Memory Allocator

Objectives

- Learn the basics of memory management by implementing minimal versions of `malloc()`, `calloc()`, `realloc()`, and `free()`.
- Accommodate with the memory management syscalls in Linux: `brk()`, `mmap()`, and `munmap()`.
- Understand the bottlenecks of memory allocation and how to reduce them.

Statement

Build a minimalistic memory allocator that can be used to manually manage virtual memory. The goal is to have a reliable library that accounts for explicit allocation, reallocation, and initialization of memory.

Support Code

The support code consists of three directories:

- `src/` will contain your solution
- `tests/` contains the test suite and a Python script to verify your work
- `utils/` contains `osmem.h` that describes your library interface, `block_meta.h` which contains details of `struct block_meta`, and an implementation for `printf()` function that does **NOT** use the heap

The test suite consists of `.c` files that will be dynamically linked to your library, `libosmem.so`. You can find the sources in the `tests/snippets/` directory. The results of the previous will also be stored in `tests/snippets/` and the reference files are in the `tests/ref/` directory.

The automated checking is performed using `run_tests.py`. It runs each test and compares the syscalls made by the `os_*` functions with the reference file, providing a diff if the test failed.

API

1. `void *os_malloc(size_t size)`

   Allocates `size` bytes and returns a pointer to the allocated memory.

   Chunks of memory smaller than `MMAP_THRESHOLD` are allocated with `brk()`. Bigger chunks are allocated using `mmap()`. The memory is uninitialized.

   - Passing `0` as `size` will return `NULL`.

2. `void *os_calloc(size_t nmemb, size_t size)`

   Allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory.

   Chunks of memory smaller than `page_size` are allocated with `brk()`. Bigger chunks are allocated using `mmap()`. The memory is set to zero.

   - Passing `0` as `nmemb` or `size` will return `NULL`.

3. `void *os_realloc(void *ptr, size_t size)`

   Changes the size of the memory block pointed to by `ptr` to `size` bytes. If the size is smaller than the previously allocated size, the memory block will be truncated.

   If `ptr` points to a block on heap, `os_realloc()` will first try to expand the block, rather than moving it. Otherwise, the block will be reallocated and its contents copied.

   When attempting to expand a block followed by multiple free blocks, `os_realloc()` will coalesce them one at a time and verify the condition for each. Blocks will remain coalesced even if the resulting block will not be big enough for the new size.

   Calling `os_realloc()` on a block that has `STATUS_FREE` should return `NULL`. This is a measure to prevent undefined behavior and make the implementation robust, it should not be considered a valid use case of `os_realloc()`.

   - Passing `NULL` as `ptr` will have the same effect as `os_malloc(size)`.
   - Passing `0` as `size` will have the same effect as `os_free(ptr)`.

4. `void os_free(void *ptr)`

   Frees memory previously allocated by `os_malloc()`, `os_calloc()` or `os_realloc()`.

   `os_free()` will not return memory from the heap to the OS by calling `brk()`, but rather mark it as free and reuse it in future allocations. In the case of mapped memory blocks, `os_free()` will call `munmap()`.

5. General

   - Allocations that increase the heap size will only expand the last block if it is free.
   - You are allowed to use `sbrk()` instead of `brk()`, in view of the fact that on Linux `sbrk()` is implemented using the `brk()`.
   - Do **NOT** use `mremap()`
   - You must check the error code returned by every syscall. You can use the `DIE()` macro for this.

Implementation

An efficient implementation must keep data aligned, keep track of memory blocks and reuse freed blocks. This can be further improved by reducing the number of syscalls and block operations.

Memory Alignment

Allocated memory should be aligned (i.e. all addresses are multiple of a given size). This is a space-time trade-off because memory blocks are padded so each can be read in one transaction. It also allows for atomicity when interacting with a block of memory.

All memory allocations should be aligned to **8 bytes** as required by 64 bit systems.

Block Reuse

`struct block_meta`

We will consider a **block** to be a continuous zone of memory, allocated and managed by our implementation. The structure `block_meta` will be used to manage the metadata of a block. Each allocated zone will comprise of a `block_meta` structure placed at the start, followed by data (**payload**). For all functions, the returned address will be that of the **payload** (not of the `block_meta` structure).

```
struct block_meta {
    size_t size;
    int status;
    struct block_meta *prev;
    struct block_meta *next;
};
```

Note: Both the `struct block_meta` and the **payload** of a block should be aligned to **8 bytes**.

Note: Most compilers will automatically pad the structure, but you should still align it for portability.

Split Block

Reusing memory blocks improves the allocator's performance, but might lead to Internal Memory Fragmentation. This happens when we allocate a size smaller than all available free blocks. If we use one larger block the remaining size of that block will be wasted since it cannot be used for another allocation.

To avoid this, a block should be truncated to the required size and the remaining bytes should be used to create a new free block.

The resulting free block should be reusable. The split will not be performed if the remaining size (after reserving space for `block_meta` structure and payload) is not big enough to fit another block (`block_meta` structure and at least **1 byte** of usable memory).

Note: Do not forget the alignment!

Coalesce Blocks

There are cases when there is enough free memory for an allocation, but it is spread across multiple blocks that cannot be used. This is called External Memory Fragmentation.

One technique to reduce external memory fragmentation is **block coalescing** which implies merging adjacent free blocks to form a contiguous chunk.

Coalescing will be used before searching for a block and in `os_realloc()` to expand the current block when possible.

Note: You might still need to split the block after coalesce.

Find Best Block

Our aim is to reuse a free block with a size closer to what we need in order to reduce the number of future operations on it. This strategy is called **find best**. On every allocation we need to search the whole list of blocks and choose the best fitting free block.

In practice, it also uses a list of free blocks to avoid parsing all blocks, but this is out of the scope of the assignment.

Note: For consistent results, coalesce all adjacent free blocks before searching.

Heap Preallocation

Heap is used in most modern programs. This hints at the possibility of preallocating a relatively big chunk of memory (i.e. **128 kilobytes**) when the heap is used for the first time. This reduces the number of future `brk()` syscalls.

For example, if we try to allocate 1000 bytes we should first allocate a block of 128 kilobytes and then split it. On future small allocations, we should proceed to split the preallocated chunk.

Note: Heap preallocation happens only once.

Building Memory Allocator

To build `libosmem.so`, run `make` in the `src/` directory:

```
student@os:~/.../mem-alloc$ cd src/
student@os:~/.../mem-alloc/src$ make
gcc -fPIC -Wall -Wextra -g -I../utils  -c -o osmem.o osmem.c
gcc -fPIC -Wall -Wextra -g -I../utils  -c -o ../utils/printf.o ../utils/printf.c
gcc -shared -o libosmem.so osmem.o helpers.o ../utils/printf.o
```

Testing and Grading

Testing is automated. Tests are located in the `tests/` directory:

```
student@so:~/.../mem-alloc/tests$ ls -F
Makefile  grade.sh@  ref/  run_tests.py  snippets/
```

To test and grade your assignment solution, enter the `tests/` directory and run `grade.sh`. Note that this requires linters being available. The easiest is to use a Docker-based setup with everything installed, as shown in the section "Running the Linters". When using `grade.sh` you will get grades for correctness (maximum `90` points) and for coding style (maximum `10` points). A successful run will provide you an output ending with:

```
### GRADE


Checker:                                                            90/ 90
Style:                                                              10/ 10
Total:                                                             100/100


### STYLE SUMMARY


```

Running the Checker

To run only the checker, use the `run_tests.py` script from the `tests/` directory.

Before running `run_tests.py`, you first have to build `libosmem.so` in the `src/` directory and generate the test binaries in `tests/snippets`. You can do so using the all-in-one `Makefile` rule from `tests/`: `make check`.

```
student@os:~/.../mem-alloc$ cd tests/
student@os:~/.../mem-alloc/tests$ make check
gcc -fPIC -Wall -Wextra -g -I../utils  -c -o osmem.o osmem.c
gcc -fPIC -Wall -Wextra -g -I../utils  -c -o helpers.o helpers.c
gcc -fPIC -Wall -Wextra -g -I../utils  -c -o ../utils/printf.o ../utils/printf.c
[...]
gcc -I../utils -fPIC -Wall -Wextra -g -o snippets/test-all snippets/test-all.c -
gcc -I../utils -fPIC -Wall -Wextra -g -o snippets/test-calloc-arrays snippets/te
gcc -I../utils -fPIC -Wall -Wextra -g -o snippets/test-calloc-block-reuse snippe
gcc -I../utils -fPIC -Wall -Wextra -g -o snippets/test-calloc-coalesce-big snipp
gcc -I../utils -fPIC -Wall -Wextra -g -o snippets/test-calloc-coalesce snippets/
gcc -I../utils -fPIC -Wall -Wextra -g -o snippets/test-calloc-expand-block snipp
[...]
test-malloc-no-preallocate       ........................ passed ...   2
test-malloc-preallocate          ........................ passed ...   3
test-malloc-arrays               ........................ passed ...   5
test-malloc-block-reuse          ........................ passed ...   3
test-malloc-expand-block         ........................ passed ...   2
test-malloc-no-split             ........................ passed ...   2
test-malloc-split-one-block      ........................ passed ...   3
test-malloc-split-first          ........................ passed ...   2
test-malloc-split-last           ........................ passed ...   2
test-malloc-split-middle         ........................ passed ...   3
test-malloc-split-vector         ........................ passed ...   2
test-malloc-coalesce             ........................ passed ...   3
test-malloc-coalesce-big         ........................ passed ...   3
test-calloc-no-preallocate       ........................ passed ...   1
test-calloc-preallocate          ........................ passed ...   1
test-calloc-arrays               ........................ passed ...   5
test-calloc-block-reuse          ........................ passed ...   1
test-calloc-expand-block         ........................ passed ...   1
test-calloc-no-split             ........................ passed ...   1
test-calloc-split-one-block      ........................ passed ...   1
test-calloc-split-first          ........................ passed ...   1
test-calloc-split-last           ........................ passed ...   1
test-calloc-split-middle         ........................ passed ...   1
test-calloc-split-vector         ........................ passed ...   2
test-calloc-coalesce             ........................ passed ...   2
test-calloc-coalesce-big         ........................ passed ...   2
test-realloc-no-preallocate      ........................ passed ...   1
test-realloc-preallocate         ........................ passed ...   1
test-realloc-arrays              ........................ passed ...   3
test-realloc-block-reuse         ........................ passed ...   3
test-realloc-expand-block        ........................ passed ...   2
test-realloc-no-split            ........................ passed ...   3
test-realloc-split-one-block     ........................ passed ...   3
test-realloc-split-first         ........................ passed ...   3
test-realloc-split-last          ........................ passed ...   3
test-realloc-split-middle        ........................ passed ...   2
test-realloc-split-vector        ........................ passed ...   2
test-realloc-coalesce            ........................ passed ...   3
test-realloc-coalesce-big        ........................ passed ...   1
test-all                         ........................ passed ...   5

Total:                                                              90/100
```

NOTE: By default, `run_tests.py` checks for memory leaks, which can be time-consuming. To speed up testing, use the `-d` flag or `make check-fast` to skip memory leak checks.

Running the Linters

To run the linters, use the `make lint` command in the `tests/` directory. Note that the linters have to be installed on your system: `checkpatch.pl`, `cpplint`, `shellcheck` with certain configuration options. It's easiest to run them in a Docker-based setup with everything configured:

```
student@so:~/.../mem-alloc/tests$ make lint
[...]
cd .. && checkpatch.pl -f checker/*.sh tests/*.sh
[...]
cd .. && cpplint --recursive src/ tests/ checker/
[...]
cd .. && shellcheck checker/*.sh tests/*.sh
```

Debugging

`run_tests.py` uses `ltrace` to capture all the libcalls and syscalls performed.

The output of `ltrace` is formatted to show only top level library calls and nested system calls. For consistency, the heap start and addresses returned by `mmap()` are replaced with labels. Every other address is displayed as `<label> + offset`, where the label is the closest mapped address.

`run_tests.py` supports three modes:

- verbose (`-v`), prints the output of the test
- diff (`-d`), prints the diff between the output and the ref
- memcheck (`-m`), prints the diff between the output and the ref and announces memory leaks

If you want to run a single test, you give its name or its path as arguments to `run_tests.py`:

```
student@os:~/.../mem-alloc/tests$ python3 run_tests.py test-all
OR
student@os:~/.../mem-alloc/tests$ python3 run_tests.py snippets/test-all
```

Debugging in VSCode

If you are using Visual Studio Code, you can use the `launch.json` configurations to run tests.

Setup the breakpoints in the source files or the tests and go to Run and Debug (`F5`). Select `Run test` script and press `F5`. This will enter a dialogue where you can choose which test to run.

You can find more on this in the official documentation: Debugging with VSCode.

If VSCode complains about `MAP_ANON` argument for `mmap()` change `C_Cpp.default.cStandard` option to `gnu11`.

Resources

- "Implementing malloc" slides by Michael Saelee
- Malloc Tutorial