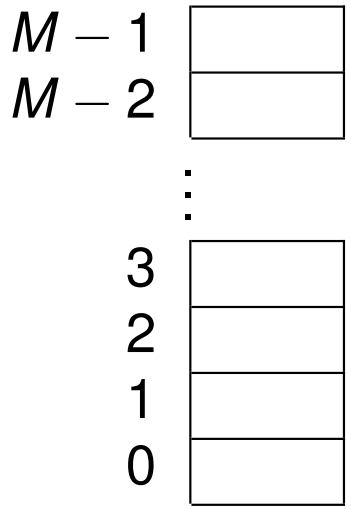
Automatic Memory Management

C. Paulin (courtesy of J.-C. Filliâtre)

M1 MPRI 2025–26

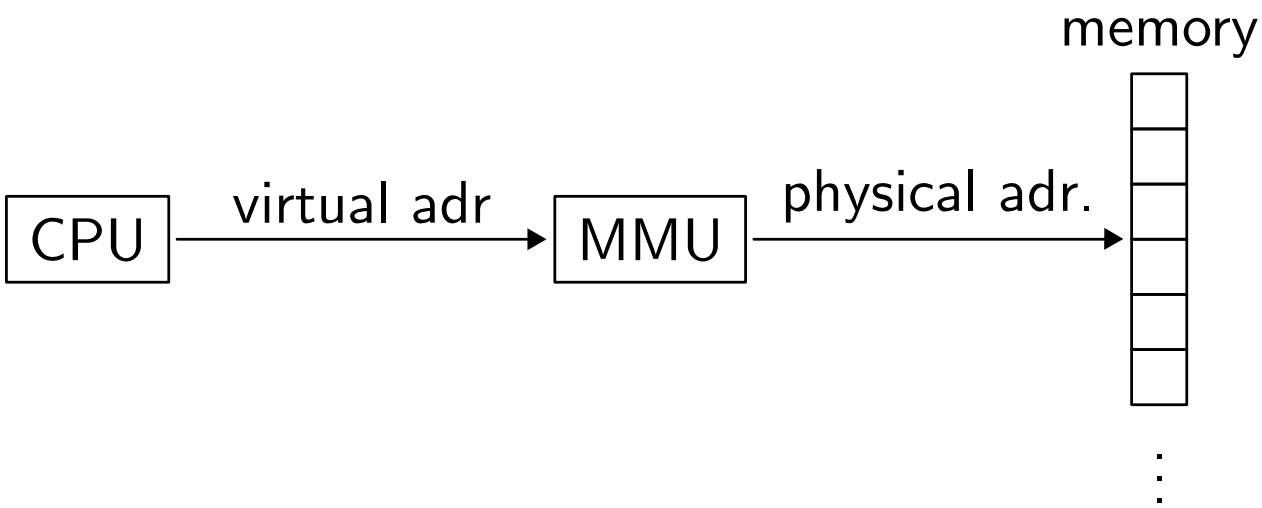
Memory

- physical memory of a computer is a huge array of M bytes,
- CPU can access (read/write) using physical addresses 0, 1, 2, etc.
- M is huge (for instance, $M = 2^{34}$ for a 16 Gio memory)



Virtual memory

No direct access to the memory
The hardware provides a mechanism of virtual memory
MMU (pour *Memory Management Unit*)



Translate virtual addresses (in $0, 1, \ldots, N-1$)
into physical addresses (in $0, 1, \ldots, M-1$)

Pages

the MMU is typically programmed by the OS
Virtual memory is organized in pages (for instance of size 4 ko each)
each page is either

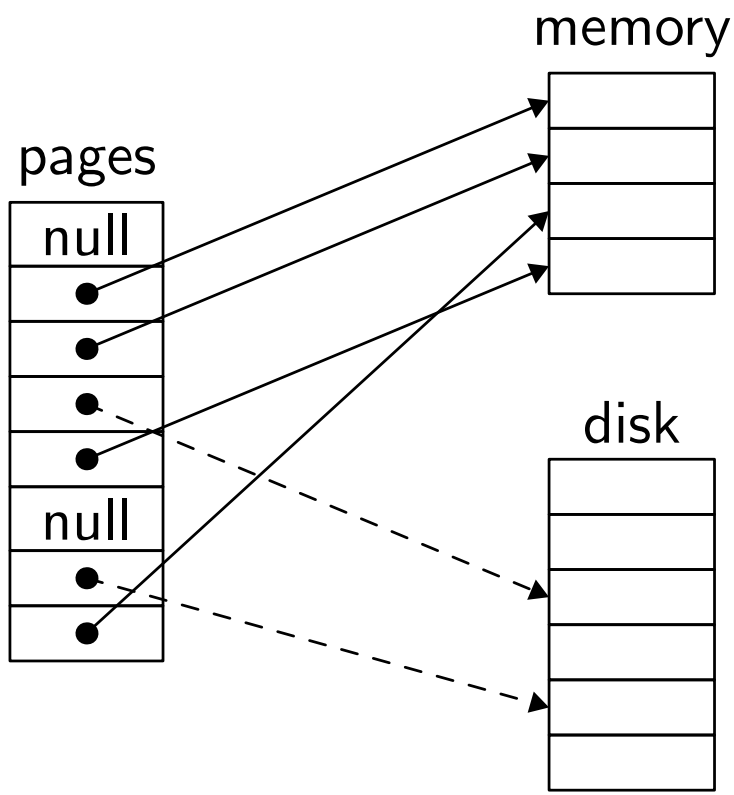
- non allocated
- allocated in physical memory (and the MMU knows the place)
- allocated on the disk

The OS keeps a table with the current status of all pages

Example

8 pages

- 2 non allocated
- 4 in physical memory
- 2 on the disk



How does it work

when the CPU want to read/write a virtual address
the MMU translates it to a physical address

- it works and the instruction is executed
- it fails and
  1. an interruption is raised (*page fault*)
  2. the page is installed into the physical memory
     (possibly by sending back another page on the disk)
  3. the execution starts again on the same instruction
- the OS has a table of pages for each process
- each program think it can use all the (virtual) memory for itself

Avantages

it makes life easier for

- the linker
  (code is always at the same address ,
  for instance `0x400000` for Linux 64 bits)
- loading a program
  (keep the pages on the disk)
- pages sharing between different processus
  (same physical page = different virtual pages)
- memory allocation
  (physical pages do not need to be consecutive)

More on virtual memory

if you want to know more, in particular on the mechanism for address translation, see

Randal E. Bryant et David R. O'Hallaron
Computer Systems : A Programmer's Perspective
chapitre 9 Virtual Memory

Automatic memory management

Static allocation

Easy to allocate static memory

- in the segment `.data` (explicitely initialized)
- in the segment `.bss` (implicitely initialized with zero)

Dynamic allocation

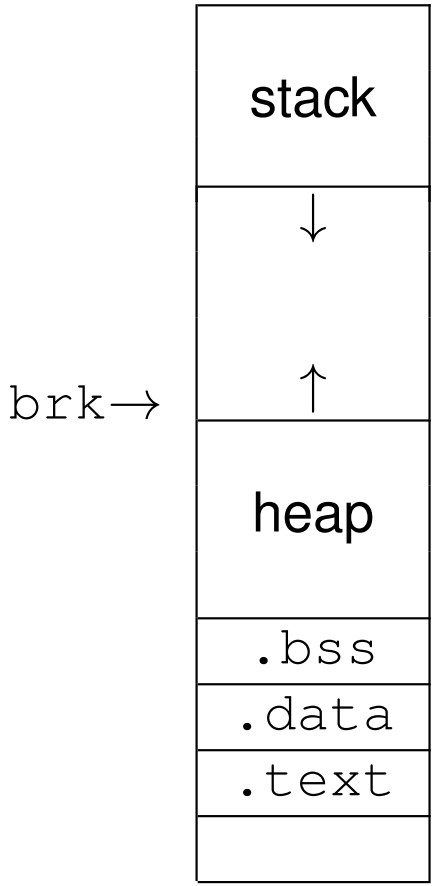
Programs usually allocate memory dynamically

- implicitely, for certain language constructions (objects, closure, etc.)
- explicitely, to store data with size not known at compile time (arrays, lists, trees, etc.)

It is also necessary to be able to free the memory

Heap

- dynamic allocation usually in the heap
- immediately above static data
- a variable `brk` (*program break*) of the system contains the address of the top of the heap (first free address)



Naive allocation

- We can allocate memory by just increasing the variable `brk`
- a system call

```
void *sbrk(int n);
```

  increments `brk` by `n` bytes and returns the old address (base of the allocated block)
- we decrements `brk` using a negative `n`
- corresponds to using the heap as a stack

Memory management

- we want a memory management system which allocates and releases memory blocks in an arbitrary order
- the block release can be
  - explicitely written in the program
    example : C library `malloc`
  - automatically done by an external program
    it is called GC (Garbage Collector)

Objective

using `sbrk`, one can provide two operations

```
void *malloc(int size);
  // returns a pointer to a new block of size
  //  at least size bytes, or NULL if it fails
```

and

```
void free(void *ptr);
  // free the block at the address ptr
  // (needs to be a block previously allocated by malloc
  //  and not yet released,
  //  otherwise the behavior is not defined)
```

Rules

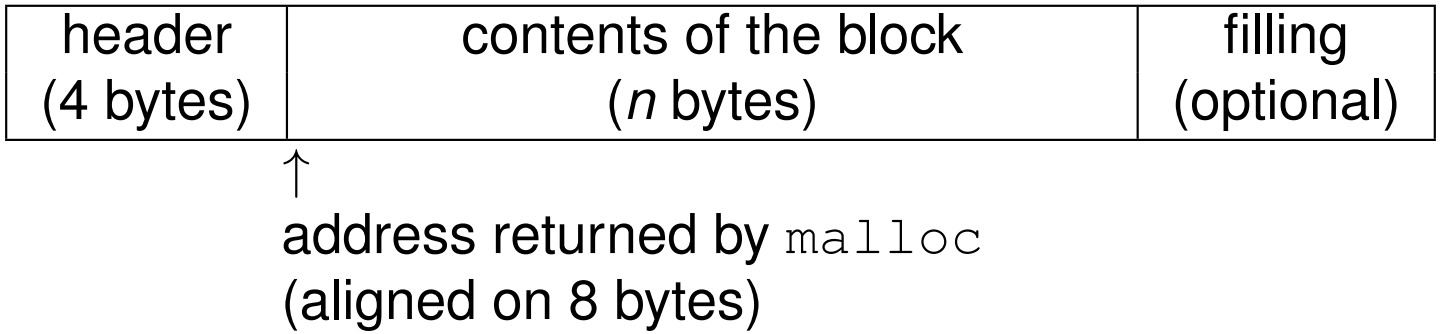
- we do not know which sequence of `malloc`/ `free` will be used
- `malloc` should answer immediately
- if `malloc` or `free` uses other structures, they are also allocated on the heap
- a block returned by `malloc` is aligned on 8 bytes (64 bits architecture)
- a block allocated will not be moved

First idea

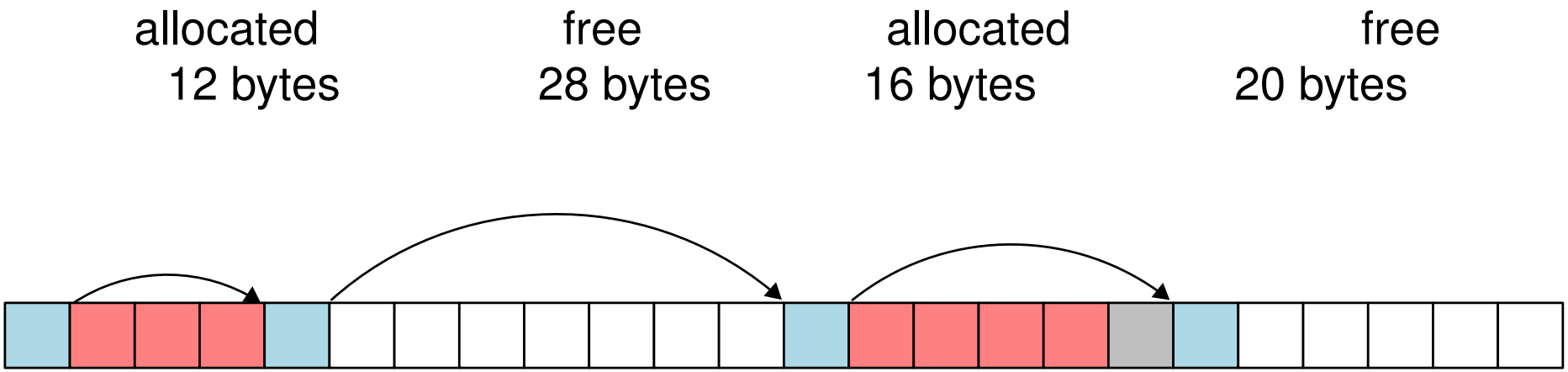
- blocks, allocated or free , are contiguous in the memory
- they are linked together
- given the address of a block, we can compute the address of next block

Implementation

- a header contains the full size of the block plus its status (allocated / free)
- then we have n bytes for the block itself
- we possibly add extra bytes in order for the total size to be a multiple of 8



Example



- square = 4 bytes
- blue = header / red = allocated / gray = filling / white = free

Trick

- because the full size is a multiple of 8, the size ends with 3 zeros
- use one bit to store the status (allocated / free)
- on the previous example

bit	5	4	3	2	1	0	
...	0	1	0	0	0	1	size 16, allocated
...	1	0	0	0	0	0	size 32, free
...	0	1	1	0	0	1	size 24, allocated
...	0	1	1	0	0	0	size 24, free

`malloc(`n`)`

we scan the list of blocks looking for a sufficiently large free block

- if we find one
  - possibly split it in 2 blocks (one allocated + one free)
  - return the allocated block
- otherwise,
  - allocate a new fresh block at the end of the list, with `sbrk`
  - return the new block

Strategy

to find a free block, we can adopt different strategies :

- take the first one large enough (*first fit*)
- take the first one large enough starting from the position of last search (*next fit*)
- choose a block large enough of minimal size (*best fit*)

`free(p)`

it is enough to change the status of block p (from allocated to free)

Problem

the memory get fragmented : smaller and smaller blocks

$\Rightarrow$ some memory is lost

$\Rightarrow$ searching free blocks becomes expansive

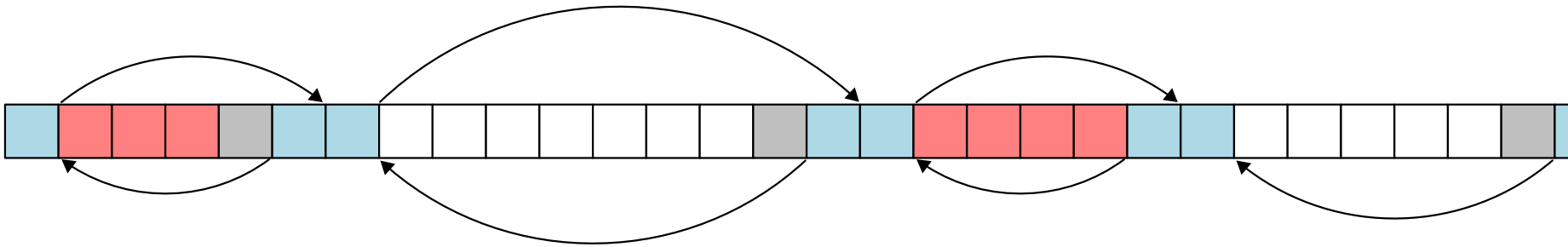
we must compact the memory

Second idea

- when a block is released, look if it can be merged with a block after or before (*coalescing*)
- it is easy to check if the next block is free
  and to merge them (just add the size of the next block to the size of the current block)
- it is more complex to merge with the previous block

Third idea

- copy the header at the end of each block
- idea due to Knuth, called *boundary tags*
- blocks are doubly linked



Fusion

When a block p is released, we look at the previous and following blocks
4 possibles situations

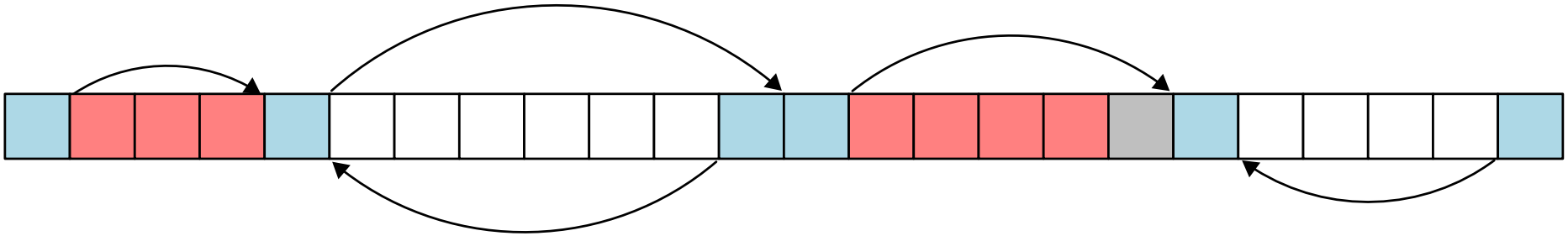
- allocated | p | allocated : do nothing
- allocated | p | free : merge with the following block
- free | p | allocated : merge with the previous block
- free | p | free : merge the three blocks

invariant there are never two consecutive free blocks

Trick

duplicating the header takes up space
instead we could

- duplicate the header only in the free blocks
- use one bit of the header of an allocated block to store if the previous block is free



Fourth idea

- Traversing all blocks is expansive
- We can link together the free blocks (*free list*)
- We use the content of the bloc to store 2 pointers (we need a minimal size for a block)

Releasing a block

when we release a block we have different choices to insert it in the free list

- at the beginning
- sort the free list by increasing base addresses
- sort the free list by increasing block sizes
- etc.

Fifth idea

- Traversing the free list is still expansive if we have many small blocks
- We may use several free lists organized by size
- example : a list of free blocks of size between 2^n and $2^{n+1} - 1$, for each n

Conclusion

- The operations `malloc`/`free` are more subtle than it seems (`malloc.c` in Linux take more than 5 000 lines of code)
- many parameters, many possible strategies
- many articles on the subject, usually using empirical evaluation
- [see for instance Wilson, Johnstone, Neely, Boles. *Dynamic Storage Allocation : A Survey and Critical Review*, 1995]

Reading

C code using these ideas in

- Brian W. Kernighan et Dennis M. Ritchie
  The C language
- Randal E. Bryant et David R. O'Hallaron
  Computer Systems : A Programmer's Perspective

Automatic memory management

GC

- many languages (Lisp, OCaml, Python, Java, etc.) use an automatic mechanism to release memory blocks,
- it is called GC for *Garbage Collector*
- in french, GC can be translated to « ramasse-miettes » or « glâneur de cellules »

GC

principle : a space allocated on the heap for a data (closure, object, records, array, constructor, etc.) which is not reachable from a program variable can be reclaimed and reused for other data

difficulty : in general, we cannot decide statically (at compile time) at what time a data is no more reachable

$\Rightarrow$ GC is part of the executable code

- either as part of the interpretor when the language is interpreted
- or in a library that will be linked with the compiled code (*runtime*)

Vocabulary

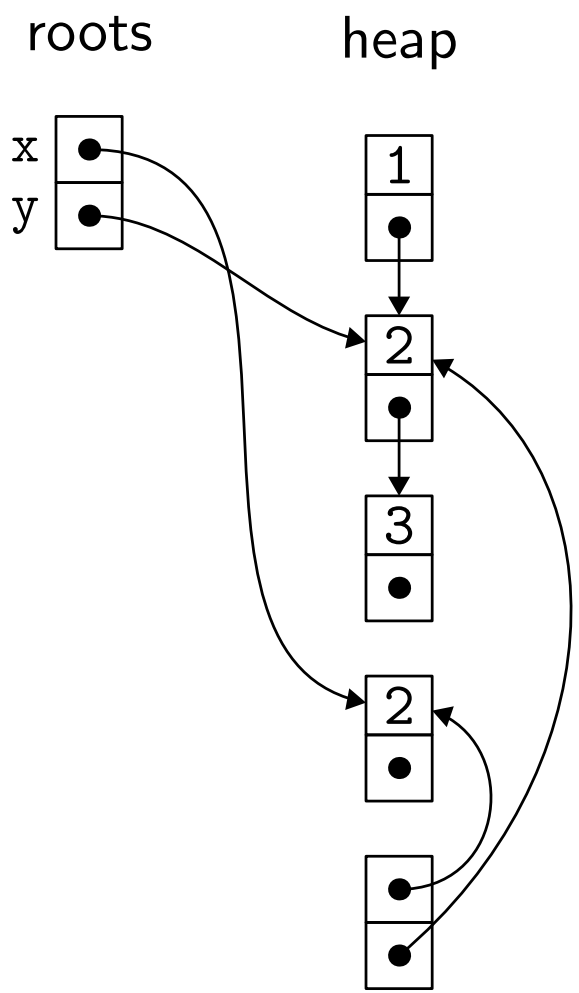
we call block an elementary portion of the heap allocated by the program
a block can contain one ore more pointers to other blocks but also other data (characters, integers, pointers outside the heap, etc.)

at each execution point in the program, we call root an active variable (global variable or local variable in a stack frame or in a register)

a block is alive if it is reachable from a root *i.e.* there exists a sequence of pointers from the root to the block

Example

```
let x, y =
  let l = [1; 2; 3] in
  (List.filter even l, List.tl l)
...
```



Reference counting

we consider a first solution, called *reference counting*

the idea is that each block contains the number of pointers pointing to its address (coming from roots or other blocks)

we update the counter when we do an assignment
(explicit or implicit as in `1::x`)
$b.f \leftarrow p$;
we need

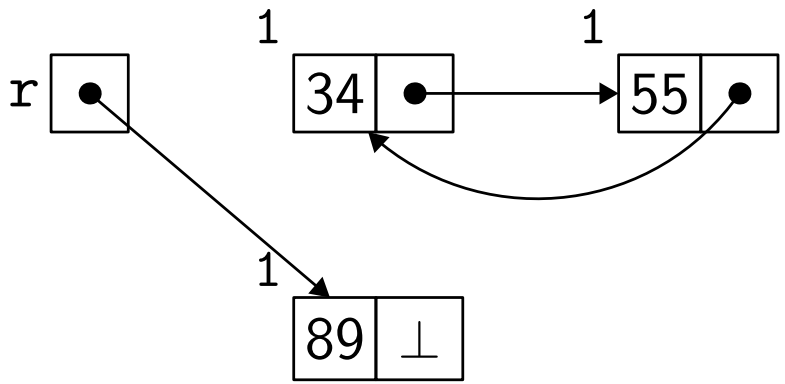
- to decrease the counter of the block corresponding to the old pointer $b.f$; if we reach 0, release the block
- to increase the counter of the block p

when we free a block b, we decrease the counters of all blocks on which b was pointing

Reference counting

Problems :

- updating counters is expensive
- cycles in data-structures leads to blocks which cannot be recovered



Reference counting is rarely used in GC (exceptions : Perl, Python) but sometimes explicitely in programs (C++, Rust,...)

Mark & Sweep

Another solution, more effective is called *mark and sweep*

Two phases :

1. mark all blocks reachable from roots
   (depth-first course, using 1 bit for the mark in each block)
2. consider all blocks and
   - release unmarked blocks
     (put them back in the *free list*)
   - remove mark from others

when doing allocation, if the free list is empty then run GC

Marking

Marking uses a depth-first search

browse(x) =
  if x is a pointer on the heap, not yet marked
    mark x
    for each field f of x
      browse($x.f$)

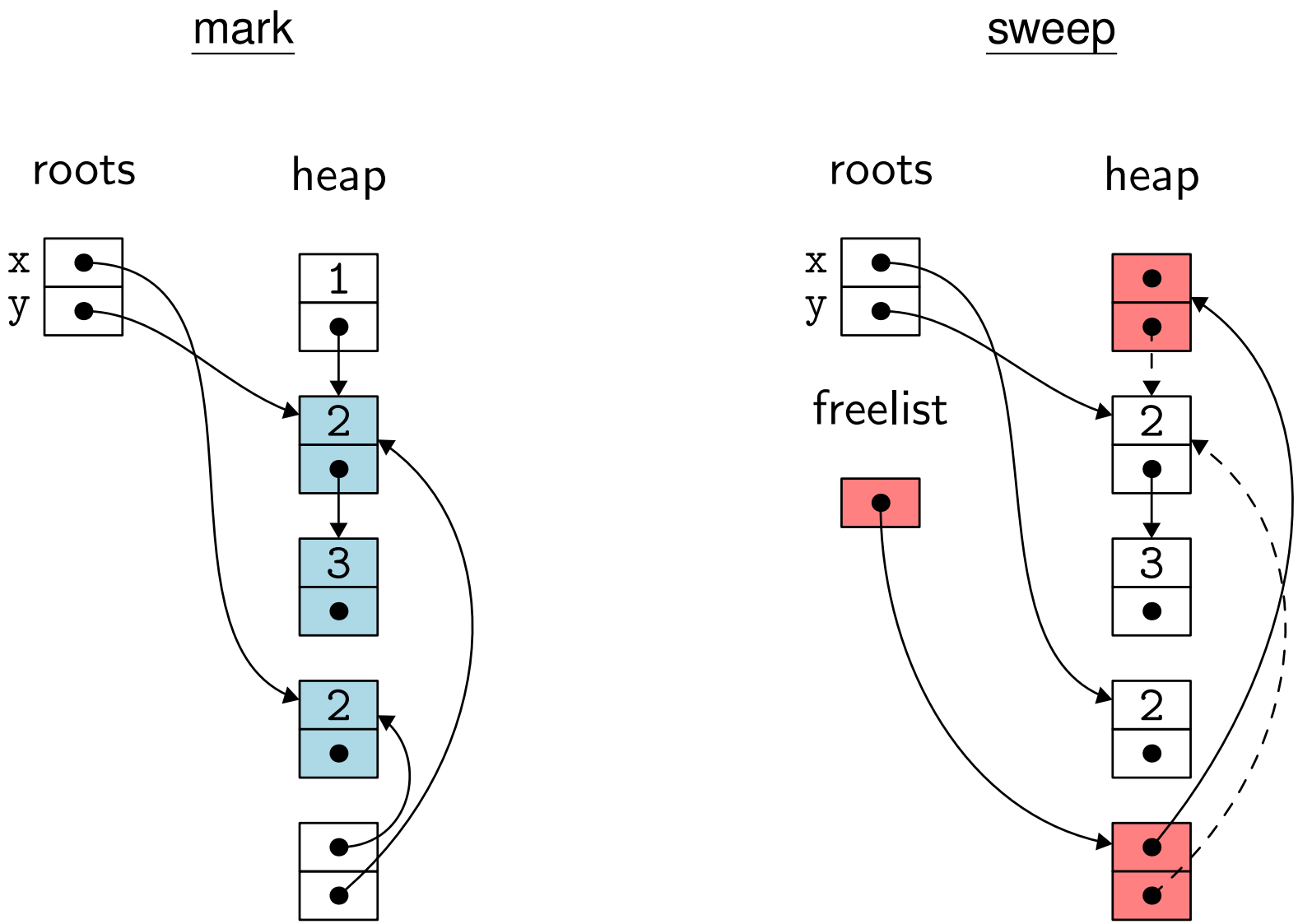
for each root r
  browse(r)

Sweep

Sweeping releases unmarked blocks

for each block *x*
  if *x* is marked
    remove the mark of *x*
  otherwise
    add *x* to the *free list*

Example



Problem

Marking is a recursive algorithm, which uses a stack proportional to the depth of the heap (possibly as large as the heap itself)

it is possible to use an explicit stack or the structure itself to encode the stack (*pointer reversal*)

Other problem

- The program has to stop during a full Mark and Sweep, it takes time
- To avoid that, we can mark the blocks incrementally while doing other operations
- it is called incremental GC
- we need to be careful to preserve invariants for correctness (only release non-reachable blocks)
  - several strategies when doing a write or a read operation to change the marks of involved blocks

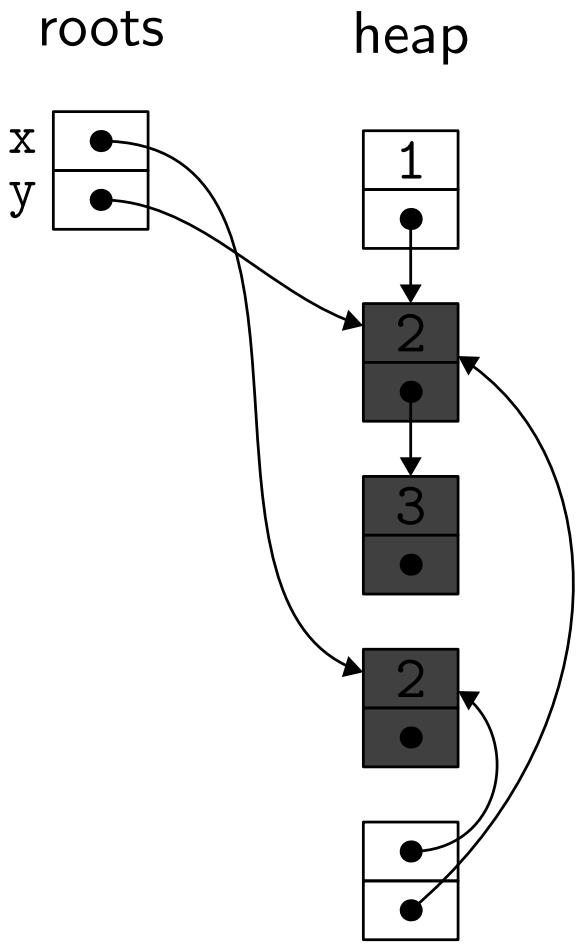
Incremental *mark and sweep*

- Instead of a mark, we use 3 colors :
- Blocks can be
  - white, candidate for release
  - black, reachable from roots, no pointer to white block
  - grey, reachable from roots, the fields have not yet been examined

  initialy the roots are grey, the other blocks are white

Incremental *mark and sweep*

- **while** there is a grey block
  choose a grey block x
  color the block in black
  for each field f of x
    if $x.f$ points to a white block
      color the block in grey



We can run this algorithm in parallel

Incremental *mark and sweep*

when there is no more grey block

- black blocks are reachable from roots
- white blocks are not
  because a black block does not point towards a white block

1. we release the white blocks
2. we color black blocks in white
3. we color the roots in grey

Mark and Sweep

Good solution for identifying unused blocks
(in particular, unused cycles remains white)

does not address the problem of fragmentation

GC with copy

Another solution, called *stop and copy*

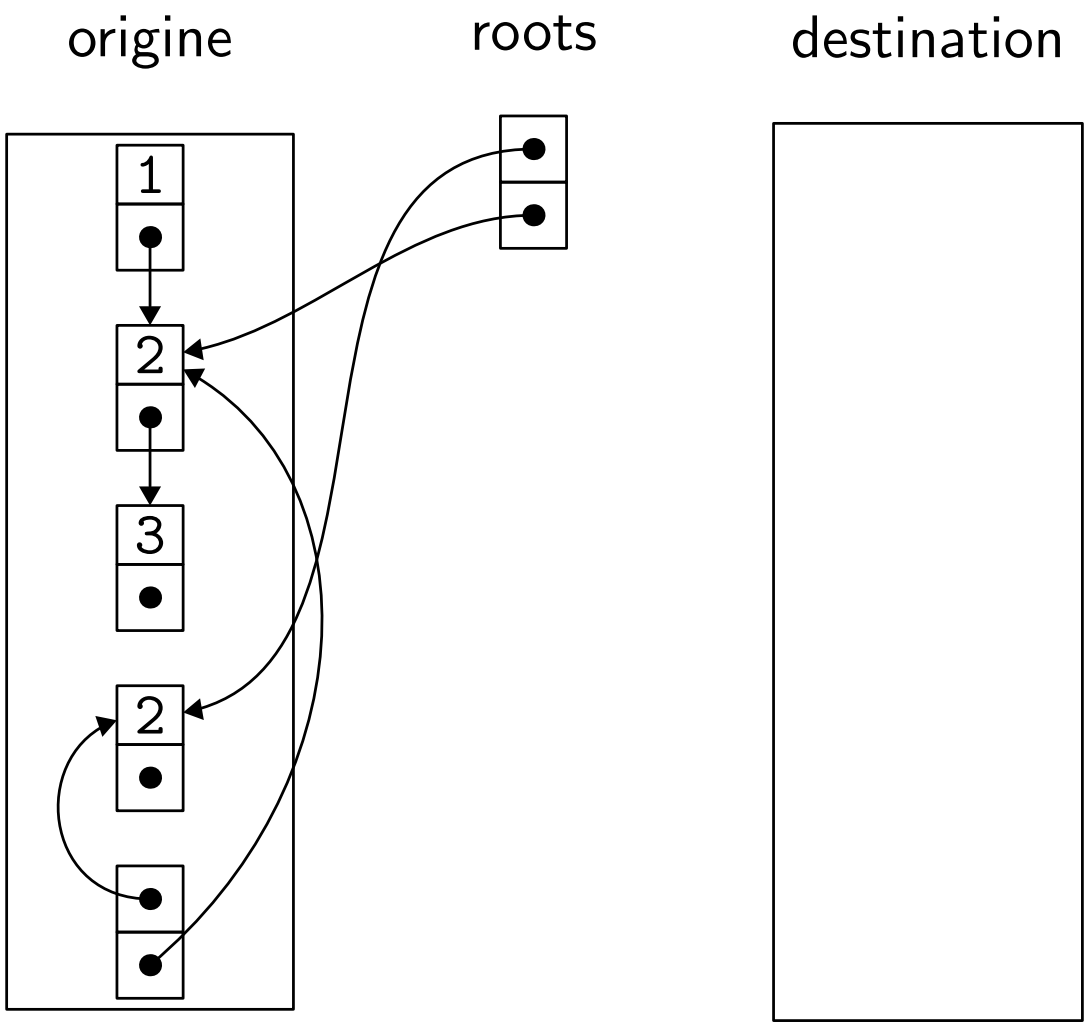
the idea is to separate the heap in 2 parts

1. use one part for linear allocation
2. when this part is full, copy what is reachable in the second part
3. exchange the role of the 2 parts

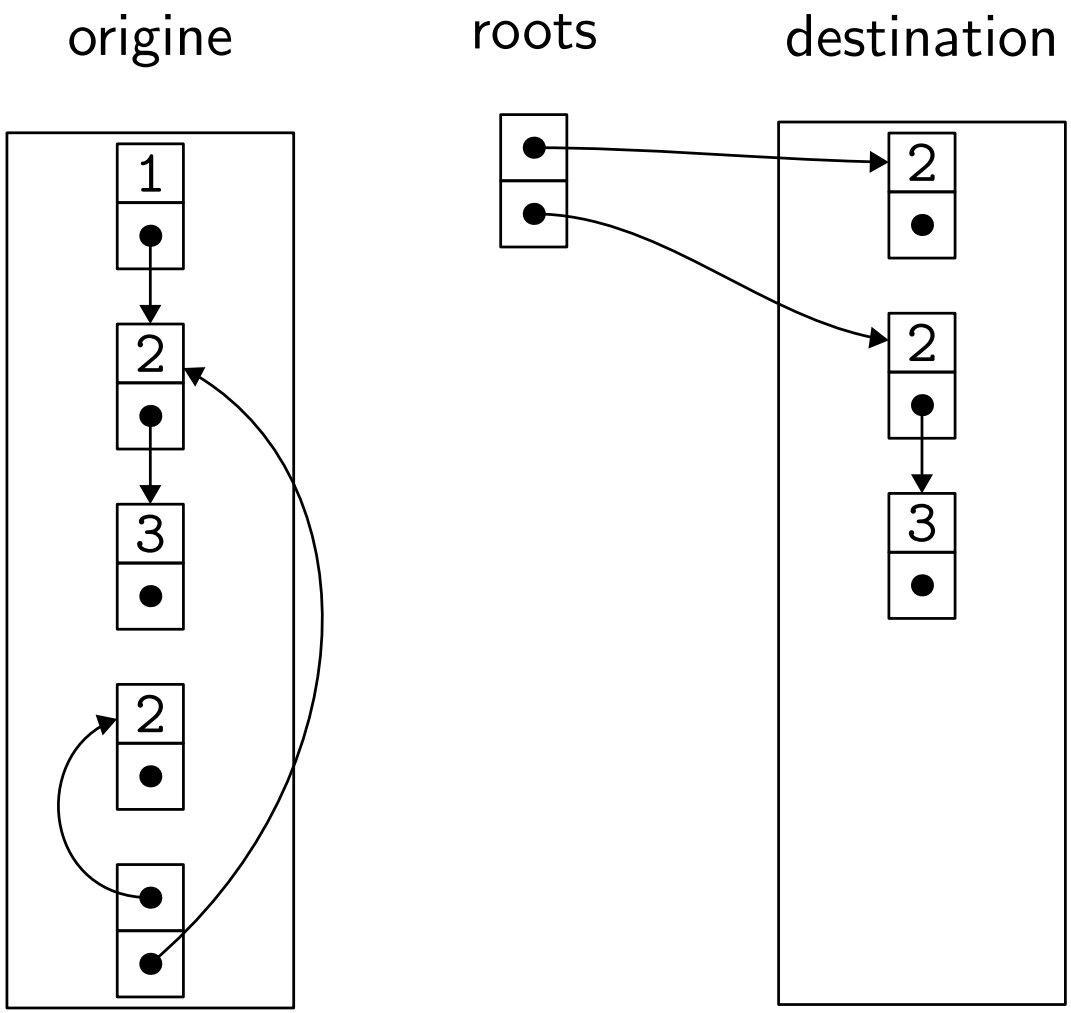
immediate benefits :

- allocating is cheap (addition + comparaison)
- no fragmentation

Example



Example



Cheney's algorithm (1970)

copy using a constant additional space

principle : breadth-first search using

- destination space to store the pointers that need to be visited
- source space to store pointers already visited :
  - when a block is moved from origin to destination, its first field is used to store the address of the destination

Cheney's algorithm

function which copies the block at the address p, if necessary

`next` is the first free address in `destination`

move(p) =
  if p points in `origin`
    if $p.f_1$ points in `destination`
      return $p.f_1$
    else
      for each field f_i of p
        `next`.$f_i \leftarrow p.f_i$
      $p.f_1 \leftarrow$ `next`
      `next` $\leftarrow$ `next` + size of block p
      return $p.f_1$
  else
    return p

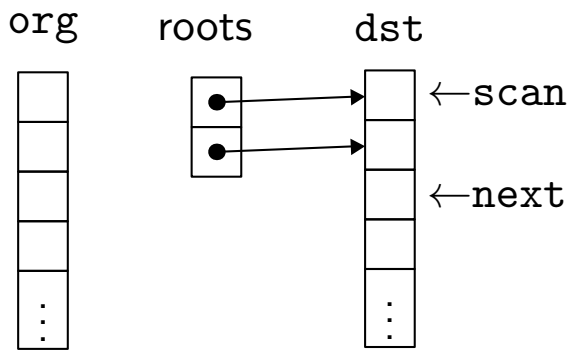
Cheney Algorithm

we start copying, starting from the root

Initially

`scan` ← `next` ← beginning of `destination`

foreach root r
  $r \leftarrow$ move(r)
while `scan` < `next`
  for each field f_i of `scan`
    `scan`.$f_i \leftarrow$ move(`scan`.f_i)
  `scan` ← `scan` + size of block `scan`



In `destination`, the zone between `scan` and `next` are blocks with fields not yet examined

both `scan` and `next` increase during the procedure !

Cheney algorithm

this algorithm is elegant but still has a drawback :
the locality of data is changed during the copy *i.e.* blocks which were close before the copy, will not necessarily be after

locality is important for memory cache

its possible to change Cheney algorithm for mixing depth-first and breadth-first search

Generational GC

in many programs, the values have a very short lifetime,
values which are still there after several collections, will probably be there for a long time

idea : organise the heap with different generations

- G_0 contains more recent values, do frequent collections on them
- G_1 contains older values, needs less frequent collections
- etc.

in practice, there are difficulties to identify roots for each generation, in particular an assignment might introduce a pointer from G_1 to G_0 ...

More on this

The Garbage Collection Handbook
Richard Jones, Antony Hosking, Eliot Moss
CRC Press, 2023

- other algorithms
- implementation details
- parallel and concurrent GC
- real-time GC

OCaml GC

OCaml GC uses 2 generations

- a minor GC (young values) : *Stop & Copy*
- a major GC (old values) : incremental *Mark & Sweep*

`destination` zone of minor GC is the major GC zone

Understanding GC requirements, we explain data representation as chosen in OCaml

Value

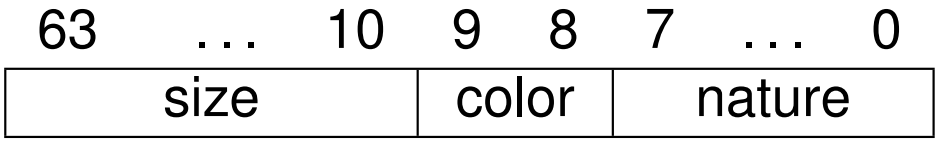
an Ocaml value can be

- an integer, representing a value of type `int` or a constant constructor (`true`, `false`, `[]`, etc.)
- a pointer, an address in or outside the heap

Arguments are always passed as values in OCaml

Block

- a pointer in the heap in OCaml points to a block of size $n + 1$ words (a word = 8 bytes on a 64 bits architecture)
- the first word is the header it contains the size n of the bloc, its nature and two bits used by the GC



(warning : not the same header as for `malloc`)

size of a block

the size of the block is encoded on 54 bits, we have consequentely

```
# Sys.max_array_length;;
- : int = 18014398509481983
```

a string is represented in a compact way (8 characters stored in a word), consequentely

```
# Sys.max_string_length;;
- : int = 144115188075855863
```

Nature of the block

The nature of a block is an integer encoded on 8 bits (0..255) ;

it allows the distinction between

- floating point number
- string
- object
- closure
- the general case of a structure block : record, array, tuple, constructor
- in the case of a constructor, the integer indicates the constructor (for pattern-matching)

Integers and pointers

When the GC looks at a block (for marking or copying), it must distinguish between integers and pointers

difficulty : the compiler cannot indicate to the GC which fields will be pointers because of polymorphic functions

```
let f x = (x, x)

f 42      (* a block which contains 2 integers *)

f [42]    (* a block which contains 2 pointers *)
```

Solution

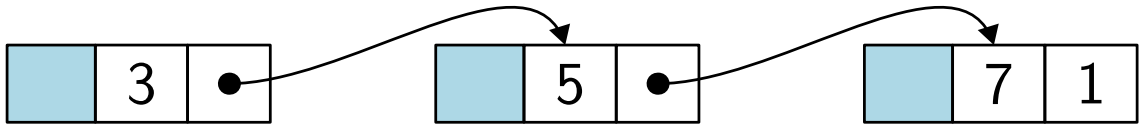
- An OCaml value is
  - either a pointer, which is a pair number because of alignments constraints
  - or an odd integer $2n + 1$, representing the value n

  GC tests the least significant bit in order to determine if a field is a pointer or not

- consequence : OCaml integers are signed 63 bits integers
- arithmetic becomes a bit more complicated
  (the standard library contains a module `Int64`)

Example

the value `1 :: 2 :: 3 :: []` is represented by



Other solution

- to avoid wasting a bit, its possible to consider as a pointer everything which looks like a pointer
- it is
  - correct *i.e.* no reachable block will be released
  - but not complete *i.e.* some unreachable blocks will not be released

  it is called conservative GC

  example : Boehm–Demers–Weiser GC for C and C++
  (see `https://www.hboehm.info/gc/`)

Yet another solution

- allocate everything in the heap, so any value is a pointer
- choice made by Python for instance

Conclusion

understanding programming languages is essential

- to be a better programmer
  - understanding the execution model
    - memory organisation : stack, heap
    - what is an object, a closure . . .
  - be able to program in different languages
- doing research in computer science
  - propose new languages
    - domain specific languages
    - languages better suited for modern architecture, security. . .
  - design tools for languages
    - static analysis, verification
    - compilers (parallel architecture, Just in Time compiling . . .)

Compilers

when compiling, we use

- several phases (typing, computing, allocating...)
- many different (advanced) technics : formal languages, semantics, execution models...

some of these technics are reusable in other contexts :

- linguistics
- proofs using computers (proof assistants, program verification)
- database requests

Evaluation

- The three practical assigments must submitted on ecampus before

  Tuesday October 21st 14 :00

  - Code + a small report explaining what has been done
- Oral exam on Thursday October 23rd (choose your 30mn slot on ecampus)
  - some theoretical questions + review of the project
  - you can use your course notes