École Polytechnique

CSC_52064 – Compilation

Jean-Christophe Filliâtre

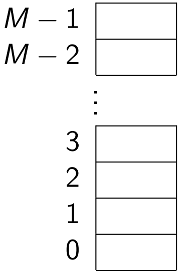
memory allocation

aujourd'hui

1. memory
2. allocation
3. GC
   - reference counting
   - mark and sweep
   - stop and copy
4. an example: OCaml's GC

the physical memory of a computer is a huge array of M bytes,

to which the CPU may access for reading and writing using physical addresses 0, 1, 2, etc.

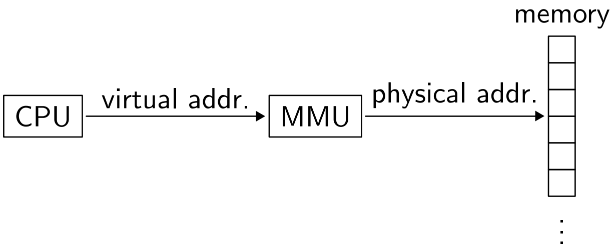


the order of magnitude of M today is several billions (e.g., $M = 2^{34}$ for 16 GiB of memory)

virtual memory

for a long time now, we do not have direct access to physical memory

we use instead a **virtual memory** mechanism provided by the hardware, namely the MMU (for *Memory Management Unit*)



it translates virtual addresses (in $0, 1, \dots, N-1$)
to physical addresses (in $0, 1, \dots, M-1$)

paged virtual memory

the MMU is typically controlled by the **operating system**

the virtual memory is split into **pages** (e.g. each of 4 KiB)

each page is either

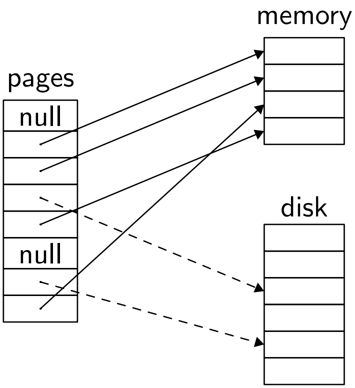
- not allocated
- allocated in physical memory (and the MMU is set)
- allocated on the disk

the operating system maintains a page table

example

8 pages

- 2 not allocated
- 4 in physical memory
- 2 on the disk



when the CPU wants to read or write at some virtual address,
the MMU translates the operation to a physical address

- either it succeeds and the instruction is executed
- or it fails and
  1. an interruption is raised (*page fault*)
  2. the handler installs the page in physical memory
     (possibly by moving another page to the disk)
  3. execution resumes on the same instruction

process

the operating system maintains a page table **per process**

so each program has the illusion to run inside the full (virtual) memory, for its own purposes

advantages

this simplifies

- linking
  (the code is always set at the same address,
  e.g. `0x400000` in 64-bit Linux)
- loading
  (pages are already on the disk)
- sharing of pages between processes
  (one physical page = several virtual pages)
- memory allocation
  (physical pages do not have to be contiguous)

further reading

more details about virtual memory in

> *Randal E. Bryant and David R. O'Hallaron*
> *Computer Systems: A Programmer's Perspective*
> *Chapter 9 Virtual Memory*

memory allocation

static allocation

it is easy to allocate memory **statically**

- either in the `.data` segment (initialized explicitly)
- or in the `.bss` segment (initialized with zero)

yet...

dynamic allocation

most programs have to allocate memory **dynamically**

- either implicitly, through programming constructs (objects, closures, etc.)
- either explicitly, to store data whose size is not known at compile time (arrays, lists, trees, etc.)

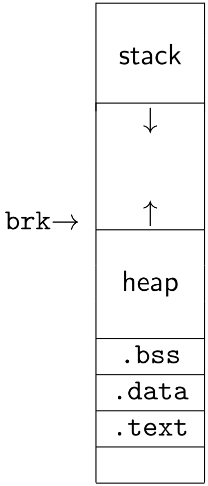
it is generally advisable to **release** it

the heap

dynamic allocated is performed on the stack and in **the heap**

the heap is located above the data segment

the system maintains its ends in a variable `brk` (*program break*)



simplistic allocation

the easiest way to allocate memory consists in increasing the value of `brk`

the system call

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

increases `brk` by `n` bytes and returns its previous value

one may decrease `brk` with a negative value for `n`,
or query its value with $\texttt{n} = 0$

this limits us to using the heap as a stack

memory management

we seek for a more flexible memory manager to allocate and free blocks in a random order

deallocation can be

- performed explicitly by the programmer
  example: the C library `malloc`
- performed automatically by a memory manager
  example : a garbage collector

the `malloc` library

goal

starting from `sbrk`, we want to provide two operations

```
  void *malloc(int size);
    // returns a pointer to a new block
    // of at least size bytes, or NULL in case of failure
```

and

```
  void free(void *ptr);
    // frees the block at address ptr
    // (must have been allocated with malloc
    //  and not yet freed,
    //  otherwise undefined behavior)
```

rules of the game

- we assume nothing about the forthcoming sequence of calls to `malloc` and `free`
- the answer to `malloc` cannot be deferred
- any data structure needed by `malloc` and `free` must itself be allocated on the heap
- any block returned by `malloc` must be 8-byte aligned
- any allocated block can no longer be moved

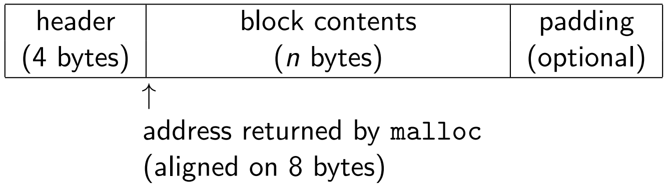
a first idea

the blocks, either allocated or free, are **contiguous** in memory

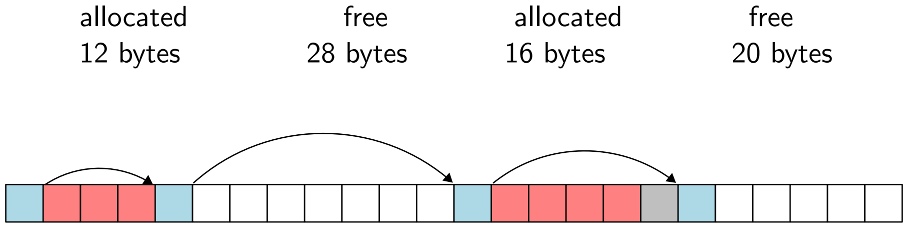
they are **chained**: given the address of a block, we can deduce the address of the next block

implementation

- a header contains the (total) size of the block and the status (allocated / free)
- then come the *n* bytes of the block
- and a possible padding, to give a total size multiple of 8



example



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

trick

the size being a multiple of 8, its three least significant bits are zero

we can use one of these bits 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 traverse the list of blocks looking for a free block that is large enough

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

strategy

to find a free block, several strategies can be used

- the first block that is large enough (**first fit**)
- same thing, but starting where the previous search stopped (**next fit**)
- we choose a block of minimal size among he blocks that are large enough (**best fit**)

`free`(p)

we simply change the status of block p (from allocated to free)

problem

with time, we get memory **fragmentation**: more and more small blocks

$\Rightarrow$ memory is wasted

$\Rightarrow$ search becomes costly

we need **compaction**

second idea

we refine the implementation with a new idea: when a block is freed, we check whether it can be **merged** with an adjacent free block (*coalescing*)

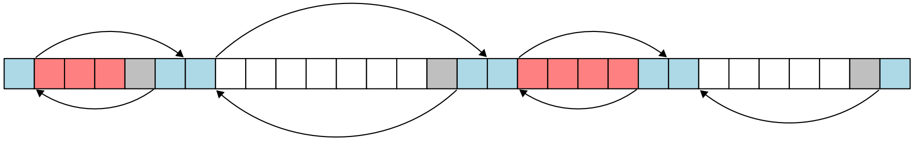
it is easy to determine whether the **next** block is free
and to merge them in that case (adding the sizes)

but there is no easy way to merge with the previous block

third idea

to do so, we duplicate the header at the **end** of each block
(idea due to Knuth, called *boundary tags*)

blocks are now doubly-linked



coalescing

when we free a block p, we check the previous and the next blocks

there are four situations

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

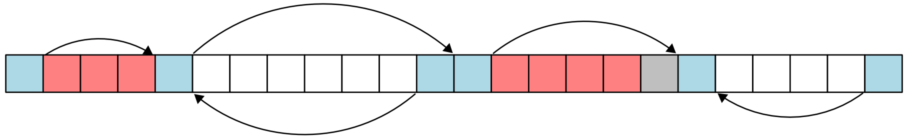
invariant: we never have two adjacent free blocks

trick

duplicate the header requires space

but we can

- do that only in free blocks
- use one bit in the header to indicate whether the previous block is free



fourth idea

it is still costly to traverse blocks when allocating

but we can chain the free blocks in a singly-linked list (**free list**)

for that we use the contents of the block, which is free, to store two pointers (this imposes a minimum block size)

deallocation

when we free a block, we now have several options to insert it back in the free list:

- insert at the beginning
- free list sorted by increasing addresses
- free list sorted by size of blocks
- etc.

fifth idea

traversing the free list can be costly when there are numerous small blocks

we can use **several free lists**, organized by size

example: a list of free blocks whose size is between 2^n and $2^{n+1} - 1$, for each n

conclusion

as we see, `malloc`/`free` are far more subtle than they appear
(Linus's `malloc.c` is more than 5 kloc)

many parameters, many possible strategies

a huge literature, with many empirical evaluations

[see for instance Wilson, Johnstone, Neely, Boles.
Dynamic Storage Allocation: A Survey and Critical Review, 1995]

further reading

we find C code implementing those ideas in

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

GC

GC

many programming languages (Lisp, Python, OCaml, Java, etc.) rely on some **automatic** memory management to free blocks, called **GC** for *Garbage Collector*

(in French, GC is translated as "ramasse-miettes" or "glâneur de cellules")

GC

principle: the space allocated to some data (closure, record, array, object, etc.) that is not **reachable** anymore from the program variables can be **reclaimed** and further reused for other data

difficulty: we typically cannot determine at compile time the moment when a block is not reachable anymore
⇒ the GC is part of the program

- either as some part of the interpreter if the language is interpreted
- or as a library linked with the user code if the language is compiled (this is called the **runtime**)

terminology

in the following, a **block** is any portion of the heap that is allocated by the program

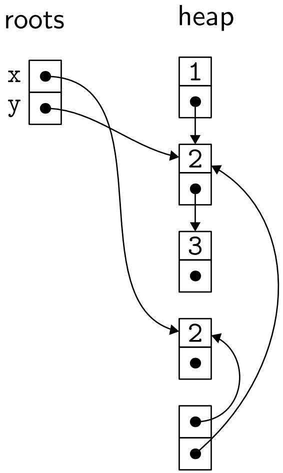
a block may contain one or several pointers to other blocks but also other kind of data (characters, integers, pointers outside of the heap, etc.)

at some point in the program execution, we call **root** any variable that is active (global variable or local variable contained in a stack frame or in a register)

we say that a block is **alive** if it is reachable from a root *i.e.* there is a path of pointers from a root to this block

example

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



reference counting

a first technique to implement a GC is **reference counting**

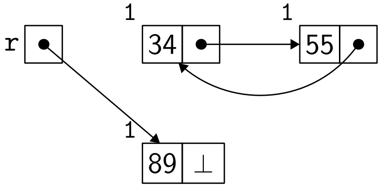
each block contains the number of pointers to that block (from roots or from other blocks)

this number must be updated when the code performs **assignments** (explicit or implicit as in `1::x`) such as $b.f \leftarrow p$; then we need

- to decrement the counter of the block previously in $b.f$; if it reaches 0, we free this block
- to increment the counter of the block p

when we free a block, we decrement the counters of all the blocks to which it points (possibly triggering other deallocations) issues:

- updating counters is costly
- **cycles** in data structures may prevent some blocks to be reclaimed



reference counting is sometimes used explicitly by programmers (e.g., the type `Rc<T>` of Rust) or combined with other techniques (e.g. CPython)

mark and sweep

let us consider another technique, called **mark and sweep**

it proceeds in two steps

1. we mark the blocks that are reachable from the roots (using a depth-first traversal and one bit per block)
2. we scan all blocks and
    - we reclaim those that are not marked (they are put back in the free list)
    - we unset the mark on the other blocks

when an allocation is requested, we look in the free list; if it is empty, this is a good opportunity for a mark and sweep

marking

marking uses a depth-first traversal, as follows

```
mark($x$) =
  **if** $x$ is a pointer to some unmarked block in the heap
    mark $x$
    **for each** field $f$ of $x$
      mark($x.f$)

**for each** root $r$
  mark($r$)
```

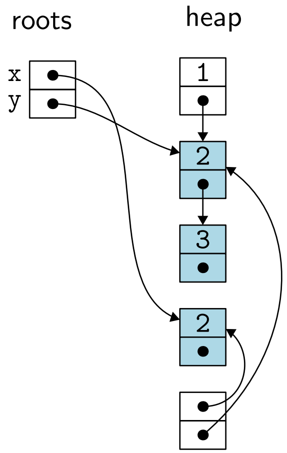
sweeping

sweeping reclaims the unmarked blocks

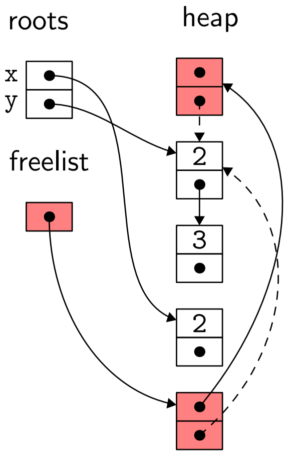
```
for each block $x$
  if $x$ is marked
    unmark $x$
  else
    insert $x$ in the free list
```

example

mark



sweep



issue

marking is a **recursive** algorithm, which may require a stack size as large as the heap

we could use an explicit stack or the blocks themselves to encode the stack (**pointer reversal**)

another issue

but more importantly, we do not want to “stop the world” to perform a long mark-and-sweep collection

to remedy this, we mark the blocks **incrementally**, during the various calls to the GC

this is **incremental garbage collection**

incremental mark and sweep

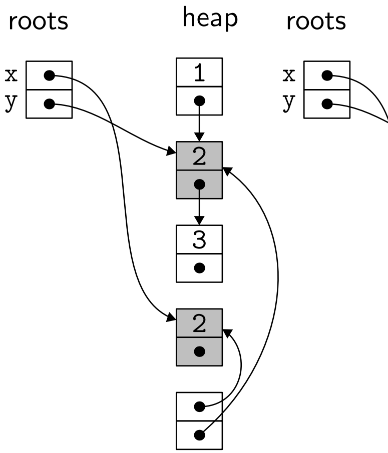
a single mark is not enough, we need three colors

we have blocks colored

- **white**, not reached (yet)
- **black**, reached from roots and without pointers to white blocks
- **gray**, reached from root but not yet scanned

initially, roots are gray and all the other blocks are white

incremental mark and sweep



```
while there are gray blocks
    pick a gray block $x$
    color it black
    for each field $f$ of $x$
        if $x.f$ points to a white block
            color this block gray
```

advantage: we can do any number of turns of this loop

incremental mark and sweep

once we do not have gray blocks anymore,

- black blocks are all reachable from the roots
- white blocks, on the contrary, are not
  for a black block never points to a white block

therefore,

1. we reclaim the white blocks
2. we whiten the black blocks
3. we color the roots in gray

mark and sweep

this is a good way to determine unreachable blocks
(in particular, we now reclaim unreachable cycles)

not a solution to the problem of fragmentation

stop and copy

let us consider a third technique, called **stop and copy**

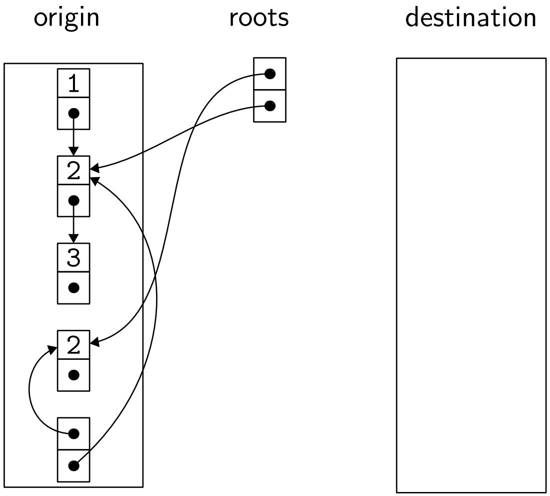
we split the heap in two halves

1. we only use one half, in which we allocate linearly
2. when it is full, we copy the reachable blocks into the other half, and we swap roles

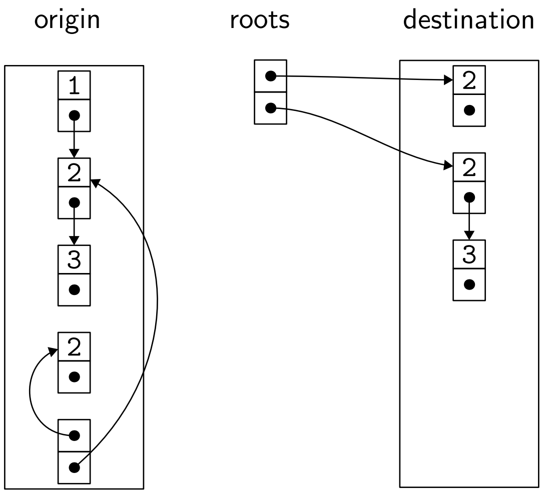
immediate benefits:

- allocation is cheap (an addition and a comparison)
- no more fragmentation issue

example



example



Cheney scanning algorithm (1970)

performs the copy using only constant extra space

principle: a breadth-first traversal that uses

- the destination space as temporary storage for blocks yet to be updated
- the origin space as storage for pointers already updated: when a block is copied, its first field indicates where it was copied

Cheney scanning algorithm

we first implement a function to copy the block at address p, if not yet already done

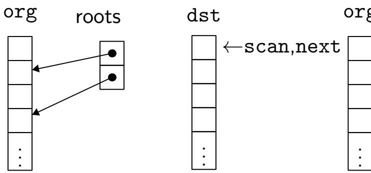
`next` points to the first available space in `destination`

```
move($p$) =
  **if** $p$ points to origin
    **if** $p.f_1$ points to destination
      **return** $p.f_1$
    **else**
      **for each** field $f_i$ in $p$
        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 scanning algorithm

we can now make the copy, starting with the roots

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



the zone in destination between scan and next contains the blocks whose fields are not yet updated

note that scan increases, but so does next!

Cheney scanning algorithm

although very elegant, this algorithm has at least one flaw: it modifies the locality of the data, *i.e.* blocks that were close before copying are not necessarily close afterwards

with memory caches, locality cannot be ignored

it is possible to modify Cheney algorithm to combine breadth-first and depth-first traversals

generational GC

in many programs, most values have a short lifetime, and those surviving several collections are likely to survive others

hence the idea to organize the heap into several **generations**

- G_0 contains the most recent data, and we make frequent collections here
- G_1 contains data older than those in G_0, and we make collections there less frequently
- etc.

in practice, this is not easy to identify the roots of each generations, notably because an assignment may introduce a pointer from G_1 to G_0

further reading

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

- other algorithms
- implementations details
- parallel and concurrent garbage collection
- real-time garbage collection

example: OCaml's GC

OCaml's GC

two generations

- a minor heap (young values): Stop & Copy
- a major heap (older values): incremental Mark & Sweep

the minor heap's `destination` is the major heap

value representation

now that we understand the GC requirements, we can explain the value representation chosen by OCaml

value

an OCaml value is

- either an integer, that is a value of type `int` or a 0-ary constructor (`true`, `false`, `[]`, etc.)
- or a pointer

reminder: OCaml's passing mode is **by value** (cf lecture 5)

block

a pointer points to block of $n + 1$ **words**
(one word = 8 bytes on a 64-bit architecture)

the first word is a **header** containing the size n, the nature of the block and two bits for the GC

63 … 10	9 8	7 … 0
size	color	nature

(this is different from `malloc`'s headers)

size of a block

the size being stored on 54 bits, we have

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

character strings are represented in a compact way (8 bytes per word), thus

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

nature of a block

the block's nature is coded on 8 bits (0..255); it is used to distinguish

- floating-point number
- character string
- object
- closure
- record, array, tuple
- constructor
  (and tells which constructor it is, needed for pattern matching)

integers and pointers

when the GC is scanning a block (for marking or copying), it must distinguish between integers and pointers

issue: the compiler cannot always tell the GC which are the fields containing pointers in the presence of polymorphism

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

```
f 42    (* a block containing two integers *)
```

```
f [42]  (* a block containing two pointers *)
```

solution

an OCaml value is

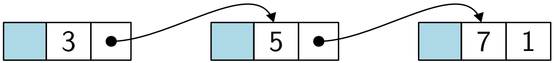
- either a pointer, necessarily **even** for alignment reasons
- or an **odd** integer $2n + 1$, standing for the value n

the GC tests the least significant bit to determine if a field is a pointer

consequence: OCaml's integers are **63-bit** signed integers
(but the OCaml standard library provides a module `Int64`)

example

the value `1 :: 2 :: 3 :: []` has the following memory layout



arithmetic

the integer n being represented by $2n+1$, arithmetic becomes a little more complicated

but the OCaml compiler makes a good use of `lea` at many places

e.g., function

```
let f x y = x + y
```

is compiled to

```
f:  leaq -1(%rax,%rbx), %rax
    ret
```

another solution

to avoid wasting a bit, another solution is to consider “any value that looks like a pointer” as a pointer

this is

- **sound** *i.e.* reachable blocks are not reclaimed
- but **incomplete** *i.e.* some unreachable blocks may not be reclaimed

this is called **conservative collection**

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

yet another solution

another solution consists in allocated all the values on the heap, so that we only have pointers (we say that all values are **boxed**)

this is what Python does for instance

take away

programming languages

understanding programming languages is essential for

- **coding**
  - have a precise execution model in mind
  - choose the right abstractions

- doing **research** in Computer Science
  - design new languages
  - design tools

in particular, we explained

- what is the stack
- various passing modes
- what is an object
- what is a closure

compilation

compilation involves

- numerous techniques
- several passes, mostly orthogonal

most of these techniques can be reused in contexts other than code generation, such as

- computational linguistics
- computer-assisted proofs
- databases

compilation also means...

many other things we didn't have time to explore

module systems
common sub-expression
program transformations
abstract interpretation
alias analysis
loop unrolling
interprocedural analysis
peephole optimization
memory caches
logic programming
just-in-time compilation
instruction scheduling
etc.

next

- lab 9
  - mini Java continued

- the project is due **Sunday March 16, 6pm**
  - via Moodle
  - graded on the basis of your report and your source code

- exam on **Monday March 17 2pm–5pm** (T5)
  - lecture notes are allowed
  - archives on the website of the course