

# Operating Systems

## (Chapter 20 ~ 22 and VM)

Memory

Young-Woo Kwon

# **MEMORY (PAGING AND VM)**

# Status Check

- At this point, we have a full-featured virtual memory system
  - Transparent, supports protection and isolation
  - Fast (via TLBs)
  - Space efficient (via multi-level tables)
- Are we done?
  - No!
- What if we completely run out of physical memory?
  - Can virtualization help?

# RAM as a Cache

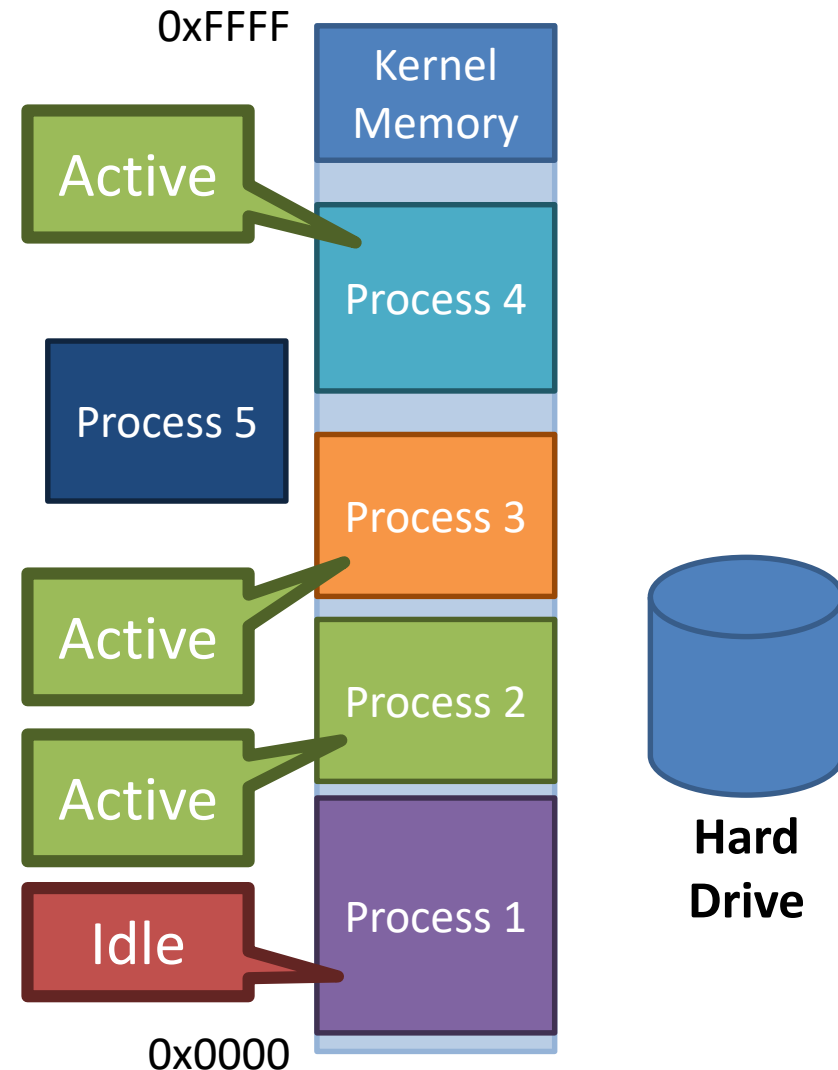
- RAM can be viewed as a high-speed cache for your large-but-slow spinning disk storage
  - You have GB of programs and data
  - Only a subset can fit in RAM at any given time
- Ideally, you want the most important things to be resident in the cache (RAM)
  - Code/data that become less important can be evicted back to the disk

# Swap Space

- Key idea: take frames from physical memory and swap (write) them to disk
  - This frees up space for other code and data
- Load data from swap back into memory on-demand
  - If a process attempts to access a page that has been swapped out...
  - A page-fault occurs and the instruction pauses
  - The OS can swap the frame back in, insert it into the page table, and restart the instruction

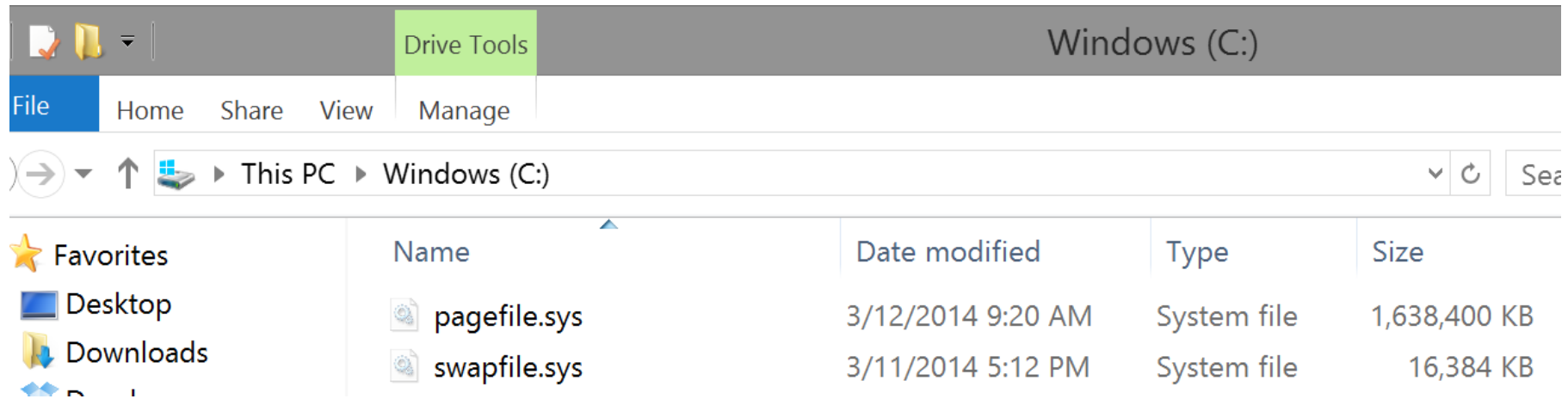
# Swapping Example

- Suppose memory is full
- The user opens a new program
- Swap out idle pages to disk
- If the idle pages are accessed, page them back in



# All Modern OSes Support Swapping

- On Linux, you create a *swap partition* along with your normal ext3/4 filesystem
  - Swapped pages are stored in this separate partition
- Windows



# Implementing Swap

1. **Data structures** are needed to track the mapping **between pages in memory and pages on disk**
2. **Meta-data** about memory pages must be kept
  - **When** should pages be evicted (swapped to disk)?
  - How do you choose **which page** to evict?
3. The functionality of the OSes **page fault handler** must be modified



# x86 Page Table Entry, Again

- On x86, page table entries (PTE) are 4 bytes

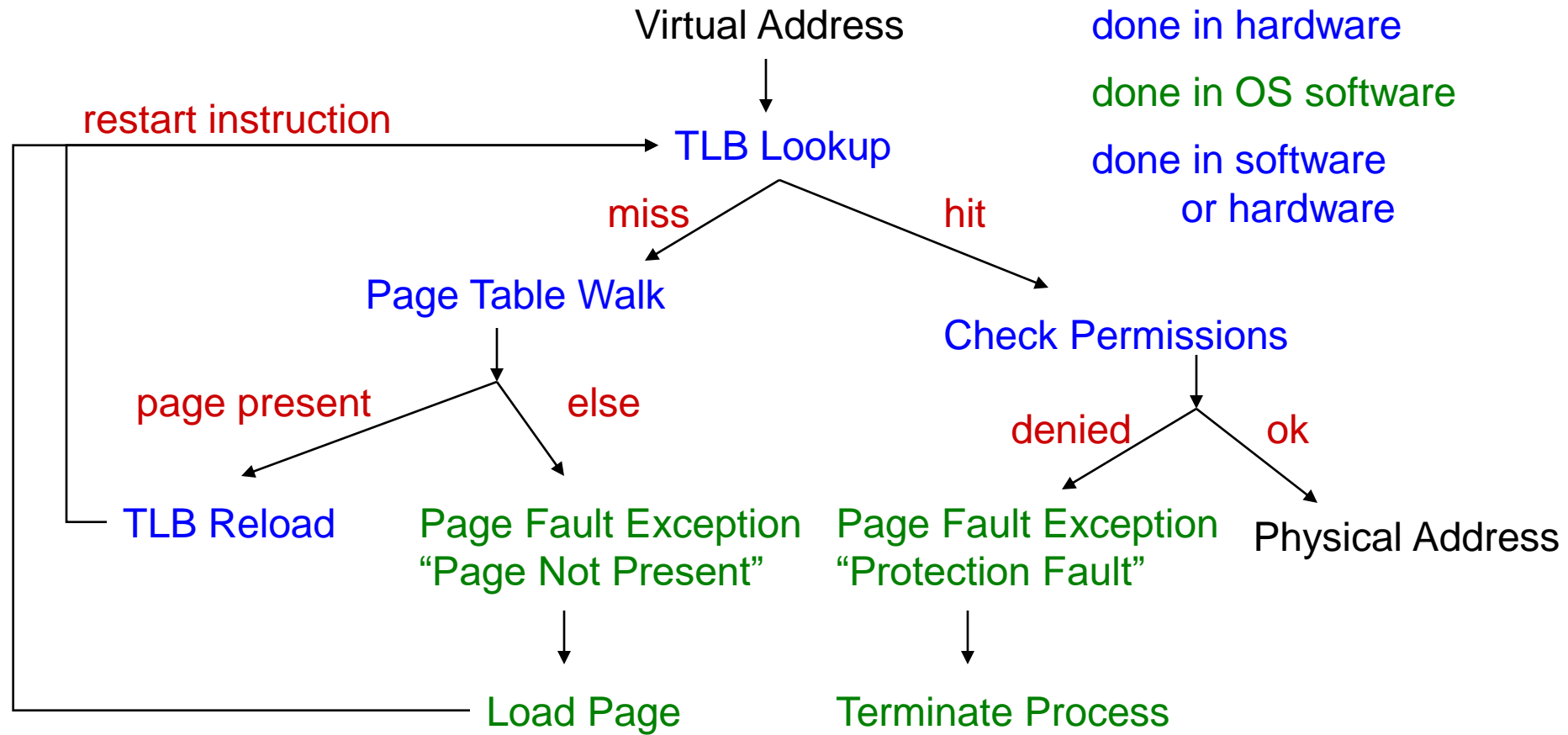
31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

- P (**present bit**) – is this page in physical memory?
  - OS sets or clears the present bit based on its swapping decisions
    - 1 means the page is in physical memory
    - 0 means the page is valid, but has been swapped to disk
  - Attempts to access an invalid page or a page that isn't present **trigger a page fault**

# Handling Page Faults

- Thus far, we have viewed **page faults as bugs**
  - i.e. when a process tries to access an invalid pointer
  - The OS kills the process that generate page faults
- However, now handling page faults is more complicated
  - **If the PTE is invalid, the OS still kills the process**
  - **If the PTE is valid, but **present = 0**, then**
    1. The OS swaps the page back into memory
    2. The OS updates the PTE
    3. The OS instructs the CPU to retry the last instruction

# Address Translation & TLB



# When Should the OS Evict Pages?

- Memory is finite, so when should pages be swapped?
- **On-demand approach**
  - If a page needs to be created and no free pages exist, swap a page to disk
- **Proactive approach**
  - Most OSes try to maintain a small pool of free pages
  - Implement a high watermark
  - Once physical memory utilization crosses the high watermark, a background process starts swapping out pages

# What Pages Should be Evicted?

- Known as the page-replacement policy
- What is the optimal eviction strategy?
  - Evict the page that **will be accessed furthest in the future**
  - Provably results in the **maximum cache hit rate**
  - Unfortunately, **impossible to implement in practice**
- Practical strategies for selecting which page to swap to disk
  - **FIFO**
  - **Random**
  - **LRU (Least recently used)**
- **Same fundamental algorithms** as in TLB eviction

# Examples of Optimal and LRU

**Assume the cache can store 3 pages**

**Optimal (Furthest in the Future)**

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

**FIFO**

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	2	3, 0, 1
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

# Examples of Optimal and LRU

**Assume the cache can store 3 pages**

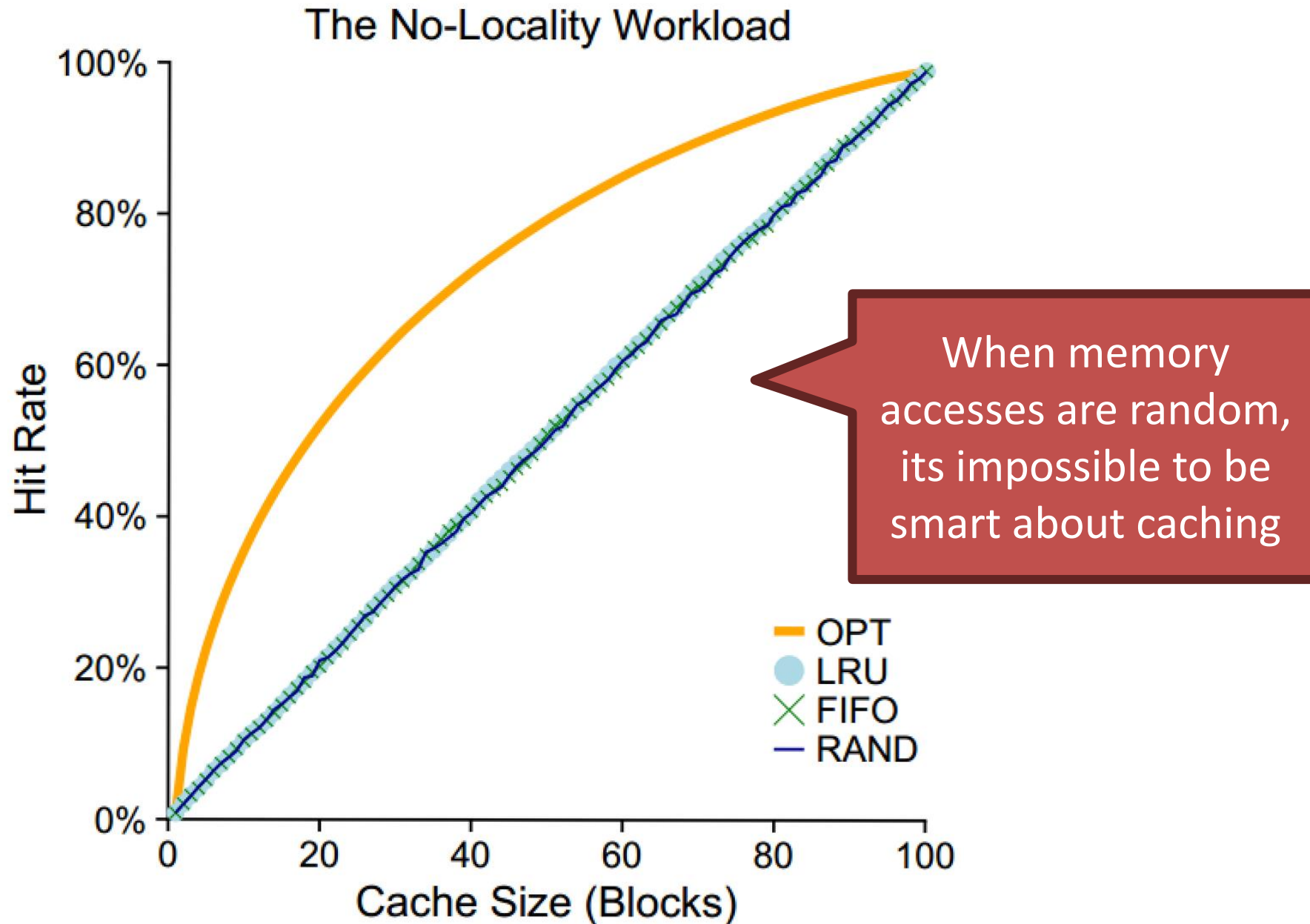
**Optimal (Furthest in the Future)**

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
0	Hit		0, 1, 2

**LRU**

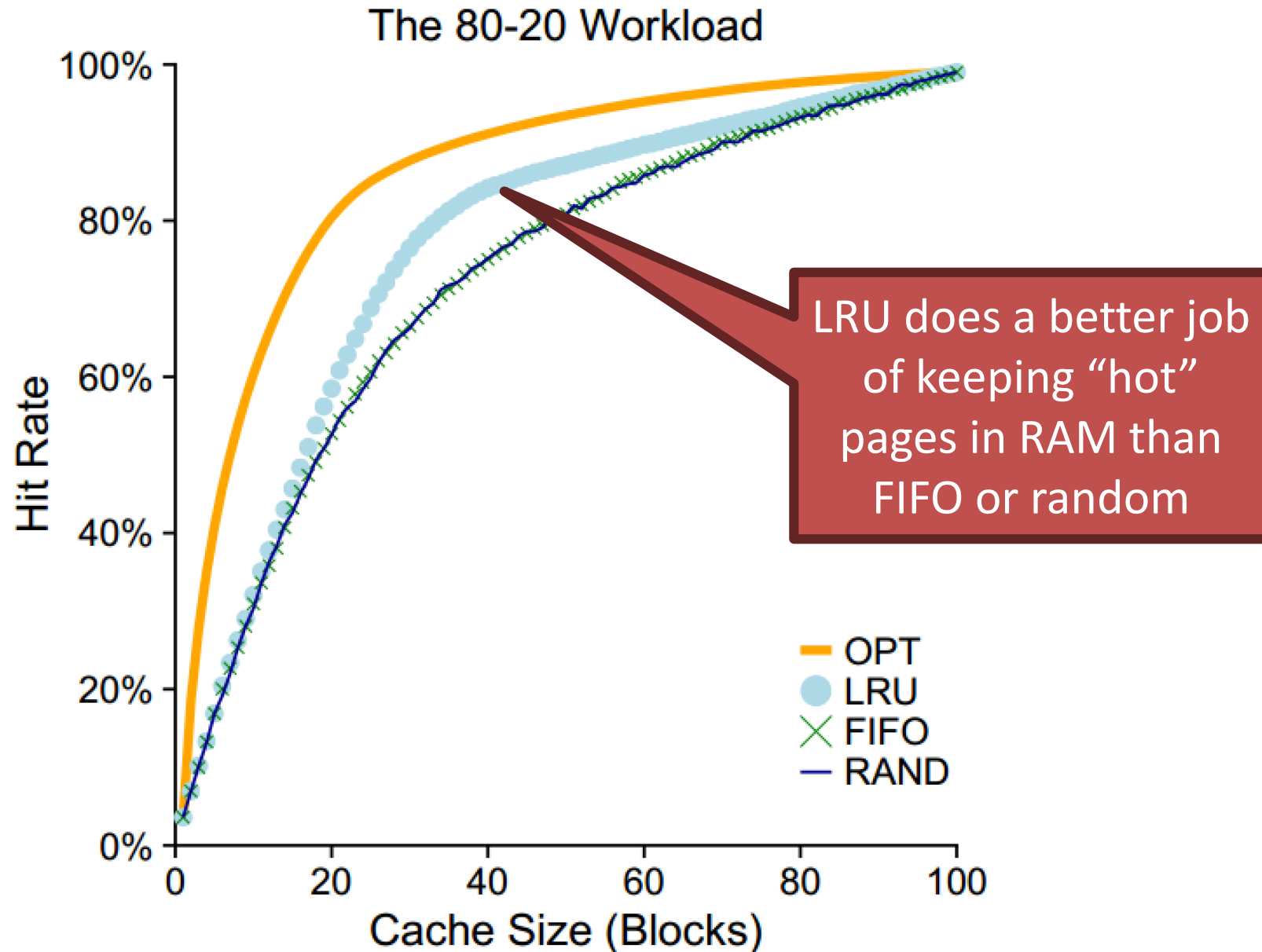
Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	0	1, 2, 3
0	Miss	3	0, 1, 2

- All memory accesses are to 100% random pages

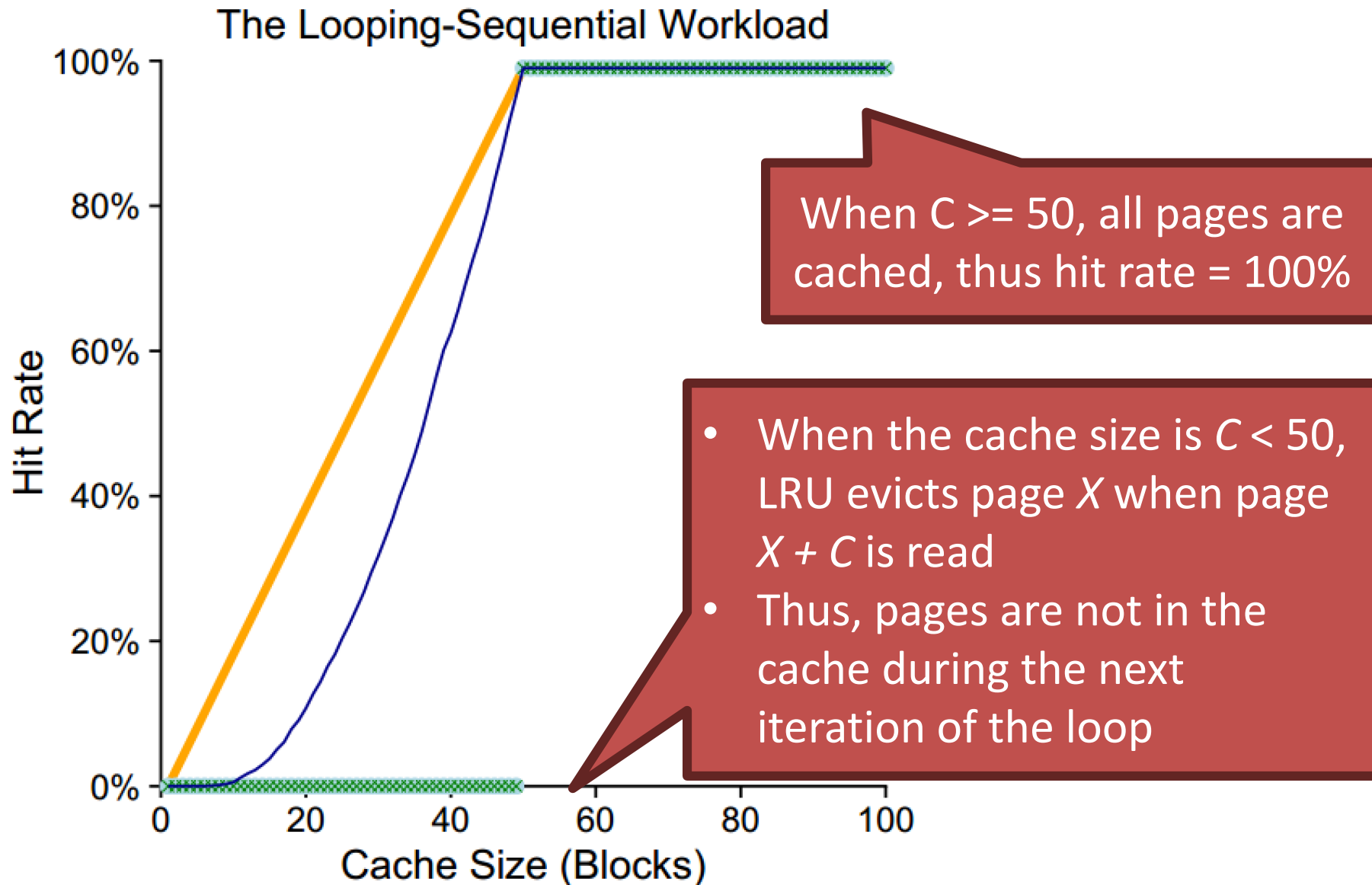




- 80% of memory accesses are for 20% of pages



- The process sequentially accesses one memory address in 50 pages, then loops



# Implementing Historical Algorithms

- LRU has high cache hit rates in most cases...
- ... but how do we know which pages have been recently used?
- **Strategy 1: record each access to the page table**
  - Problem: adds additional overhead to page table lookups
- **Strategy 2: approximate LRU with help from the hardware**

# x86 Page Table Entry, Again

- On x86, page table entries (PTE) are 4 bytes

31 - 12	11 - 9	8	7	6	5	4	3	2	1	0
Page Frame Number (PFN)	Unused	G	PAT	D	A	PCD	PWT	U/S	W	P

- Bits related to swapping
  - A – **accessed bit** – has this page been read recently?
  - D – **dirty bit** – has this page been written recently?
  - The MMU sets the accessed bit when it reads a PTE
  - The MMU sets the dirty bit when it writes to the page referenced in the PTE
  - The OS may clear these flags as it wishes

# Approximating LRU

- The accessed and dirty bits tell us which pages have been recently accessed
- But, LRU is still difficult to implement
  - On eviction, LRU needs to scan all PTEs to determine which have not been used
  - But there are millions of PTEs!

# When virtual memory works well

- Locality
  - 80% of accesses are to 20% of pages
  - 80% of accesses are made by 20% of code
- Temporal locality:
  - Page that's accessed will be accessed again in near future
- Spatial locality:
  - Prefetching pays off: if a page is accessed, neighboring page will be accessed
- If VM works well, average access to all memory is about as fast as access to physical memory

# Thrashing: When Virtual Memory Does Not Work Well

- System accesses a page, evicts another page from its frame, and next access goes to just-evicted page which must be brought in
- Worst case a phenomenon called **Thrashing**
  - leads to constant swap-out/swap-in
  - 100% disk utilization, but no process makes progress
    - CPU most idle, memory mostly idle

# What to do about Thrashing?

- Buy more memory
  - ultimately have to do that
  - increasing memory sizes ultimately reason why thrashing is nowadays less of a problem than in the past – still OS must have strategy to avoid worst case
- Ask user to kill process
- Let OS decide to kill processes that are thrashing
  - Linux has an option to do that (see next slide)
- In many cases, still: reboot only time-efficient option
  - But OS should have reasonable strategy to avoid it if it can



An aircraft company discovered that it was cheaper to fly its planes with less fuel on board. The planes would be lighter and use less fuel and money was saved. On rare occasions however the amount of fuel was insufficient, and the plane would crash. This problem was solved by the engineers of the company by the development of a special OOF (out-of-fuel) mechanism. In emergency cases a passenger was selected and thrown out of the plane. (When necessary, the procedure was repeated.) A large body of theory was developed and many publications were devoted to the problem of properly selecting the victim to be ejected. Should the victim be chosen at random? Or should one choose the heaviest person? Or the oldest? Should passengers pay in order not to be ejected, so that the victim would be the poorest on board? And if for example the heaviest person was chosen, should there be a special exception in case that was the pilot? Should first class passengers be exempted? Now that the OOF mechanism existed, it would be activated every now and then, and eject passengers even when there was no fuel shortage. The engineers are still studying precisely how this malfunction is caused.

Source: [\*Ikml \(Andries Brouwer\), 2004\*](#)

# **IMPLICIT MEMORY MANAGEMENT**

# Implicit Memory Management

- Motivation: manually (or explicitly) reclaiming memory is difficult:
  - Too early: risk access-after-free errors
  - Too late: memory leaks
- Requires principled design
  - Programmer must reason about ownership of objects
- Difficult & error prone, especially in the presence of object sharing
- Complicates design of APIs

# Garbage Collection

- Invented in 1959
- Automatic memory management
  - The GC reclaims memory occupied by objects that are no longer in use
  - Such objects are called garbage
- Conceptually simple
  - Scan objects in memory, identify objects that cannot be accessed (now, or in the future)
  - Reclaim these garbage objects
- In practice, very tricky to implement

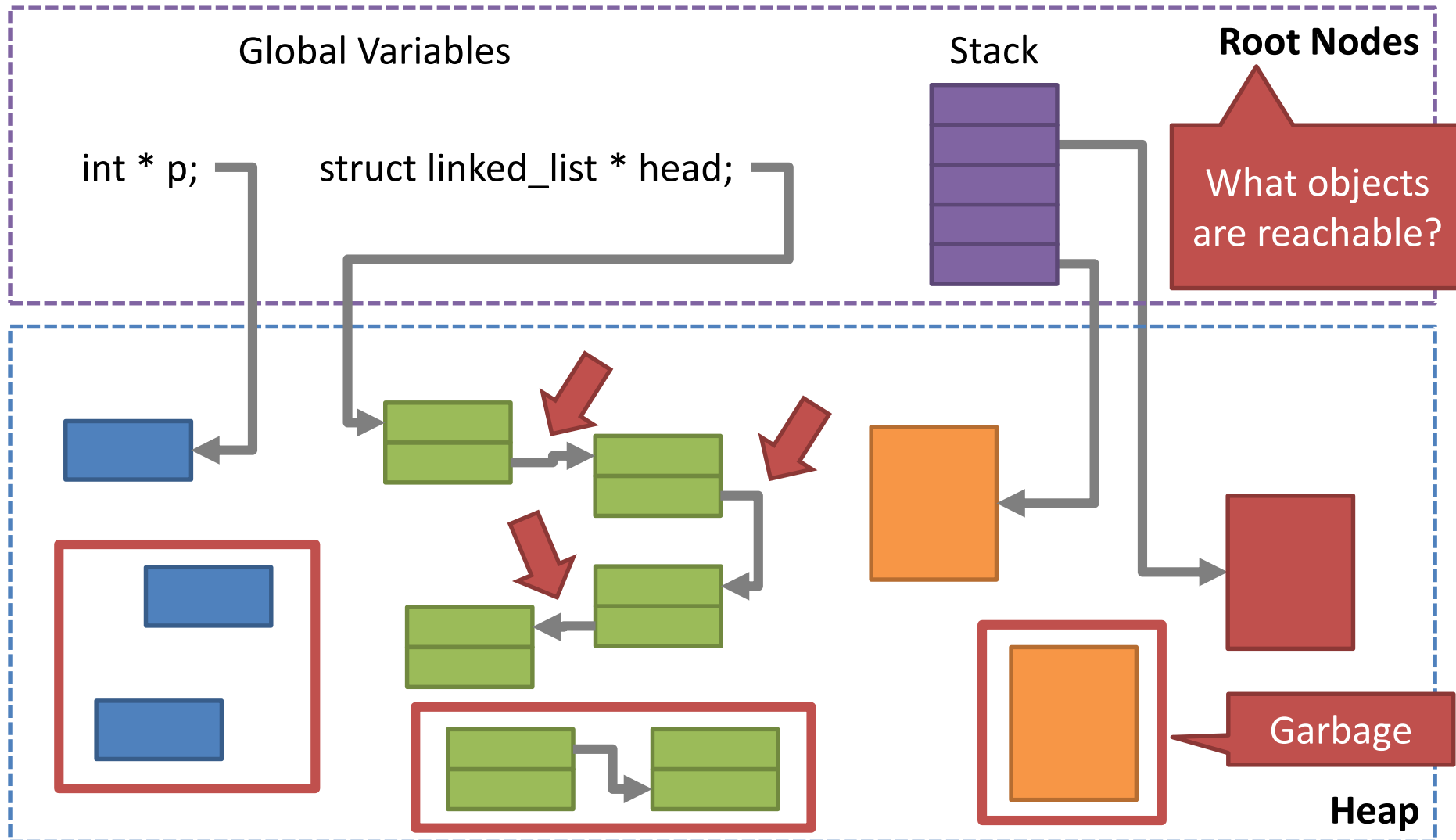
# Manual Reference Counting

- Idea: keep track of how many references there are to each object in a reference counter stored with each object
  - Copy a reference to an object globalvar = q
    - increment count: “addref”
  - Remove a reference p = NULL
    - decrement count: “release”
- Uses set of rules programmers must follow
  - E.g., must ‘release’ reference obtained from OUT parameter in function call
  - Must ‘addref’ when storing into global
  - May not have to use addref/release for references copied within one function
- Programmer must use addref/release correctly
  - Still somewhat error prone, but rules are such that correctness of the code can be established locally without consulting the API documentation of any functions being called; parameter annotations (IN, INOUT, OUT, return value) imply reference counting rules

# Automatic Reference Counting

- Idea: *force* automatic reference count updates when pointers are assigned/copied
- Most common variant:
  - C++ allows programmer to interpose on assignments and copies via operator overloading/special purpose constructors.
- Disadvantage of all reference counting schemes is their inability to handle cycles
  - But great advantage is immediate reclamation: no “drag” between last access & reclamation

# Garbage Collection Concepts



# Approaches to GC

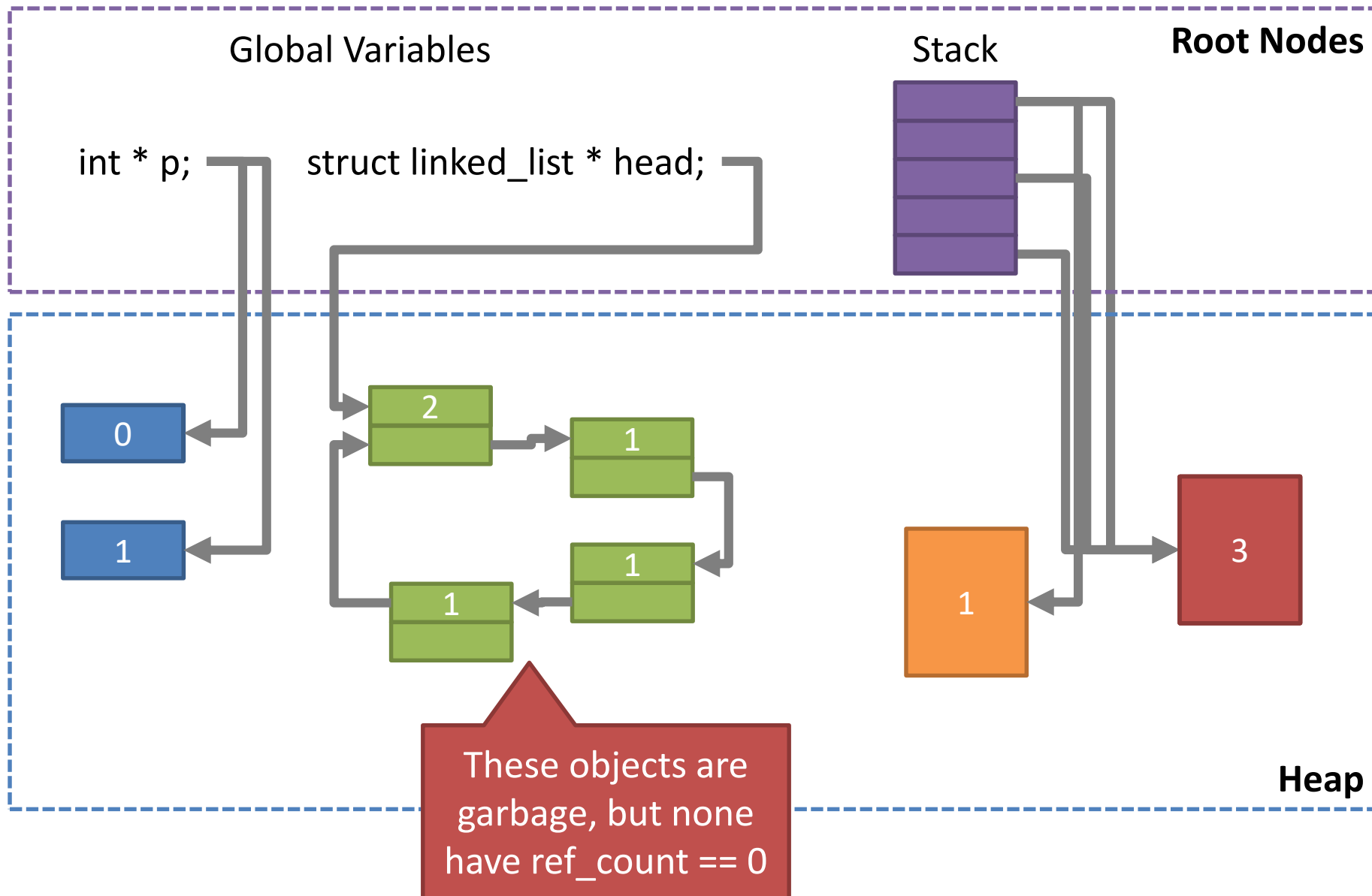
- Reference Counting
  - Each object keeps a count of references
  - If an objects count == 0, it is garbage
- Mark and Sweep
  - Starting at the roots, traverse objects and “mark” them
  - Free all unmarked objects on the heap
- Copy Collection
  - Extends mark & sweep with compaction
  - Addresses CPU and external fragmentation issues
- Generational Collection
  - Uses heuristics to improve the runtime of mark & sweep



# Reference Counting

- Key idea: each object includes a `ref_count`
  - Assume `obj * p = NULL;`
  - `p = obj1; // obj1->ref_count++`
  - `p = obj2; // obj1->ref_count--, obj2->ref_count++`
- If an object's `ref_count == 0`, it is garbage
  - No pointers target that object
  - Thus, it can be safely freed

# Reference Counting Example



# Pros and Cons of Reference Counting

## The Good

- Relatively easy to implement
- Easy to conceptualize

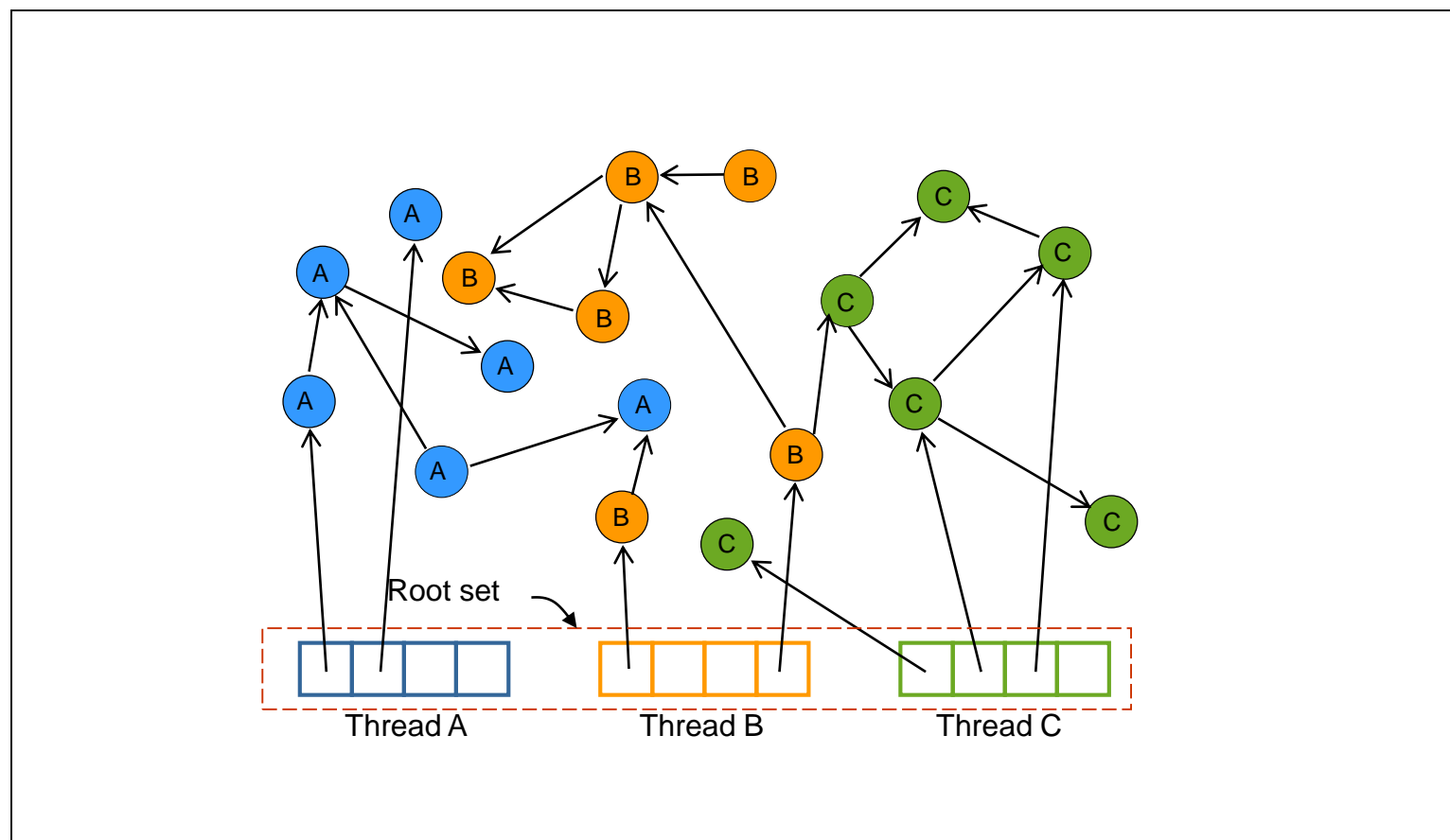
## The Bad

- Not guaranteed to free all garbage objects
- Additional overhead (`int ref_count`) on all objects
- Access to `obj->ref_count` must be synchronized

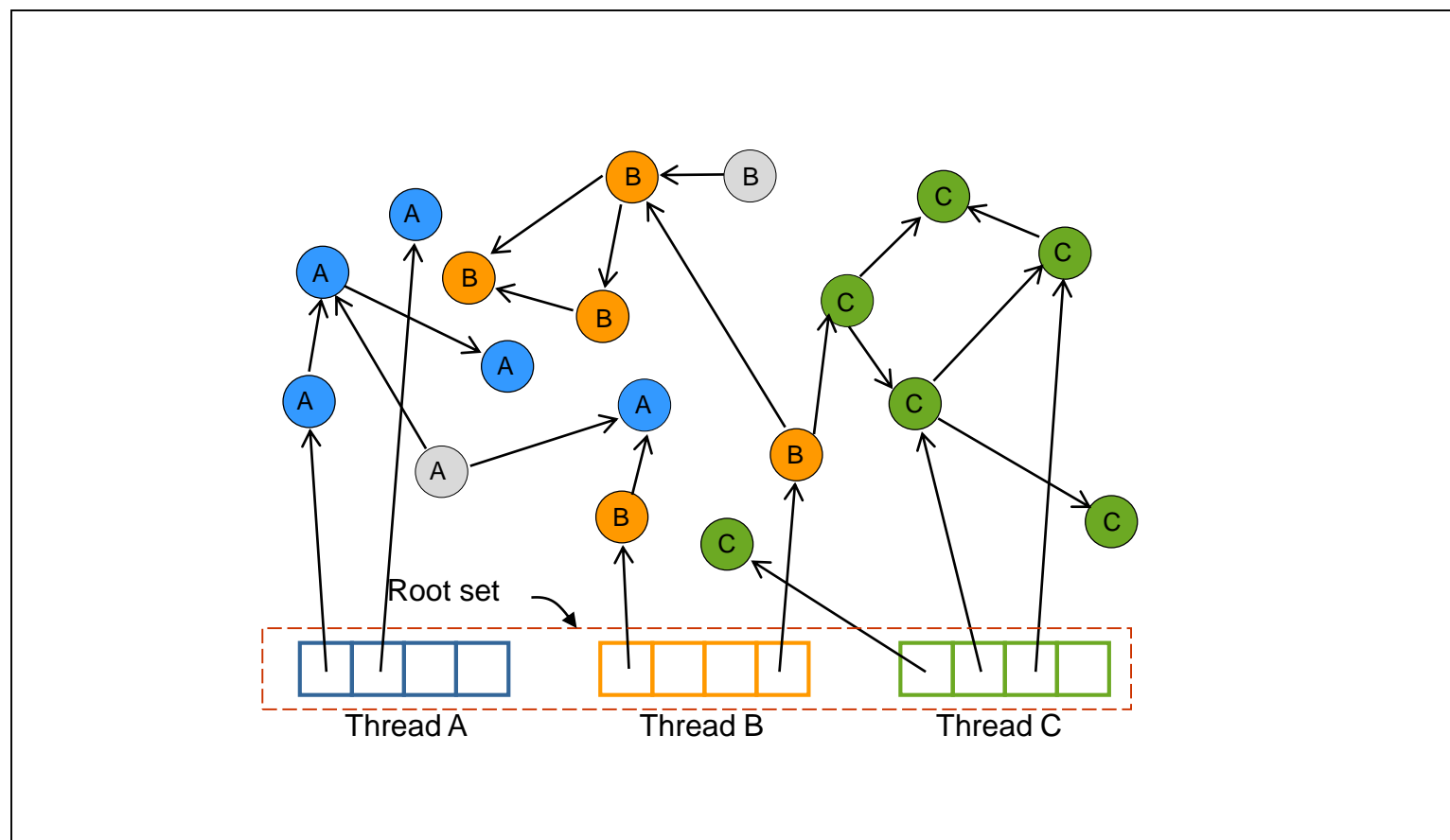
# Mark and Sweep

- Key idea: periodically scan all objects for reachability
  - Start at the roots
  - Traverse all reachable objects, mark them
  - All unmarked objects are garbage

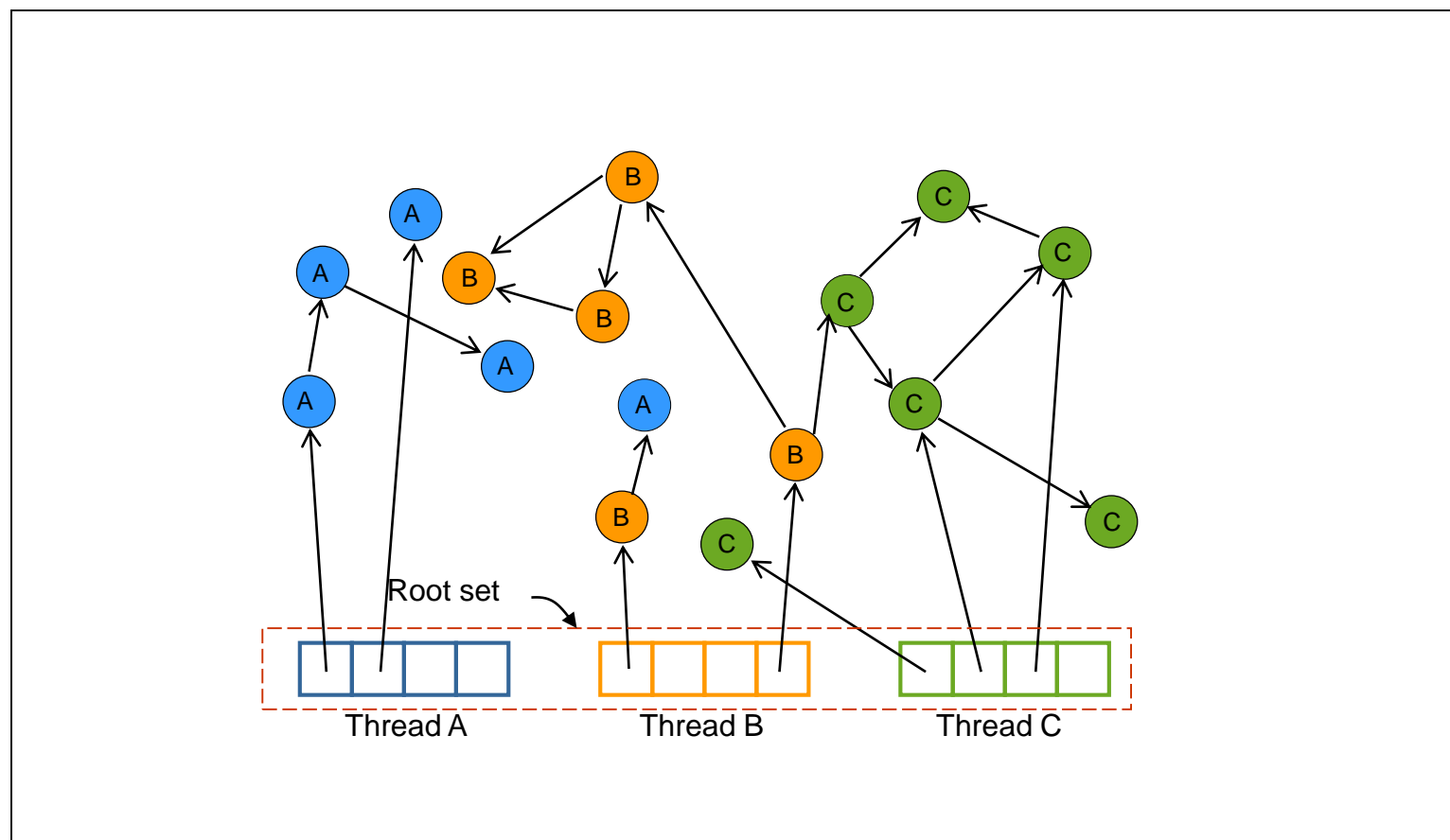
# Reachability Graph



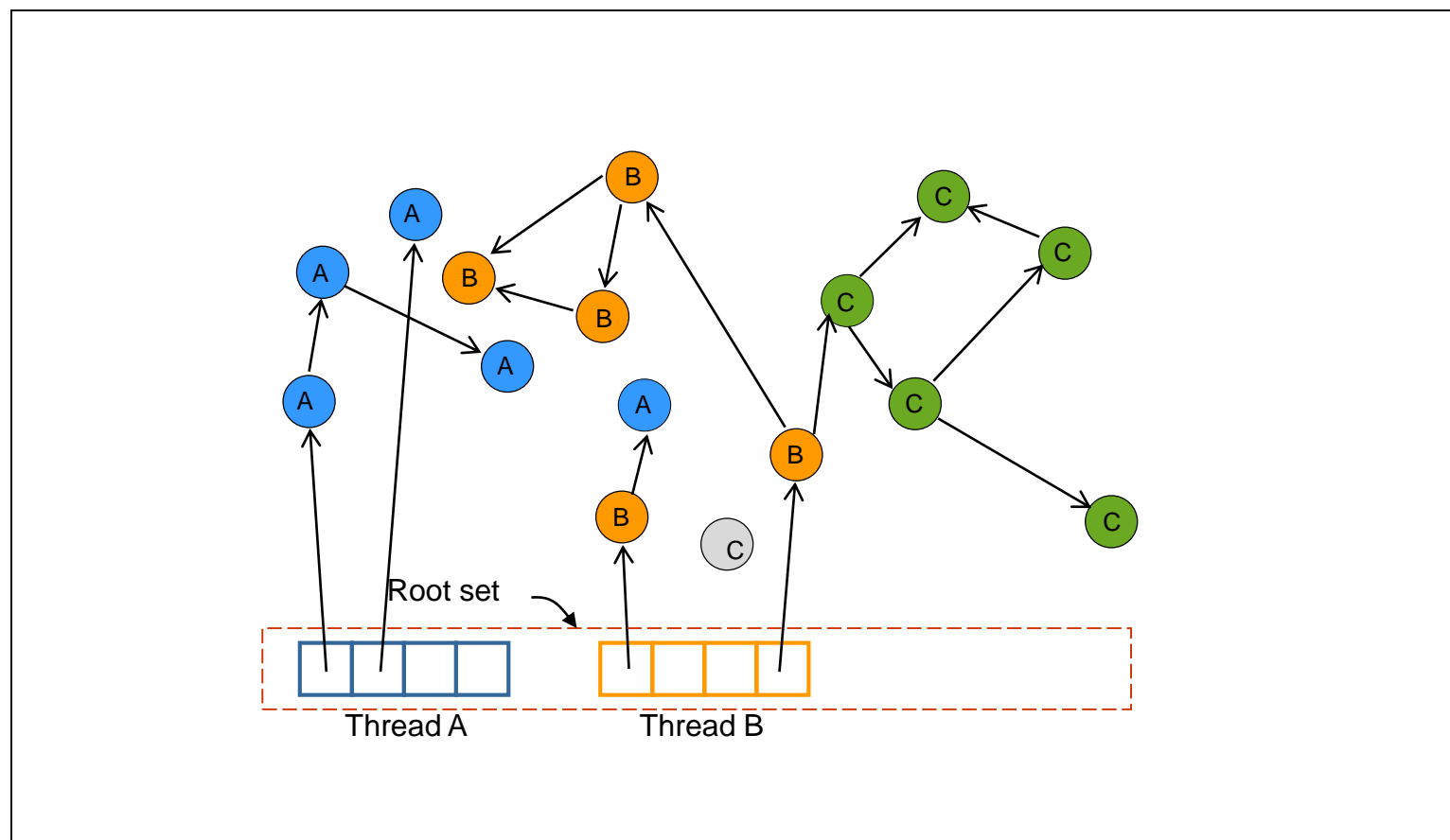
# Reachability Graph



# Reachability Graph

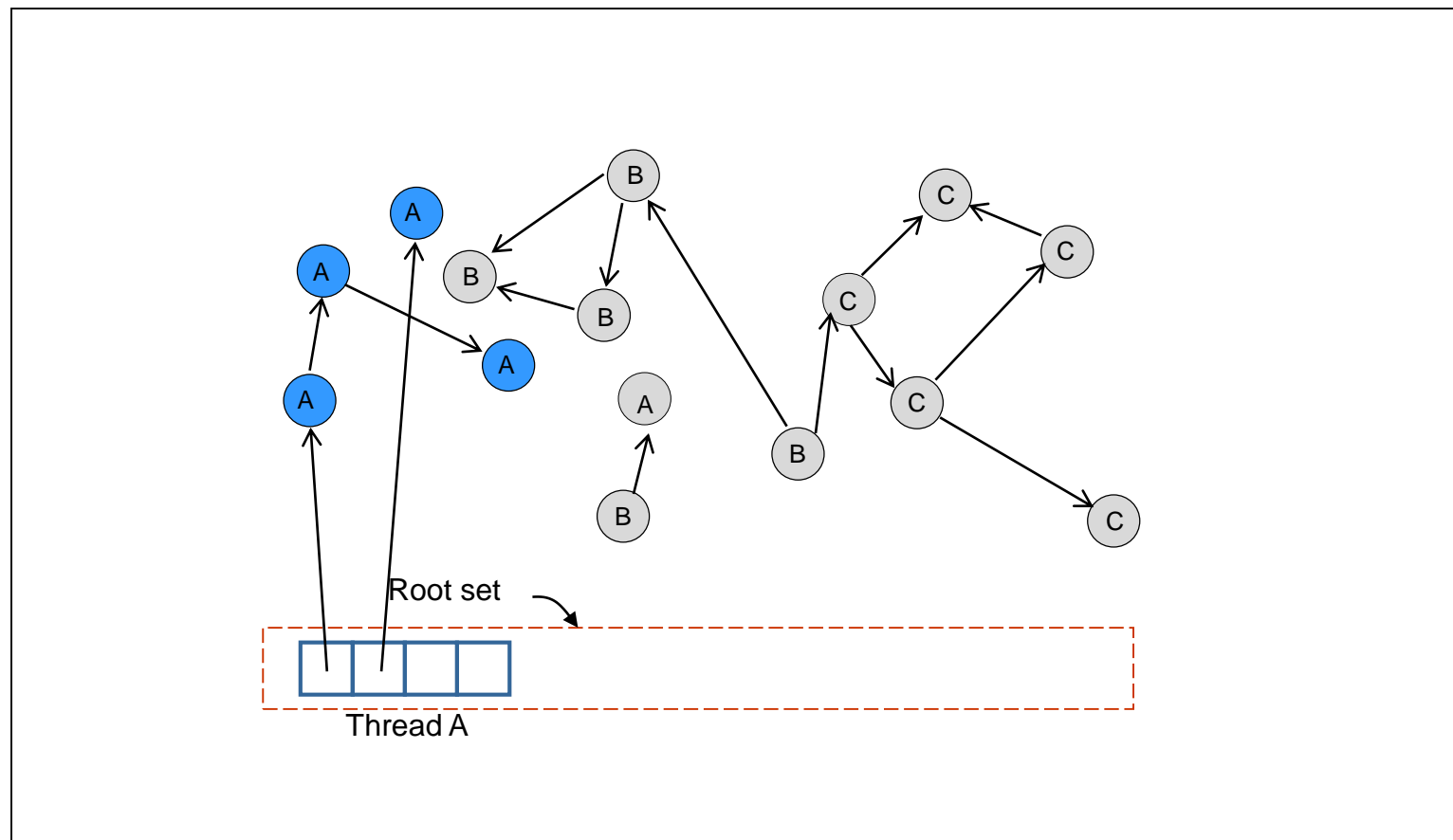


# Reachability Graph

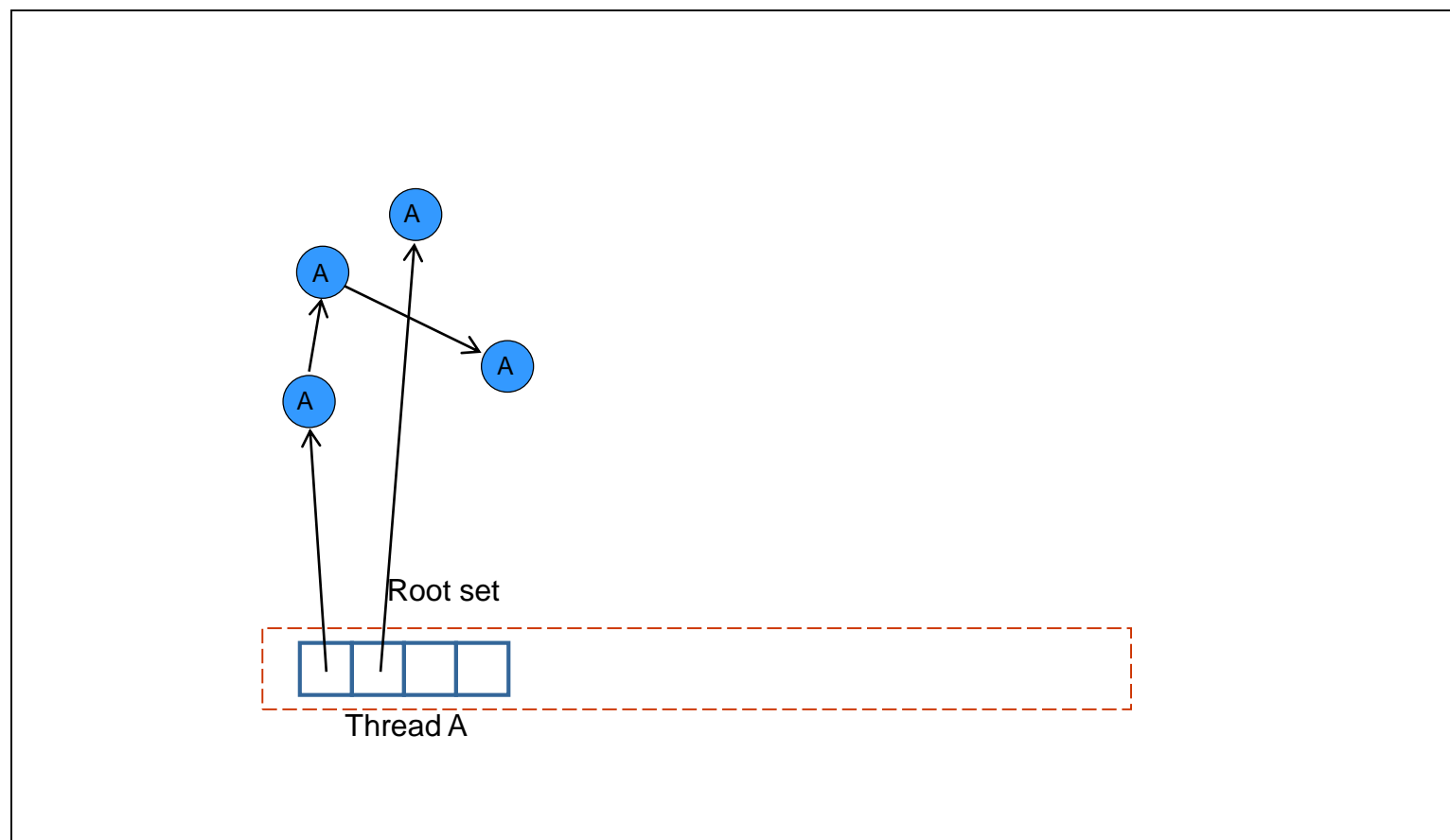




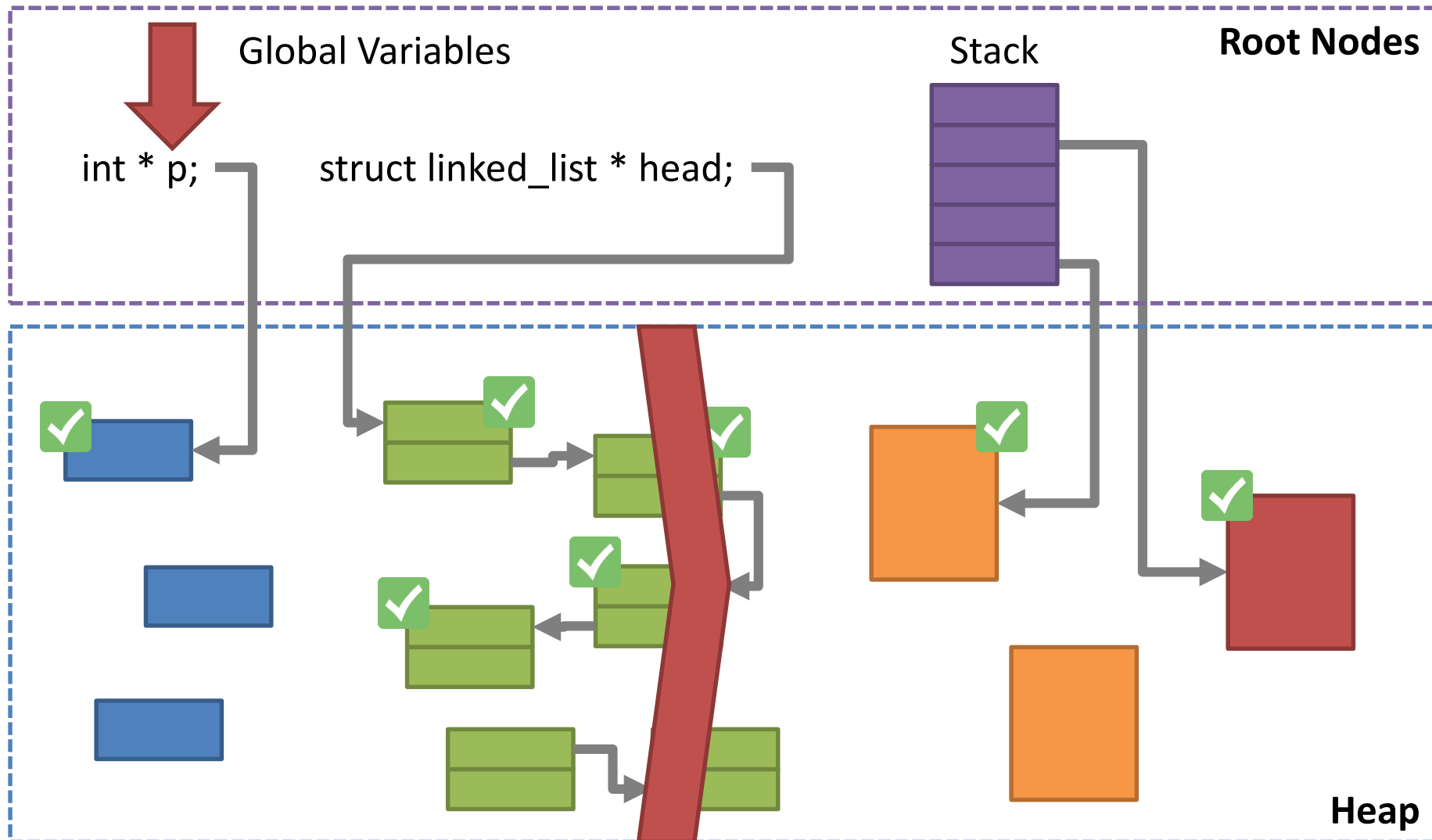
# Reachability Graph



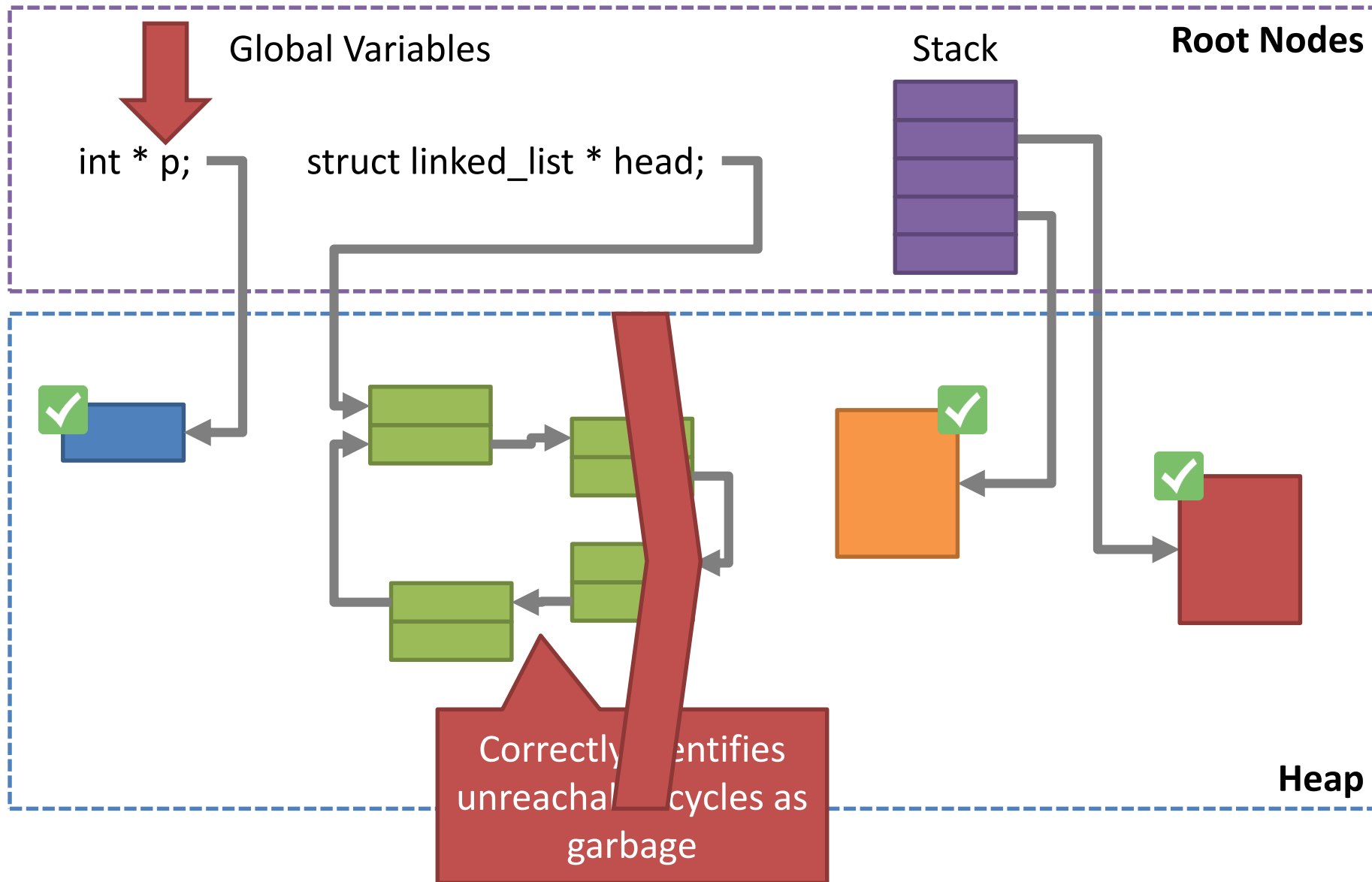
# Reachability Graph



# Mark and Sweep Example



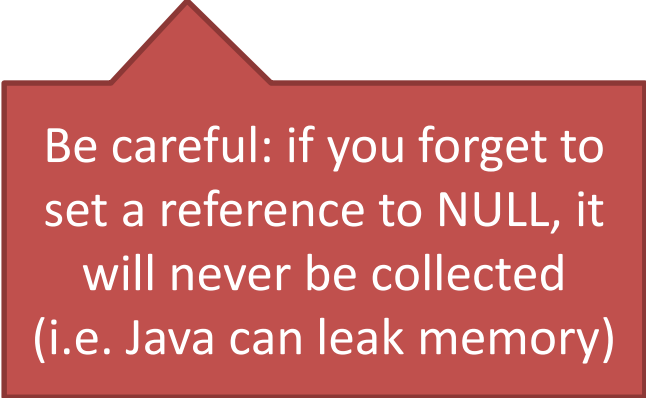
# Mark and Sweep Example



# Pros and Cons of Mark and Sweep

## The Good

- Overcomes the weakness of reference counting
- Fairly easy to implement and conceptualize
- Guaranteed to free all garbage objects



Be careful: if you forget to set a reference to NULL, it will never be collected (i.e. Java can leak memory)

## The Bad

- Mark and sweep is CPU intensive
  - Traverses all objects reachable from the root
  - Scans all objects in memory freeing unmarked objects
- Naïve implementations “stop the world” before collecting
  - Threads cannot run in parallel with the GC
  - All threads get stopped while the GC runs

# Copy Collection

- Problem with mark and sweep:
  - After marking, all objects on the heap must be scanned to identify and free unmarked objects
- Key idea: use compaction (aka relocation)
  - Divide the heap into *start space* and *end space*
  - Objects are allocated in *start space*
  - During GC, instead of marking, copy live object from *start space* into *end space*
  - Switch the *space* labels and continue

# Compaction/Relocation

String str2 = new String();

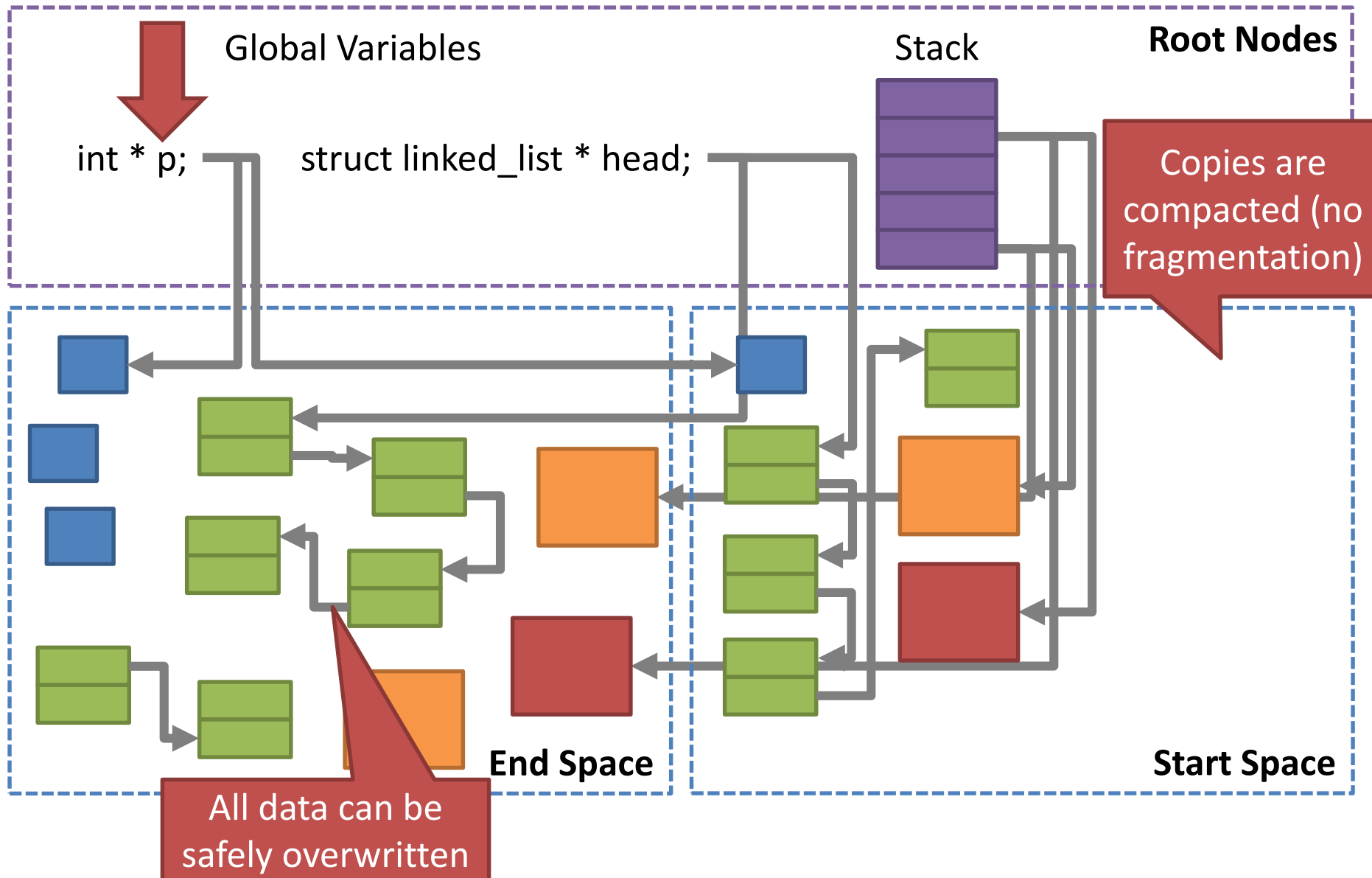
Pointer	Value	Location
obj1	0x0C00	0x0C00
ht	<b>0x0D90</b>	<b>0x0D90</b>
str	<b>0x0F20</b>	<b>0x0F20</b>
str2	0x10B0	0x10B0



- One way to deal with fragmentation is **compaction**
  - Copy allocated blocks of memory into a contiguous region of memory
  - Repeat this process periodically
- This only works if pointers are boxed, i.e. managed by the runtime



# Copy Collection Example





# Pros and Cons of Copy Collection

## The Good

- Improves on mark and sweep
- No need to scan memory for garbage to free
- After compaction, there is no fragmentation

## The Bad

- Copy collection is slow
  - Data must be copied
  - Pointers must be updated
- Naïve implementations are not parallelizable
  - “Stop the world” collector

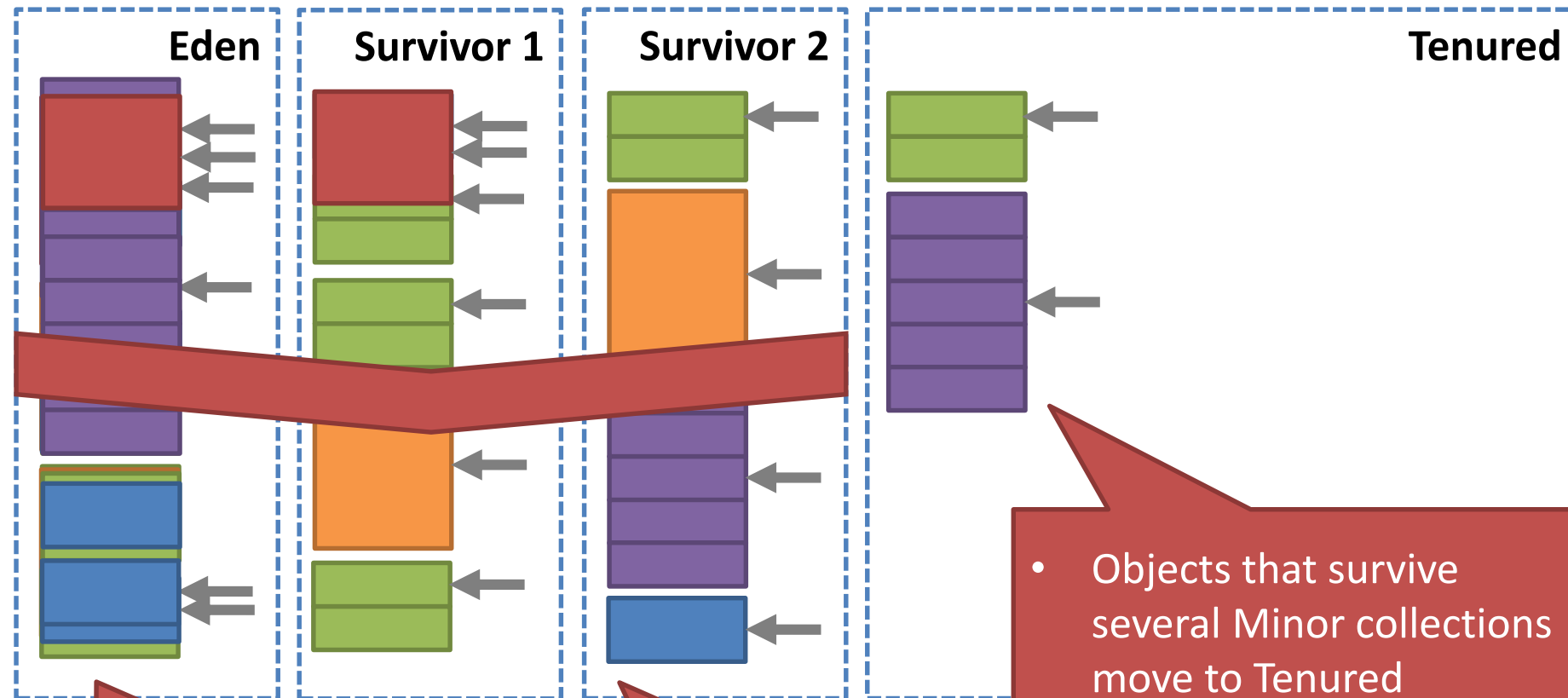
# Generational Collection

- Problem: mark and sweep is slow
  - Expensive full traversals of live objects
  - Expensive scan of heap memory
- Problem: copy collection is also slow
  - Expensive full traversals of live objects
  - Periodically, all live objects get copied
- Solution: leverage knowledge about object creation patterns
  - Object lifetime tends to be inversely correlated with likelihood of becoming garbage (generational hypothesis)
  - Young objects die quickly – old objects continue to live

# Garbage Collection in Java

- By default, most JVMs use a generational collector
- GC periodically runs two different collections:
  - Minor collection – occurs frequently
  - Major collection – occurs infrequently
- Divides heap into 4 regions
  - Eden: newly allocated objects
  - Survivor 1 and 2: objects from Eden that survive minor collection
  - Tenured: objects from Survivor that survive several minor collections

# Generational Collection Example

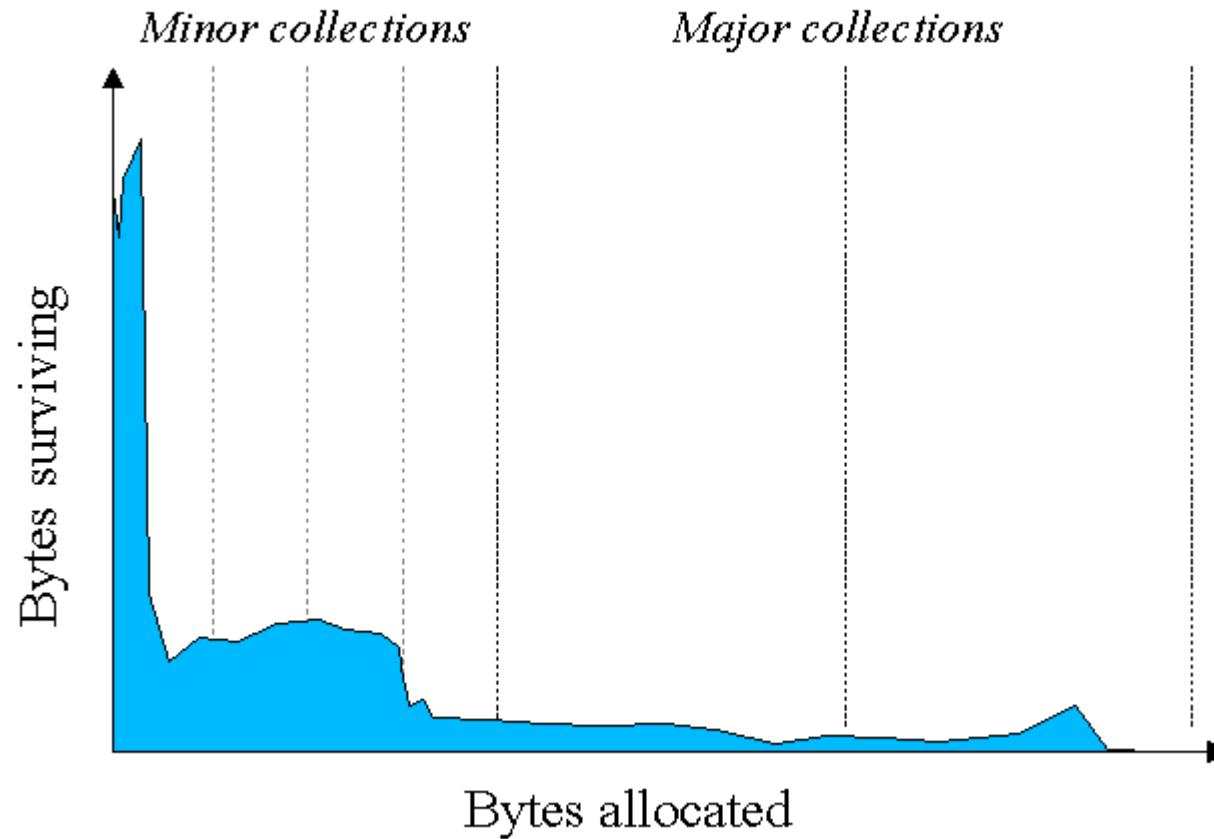


- Minor collection occurs whenever Eden gets full
- Live objects are copied to Survivor

- Survivor 1 and 2 rotate as destinations for a copy collector

- Objects that survive several Minor collections move to Tenured
- Tenured objects are only scanned during Major collection
- Major collections occur infrequently

# Infant Mortality



Source: [http://java.sun.com/docs/hotspot/gc5.0/gc\\_tuning\\_5.html](http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html)

# Generational Collection

- Observation: “most objects die young”
- Allocate objects in separate area (“nursery”, “Eden space”), collect area when run out of space
  - Will typically have to evacuate few survivors
  - “minor garbage collection”
- But: must treat all pointers into Eden as roots
  - Typically, requires cooperation of the mutator threads to record assignments: if ‘b’ is young, and ‘a’ is old,  $a.x = b$  must add a root for ‘b’.
    - Aka “write barrier”

# malloc()/free() vs. GC

## Explicit Alloc/Dealloc

- Advantages:
  - Typically faster than GC
  - No GC “pauses” in execution
  - More efficient use of memory
- Disadvantages:
  - More complex for programmers
  - Tricky memory bugs
    - Dangling pointers
    - Double-free
    - Memory leaks
  - Bugs may lead to security vulnerabilities

## Garbage Collection

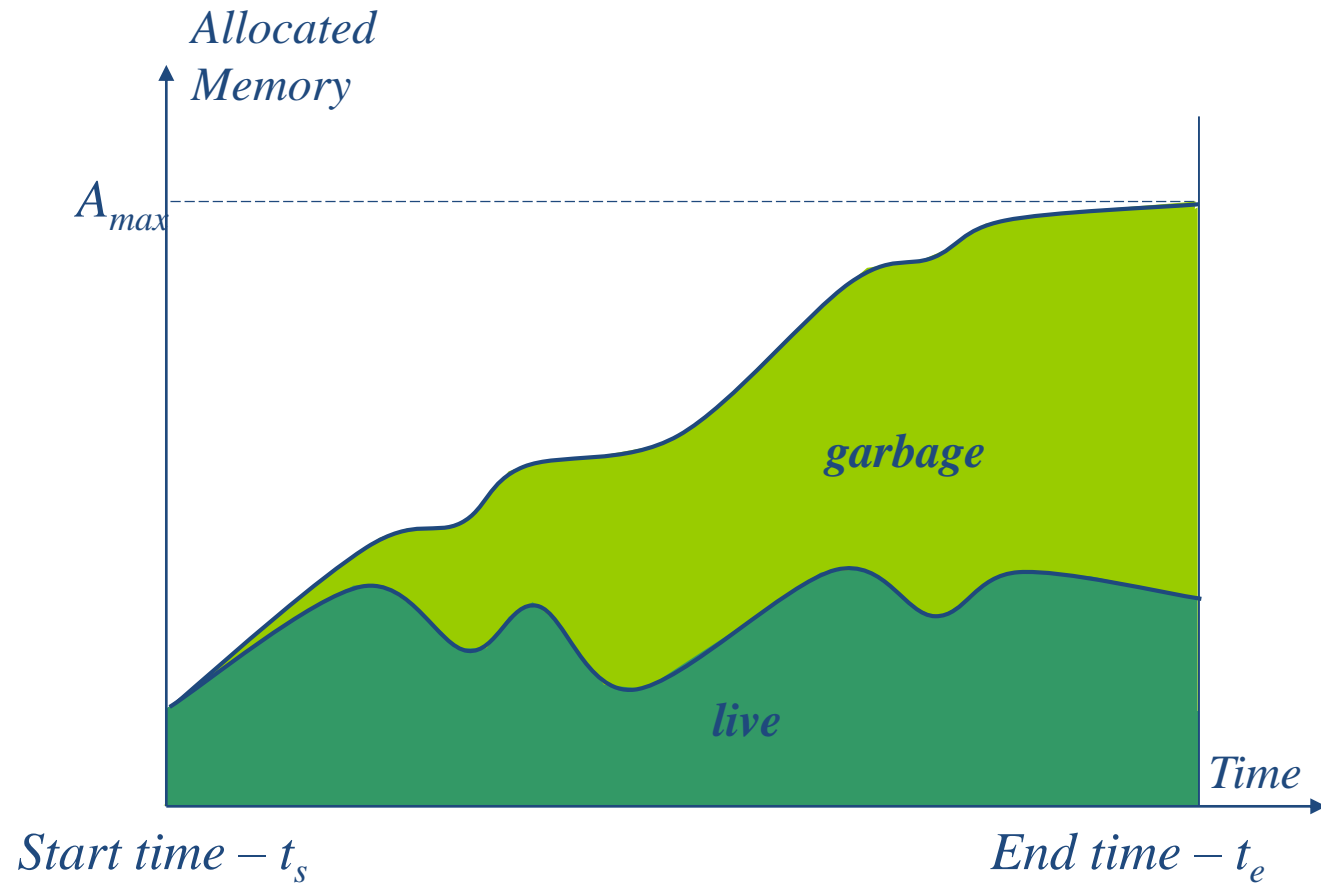
- Advantages:
  - Much easier for programmers
- Disadvantages
  - Typically slower than explicit alloc/dealloc
  - Good performance requires careful tuning of the GC
  - Less efficient use of memory
  - Complex runtimes may have security vulnerabilities
    - JVM gets exploited all the time

# Heap Size vs. GC Frequency

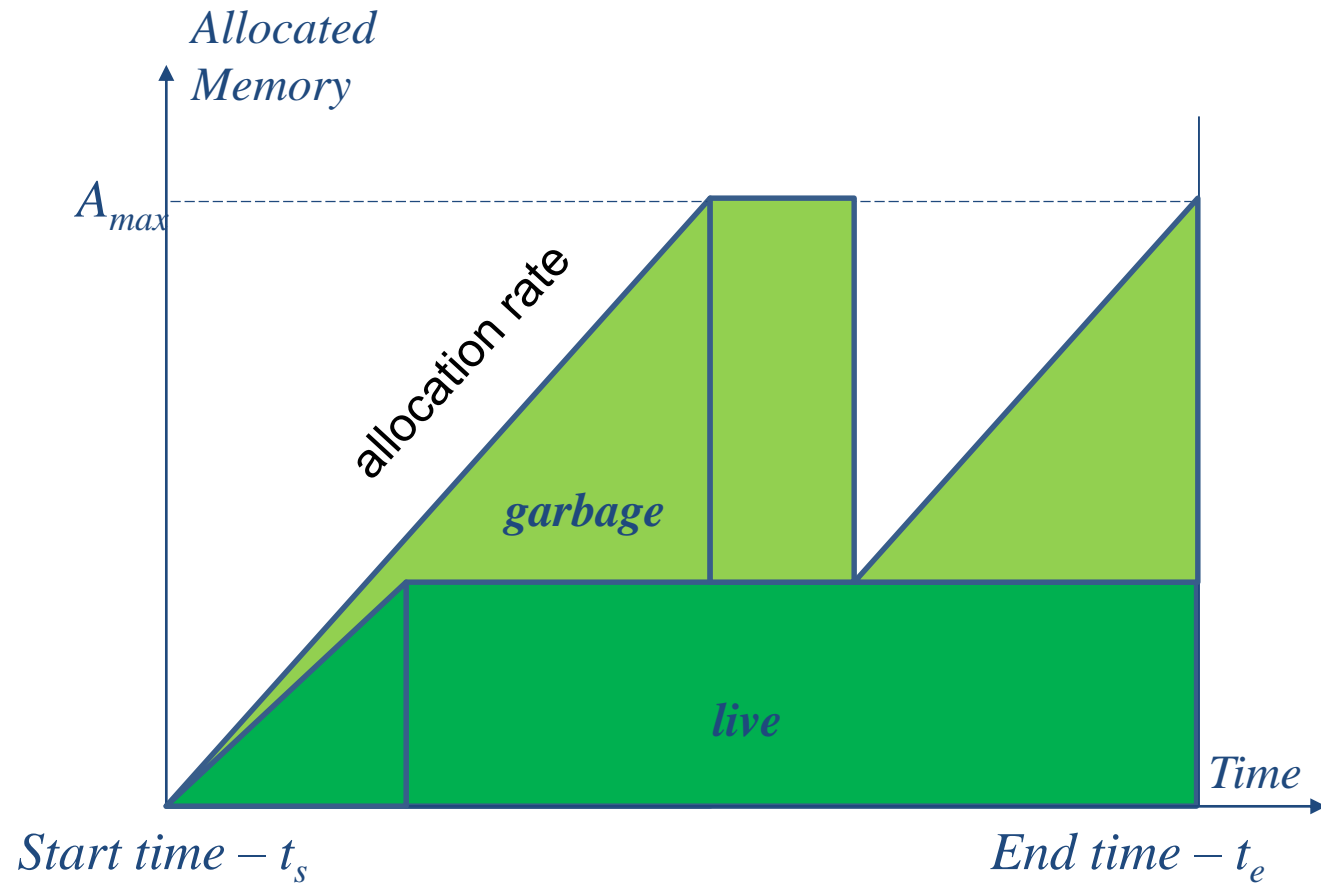
- All else being equal, smaller maximum heap sizes necessitate more frequent collections
  - Old rule of thumb: need between 1.5x and 2.5x times the size of the live heap to limit collection overhead to 5-15% for applications with reasonable allocation rates
  - [[Hertz 2005](#)] finds that GC outperforms explicit MM when given 5x memory, is 17% slower with 3x, and 70% slower with 2x
  - Performance degradation occurs when live heap size approaches maximum heap size



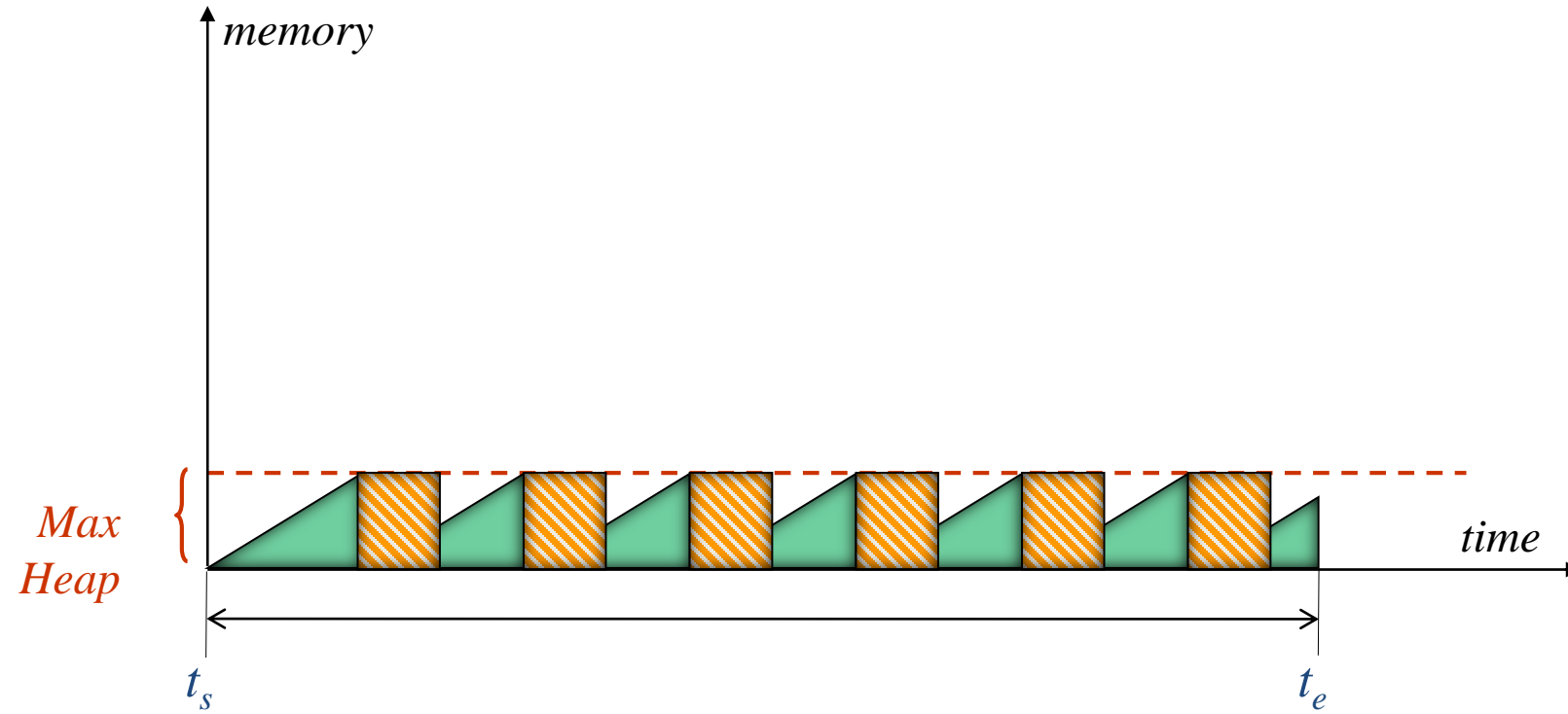
# Memory Allocation Time-Profile



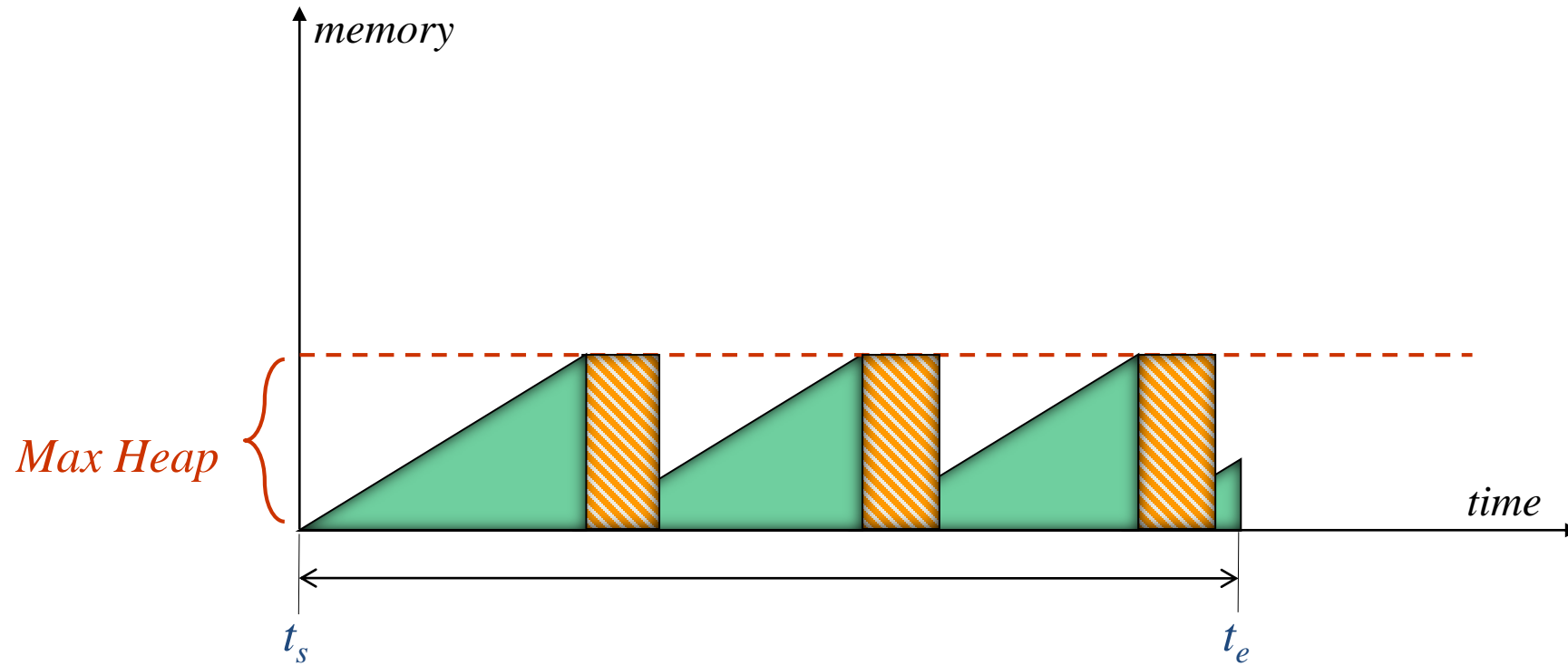
# Modeling Memory Allocation



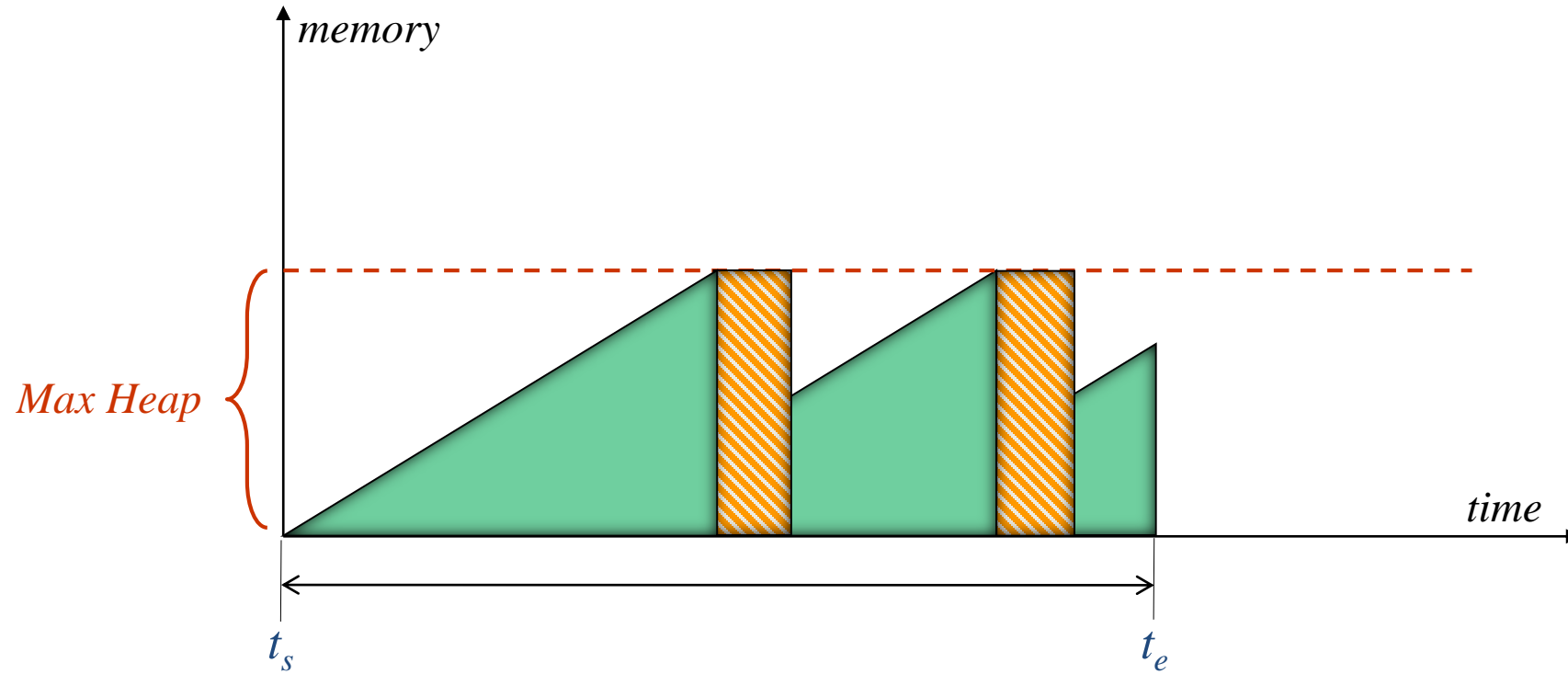
# Execution Time vs. Memory



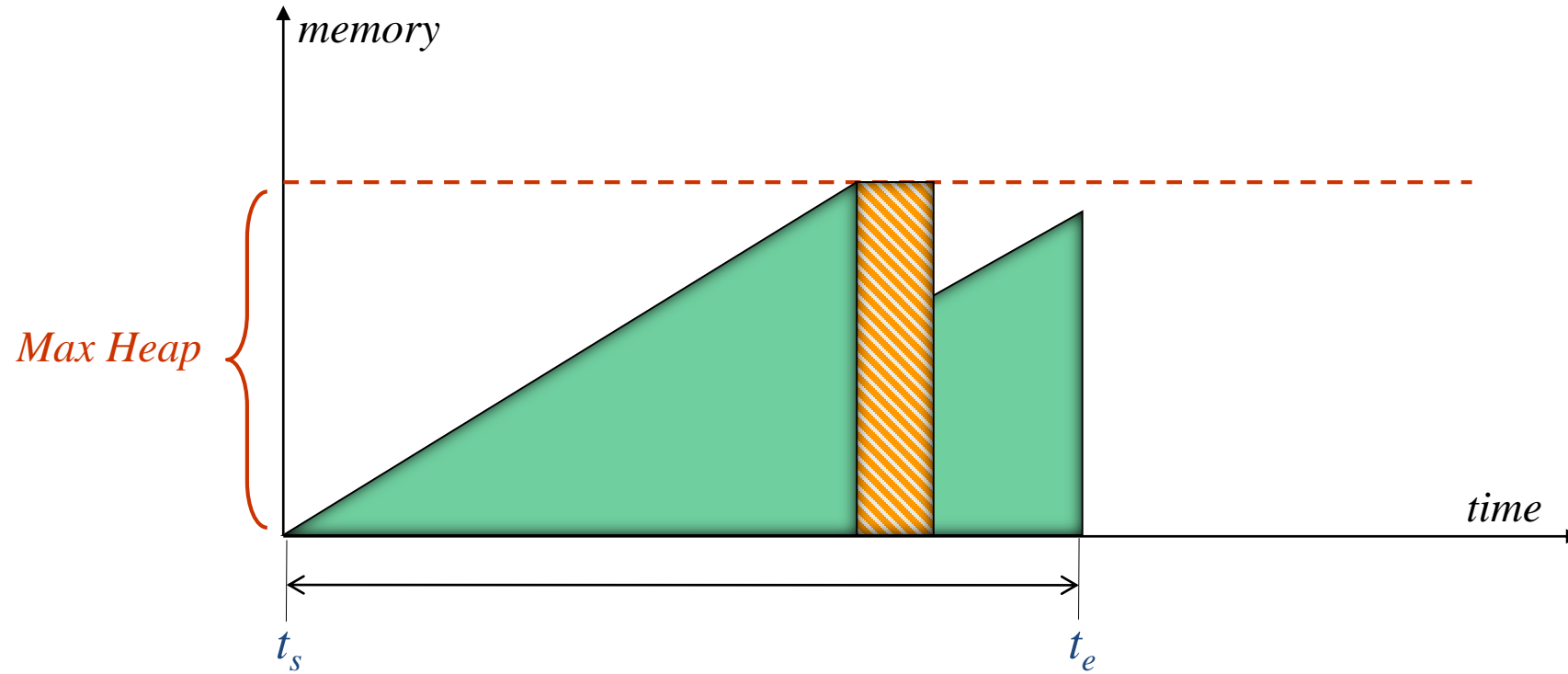
# Execution Time vs. Memory



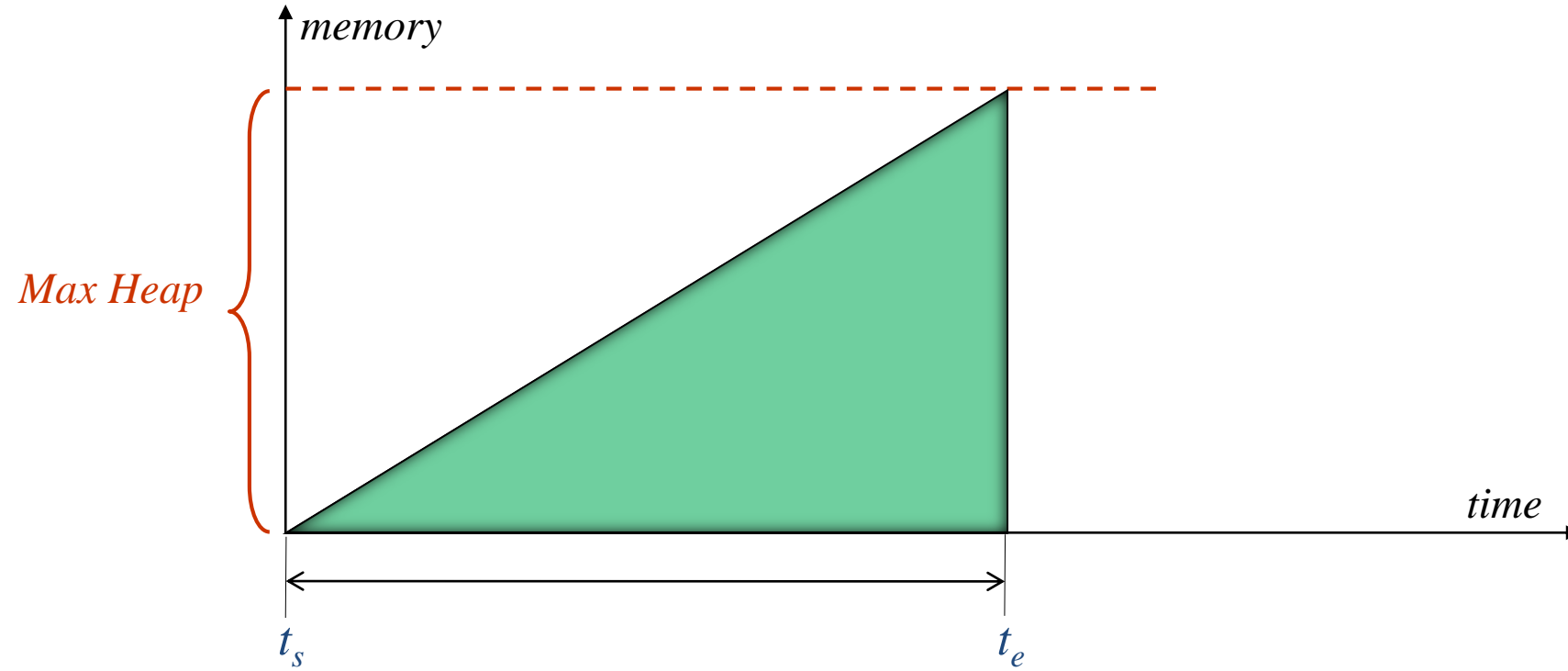
# Execution Time vs. Memory



# Execution Time vs. Memory



# Execution Time vs. Memory



# When to collect

- “Stop-the-world”
  - All mutators stop while collection is ongoing
- Incremental
  - Mutators perform small chunks of marking during each allocation
- Concurrent/Parallel
  - Garbage collection happens in concurrently running thread – requires some kind of synchronization between mutator & collector



# Programmer's Perspective

- Your program is running out of memory. What do you do?
- Possible reasons:
  - Leak
  - Bloat
- Your program is running slowly and unpredictably
  - Churn
  - “GC Thrashing”

# Memory Leaks

- Objects that remain reachable, but will not be accessed in the future
  - Due to application semantics
- Will ultimately lead to out-of-memory condition
  - But will degrade performance before that
- Common problem, particularly in multi-layer frameworks
  - Containers are a frequent culprit
  - Heap profilers can help

# Bloat and Churn

- Bloat: use of inefficient, pointer-intensive data structures increases overall memory consumption [Chis et al 2011]
  - E.g. HashMaps for small objects
- Churn: frequent and avoidable allocation of objects that turn to garbage quickly