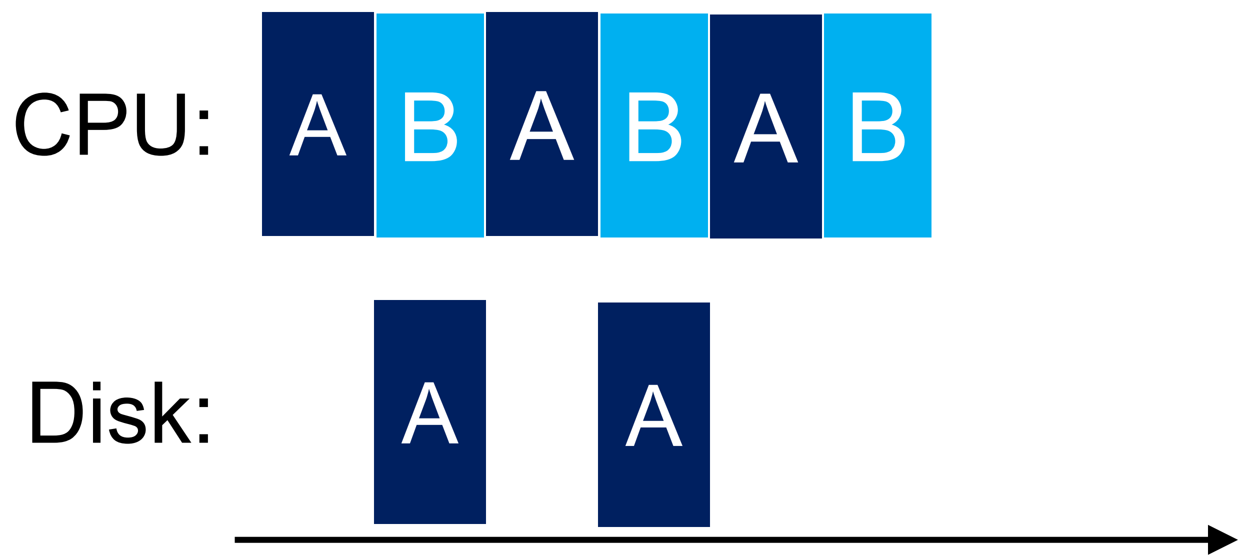
- **CPU utilization**

(a) Not interleaved

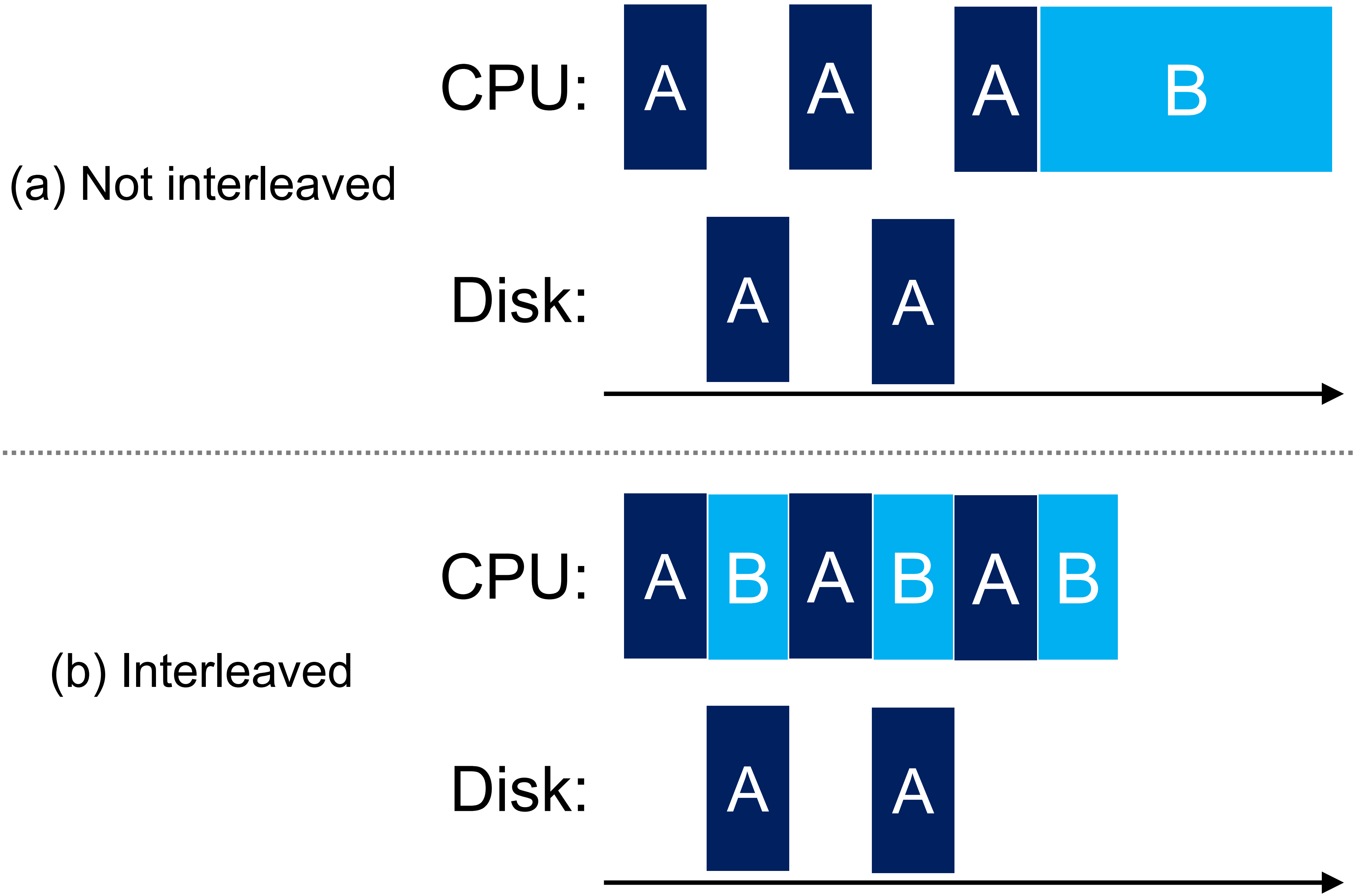




(a) Not interleaved

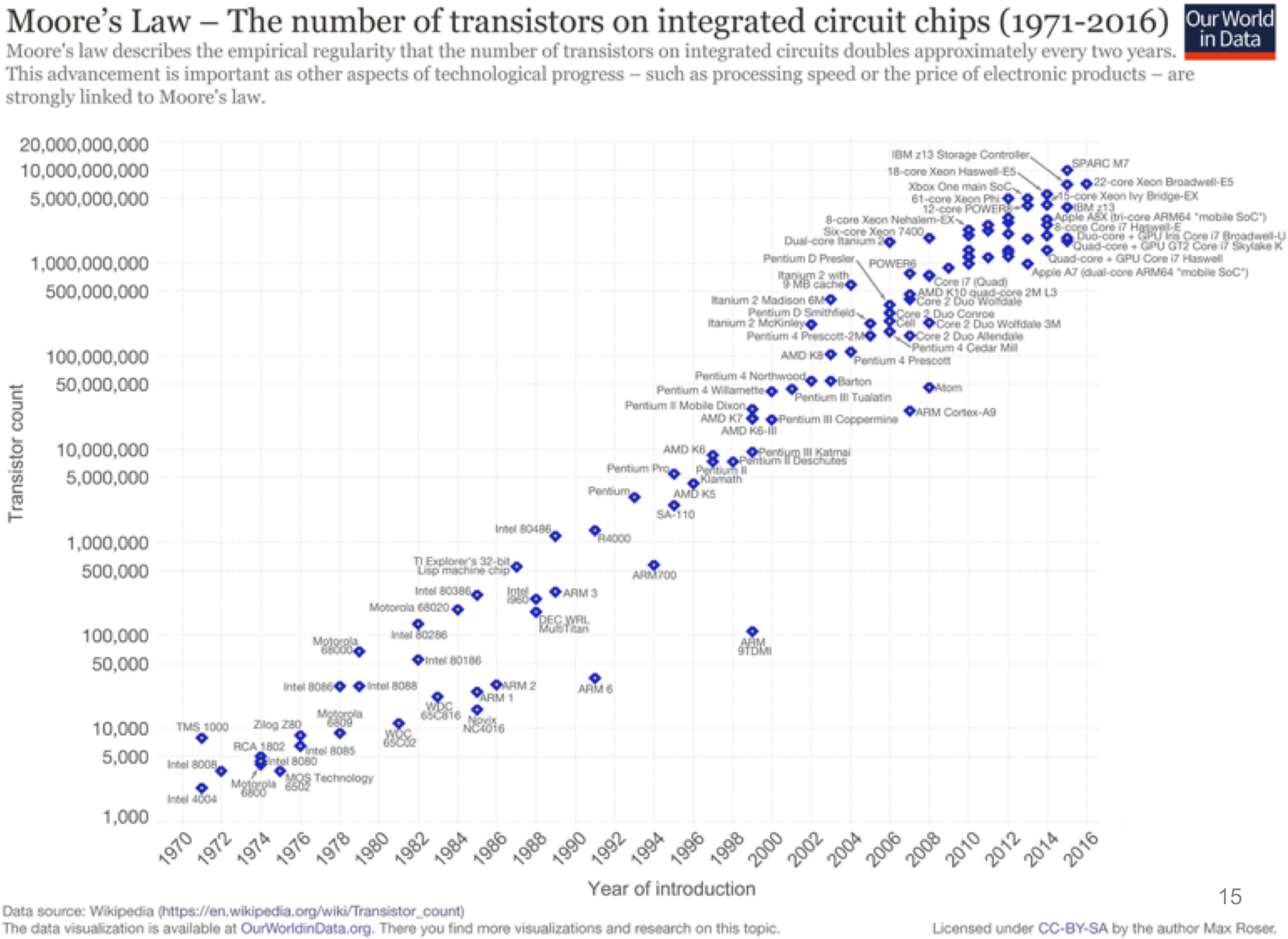


(b) Interleaved

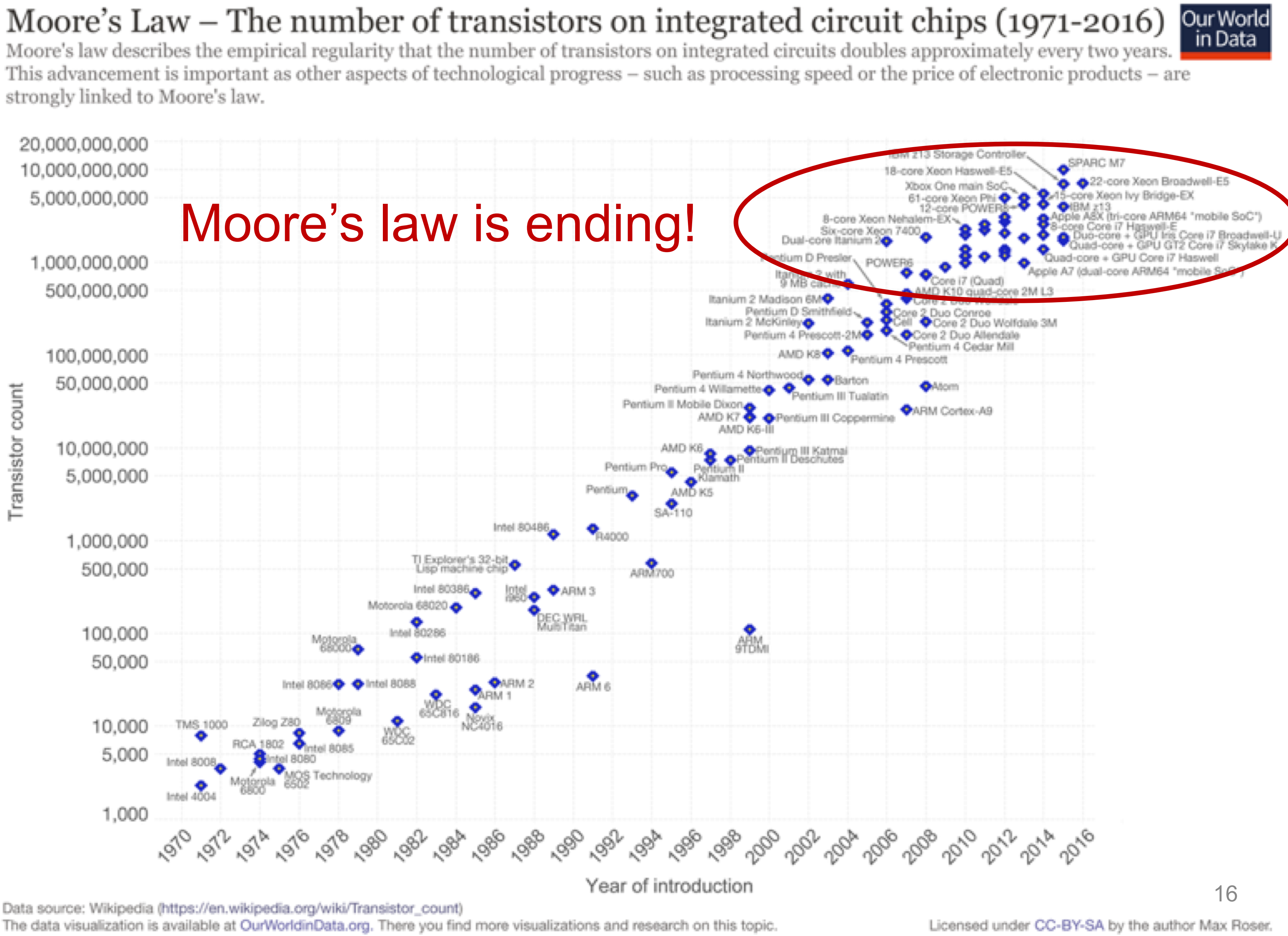


What if there is only one process?

Moore's law: # transistors doubles every ~2 years



Moore’s law: # transistors doubles every ~2 years



Moore’s law is ending!

CPU Trends – What Moore's Law Implies…

- The future
  - Same CPU speed
  - More cores (to scale-up)

- Faster programs => concurrent execution

- **Goal**: Write applications that fully utilize many CPU cores…

Goal

- Write applications that fully utilize many CPUs…

Strategy 1

- Build applications from many communication processes
  - Like Chrome (process per tab)
  - Communicate via `pipe()` or similar

- Pros/cons?

Strategy 1

- Build applications from many communication processes
  - Like Chrome (process per tab)
  - Communicate via `pipe()` or similar

- Pros/cons? – That we've talked about in previous slides
  - Pros: Don't need new abstractions!
  - Cons:
    - Cumbersome programming using IPC
    - Copying overheads
    - Expensive context switching

Strategy 2

- New abstraction: the **thread**

Introducing Thread Abstraction

- New abstraction: the **thread**

- Threads are just like processes, but threads share the address space

Thread

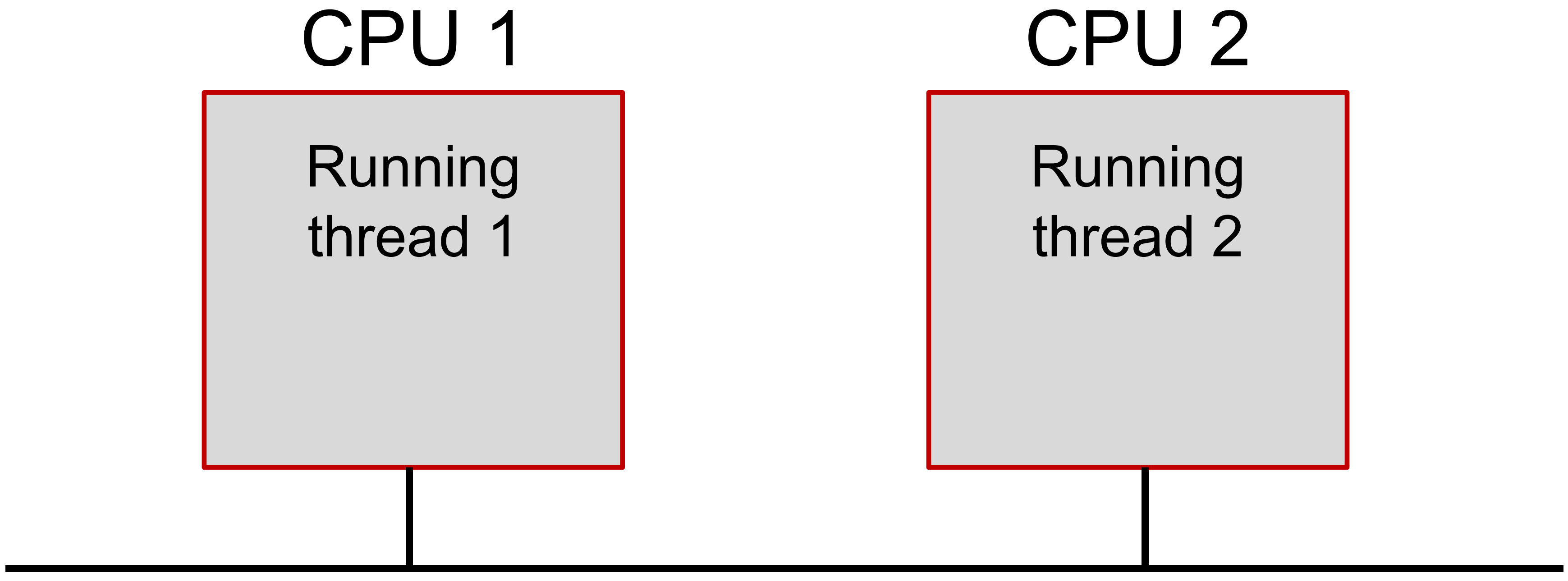
- A process, as defined so far, has only one thread of execution

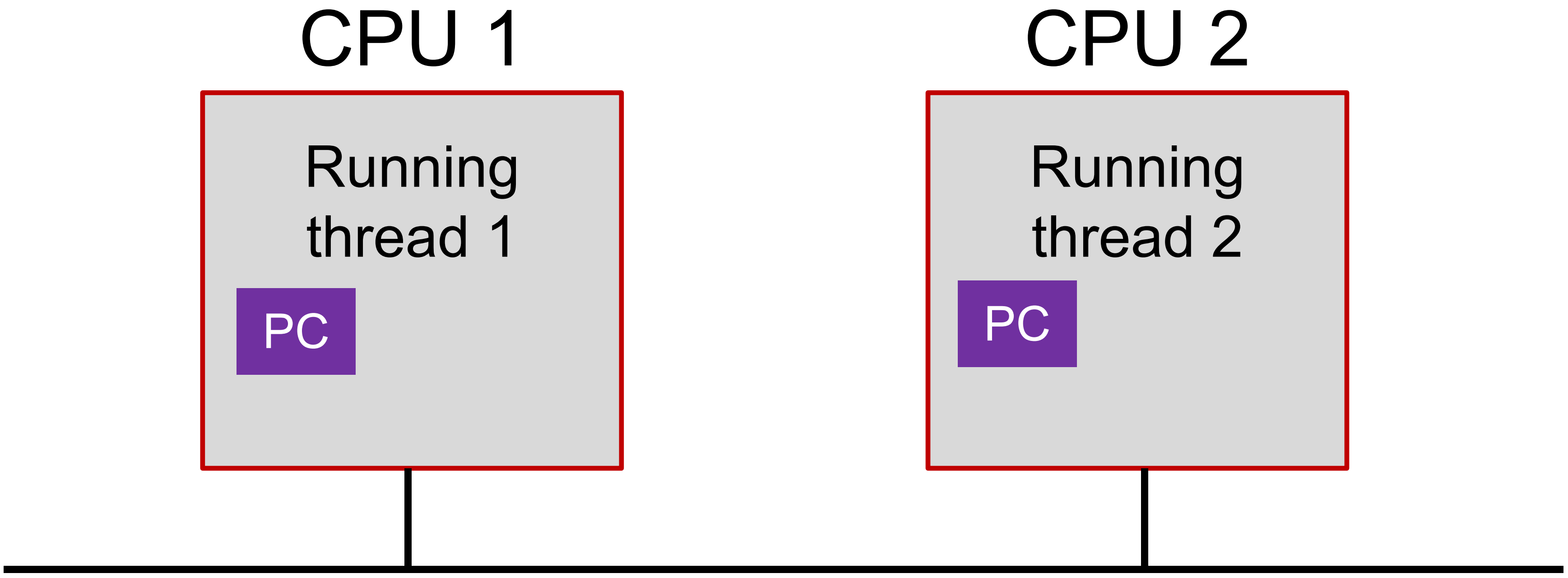
- **Idea**: Allow multiple threads of concurrently running execution within the same process environment, to a large degree independent of each other
  - Each thread may be executing different code at the same time

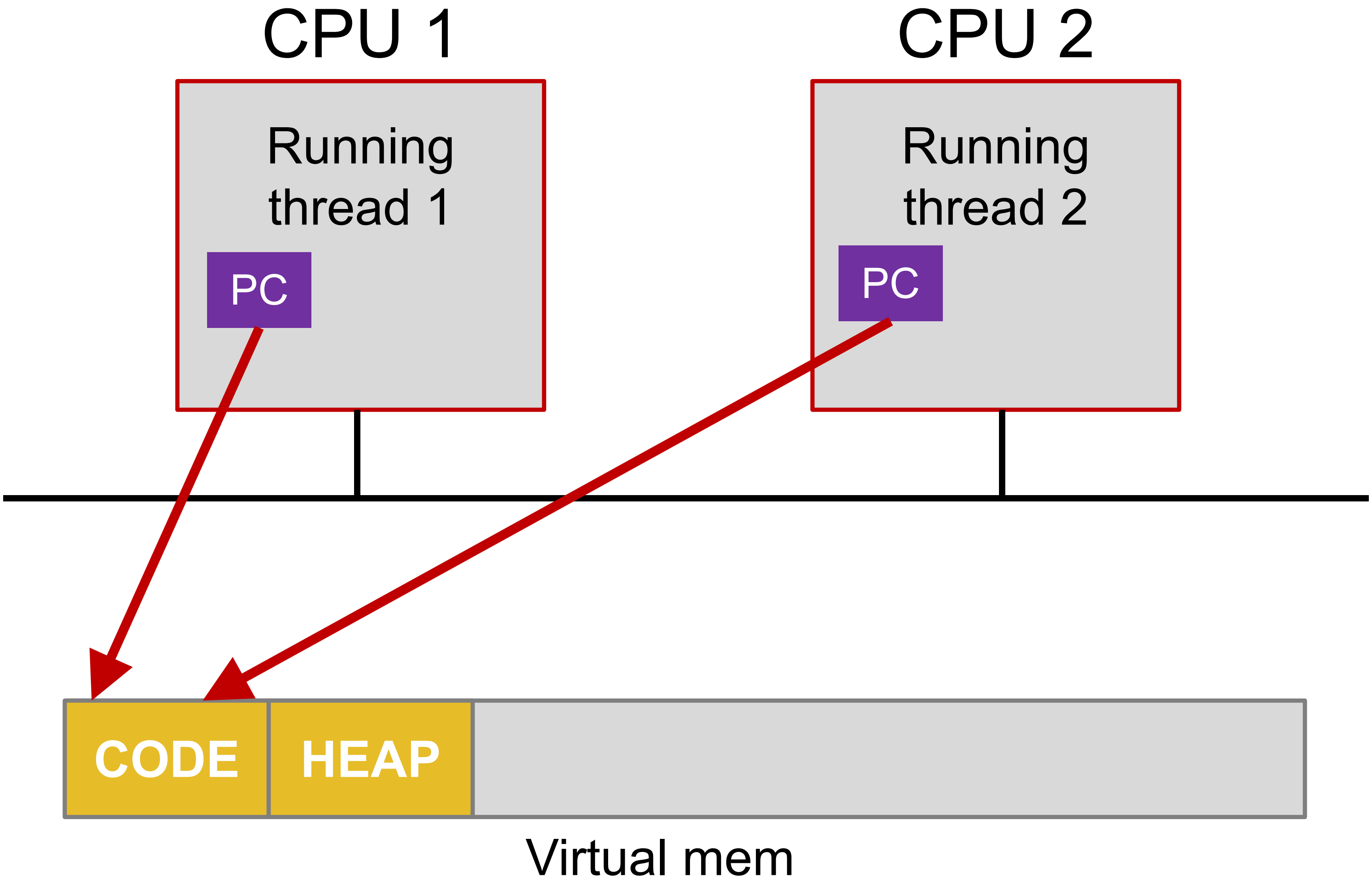
Process vs. Thread

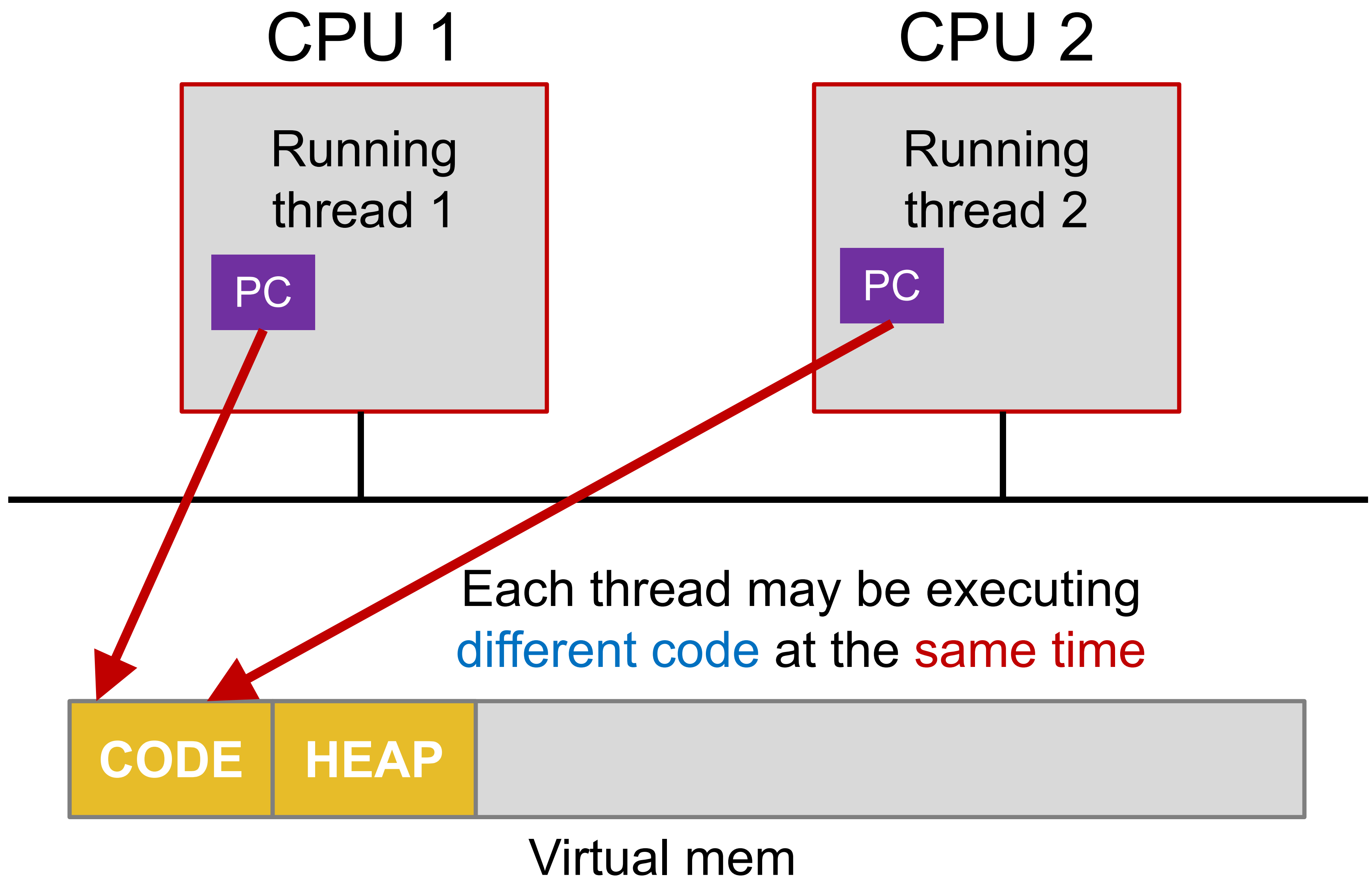
- Multiple threads within a process will share
  - The address space
  - Open files (file descriptors)
  - Other resources

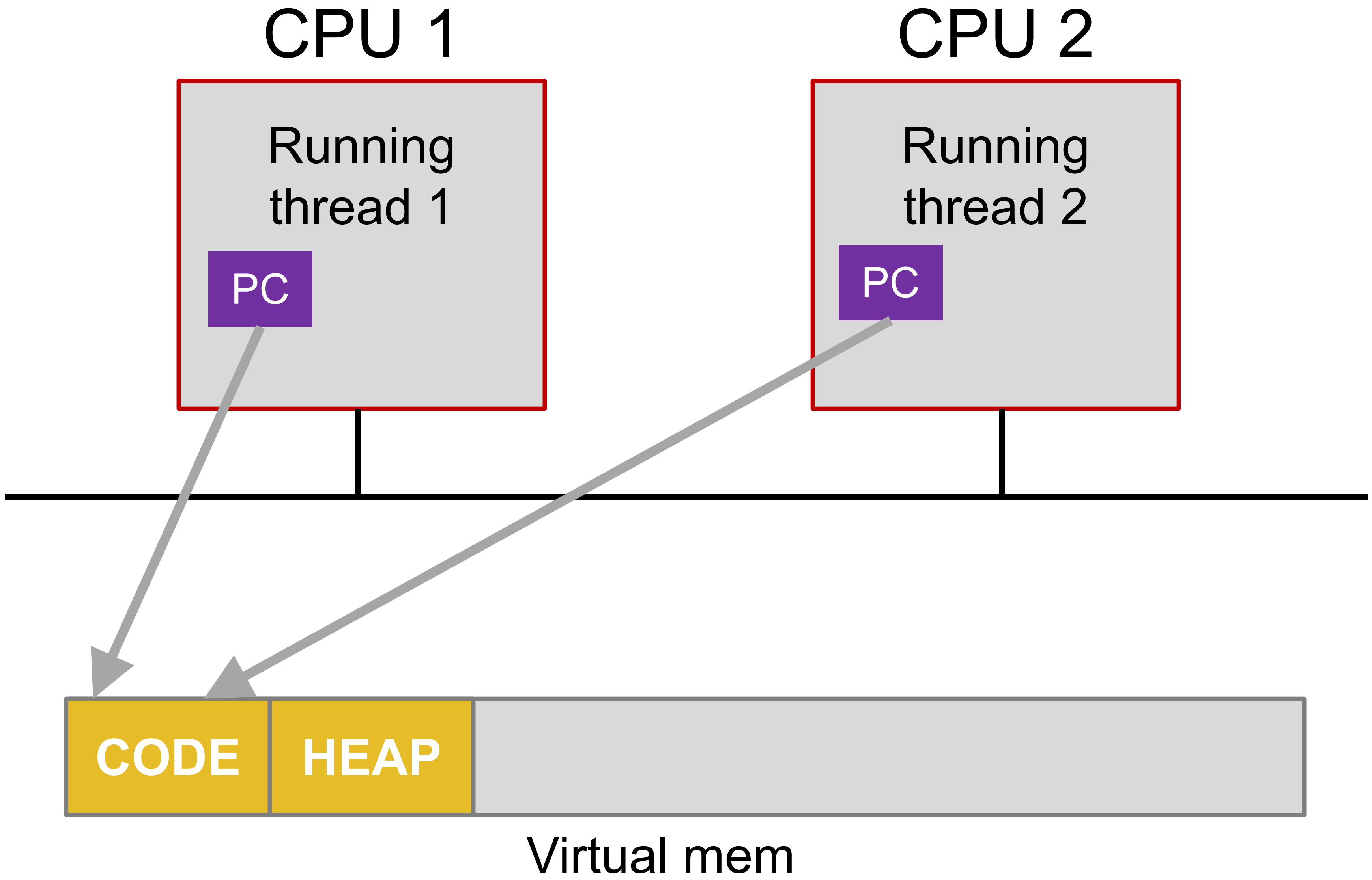
- Thread
  - Efficient and fast resource sharing
  - Efficient utilization of many CPU cores with only one process
  - Less context switching overheads

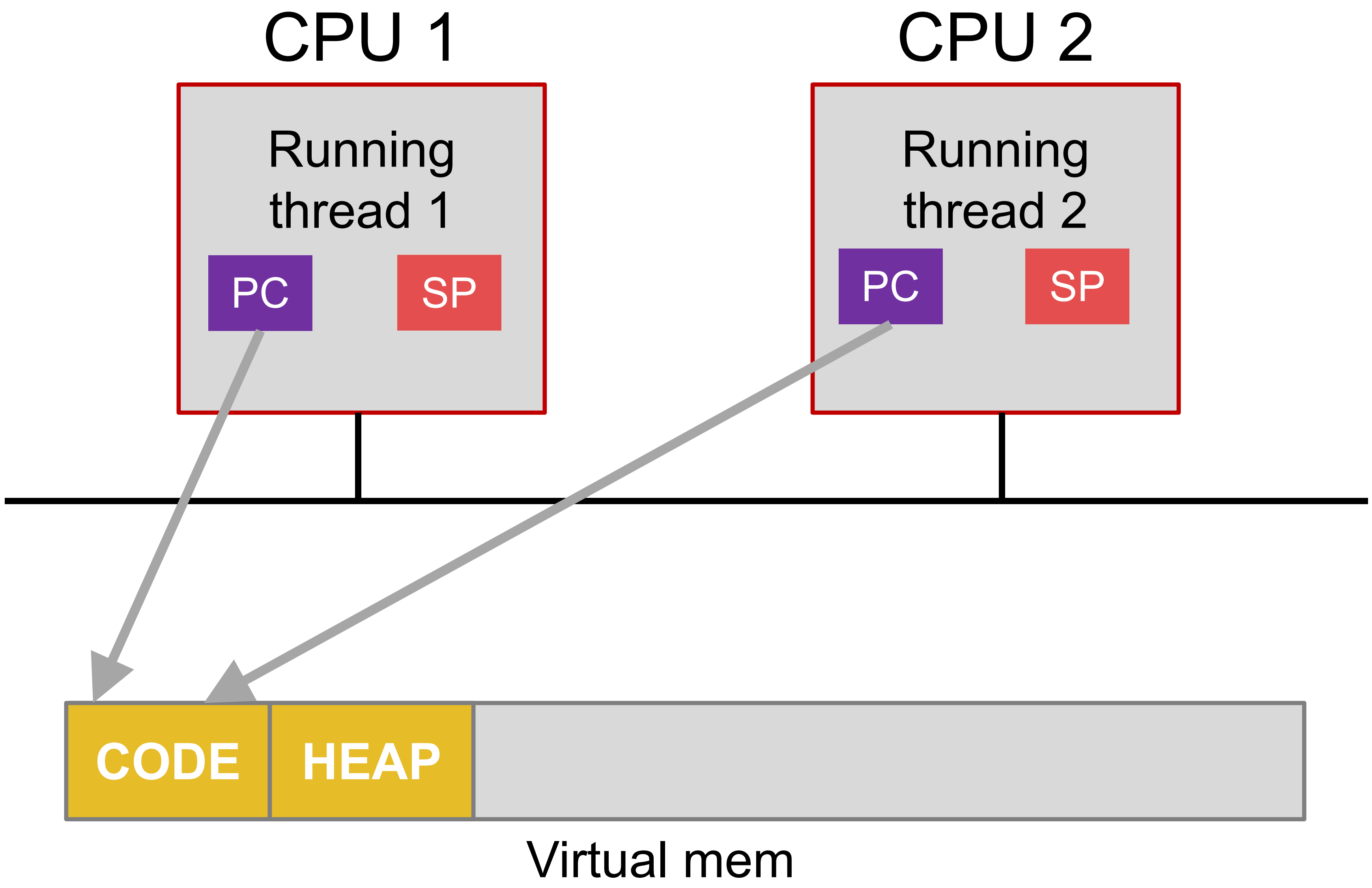

CPU 1
Running thread 1
CPU 2
Running thread 2

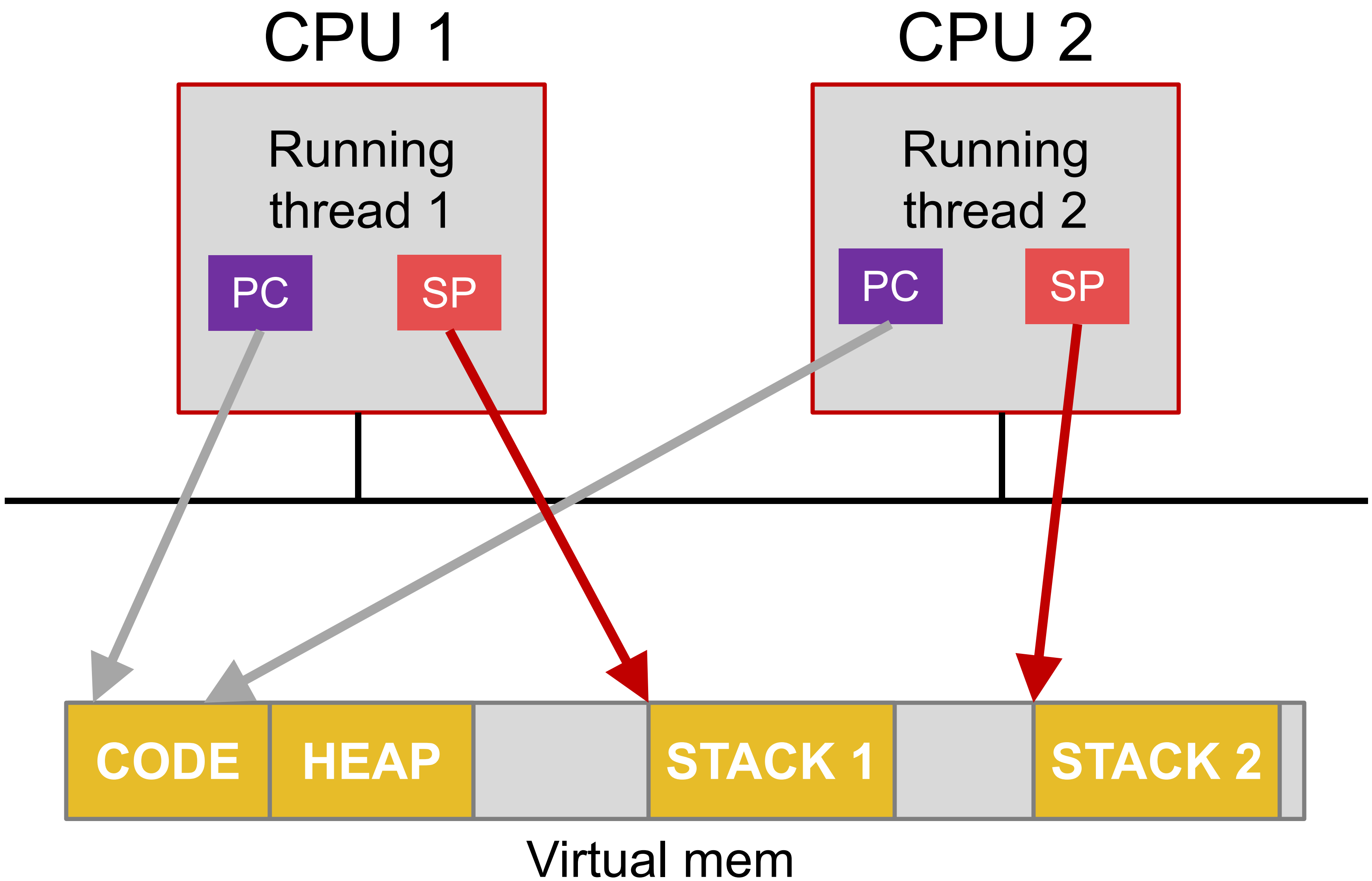

CPU 1
Running thread 1
PC
CPU 2
Running thread 2
PC

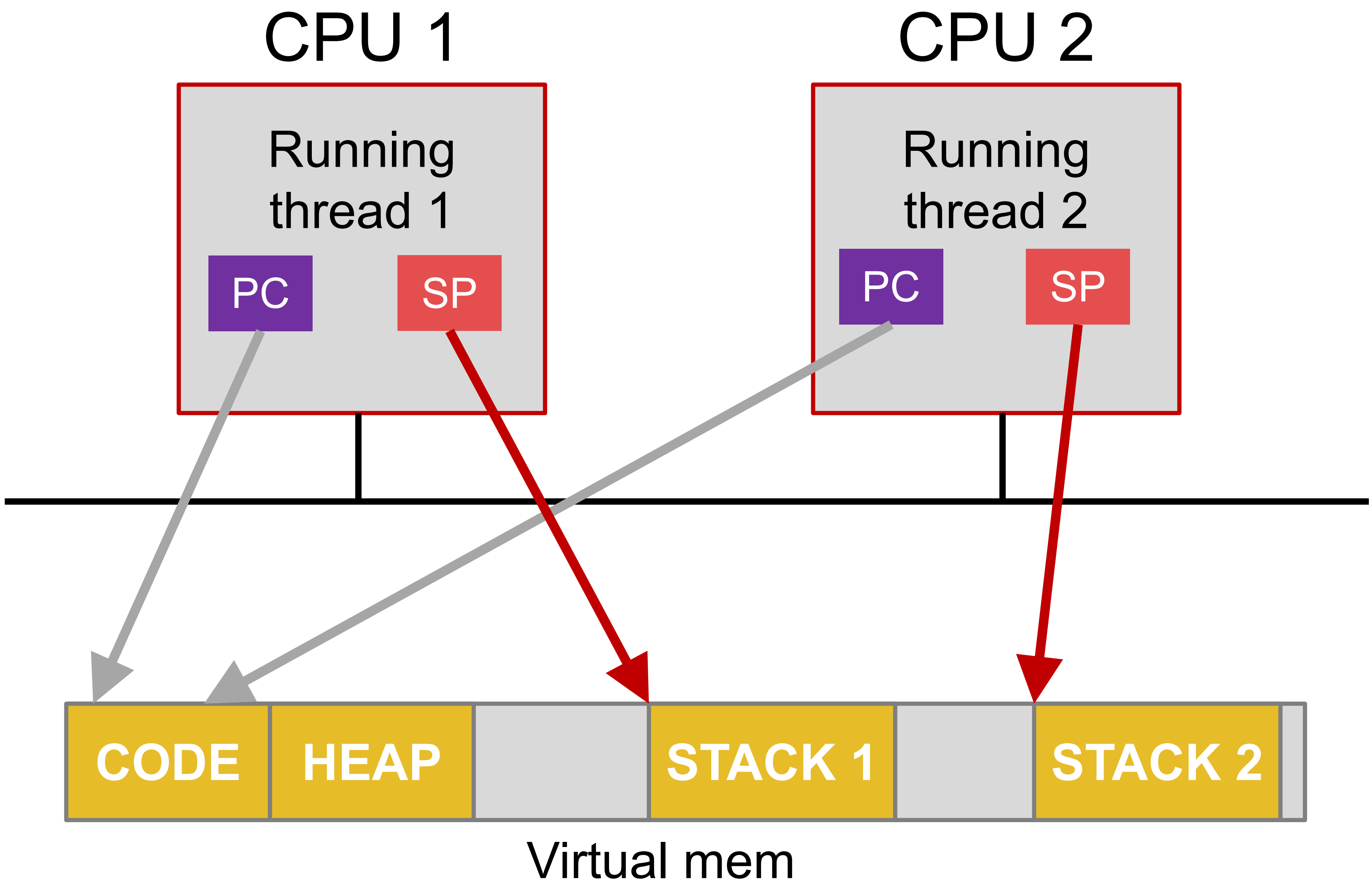

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
CODE
HEAP
Virtual mem


CPU 1
Running thread 1
PC
CPU 2
Running thread 2
PC
Each thread may be executing different code at the same time
CODE
HEAP
Virtual mem

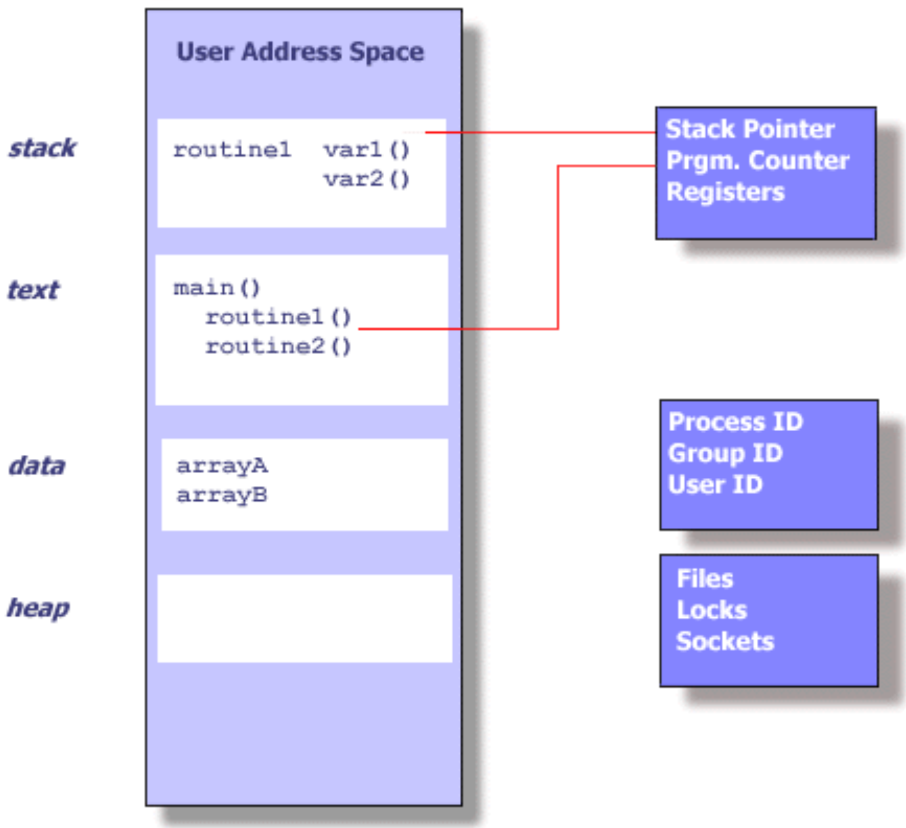

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
PC
CODE
HEAP
Virtual mem

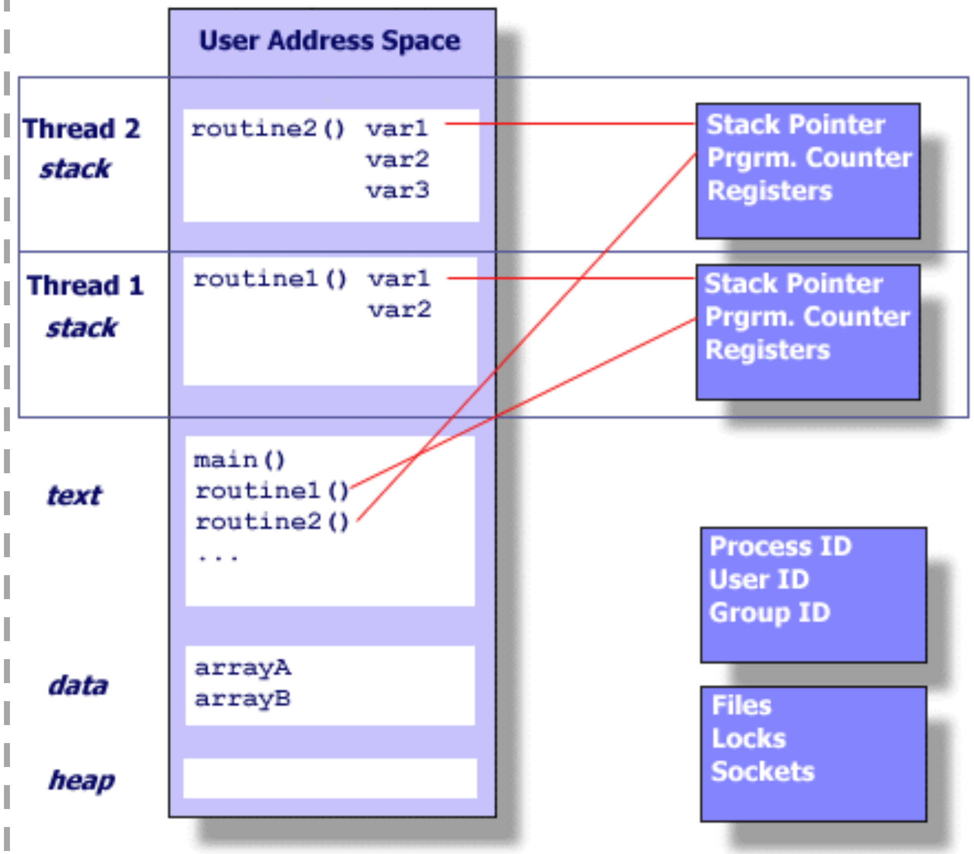

CPU 1
CPU 2
Running thread 1
Running thread 2
PC
SP
PC
SP
CODE
HEAP
Virtual mem


CPU 1
Running thread 1
PC
SP
CPU 2
Running thread 2
PC
SP
CODE
HEAP
STACK 1
STACK 2
Virtual mem



Thread executing different functions need different stacks



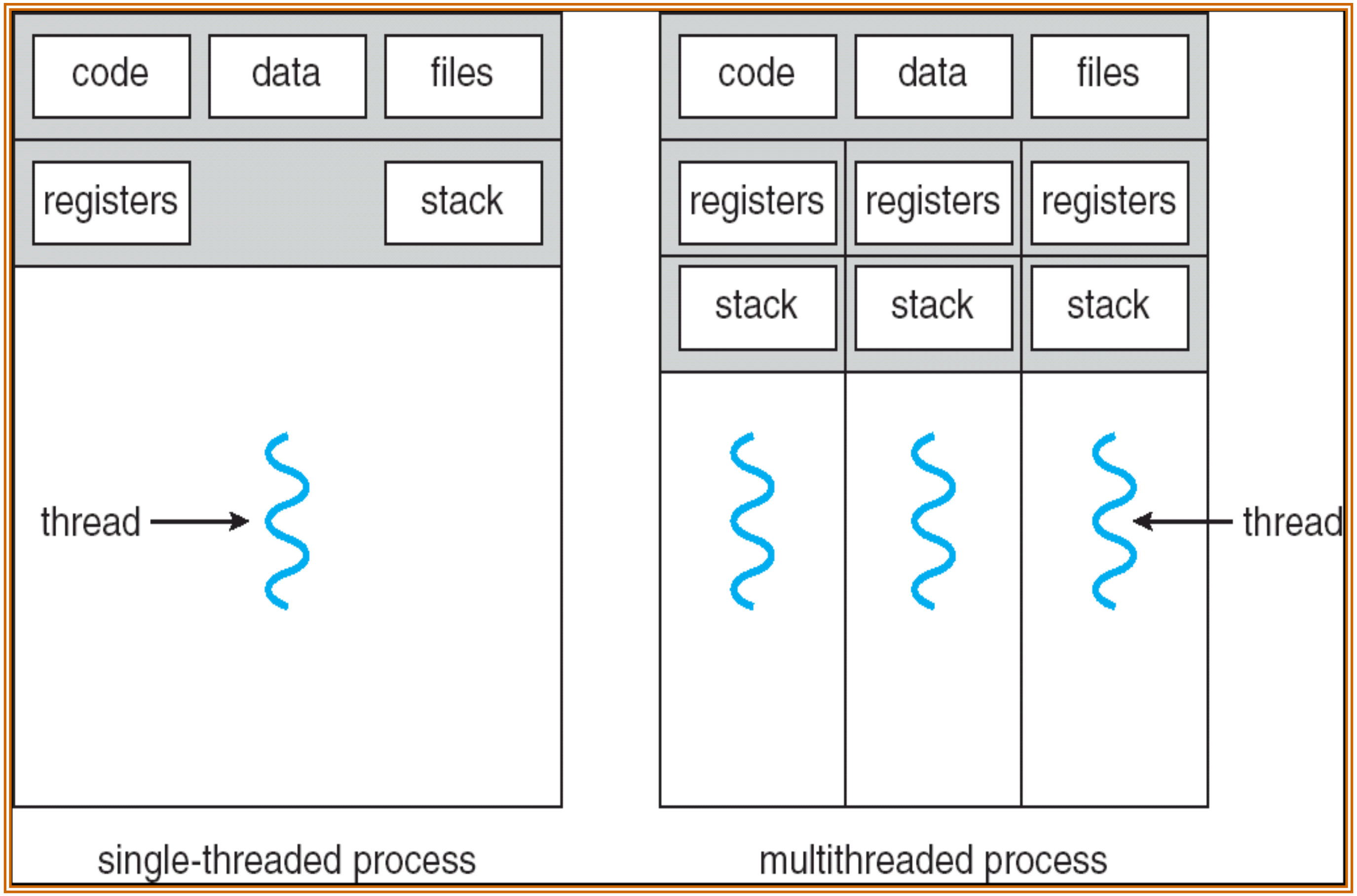


Linux process

Threads within a Linux process

*: https://computing.llnl.gov/tutorials/pthreads/

Single- vs. Multi-threaded Process



Using Threads

- Processes usually start with a single thread
- Usually, library procedures are invoked to manage threads
  - `thread_create`: typically specifies the name of the procedure for the new thread to run
  - `thread_exit`
  - `thread_join:` blocks the calling thread until another (specific) thread has exited
  - `thread_yield:` voluntarily gives up the CPU to let another thread run

Pthread

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX (e.g., Linux) OSes

Pthread APIs

Thread Call	Description
`pthread_create`	Create a new thread in the caller's address space
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a thread to terminate
`pthread_mutex_init`	Create a new mutex
`pthread_mutex_destroy`	Destroy a mutex
`pthread_mutex_lock`	Lock a mutex
`pthread_mutex_unlock`	Unlock a mutex
`pthread_cond_init`	Create a condition variable
`pthread_cond_destroy`	Destroy a condition variable
`pthread_cond_wait`	Wait on a condition variable
`pthread_cond_signal`	Release one thread waiting on a condition variable

Pthread APIs

Thread Call	Description
`pthread_create`	Create a new thread in the caller's address space
`pthread_exit`	Terminate the calling thread
`pthread_join`	Wait for a thread to terminate
`pthread_mutex_init`	Create a new mutex
`pthread_mutex_destroy`	Destroy a mutex
`pthread_mutex_lock`	Lock a mutex
`pthread_mutex_unlock`	Unlock a mutex
`pthread_cond_init`	Create a condition variable
`pthread_cond_destroy`	Destroy a condition variable
`pthread_cond_wait`	Wait on a condition variable
`pthread_cond_signal`	Release one thread waiting on a condition variable

Thread creation

Thread lock

Thread CV

Example of Using Pthread

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

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

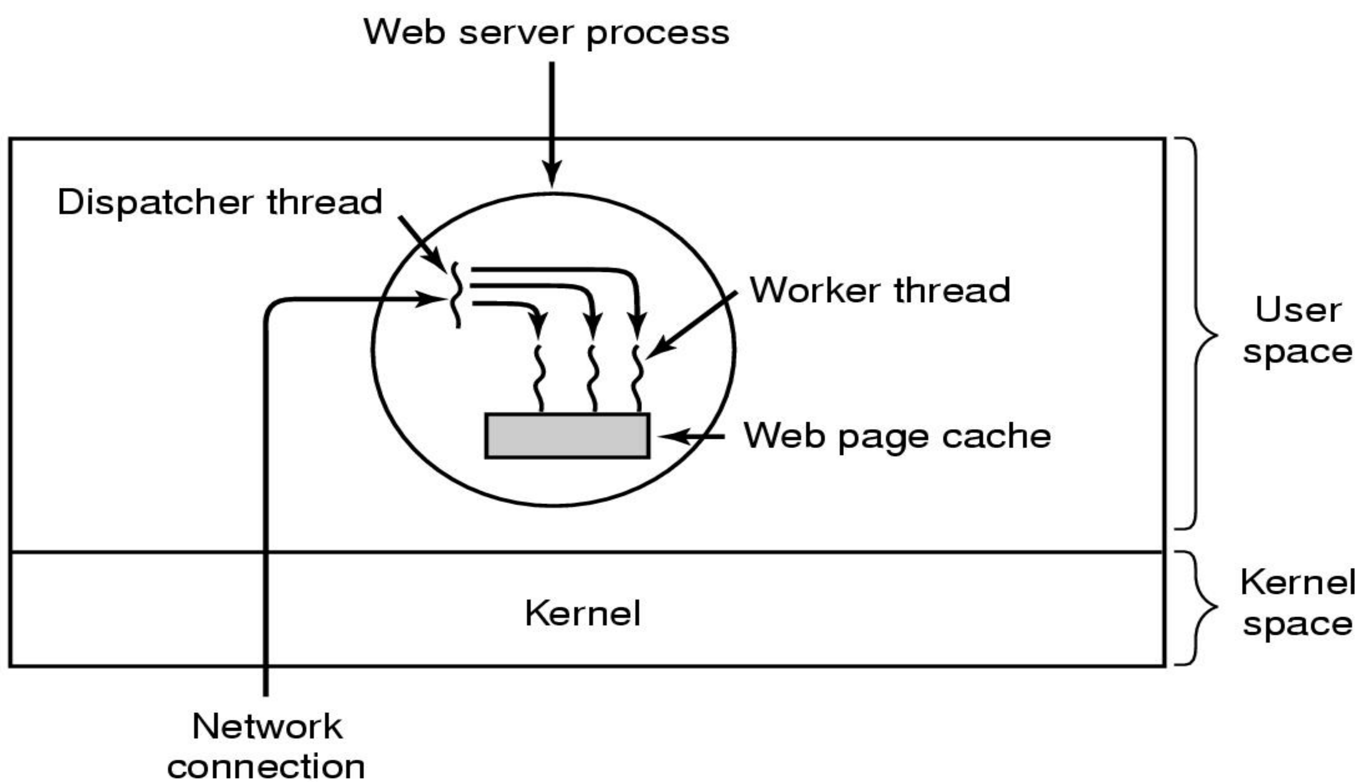
Demo: Basic Threads

Fork the demo code repo at: https://github.com/tddg/demo-ostep-code

In today's lecture, we showed the demo in dir: thread-api.

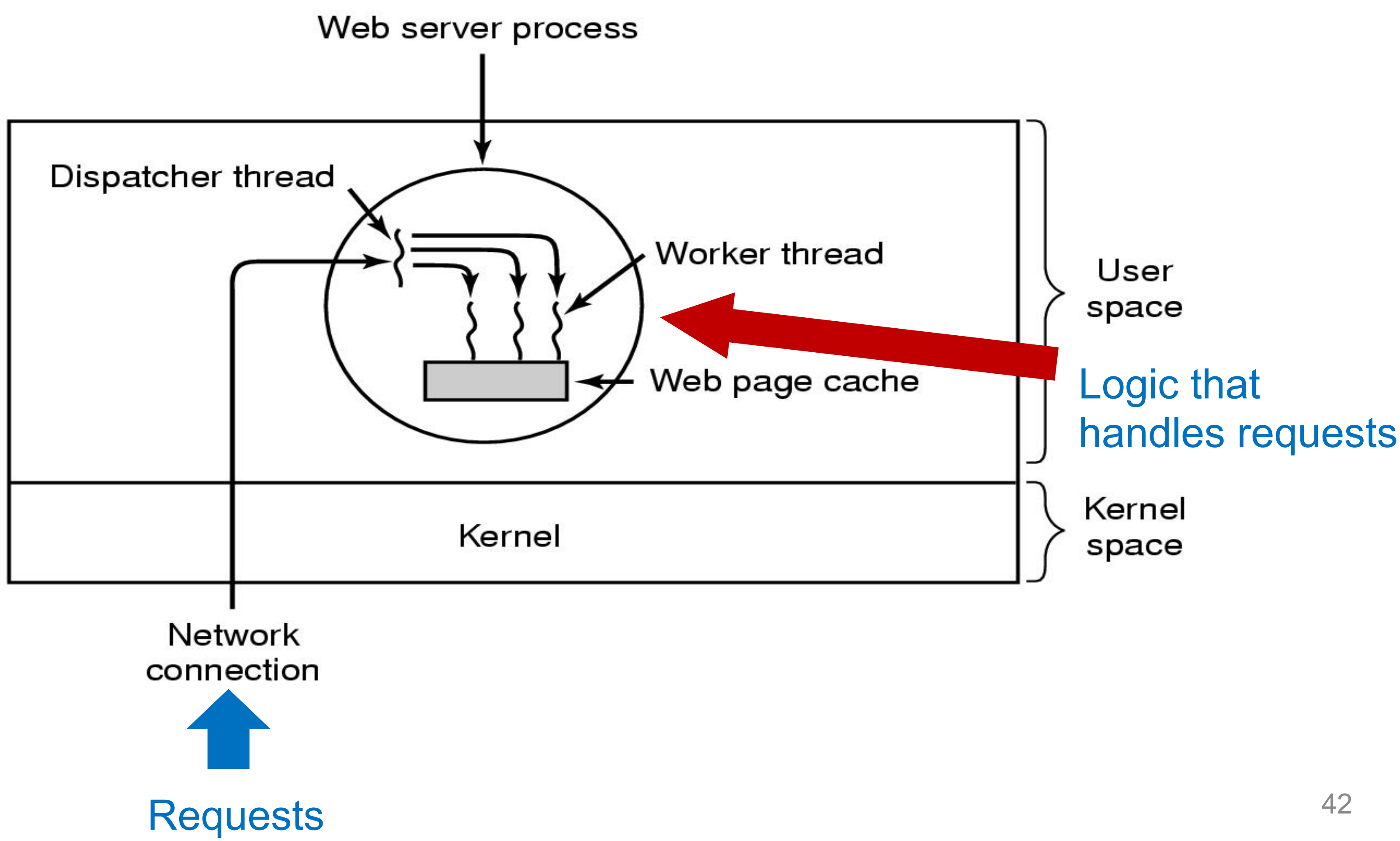
Example Multithreaded Applications

A multithreaded web server



Example Multithreaded Applications

A multithreaded web server



Code Sketch

```
while (TRUE) {
   get_next_request(&buf);
   handoff_work(&buf);
}
```

(a) Dispatcher thread

```
while (TRUE) {
   wait_for_work(&buf);
   check_cache(&buf; &page);
   if (not_in_cache)
      read_from_disk(&buf,  &page);
   return_page(&page);
}
```

(b) Worker thread

Benefits of Multithreading

- **Resource Sharing**
  – Sharing the address space and other resources may result in high degree of cooperation
- **Economy**
  – Creating/managing processes much more time consuming than managing threads: e.g., context switch
- **Better Utilization of Multicore Architectures**
  – Threads are doing job concurrently (in parallel)
  – Multithreading an interactive application may allow a program to continue running even if part of it is blocked or performing a lengthy operation

Real-world Example: Memcached

- Memcached—A high-performance memory-based caching system
  - 14k lines of C source code
  - https://memcached.org/
- A typical multithreaded server implementation
  - `Pthread` + `libevent`
  - A dispatcher thread dispatches newly coming connections to the worker threads in a round-robin manner
  - Event-driven: Each worker thread is responsible for serving requests from the established connections

Multithreading vs. Multi-processes

- Real-world debate
  - Multithreading vs. Multi-processes
  - Memcached vs. Redis
- Redis—A single-threaded memory-based data store
  - https://redis.io/

Memcached

redis

Wish List for Redis…

http://goo.gl/N9UTKD

Wish List For Redis

- Explicit memory management.
- **Deployable (Lua) Scripts.** Talked about near the start.
- **Multi-threading.** Would make cluster management easier. Twitter has a lot of "tall boxes," where a host has 100+ GB of memory and a lot of CPUs. To use the full capabilities of a server a lot of Redis instances need to be started on a physical machine. With multi-threading fewer instances would need to be started which is much easier to manage.