Lecture 20

Tuesday, November 2, 2021 12:34 PM

Memory Management

- Program
- Language runtime (Pointers, memory allocation, garbage collection)
- Operating System (Virtual memory, page tables, swapping, ...)
- Hardware

libc

(Memory controller, Direct Memory Access (DMA) Translation Lookaside Buffer (TLB), ...)

```
utop # let f n =
  let l = List.range 0 n in
  fun i -> List.nth_exn l i;;
val f : int -> int -> int = <fun>
```

```
utop # let g = f 10;;
val g : int -> int = <fun>
-( 14:15:20 )-< command 5 >-
utop # g 3;;
- : int = 3
-( 14:15:30 )-< command 6 >-
utop # g 5;;
- : int = 5
-( 14:15:34 )-< command 7 >-
utop # g 7;;
- : int = 7
```

This call to f creates one list l

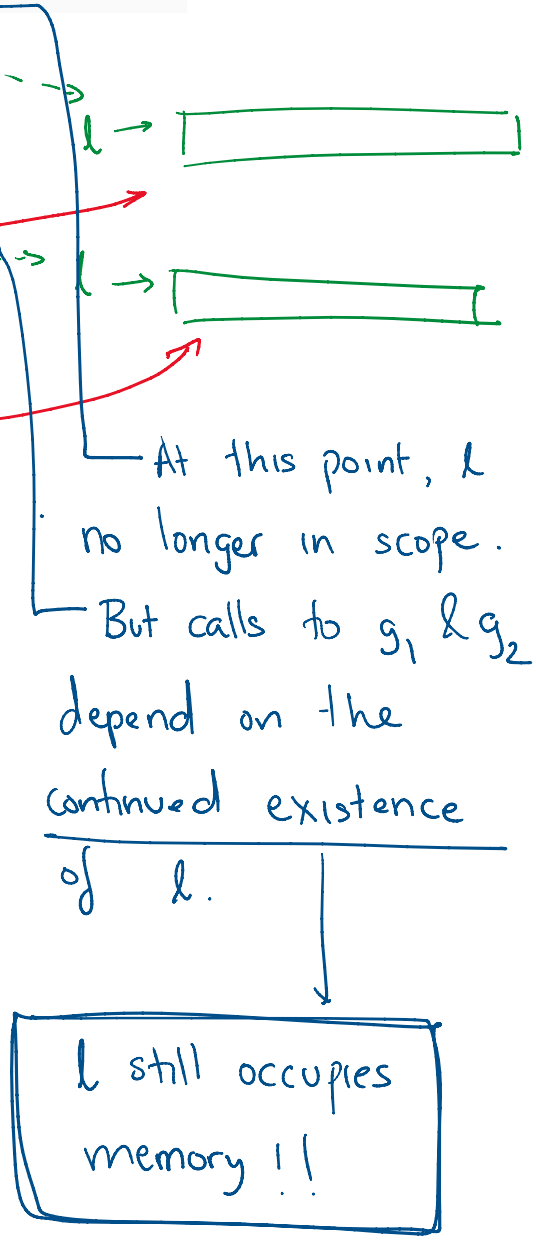
All these calls are accessing the same list

```
utop # let f n =
```

```
utop # let f n =
  let l = List.map (List.range 0 n) ~f:(fun x -> ref x) in
  fun i -> List.nth_exn l i;;
val f : int -> int -> int ref = <fun>
```

```
-( 14:20:52 )-< command 13 >-
utop # let g1 = f 10;;
val g1 : int -> int ref = <fun>
-( 14:22:05 )-< command 14 >-
utop # let g2 = f 10;;
val g2 : int -> int ref = <fun>
```

```
-( 14:22:11 )-< command 15 >-
utop # g1 3;;
- : int ref = {Base.Ref.contents = 3}
-( 14:22:15 )-< command 16 >-
utop # g2 3;;
- : int ref = {Base.Ref.contents = 3}
-( 14:22:22 )-< command 17 >-
utop # !(g1 3);;
- : int = 3
-( 14:22:39 )-< command 18 >-
utop # !(g2 3);;
- : int = 3
-( 14:22:46 )-< command 19 >-
utop # !(g1 3);;
- : int = 3
-( 14:22:53 )-< command 20 >-
utop # (g1 3) := 5;;
- : unit = ()
-( 14:22:59 )-< command 21 >-
utop # !(g1 3);;
- : int = 5
-( 14:23:18 )-< command 22 >-
utop # !(g2 3);;
- : int = 3
```



Job of the language runtime

- What things are in memory
- What memory is used
- What memory is available

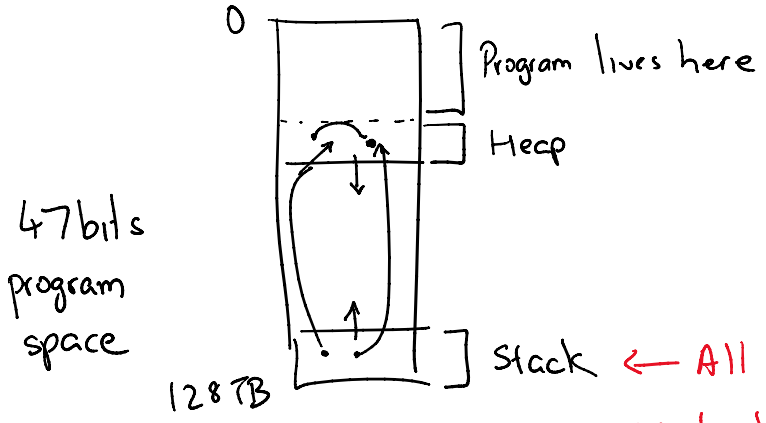
```
-( 14:36:09 )-< command 26 >-
utop # let f n =
  let l = List.map (List.range 0 n) ~f:(fun x -> ref x) in
  fun i -> List.nth_exn l i;;
val f : int -> int -> int ref = <fun>
```

```
  let l = List.map (List.range 0 n) ~f:(fun x -> ref x) in
  fun i -> List.nth_exn l i;;
val f : int -> int -> int ref = <fun>
─( 14:36:16 )─< command 27 >──────────────────────────────────────
utop # let g1 = f 30;;
val g1 : int -> int ref = <fun>
─( 14:36:21 )─< command 28 >──────────────────────────────────────
utop # let g2 = f 30;;
val g2 : int -> int ref = <fun>
─( 14:36:23 )─< command 29 >──────────────────────────────────────
utop # !(g1 5);;
- : int = 5
─( 14:36:26 )─< command 30 >──────────────────────────────────────
utop # !(g2 16);;
- : int = 16
─( 14:36:36 )─< command 31 >──────────────────────────────────────
utop # let g1 = f 30;;
val g1 : int -> int ref = <fun>
```

← At this point, first g1 can no longer be accessed.

So: this list can no longer be accessed either.

Layout of Virtual Address Space

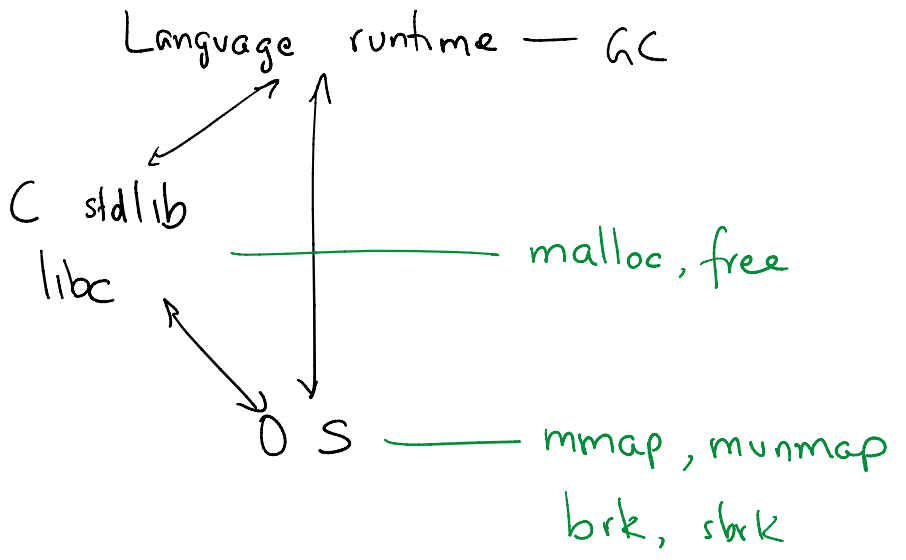


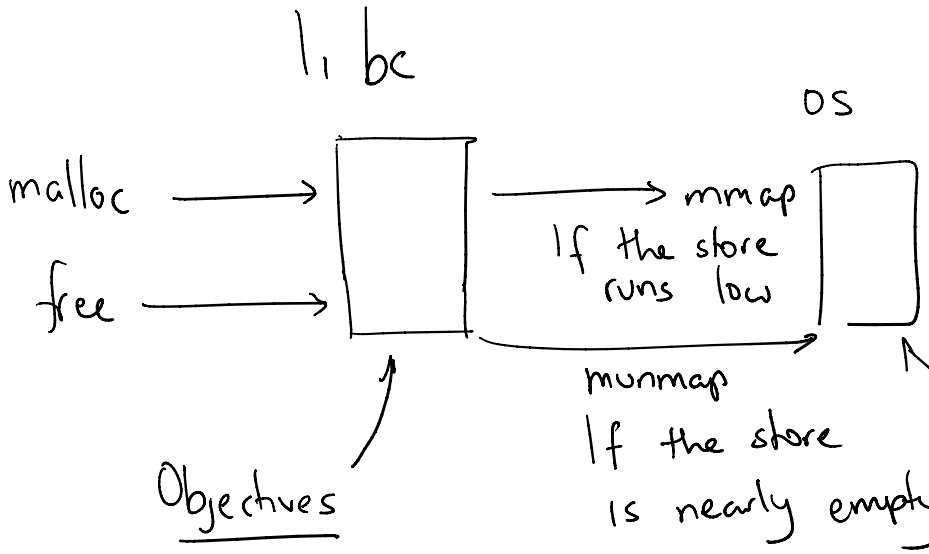
47bits program space

x86_64

https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

← All currently accessible values must be reachable from the stack.





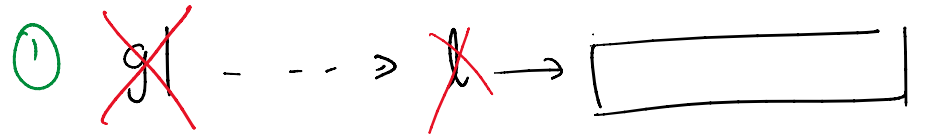
Memory allocators
- Glibc
- jemalloc
- dlmalloc

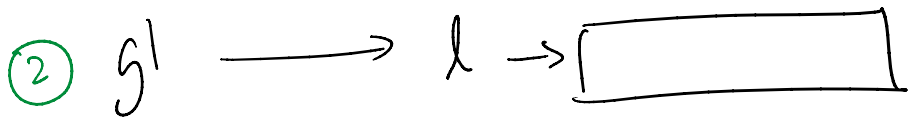
|- dlmalloc |

Garbage Collection

Question: When is it safe to deallocate an object?

Proposal: If nothing points to it. (ref-counting)



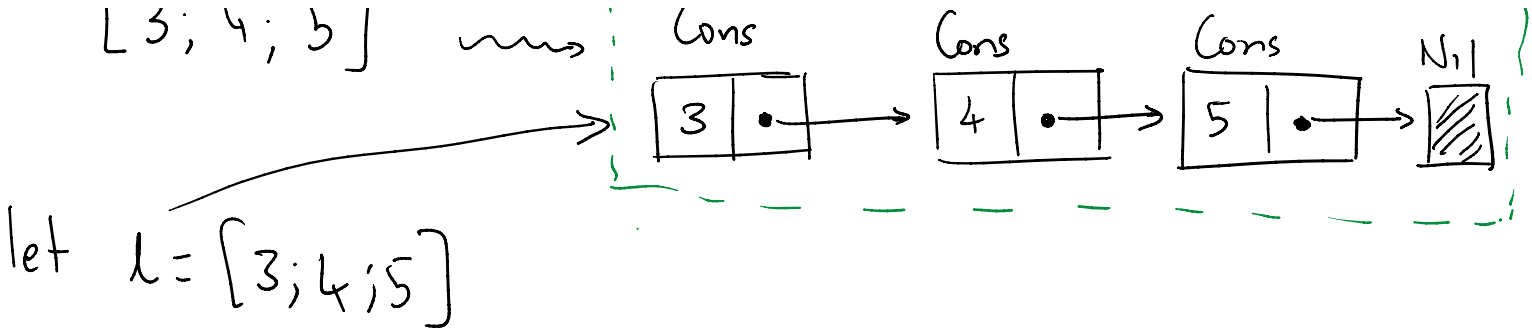


③ Now, nothing points to the first gl. So deallocate

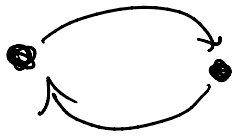
④ Now, nothing points to the first l. Deallocate!

stack 

[3; 4; 5] 



Main challenge of ref-counting: Cycles



Alternative: Mark-and-sweep