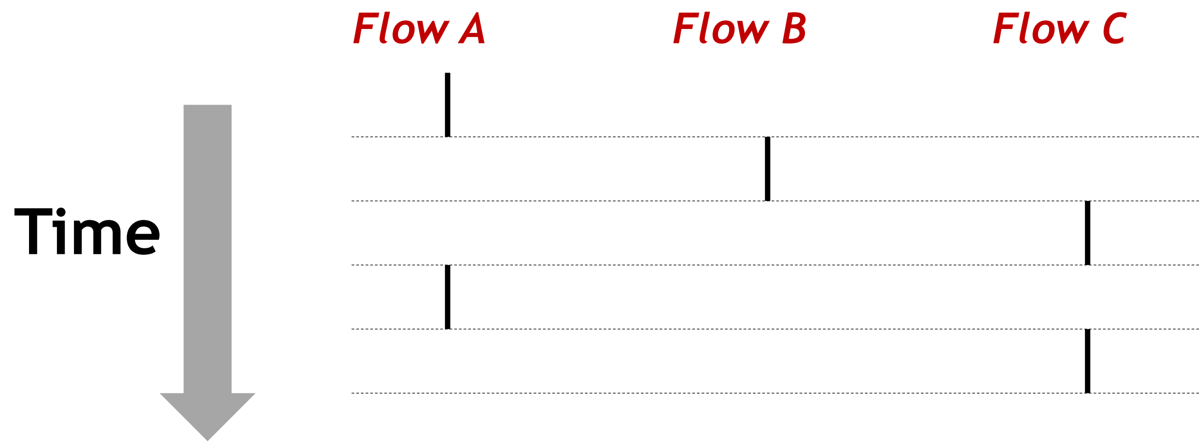
Concurrent Programming

+ 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: examples**
  - e.g. single CPU interleaving instructions from two flows
  - e.g. multiple CPU cores executing two flows at the same time
  - 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**
- **Reasoning about all possible sequences of interleaved control flows is at least error prone and often impossible.**
  - Imagine two control flows of 2 instructions.
    - `A,B`
    - `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
  - Concurrent flows spawned by forking child processes
  - Each flow has its *own private address space*
- **Thread-based**
  - Kernel automatically interleaves multiple logical flows
  - Concurrent flows spawned by creating threads
  - Each flow *shares the same address space*
  - *Threads exist within a process, possibly many of them*

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? 500**
- **Two processes concurrently sum parts of the array.**
- **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*
  - via *shared memory objects*
  - ….there are others

+ Sockets

- **What is a socket?**
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
- **Clients and servers communicate with each other by reading from and writing to sockets**

- **For IPC, however, the "client" and the "server" can just be different processes on the same machine!**
- **This provides a way for parent and child processes to share data.**

+ Pros & Cons of Sockets

- **Pros**
  - Persistent bi-directional communication.
  - Processes can be on same or different computers.
  - Easy to create, well-know programming interface.
- **Cons**
  - Multiple sockets necessary if you want to send the same data to multiple processes.
  - All messages pass through OS, so resource intensive.

+ 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.**
- **There are a few ways to use pipes, here we will see two.**

Pipes in C

- The `pipe` system call is called with a pointer to an array of two integers.
- The 0th element of the array contains the file descriptor that corresponds to the output of the pipe
- The 1st 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.
  - Very useful on the command line (as in, everyday useful)
- **Cons**
  - Can be confusing quickly in non-trivial programs
  - Processes using pipes must have a common parent process
  - Uni-directional
  - Multiple pipes necessary if you want to send the same data to multiple processes.
  - All messages pass through OS, so resource intensive.

+ 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`()**
- **Example of two processes using shared memory**
  - 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
    - File descriptors (yes)
    - Address space (no)
- **Cons**
  - Nontrivial to share data between processes
    - Requires interprocess communication
  - Systems calls necessary

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

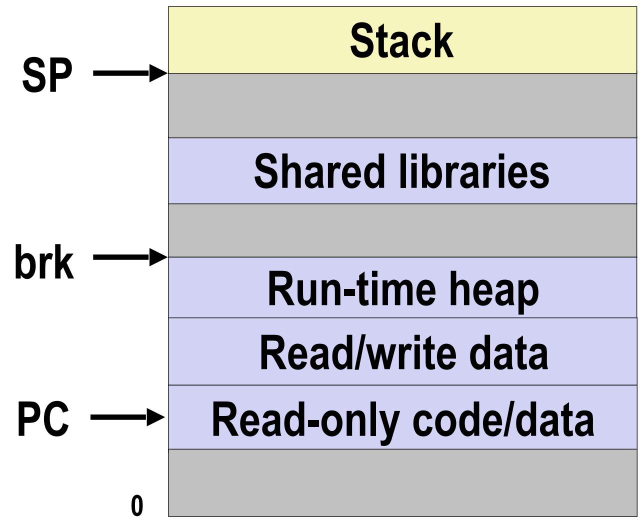
- Process = process context + code, data & stack

Process context

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

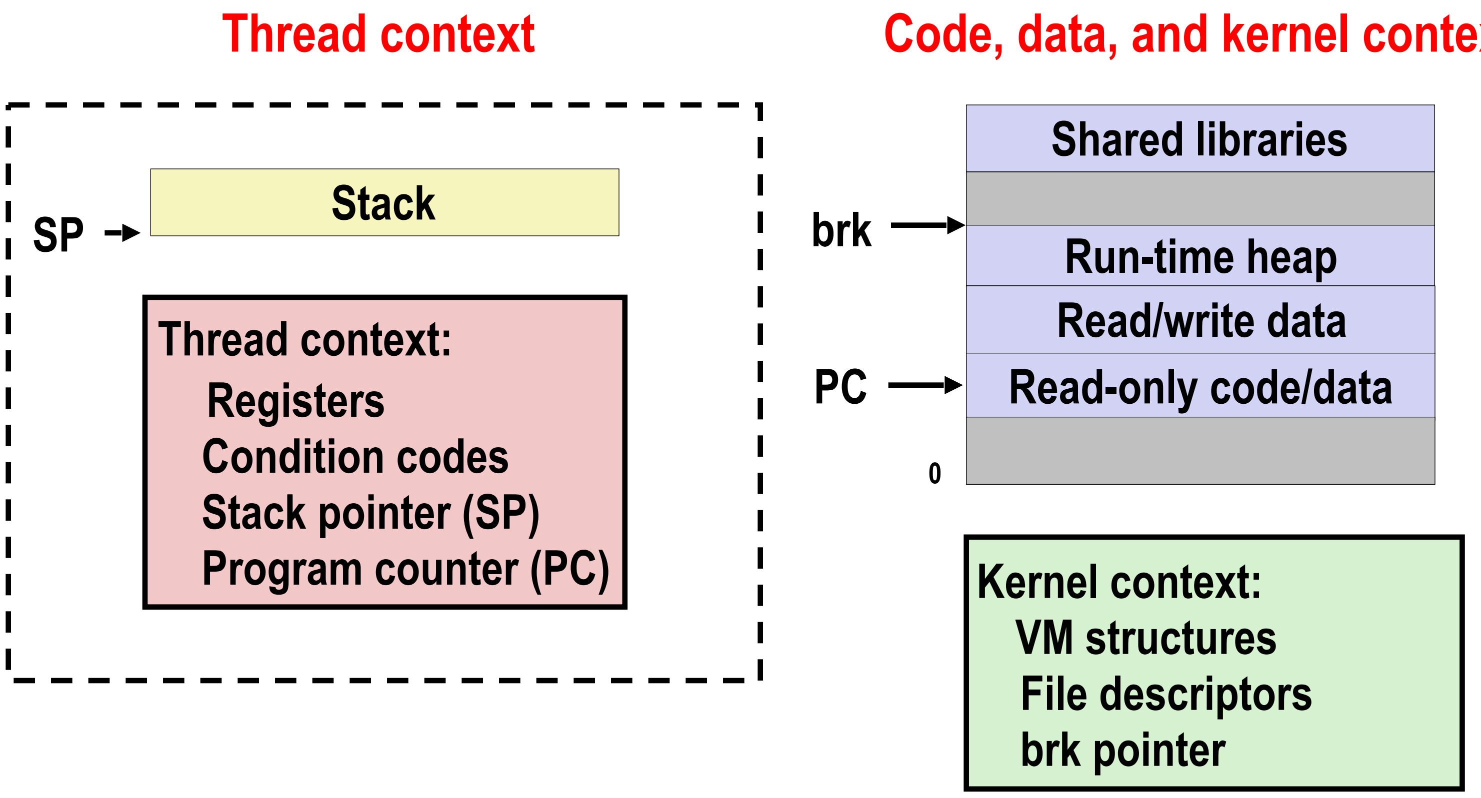
Kernel context:
- VM structures
- File descriptors
- brk pointer

Code, data, and stack



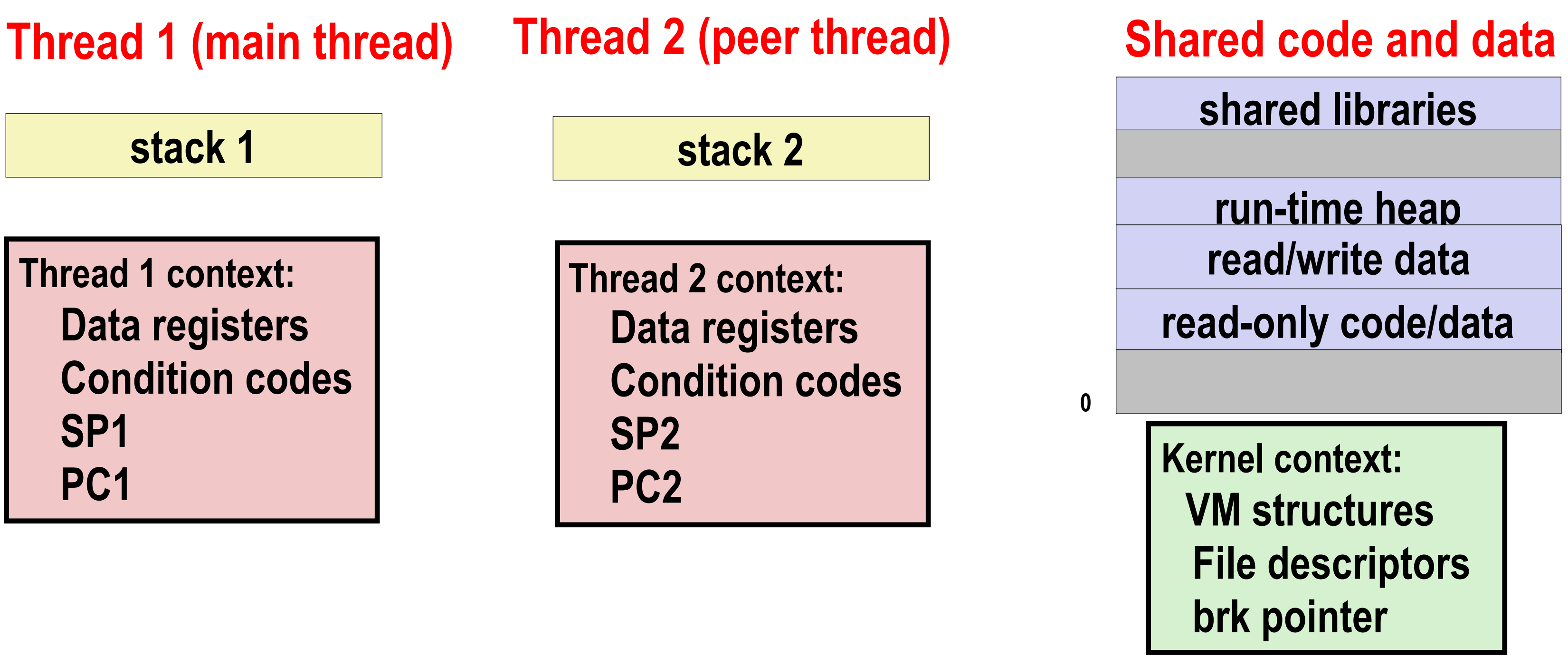
+ Alternate View of a Process

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



A Process With Multiple Threads

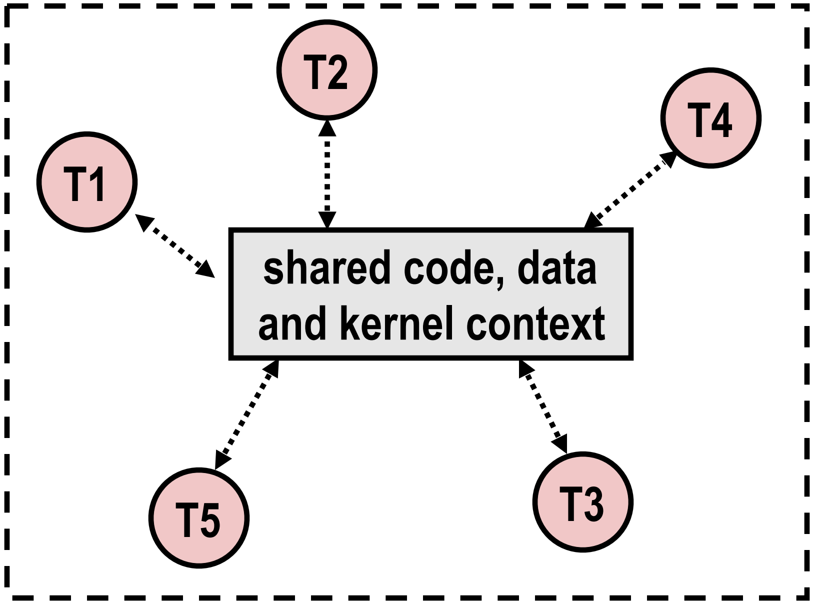
- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, heap, and kernel context
  - Each thread has its own stack for local variables
  - Each thread has its own thread id (TID)



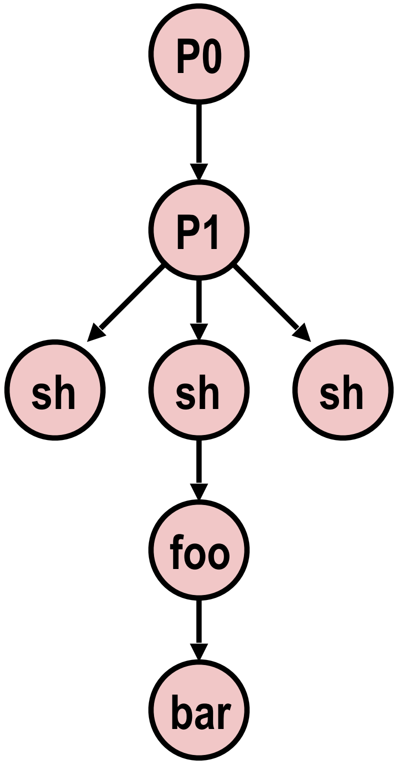
+ Logical View of Threads

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

Threads associated with some process



Process hierarchy



+ 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 code and heap
  - Threads are less expensive than processes
    - Process control (creating/reaping) ***2x*** as expensive as thread

Threads in C

+ Posix Threads (Pthreads) Interface

- **Pthreads: ~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()` *(kills all threads)*
  - Most threaded programs use a small subset
  - *See book for more*

+ The Pthreads "hello, world" Program

```c
/*
 * 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)

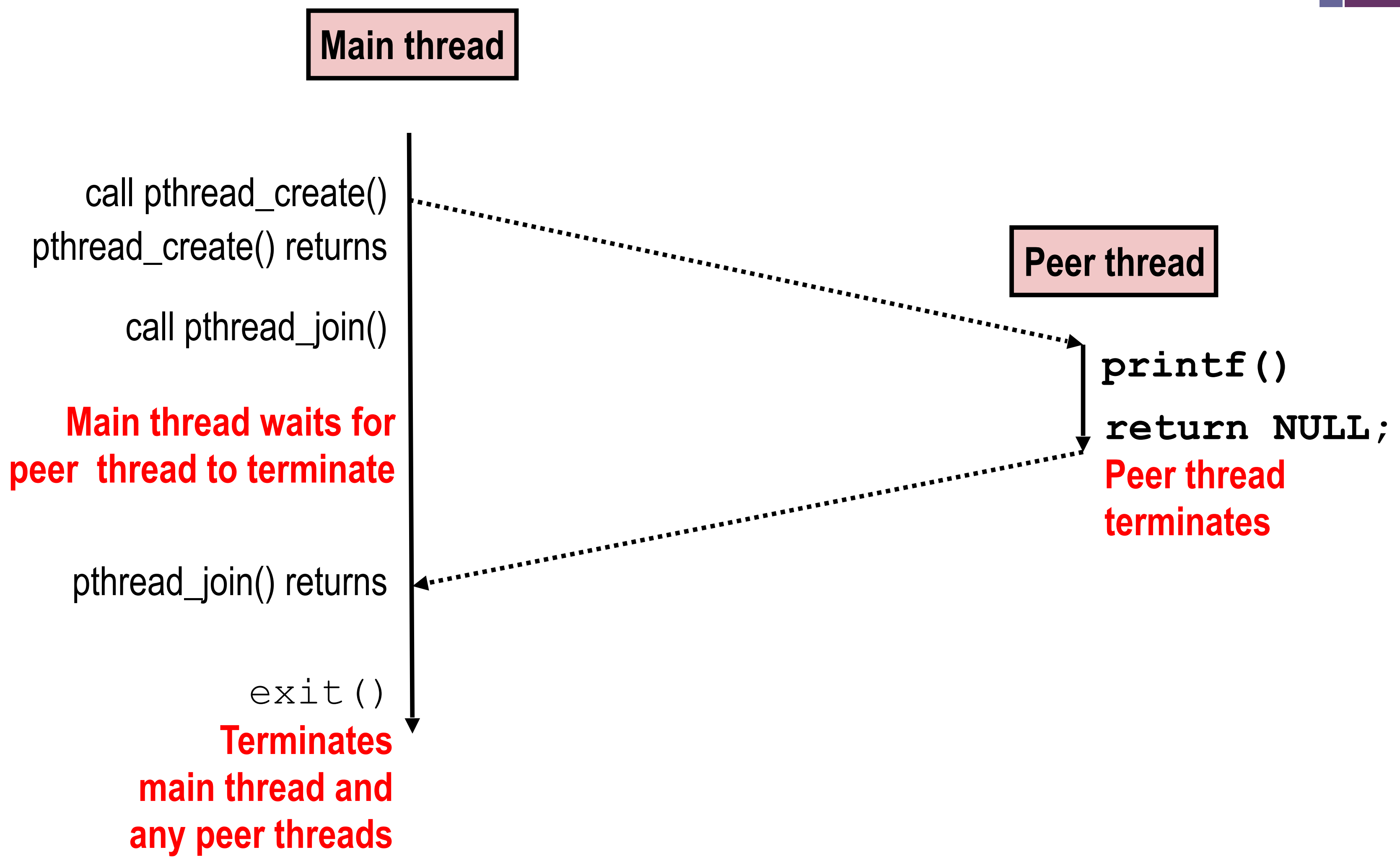
Thread routine

Thread arguments (void p)*

*Return value (void** p)*

```c
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.