15-213 Recitation

5 March, 2001

Urs Hengartner

Overview

- Dynamic Memory Allocation
- Garbage Collection
- Lab 3
- Pointer Arithmetic
- `gc_malloc()`

Logistics for Lab 3

- Checkpoint 1: next Tuesday
  - Implement **correct** garbage collector, does not need to provide coalescing of freed memory blocks
- Assigned: Wednesday in two weeks
  - Implement **coalescing**, add optimizations
- Be warned:
  - Not much coding (100 and 50 LOC, respectively), but debugging can be time consuming

Dynamic Memory Allocation

- Memory has to be allocated at runtime if size is unknown at compile time
- Dynamic allocation:

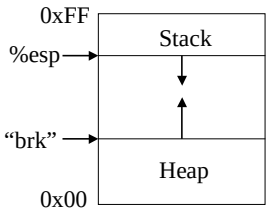
```
int *array = (int*) malloc(len *
sizeof(int));
```

- C (unlike Java) requires progammer to **explicitly** free dynamically allocated memory:

```
free(array);
```

Heap

- Dynamic memory is allocated on the **heap**
- Memory layout (simplified):



5

Garbage Collection

- Garbage: Dynamic memory that can no longer be accessed
- Example:

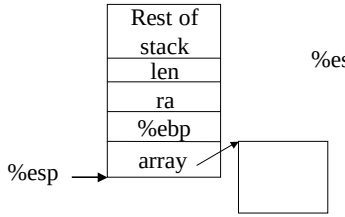
```
void foo(int len)
{
    int *array = (int*)malloc(len*sizeof(int));
}
```

memory block `array` is pointing to becomes garbage after `foo()` returns
- Let memory management detect garbage and collect (free) it **automatically**

6

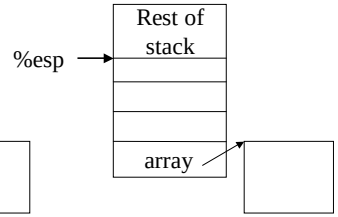
Detecting Garbage

- **While** in `foo()`



- Memory block is reachable by pointer on stack and cannot be collected

- **After leaving** `foo()`



- Memory block is no longer reachable by pointer on stack and can be collected

7

Removing Garbage

- Mark and Sweep-Algorithm (first attempt):
  - Scan **rootset** for pointers pointing to allocated memory block in heap
  - Mark all these memory blocks
  - Free all un-marked memory blocks
- Rootset: Pointers in
  - Stack
  - Registers
  - Global variables
- Are these all the pointers we have to consider?
  - No, allocated memory blocks in heap could also contain pointers (e.g., in a linked list)

8

Mark and Sweep-Algorithm

- Scan rootset for pointers pointing to allocated memory block in heap, for each of these pointers, call `mark(block)`
- `void mark(block)`
  `{`
  If memory block `block` has not already been marked
  - Mark it
  - Scan it for further pointers to allocated memory blocks in heap and call `mark()` for each of them

  `}`
- Free all unmarked memory blocks

9

Detecting Pointers

- How can we detect pointers (to allocated memory block) in global data, stack, registers, and memory blocks?
  - Four bytes-aligned value
  - Value between lower and upper bound of heap
- Things that make detection hard
  - C does not tag pointers, anything could be a pointer (pointer could be assigned to integer,…)
  - C allows pointer arithmetic, that is, pointer does not have to point to beginning of allocated memory block in heap, but to some random location within such a block

10

Conservative Approach

- **Any** four byte-aligned value that points to an allocated memory block in the heap is considered to be a "pointer" (thus memory block won't be freed)
- Such a "pointer" is allowed to point to beginning of allocated memory block or to some random location **within** an allocated memory block

11

Lab 3

- **Given**: `gc_malloc()` which allocates memory from its own heap (`dseg_lo - dseg_hi`)
- **Todo**: add garbage collection to `gc_malloc()`
- **Possible approach**: whenever `gc_malloc()` fails to satisfy memory request, collect garbage and try again
  (Alternative: collect garbage upon every call to `gc_malloc()`)

12

Data structures

- **Free list**: keeps circular linked list of free blocks
- **Splay Tree**: keeps tree of allocated blocks
- Note: free list and splay tree are also allocated in the heap managed by `gc_malloc()` (see Figure 1), `dseg_lo` points to free list and `dseg_lo + sizeof(long)` to root of splay tree

13

Free List

- Free memory blocks in the heap are kept in a circular linked list
- Nodes of list consist of header and the actual free memory block
- Header contains pointer to next and previous free block and size of free block
- Note: order of blocks in list does not correspond to the order implied by the address of a free block
- Use `gc_add_free()` and `gc_remove_free()` to add/remove free blocks to/from free list

14

Splay Tree I

- Allocated memory blocks in the heap are kept in a splay tree
- Nodes of tree consist of header and the actual memory block
- Header contains pointer to left and right child and size of allocated block
- Nodes are ordered based on their memory address (binary search tree)
- Use `insert()` and `delete()` to add/remove blocks to/from splay tree

15

Splay Tree II

- Use `contains()` for querying whether "pointer" points to allocated memory block in heap (either to its beginning or to some location within)
- **Notes**:
  - Size stored in header does not include size of header
  - Size of header of a splay tree node is identical to size of header of a free list node
  - `insert()`, `remove()`, and `contains()` re-balance splay tree, that is, have to reset pointer to root (`dseg_lo + sizeof(long)`) after each call

16

Marking

- How does marking work?
- We could allocate another bit in the header part of a splay tree node and use it as mark
- **Observation**: size of allocated block is always multiple of eight bytes (convention), thus lower three bits of size entry in header are always zero
- Use lowest bit for marking

17

Pointer Arithmetic I

- Take a look at slides from first recitation
- Do not use `void*` pointers for pointer arithmetic
- `ptr[i]` is the same as `ptr+i`, actual number of bytes added to `ptr` depends on type of pointer:

```
int *int_ptr = 0;
char *char_ptr = 0;
int_ptr++;
/* int_ptr is now 4 */
char_ptr++;
/* char_ptr is now 1 */
```

18

Pointer Arithmetic II

- Use type casting to convert between different types of pointers (`list_t*`, `Tree*`, `ptr_t`, `char*`, `void*`)
- Example:

```
/* tree is pointer to unmarked node in
   splay tree, thus it can be freed */
Tree *tree = …;
/* put it into free list, second
   argument to gc_add_free() has to be
   of type list_t* */
gc_add_free(dseg_lo, (list_t*) tree);
```

19

`gc_malloc()`

- 15-16: Get callee-save registers and current frame pointer
- 19: Round up requested size to a power of 8 (similar to `malloc()`)
- 22-34: Get pointer to free list stored at `dseg_lo` and search for memory block that is big enough (**first fit**)
- 37-49: If there is no such block, increase upper boundary of heap (`dseg_hi`) and put new memory block into free list

20

gc_malloc()

- 53: Test whether found (or new) block is big enough for a split
- 54-63: If not, return entire block
  - Remove it from free list
  - Insert it into splay tree
  - Fix pointer to root of splay tree stored at `dseg_log + sizeof(long)`
- 66-87:If so, split block and return first part
  - Create new free list block consisting of second part and replace found entry with this new block
  - Update size entry in header of found entry and put it into splay tree (fix pointer to root)

How to proceed

- Make sure you understand everything I just explained
- Read through handout, ask your TA if there is something you don't understand
- Study `malloc.c` and comments in `splay_tree.c`
- Implement garbage collection algorithm outlined earlier
- Checkpoint 1 does not require coalescing, just call `gc_add_free()` for each freed block