Scribe Notes 9/25: Memory Allocation && Conservative Garbage Collector

Announcements
- Code for lectures, problem sets, and section all available at [https://code.seas.harvard.edu/cs61](https://code.seas.harvard.edu/cs61)
  - To grab code for lectures, use git://code.seas.harvard.edu/cs61/cs61-lectures.git
- Start on Assignment 1! *(Kevin Schmid* is not done with Assignment 1!)
- Piazza
  - Do not post code to Piazza anonymously
  - Make sure to use the “preformatted” toggle when entering code snippets in a Piazza question to keep spaces and indentations

Storage Durations
- Advantages and disadvantages of each of STATIC, AUTOMATIC, and DYNAMIC

<table>
<thead>
<tr>
<th>STATIC (globals)</th>
<th>AUTOMATIC (locals, stack)</th>
<th>DYNAMIC (heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ simple</td>
<td>+ simple</td>
<td>- complicated</td>
</tr>
<tr>
<td>+ managed by compiler</td>
<td>+ managed by compiler</td>
<td>- managed by programmer (much easier to have memory errors)</td>
</tr>
<tr>
<td>+ cheap</td>
<td>+ cheap</td>
<td>- nameless objects, require pointers</td>
</tr>
<tr>
<td>+ lifetime is program execution (no memory errors!)</td>
<td>- lifetime bounded by function</td>
<td>+ flexible! lifetime determined by code</td>
</tr>
<tr>
<td>- fixed size determined at compile time, not dependent on input</td>
<td>~ number of stack frames dependent on input (e.g. recursion)</td>
<td>+ size can be totally input dependent</td>
</tr>
</tbody>
</table>
C vs. other “higher” level langs
- C - explicit dynamic storage duration
  - e.g. malloc() or free()
- PHP, ML, Java, etc. - implicit dynamic storage duration
  - garbage collection - language takes care of allocations and frees
    - Programmer does not have to take care of memory manipulation
  - + no memory leaks (for the most part)
  - + no double frees

What is Garbage Collection?
1. track every dynamic object
2. find all accessible objects (difficult process!)
3. free all inaccessible objects

   e.g. example.c

   int main () {
     int *x = malloc(4);
     *x = 0;
     x = NULL;
   }

   In this example, the memory allocated is now no longer referenced by any pointers in the program. That is, once we set x to NULL, we no longer have a pointer to the allocated 4 bytes of memory we asked for from Malloc. Therefore, that block of memory should be freed. Garbage Collection will accomplish this automatically.

- What is conservative garbage collection?
  - Being extra sure to collect all the memory that we can, without much regard to efficiency
  - Today, we will build a conservative garbage collector

Examples using singly and circularly linked lists:

In this first example of a singly linked list, all the nodes are accessible since we have a root pointer pointing to the heap, specifically to the beginning of the linked list.
In this second diagram, we see that if we instead have a pointer to another node in the linked list (not necessarily the first) the earlier nodes in the linked list become inaccessible.
Finally, if we were to implement a circularly linked list and removed the root pointer to it, it then becomes inaccessible, even though we have pointers pointing to each of the nodes.

Which Objects are Inaccessible?

- Let $A$ be the set of all allocated objects
- Let $R$ be a set of roots (pointers in the stack that point to the heap)
- Find $A$
- Then $R \subset A' \subset A$, s.t. $A'$ is closed and self-contained
  - Note that $A'$ is the set of all accessible objects
  - Note that Garbage Collection frees the set difference $A - A'$
Example: whichdict.c

whichdict is a program that takes DICTFILE and WORD and reports which dict contains each word

Memory

Structure:

In the figure above, for any dictionary that we read in

- ptr *d is a pointer that points to the entire dictionary
  - This is the root -- the only root. Why? This pointer allows us to access all of dictionary. Without it, we cannot access any of dictionary.
  - It points to a structure that keeps track of all the words in the dictionary, and the number of words
- We keep track of all the words with an array filled with the following data: dictionary name, line number, and a pointer to a string that contains the actual word (ptr *word)
- If we ever set ptr *d to NULL, we lose access to the entire dictionary. At this time, the garbage collector should come “sweep up” all the memory.
  - In a similar vein, if we ever set the ptr *word for “hi” to NULL, the GC should come by and sweep up the memory that contains “hi,” as it is no longer accessible.

In order to help us see what’s going on with the memory management, we can write a little bit of Assignment 1 to report number of total and active allocations.
What Do We Need for a Garbage Collector?

- instead of using linked list of metadata, try using a large array that looks like so

```
<table>
<thead>
<tr>
<th>MR (pointer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTR</td>
</tr>
<tr>
<td>PTR</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

- where each cell is a memregion, consisting of a pointer and a size
- have as many memregions as allocations requested
- How do you record allocated pointers in this array? Look at the following code:

```c
void m61_malloc(size_t sz){
    void *ptr=malloc(sz)
    ...
    mr[nmr].ptr=ptr
    mr[nmr].sz=sz;
}
```

- We must use arrays to reference the blocks further in the array. This code in particular references the block one down from the most currently allocated region.
- Next, we add function that finds ptr that you are trying to free

```c
memregion *m61_find_mr(char *ptr){
    for (size_t i=0; i!=nmr; ++i)
        if(ptr >= mr[i].ptr && ptr < mr[i].ptr +mr[i].sz)
            return &mr[i];
    return NULL;
}
```

- Once you find this pointer, how do you free it?

```c
void m61_free(void *ptr){
    memregion *m61_find_mr(ptr);
    if(m){
        memcpy(m, &m[nmr-1], sizeof(memregion));
        --nmr;
    }
    free(ptr);
}
```
The code above simply copies the data from the last element in the array over the pointer you are trying to free. This way, you don't have to move every element in the array down.

Next, we write function called `m61_print_accessible(char *ptr, size_t sz)` which goes over contents in memory and prints all the pointers it finds
- check every byte to see if it is the start of a pointer address
- usually not necessary to check for overlapping ranges, but we do in case someone packs their pointers very closely
- char *address points to the beginning of one of 4-byte chunks
- check every byte up through sz – 4 because otherwise you check past the length of the array (since each pointer address takes up 4 bytes), called nasal demons
  - this kind of memory violation can lead the compiler to do anything
- if had circularly linked structure, it would go forever, so need to stop when encounter a pointer that has already been seen
- to get address of pointer, cast the 4-byte chunk to a char * and dereference with *

Depth-First Search
- Iterate through, marking a pointer each time that encounter one that hasn’t been seen yet, and traverse that pointer when it is encountered by recursively calling the function on that found pointer

Write function called `mark_accessible(char *ptr, size_t sz)` to mark accessible parts of heap
- start out with all of memory region as unmarked
- mark all accessible regions, then free anything that wasn’t marked
- only going to do the stack, although globals would be covered with a really conservative garbage collector
  - address of function is not in the stack, but rather in globals
  - however, local variables created in a function live in the stack
- bottom of stack is address of local variable in main and top of stack is address of local variable in other function
  - useful for knowing what region of address to call `m61_mark_accessible` on
  - create arbitrary local variable at top of main function to have a global variable with address at bottom of stack
- garbage collector is looking for accessible objects, and the accessible variables in main (variables in the heap) are stored on the stack
- when call garbage collector on `whichdict`, still thinks that there are 3 allocated objects
  - in main the dictionary pointer d created in line `dict *d = dict_new()` allows us to access dynamically allocated memory
  - to make the dictionary unaccessible, set pointer to dictionary, d, to NULL
  - a stack frame has local variables and arguments, and the local variable in main is d
  - using large dictionary and searching for word at end of dictionary causes garbage collector to take a long time to exit
  - conservative garbage collector doesn’t know where the pointers are, so it needs to search through memory

Garbage collection is often much slower than explicit dynamic memory management
- conservative garbage collection is especially slower
- in Java, if it knows that only strings have been allocated, then can run much faster

- **Experiment** - Run whichdict with really big dictionary and search for word at end of dictionary
  - garbage collector takes a long time to exit
  - run whichdict using gdb to see where the program is at various points in time
    - finds zygote quickly, but then continues to run
    - going over list of all allocated objects
    - run program using retard sampling (run program many times and break out of program to see where the program spends most of its time)
    - using backtrace function (which examines previous functions on the call stack), notice that when break out of dictionary, still only checking words in the “a” range
    - since majority of “pointers” we find in array are fake pointers, we run find_mr way too many times
    - if sort pointers using binary search, takes 5 seconds to complete whichdict on call to word zygote
    - however, on version of program with explicit memory management, takes 0.1 seconds
    - why is garbage collection expensive?
      - explicit version knows where all the pointers are, but a conservative garbage collector doesn’t know where the pointers are

- How can we change the representation of a dictionary so that it’s faster?
  - hash table would speed up look-ups but not creation of dictionary
  - malloc is one of slowest things you can do
    - requires interacting with operating system and changes various aspects of the machine
    - needs to handle crazy memory situations as well
  - how can we avoid some of these allocations?
    - malloc single enormous string to contain the whole file and point into that object at different places
    - this is allocating the same amount of memory as before, but in fewer units
    - this modification reduces whichdict time by ~60%
      - less malloc calls, and when free object, just need to free one object
      - word pointers still exist, they just point to different addresses within the same block of memory