02335 Operating Systems, Fall 2024
Instructions for Assignment 2:
Memory Management — v.1

September 16, 2024

In this assignment, you will implement a simple memory management component which might be used, e.g. in an embedded system. You will implement two functions: `simple_malloc` and `simple_free`. `simple_malloc` allows you to dynamically allocate memory blocks; `simple_free` is used to free up previously allocated memory blocks.

You will use your solution from Assignment 1 to verify your solution to this assignment. To do that you will need to modify your assignment 1 solution to use the `simple_malloc` and `simple_free` functions you create in this assignment instead of those found in the standard C library.

You must not depend on any memory allocation mechanisms, neither in the standard C library nor elsewhere.

We have provided two build targets for this assignment. The first target, `malloc_check`, mostly contained in `check_mm.c`, is a small program that you may use to test your allocation routines.

The second target, `combined`, which includes the `main.c` file you modified in Assignment 1. You are here provided with a replacement implementation of the `io.c` that you made in Assignment 1.

1 Learning objectives

During this assignment you will be working towards the following learning objectives:

- You can explain how the compiler, assembler and linker are used to create executables.
- You can apply standard programming methodologies and tools such as test-driven development, build systems, debuggers.
- You can explain the role of each component of a compilation tool-chain used in system programming and how the components interact.
- Given reference material, you can implement C programs with pointers, pointer arithmetic, arrays, structs, memory management and low level I/O.
- You can explain in your own words how static, automatic and dynamic memory management can be used to handle memory in C programs.

- You can explain in your own words what a C pointer is, its relation to arrays in C and the result of pointer arithmetic in terms of memory addresses.
- You can explain in your own words how memory leaks can occur in systems with dynamic memory management.

2 Reference material

When reading literature, it is helpful to remember that the AMD64 architecture is known under many synonyms, including AMD64, x86-64, x86_64, EM64T, IA-32e and Intel 64.

See Assignment 1 for reference material on C, Unix and development tools.

For this assignment, you should consult:

- Sections 3 - 3.2 in Tanenbaum and Bos' book.
- The documentation for the `check` libraries for basic unit testing of C programs as found at `https://libcheck.github.io/check/`.

3 Programming Environment

For all the assignments, you are supposed to be programming for a Unix environment. This may in particular be a Linux environment, but any Posix-compliant system (like MacOS) should be feasible.

On DTU Learn you will find instructions for establishing and testing a Unix environment on your platform.

The tests provided in this assignment use the `check` framework for testing. This may not be installed by default on your system.

On Linux systems the test framework may be installed as the package `check`. On MacOS it might be installed with MacPorts or HomeBrew.

4 Source code tree

For this assignment, you are provided with a source code file `assignment_2.tgz` which is organized into various `.c` and `.h` files and a Makefile for building and testing your program.

The files your are given in this assignment are:

- `mm.h`

  Contains the prototypes and definitions you need to use for dynamic memory allocation.

- `mm.c`

  A skeleton file providing the data structure and macros you need to use to implement the two functions `simple_malloc` and `simple_free`.

- `check_mm.c`

  Contains example unit tests testing `simple_malloc` and `simple_free`. You should add more tests as part of the assignment (see below).

- `io.h` and `io.c`

  This header files is the same file provided to you in Assignment 1 whereas the C file is a replacement implementation.

- `main.c`

  This is a place-holder for the `main.c` file you created in Assignment 1. You should replace this with the `main.c` file you made in that assignment and modify this to use the simple memory management which you have implemented.

- `Makefile`

  A makefile for building a test target `malloc_check` and an application target `cmd_int`. There should be no need to modify this file.

4.1 Testing the environment

To test the environment, first make sure you are in the directory where you unpacked the tar file and then type:

```
make
```

This will produce two binaries (programs), `malloc_check` and `cmd_int`. You can build each program individually by typing

```
make <name of executable>
```

This can be useful if, for instance, your combined program doesn't build but you still want to test your malloc implementation.

After successful compilation type:

```
./malloc_check
```

You should see all the example unit tests fail as `simple_malloc` and `simple_free` have not been correctly implemented yet.

To run the actual program, type

```
./cmd_int
```

This program should operate in the same was as in Assignment 1.

5 Working on the assignment

In this assignment you are to implement two functions in `mm.c`: `simple_malloc` and `simple_free`. You will be given the basic data structure for managing the system's memory. You will then have to incorporate these memory routines into your program from Assignment 1.

`simple_malloc` takes a single argument: the size in bytes of the requested memory block. The `simple_malloc` function returns the address of the allocated memory block if successful, or `NULL` if not. `simple_malloc` must ensure that all allocations are aligned to 8 byte addresses — that is, the address must be a multiple of 8. On most computer architectures incorrect alignment can have a huge performance impact and even cause certain instructions to crash. A simple way of guaranteeing alignment, is to allocate slightly more memory than requested and then return a pointer to a 8-byte boundary. By requesting an 8 byte aligned address, pointers on a 64-bit machine will be properly aligned.

`simple_free` takes a single argument: the address of a memory block previously allocated through `simple_malloc`. The `simple_free` function frees the memory block and makes the memory available to subsequent `simple_malloc` calls.

The memory that you will manage is *one* large contiguous region of memory. It is defined by two variables:
`memory_start` which is the address of the first byte of the memory region and `memory_end` which is the address of the byte *following right after the memory memory region.* I.e. the byte at address `memory_end` is **not** part of the memory region.

Note: The values of `memory_start` and `memory_end` are of the standard type `uintptr_t` which is an unsigned integer type big enough to hold a pointer for the given machine architecture. (On a 64-bit machine it will be identical to `uint64_t`.) You should not assume anything about these values except that the the start address does not exceed the end address. Do not make any assumptions about the alignment of the pointers, or the size of the memory region itself. These values may change in our testing.

The memory region is illustrated in figure 1.

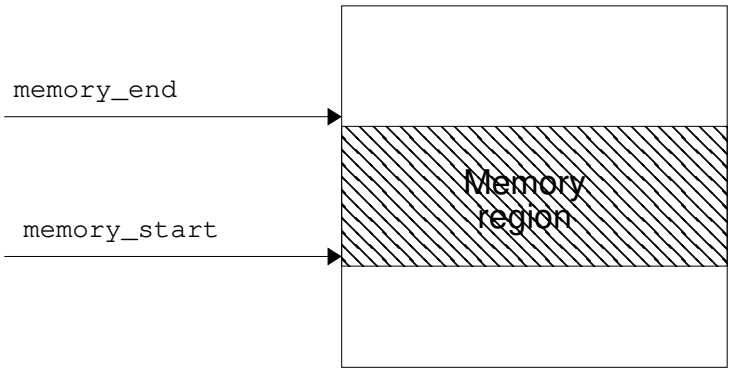

Figure 1: The region of memory you need to manage is defined by two variables.

As described in Tanenbaum and Bos, Chapter 3, memory management can be implemented using many different data structures. In this assignment, you are asked to use a particular data structure which is simple yet having a very small space overhead.

You are going to use a singly-linked, circular list of of memory *blocks*. Each block has a *size* which indicates the number of bytes available for the user to allocate. A block may either be free or in use (allocated, non-free). A block is defined by a *block header* which links to the next, adjacent in memory. The list is made circular by having a *dummy block* at the end of the memory region which points back to the first memory block. The size of the dummy block is 0, and it should always be considered allocated.

The block header should be positioned right before the block of memory allocated to the user. The header should be 8-byte aligned and have size which is a multiple of 8 bytes.

The particular header is given by a struct `BlockHeader` which is optimal in the sense that it will be only 8 bytes long — just enough to hold the next block pointer on a 64-bit machine.

In addition to the next pointer, it must be possible to determine how large the block is. This is accomplished by using the `next` pointer to calculate the size. Furthermore, is must recorded whether the block is free or in use. This is done by (ab)using the least significant bit (bit 0) of the pointer value to hold a *free flag*.

This compact representation of the header information is supposed to be accessed through a set of macros for getting and setting the pointer and the free flag as well as calculating the block size.

The data structure should be initialized upon the first usage of `simple_malloc` by calling the function `simple_init` as shown in the skeleton implementation.

The memory should be allocated according to the *next fit* strategy. For this a `current` pointer is used as a starting point for the next allocation search.

[The memory manager is assumed to be used from a uset program with only one (main) thread. Hence it should not be concerned with *thread safety*.]

5.1 Task 1: Preparatory analysis

Before embarking on the tricky coding of this component, it is of paramount importance to have a good understanding of how the data structure is going to work. For the first task, you should therefore do the following analysis points:

Reporting for Task 1

In the report section for this task, you should address the following points:

1. Explain why bit 0 of the `next` pointer may be used for the free flag. Coult other bits have been used?
2. Show (i.e. give a drawing) how the data structure should look after the initialization. The drawing should show the blocks within the memory region including their headers with values of pointers and free flags.
3. Suppose that the memory region starts at address 0x100 and ends at address 0x180. Explain how you can readily see that these addresses are 8-byte aligned.
4. For the memory region above, assumme that the following sequence of calls calls have been made:

```
a = simple_malloc(32);
b = simple_malloc(20);
c = simple_malloc(10);
simple_free(a);
```

Determine the values obtained for `a`, `b` and `c` (hex values) and make a drawing of the data structure. Remember to take the necessary size alignments into account.

5. Now suppose the following further calls are made:

```
simple_free(b);
d = simple_malloc(40);
e = simple_malloc(4);
```

Determine the values of `d` and `e`.

5.2 Task 2: Implementing the memory manager

In this task, you should implement the memory manager functions in `mm.c` by filling out the provided skeleton following the data structure principles outlined in the beginning of Section 5

Note that the BlockHeader nodes should be accessed only through the given macros. To handle pointers as numbers, they must be cast to the standard type `uintptr_t`.

Reporting for Task 2

In the report section for this task, you should address the following points:

1. Explain on of your macros in detail.
2. Give a brief account of those implementation parts that you have found most challenging.
3. Device a test (extending the given `check`-based test suite) that demonstrates the the memory management system does **not** use a *first fit* strategy.

   State the code of your test in the report and explain how it works.

5.3 Task 3: Application of the memory manager

Now you are going to verify that your memory management system may replace the standard `malloc/free` used in Assignment 1.

First, you need to copy your `main.c` file from assignment 1 into your assignment 2 directory. Then, you need to replace all calls to `malloc` and `free` with calls to `simple_malloc` and `simple_free` and add an include of `"mm.h"`.

Finally, you should check that the command interpreter still works as expected including running your extended shell test script `test.sh`.

Reporting for Task 3

In the report section for this task, you should address the following points:

1. State the result of the combined system. Does it still work?

2. Discuss what will happen if the user program forgets to free allocated collection nodes. What is the phenomenon called?
3. Another common problem with explicit memory management is the notion of *dangling pointers*. Explain this notion briefly.
4. Suggest a measure which could be implemented in the memory management system in order to help detecting occurrences of dangling pointers.

6 Hand-in and reporting

On Assignment 2 on DTU Learn, you must hand in the following:

- A file named `group_NN_assign_2.tgz` containing your solution files `mm.c`, `check_mm.c` , and `main.c`. If you have changed or added other files, these should be included also. As suggested by the suffix, the format of the file must be a gzipped tar-archive.
- A file named `group_NN_assign_2.pdf` with a report for the assignment. See below.

In the above file names, *NN* is your (two-digit) group number, e.g. `group_05_assign_2.tgz` .

6.1 Report requirements and rules

The report may be written in Danish or English. It must be a pdf file in A4 format but otherwise the layout is free.

The report should have a front page with the following information:

- **Who:** Group no. + name and study no. of each participant
- **What:** Assignment name and no.
- **Where:** University, department, course number and course name
- **When:** Date of hand-in

In the report, each task should be addressed in order and the reporting points properly answered.

There is no page limit, but the report should be kept concise and to the point. You may use code snippets to illustrate your text.

References to used material and solutions may be given as footnotes or by a reference list. Any use of generative AI must be declared: What did you use it for and how?

By handing in a report with all your names, all members of the group implicitly confirm that all work has been done by the group itself and that each member has made a significant contribution to the work.

Also note:

- Any collaboration with other groups on smaller parts of the assignments must be declared and clearly identified. Collaboration on major parts is not acceptable.
- Any undeclared use or adaption of others' work is *strictly prohibited*