Managing your data

Mayer Goldberg

January 22, 2018

Contents

1 The issue

There are many ways to handle the issue of allocating memory. In this document I want to explain in detail two possible choices you have, the rationale behind each choice, and to recommend a course for your project. Of course, you are free to do anything that works and makes sense to you, as we shall be testing your compiler as black boxes.

The key constraints we have are:

- We are using tagged memory (tagging it with *run-time type information*, RTTI).
- The RTTI takes up 4 bits, so we have less room for our actual data. The size of the RTTI data is determined by the *number of different types* we need to distinguish. You can have a look at the file `scheme.s`, which I posted, and see that 4 is the bare minimum.
- In the design I offered you, we are representing pairs as a single 64-bit. You are not obligated to follow this design, but it's nice, and works.
- If you follow my design, you will have 30-bit data pointers, as I have explained in class, and so the size of your data will be limited to 1Gbyte.

2 Using `malloc`

You can choose to use the procedure `malloc` that is available with `gcc`. You've used `malloc` before, and are familiar with how to use it in C. If you choose to use `malloc` from assembly, you need to be aware of the following issues:

- If you call `malloc`, you will need to save your registers before the call, and restore them after the call, as `malloc` changes many of them. One thing you can do is write your own procedure `safe_malloc`, which does just that, and places in `rax` a pointer to the memory it allocated.
- Allocating small amounts of memory is wasteful, because allocation is done in blocks of $\texttt{blockSize} * 2^n$, for various values of n. So you won't get the number of bytes you asked for but in fact, the smallest chuck that will contain the number of bytes you asked for. The problem is that you don't have a *garbage collector*, and you only have 1Gbyte of data with which to play, and you shall be calling `malloc` quite often, so a wasteful solution.

If you want to use `malloc`, then:

- Allocate 1 byte initially, and use the value of the pointer as the `start_of_data`.
- Change the macros in my code (or write your own) that use this value of `start_of_data` as the base, i.e., subtract or add `start_of_data` to each data pointer. This means that while `malloc` will allocate memory within a 64-bit address space, you will only keep the *delta*, the difference from `start_of_data`, and if you can keep everything within 1Gbyte, you'll be fine.

3 Not using `malloc`

You can choose not to use the procedure `malloc`, and managing memory on your own. This isn't as difficult as it sounds; In fact, you may be surprised to learn that it's actually easier than using `malloc`:

- Keep in mind that you're not actually planning on `free`-ing any memory. This means that no bookkeeping and no special data structures are really necessary.

- [Consequently], You can portion chunks of memory with absolutely no waste! This will let you use your 1Gbyte more effectively.
- No surprises with `malloc` overwriting your registers.

If you want to manage your own memory, here's what you need to do:

- Create macros for chunks of memory, e.g., `gigabyte(n)`, `megabyte(n)`, `kilobyte(n)`. They take *literals*, and simply multiply by the appropriate power of 2.
- In the `.bss` section, have as the first label:

```
malloc_pointer:
  resq 1
start_of_data:
  resb gigabyte(1)
```

- Then, before you use this memory, execute the following instructions:

```
mov rax, malloc_pointer
mov qword [rax], start_of_data
```

  These will guarantee that `malloc_pointer` will point to the next available byte.

- To allocate n bytes, all you need to do is

```
mov rbx, malloc_pointer
mov rax, qword [rbx]
add qword [rbx], n
```

  If you like, you can wrap this up as an assembly-language procedure called `my_malloc`.

4 Final reocmmendations

Under our given constraints, managing your own memory, i.e., writing your own `malloc`, is simpler and more economic, since there is no need to deallocate memory. Your data will be more compact, and you will get more objects in the same 1Gbyte limit.