

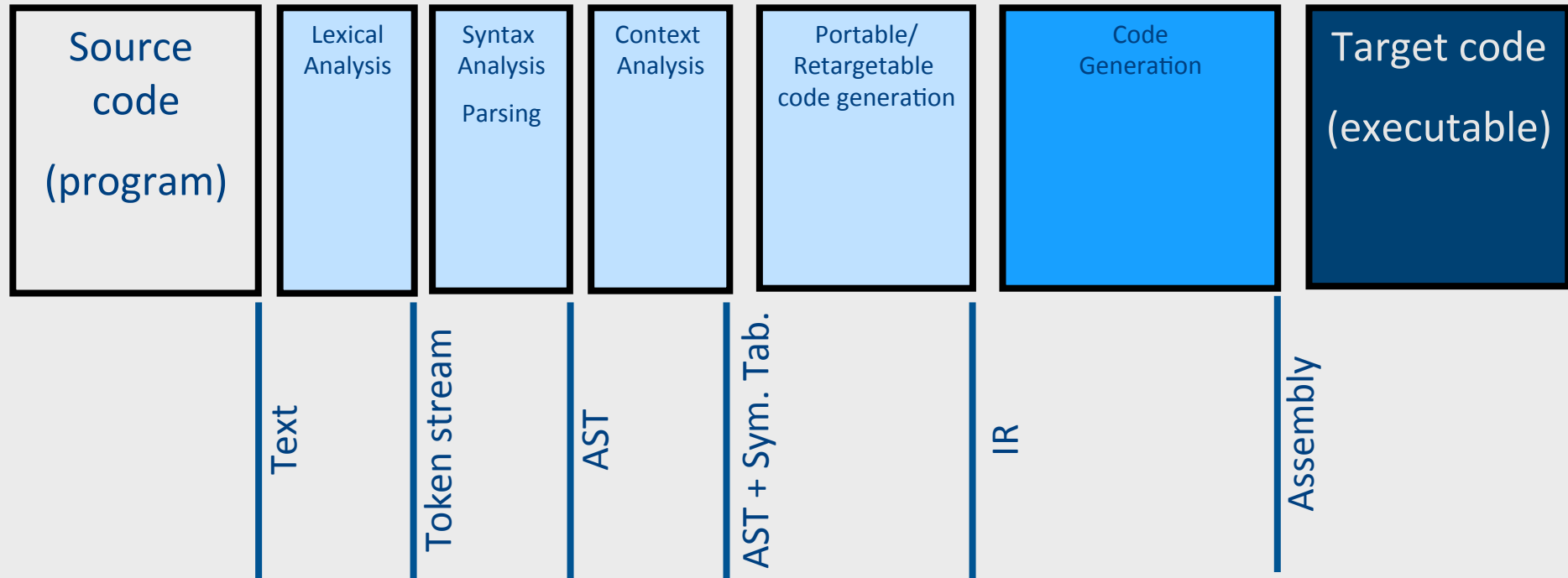
Compilation

0368-3133 (Semester A, 2013/14)

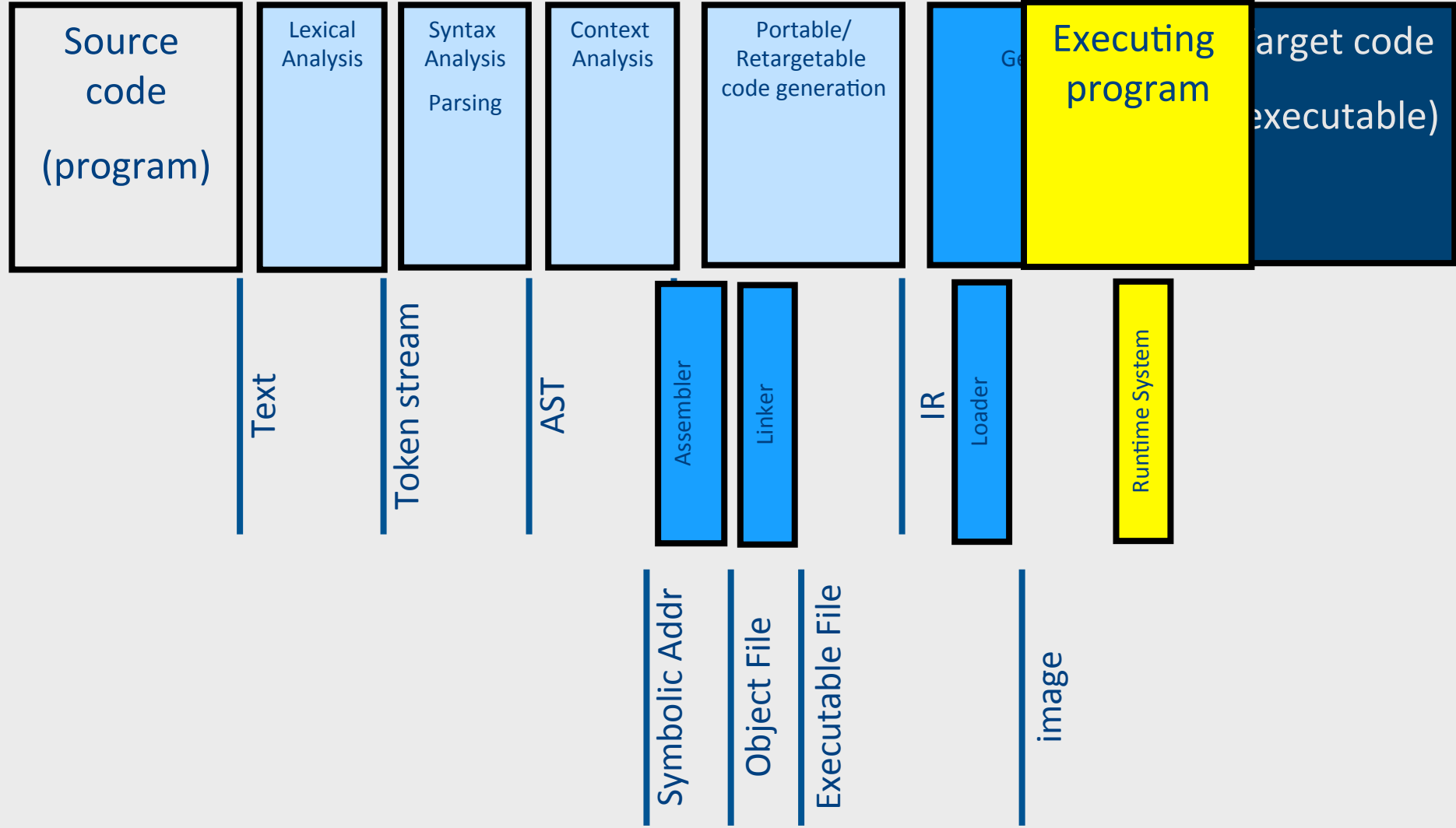
Lecture 13b: Memory Management

Noam Rinetzky

Stages of compilation



Compilation → Execution



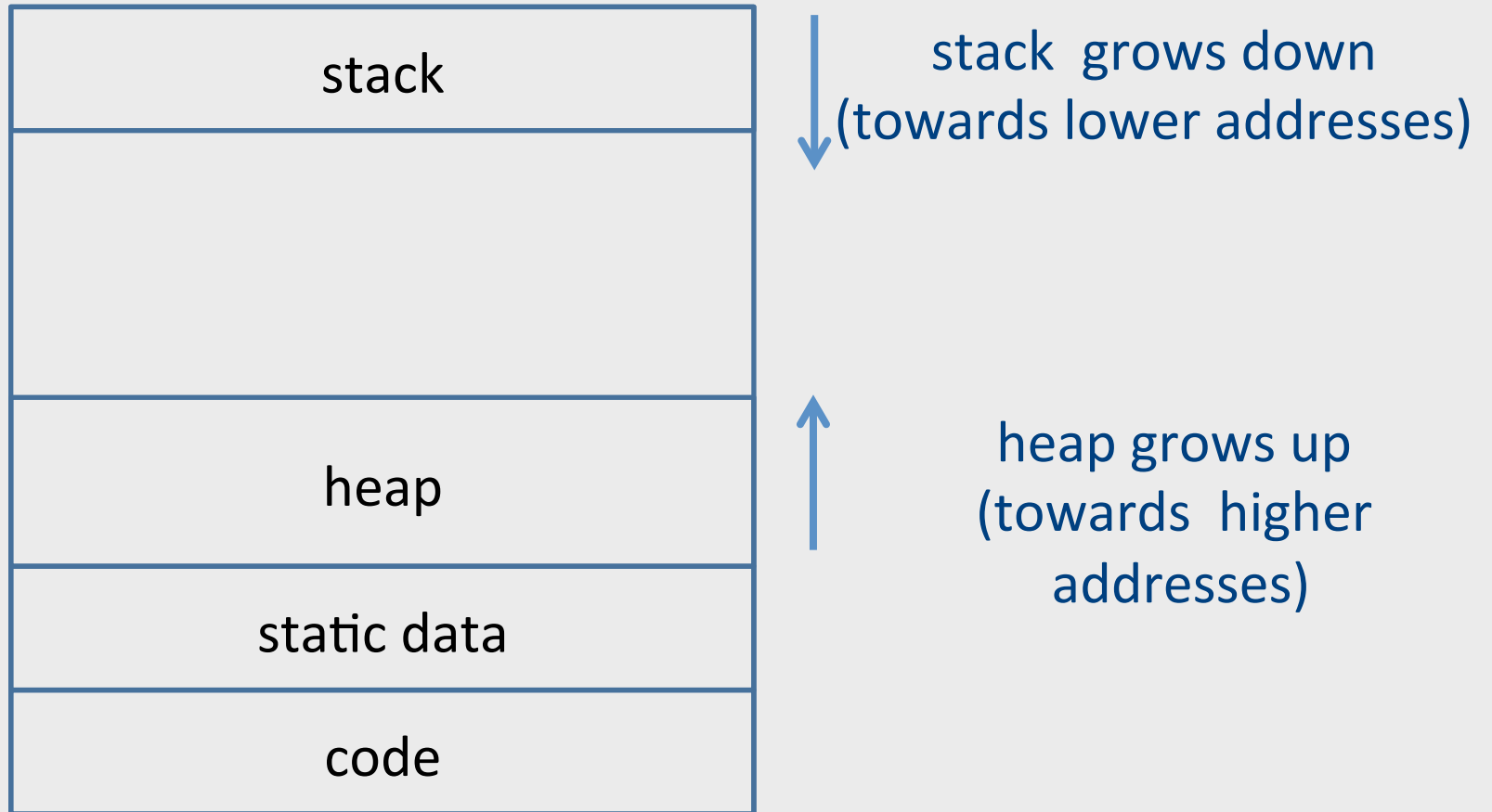
Runtime Environment

- 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

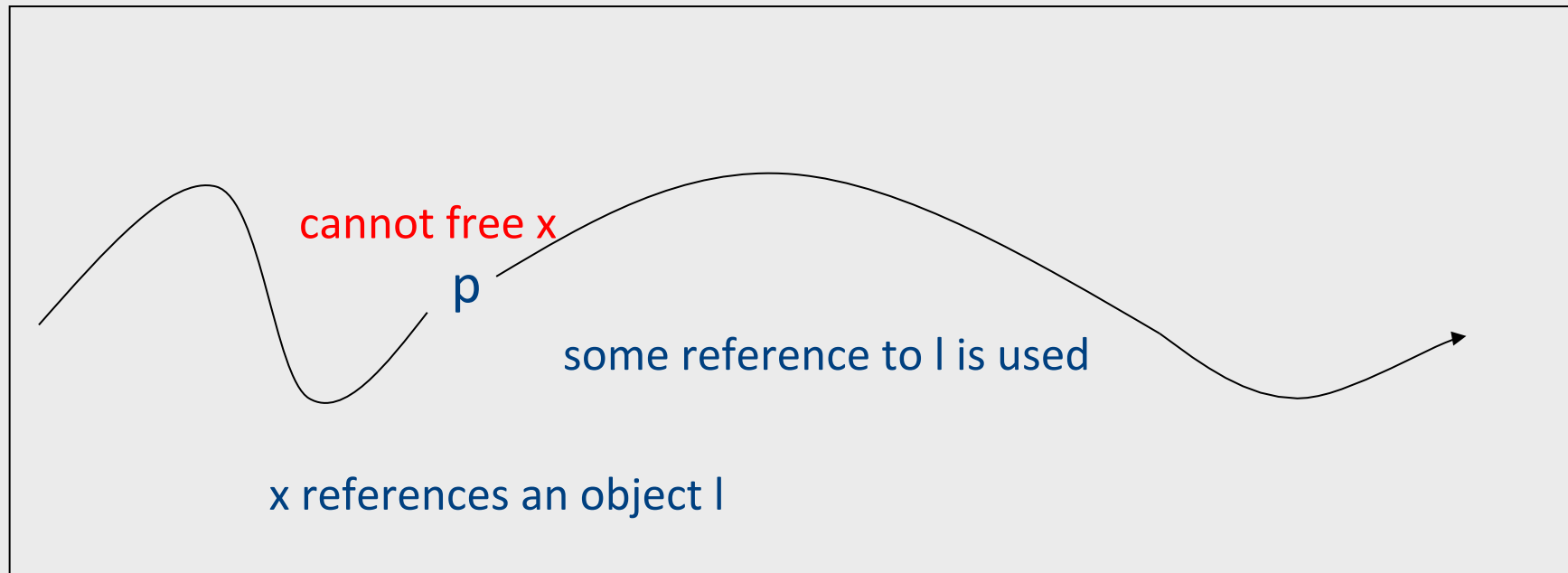
- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error

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

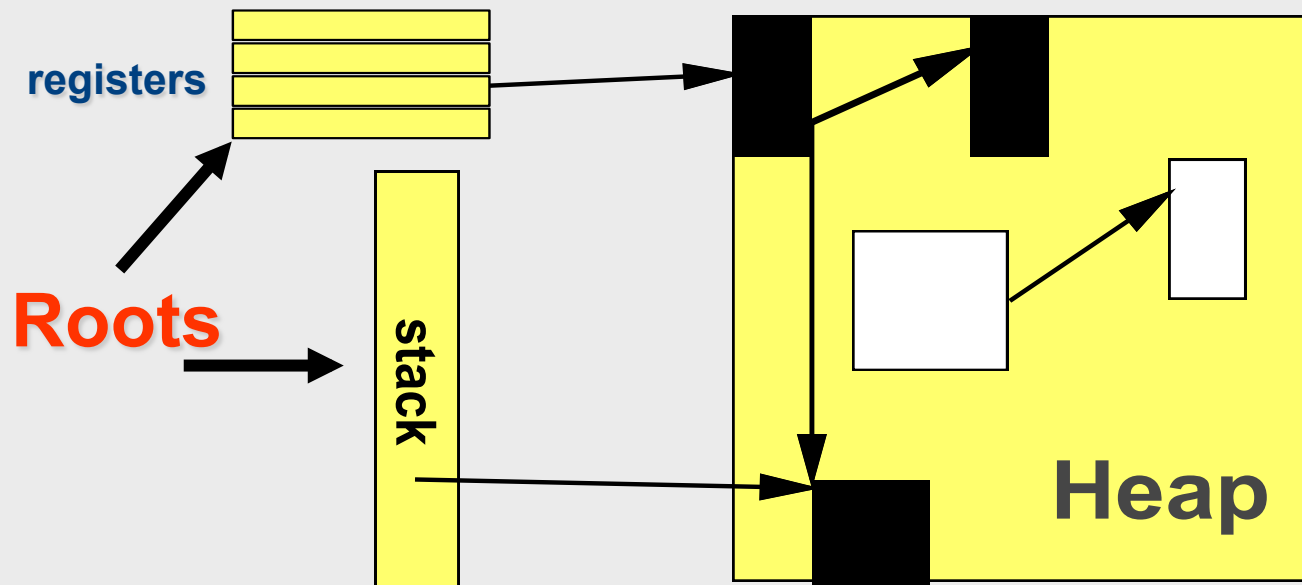
The Mark-and-Sweep Algorithm

[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.



Note!
This is not
the heap data
structure!

Triggering

Garbage collection is triggered by allocation

```
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 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)

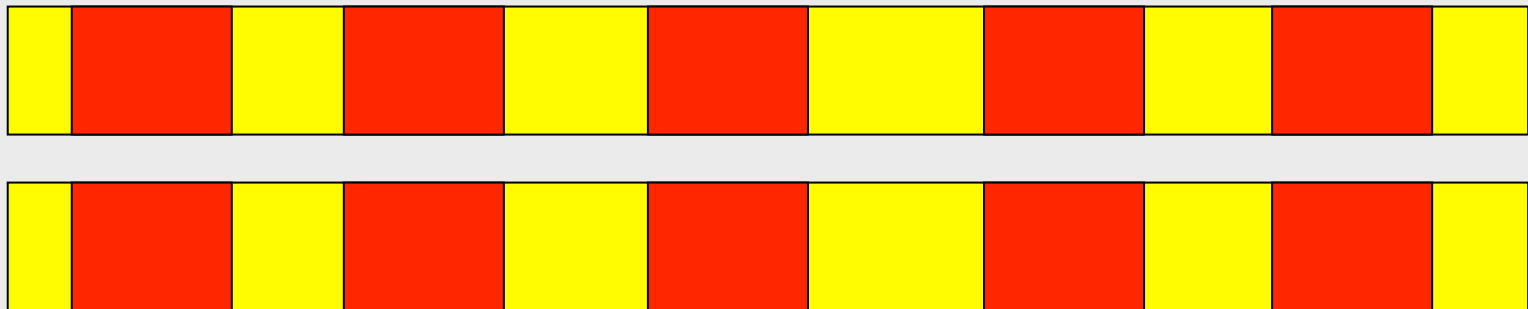
Properties of Mark & Sweep

- 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

The
Heap



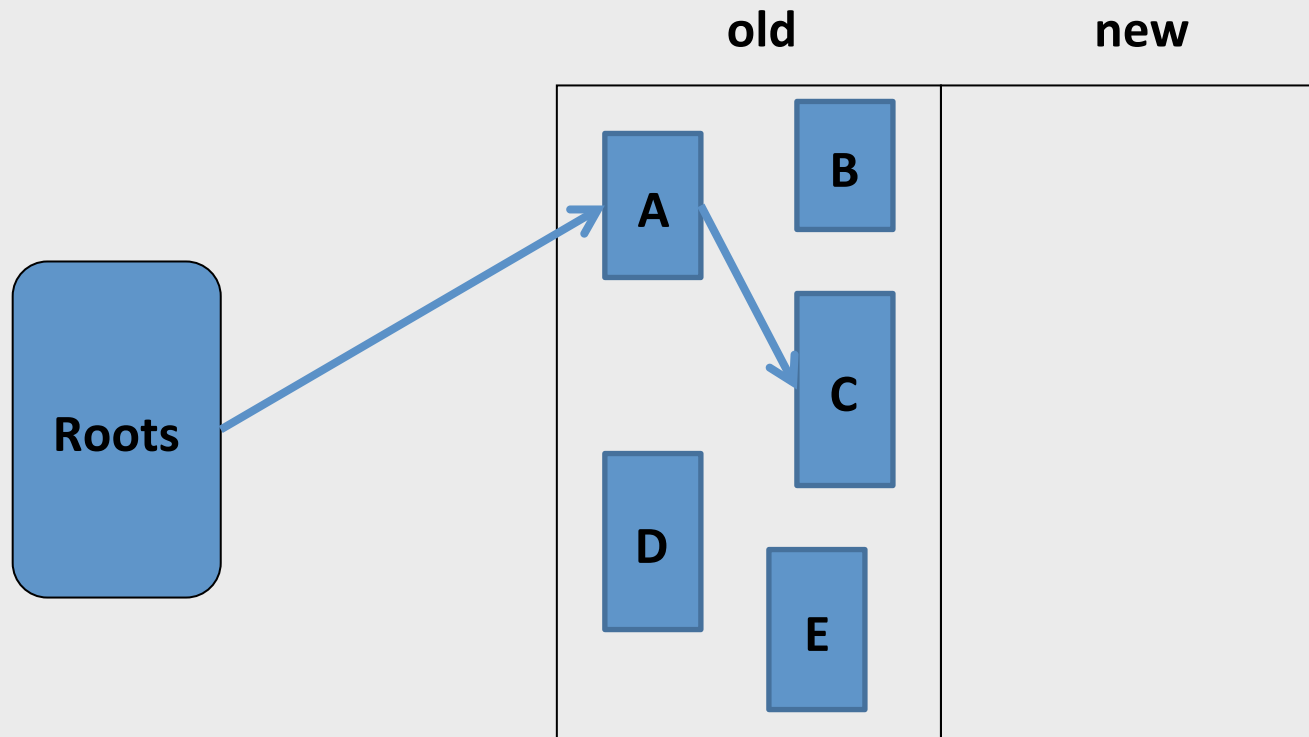
Mark Compact

- 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

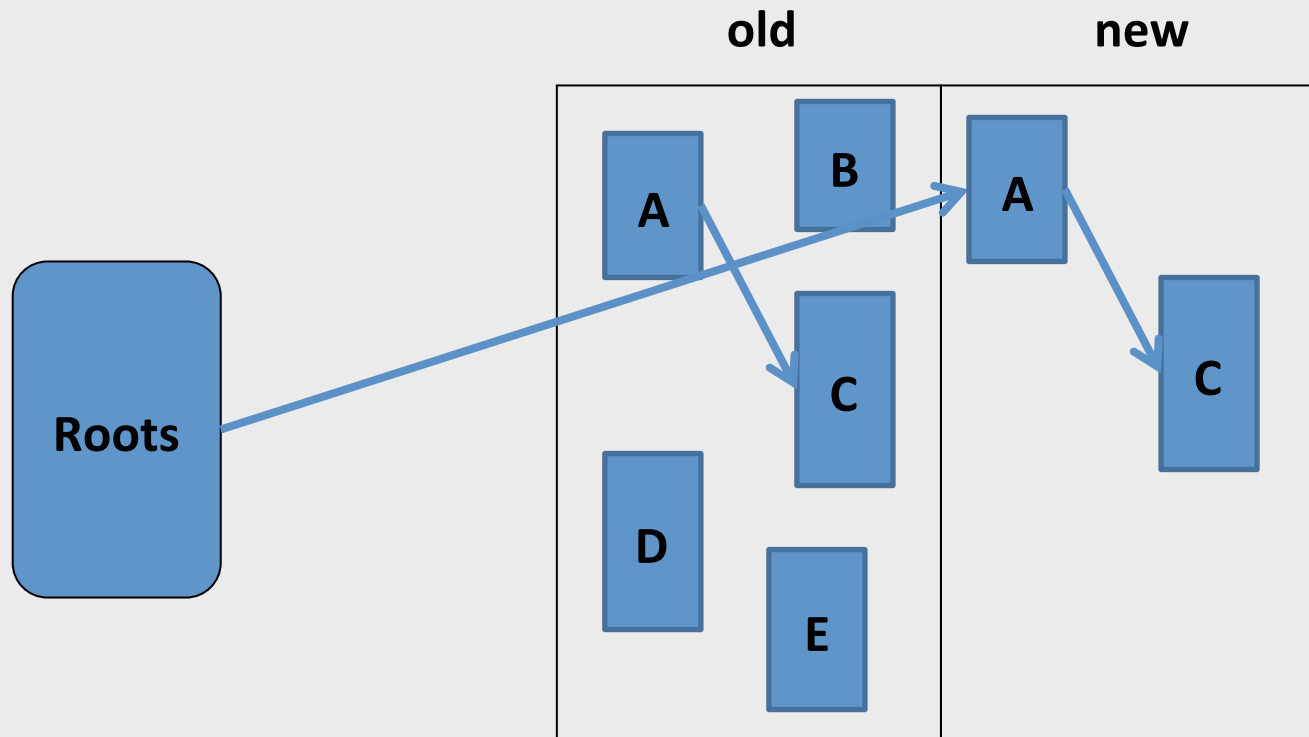
Copying GC

- 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



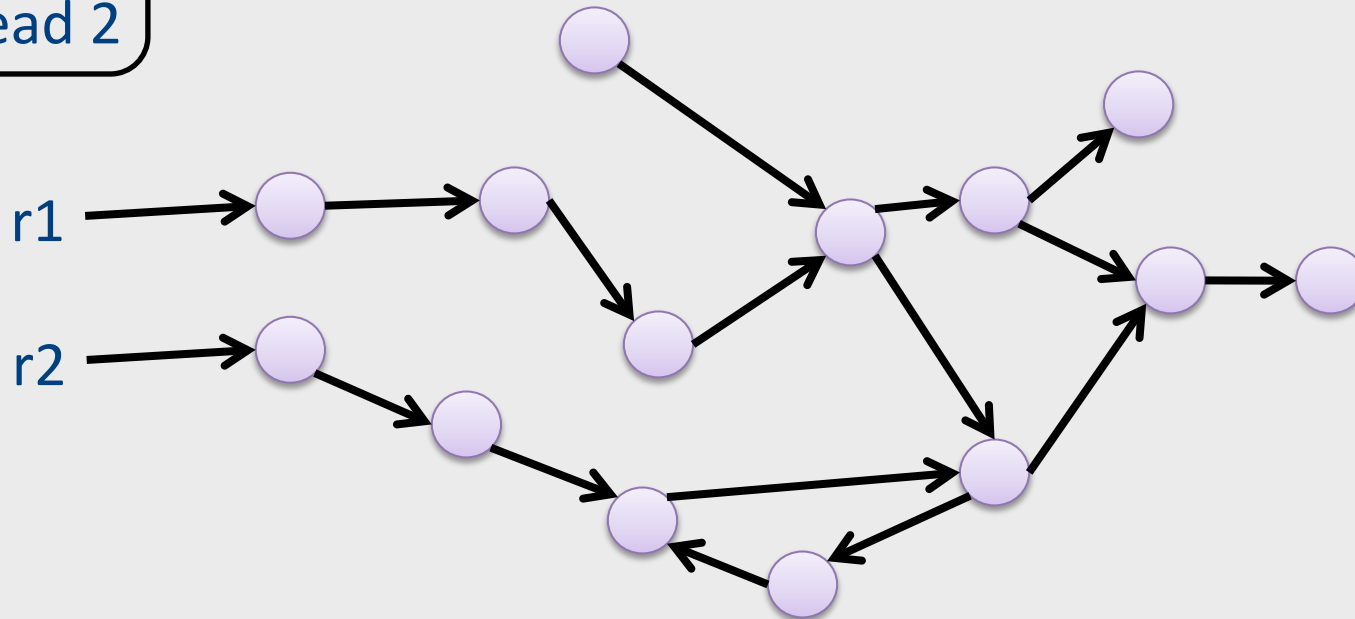
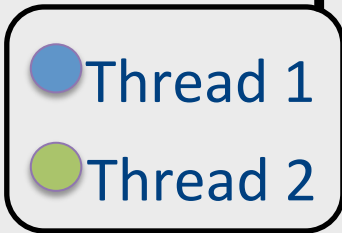
Properties of Copying Collection

- 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

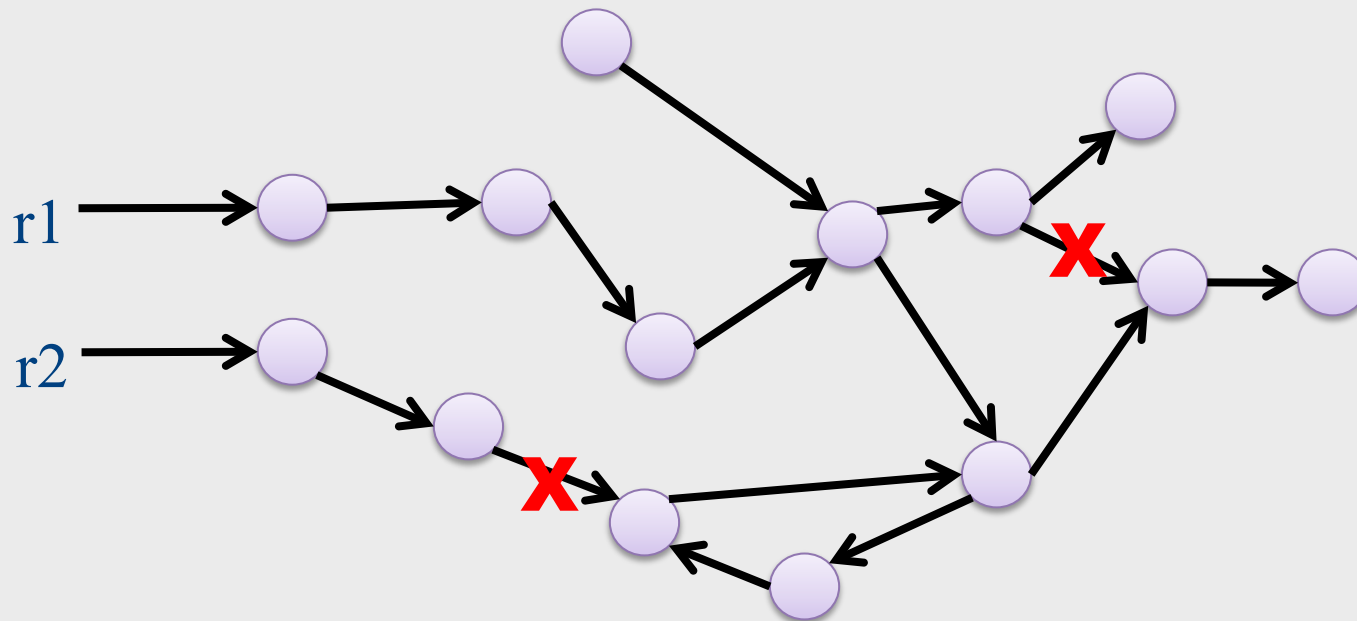
	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Size of heap (live objects)	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More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

Parallel Mark&Sweep GC



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

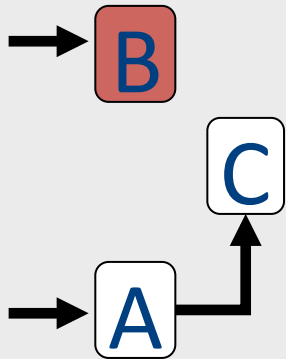
Concurrent Mark&Sweep Example



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

Problem: Interference

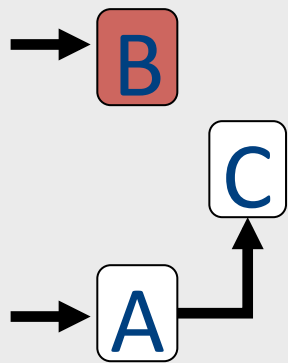
SYSTEM = MUTATOR || GC



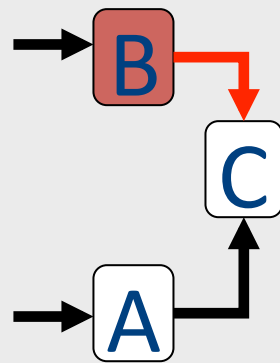
1. GC traced B

Problem: Interference

SYSTEM = MUTATOR || GC



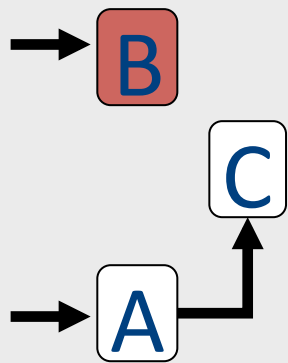
1. GC traced B



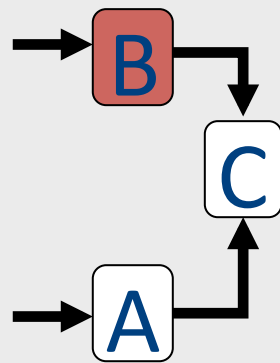
2. Mutator
links C to B

Problem: Interference

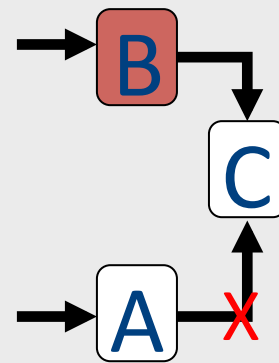
SYSTEM = MUTATOR || GC



1. GC traced B



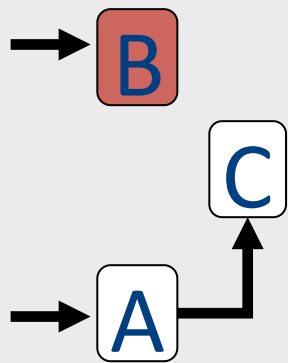
2. Mutator links C to B



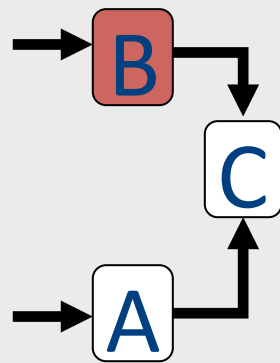
3. Mutator unlinks C from A

Problem: Interference

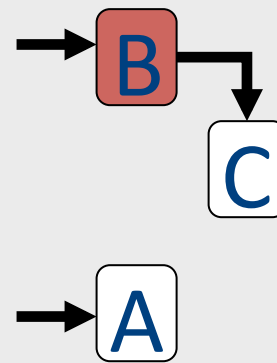
SYSTEM = MUTATOR || GC



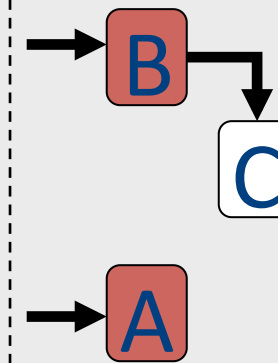
1. GC traced B



2. Mutator links C to B



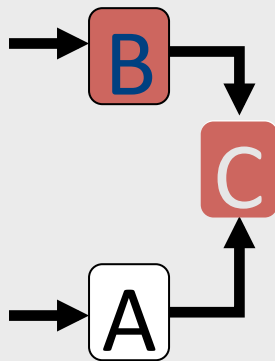
3. Mutator unlinks C from A



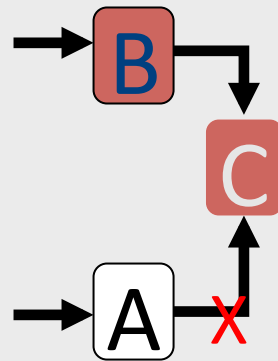
4. GC traced A

The 3 Families of Concurrent GC Algorithms

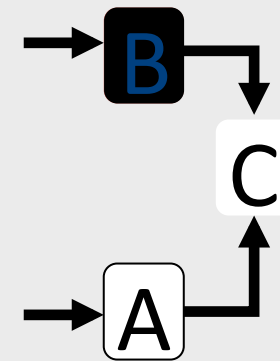
1. Marks C when C is linked to B (DIJKSTRA)



2. Marks C when link to C is removed (YUASA)



3. Rescan B when C is linked to B (STEELE)



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

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**

Terminology Recap

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

The End