

# **CSC 252: Computer Organization**

## **Spring 2023: Lecture 24**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester



# Announcements

- Virtual Memory problem set: <https://www.cs.rochester.edu/courses/252/spring2023/handouts.html>
  - Not to be turned in. Won't be graded.
- Assignment 5 due April 21.

9	10	11	12 <b>Today</b>	13	14	15
16	17	18	19	20	21 <b>Due</b>	22
23	24	25	26	27	28	29
30	May 1 <b>Final</b>	2	3	4	5	6



# Explicit/Implicit Memory Management

- So far we have been talking about explicitly memory management: programmers explicitly calling malloc/free (C/C++)



# Explicit/Implicit Memory Management

- So far we have been talking about explicitly memory management: programmers explicitly calling malloc/free (C/C++)
- Downside: potential memory leaks

```
void foo() {  
    int *p = malloc(128);  
    p = malloc(32);  
    return; /* both blocks are now garbage */  
}
```



# Explicit/Implicit Memory Management

- So far we have been talking about explicitly memory management: programmers explicitly calling malloc/free (C/C++)
- Downside: potential memory leaks

```
void foo() {  
    int *p = malloc(128);  
    p = malloc(32);  
    return; /* both blocks are now garbage */  
}
```

- Alternative: implicit memory management; the programmers never explicitly request/free memory



# Explicit/Implicit Memory Management

- So far we have been talking about explicitly memory management: programmers explicitly calling malloc/free (C/C++)
- Downside: potential memory leaks

```
void foo() {  
    int *p = malloc(128);  
    p = malloc(32);  
    return; /* both blocks are now garbage */  
}
```

- Alternative: implicit memory management; the programmers never explicitly request/free memory
- Common in many dynamic languages:
  - Python, Ruby, Java, JavaScript, Perl, ML, Lisp, Mathematica



# Explicit/Implicit Memory Management

- So far we have been talking about explicitly memory management: programmers explicitly calling malloc/free (C/C++)
- Downside: potential memory leaks

```
void foo() {  
    int *p = malloc(128);  
    p = malloc(32);  
    return; /* both blocks are now garbage */  
}
```

- Alternative: implicit memory management; the programmers never explicitly request/free memory
- Common in many dynamic languages:
  - Python, Ruby, Java, JavaScript, Perl, ML, Lisp, Mathematica
- The key: **Garbage collection**
  - Automatic reclamation of heap-allocated storage—application never has to free



# Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?



# Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?
  - If a block will never be used in the future. How do we know that?



# Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?
  - If a block will never be used in the future. How do we know that?
  - In general we cannot know what is going to be used in the future since it depends on program's future behaviors



# Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?
  - If a block will never be used in the future. How do we know that?
  - In general we cannot know what is going to be used in the future since it depends on program's future behaviors
  - But we can tell that certain blocks cannot possibly be used ***if there are no pointers to them***



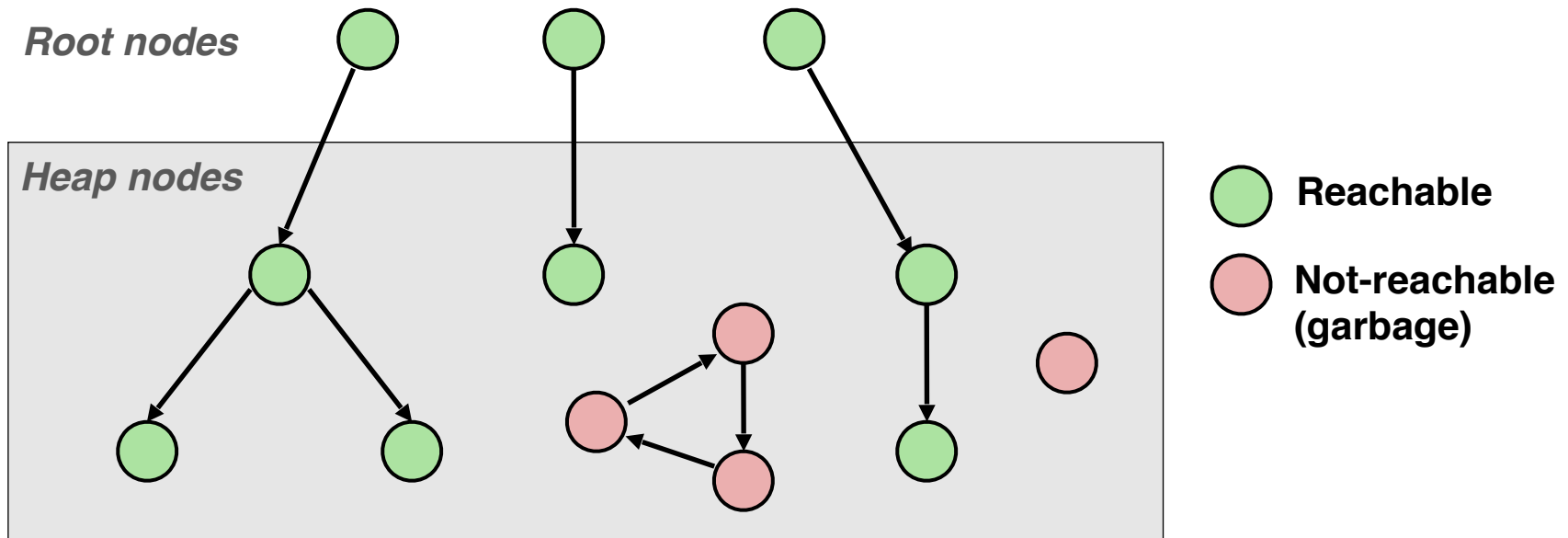
# Garbage Collection

- How does the memory manager know when certain memory blocks can be freed?
  - If a block will never be used in the future. How do we know that?
  - In general we cannot know what is going to be used in the future since it depends on program's future behaviors
  - But we can tell that certain blocks cannot possibly be used ***if there are no pointers to them***
  - Garbage collection is essentially to obtain all **reachable** blocks and discard unreachable blocks.



# Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called **root** nodes (e.g. registers, locations on the stack, global variables)



A node (block) is **reachable** if there is a path from any root to that node.

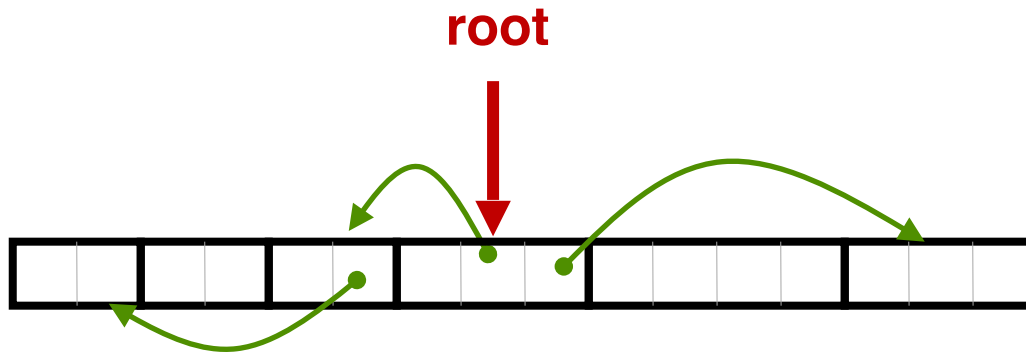
Non-reachable nodes are **garbage** (cannot be needed by the application)



# Mark and Sweep Collecting

- Idea:

- Use extra **mark bit** in the header to indicate if a block is reachable
- **Mark**: Start at roots and set mark bit on each reachable block
- **Sweep**: Scan all blocks and free blocks that are not marked



*Note: arrows here denote memory refs, not free list ptrs.*

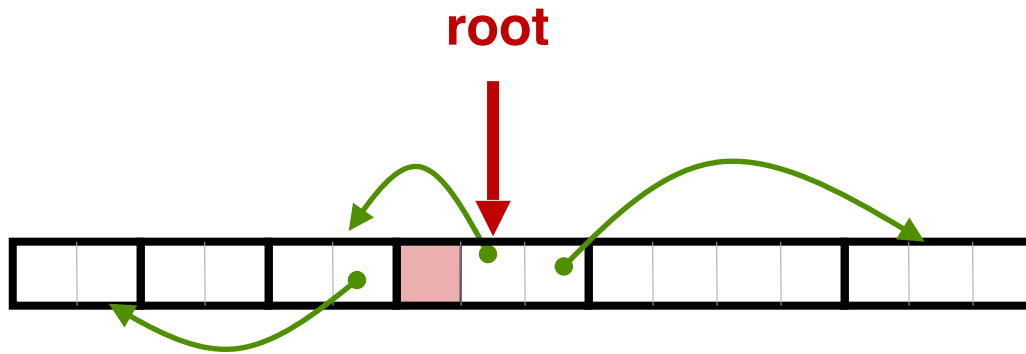
 **Mark bit set**



# Mark and Sweep Collecting

- Idea:

- Use extra **mark bit** in the header to indicate if a block is reachable
- **Mark**: Start at roots and set mark bit on each reachable block
- **Sweep**: Scan all blocks and free blocks that are not marked



*Note: arrows here denote memory refs, not free list ptrs.*

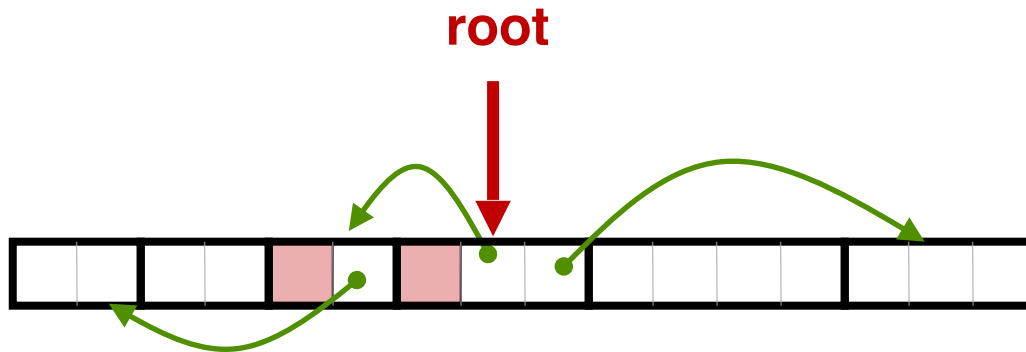
 **Mark bit set**



# Mark and Sweep Collecting

- Idea:

- Use extra **mark bit** in the header to indicate if a block is reachable
- **Mark**: Start at roots and set mark bit on each reachable block
- **Sweep**: Scan all blocks and free blocks that are not marked



*Note: arrows here  
denote memory refs,  
not free list ptrs.*

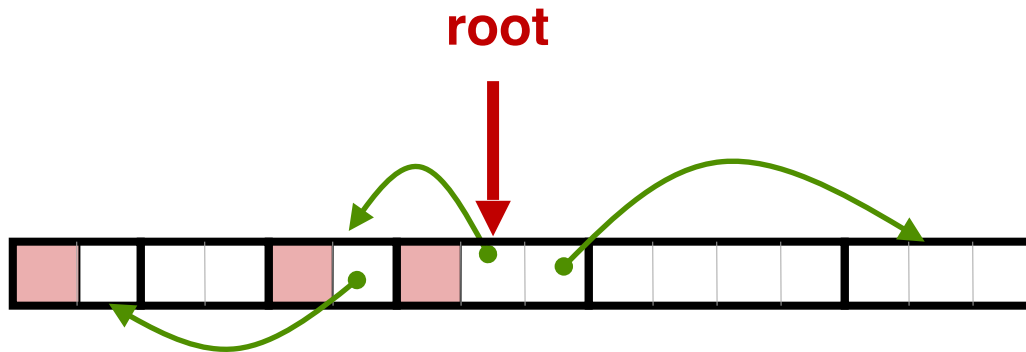
 **Mark bit set**



# Mark and Sweep Collecting

- Idea:

- Use extra **mark bit** in the header to indicate if a block is reachable
- **Mark**: Start at roots and set mark bit on each reachable block
- **Sweep**: Scan all blocks and free blocks that are not marked



*Note: arrows here  
denote memory refs,  
not free list ptrs.*

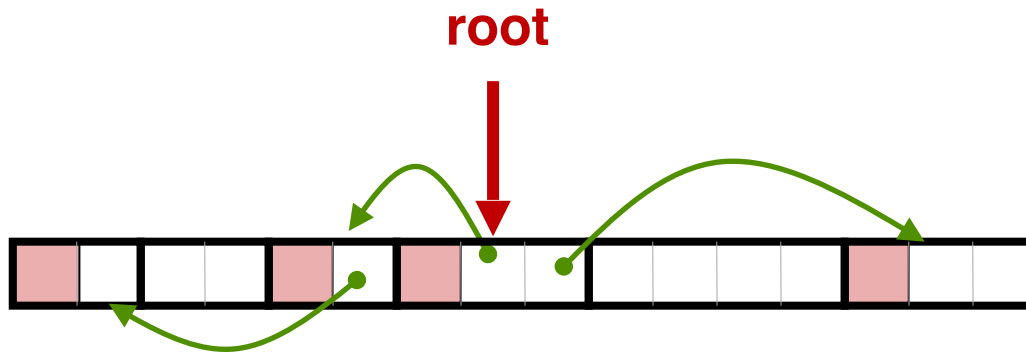
 **Mark bit set**



# Mark and Sweep Collecting

- Idea:

- Use extra **mark bit** in the header to indicate if a block is reachable
- **Mark**: Start at roots and set mark bit on each reachable block
- **Sweep**: Scan all blocks and free blocks that are not marked



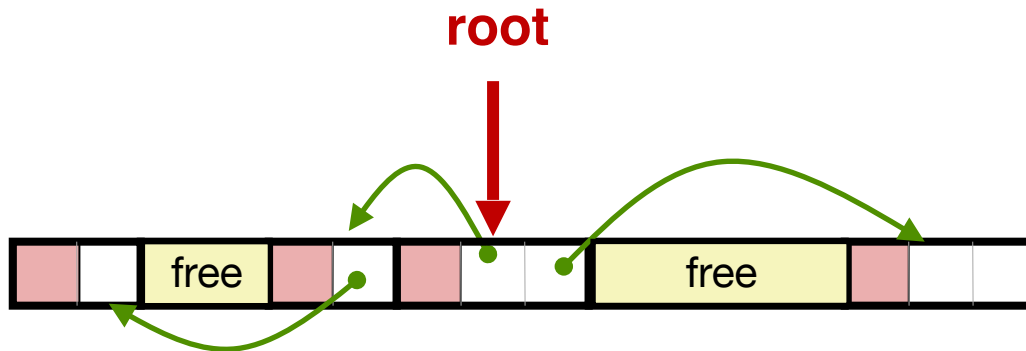
*Note: arrows here  
denote memory refs,  
not free list ptrs.*

 **Mark bit set**



# Mark and Sweep Collecting

- Idea:
  - Use extra **mark bit** in the header to indicate if a block is reachable
  - **Mark:** Start at roots and set mark bit on each reachable block
  - **Sweep:** Scan all blocks and free blocks that are not marked



*Note: arrows here  
denote memory refs,  
not free list ptrs.*

 **Mark bit set**



# Mark and Sweep (cont.)

## Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {  
    if (!is_ptr(p)) return;           // do nothing if not pointer  
    if (markBitSet(p)) return;        // check if already marked  
    setMarkBit(p);                    // set the mark bit  
    for (i=0; i < length(p); i++)    // call mark on all words  
        mark(p[i]);                  // in the block  
    return;  
}
```



# Mark and Sweep (cont.)

## Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {  
    if (!is_ptr(p)) return;           // do nothing if not pointer  
    if (markBitSet(p)) return;       // check if already marked  
    setMarkBit(p);                   // set the mark bit  
    for (i=0; i < length(p); i++)    // call mark on all words  
        mark(p[i]);                  // in the block  
    return;  
}
```

## Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {  
    while (p < end) {  
        if markBitSet(p)  
            clearMarkBit();  
        else if (allocateBitSet(p))  
            free(p);  
        p += length(p);  
    }  
}
```



# Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.



# Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.



# Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.



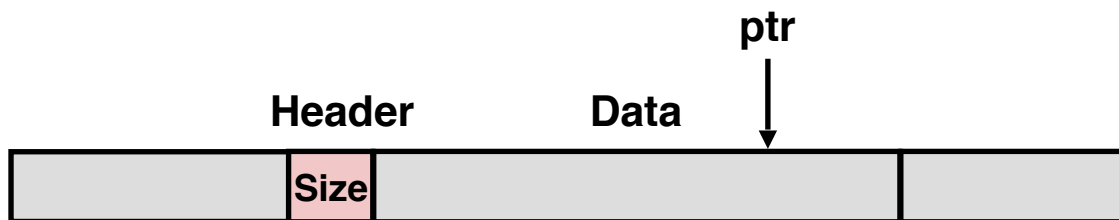
# Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?



# Conservative Mark & Sweep in C

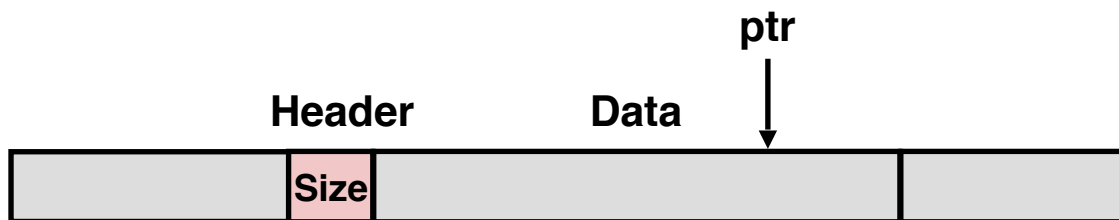
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?





# Conservative Mark & Sweep in C

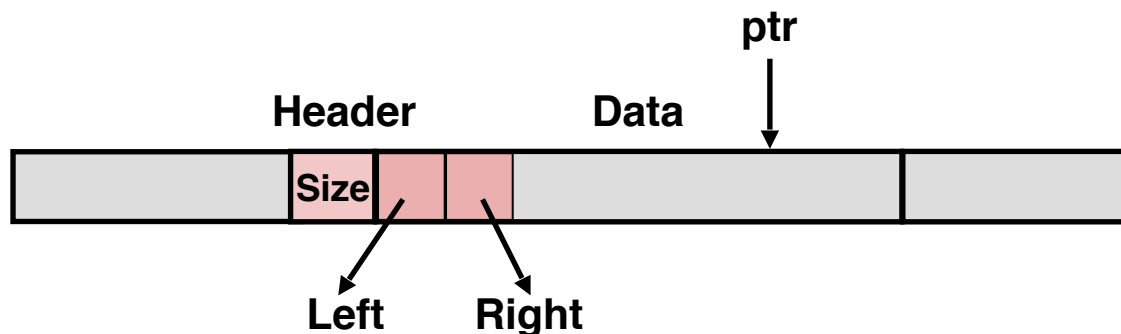
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)





# Conservative Mark & Sweep in C

- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.
- C pointers can point to the middle of a block. How do you find the header of a block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)

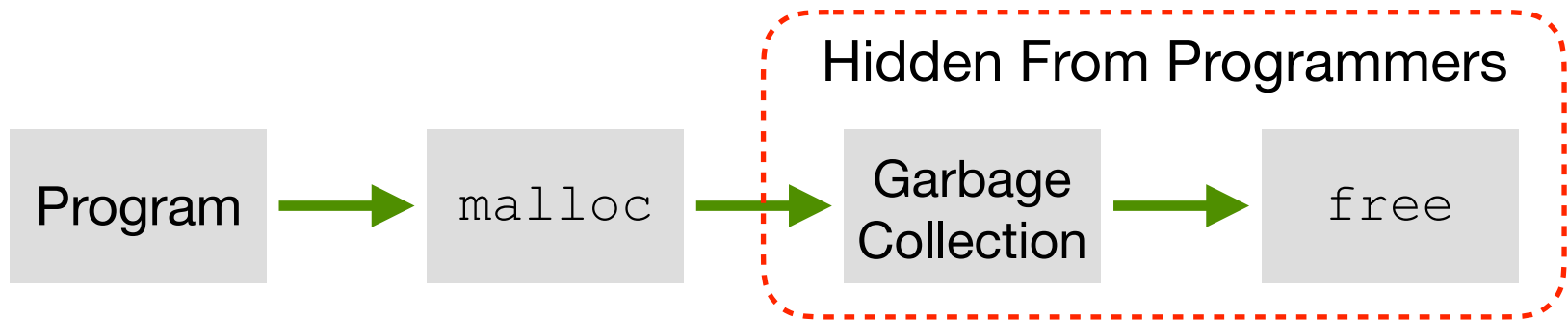


**Left:** smaller addresses  
**Right:** larger addresses



# Potential GC Implementations (in C)

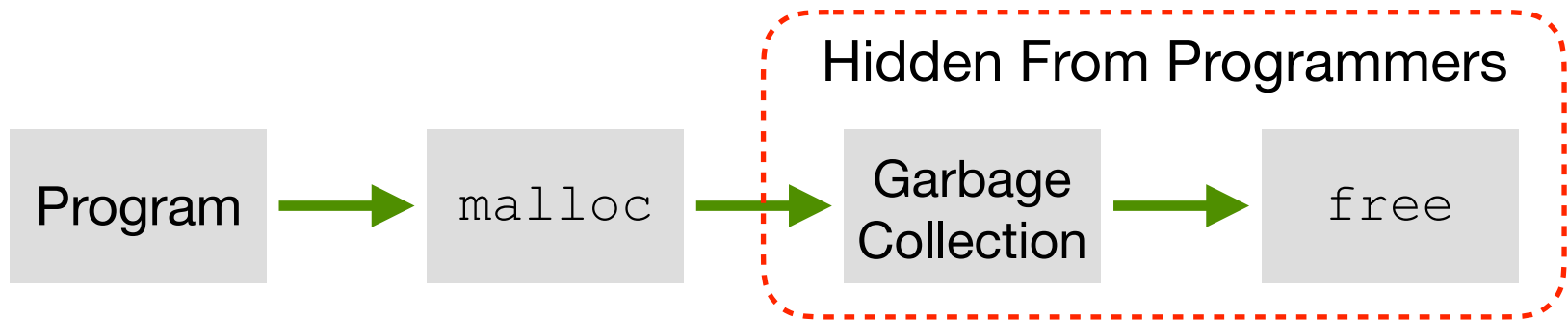
- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.





# Potential GC Implementations (in C)

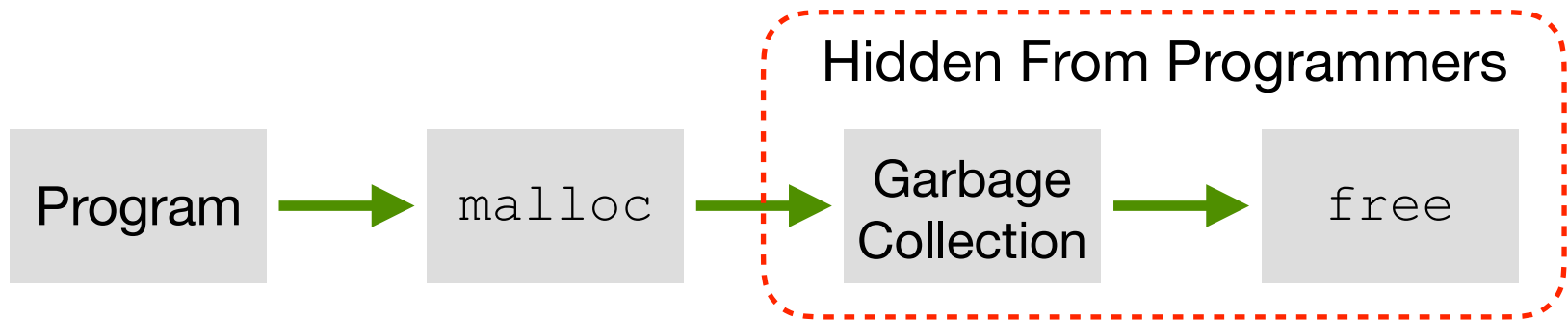
- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - **Stop-the-world GC**. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.





# Potential GC Implementations (in C)

- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - **Stop-the-world GC**. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.

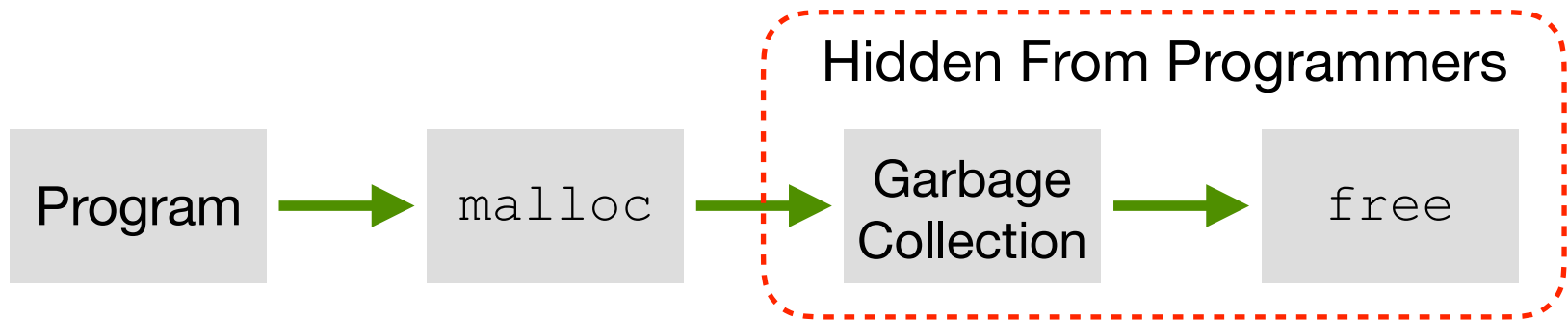


- To minimize main application (called mutator) pause time:



# Potential GC Implementations (in C)

- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - **Stop-the-world GC**. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.

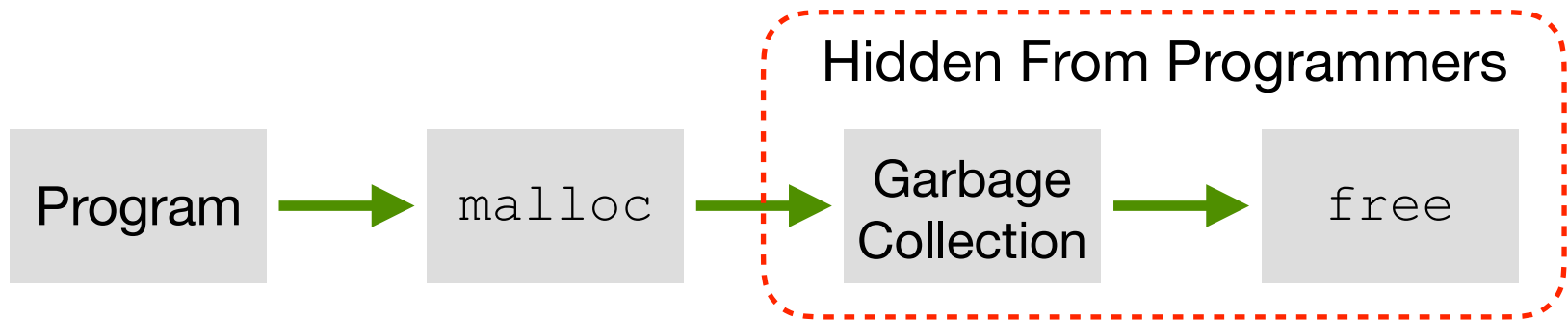


- To minimize main application (called mutator) pause time:
  - **Incremental GC**: Examine a small portion of heap every GC run



# Potential GC Implementations (in C)

- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - **Stop-the-world GC**. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



- To minimize main application (called mutator) pause time:
  - **Incremental GC**: Examine a small portion of heap every GC run
  - **Concurrent GC**: Run GC service in a separate process/thread



# Garbage Collection Implications

- GC is a great source of performance non-determinisms
  - Generally can't predict when GC will happen



# Garbage Collection Implications

- GC is a great source of performance non-determinisms
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive



# Garbage Collection Implications

- GC is a great source of performance non-determinisms
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts



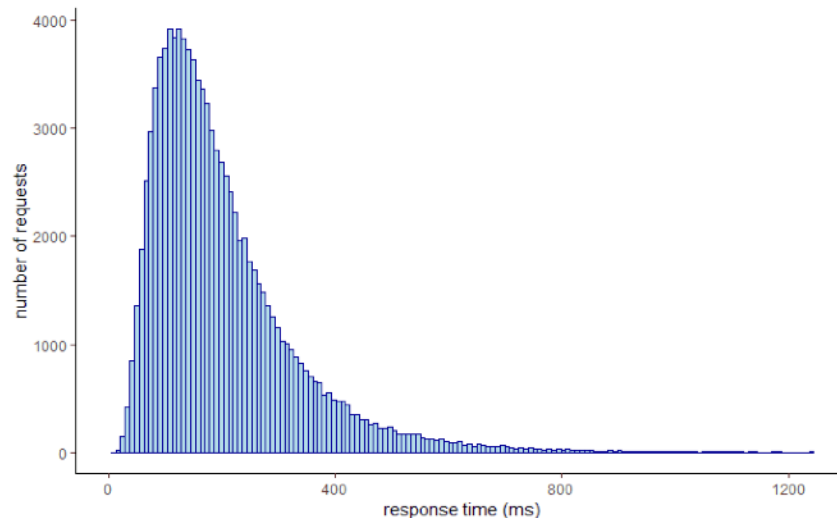
# Garbage Collection Implications

- GC is a great source of performance non-determinisms
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts
  - Bad for real-time systems: think of a self-driving car that needs to decide whether to avoid a pedestrian but a GC kicks in...



# Garbage Collection Implications

- GC is a great source of performance non-determinisms
  - Generally can't predict when GC will happen
  - Stop-the-world GC makes program periodically unresponsive
  - Concurrent/Incremental GC helps, but still has performance impacts
  - Bad for real-time systems: think of a self-driving car that needs to decide whether to avoid a pedestrian but a GC kicks in...
  - Bad for server/cloud systems: GC is a great source of *tail latency*





# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)



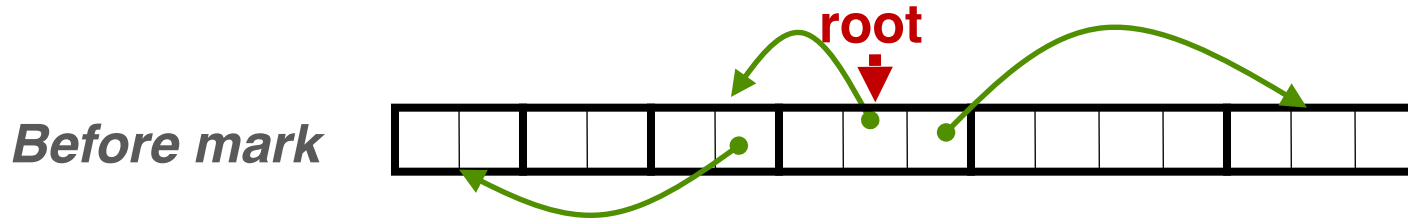
# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.



# Classical GC Algorithms

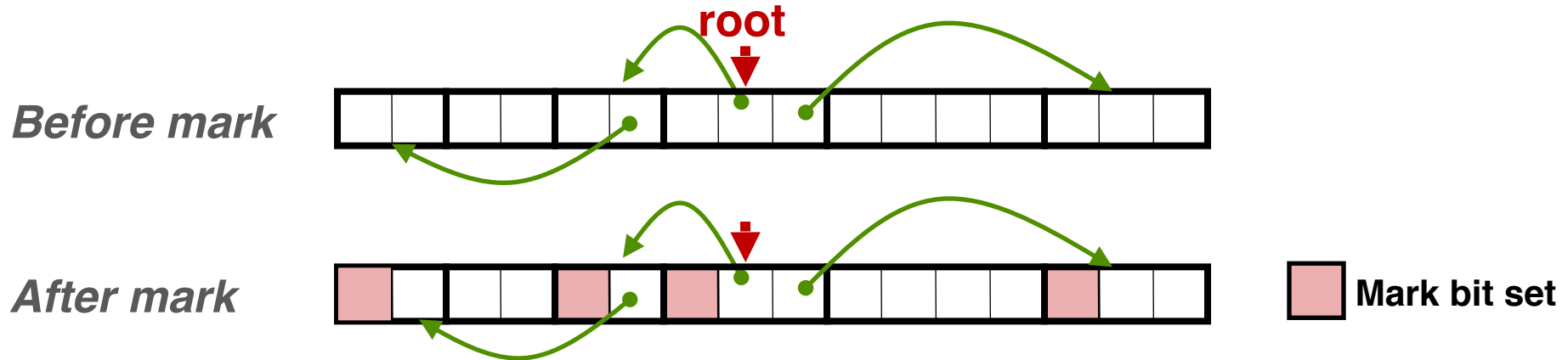
- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.





# Classical GC Algorithms

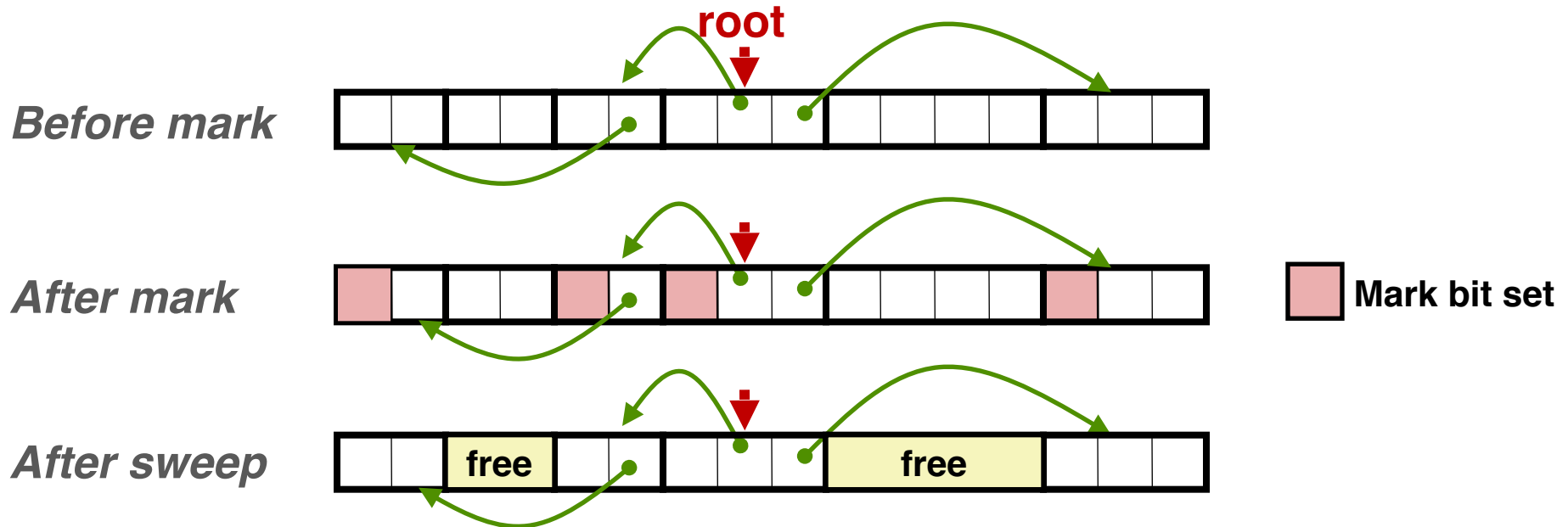
- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.





# Classical GC Algorithms

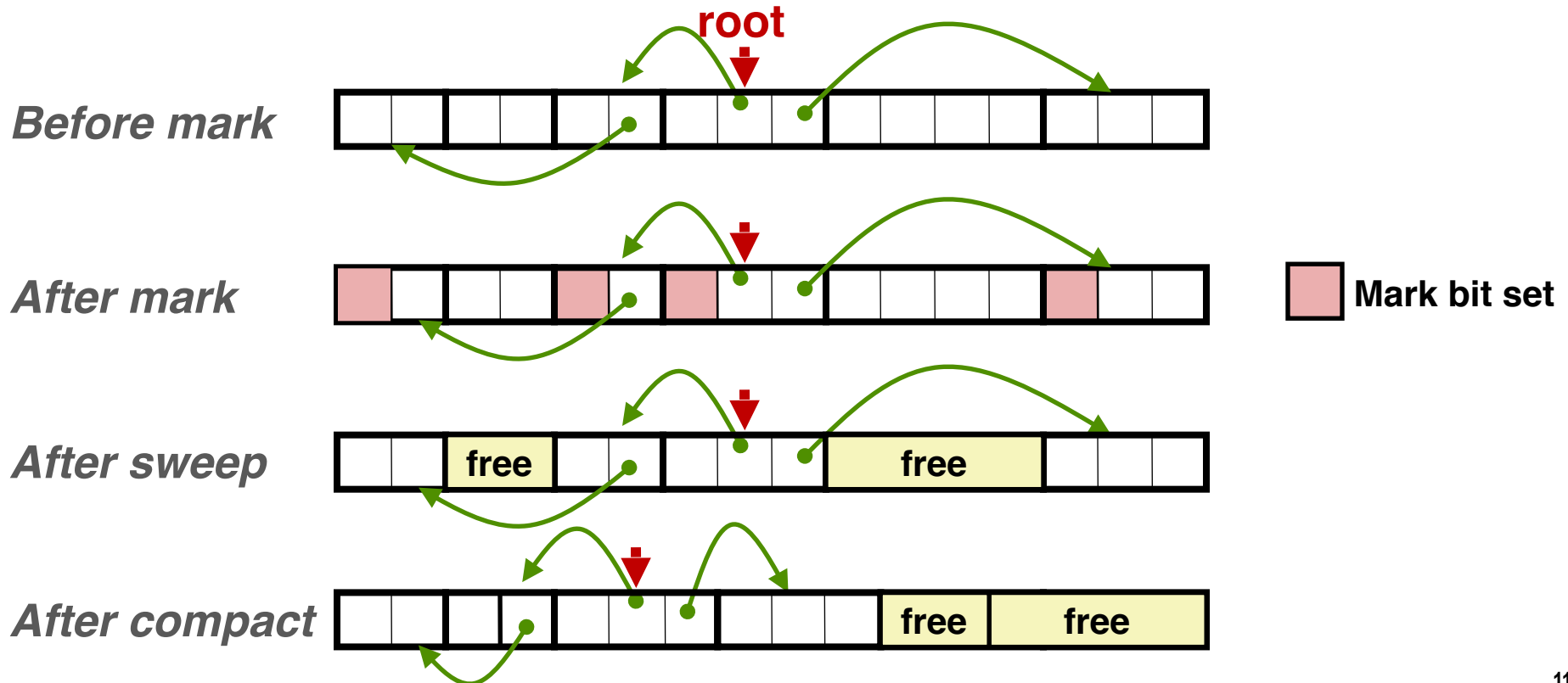
- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.





# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
  - After M&S, compact allocated blocks to consecutive memory region.
  - Reduce external fragmentation. Allocation is also easier.





# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)



# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.



# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)



# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon (“infant mortality”); others will survive for a long time.



# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon (“infant mortality”); others will survive for a long time.
  - Wasteful to scan long-lived objects every collection time



# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon (“infant mortality”); others will survive for a long time.
  - Wasteful to scan long-lived objects every collection time
  - Idea: divide heap into two generations, young and old. Allocate into young gen., and promote to old gen. if lived long enough. Collect young gen. more often than old gen.



# Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
- Mark-sweep-compact collection (Styger, 1967)
- Mark-copy collection (Minsky, 1963)
  - After mark, copy reachable objects to another region of memory as they are being traversed. Can be done without auxiliary storage.
- Generational Collectors (Lieberman and Hewitt, 1983)
  - Observation: most allocations become garbage very soon (“infant mortality”); others will survive for a long time.
  - Wasteful to scan long-lived objects every collection time
  - Idea: divide heap into two generations, young and old. Allocate into young gen., and promote to old gen. if lived long enough. Collect young gen. more often than old gen.
- **Question: Can any of these algorithms be used for GC in C?**

Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, 1996.



# Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.



# Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object



# Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses



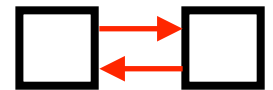
# Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing



# Classical GC Algorithms

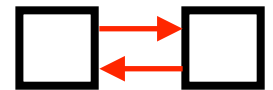
- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing





# Classical GC Algorithms

- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing to the object
  - Decrement the counter if a pointer is taken off the object
  - When the counter reaches zero, collect the object
- Advantages of Reference Counting
  - Simpler to implement
  - Collect garbage objects immediately; generally less long pauses
- Disadvantages of Reference Counting
  - A naive implementation can't deal with self-referencing
- A heterogeneous approach (RC + tracing) is often used





# Today

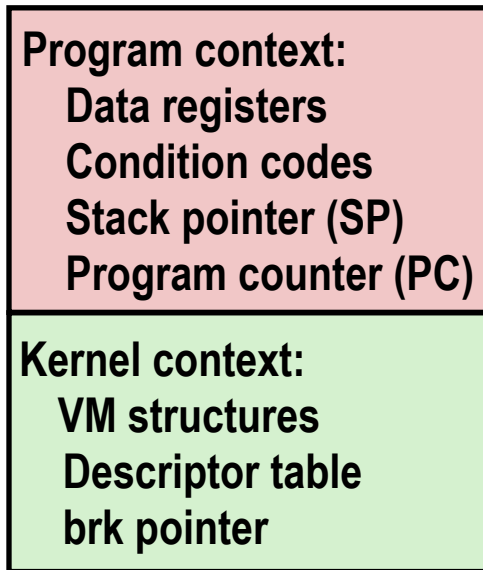
- From process to threads
  - Basic thread execution model
- Multi-threading programming
- Hardware support of threads
  - Single core
  - Multi-core
  - Hyper-threading
  - Cache coherence



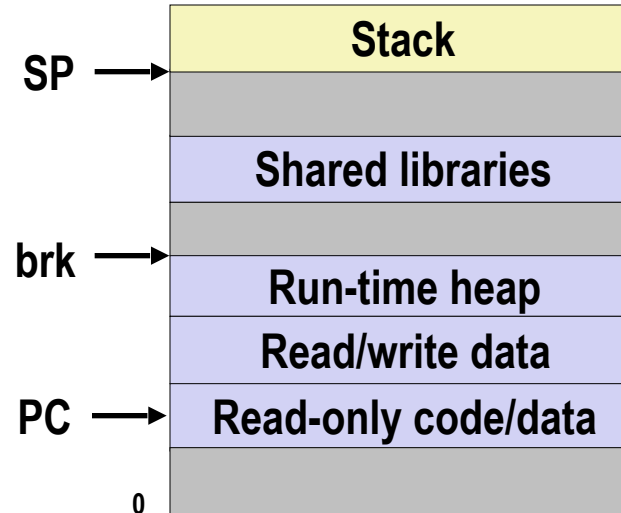
# Programmers View of A Process

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

## Process context



## 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)

**Thread 1 (main thread)**

**Thread 2 (peer thread)**

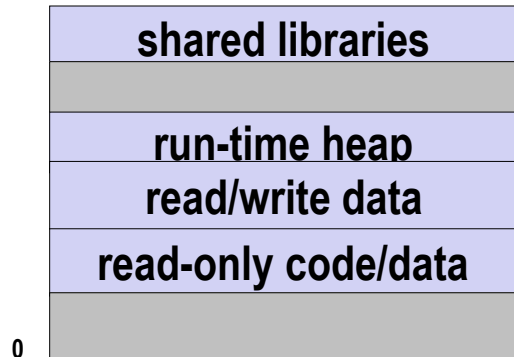
**Shared code and data**

**stack 1**

**stack 2**

Thread 1 context:  
Data registers  
Condition codes  
SP1  
PC1

Thread 2 context:  
Data registers  
Condition codes  
SP2  
PC2



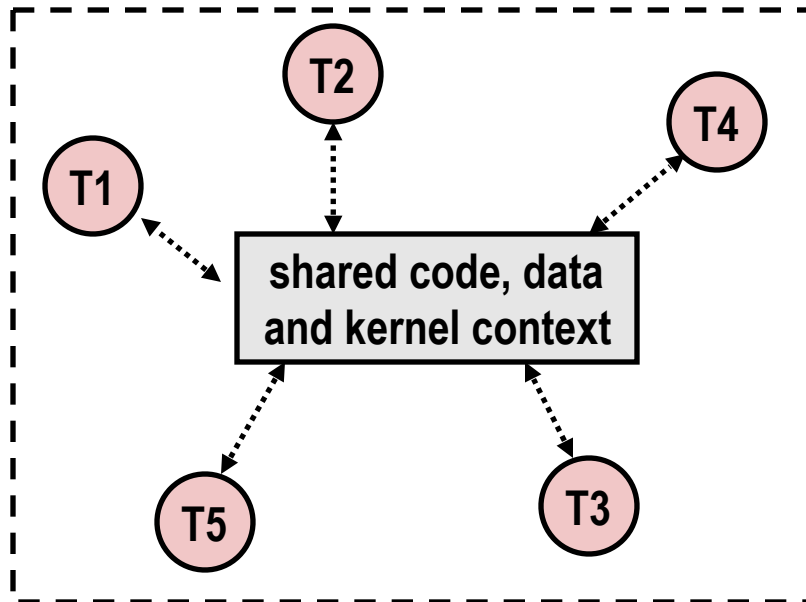
Kernel context:  
VM structures  
Descriptor table  
brk pointer



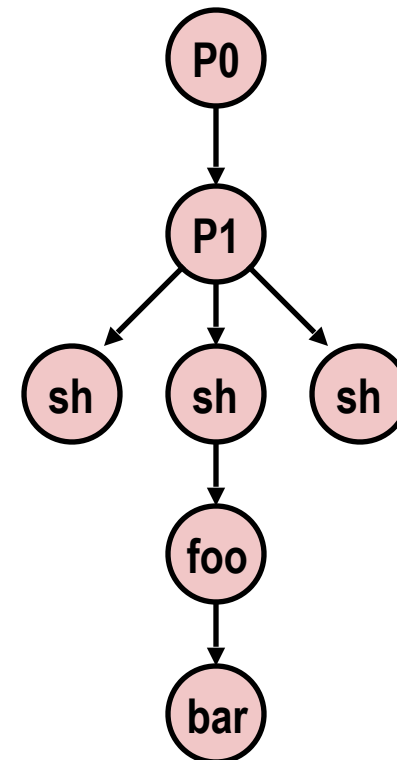
# Logical View of Threads

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

Threads associated with process foo



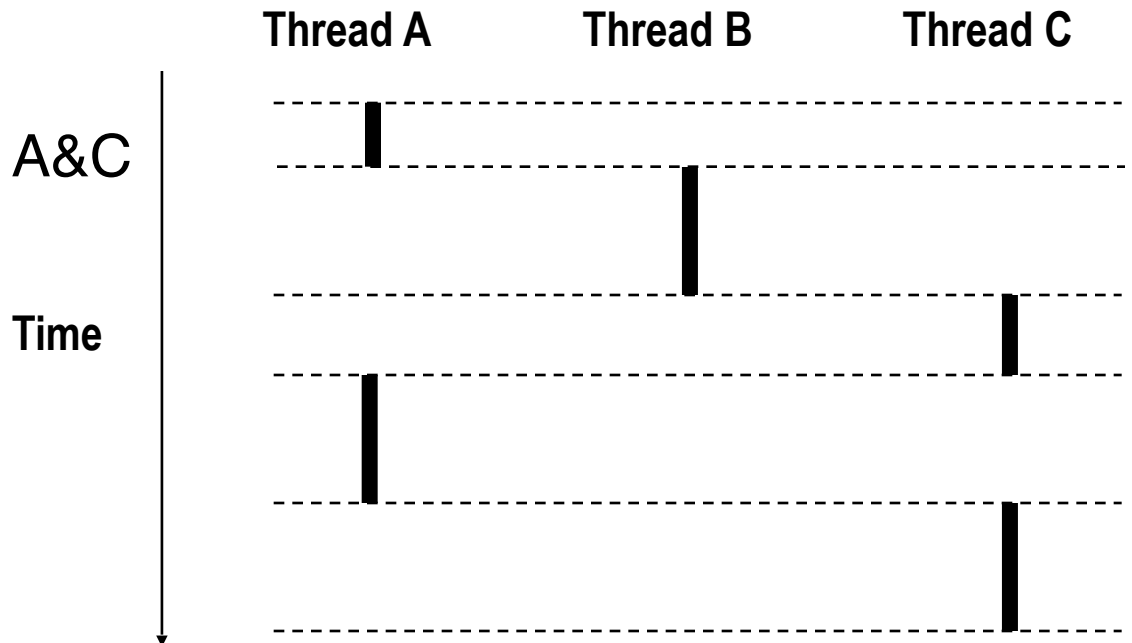
Process hierarchy





# Concurrent Threads

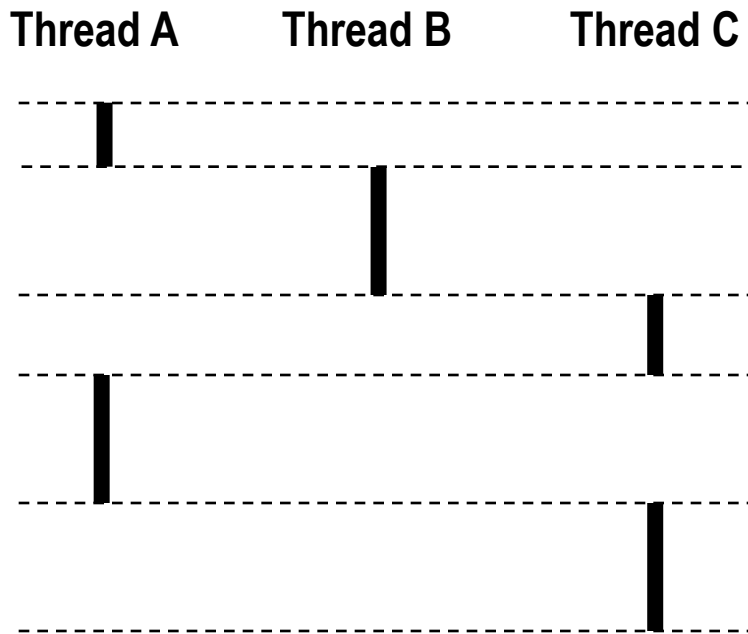
- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential
- Examples:
  - Concurrent: A & B, A&C
  - Sequential: B & C



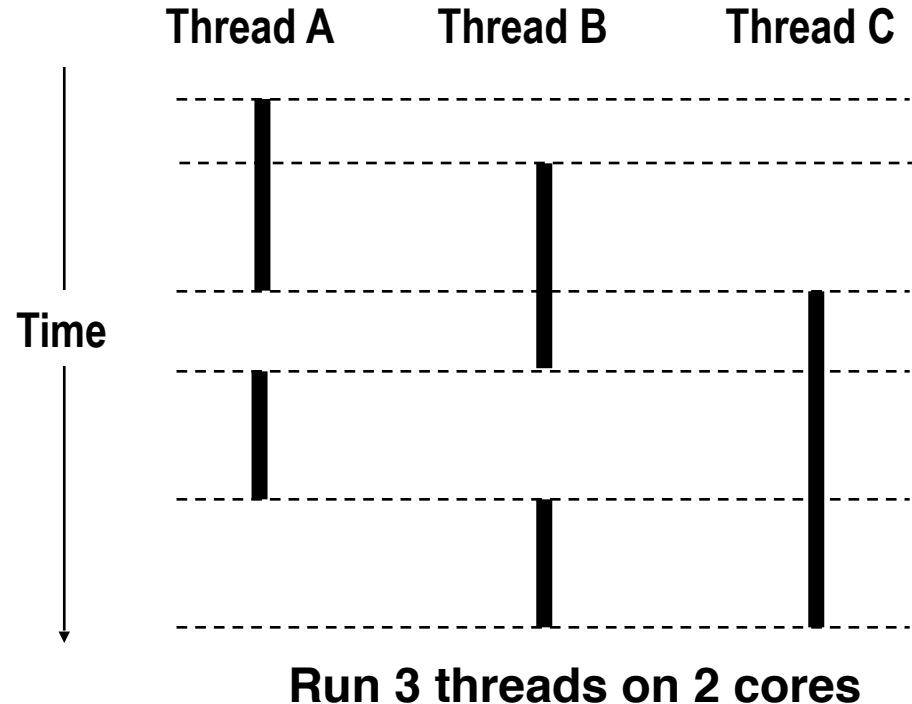


# Concurrent Thread Execution

- Single Core Processor
  - Simulate parallelism by time slicing



- Multi Core Processor
  - 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

```
/*
 * 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;
}
```

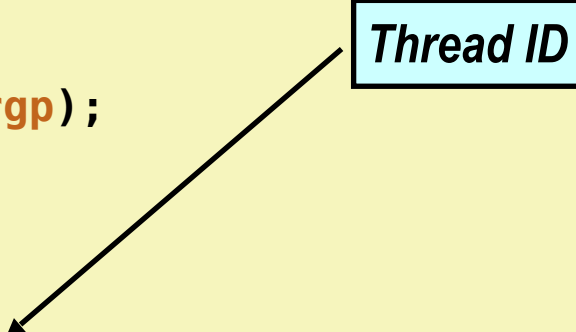
hello.c



# 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);  
}
```

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)

```
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

Thread attributes  
(usually NULL)

Thread routine

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

Thread attributes  
(usually NULL)

Thread routine

Thread arguments  
(void \*p)

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

Thread attributes  
(usually NULL)

Thread routine

Thread arguments  
(void \*p)

hello.c

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

hello.c

Return value  
(void \*\*p)



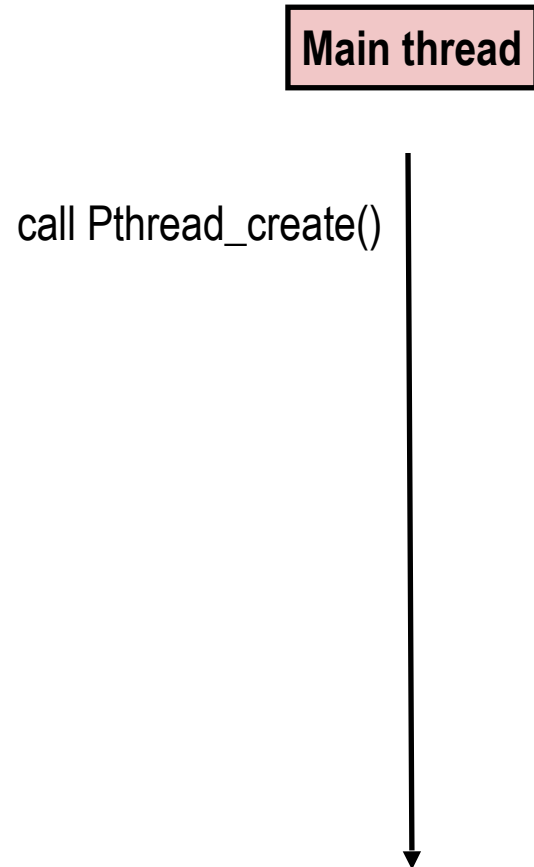
# Execution of Threaded “hello, world”

Main thread



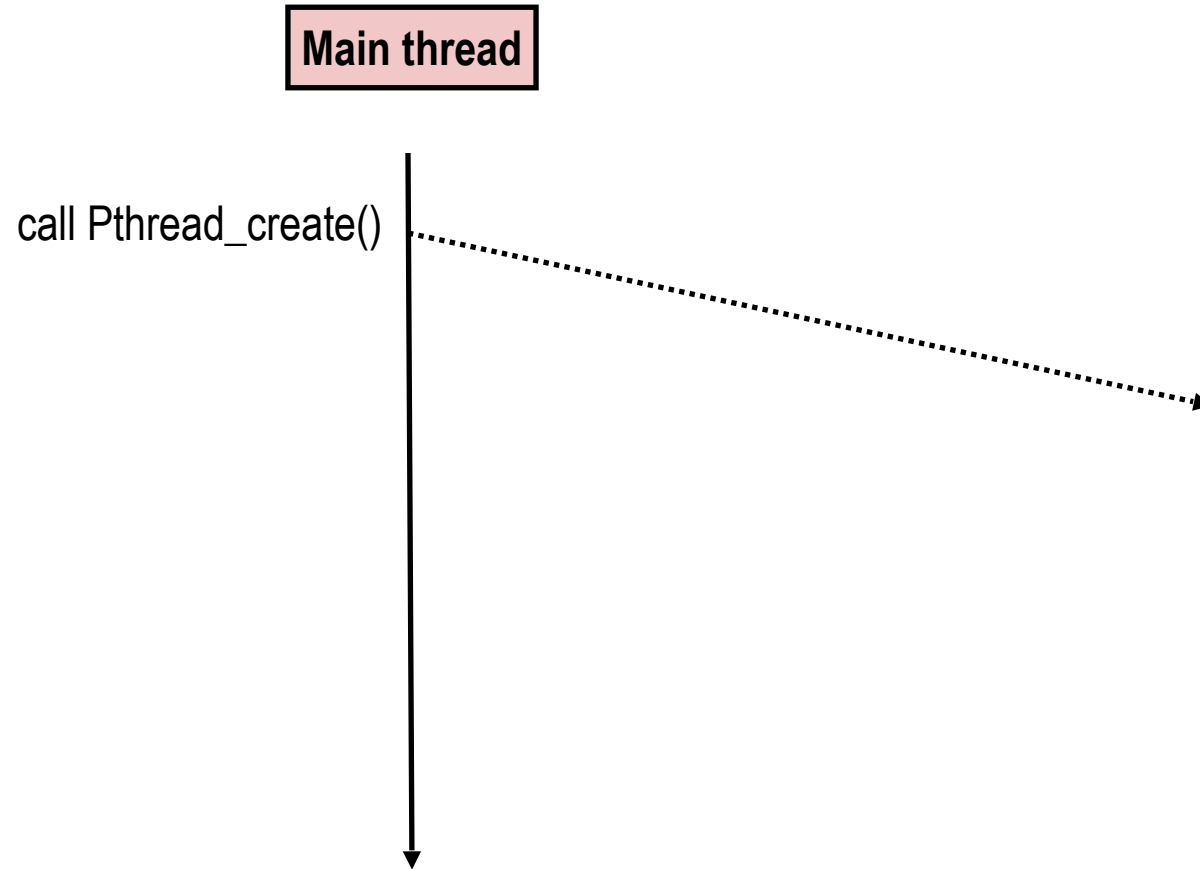


# Execution of Threaded “hello, world”



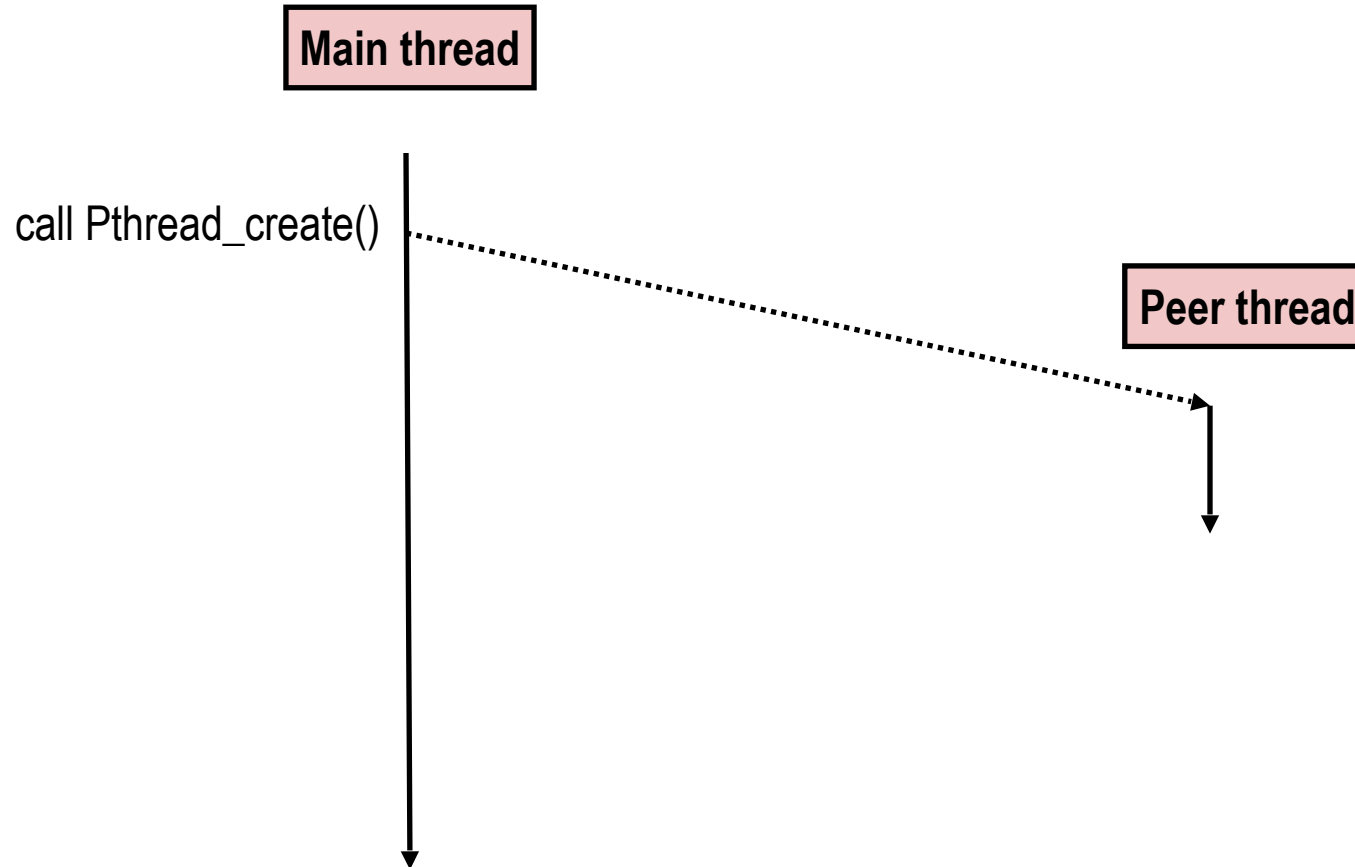


# Execution of Threaded “hello, world”



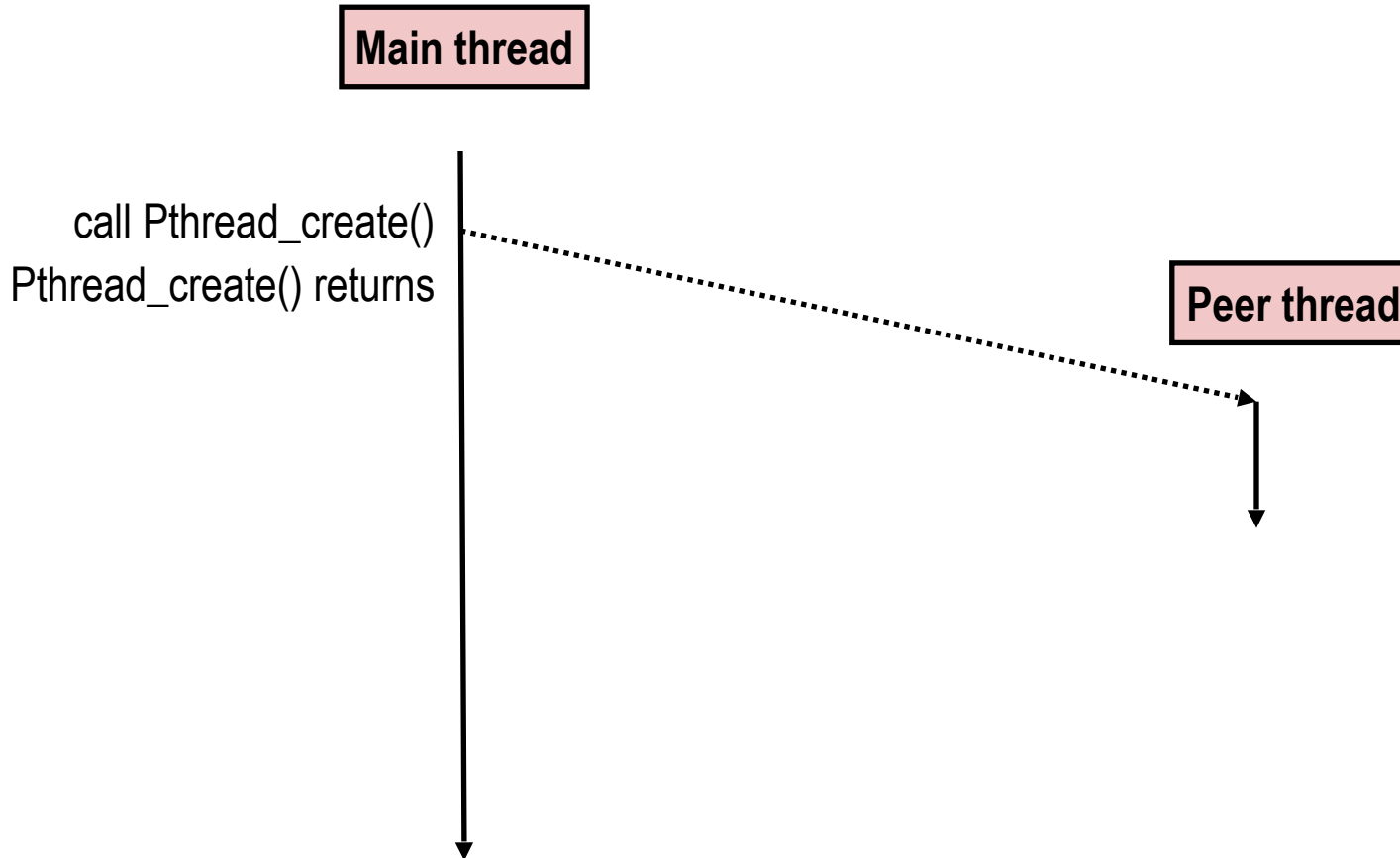


# Execution of Threaded “hello, world”



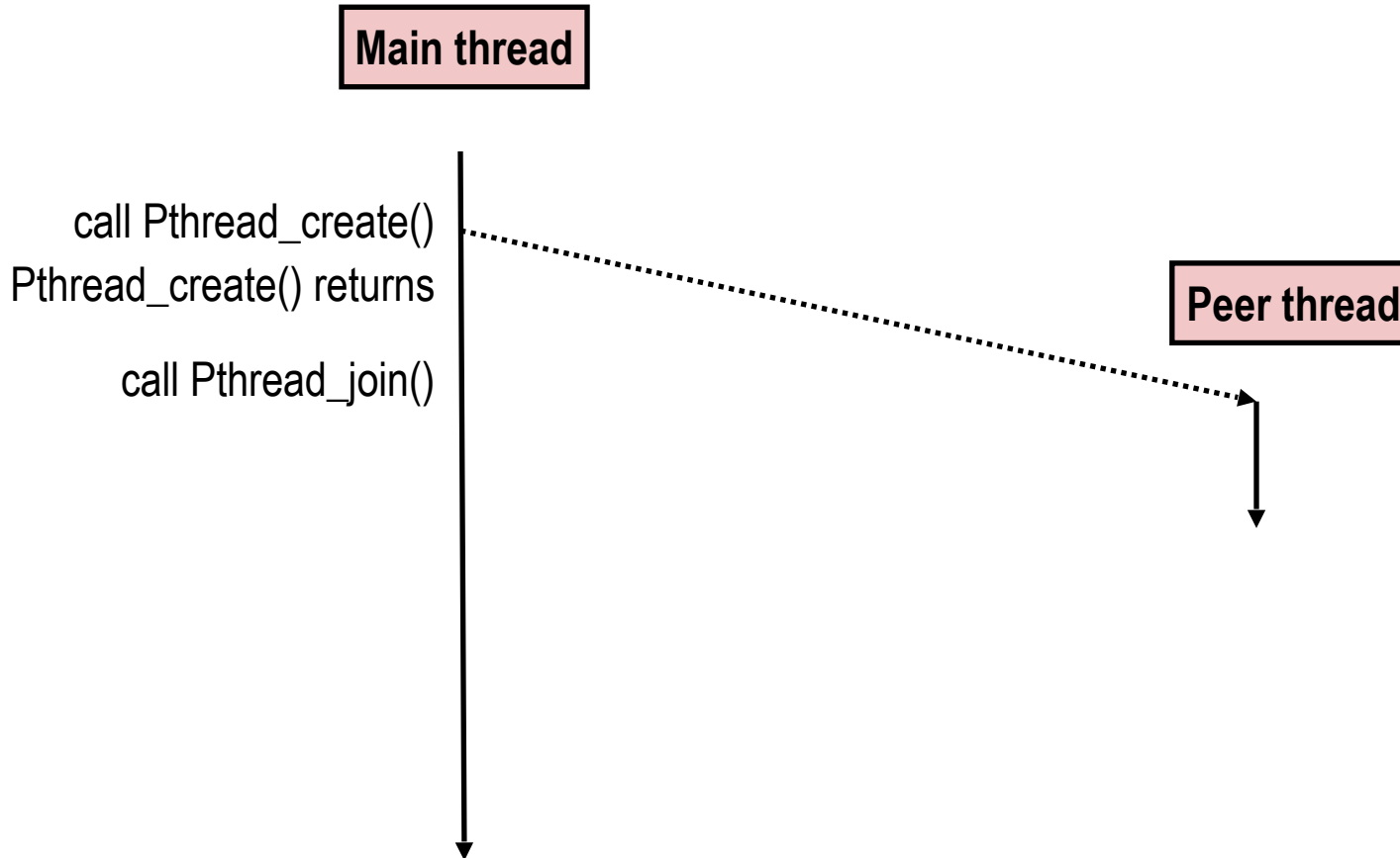


# Execution of Threaded “hello, world”



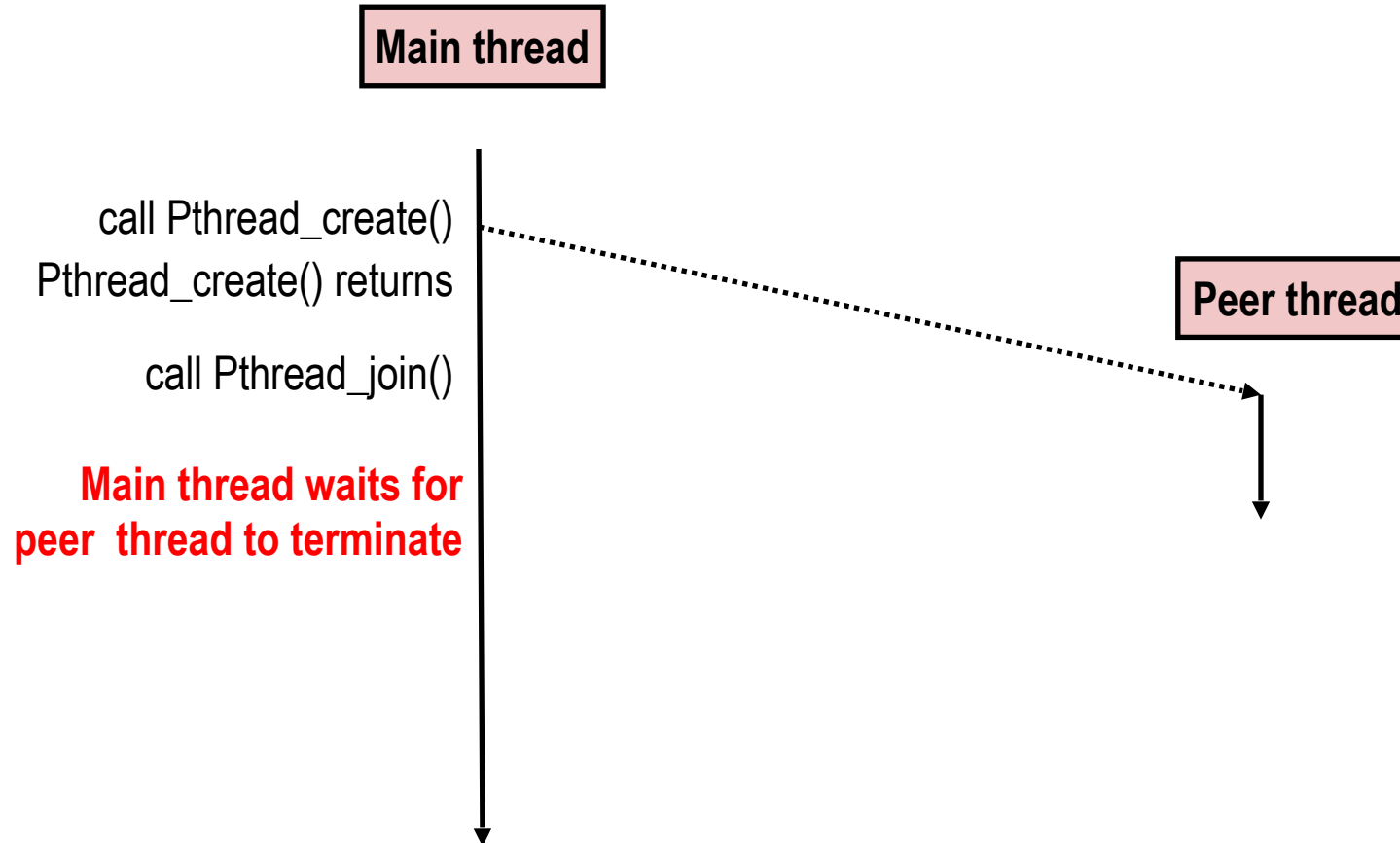


# Execution of Threaded “hello, world”



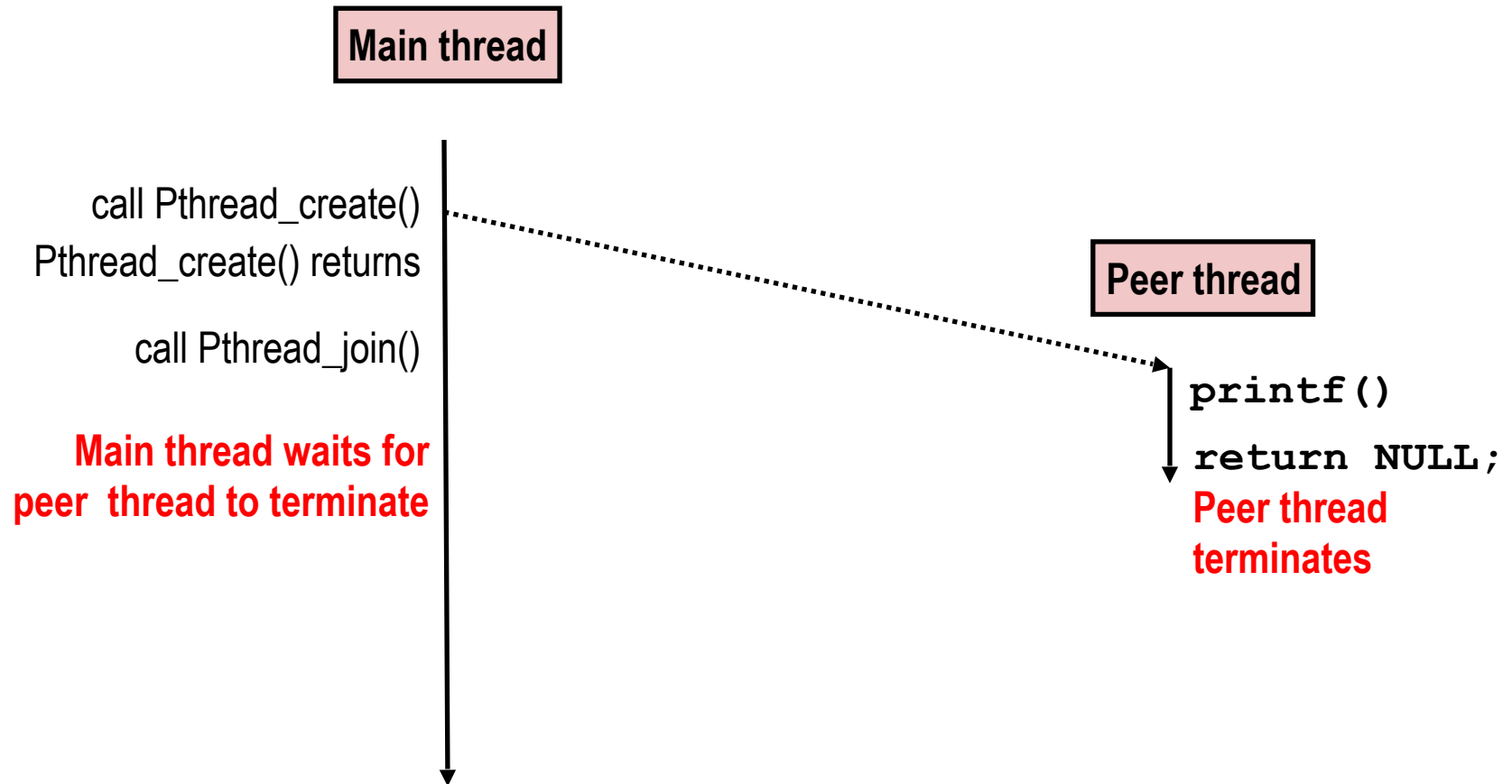


# Execution of Threaded “hello, world”



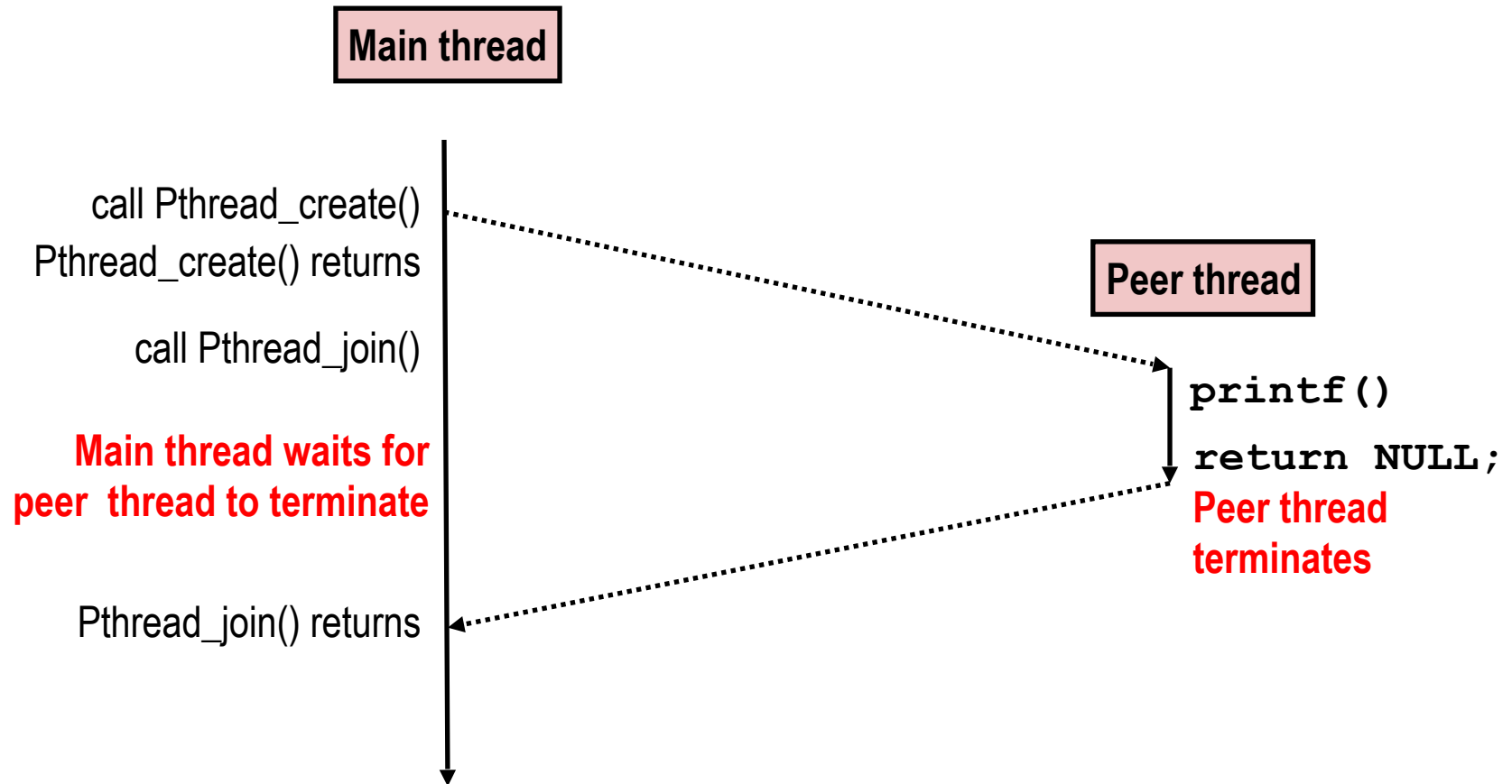


# Execution of Threaded “hello, world”



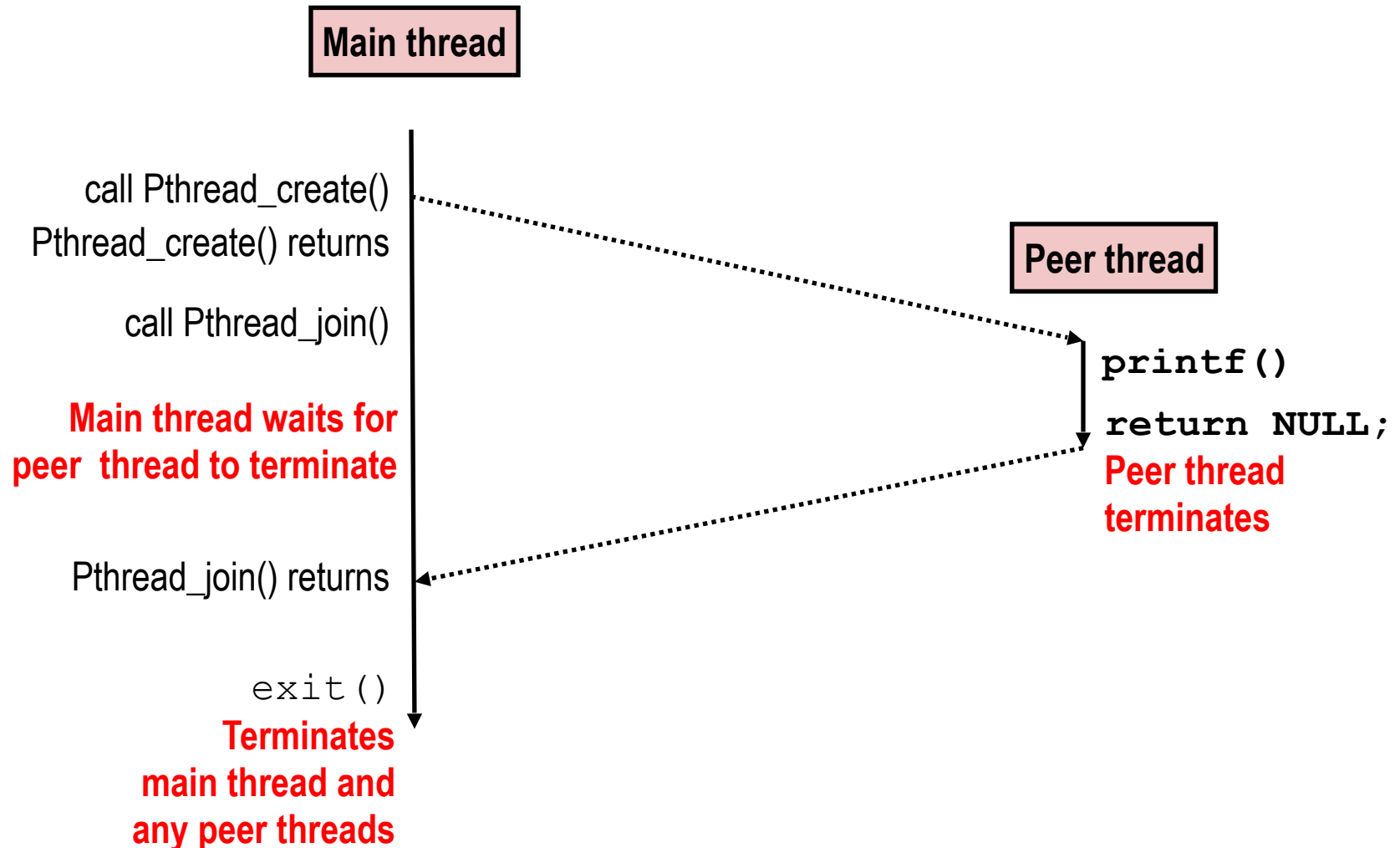


# Execution of Threaded “hello, world”





# Execution of Threaded “hello, world”





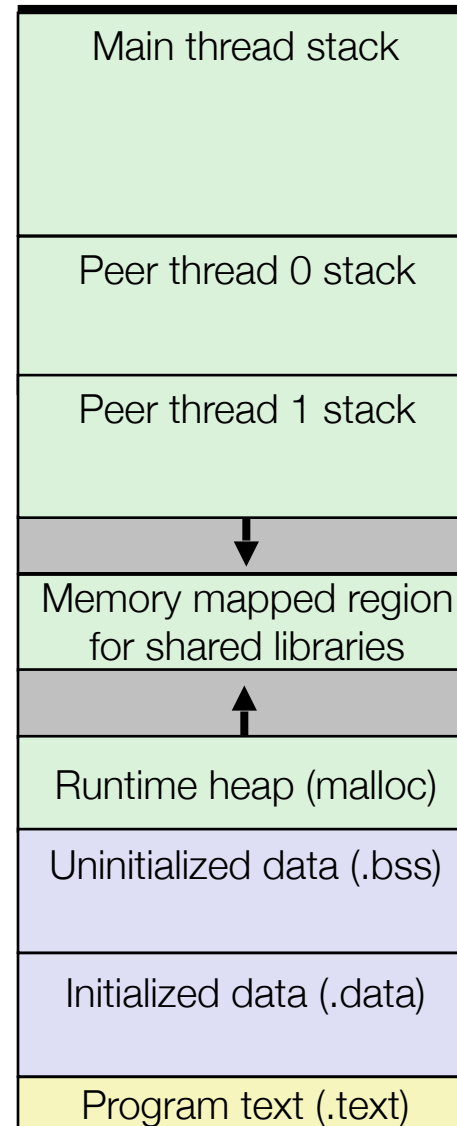
# Example Program to Illustrate Sharing

```
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





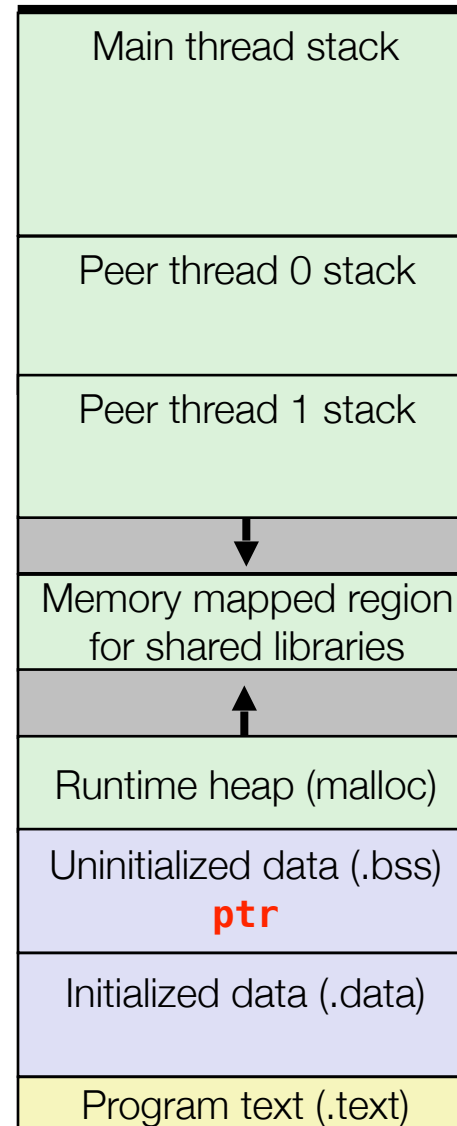
# Example Program to Illustrate Sharing

```
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





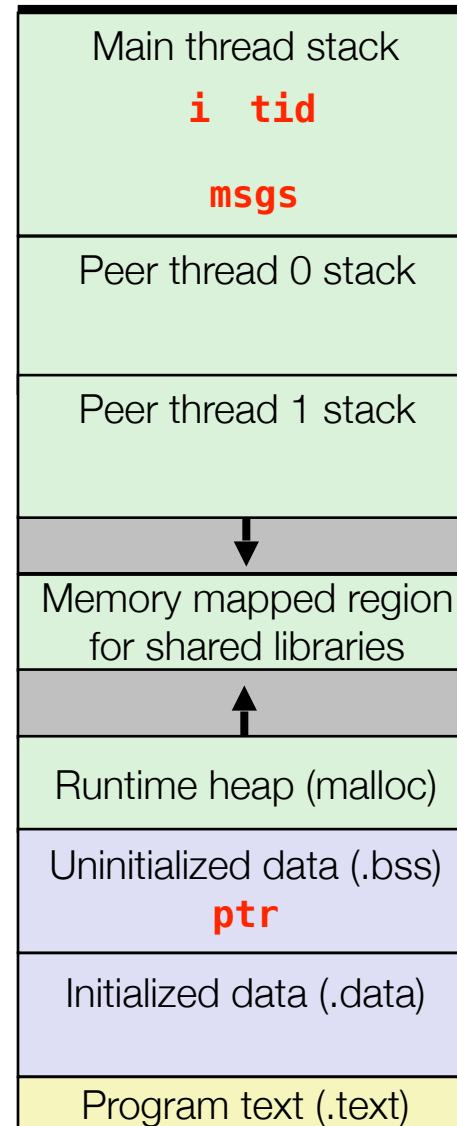
# Example Program to Illustrate Sharing

```
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





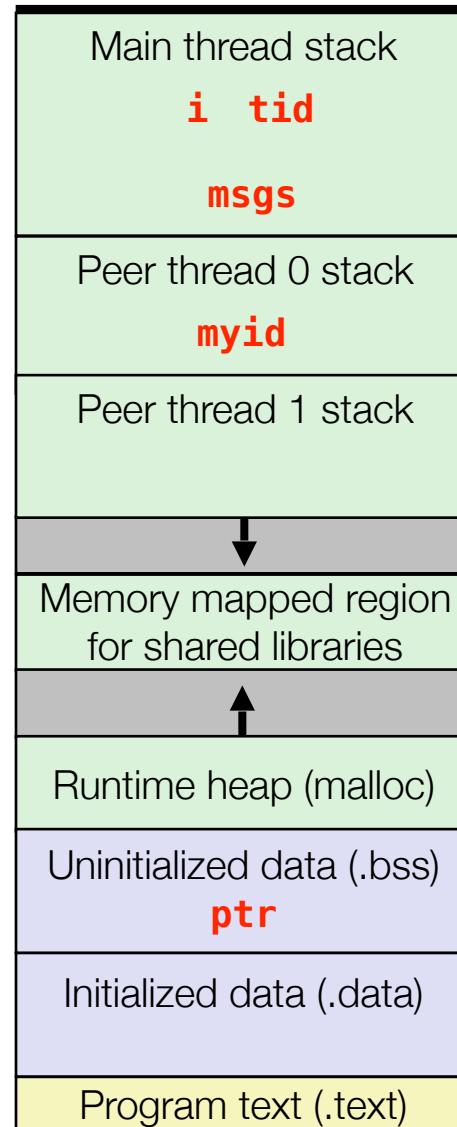
# Example Program to Illustrate Sharing

```
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





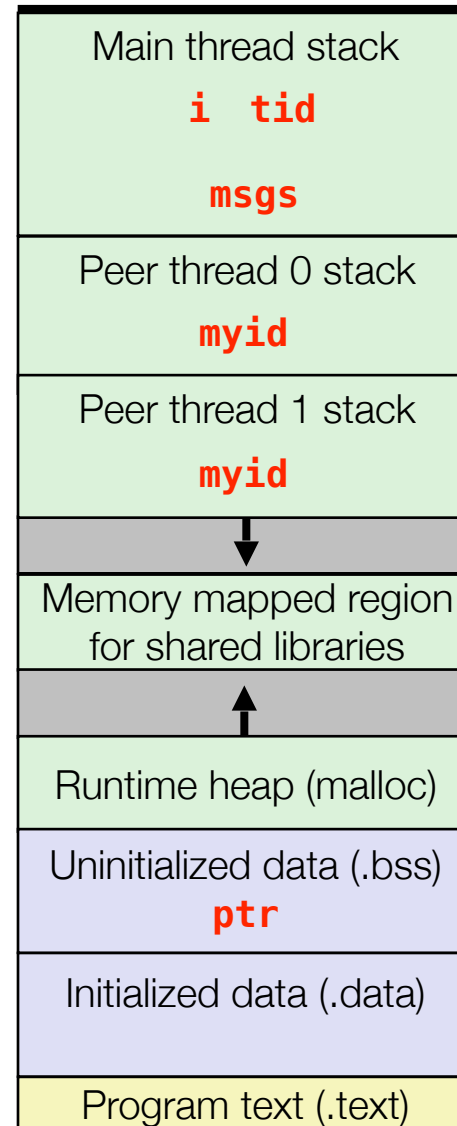
# Example Program to Illustrate Sharing

```
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





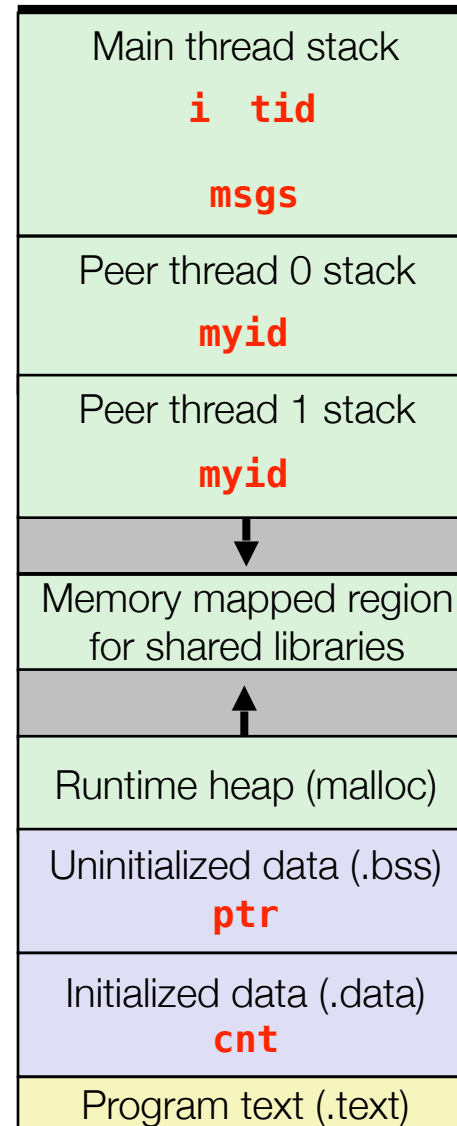
# Example Program to Illustrate Sharing

```
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





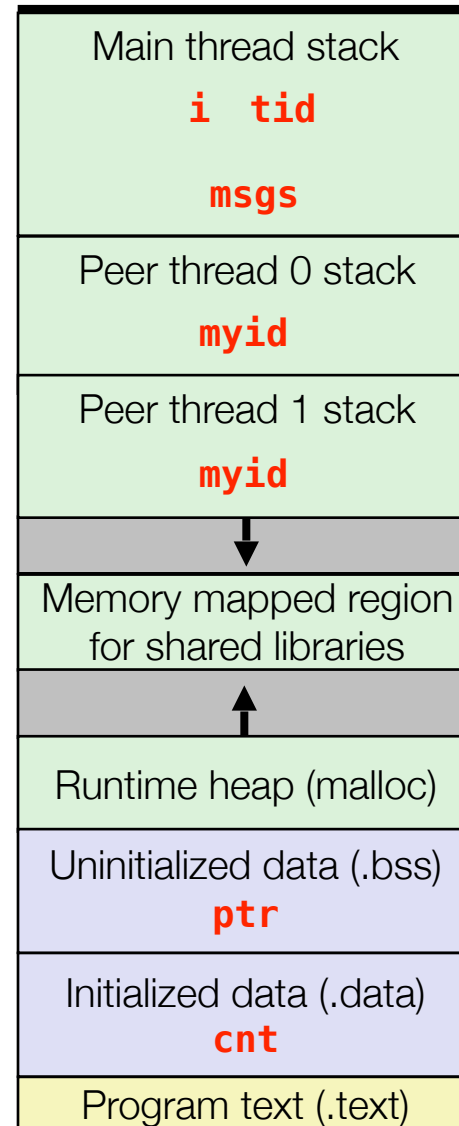
# Example Program to Illustrate Sharing

```
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



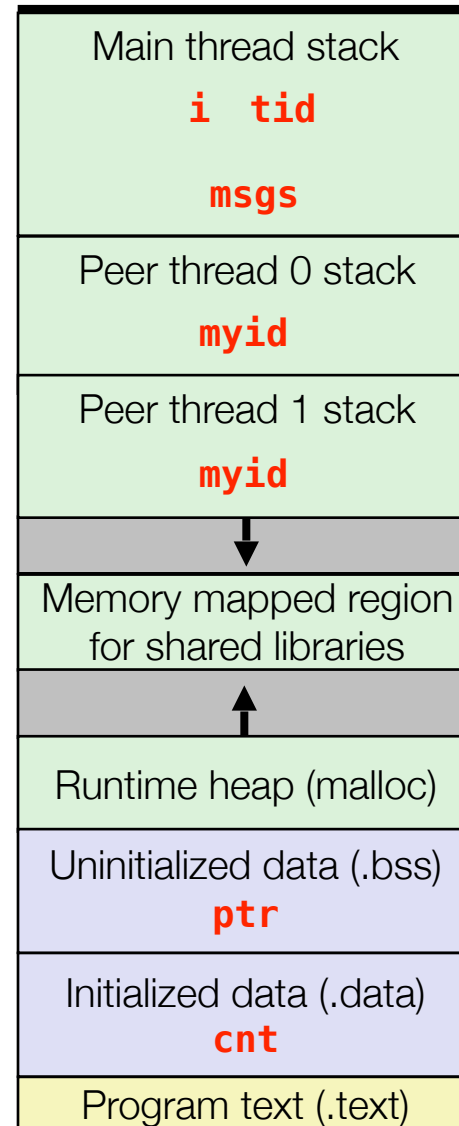
# Example Program to Illustrate Sharing

```
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



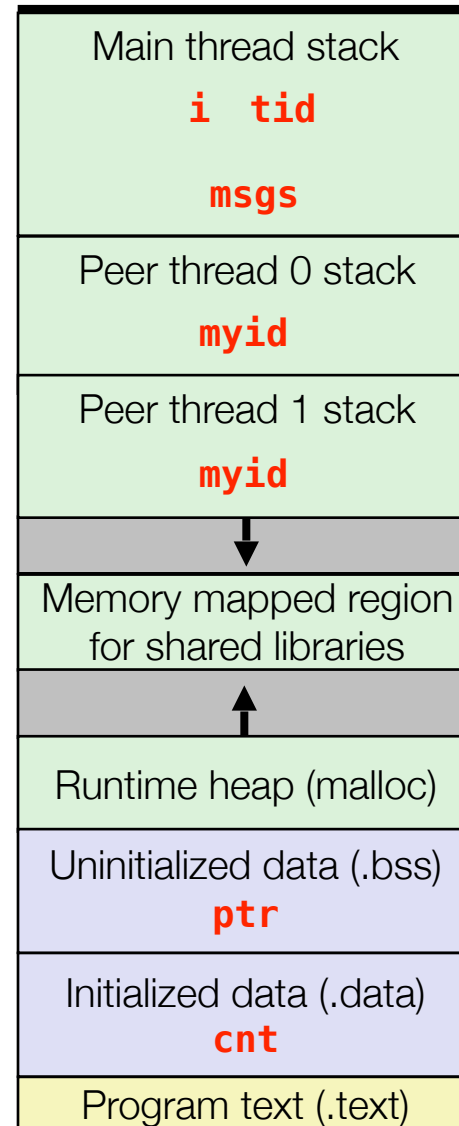
# Example Program to Illustrate Sharing

```
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



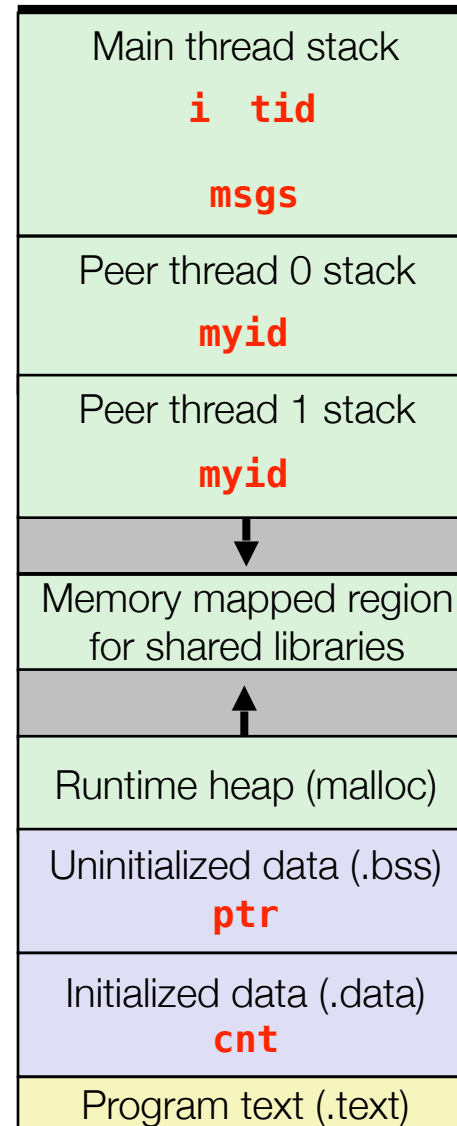
# Example Program to Illustrate Sharing

```
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



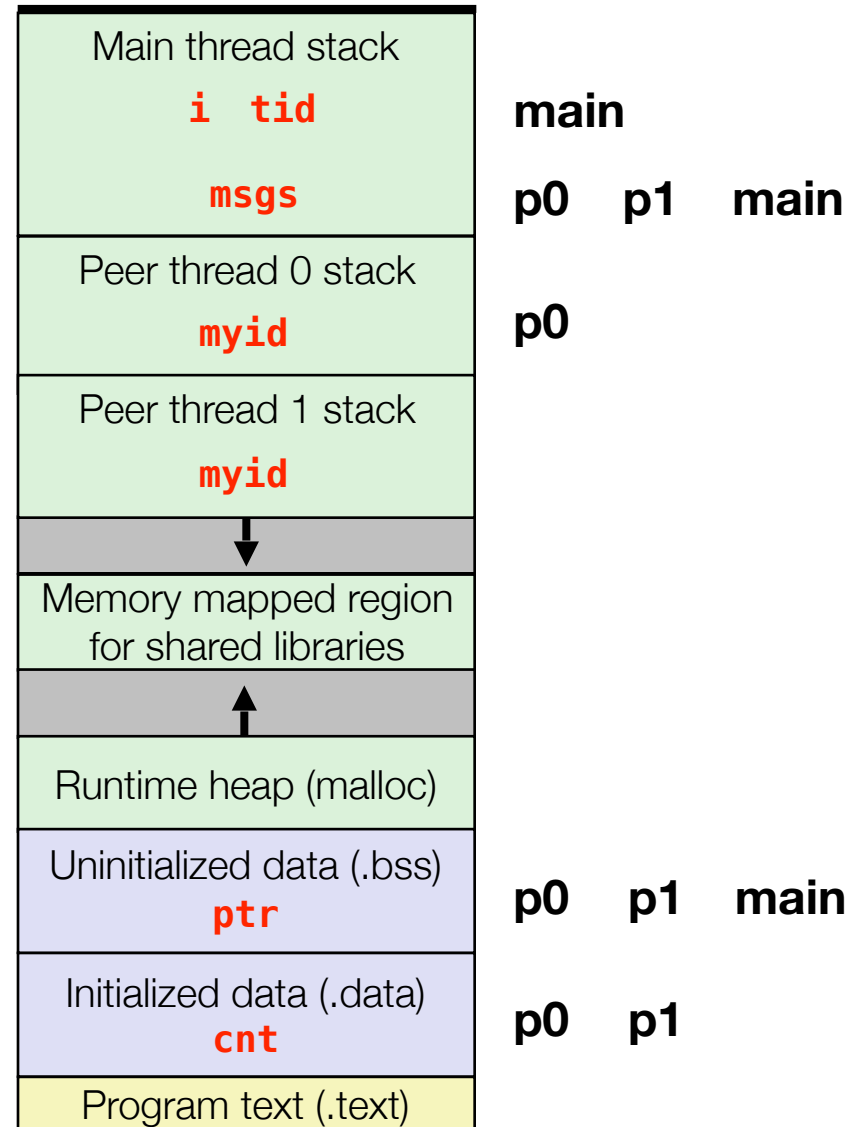
# Example Program to Illustrate Sharing

```
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





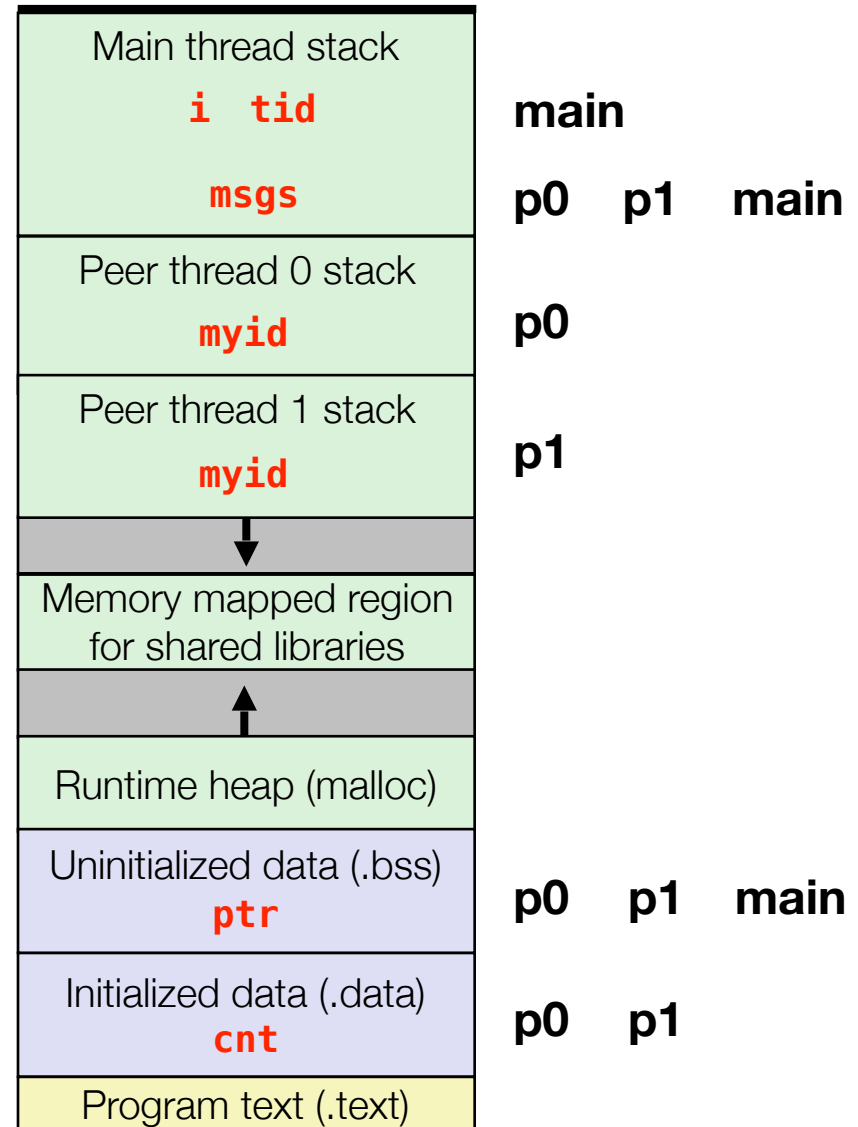
# Example Program to Illustrate Sharing

```
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





# Threads Memory Model



# Threads Memory Model

- Conceptual model:
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments of the process virtual address space
    - Open files and installed handlers
- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

*The mismatch between the conceptual and operation model is a source of confusion and errors*