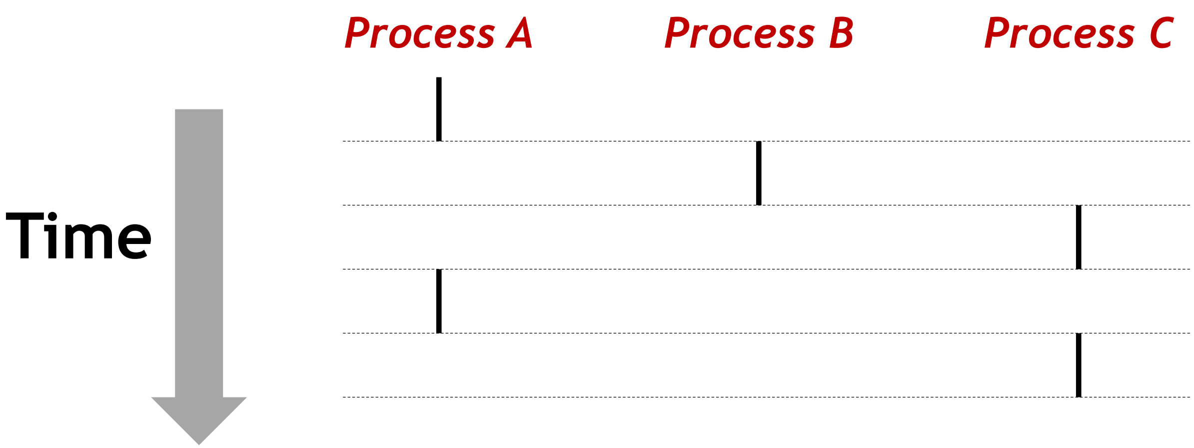
# Concurrent Programming

# + Concurrency (Review)

- **Multiple logical control flows.**
- **Flows run concurrently if they overlap in time**
  - Otherwise, they are sequential
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



# + What & Why is Concurrency?

- **What: things happening “simultaneously"**
  - e.g. single CPU interleaving instructions from two flows
  - e.g. multiple CPU cores concurrently executing instructions
  - e.g. CPU and network card concurrently doing processing
- **Why: efficiency**
  - Due to ‘power wall’ cores not getting faster, just more numerous
    - To speed up programs using multiple CPUs we have to write concurrent code.
  - From systems perspective, don’t idle CPU while IO is performed
    - To speed up programs the system interleaves CPU processing and I/O.

# + Concurrent Programming is Hard!

- **The human mind tends to be sequential**
- **Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible**
  - Imagine two control flows of 2 instructions each A, B and C, D
    - Possible interleaved execution orders
      - A,B,C,D
      - A,C,B,D
      - A,C,D,B
      - C,D,A,B
      - C,A,D,B
      - C,A,B,D
    - Some orderings might yield unexpected results.

# + Approaches for Writing Concurrent Programs

- **Process-based**
  - Kernel automatically interleaves multiple logical flows
  - Each flow has its *own private address space*
- **Thread-based**
  - Kernel automatically interleaves multiple logical flows
  - Each flow *shares the same address space*
  - Threads are less expensive for the system!

# Process-based Concurrency

# + Process-Based Concurrent Program

- **What does this program do?**
- **What would be printed from line 18?**

```c
1  int numbers[1000];
2  int sum1 = 0, sum2 = 0;
3
4  int main() {
5      for (int i = 0; i < 1000; i++)
6        numbers[i] = 1;
7
8      int pid = fork();
9      if (pid != 0) {
10         for (int i = 0; i < 500; i++)
11             sum1 += numbers[i];
12     } else {
13         for (int i = 0; i < 500; i++)
14             sum2 += numbers[500+i];
15         return 0;
16     }
17     waitpid(pid, NULL, 0);
18     printf("sum is %d\n", sum1 + sum2);
19
20     return 0;
21 }
```

# + Process-Based Concurrent Program

- **What does this program do?**
- **What would be printed from line 18?**
- **Two processes concurrently sum the numbers.**
- **However, it is not simple to share data between them because they have *separate address spaces*.**

```
1  int numbers[1000];
2  int sum1 = 0, sum2 = 0;
3
4  int main() {
5      for (int i = 0; i < 1000; i++)
6        numbers[i] = 1;
7
8      int pid = fork();
9      if (pid != 0) {
10         for (int i = 0; i < 500; i++)
11             sum1 += numbers[i];
12     } else {
13         for (int i = 0; i < 500; i++)
14             sum2 += numbers[500+i];
15         return 0;
16     }
17     waitpid(pid, NULL, 0);
18     printf("sum is %d\n", sum1 + sum2);
19
20     return 0;
21 }
```

# + Interprocess Communication

- **How to communicate across processes? (*inter-process communication or IPC*)**
  - via *sockets*
  - via *pipes* (file system)
  - via *shared memory objects*

# + Unix Pipes

- **Unlike other forms of interprocess communication, a pipe is one-way communication only**
- **Via a pipe, output of one process is the input to another process.**
- **A limitation of pipes is that the processes using pipes must have a common parent process**

# + Pipes in C

- The `pipe` system call is called with a pointer to an array of two integers.
- The first element of the array contains the file descriptor that corresponds to the output of the pipe
- The second element of the array contains the file descriptor that corresponds to the input of the pipe.

```c
int main()
{
    int fd[2];
    pipe(fd);

    int pid = fork();
    if (pid != 0) { // parent
        write(fd[1], "This is a message!", 18);
    }
    else // child
    {
        int n;
        char buf[1025];
        if ((n = read(fd[0], buf, 1024)) >= 0)
        {
            buf[n] = 0; // null terminate string
            printf("Child -> %s \n", buf);
        }
        return 0;
    }

    waitpid(pid, NULL, 0);
    return 0;
}
```

# Pipes from the Shell

- **Using the terminal you can  two commands together so that the output from one program becomes the input of the next program.**
- **When you pipe commands together in the terminal in this way, it is called a *pipeline***
- **Example…**

```
ls | grep ".c" | sort -r | cut -c 1-5
```

# + Pros & Cons of Pipes

- **Pros**
  - Efficient use of memory and CPU time
  - Easy to create
- **Cons**
  - Can be confusing quickly in non-trivial programs
  - Uni-directional
  - Little capability for error handling
  - Requires system call

# + Shared Memory

- **Shared Memory is an efficient means of passing data between programs.**
- **Allow two or more *processes* access to the same address space for reading and writing.**
- **A process creates or accesses a shared memory segment using `shmget`()**
- **You can review an example of two processes sharing some memory in the following files**
  - lecture25/shared_memory_server.c
  - lecture25/shared_memory_client.c

# + Pros & Cons of Shared Memory

- **Pros**
  - Highly performant, bidirectional communication
- **Cons**
  - Error prone, difficult to debug
  - Requires system call
  - All the same *synchronization* problems as threads (which we will understand soon!)

# + Pros & Cons of Process-based Concurrency

- **Pros**
  - Clean sharing model
    - descriptors (no)
    - file tables (yes)
    - global variables (no)
- **Cons**
  - Systems calls necessary
  - Nontrivial to share data between processes
    - Requires IPC (interprocess communication) mechanisms

# Thread-based Concurrency

# + What is a Thread?

- **A thread is…**
  - a unit of execution, associated with a process.
  - the smallest sequence of instructions that can be managed independently by the OS scheduler
- **Multiple threads can..**
  - exist within one process
  - be executing concurrently
  - *share resources* such as memory

# + Traditional View of a Process

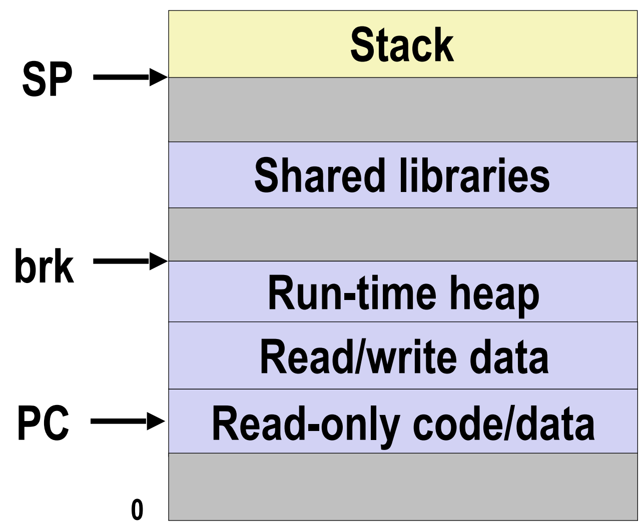
- **Process = process context + code, data & stack**

**Process context**

**Program context:**
- Registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

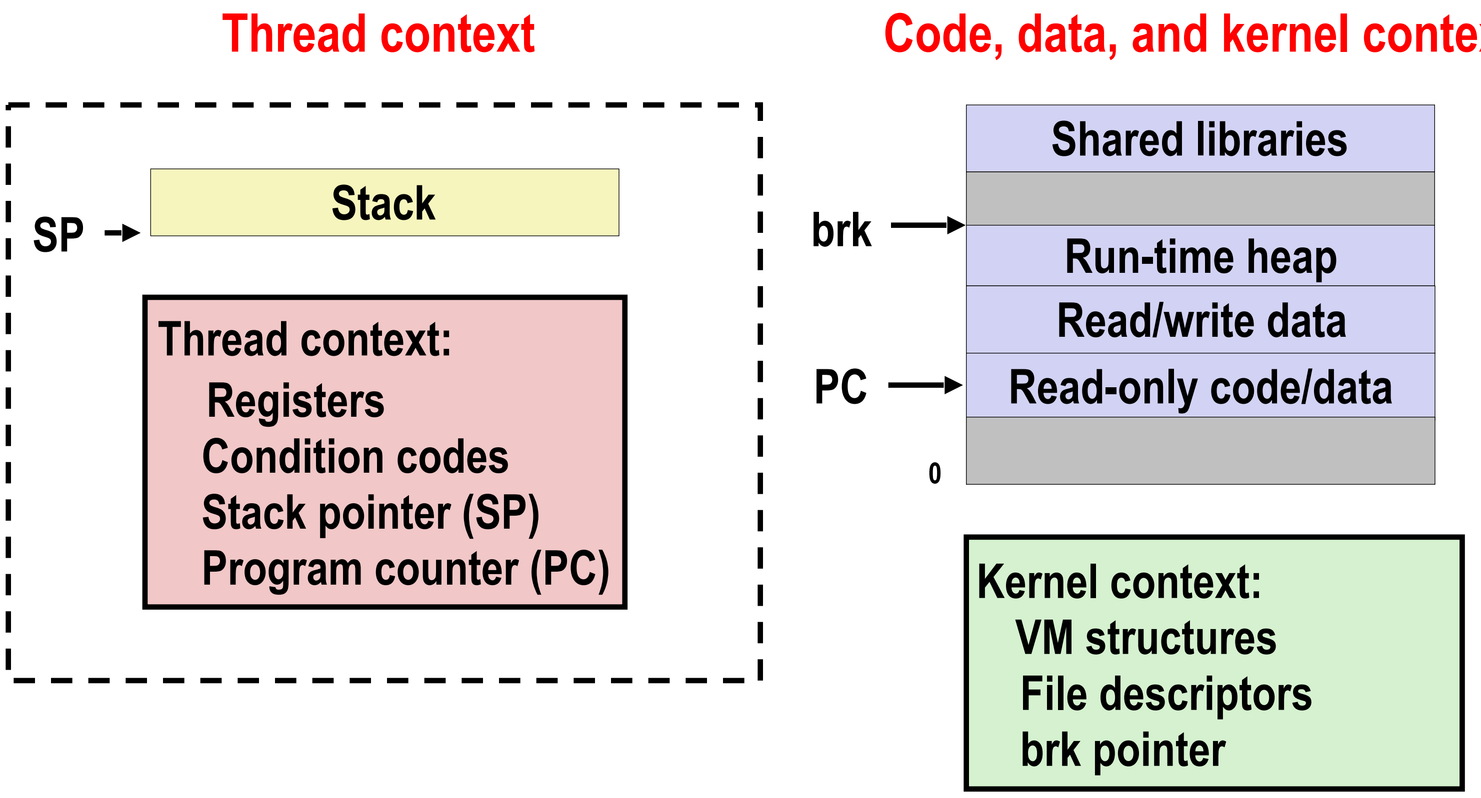
**Kernel context:**
- VM structures
- File descriptors
- brk pointer

**Code, data, and stack**



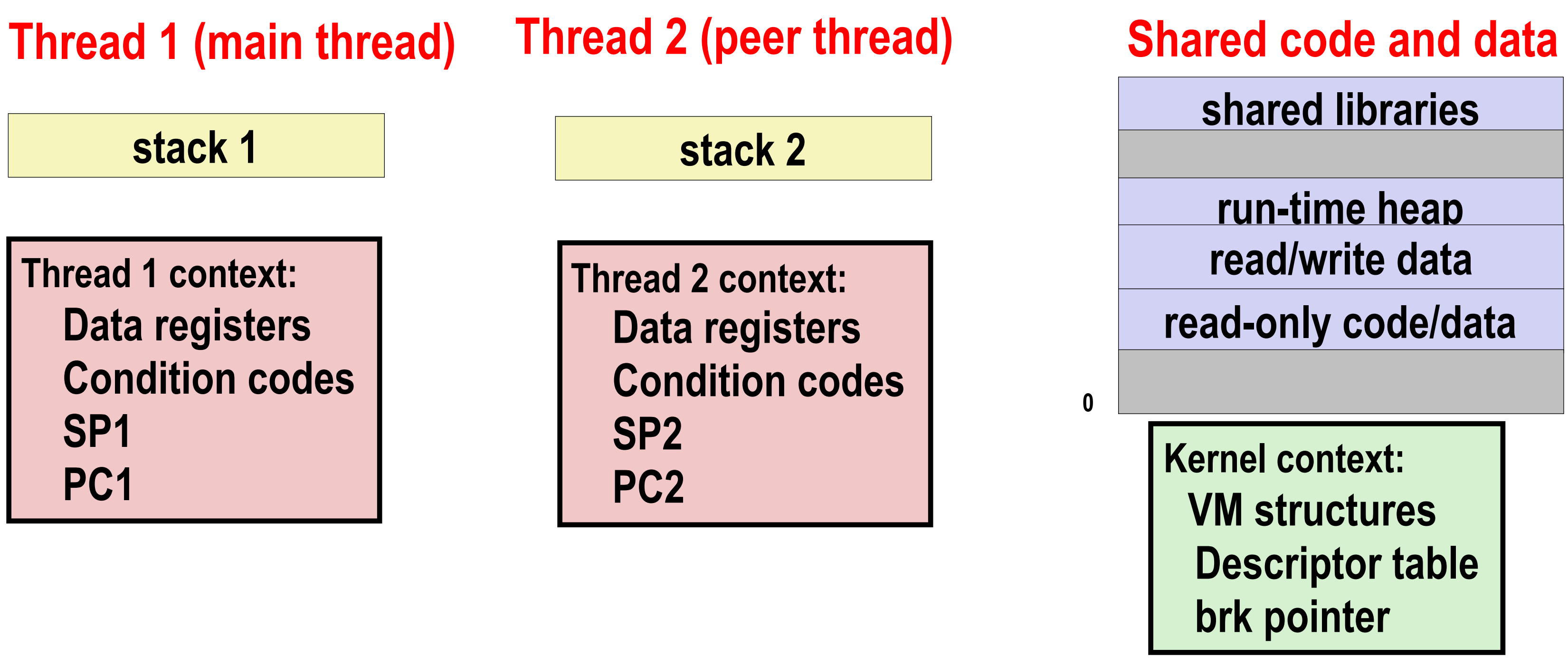
# Alternate View of a Process

- **Process = thread context(s) + code, data & kernel context**



# A Process With Multiple Threads

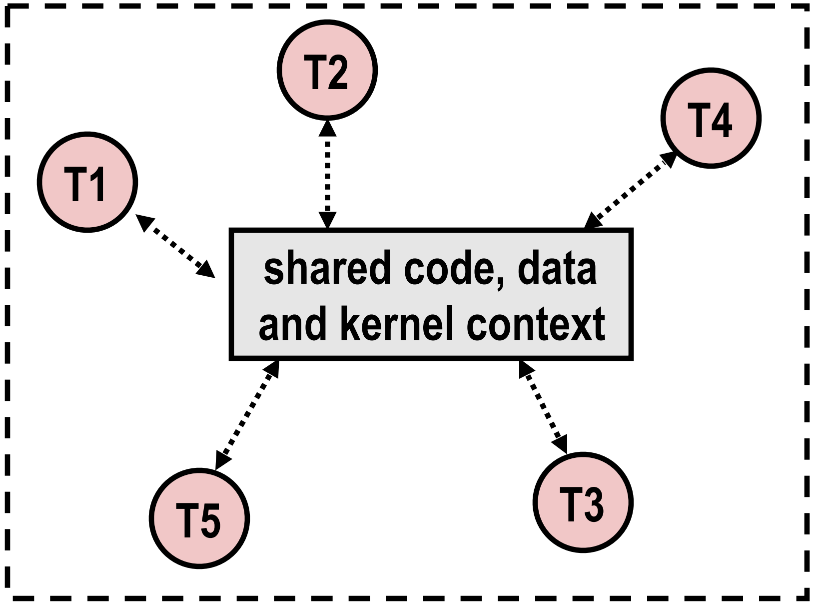
- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
  - Each thread has its own thread id (TID)



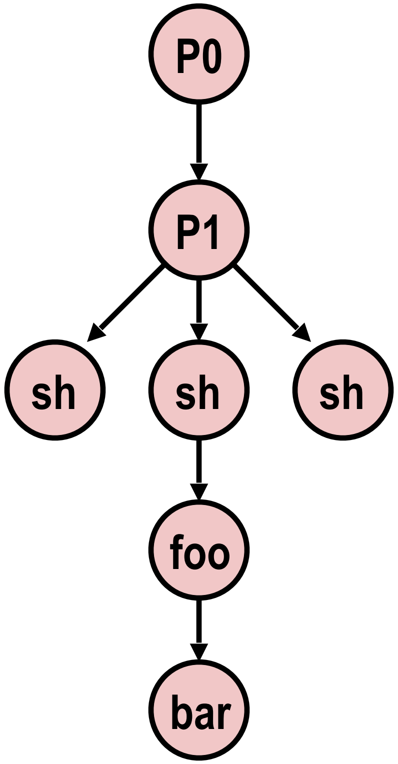
# Logical View of Threads

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

Threads associated with some process

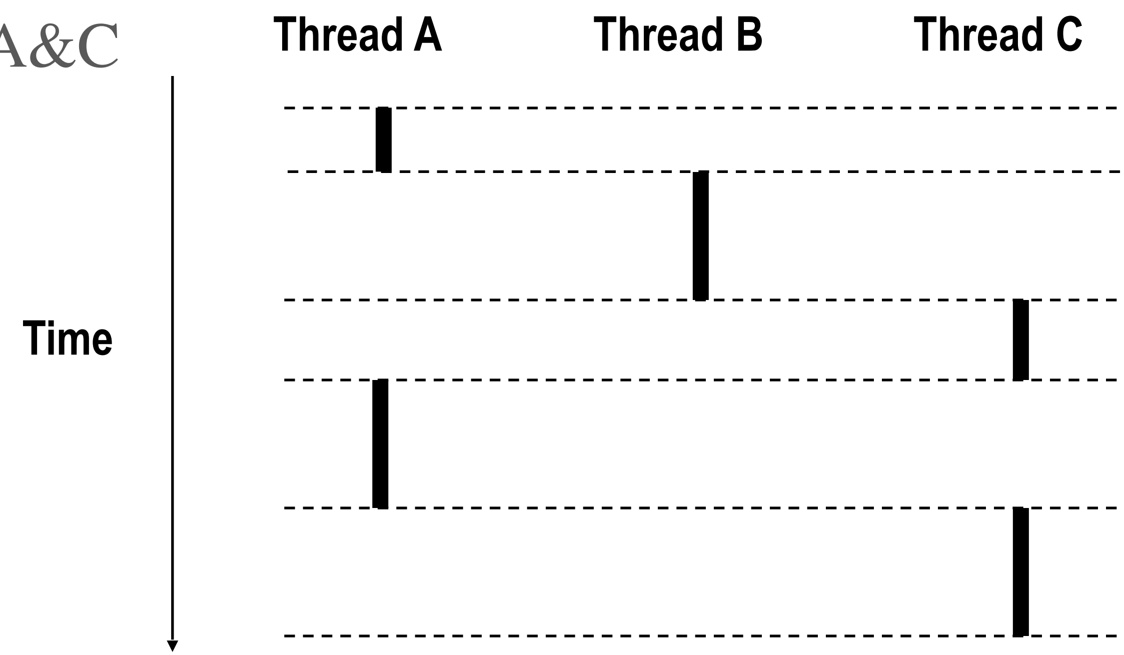



Process hierarchy



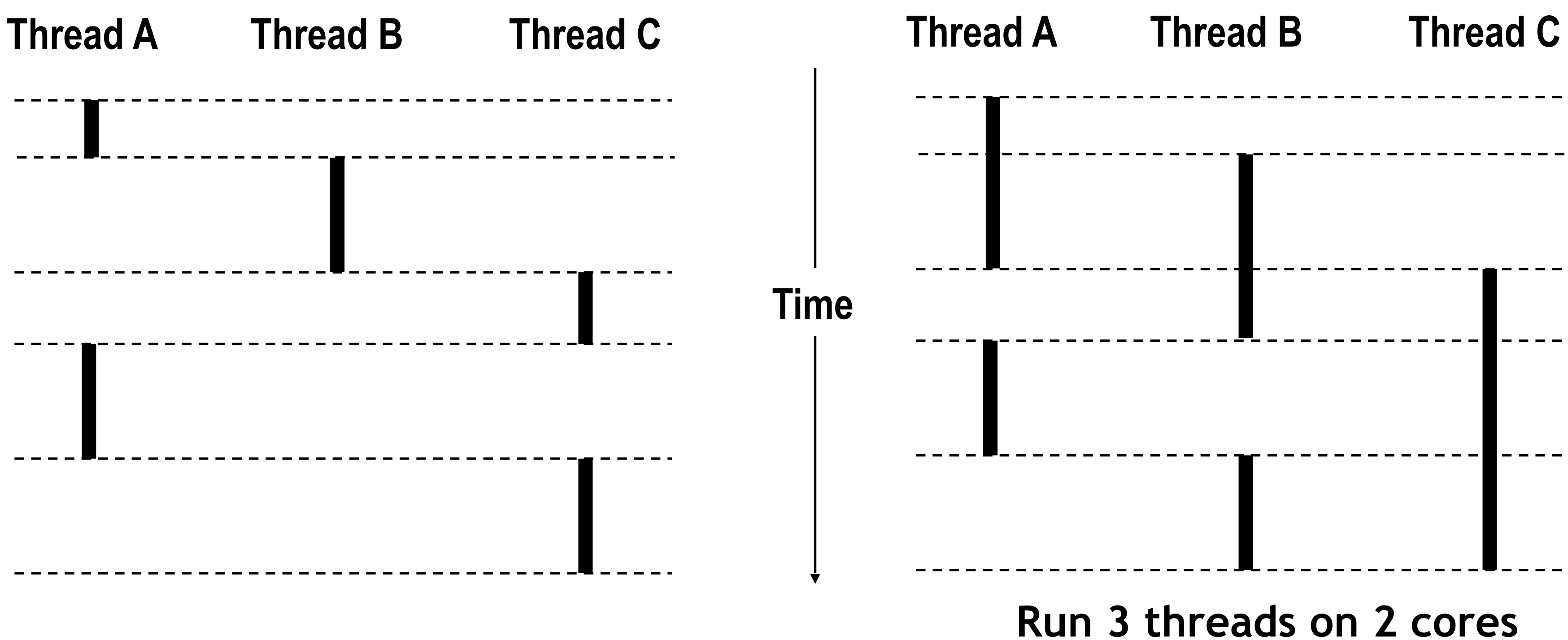
# + Concurrent Threads

- **Two threads are concurrent if their flows overlap in time**
- **Otherwise, they are sequential**
- **Examples:**
  - Concurrent: A & B, A&C
  - Sequential: B & C



# Concurrent vs Parallel Thread Execution

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



# Threads vs. Processes

- **Similarities**
  - Each has its own logical control flow
  - Each can run concurrently (possibly on different cores)
  - Each is context switched
- **Differences**
  - Threads share all code and data (except local stacks)
    - Processes do not
  - Threads are somewhat less expensive than processes
    - Process control (creating/reaping) ***2x*** as expensive as thread control

# + 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] , `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

# The Pthreads "hello, world" Program

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

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}
```

Thread ID

Thread attributes (usually NULL)

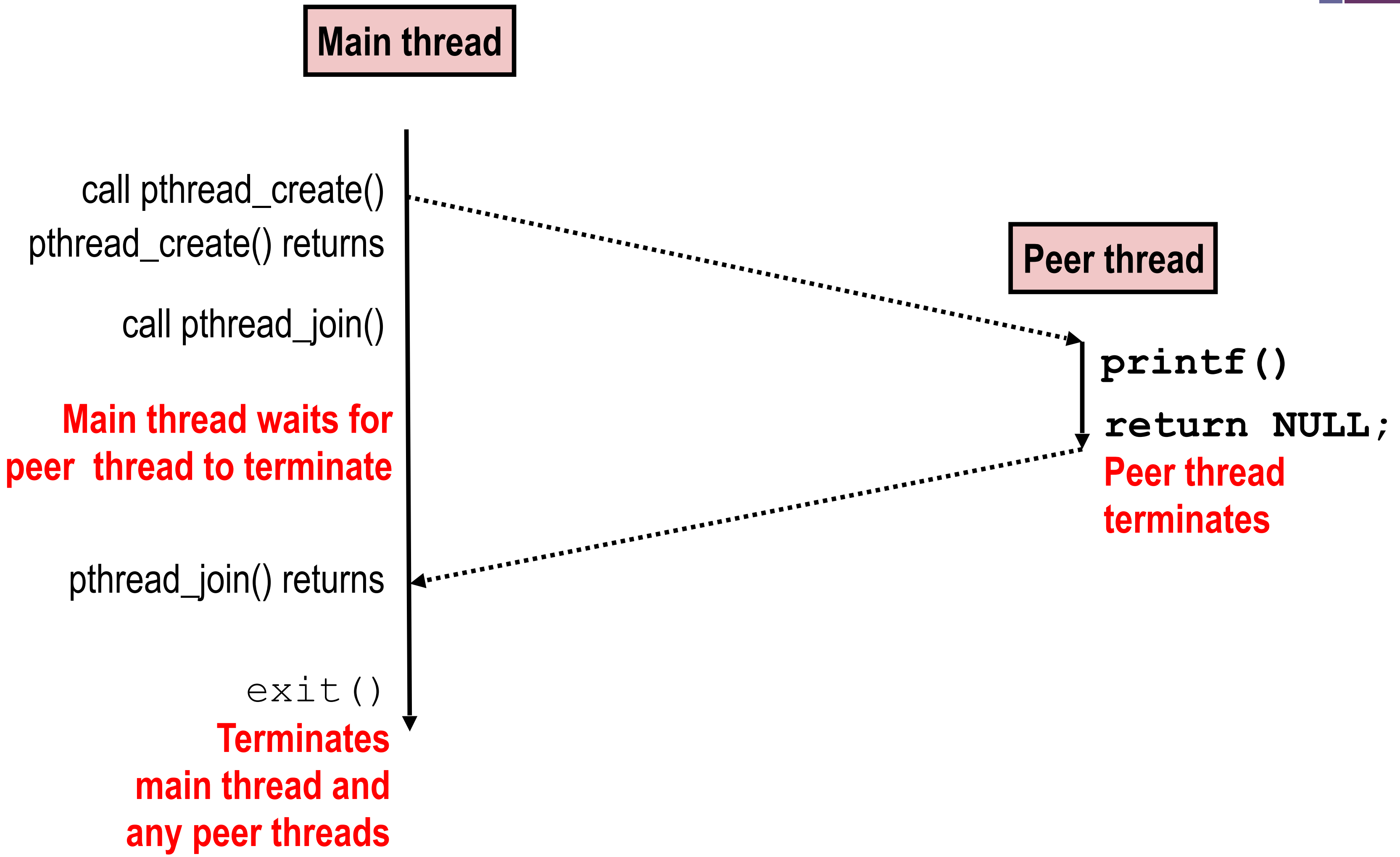
Thread routine

Thread arguments (void *p)

Return value (void **p)

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

# + Execution of Threaded "hello, world"




- See lecture25/thread_sum.c for another example.

# + Concurrent Programming is Hard!

- **The ease with which threads share data and resources also makes them vulnerable to subtle and baffling errors.**
- **Classical problem classes of concurrent programs:**
  - *Races*: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - *Deadlock*: improper resource allocation prevents forward progress
  - *Livelock / Starvation / Fairness*: external events and/or system scheduling decisions can prevent sub-task progress

# + Race Example

- **What's the expected output on line 11?**
  - 2
- **Possible output…**
  - 1
- **??????**

```c
1  int numbers[2] = { 1, 1 };
2  int sum = 0;
3
4  int main() {
5    pthread_t tid;
6    pthread_create(&tid,NULL,run,numbers[1]);
7    for (int i = 0; i < 1; i++) {
8        sum += numbers[i];
9    }
10   pthread_join(tid, NULL);
11   printf("sum is %d\n", sum);
12 }
13
14 void* run(void* arg) {
15   int* numbers = (int*) arg;
16   for (int i = 0; i < 1; i++) {
17       sum += numbers[i];
18   }
19   return NULL;
20 }
```

# Race Example *con't*

- **Why can the outcome be 1? This line is the culprit.**

```
sum += numbers[i];
```

- **What does this look like in assembly?**

```
movq ...(,%rdi,4), %rcx
movq ...(%rsi), %rdx
addq %rcx, %rdx
movq %rdx, ...(%rsi)
```

- **Two threads T, T' have combinatorial number of interleavings**
  - **OK:** `T1, T2, T3, T4, T'1, T'2, T'3, T'4`
  - **BAD:** `T1, T'1, T2, T'2, T3, T'3, T4, T'4`
- Global variable `sum` is written as 1 by both threads at `T4` & `T'4`

# + Pros and Cons of Thread-Based Designs

- **Pros**
  - Easy to share data structures between threads
  - Threads are more efficient than processes
- **Cons**
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors.
    - Hard to detect by testing since probability of bad outcome can be low

# + Summary: Approaches to Concurrency

- **Process-based**
  - Hard to share resources: easy to avoid unintended sharing
  - High overhead in adding/removing clients
- **Thread-based**
  - Easy to share resources: perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug - event orderings not repeatable