CSC 252/452: Computer Organization
Fall 2025: Lecture 24

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Announcements

- Final exam: Dec. 13th, 8:30 AM -- 11:30 AM
- Open book test: any sort of paper-based product, e.g., book, **notes**, magazine, old tests.
  - **No** electronic devices
- Problem sets and previous exams are helpful.
- Course evaluation starts today.
  - If >60% students do course evaluation, two extra points are given to each student

Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
  - Single core
  - Multi-core
  - Hyper-threading
  - Cache coherence

Programmers View of A Process

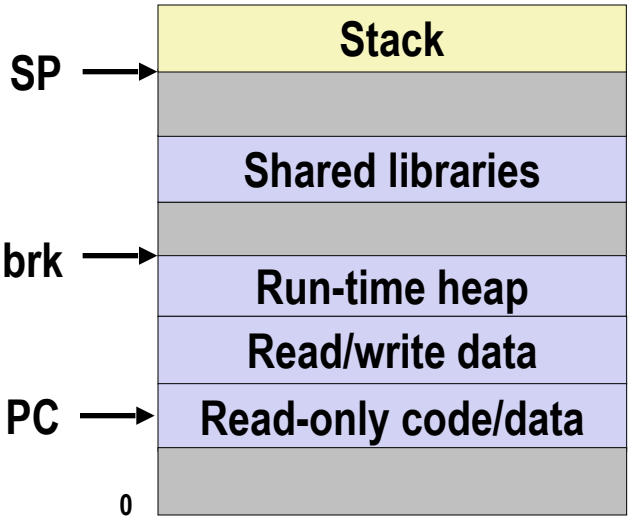
- Process = process context + code, data, and stack

Process context

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

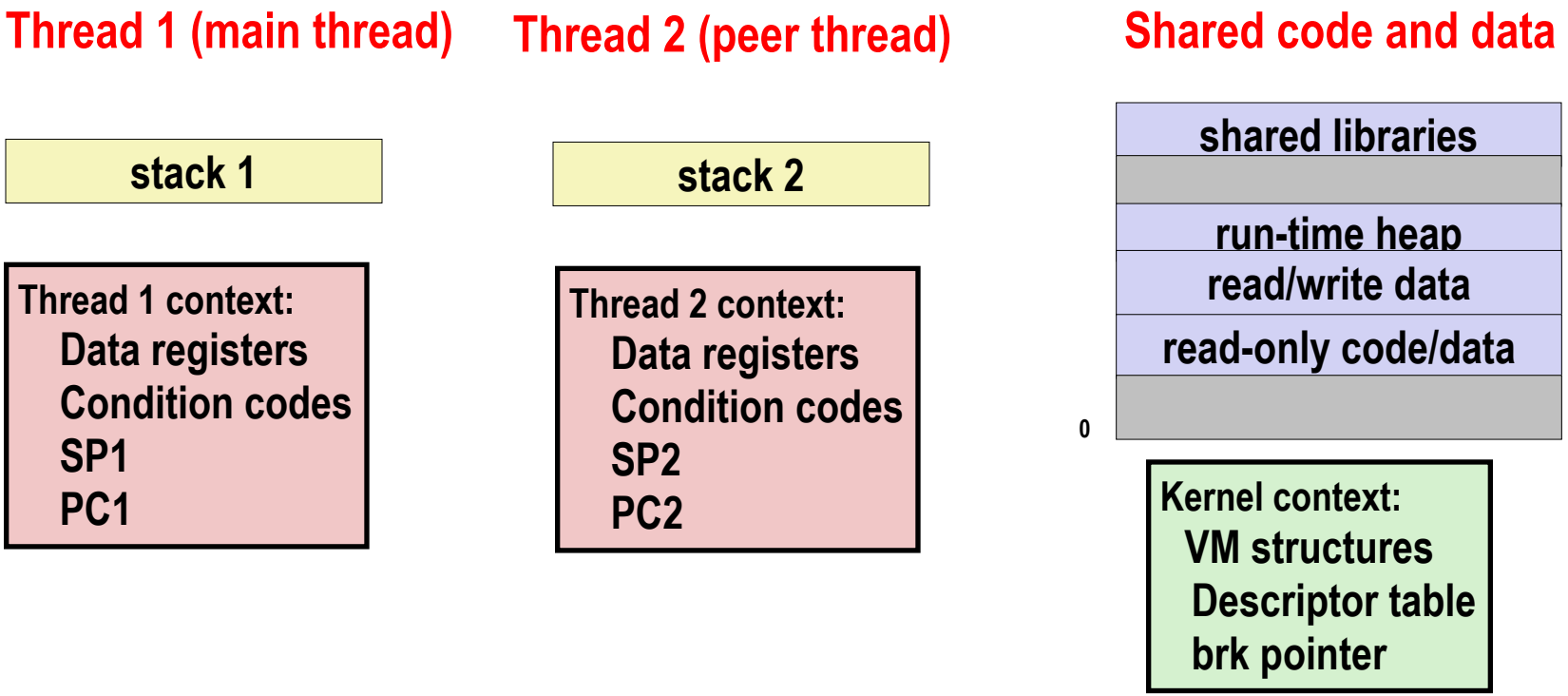
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



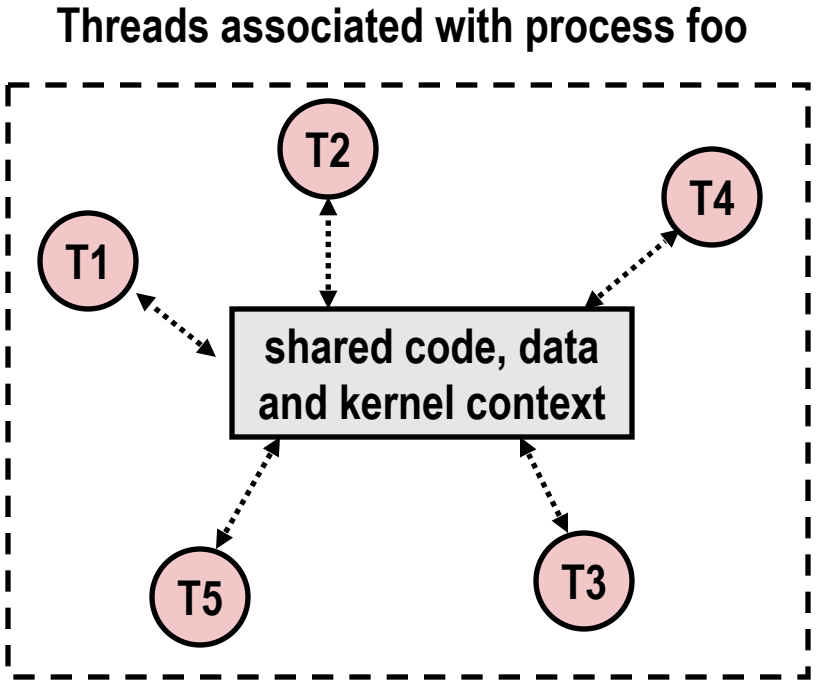
A Process With Multiple Threads

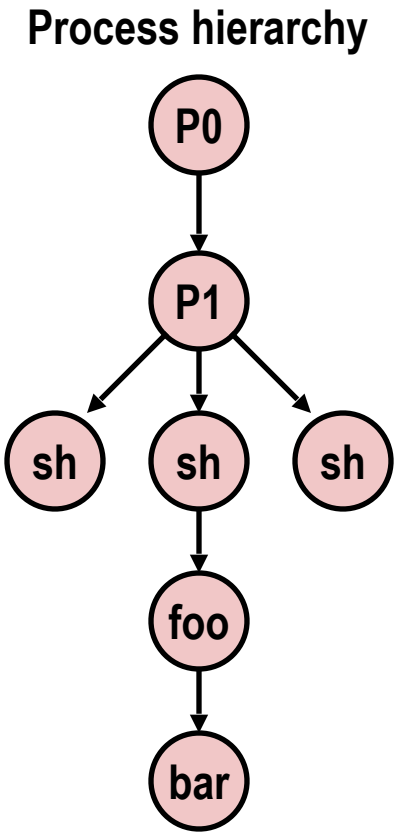
- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)



Logical View of Threads

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

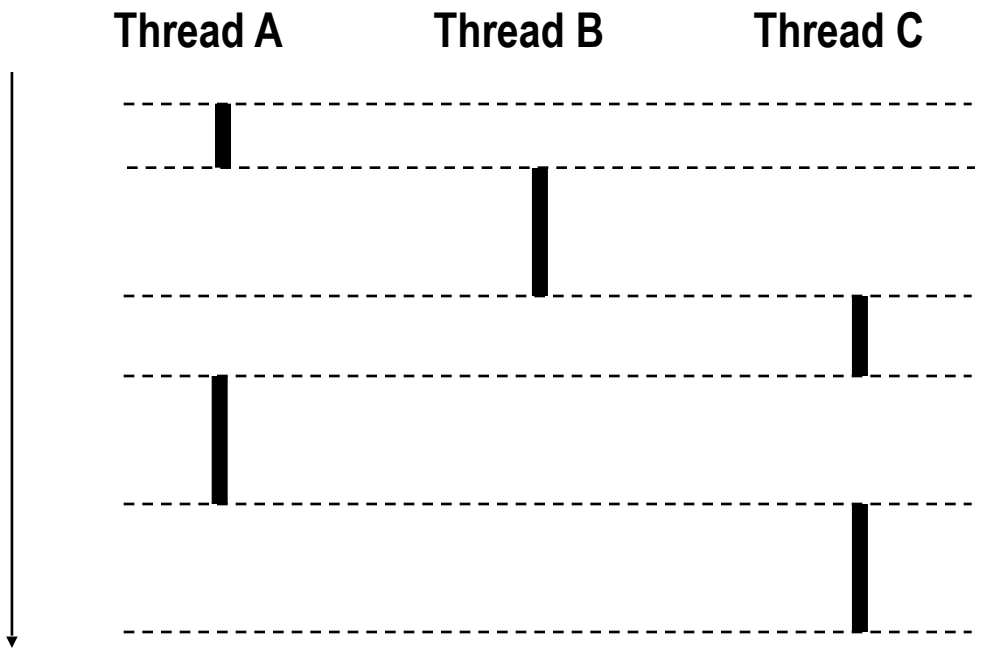




Concurrent Threads

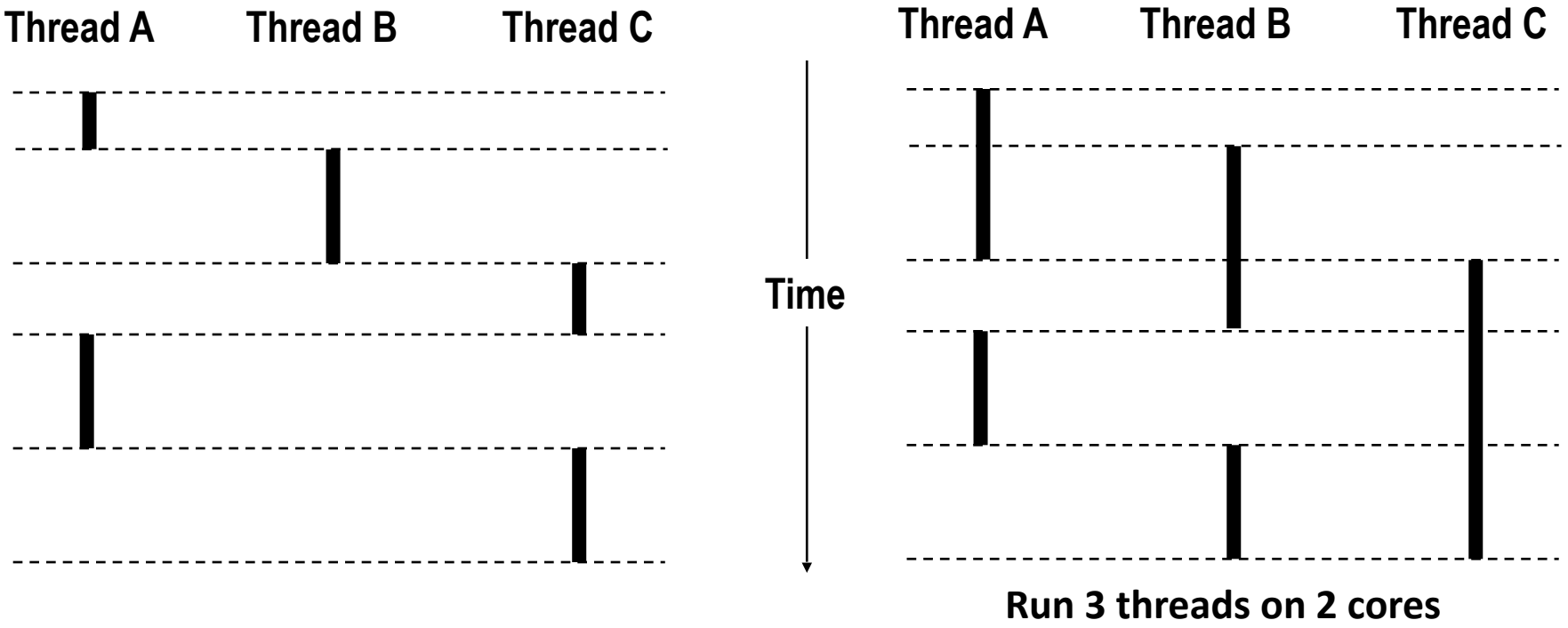
- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

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



Concurrent Thread Execution

- Single Core Processor
  - Simulate parallelism by time slicing
- Multi Core Processor
  - Threads can have true parallelisms



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, controlled by kernel

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, controlled by kernel
- How threads and processes are different
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are less expensive than processes
    - Space: threads share the same virtual address space except stacks, but processes have their own virtual address space
    - Process control (creating and reaping) twice as expensive
    - Typical 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()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

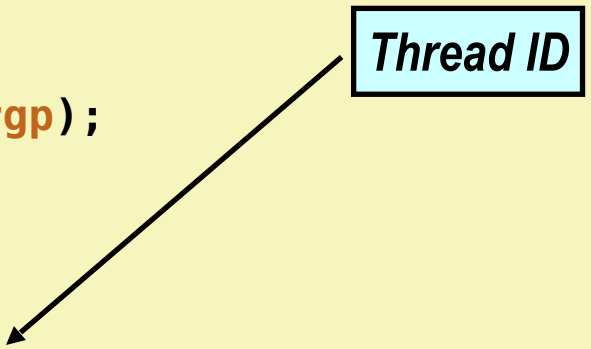
```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

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

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

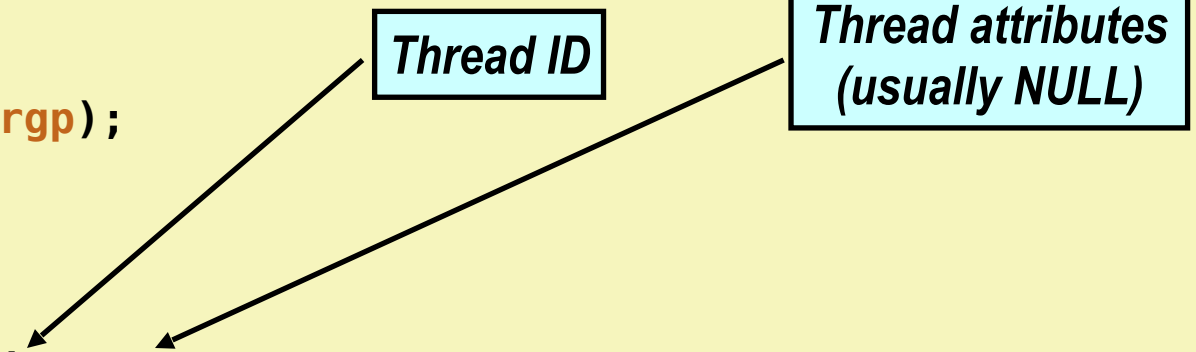
hello.c

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




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

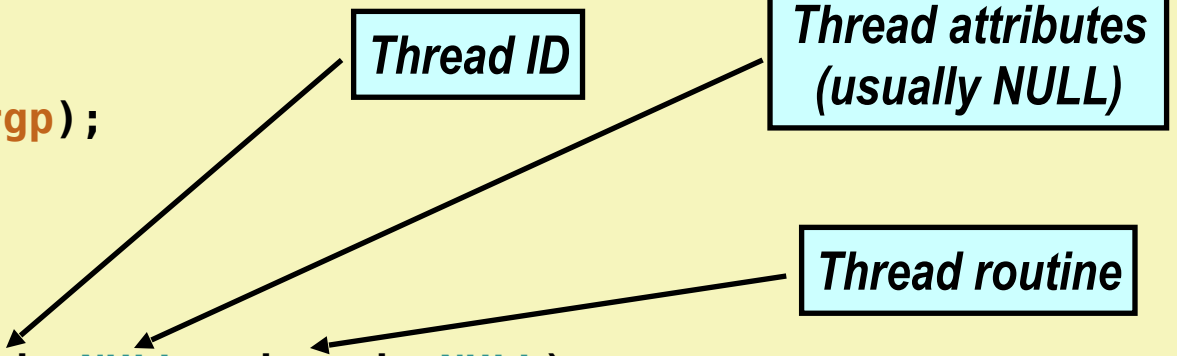
hello.c

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




```c
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)

Thread routine

Thread arguments (void *p)

```c
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)

Thread routine

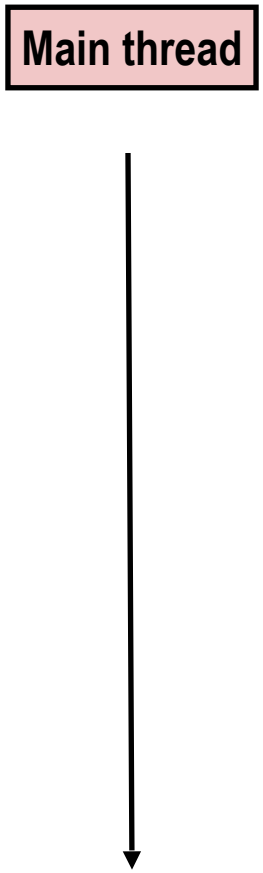
Thread arguments (void *p)

Return value (void **p)

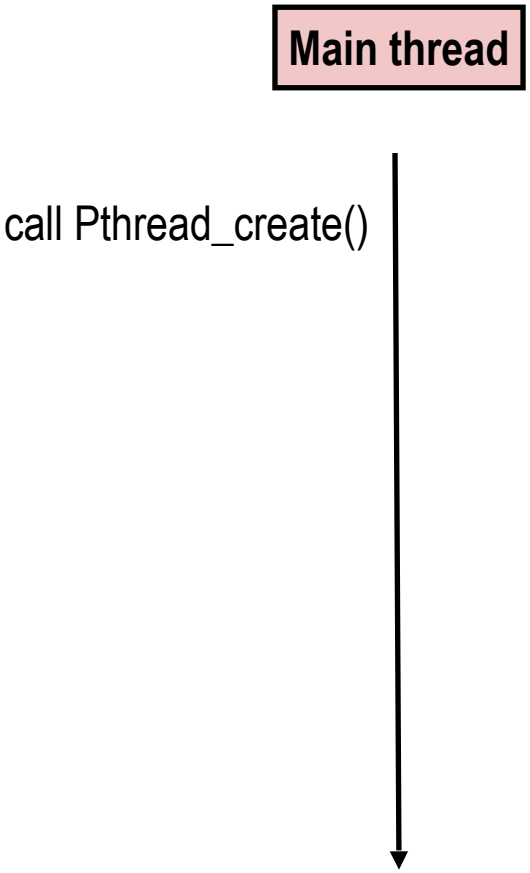
```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

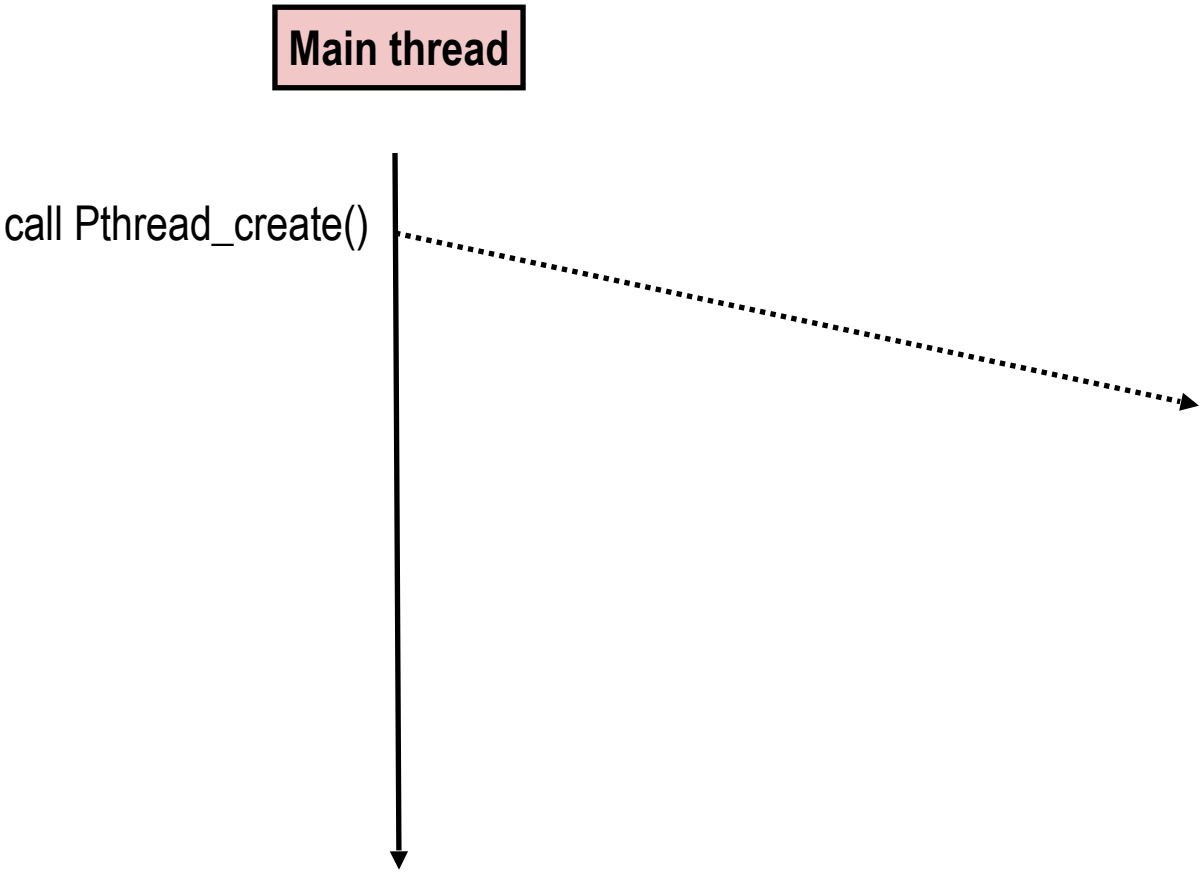
Execution of Threaded “hello, world”



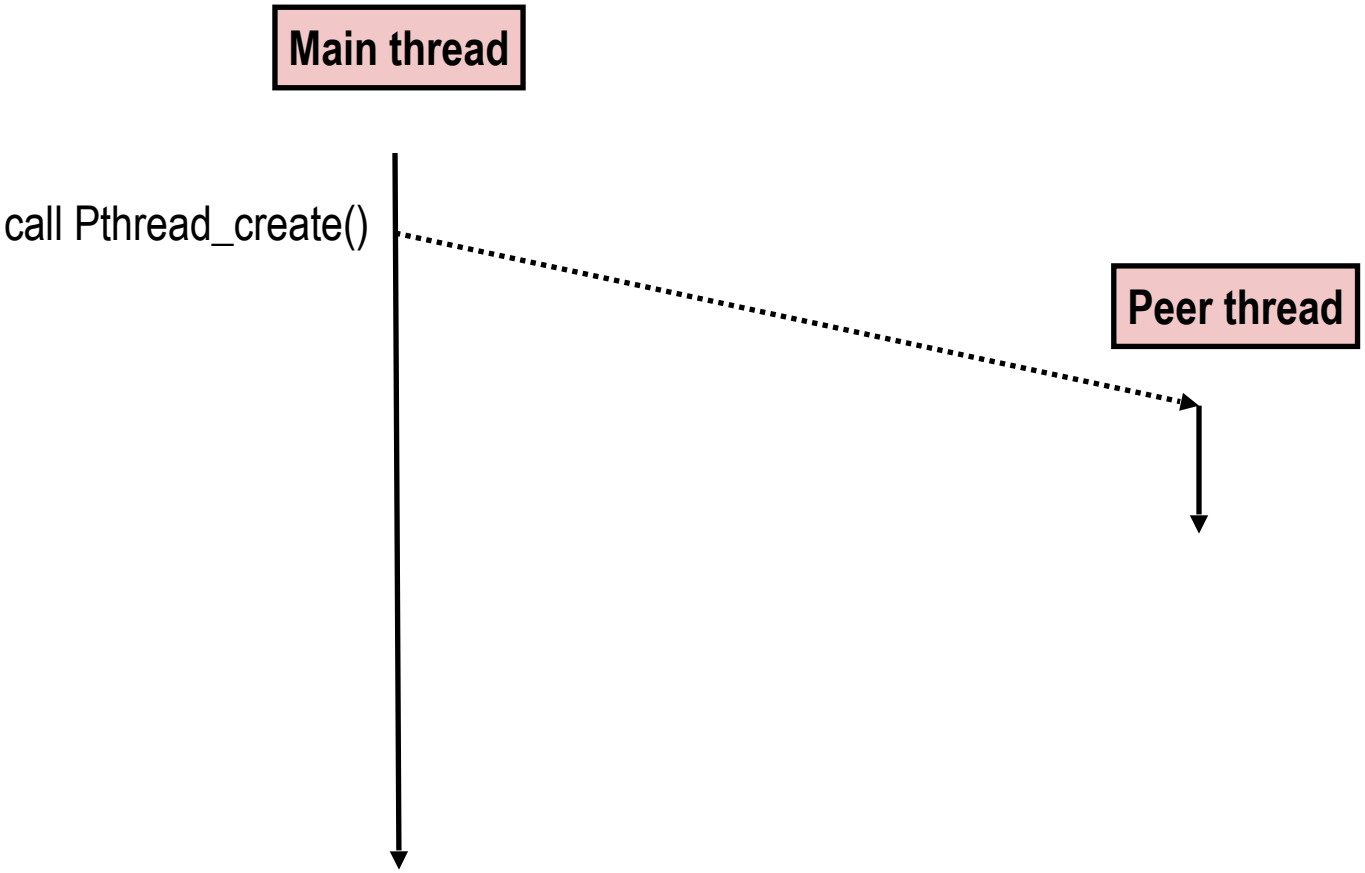
Execution of Threaded “hello, world”



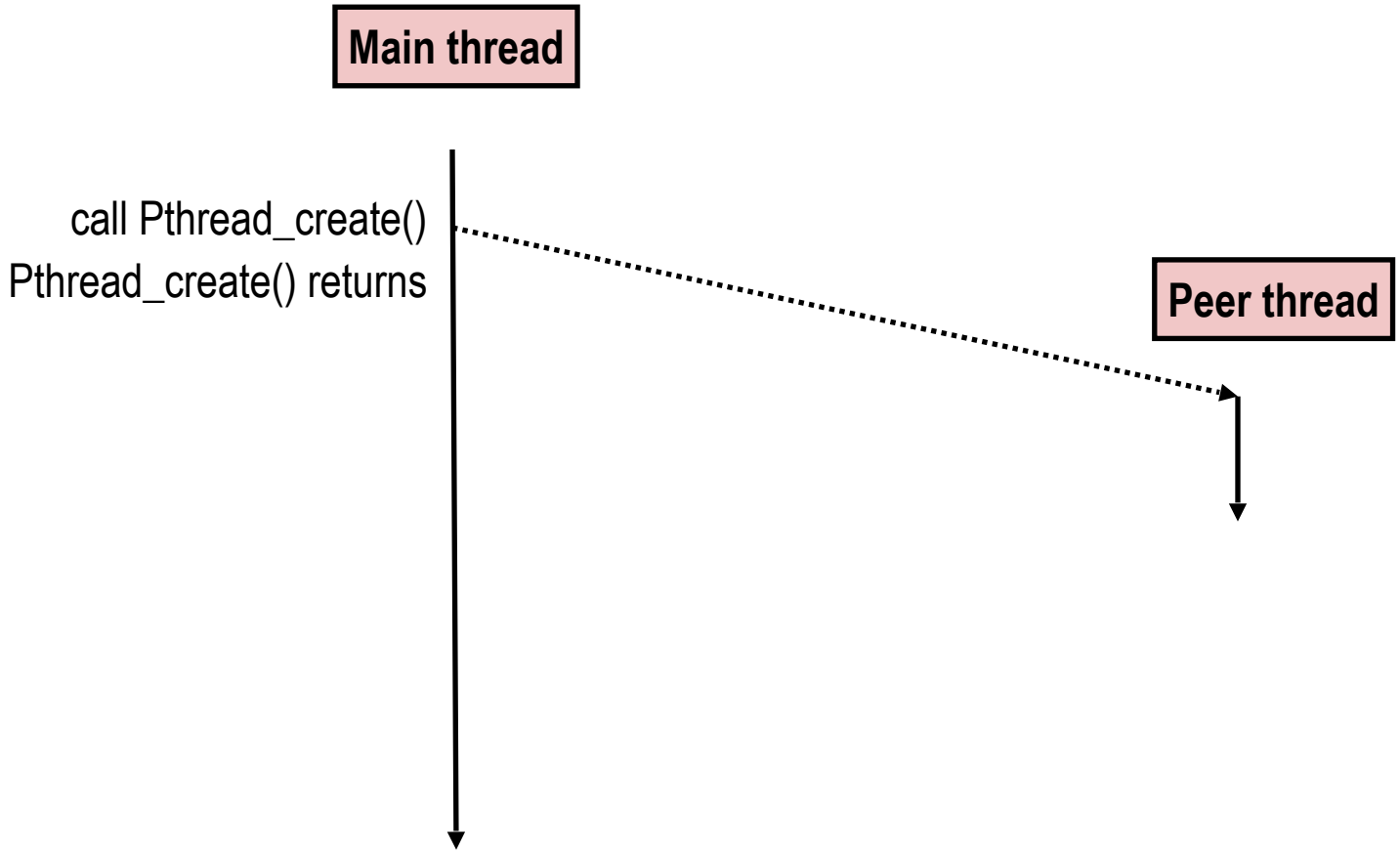
Execution of Threaded “hello, world”



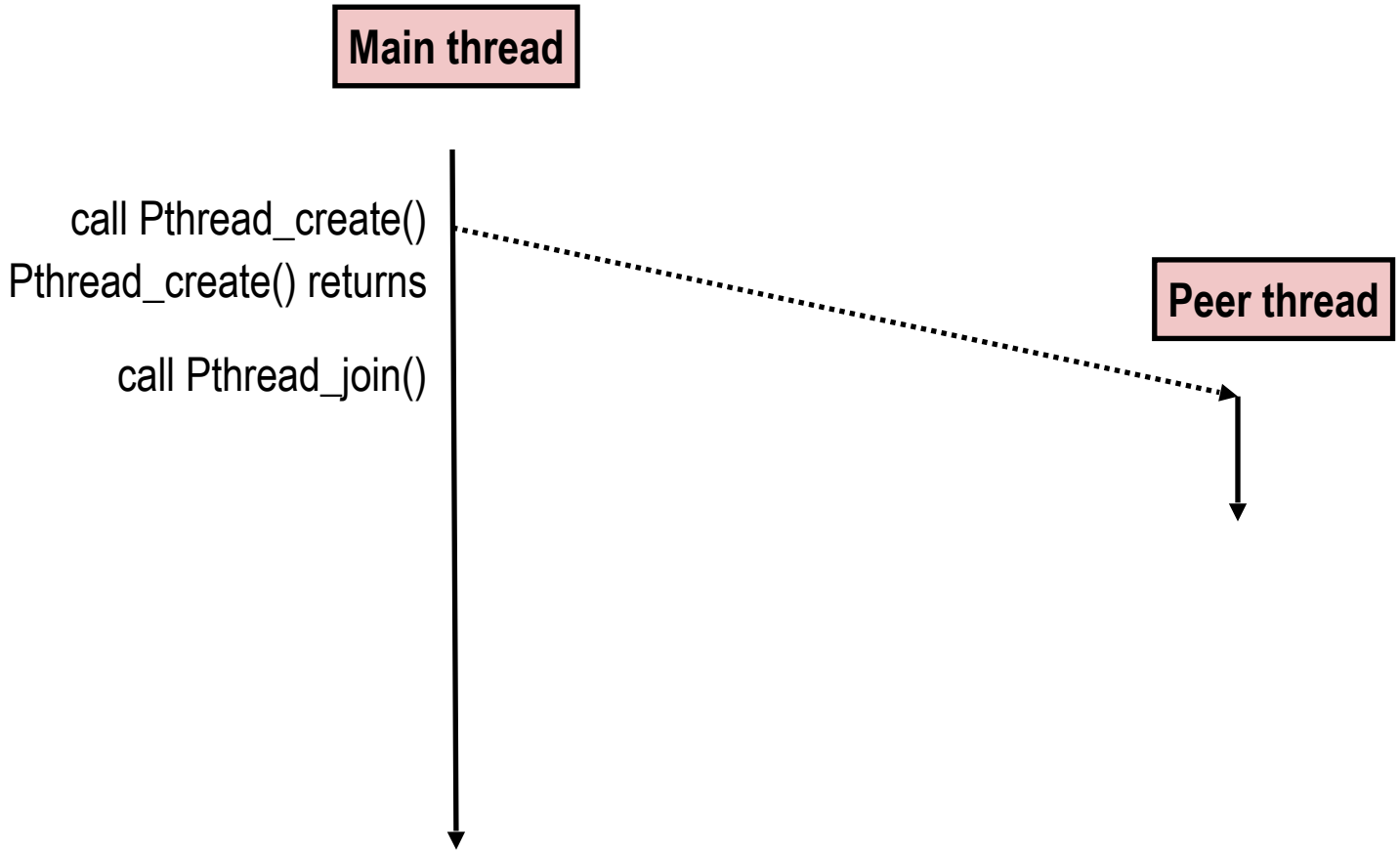
Execution of Threaded “hello, world”



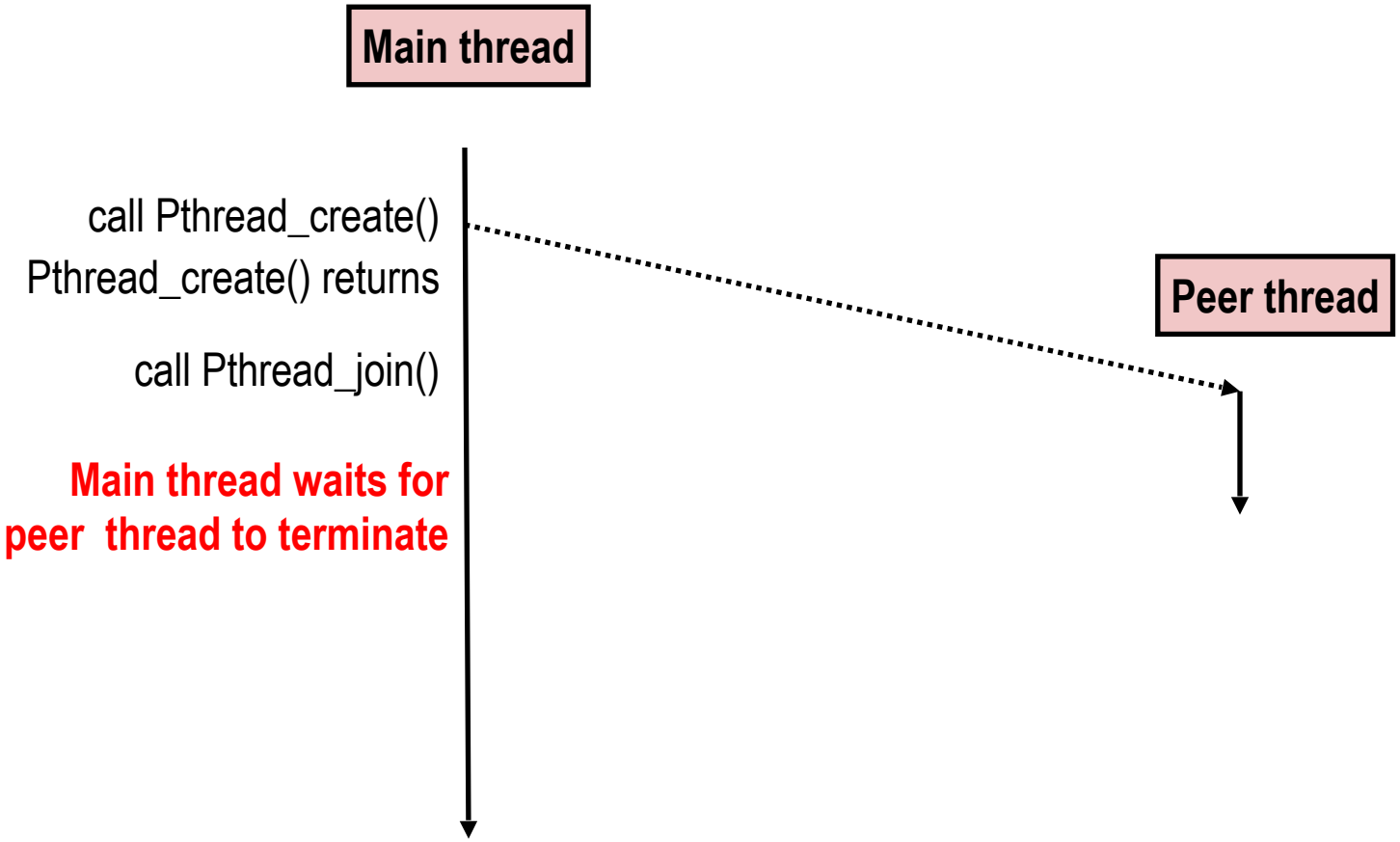
Execution of Threaded “hello, world”



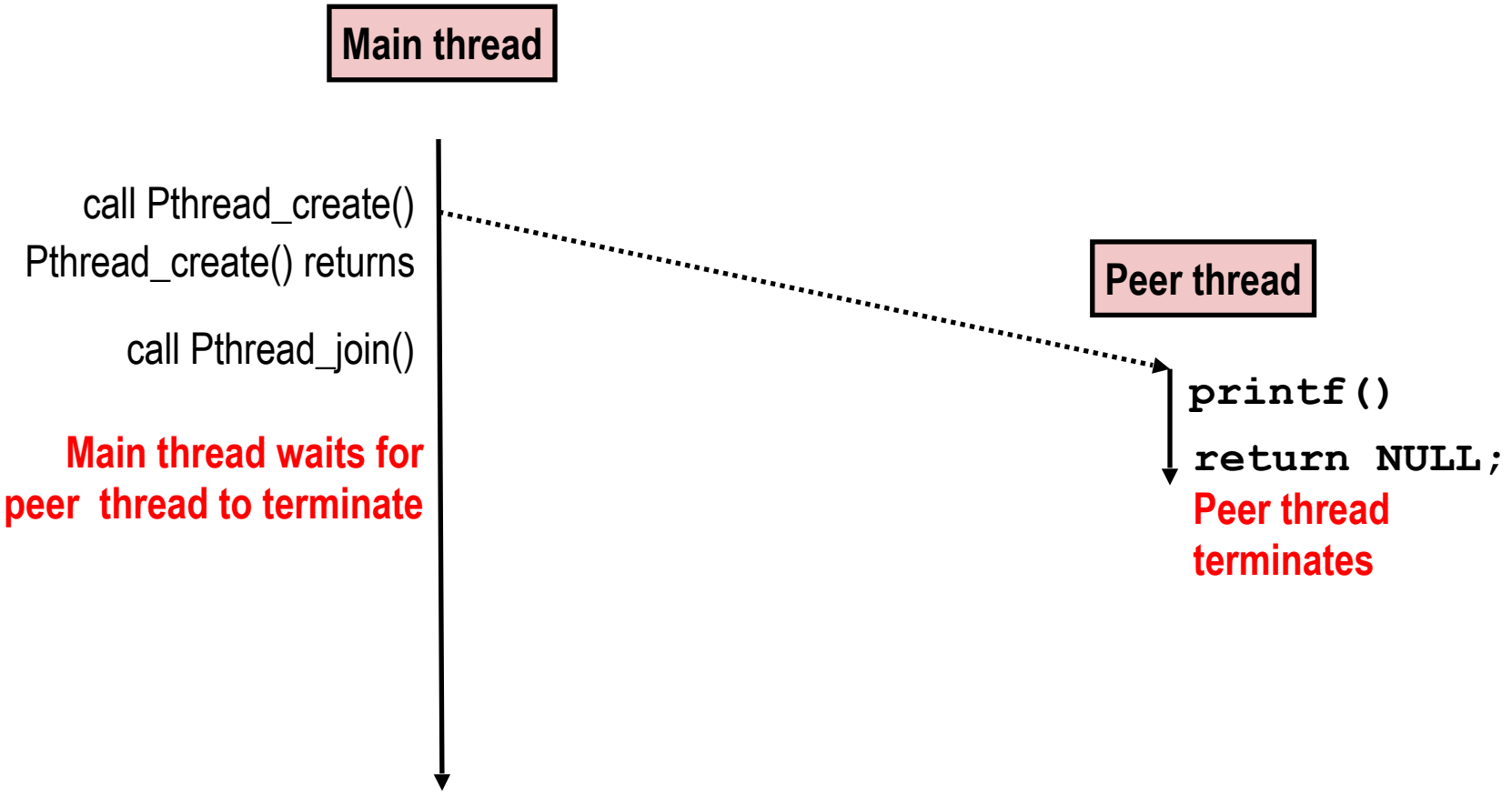
Execution of Threaded “hello, world”



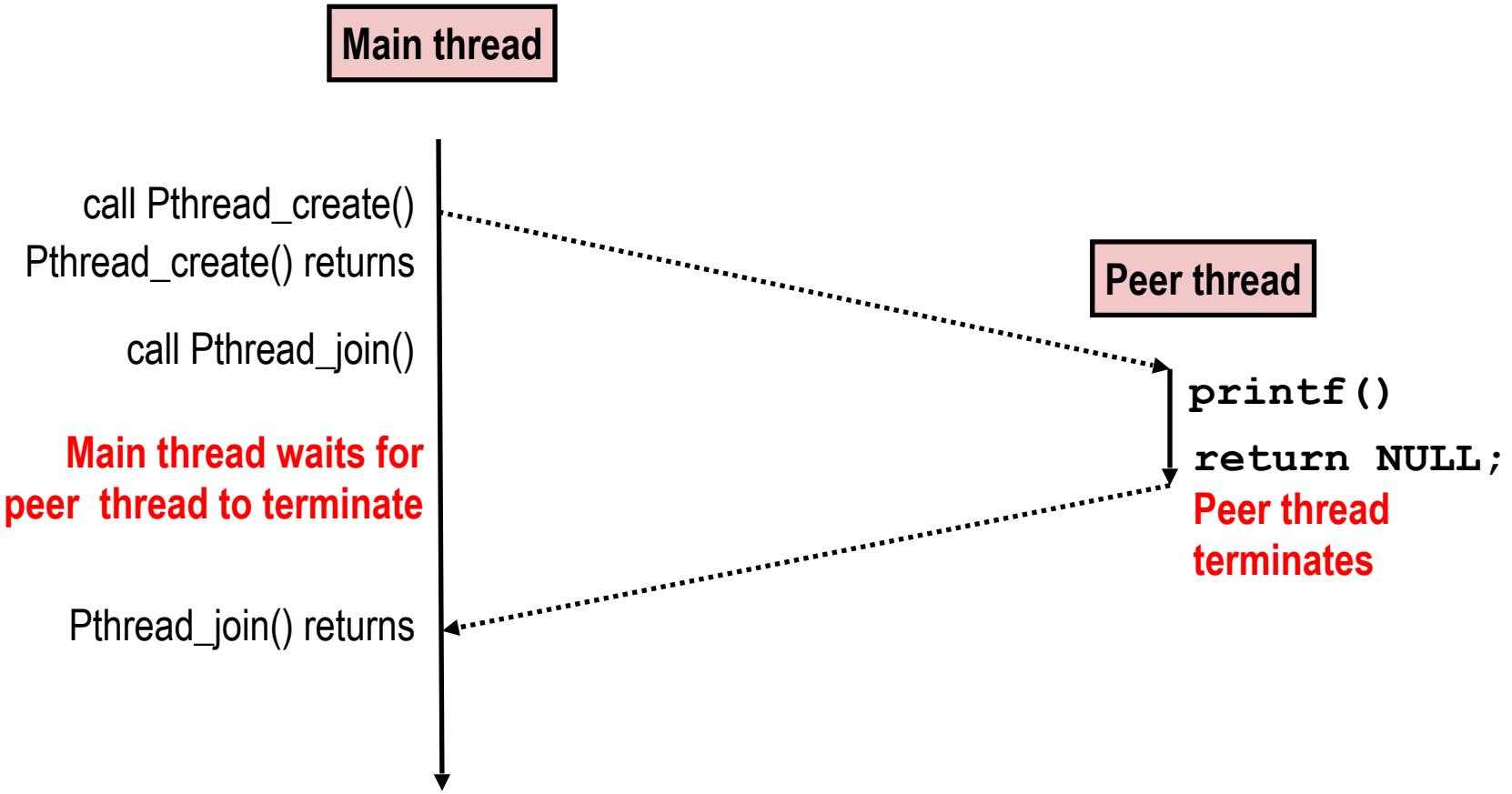
Execution of Threaded “hello, world”



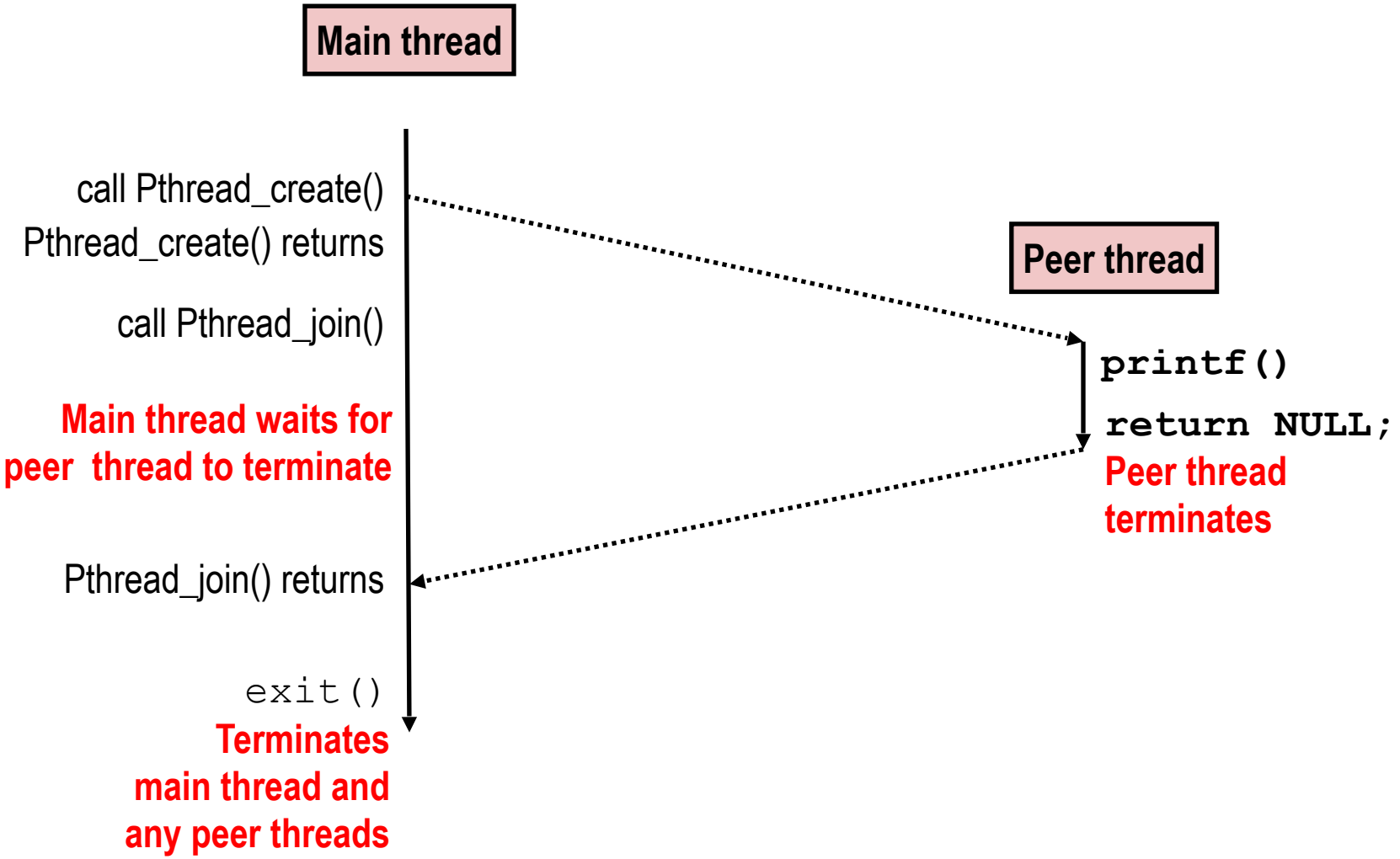
Execution of Threaded “hello, world”



Execution of Threaded “hello, world”



Execution of Threaded “hello, world”

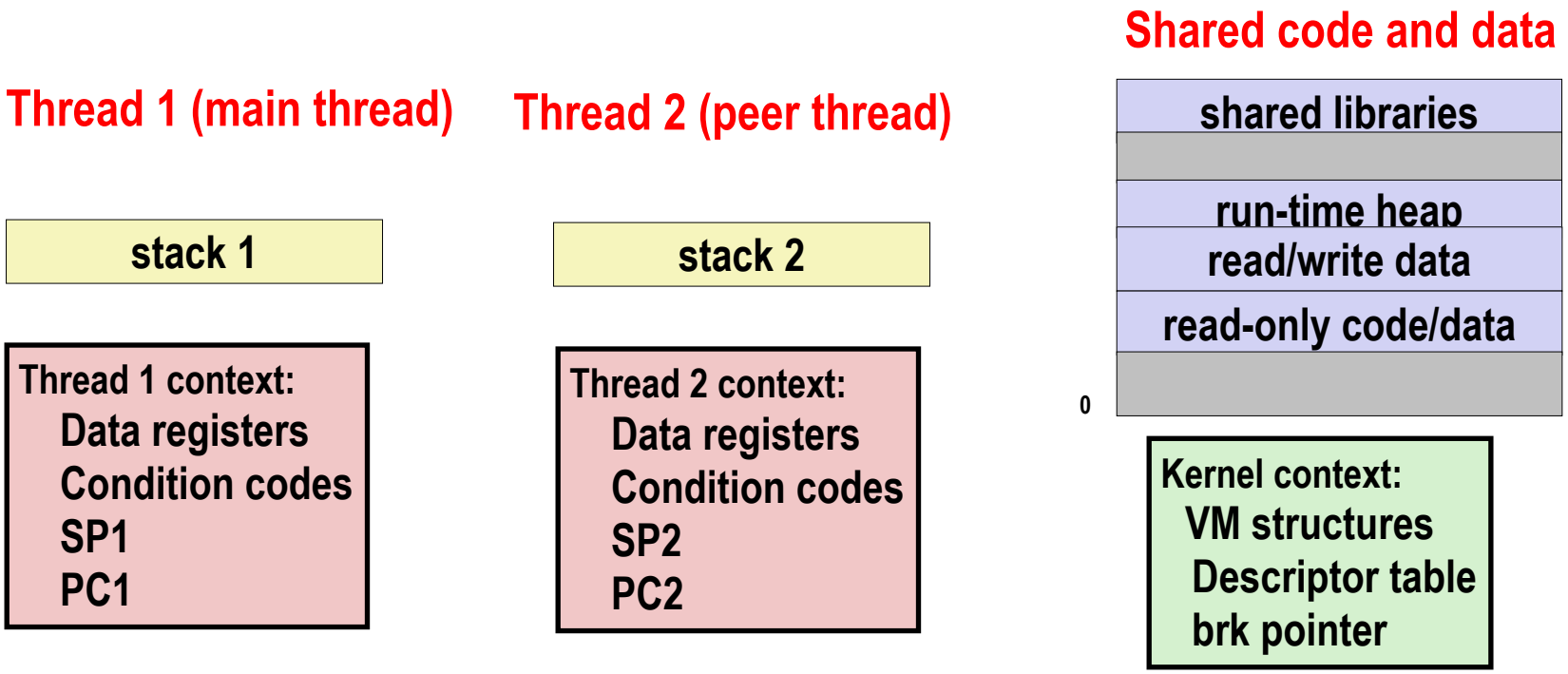


Today

- From process to threads
  - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
  - Single core
  - Multi-core
  - Cache coherence

Shared Variables in Threaded C Programs

- One great thing about threads is that they can share same program variables.
- Question: Which variables in a threaded C program are shared?
- Intuitively, the answer is as simple as "*global variables are shared*" and "*stack variables are private*". Not so simple in reality.



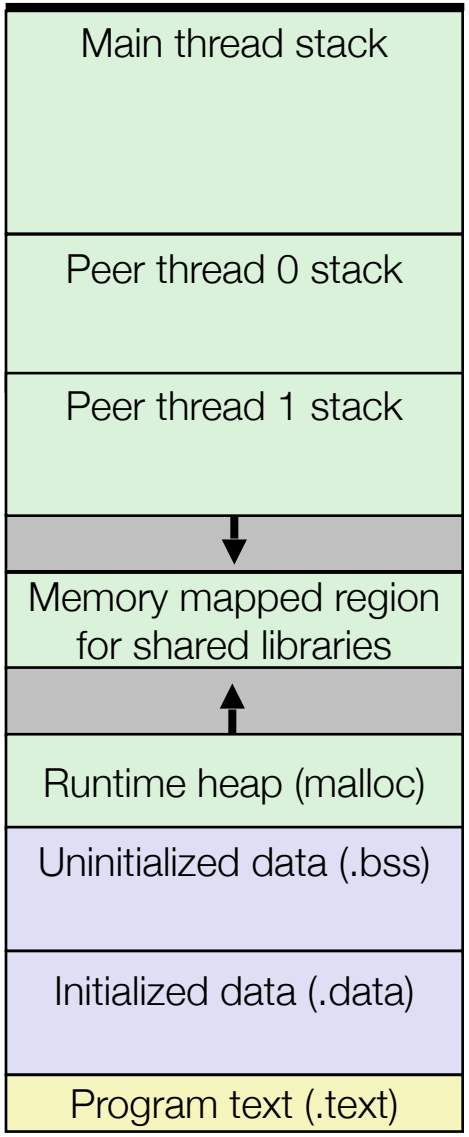
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



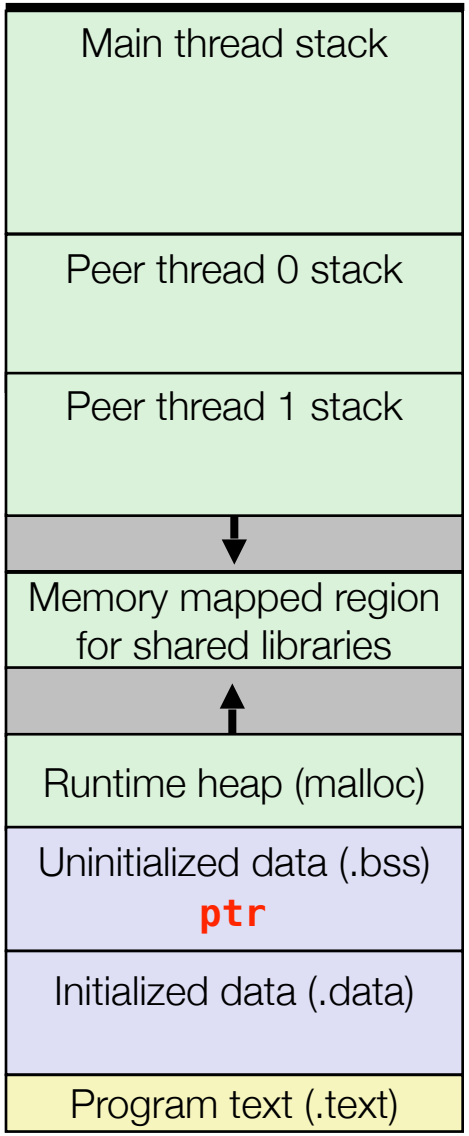
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



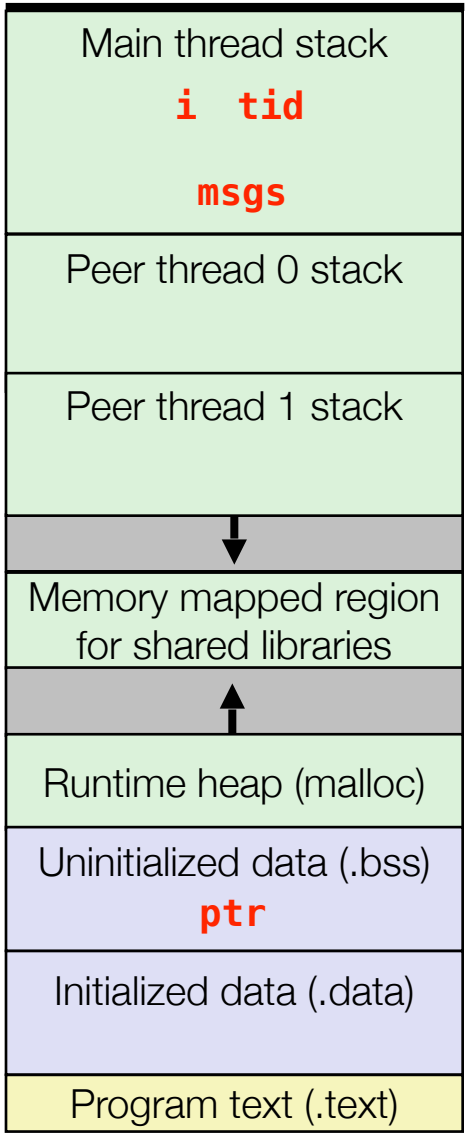
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



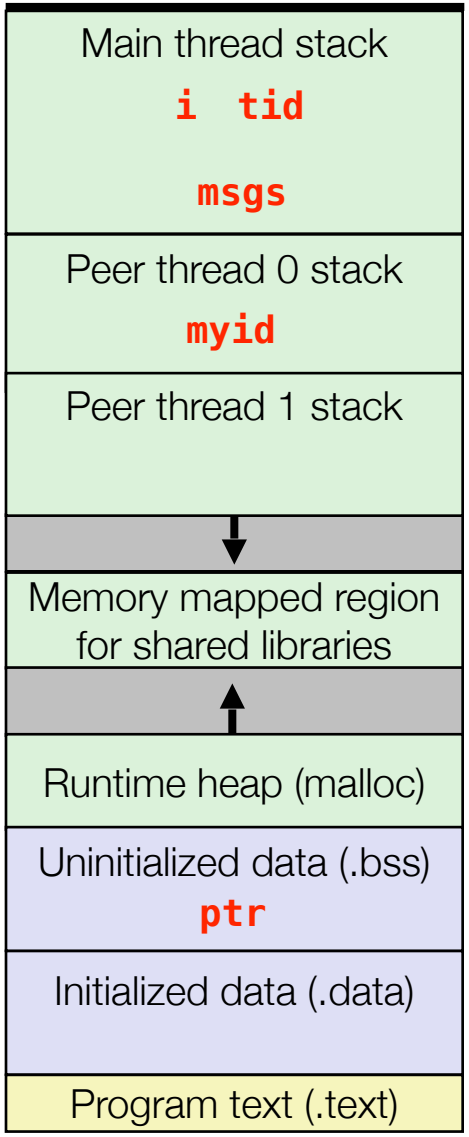
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



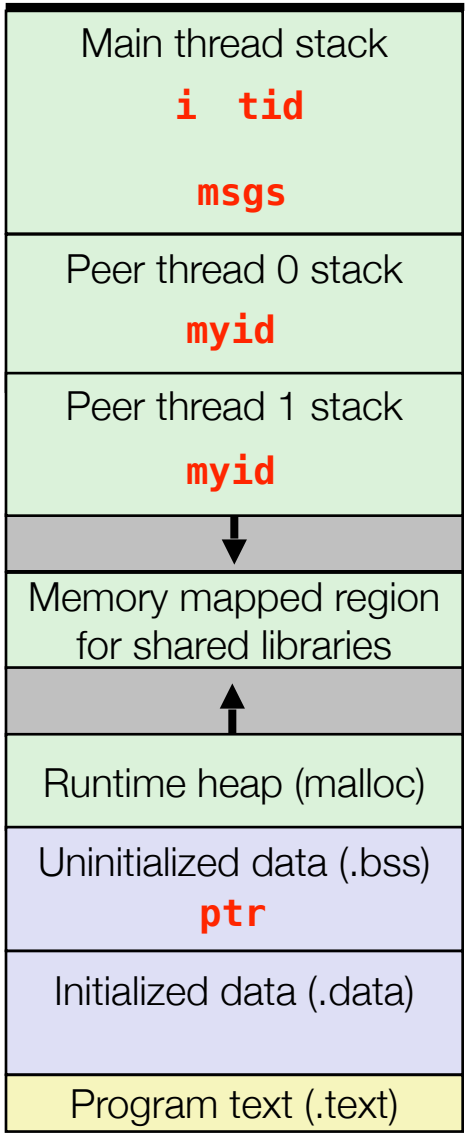
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



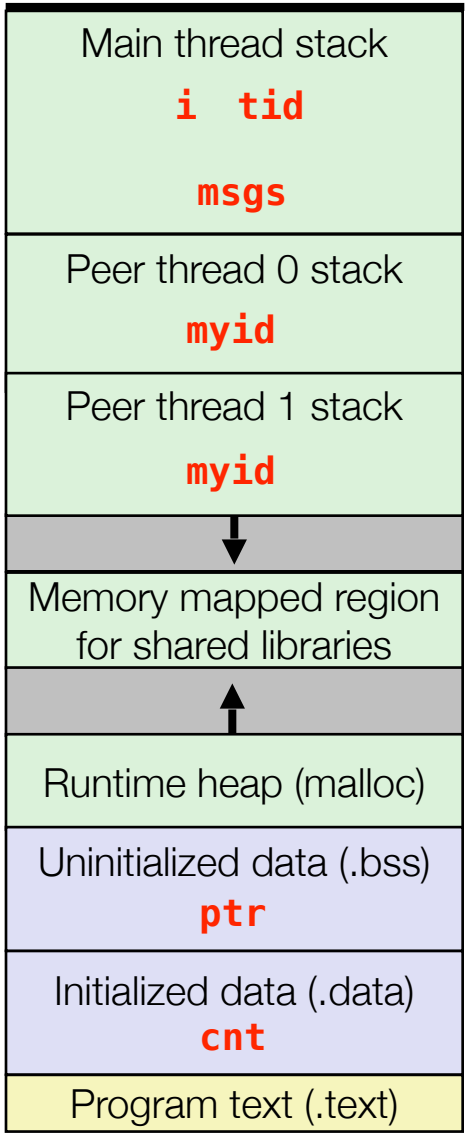
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



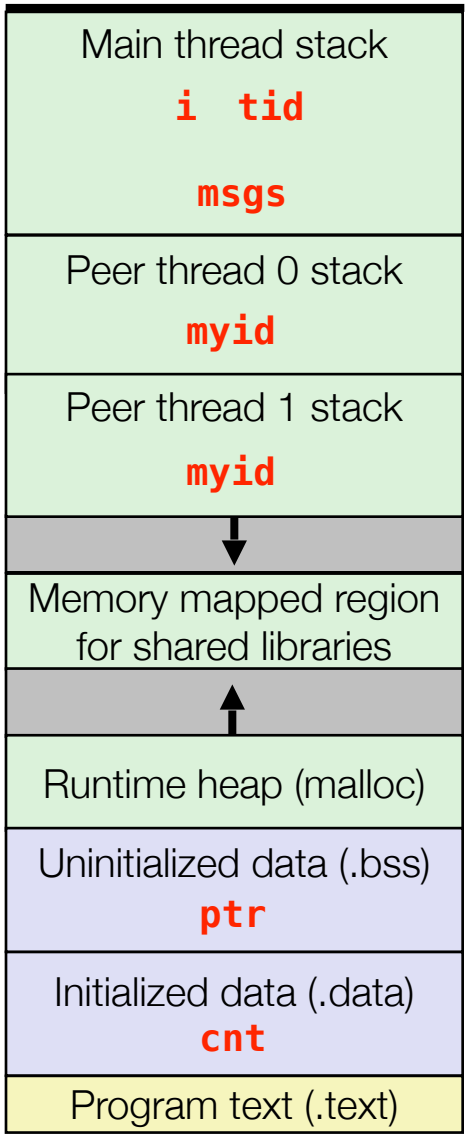
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



p0 p1 main

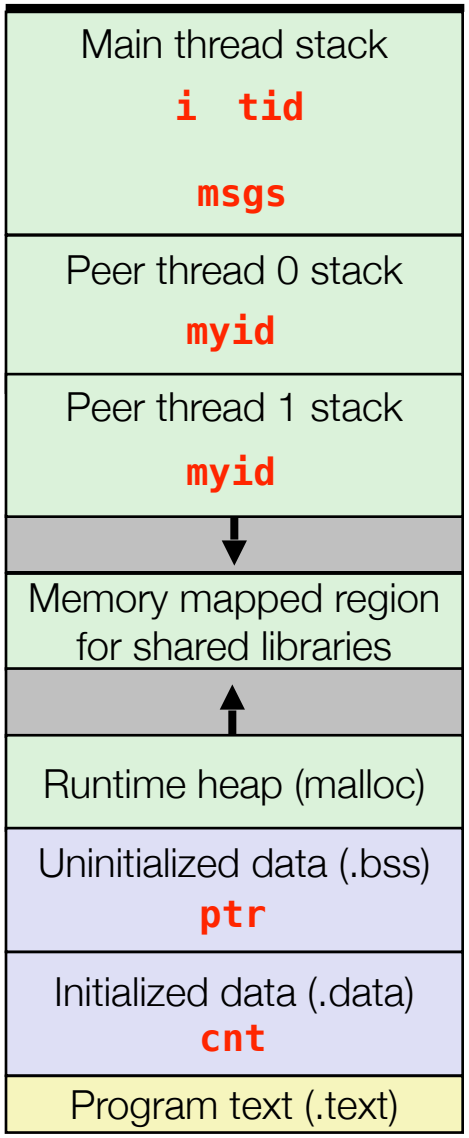
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



p0 p1 main

p0 p1

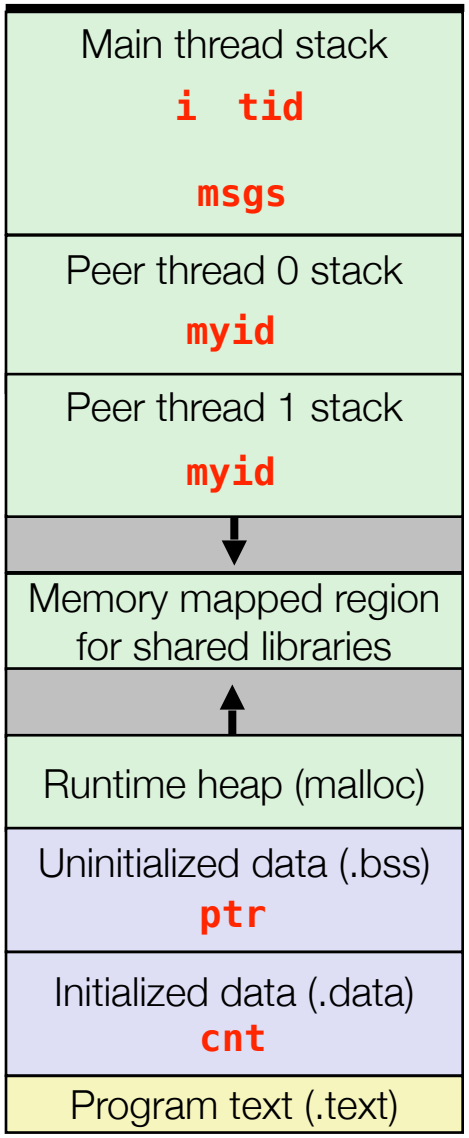
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



main

p0 p1 main

p0 p1

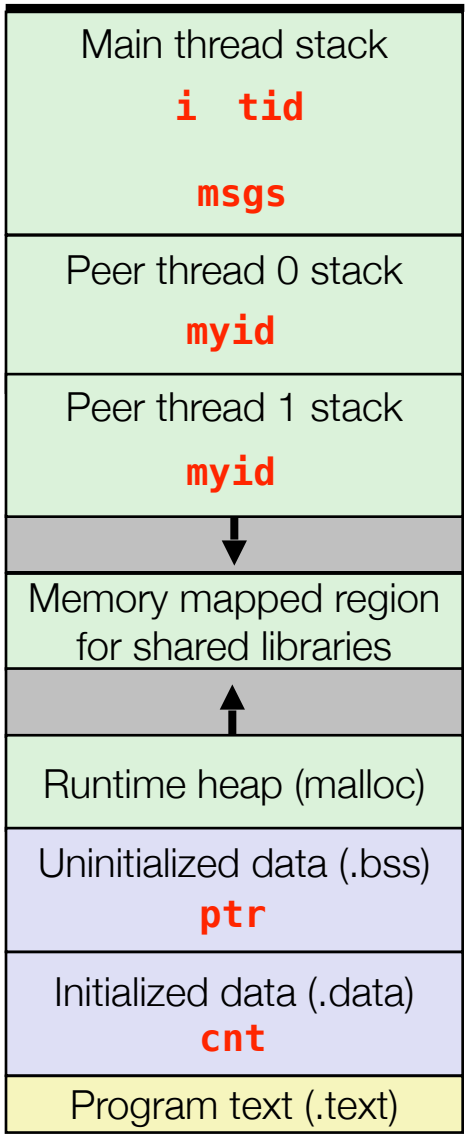
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



main

p0 p1 main

p0 p1 main

p0 p1

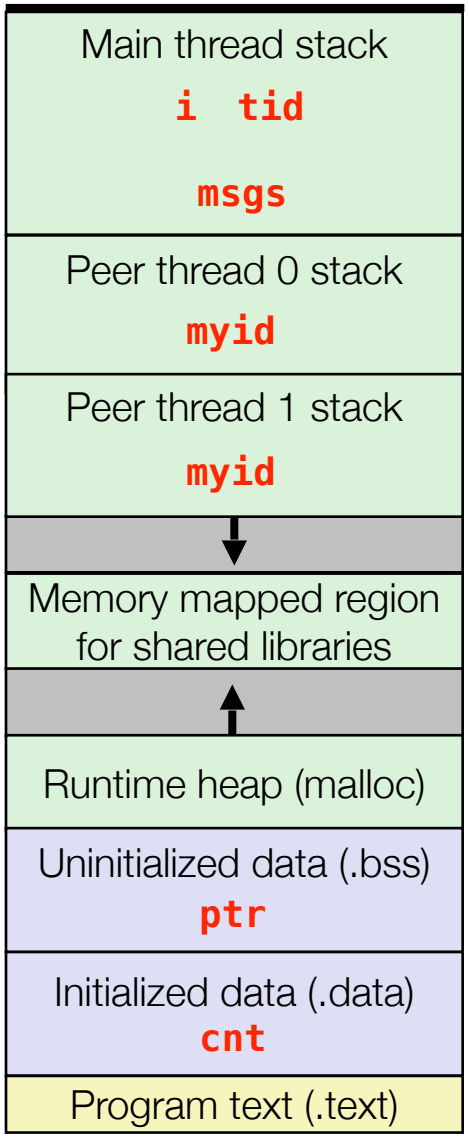
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



main

p0 p1 main

p0

p0 p1 main

p0 p1

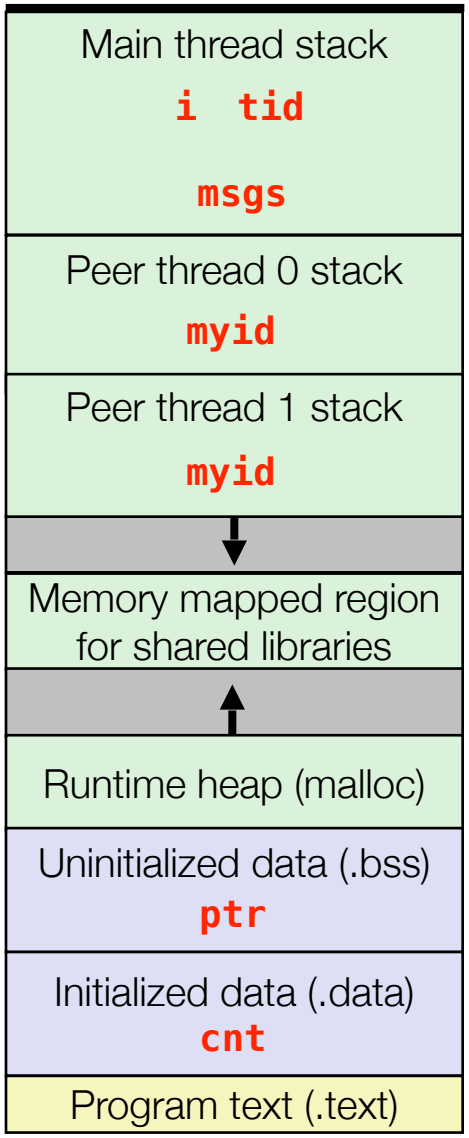
Example Program to Illustrate Sharing

```c
char **ptr;  /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    pthread_exit(NULL);
}
```

sharing.c



main

p0 p1 main

p0

p1

p0 p1 main

p0 p1

Synchronizing Threads

- Shared variables are handy...

- …but introduce the possibility of nasty *synchronization* errors.

Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
   pthread_t tid1, tid2;
   long niters = 10000;

   Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * 10000))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

https://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/

Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
   pthread_t tid1, tid2;
   long niters = 10000;

   Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * 10000))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt
OK cnt=20000

linux> ./badcnt
BOOM! cnt=13051
```

`cnt` should be 20,000.

What went wrong?

https://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
-------------------------------------
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
-------------------------------------
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

H_i : Head

L_i : Load `cnt`

U_i : Update `cnt`

S_i : Store `cnt`

T_i : Tail

Concurrent Execution

- Key observation: In general, any sequentially consistent interleaving is possible, but some give an unexpected result!

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt (shared)
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2

Thread 1 critical section

Thread 2 critical section

L_i	`movq cnt(%rip),%rdx`
U_i	`addq $1, %rdx`
S_i	`movq %rdx, cnt(%rip)`

Concurrent Execution (cont)

- A legal (feasible) but undesired ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt (shared)
1	L_1	0	-	0
1	U_1	1	-	0
2	L_2	-	0	0
1	S_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1

L_i	`movq cnt(%rip),%rdx`
U_i	`addq $1, %rdx`
S_i	`movq %rdx, cnt(%rip)`

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

```
    movq   (%rdi), %rcx
    testq  %rcx,%rcx
    jle    .L2
    movl   $0, %eax
.L3:
    movq   cnt(%rip),%rdx
    addq   $1, %rdx
    movq   %rdx, cnt(%rip)
    addq   $1, %rax
    cmpq   %rcx, %rax
    jne    .L3
.L2:
```

H_i : Head

L_i : Load `cnt`

U_i : Update `cnt`

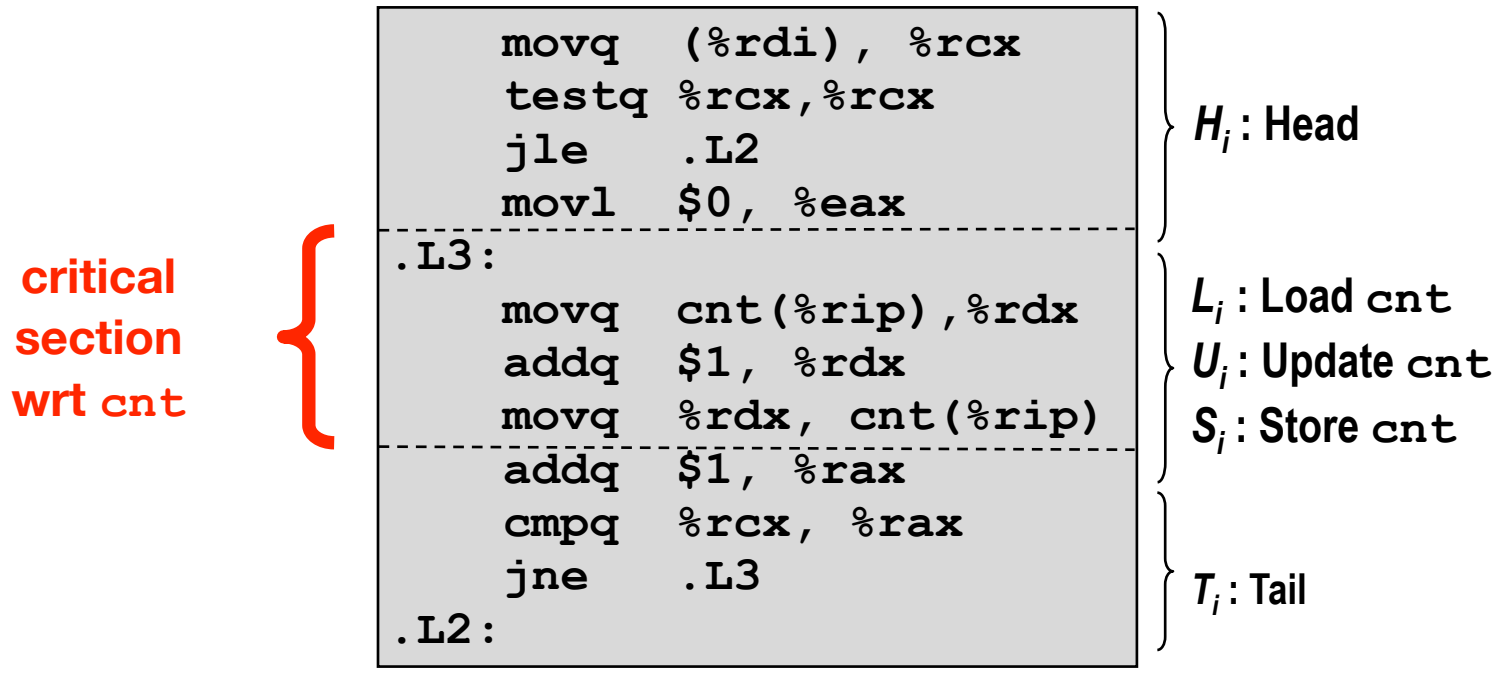
S_i : Store `cnt`

T_i : Tail

critical section wrt `cnt`

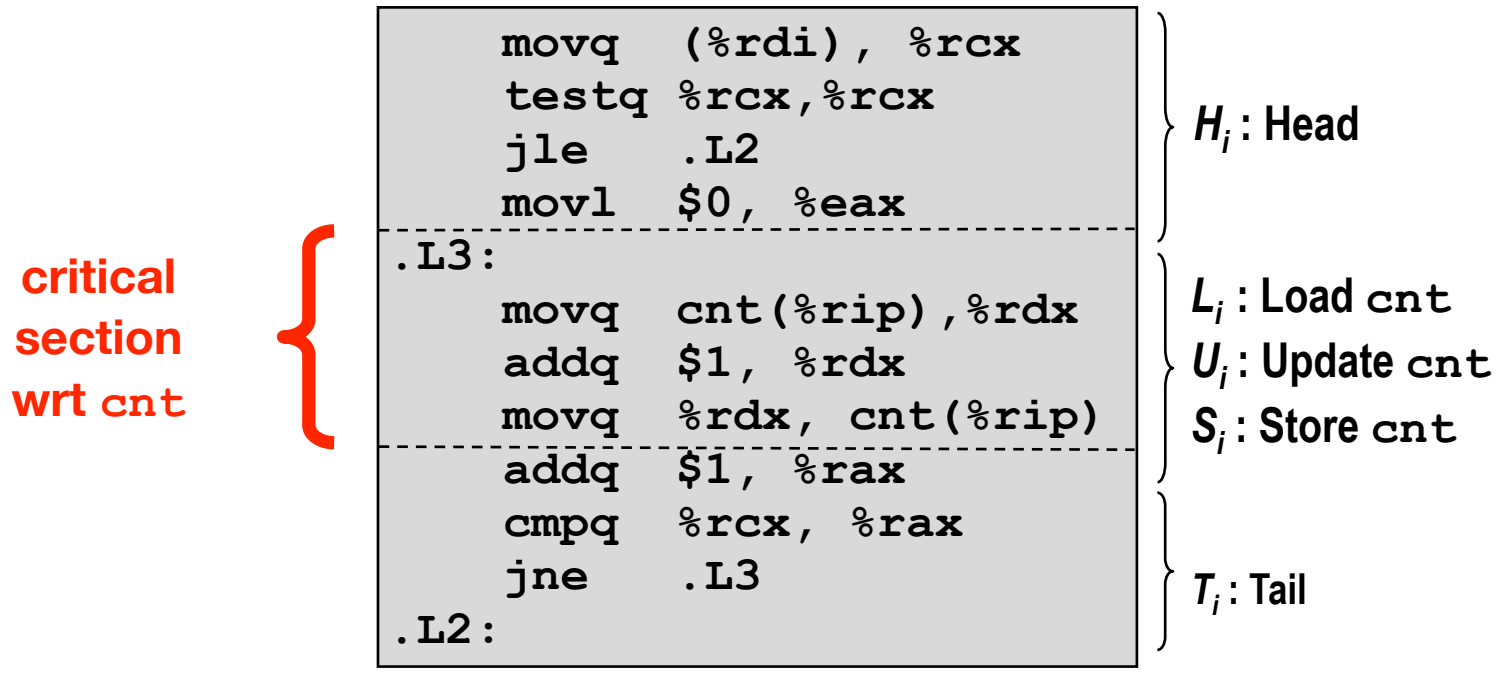
Critical Section

- Code section (a sequence of instructions) where no more than one thread should be executing concurrently.
  - Critical section refers to code, but its intention is to protect data!



Critical Section

- Code section (a sequence of instructions) where no more than one thread should be executing concurrently.
  - Critical section refers to code, but its intention is to protect data!
- Threads need to have ***mutually exclusive*** access to critical section. That is, the execution of the critical section must be ***atomic***: instructions in a CS either are executed entirely without interruption or not executed at all.



Enforcing Mutual Exclusion

- We must *coordinate*/*synchronize* the execution of the threads
  - i.e., need to guarantee ***mutually exclusive access*** for each critical section.

- Classic solution:
  - Semaphores/mutex (Edsger Dijkstra)

- Other approaches
  - Condition variables
  - Monitors (Java)
  - 254/258 discusses these

Using Semaphores for Mutual Exclusion

- Basic idea:

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
  - No more than one thread can be in the critical section at a time.

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
  - No more than one thread can be in the critical section at a time.
- Terminology

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
  - No more than one thread can be in the critical section at a time.
- Terminology
  - **Binary semaphore** is also called **mutex** (i.e., the semaphore value could only be 0 or 1)

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
  - No more than one thread can be in the critical section at a time.
- Terminology
  - **Binary semaphore** is also called **mutex** (i.e., the semaphore value could only be 0 or 1)
  - Think of P operation as "**locking**", and V as "**unlocking**".

Proper Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;  /* Counter */
sem_t mutex;            /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with P and V:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Warning: It's orders of magnitude slower than `badcnt.c`.

Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.
  - Checking mutex value and setting its value must be an atomic unit: they either are performed entirely or not performed at all.

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

Problem?

- Wouldn't there be a problem when multiple threads access the mutex? How do we ensure exclusive accesses to mutex itself?
- Hardware MUST provide mechanisms for atomic accesses to the mutex variable.
  - Checking mutex value and setting its value must be an atomic unit: they either are performed entirely or not performed at all.
  - on x86: the atomic test-and-set instruction.

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

Deadlock

- Def: A process/thread is *deadlocked* if and only if it is waiting for a condition that will never be true
- General to concurrent/parallel programming (threads, processes)
- Typical Scenario
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!

Deadlocking With Semaphores

```c
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}

int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;

Avoiding Deadlock

Acquire shared resources in same order

```
Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s1);
```

```
Tid[1]:
P(s1);
P(s0);
cnt++;
V(s1);
V(s0);
```

```
Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s1);
```

```
Tid[1]:
P(s0);
P(s1);
cnt++;
V(s1);
V(s0);
```

Summary of Multi-threading Programming

- Concurrent/parallel threads access shared variables
- Need to protect concurrent accesses to guarantee correctness
- Semaphores (e.g., mutex) provide a simple solution
- Can lead to deadlock if not careful
- Take CSC 254/258 to know more about avoiding deadlocks (and parallel programming in general)

Thread-level Parallelism (TLP)

- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task

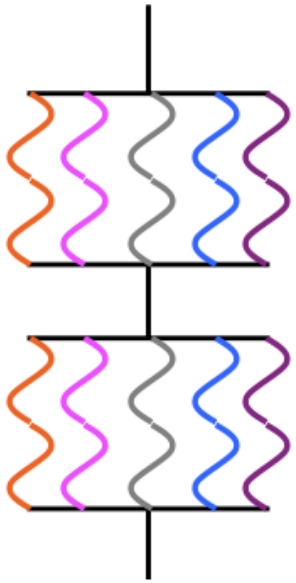
Thread-level Parallelism (TLP)

- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task
- Example: Parallel summation of N number
  - Partition values *0, …, n-1* into *t* ranges, ⌊n/t⌋ values each range
  - Each of t threads processes one range (sub-task)
  - Sum all sub-sums in the end

Thread-level Parallelism (TLP)

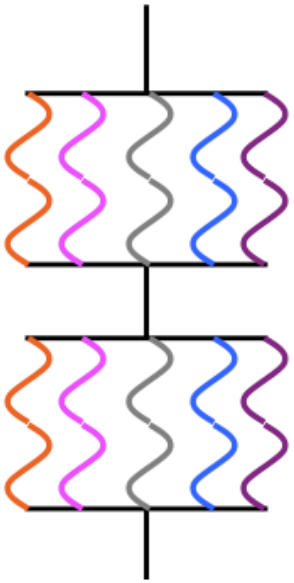
- Thread-Level Parallelism
  - Splitting a task into independent sub-tasks
  - Each thread is responsible for a sub-task
- Example: Parallel summation of N number
  - Partition values *0, …, n-1* into *t* ranges, ⌊n/t⌋ values each range
  - Each of t threads processes one range (sub-task)
  - Sum all sub-sums in the end
- Question: if you parallel you work N ways, do you always an N times speedup?

Why the Sequential Bottleneck?



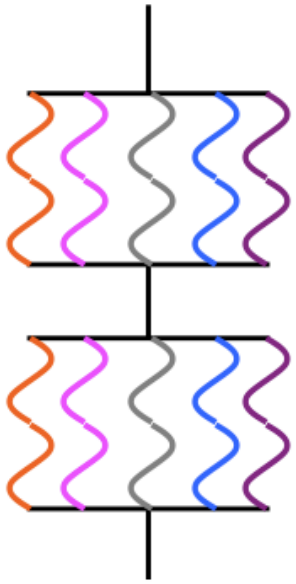
- Maximum speedup limited by the sequential portion
- Main cause: Non-parallelizable operations on data

Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: Non-parallelizable operations on data
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

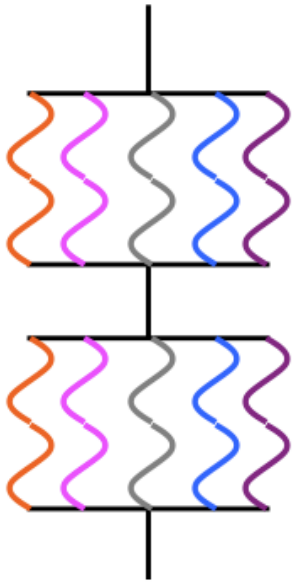
Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: Non-parallelizable operations on data
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

```
Each thread:
    loop {
        Compute
        P(A)
            Update shared data
        V(A)
     }
```

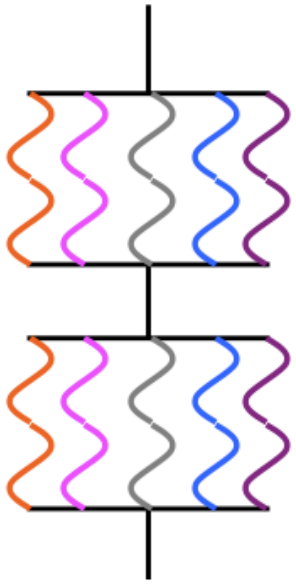
Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: Non-parallelizable operations on data
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

```
Each thread:
    loop {
      Compute                    N
      P(A)
          Update shared data
      V(A)
    }
```

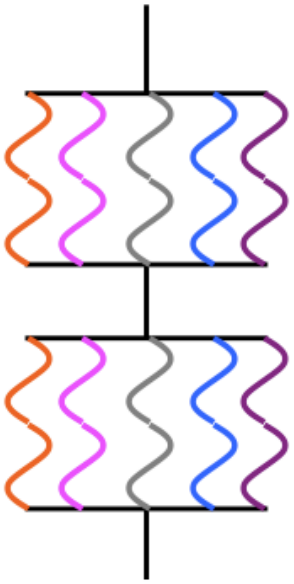
Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: Non-parallelizable operations on data
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

```
Each thread:
    loop {
      Compute                    N
      P(A)
          Update shared data
      V(A)                       C
    }
```

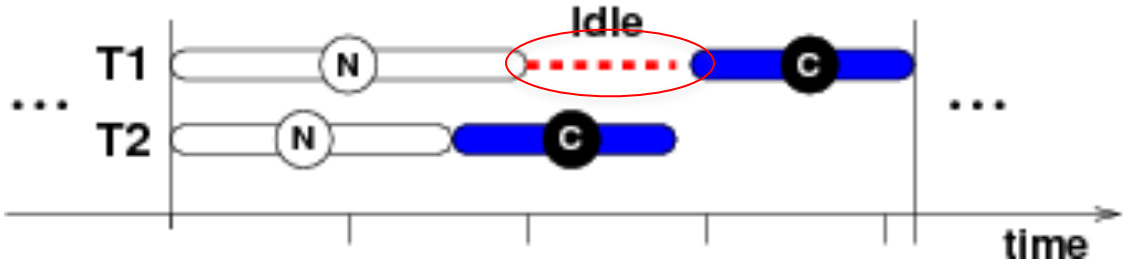
Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: Non-parallelizable operations on data
- Parallel portion is usually not perfectly parallel as well
  - e.g., Synchronization overhead

Each thread:

```
loop {
    Compute                    N
    P(A)
        Update shared data
    V(A)                       C
}
```



Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

```
1 - f
```

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$1 - f \quad +$$

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\frac{1}{1 - f + \frac{f}{N}}$$

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable (f = 1): Speedup = N

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable (f = 1): Speedup = N
- Completely sequential (f = 0): Speedup = 1

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable (f = 1): Speedup = N
- Completely sequential (f = 0): Speedup = 1
- Mostly parallelizable (f = 0.9, **N = 1000**): **Speedup = 9.9**

Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

Today

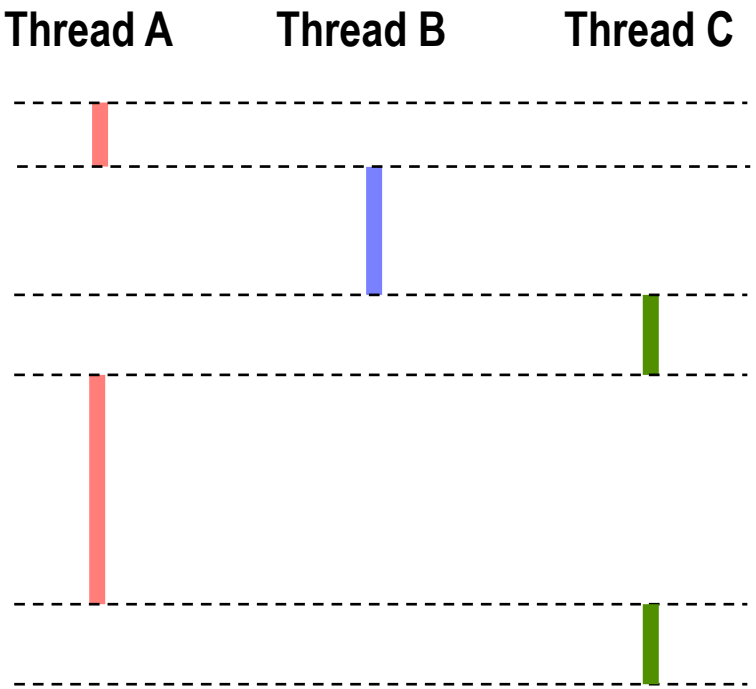
- From process to threads
  - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
  - Single core
  - Multi-core
  - Cache coherence

Can A Single Core Support Multi-threading?

- Need to multiplex between different threads (time slicing)

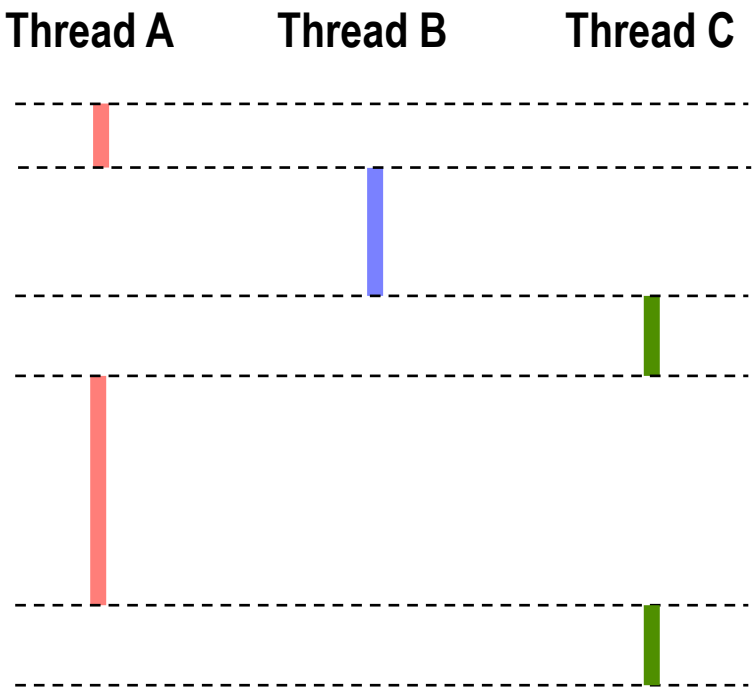
Sequential

Multi-threaded



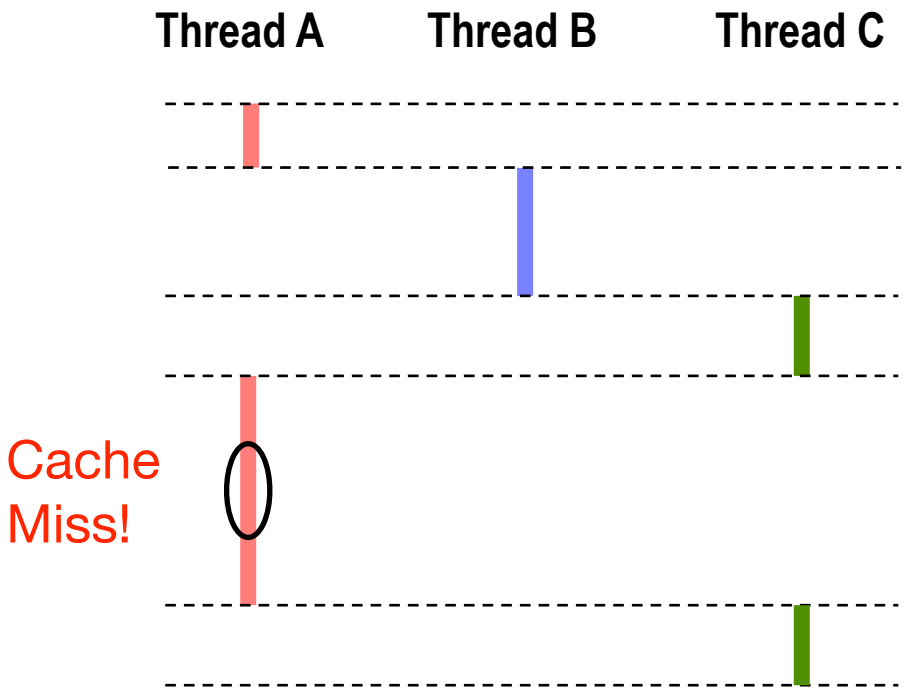
Any benefits?

- Can single-core multi-threading provide any performance gains?



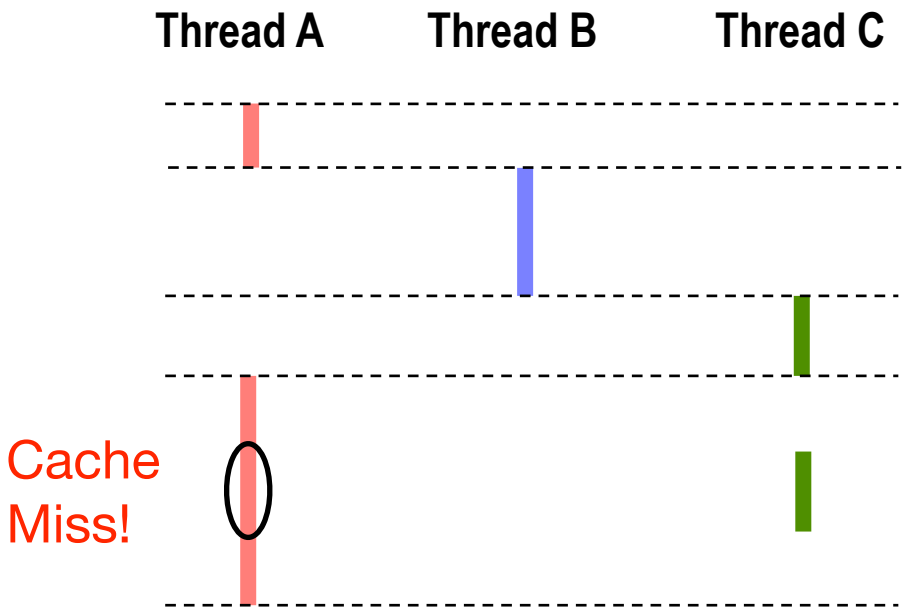
Any benefits?

- Can single-core multi-threading provide any performance gains?



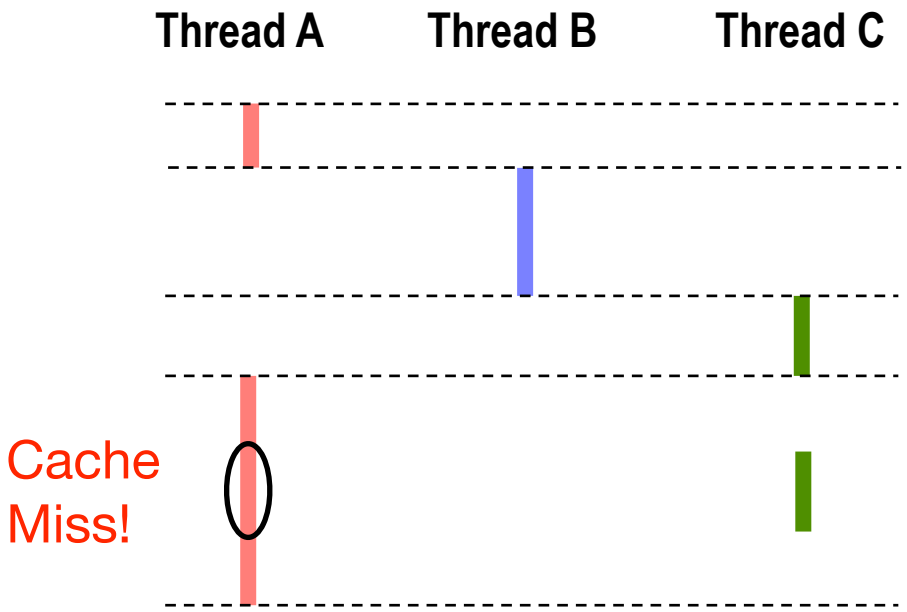
Any benefits?

- Can single-core multi-threading provide any performance gains?



Any benefits?

- Can single-core multi-threading provide any performance gains?
- If Thread A has a cache miss and the pipeline gets stalled, switch to Thread C. Improves the overall performance.



When to Switch?

- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)

When to Switch?

- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)
- Fine grained
  - Cycle by cycle
  - Thornton, “CDC 6600: Design of a Computer,” 1970.
  - Burton Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978. The HEP machine. A seminal paper that shows that using multi-threading can avoid branch prediction.

When to Switch?

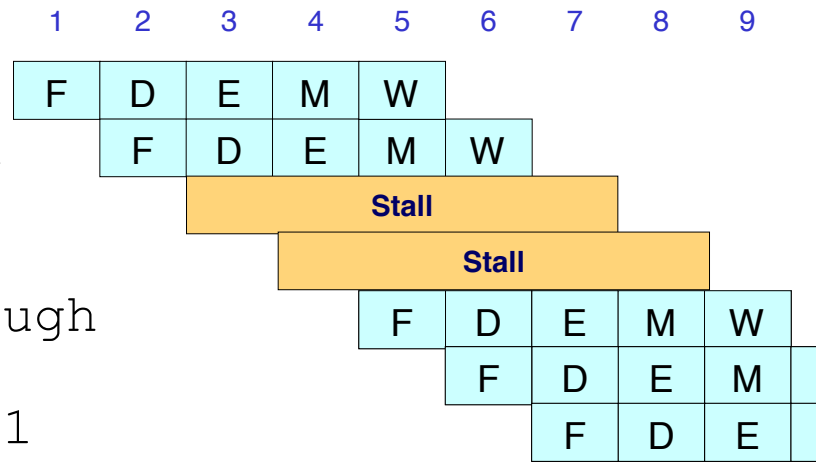
- Coarse grained
  - Event based, e.g., switch on L3 cache miss
  - Quantum based (every thousands of cycles)
- Fine grained
  - Cycle by cycle
  - Thornton, "CDC 6600: Design of a Computer," 1970.
  - Burton Smith, "A pipelined, shared resource MIMD computer," ICPP 1978. The HEP machine. A seminal paper that shows that using multi-threading can avoid branch prediction.
- Either way, need to save/restore thread context upon switching.

Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

The stalling approach

```
     xorg %rax, %rax
     jne L1                 # Not taken
     Stall
     Stall
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```

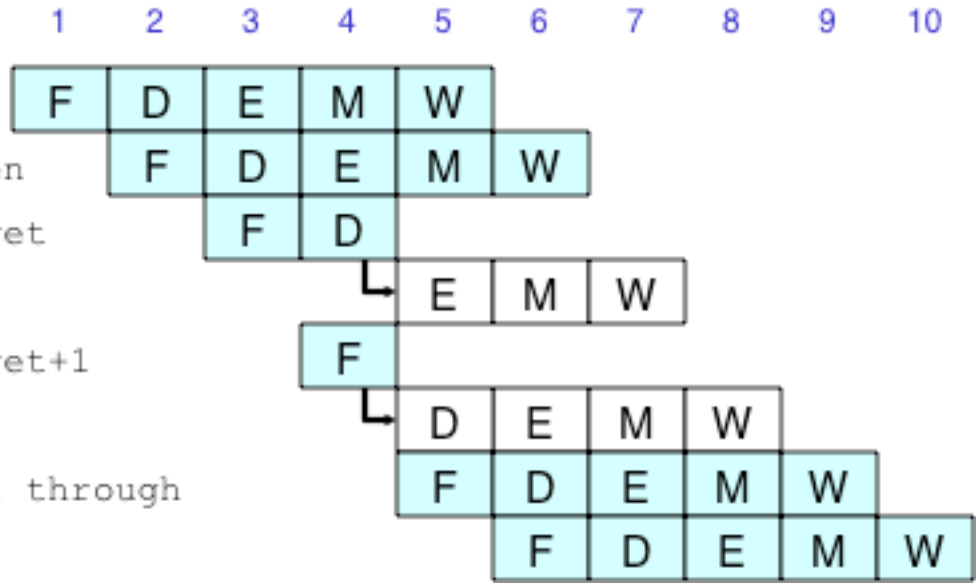


Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

The branch prediction approach

```
# demo-j.ys
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovq $2,%rdx # Target
       bubble
0x020: irmovq $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

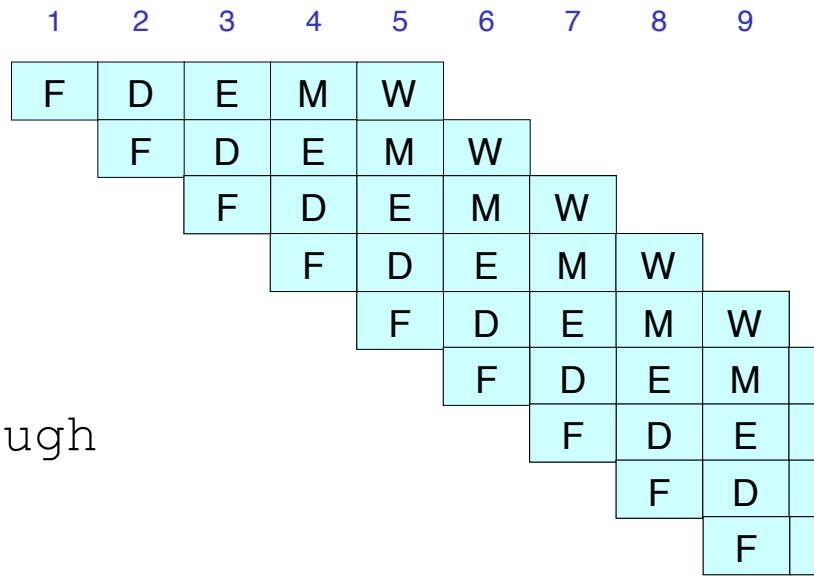


Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1                  # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax         # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
… …
```

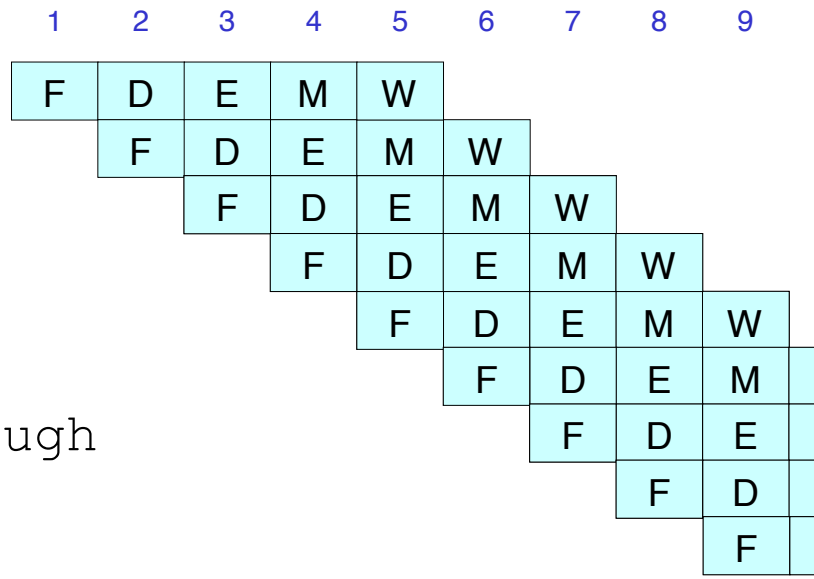


Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!
  - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).

The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1                  # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax         # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
… …
```



Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!
  - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).
  - GPUs do this (among other things).

The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1                  # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax         # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
… …
```