Programming Language & Compiler

Memory Management & Garbage Collection

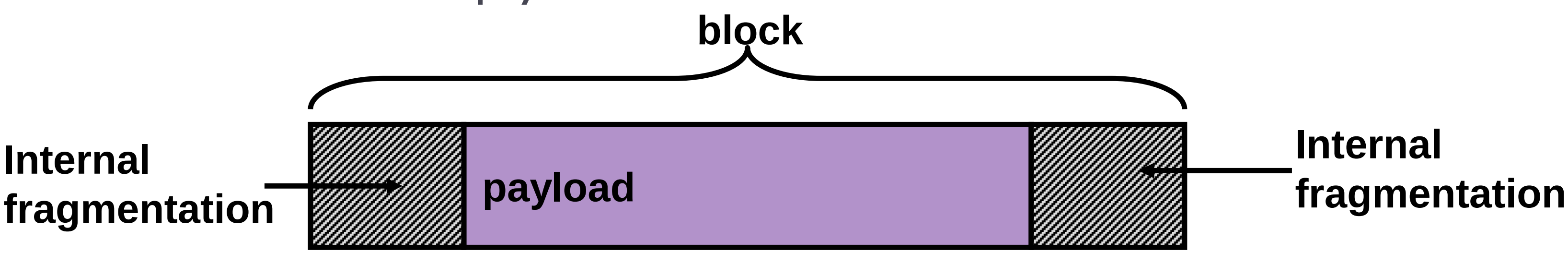
Hwansoo Han

Good Memory Management

- Primary goals
  - Good time performance for `malloc` and `free`
    - Ideally should take constant time (not always possible)
    - Should certainly not take linear time in the number of blocks
  - Good space utilization
    - User allocated structures should be large fraction of the heap.
    - Want to minimize "fragmentation".
- Some other goals
  - Good locality properties
    - Structures allocated close in time should be close in space
    - "Similar" objects should be allocated close in space
  - Robust
    - Can check that `free(p1)` is on a valid allocated object `p1`
    - Can check that memory references are to allocated space

Internal Fragmentation

- Poor memory utilization caused by *fragmentation*.
  - Comes in two forms: internal and external fragmentation
- Internal fragmentation
  - For some block, internal fragmentation is the difference between the block size and the payload size.



  - Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).
  - Depends only on the pattern of *previous* requests, and thus is easy to measure.

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

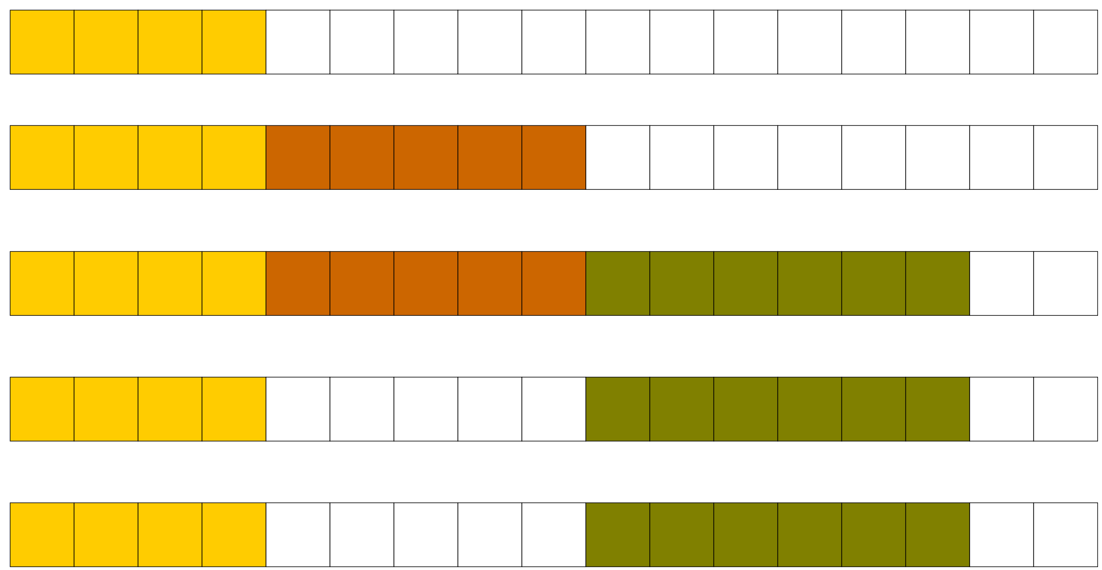
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```

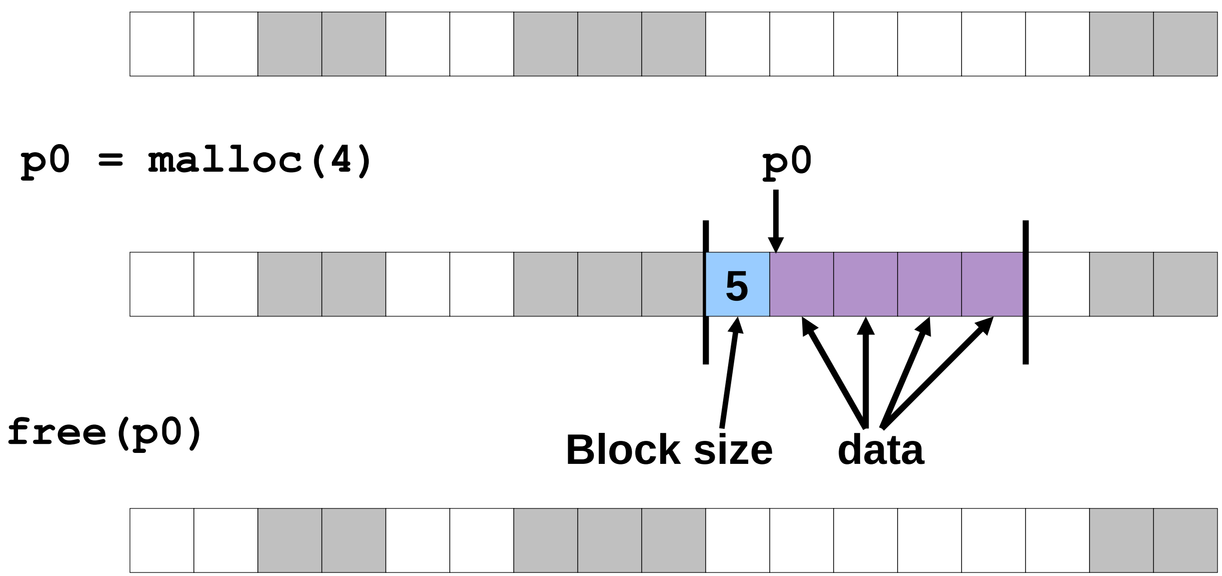


oops!

External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

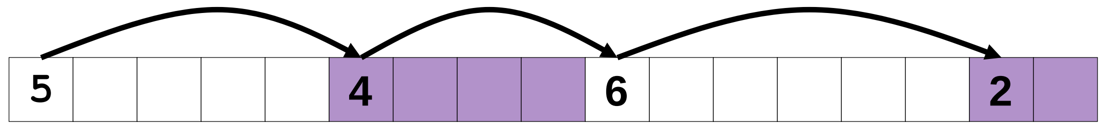
Knowing How Much to Free

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the *header field* or *header*
  - Requires an extra word for every allocated block

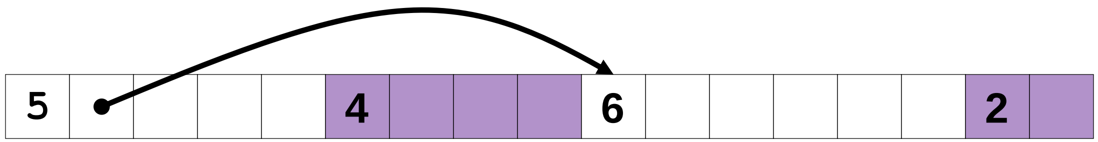


Keeping Track of Free Blocks

- ***<u>Method 1</u>***: *Implicit list* using lengths -- links all blocks



- *<u>Method 2</u>*: *Explicit list* among the free blocks using pointers within the free blocks

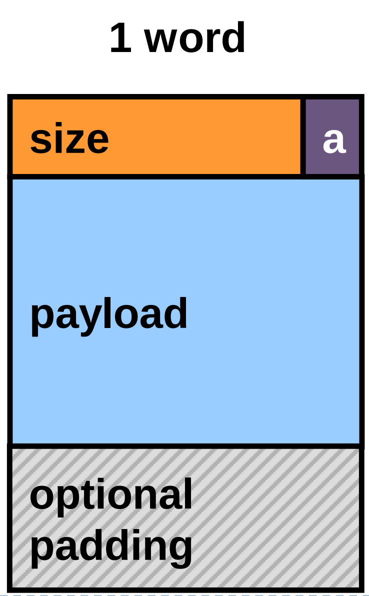


- *<u>Method 3</u>*: *Segregated free list*
  - Different free lists for different size classes

Method 1: Implicit List

- Need to identify whether each block is free or allocated
  - Can use extra bit
  - Bit can be put in the same word as the size if block sizes are always multiples of 2
    - Mask out low order bit when reading size
    - If you aligned on 8 bytes, mask out low order 3 bits

Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size (in bytes)

payload: application data
(allocated blocks only)

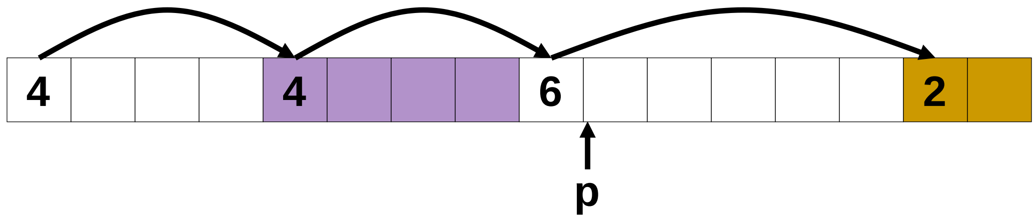
Implicit List: Finding a Free Block

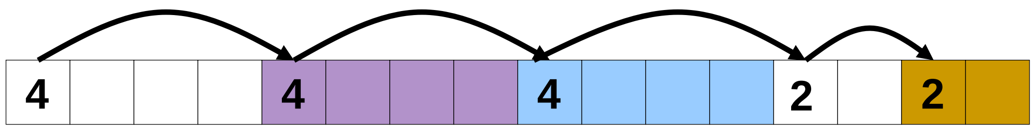
- *First fit:*
  - Search list from beginning, choose first free block that fits
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- *Next fit:*
  - Like first-fit, but search list from location of end of previous search
- *Best fit:*
  - Search the list, choose the free block with the closest size that fits
  - Keeps fragments small --- usually helps avoid fragmentation
  - Will typically run slower than first-fit

Implicit List: Allocating & Freeing

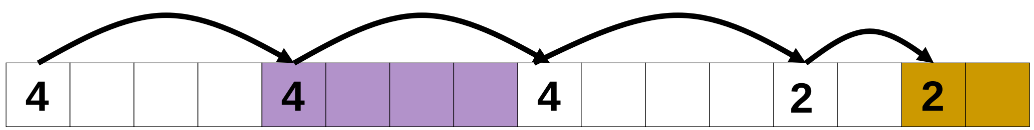
- Allocating in a free block – *splitting if needed*

```
P = malloc(3)
```






```
free(p)
```

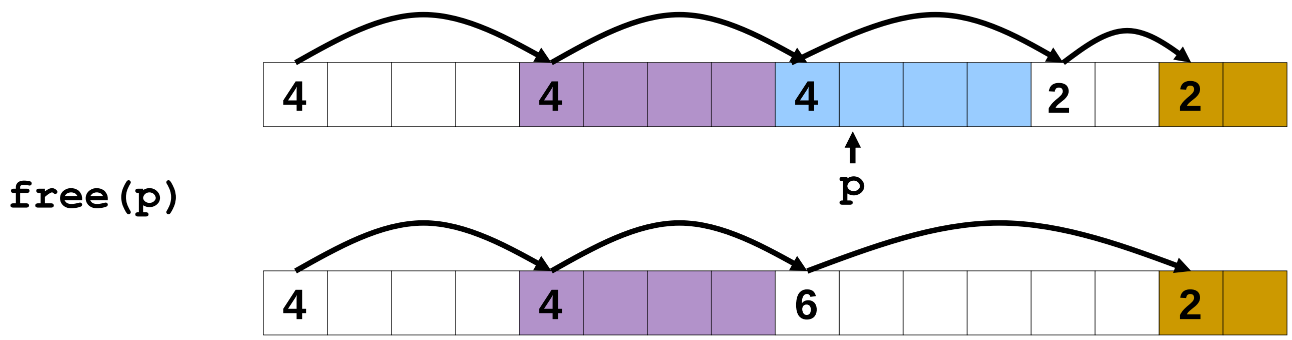



```
malloc(5)
```

Oops!

Implicit List: Coalescing

- Join (*coalesce*) with next and/or previous block if they are free

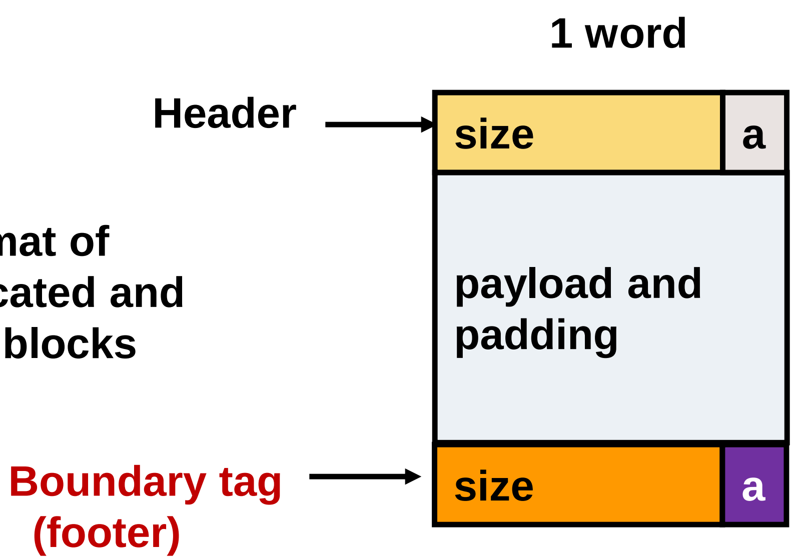


- Coalescing with previous free block?

Implicit List: Bidirectional Coalescing

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at bottom of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!

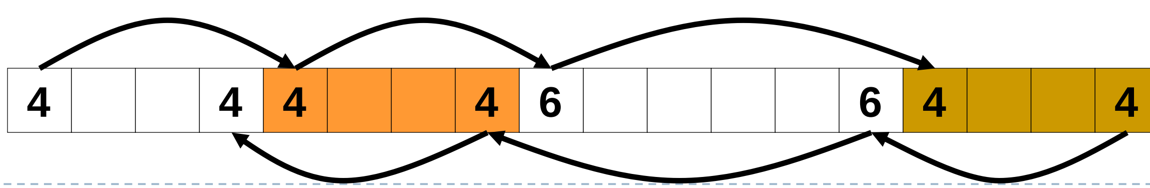
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

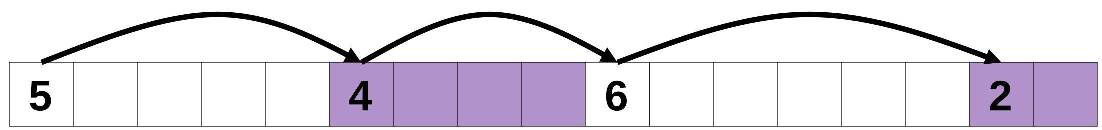
size: total block size

payload: application data (allocated blocks only)

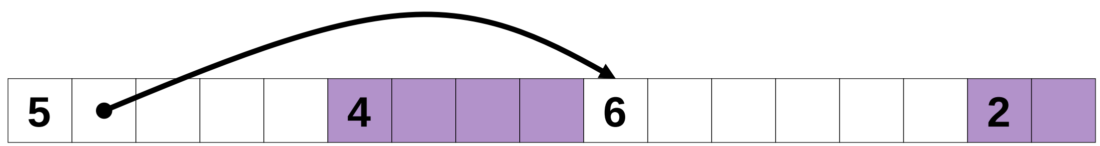


Keeping Track of Free Blocks

- *Method 1*: Implicit list using lengths -- links all blocks



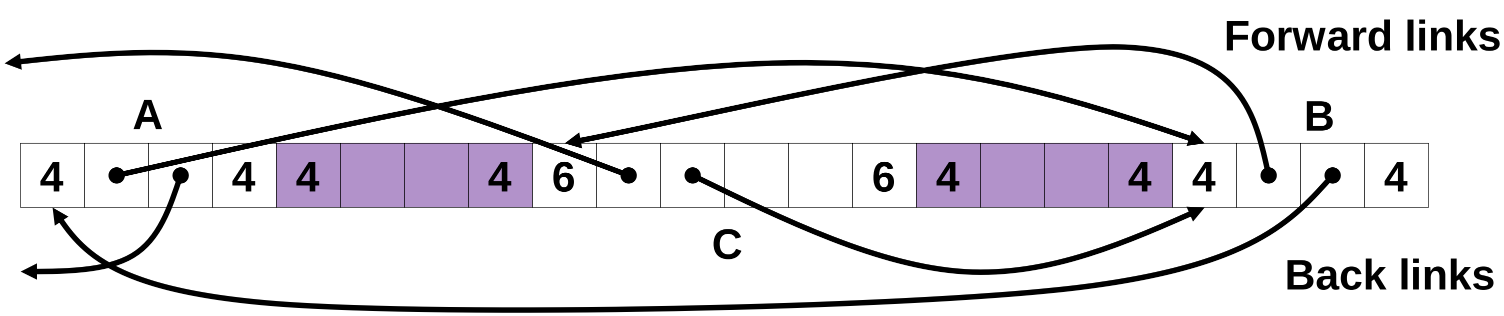
- ***Method 2*:** Explicit list among the free blocks using pointers within the free blocks



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

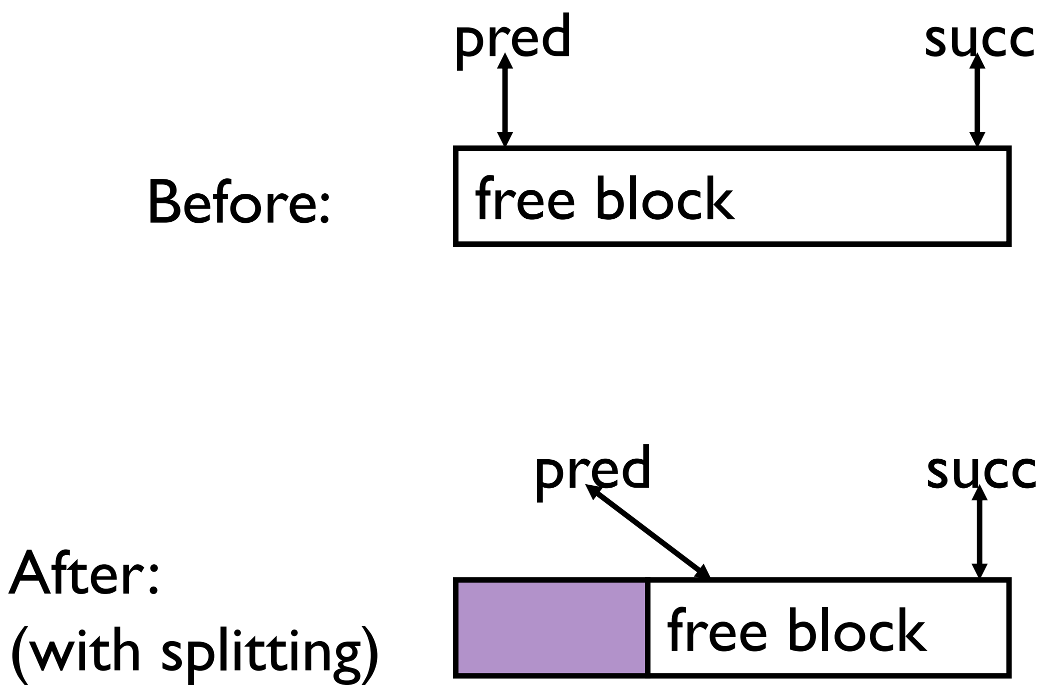
Explicit Free Lists

- Use data space for link pointers
  - Typically doubly linked
  - Still need boundary tags for coalescing



- It is important to realize that links are not necessarily in the same order as the blocks

Allocating From Explicit Free Lists

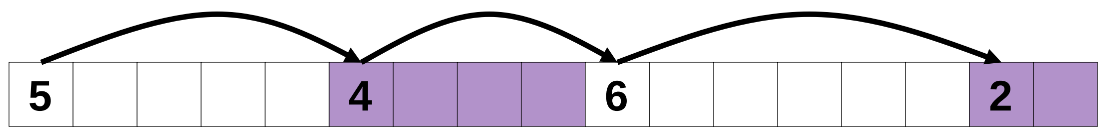


Freeing With Explicit Free Lists

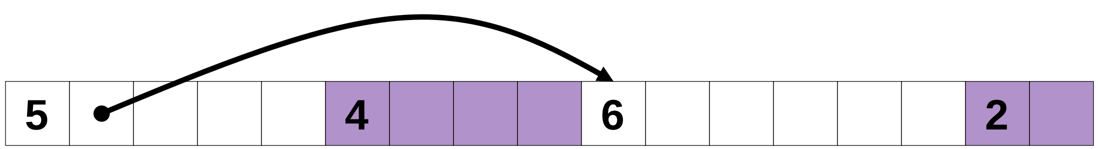
- *Insertion policy:* Where in the free list do you put a newly freed block?
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
    - Pro: simple and constant time
    - Con: studies suggest fragmentation is worse than address ordered.
  - Address-ordered policy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

Keeping Track of Free Blocks

- *__Method 1__*: Implicit list using lengths -- links all blocks



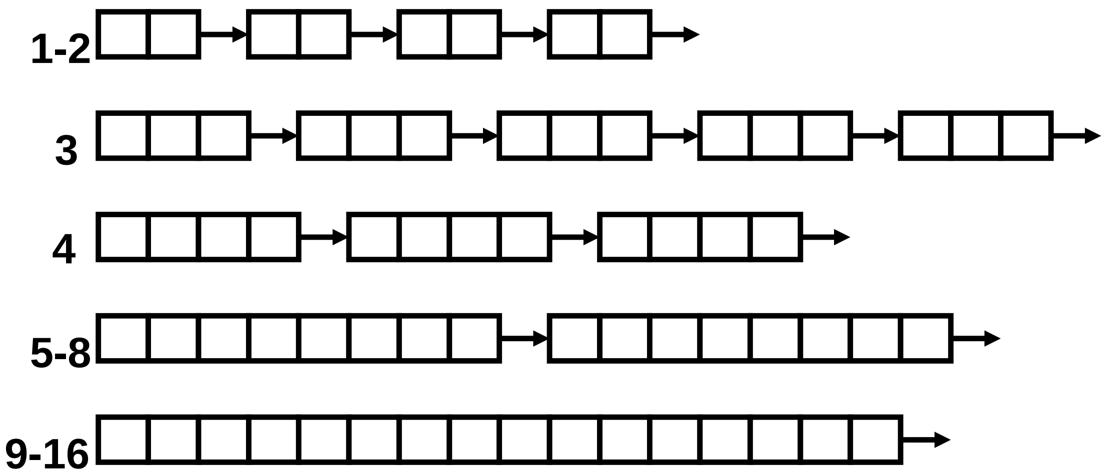
- *__Method 2__*: Explicit list among the free blocks using pointers within the free blocks



- ***__Method 3__***: Segregated free lists
  - Different free lists for different size classes

Segregated Storage

- Each *size class* has its own collection of blocks



- General principles
  - Often have separate size class for every small size (2,3,4,…)
  - For larger sizes typically have a size class for each power of 2
- 128 size classes for Doug Lea's malloc.c
  - 63 exact bins (spaced by 8 byte) : 16,24,32,…,512
  - 64 sorted bins (approx. logarithmically spaced) : 576, 640, … 2^{31}

Segregated Fits

- 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
- To free a block:
  - Coalesce and place on appropriate list (optional)
- Tradeoffs
  - Faster search than sequential fits
  - Controls fragmentation of simple segregated storage
  - Coalescing can increase search times
    - Deferred coalescing can help

GC - Automated Free for Heap Objects

- Garbage collection
  - Automatically reclaim the space that the running program can never access again
  - Performed by the runtime system

- Two parts of a garbage collector
  - Garbage detection
  - Reclamation of the garbage objects’ storage

Liveness in GC

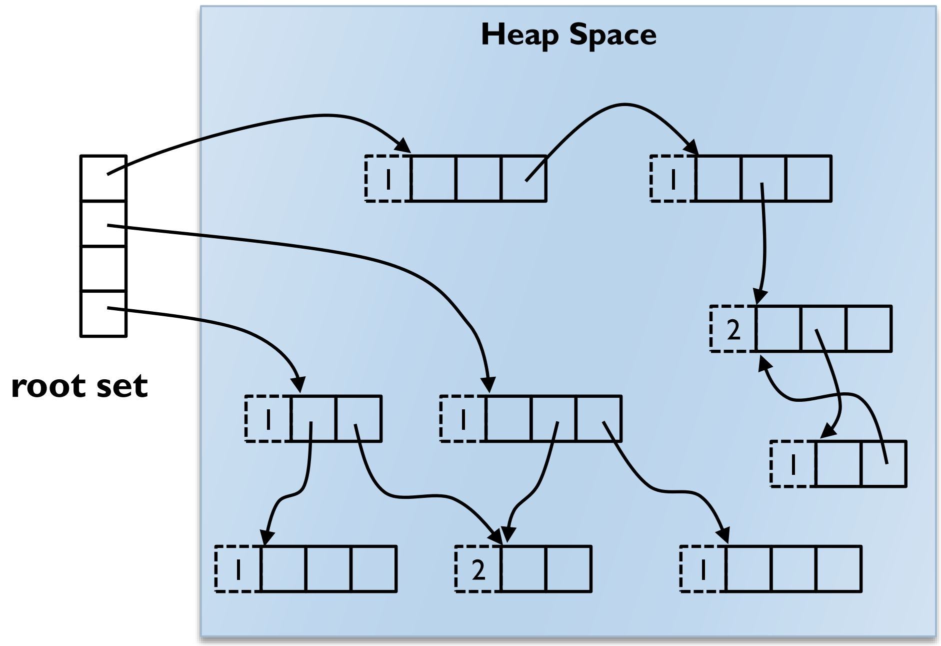
- A root set
  - Global variables
  - Local variables in the activation stack
  - Any registers used by active procedures
- Live objects
  - Objects on any directed path of pointers from the roots

Basic Garbage Collection Techniques

- Assumption – heap objects are self-identifying

- Basic techniques for GC
  - Reference counting
  - Mark-Sweep collection
  - Mark-Compact collection
  - Copying collection

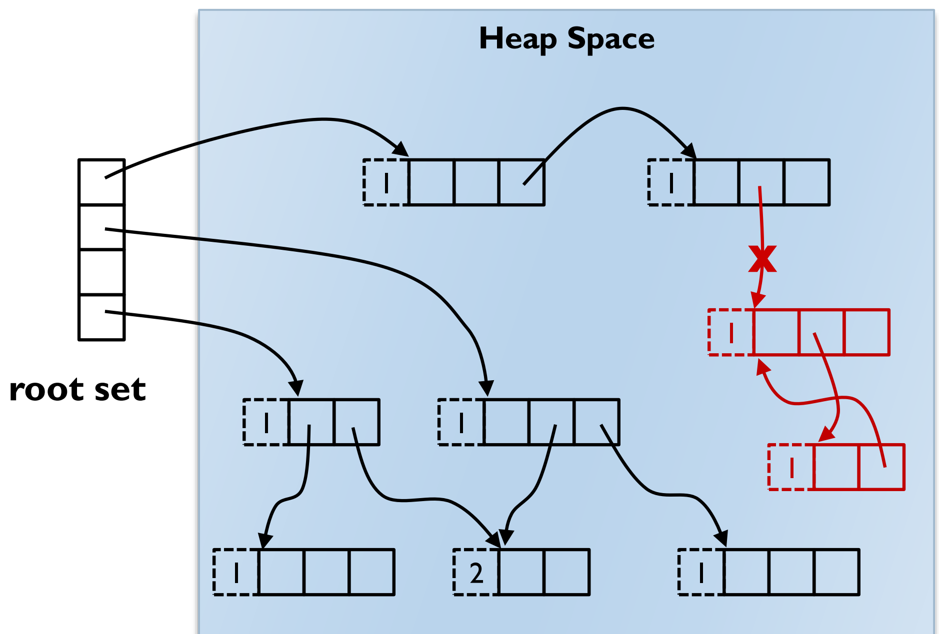
Reference Counting



- Keeping track of how many pointers point to each record

Reference Counting - problems

- Cost of reference counting
  - Too many ref-count increments and decrements
- Fail to reclaim circular structures
  - Not always effective



Mark-Sweep Collection

- Garbage detection
  - Traverse the graph of pointer relationships *and*
  - Mark all reachable objects
- Reclamation
  - Sweep unmarked objects
- Problems
  - Fragmentation
  - Collection cost - proportional to the size of heap
  - Poor locality of reference

Mark-Compact Collection

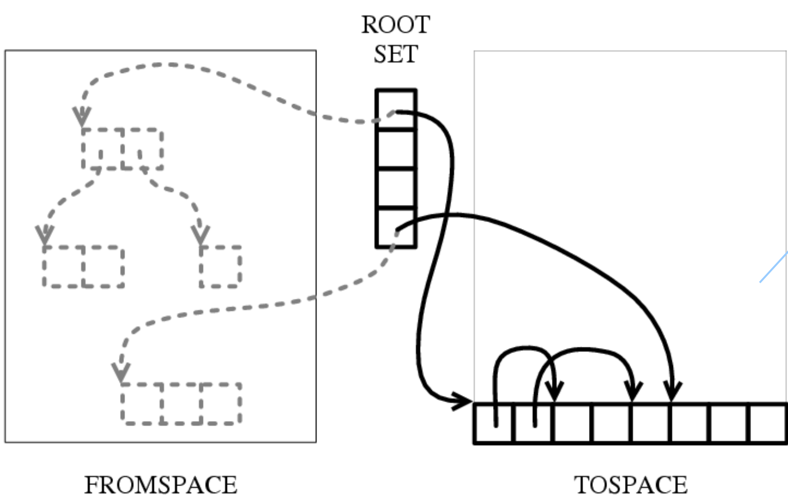
- The same detection phase by marking
- Objects are compacted
  - Moving most of the live objects until all of the live objects are contiguous
- Pros
  - No fragmentation problem
  - Allocation order preserved
- Cons
  - Slower than mark-sweep
  - Need fast compacting algorithms

Copying Collection

- Move the live objects to a contiguous area
  - Integrate the traversal of the data and the copying process
- A simple copying collector
  - "Stop-and-Copy" using semi-spaces

"Stop-and-Copy" Using Semi-Spaces

- Subdivided heap into two contiguous semi-spaces
- When program demands more unused area of the current semi-space
  - Stop and copy to reclaim space



Cheney's algorithm: breadth-first search to traverse the reachable data

Incremental Tracing Collection

- Interleave GC with program execution
  - Small units of garbage collection interleaved with small units of program execution
  - Needed for real-time applications

- Difficulty
  - While the collector is tracing out the graph of reachable objects, the graph may change by the running program

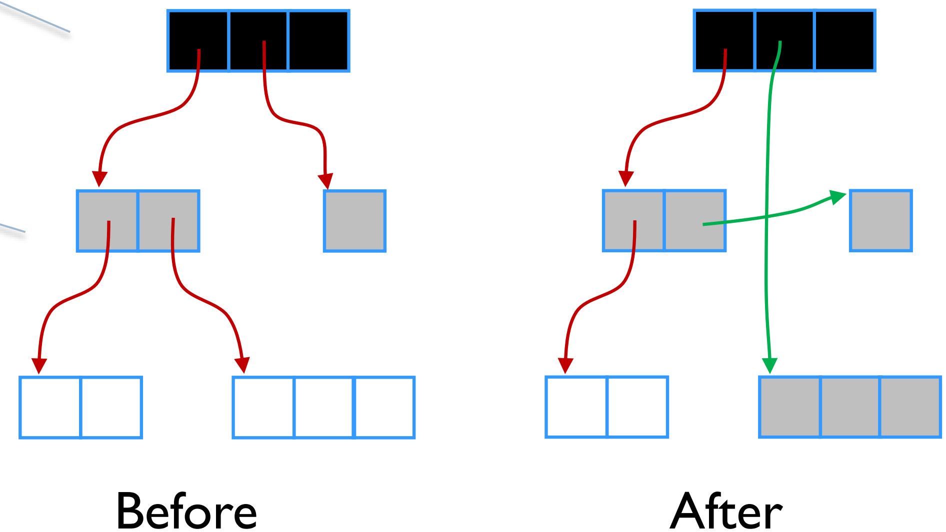
- Mark-sweep or copying GC can be made incremental
  - Tricolor marking

Tricolor Marking

Black objects have been visited, and their children also visited

Grey objects have been visited, but their children have not yet

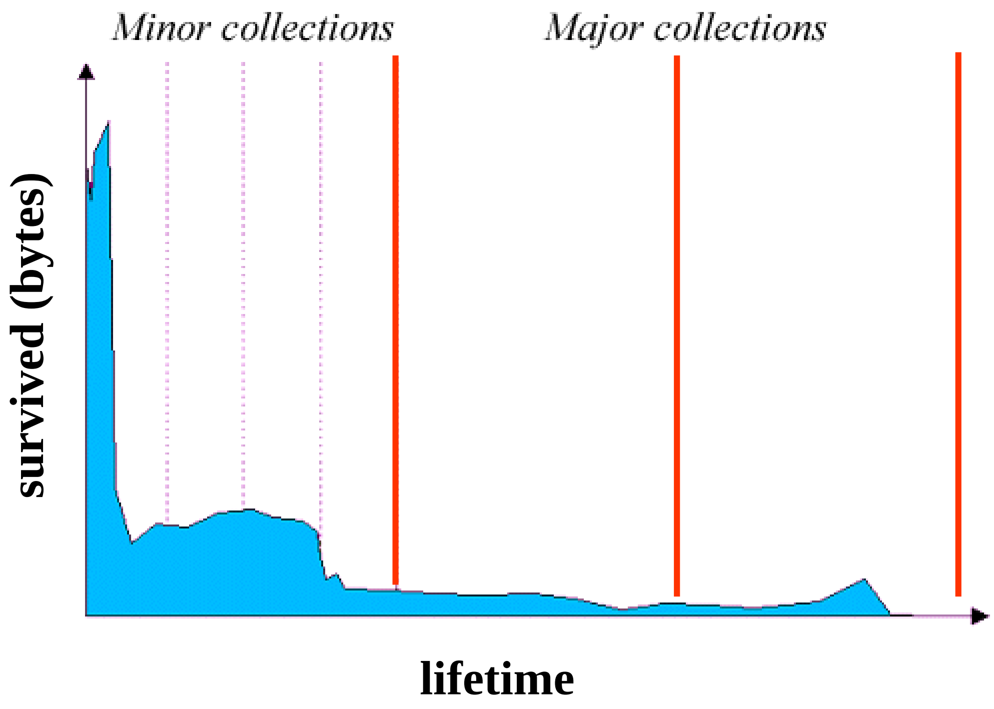
White objects are not yet visited



- Live objects' color becomes white, grey, and then black
  - Collection ends when there are no grey objects
  - All white objects are garbage
- Need to coordinate the collector with the "mutator"
  - Invariant - No black object points to a white object

Infant Mortality

- Most objects die young
- Marking entire heap space is time-consuming



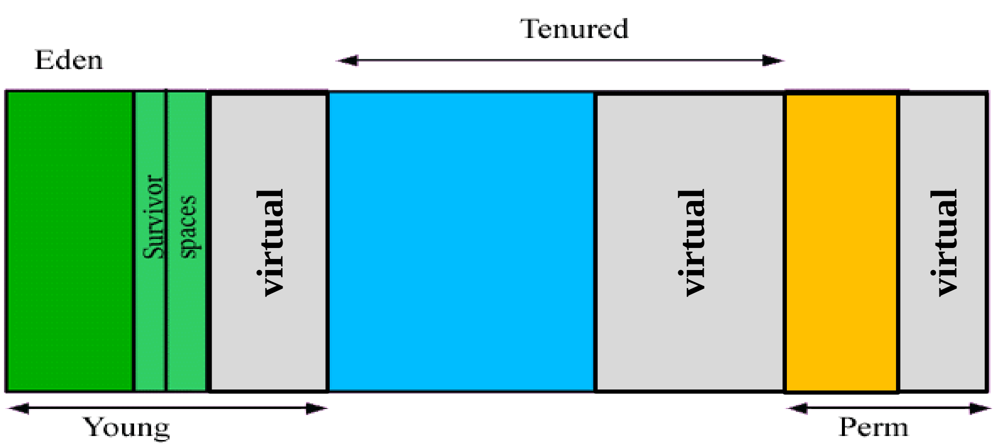
Generational Collection

- Most objects live a very short time, while a small percentage of them live much longer

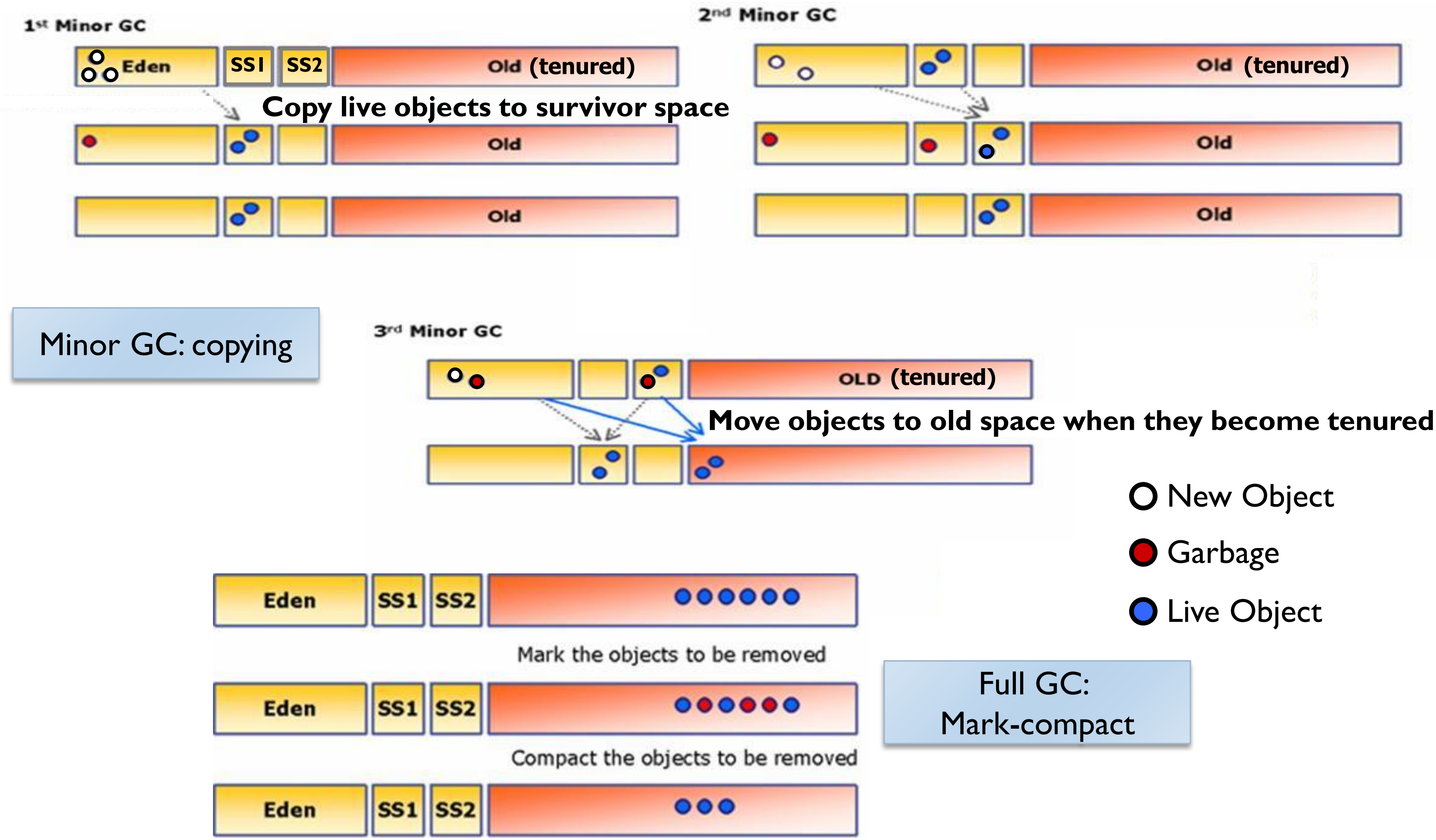
- When using copying collection, need to avoid much repeated copying of old objects
  - Divide the heap into generations
    - The younger generation is typically several times smaller than the old one
  - Younger generations are collected more often

Generational Collection in J2SE

- Arrangement of generations
  - Young generation (nursery)
    - Eden + two survivor spaces
  - Tenured generation
  - Permanent generation
    - Code area used by JVM - class and method objects

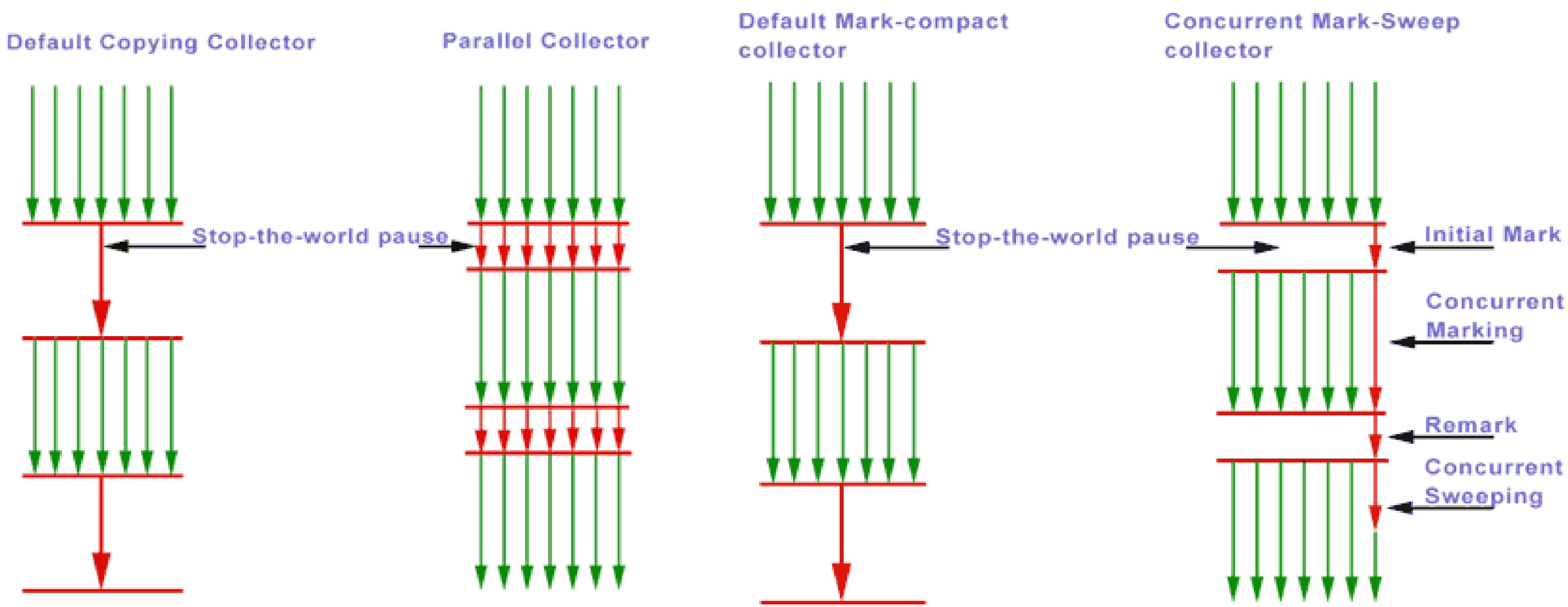


Default GC in J2SE



GC Algorithms in J2SE 1.4.1+

- Young generation
  - Copying collector
  - Parallel collector (2P+)
- Old generation
  - Mark-compact collector
  - Concurrent Mark-Sweep collector (2P+)



Summary

- Memory allocation
  - First fit, next fit, best fit, segregated fit

- Free list management
  - Implicit list, explicit list, segregated list
  - Coalescing with boundary tags

- Garbage collection
  - Reference counting
  - Mark-sweep, mark-compact, copying-collection
  - Stop-the-world vs. incremental GC
  - Generational GC
  - Concurrent GC