Memory management
Part I

Michel Schinz (based on Erik Stenman's slides)

Advanced Compiler Construction / 2006-03-31

Why 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 aim of memory management is to use that finite resource as efficiently as possible, according to some criterion.

Memory areas

The memory used by a program can be allocated from three different areas:

- a **static area**, which is laid out at compilation time, and allocated when the program starts,
- a **stack**, from which memory is allocated and freed dynamically, in LIFO order,
- a **heap**, from which memory is allocated and freed dynamically, in any order.

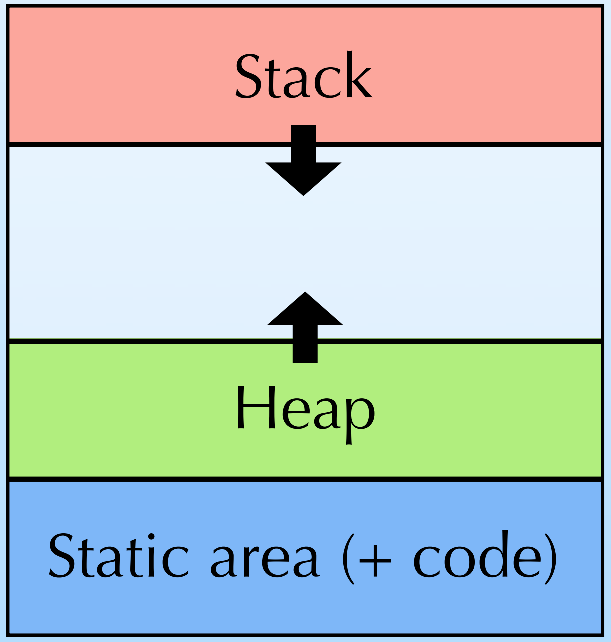
Location of data

Each of the areas presented before is useful to store different kinds of data:

- global variables and constants go into the static area,
- local variables and function arguments go into the stack,
- all data outliving the function which created them go into the heap.

Memory organisation

The three areas described before can be laid out as follows in memory:



Memory management

Managing the static area and the stack is trivial.

Managing the heap is much more difficult because of the irregular lifetimes of the blocks allocated from it.

All the techniques we will see apply exclusively to the management of the heap.

Memory deallocation

Memory deallocation can be either **explicit** or **implicit**.

It is explicit when the language offers a way to declare a memory block as being free – *e.g.* using `delete` in C++ or `free` in C.

It is implicit when the run time system infers that information itself, usually by finding which allocated blocks are not *reachable* anymore.

The dangers of explicit memory deallocation

There are several problems with explicit memory deallocation:

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

The danger of implicit memory deallocation

Implicit memory deallocation is based on the following conservative assumption:

> If a block of memory is still reachable, then it will be used again in the future.

Since this assumption is conservative, it is possible to have space leaks even with implicit memory deallocation – by keeping a reference to a memory block without accessing it anymore.

Management of free memory

The memory management system must keep track of which parts of the heap are free, and which are allocated.

For that purpose, free blocks are stored in a data-structure which can be as simple as a linked list. We will call that data-structure the **free list** even though it is technically not always a list.

Allocation and deallocation

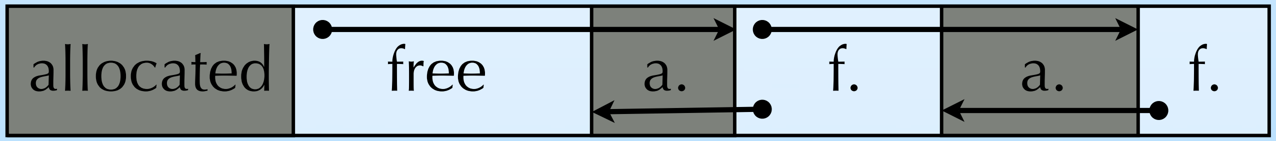
The aim of allocation is to find a free block big enough to satisfy the request, and possibly **split** it in two if it is too big: one part is then returned as the result of the allocation, while the other is put back in the free list.

On deallocation, adjacent free blocks can be **coalesced** to form bigger free blocks.

Free list encoding

Since free blocks are not used by the program, they can be used to store the data required to encode the free list – *e.g.* links to successors and predecessors.

This implies that the smallest possible free block must be big enough to contain that information.



Header field

Allocated blocks are not linked in the free list, and hence must not hold any kind of link.

However, the size of all blocks, allocated or not, must be stored in them: it is required both during allocation and deallocation.

This size is stored in a **header field** at the beginning of the block. This header word is also used for garbage collection.

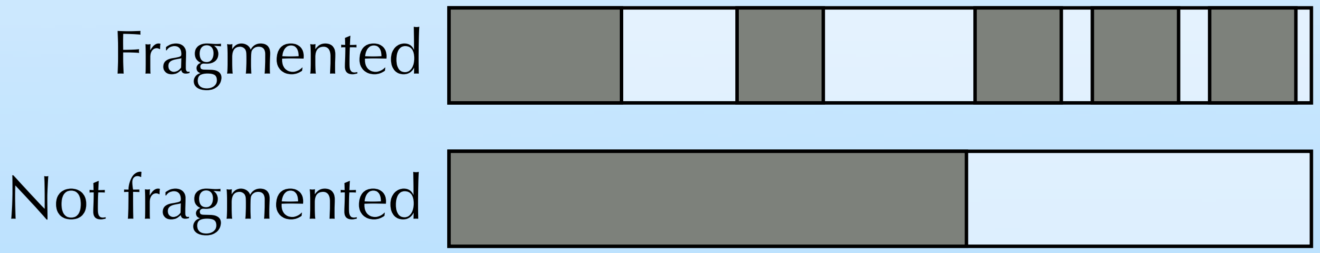
Fragmentation

The term **fragmentation** is used to designate two different – but similar – problems associated with memory management:

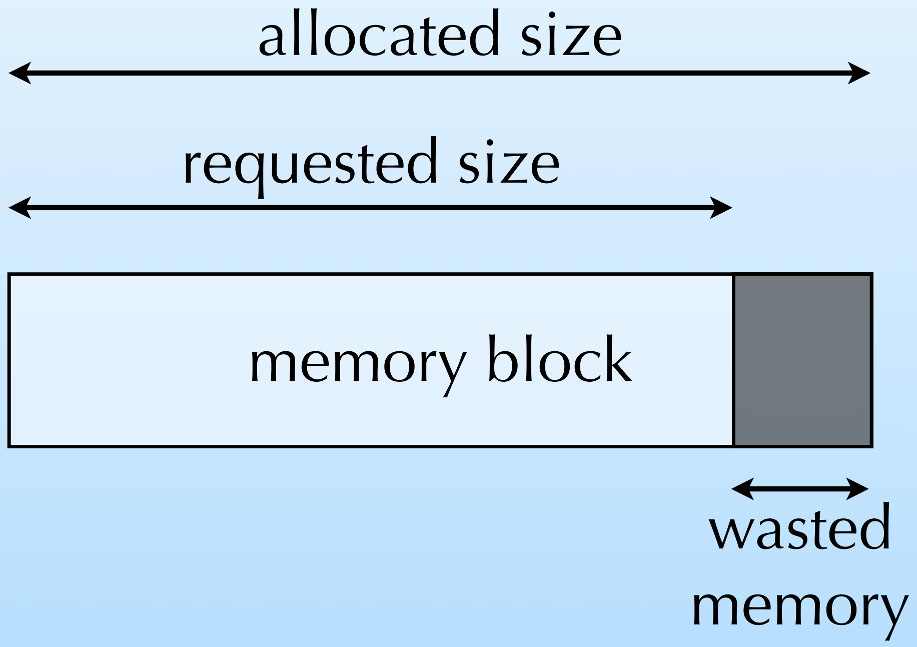
- **external fragmentation** refers to the fragmentation of free memory in many small blocks,
- **internal fragmentation** refers to the waste of memory due to the use of a free block larger than required to satisfy an allocation request.

External fragmentation

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



Internal fragmentation



Allocation policies

Whenever a block of memory is requested, there will in general be several free blocks big enough to satisfy the request.

A policy must therefore be used to decide which of those candidates to choose.

There are several such policies: first fit, next fit, best fit, worst fit, etc.

First and next fit

First fit chooses the first block in the free list big enough to satisfy the request, and split it.

Next fit is like first fit, except that the search for a fitting block will start where the last one stopped, instead of at the beginning of the free list.

It appears that next fit results in significantly more fragmentation than first fit, as it mixes blocks allocated at very different times.

Best and worst fit

Best fit chooses the smallest block bigger than the requested one.

Worst fit chooses the biggest, with the aim of avoiding the creation of too many small fragments – but doesn't work well in practice.

The major problem of these techniques is that they require an exhaustive search of the free list, unless segregation techniques are used.

Segregated free lists

Instead of having a single free list, it is possible to have several of them, each holding free blocks of (approximately) the same size.

These **segregated free lists** are organised in an array, to quickly find the appropriate free list given a block size.

When a given free list is empty, blocks from “bigger” lists are split in order to repopulate it.

Buddy system

Buddy systems are a variant of segregated free lists.

The heap is viewed as one large block which can be split in two smaller blocks, called buddies, of a given size. Those smaller blocks can again be split in two smaller buddies, and so on.

Coalescing is fast in such a system, since a block can only be coalesced with its buddy, provided it is free too.

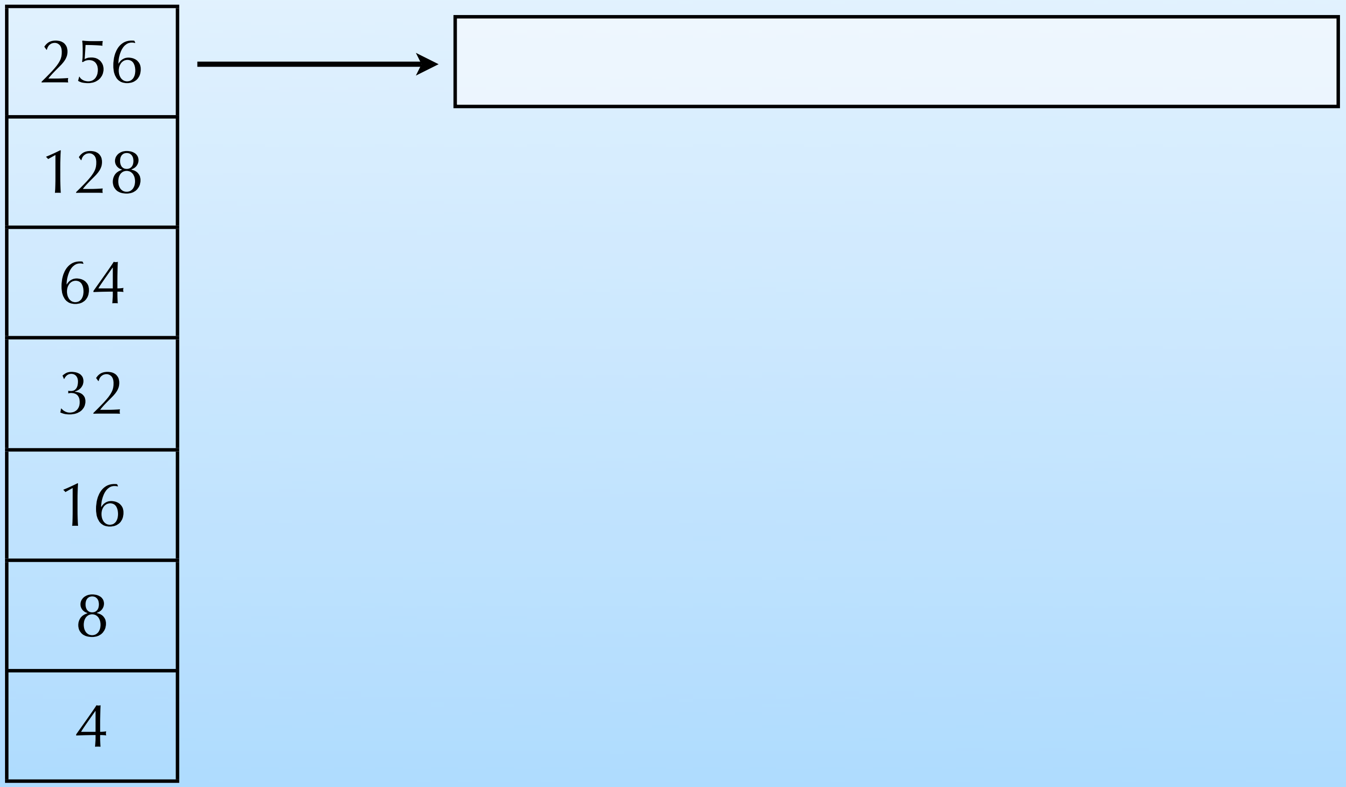
Kinds of buddy systems

Examples of buddy systems:

- In a **binary buddy system** – the most common kind – the blocks of a given free list are twice as big as those in the previous free list.
- In a **Fibonacci buddy system**, the size of the blocks of successive free lists forms a Fibonacci sequence ($s_n = s_{n-1} + s_{n-2}$).

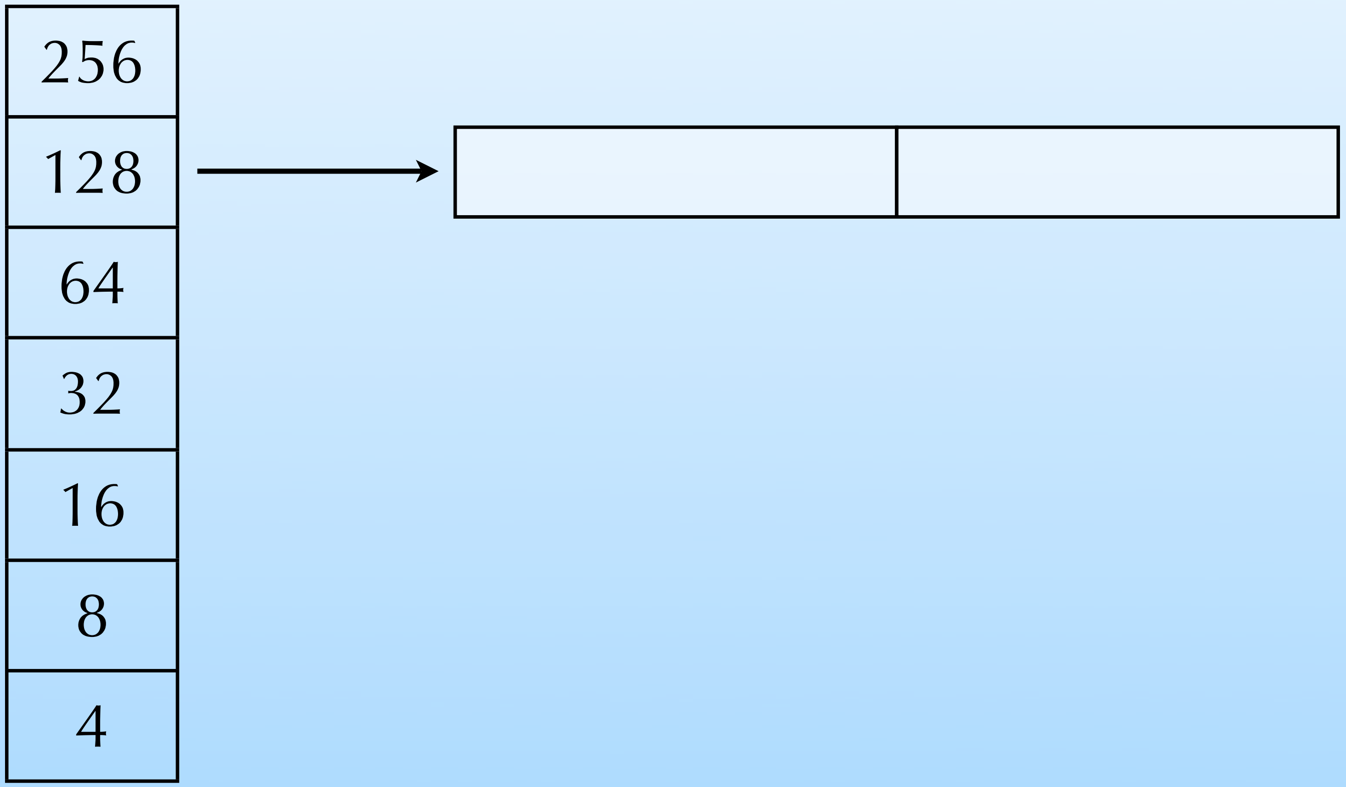
Binary buddy system example

Allocation of a 10 bytes block.



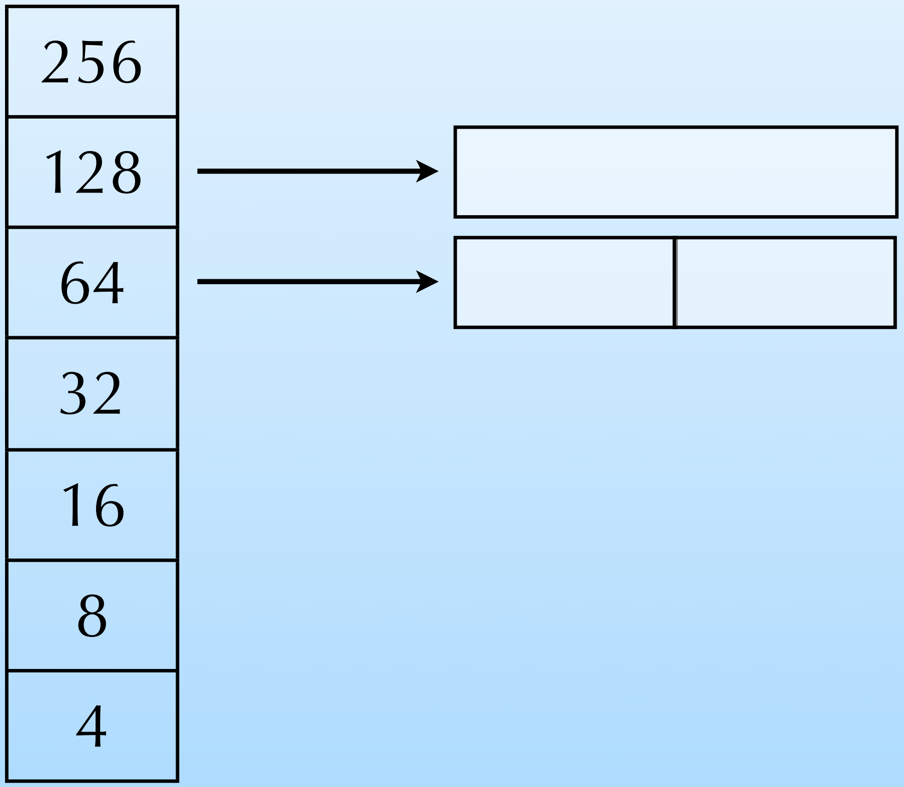
Binary buddy system example

Allocation of a 10 bytes block.



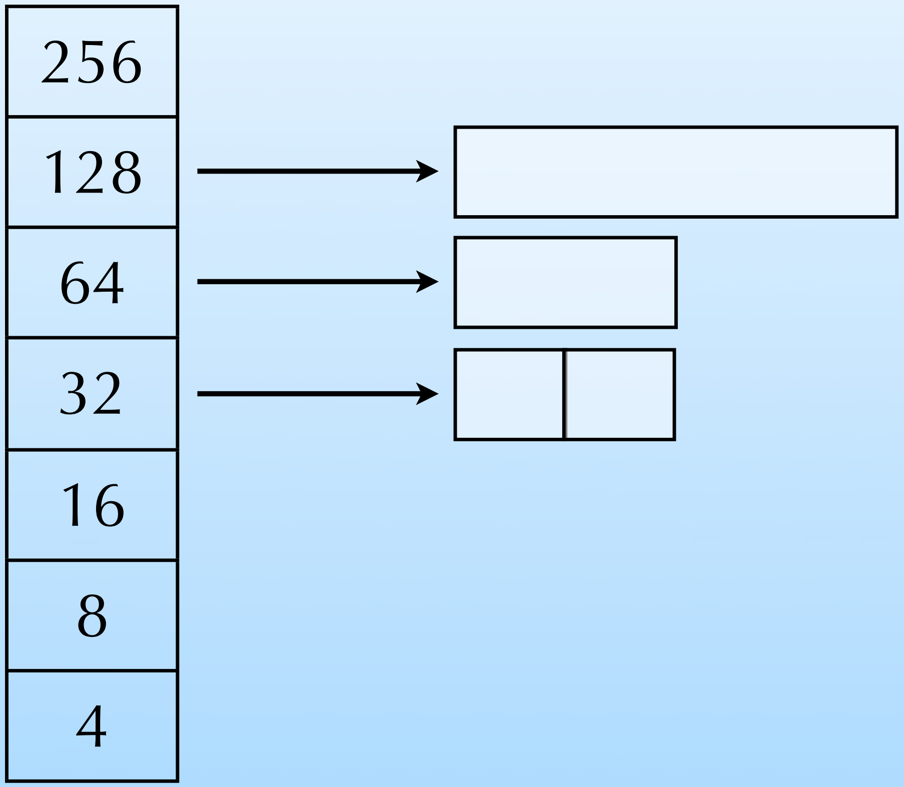
Binary buddy system example

Allocation of a 10 bytes block.



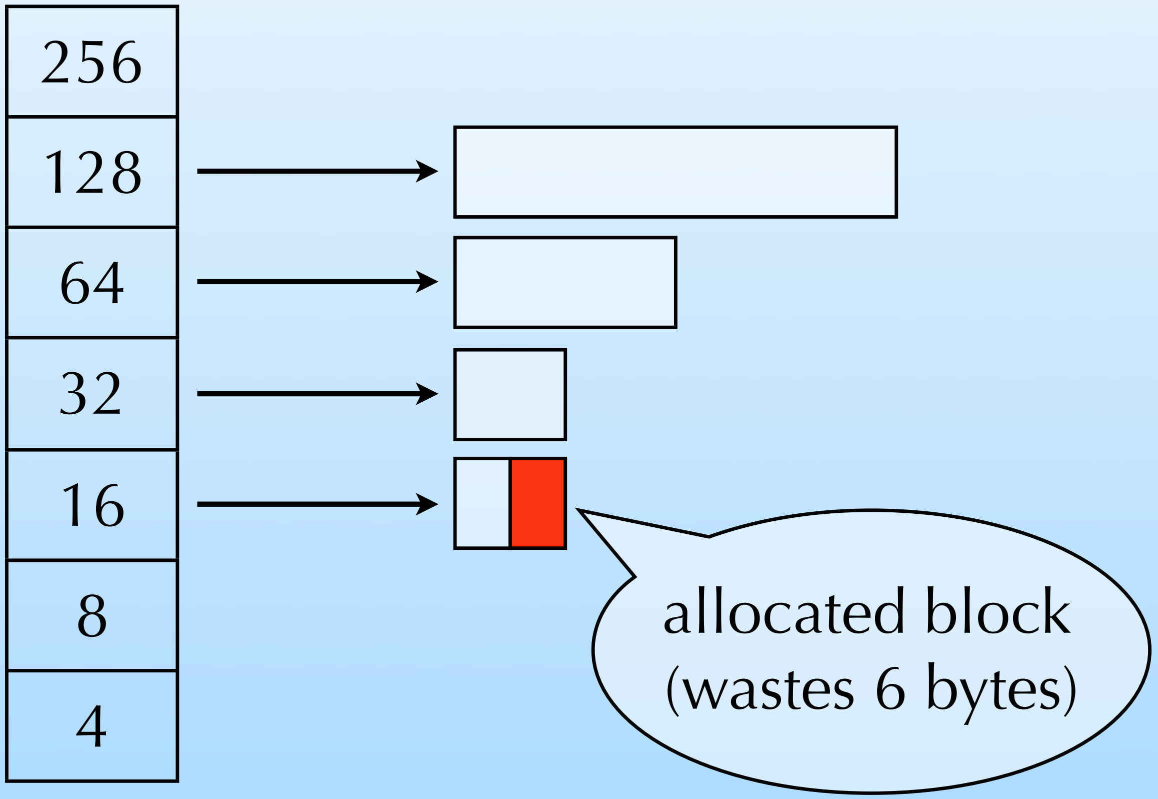
Binary buddy system example

Allocation of a 10 bytes block.



Binary buddy system example

Allocation of a 10 bytes block.



Automatic memory management

The (unattainable) goal of automatic memory management is to automatically deallocate dead objects.

Dead objects are those which will not be accessed anymore in the future. Objects which are not dead are said to be **live**.

Since liveness is undecidable in general, reachability (to be defined) is used as a conservative approximation.

The reachability graph

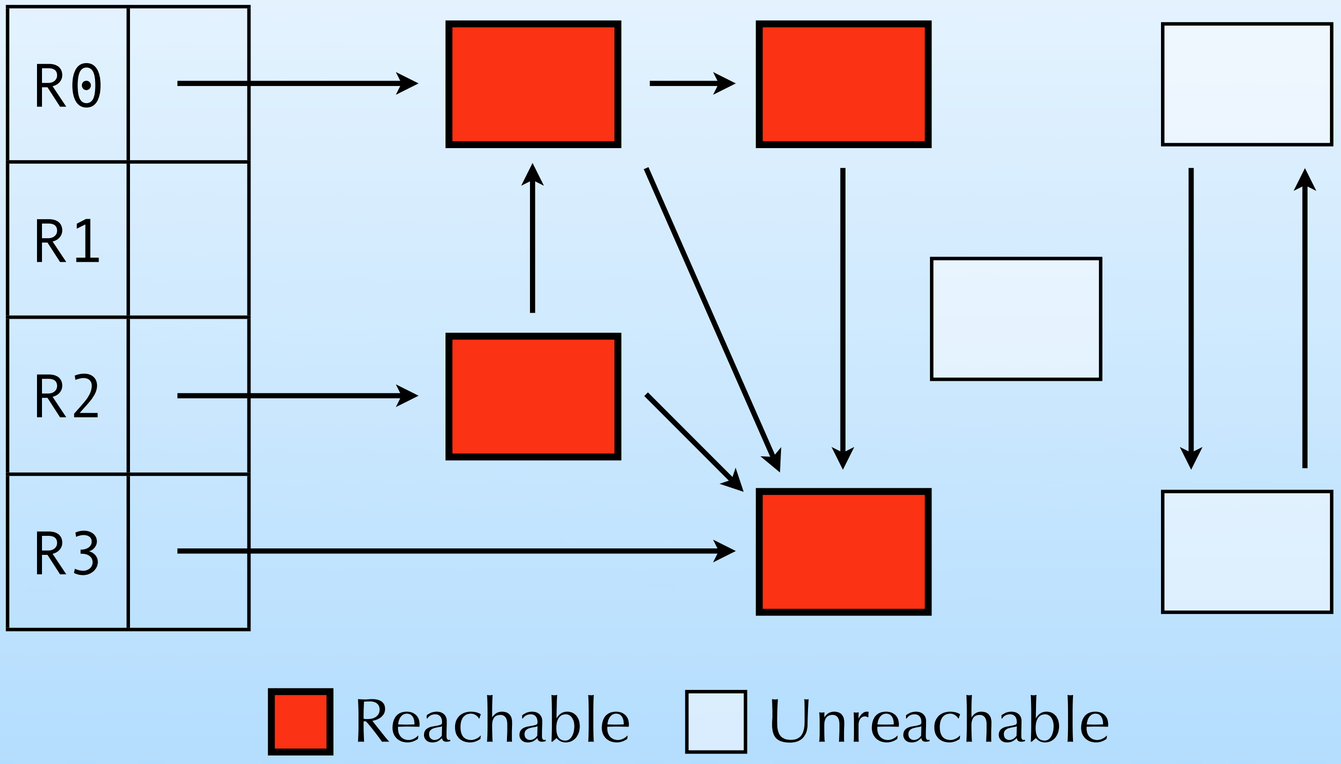
At any time during the execution of a program, we can define the set of **reachable** objects as being:

roots

- the objects immediately accessible from global variables, the stack or registers,
- the objects which are reachable from other reachable objects, by following pointers.

This forms the **reachability graph**.

Reachability graph example



Garbage collection

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

We will examine several garbage collection techniques: reference counting, mark & sweep GC and copying GC.

Reference counting

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

> Every object carries a count of the number of pointers which reference it. When this count is zero, the object is unreachable and can be deallocated.

Reference counting requires collaboration from the compiler – or the programmer – to make sure that reference counts are properly maintained.

Reference counting pros and cons

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

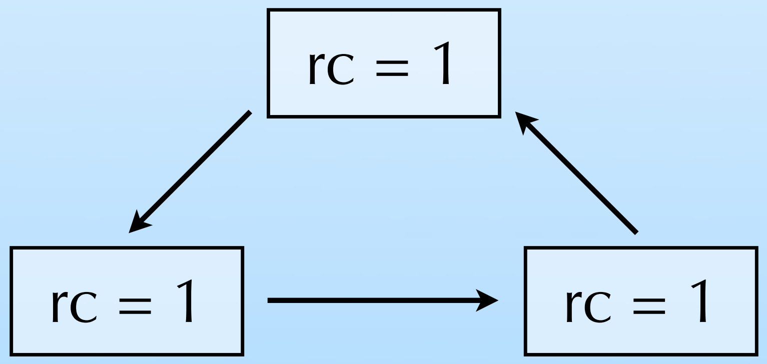
However, it has an important impact on space consumption, and speed of execution: every object must contain a counter, and every pointer write must update it.

But the biggest problem is cyclic structures...

Reference count of cyclic structures

The reference count of objects which 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 and reference counting

The problem with cyclic structures is due to the fact that reference counts do *not* compute reachability, but a weaker approximation.

In other words, we have:

$$\text{reference\_count}(x) = 0 \;\Rightarrow\; x \text{ is unreachable}$$

but not the other way around.

Uses of reference counting

Due to its problem with cyclic structures, reference counting is seldom used.

It is still interesting for systems which do not allow cyclic structures to be created (*e.g.* hard links on Unix file systems).

It has also been used in combination with a mark & sweep GC, the latter being run infrequently to collect cyclic structures.

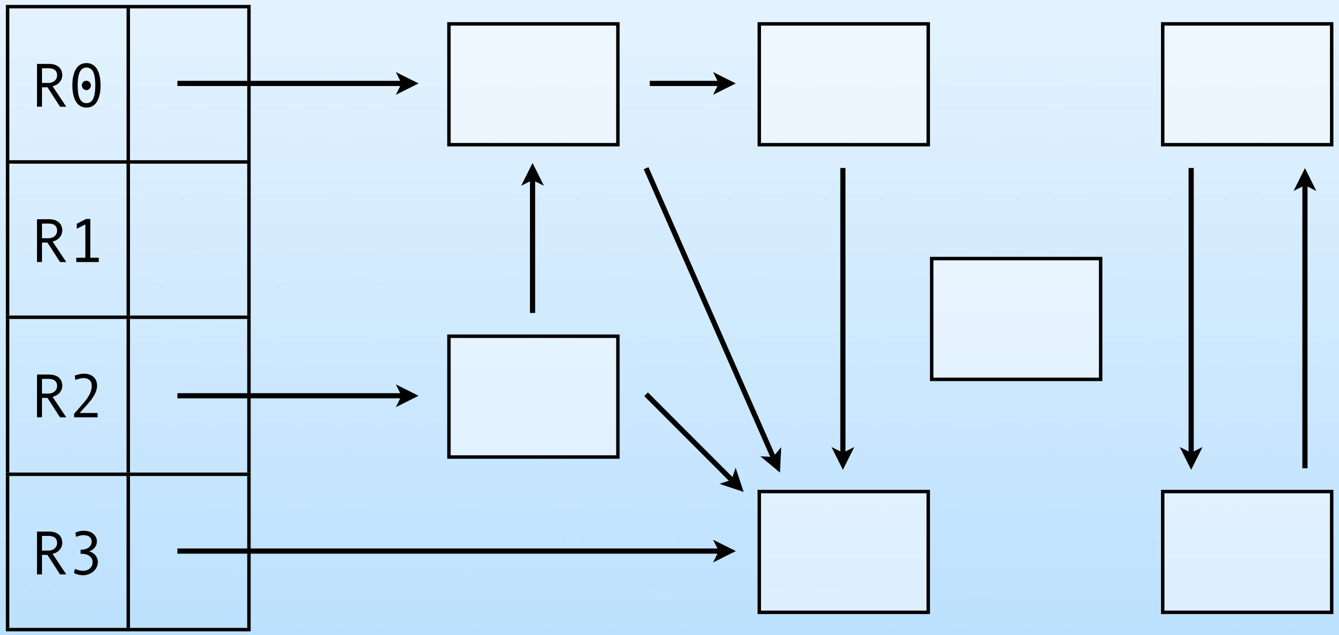
Mark & sweep GC

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

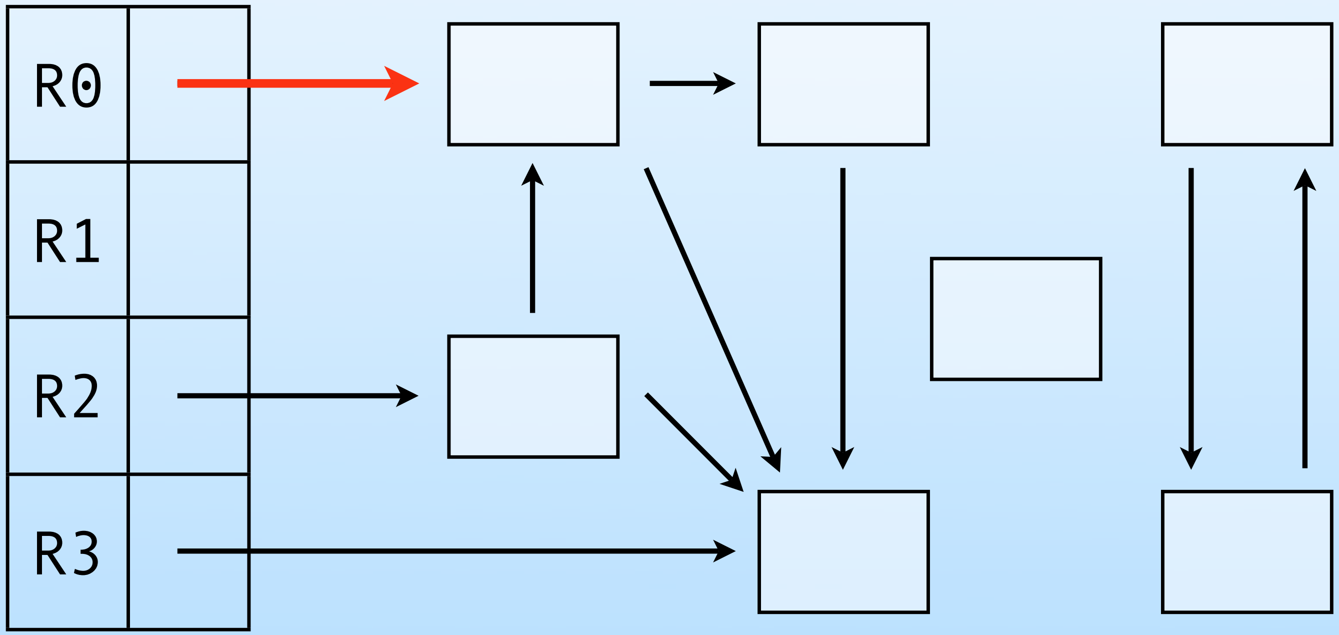
- in the marking phase, the reachability graph is traversed and reachable objects are marked,
- in the sweeping phase, all allocated objects are examined, and unmarked ones are freed.

GC is triggered by a lack of memory, and *must* complete before the program can be resumed.

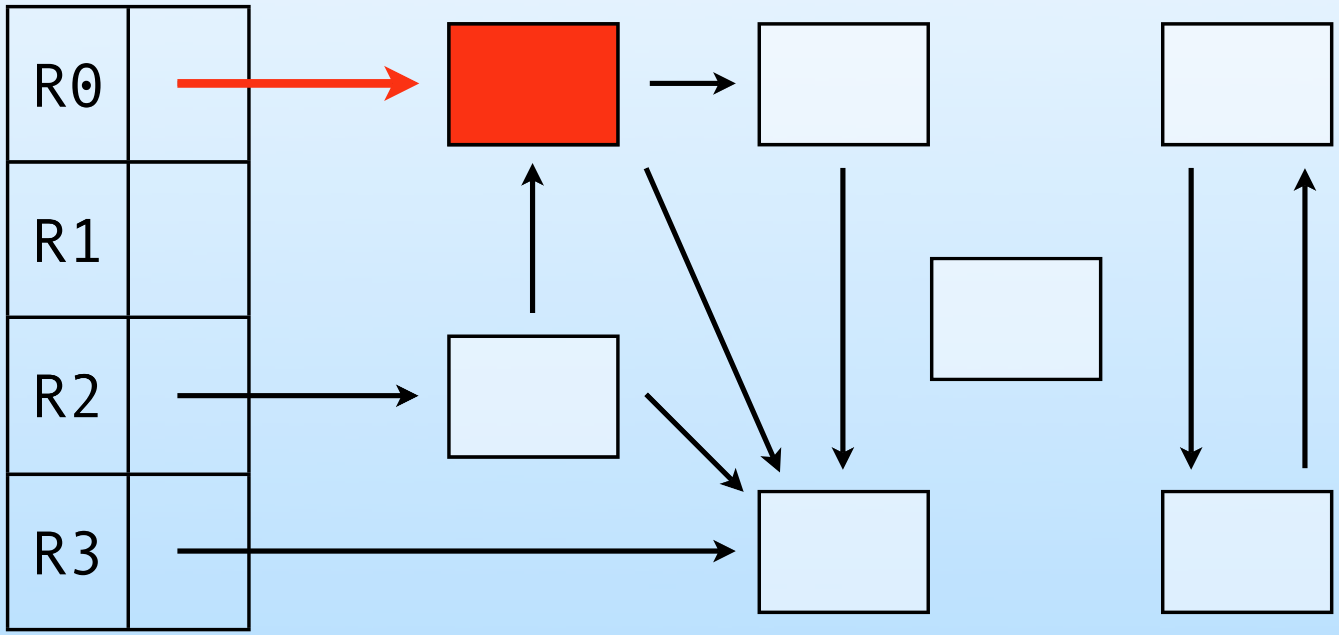
Mark & sweep GC



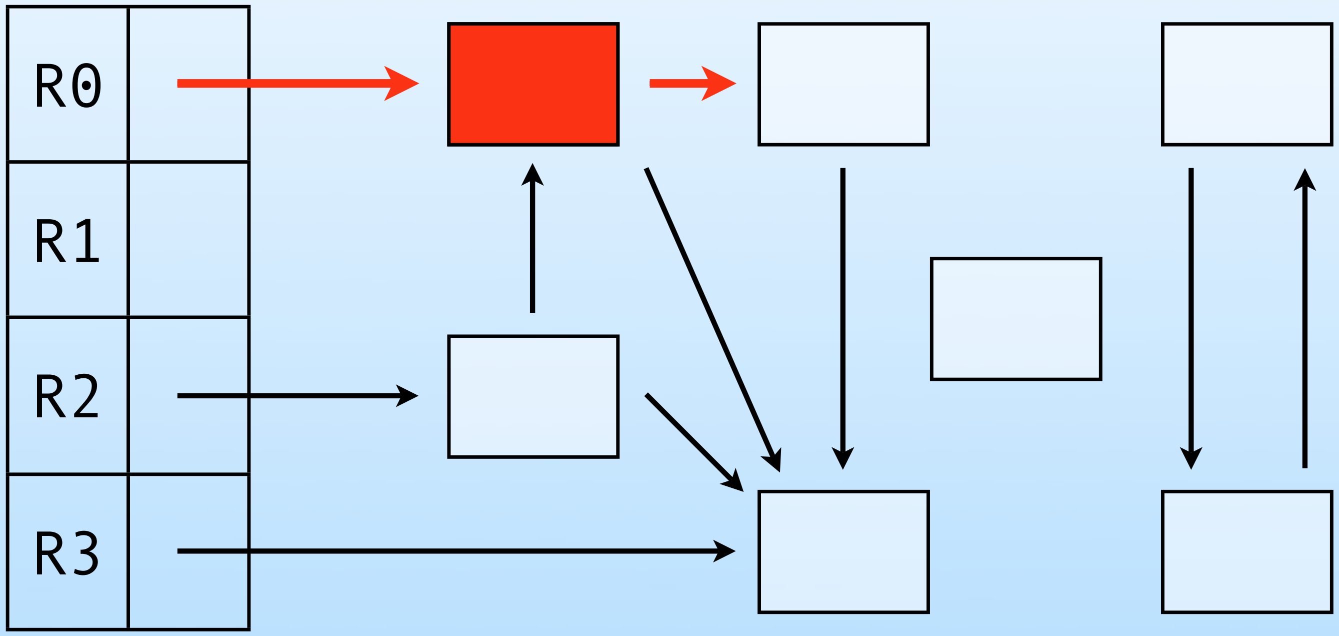
Mark & sweep GC



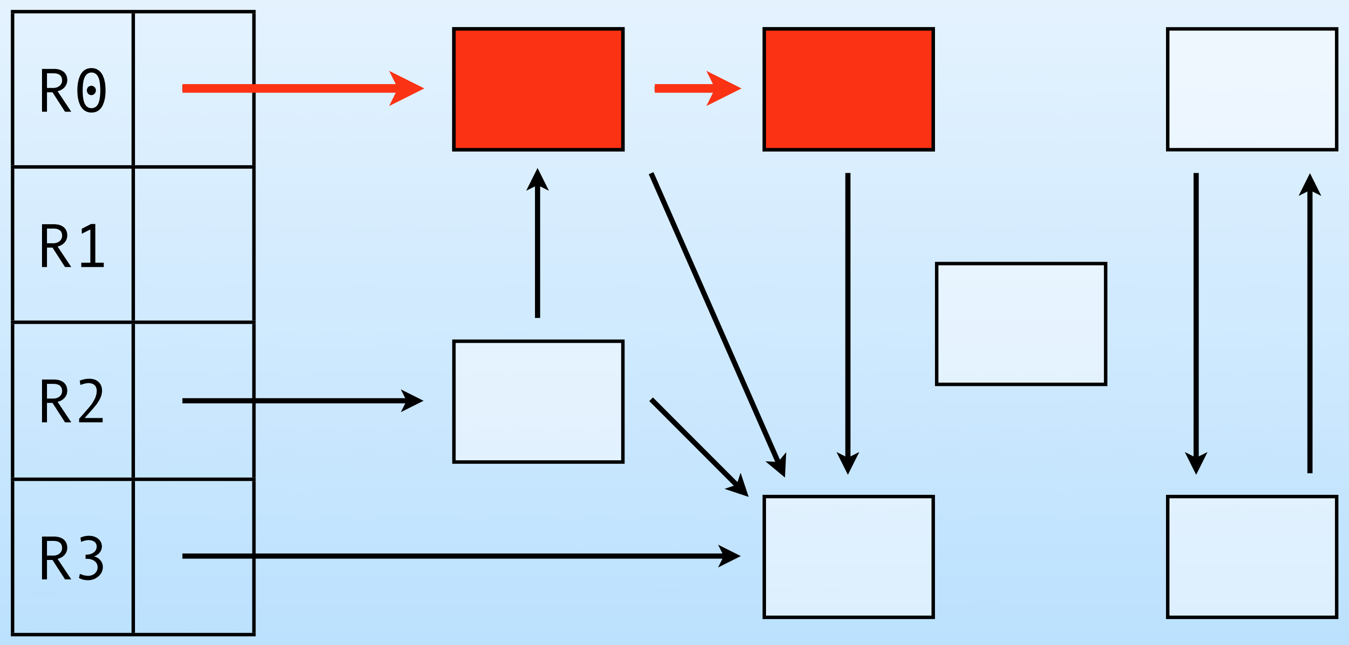
Mark & sweep GC



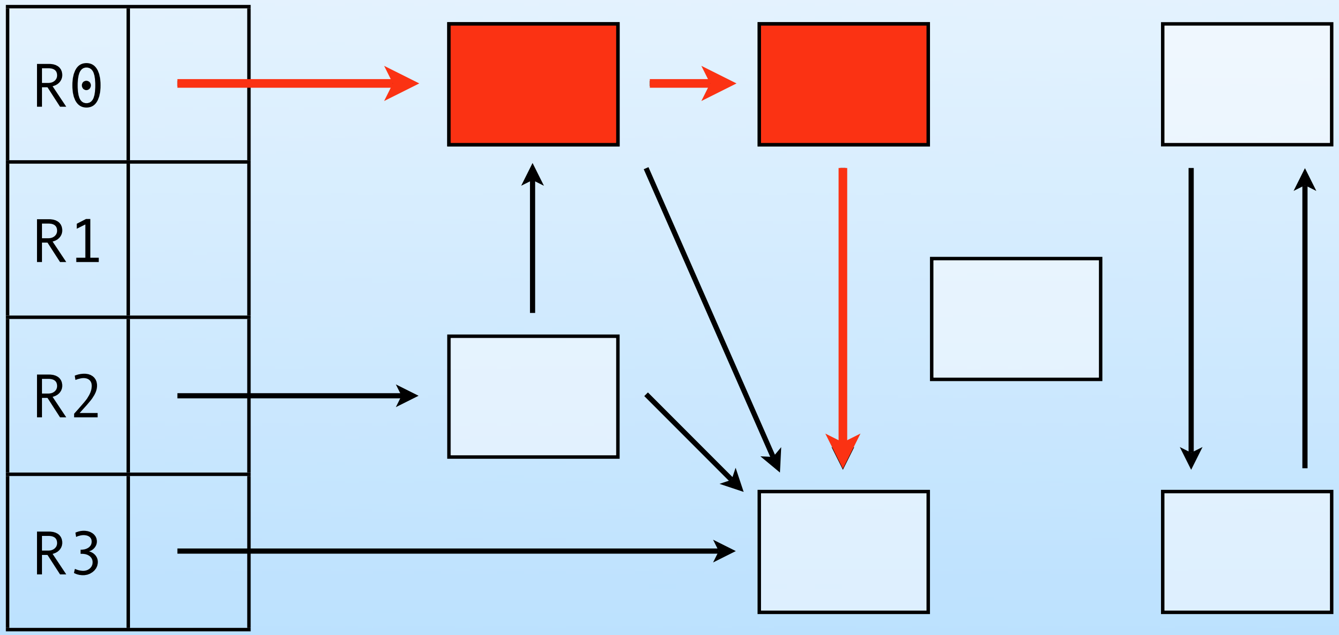
Mark & sweep GC



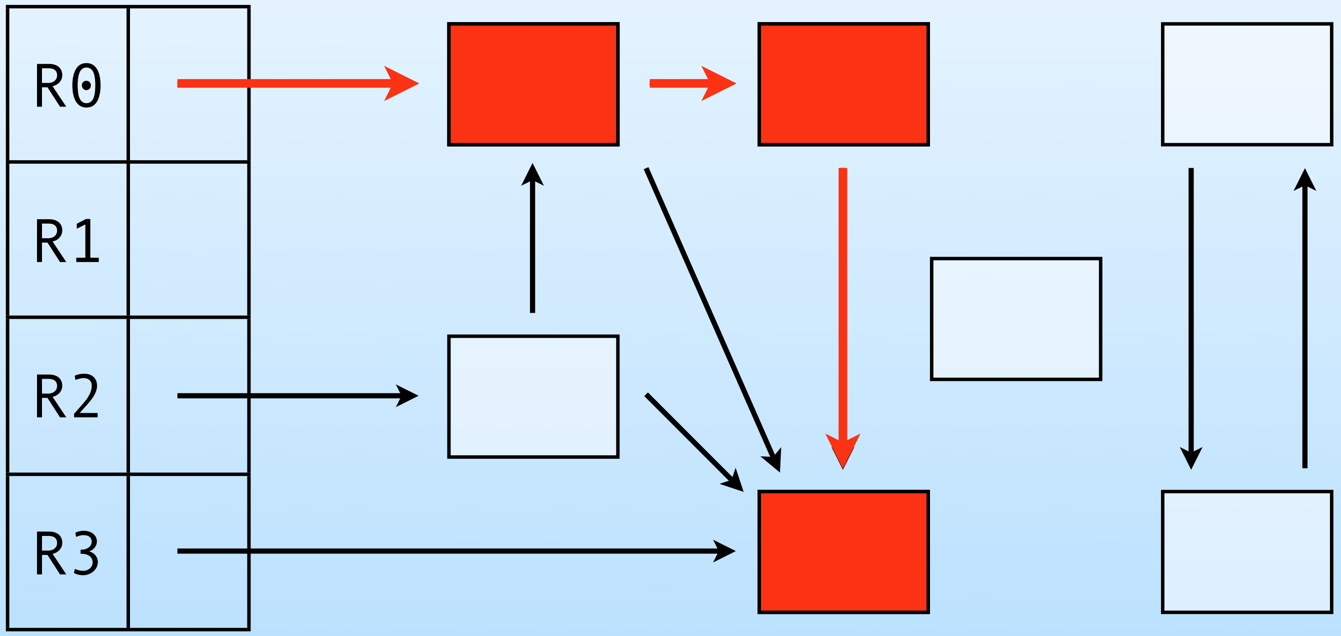
Mark & sweep GC



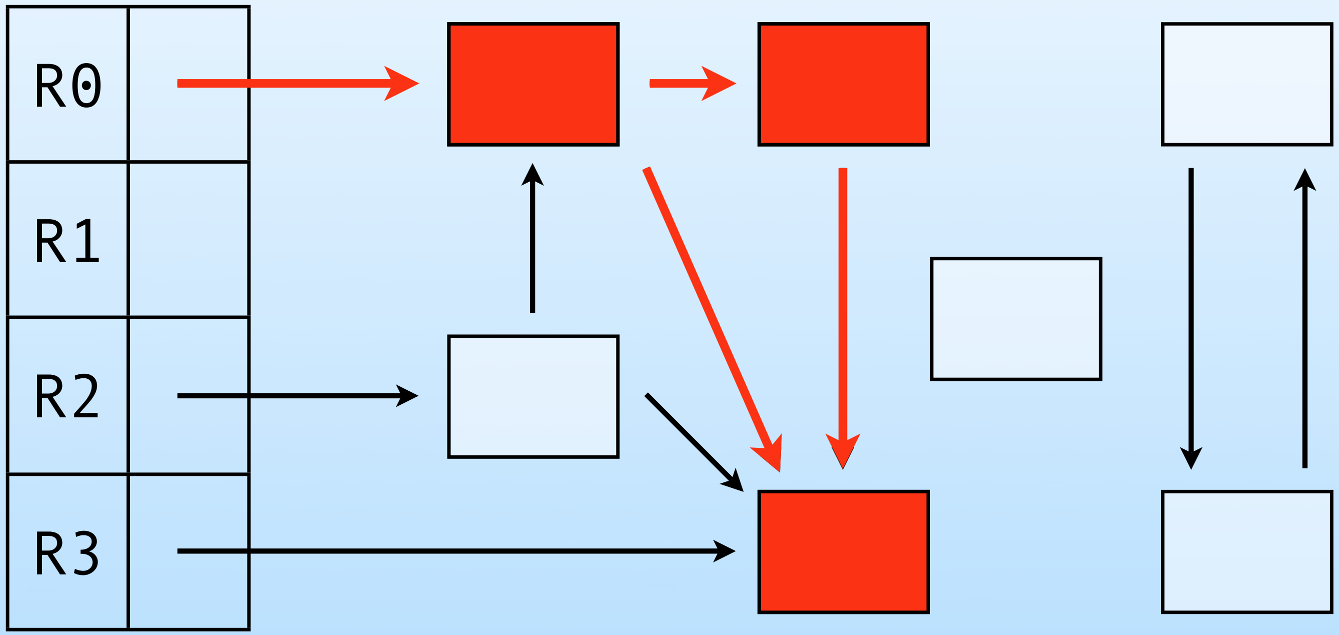
Mark & sweep GC



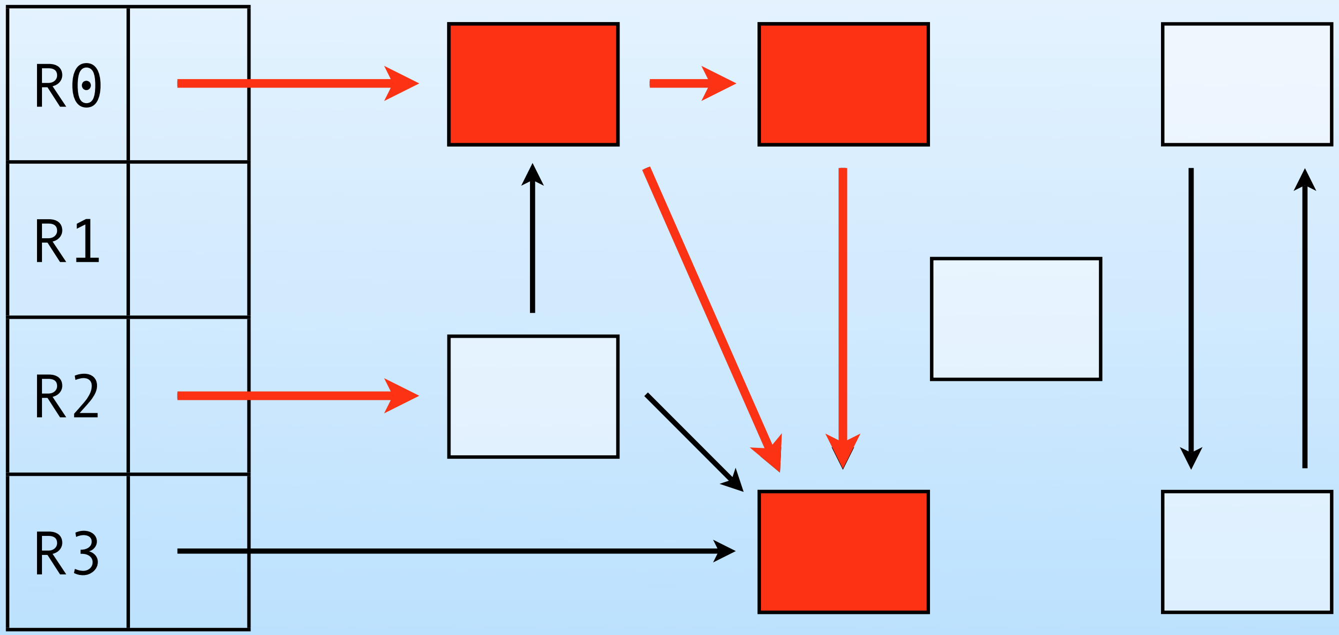
Mark & sweep GC



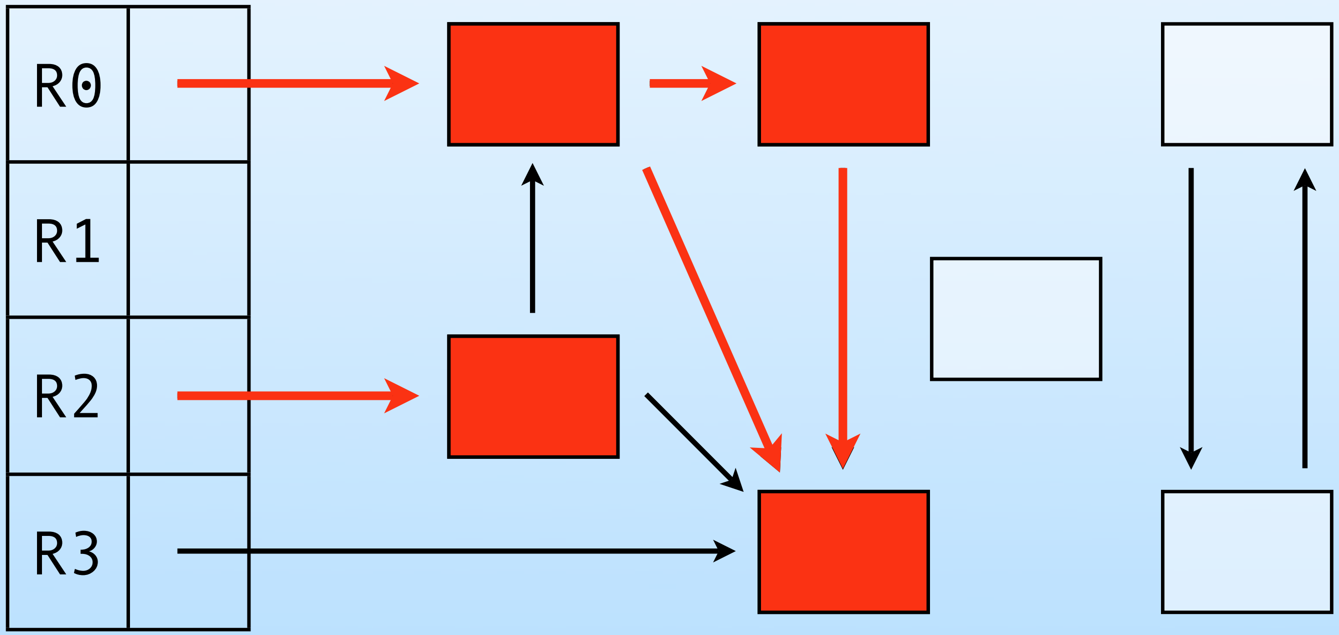
Mark & sweep GC



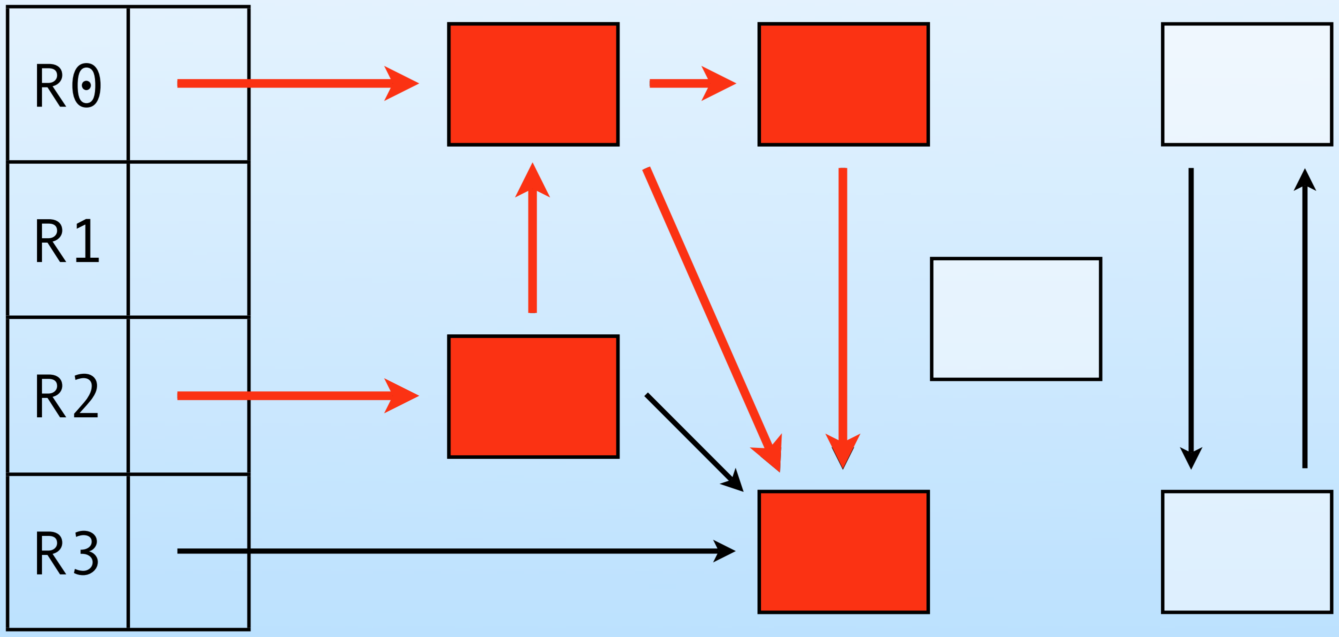
Mark & sweep GC



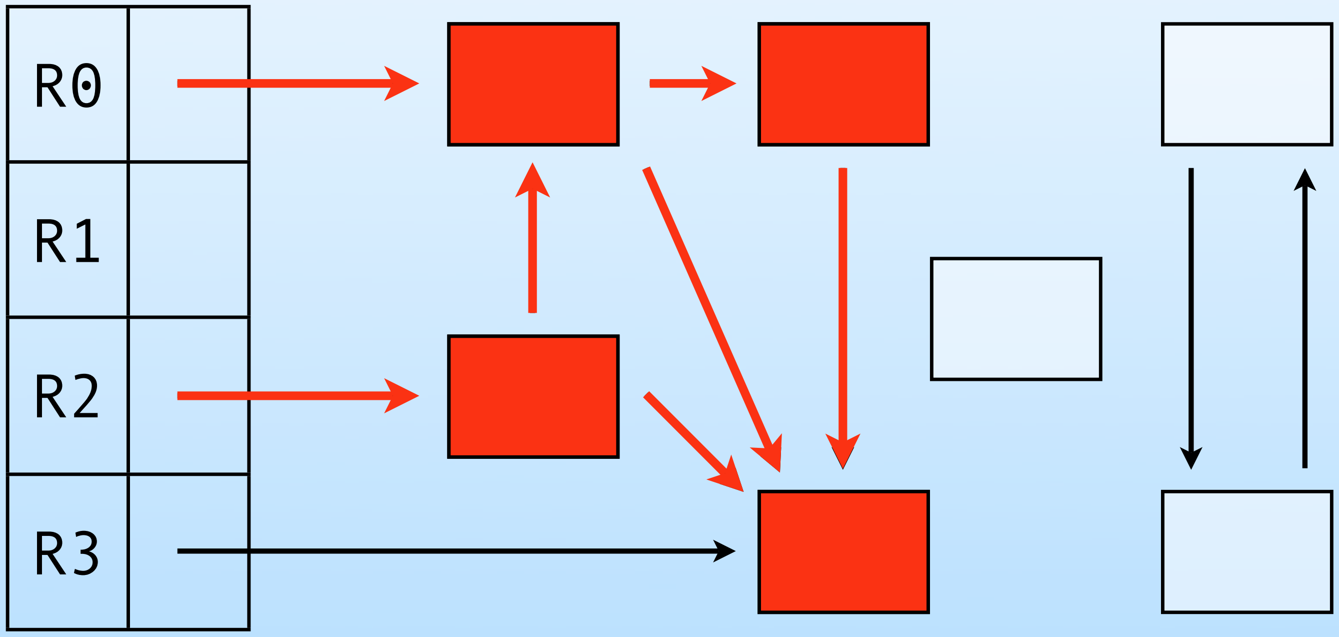
Mark & sweep GC



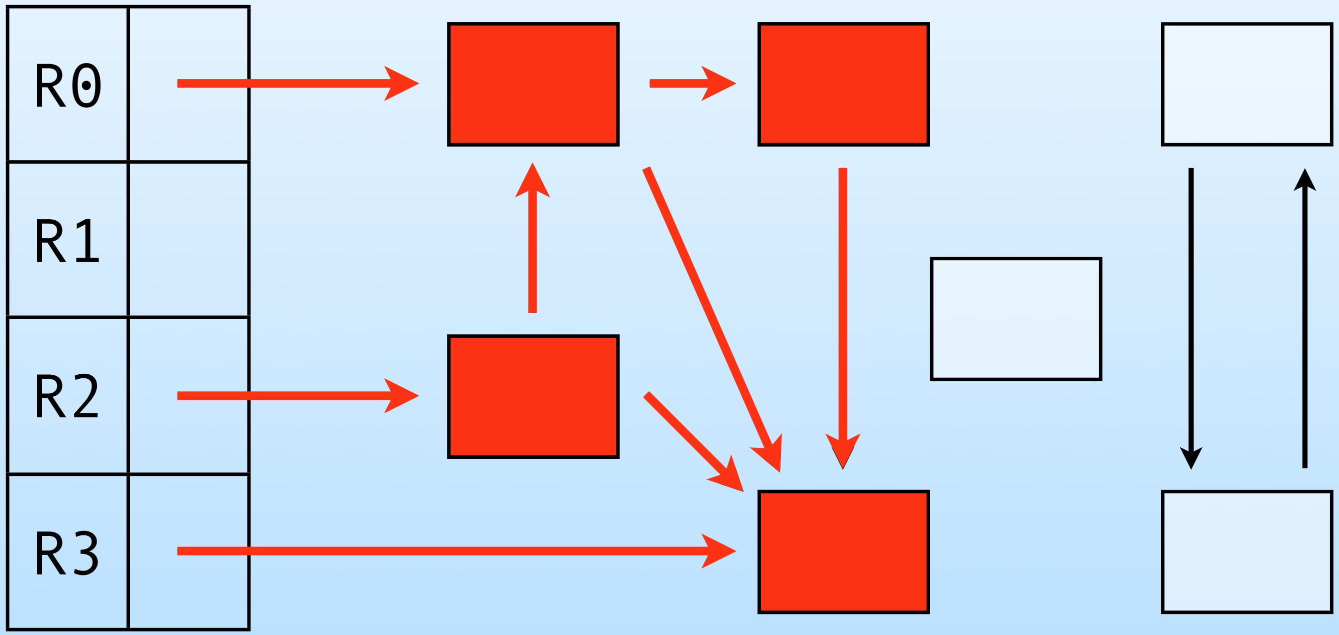
Mark & sweep GC



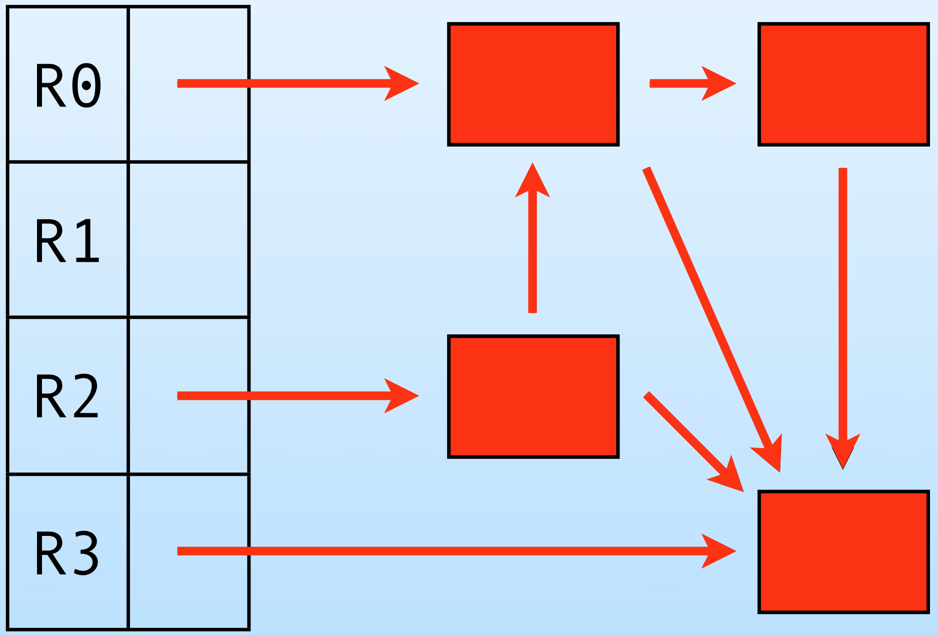
Mark & sweep GC



Mark & sweep GC



Mark & sweep GC



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 header word, along with the size.

It is also possible to use “external” bit maps to store mark bits.

Reachable graph traversal

The mark phase requires a depth-first traversal of the reachable graph. This is usually implemented by recursion.

Recursive function calls use stack space, and since the depth of the reachable graph is not bounded, the GC can overflow its stack!

Several techniques have been developed to either recover from those overflows, or avoid them by storing the stack in the objects being traced.

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 moving them to the free list.

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

Finding reachable objects

Until now, we have assumed that the reachability graph can be computed by the GC.

This is a strong assumption: the GC must be able to identify at run time *all* pointers found in the root set, and in allocated objects. Clearly, this requires collaboration from the compiler.

When the compiler does not – or cannot, due to language characteristics – collaborate, the GC must conservatively approximate reachability.

Identifying the root set

To identify the root set, the GC must know which registers and stack locations contain live pointers.

To enable that identification, the compiler emits **pointer maps**, which describe the location of live pointers at every point where a GC can potentially be triggered (*e.g.* allocation, function call).

Identifying pointers

To locate pointers appearing inside of objects, several techniques can be used:

- if the type of the object can be extracted from it (often the case in OO languages), then the location of pointers can be found that way,
- a **tagging scheme** can be used, to distinguish pointers from other values like integers.

Tagging

If the allocator makes sure that all objects are allocated at 2^m bytes boundaries, then the lowest m bits of all pointers will be zero.

If the system moreover ensures that integers always have a lowest bit of one, by representing n as $2n+1$, then it becomes possible to distinguish integers from pointers by looking at the low bit.

This technique is called **tagging**.

Mark & sweep pros and cons

Mark & sweep GC is better than reference counting in that it reclaims circular structures. It is also relatively easy to implement.

Its main disadvantages result from the fact that memory is not compacted after collection, hence:

- fragmentation can be a problem,
- allocation is “slow” – at least compared with the copying GCs we will examine later.

Cost of mark & sweep

The mark phase takes time proportional to the amount of reachable data R. The sweep phase takes time proportional to the heap size H. This is done to recover $H - R$ words of memory.

Therefore, the amortised cost of mark & sweep GC is: $(c_1 R + c_2 H) / (H - R)$.

That cost is high if $R \approx H$, that is if few objects are unreachable.

Conservative mark & sweep GC

Sometimes, the compiler does not (or simply cannot) enable the GC to identify pointers.

It is still possible in that case to perform mark & sweep GC, provided that the approximation of the reachability graph errs on the safe side. That is, it sometimes includes unreachable objects, but never excludes reachable ones.

This is the idea behind **conservative GC** (non-conservative GC is said to be **precise**).

Conservative identification of pointers

A conservative GC scans the registers, the stack, global variables and all allocated objects, looking for potential pointers to heap objects.

A value is considered to be a valid pointer if it represents the address of an allocated block.

Whenever such a value is found, the corresponding block is marked and recursively searched for pointers, as usual.

Reducing misidentifications

Some characteristics of the architecture and the compiler can be used to reduce the amount of misidentifications – non-pointers mistaken for pointers.

- With most compilers one has the guarantee that if a block is reachable, then exists at least one pointer referencing its beginning.
- Some architecture require pointers to be aligned in memory.

Summary

Memory management is an important part of the run time system, especially for languages offering implicit memory deallocation.

Implicit memory deallocation generally uses reachability as a good but conservative approximation of liveness.

Reference counting cannot reclaim cyclic structures while other forms of garbage collection, like mark & sweep, can.