Compilation

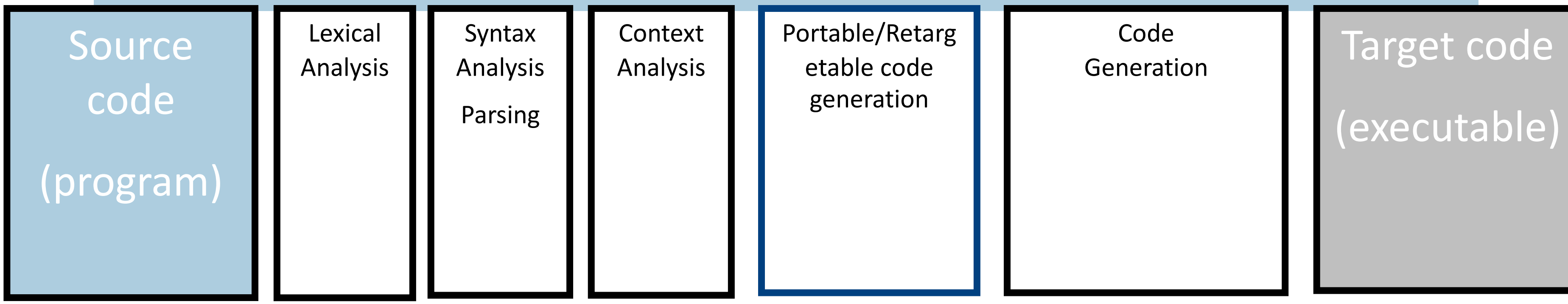
0368-3133

Lecture 10

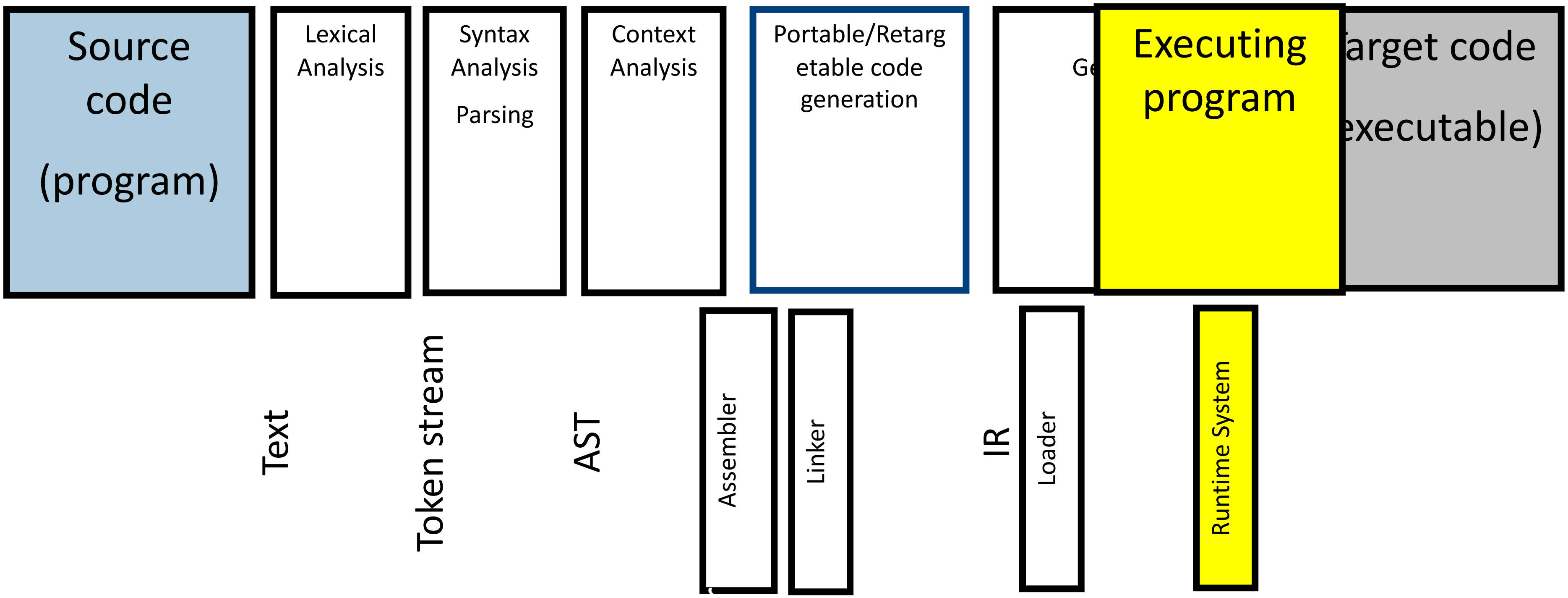
Memory Management

Noam Rinetzky

Stages of compilation



Compilation ➔ Execution



Runtime Environment

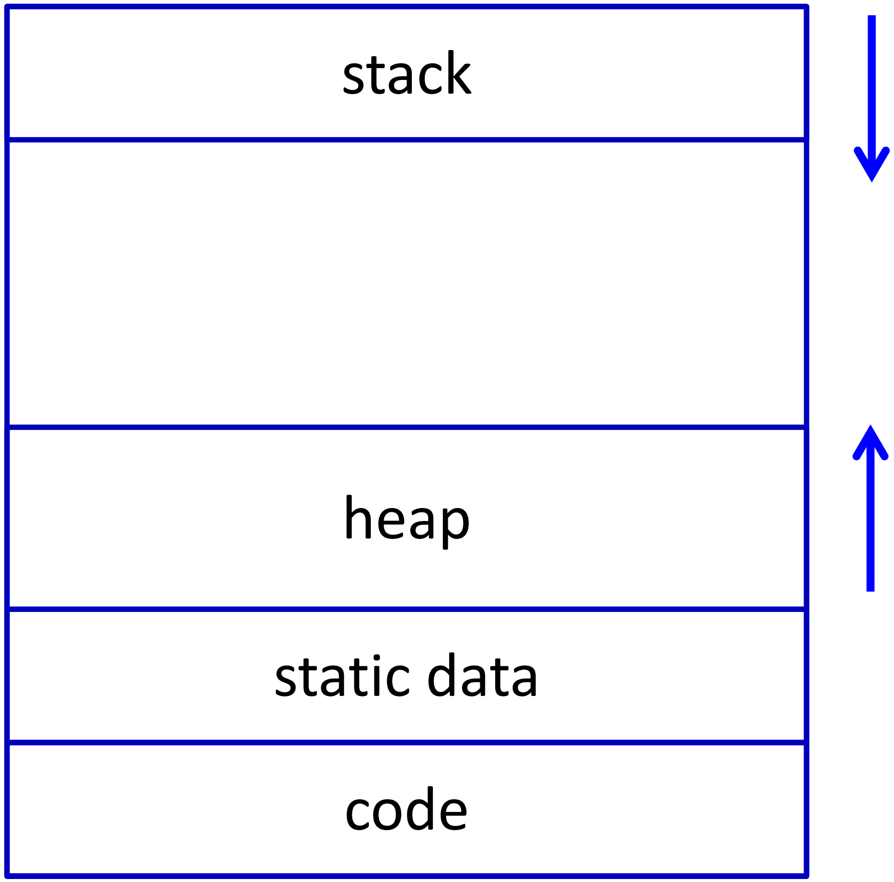
- Mediates between the OS and the programming language
- Hides details of the machine from the programmer
  - Ranges from simple support functions all the way to a full-fledged virtual machine
- Handles common tasks
  - Runtime stack (activation records)
  - **Memory management**
  - Dynamic optimization
  - Debugging
  - …

Where do we allocate data?

- Activation records
  - Lifetime of allocated data limited by procedure lifetime
  - Stack frame deallocated (popped) when procedure return

- **Dynamic memory allocation on the heap**

Memory Layout



Alignment

- Typically, can only access memory at aligned addresses
  - Either 4-bytes or 8-bytes
- What happens if you allocate data of size 5 bytes?
  - **Padding** – the space until the next aligned addresses is kept empty
- (side note: x86, is more complicated, as usual, and also allows unaligned accesses, but not recommended)

Allocating memory

- In C - malloc
- `void *malloc(size_t size)`

- Why does malloc return void* ?
  - It just allocates a chunk of memory, without regard to its type
- How does malloc guarantee alignment?
  - After all, you don't know what type it is allocating for
  - It has to align for the largest primitive type
  - In practice optimized for 8 byte alignment (glibc-2.17)

Memory Management

- Manual memory management
- Automatic memory management

Manual memory management

- malloc
- free

```
a =  malloc(…) ;
// do something with a
free(a);
```

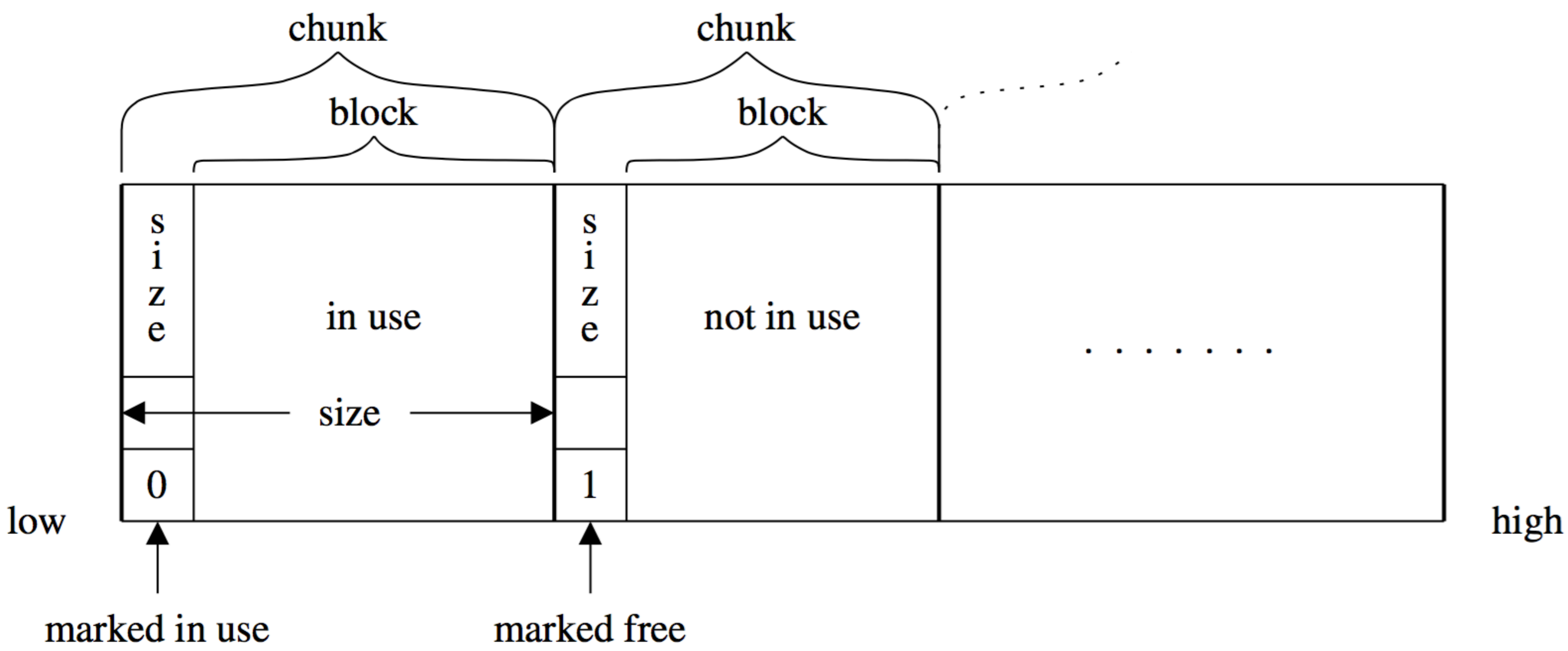
malloc

- where is malloc implemented?
- how does it work?

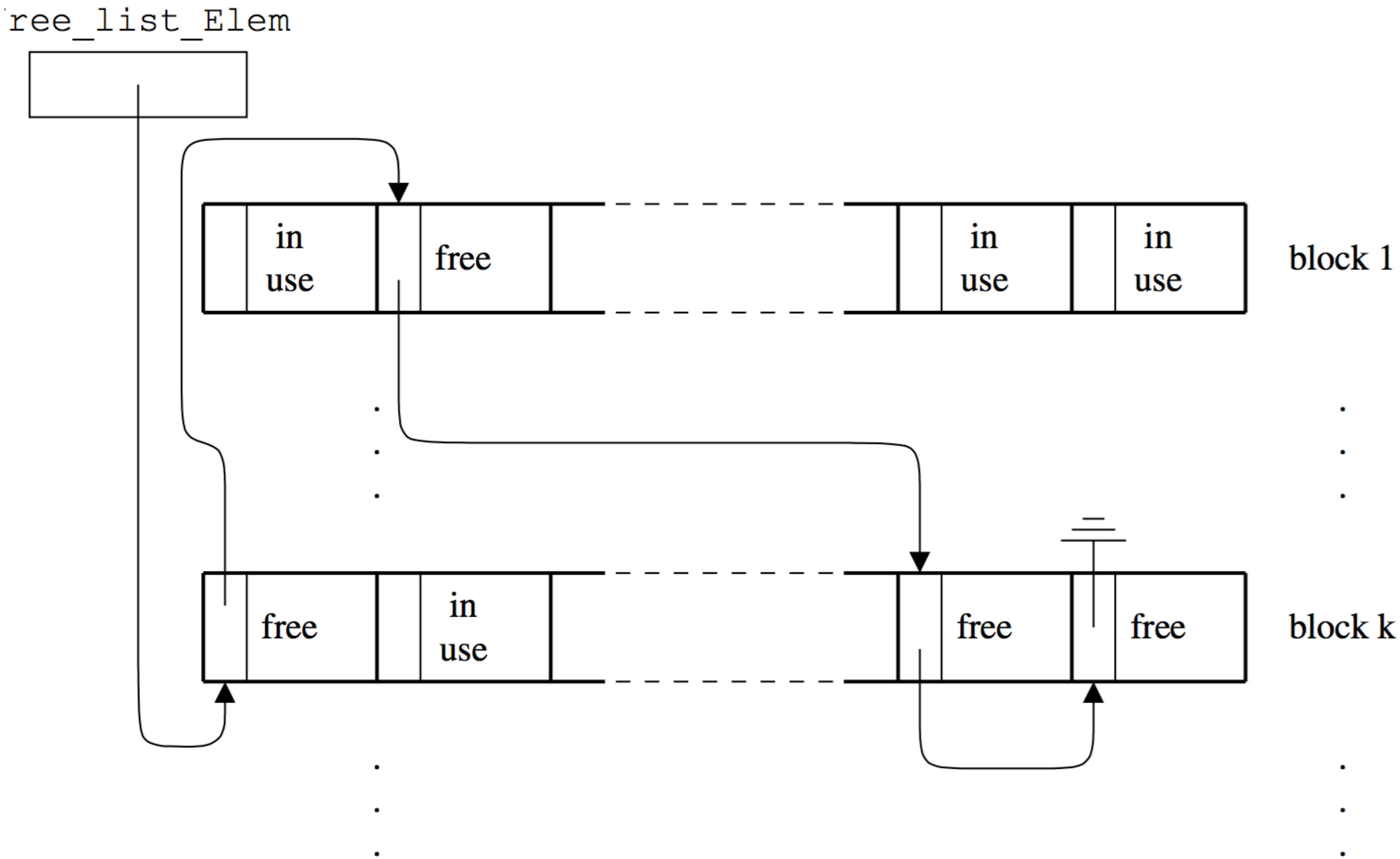
Free-list Allocation

- A data structure records the location and size of free cells of memory.
- The allocator considers each free cell in turn, and according to some policy, chooses one to allocate.
- Three basic types of free-list allocation:
  - First-fit
  - Next-fit
  - Best-fit

Memory chunks

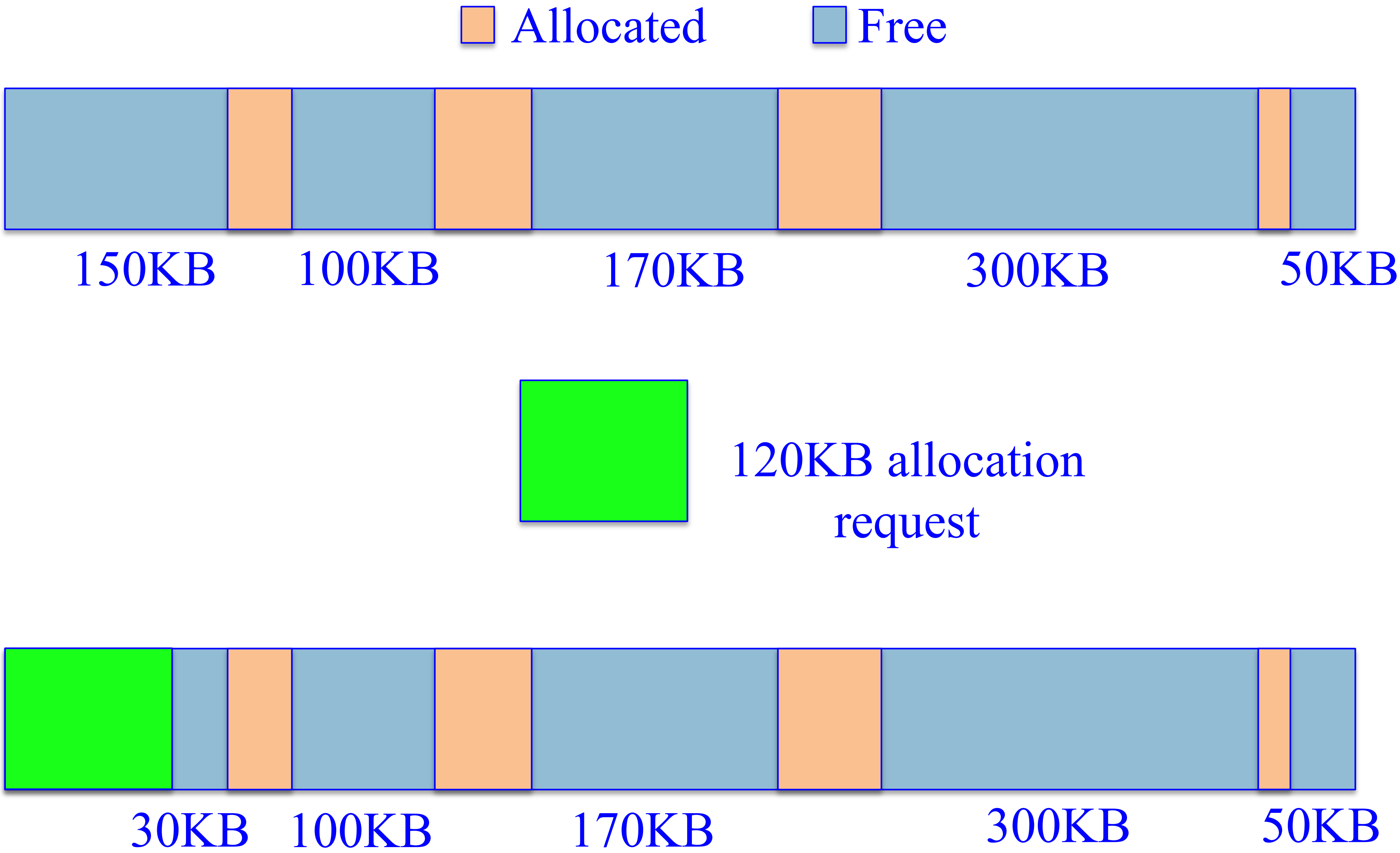


Free list



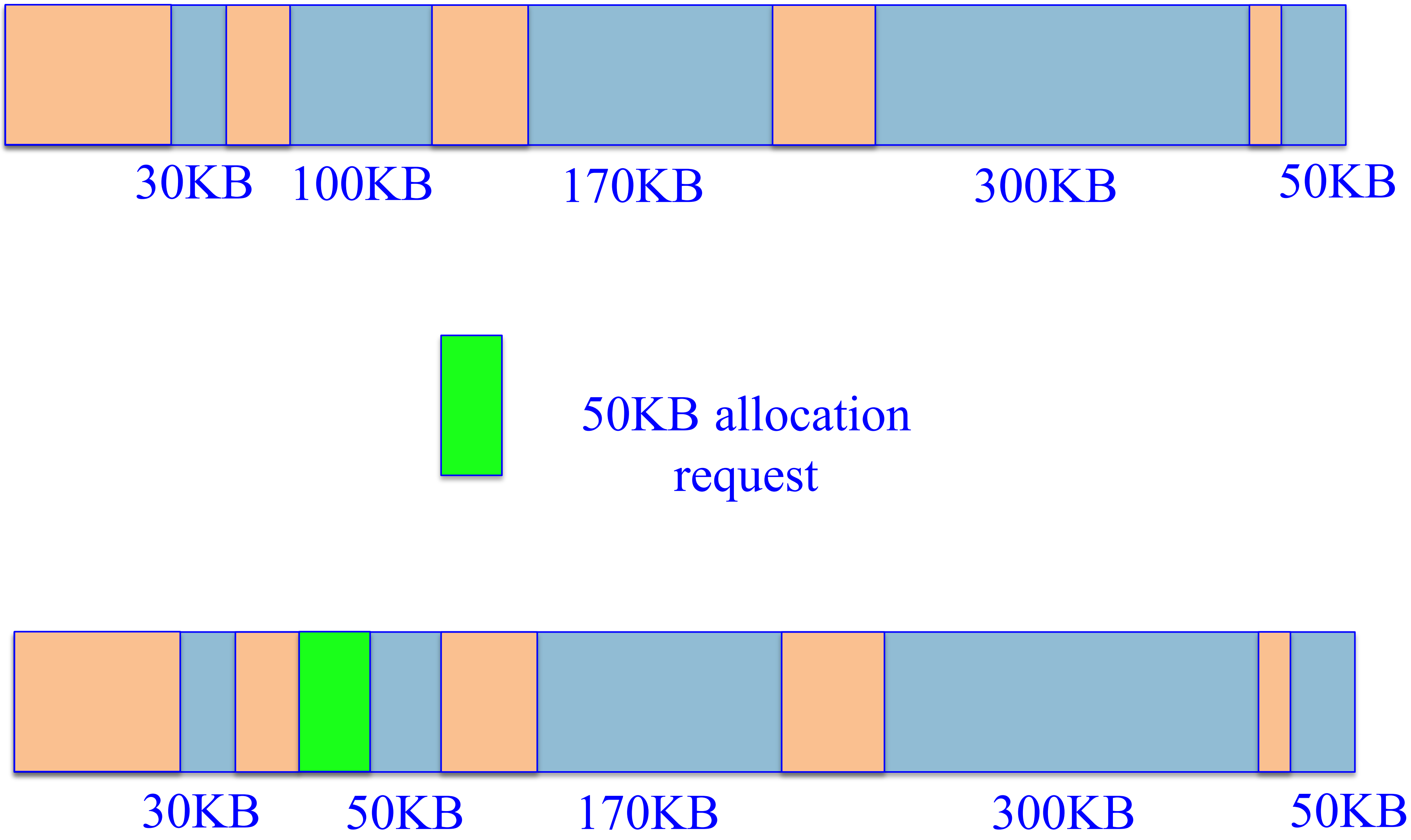
ree_list_Elem
in
use
free
in
use
in
use
block 1
free
in
use
free
free
block k

First-Fit

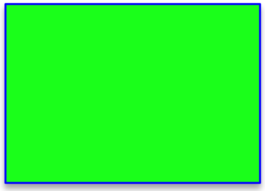


Allocated
Free
150KB
100KB
170KB
300KB
50KB
120KB allocation request
30KB
100KB
170KB
300KB
50KB

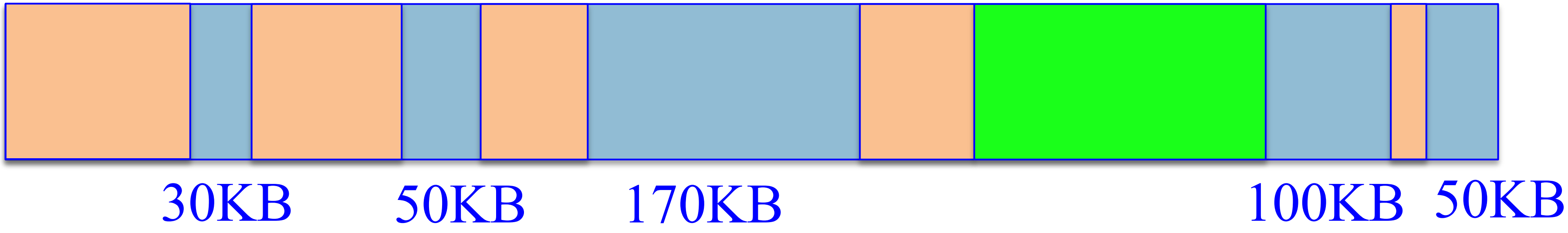
First-Fit


30KB
100KB
170KB
300KB
50KB
50KB allocation request
30KB
50KB
170KB
300KB
50KB

First-Fit



200KB allocation request



Fragmentation

- Dispersal of free memory across a possibly large number of small free cells.
- Negative effects:
  - Can prevent allocation from succeeding
  - May cause a program to use more address space, more resident pages and more cache lines.
- Fragmentation is impractical to avoid:
  - Usually the allocator cannot know what the future request sequence will be.
  - Even given a known request sequence, doing an optimal allocation is NP-hard.
- Usually There is a trade-off between allocation speed and fragmentation.

Segregated-fits Allocation

- Idea – use multiple free-list whose members are segregated by size in order to speed allocation.
- Usually a fixed number k of size values $s_0 < s_1 < \ldots < s_{k-1}$
- k+1 free lists $f_0,\ldots,f_k$
- For a free cell, b, on list f_i,

$$\text{size(b)} = s_i \;\; \forall\, 1 \leq i \;\leq k-1$$

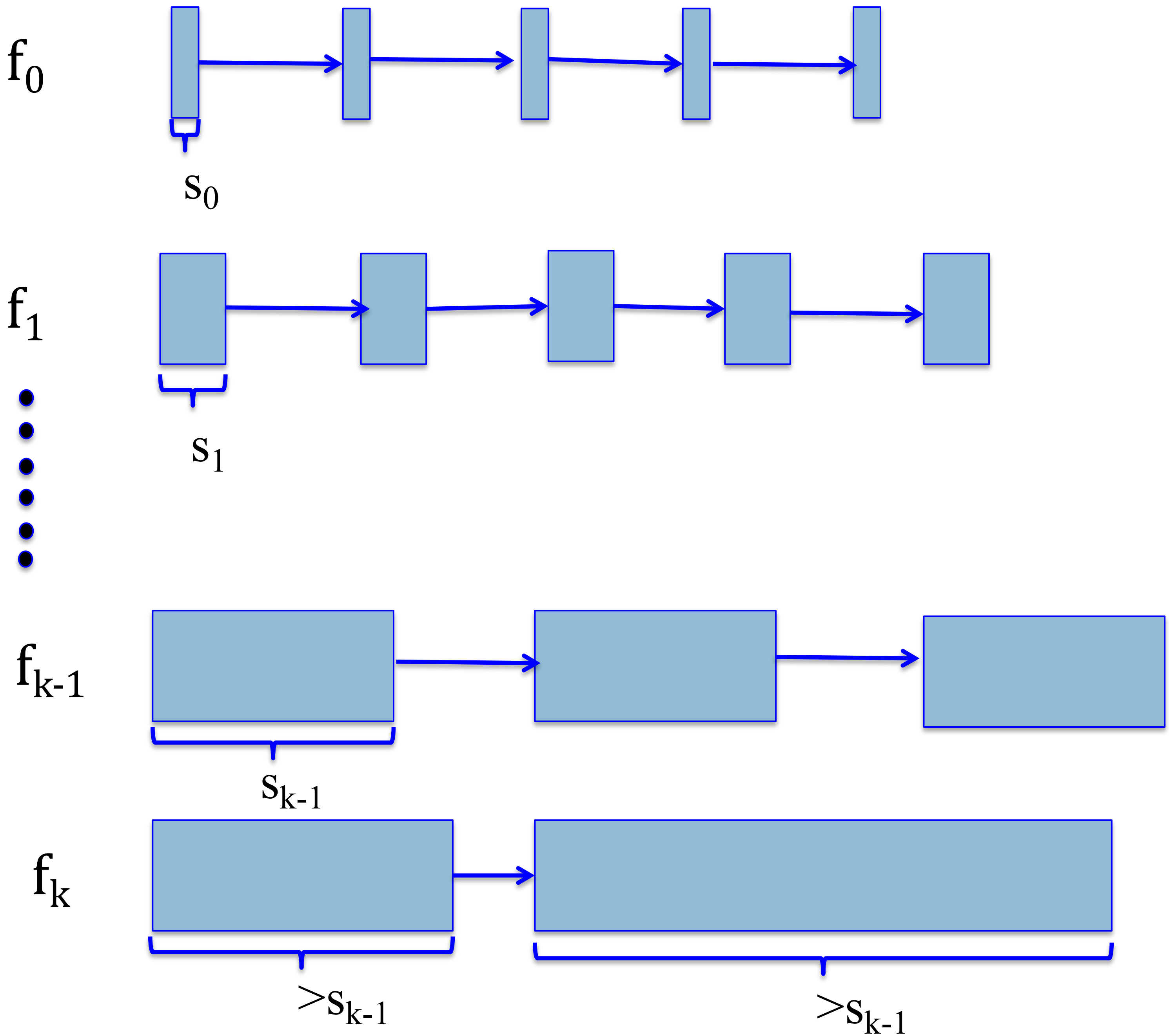
$$\text{size(b)} > s_{k-1} \text{ if } i=k$$

- When requesting a cell of size $b \leq s_{k-1}$, the allocator rounds the request size up to the smallest s_i such that $b \leq s_i$.
- S_i is called a **size class**

Segregated-fits Allocation

```
SegregatedFitAllocate(j):
        result ← remove(freeLists[j])
        if result = null
                large ← allocateBlock()
                if large = null
                        return null
                initialize(large, sizes[j])
                result ← remove(freeList[j])
        return result
```

- List f_k, for cells larger than s_k, is organized to use one of the basic single-list algorithms.
- Per-cell overheads for large cell are a bit higher but in total it is negligible.
- The main advantage: for size classes other than s_k, allocation typically requires constant time.



f_0
s_0
f_1
s_1
f_{k-1}
s_{k-1}
f_k
$>s_{k-1}$
$>s_{k-1}$

Runtime support for MM

- C’s standard library provides basic memory management
  - Gets memory pages from the OS
  - Maintains inventory of free memory cells
    - mmap(), brk()

free

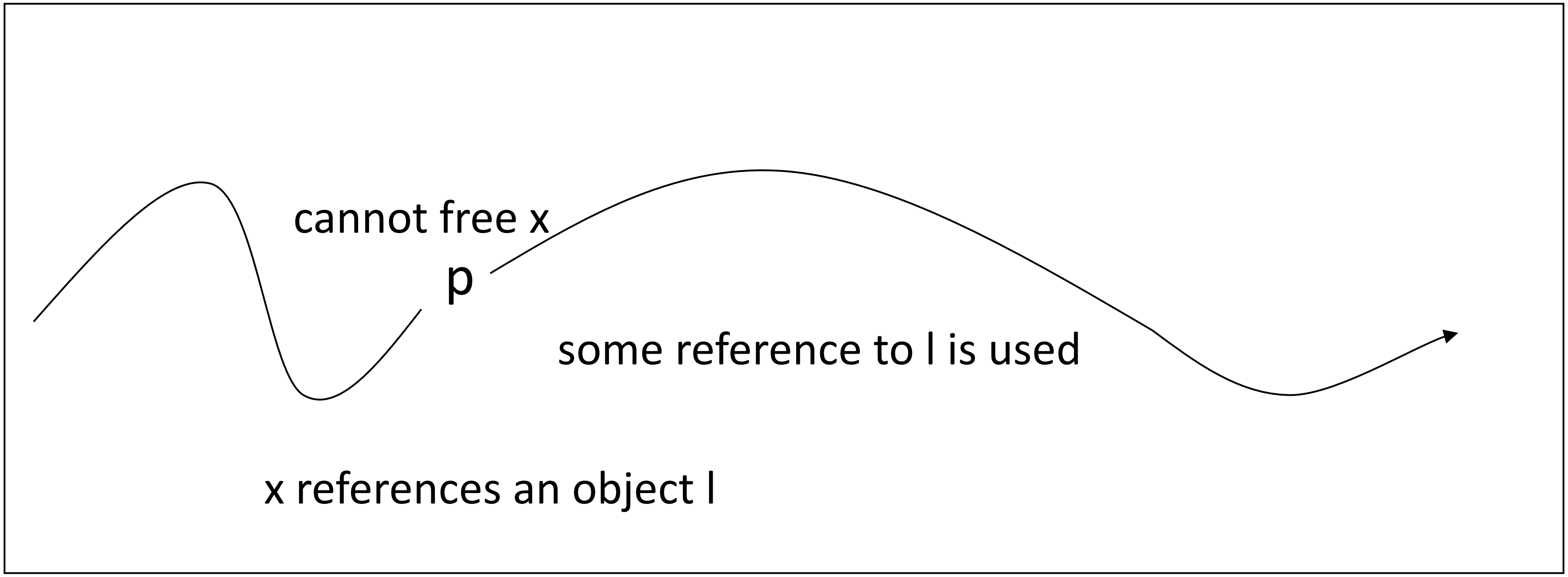
- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

When can we free an object?

```
a =  malloc(…) ;
b = a;
// free (a); ?
c = malloc (…);
if  (b == c)
     printf("unexpected equality");
```

Cannot free an object if it has a reference with a future use!

When can free x be inserted after p?



On all execution paths after p there are no uses of references to the object referenced by x ➔ inserting free x after p is valid

Automatic Memory Management

- automatically free memory when it is no longer needed
- not limited to OO languages
- prevalent in OO languages such as Java
  - also in functional languages

Garbage collection

- approximate reasoning about object liveness
- use reachability to approximate liveness
- **assume reachable objects are live**
  - non-reachable objects are dead

Garbage Collection – Classical Techniques

- reference counting
- mark and sweep
- copying

GC using Reference Counting

- add a reference-count field to every object
  - how many references point to it
- when (rc==0) the object is non reachable
  - non reachable => dead
  - can be collected (deallocated)

Managing Reference Counts

- Each object has a reference count o.RC
- A newly allocated object o gets o.RC = 1
  - why?

- write-barrier for reference updates

```
update(x,old,new) {
  old.RC--;
  new.RC++;
  if (old.RC == 0) collect(old);
}
```

- collect(old) will decrement RC for all children and recursively collect objects whose RC reached 0.

Cycles!

- cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
- several approaches for dealing with cycles
  - ignore
  - periodically invoke a tracing algorithm to collect cycles
  - specialized algorithms for collecting cycles

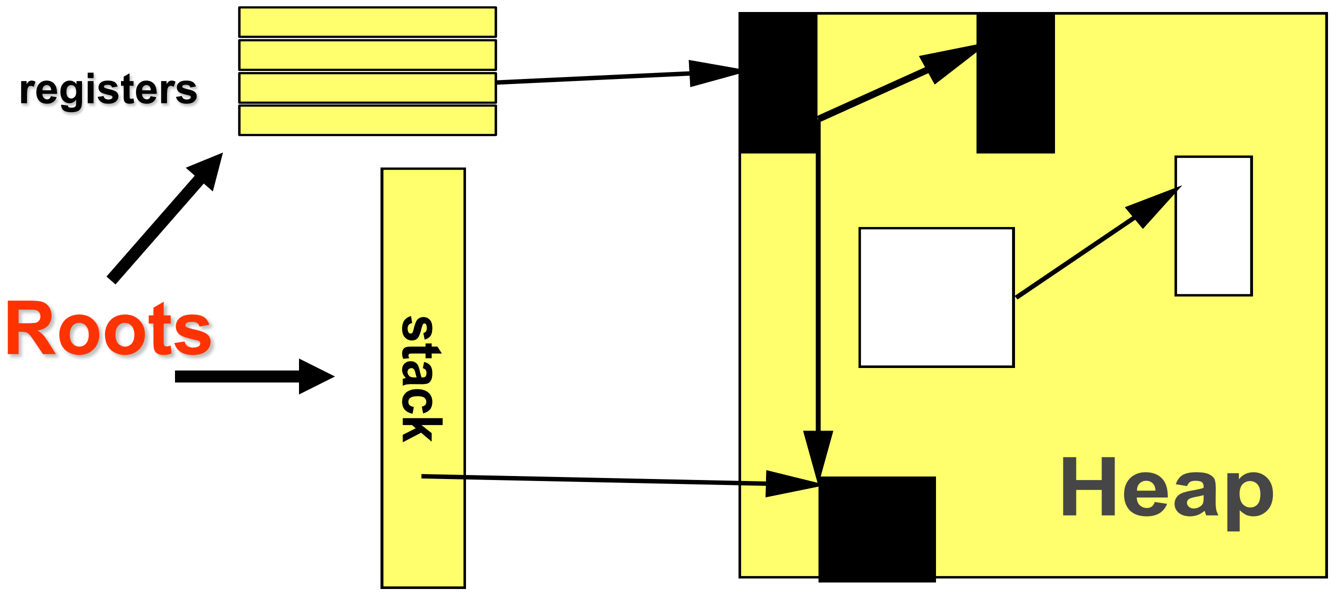
The Mark-and-Sweep Algorithm [McCarthy 1960]

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object

- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects

The Mark-Sweep algorithm

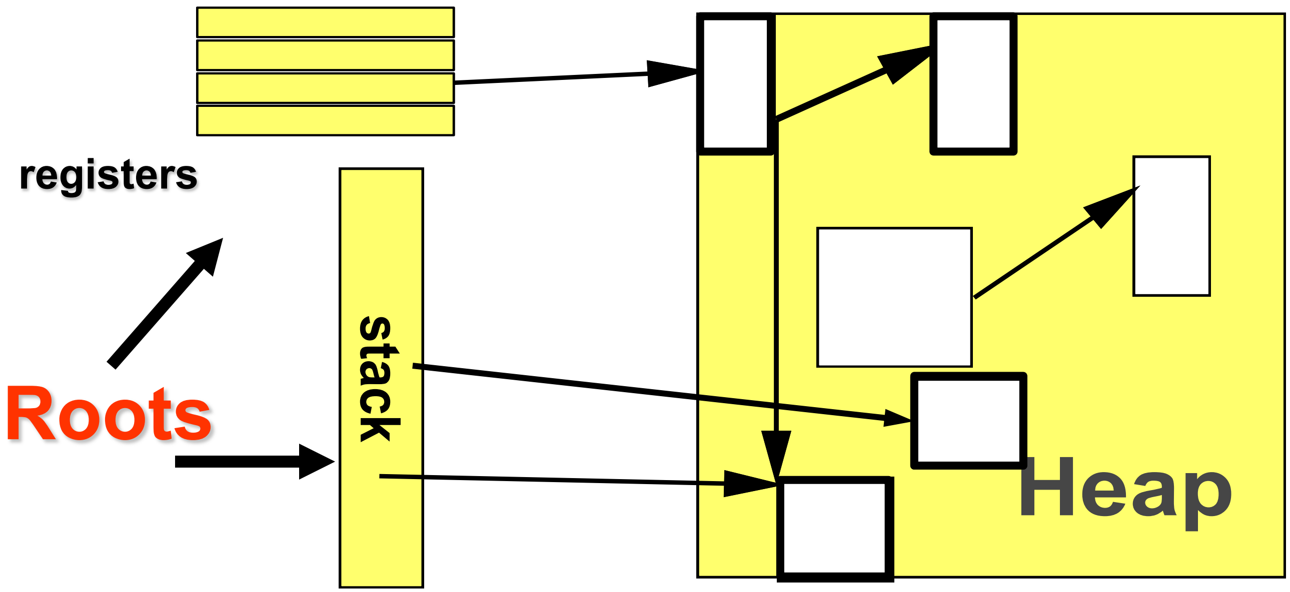
- Traverse live objects & mark black.
- White objects can be reclaimed.



heap

The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.



Triggering

```
New(A)=
   if free_list is empty
          mark_sweep()
           if free_list is empty
                   return ("out-of-memory")
   pointer = allocate(A)
   return (pointer)
```

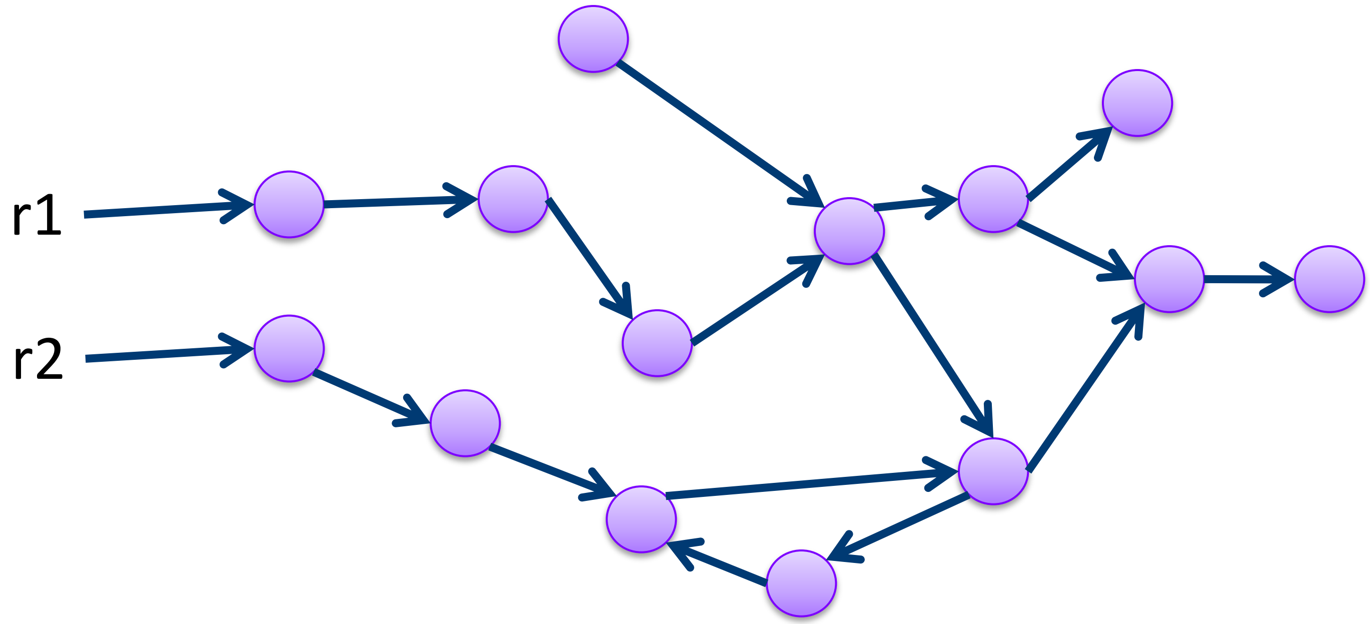
Basic Algorithm

```
mark_sweep()=
for Ptr in Roots
        mark(Ptr)
sweep()
```

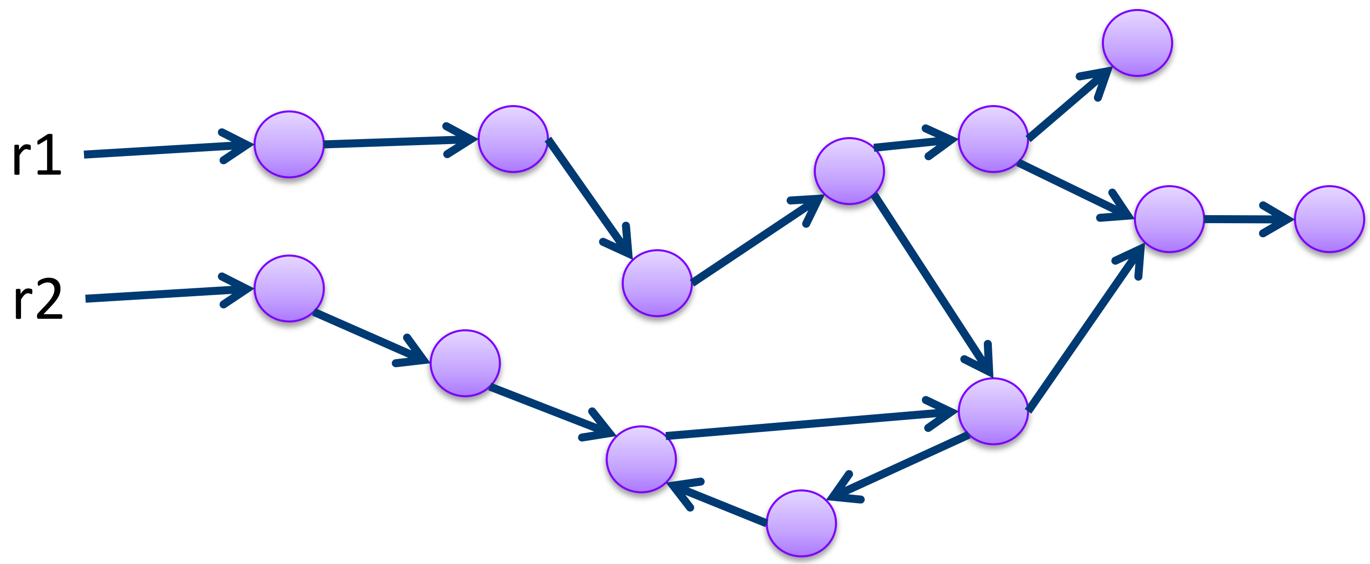
```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

```
Sweep()=
p = Heap_bottom
while (p < Heap_top)
        if (mark_bit(p) == unmarked) then free(p)
        else mark_bit(p) = unmarked;
        p=p+size(p)
```

Mark&Sweep Example



Mark&Sweep Example



Mark&Sweep in Depth

```
mark(Obj)=
if mark_bit(Obj) == unmarked
        mark_bit(Obj)=marked
        for C in Children(Obj)
                mark(C)
```

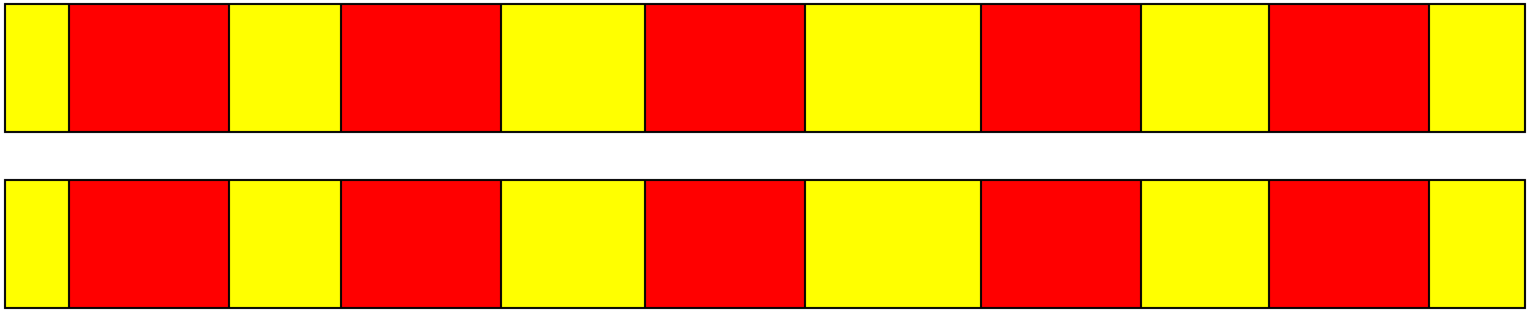
- How much memory does it consume?
  - Recursion depth?
  - Can you traverse the heap without worst-case O(n) stack?
    - Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

Properties of Mark & Sweep

- Most popular method today
- Simple
- Does not move objects, and so **heap may fragment**
- Complexity
  - ☺ Mark phase: live objects (dominant phase)
  - ☹ Sweep phase: heap size
- Termination: each pointer traversed once
- Engineering tricks used to improve performance

Mark-Compact

- During the run objects are allocated and reclaimed
- Gradually, the heap gets fragmented
- When space is too fragmented to allocate, a compaction algorithm is used
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations
- Compaction is very costly and we attempt to run it infrequently, or only partially



Mark Compact

- Important parameters of a compaction algorithm
  - Keep order of objects?
  - Use extra space for compactor data structures?
  - How many heap passes?
  - Can it run in parallel on a multi-processor?

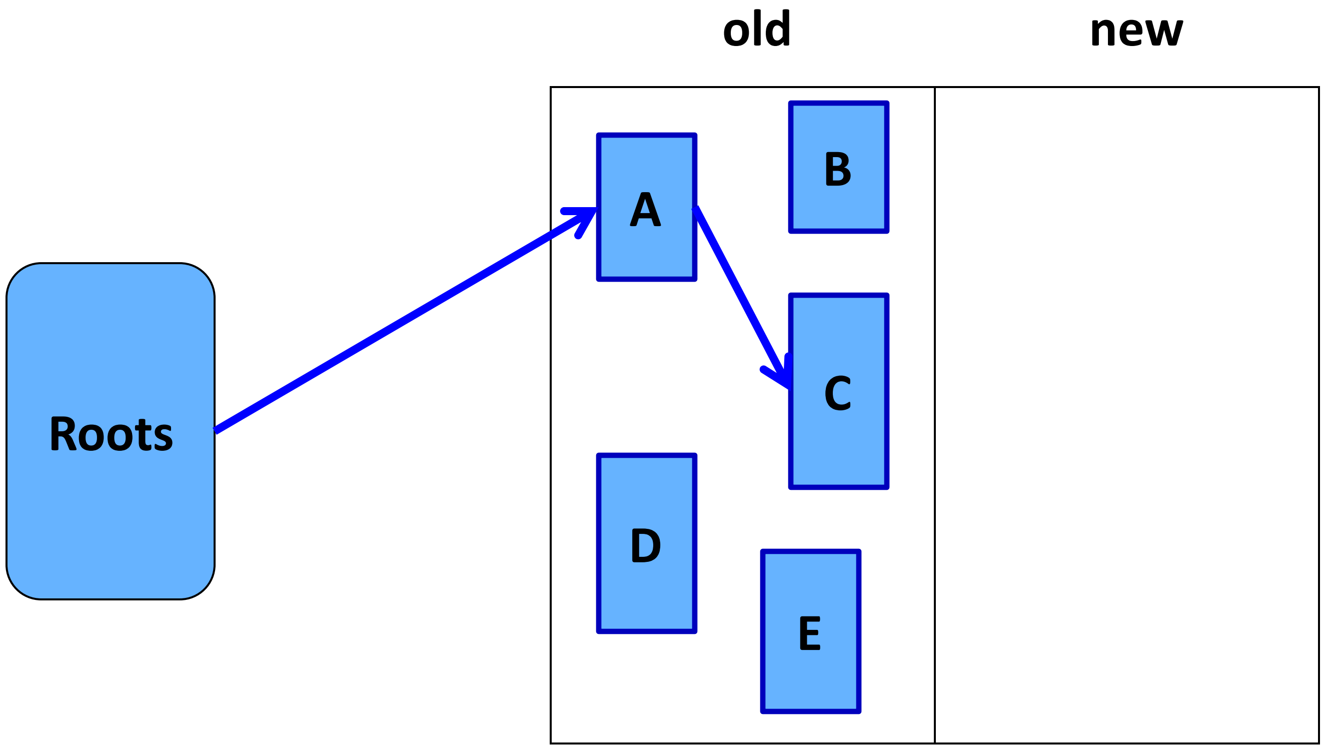
- We do not elaborate in this intro

Copying GC

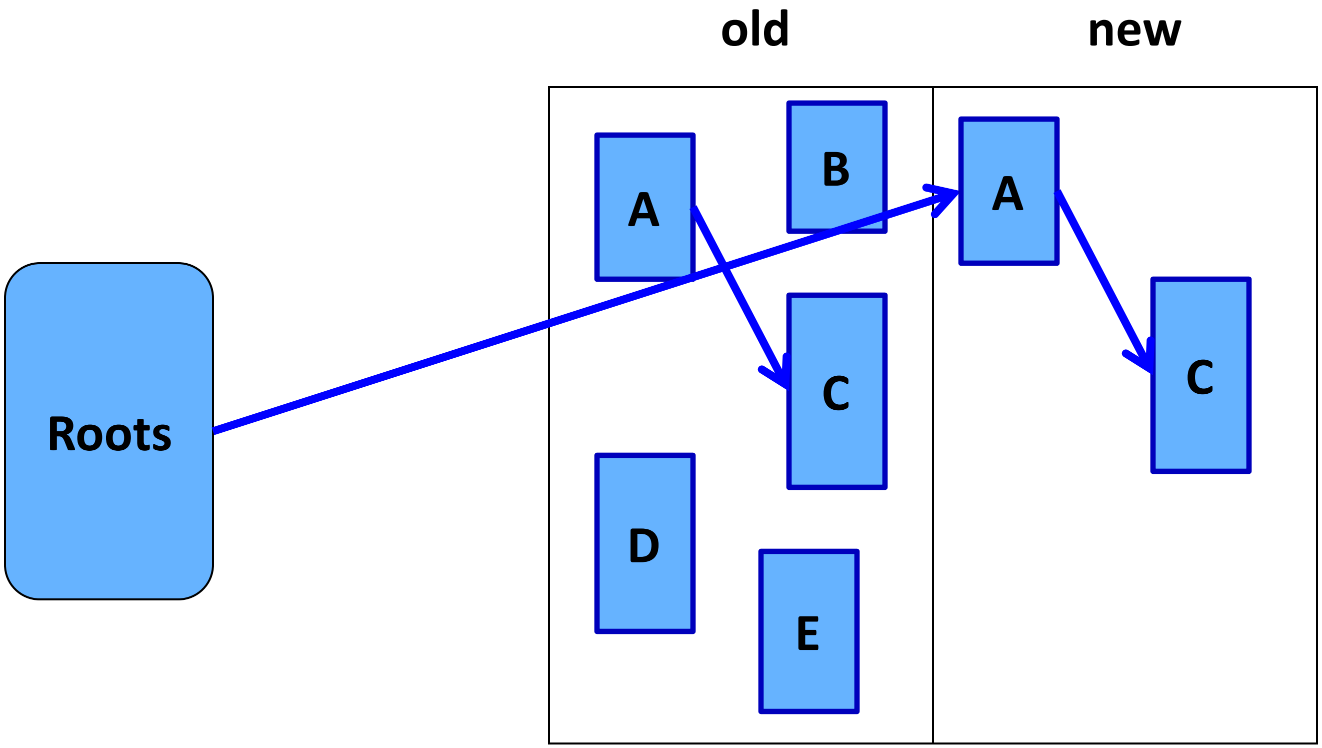
- partition the heap into two parts
  - old space
  - new space

- Copying GC algorithm
  - copy all **reachable** objects from old space to new space
  - swap roles of old/new space

Example



Example



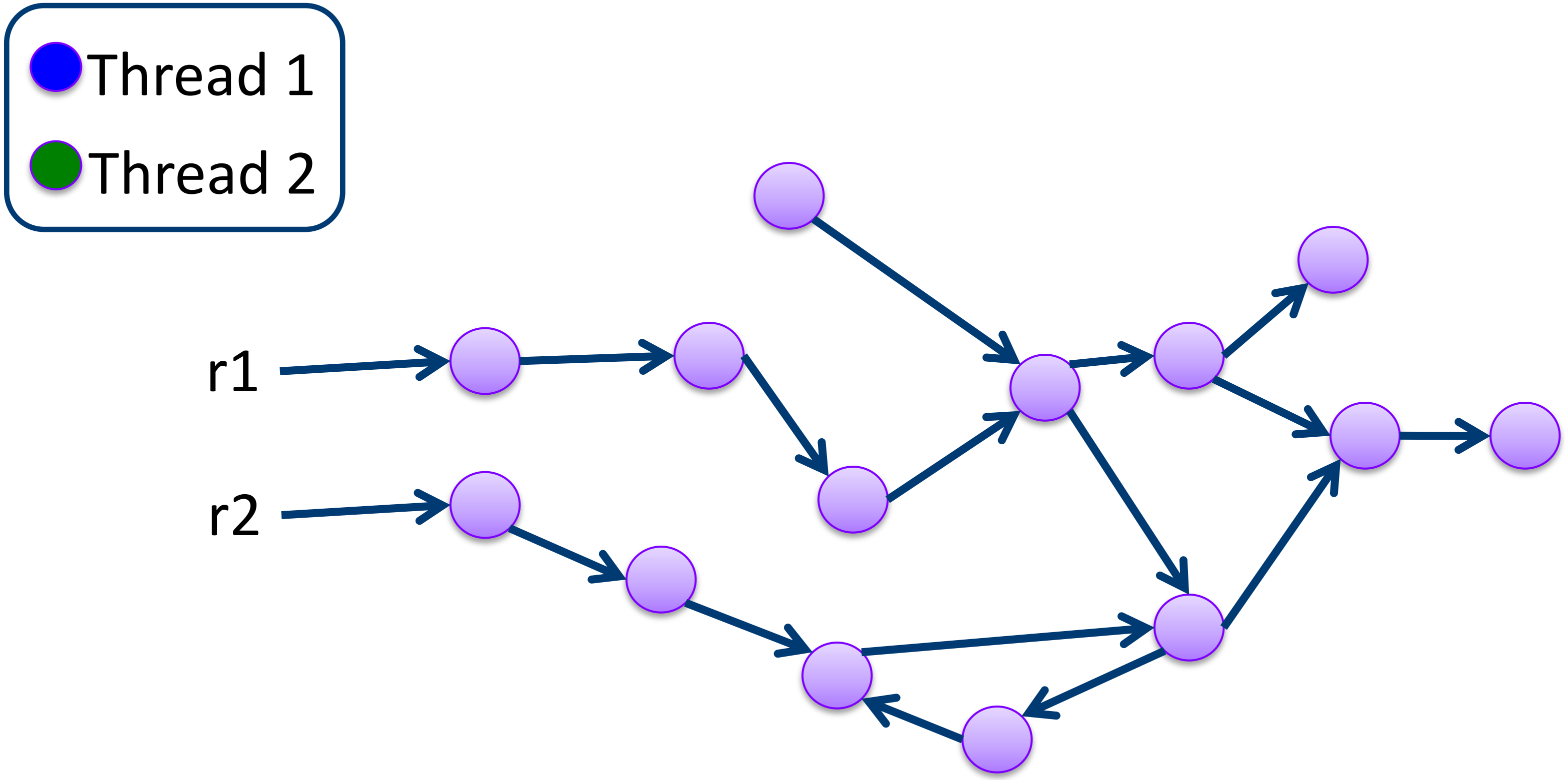
Properties of Copying Collection

- Compaction for free
- Major disadvantage: **half of the heap is not used**
- "Touch" only the live objects
  - Good when most objects are dead
  - Usually most new objects are dead
    - Some methods use a small space for young objects and collect this space using copying garbage collection

A very simplistic comparison

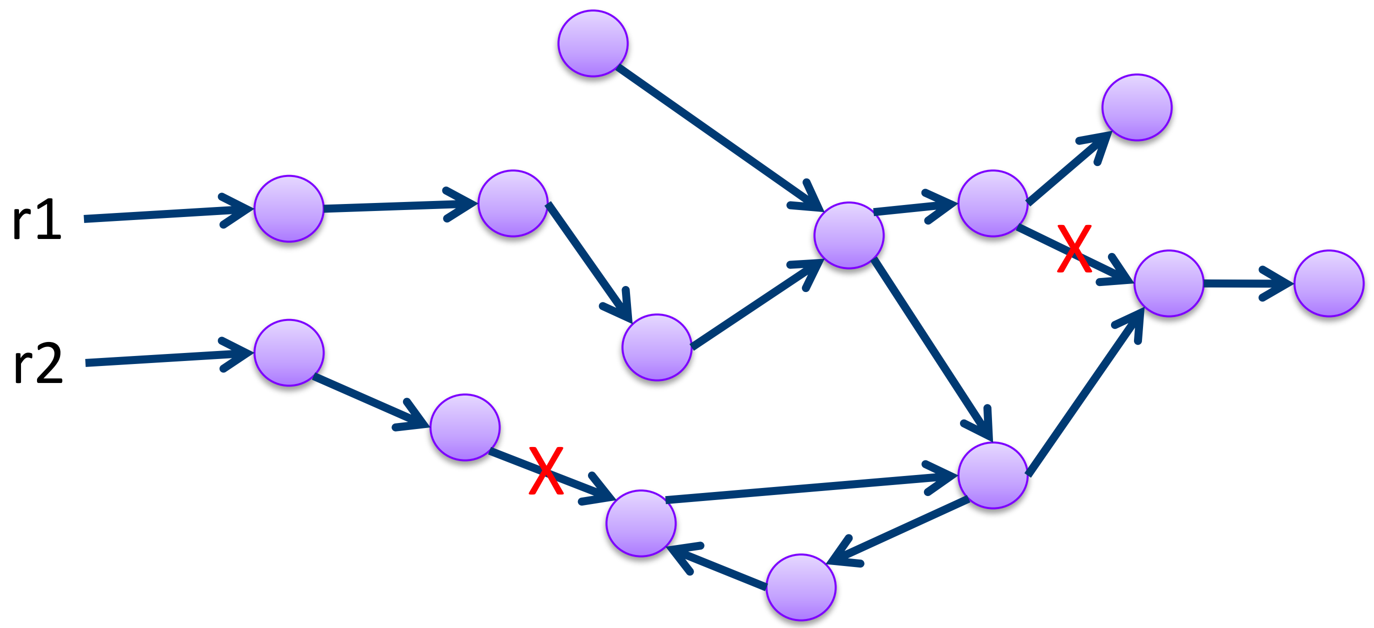
	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Mark = live objects Sweep = Size of heap	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

Parallel Mark&Sweep GC



Parallel GC: mutator is stopped, GC threads run in parallel

Concurrent Mark&Sweep Example



Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator

Conservative GC

- How do you track pointers in languages such as C ?
  - Any value can be cast down to a pointer
- **How can you follow pointers in a structure?**

- Easy – be conservative, consider anything that can be a pointer to be a pointer
- Practical! (e.g., Boehm collector)

Conservative GC

- Can you implement a conservative **copying GC**?
- What is the problem?
- **Cannot update pointers to the new address… you don't know whether the value is a pointer, cannot update it**

Modern Memory Management

- Considers standard program properties
- Handle parallelism
  - Stop the program and collect in parallel on all available processors
  - Run collection concurrently with the program run
- Cache consciousness
- Real-time

Terminology Recap

- Heap, objects
- Allocate, free (deallocate, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, compaction, tracing algorithms
- Fragmentation