ECS652: Compilers (Spring 2018)

Lecture 10

Runtime system

Greta Yorsh

Outline

✓ Code generation for expressions

✓ Code generation for methods

✓ Code generation for objects

✓ Operational semantics

- Runtime system

Runtime system

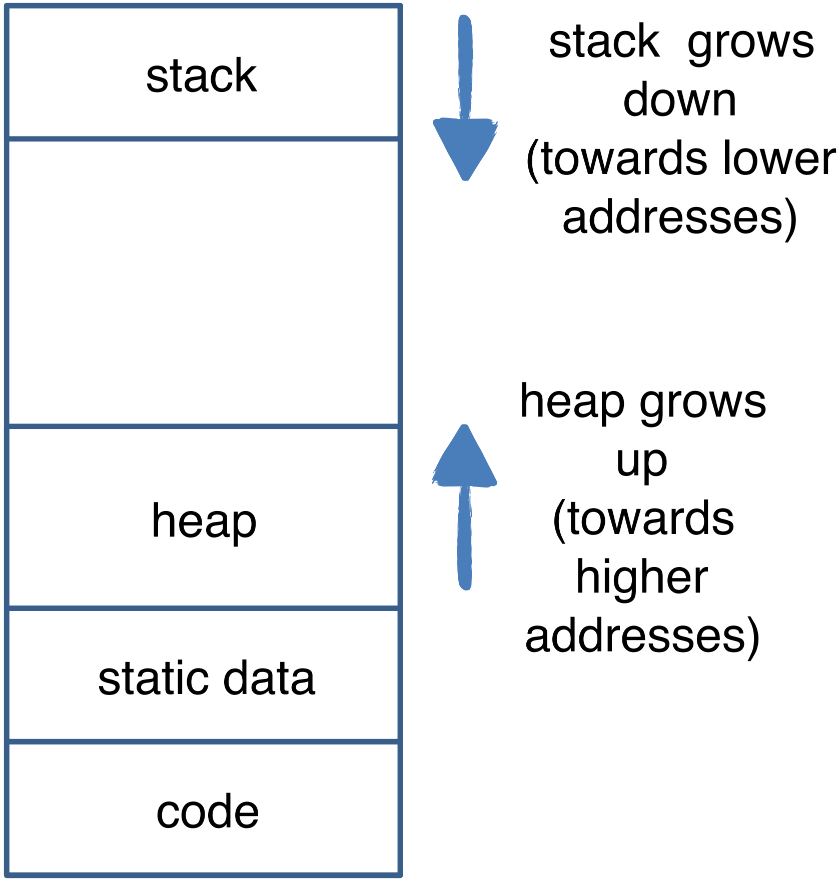
- **Mediates between OS and programming language**
- Hides details of the os and the machine from the programmer
- Ranges from simple support functions all the way to a full-fledged VM
  - Microsoft CLR for C# and other .Net languages
  - Java Virtual Machine (JVM) by Sun/Oracle or IBM for Java, Scala, …
  - Mozilla SpiderMonkey for JavaScript
- Handles common tasks
  - memory management, including garbage collector (GC)
  - dynamic optimizations such as Just-In-Time (JIT) compiler
  - thread management
  - exception handling
  - security
  - debugging

Memory management

- Tasks
    - allocation
    - deallocation
- **Manual** memory management
    - programmer responsible for calling allocation and deallocation explicitly
- **Automatic** memory management
    - garbage collector inside runtime system automatically deallocates memory

Where do we allocate data?

- Runtime stack
  - stack frame deallocated (popped) upon method return
  - lifetime of allocated data is limited by method lifetime
- Dynamic memory allocation on the heap



stack
heap
static data
code
stack grows down (towards lower addresses)
heap grows up (towards higher addresses)

Alignment

- Typically, memory access is **word-aligned**: address is a multiple of 4-bytes or 8-bytes
- Unaligned access is often slower

Alignment

- Typically, memory access is **word-aligned**: address is a multiple of 4-bytes or 8-bytes
- Unaligned access is often slower

- How do we allocate data of size 5 bytes?

Alignment

- Typically, memory access is **word-aligned**: address is a multiple of 4-bytes or 8-bytes
- Unaligned access is often slower

- How do we allocate data of size 5 bytes?
- Padding: the space until the next aligned addresses is kept empty

Manual memory management

- Examples: C, C++, Pascal, Modula, Rust

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

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
  - allocates a chunk of memory, without regard to its type

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
  - allocates a chunk of memory, without regard to its type
- Where is malloc implemented?

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
  - allocates a chunk of memory, without regard to its type
- Where is malloc implemented?
- How does it work?

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
  - allocates a chunk of memory, without regard to its type
- Where is malloc implemented?
- How does it work?
- How does malloc guarantee alignment?
  - after all, you don’t know what type it is allocating for

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
    - allocates a chunk of memory, without regard to its type
- Where is malloc implemented?
- How does it work?
- 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

Allocating memory

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
  - allocates a chunk of memory, without regard to its type
- Where is malloc implemented?
- How does it work?
- 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-2.25)

Deallocating memory

- 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); // ?????
...
```

When can we free an object?

```
a = malloc(…) ;
b = a;
free (a); // ?????
```

When can we free an object?

```
a = malloc(...) ;
b = a;
free (a); // ?????
*b = c;
```

When can we free an object?

```
a = malloc(…) ;
b = a;
free (a); // ?????
*b = c;
```

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

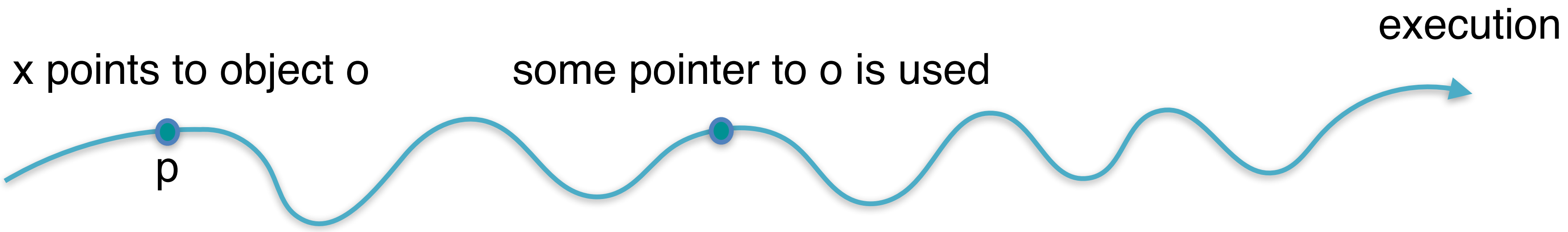
When can we free an object?

```
a = malloc(…) ;
b = a;
free (a); // ?????
*b = c;
```

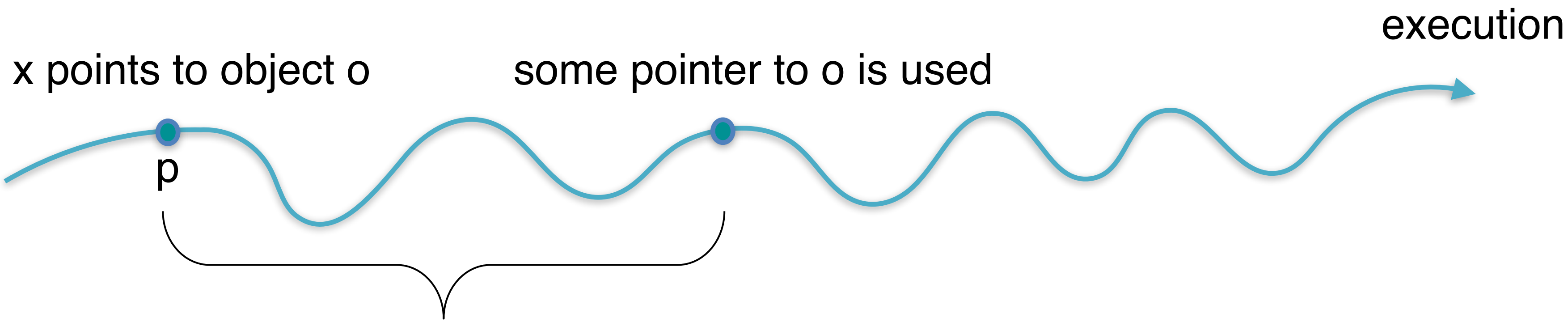
```
a = malloc(…) ;
b = a;
free (a); // ?????
c = malloc (…);
if  (b == c)
        printf("unexpected equality");
```

Cannot free an object if there is a pointer to it with a future use!

When can **free x** be inserted after **p**?



When can **free x** be inserted after **p**?



execution

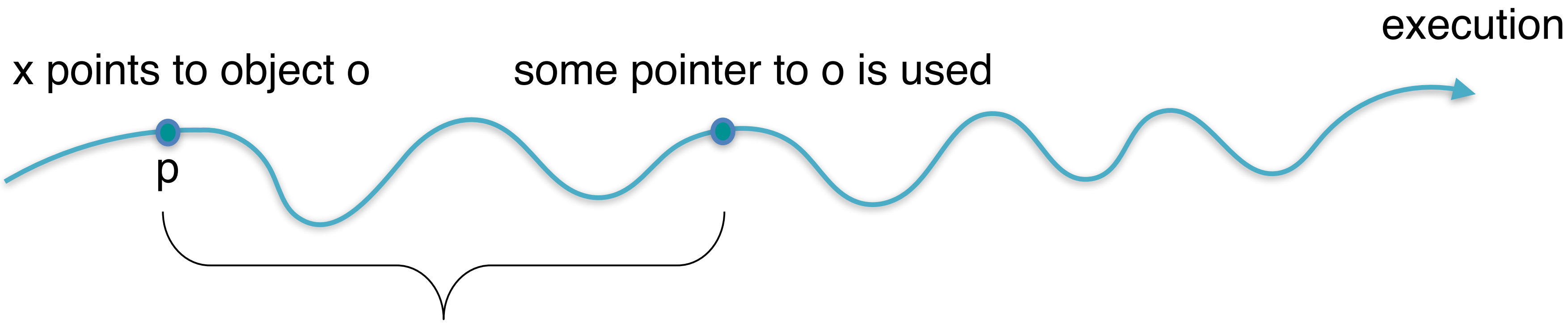
x points to object o

some pointer to o is used

p

cannot free x
because the object it points to is **live**

When can **free x** be inserted after **p**?



cannot free x
because the object it points to is **live**

on **all execution paths** after program point p
there are **no uses** of pointers to the object pointed to by **x**

inserting **free x** after p is safe

Automatic memory management

- Prevalent in object oriented languages and functional languages
- Garbage collection: automatically free memory when it is no longer needed
- Garbage collector (GC) is triggered by allocation

```
New(A) {
  if free_list is empty then gc()
  if free_list is empty then error(“out of memory”);
  pointer := allocate(A);
  return pointer
}
```

Garbage collection

- Approximate reasoning about object liveness
- Use **reachability to approximate liveness**
- Assume reachable objects are live and unreachable objects are dead

Garbage collection: classical techniques

- Reference counting
- Tracing: mark and sweep
- Copying

Reference Counting GC

- Add a reference-count field to every object
  - **o.RC** how many references point to object o
- Newly allocated object o gets **o.RC=1**
- When **o.RC=0** the object o is unreachable
  - unreachable implies dead
  - can be collected (deallocated)

Reference Counting GC

- Add a reference-count field to every object
  - **o.RC** how many references point to object o
- Newly allocated object o gets **o.RC=1**

Why?

- When **o.RC=0** the object o is unreachable
  - unreachable implies dead
  - can be collected (deallocated)

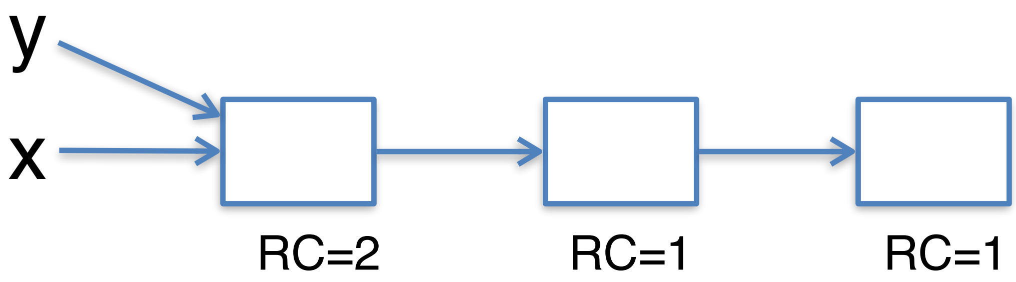
Write-barrier for reference updates

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

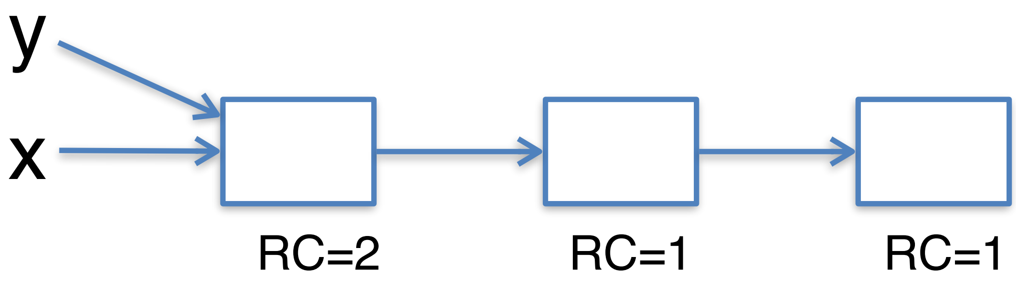
```
update(x,old,new) {
  old.RC--
  new.RC++
  if (old.RC=0) collect(old)
}
```

```
collect(o) {
  free(o)
  for c in Children(o) {
    c.RC--;
    if (c.RC=0) collect(c)
  }
}
```

Example: reference counting

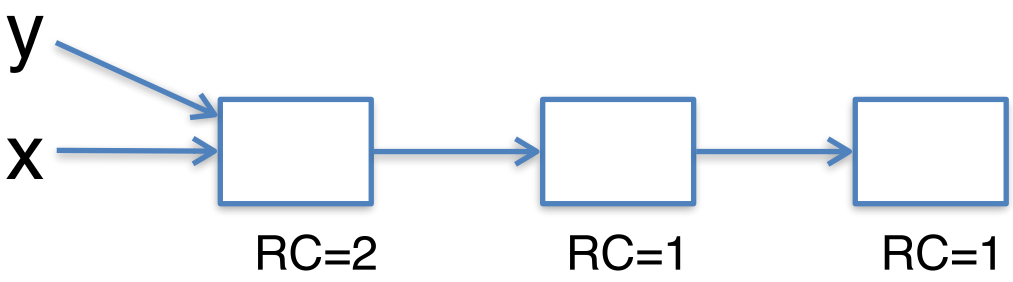


Example: reference counting

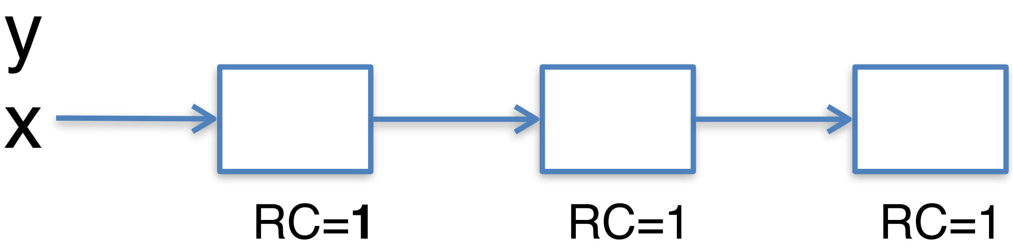


```
y := null
```

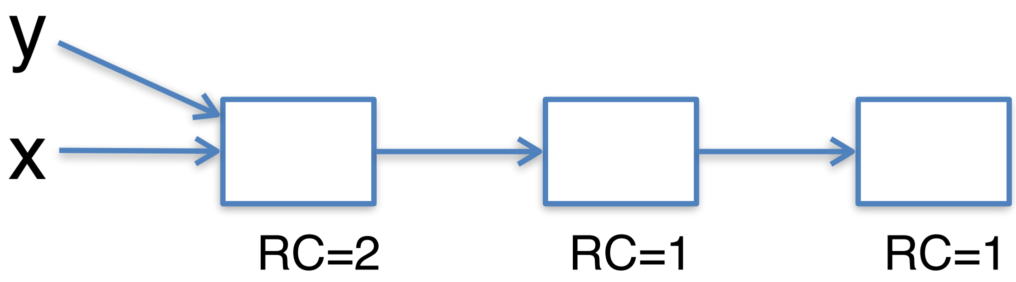
Example: reference counting



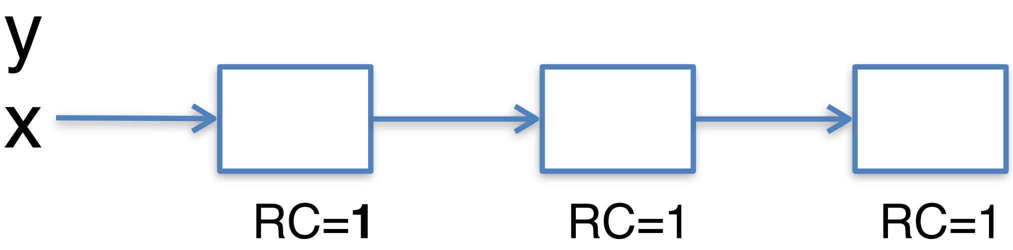
```
y := null
```



Example: reference counting

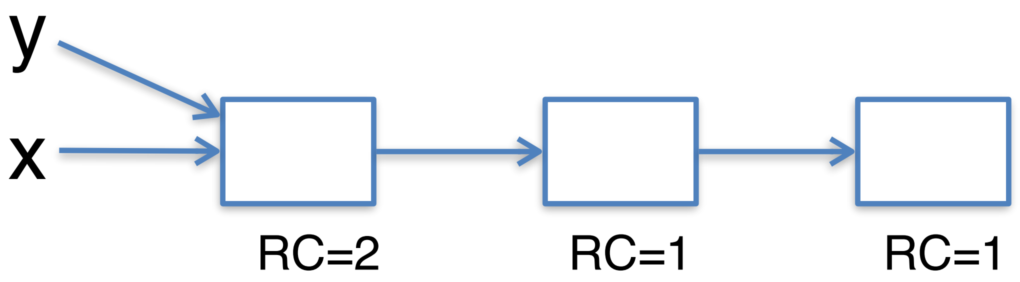


y := null

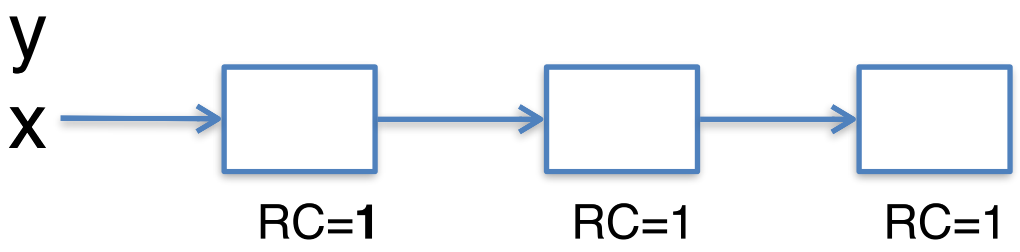


x := null

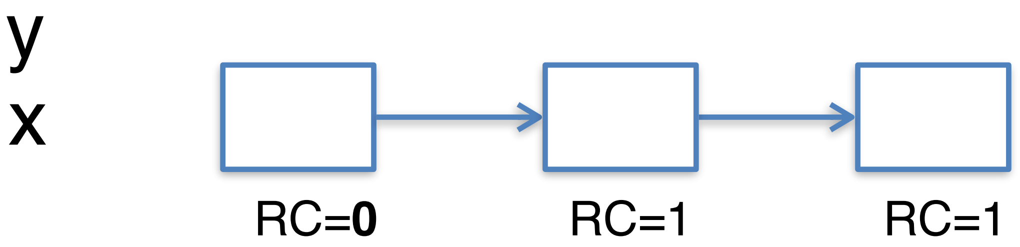
Example: reference counting



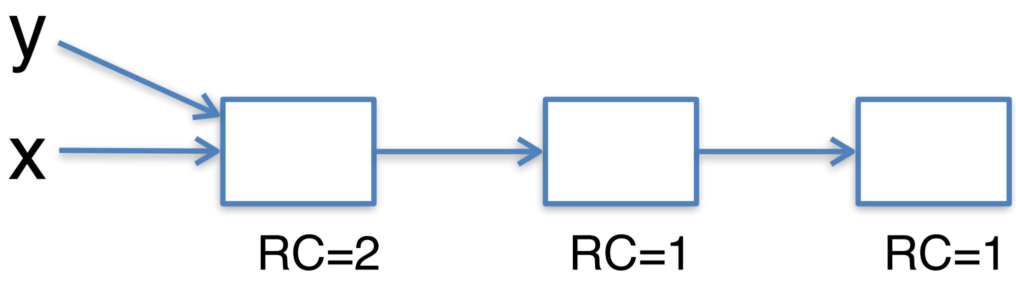
`y := null`



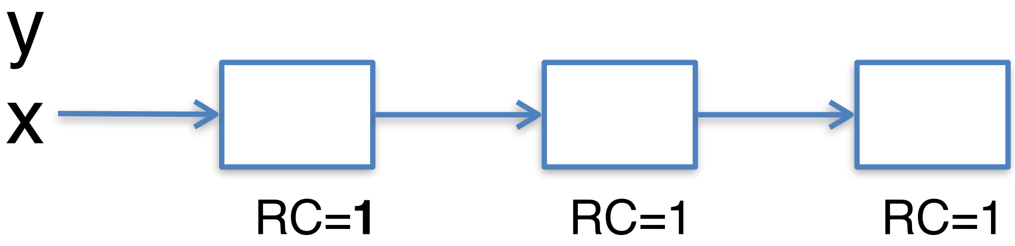
`x := null`



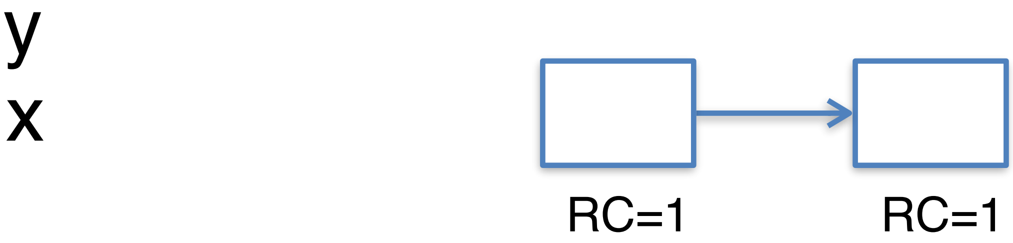
Example: reference counting



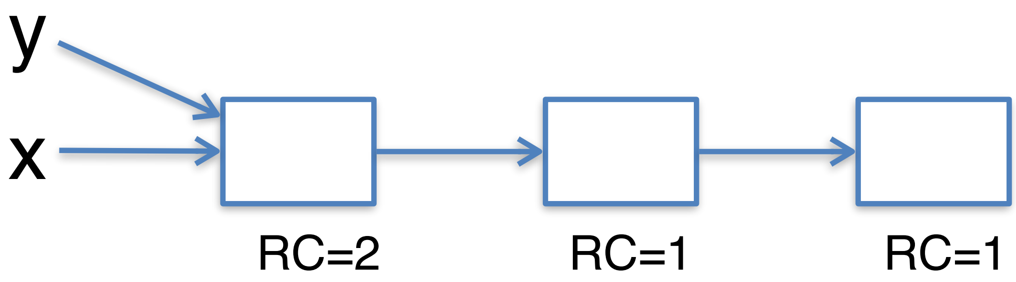
y := null



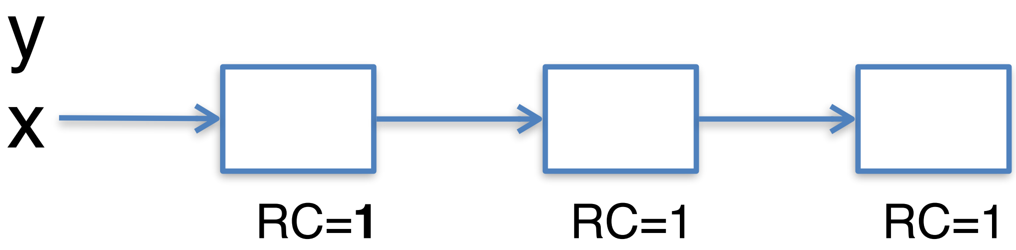
x := null



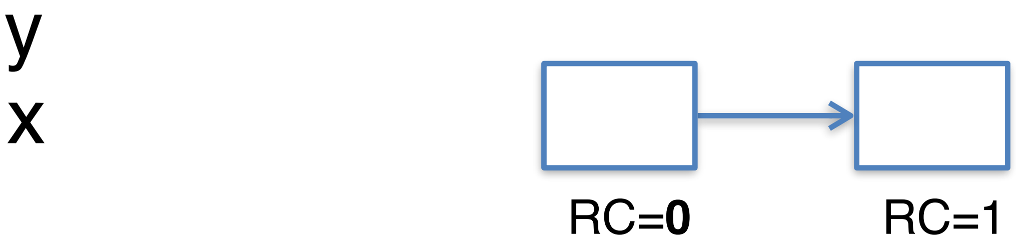
Example: reference counting



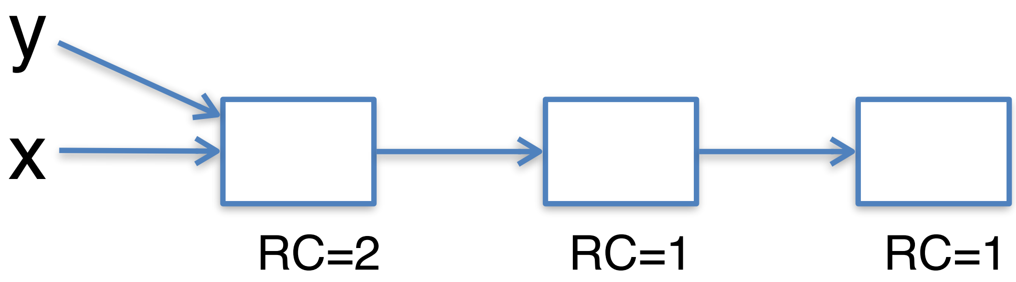
y := null



x := null



Example: reference counting

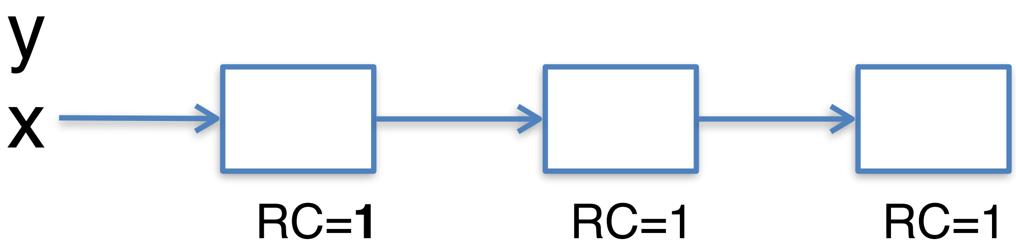


y

x

RC=2 RC=1 RC=1

```
y := null
```

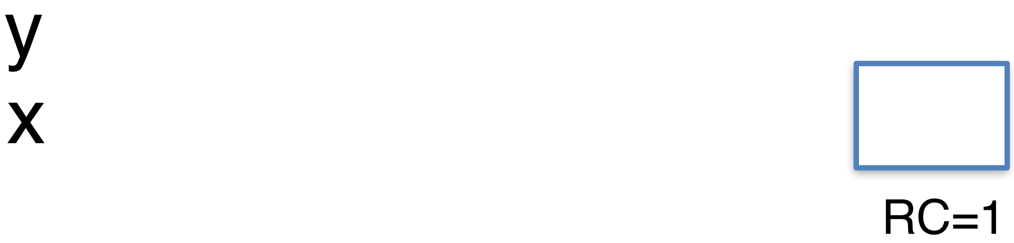


y

x

RC=1 RC=1 RC=1

```
x := null
```

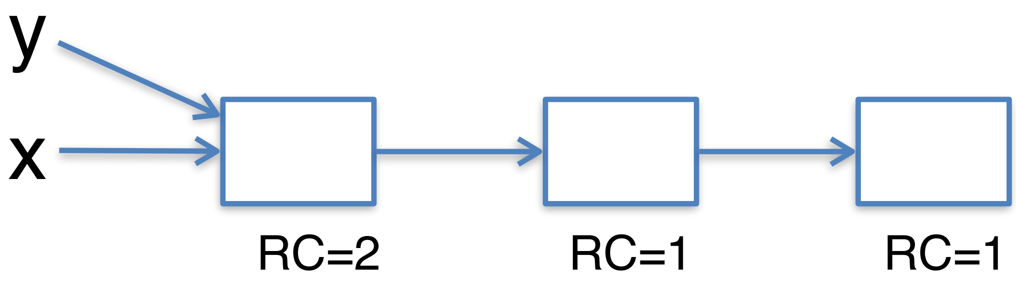


y

x

RC=1

Example: reference counting



y

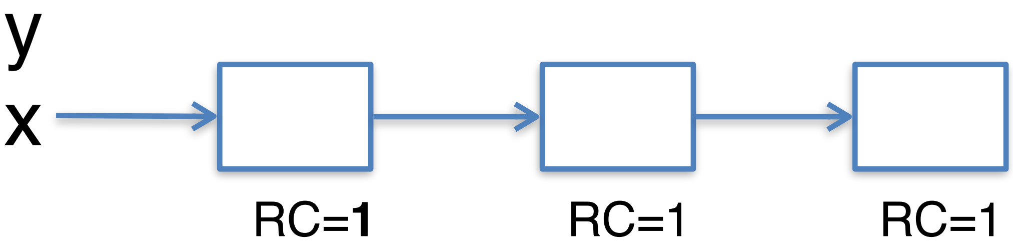
x

RC=2

RC=1

RC=1

y := null



y

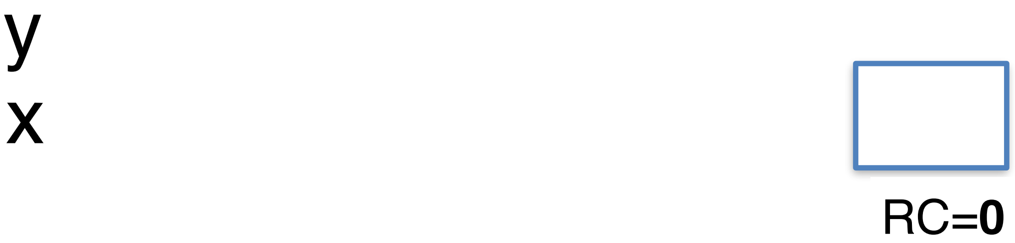
x

RC=1

RC=1

RC=1

x := null

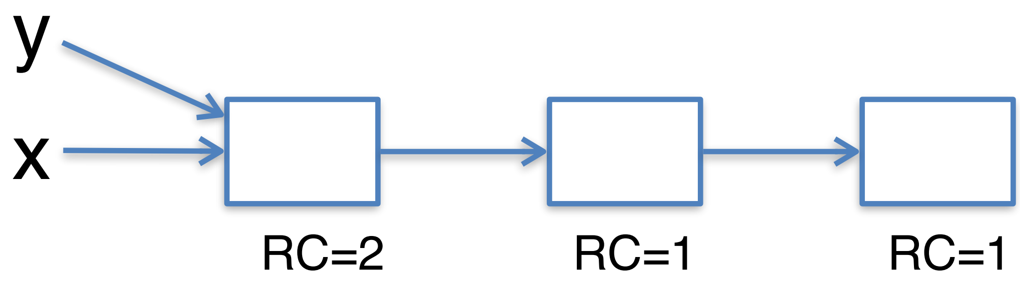


y

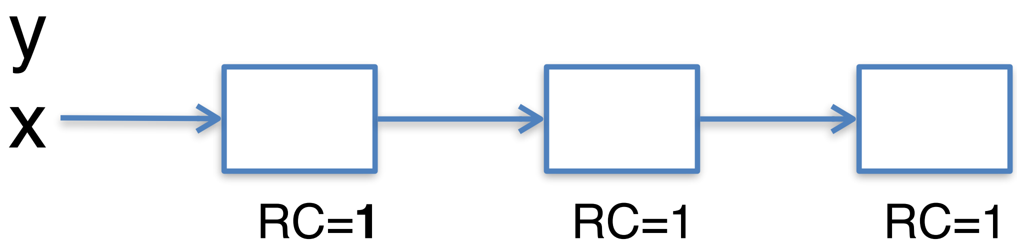
x

RC=0

Example: reference counting



```
y := null
```

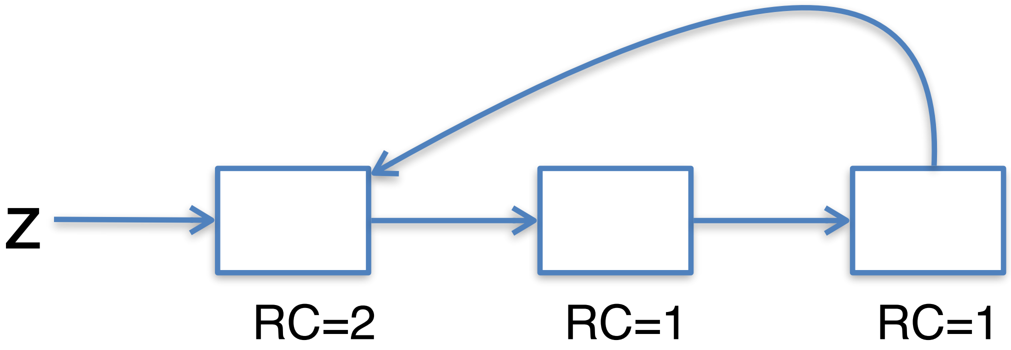


```
x := null
```

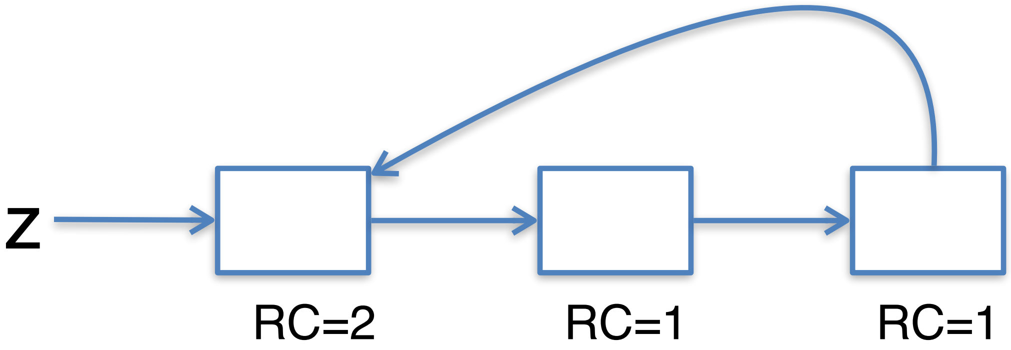
y

x

Example: reference counting

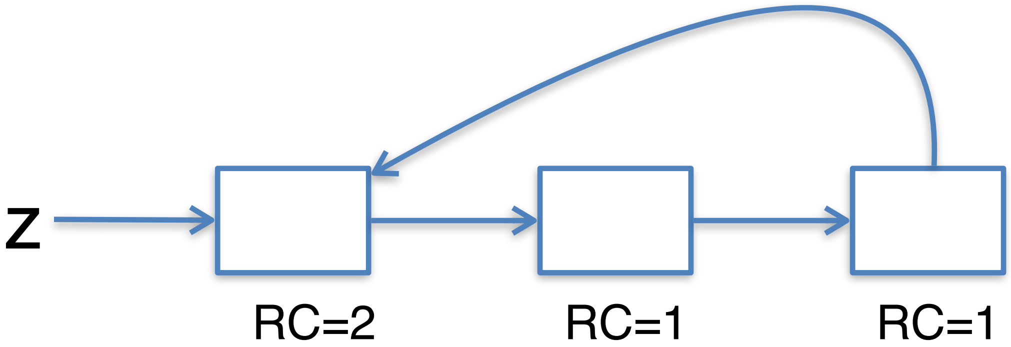


Example: reference counting

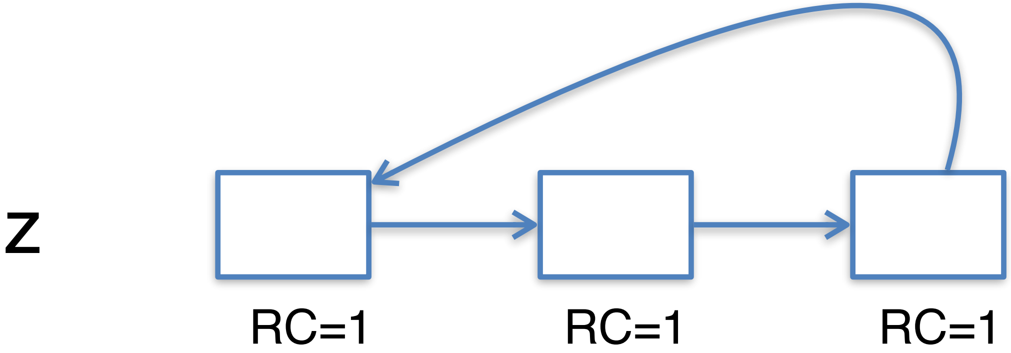


```
z := null
```

Example: reference counting



z := null



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

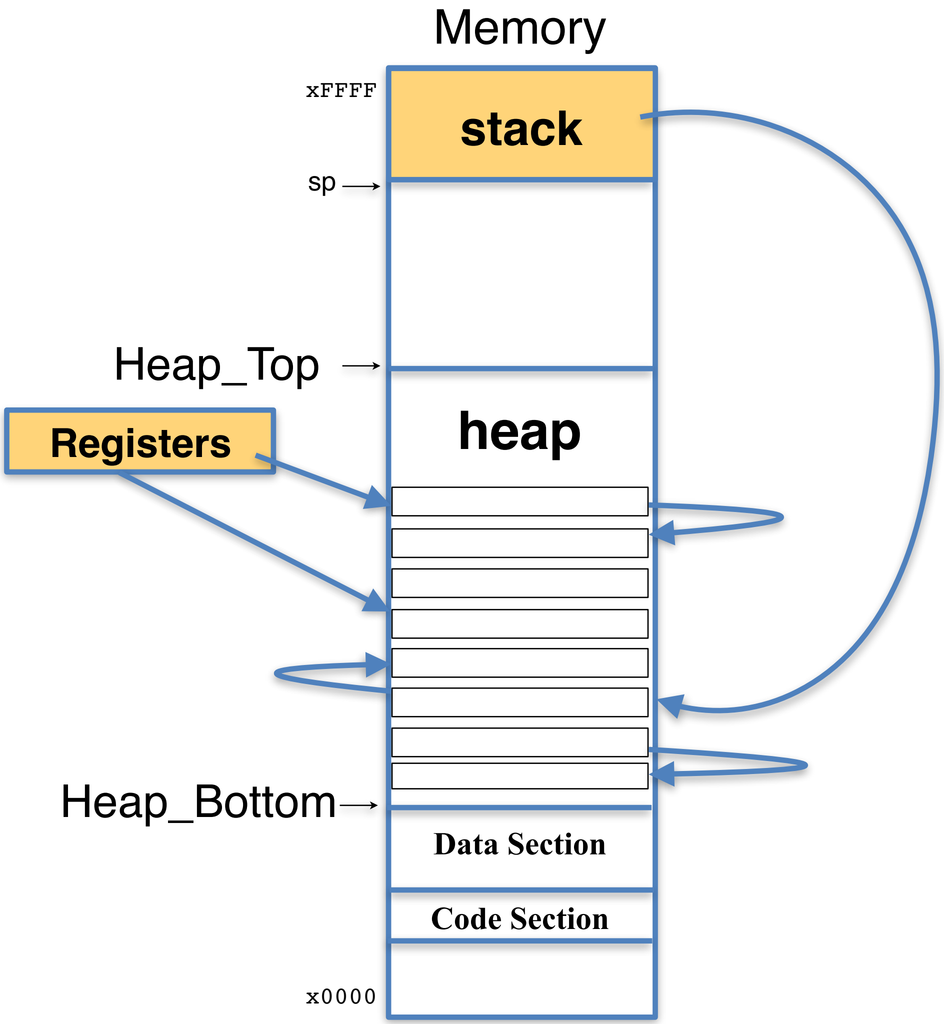
Mark and Sweep GC

[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

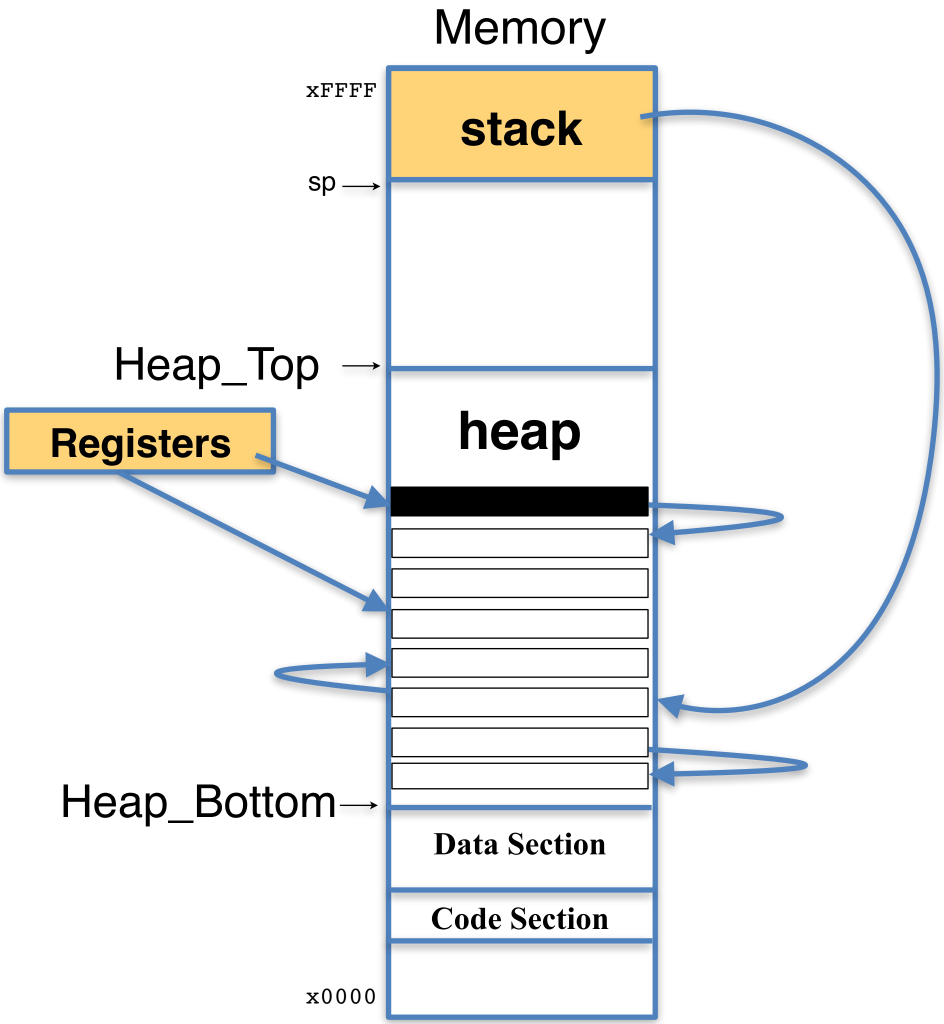
Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



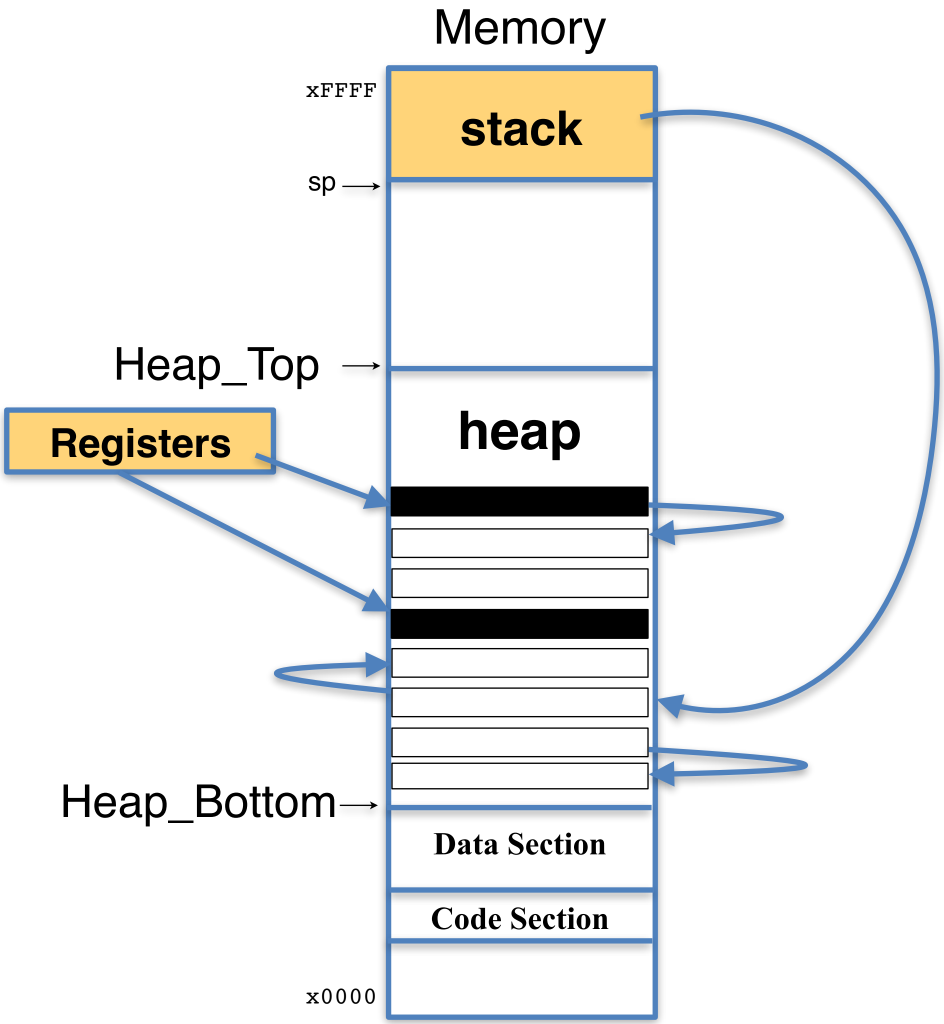
Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



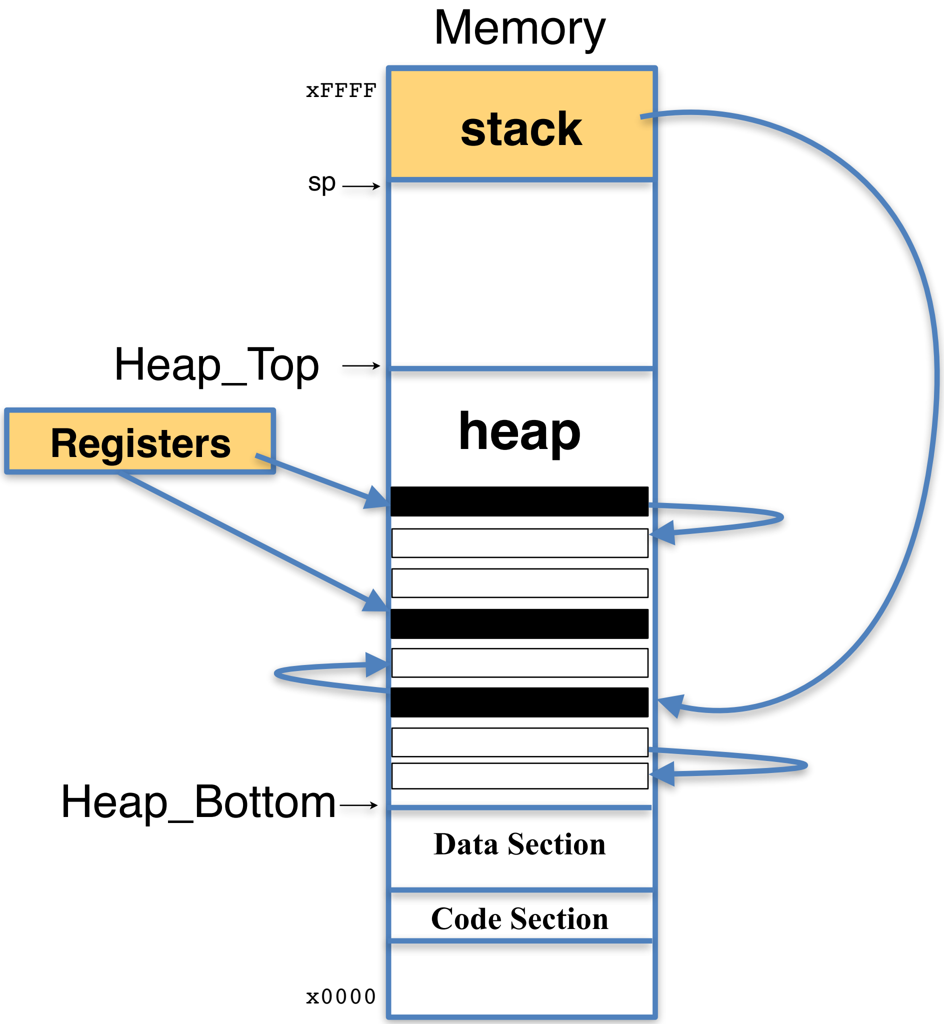
Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



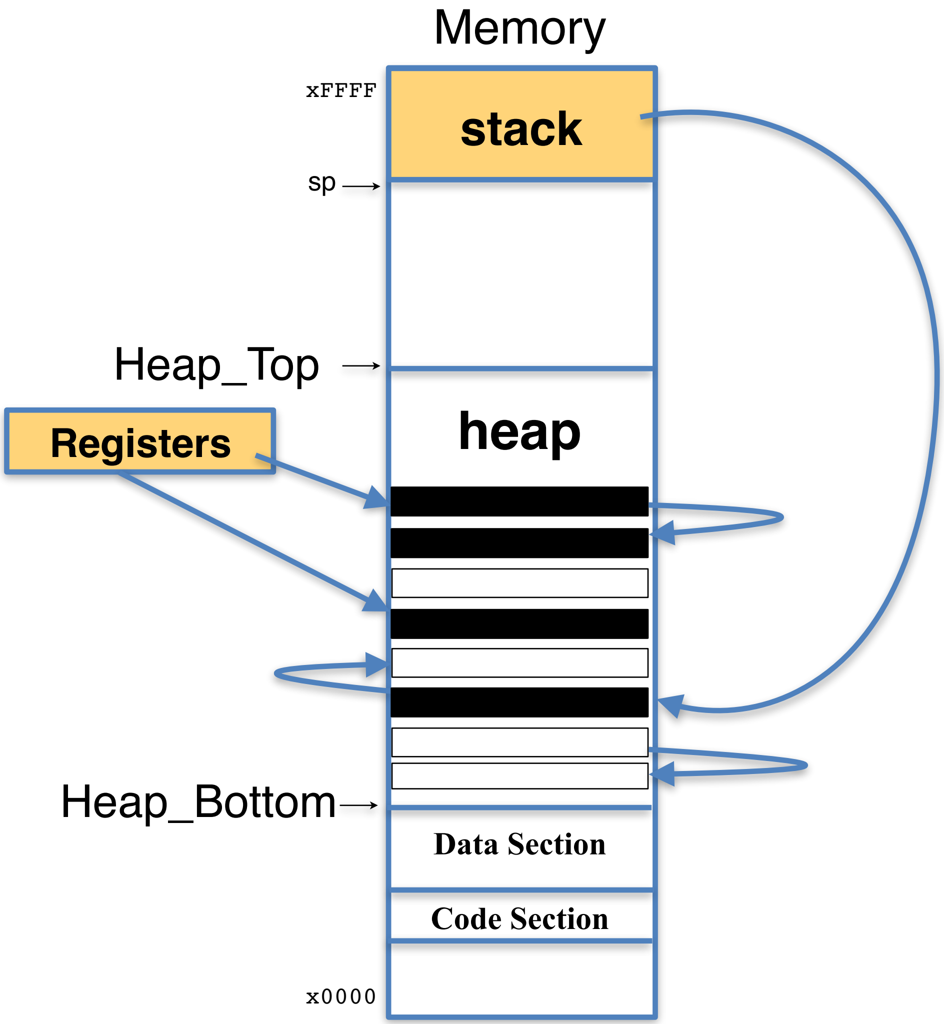
Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



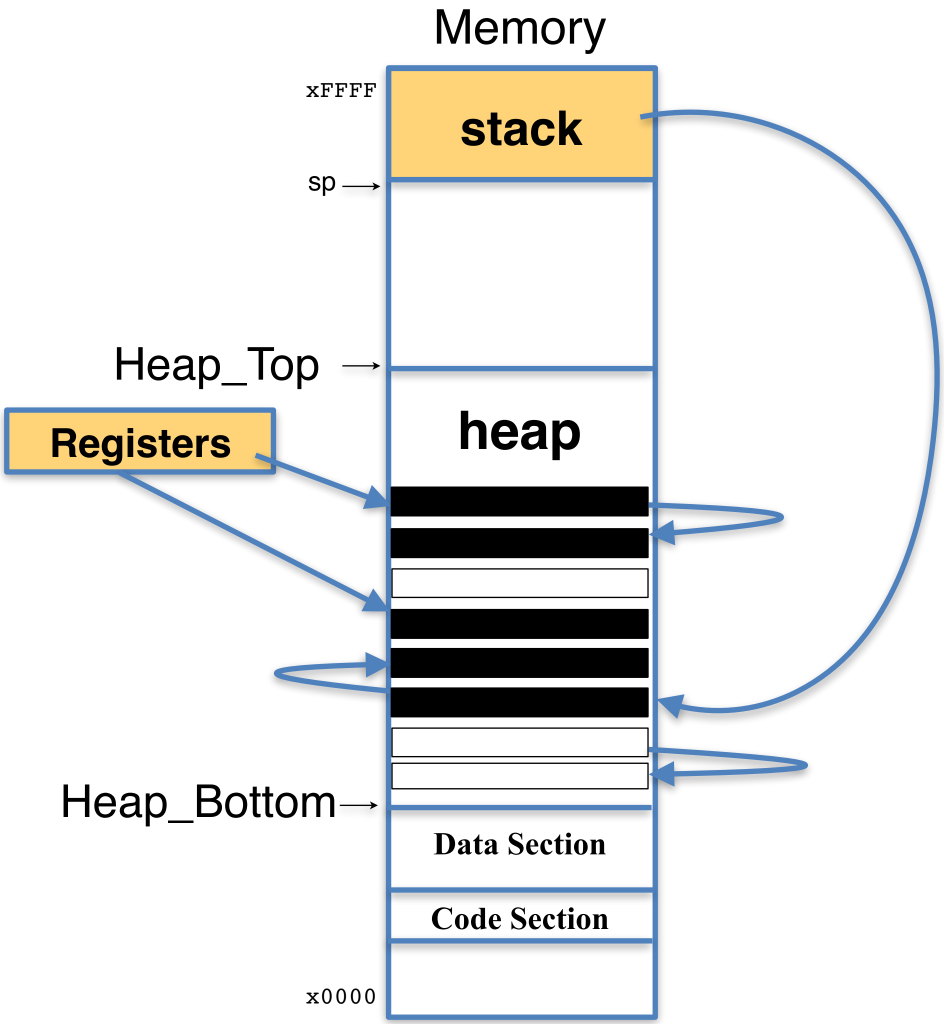
Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



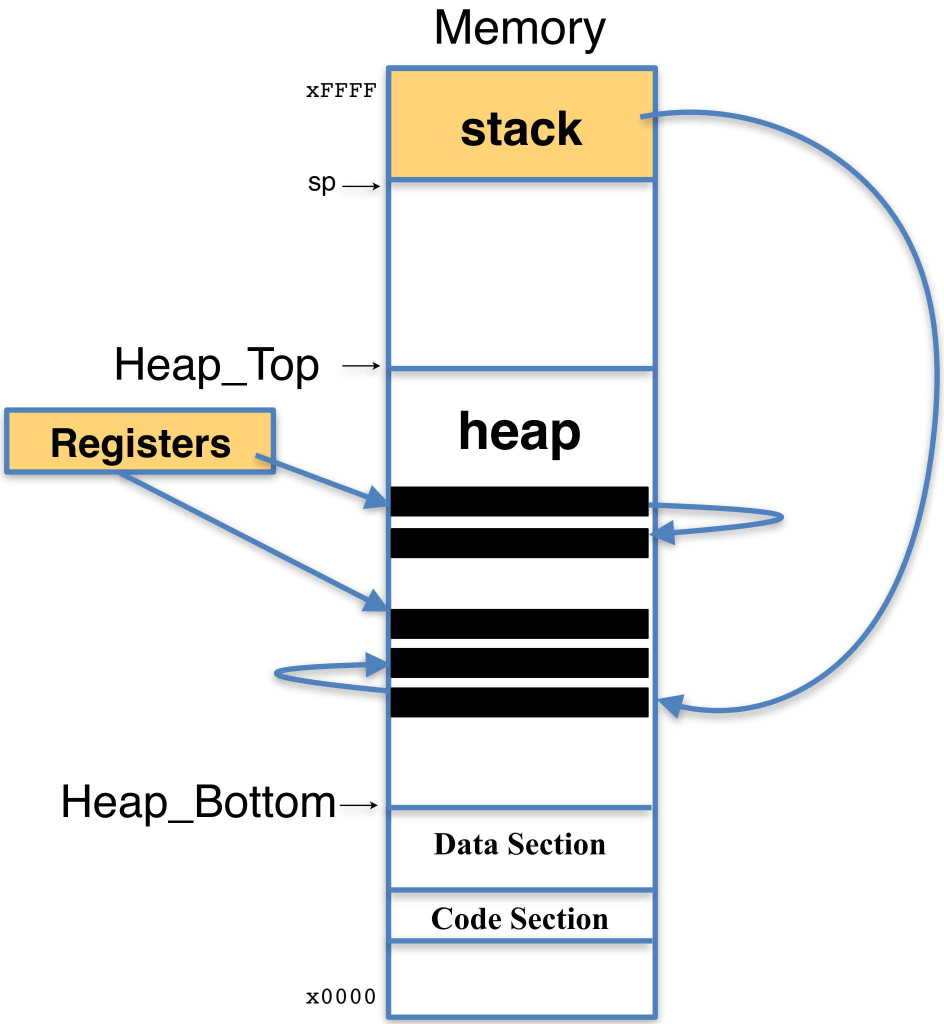
Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



Mark and Sweep GC

- **Roots**: pointers in **registers** and on **stack**
- Traverse live objects and mark black
- Reclaim white objects



Mark and Sweep GC

```
mark_sweep_gc () {
  for p in Roots
    mark(p)
  sweep()
}
```

Mark and Sweep GC

```
mark (o) {
 if (mark_bit(o) = unmarked) {
     mark_bit(o) :=marked
     for c in Children(o)
         mark(c)
  }
}
```

```
mark_sweep_gc () {
 for p in Roots
   mark(p)
 sweep()
}
```

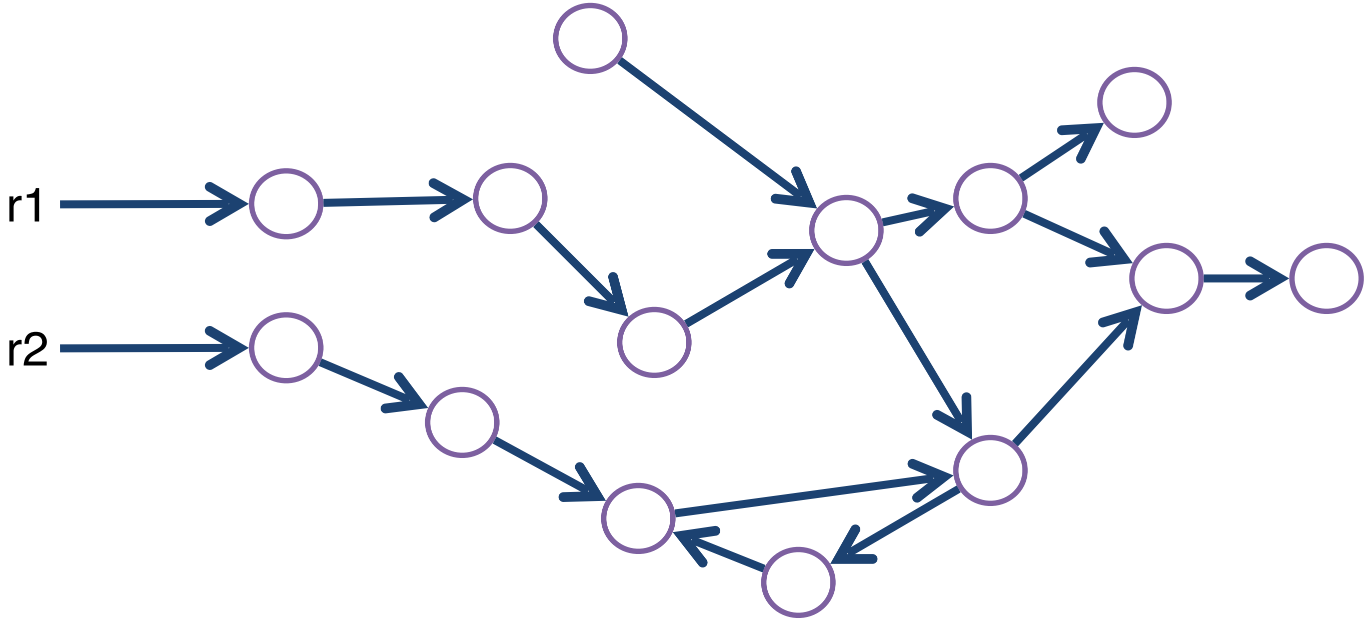
Mark and Sweep GC

```
mark (o) {
  if (mark_bit(o) = unmarked) {
      mark_bit(o) :=marked
      for c in Children(o)
          mark(c)
   }
}
```

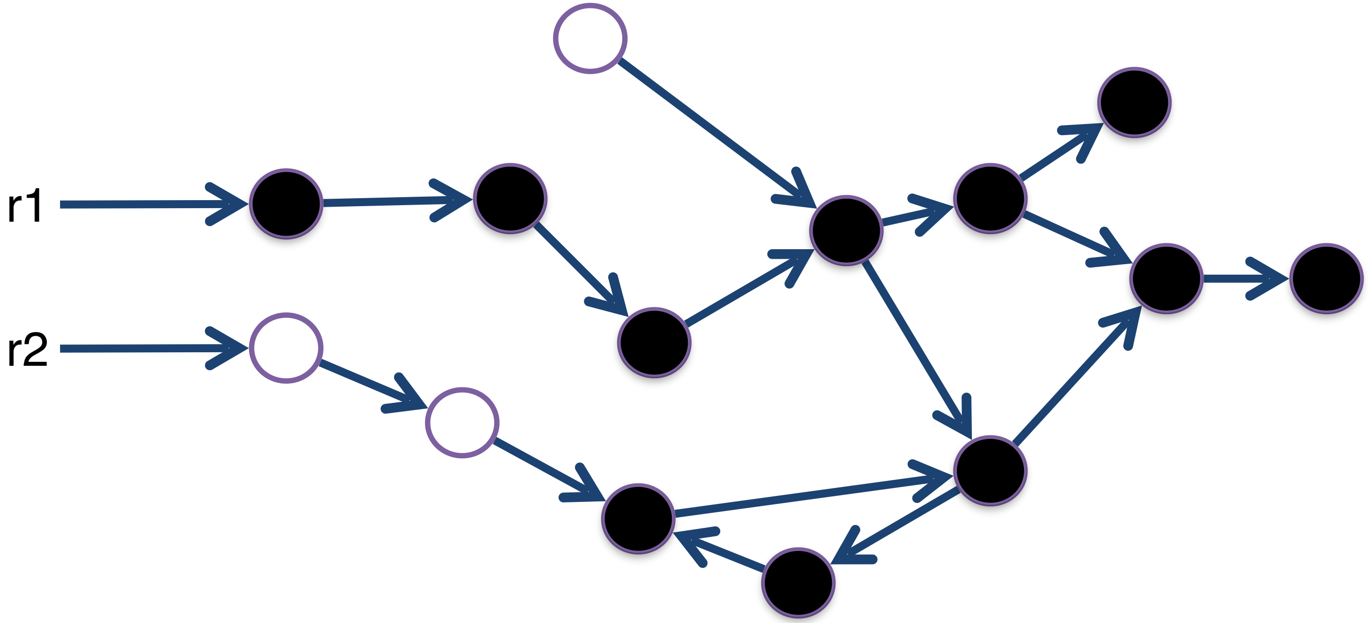
```
mark_sweep_gc () {
  for p in Roots
    mark(p)
  sweep()
}
```

```
sweep() {
  p := Heap_bottom
  while (p < Heap_top) {
           if (mark_bit(p) = unmarked) free(p)
           else mark_bit(p) := unmarked;
           p:= p + size(p)
   }
}
```

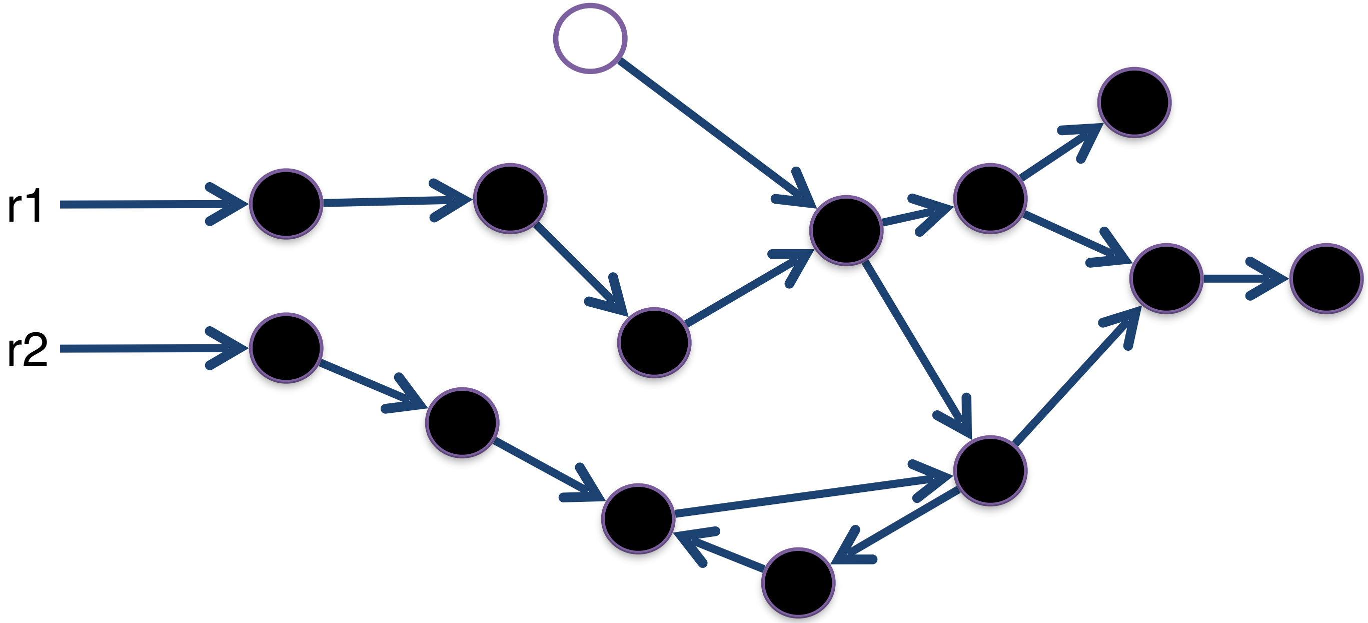
Example: Mark and Sweep



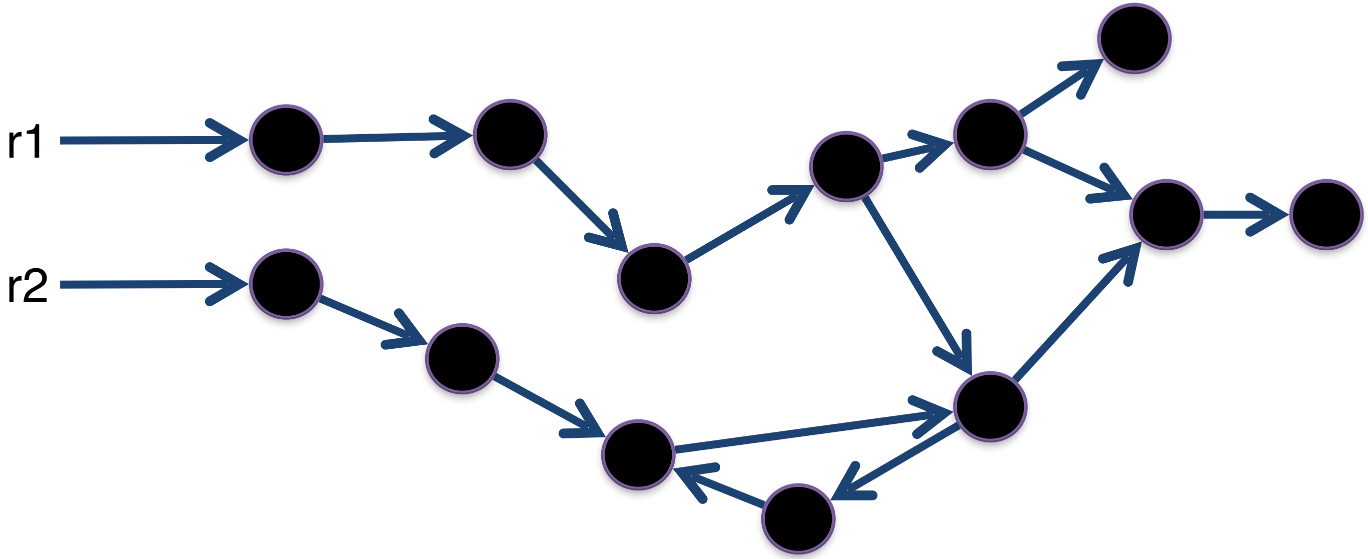
Example: Mark and Sweep



Example: Mark and Sweep



Example: Mark and Sweep



Properties of Mark Phase

```
mark (o) {
  if (mark_bit(o) = unmarked) {
      mark_bit(o) :=marked
      for c in Children(o)
          mark(c)
   }
}
```

- How much memory overhead per object?

Properties of Mark Phase

```
mark (o) {
  if (mark_bit(o) = unmarked) {
      mark_bit(o) :=marked
      for c in Children(o)
          mark(c)
   }
}
```

- How much memory overhead per object?
- Recursion depth?

Properties of Mark Phase

```
mark (o) {
  if (mark_bit(o) = unmarked) {
      mark_bit(o) :=marked
      for c in Children(o)
          mark(c)
   }
}
```

- How much memory overhead per object?
- Recursion depth?
- Can we traverse the heap without worst-case O(n) stack?

Properties of Mark Phase

```
mark (o) {
 if (mark_bit(o) = unmarked) {
     mark_bit(o) :=marked
     for c in Children(o)
         mark(c)
  }
}
```

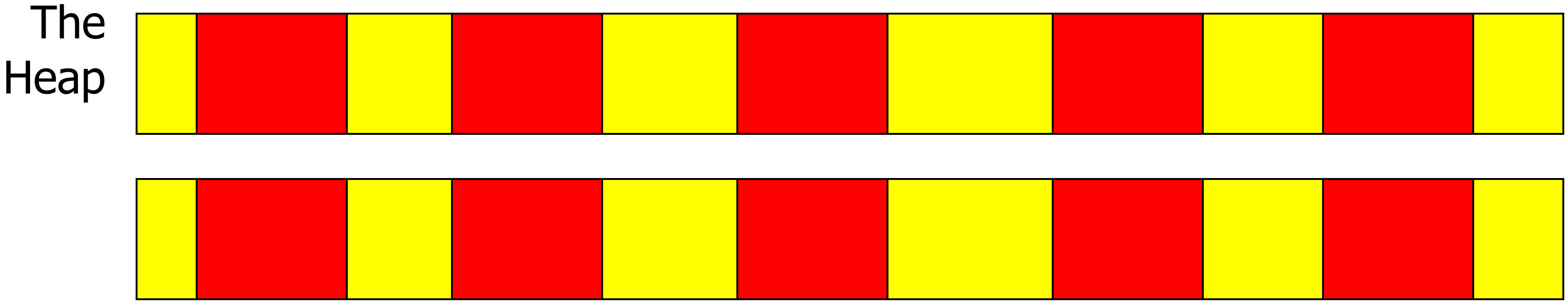
- How much memory overhead per object?
- Recursion depth?
- Can we traverse the heap without worst-case O(n) stack?
- Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)

Properties of Mark and Sweep GC

- Most popular method today
- Simple
- Does not move objects, so heap may **fragment**
- Complexity
  - mark phase: live objects
  - sweep phase: heap size
- Termination: each pointer traversed once
- Engineering tricks used to improve performance

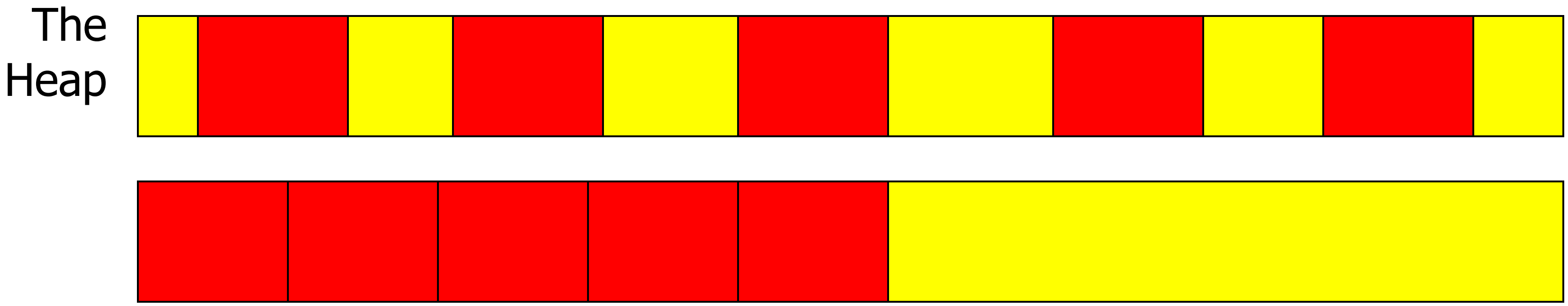
Mark-Compact

- At runtime, objects are allocated and reclaimed
- Gradually, the **heap** gets **fragmented**
- When space becomes too fragmented to allocate, run **compaction algorithm**
    - **move all live objects to the beginning of the heap**
    - **update all pointers to reference the new locations**



Mark-Compact

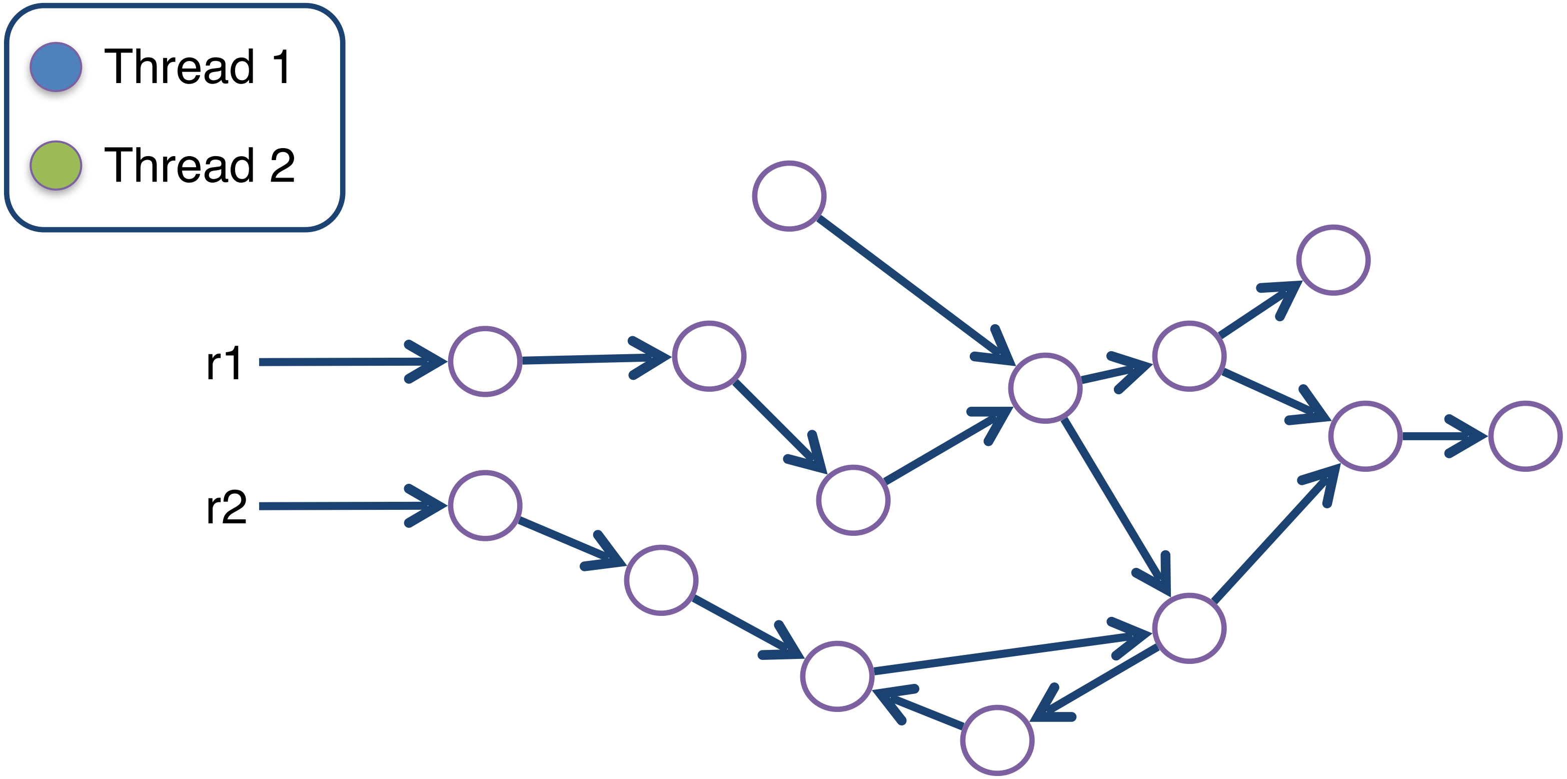
- At runtime, objects are allocated and reclaimed
- Gradually, the **heap** gets **fragmented**
- When space becomes too fragmented to allocate, run **compaction algorithm**
  - **move all live objects to the beginning of the heap**
  - **update all pointers to reference the new locations**



Mark-Compact

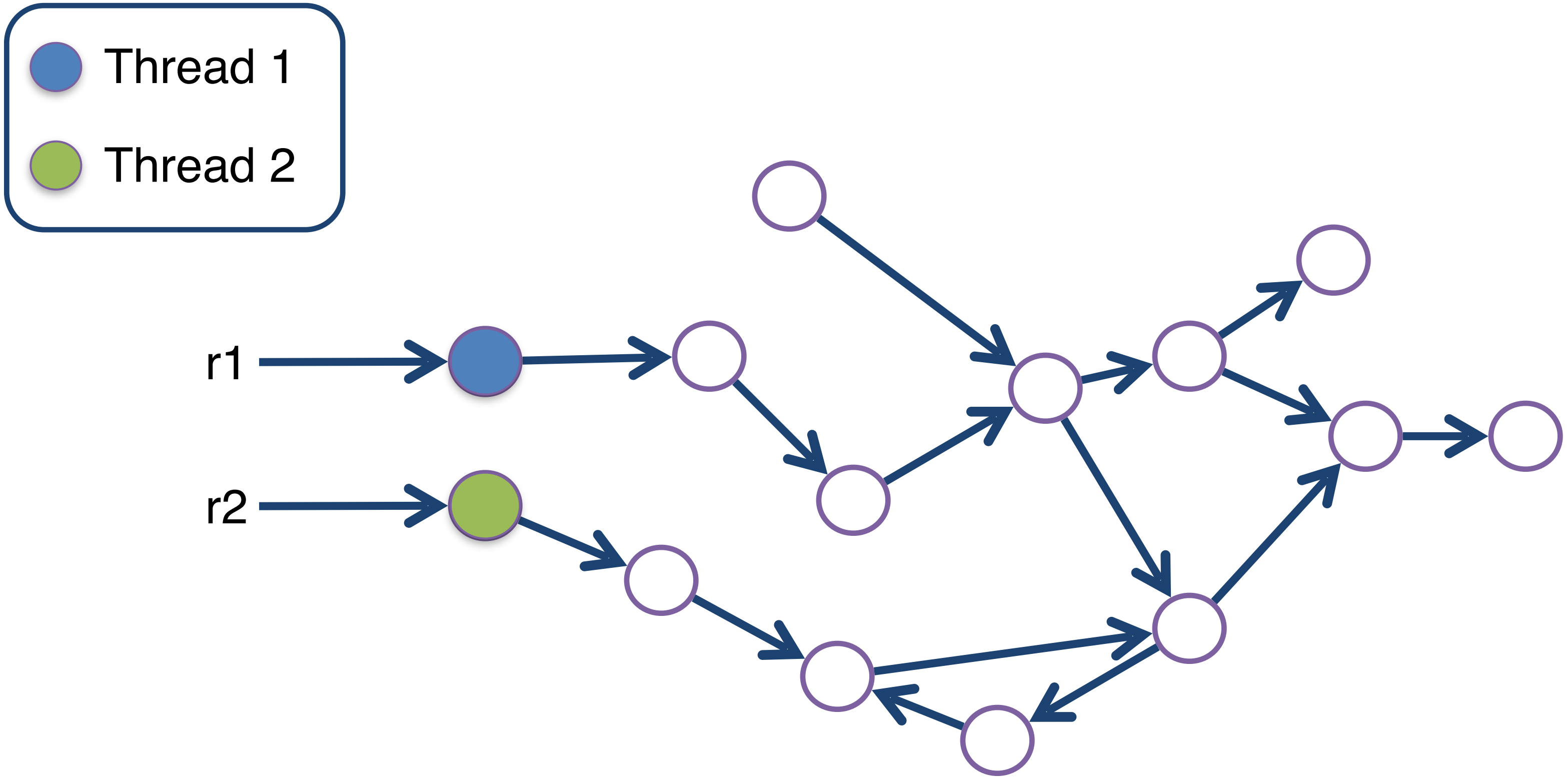
- Compaction is very costly and we attempt to run it infrequently, or only partially
- Important parameters of compaction algorithm
  - keep order of objects?
  - use extra space for compactor data structures?
  - how many heap passes?
  - preserve alignment?
  - can it run in parallel on a multi-processor?

Parallel Mark and Sweep GC



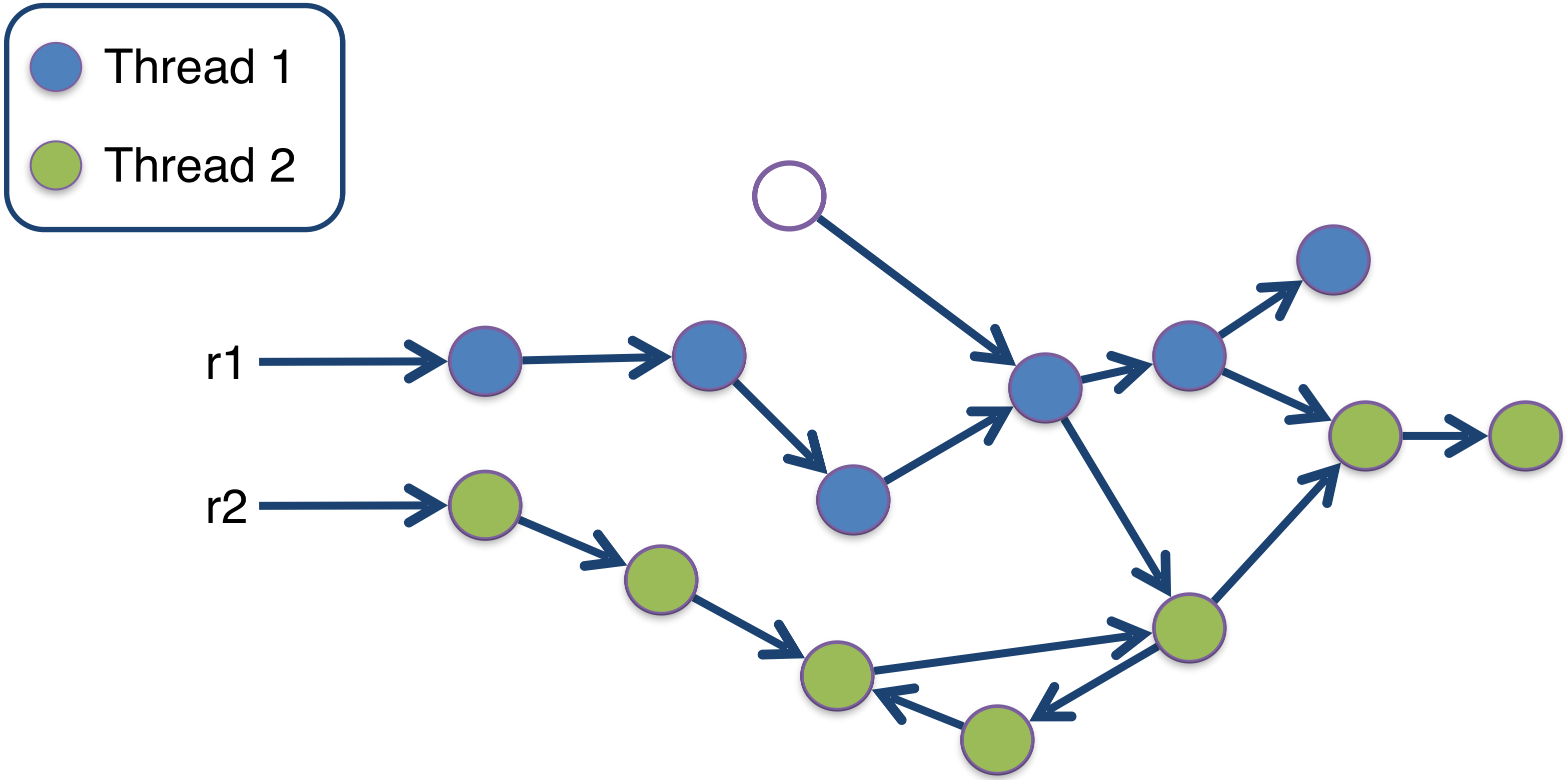
Parallel GC: mutator is stopped, collector threads run in parallel

Parallel Mark and Sweep GC



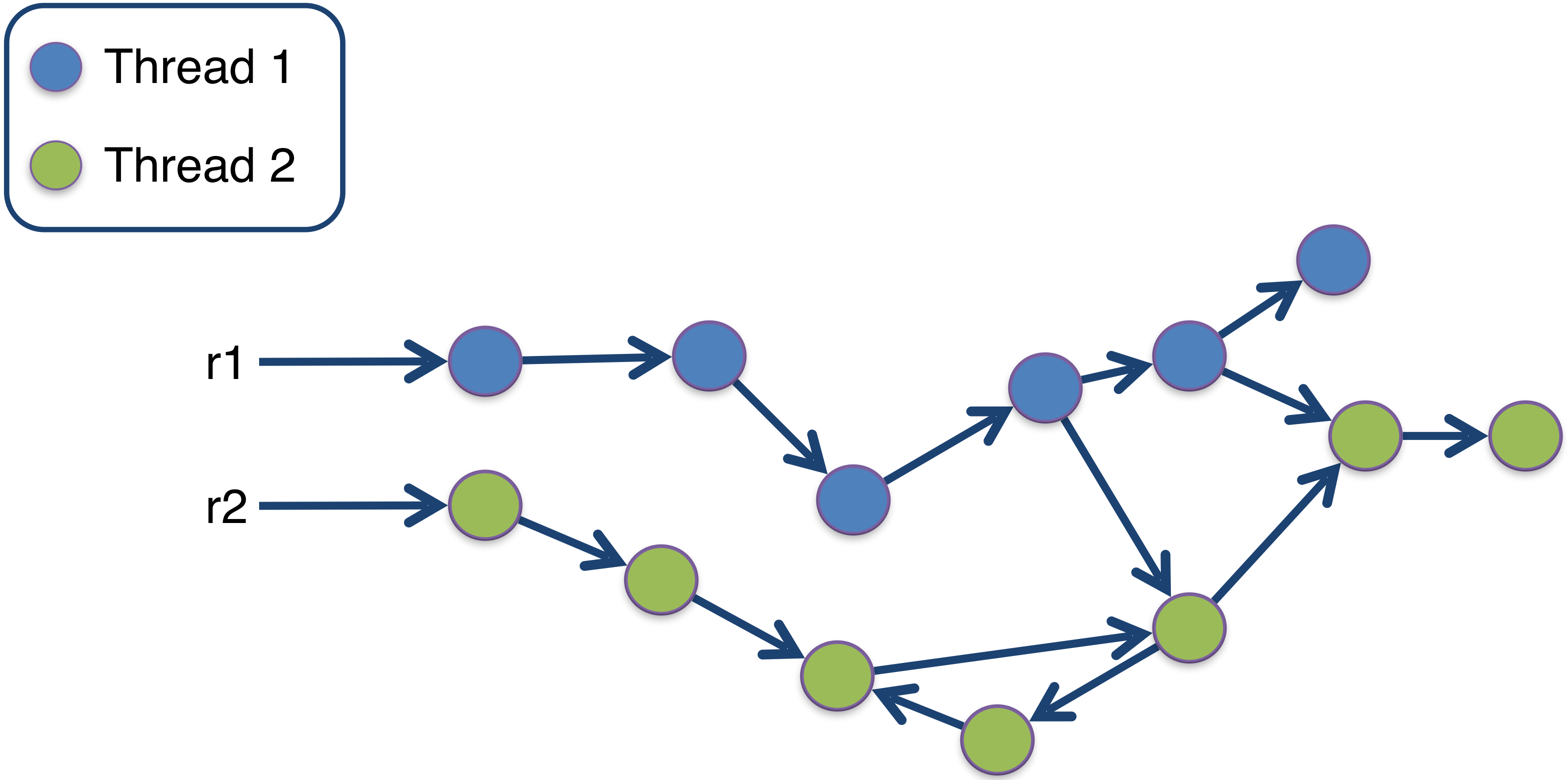
Parallel GC: mutator is stopped, collector threads run in parallel

Parallel Mark and Sweep GC



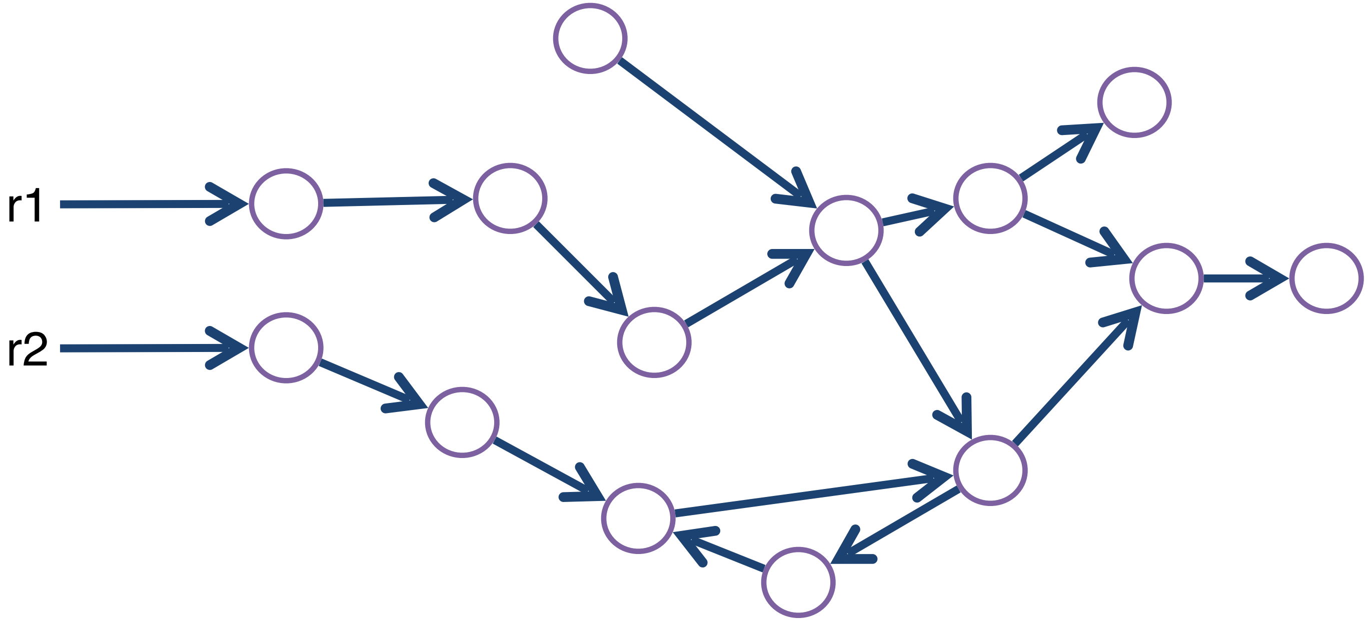
Parallel GC: mutator is stopped, collector threads run in parallel

Parallel Mark and Sweep GC



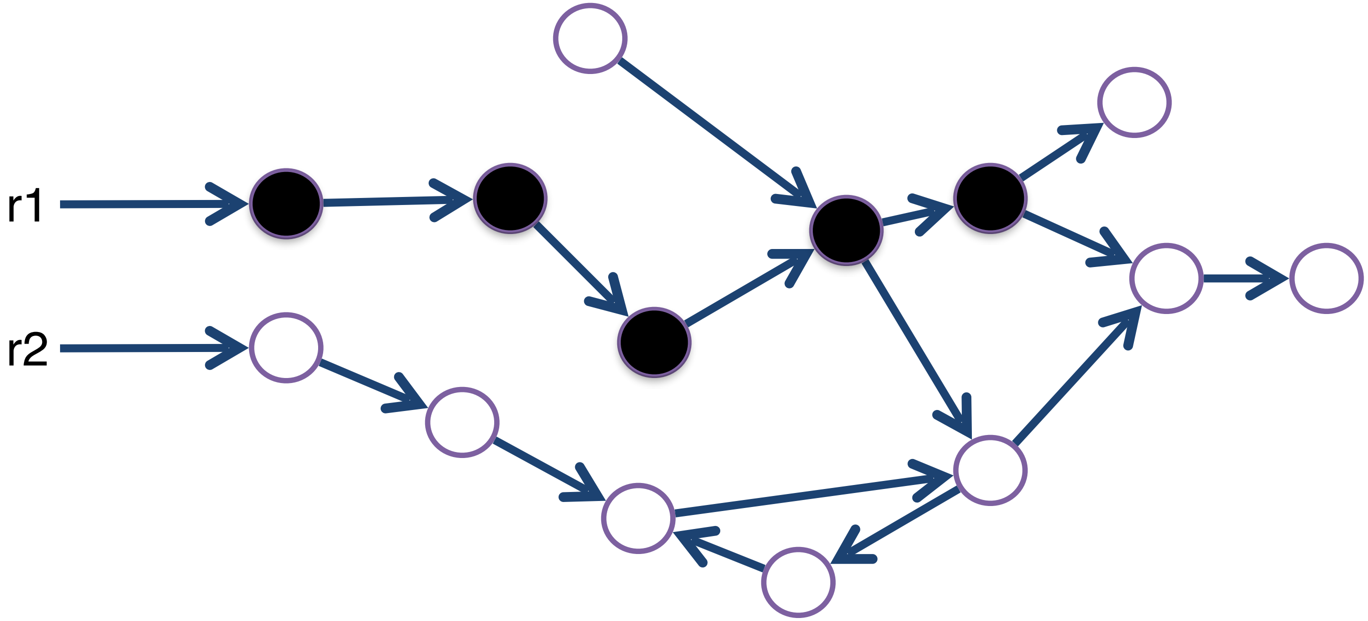
Parallel GC: mutator is stopped, collector threads run in parallel

Concurrent Mark and Sweep GC



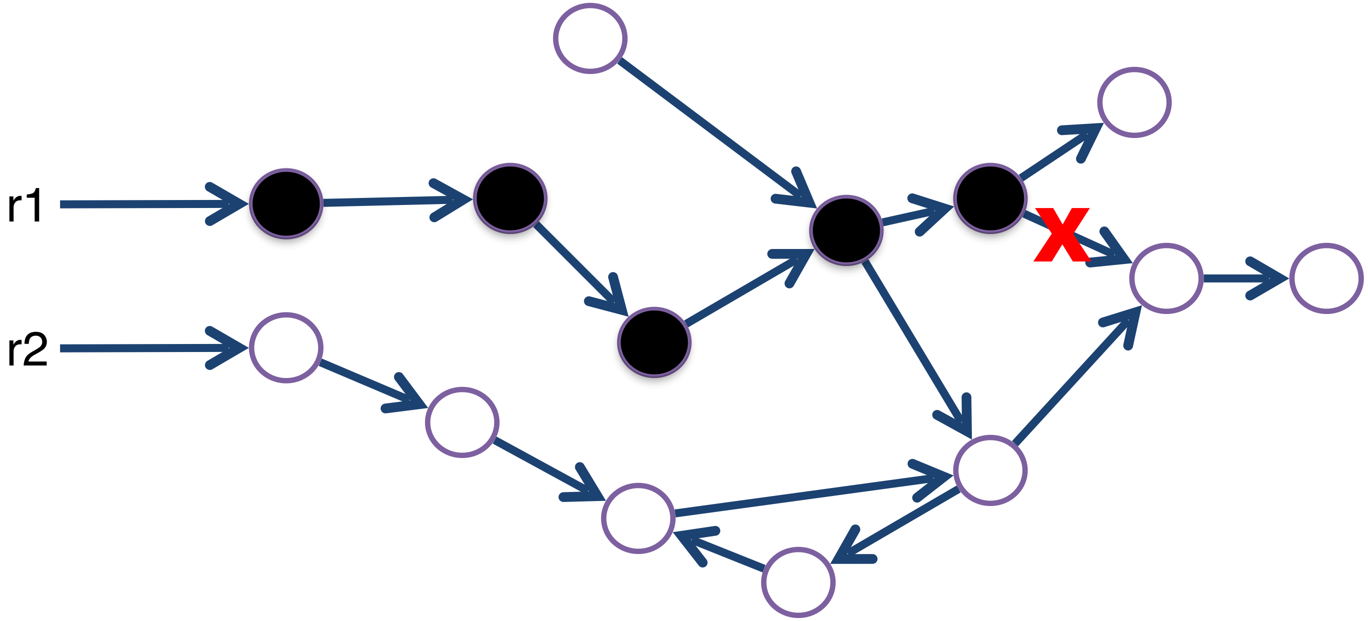
Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



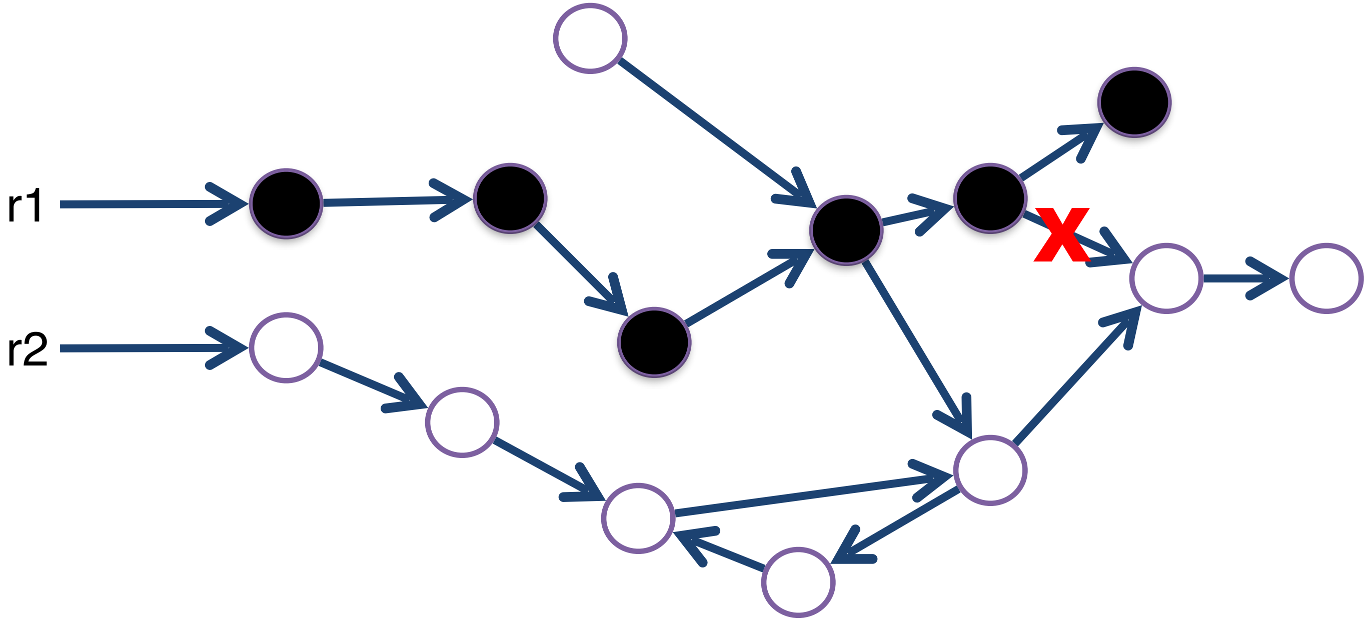
Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



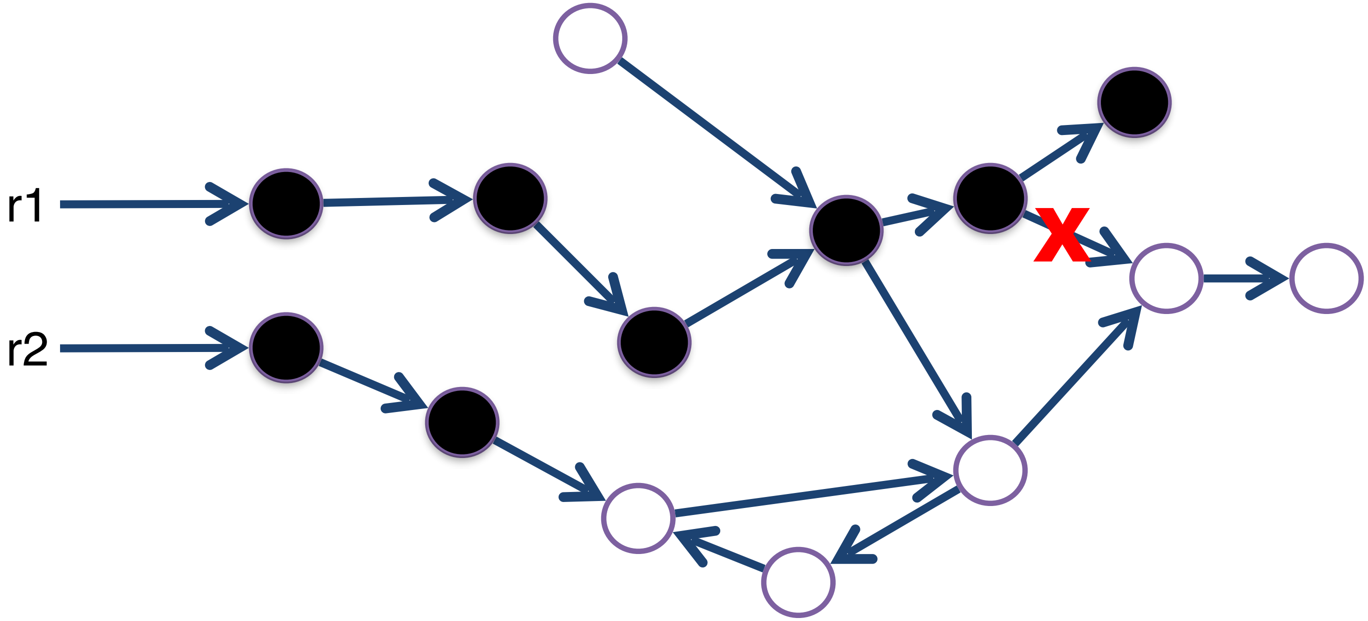
Concurrent GC: mutator and collector threads run in parallel
no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



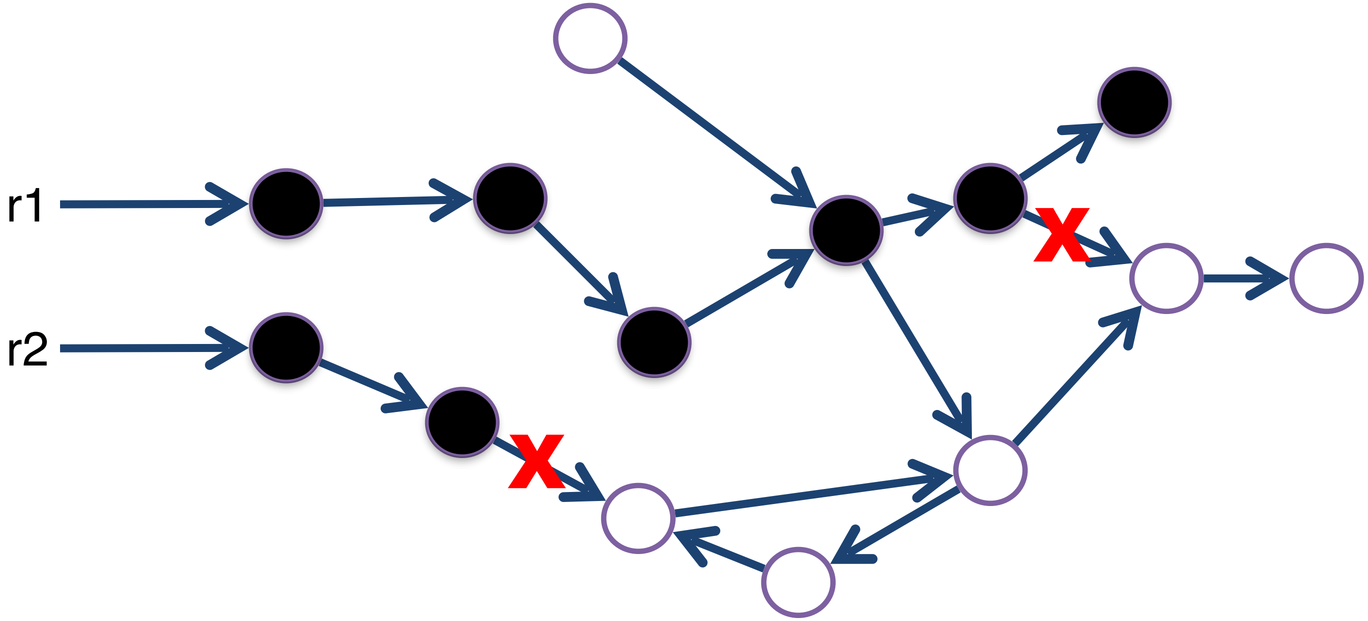
Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



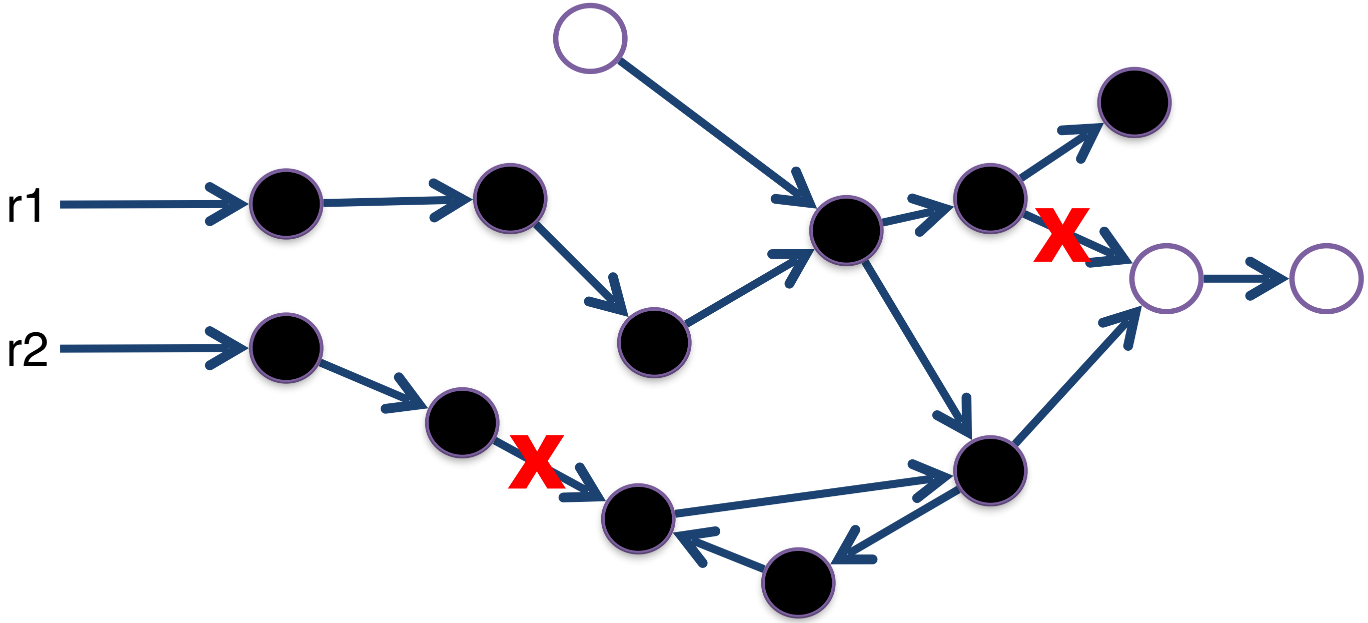
Concurrent GC: mutator and collector threads run in parallel
no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



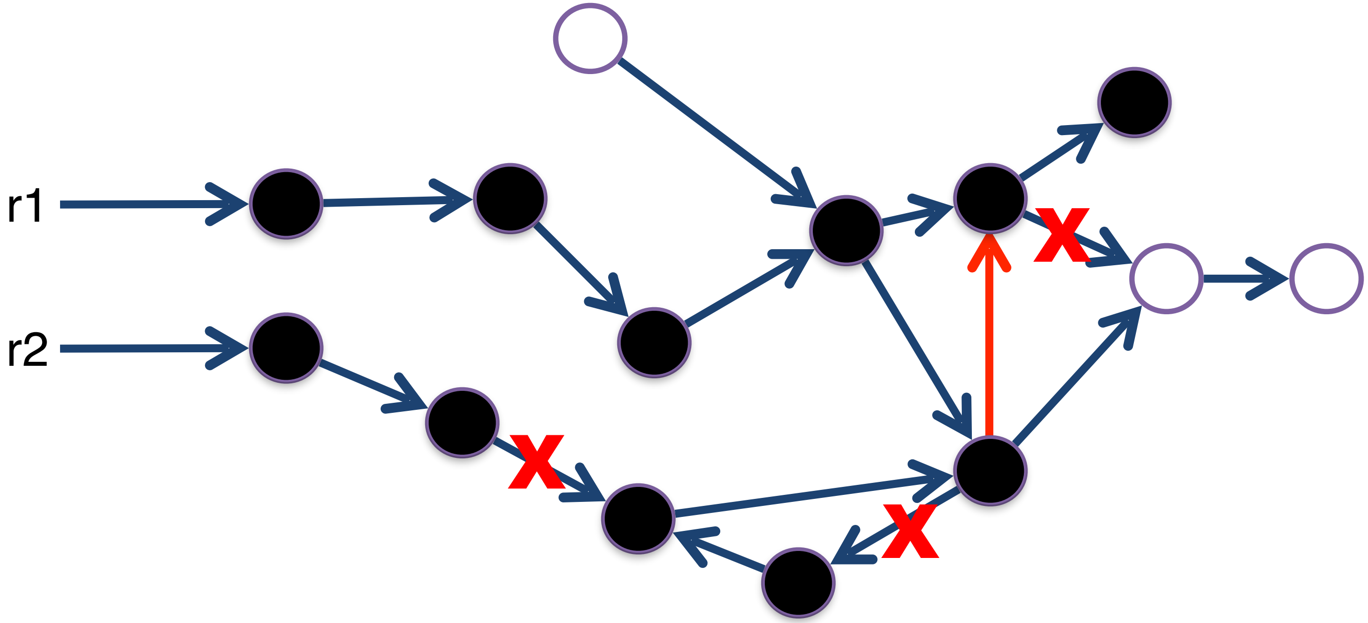
Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



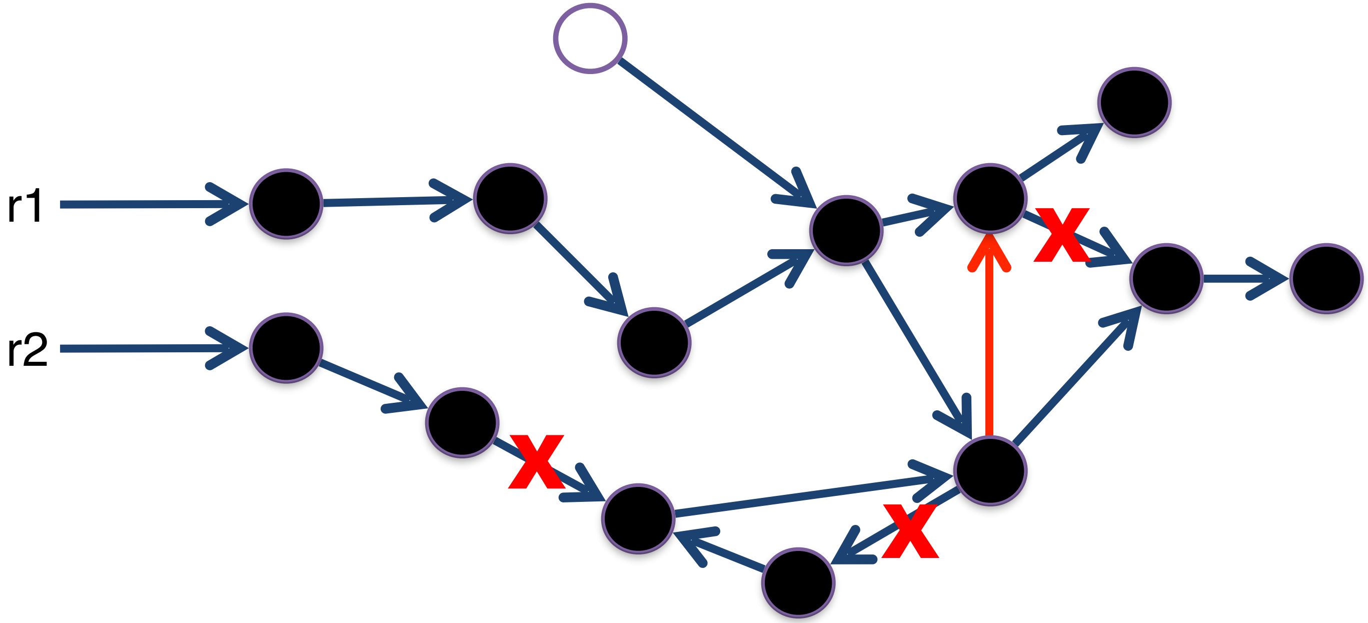
Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



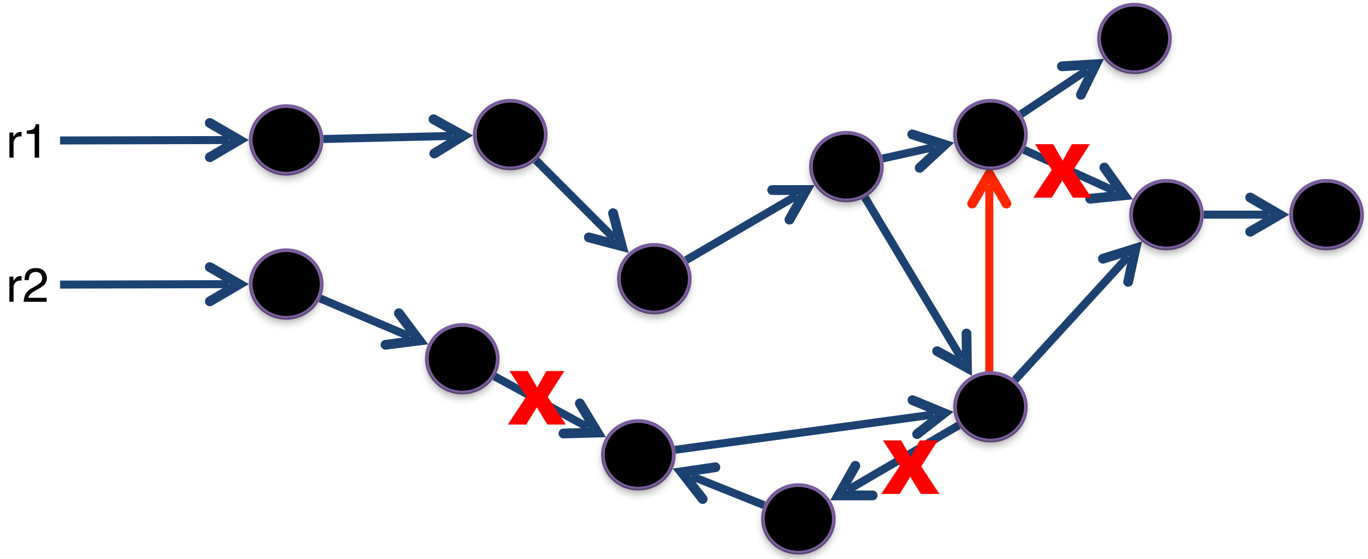
Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



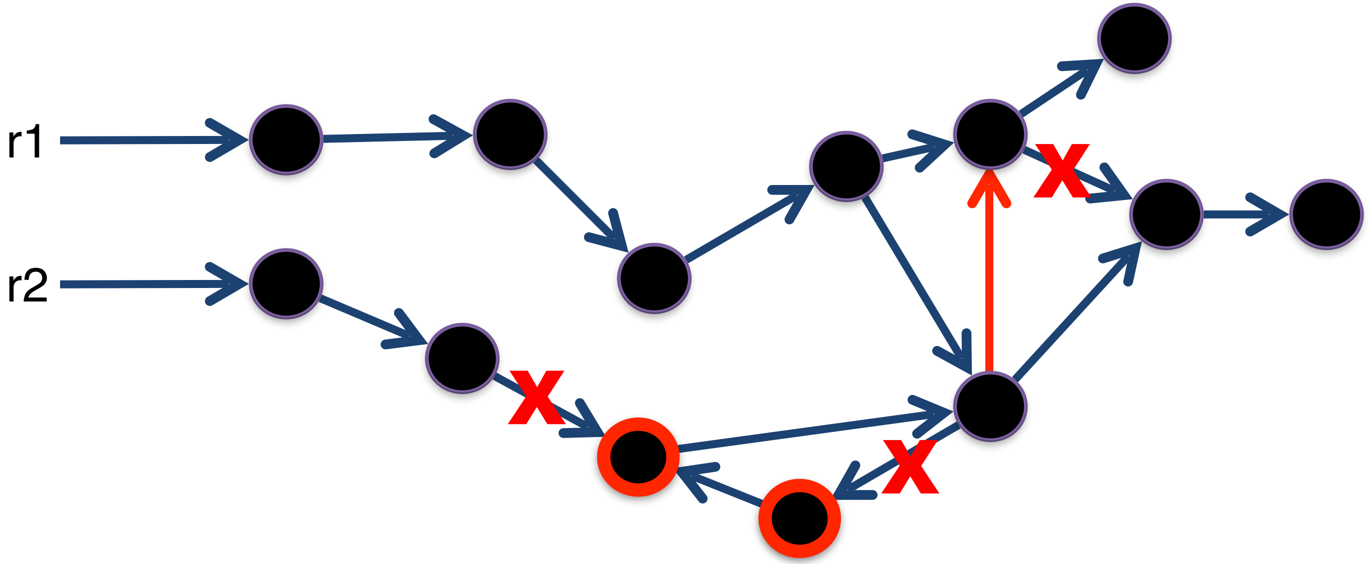
Concurrent GC: mutator and collector threads run in parallel
no need to stop mutator (after roots marked)

Concurrent Mark and Sweep GC



Concurrent GC: mutator and collector threads run in parallel
no need to stop mutator (after roots marked)

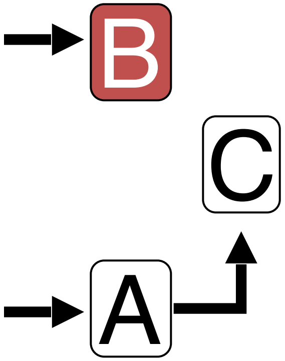
Concurrent Mark and Sweep GC



Concurrent GC: mutator and collector threads run in parallel no need to stop mutator (after roots marked)

Concurrent GC: interference

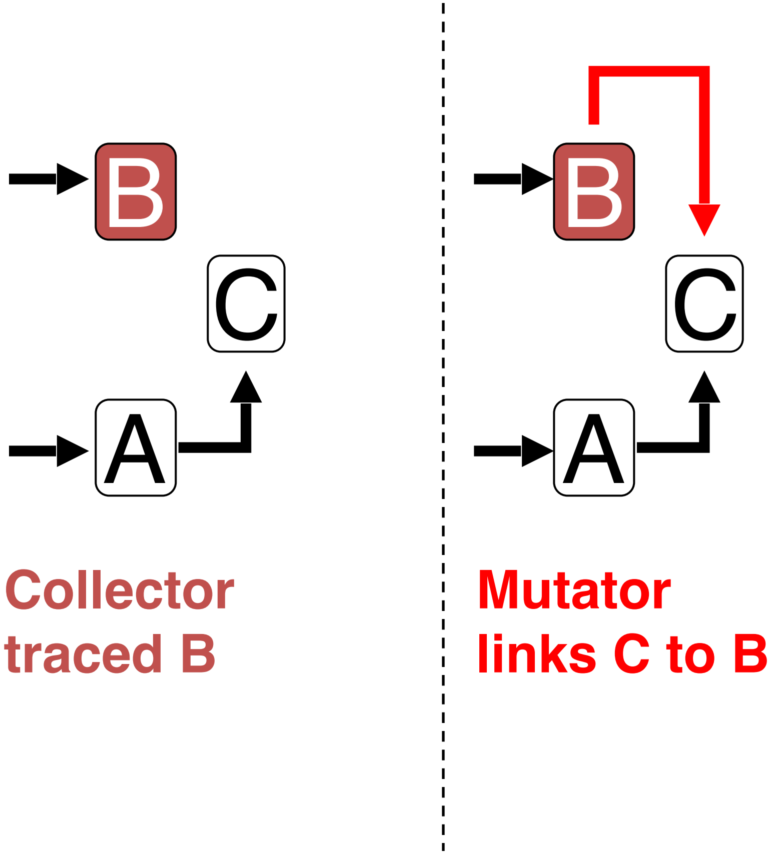
SYSTEM = MUTATOR || COLLECTOR



Collector traced B

Concurrent GC: interference

SYSTEM = MUTATOR II COLLECTOR

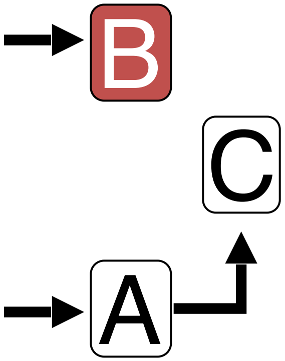


Collector traced B

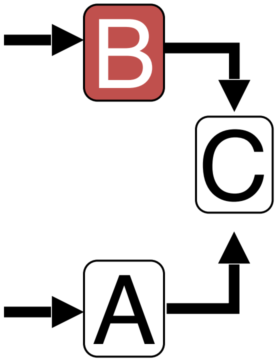
Mutator links C to B

Concurrent GC: interference

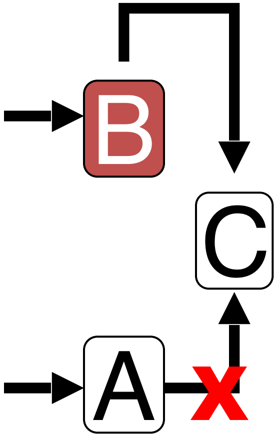
SYSTEM = MUTATOR II COLLECTOR



Collector
traced B



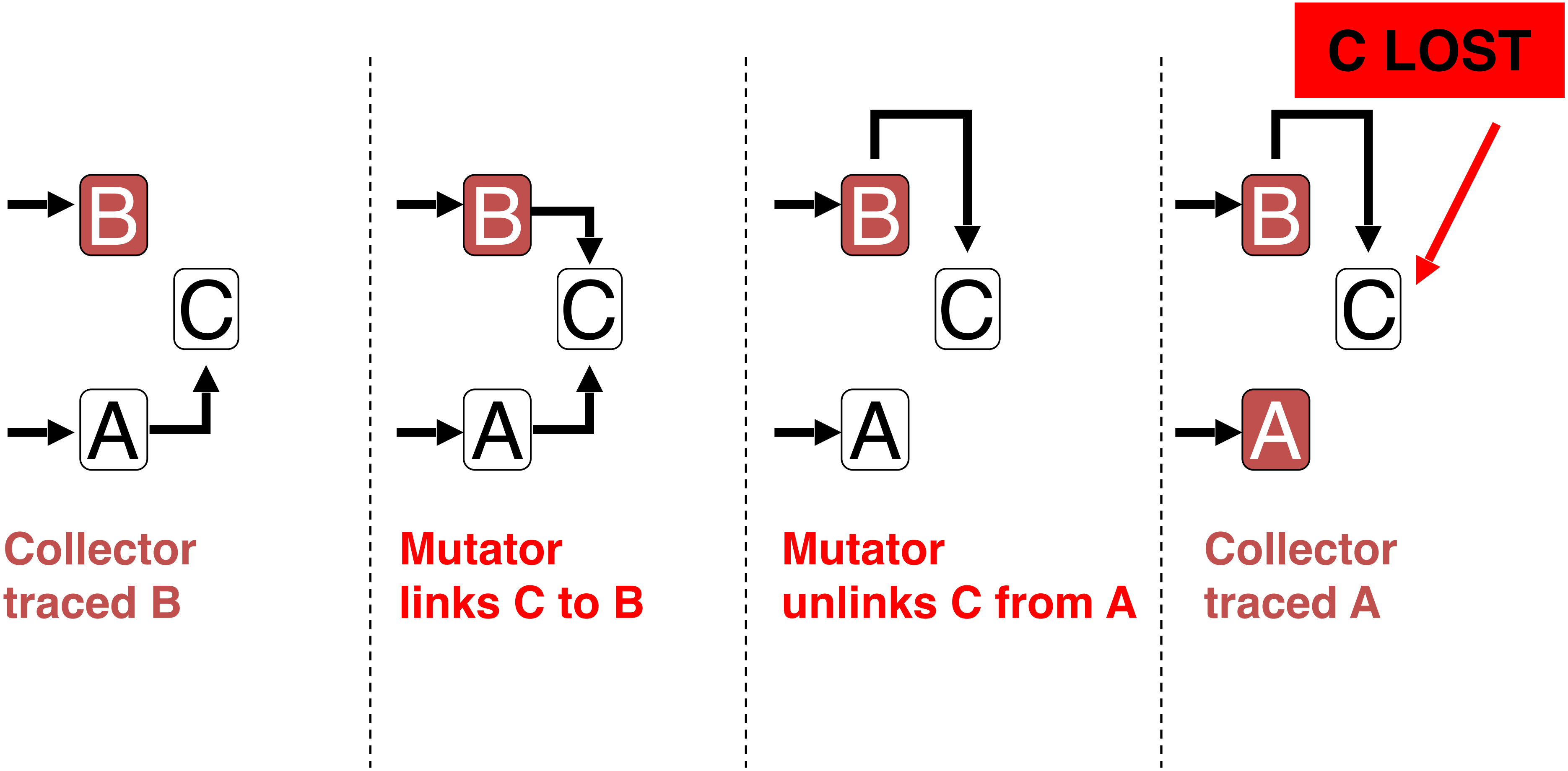
Mutator
links C to B



Mutator
unlinks C from A

Concurrent GC: interference

SYSTEM = MUTATOR II COLLECTOR



C LOST

Collector traced B

Mutator links C to B

Mutator unlinks C from A

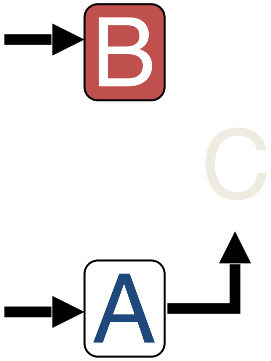
Collector traced A

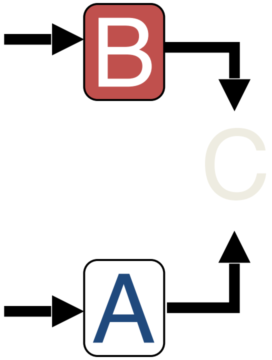
Concurrent GC: families of algorithms

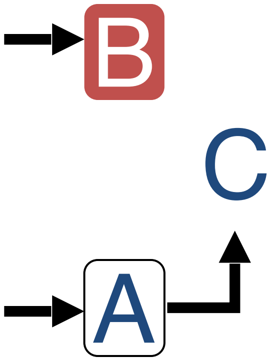
DIJKSTRA
Marks C when
C is linked to B

YUASA
Marks C when
link to C is removed

STEELE
Rescan B when
C is linked to B

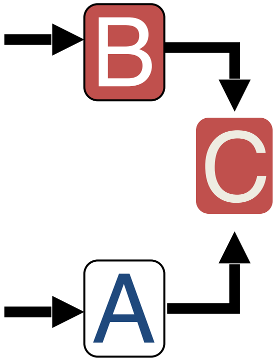




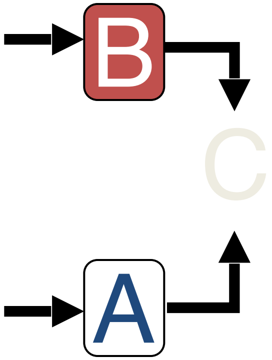


Concurrent GC: families of algorithms

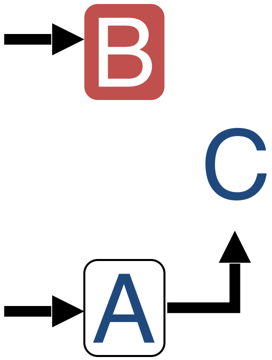
DIJKSTRA
Marks C when
C is linked to B



YUASA
Marks C when
link to C is removed

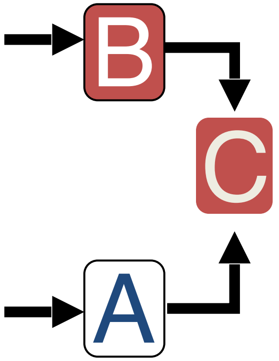


STEELE
Rescan B when
C is linked to B

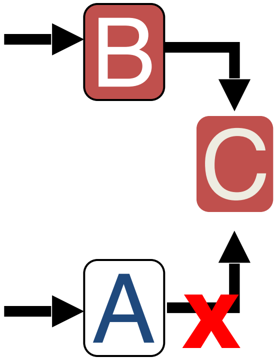


Concurrent GC: families of algorithms

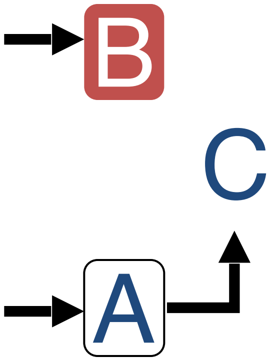
DIJKSTRA
Marks C when
C is linked to B



YUASA
Marks C when
link to C is removed

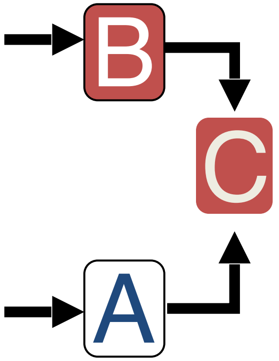


STEELE
Rescan B when
C is linked to B

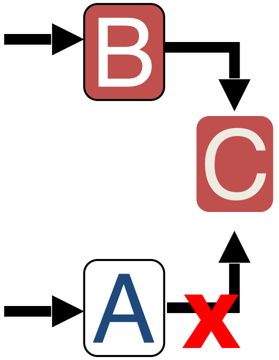


Concurrent GC: families of algorithms

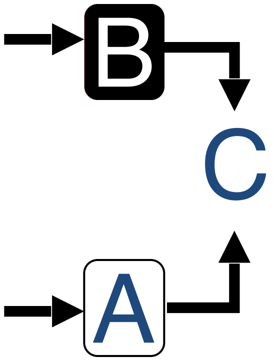
DIJKSTRA
Marks C when
C is linked to B



YUASA
Marks C when
link to C is removed



STEELE
Rescan B when
C is linked to B

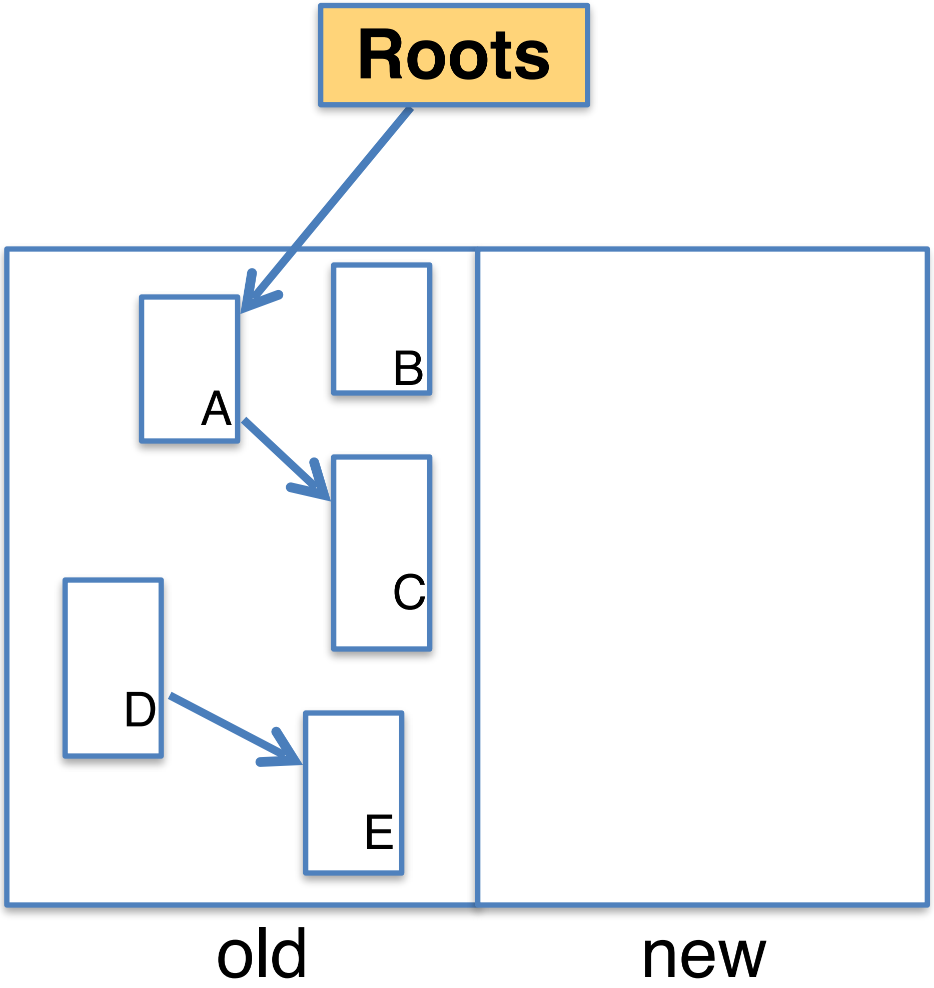


sequential

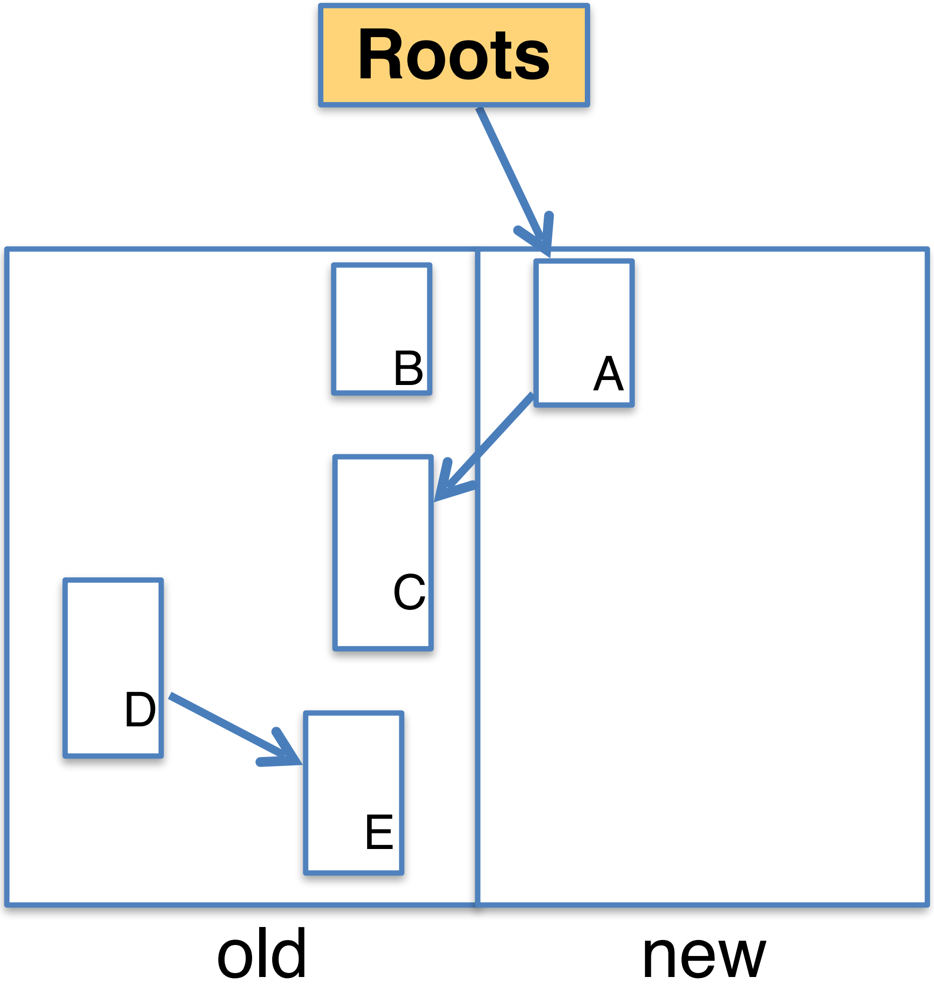
Copying GC

- Partition the heap into two halves
    - old space
    - new space
- Copy all reachable objects from old space to new space
- Swap roles of old and new spaces

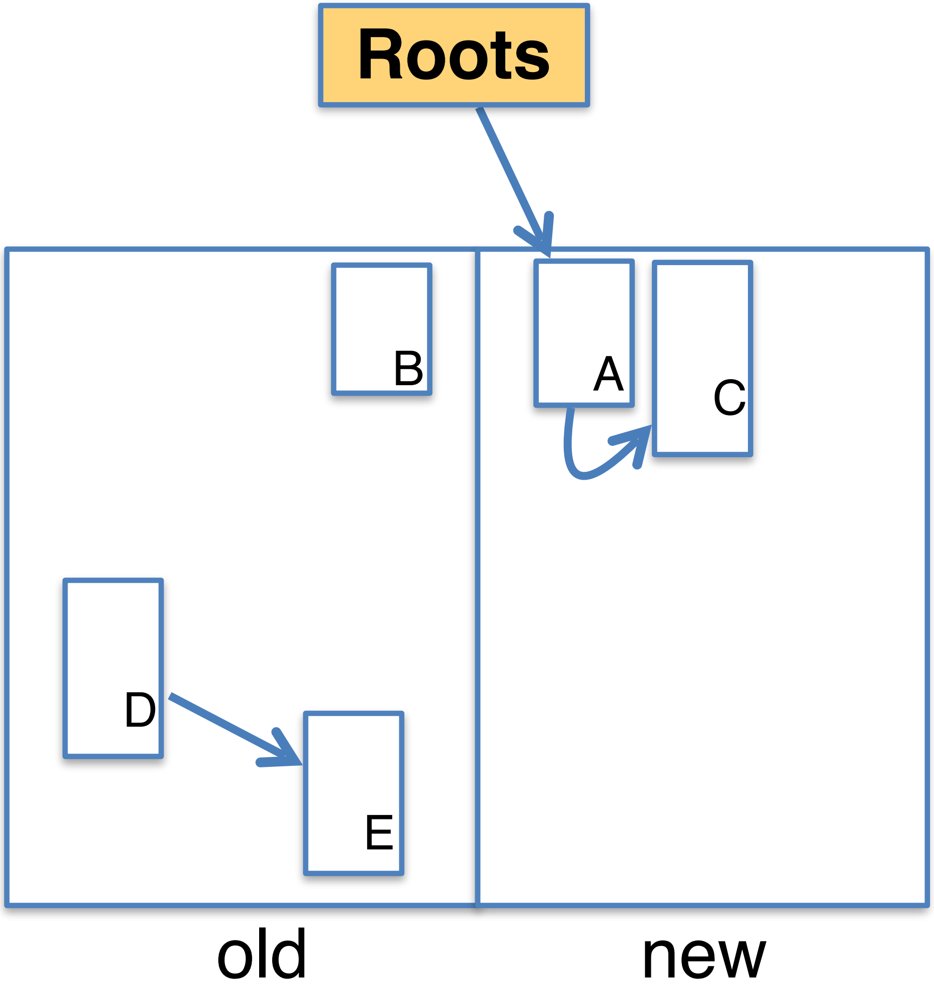
Example: Copying GC



Example: Copying GC



Example: Copying GC



Properties of Copying GC

- Major disadvantage: **half of the heap is not used**
- Compaction for free
- Touch only the live objects
  - **good when most objects are dead**
  - usually most new objects are dead
- **Generational GC:** use a small space for **young** objects and collect this space using copying GC

Very simplistic comparison

	Reference Counting	Mark and 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
Compactio n	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

Modern memory management

- Considers standard program properties
- 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

- Any value can be cast down to a pointer in C
- How can we follow pointers in a structure?

Conservative GC

- Any value can be cast down to a pointer in C
- How can we follow pointers in a structure?
- Easy: conservatively consider anything that can be a pointer to be a pointer
- Practical! Boehm collector

Conservative GC

- Any value can be cast down to a pointer in C
- How can we follow pointers in a structure?
- Easy: conservatively consider anything that can be a pointer to be a pointer
- Practical! Boehm collector
- Can we implement a conservative copying GC?

Conservative GC

- Any value can be cast down to a pointer in C
- How can we follow pointers in a structure?
- Easy: conservatively consider anything that can be a pointer to be a pointer
- Practical! Boehm collector
- Can we implement a conservative copying GC?
- No. Cannot update pointers to the new address: if we don't know whether the value is a pointer, we cannot update it

Summary: Garbage Collection

- Reference counting
- Mark and sweep
- Compaction
- Copying
- Generational
- Parallel
- Concurrent

ERROR HANDLING

Runtime checks

- Generate code for checking attempted illegal operations
  - null pointer check
    - array length, virtual call, reference arguments to library call
  - array bounds check
  - array allocation size check
  - division by zero
  - …
- If check fails jump to **error handler** code that prints a message and gracefully exists program
- Alternatively, use an **exception handling** mechanism

Null pointer check

```
# null pointer check
cmp $0,%eax
je labelNPE
```

Single generated handler for entire program

```
labelNPE:
    push $strNPE    # error message
    call __println
    push $1         # error code
    call __exit
```

Array bounds check

```
# array bounds check
mov -4(%eax),%ebx  # ebx = length
mov $0,%ecx        # ecx = index
cmp %ecx,%ebx
jle labelABE       # ebx <= ecx ?
cmp $0,%ecx
jl  labelABE       # ecx < 0 ?
```

Single generated handler for entire program

```
labelABE:
   push $strABE    # error message
   call __println
   push $1         # error code
   call __exit
```

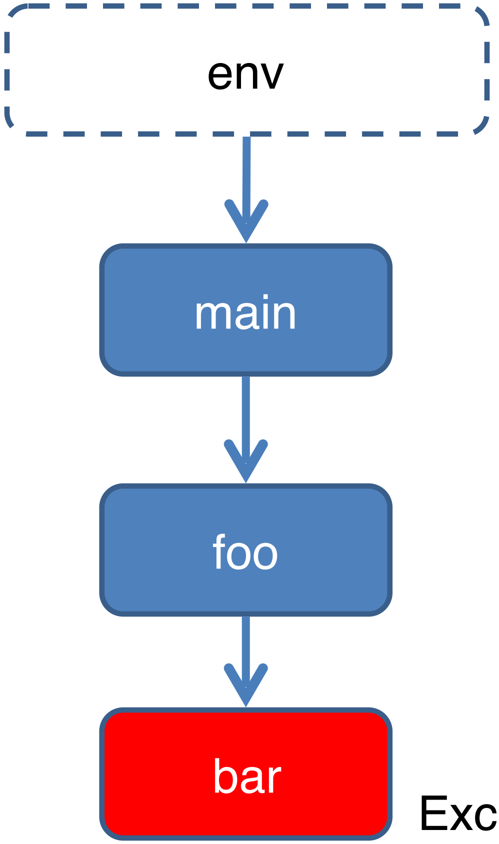
Array allocation size check

```
# array size check
cmp $0,%eax        # eax == array size
jle labelASE       # eax <= 0 ?
```

Single generated handler for entire program

```
labelASE:
    push $strASE       # error message
    call __println
    push $1            # error code
    call __exit
```

Exceptions



Exception thrown

Exception example

```
org.eclipse.swt.SWTException: Graphic is disposed
        at org.eclipse.swt.SWT.error(SWT.java:3744)
        at org.eclipse.swt.SWT.error(SWT.java:3662)
        at org.eclipse.swt.SWT.error(SWT.java:3633)
        at org.eclipse.swt.graphics.GC.getClipping(GC.java:2266)
        at com.aelitis.azureus.ui.swt.views.list.ListRow.doPaint(ListRow.java:260)
        at com.aelitis.azureus.ui.swt.views.list.ListRow.doPaint(ListRow.java:237)
        at com.aelitis.azureus.ui.swt.views.list.ListView.handleResize(ListView.java:867)
        at com.aelitis.azureus.ui.swt.views.list.ListView$5$2.runSupport(ListView.java:406)
        at org.gudy.azureus2.core3.util.AERunnable.run(AERunnable.java:38)
        at org.eclipse.swt.widgets.RunnableLock.run(RunnableLock.java:35)
        at org.eclipse.swt.widgets.Synchronizer.runAsyncMessages(Synchronizer.java:130)
        at org.eclipse.swt.widgets.Display.runAsyncMessages(Display.java:3323)
        at org.eclipse.swt.widgets.Display.readAndDispatch(Display.java:2985)
        at org.gudy.azureus2.ui.swt.mainwindow.SWTThread.<init>(SWTThread.java:183)
        at org.gudy.azureus2.ui.swt.mainwindow.SWTThread.createInstance(SWTThread.java:67)
     ….
```