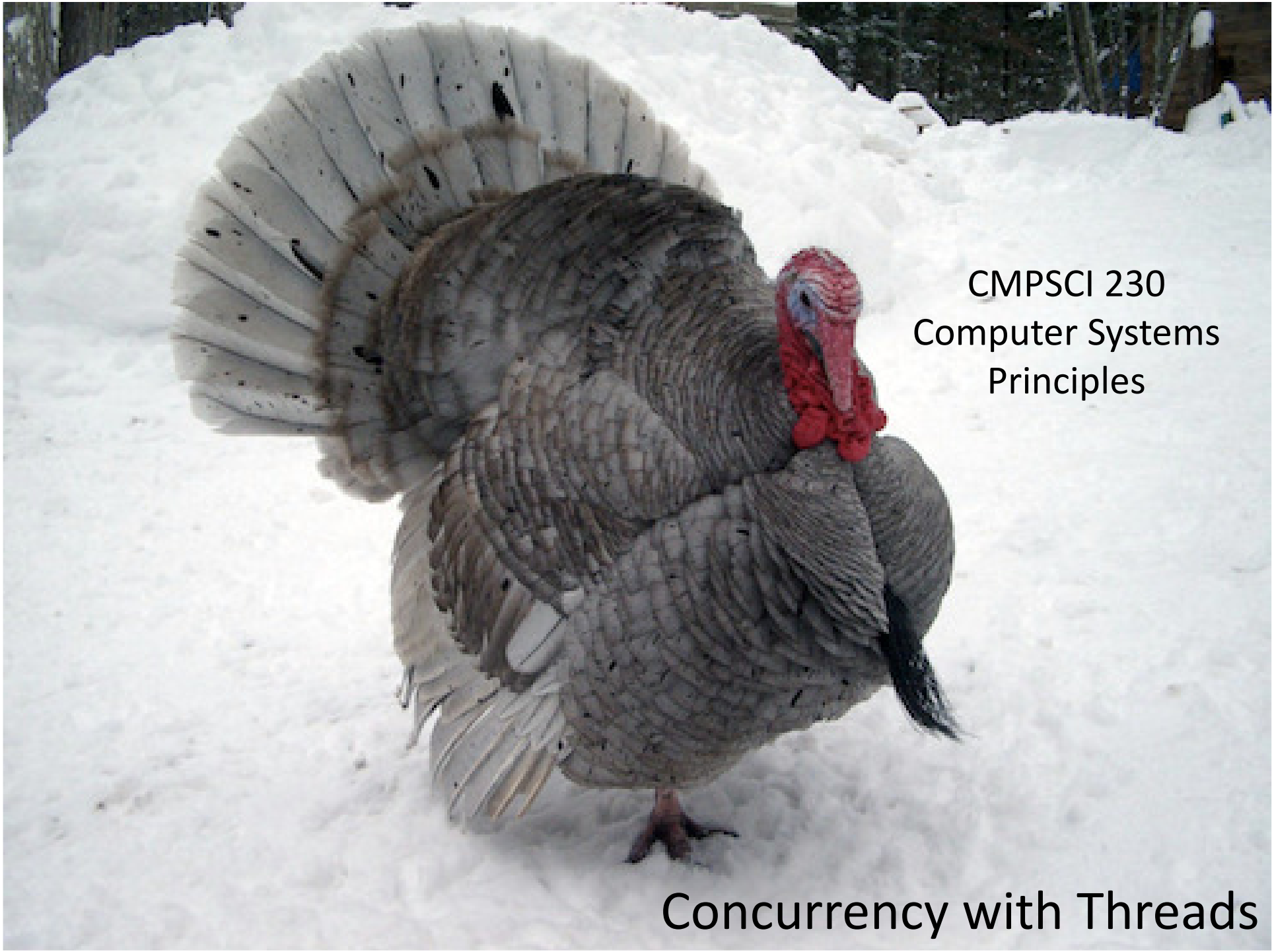

CMPSCI 230
Computer Systems
Principles
Concurrency with Threads

Objectives

- To learn what a thread is
- To understand the difference between between processes and threads
- To learn about programming with threads using the pthread library
- To learn about unintended sharing

iClicker question

After forking, a parent and child process:

A) Share virtual memory and file descriptors
B) Share virtual memory but not file descriptors
C) Share file descriptors but not virtual memory
D) Share neither file descriptors nor virtual memory

iClicker question: Solution

After forking, a parent and child process:

A) Share virtual memory and file descriptors

B) Share virtual memory but not file descriptors

C) Share file descriptors but not virtual memory

D) Share neither file descriptors nor virtual memory

Process Pros

- Clean Sharing Model
  - Global variables (address spaces) **are not shared**
  - File tables (file descriptors) **are shared**
- Simple and straightforward
  - Forking a child process is easy
  - Reaping child processes is simple
  - Processes are isolated for protection

Process Cons

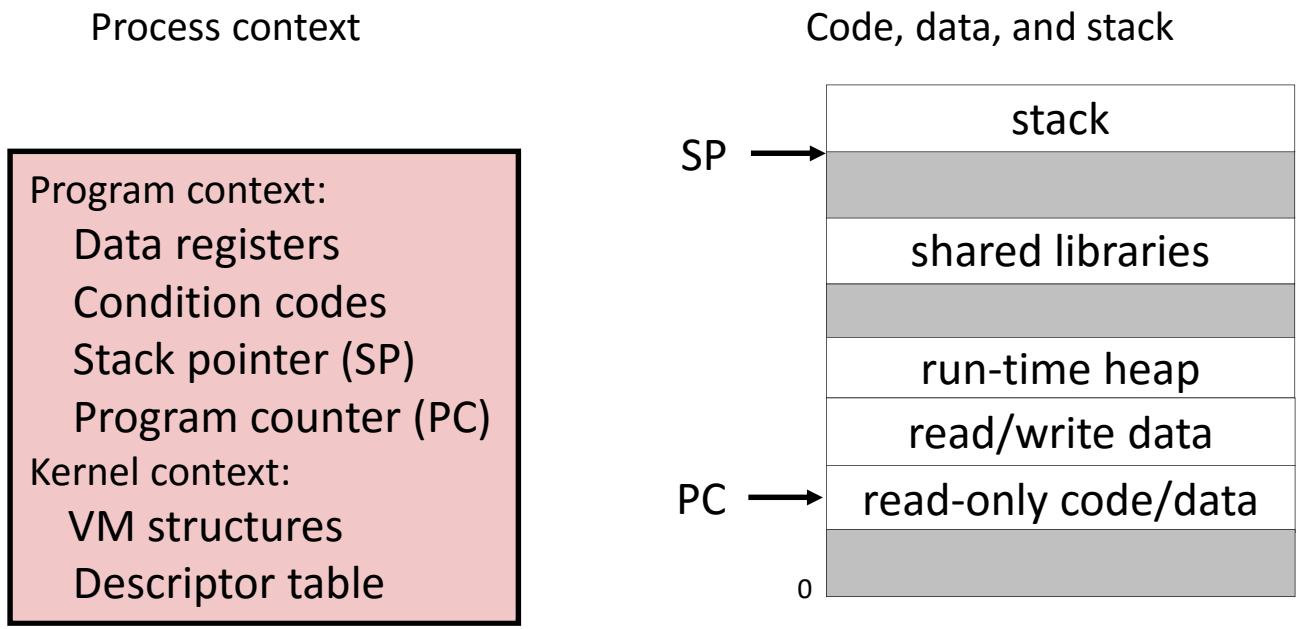
- Forking a process is expensive
  - Need to clone parent's memory space
  - Need to setup a separate context for child

Process Cons

- Forking a process is expensive
  - Need to clone parent's memory space
  - Need to setup a separate context for child
- Non-trivial to share data between processes
  - Need to use files/pipes to share data
  - Can use shared memory
  - Lots of overhead!

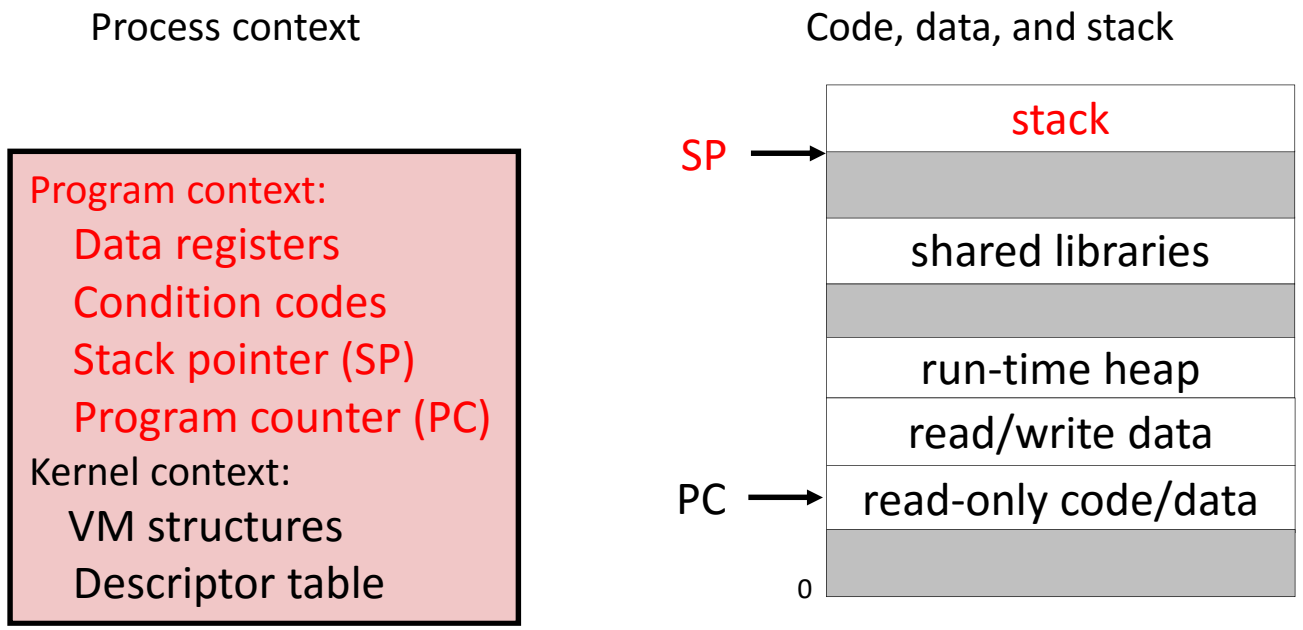
View of a Process

- Process = context + code, data, and stack



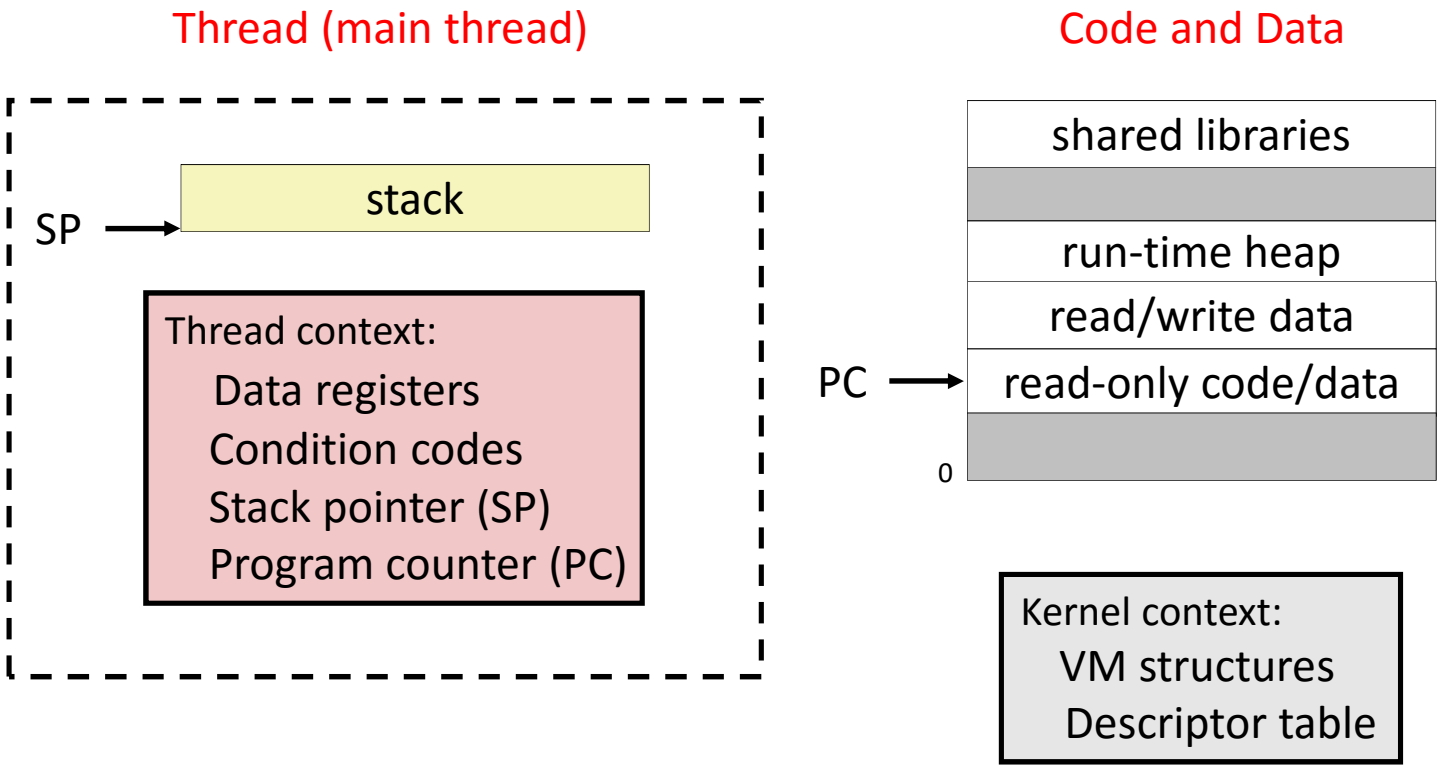
View of a Process

- Process = context + code, data, and stack



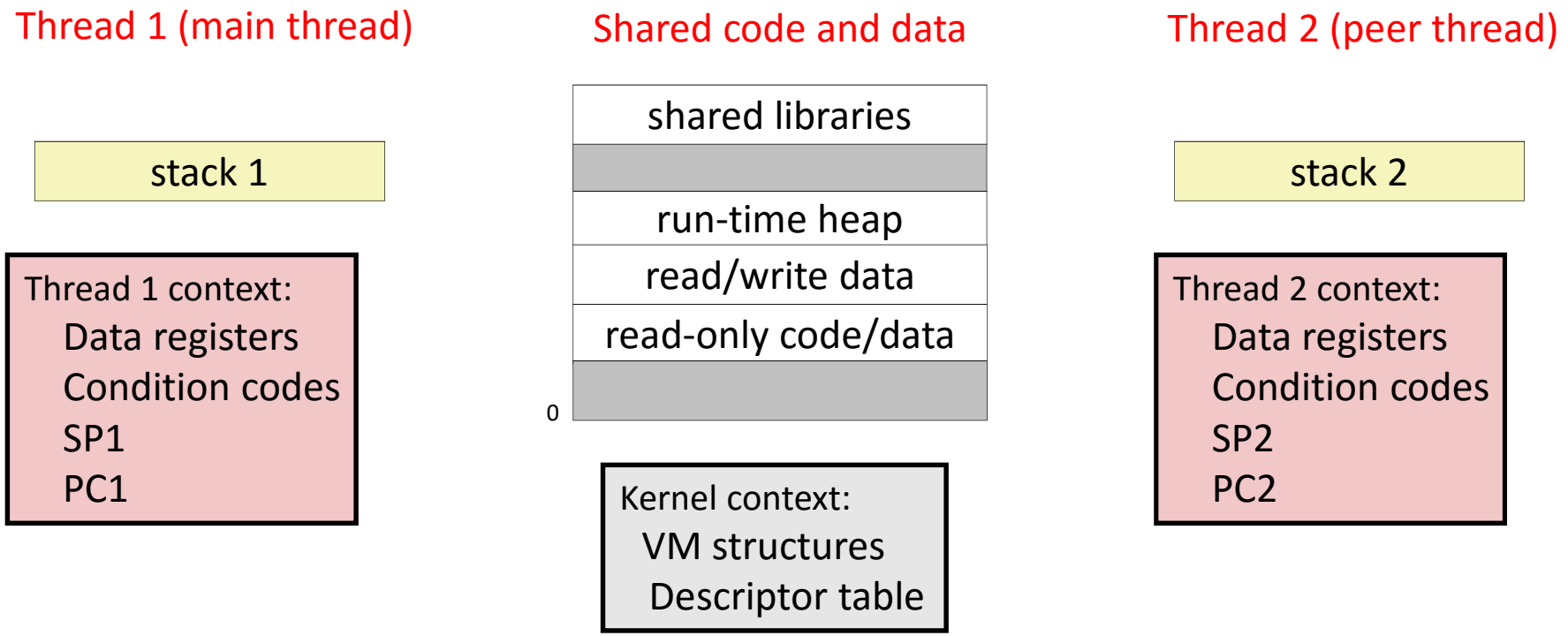
Alternate View of a Process

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



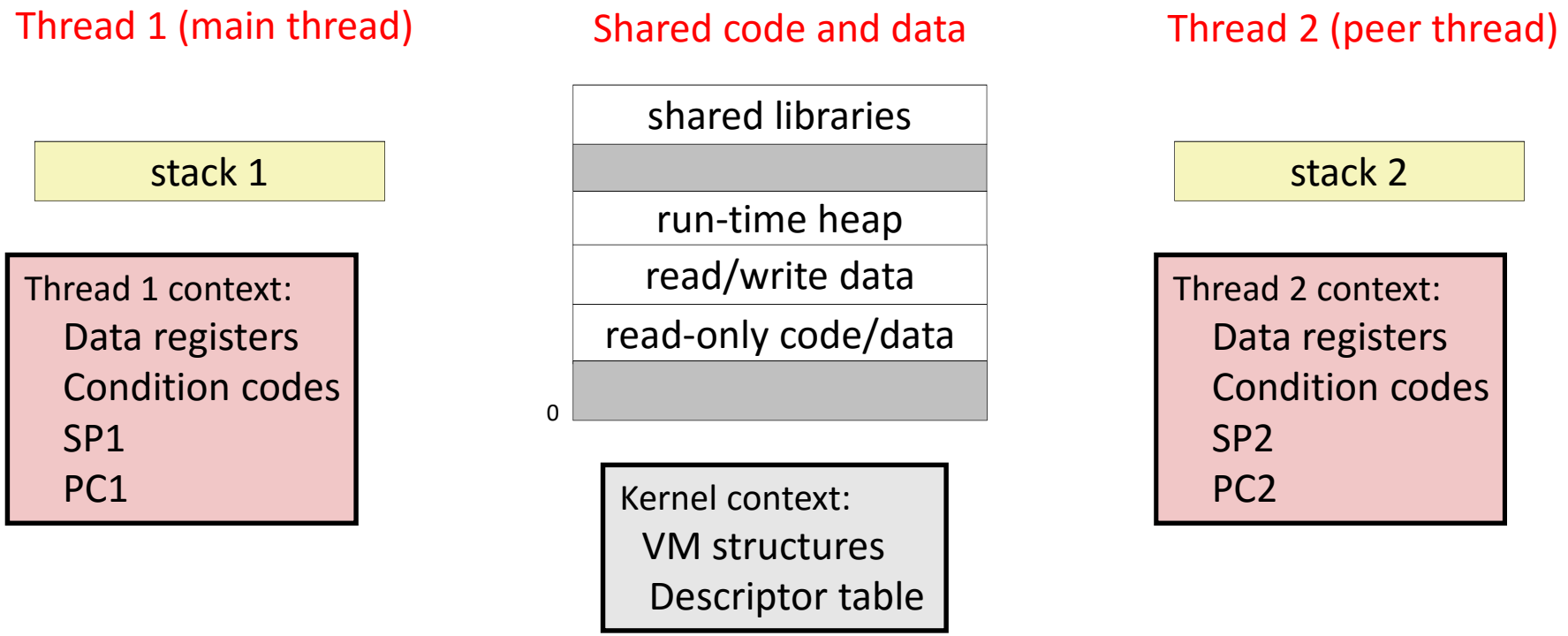
A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow



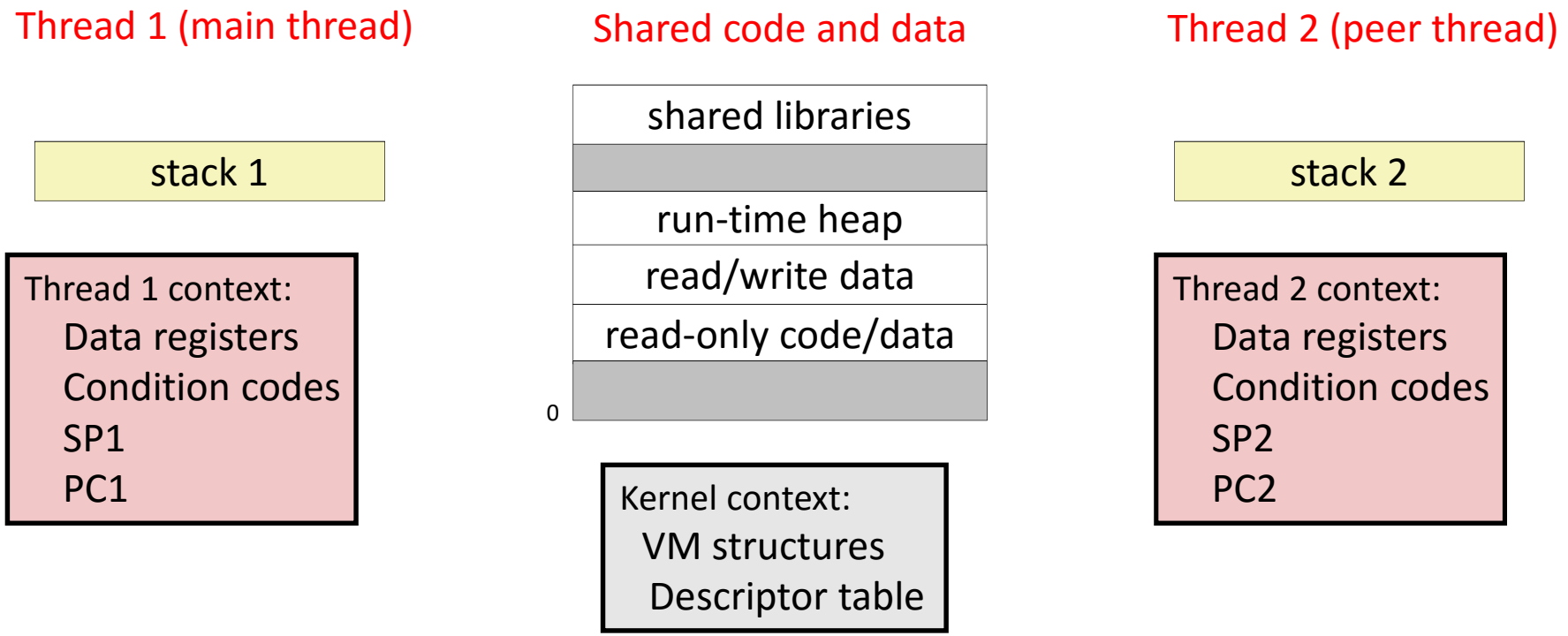
A Process With Multiple Threads

- 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



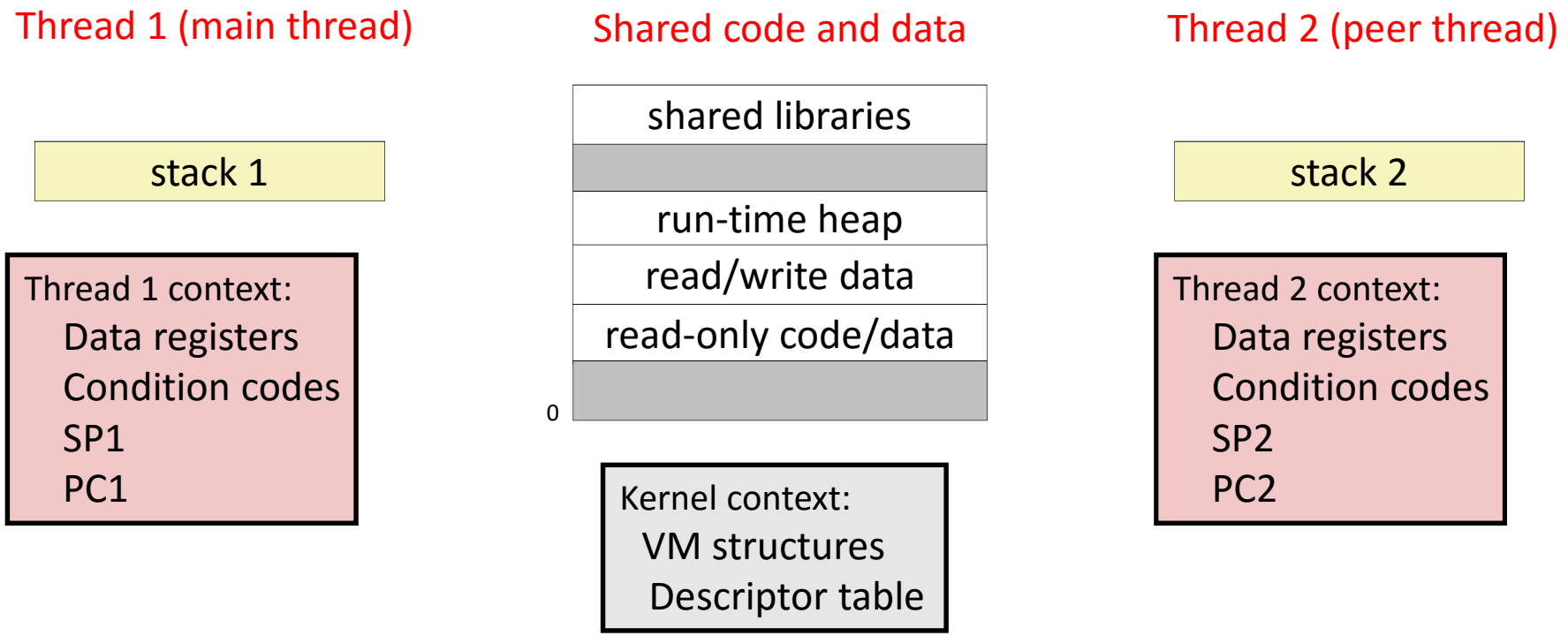
A Process With Multiple Threads

- 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



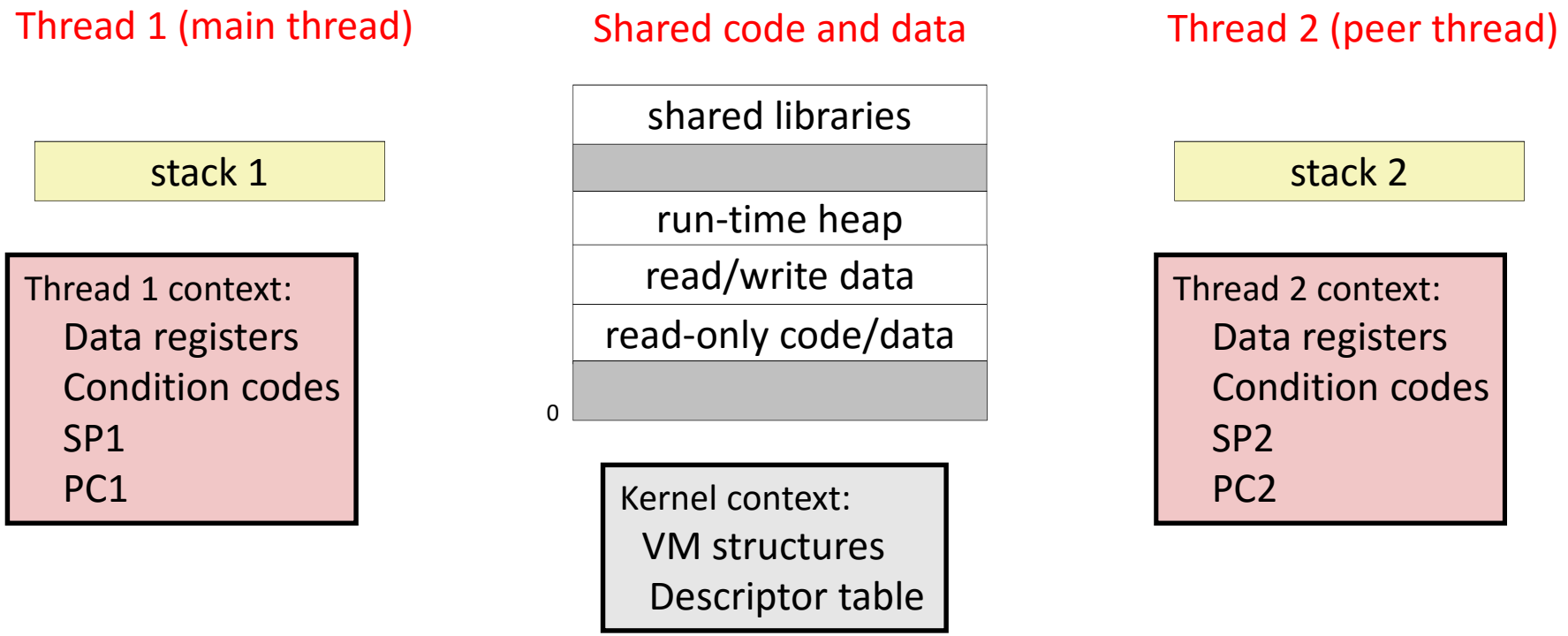
A Process With Multiple Threads

- 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
    - but not protected from other threads



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
    - but not protected from other threads
  - Each thread has its own thread id (TID)



OS: Concept of a Thread

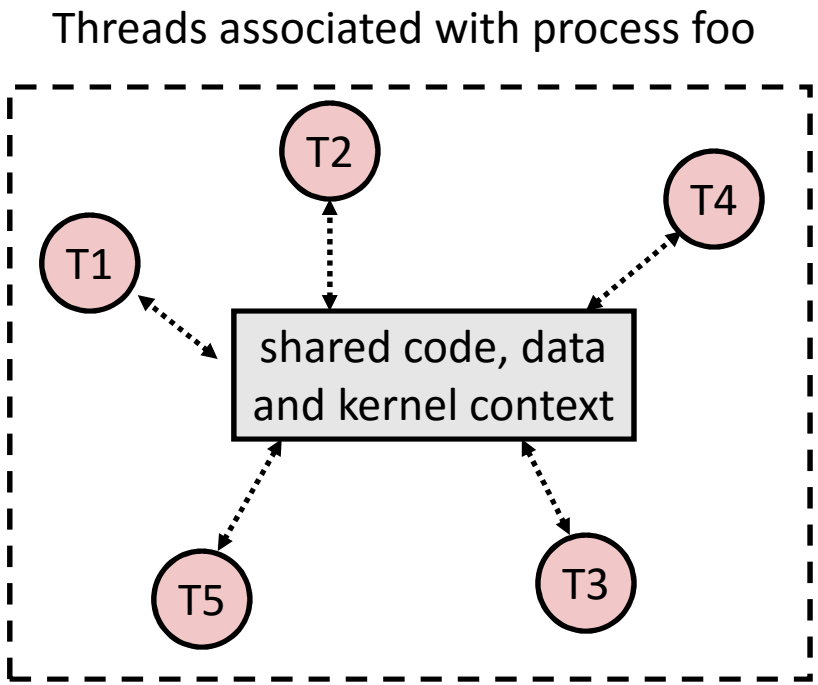
- **Processes**
  - Represent an entire program
- **Threads**
  - Represents *smaller chunks* of code

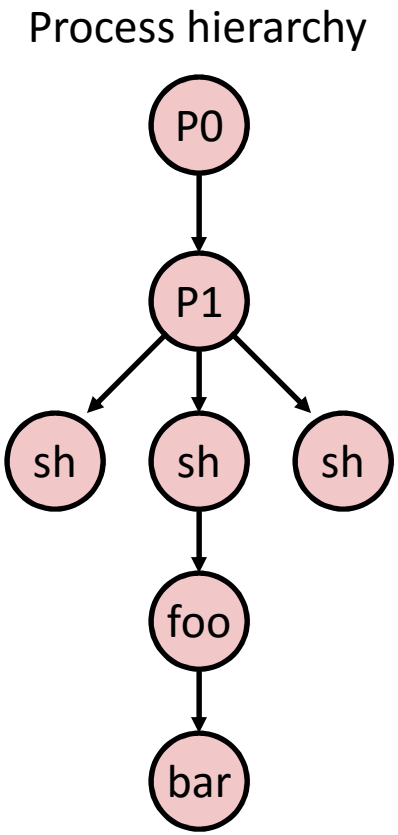
OS: Concept of a Thread

- **Processes**
  - Represent an entire program
  - High Overhead context switch
- **Threads**
  - Represents *smaller chunks* of code
  - Lower Overhead context switch

Logical View of Threads

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





Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched
- **How threads and processes are different**
  - Threads share code and some data
    - Processes (typically) do not

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched
- **How threads and processes are different**
  - Threads share code and some data
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control is 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

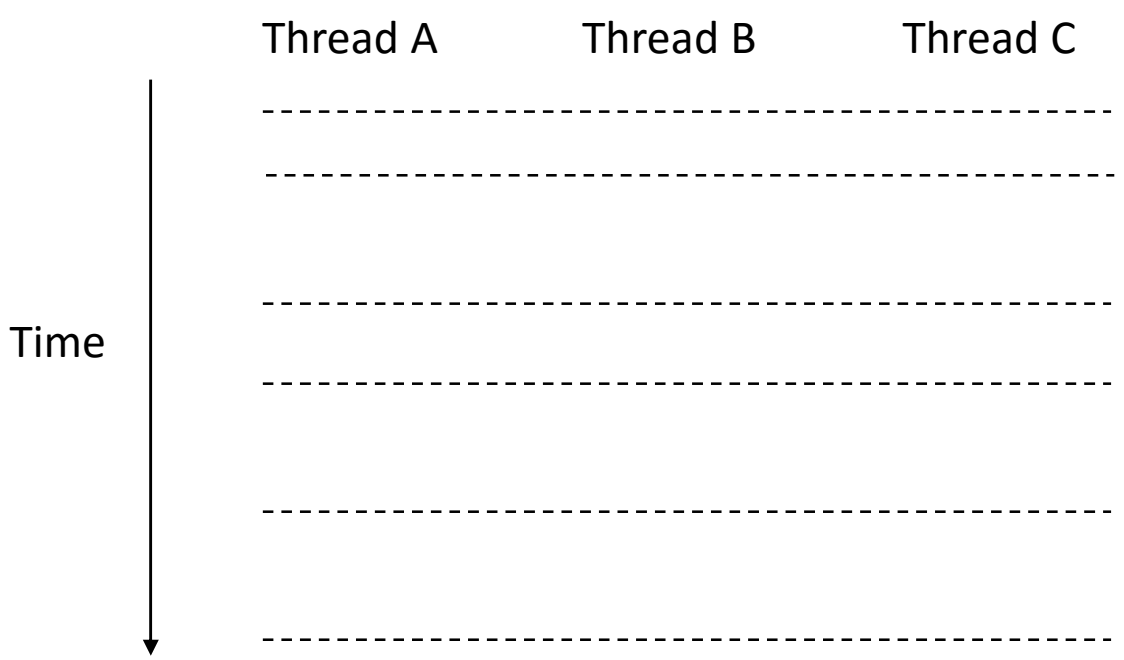
Concurrent Threads

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

Concurrent Threads

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

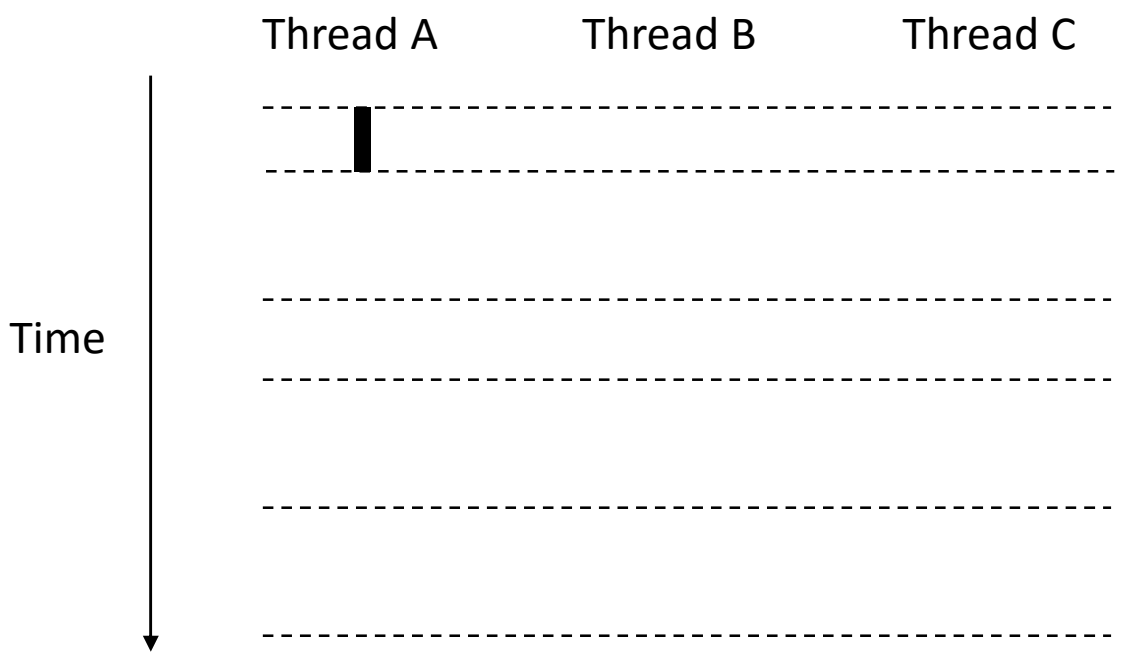
- Examples:



Concurrent Threads

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

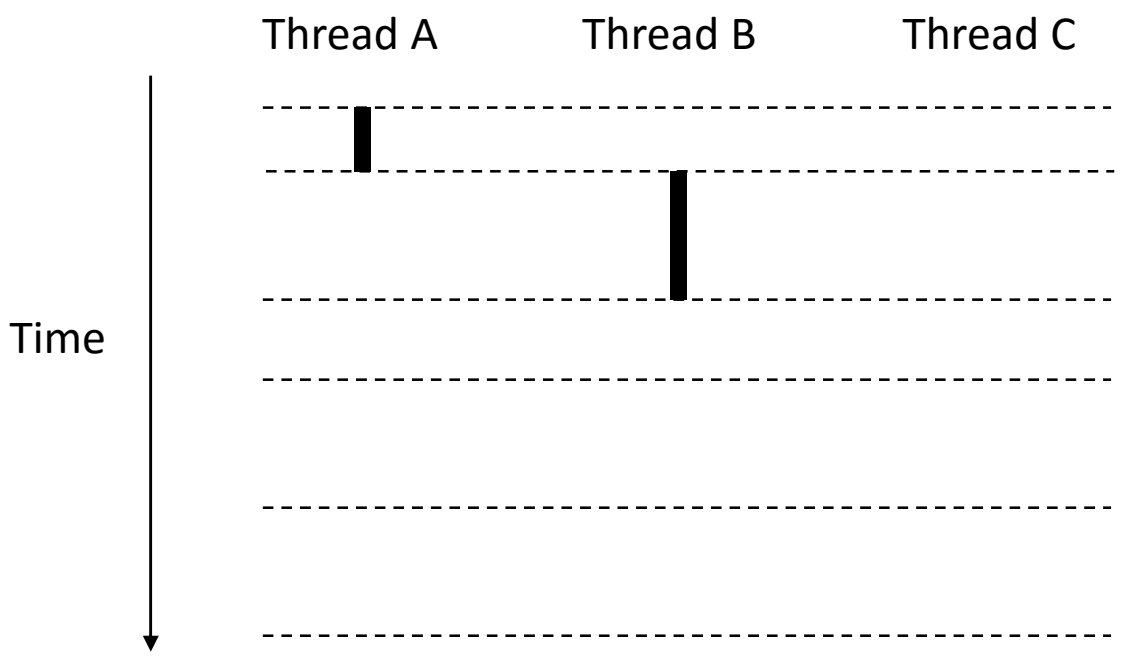
- Examples:



Concurrent Threads

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

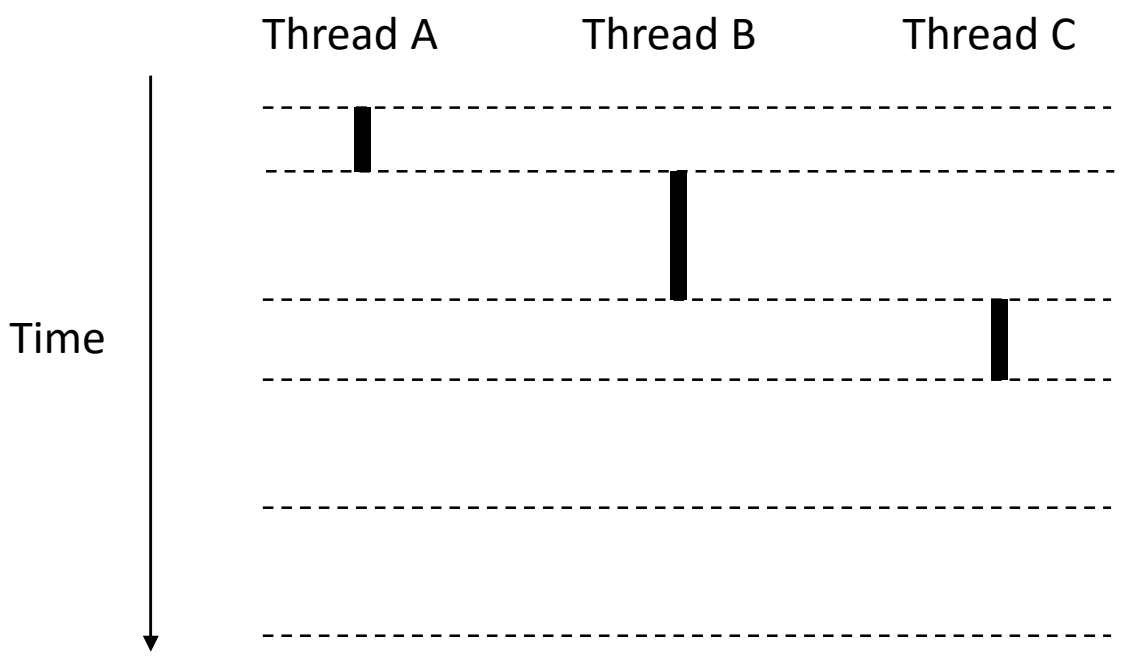
- Examples:



Concurrent Threads

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

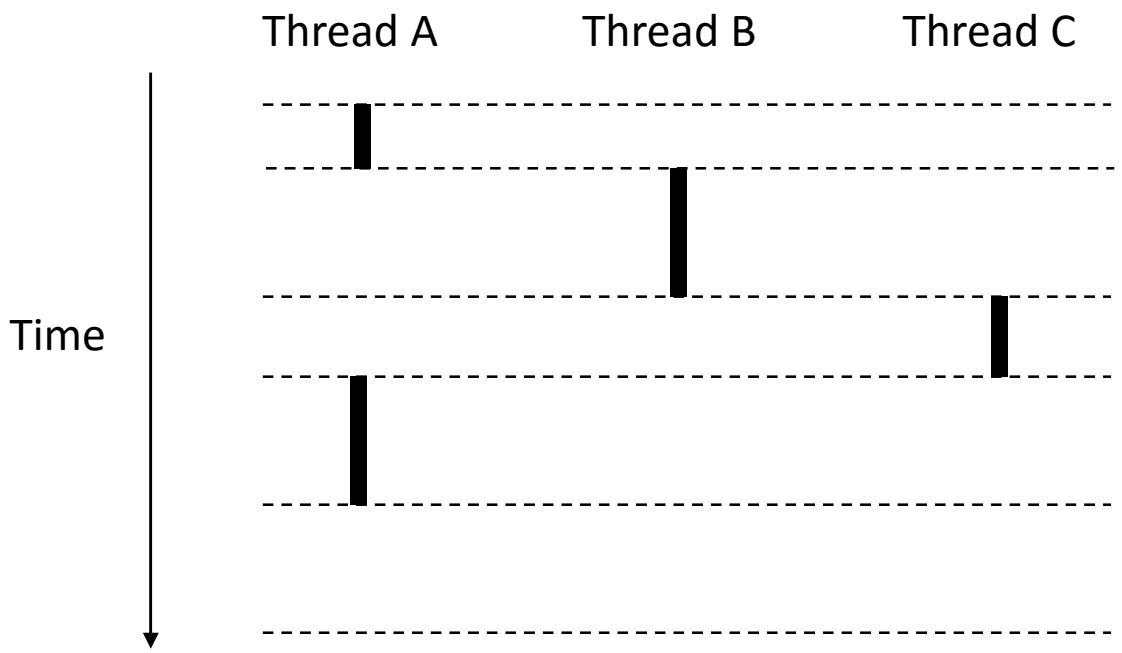
- Examples:



Concurrent Threads

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

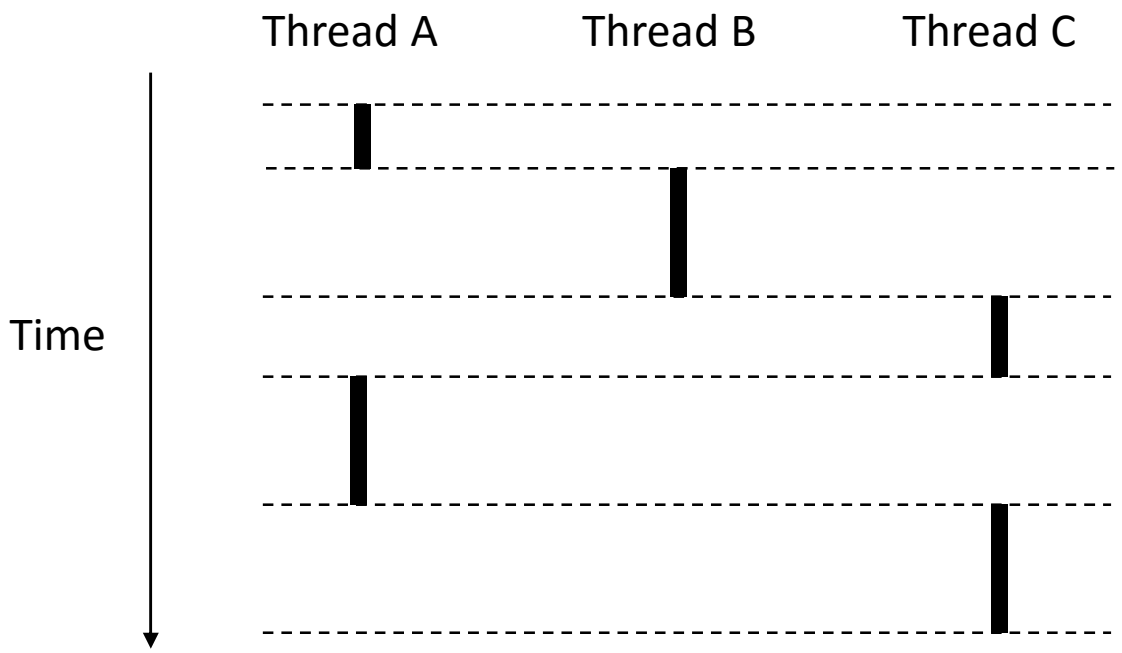
- Examples:



Concurrent Threads

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

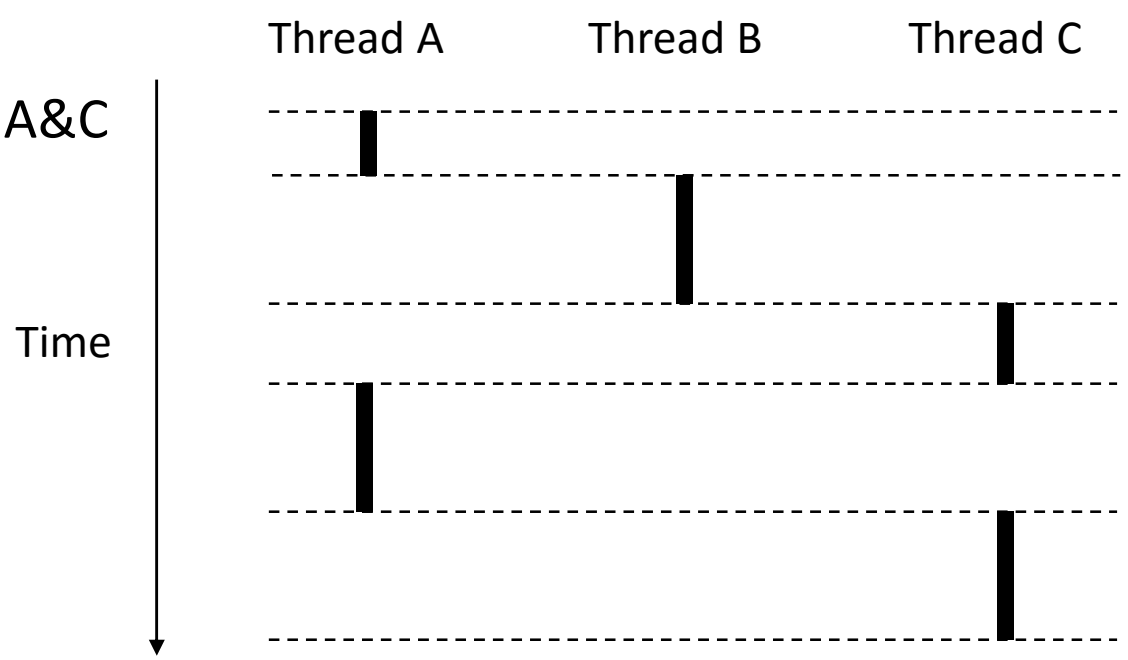
- Examples:



Concurrent Threads

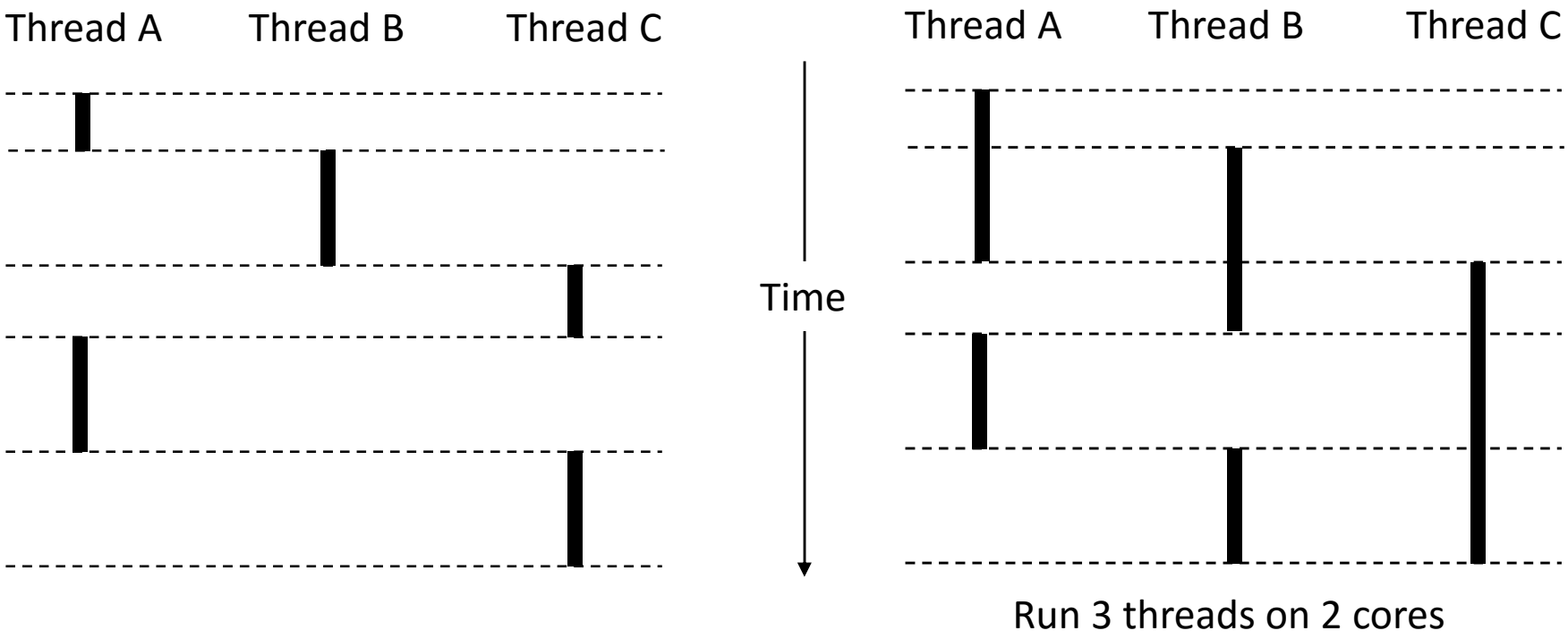
- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

- 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



Posix Threads (Pthreads) Interface

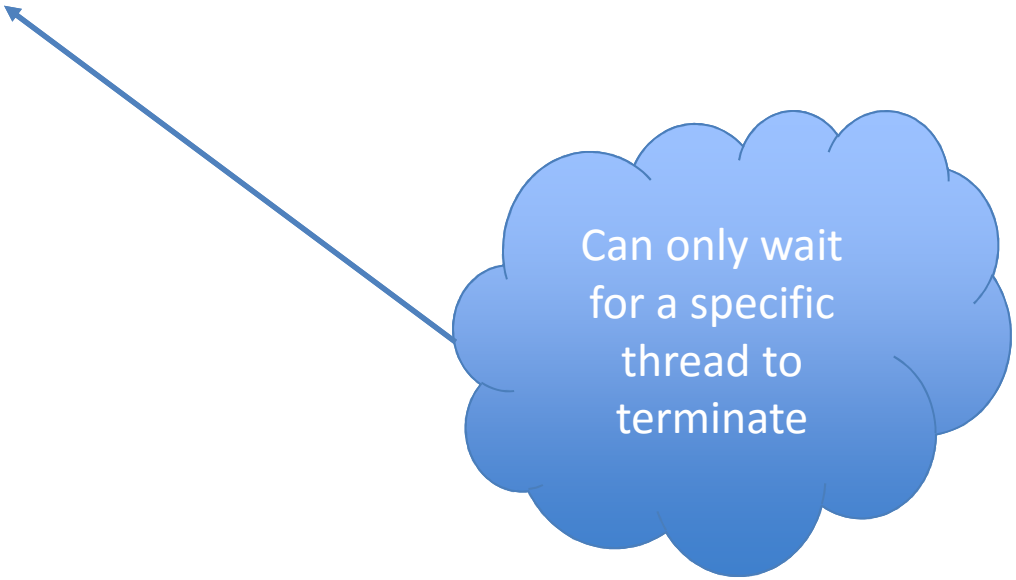
- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs

Posix Threads (Pthreads) Interface

- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`

Posix Threads (Pthreads) Interface

- *Pthreads:* Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`



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()`

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()`

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()` [terminates current threads]
    - `exit()` [terminates all threads]

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

The Pthreads "hello, world" Program

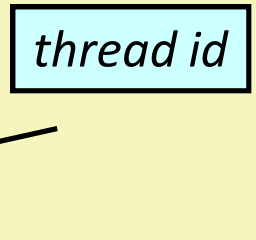
```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
```

thread routine that the new thread executes

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

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

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 attributes (usually NULL)

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

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 arguments (void *p)*

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

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);
}
```

return value
*(void **p)*

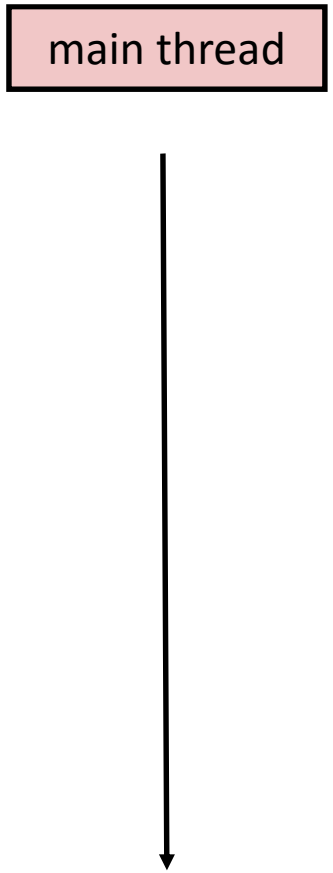
```
/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```

The Pthreads "hello, world" Program

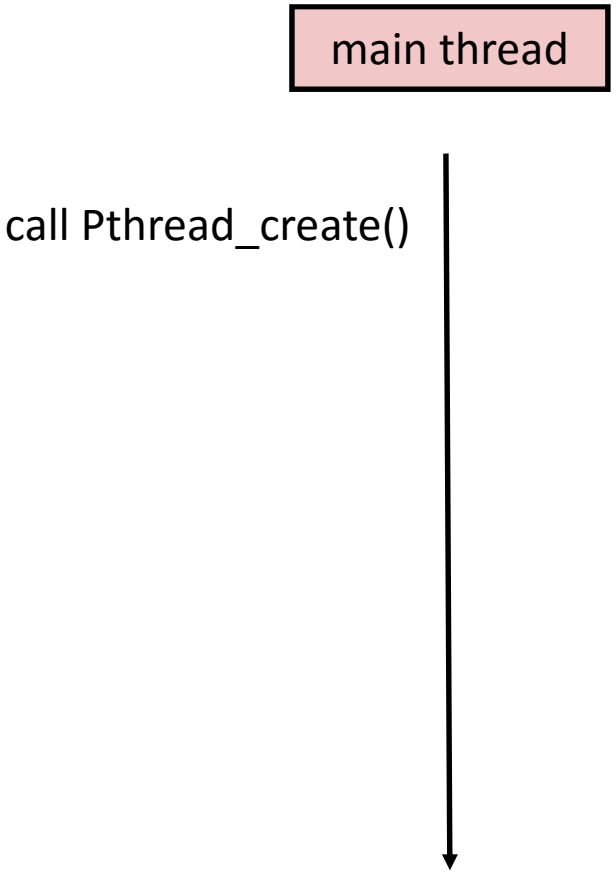
```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
```

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

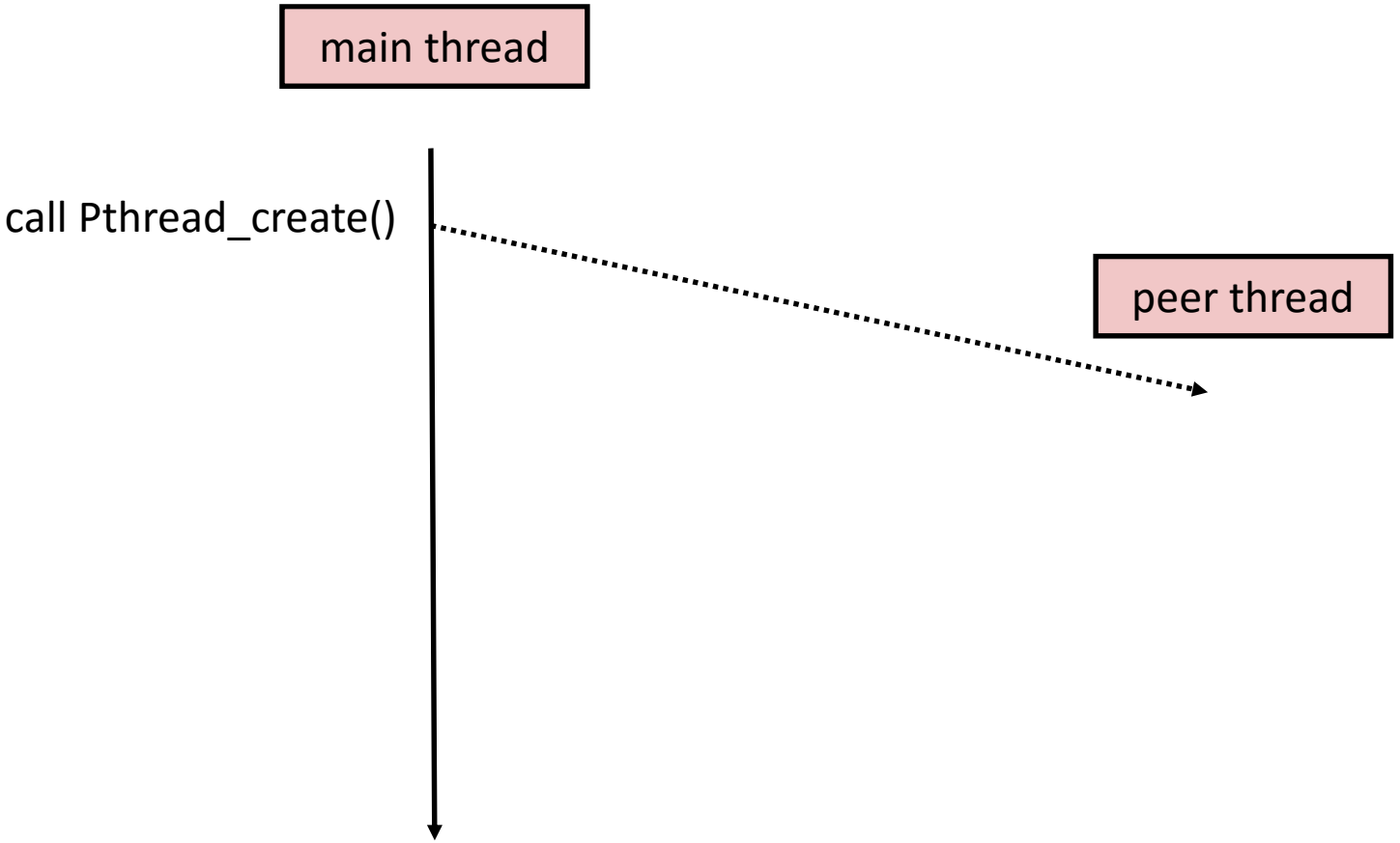
Execution of Threaded“hello, world”



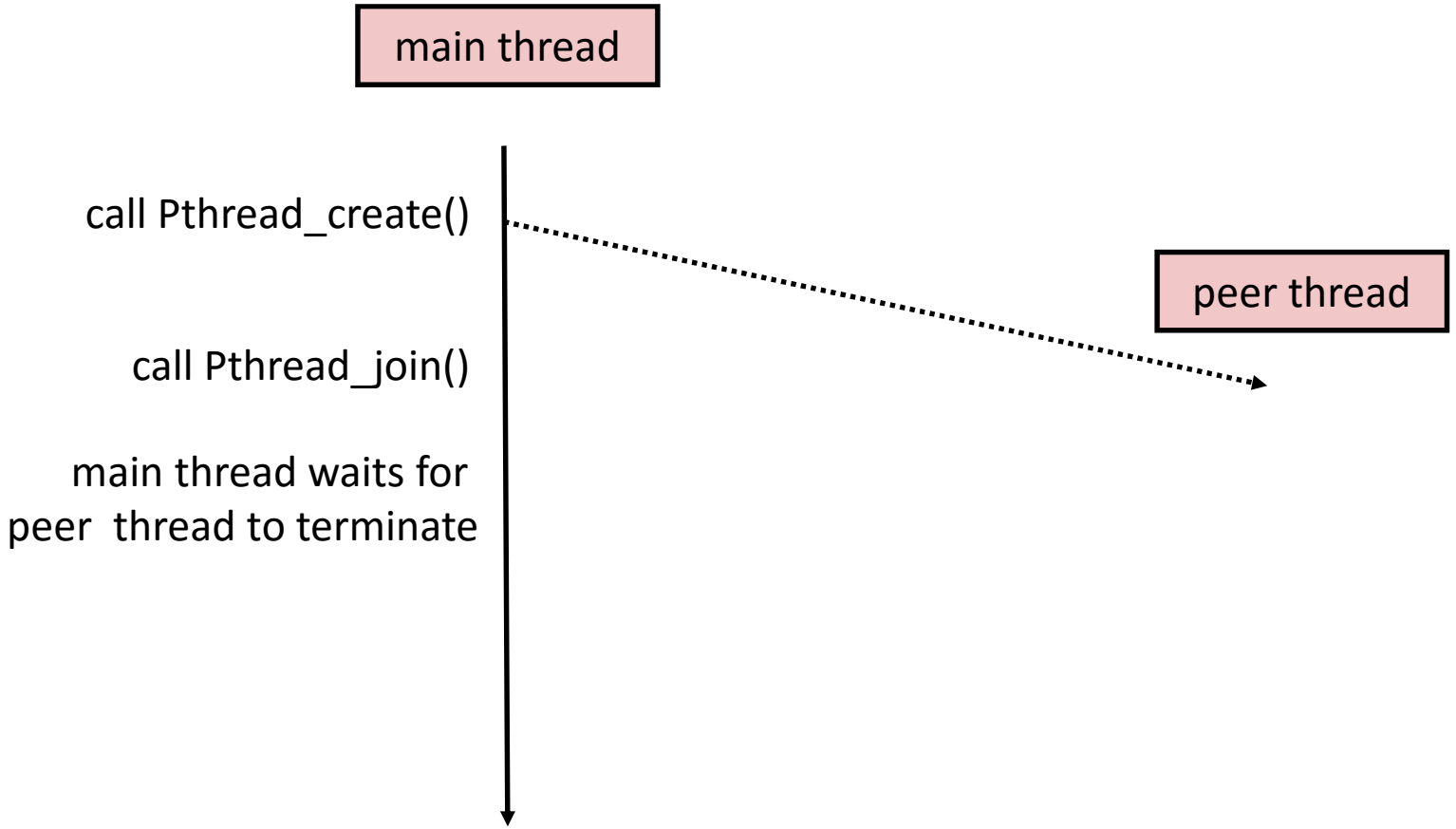
Execution of Threaded“hello, world”



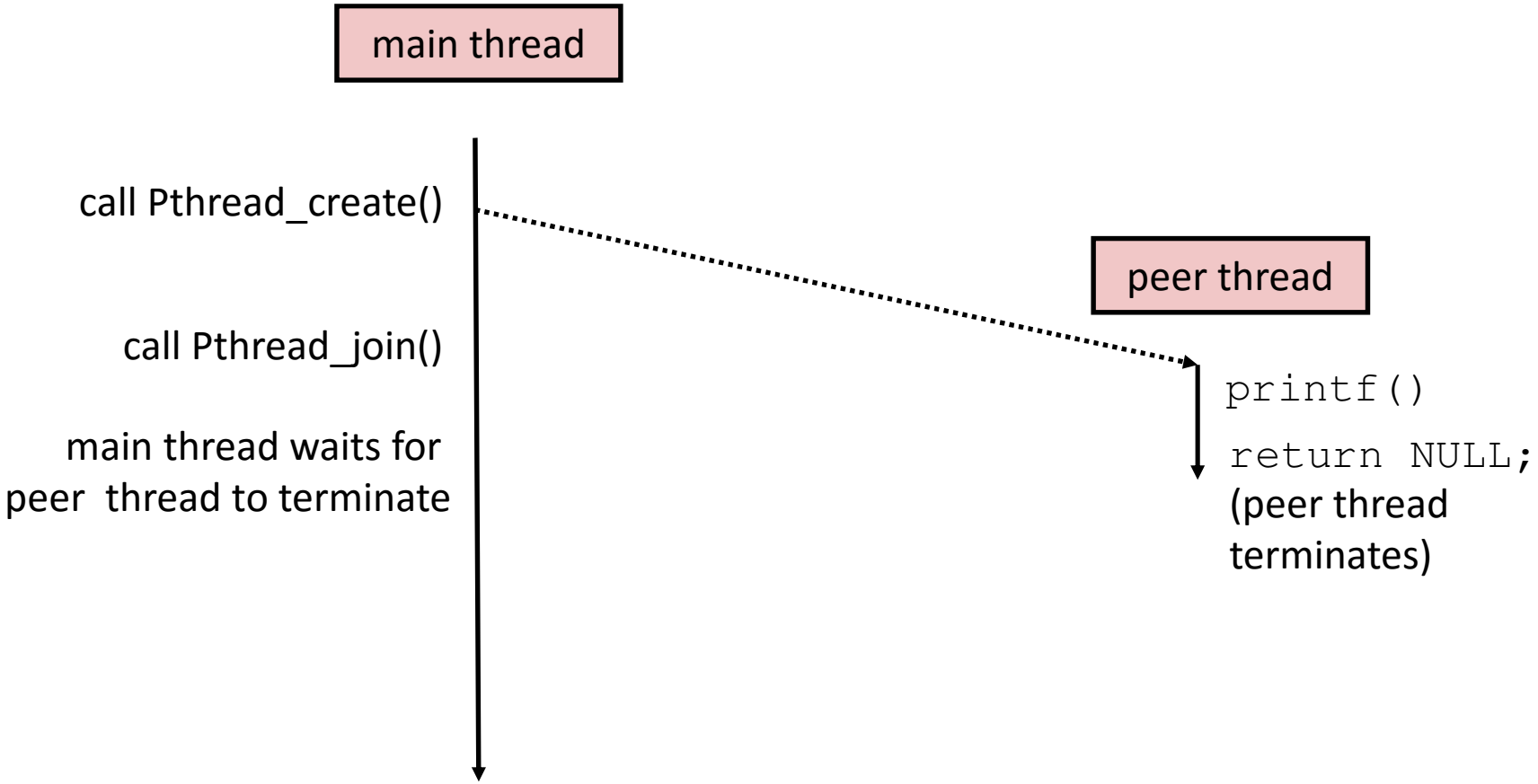
Execution of Threaded“hello, world”



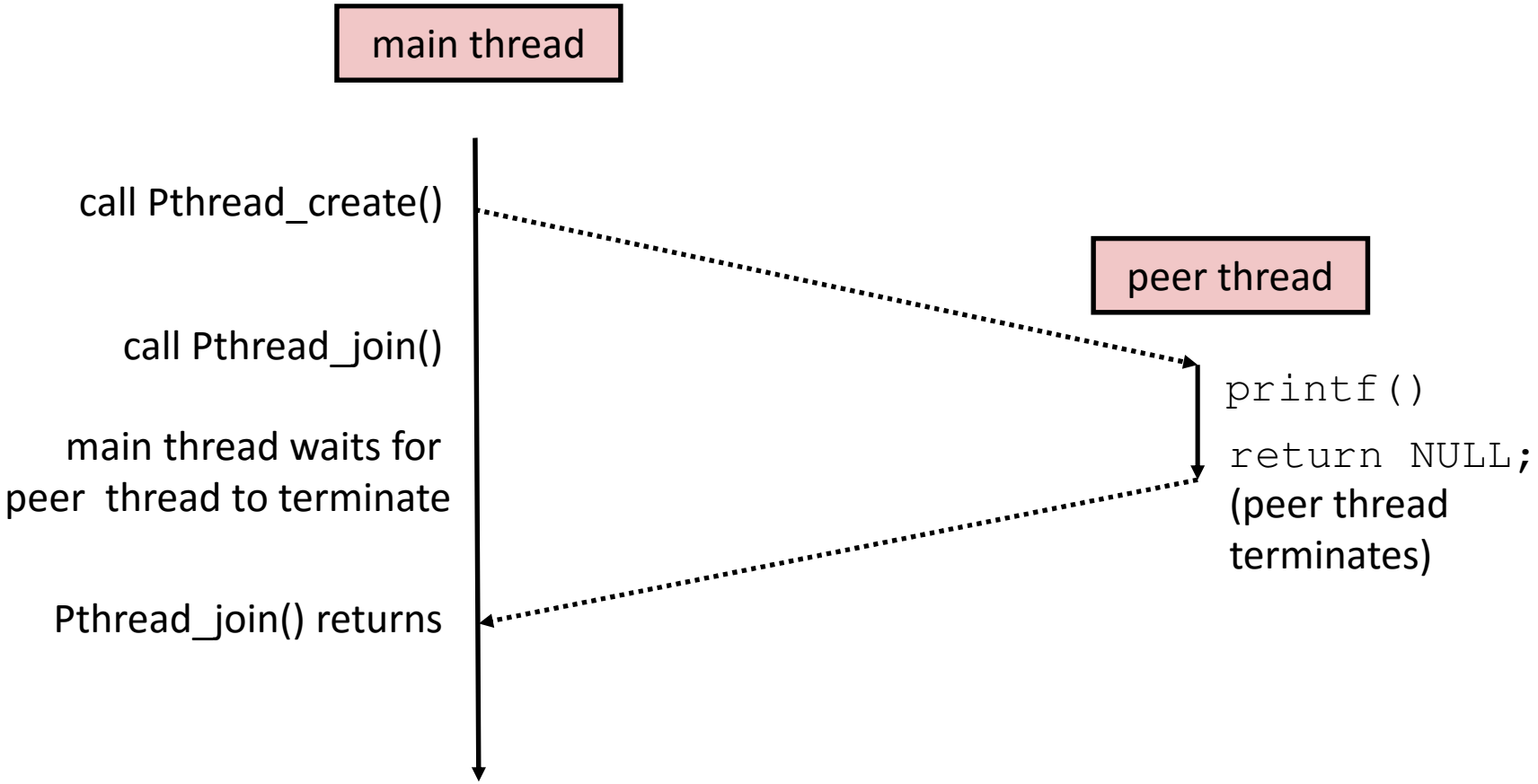
Execution of Threaded“hello, world”



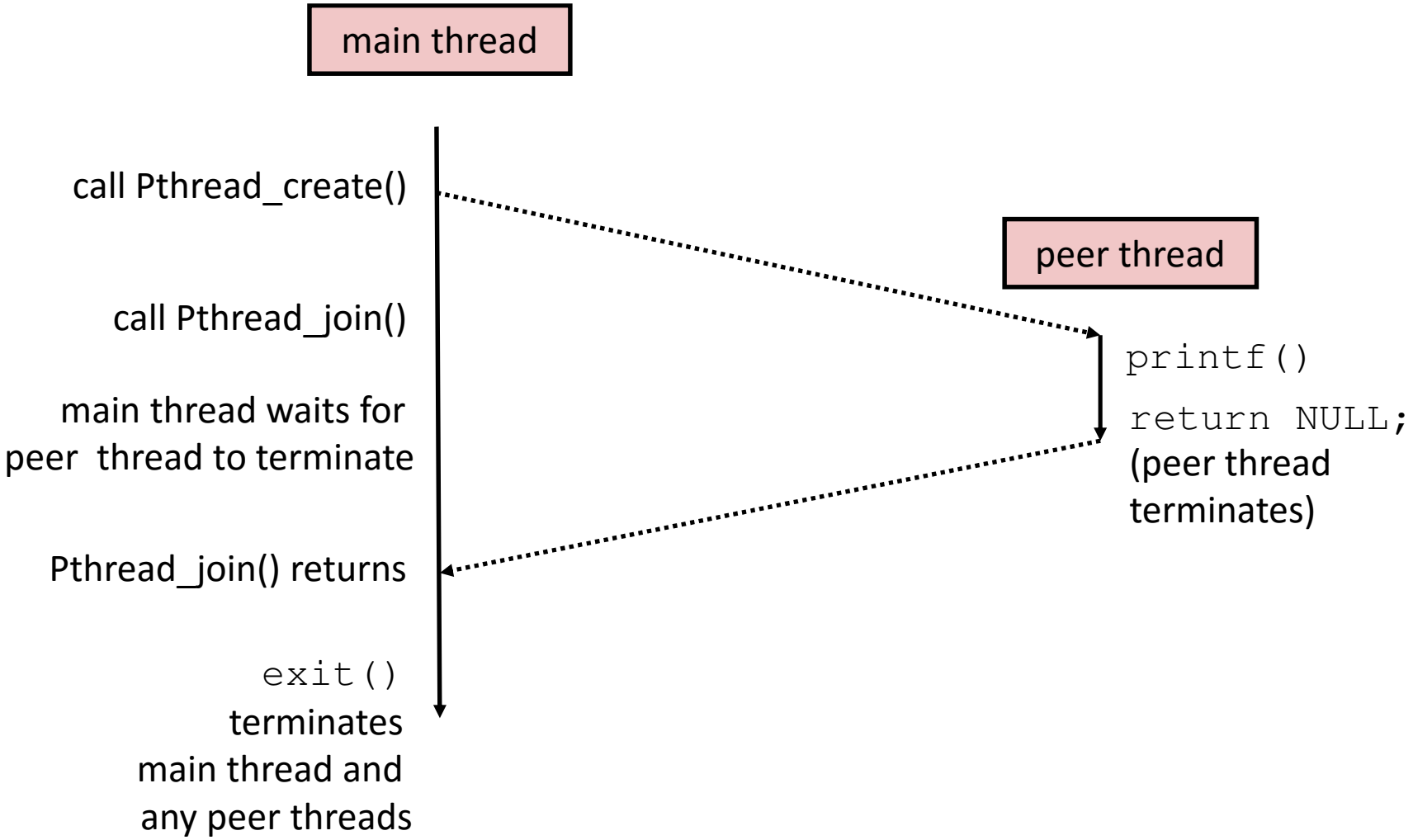
Execution of Threaded“hello, world”



Execution of Threaded“hello, world”



Execution of Threaded“hello, world”



Passing argument

```
int main()
{
    int i;
    pthread_t tid;

    for (i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void *)i);
    pthread_exit(NULL);
}
```

```
void *thread(void *vargp)
{
    int myid = (int)vargp;
    printf("hello from [%d]\n", myid);
    return NULL;
}
```

Passing argument

```
int main()
{
    int i;
    pthread_t tid;

    for (i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void *)i);
    pthread_exit(NULL);
}
```

```
void *thread(void *vargp)
{
    int myid = (int)vargp;
    printf("hello from [%d]\n", myid);
    return NULL;
}
```

Passing pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;

  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int) tid);
  pthread_join(tid, NULL);
  printf("Main thread joined with peer thread.\n");

  return 0;
}
```

```
void *thread (void *vargp) {
  int arg = *((int*) vargp);
  printf("hello from thread %X!\n", arg);
  return NULL;
}
```

Passing pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;

  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int) tid);
  pthread_join(tid, NULL);
  printf("Main thread joined with peer thread.\n");

  return 0;
}
```

```
void *thread (void *vargp) {
  int arg = *((int*) vargp);
  printf("hello from thread %X!\n", arg);
  return NULL;
}
```

Passing pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;

  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int) tid);
  pthread_join(tid, NULL);
  printf("Main thread joined with peer thread.\n");

  return 0;
}
```

```
void *thread (void *vargp) {
  int arg = *((int*) vargp);
  printf("hello from thread %X!\n", arg);
  return NULL;
}
```

Returning value

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int ret_value;

  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
ret_value);

  return 0;
}
```

```
void *thread (void *vargp) {
  return (int *) 0xDEADBEEF;
}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int)(tid));
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
*ret_value);
  free(ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int)(tid));
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
*ret_value);
  free(ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int)(tid));
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
*ret_value);
  free(ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int)(tid));
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
*ret_value);
  free(ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int)(tid));
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
*ret_value);
  free(ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, &thread_arg);
  printf("Main thread waiting for peer thread %u to
complete...\n", (unsigned int)(tid));
  pthread_join(tid, (void **)(&ret_value));
  printf("Main thread joined with peer thread: %X.\n",
*ret_value);
  free(ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

Returning pointer

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  printf("Main thread creating peer
  pthread_create(&tid, NULL, threa
  printf("Main thread waiting fo
complete...\n", (unsigned int)(
  pthread_join(tid, (void **)(&r
  printf("Main thread joined wit
*ret_value);
  free(ret_value);
  return 0; }
```


```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  printf("hello from thread: %X!\n", arg);
  int *ret = (int*)malloc(sizeof(int));
  *ret = arg;
  return ret;}
```

i-clicker: What is the output?

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  pthread_create(&tid, NULL, thread, &thread_arg);
  pthread_join(tid, (void **)(&ret_value));
  printf("%X\n", *ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  return &arg;}
```

A. 0xDEADBEEF
B. Segfault
C. 0
D. -1

i-clicker solution: What is the output?

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int thread_arg = 0xDEADBEEF;
  int* ret_value;
  pthread_create(&tid, NULL, thread, &thread_arg);
  pthread_join(tid, (void **)(&ret_value));
  printf("%X\n", *ret_value);
  return 0; }
```

```
void *thread (void *vargp) {
  int arg = *((int*)vargp);
  return &arg;}
```

A. 0xDEADBEEF
B. Segfault
C. 0
D. -1

Detached mode

```
int main (int argc, char *argv[]) {
  // This is used to record the thread id:
  pthread_t tid;

  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, NULL);

  pthread_exit(NULL);
  return 0;
}
```

```
void *thread (void *vargp) {
  pthread_detach(pthread_self());
  printf("hello from thread world!\n");
}
```

Detached mode

```
int main (int argc, char *argv[]) {
  // This is used to record the thread id:
  pthread_t tid;

  printf("Main thread creating peer thread.\n");
  pthread_create(&tid, NULL, thread, NULL);

  pthread_exit(NULL);
  return 0;
}
```

```
void *thread (void *vargp) {
  pthread_detach(pthread_self());
  printf("hello from thread world!\n");
}
```

Detached mode

```
void *thread (void *vargp) {
  pthread_detach(pthread_self());
  printf("hello from thread world!\n");
}
```

- Run thread in “detached” mode.
  - Runs independently of other threads
  - Reaped automatically (by kernel) when it terminates

Concurrent Programming is Nice!

- Easy to share global data structure
- Easy to implement a cache that all the threads can use

Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading

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 frequently impossible

Concurrent Programming is Hard!

Concurrent Programming is Hard!

- Classical problem classes of concurrent programs:
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?

Concurrent Programming is Hard!

- Classical problem classes of concurrent programs:
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock

Concurrent Programming is Hard!

- Classical problem classes of concurrent programs:
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line

Unintended Sharing

- Threads share the heap
- Threads have separate stacks
- But, C has pointers
  - which can point to anywhere, even the stack!
- What if…
  - I pass a pointer to a peer thread that points to a location on the stack
  - Unintended Sharing!

Unintended sharing

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  for(int i=0; i<10; i++) {
    pthread_create(&tid, NULL, thread, &i);
  }
  pthread_exit(NULL);
  return 0;
}
```

```
void *thread (void *vargp) {
  pthread_detach(pthread_self());
  int arg = *((int*) vargp);
  printf("hello from thread %d!\n", arg);
  return NULL;
}
```

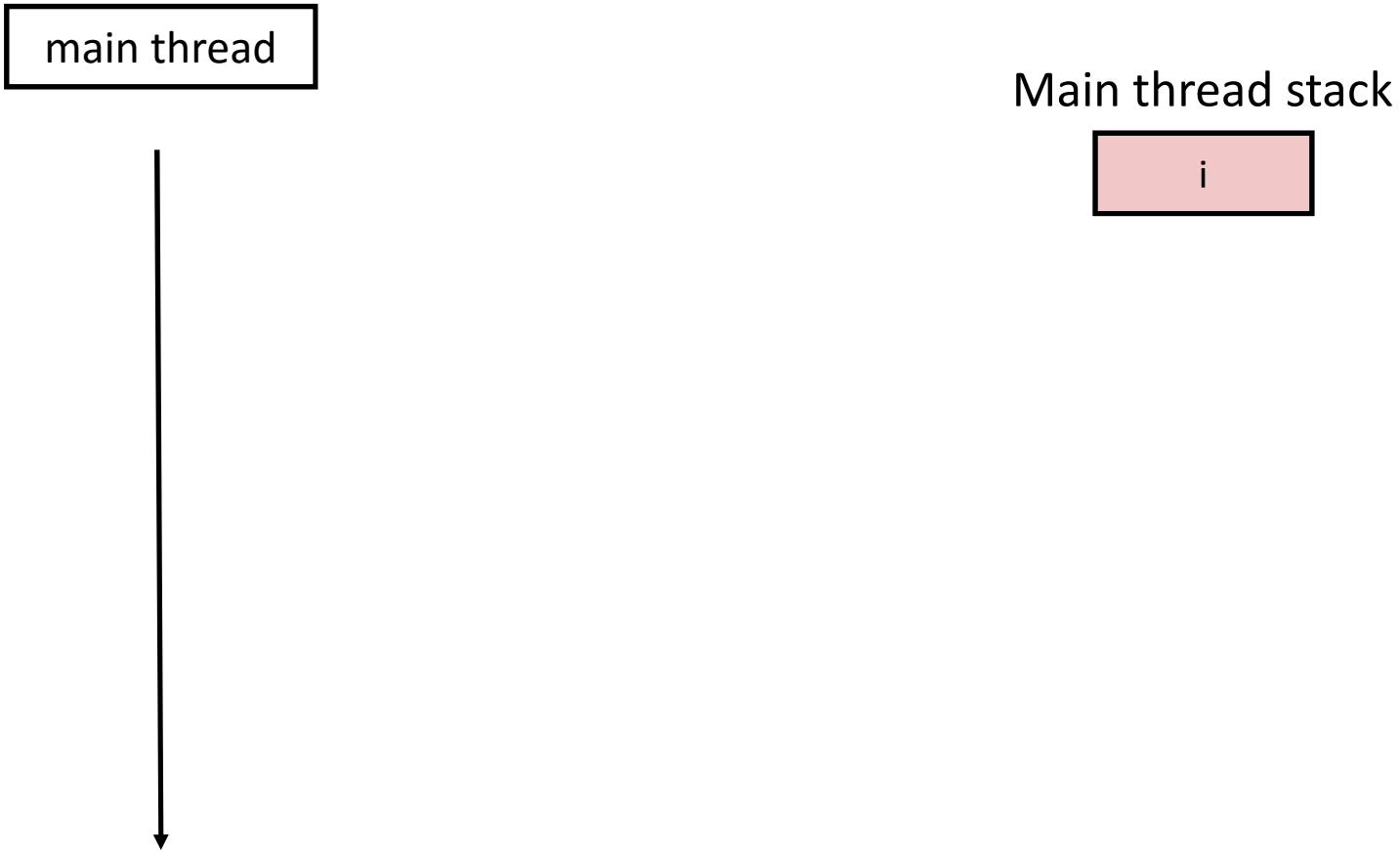
Unintended sharing

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  for(int i=0; i<10; i++) {
    pthread_create(&tid, NULL, thread, &i);
  }
  pthread_exit(NULL);
  return 0;
}
```

```
void *thread (void *vargp) {
  pthread_detach(pthread_self());
  int arg = *((int*) vargp);
  printf("hello from thread %d!\n", arg);
  return NULL;
}
```

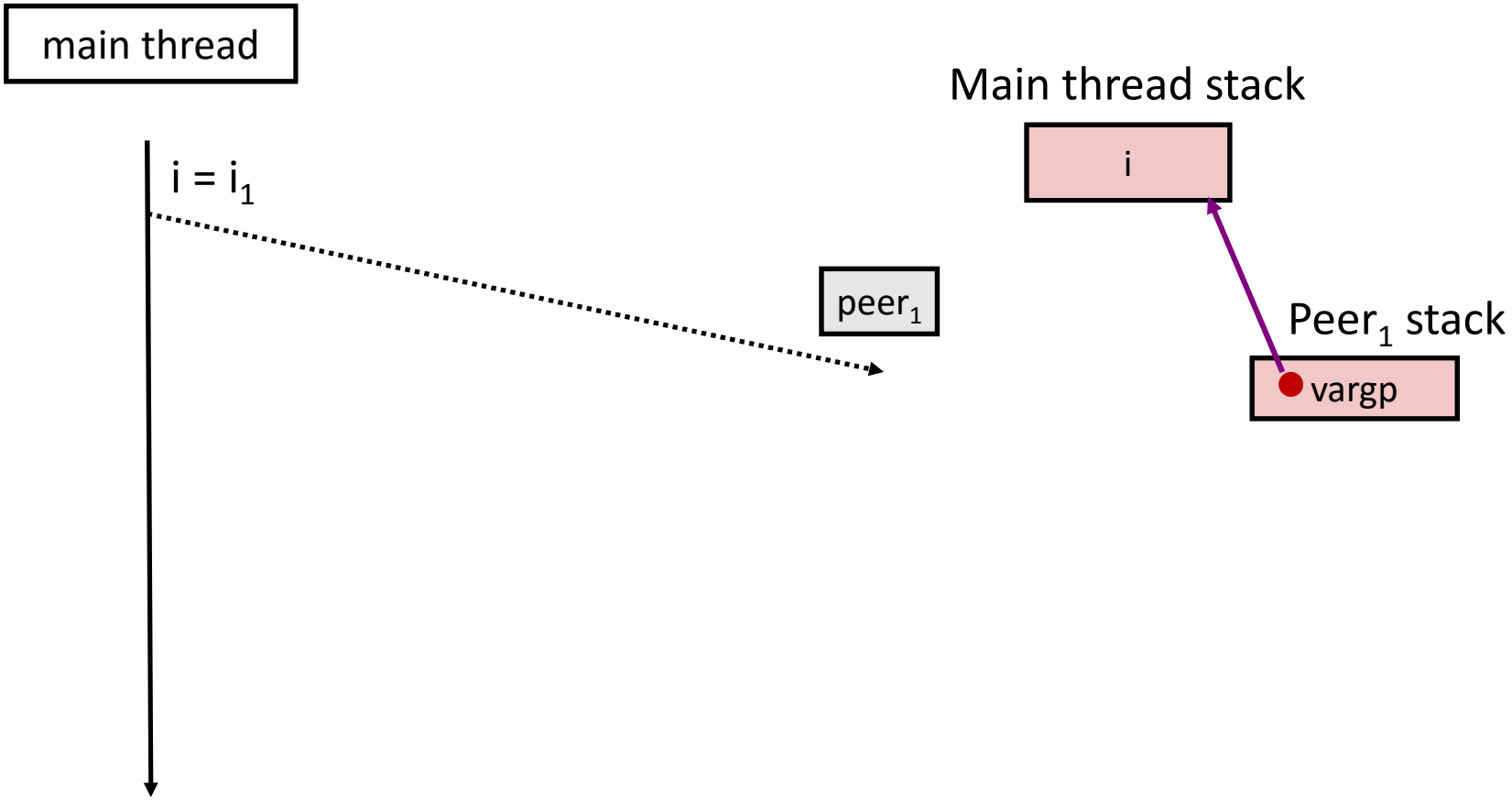
Potential Form of Unintended Sharing

```
for(int i=0; i<10; i++) {
  pthread_create(&tid, NULL, thread, &i);
}
```



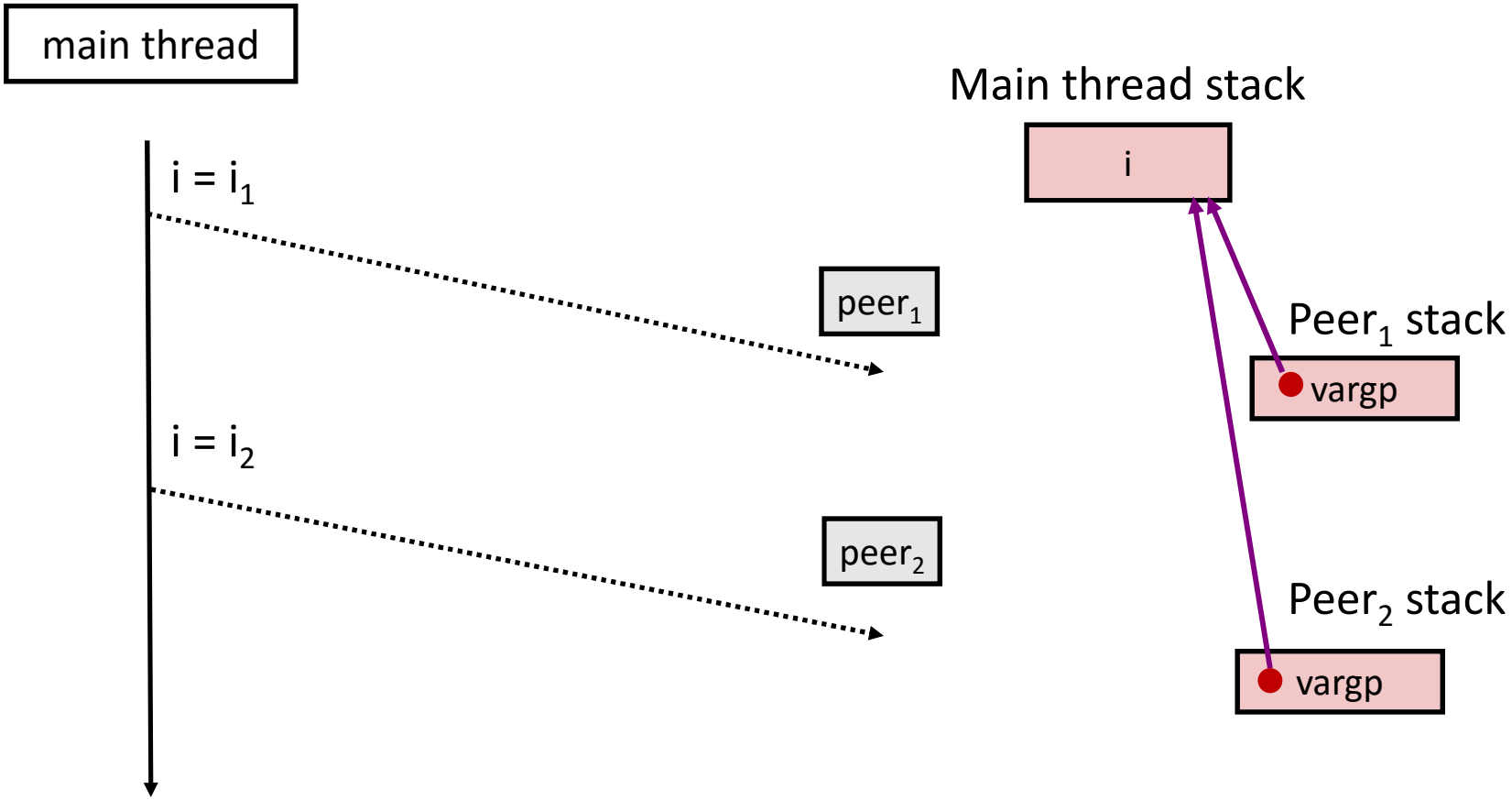
Potential Form of Unintended Sharing

```
for(int i=0; i<10; i++) {
  pthread_create(&tid, NULL, thread, &i);
}
```



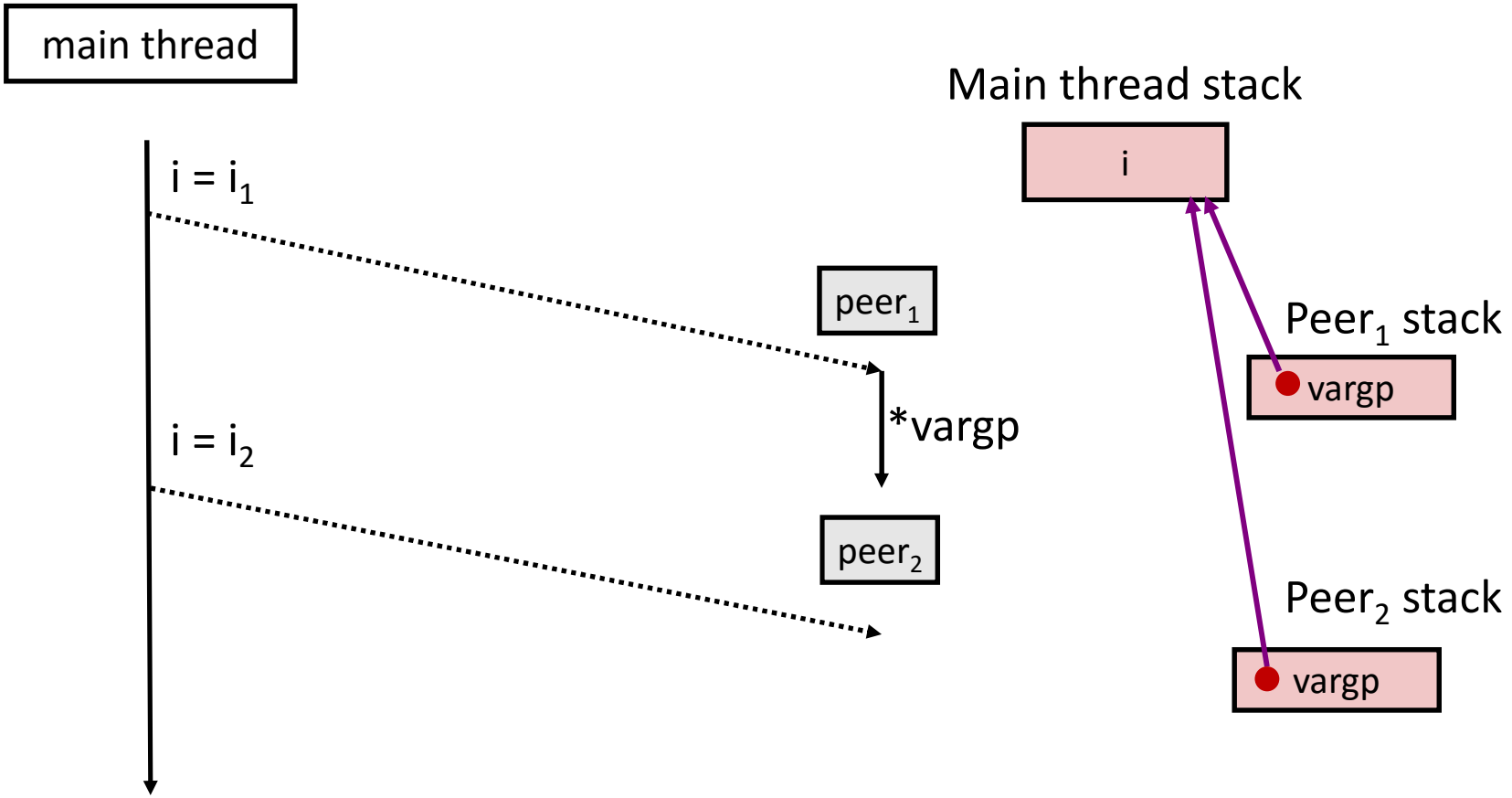
Potential Form of Unintended Sharing

```
for(int i=0; i<10; i++) {
  pthread_create(&tid, NULL, thread, &i);
}
```



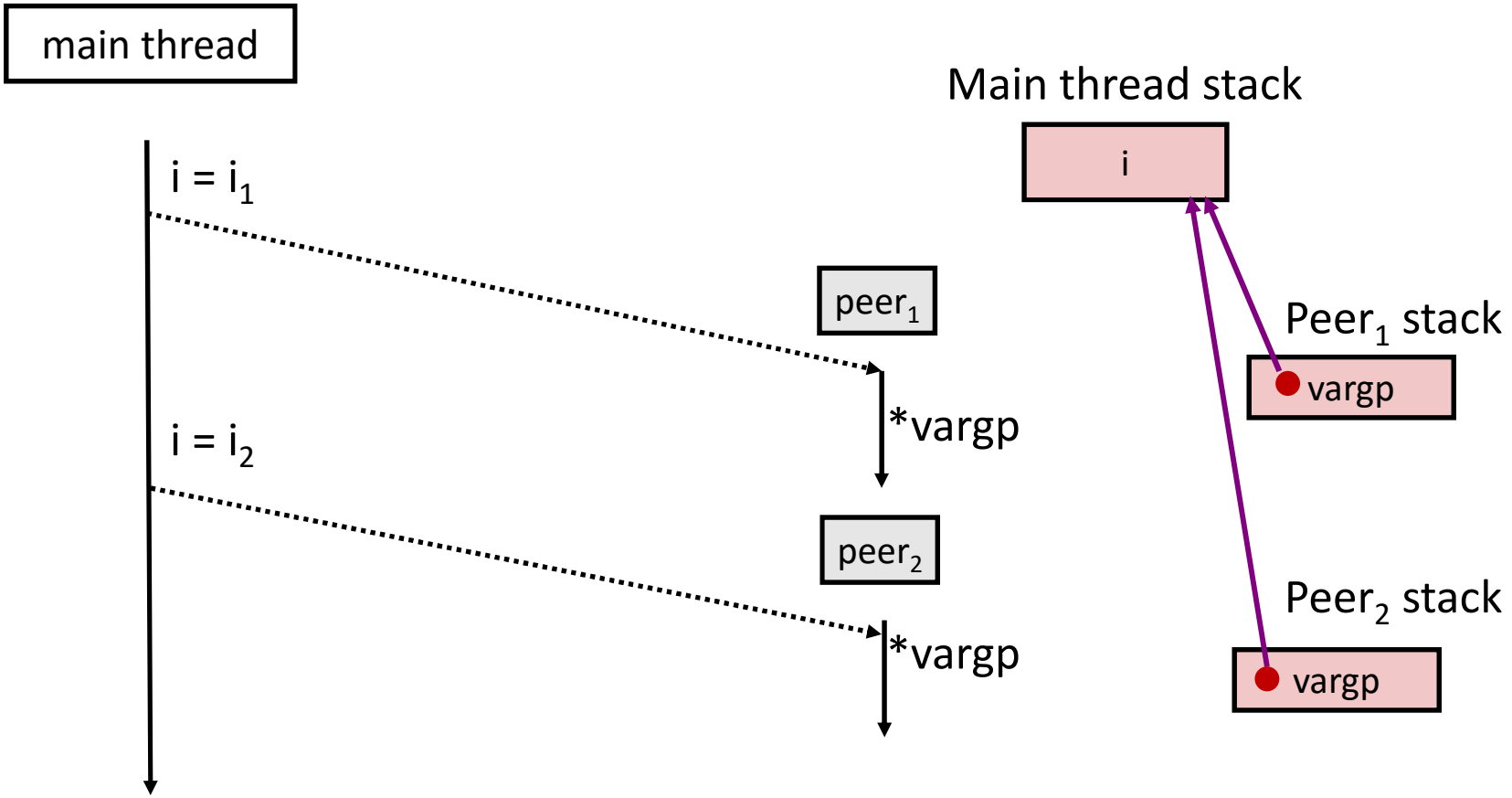
Potential Form of Unintended Sharing

```
for(int i=0; i<10; i++) {
  pthread_create(&tid, NULL, thread, &i);
}
```



Potential Form of Unintended Sharing

```
for(int i=0; i<10; i++) {
  pthread_create(&tid, NULL, thread, &i);
}
```



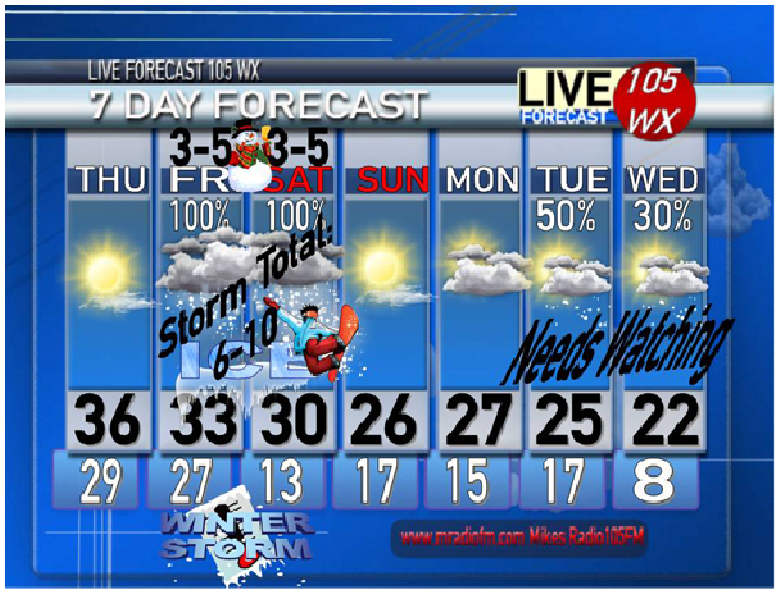
Fix unintended sharing

```
int main (int argc, char *argv[]) {
  pthread_t tid;
  int *argp;
  for(int i=0; i<10; i++) {
    argp = (int*)malloc(sizeof(int));
    *argp = i;
    pthread_create(&tid, NULL, thread, argp);
  }
  pthread_exit(NULL);
  return 0;
}
```

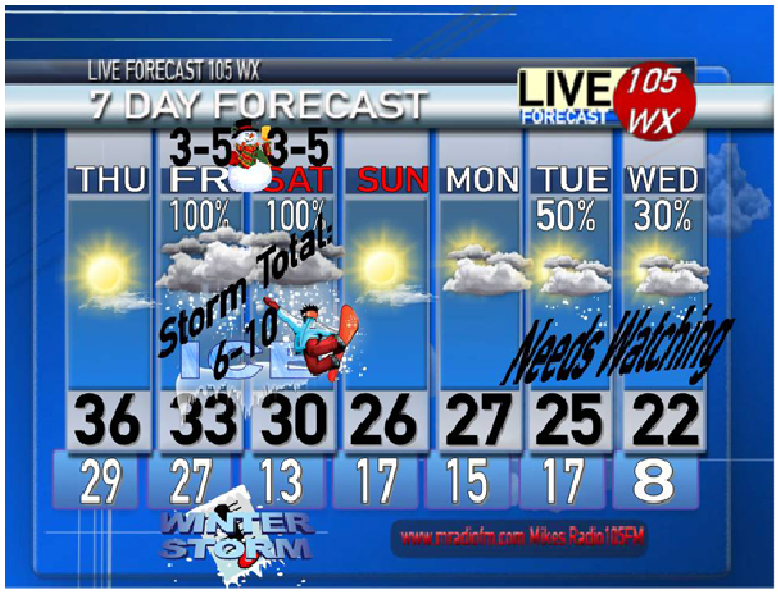
```
void *thread (void *vargp) {
  pthread_detach(pthread_self());
  int arg = *((int*) vargp);
  printf("hello from thread %d!\n", arg);
  return NULL;}
```

Race Conditions

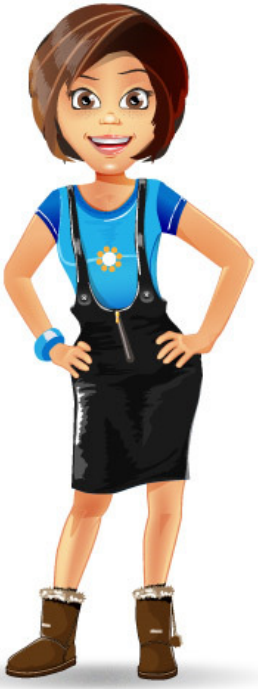
- A **race condition** occurs when two or more threads can access shared data and one of them is a write. Because the thread scheduling algorithm can swap between threads at any time, you don’t know the order in which the threads will attempt to access the shared data.

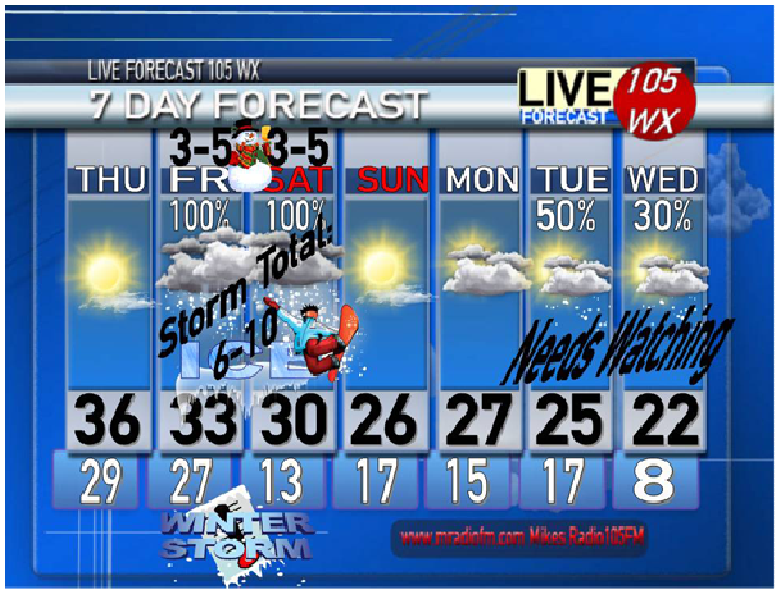


Theres gonna be a snowstorm on Friday.



It is time for some shopping and then a nap at home.




LIVE FORECAST 105 WX
7 DAY FORECAST
LIVE FORECAST 105 WX
3-5
3-5
THU
FRI
SAT
SUN
MON
TUE
WED
100%
100%
50%
30%
Storm Total:
6-10
Needs Watching
36
33
30
26
27
25
22
29
27
13
17
15
17
8
WINTER STORM
www.mradiofm.com Mikes Radio 105FM


Let me help you.


%
SALE-


%
SALE-


Got my shopping done.
MILK


MILK

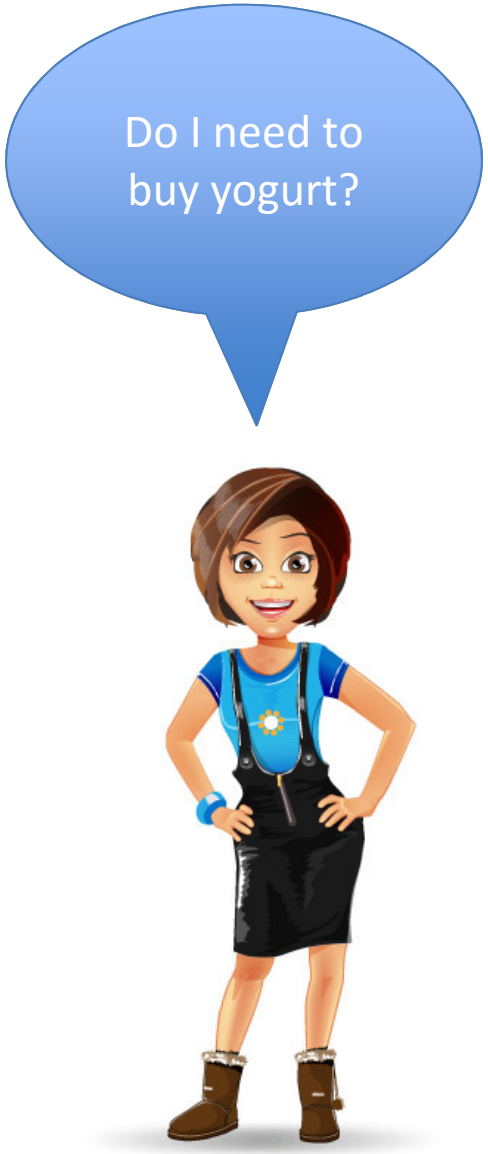

MILK


I hate race condition!
MILK
MILK


MILK


MILK


MILK


Do I need to
buy yogurt?