CS107: Memory Management

Guannan Wei
guannan.wei@tufts.edu

April 7, 2026

Spring 2026

Tufts University

Memory Management

The memory of a computer is a finite resource. Typical programs use a lot of memory over their lifetime, but not all of it at the same time.

The goal of **memory management** is to use that finite resource as efficiently as possible, according to some criterion.

In general, programs dynamically allocate memory from two different areas: the stack and the heap. Since the management of the stack is trivial, the term memory management usually designates that of the heap.

The Memory Manager

The **memory manager** is the part of the **run time system** in charge of managing heap memory.

The Memory Manager

The **memory manager** is the part of the **run time system** in charge of managing heap memory.

Allocation: Its job consists in maintaining the set of free **memory blocks** (also called objects later) and to use them to fulfill allocation requests from the program.

The Memory Manager

The **memory manager** is the part of the **run time system** in charge of managing heap memory.

Allocation: Its job consists in maintaining the set of free **memory blocks** (also called objects later) and to use them to fulfill allocation requests from the program.

Deallocation:

- **Explicit**: the program asks for a block to be freed (e.g. `free` in C, `delete` in C++, etc.).
- **Implicit**: the memory manager automatically tries to free unused blocks when it does not have enough free memory to satisfy an allocation request.

Explicit Deallocation

Explicit memory deallocation presents several problems:

1) memory can be freed too early, which leads to **dangling pointers** - and then to data corruption, crashes, security issues, etc.
2) memory can be freed too late - or never - which leads to **space leaks**.

Due to these problems, most modern programming languages are designed to provide **implicit deallocation**, also called **automatic memory management** - or **garbage collection**, even though garbage collection refers to a specific kind of automatic memory management.

Implicit Deallocation

Implicit memory deallocation is based on the following conservative assumption:

> *If a block of memory is reachable, then it will be used again in the future, and therefore it cannot be freed. Only unreachable memory blocks can be freed.*

Implicit Deallocation

Implicit memory deallocation is based on the following conservative assumption:

> *If a block of memory is reachable, then it will be used again in the future, and therefore it cannot be freed. Only unreachable memory blocks can be freed.*

Since this assumption is conservative, it is possible to have memory leaks even with implicit memory deallocation. This happens whenever a reference to a memory block is kept, but the block is not accessed anymore.

However, implicit deallocation completely eliminates dangling pointers (ensuring memory safety).

Garbage Collection

Garbage collection (GC) is a common name for a set of techniques that automatically reclaim objects that are not reachable anymore.

We will examine several garbage collection techniques:

1) reference counting,
2) mark & sweep garbage collection,
3) copying garbage collection.

All these techniques have concepts in common, which we introduce now.

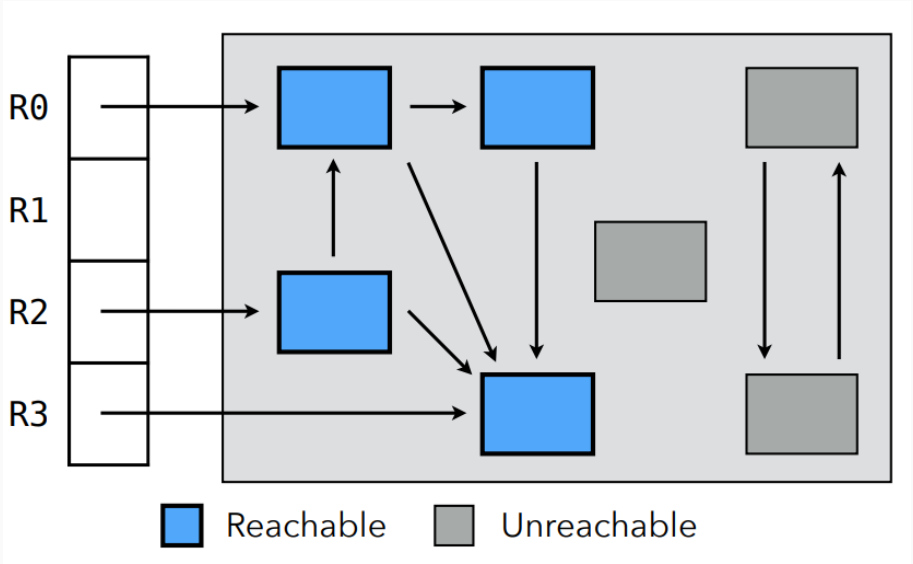
Reachable Objects

At any time during the execution of a program, we can define the set of **reachable objects** as being:

- the objects immediately accessible from global variables, the stack or registers - called the **roots** or **root set**,
- the objects transitively reachable from other reachable objects, by following pointers.

These objects form the **reachability graph**.

Reachability Graph Example



(Im)precision

To compute the reachability graph at run time, it needs to *unambiguously identify pointers* in registers, in the stack, and in heap objects.

If this is not possible, the reachability graph must be approximated. Such an approximation must be safe for the task at hand!

For example, if a sequence of bytes looks like pointer, then we have to assume it can be a pointer, even if it is not.

GC Technique #1: Reference Counting

The idea of **reference counting** is simple:

> *Every object carries a count of the number of pointers that reference it. When this count reaches zero, the object is guaranteed to be unreachable and can be deallocated.*

Reference counting requires collaboration from the compiler and/or programmer to make sure that reference counts are properly maintained!

Pros And Cons

Reference counting is relatively easy to implement, even as a library. It reclaims memory immediately.

Examples: C++'s `std::shared_ptr`, Rust's `Rc<T>`, etc.

Pros And Cons

Reference counting is relatively easy to implement, even as a library. It reclaims memory immediately.

Examples: C++'s `std :: shared_ptr` , Rust's `Rc<T>`, etc.

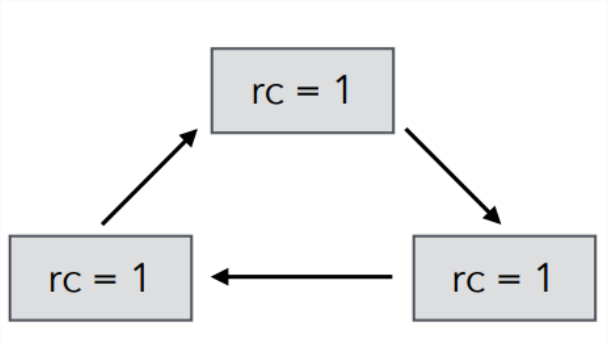
However, it has an important impact on space consumption, and speed of execution: every object must contain a counter, and every pointer acquisition/release must update it.

But the biggest problem is cyclic structures...

Cyclic Structures

The reference count of objects that are part of a cycle in the object graph never reaches zero, even when they become unreachable!

This is **the** major problem of reference counting.



Cyclic Structures

The problem with cyclic structures is due to the fact that reference counts provide only an approximation of reachability.

In other words, we have:

`reference_count(x) = 0` $\implies$ `x is unreachable`

but the opposite is not true!

Uses Of Reference Counting

Due to its problem with cyclic structures, reference counting is rarely used alone. It has also been used in combination with a mark & sweep GC, the latter being run infrequently to collect cyclic structures.

It is still interesting for systems that do not allow cyclic structures to be created — e.g. hard links in Unix file systems.

GC Technique #2: Mark & Sweep

Mark & sweep garbage collection is a GC technique that proceeds in two successive phases:

1) in the **marking** phase, the reachability graph is traversed and reachable objects are marked,
2) in the **sweeping** phase, all allocated objects are examined, and unmarked ones are freed.

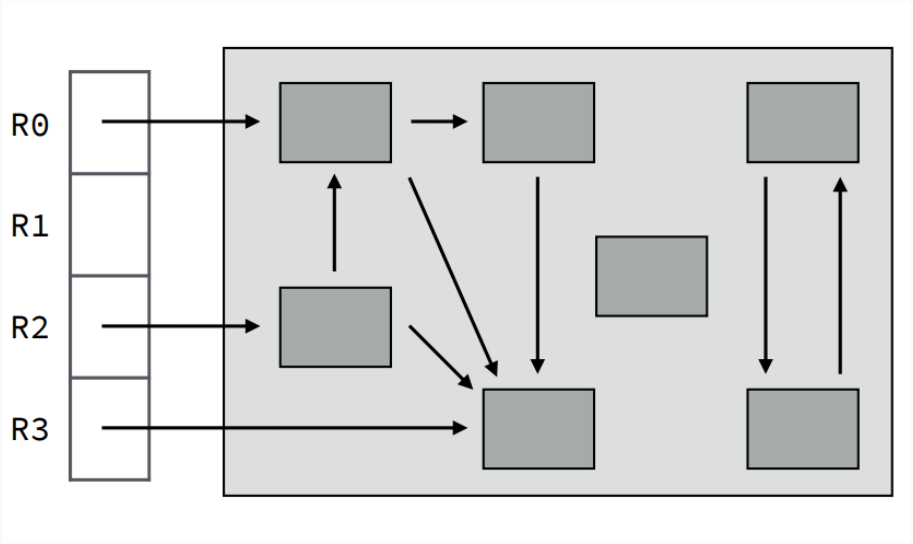
GC Technique #2: Mark & Sweep

Mark & sweep garbage collection is a GC technique that proceeds in two successive phases:

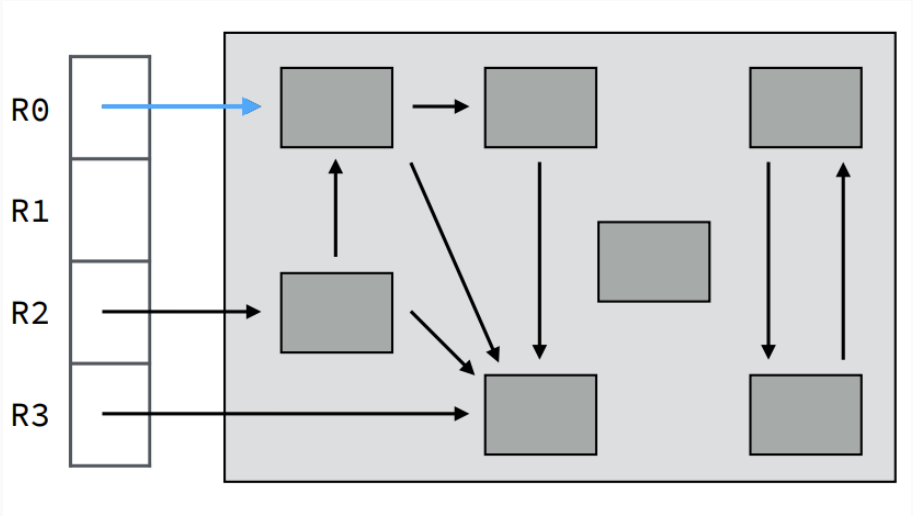
1) in the **marking** phase, the reachability graph is traversed and reachable objects are marked,
2) in the **sweeping** phase, all allocated objects are examined, and unmarked ones are freed.

GC is triggered by a lack of memory, and must complete before the program can be resumed ("stop the world''). This is necessary to ensure that the reachability graph is not modified by the program while the GC traverses it.

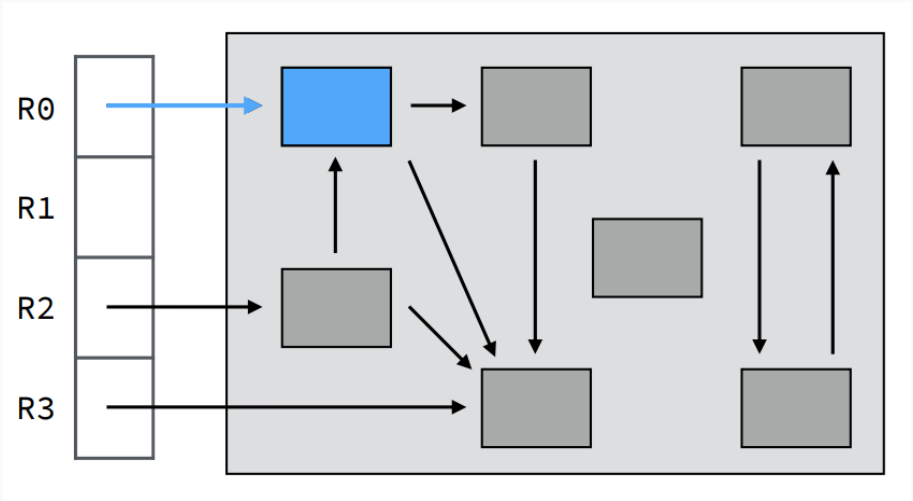
Mark & Sweep GC



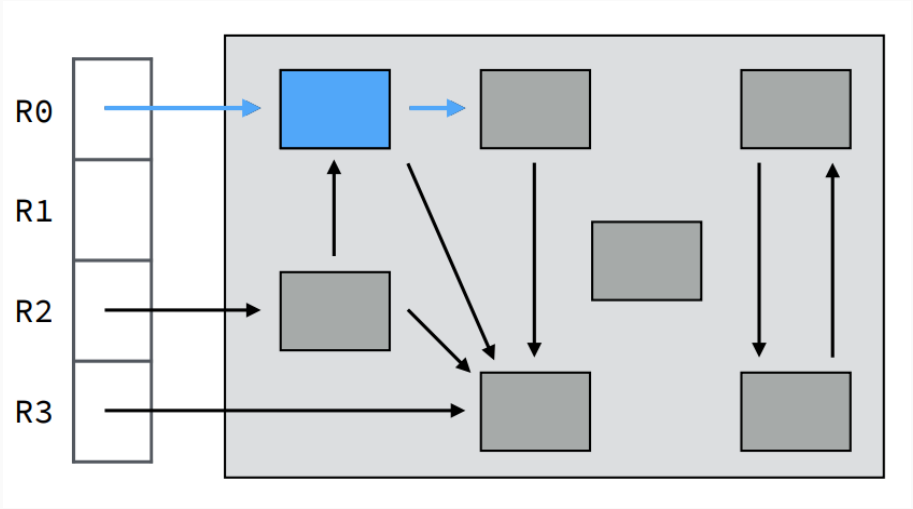
Mark & Sweep GC



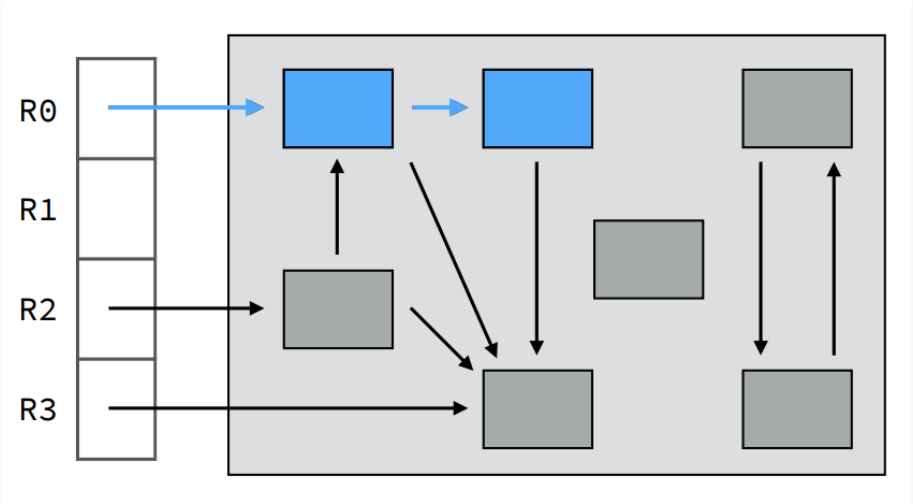
Mark & Sweep GC



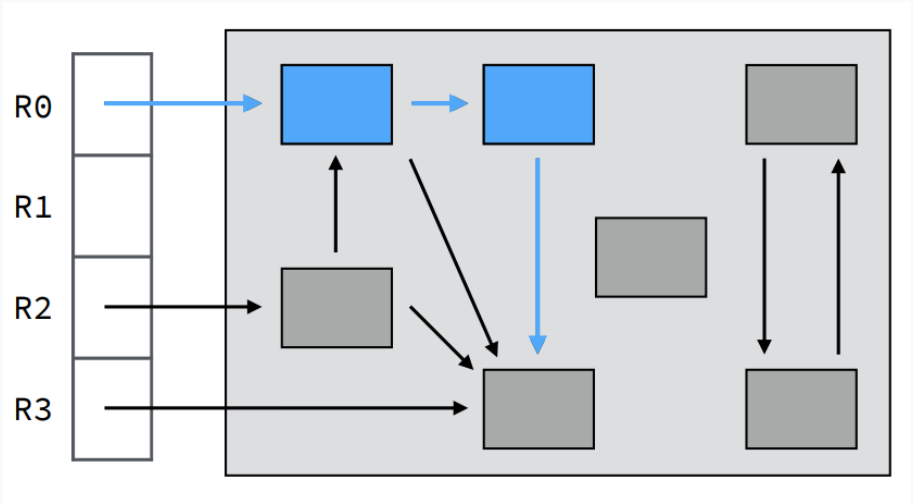
Mark & Sweep GC



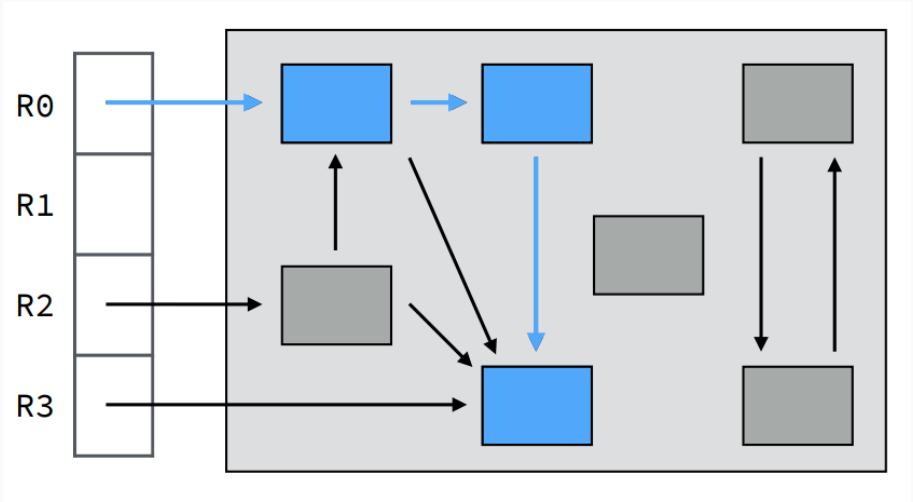
Mark & Sweep GC



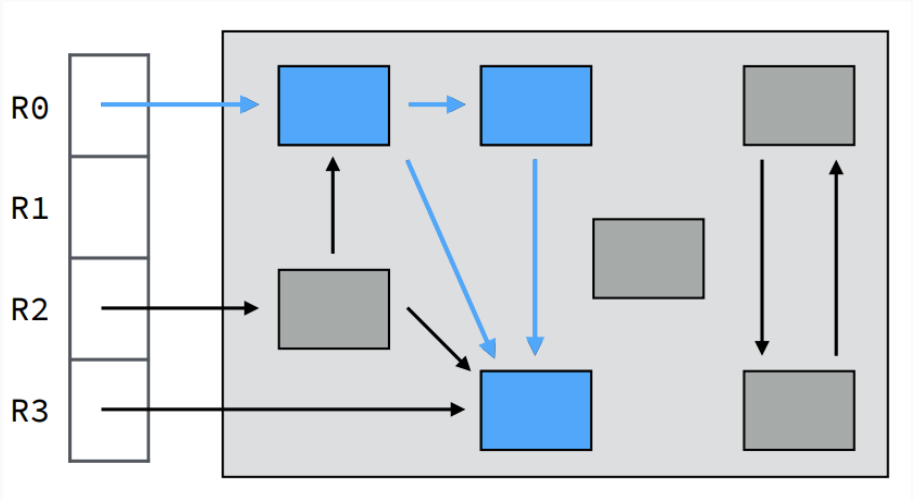
Mark & Sweep GC



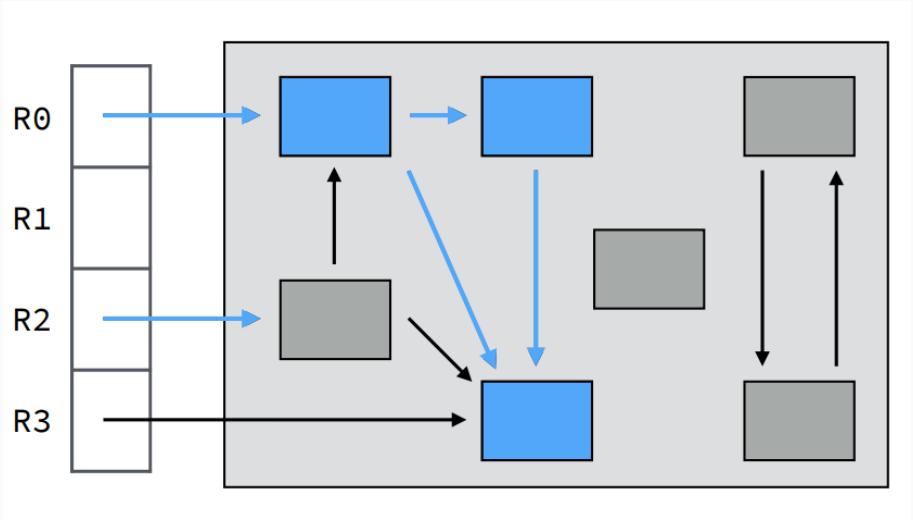
Mark & Sweep GC



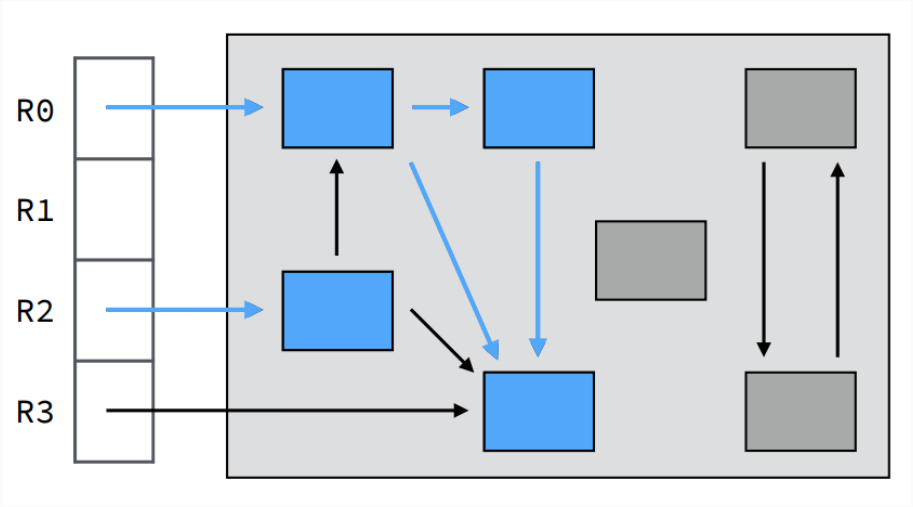
Mark & Sweep GC



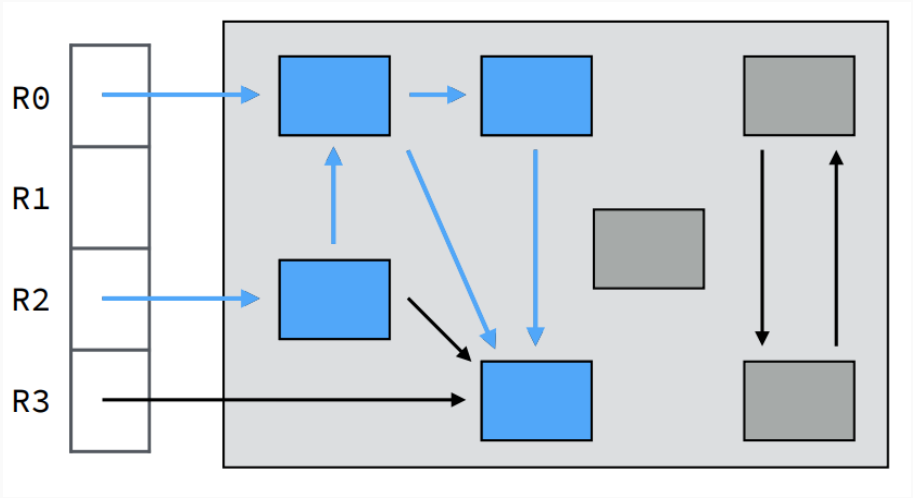
Mark & Sweep GC



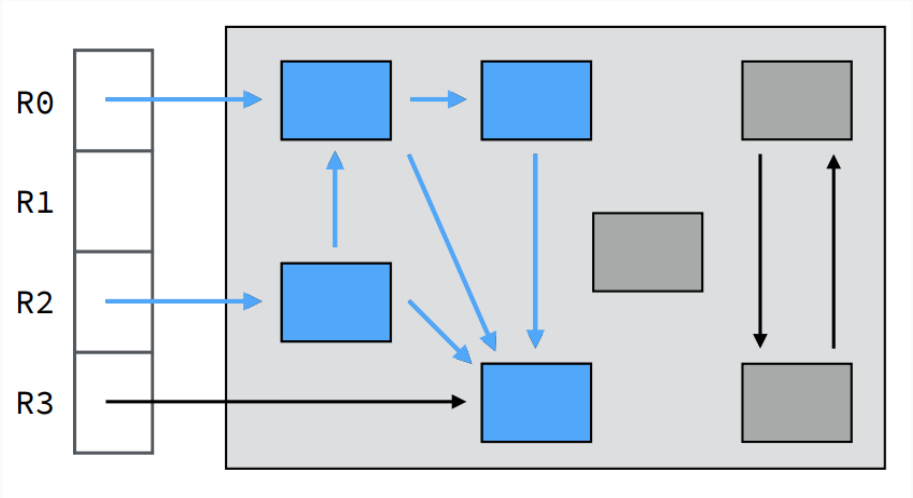
Mark & Sweep GC



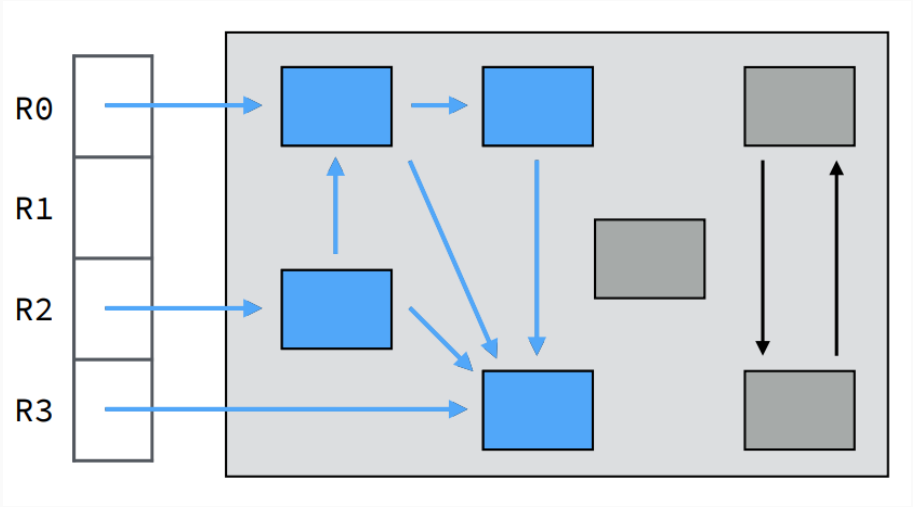
Mark & Sweep GC



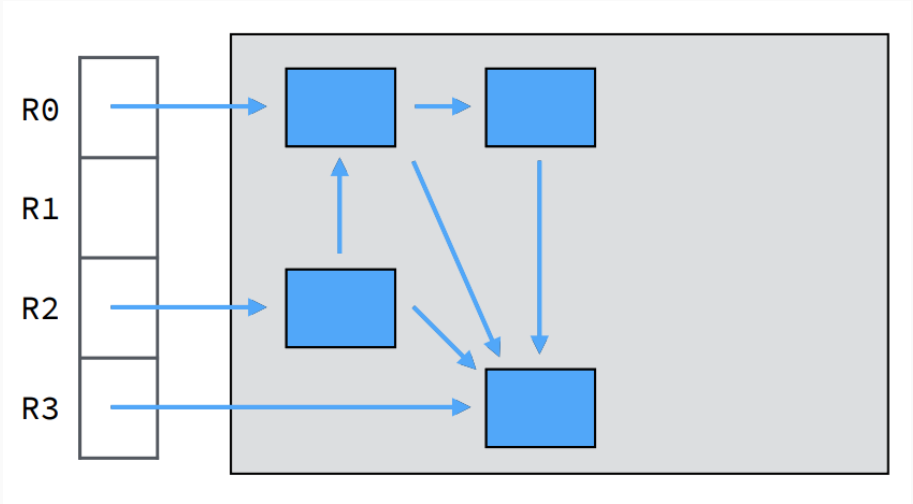
Mark & Sweep GC



Mark & Sweep GC



Mark & Sweep GC



Memory Manager Data Structures - Free List

How does the memory manager organize the memory and track whether blocks are free or allocated?

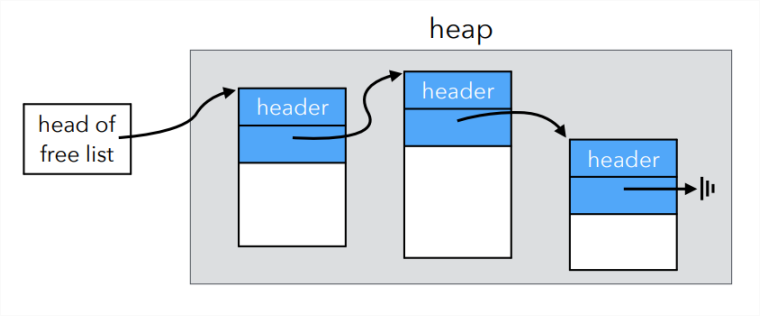
Memory Manager Data Structures - Free List

How does the memory manager organize the memory and track whether blocks are free or allocated?

In a mark & sweep GC, free blocks are **not** contiguous in memory. Therefore, they have to be stored in a data structure usually known as the **free list**.

In a simple GC, this free list could effectively be a linked list. In practice, the representation of free lists varies depending on the organization of the heap and the properties of the garbage collector.

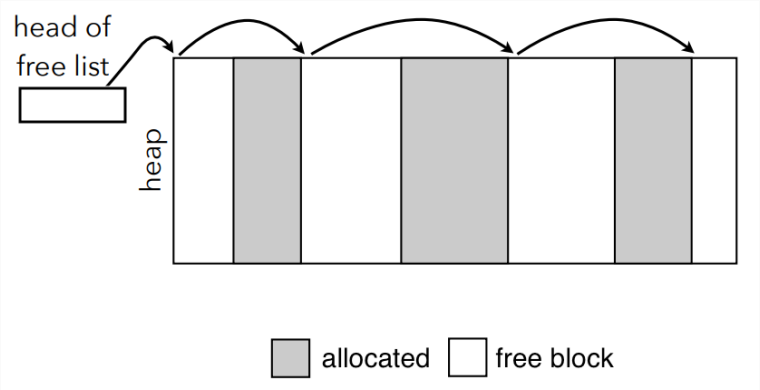
Memory Manager Data Structures - Free List



Since the free blocks are, by definition, not used by the program, the links can be stored in the blocks themselves!

Memory Manager Data Structures - Free List

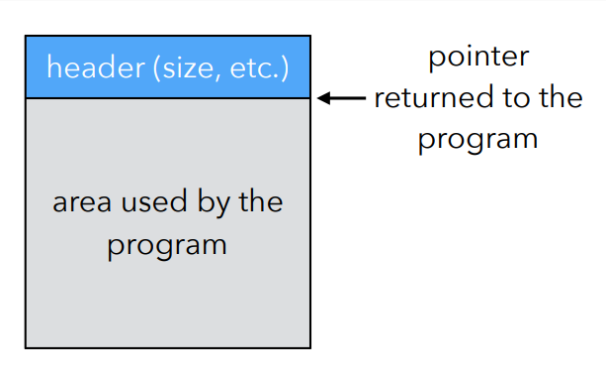
The "flat'' view of the heap and free list:



Memory Manager Data Structures - Block Header

The memory manager must associate a set of properties to the blocks it manages, for example their size.

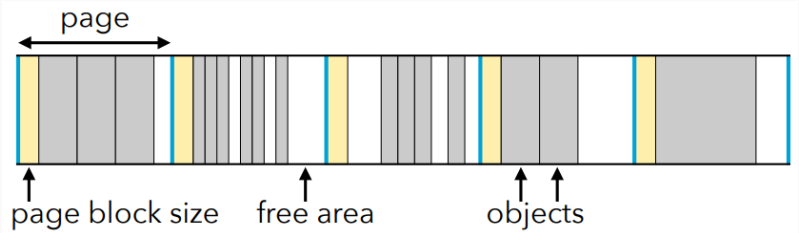
This is typically done by storing these properties in a **block header**, stored just before the area used by the program.



BiBoP

To decrease the overhead of headers, the technique known as **BiBoP** (for **big bag of pages**) groups objects of the same size into contiguous areas of memory called pages.

- All pages have the same power-of-two size $s = 2^b$ (e.g. 4096 bytes) and start at an address which is a multiple of s.
- The size of all the objects on a page is stored at the page's beginning, and can be retrieved by masking the b least-significant bits of an object's address. (E.g. object at 0b10010000 (128 + 16) so base is 0b10000000.)



Marking Objects

Reachable objects must be marked in some way.

Since only one bit is required for the mark, it is possible to store it in the block header, along with the size.

Marking Objects

Reachable objects must be marked in some way.

Since only one bit is required for the mark, it is possible to store it in the block header, along with the size.

- For example, if the system guarantees that all blocks have an even size, then the least significant bit (LSB) of the block size can be used for marking.
- It is also possible to use "external" bit maps - stored in a memory area that is private to the GC - to store mark bits.

Reachability Graph Traversal

The mark phase requires a depth-first traversal of the reachability graph. This is usually implemented by recursion. Recursive function calls use stack space, and since the depth of the reachability graph is not bounded, the GC can overflow its stack!

Several techniques - not presented here - have been developed to either recover from those overflows, or avoid them altogether by storing the stack in the objects being traced.

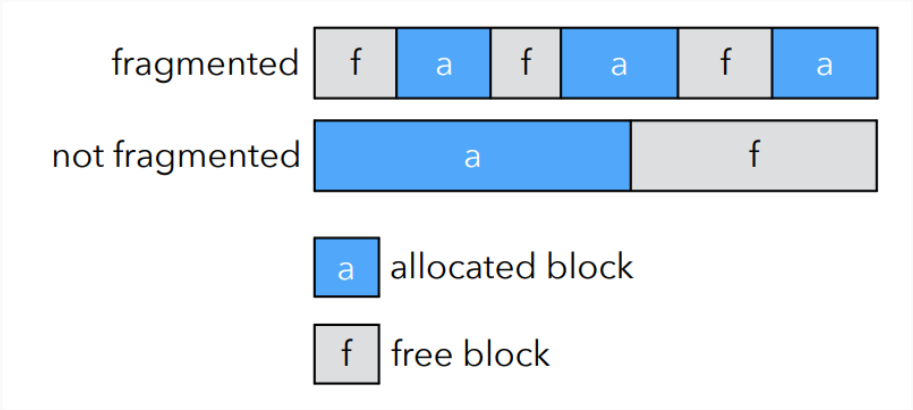
Fragmentation

The term **fragmentation** refers to two different but similar problems associated with memory management:

- **External fragmentation** refers to the fragmentation of free memory in many small blocks.
- **Internal fragmentation** refers to the waste of memory due to the use of a free block larger than required to satisfy an allocation request.

External Fragmentation

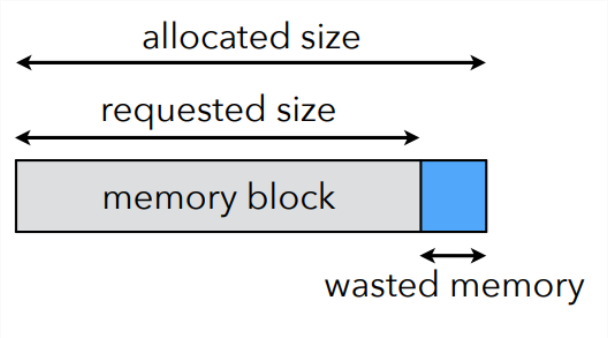
The following two heaps have the same amount of free memory, but the first suffers from **external fragmentation** while the second does not. As a consequence, some requests can be fulfilled by the second but not by the first.



Internal Fragmentation

For various reasons - e.g. alignment constraints - the memory manager sometimes allocates slightly more memory than requested by the client. This results in small amounts of wasted memory scattered in the heap.

This phenomenon is called **internal fragmentation**.



Allocation Policy

When a block of memory is requested, there are in general many free blocks large enough to satisfy the request.

An **allocation policy** must therefore be used to decide which of those candidates to choose. A good allocation policy should minimize fragmentation while being fast to implement.

The most commonly used policies are:

- **first fit**, which uses the first suitable block,
- **best fit**, which uses the smallest block big enough to satisfy the request.

Splitting And Coalescing

When the memory manager has found a free block big enough to satisfy an allocation request, it is possible for that block to be bigger than the size requested.

In that case, the block can be **split** in two parts: one part is returned to the client, while the other is put back into the free list.

Splitting And Coalescing

When the memory manager has found a free block big enough to satisfy an allocation request, it is possible for that block to be bigger than the size requested.

In that case, the block can be **split** in two parts: one part is returned to the client, while the other is put back into the free list.

The opposite must be done during deallocation: if the block being freed is adjacent to one or two other free blocks, then they all should be **coalesced** to form a bigger free block.

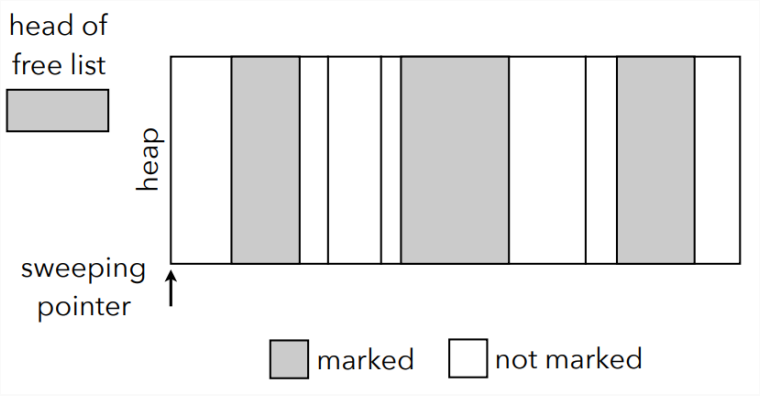
Sweeping Objects

Once the mark phase has terminated, all allocated but unmarked objects can be freed. This is the job of the sweep phase, which traverses the whole heap sequentially, looking for unmarked objects and adding them to the free list.

Notice that unreachable objects cannot become reachable again. It is therefore possible to sweep objects on demand, to only fulfill the current memory need. This is called **lazy sweep**.

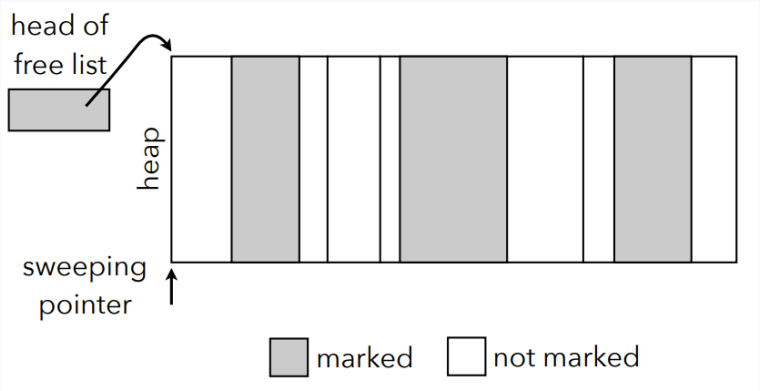
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



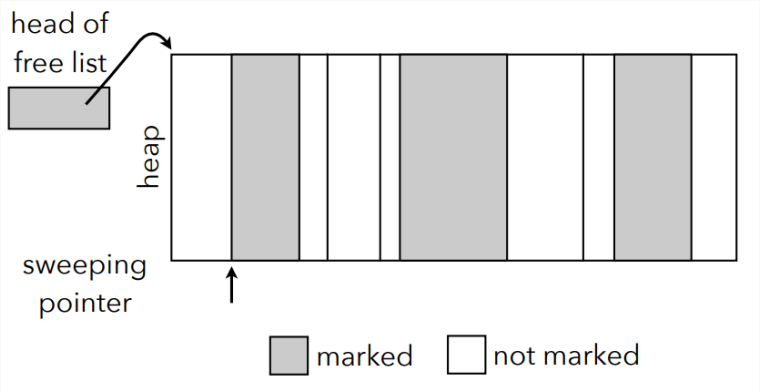
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



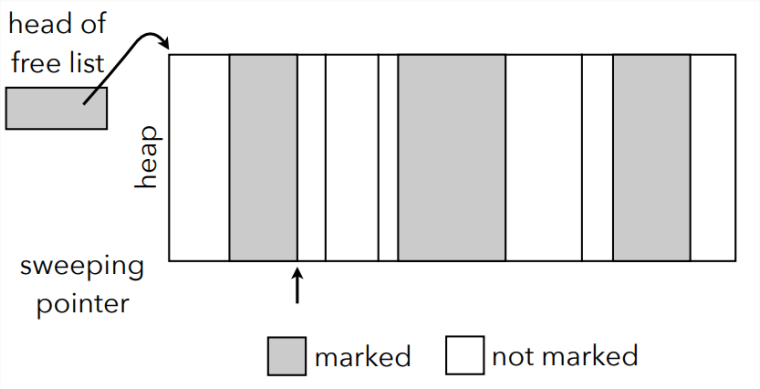
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



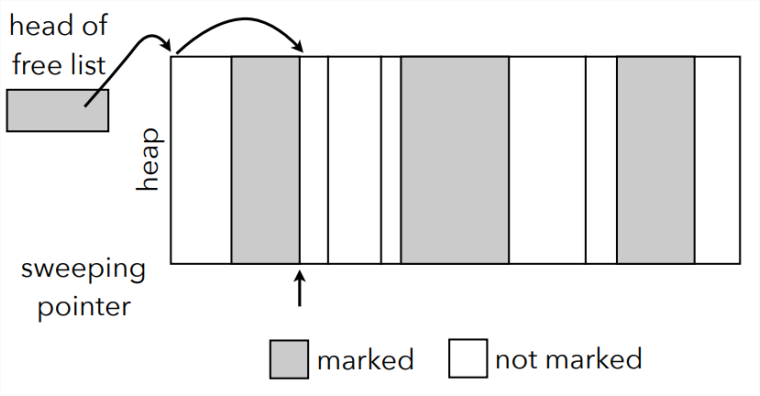
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



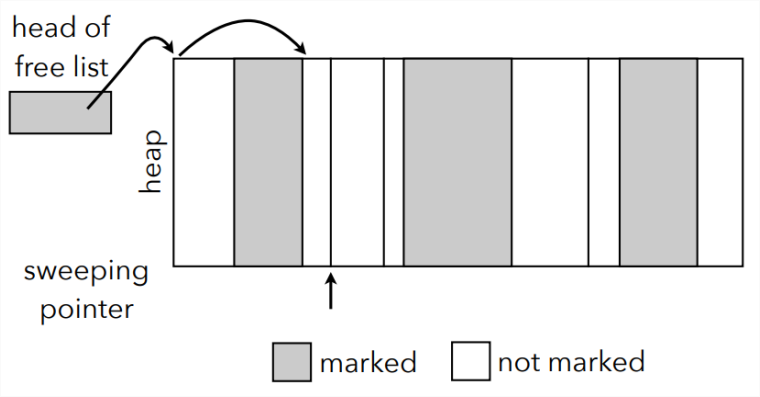
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



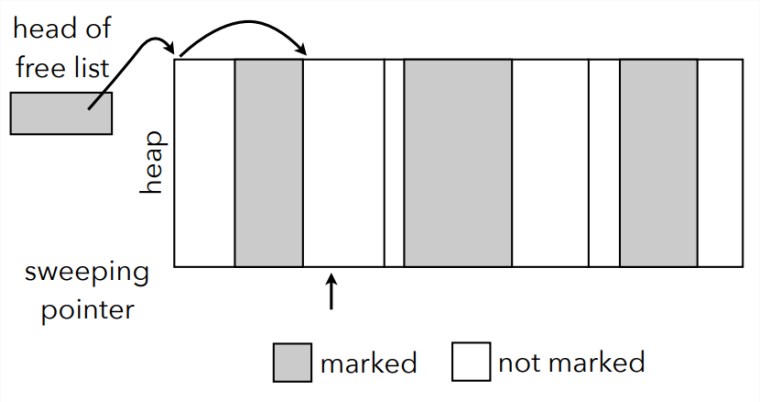
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



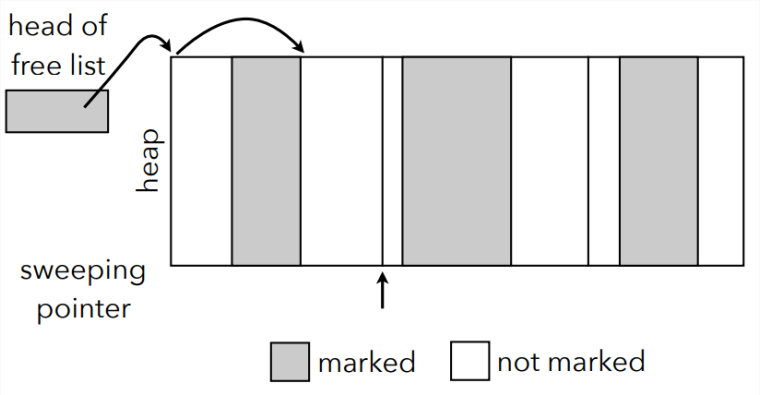
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



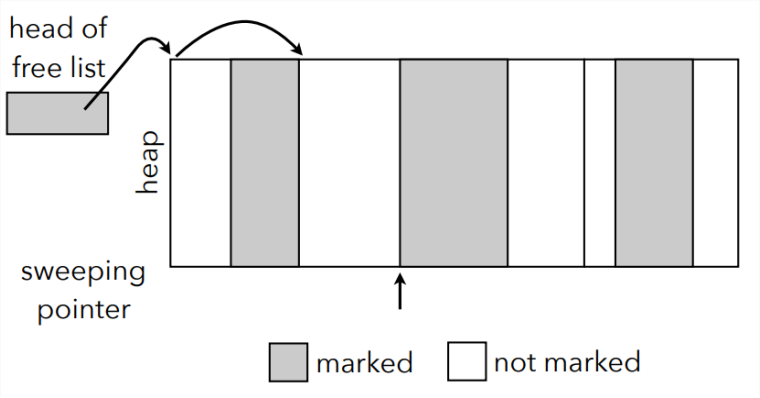
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



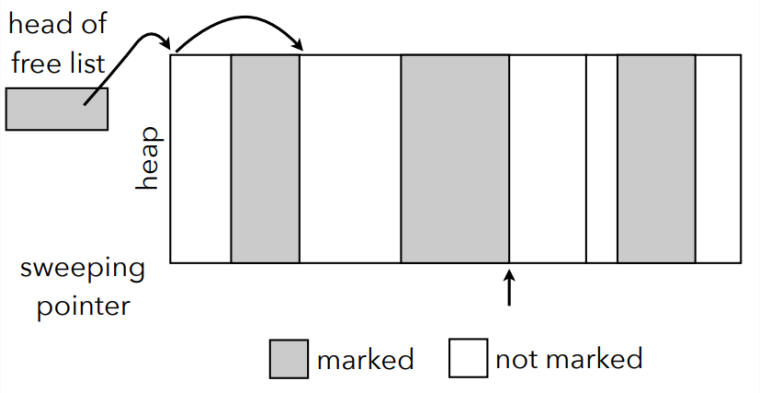
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



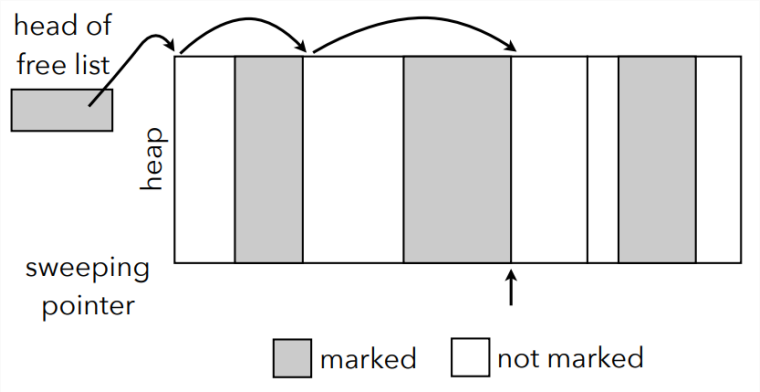
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



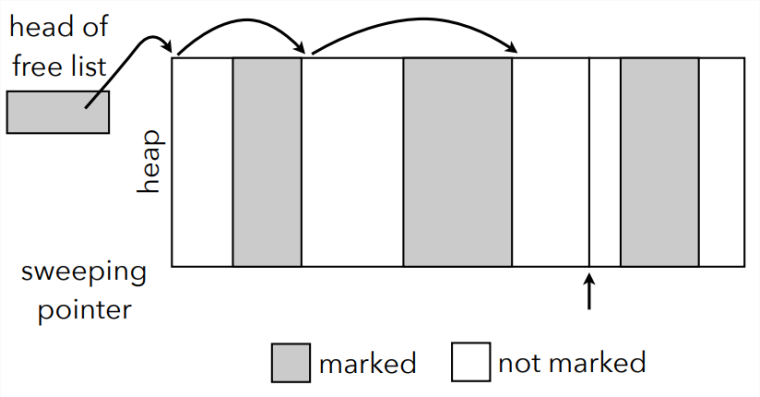
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



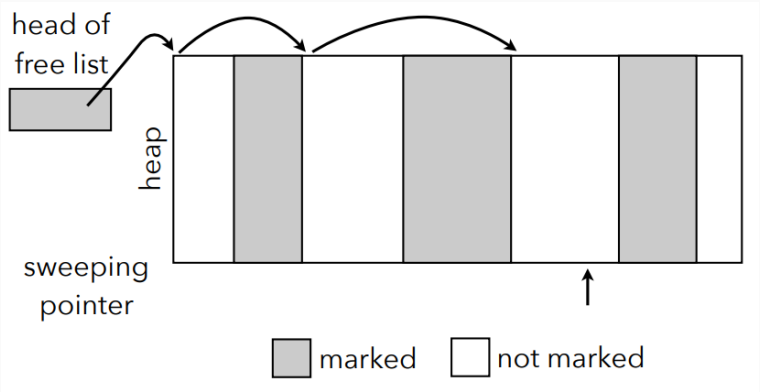
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



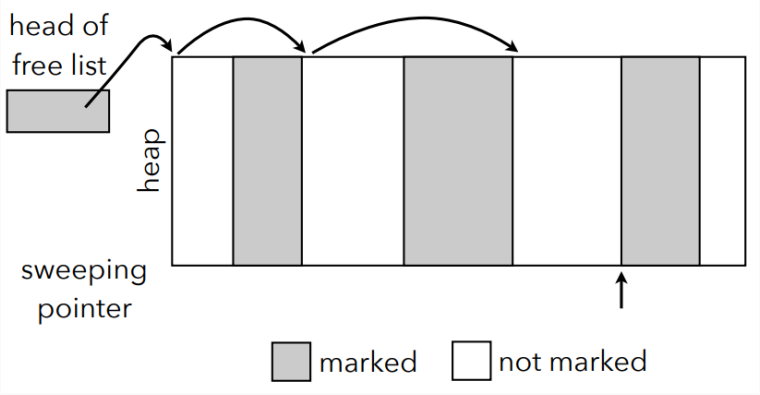
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



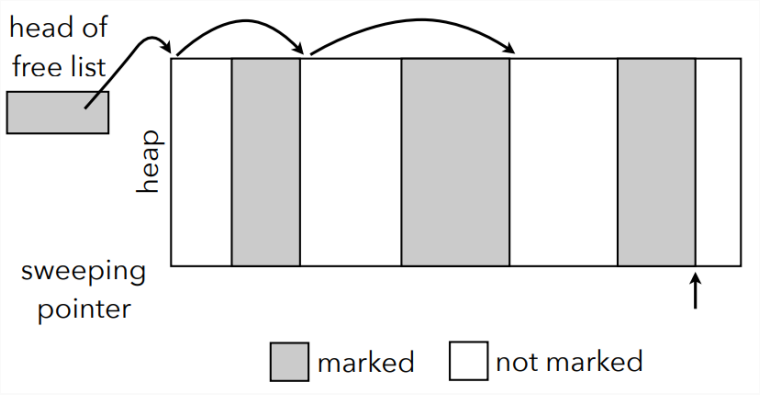
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



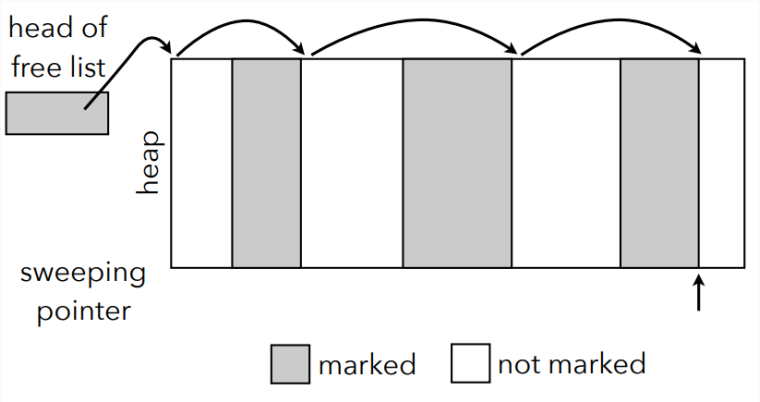
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



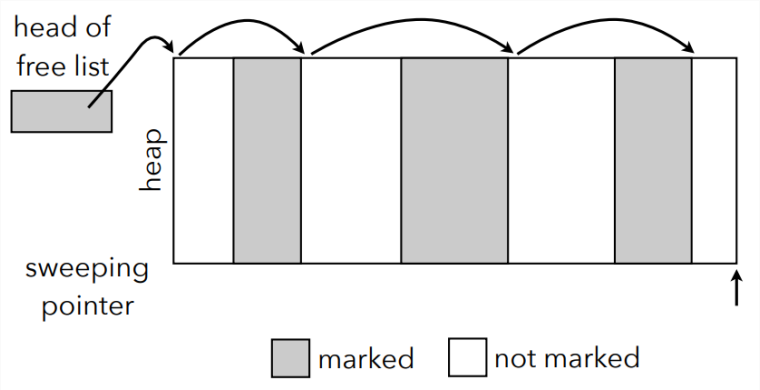
Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



Sweeping And Coalescing

The sweeping phase traverses the whole heap from beginning to end, and completely rebuilds the free list. It performs coalescing in the process.



Conservative M&S GC

A **conservative (mark & sweep) garbage collector** GC assumes that everything that looks like a pointer to an allocated object is a pointer to an allocated object.

This assumption is conservative - in that it can lead to the retention of dead objects - but safe - in that it cannot lead to the freeing of live objects.

It is however very important that the GC uses as many hints as possible to avoid considering non-pointers as pointers, as this can lead to memory being retained needlessly.

Pointer Identification

Several characteristics of the architecture or compiler can be used to filter the set of potential pointers, e.g.:

- Many architectures require pointers to be aligned in memory on 2 or 4 bytes boundaries. Therefore, unaligned potential pointers can be ignored.

Pointer Identification

Several characteristics of the architecture or compiler can be used to filter the set of potential pointers, e.g.:

- Many architectures require pointers to be aligned in memory on 2 or 4 bytes boundaries. Therefore, unaligned potential pointers can be ignored.
- Many compilers guarantee that if an object is reachable, then there exists at least one pointer to its beginning (i.e. the base address). Therefore, (potential) interior pointers can be ignored.

```
struct Pair { int x; int y; };
Pair* p = malloc(sizeof(Pair)); // the base pointer
int* q = &p->y;                 // an interior pointer
```

Exercise

The POSIX `malloc` function does not clear the memory it returns to the user program, for performance reasons.

In a garbage collected environment, is it also a good idea to return freshly-allocated blocks to the program without clearing them first?

Exercise

The POSIX `malloc` function does not clear the memory it returns to the user program, for performance reasons.

In a garbage collected environment, is it also a good idea to return freshly-allocated blocks to the program without clearing them first?

Typically no, as we would return blocks potentially retaining pointers to already-collected blocks.

GC Technique #2: Mark & Sweep

Summary:

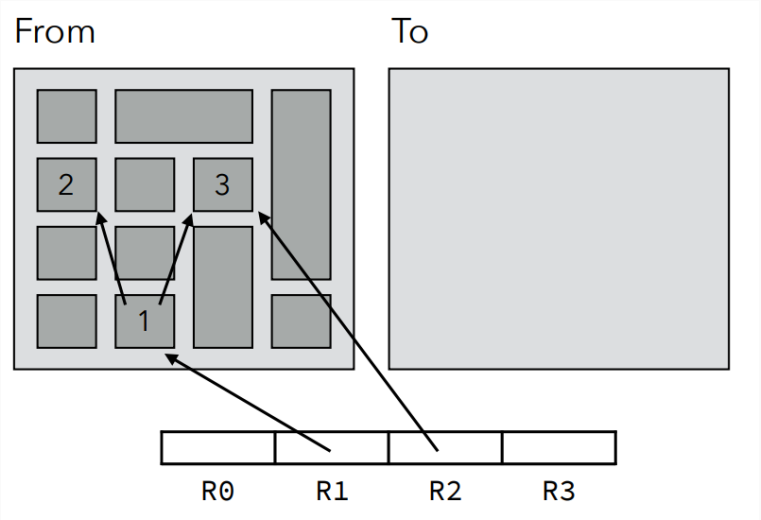
- Use free list(s) to track free blocks.
- Allocation needs to search the free list for a block big enough to satisfy the request.
- GC consists in a mark phase, which traverses the reachability graph and marks reachable objects, followed by a sweep phase, which traverses the heap and reconstructs the free list.

GC Technique #3: Copying GC

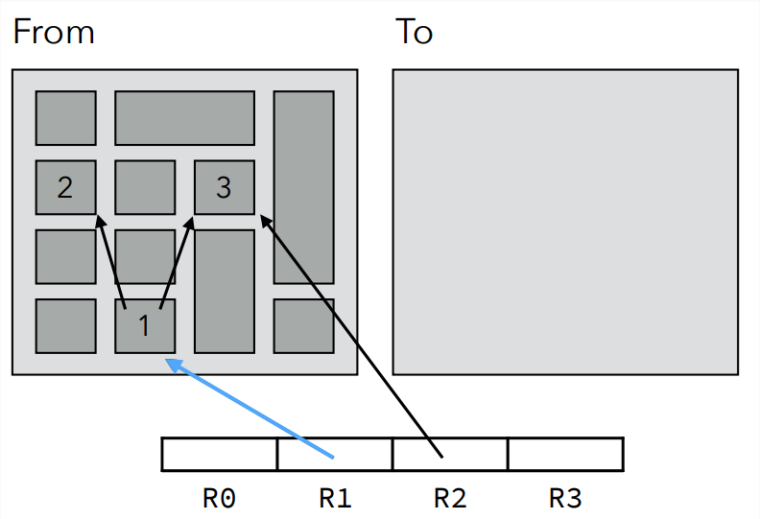
The idea of **copying garbage collection** is to split the heap in two semi-spaces of equal size: the **from-space** and the **to-space**.

- Memory is allocated in from-space, while to-space is left empty.
- When from-space is full, all reachable objects in from-space are copied to to-space, and pointers to them are updated accordingly.
- Finally, the role of the two spaces is exchanged, and the program resumed.

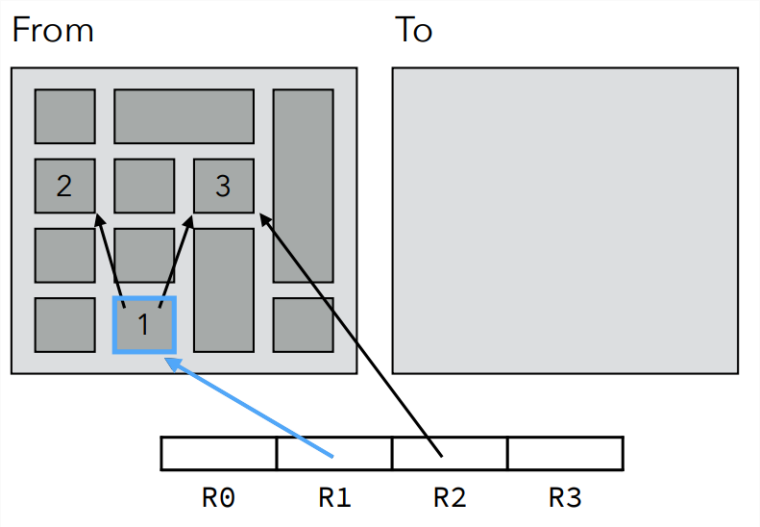
Copying GC



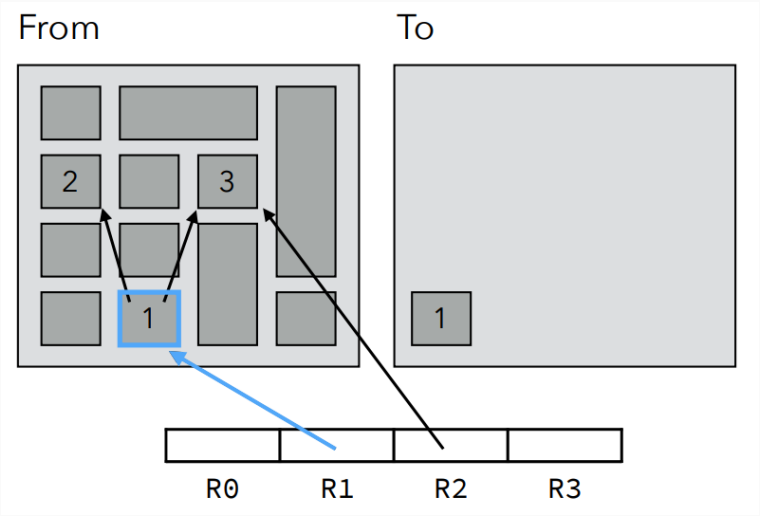
Copying GC



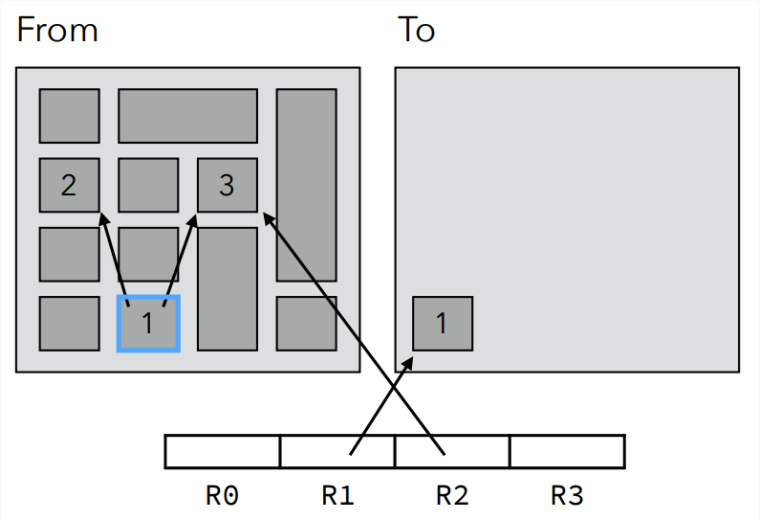
Copying GC



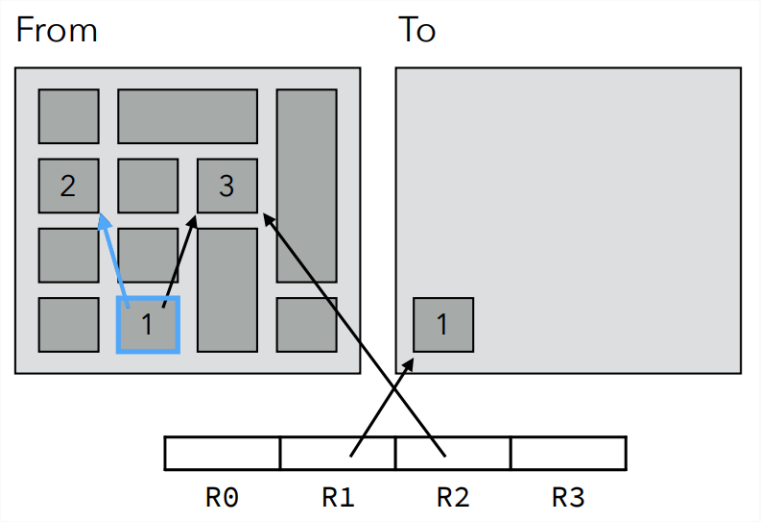
Copying GC



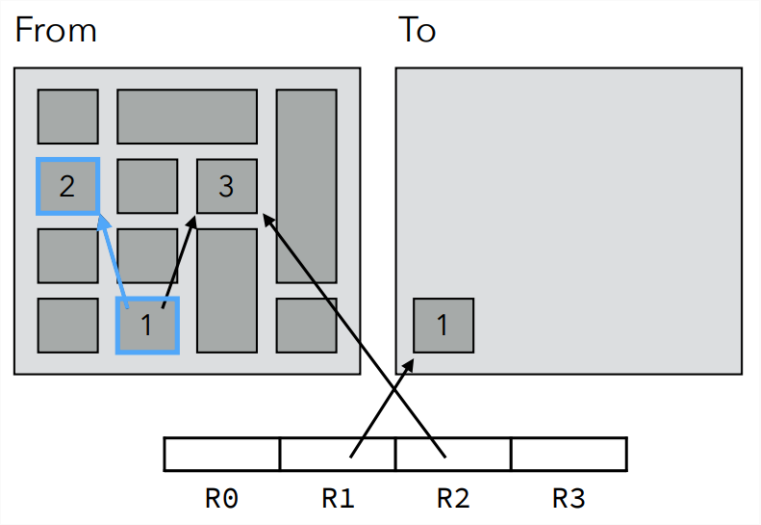
Copying GC



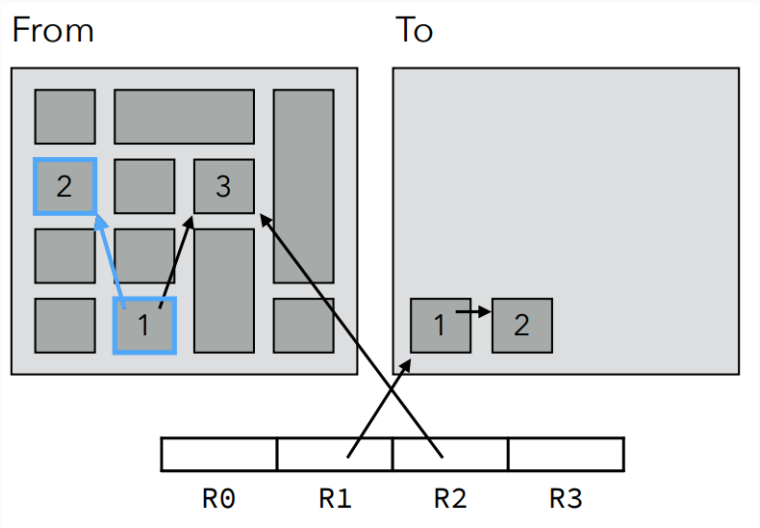
Copying GC



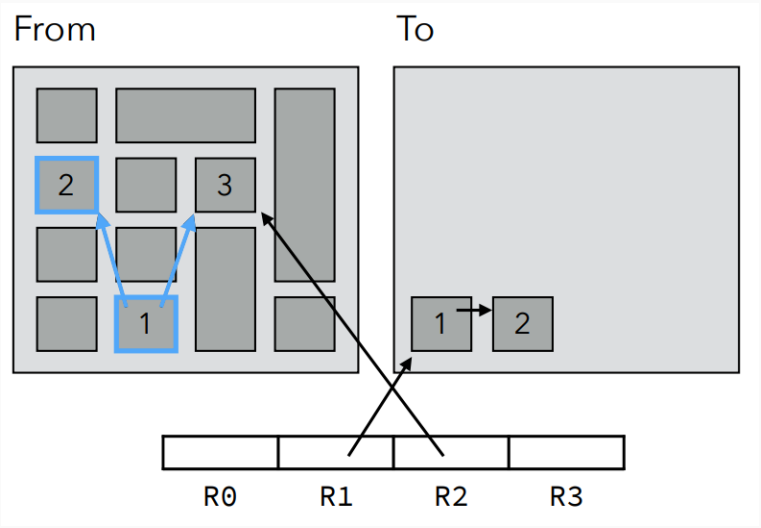
Copying GC



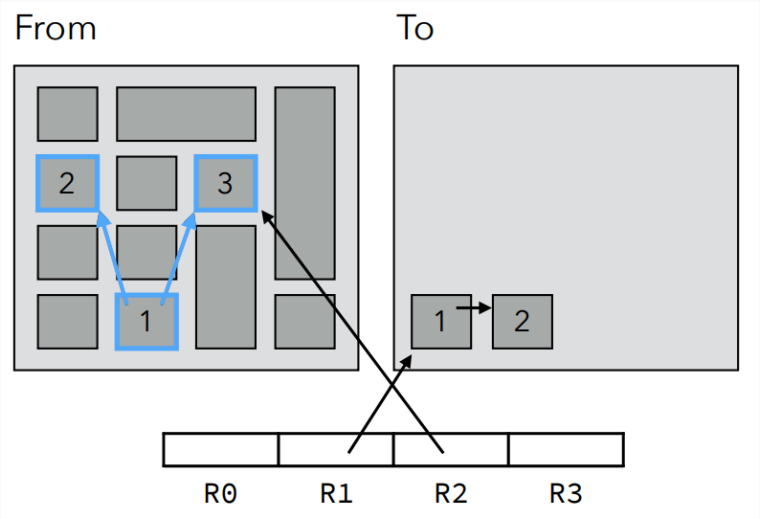
Copying GC



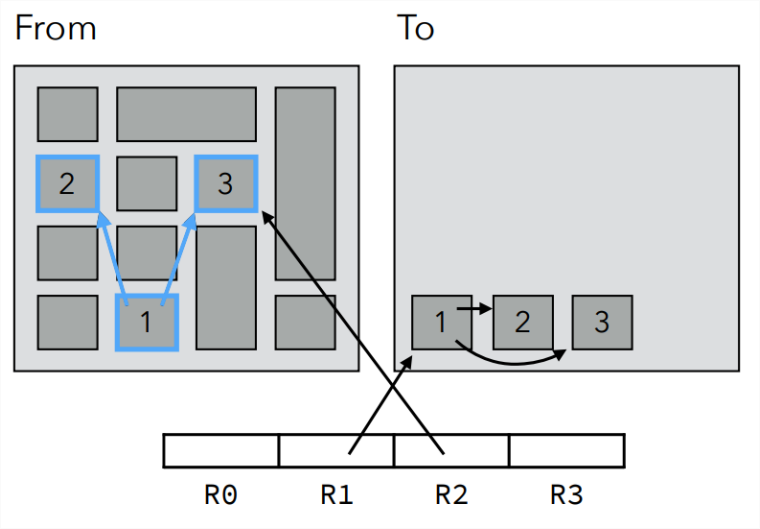
Copying GC



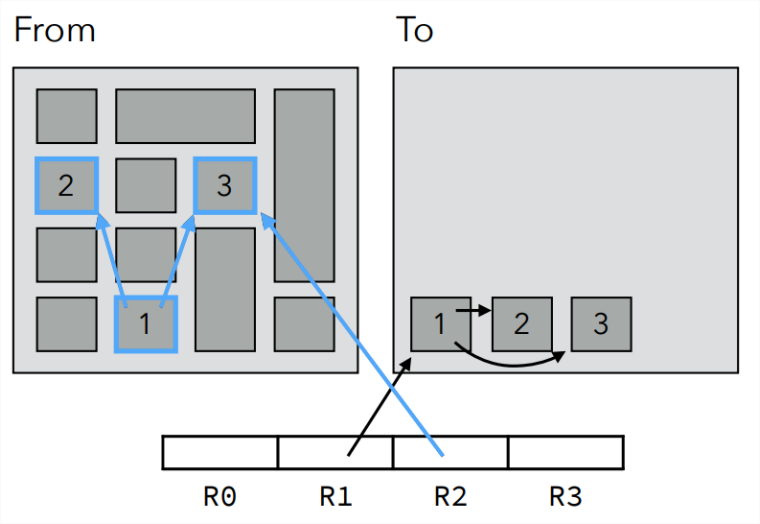
Copying GC



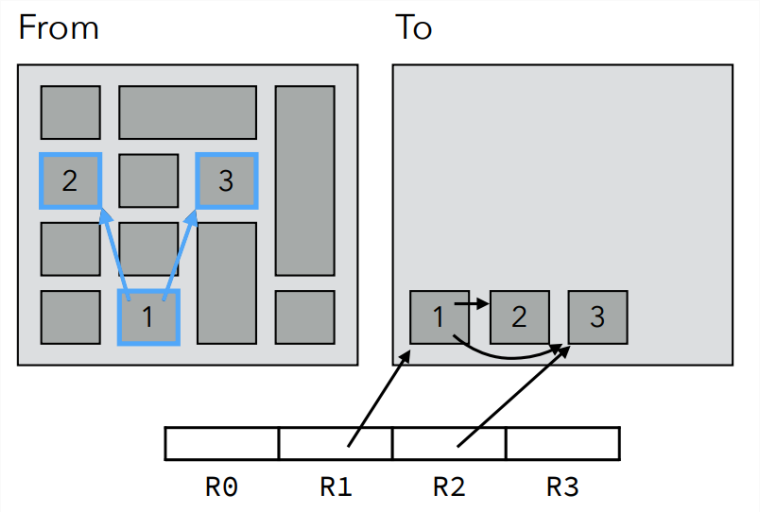
Copying GC



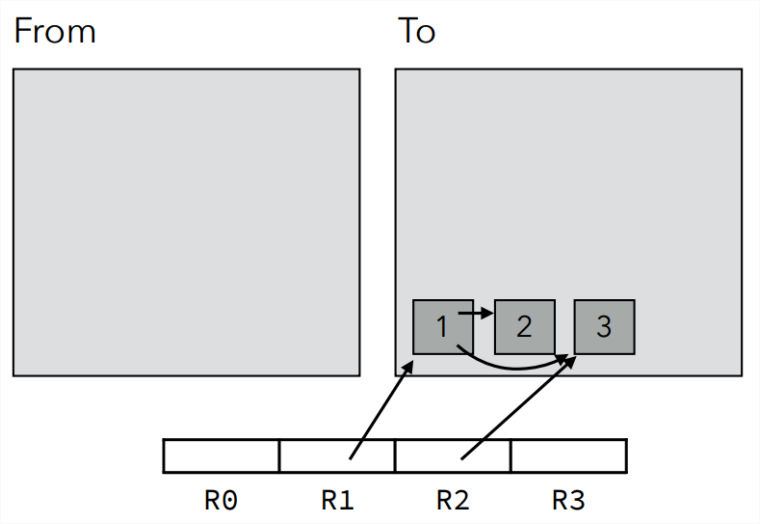
Copying GC



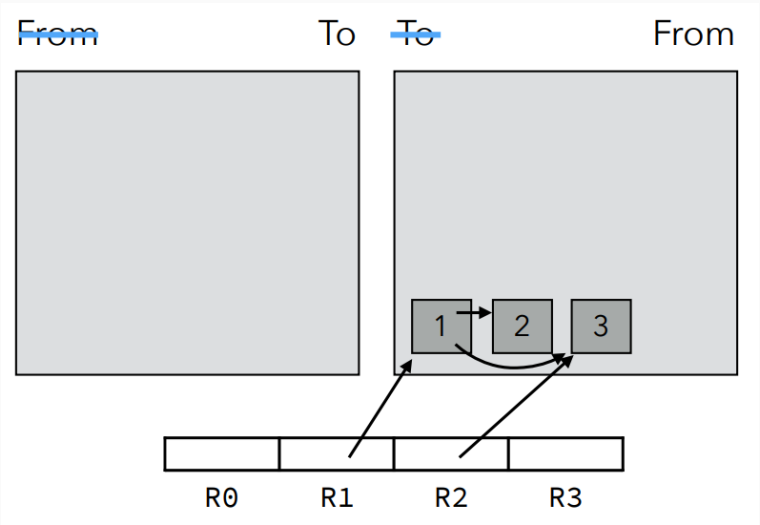
Copying GC



Copying GC



Copying GC



Copying GC

Advantages:

- No fragmentation
- No need for free lists
- Allocation is fast (next lecture)

Disadvantages:

- Need twice the memory
- Copying can be expensive