

# **CSC 252: Computer Organization**

## **Spring 2021: Lecture 24**

Instructor: Yuhao Zhu

Department of Computer Science  
University of Rochester

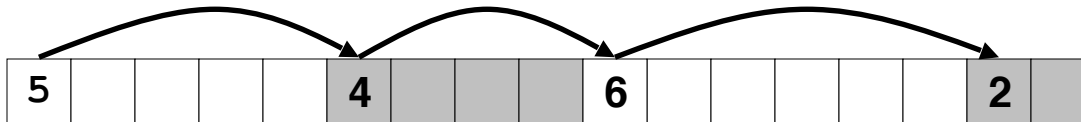
# Announcements

- Lab5: <https://www.cs.rochester.edu/courses/252/spring2021/labs/assignment5.html>
- Released a virtual memory problem set; not to be turned in: <https://www.cs.rochester.edu/courses/252/spring2021/handouts.html>

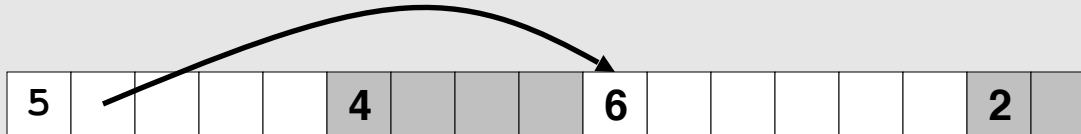
SUN 25	MON 26	TUE 27	WED 28	THU 29	FRI 30	SAT May 1
2	3	4	5 <b>A5 Due</b>	6 <b>Today Last Lecture</b>	7	8
9	10	11	12 <b>Final</b>	13	14	15

# Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



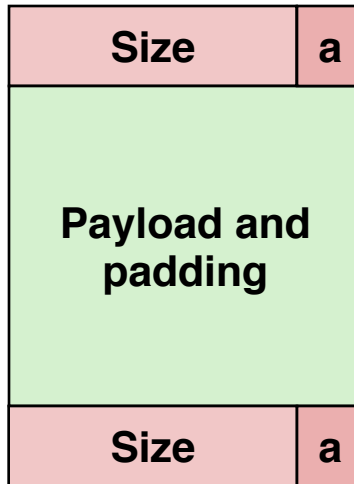
- Method 2: *Explicit list* among the free blocks using pointers



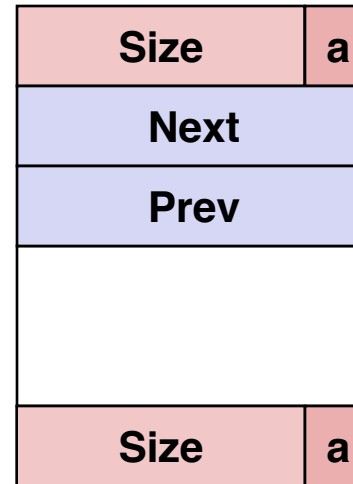
- Method 3: *Segregated free list*
  - Different free lists for different size classes

# Explicit Free Lists

Allocated block  
(same as before)



Free block



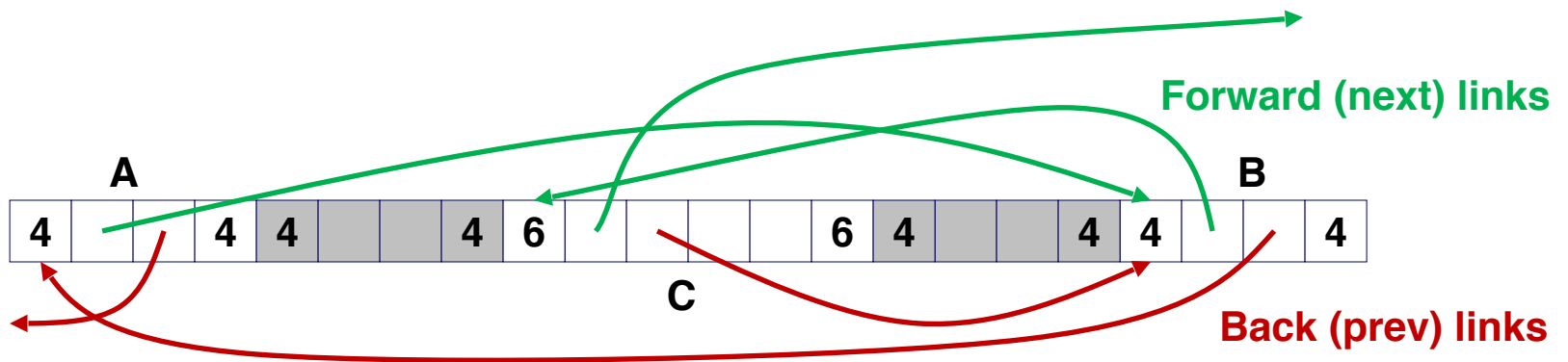
- Maintain list(s) of *free* blocks, not *all* blocks
  - The “next” free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
    - These pointers exist only in free blocks, occupying the would-be payload area, so not really an overhead.
  - Still need boundary tags for coalescing.

# Explicit Free Lists

- Logically:

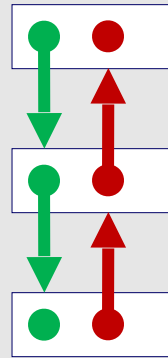


- Physically: blocks can be in any order



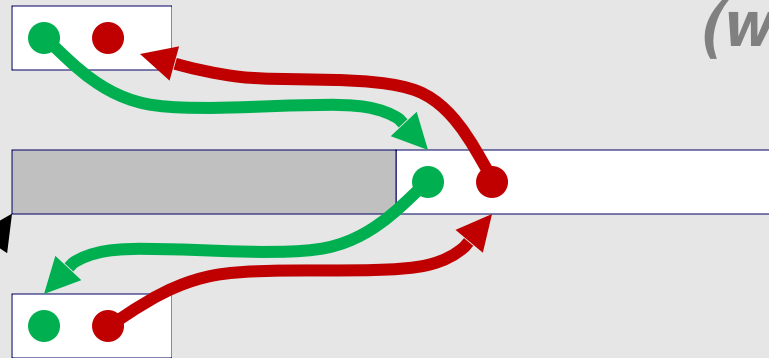
# Allocating From Explicit Free Lists

*Before*



conceptual graphic

*After*



*(with splitting)*

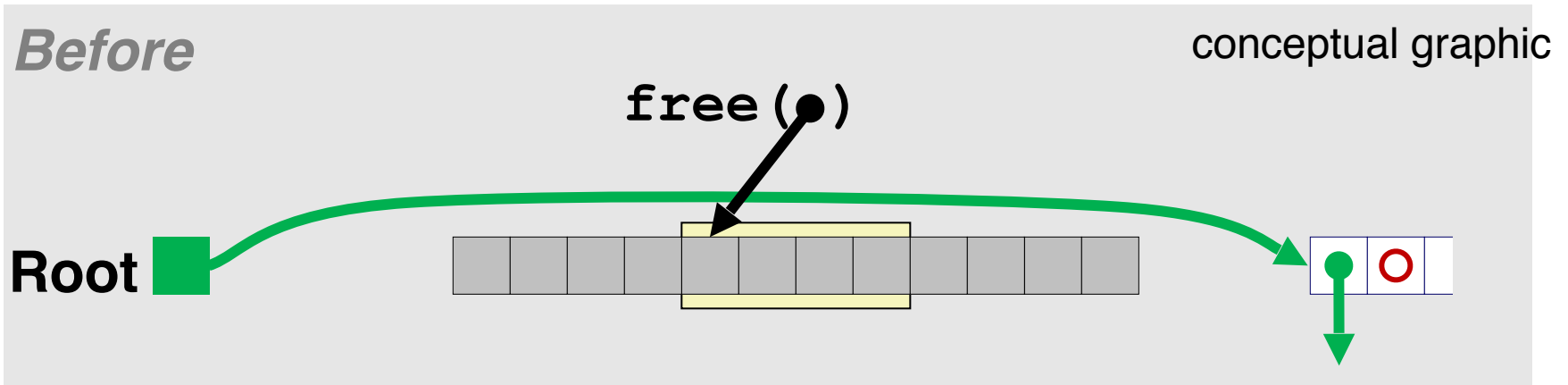
● = malloc(...)

# Freeing With Explicit Free Lists

- *Insertion policy*: Where in the free list do you put a newly freed block?
- LIFO (last-in-first-out) policy
  - Insert freed block at the beginning of the free list
  - *Pro*: simple and constant time
  - *Con*: studies suggest fragmentation is worse than address ordered
- Address-ordered policy
  - Insert freed blocks so that free list blocks are always in address order:  
$$addr(prev) < addr(curr) < addr(next)$$
  - *Con*: requires search
  - *Pro*: studies suggest fragmentation is lower than LIFO

# Freeing With a LIFO Policy (Case 1)

- Insert the freed block at the root of the list



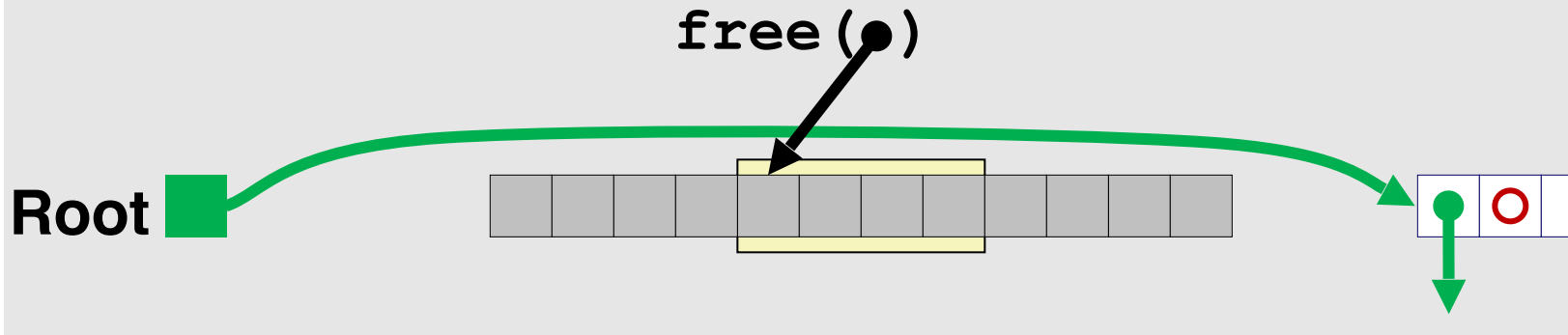


# Freeing With a LIFO Policy (Case 1)

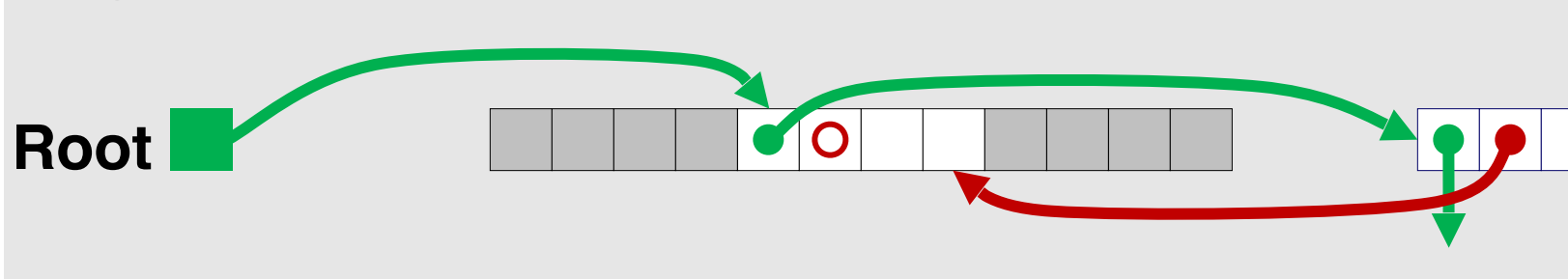
- Insert the freed block at the root of the list

*Before*

conceptual graphic

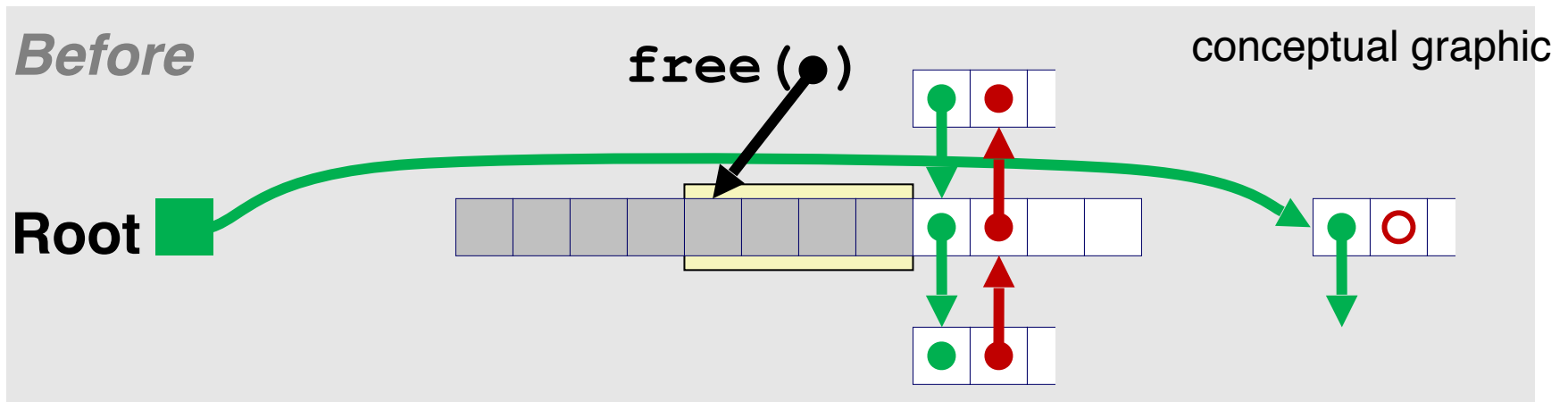


*After*



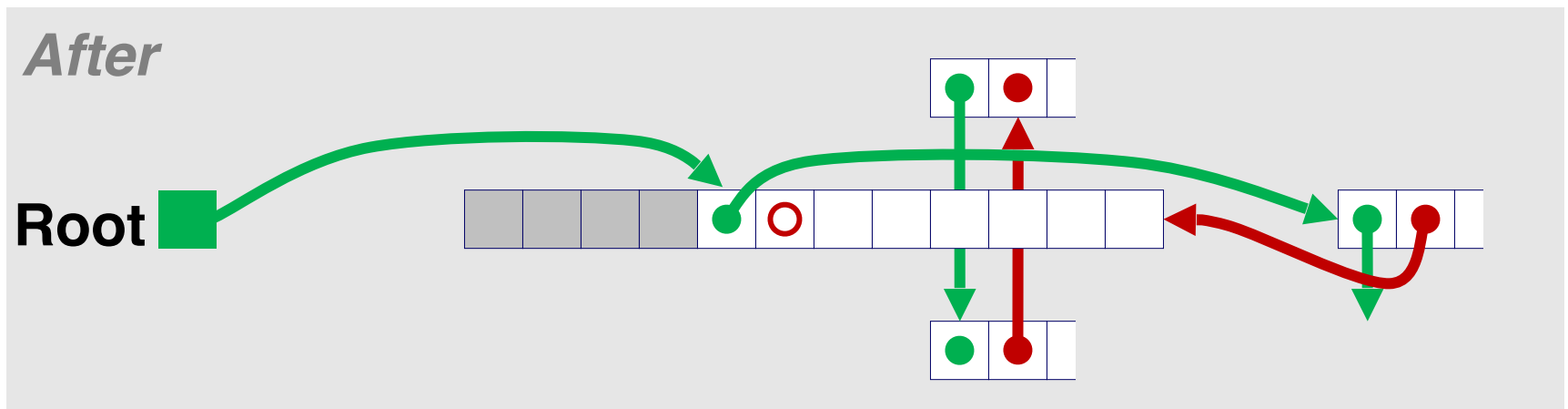
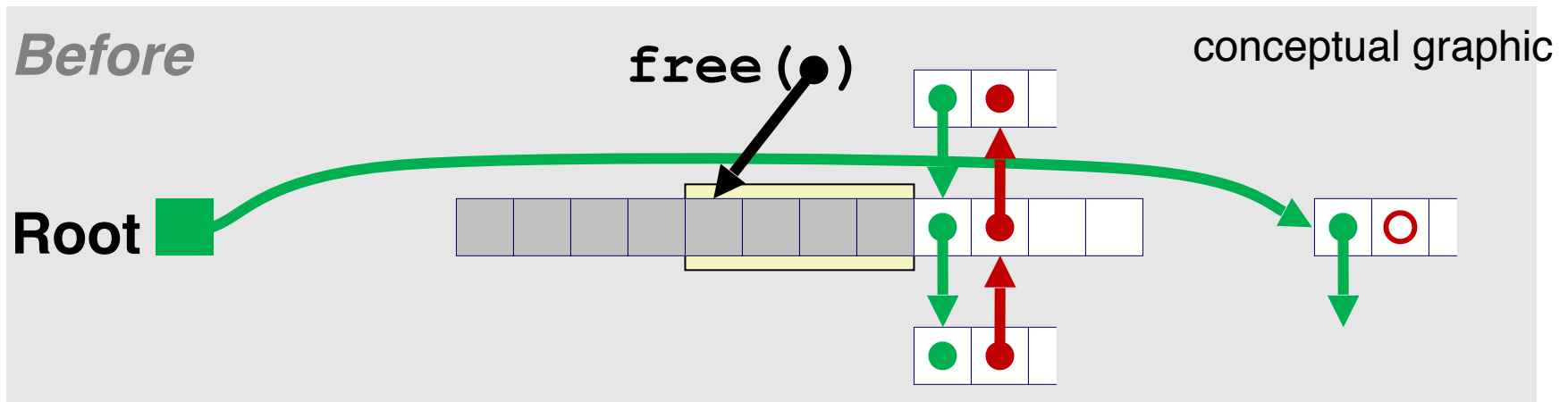
# Freeing With a LIFO Policy (Case 2)

- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



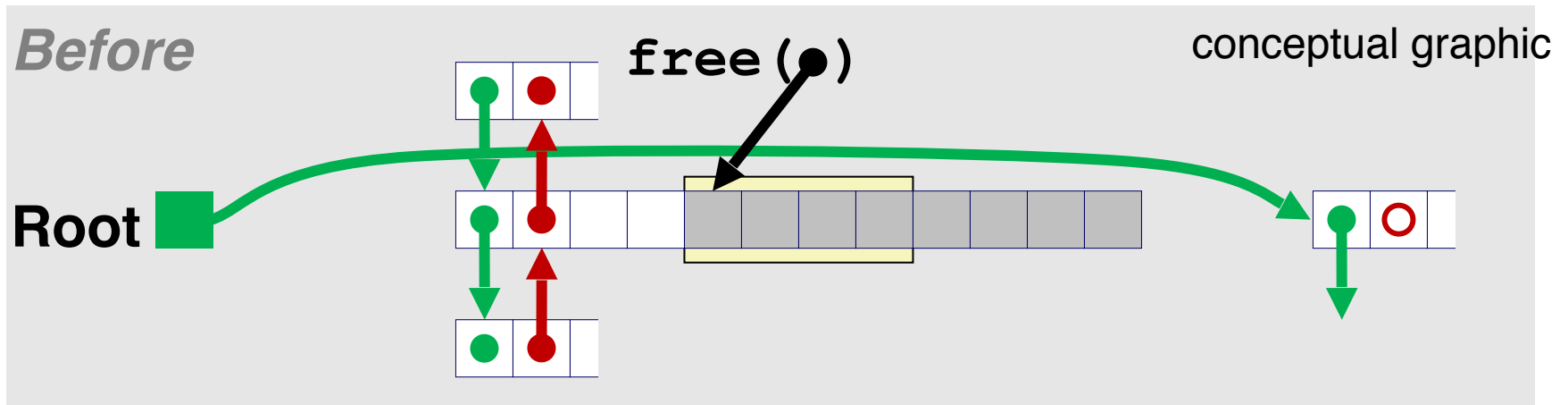
# Freeing With a LIFO Policy (Case 2)

- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list



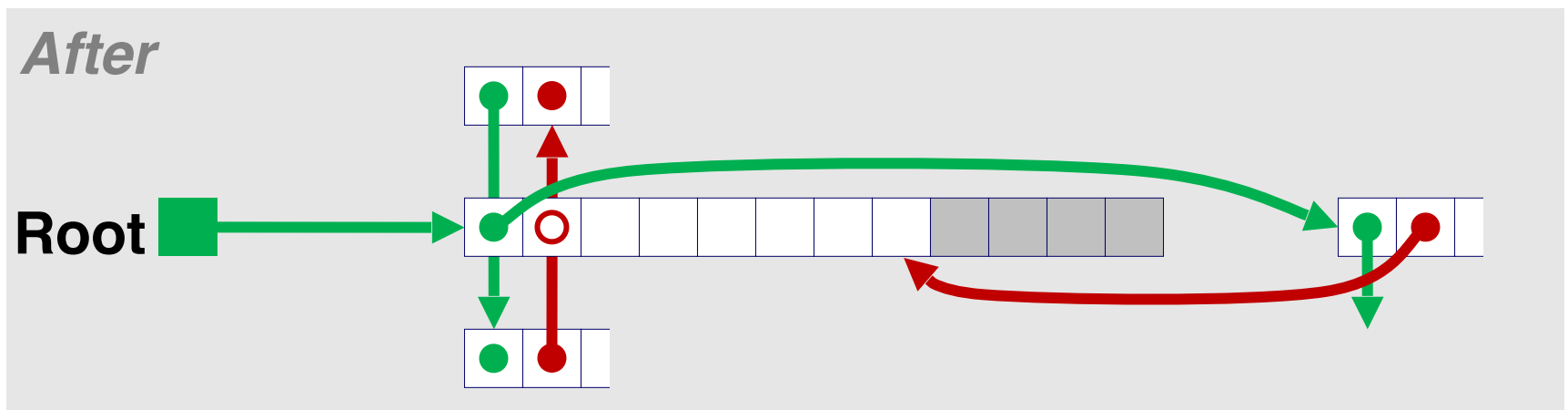
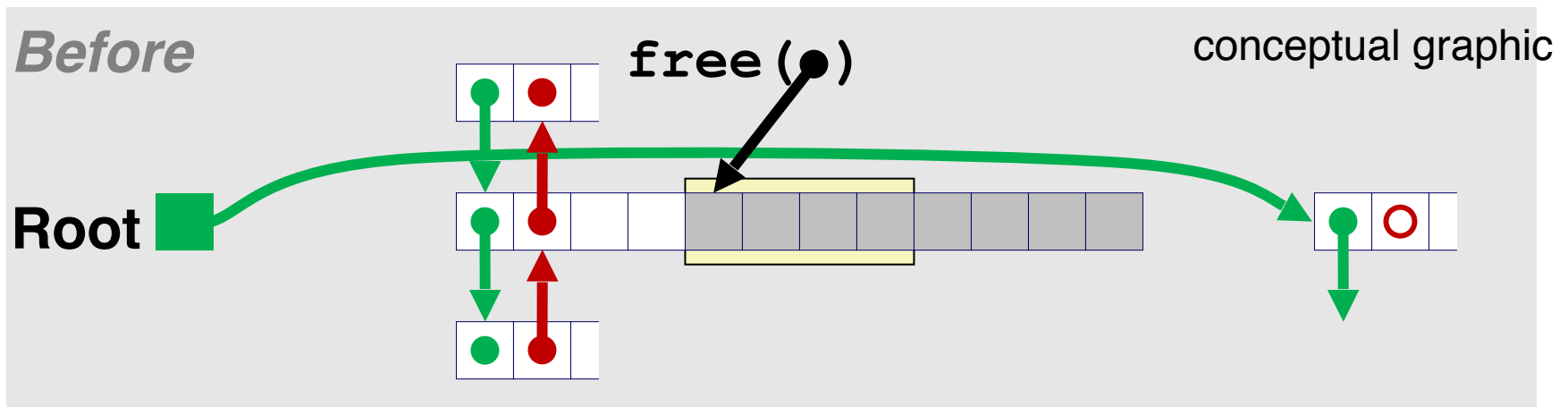
# Freeing With a LIFO Policy (Case 3)

- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



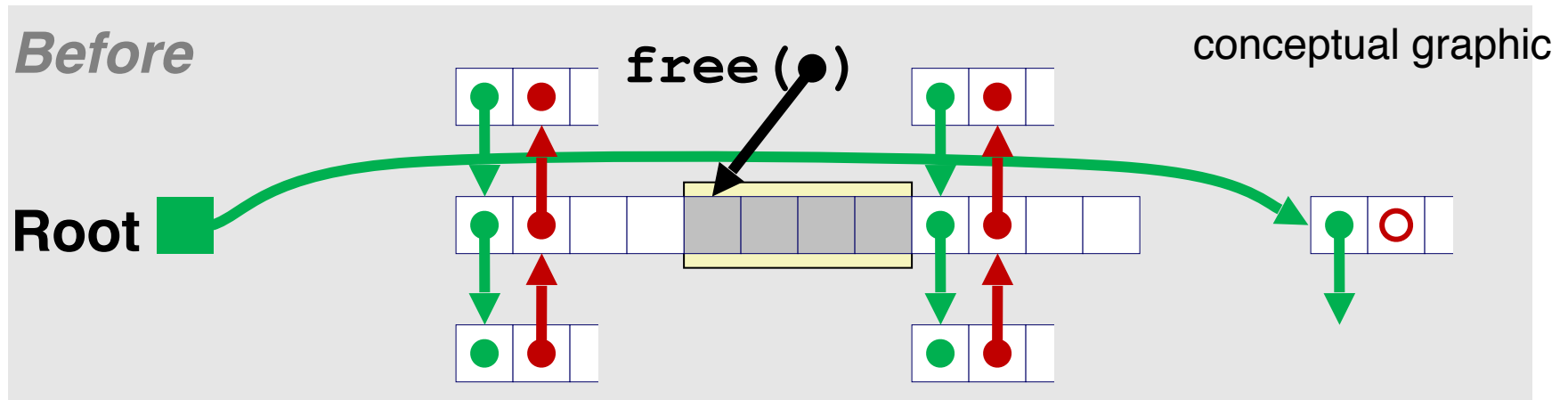
# Freeing With a LIFO Policy (Case 3)

- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list



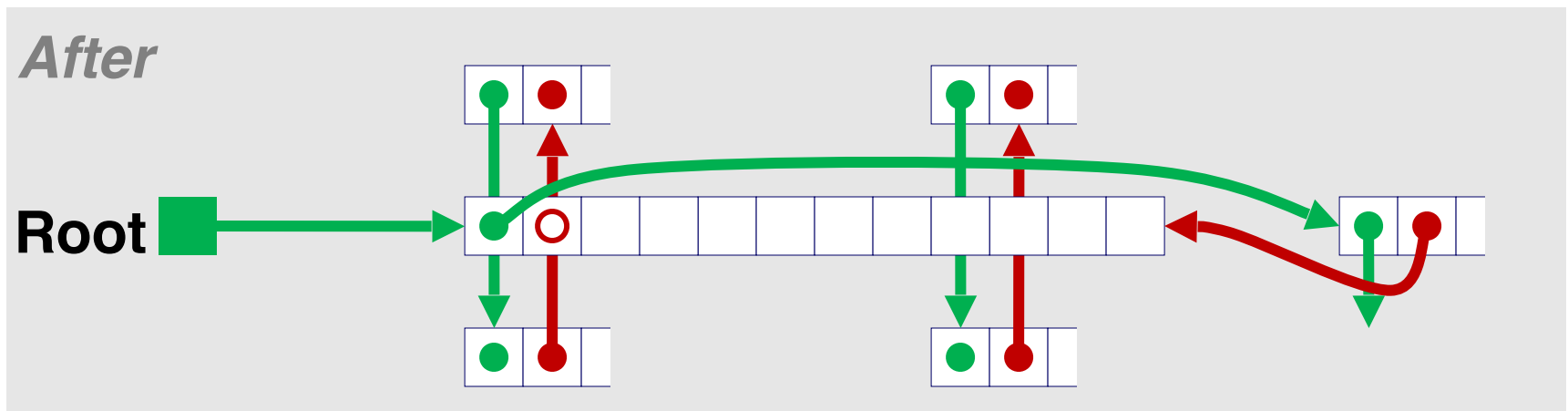
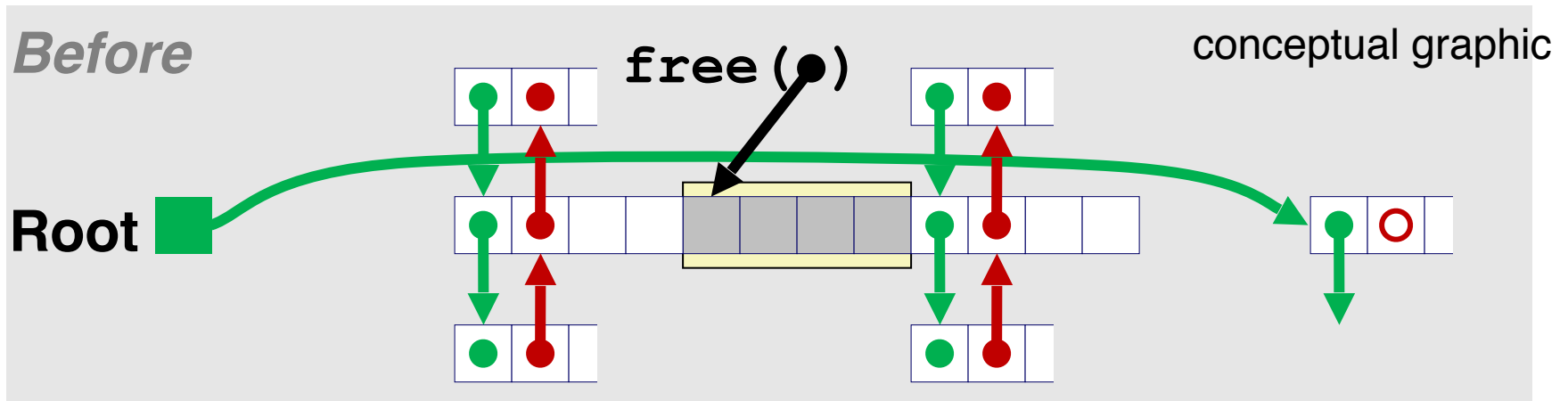
# Freeing With a LIFO Policy (Case 4)

- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



# Freeing With a LIFO Policy (Case 4)

- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



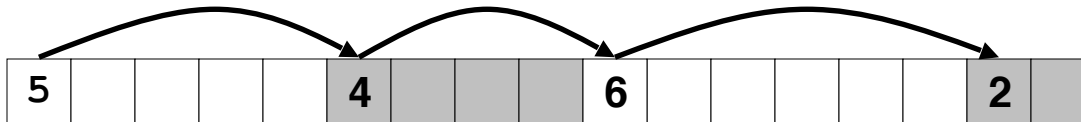
# Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of **free** blocks instead of **all** blocks. Much faster when most of the memory is full.
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links in free blocks (2 extra words needed for each block).

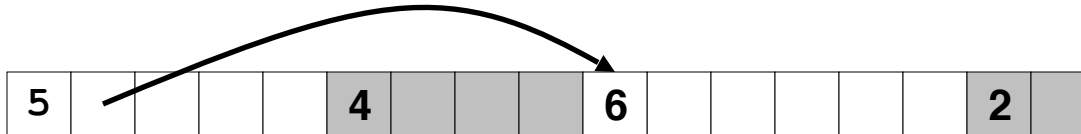


# Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

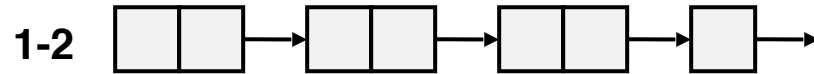


- Method 2: *Explicit list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes

# Segregated List (Seglist) Allocators



- Each *size class* of blocks has its own free list
- Organize the Seglist
  - Often have separate classes for each small size
  - For larger sizes: One class for each two-power size (why?)

# Seglist Allocator

- Given an array of free lists, each one for some size class

# Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size  $n$ :
  - Search appropriate free list for block of size  $m > n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

# Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size  $n$ :
  - Search appropriate free list for block of size  $m > n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Remember heap is in VM, so request heap memory in pages
  - Allocate block of  $n$  bytes from this new memory
  - Place remainder as a single free block in largest size class.

# Seglist Allocator

- Given an array of free lists, each one for some size class
- To allocate a block of size  $n$ :
  - Search appropriate free list for block of size  $m > n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Remember heap is in VM, so request heap memory in pages
  - Allocate block of  $n$  bytes from this new memory
  - Place remainder as a single free block in largest size class.
- To free a block:
  - Coalesce and place on appropriate list

# Advantages of Seglist allocators

- Higher throughput
  - Constant time allocation and free for requests that have a dedicated free list (most of the cases)
  - log time for power-of-two size classes (searching the lists)
- Better memory utilization
  - First-fit search of segregated free list approximates a best-fit search of entire heap.
  - Extreme case: Giving each block its own size class is equivalent to best-fit.

# 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***

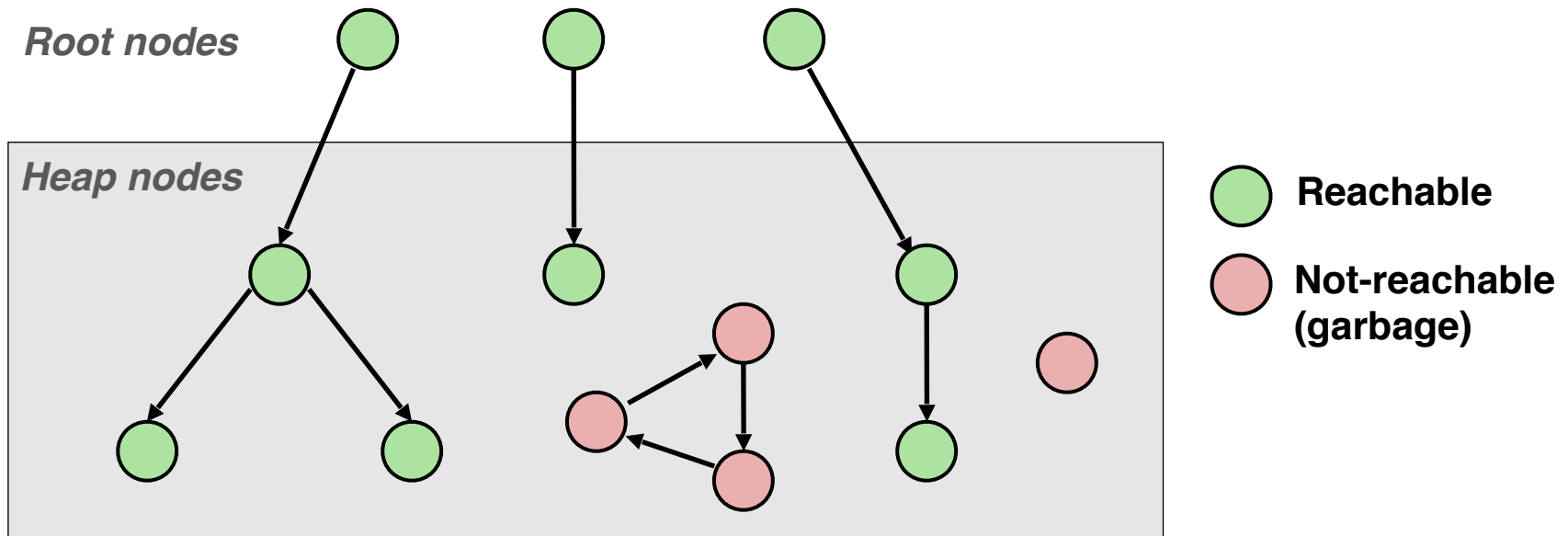


# Garbage Collection

- 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)



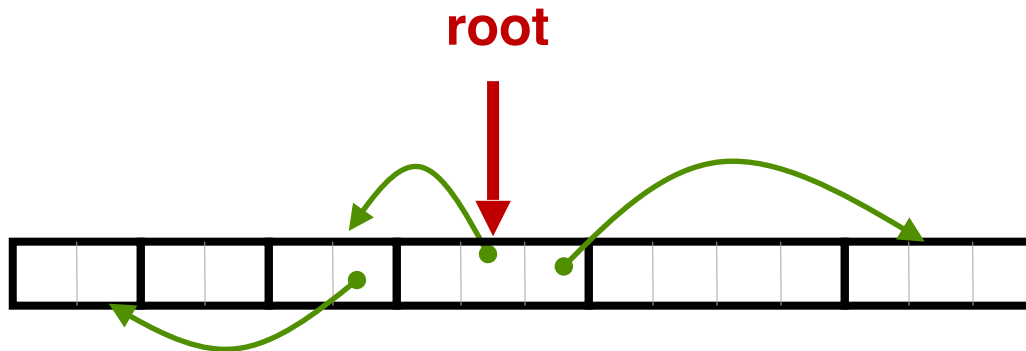
A node (block) is **reachable** if there is a path from any root to that node.

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



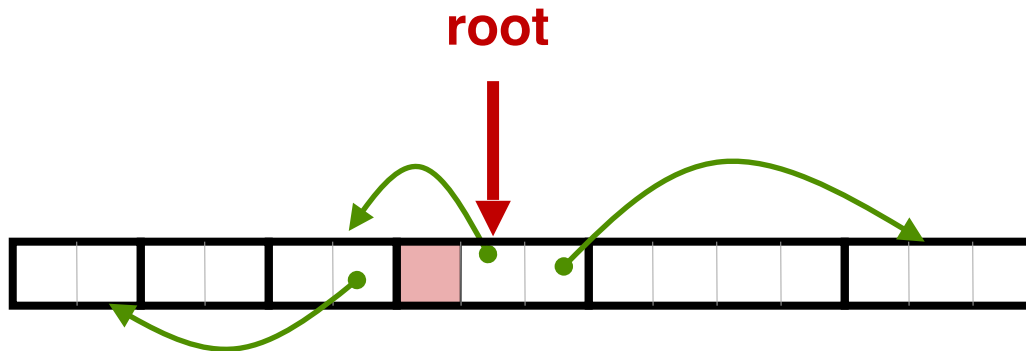
*Note: arrows here denote memory refs, not free list ptrs.*

 **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



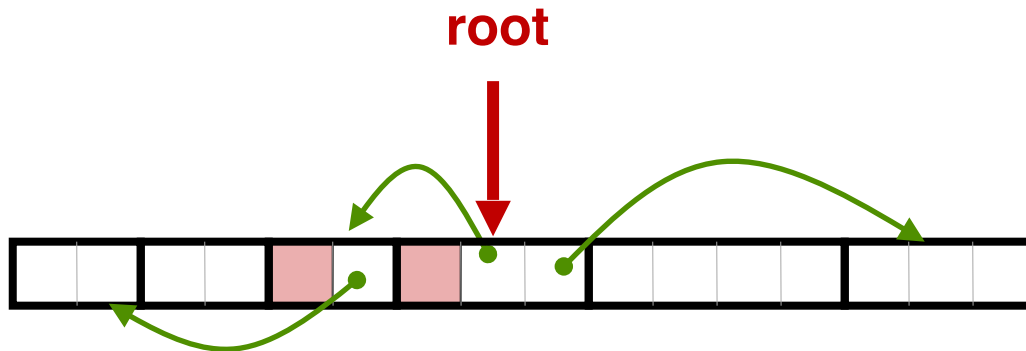
*Note: arrows here denote memory refs, not free list ptrs.*

 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

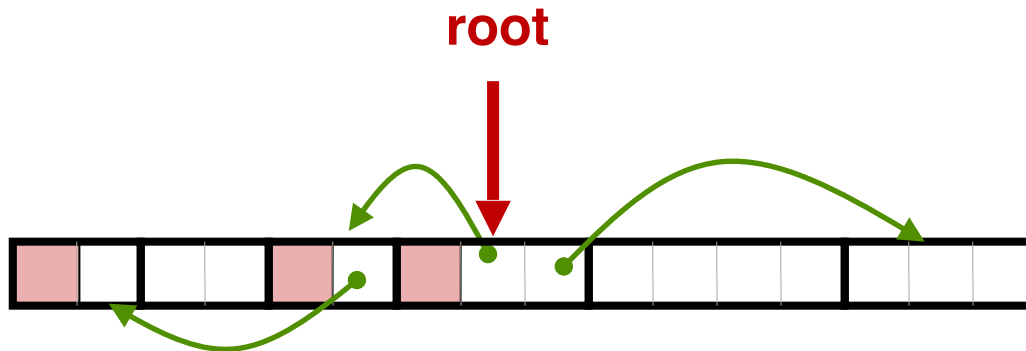


*Note: arrows here denote memory refs, not free list ptrs.*

 **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



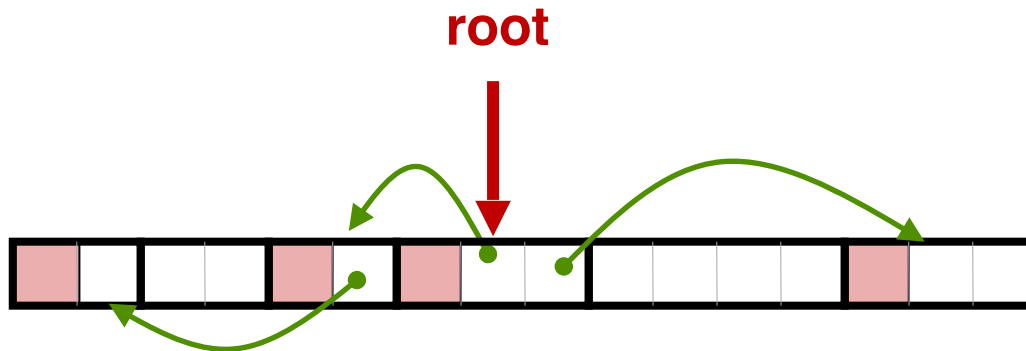
*Note: arrows here denote memory refs, not free list ptrs.*

 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



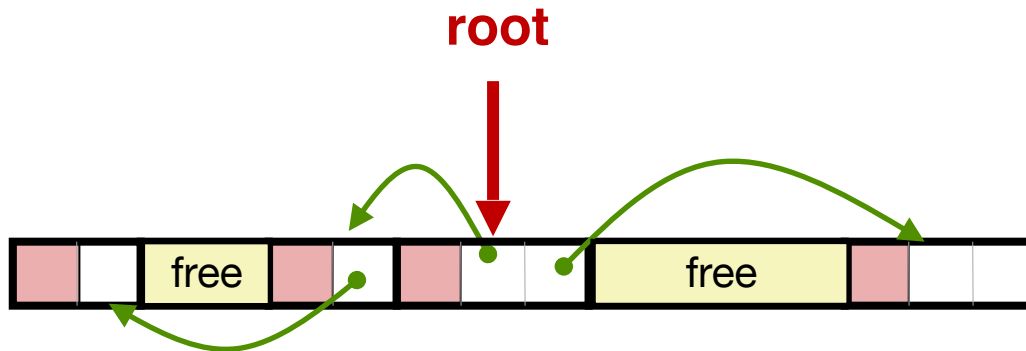
*Note: arrows here denote memory refs, not free list ptrs.*

 **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.*



# 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

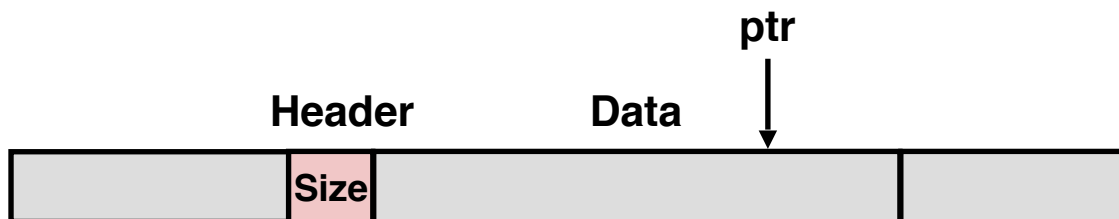
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.

# 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?

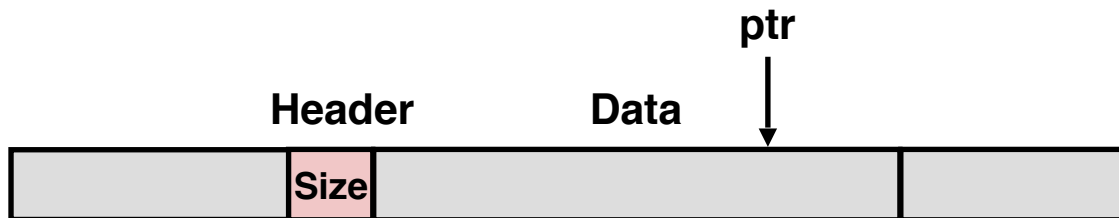
# Conservative Mark & Sweep in C

- 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?



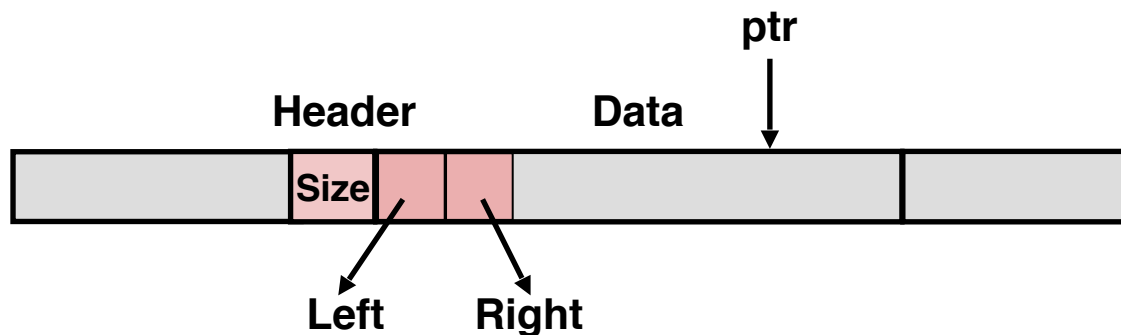
# Conservative Mark & Sweep in C

- 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

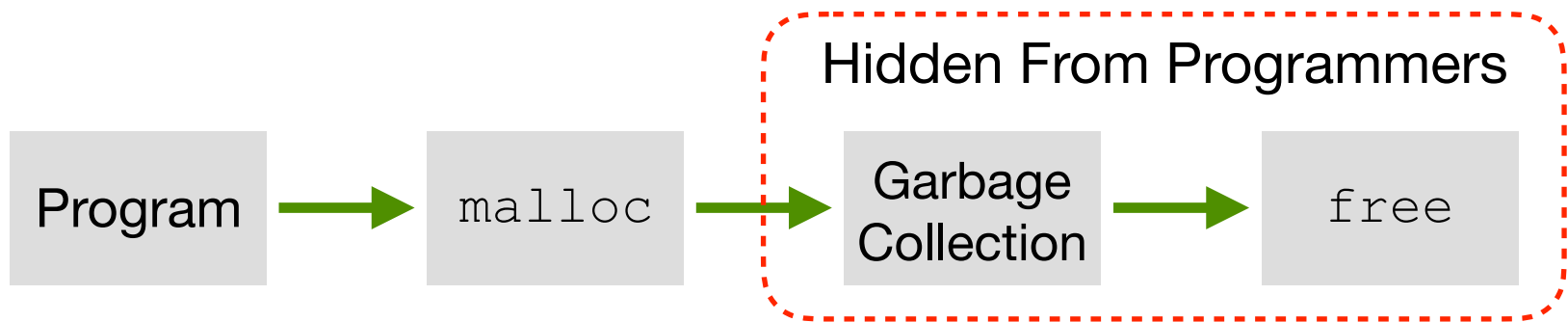
- Garbage Collection in C is tricky.
- How do you know a pointer is a pointer? After all, a pointer is just a 8-byte value. Any consecutive 8 bytes could be disguised as a pointer.
  - Must be conservative. Any 8 bytes whose values fall within the range of the heap must be treated as a pointer.
- C pointers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

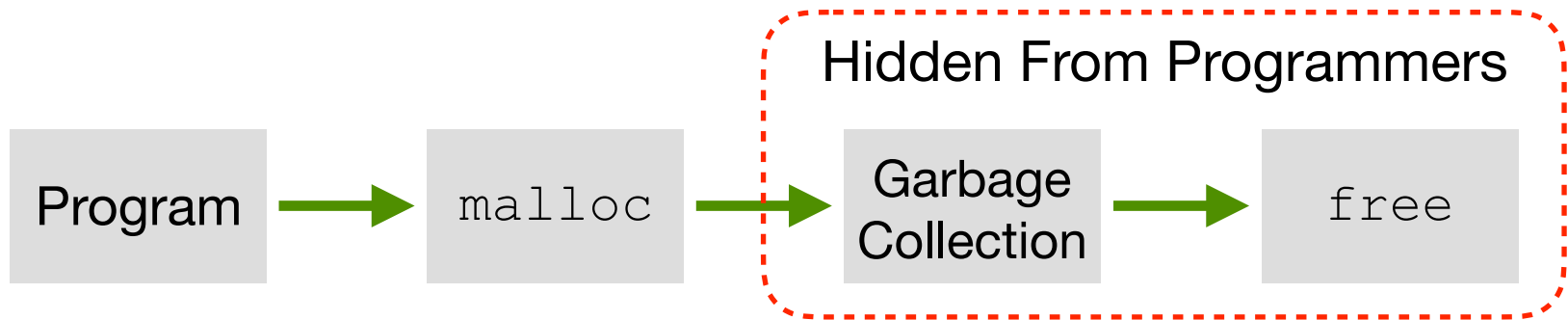
- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.





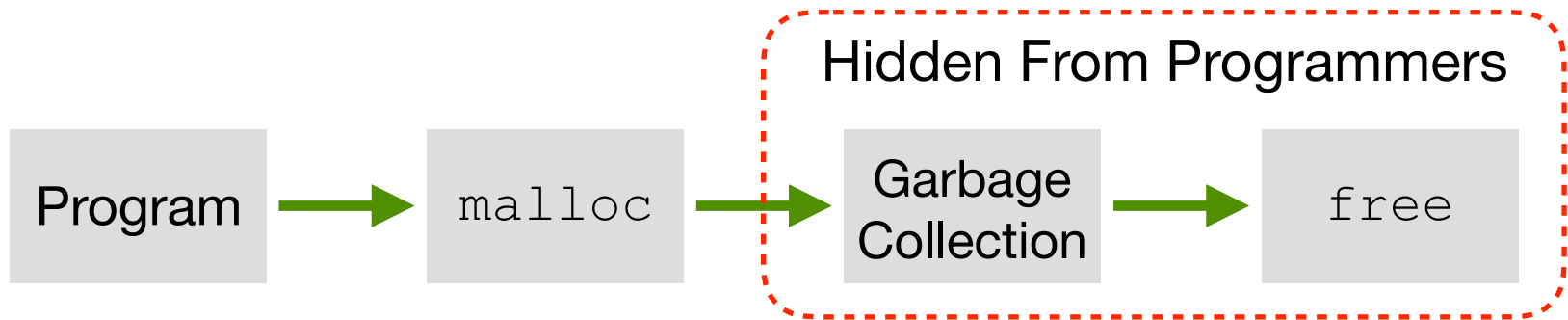
# Potential GC Implementations (in C)

- 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)

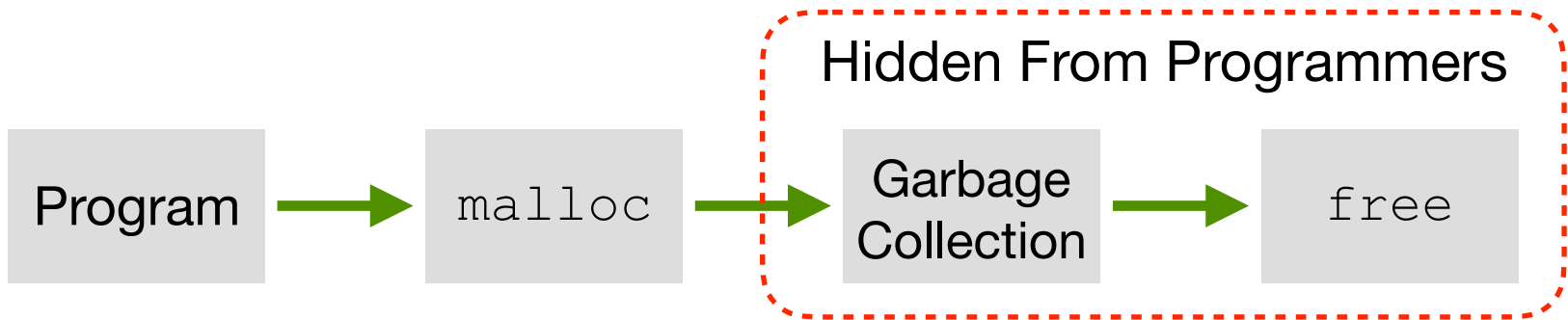
- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - **Stop-the-world GC**. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



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

# Potential GC Implementations (in C)

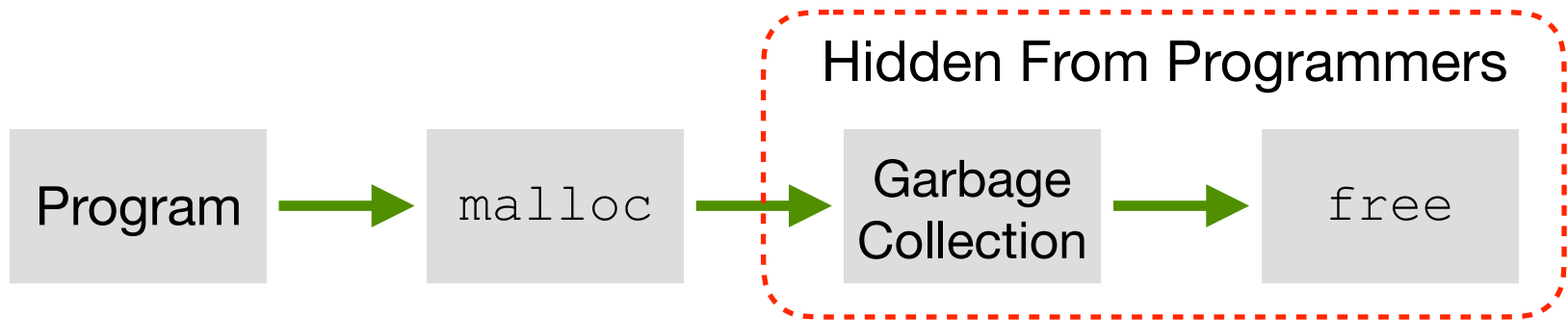
- Can build on top of `malloc/free` function
  - Call `malloc` until you run out of space. Then `malloc` will call GC.
  - **Stop-the-world GC**. When performing GC, the entire program stops. Some calls to `malloc` will take considerably longer than others.



- 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

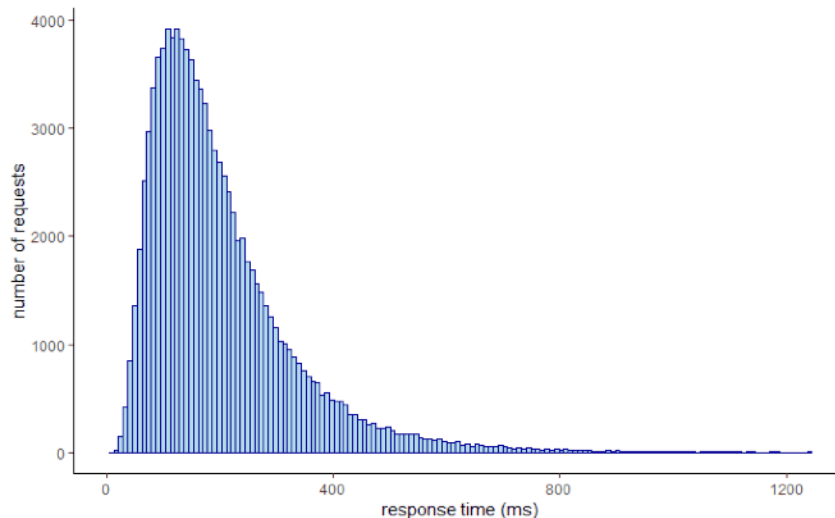
# Garbage Collection Implications

- 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

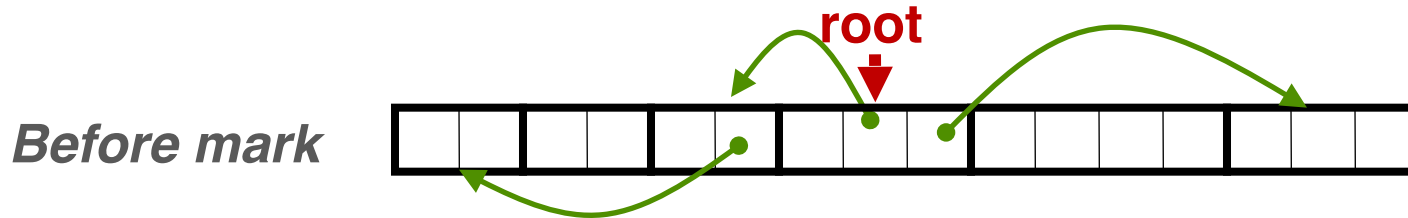
- Mark-and-sweep collection (McCarthy, 1960)

# 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.

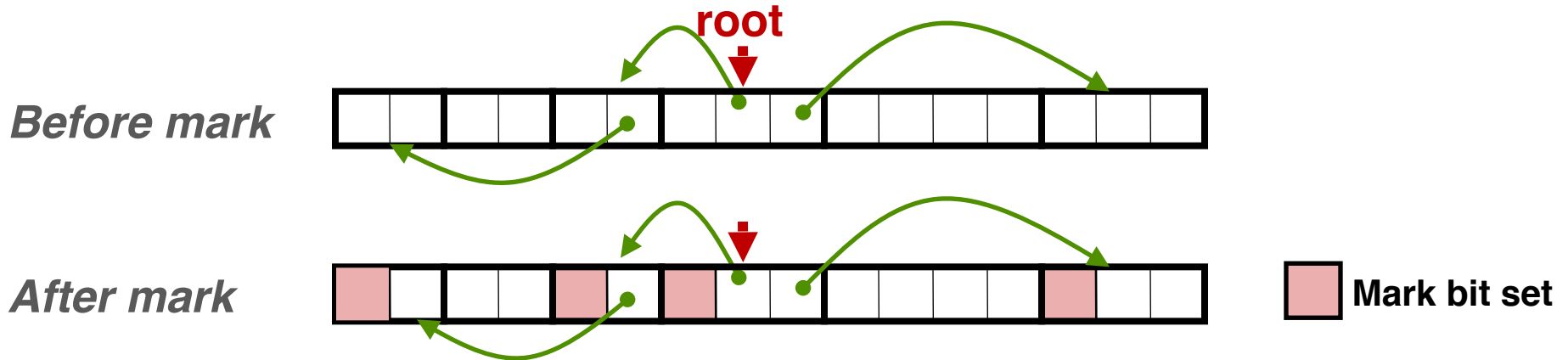
# Classical GC Algorithms

- 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.



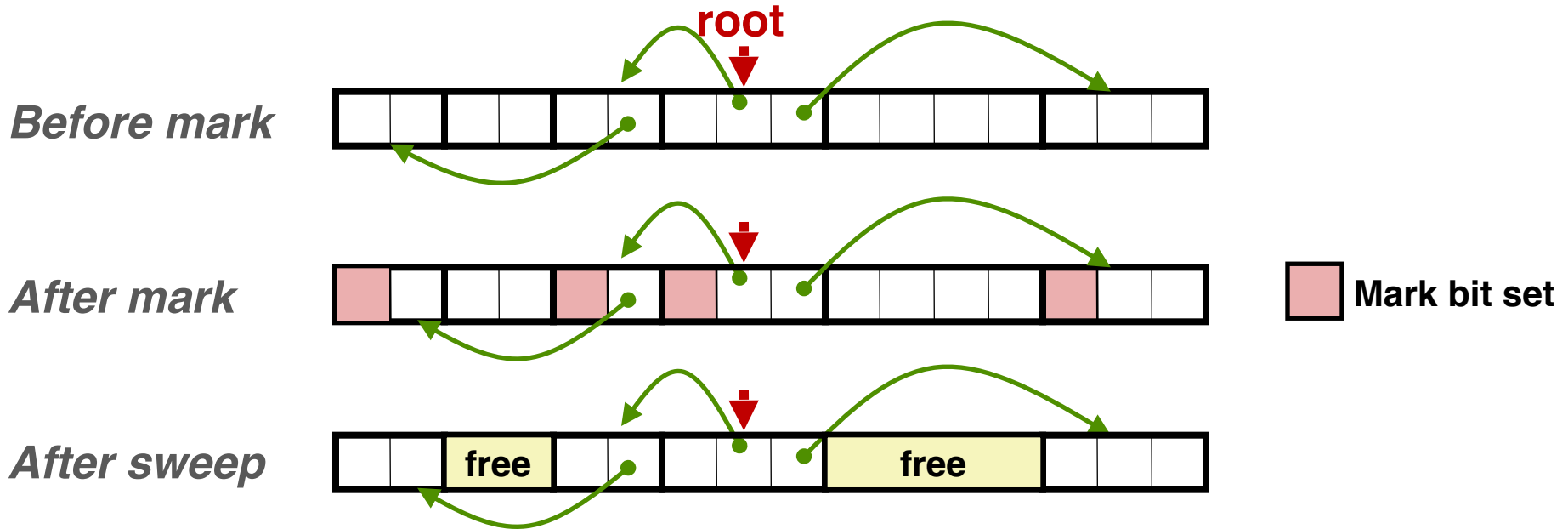
# Classical GC Algorithms

- 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.



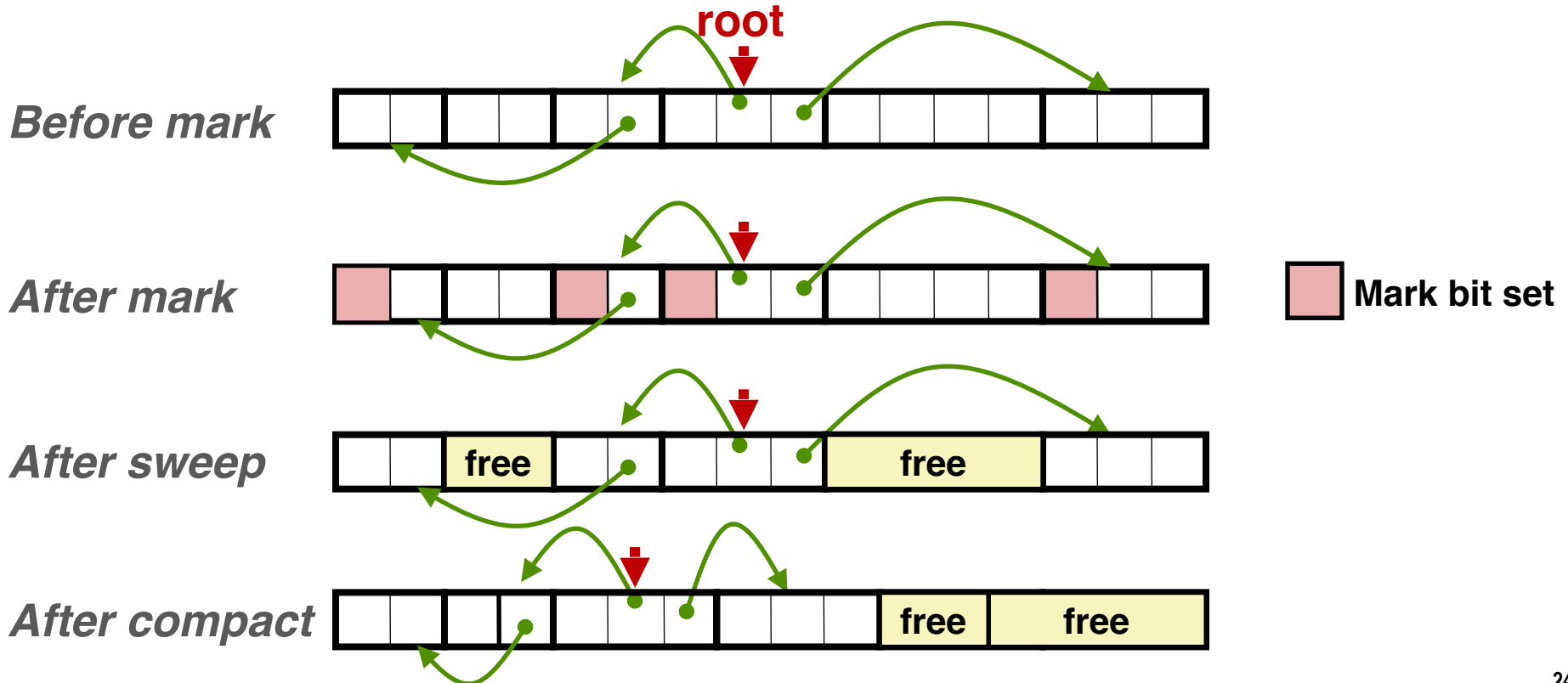
# Classical GC Algorithms

- 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

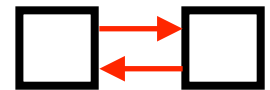
- All the GC algorithms described so far are tracing-based
  - Start from the root pointers, trace all the reachable objects
  - Need graph traversal. Different to implement.
- Reference counting (Collins, 1960)
  - Keep a counter for each object
  - Increment the counter if there is a new pointer pointing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

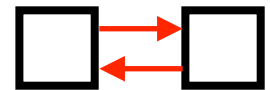
# Classical GC Algorithms

- 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



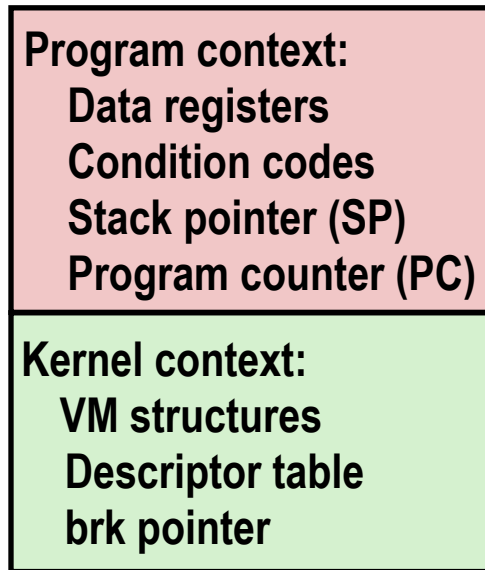
# Today

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

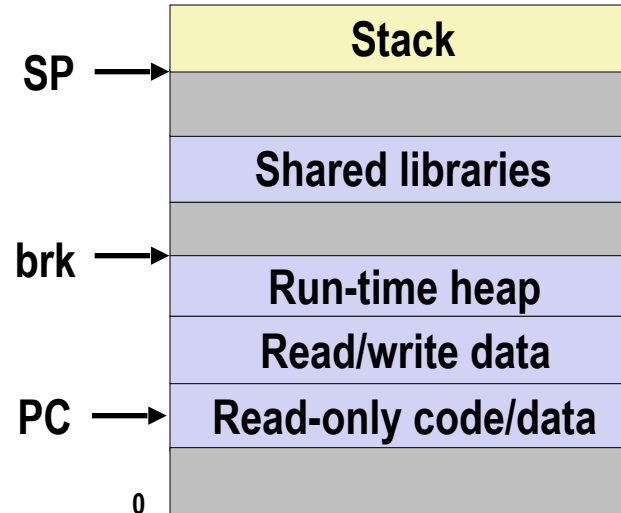
# Programmers View of A Process

- 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)**

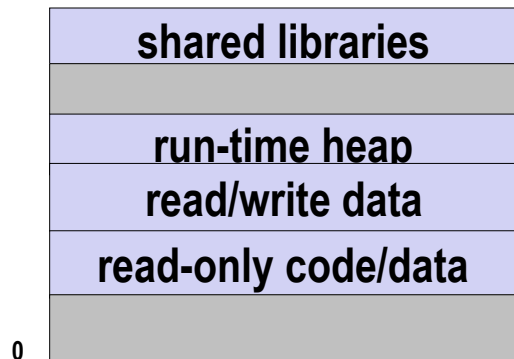
**Shared code and data**

stack 1

stack 2

Thread 1 context:  
Data registers  
Condition codes  
SP1  
PC1

Thread 2 context:  
Data registers  
Condition codes  
SP2  
PC2

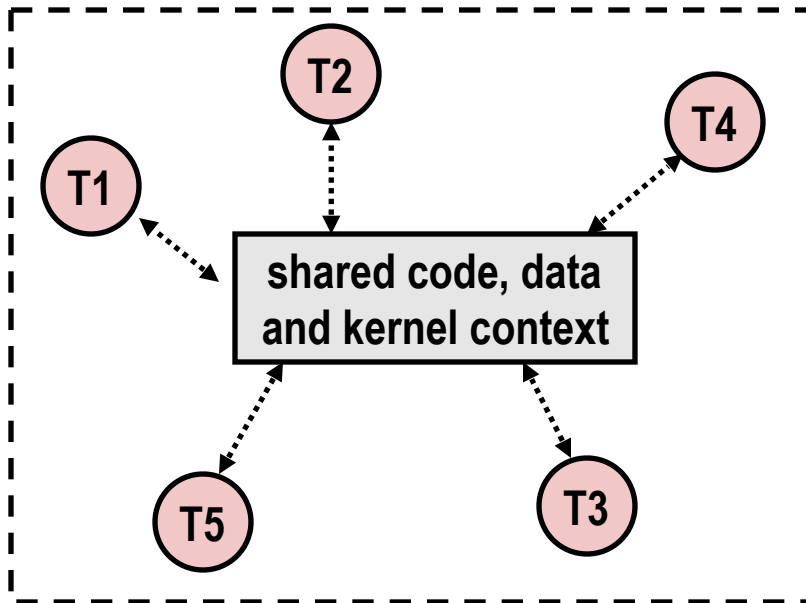


Kernel context:  
VM structures  
Descriptor table  
brk pointer

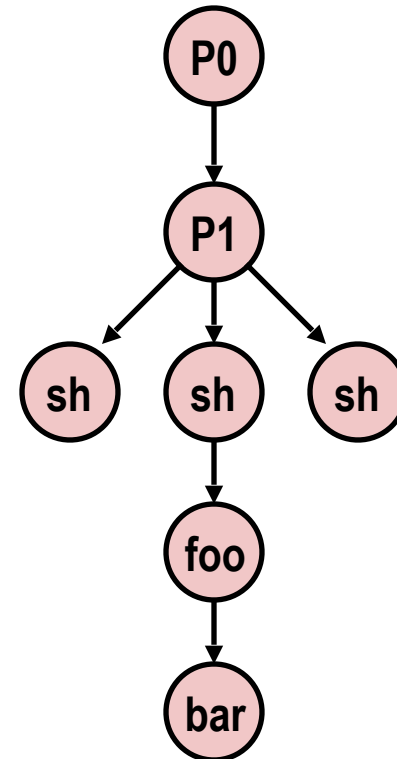
# Logical View of Threads

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

Threads associated with process foo



Process hierarchy





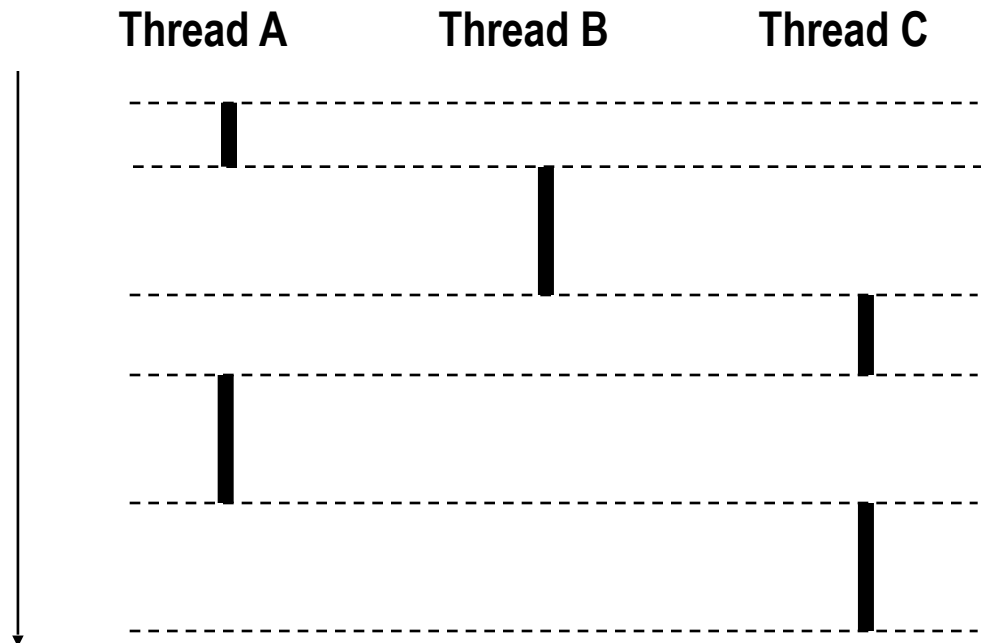
# Concurrent Threads

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

- Examples:

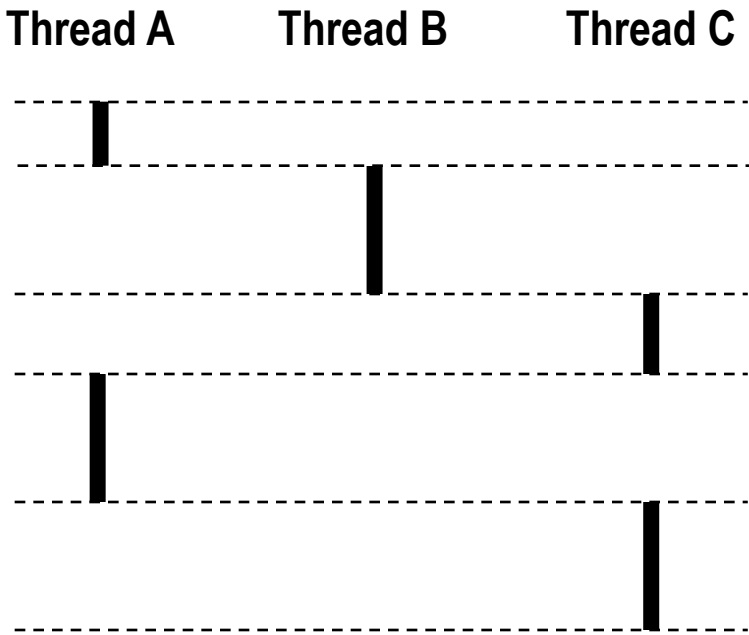
- Concurrent: A & B, A&C
- Sequential: B & C

Time

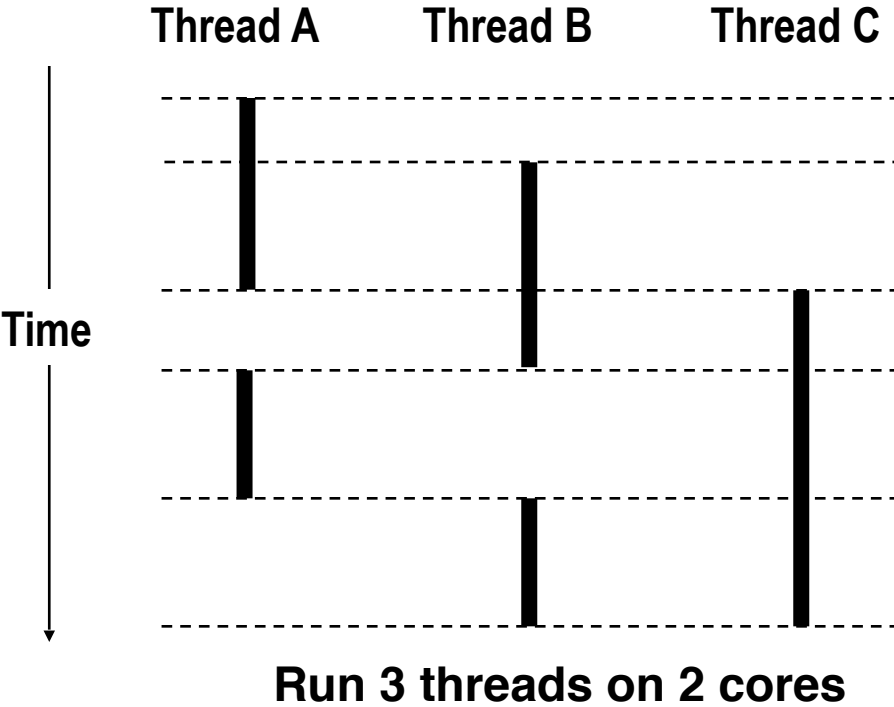


# Concurrent Thread Execution

- 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);  
}
```

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;  
}
```

Return value  
(void \*\*p)

hello.c

# Execution of Threaded “hello, world”

Main thread



# Execution of Threaded “hello, world”

Main thread

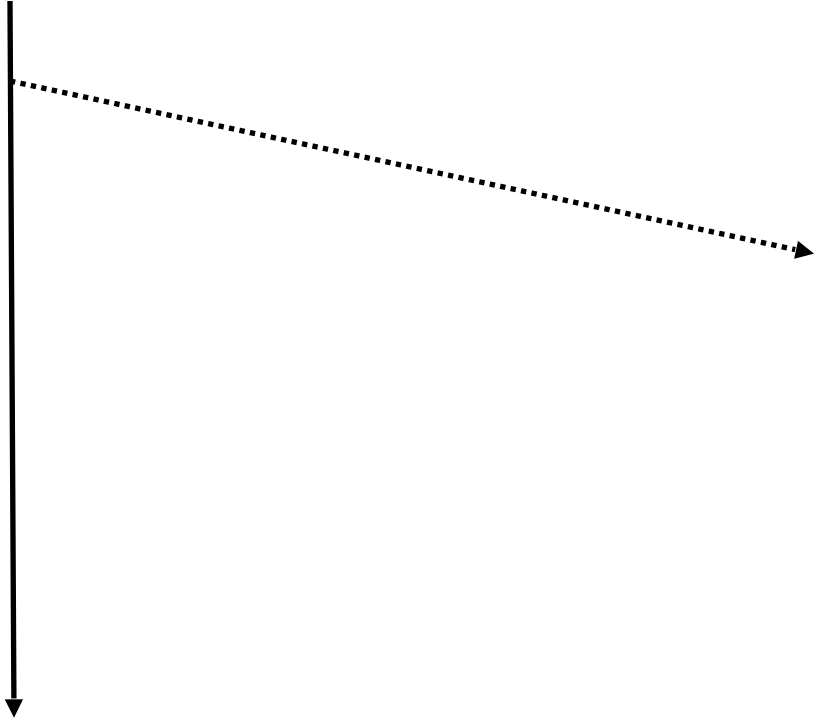
call Pthread\_create()



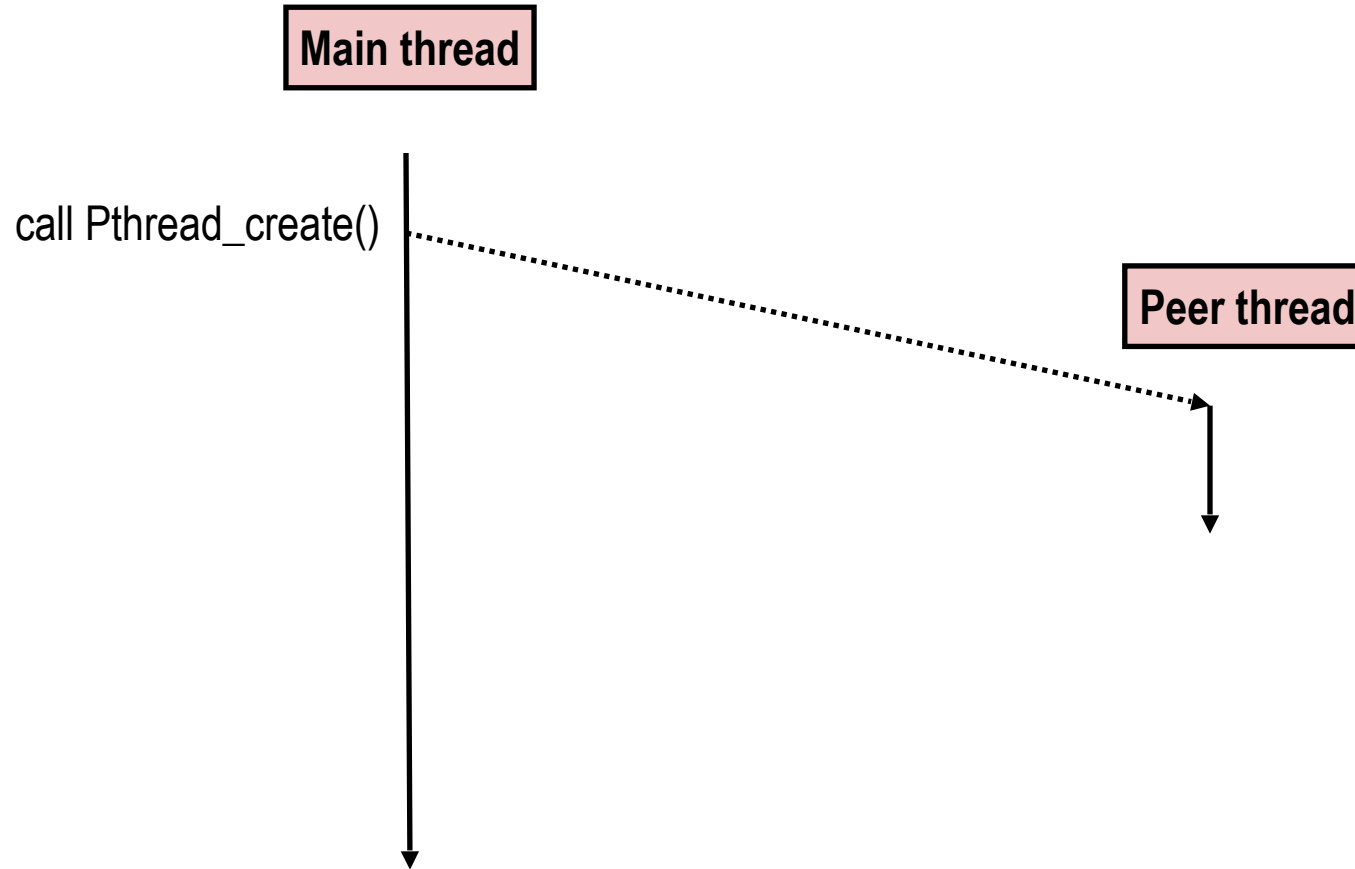
# Execution of Threaded “hello, world”

Main thread

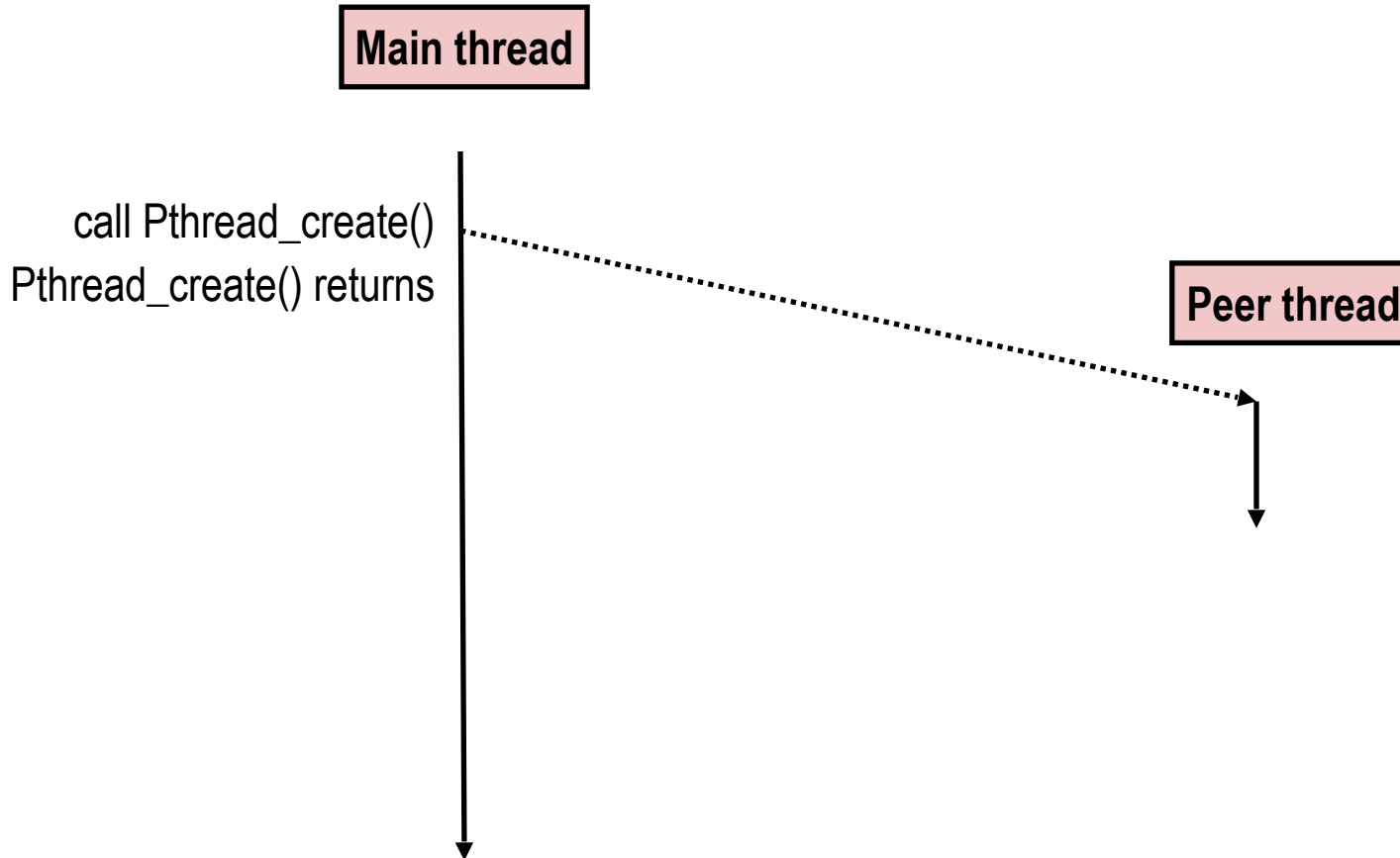
call Pthread\_create()



# Execution of Threaded “hello, world”

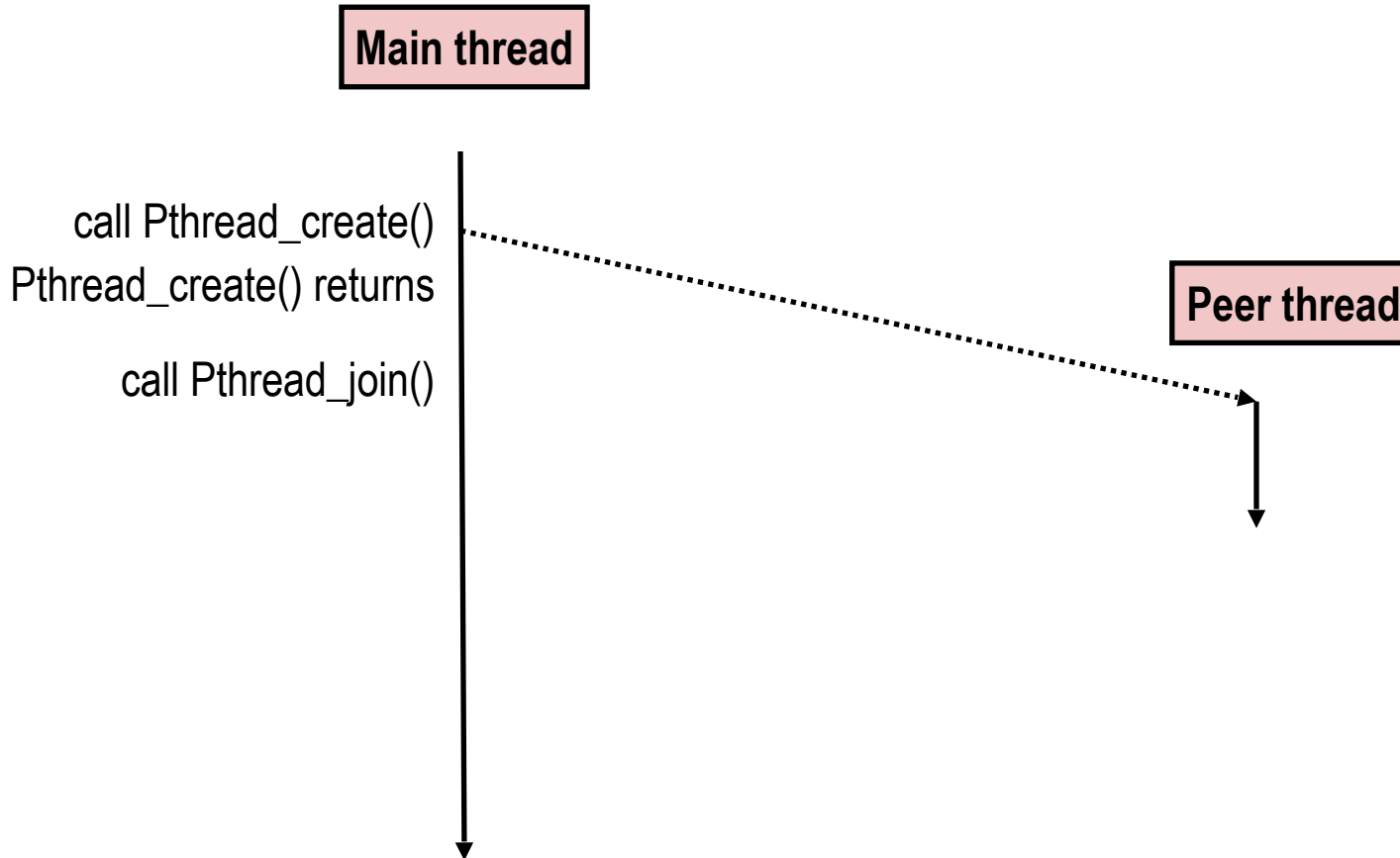


# Execution of Threaded “hello, world”

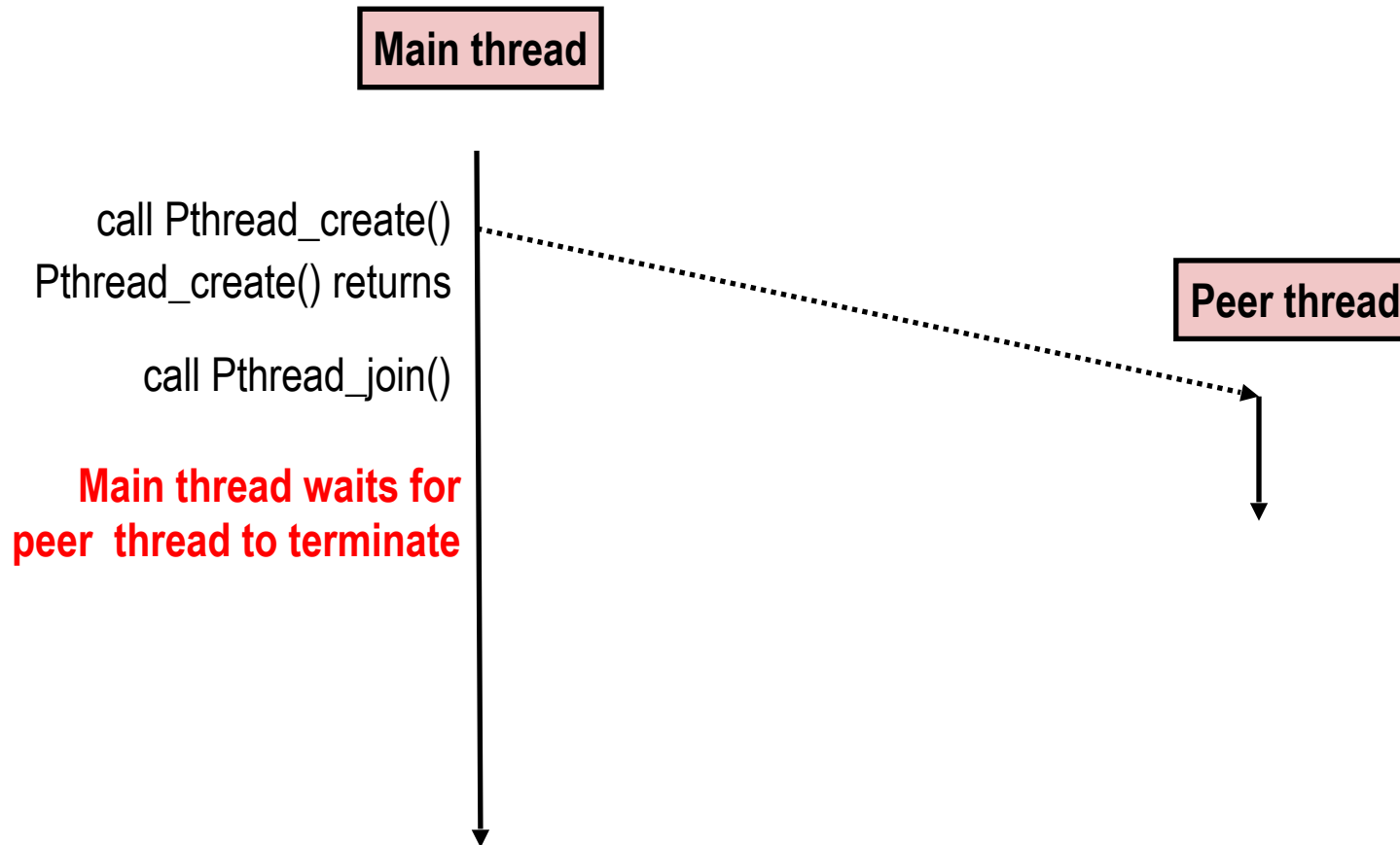




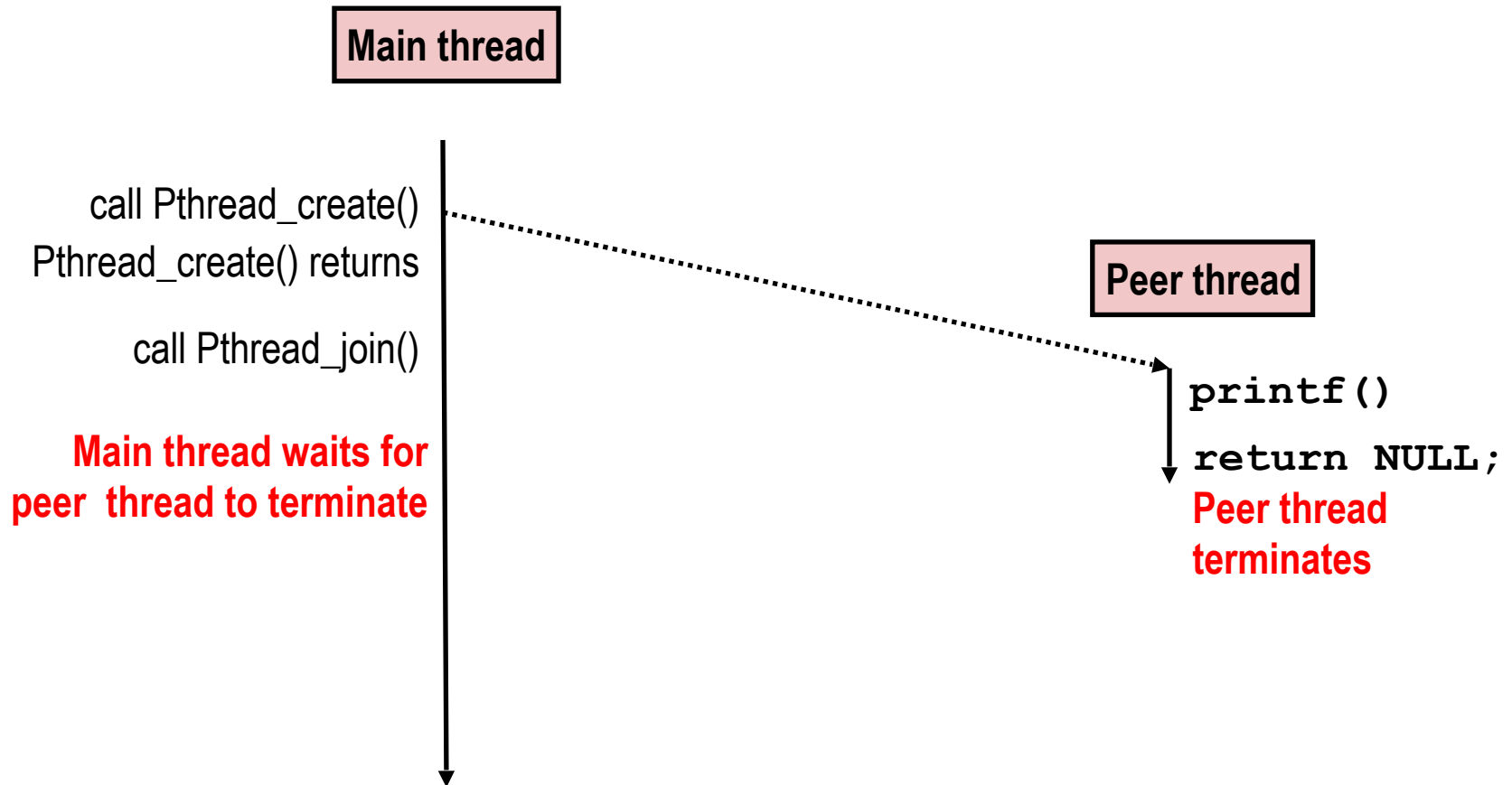
# Execution of Threaded “hello, world”



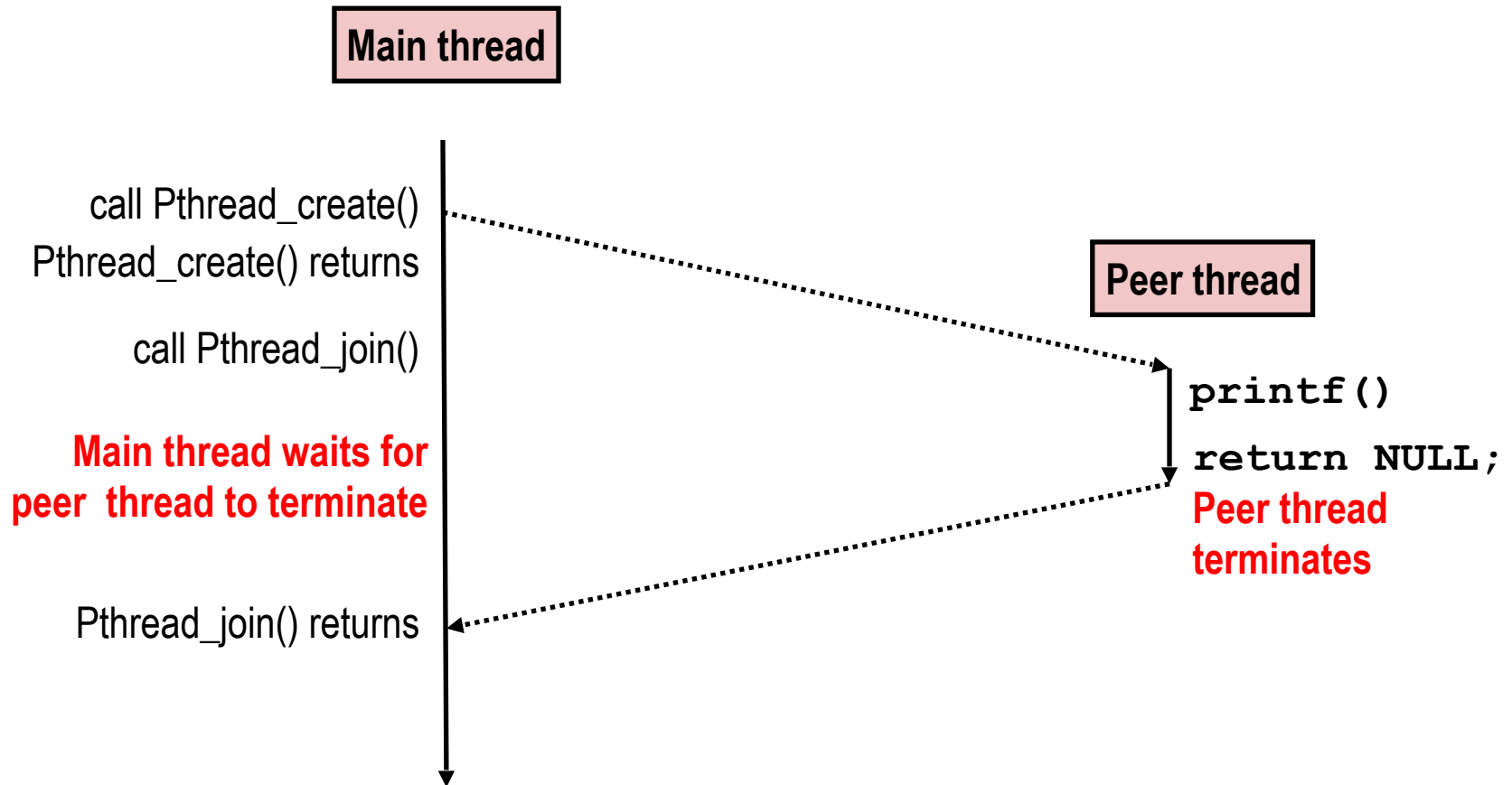
# Execution of Threaded “hello, world”



# Execution of Threaded “hello, world”



# Execution of Threaded “hello, world”



# Execution of Threaded “hello, world”

