Concurrent Programming:
Concurrent Servers:
3. *Thread based*

Approaches for Writing Concurrent Servers

Allow server to handle multiple clients concurrently

1. Process-based

- Kernel automatically interleaves multiple logical flows
- Each flow has its own private address space

2. Event-based

- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

3. Thread-based

- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of of process-based and event-based.

Traditional View of a Process

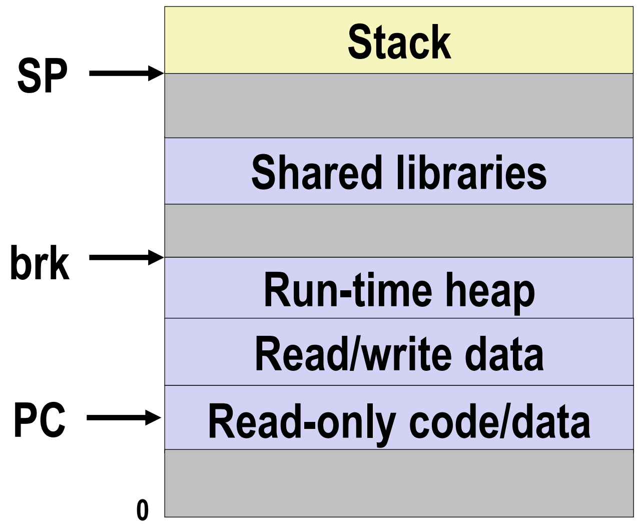
- Process = process context + code, data, and stack

Process context

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

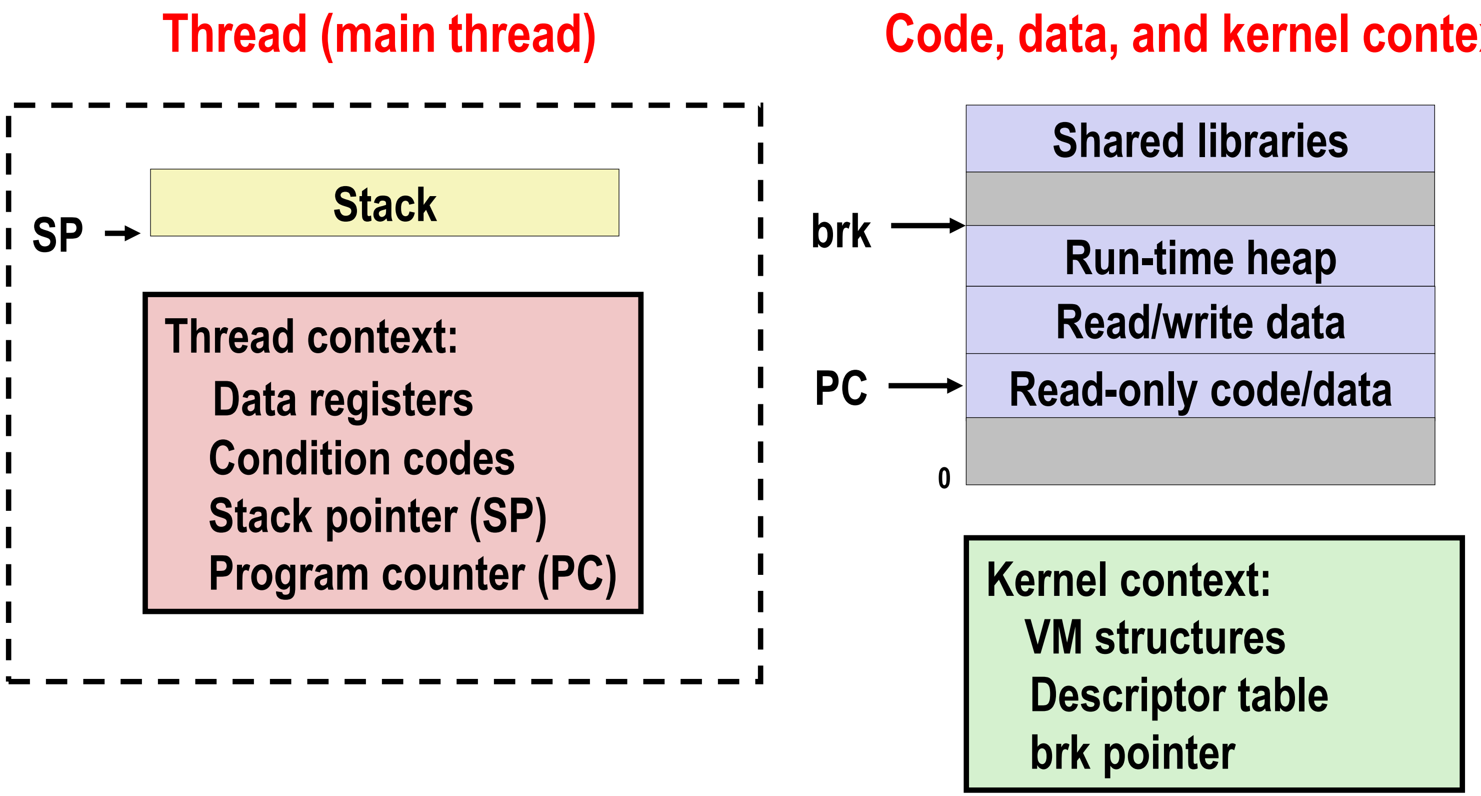
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



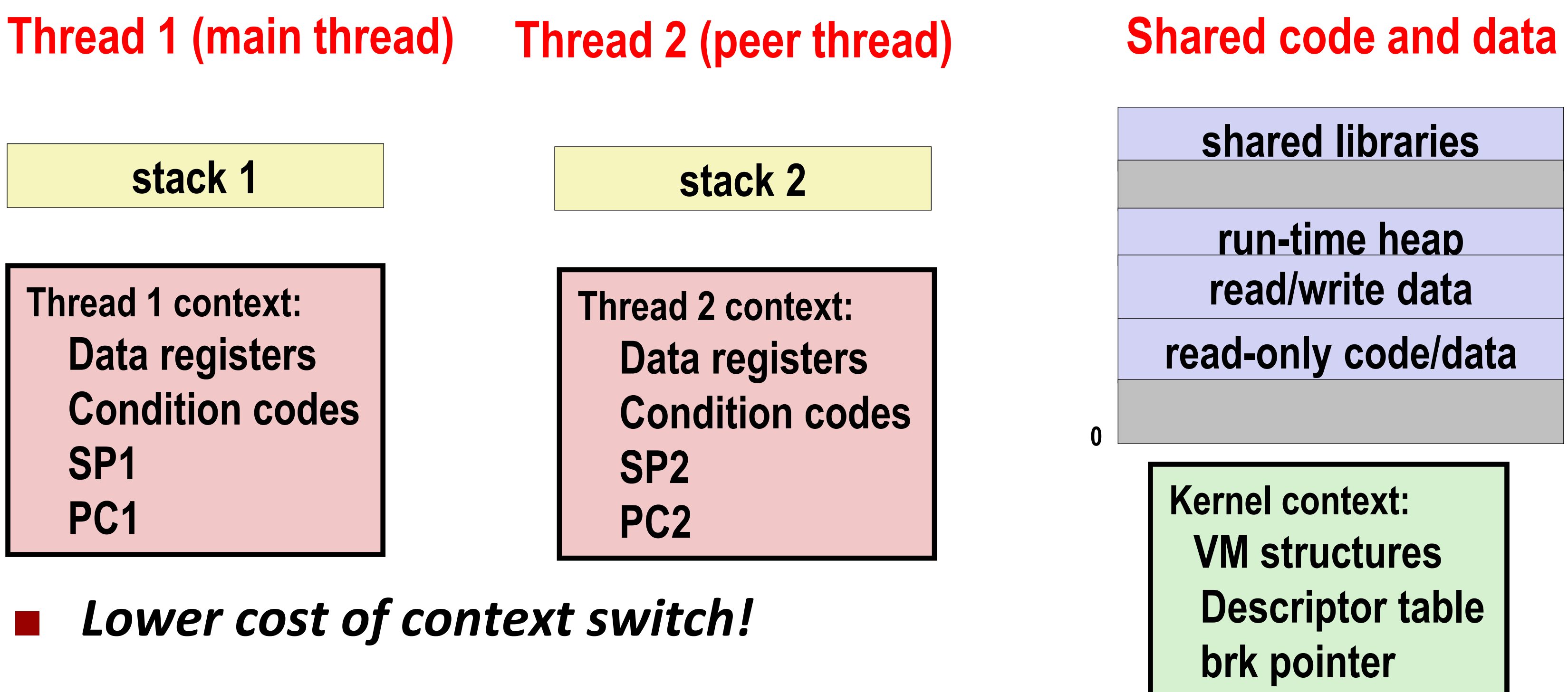
Alternate View of a Process

- Process = thread + code, data, and kernel context



A Process With Multiple Threads

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

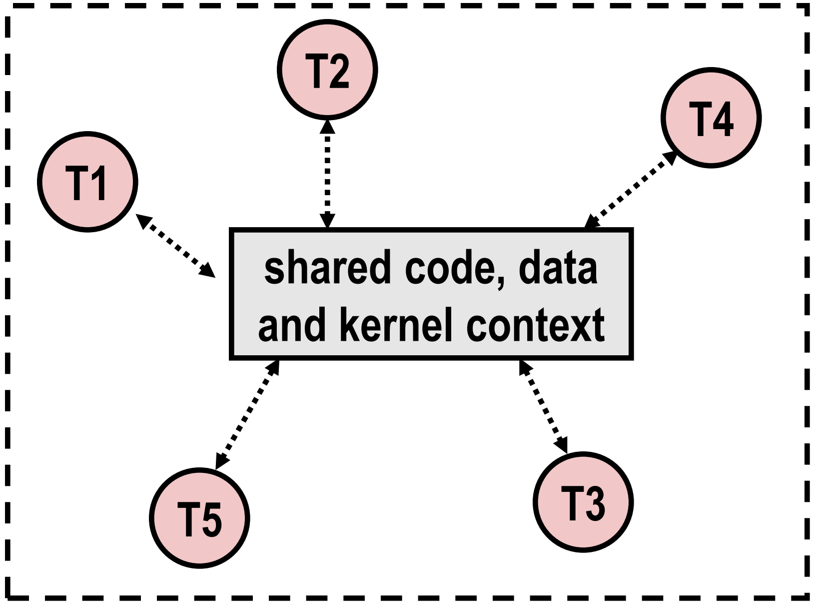


- ***Lower cost of context switch!***

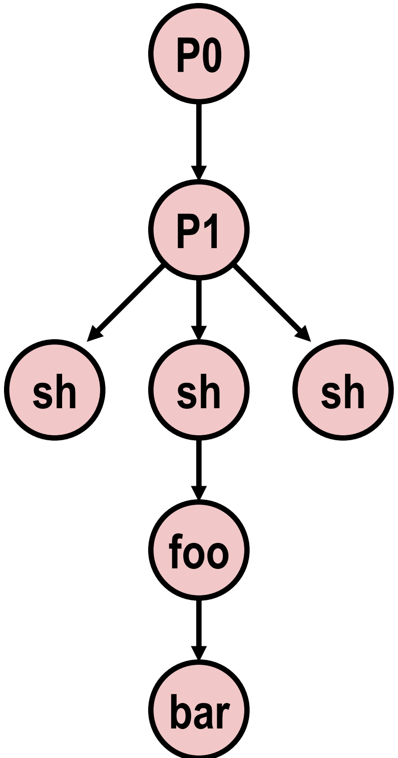
Logical View of Threads

- **Threads associated with process form a *pool* of peers**
  - Unlike processes which form a tree hierarchy
  - Possibility of mix-and-matching, but gets complex

Threads associated with process foo



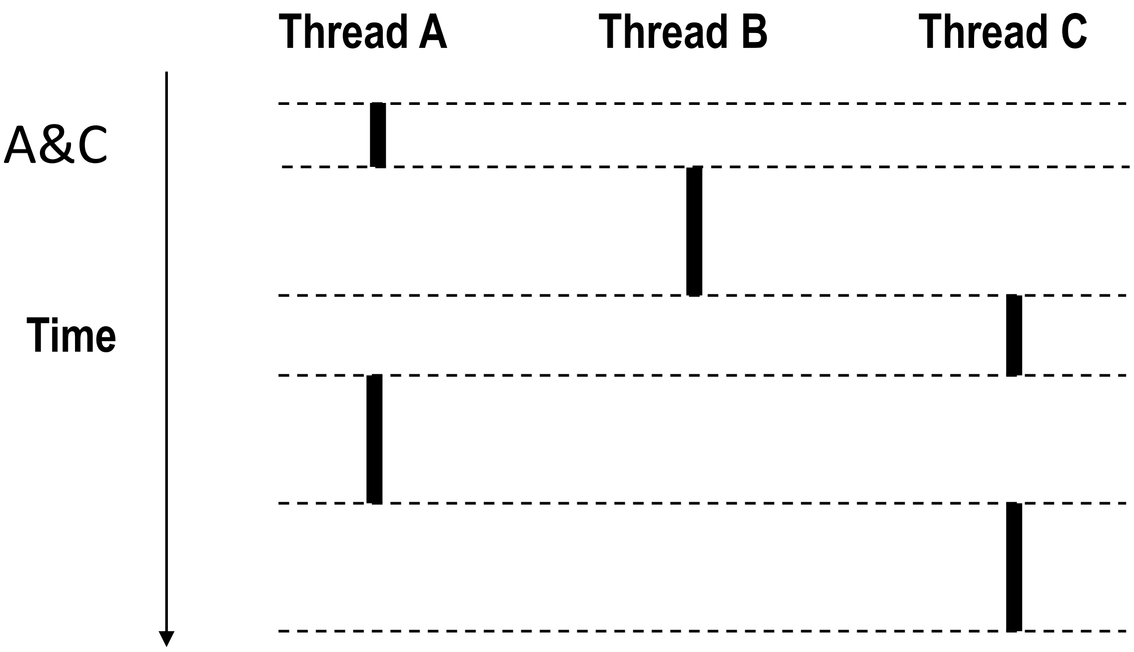
Process hierarchy



Concurrent Threads

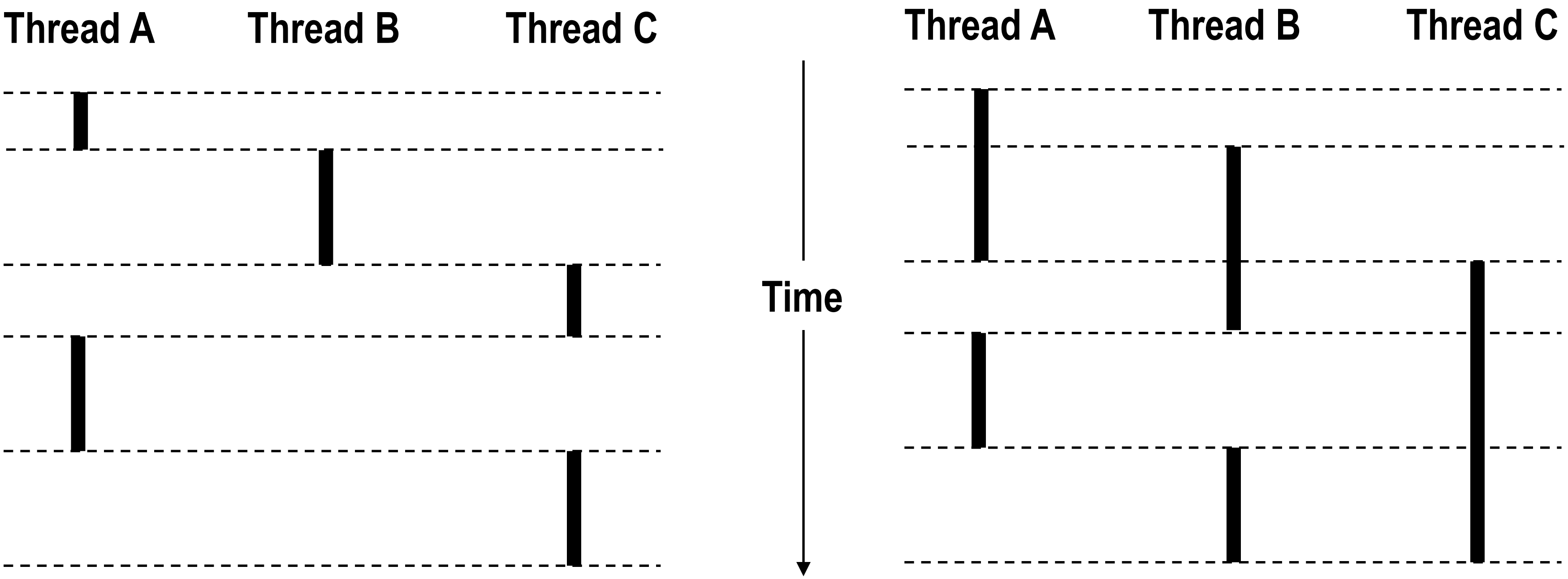
- **Two threads are *concurrent* if their flows overlap in time**
- **Otherwise, they are sequential**

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



Concurrent Thread Execution

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



Run 3 threads on 2 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 (typically) 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 thread interface (~60 functions)**
  - Creating and reaping threads
    - `pthread_create(3)`
    - `pthread_join(3)`
  - Determining your thread ID
    - `pthread_self(3)`
  - Terminating threads
    - `pthread_cancel(3)`
    - `pthread_exit(3)`
    - `exit(3)` [terminates all threads]
  - Synchronizing access to shared variables
    - `pthread_mutex_init(3)`
    - `pthread_mutex_[un]lock(3)`
- **libc implemented `(-lpthread)`, syscall is `clone(2)`**
  - See https://nullprogram.com/blog/2015/05/15/ for raw calls

The Pthreads "hello, world" Program

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

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

hello.c

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

hello.c

The Pthreads "hello, world" Program

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

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

Thread ID

hello.c

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

hello.c

The Pthreads "hello, world" Program

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

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

hello.c

Thread ID

Thread attributes (usually NULL, see pthread_attr_init(3))

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

hello.c

The Pthreads "hello, world" Program

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

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

hello.c

Thread ID

Thread attributes (usually NULL, see pthread_attr_init(3))

Thread routine

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

hello.c

The Pthreads "hello, world" Program

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

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

hello.c

Thread ID

Thread attributes (usually NULL, see pthread_attr_init(3))

Thread routine

Thread arguments (void *p)

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

hello.c

The Pthreads "hello, world" Program

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

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

hello.c

Thread ID

Thread attributes (usually NULL, see pthread_attr_init(3))

Thread routine

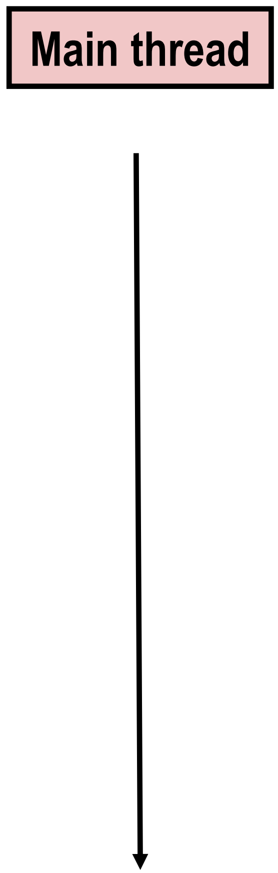
*Thread arguments (void *p)*

*Return value (void **p)*

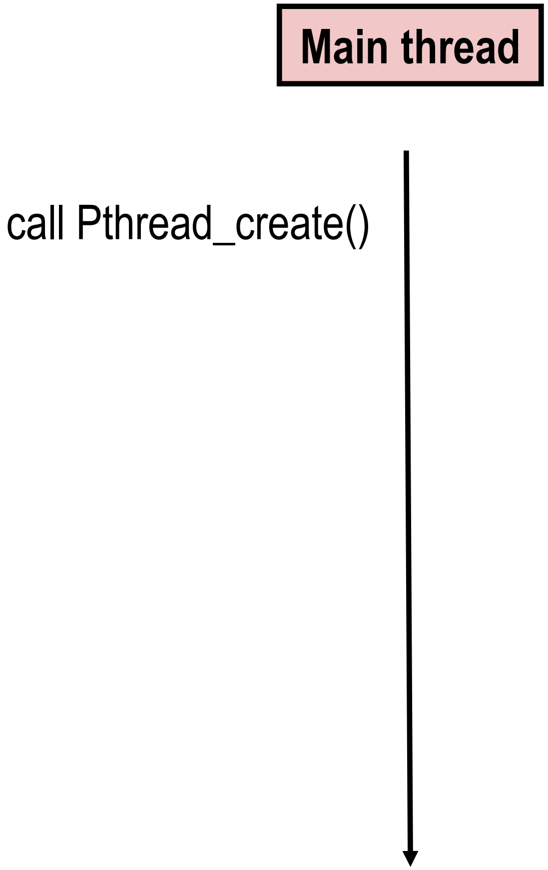
```
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

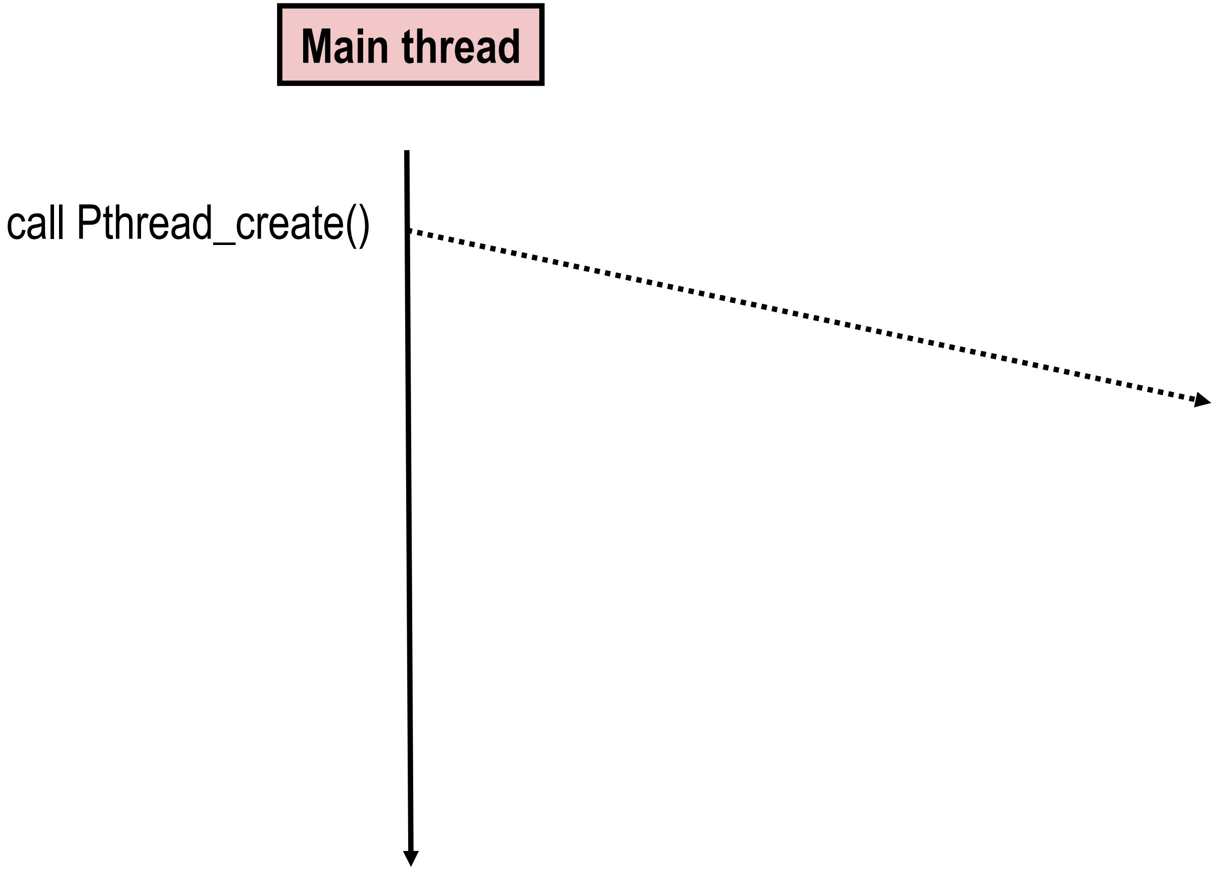
Execution of Threaded "hello, world"



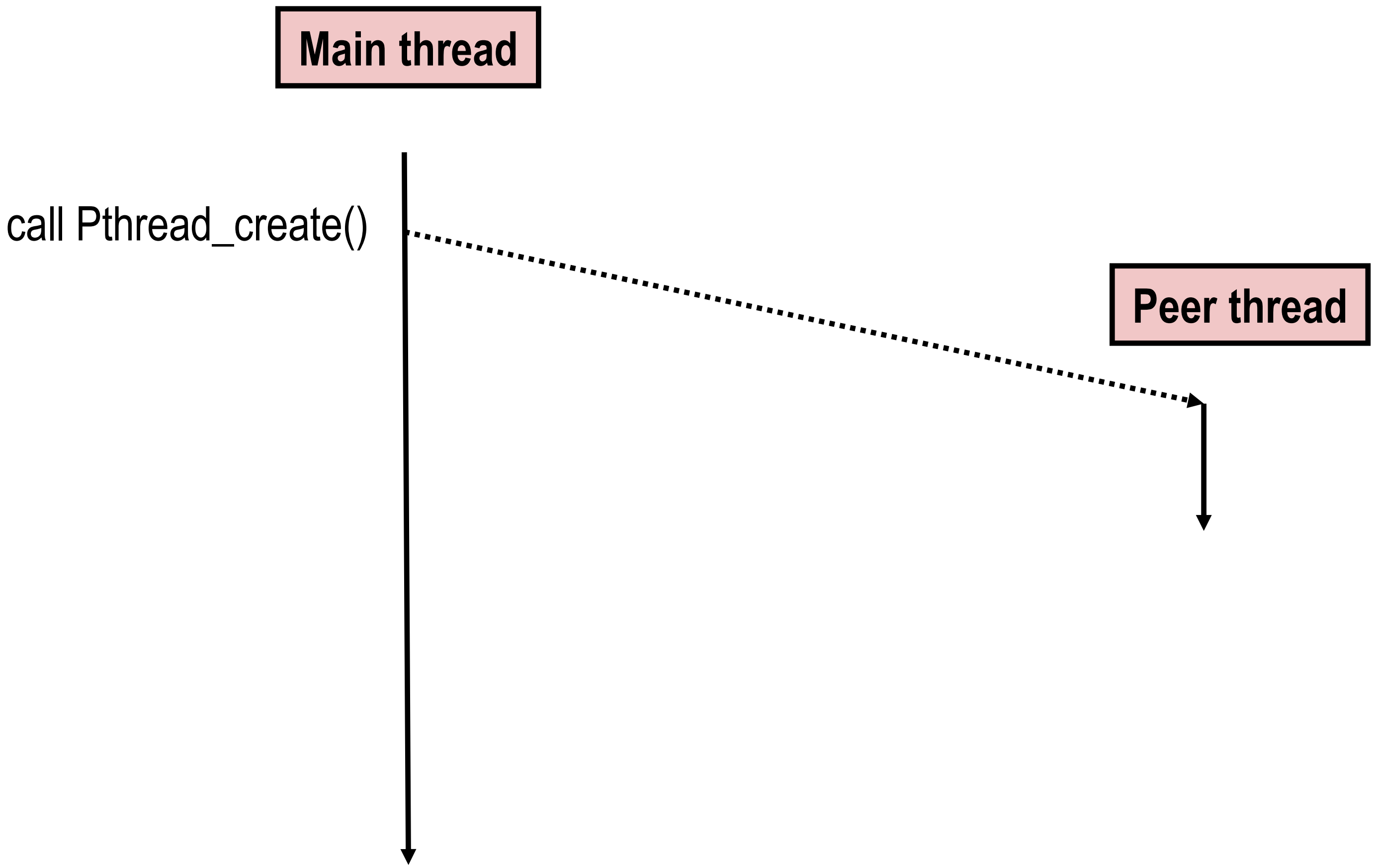
Execution of Threaded "hello, world"



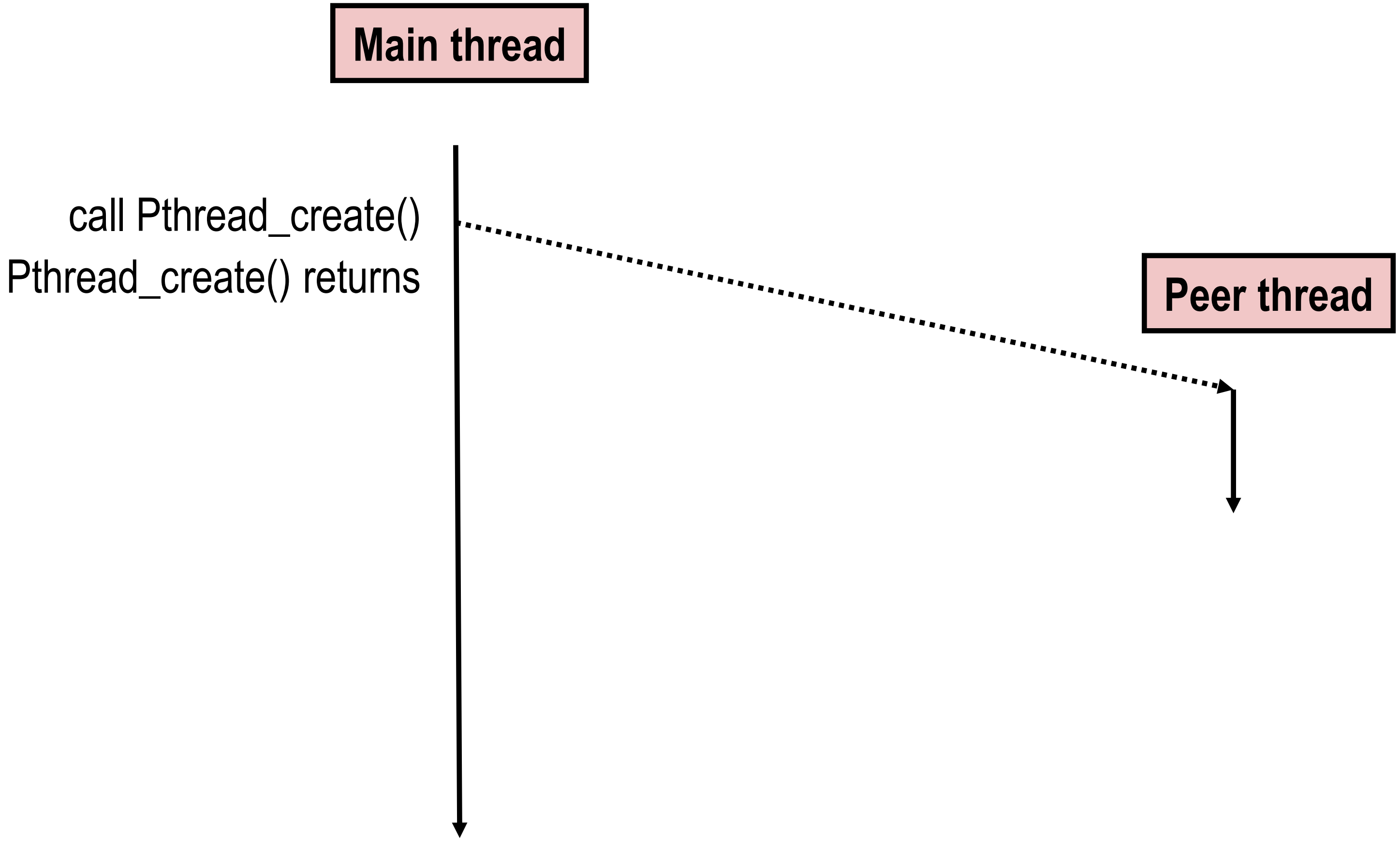
Execution of Threaded “hello, world”



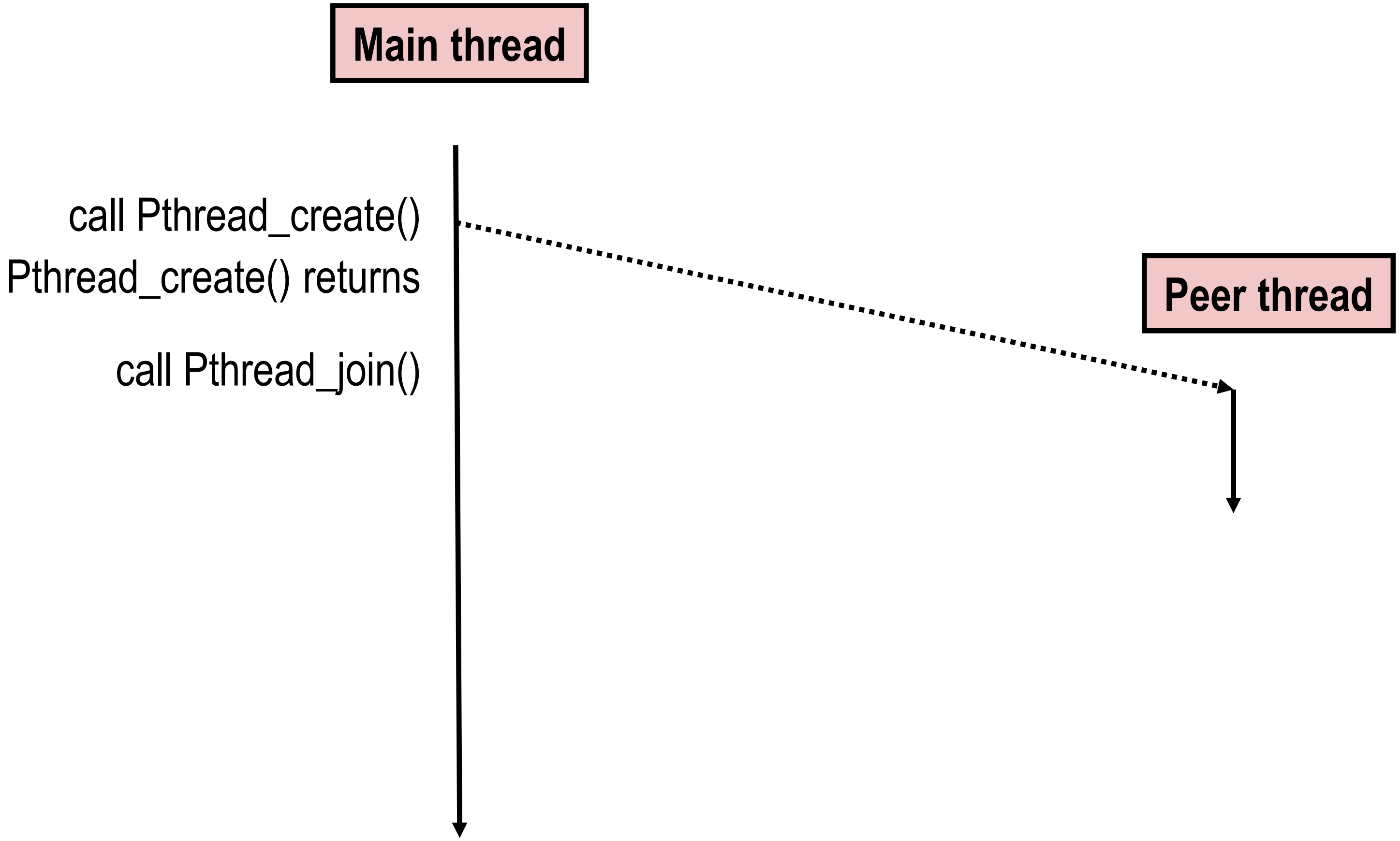
Execution of Threaded "hello, world"



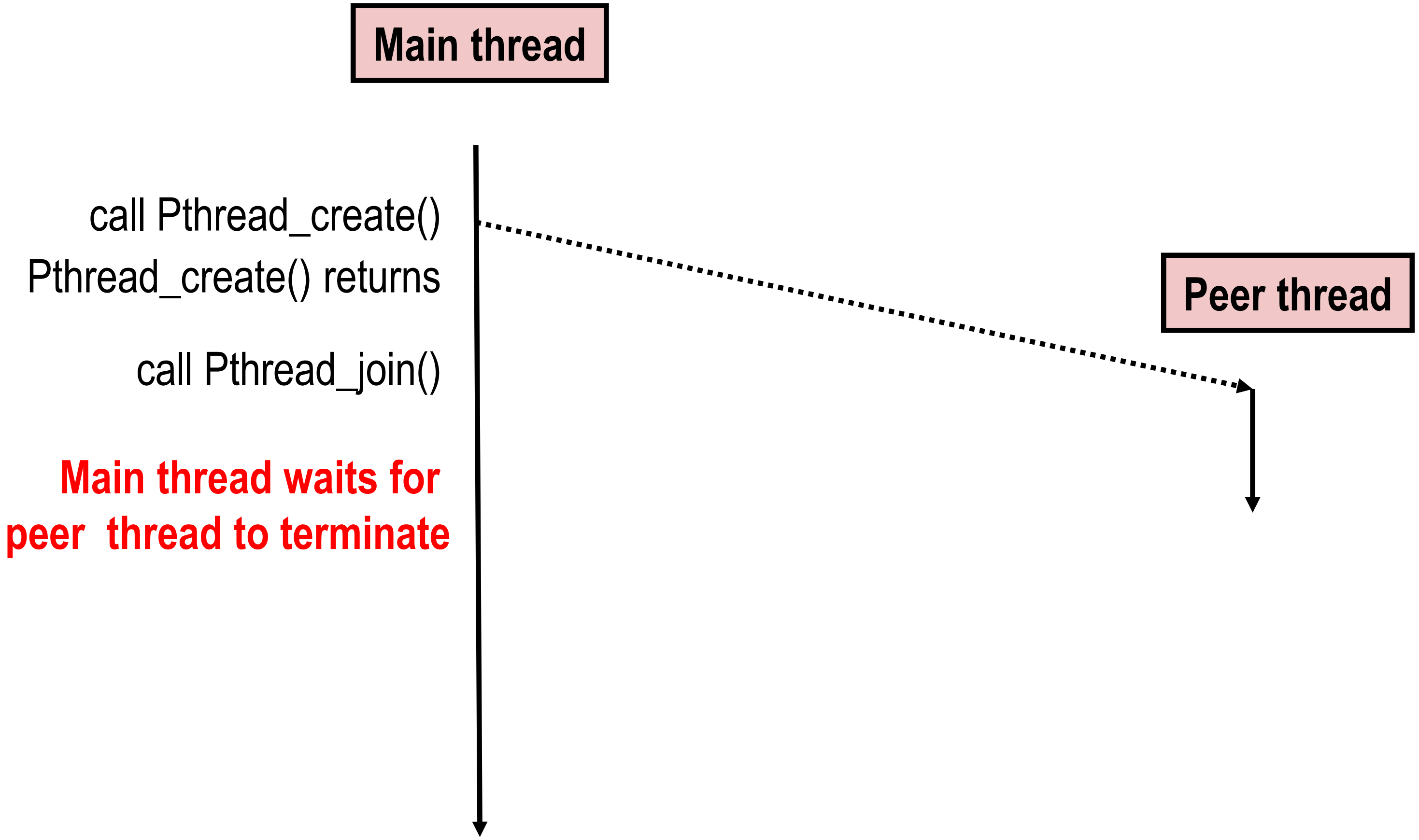
Execution of Threaded “hello, world”



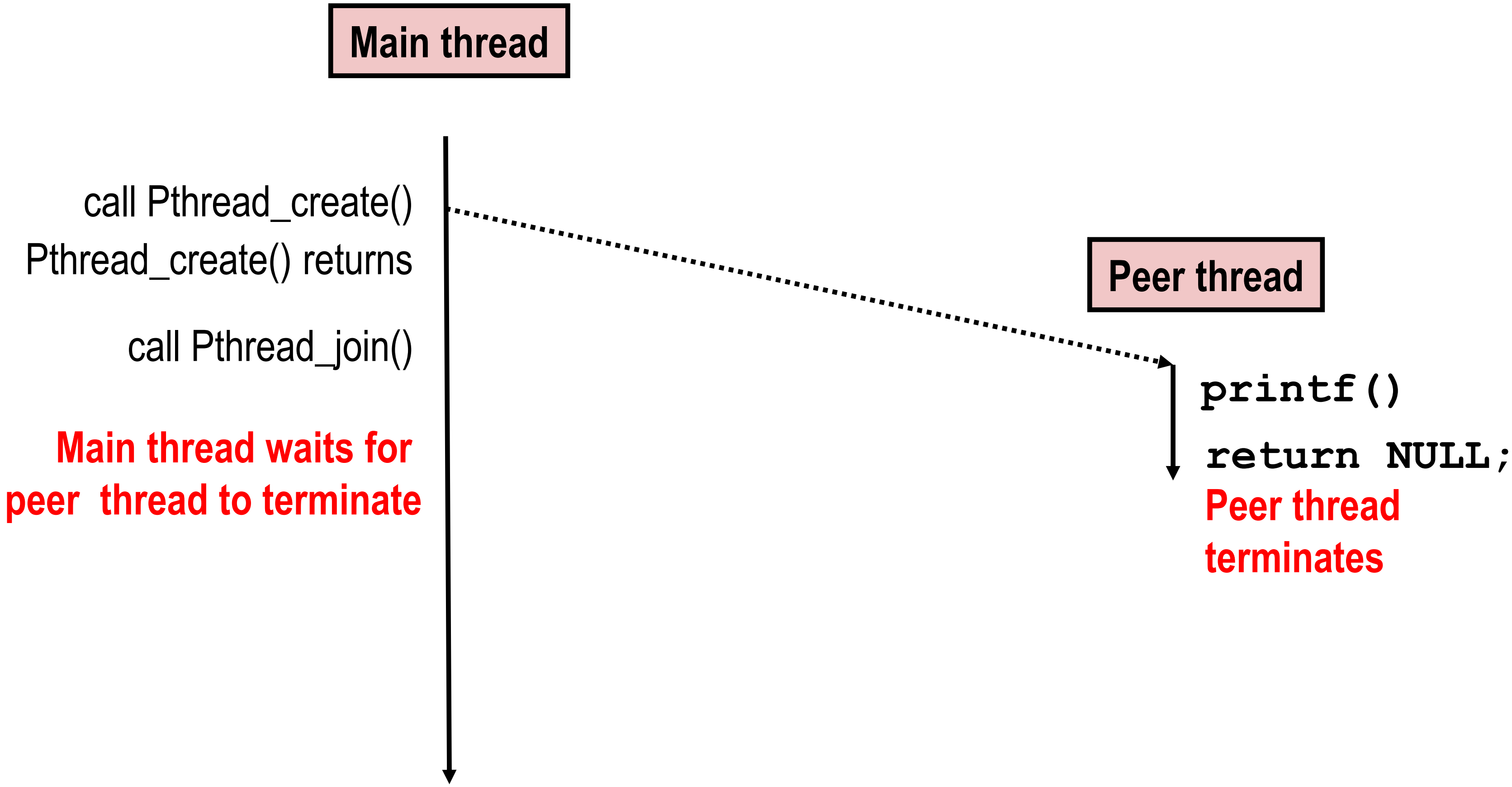
Execution of Threaded “hello, world”



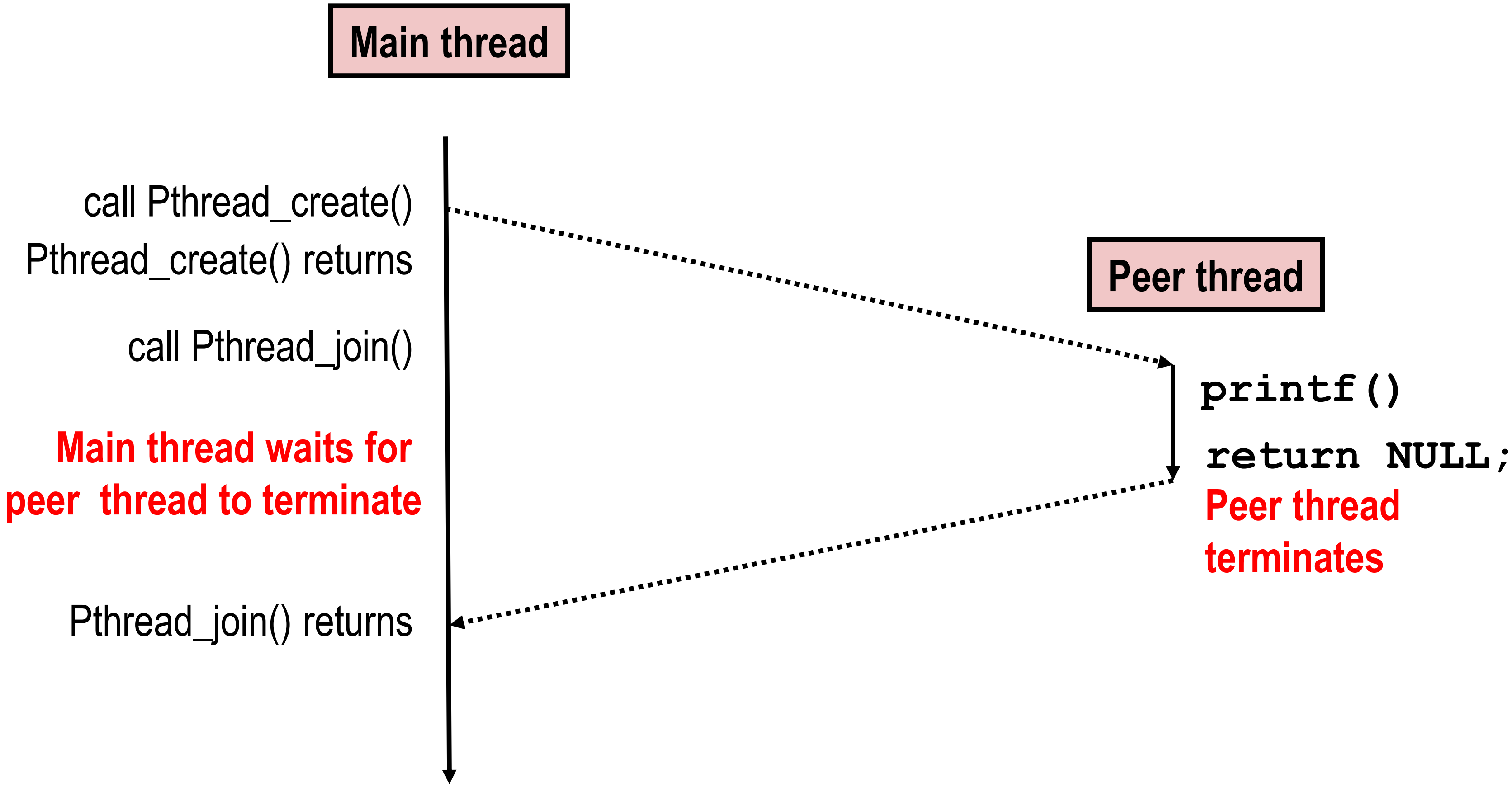
Execution of Threaded "hello, world"



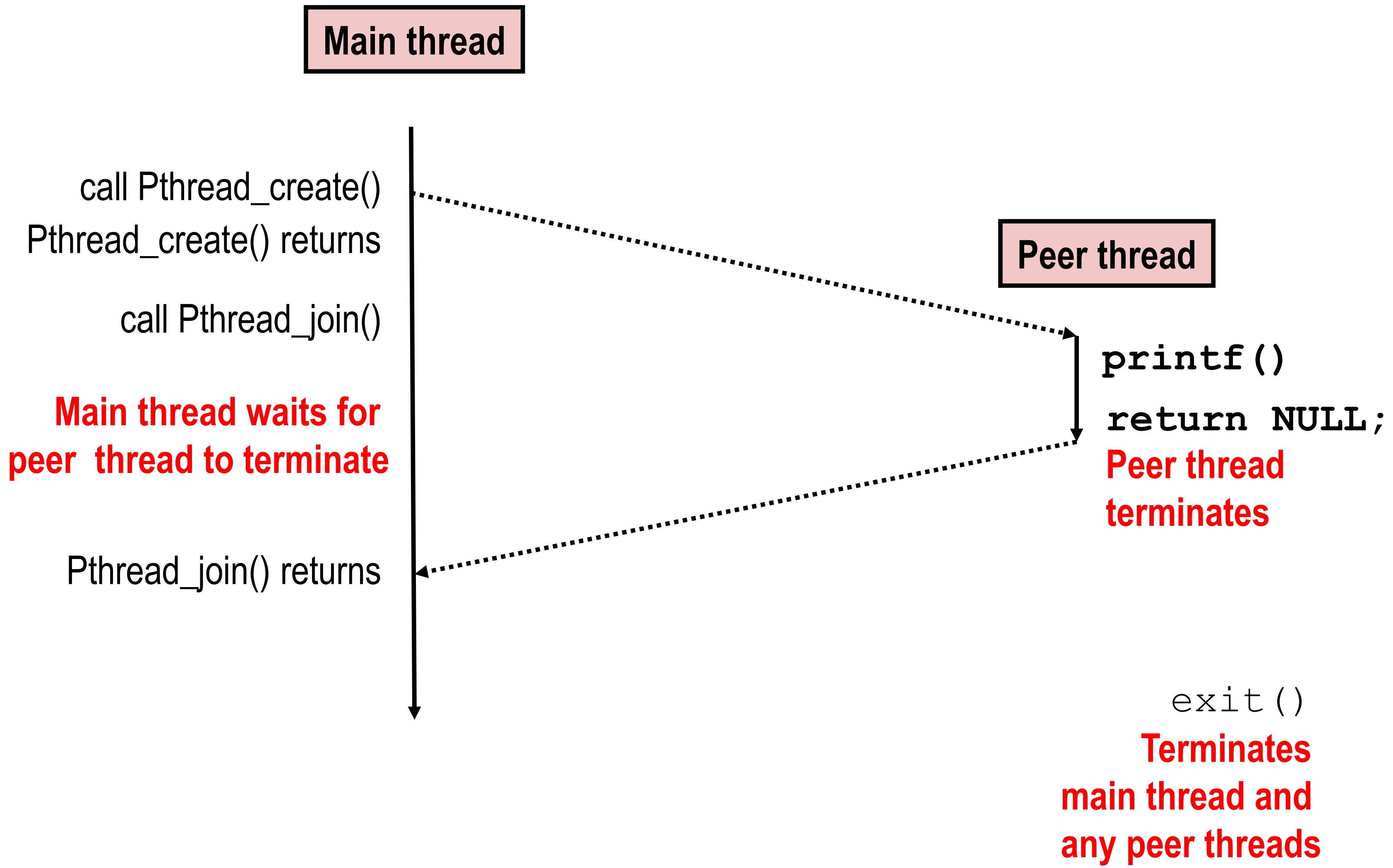
Execution of Threaded "hello, world"



Execution of Threaded "hello, world"



Execution of Threaded "hello, world"



Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                     (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

echoservert.c

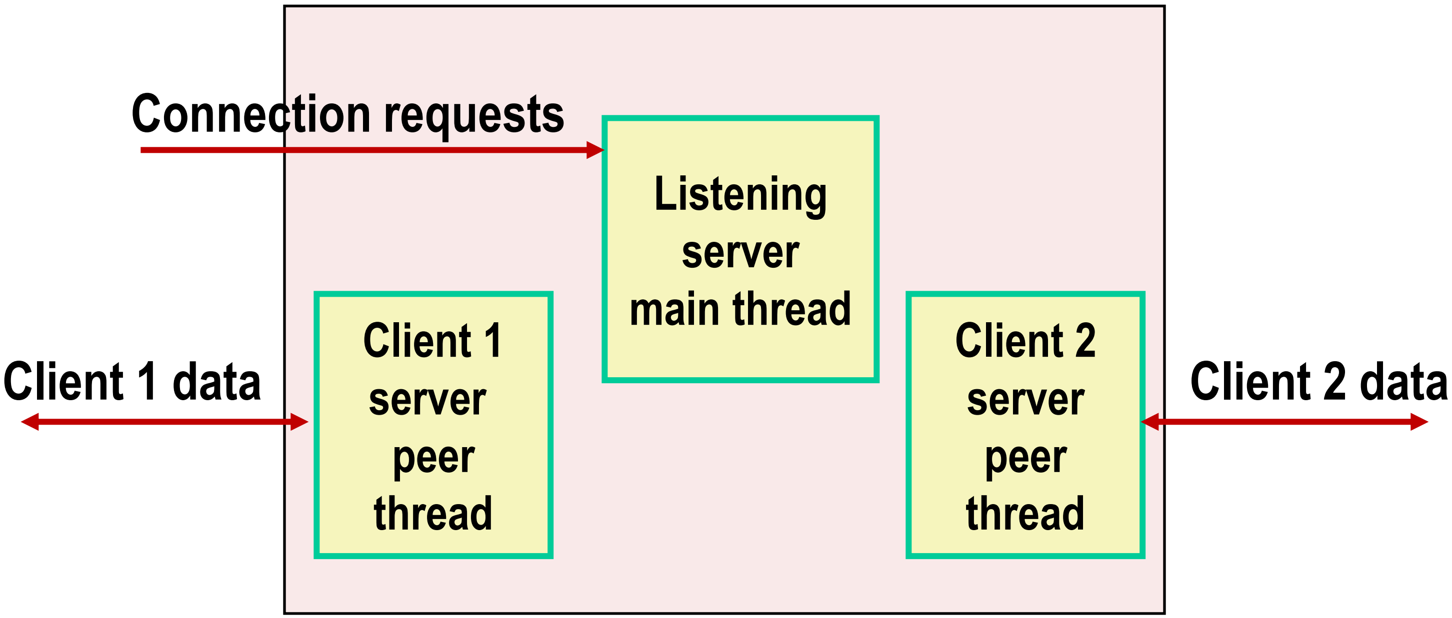
- **`malloc` of connected descriptor necessary to avoid deadly race (later)**
- **Should NOT close connfd!**

Thread-Based Concurrent Server (cont)

```
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```
echoservert.c

- Run thread in “detached” mode.
  - Can’t be made joinable again
  - Reaped automatically (by kernel) when it terminates
  - Either `pthread_detach` or `pthread_join` should be called for each thread, but never both!
- Must free storage allocated to hold `connfd`.
- Must close `connfd` (important, shared FD table!)

Thread-based Server Execution Model



- Each client handled by individual peer thread
- Threads share all process state in kernel, except TID
- Each thread has a separate stack for local variables, and own registers

Issues With Thread-Based Servers

- **Must join or detach**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable (useful for splitting task: “fork-join model”)
    - use `pthread_detach(pthread_self())` to make detached
    - Or create in detach mode with `pthread_attr_setdetachstate`

Issues With Thread-Based Servers

- **Must join or detach**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable (useful for splitting task: "fork-join model")
    - use `pthread_detach(pthread_self())` to make detached
    - Or create in detach mode with `pthread_attr_setdetachstate`
- **Must be careful to avoid unintended sharing**
  - For example, passing pointer to main thread's stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`

Issues With Thread-Based Servers

- **Must join or detach**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable (useful for splitting task: "fork-join model")
    - use `pthread_detach(pthread_self())` to make detached
    - Or create in detach mode with `pthread_attr_setdetachstate`
- **Must be careful to avoid unintended sharing**
  - For example, passing pointer to main thread's stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`
    - Assumes that address dereferenced in peer thread before changed!

Issues With Thread-Based Servers

- **Must join or detach**
  - At any point in time, a thread is either *joinable* or *detached*
  - *Joinable* thread can be reaped and killed by other threads
    - must be reaped (with `pthread_join`) to free memory resources
  - *Detached* thread cannot be reaped or killed by other threads
    - resources are automatically reaped on termination
  - Default state is joinable (useful for splitting task: “fork-join model”)
    - use `pthread_detach(pthread_self())` to make detached
    - Or create in detach mode with `pthread_attr_setdetachstate`
- **Must be careful to avoid unintended sharing**
  - For example, passing pointer to main thread’s stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);`
    - Assumes that address dereferenced in peer thread before changed!
- **All functions called by a thread must be *thread-safe***
  - Much more on that later

Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**

- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!
  - Future lectures

Summary: Approaches to Concurrency

- **Process-based**
  - Hard to share resources/Easy to avoid unintended sharing
  - High overhead in adding and removing clients
- **Event-based**
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core
- **Thread-based**
  - Easy to share resources… Perhaps too easy!
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable