Systems Programming

Threads

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.

Today

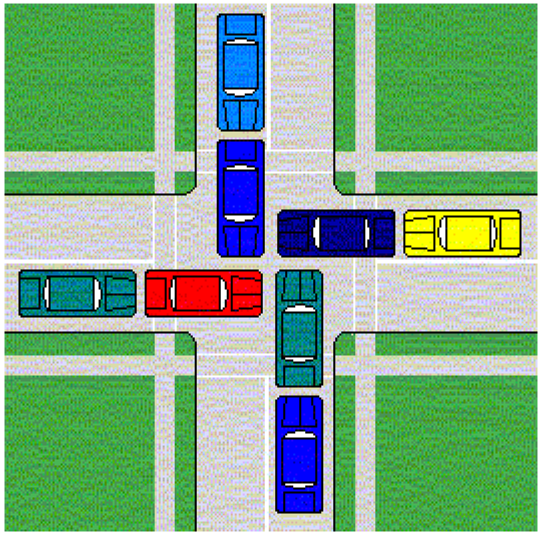
- **Concurrent Programming is Hard**
- Threads

Concurrent Programming is Hard!

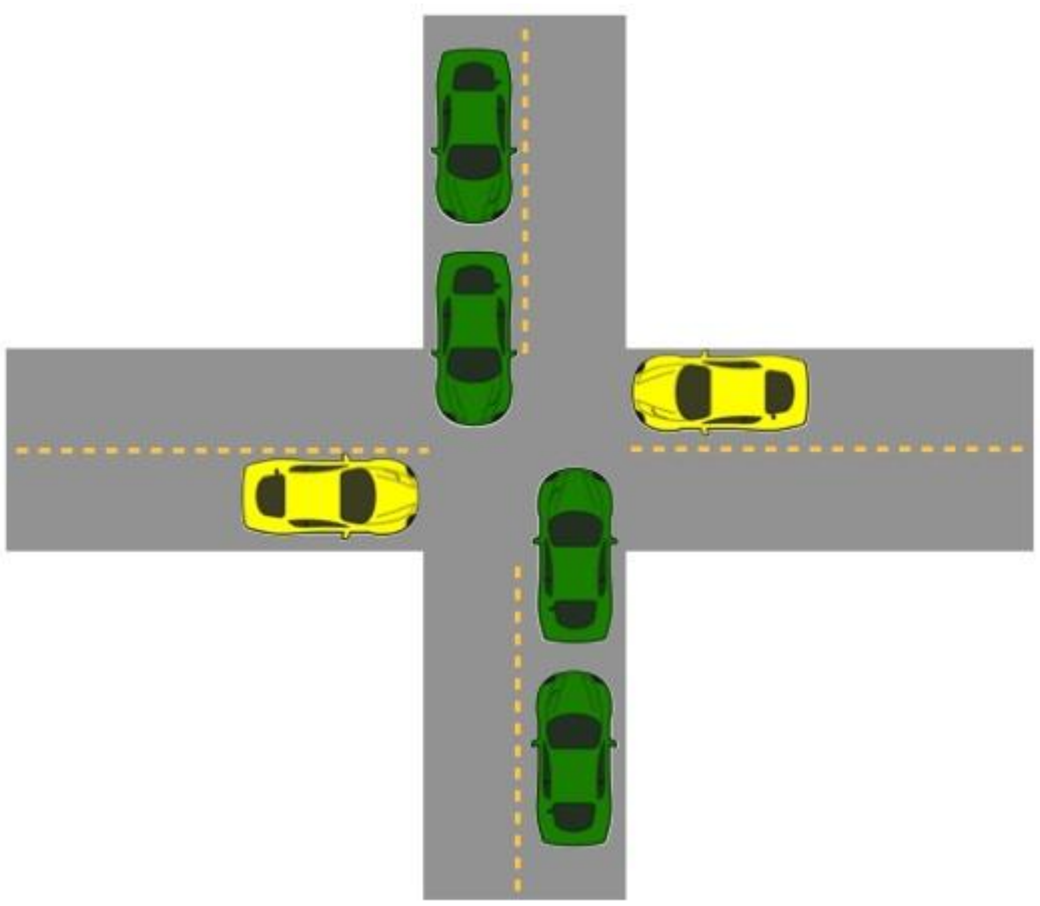
- **The human mind tends to be sequential**
- **The notion of time is often misleading**
- **Thinking about all possible sequences of events in a computer system is at least error prone and often impossible**
- **Often leads to concurrency bugs**
  - Data race, deadlock, livelock, and starvation

Data Race

Deadlock



Starvation



- Yellow must yield to green
- Continuous stream of green cars
- Overall system makes progress, but some individuals wait indefinitely

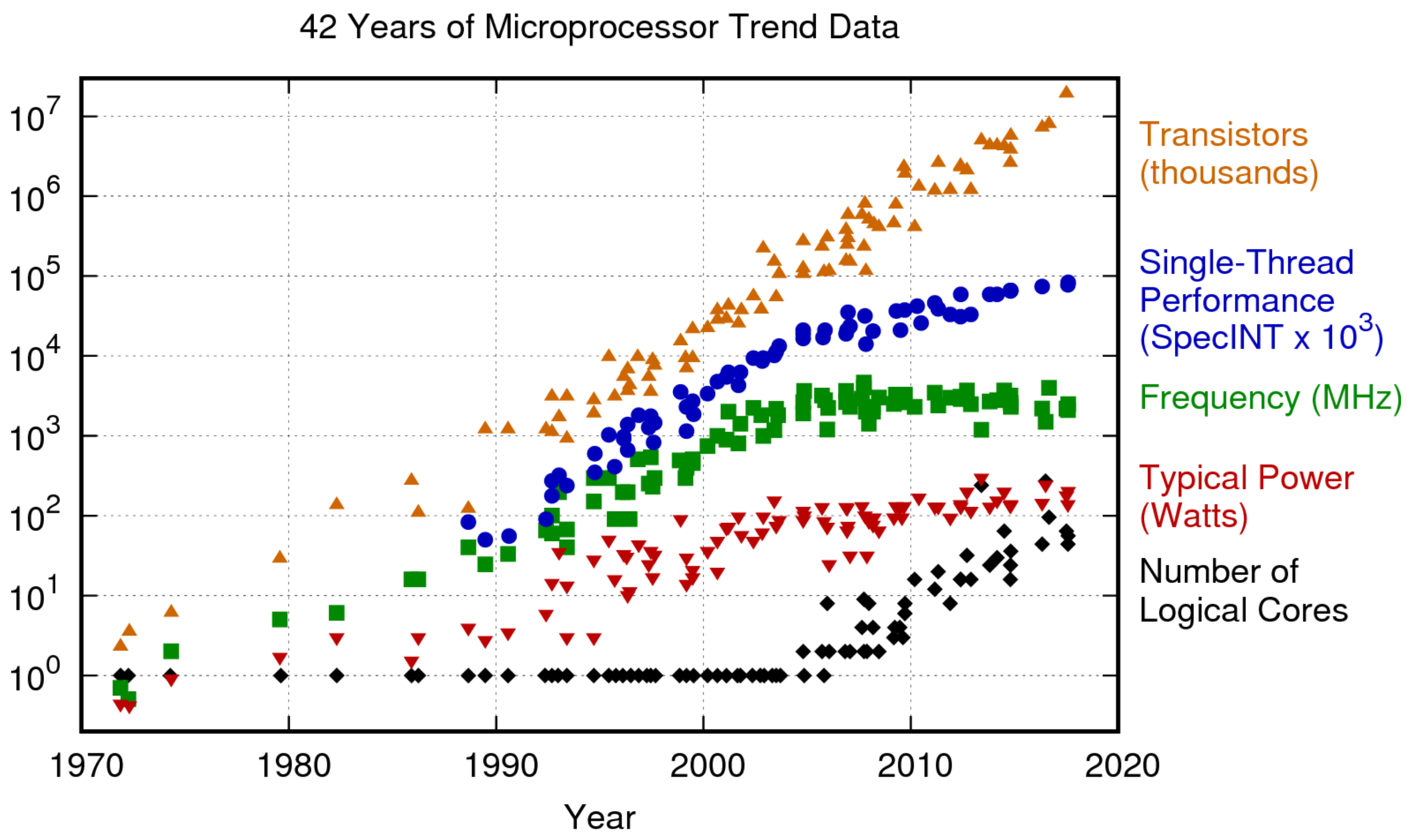
Concurrent Programming is Hard!

- **Classical problem classes of concurrent programs:**
  - ***Races:*** outcome depends on arbitrary scheduling decisions
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Starvation***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line
- **Many aspects of concurrent programming are beyond the scope of our course..**
  - We'll cover some of these aspects in the next few lectures.

Concurrent Programming is Hard!

It may be hard, but …

it is useful and become more and more necessary!



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

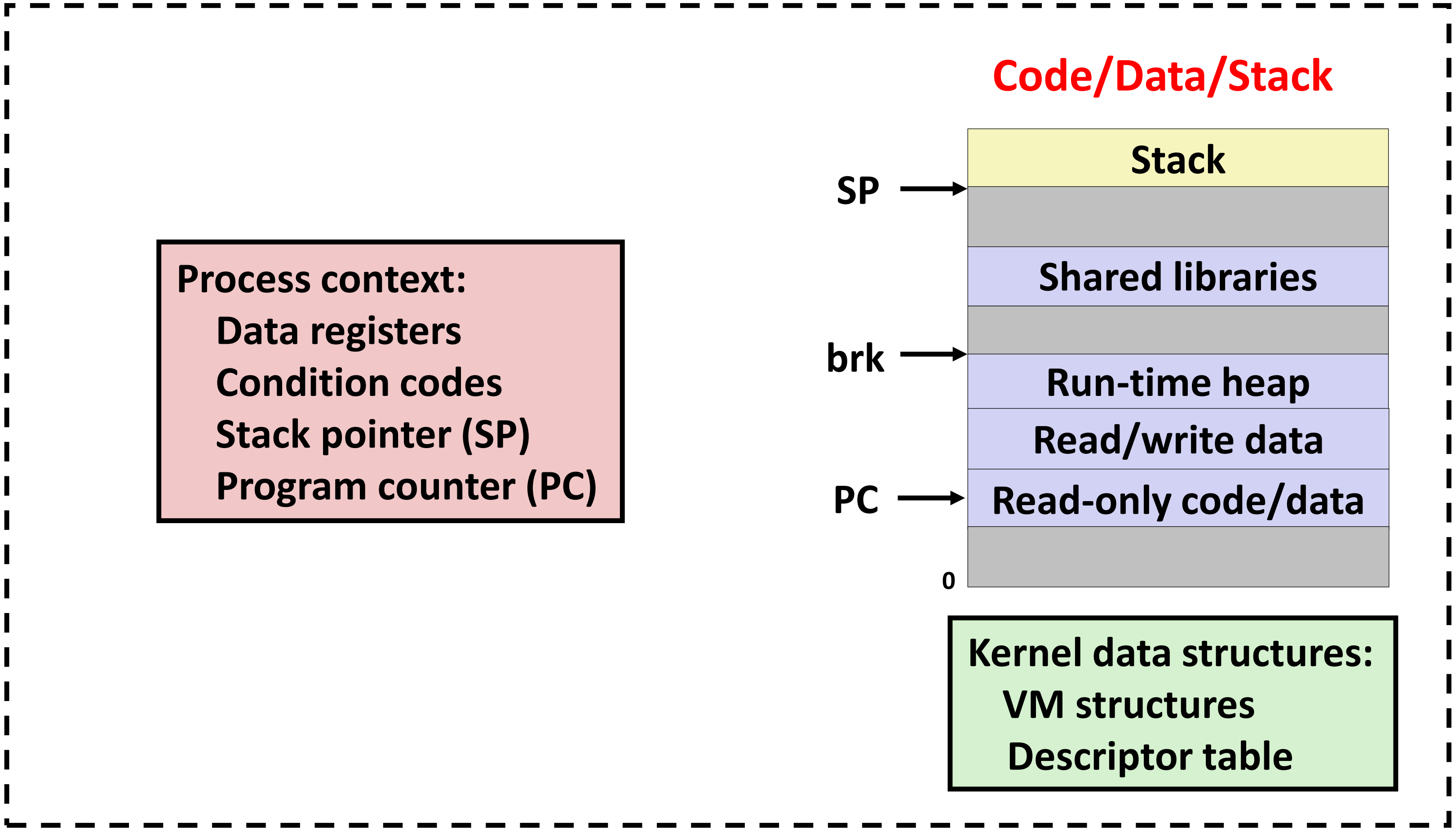
Today

- Concurrent Programming is Hard
- **Threads**

Process is defined by …

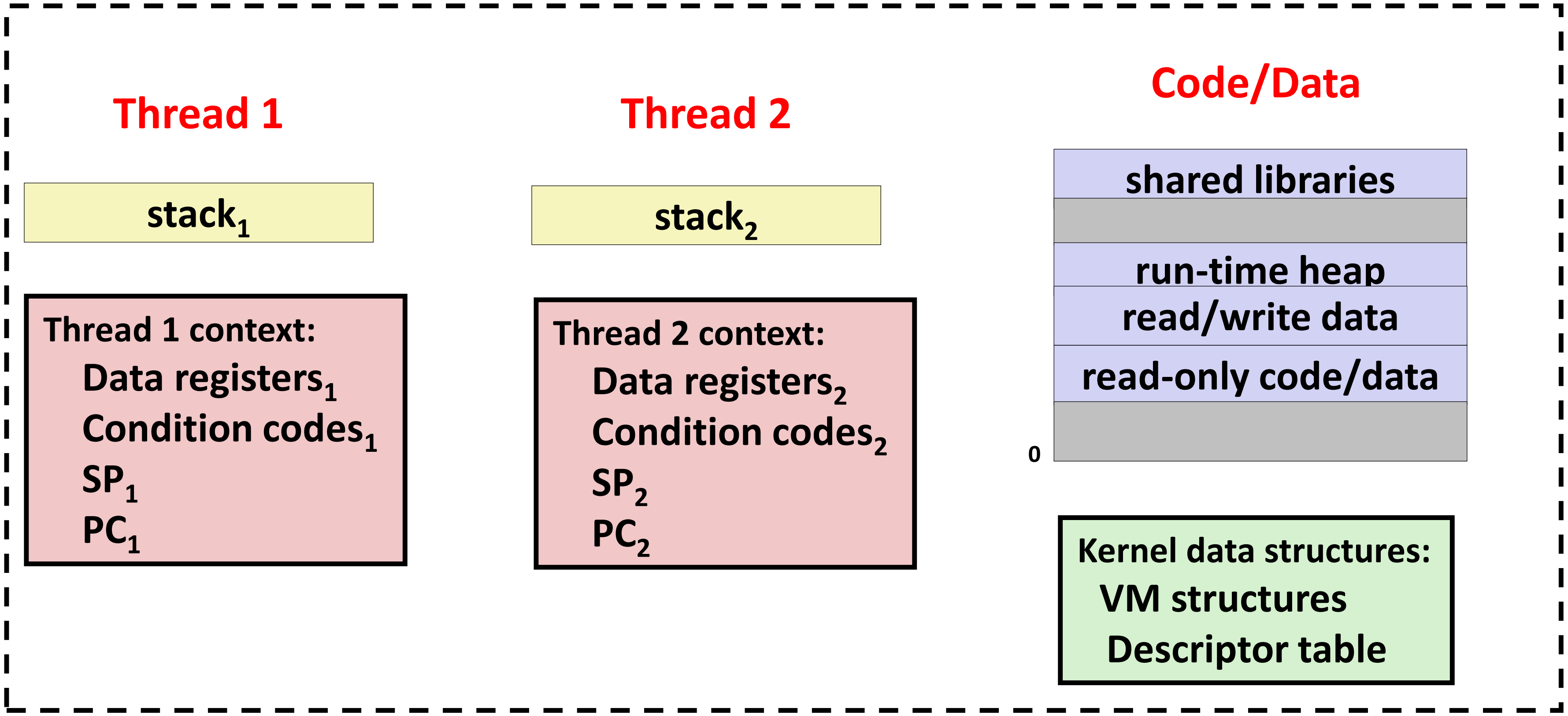
- Process = process context + code/data/stack

  + kernel data structures



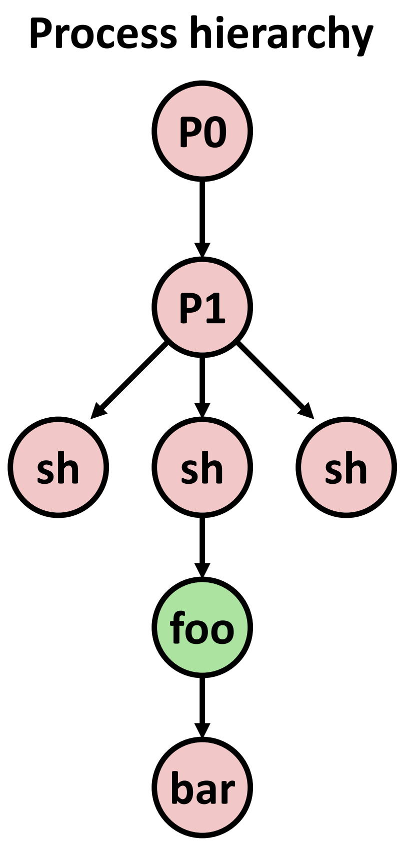
A Process With Multiple Threads

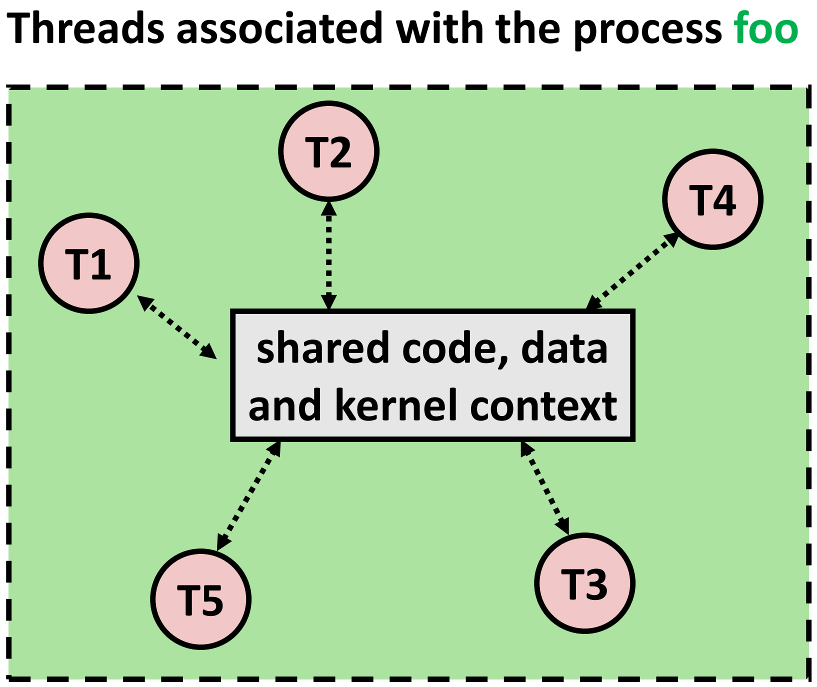
- **Multiple threads can be associated with a process**
  - Each thread has its own
    - logical control flow
    - stack (but not protected from other threads)
    - thread id (TID)
  - Threads share the same
    - code, data, and kernel context



Logical View of Threads

- **Threads associated with process form a pool of peers**
  - Unlike processes which form a tree hierarchy

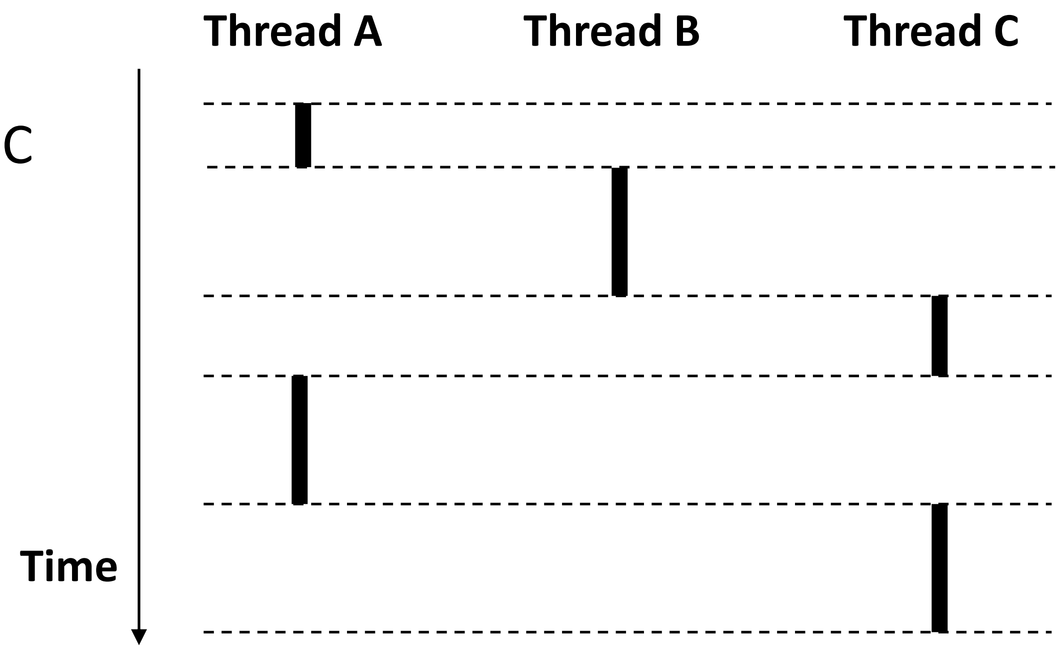



Threads associated with the process foo
T1
T2
T4
shared code, data and kernel context
T5
T3

Concurrent Threads

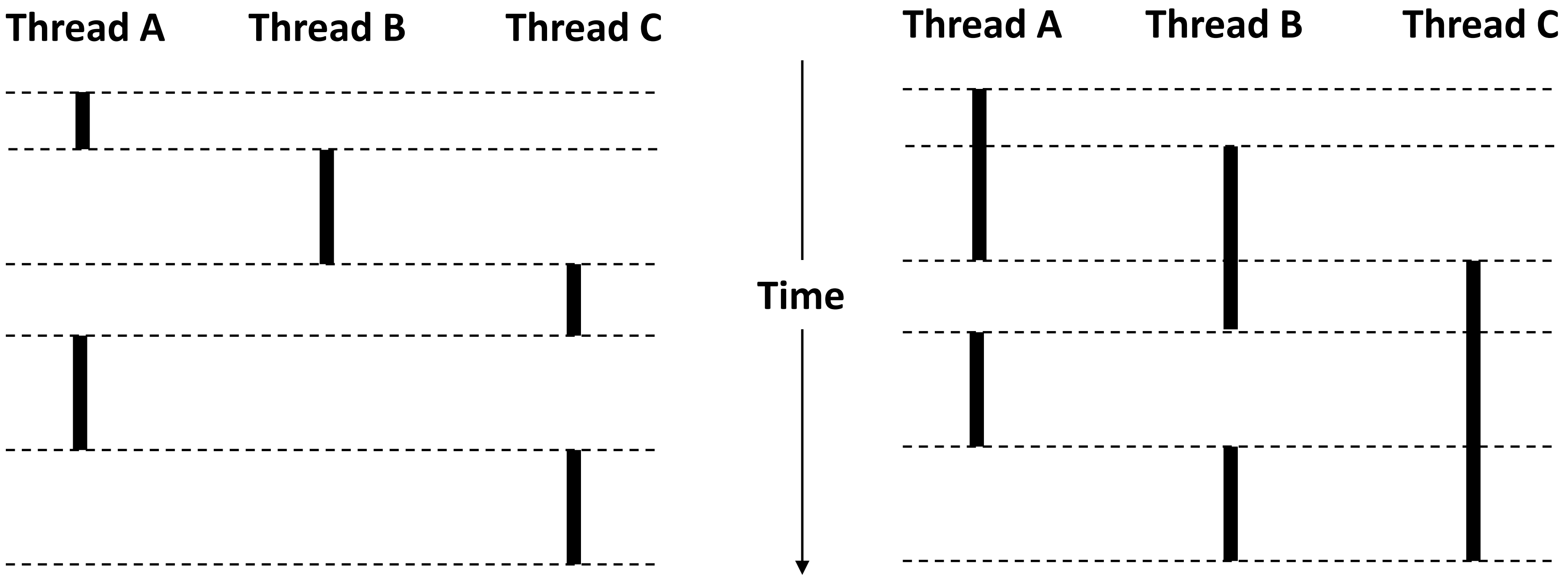
- **Two threads are *concurrent* if their flows overlap in time**
- **Otherwise, they are sequential**

- **Examples:**
  - Concurrent: A & B, A&C
  - Sequential: B & C



Concurrent Thread Execution

- **Single Core Processor**
  - Simulate parallelism by time slicing
- **Multi-Core Processor**
  - Can have true parallelism



Run 3 threads on a single CPU core

Run 3 threads on two CPU cores

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Posix Threads (pthreads) Interface

- ***pthreads:* Standard interface for ~60 functions that manipulate threads from C programs**
  - Creating and reaping threads
    - **`pthread_create()`**
    - **`pthread_join()`**
  - Determining your thread ID
    - **`pthread_self()`**
  - Terminating threads
    - **`pthread_cancel()`**
    - **`pthread_exit()`**
    - **`exit()`** [terminates all threads]
    - **`return`** [terminates current thread]
  - Synchronizing access to shared variables
    - **`pthread_mutex_init`**
    - **`pthread_mutex_[un]lock`**

The pthreads "hello, world" Program

```c
/*
 * hello.c - pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main(int argc, char** argv)
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    return 0;
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

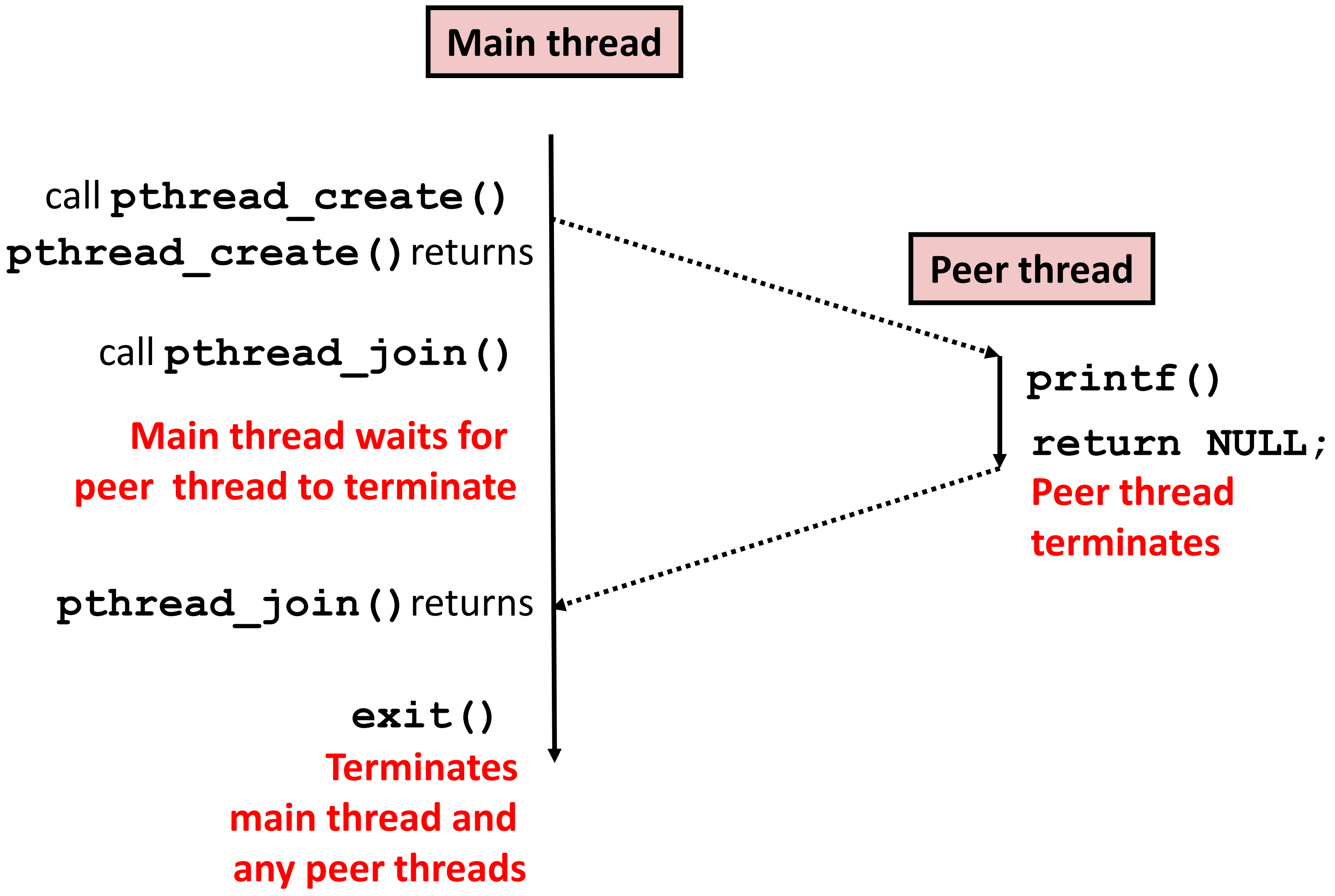
*Thread arguments (void *p)*

*Return value (void **p)*

```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

Execution of Threaded "hello, world"



Main thread
call pthread_create()
pthread_create() returns
call pthread_join()
Main thread waits for peer thread to terminate
pthread_join() returns
exit()
Terminates main thread and any peer threads
Peer thread
printf()
return NULL;
Peer thread terminates