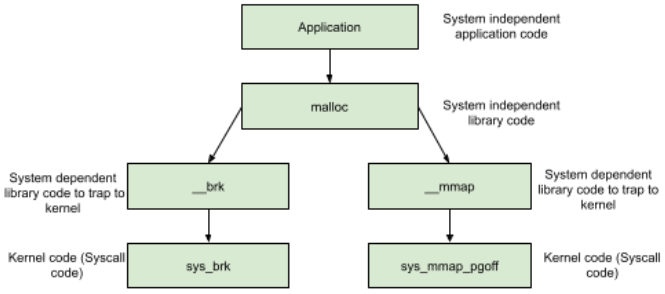
# Programming Assignment 3: Arena Allocator
# Due: Monday November 11th, 2022 5:30PM CST

## Memory Allocation with malloc()

In the glibc `malloc()` implementation `malloc()` either invokes either brk() or mmap () syscall to obtain memory.




While this implementation works fine for most applications applications with performance needs, e.g. games, aircraft, spacecraft, etc. can not afford to make a system call every time an allocation is needed. In these cases a large memory pool or arena is allocated on application startup. The application then manages memory on its own by implementing its own allocators to handle requests.

1. You will implement a memory arena allocator that supports First Fit, Best Fit, Worst Fit and Next Fit algorithms.

2. You must also benchmark your four implementations of an allocator against the standard system call malloc(). Design and develop a program and capture execution time for your four implementations of the arena allocator and compare their performance against malloc().

3. The results of your benchmarking will be written in a report detailing your findings.

You may complete this assignment in groups of two or by yourself. If you wish to be in a group of two the group leader must email me your group member's names by November 4th, 2022. Your email must have the subject line "3320 Section # Assignment 3 Group" where section number is 002 or 003. ( 003 is the 5:30pm class, 002 is 7:00pm ). For example, if you are in the 5:30pm class your subject line must be "3320 Section 003 Assignment 3 Group"

The code you submit for this assignment will be verified against a database consisting of kernel source, github code, stackoverflow, previous student's submissions and other internet resources. Code that is not 100% your own code will result in a grade of 0 and referral to the Office of Student Conduct.

## Important Warning

This program involves a lot of pointers and pointer arithmetic. You will seg fault your code if you are not careful with your pointers. Verify **ALL** pointers pointers before you dereference them.

Bad: if ( ptr -> next )

Good: if( ptr && ptr->next )

You will need to use gdb to find and fix your pointers errors.

## Your Memory Allocator API

Your memory allocator must implement the following four functions:

```
int mavalloc_init( size_t size, enum ALGORITHM algorithm )
```

This function will use malloc to allocate a pool of memory that is `size` bytes big. If the size parameter is less than zero then return -1. If allocation fails return -1. If the allocation succeeds return 0. This is the only `malloc()` your code will call.

`size` must be 4-byte aligned. You are provided a macro ALIGN4 to perform this alignment.

The second parameter will set which algorithm you will use to allocate memory from your pool. The enumerated value is:

```
enum ALGORITHM
{
  FIRST_FIT = 0,
```

```
    NEXT_FIT,
    BEST_FIT,
    WORST_FIT
  };
```

```
void *mavalloc_alloc( size_t size )
```

This function will allocate `size` bytes from your preallocated memory arena using the heap allocation algorithm that was specified during `mavalloc_init`. This function returns a pointer to the memory on success and NULL on failure.

```
void mavalloc_free( void * pointer )
```

This function will free the block pointed to by the pointer. This function returns no value. If there are two consecutive blocks free then combine (coalesce) them. Do not call free( ) for this. This should return the block back to your area.

```
void mavalloc_destroy( )
```

This function will free the allocated arena and empty the linked list <u>This is the only `free()` that your code will call.</u>

```
int mavalloc_size( )
```

This function will return the number of nodes in the memory area linked list.

## Tracking Your Memory

You will keep a ledger to maintain a list of allocated and free memory segments of your memory pool, where a segment is either allocated to a process or is an empty hole between two allocations. Your ledger should be implemented as a linked list using an array and not use dynamic memory allocations.

```
struct Node {
  size_t size;
  enum TYPE type;
  void * arena;
  int next;
  int previous;
```

```
};
```

Preallocate the memory for your ledger in your `mavalloc_init` function. Assume a maximum of 10000 allocations.

Each entry in the list specifies a hole (H) or process allocation (P), the address at which it starts, the length, and a pointer to the next item.

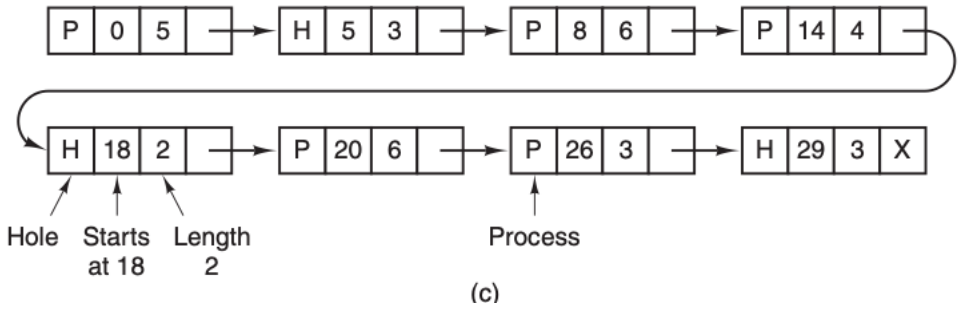



Figure 1

For example, after the user initializes your memory allocator with:

```
mavalloc_init( 65535, FIRST_FIT );
```

Then you will have a single node in your linked list specifying a hole, at the starting address, a length of 65535, and a next pointer of NULL.

If the user then requests a memory allocation as:

```
mavalloc_alloc( 5000 );
```

you will then have a linked list with the first node specifying a process allocation (P), a pointer to the starting address, a length of 5000, and a next pointer pointing to the next node which is a hole (H), pointing to the starting address plus 5000, a length of 60535, and a next pointer of NULL.

Your list will be kept sorted by address. Sorting this way has the advantage that when a allocation is released (terminated in figure 1 terminology), updating the list is straightforward. An allocated block of memory normally has two neighbors (except when it is at the very top or bottom of memory). These may be either allocations or holes, leading to the four combinations shown in Fig. 2. In Fig. 1(a) updating the list requires

replacing a P by an H. In Fig. 2(b) and Fig. 2(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 2(d), three entries are merged and two items are removed from the list.

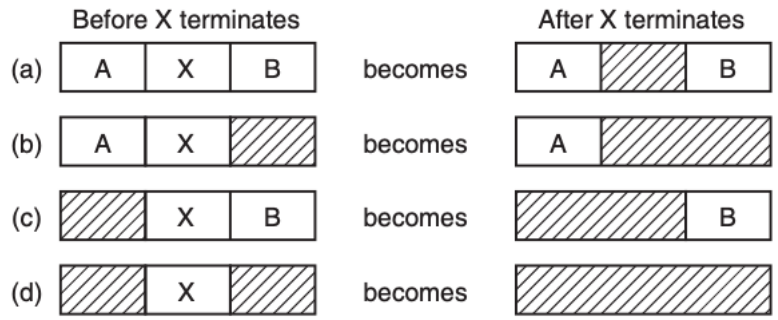



Figure 2

It may be more convenient to have the list as a double-linked list, rather than the single-linked list. This structure makes it easier to find the previous entry and to see if a merge is possible. You may use either a single or doubly linked list.

## Aligning Memory Requests

All memory allocations must be word-aligned. You must use the following `#define` in your code:

```
#define ALIGN4(s)          ((((s) - 1) >> 2) << 2) + 4)
```

When the user requests a size of memory you must pass it through that macro to ensure that the request is 4-byte aligned.

For example, your `void *mavalloc_alloc( size_t size )` function must treat the parameter size as:

```
void *mavalloc_alloc( size_t size )
{
  size_t requested_size = ALIGN4( size );

  // allocate requested_size according to the selected algorithm
```

Your allocator will then allocate a memory block of `requested_size` number of bytes.

## Getting and Building the Code

The code for this assignment is at https://github.com/CSE3320/Arena-Allocator-Assignment. A Makefile is provided so to build the code type: `make`

## Non-Functional Requirements

Tabs or spaces shall be used to indent the code. Your code must use one or the other. All indentation must be consistent.

No line of code shall exceed 100 characters.

All code must be well commented. This means descriptive comments that tell the intent of the code, not just what the code is executing. When in doubt over comment your code.

Keep your curly brace placement consistent. If you place curly braces on a new line , always place curly braces on a new end. Don't mix end line brace placement with new line brace placement.

Remove all extraneous debug output before submission. The only output shall be the output of the unit tests and your benchmark programs.

## Unit Tests

There are 20 unit tests that will help you debug your code. They are contained in `main.c` in the repo. After building the code with `make` you can run the tests by typing `./unit_tests .` There will be 20 test cases with each test case being worth 2.5% of the total grade. When a unit test fails the assert message will tell you what assert didn't pass. You can then look at the unit test and a comment before the assert will tell you what part of your functionality broke the code. In the screenshot below, if the assert on line 18 killed your program, you can read the comment on line 17 that tells you

```
13    char * ptr = ( char * ) mavalloc_alloc ( 65535 );
14
15    int size = mavalloc_size();
16
17    // If you failed here your allocation on line 13 failed
18    TINYTEST_ASSERT( ptr );
```

the mavalloc_alloc call on line 13 failed. You would then need to fix mavalloc_alloc to pass the test.

You may not modify main.c. 50% of your grade for this assignment will come from the unit tests.

## Benchmarking your code

You may use benchmark1.c, benchmark2.c, benchmark3.c, benchmark4.c, and benchmark5.c to create your benchmarks. These programs must be identical with the exception of the calls to malloc() in one program and the mavalloc_init() call in the other four to ensure your comparisons are valid.

You are to determine which implementation is best given your benchmark programs. You must design your code and testing to provide you with enough valid data to allow you to interpret and report your findings. These programs must be thorough in their behavior and must test a large range of different allocations and edge cases. Benchmarks and reports which only test a few use cases will result in a loss of significant points on the report.

You must demonstrate enough rigor in your data collection that variations in CPU loading on your system provides a negligible amount of error in your benchmarks.

Timings must be collected and presented in units of milliseconds or smaller. Results that are collected in units of seconds will result in 0 points being assigned for the benchmark / report portion of the assignment.

## The Report

Your report must start with an executive summary paragraph that succinctly explains your project, the benchmarks, the results of your benchmarks and your conclusion as to which is the best allocation algorithm implementation.

Following the executive summary, explain the purpose of the benchmarks and your benchmark design in detail. Be sure to clearly state the design considerations and strategy used to throughly benchmark the performance of your implementations.

Your data must be presented in a table and graph form.

You must provide an interpretation, in text, of your data with enough description and explanations that a reader will understand the results and understand your conclusions.

If there are any anomalous results be sure to note them and provide an explanation as to why they occurred.

After presenting your results follow with your conclusions. Explain, with references back to your data, why your winning algorithm is the best allocation algorithm. References back to the lecture noting that Worst Fit leaves the largest available heap blocks will not be accepted and will result in 0 points for the report. You must have data supporting your conclusion and your interpretation must rely on your data.

**Report Format**

Your report must be submitted as a PDF file. Reports that are submitted as Microsoft Word documents, text files, etc. will result in loss of half the report points. The report must be written with a 12pt Times New Roman typeface with headings in a 16pt Bold Times New Roman typeface.

Reports that make me install Open Office or any other application to read your report will not be read or graded and will be assumed to contain no information. I can't support 144 different file formats and 280+ different typefaces for the reports. PDF in an industry standard so we will use it. And while Times New Roman is a little dull, it's easy to read and is a common industry standard as well.

In the upper right corner of the first page put the names of the group members and their student ID numbers.

## What Do I Submit?

The deliverables for this assignment are:

1. mavalloc.c
2. benchmark1.c
3. benchmark2.c
4. benchmark3.c
5. benchmark4.c

6. benchmark5.c

7. Your report in PDF format.

Do not zip, gzip, tar, bz2, the files. Submit them each individually.

## Point Allocations

| Category | | Percentage of Grade |
|---|---|---|
| **Unit Tests** | 2.5% per unit test | 50% |
| **Report and Code** | | 50% |

## Administrative

This assignment must be coded in C. Any other language will result in 0 points. You programs will be compiled and graded on the course container. Please make sure they compile and run in the container before submitting it. Code that does not compile with the provided Makefile will result in a 0.

Your program is to be turned in via Canvas. Submission time is determined by the Canvas system time. You may submit your programs as often as you wish. Only your last submission will be graded.

There are coding resources, working code, and test code on GitHub at: https://github.com/CSE3320/Arena-Allocator-Assignment . You may use no other outside code.

## Academic Integrity

This assignment must be 100% your own work. No code may be copied from friends, previous students, books, web pages, etc. All code submitted is automatically checked against a database of previous semester's graded assignments, current student's code and common web sources. By submitting your code on Canvas you are attesting that you have neither given nor received unauthorized assistance on this work. **Code that is copied from an external source or used as inspiration, excluding the course github or Canvas, will result in a 0 for the assignment and referral to the Office of Student Conduct.**