CS 312

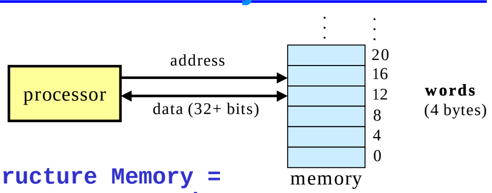
Spring 2003

Lecture 19:
Memory Management

The grand illusion

- Evaluation models say: infinite universe of SML values
  - primitives, tuples, datatype constructors
  - arbitrary number of distinct ref cells
- Reality: finite computer memory
  - huge array of ~5 billion bits of information
  - laid out sequentially on silicon
- How does SML (Java, ...) provide this abstraction of the hardware?

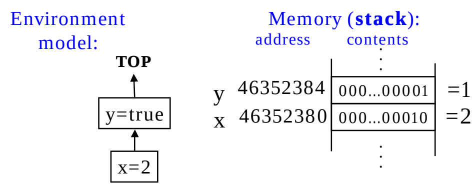
The memory interface




```
structure Memory =
  type memory = int array
  type address = int
  type data = int
  exception UnalignedAccess
  val read: address -> data = …
  val write: address * data -> unit = …
end
```

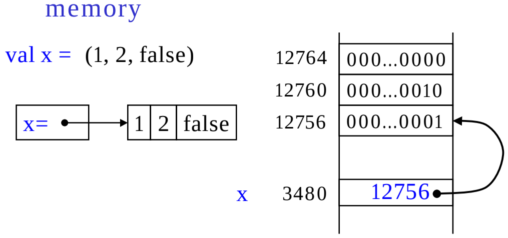

A simple model

- SML values stored in memory
- Variables take up one memory location
- Primitives (int, bool) stored in one word



Boxes

- Tuple of values stored sequentially in memory

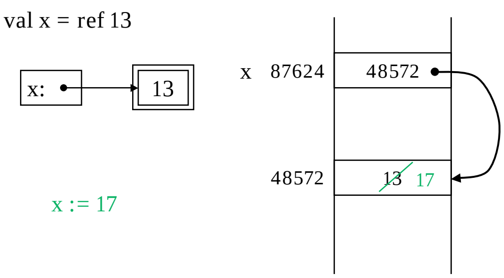


- Variable bound to a tuple contains *address* of tuple in memory (in SML)

Refs

- Ref is just a memory cell

val x = ref 13



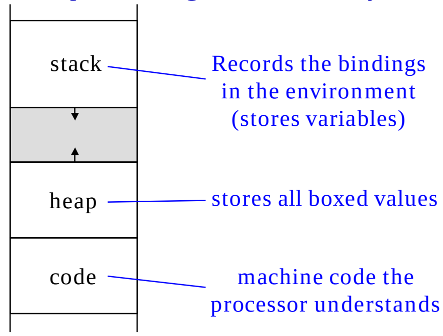
Memory management

- How does system know where to put things in memory? How to:
  - find memory for a new variable?
  - find memory for a new value?
  - avoid putting two values in same place?
  - avoid leaving memory unused?
  - reuse memory if value stored there is not needed?

7

A typical memory layout

- Three important regions of memory

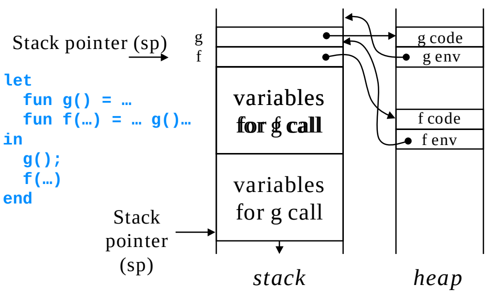


8

Stack

- Stack grows downward in memory
- Stores variables for each function call

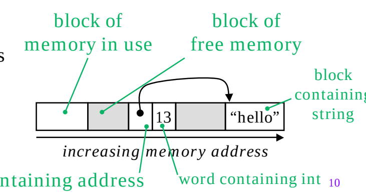
```
let
  fun g() = …
  fun f(…) = … g()…
in
  g();
  f(…)
end
```



9

Heap

- Memory heap ≠ Binary heap
- Memory management:
  - where things go in the heap   val x = (1, 2, y) ...
  - when to get rid of things in the heap
  - possibly: moving things in the heap
  - must be done at run time; can't preallocate space
- Things in heap take space:
  - Tuples, records
  - Refs
  - Closures
  - Strings



10

Allocator interface (explicit free)

```
signature ALLOCATOR = sig
  (* malloc(n): allocate an unused block of
   * n bytes and return the address.
   * Requires: n > 0 *)
  val malloc: int -> address

  (* free(a): release the previously
   * allocated block at address a.
   * Requires: a was previously returned
   * by malloc and has not been freed
   * already *)
  val free: address -> unit
end
```

Requires clause on `free` makes C programming difficult -- hard to share values between different modules

11

Allocator interface (with GC)

```
signature ALLOCATOR = sig
  (* malloc(n): allocate an unused block of
   * n bytes and return the address.
   * Requires: n > 0 *)
  val malloc: int -> address

 (* collect_garbage(roots): find blocks
  * of memory previously allocated by
  * malloc(), but that are not now
  * reachable from roots. Mark these
  * blocks unused. *)
  val collect_garbage: address list
end
```

12

Fixed-size blocks

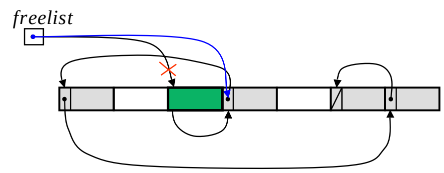
```
signature ALLOCATOR = sig
  val size = 16
  (* malloc(n): allocate an unused block of
   * n bytes and return the address.
   * Requires: n = size *)
  val malloc: int -> address

  (* free(a) releases the previously
   * allocated block at address a.
   * Requires: a was previously returned
   * by malloc and has not been freed
   * already *)
  val free: address -> unit
end
```

Much easier to implement...

13

Freelist

- Idea: keep all the unused blocks of memory in a linked list
  - Use first word of each block to store pointer
  - On malloc, update freelist to tail, return head
  - On free, do cons



14

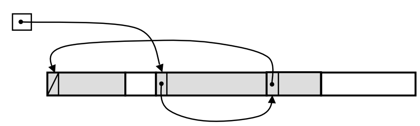
Fixed-size allocator

```
structure Allocator :> ALLOCATOR =
  (* freelist actually stored in memory *)
  val freelist: address ref = ref 0
  val memory: Memory.memory = …

  fun malloc(n) = let
    val ret = !freelist
    val next = Memory.read(memory, !freelist)
  in
    freelist := next;
    ret
  end

  fun free(a) =
    (Memory.write(memory, a, !freelist);
     freelist := a)
end
```

15

Variable-sized blocks

- Problem: different values take different amounts of memory
- Idea: use freelist just like before, but with variable-sized blocks of memory

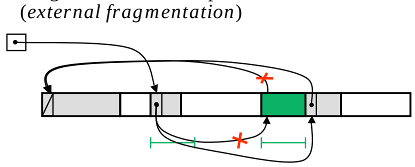


- Problems:
  - Head of freelist may not be big enough (search!)
  - Head of freelist may be too big

16

First-fit

- On allocation, walk down freelist until first large-enough block is found
- Split into allocated part, unused part, put unused part back on freelist
- Problems:
  - Can be slow: may need to see entire list
  - Fragmentation of heap into small unusable blocks (*external fragmentation*)



17

Buddy system

- Idea 1: accelerate allocation by having multiple freelists, for different sizes
- Idea 2: free block can be split into two free "buddies" that know about each other

exponential buddy	Fibonacci buddy
1 → …	1 → …
2 → …	2 → …
4 → …	3 → …
8 → …	5 → …

- malloc: find smallest non-empty freelist larger than requested block size.
- Advantage: merge adjacent free blocks ("buddies") to make free block for next-larger freelist
- O(1) malloc, free! (need doubly-linked freelist)
- Disadvantage: *internal fragmentation* (~27% space wasted)

18

Simple allocator

- A fast allocator that doesn't support **`free`**:

```
structure Allocator :> ALLOCATOR = struct
  (* freelist actually stored in memory *)
  val curr: address ref = ref LOW_MEM
  val memory: Memory.memory = …

  fun malloc(n) = let
    val ret = !curr
  in
    curr := ret + n;
    if curr > HI_MEM then raise OutOfMemory
    else ret
  end
end
```

- Idea: reclaim memory using an automatic garbage collector

19