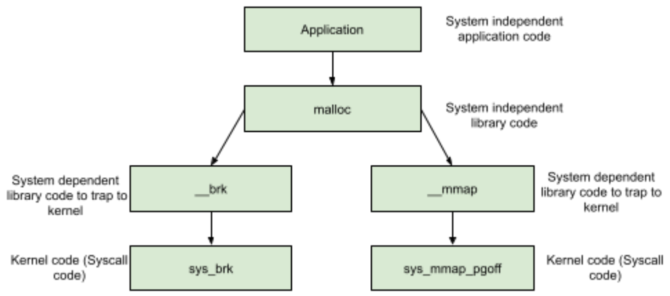
Programming Assignment 3: Arena Allocator
Due: Monday April 11th, 2022 5:30PM CST

Memory Allocation with malloc()

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


Application
System independent application code
malloc
System independent library code
System dependent library code to trap to kernel
__brk
__mmap
System dependent library code to trap to kernel
Kernel code (Syscall code)
sys_brk
sys_mmap_pgoff
Kernel code (Syscall code)

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.

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

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 March 31, 2022. Your email must have the subject line "3320 Section # Assignment 3 Group" where section number is 001 or 003. (003 is the 5:30pm class, 001 is 7:00pm)

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.

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 back to your preallocated memory arena. This function returns no value. If there are two consecutive blocks free then combine (coalesce) them.

```
void mavalloc_destroy( )
```

This function will free the allocated arena and empty the linked list `int mavalloc_size( )`

This function will return the number of nodes in the memory area

Tracking Your Memory

You will keep maintain a linked 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. 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.

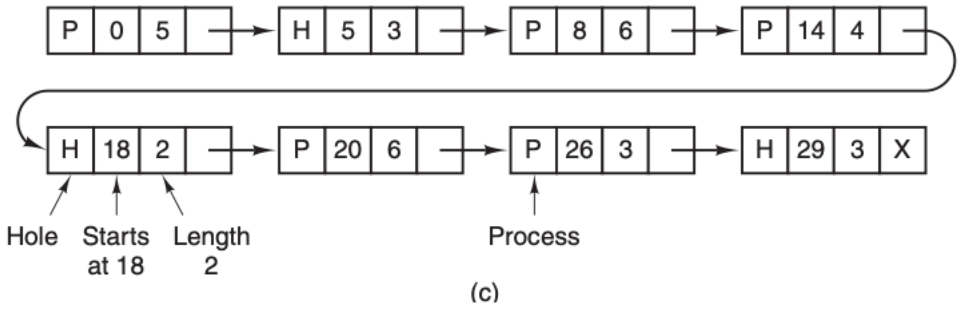


Figure 1

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.

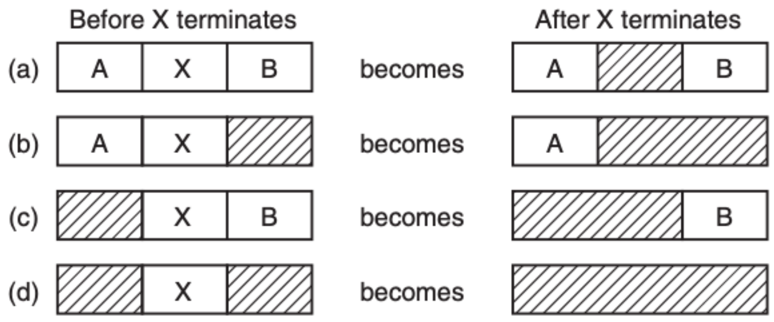


Figure 2

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.

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 commands entered or the shell prompt.

Testing your code

You will be provided a set of unit tests to test your code. They will be contained in `main.c` in the repo. You can run the tests by typing `./unit_tests .` There will be 20 test cases with each test case being worth 5% of the total grade.

Administrative

This assignment must be coded in C. Any other language will result in 0 points. You programs will be compiled and graded on omega.uta.edu. Please make sure they compile and run on omega 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.**