Assignment 6 - Heap Management Simulator

Introduction

Running programs on a computer requires memory. Operating systems comprise a memory management service that provides transparent memory access: in particular allocation and deallocation. For this assignment, you will design and implement your own heap management simulator.

Memory allocation

In order to manipulate data along with the execution of code, a user must acquire a zone of memory to store the data. The `malloc` primitive fulfills this request: given an amount of memory required by the user, it uses a predetermined strategy to find a zone of contiguous free memory, allocates it to the user, updates the set of free zones accordingly, and returns a pointer to the newly allocated zone.

The memory allocation call has the following format:

```
void* heap_malloc(int size);
```

where `<size>` represents the amount of memory the user wishes to acquire, and the returned address points to the first word of memory that is writable in the allocated zone. If there isn't enough free memory to satisfy the request, then the call returns NULL.

Memory allocation strategies

Upon handling an allocation request, the memory management service must find a zone of contiguous free memory that is large enough to contain the resulting allocated zone.

The generic call for searching a suitable free zone has the following format:

```
void find_free_zone(int size, freezone* fz);
```

where `<size>` represents the amount of memory the user wishes to acquire, and `<fz>` is a pointer to the result of the search. The result is a couple of index values: the first index value corresponds to the first cell of the free zone in the heap array that is now targeted for allocation, while the second index value corresponds to the first cell of the free zone in the heap array that precedes the zone targeted for allocation.

Memory deallocation

In order to increase the amount of free memory available, a user can deallocate a previously acquired zone of memory. The `free` primitive adds the deallocated zone to the set of free zones. After this call, the user cannot access the deallocated zone anymore.

A memory deallocation call has the following format:

```
int heap_free(void *dz);
```

where `<dz>` points to the first word of memory that is writable in the allocated zone. The call returns 0 if the deallocation is successful, -1 otherwise.

Memory compaction (*Bonus*)

Successive allocations and deallocations may lead to external fragmentation: the set of free memory zones contains a significant number of non contiguous zones that are too small to satisfy allocation requests. The memory management service can solve external fragmentation via *compaction*. It repositions every allocated memory zone so that they become contiguous. In doing so, the service reshapes the set of free memory into a single contiguous free zone, thus improving the response for future allocation requests. Of course, this means that the service must also update all of the pointers to zones it has previously allocated.

A memory compaction call has the following format:

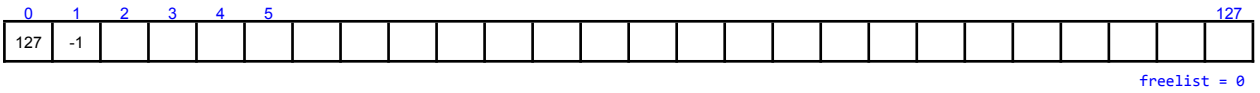
```
int heap_defrag();
```

and returns the size of the unique free zone that results from the compaction.

Heap management model

For simulation purposes, we model the heap as a static array of `HEAP_SIZE` characters. Every memory zone in the heap consists of a set of contiguous array cells. The first cell of every zone is reserved for metadata: it contains the size of the zone.

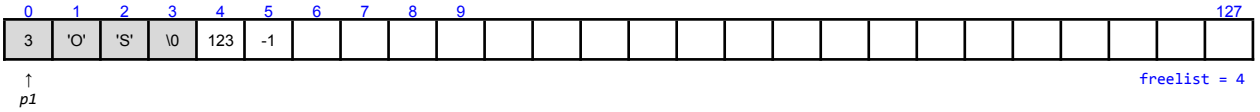
In the case of a free zone, the second cell contains the index of the next free zone in the array; in the last free zone, the second cell contains -1. A variable `freelist` keeps track of the first free memory zone. Initially, that is when no allocation has been carried out yet, a heap of capacity `HEAP_SIZE=128` looks as follows:



In the case of an allocated zone, data can be stored after the cell reserved for metadata, from the second cell onwards to the last. Consider the following code:

```
p1 = (char*) heap_malloc(3);
strcpy(p1, "OS");
```

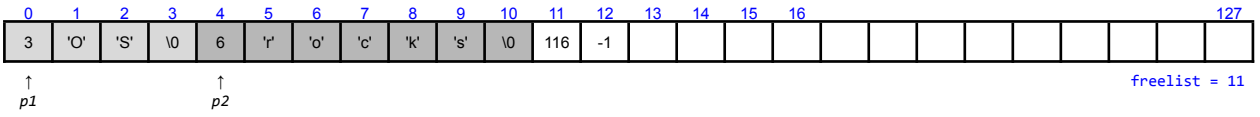
After the execution of the above code, the 128-words heap looks as follows:



Let's carry out a second allocation, as a result of the following code:

```
p2 = (char*) heap_malloc(6);
strcpy(p2, "rocks");
```

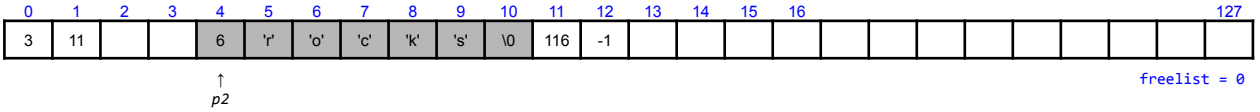
The heap now looks as follows:



Now let's deallocate a zone, as a result of the following code:

```
heap_free(p1);
```

The heap now looks as follows:



Note that this model doesn't allow the heap management service to prevent buffer overflows. For instance if the user requests a new allocation for 2 characters and then writes 5 in the allocated zone, the damage to the metadata cannot be repaired. *In this assignment, we assume that user code never causes buffer overflows.*

Coding canvas

You can download a coding canvas for the assignment: heap-simulator-canvas.tgz

It contains the following files:

1. `Makefile`
2. `include/os-memory.h`
3. `src/memory-simulator.c`
4. `src/memory-simulation-run.c`
5. `src/memory-management.c`
6. `output.txt`

`Makefile`	Provides commands to compile and run the simulator. *You may only modify it to change the memory allocation strategy by uncommenting the corresponding line.*
`include/os-memory.h`	Header file with elements to get you started: macros, constants, data structures, and signatures for functions. *You are allowed to add more elements, but not to modify existing ones.*
`src/memory-simulator.c`	Internal code of the simulator, including the main function that runs the program. *You may not modify this file.*
`src/memory-simulation-run.c`	contains a single function, `run_simulation()`, which defines the test simulation to be performed when running the program. *You can rewrite the code of `run_simulation()` as much as you like.*
`src/memory-management.c`	Source code of the memory management functions. ***Your assignment is mostly about writing the missing code of the functions inside this file.*** *You are allowed to add more functions.*
`output.txt`	Output that should result from running the default simulation code of function `run_simulation()` in file `src/memory-simulation-run.c`

Objectives

The memory management service uses the *first fit* strategy by default. To complete this assignment, you must implement **two more strategies: *best fit*, and *worst fit***. Additionally, you must implement the **memory deallocation call**.

As a bonus, you can also implement the **memory compaction call**. Since this is a bonus question, you will get full points for handing in your assignment with correct solutions for allocation strategies and for deallocation only. If you provide a solution for compaction, the associated points will be added to your total, but the total cannot exceed the maximum. In other words, compaction is a bonus question in the sense that it acts as a safety net for your assignment grade.