Section 13: Garbage Collection

A great resource on garbage collection is Paul R. Wilson's Uniprocessor Garbage Collection Techniques. It's a very readable introduction to some very important techniques! *You are **not** required to read this paper, it is purely supplemental!*

1. Memory Regions

Consider the following C code (https://godbolt.org/z/EaEc9E3j6):

```c
#include <stdlib.h>

int a = 0;

int main() {
   int b = foo() + 1;
   int* d = (int*)malloc(sizeof(int));
   *d = b + 1;
   a = *d + 1;
   free(d);
   return a;
}

int foo() {
   int c = 0;
   return c;
}
```

1.1. Which memory region (stack, heap, global) do the variables live in?

a: _____ b: _____ c: _____ *d: _____

1.2. For each variable above, mark the point in the source code where memory for it is **allocated**.

1.3. For each variable above, mark the point in the source code where memory for it is **freed**.

2. Bugs with Manual Memory Management

For each of the following programs:

1. Describe in your own words what is going wrong.
2. Is there a common name for this kind of bug?
3. Would the same type of bug be possible if `d` was allocated on the stack instead of the heap?

Program 1

(https://godbolt.org/z/f4a6Wd7e4)

```
int main() {
   int* d = (int*)malloc(sizeof(int));
   *d = 2;
   free(d);
   return *d;
}
```

Program 2

(https://godbolt.org/z/9bG1Gxq4r)

```
int main() {
   int* d = (int*)malloc(sizeof(int));
   *d = 2;
   free(d);
   free(d);
}
```

Program 3

(https://godbolt.org/z/77cneoG3e)

```
int main() {
   int* d = (int*)malloc(sizeof(int));
   *d = 2;
   return *d;
}
```

Program 4

(https://godbolt.org/z/ePnsnYhzd)

```
int main() {
   int* d = (int*)malloc(sizeof(int));
   d[0] = 2;
   d[1] = 2;
   free(d);
}
```

Task 3: Reference Counting

The following C program (https://godbolt.org/z/fn1e81ozf) has memory leaks. In this task, we will investigate how we could add a form of garbage collection known as *reference counting* to automatically free memory.

```c
typedef struct node {
   int value;
   struct node* a;
   struct node* b;
} node;


node* create(int value) {
   node* new = malloc(sizeof(node));
   new->value = value;
   return new;
}

void print(node* n) {
   // ....
}
```

```c
node* n0;
int value0 = 0;

int main() {
    n0 = create(value0);
    n0->a = create(1);
    n0->b = create(2);
    n0->a->a = n0->b;
    n0->b->b = create(3);
    n0->b->b->b = create(4);
    n0->b->b->b->b = create(5);
    node* leaf = n0->b->b->b->b;
    print(leaf);
    leaf = NULL;
    node* n1 = n0->a;
    n0 = NULL;
    n1->a->a = n1;
    n1->a->b = NULL;
    n1 = NULL;
}
```

	counter for node					
	0	1	2	3	4	5

3.1. Draw a diagram of the object graph that is created by the `main` function. Label each node with its value and ignore any pointers that are `NULL`.

3.2. Now, let us assume that every object is automatically extended to include a `counter` field that r**ecords the number of pointers that are currently pointing to it**. For each line in the main function, annotate how it affects the counts of each heap object (labeled 0–5). Moreover, if a pointer counter drops to zero, **decrement the counter of what it points to**.

3.3. When a counter reaches zero, the object is deallocated. For each node (0–5) mark the line on which it would be deallocated.

3.4. Do we still have a memory leak? Why or why not?

Task 4: Tracing Garbage Collection

Instead of using reference counting, we can periodically **stop** program execution, find all objects that are no longer "live" (i.e., accessible from the *root set*, which are global pointers and pointers living on the stack) and free them. In this task we will analyze the same source code as in Task 3.

4.1. Assume we pause execution while the main function is executing. Which variables form the root set?

4.2. If execution is paused while we create node 5 (during the call to `create(5)`), how does that change the root set?

4.3. What is the earliest point in the program at which node 4 will be garbage collected? What is the latest?

4.4. What is the earliest point in the program at which node 1 will be garbage collected? What is the latest?

4.5. Assuming we have full knowledge of the program execution including statements that haven't executed yet (which is something a runtime will never have!) what would be the earliest point at which we could free node 4 without changing the visible behavior of the program?

5. Mark-and-Sweep vs. Mark-and-Copy

In a so-called *mark-and-sweep* garbage collection scheme, we simply call `free` on the objects which were found to be unreachable. This works great for our C example where we essentially just shift the burden of calling `free` from the user to our garbage collector.

5.1. What are the (negative) implications of this scheme for memory accesses and allocations?

An alternative strategy that avoids this negative effect is known as *mark-and-copy*. It works by dividing the memory into two "semispaces," the *fromspace* and the *tospace*. After the marking phase, the garbage collector copies the live objects in the fromspace to the tospace, then the fromspace and tospace switch roles. Here are a couple of hints for how this works in practice:

1. The root set is initially copied to the tospace.
2. The garbage collector does a linear scan of the tospace looking for object pointers, copies over any objects that it sees are pointed to, and updates the pointer to the new location.
3. To keep track of what it has already copied, it overwrites copied objects in the fromspace with pointers to the object in the tospace.

5.2. Try out the algorithm below assuming a root set of { A, E }!

fromspace	tospace
`A: { next: B }`	
`B: { next: C }`	
`C: { next: A }`	
`D: { next: E }`	
`E: { next: F }`	
`F: { next: H }`	
`G: { next: D }`	
`H: { next: NULL }`	
`I: { next: I }`	

5.3. When might we choose mark-and-sweep over mark-and-copy?

5.4. The *generational hypothesis* posits that most objects become unused very quickly, but a small number live for a very long time. Assuming the hypothesis is true, how might we adapt the mark-and-copy scheme to exploit this fact?