Weaver Memory Manager

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article describes using literary programming a memory manager written for Weaver Game Engine. It aims to be a very simple and fast memory manager for programs where memory is allocated and freed in a stack-based order and we know the maximum ammount of memory that the program will need. It allows users creating markings during the program execution that after allows them to free at once all the memory allocated after the last marking. After the memory manager creation, we also run some benchmarks comparing its performance against malloc from standard library running at Linux, Windows and in a web browser using Web Assembly.*

1. Introduction

1.1. Memory Managers in Game Engines

Many game engines run custom memory managers instead of using the system's library to obtain more memory dinamically. According with [Gregory 2019], this happens because implementations like *malloc* and *free* could be relatively slow when used in games compared with custom memory managers. Custom memory managers also can deal with memory fragmentation.

Gregory identifies five design pattern for managing memory in games:

Stack-based Allocators: These allocators always return sequential regions of memory and deallocations must happen in reverse order than allocation. The implementation is very simple and it is easy to ensure spatial locality of used memory regions. All the memory allocatyed since some specific time of execution could be rapidly deallocated with a single function call.

Pool Allocators: This technique can be used when we know that we will need at most n elements, all with the same size. We can allocate previously the memory for these elements and keep their memory region always at disposal using variables to store when a given region is occupied by an element and when we can treat it as free memory.

Alligned Allocations: Different types of data and computer architectures can have different alignment restrictions for memory. In *Playstation 3*, for example, any memory position passed using DMA (*Direct Memory Access*) must be 128-bit aligned. This means that its address must be a multiple of 128 bits.

Single Frame and Double-Buffered Memory Allocators: A game engine runs all its computations in frames, and in each frame a new image is sent to the screen. Some allocated variables must have a lifetime of only one or two frames, and could be deallocated automatically after this time.

Memory Defragmenter: When memory is allocated and freed in unpredictable order, small unused memory regions could accumulate. Sometimes they are too small to be useful, and they are many and so they waste considerable ammounts of memory. To avoid this, instead ofusing pointers for allocated objects, we could use indexes which indentifies the pointers. This way, allocated memory could be moved incrementally and peridically to avoid

fragmentation and their pointers can be updated.

This article objective is to define a stack-based allocator with support for memory alignment and which can store two stacks in its memory region. Manage which stack to use is user's responsability. This technique is described in [Ranck, 2000] which shows how it was used in the game *Hydro Thunder* made by *Midway*.

Pool allocators won't be supported here, but they can easily be built using the allocator defined here. The fragmentation problem also won't be addressed, as stack-based allocators usually do not have memory fragmentation when correctly used.

1.2. Executon Environments

In this article we will focus in creating functions which to run in four different environments: Windows 10, MacOS Sierra, Linux and Web Assembly. The three first are operating systems for personal computers. Thelast one is a virtual machine specification specialized in run a optimized subset of javascript. It permits the execution of complex computer programs in environments like web browsers. Its development started as a method proposed by [Zakai, 2011] who presented a novel way to compile code in C and C++ for javascript with advanced optimization techniques.

To develop portable code in all four environments, we will use conditional macros and different methods of obtaining memory in all different systems will be compared.

1.3. Literary Programming and this Article Notation

This article utilizes the technique of "Literary Programming" in the development of the memory manager. This technique was presented in [Knuth, 1984] and consists in a philosophy of software development where the programmer develop the software writting and explaining didatically all the necessary code, focusing in writting a clear explanation for people reading the explanation. Automatic tools are employed to extract the code from the explanation, change the order of the code when necessary and produce an executable program from the extracted code.

For example, in this article will be defined two different files: `memory.c` and `memory.h`, both could be statically included in any project, or compiled as a shared library. What the file `memory.h` contains is:

File: `src/memory.h`**:**

```
#ifndef WEAVER_MEMORY_MANAGER
#define WEAVER_MEMORY_MANAGER
#ifdef __cplusplus
extern "C" {
#endif
                    <Section to be inserted: Memory Declarations>
#ifdef __cplusplus
}
#endif
#endif
```

The two first lines and the last one are security macros to avoid that functions and variables declared there be included more than once, even if an user includes this header file more than once using include macros. The other lines contains macros to check if we are compiling using C++ instead of C. In this case we declare all functions in this file as C style functions to warn the compiler that they are not modifiable by operator overloading and because of this is not necessary to store additional information besides the function name to

recognize them.

In the code above, we use the red letters to indicate that in the future we will insert new code there called "Memory Declarations", with all the necessary function declaraions. Searching in this article, you can find in the next pages another pieces of code where the title is not `memory.h` as above, but "Memory Declarations". These pieces of code will be inserted in the code above. And could be more than one piece of code with he same title. In this case, to produce functional code for the compiler, we should concatenate all these pieces of code with the same title and put in the part marked as red in the source code. his allow us to introduce function declarations as we explain them in the article, not needing to declare all them once just because they are part of the same piece of code.

1.4. Functions to be Defined

Our memory manager will define a total of 6 new functions. The first one receives a size in bytes and return a new contiguous region of memory to be managed. We call this region a "arena":

Section: Memory Declarations:

```
#include <stdlib.h> // Include 'size_t'
void *_Wcreate_arena(size_t size);
```

The second function receives an arena created by the first function and free all the reserved memory. If that arena had not deallocated elements, we return false (a program could print a warning to the user if the function return false) and otherwise the function returns true:

Section: Memory Declarations (continuation):

```
#include <stdbool.h> // Include 'bool'
bool _Wdestroy_arena(void *);
```

The third one is equivalent to `malloc` and receives an arena, a number which should be a power of two or zero representing a bit alignment which should be respected, another number which selects if we should allocate in ther left (0) or right (1) stack, and finally a size in bytes. The function always returns an address multiple of the alignment if a power of two was passed. And it returns `NULL` in case of no enough memory in the arena:

Section: Memory Declarations (continuation):

```
void *_Walloc(void *arena, unsigned alignment, int right, size_t size);
```

The fourth function puts a marking, which we will call "memory point" in our arena. This marking can be used to determine which allocations happened before and after the marking. The last argument also selects if we should put the marking in the left (0) or right (1) stack. We also specify an aligment like in the previous allocation function. The function returns if this was sucessful or not:

Section: Memory Declarations (continuation):

```
bool _Wmempoint(void *arena, unsigned alignment, int region);
```

The last function use the marking of the previous function and deallocate all the allocated memory since the last marking was created. It receives a flag to know if this should be done with the left (0) or right (1) stack.

Section: Memory Declarations (continuation):

```
void _Wtrash(void *arena, int region);
```

2. Implementation

2.1. Obtaining a Beginning Region of Memory

In a general purpose memory manager we don't need to know the maximum ammount of memory needed in our program. But in a game memory manager, it is essential to estabilish a maximum limit for memory usage. In this case we are more interested in ensuring a continuous performance instead of getting a precise result for some computation. We want to avoid at all costs the subit performance decrease when a supported machine don't have more memory and needs to use memory swap. In some architectures, we do not even have a disk to use swap memory and consuming all RAM can make the game crash. Because of this, in games we can allocate during initialization the maximum ammount of needed memory and never use more memory than what was allocated in this moment.

How we obtain this initial memory varies according with the execution environment. We could use a simple `malloc` for this, but we will prefer alternatives with less waste of memory. Ideally a function which returns a raw region of memory, without aditional informations or data structures.

Both in Unix systems (Linux, OpenBSD, MacOS) as in Web Assembly compiled usng Emscripten, the more economical choice is using `mmap`. It is a very customizable function which can map to memory anithing, from disk files to new allocated regions.

To use `mmap`, we need the following headers:

Section: Incluir Cabealhos Necessrios:

```
#if defined(__EMSCRIPTEN__) || defined(__unix__) || defined(__APPLE__)
#include <sys/mman.h>
#endif
```

After defining the headers, to allocate a region of M bytes not associated with any file, which can be read and written, we invoke the function with the following arguments:

Section: Allocate in 'arena' region of 'M' bytes:

```
#if defined(__EMSCRIPTEN__) || defined(__unix__) || defined(__APPLE__)
arena = mmap(NULL, M, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON,
             -1, 0);
#endif
```

The null, zero and -1 arguments are just the ones not necessary in this usage of `mmap`.

As we can create a new region of memory, we can also destroy that region, deallocating it. In this case, we should use `munmap`:

Section: Deallocate 'arena' of size 'M' bytes:

```
#if defined(__EMSCRIPTEN__) || defined(__unix__) || defined(__APPLE__)
munmap(arena, M);
#endif
```

In Windows environments, the equivalent function is `CreateFileMapping`, with the difference that it returns a handle which needs another function to give us a pointer to the allocated region. Fortunately, according with Windows API documentation, we can close that handle with `CloseHandle` before deallocate and undo the reserver region. Thanks for this, we can keep the simmetry between Windows and other platform's code, as we can close the handle in this code without needing to memorize it in some additional structure:

Section: Allocate in 'arena' region of 'M' bytes (continuation):

```
#if defined(_WIN32)
{
  HANDLE handle;
  handle = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,
                              PAGE_READWRITE,
                              (DWORD) ((DWORDLONG) M) / ((DWORDLONG) 4294967296),
                              (DWORD) ((DWORDLONG) M) % ((DWORDLONG) 4294967296),
```

```
                                NULL);
  arena = MapViewOfFile(handle, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);
  CloseHandle(handle);
}
#endif
```

To deallocate the region in Windows, we use `UnmapViewOfFile`:

Section: Deallocate 'arena' of size 'M' bytes (continuation):

```
#if defined(_WIN32)
UnmapViewOfFile(arena);
#endif
```

Using the previous functions require the following headers:

Section: Include Headers (continuation):

```
#if defined(_WIN32)
#include <windows.h>  // Include 'CreateFileMapping', 'MapViewOfFIle',
#include <memoryapi.h> // 'UnmapViewOfFile', 'CloseHandle'
#endif
```

2.2. Obtaining Page Size

In a real computer, contrasting with some virtual machines, when a program asks for memory for the Operating System, it always receive memory in multiples of the page size used internally by the machine. Tipically the page size is 4 KiB. In this case, it's useless asking for non-multiples of KiB. If we ask for just 2 KiB, we keep receiving 4 KiB. If we ask for 5 KiB, we receive 8KiB.

It's important that in these environments, our memory manager be aware of this and even if the user asks for a non-multiple of page size ammount of memory, it always should adjust the requested size for a multiple of page size, avoiding the waste of memory.

In the majority of Unix systems, if they are POSIX compatible, we can get the page size using the function `sysconf`:

Section: Get page size 'p':

```
#if defined(__unix__)
p = sysconf(_SC_PAGESIZE);
#endif
```

In MacOS, while its documentation says that the system is POSIX compatible, it also recommends using the BSD function `getpagesize` to get this information. So to follow the documentation guidelines, we use this code:

Section: Get page size 'p' (continuation):

```
#if defined(__APPLE__)
p = getpagesize();
#endif
```

Both the BSD function as the POSIX function are defined in the same header:

Section: Include Headers (continuation):

```
#if defined(__APPLE__) || defined(__unix__)
#include <unistd.h>
#endif
```

In Windows, we can get the page size using the more complex `GetSystemInfo`, which also returns a series of additional information about the system, which we don't need to keep at the moment.

Section: Get page size ‘p’ (continuation):

```
#if defined(_WIN32)
{
  SYSTEM_INFO info;
  GetSystemInfo(&info);
  p = info.dwPageSize;
}
#endif
```

To use this function, the documentation recommends to include directly `windows.h` header:

Section: Include Headers (continuation):

```
#if defined(_WIN32)
#include <windows.h> // Include 'GetSystemInfo'
#endif
```

Finally the Web Assembly environment. Here the dynamically allocated memory is obtained from a contiguous region of memory which starts with a fixed size and can grow calling the operator `grow_memory`, which also is configured with a maximum size. Anyway, here the memory is also allocaten in multiples of a fixed value. But the value is always 64 KiB:

Section: Get page size ‘p’ (continuation):

```
#if defined(__EMSCRIPTEN__)
p = 64 * 1024; // 64 KiB
#endif
```

2.3. Running in Multiple Threads

It’s important to ensure that he code defined here works even when invoked simultaneously in different parts of code. For this we will need mutexes to ensure that the same region of memory won’t be accessed by more than one thread at same time. If more than one thread wants toallocate memory, probably will be more efficient give to each one its own arena to avoid that each thread block the others in the process.

Not all environments supports threads. n the virtual machne Web Assembl, at least in the version implemented in web browsers at the time of this writting, don’t support threads except in experimental versions. In Windows we also would prefer to use “critical sections” instead of a mutex. The main reason is avoiding a system call to the kernel when a mutex is not blocked. The critical sections ensure this with the rrelevant drawback of not being able to be shared between different programs. Besides that, the critical sections are equivalent to mutexes.

In Unix based systems, we include the POSIX header “pthreads”. In Windows, the relevant heraders are already included in `windows.h`.

Section: Include Headers (continuation):

```
#if defined(__unix__) || defined(__APPLE__)
#include <pthread.h>
#endif
```

A mutex is declared with the following code:

Section: Declarao de Mutex:

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_t mutex;
#endif
```

```
#if defined(_WIN32)
CRITICAL_SECTION mutex;
#endif
```

When the mutex is initialized, we store in a boolean variable `error` if some problem happened. In the case of the library *pthreads*, its initialization function already returns a non-null value if some problem happened. In Windows, the operating system ensures that the function never fails. In both cases we assume that we have a generic pointer for our mutex:

Section: Initialize '*mutex':

```
#if defined(__unix__) || defined(__APPLE__)
error = pthread_mutex_init((pthread_mutex_t *) mutex, NULL);
#endif
#if defined(_WIN32)
InitializeCriticalSection((CRITICAL_SECTION *) mutex);
#endif
```

The code to destroy the mutex is below. In this case we don't need to care about errors:

Section: Ending '*mutex':

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_destroy((pthread_mutex_t *) mutex);
#endif
#if defined(_WIN32)
DeleteCriticalSection((CRITICAL_SECTION *) mutex);
#endif
```

The classical operation of *wait* when a thread try to enter a critical section protected by mutexes:

Section: '*mutex':WAIT():

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_lock((pthread_mutex_t *) mutex);
#endif
#if defined(_WIN32)
EnterCriticalSection((CRITICAL_SECTION *) mutex);
#endif
```

And this is the function *signal* to free a mutex resedved by the thread:

Section: '*mutex':SIGNAL():

```
#if defined(__unix__) || defined(__APPLE__)
pthread_mutex_unlock((pthread_mutex_t *) mutex);
#endif
#if defined(_WIN32)
LeaveCriticalSection((CRITICAL_SECTION *) mutex);
#endif
```

2.3. Memory Arena Header

After allocating a region of memory (or "arena"), the first requirement to initialize it is reserving the initial byes to store general informations abou the region. The necessary informations are: remaining size in bytes (`remaining_space`), arena total size (`total_size`) and pointers to the beginning of the next free region in the left and right stack (`left_free`, `right_free`). In `right_allocations` we store the quantity in bytes allocated in right stack and in `left_allocations` the quantity in bytes of memory allocated in the left stack.

We also need a mutex to protect the arena header of being manipulated by two threads at same time.

Another information useful during debugging is the maximum ammount of memory that our arena allocated during its lifetime. This is useful because after deallocating the arena we can check if we gave too much memory to it that in the end wasn't used. In this case, we could want to lower the ammount os memory reserved to it. We will obtain this information wih the variable `smallest_remaining_space` that store the minimum value of remaining space during the arena lifetime. This variable will be defined only if the macro `W_DEBUG_MEMORY` is defined. We prefer to store the minimum ammount of remaining space instead of the maximum ammount of used space because is usually easier to read values which should be small, close to zero, instead of values which usually should be big, close to the total arena size.

We will also have two other pointers: `left_point` and `right_point`. They are pointers for "memory points", information about the arena status during a specific moment which permit us to restore the arena to that given status. These memory points will be better defined at section 2.9.

The header in our memory arena have the following format:

Section: Arena Header:

```
struct arena_header{
                    <Section to be inserted: Declarao de Mutex>
  void *left_free, *right_free;
  void *left_point, *right_point;
  size_t remaining_space, total_size, right_allocations, left_allocations;
#if defined(W_DEBUG_MEMORY)
  size_t smallest_remaining_space;
#endif
};
```

Let's assume that we have a pointer for a non-initialized arena, which we call `arena` and that we know as `M` the arena size in bytes. With these information we can initialize the arena header with the fllowing code. To compute the next empty positions where we could store data, we convert the arena pointer for a characere pointer. Doing his we ensure that the pointer arithmetic will work with multiples of 1 byte. The next free space in the left stack is the first byte after the header. And the next free space in the right stack is the last byte in the arena.

Section: Initialize header in 'arena' with size 'M':

```
{
  struct arena_header *header = (struct arena_header *) arena;
  header -> right_free = ((char *) header) + M - 1;
  header -> left_free = ((char *) header) + sizeof(struct arena_header);
  header -> remaining_space = M - sizeof(struct arena_header);
  header -> right_allocations = 0;
  header -> left_allocations = 0;
  header -> total_size = M;
  header -> left_point = NULL;
  header -> right_point = NULL;
#if defined(W_DEBUG_MEMORY)
  header ->  smallest_remaining_space = header -> remaining_space;
#endif
  { // Mutex initialization
    void *mutex = &(header -> mutex);
                    <Section to be inserted: Initialize '*mutex'>
```

```
  }
}
```

2.4. Allocating New Arena

With the previously defined code we already can define the function `_Wcreate_arena`. This function is responsible for the following 6 operations:

1. Get from the user a size `t` in bytes to allocate.
2. Discover the page size `p` in our environment.
3. Get `M`, as the lesser multiple of `p` greater than `t`. If `M` is smaller than the arena header size, it becomes equal the lesser multiple than `p` greater than header size.
4. Allocate a new arena of size `M`.
5. Initialize its header.
6. Return a pointer for the beginning of arena, where its header is stored, or `NULL` in case of problems.

The code to do this is:

Section: Definition for '_Wcreate_arena':

```
void *_Wcreate_arena(size_t t){
  bool error = false;
  void *arena;
  size_t p, M, header_size = sizeof(struct arena_header);
  // Operation 2:
                    <Section to be inserted: Get page size 'p'>
  // Operation 3:
  M = (((t - 1) / p) + 1) * p;
  if(M < header_size)
    M = (((header_size - 1) / p) + 1) * p;
  // Operation 4:
            <Section to be inserted: Allocate in 'arena' region of 'M' bytes>
  // Operation 5:
           <Section to be inserted: Initialize header in 'arena' with size 'M'>
  // Operation 6:
  if(error) return NULL;
  return arena;
}
```

2.5. The function "_Wdestroy_arena"

Once we provided a function to create new arenas, we need to define a function to destroy and deallocate them. This is a simpler function to do four things:

1. Destroy the arena mutex.
2. Print a warning if there are still allocated memory in the arena.
3. If we are in debug mode, print the ammount of memory which never was used int he arena.
4. Free to the operating system the arena memory.

Section: Definition for '_Wdestroy_arena':

```
bool _Wdestroy_arena(void *arena){
  struct arena_header *header = (struct arena_header *) arena;
  void *mutex = (void *) &(header -> mutex);
  size_t M = header -> total_size;
  bool ret = true;
                    <Section to be inserted: Ending '*mutex'>
  if(header -> total_size != header -> remaining_space +
```

```
     sizeof(struct arena_header))
    ret = false;
#if defined(W_DEBUG_MEMORY)
  printf("Unused memory: %zu/%zu (%f%%)\n",
         header -> smallest_remaining_space, header -> total_size,
         100.0 *
         ((float) header -> smallest_remaining_space) / header -> total_size);
#endif
```

<Section to be inserted: Deallocate 'arena' of size 'M' bytes>

```
  return ret;
}
```

If we are in debug mode, we need to include the correct header to print messages in the screen:

Section: Include Headers (continuation):

```
#if defined(W_DEBUG_MEMORY)
#include <stdio.h>
#endif
```

2.6. Keeping Memory Alignment in Allocations

When we allocate new memory, we begin with an available address p in which we will put our allocated memory. But we should respect the alignment a: a power of two or the number zero (meaning "any alignment"). How to do this depends if we are in the left stack (initial positions in the arena) or right stack (ending positions in the arena). One of the stacks grows from lesser to greater addresses and the other grows form greater to lesser addresses.

In the left stack, we want to put our allocation in the address p, but we may need to shift p by some positions to respect the alignment. We consider that $a-1$ is the worst case of how many bytes we need to shift. As a is a power of two, $a-1$ is also a bitmask of what bits should be zero in the final address. Using these informations, we can ensure memory alignment with:

Section: Align 'p' and store 'offset' according with 'a' (left):

```
offset = 0;
if (a > 1){
  void *new_p = ((char *) p) + (a - 1);
  new_p = (void *) (((uintptr_t) new_p) & (~((uintptr_t) a - 1)));
  offset = ((char *) new_p) - ((char *) p);
  p = new_p;
}
```

We need the casting to `uintptr_t` because we want to do bit-to-bit operations in pointers. So we need the header below:

Section: Include Headers (continuation):

```
#include <stdint.h>
```

In the right stack, we will try to allocate in address p, but we probably need to lower the initial address to keep the alignment. So we don't need to sum the initial address with the worst case value $a-1$. We can just remove directly the ending bits of using bit mask $a-1$:

Section: Align 'p' and store 'offset' according with 'a' (right):

```
offset = 0;
if (a > 1){
  void *new_p = (void *) (((uintptr_t) p) & (~((uintptr_t) a - 1)));
```

```
  offset = ((char *) p) - ((char *) new_p);
  p = new_p;
}
```

2.7. Allocating Memmory

Before memory allocation, we need to check if we have enough space in the arena. We just need to read values from arena header and compare them with the value which we need to allocate, considering the worst case in alignment, where we need additional $a-1$ bytes. If we don't have enough space, we need to return `NULL`.

f we are allocating in the left stack, our allocation address will be the next position after the header and previously allocated blocks after passing for alignment correction. We also need to update how many bytes are already allocated in variable `left_allocations`.

If we are allocating in the right stack, we check the next free region stored in the header and subtract from the address the size of the new allocation minus 1. This ensure that the allocated region have he right size and that the found address can be returned to the user. Of course, before returning we also need to do the alignment correction. And as we are updating the right stack, we store the new allocation size in the variable `right_allocations`.

Once this is done, we only need to update in the arena header what are the next free regions, as we occupied the previous one. We also need to compute a newvalue for variable `remaining_space`. Only after this we can return `p`.

Section: Allocating 'p' with size 't' in 'arena', alignment 'a':

```
{
  int offset;
  struct arena_header *head = (struct arena_header *) arena;
  if(head -> remaining_space >= t + ((a == 0)?(0):(a - 1))){
    if(right){
      p = ((char *) head -> right_free) - t + 1;
       <Section to be inserted: Align 'p' and store 'offset' according with 'a' (right)>
      head -> right_free = (char *) p - 1;
      head -> right_allocations += (t + offset);
    }
    else{
      p = head -> left_free;
        <Section to be inserted: Align 'p' and store 'offset' according with 'a' (left)>
      head -> left_free = (char *) p + t;
      head -> left_allocations += (t + offset);
    }
    head -> remaining_space -= (t + offset);
#if defined(W_DEBUG_MEMORY)
    if(head -> remaining_space < head -> smallest_remaining_space)
      head -> smallest_remaining_space = head -> remaining_space;
#endif
  }
}
```

2.8. The _Walloc Function

We can combine the previous code to define the allocation function. It will get `arena` (the arena where we will allocate), `a` (the alignment), `right` (1 if we should allocate in the right stack and 0 if we should allocate in the left stack) and `t` (size in bytes for the region to be allocated).

First we use a "wait" in the arena mutex. After we do the allocation and then use a

"signal" to free the mutex. We also need a variable `p` to be returned and tha should point to the newly allocated memory.

Section: Definition for '_Walloc':

```
void *_Walloc(void *arena, unsigned a, int right, size_t t){
  struct arena_header *header = (struct arena_header *) arena;
  void *mutex = (void *) &(header -> mutex);
  void *p = NULL;
                    <Section to be inserted: '*mutex':WAIT()>
      <Section to be inserted: Allocating 'p' with size 't' in 'arena', alignment 'a'>
                    <Section to be inserted: '*mutex':SIGNAL()>
  return p;
}
```

2.9. Definition for Memory Points

If we have an arena that is being utilized and is storing allocated regions, but we want to free at once all allocations and restart the arena, we could do this just restarting the values stored in arena header. The pointers for the next free regions and the variable with the remaining free space should also be reset to the initial values.

If we are interested not in restart all the arena, but just the left of right stack, we can just check how many bytes were allocated in the corresponding stack thanks for the variables `right_allocations` and `left_allocations` stored in arena header. These values can be easily reseted and this empties only the corresponding stack:

Section: Restart memory in stack from 'arena':

```
{
  struct arena_header *header = arena;
  if(right){
    header -> right_free = ((char *) arena) + header -> total_size - 1;
    header -> remaining_space += header -> right_allocations;
    header -> right_allocations = 0;
  }
  else{
    header -> left_free = ((char *) arena) + sizeof(struct arena_header);
    header -> remaining_space += header -> left_allocations;
    header -> left_allocations = 0;
  }
}
```

But what if instead of resetting all allocations we were interested in restore the state to a previous one saved in the past? We call the previous state a "memory point". To save the state, we need only to store the content of variables `left_allocations` for the left stack and `right_allocations` for the right stack.

The pointer for the next free region `left_free` or `right_free` can be updated moving it a number of positions equal to the difference between the current quantity of allocated memory and the quantity from a previous memory point.

Howeber, we want to store not a single one memory point, but a list of them. We can organize all stored memory points as a linked list. So we can define a memory point with the following header:

Section: Memory Point Header:

```
struct memory_point{
  size_t allocations; // Left or right
  struct memory_point *last_memory_point;
```

```
};
```

2.10. Creating a Memory Point

Creating memory points means calling *wait* for the mutex, allocate memory for the memory point, initialize it and update information in the arena about what is the last memory point, noting if we stored it in the left or right stack. Finally we free the mutex with *signal*:

Section: Definition for '_Wmempoint':

```
bool _Wmempoint(void *arena, unsigned a, int right){
  struct arena_header *header = (struct arena_header *) arena;
  void *mutex = (void *) &(header -> mutex);
  char *p = NULL;
  struct memory_point *point;
  size_t allocations, t = sizeof(struct memory_point);
                    <Section to be inserted: '*mutex':WAIT()>
  if(right)
    allocations = header -> right_allocations;
  else
    allocations = header -> left_allocations;
      <Section to be inserted: Allocating 'p' with size 't' in 'arena', alignment 'a'>
  point = (struct memory_point *) p;
  if(point != NULL){
    point -> allocations = allocations;
    if(right){
      point -> last_memory_point = header -> right_point;
      header -> right_point = point;
    }
    else{
      point -> last_memory_point = header -> left_point;
      header -> left_point = point;
    }
  }
                    <Section to be inserted: '*mutex':SIGNAL()>
  if(point == NULL)
    return false;
  return true;
}
```

2.10. Restoring a Memory Point

Restore a memory point means restoring the state of the memory stack (at left or right) exactly as it was before saving the last memory point. If here is no memory point saved, we empty all the memory stack. For this we use the function `_Wtrash`:

Section: Definition for '_Wtrash' (continuation):

```
void _Wtrash(void *arena, int right){
  struct arena_header *head = (struct arena_header *) arena;
  void *mutex = (void *) &(head -> mutex);
  struct memory_point *point;
                    <Section to be inserted: '*mutex':WAIT()>
  if(right){
    point = head -> right_point;
```

```
  }
  else{
    point = head -> left_point;
  }
  if(point == NULL){
            <Section to be inserted: Restart memory in stack from ‘arena’>
  }
  else{
    if(right){
      head -> remaining_space += (head -> right_allocations -
                                  point -> allocations);
      head -> right_point = point -> last_memory_point;
      head -> right_allocations = point -> allocations;
      head -> right_free = ((char *) point) + sizeof(struct memory_point) - 1;
    }
    else{
      head -> remaining_space += (head -> left_allocations -
                                  point -> allocations);
      head -> left_point = point -> last_memory_point;
      head -> left_allocations = point -> allocations;
      head -> left_free = point;
    }
  }
                    <Section to be inserted: ‘*mutex’:SIGNAL()>
}
```

2.11. Final Organization of Source File

We save all the code for function definition in the file below to be compiled:

File: `src/memory.c`:

```
                    <Section to be inserted: Include Headers>
#include "memory.h"
                    <Section to be inserted: Arena Header>
                  <Section to be inserted: Memory Point Header>
                <Section to be inserted: Definition for ‘“’Wcreate“’arena’>
               <Section to be inserted: Definition for ‘“’Wdestroy“’arena’>
                  <Section to be inserted: Definition for ‘“’Walloc’>
                <Section to be inserted: Definition for ‘“’Wmempoint’>
                  <Section to be inserted: Definition for ‘“’Wtrash’>
```

3. Performance

We measure the performance of this code with a prgram which runs the functions defined here one hundred thousand times and we measure the mean and standard deviation comparing our measures with measures for the functions `malloc` and `free` from the standard system library. Measures with higher standard deviations were discarded and not considered as we assumed that in these cases the Operating System probably was running jobs not related with our functions and these parallel jobs interferred with the measures.

In all these allocations, we choosed the value of 5 KiB expecting that with this size we avoid penalties for some `malloc` implementations with problems with memory alignment. At the same time this was expected to be higher enough to make `malloc` implementations ask for more memory to the Operating System, as this value is higher than the page system

in the tested system. In the function `_Walloc`, we opted for always use the left stack, as this should be the faster stack benefiting from spatial locality.

It's important to notice that while the functions defined here in this document were desgned to run in constant time, independent of allocation size, the `malloc` and `free` functions of operating systems not always follow the same philosophy. During our measures, the allocation size was increased to check if this affected execution times. When a function don't run in constant time, we measured for which values the function has an equivalent performance and for which the performance was worst. But we didn't risked more precise predictions about its behaviour, as this would require analysis of its implementation.

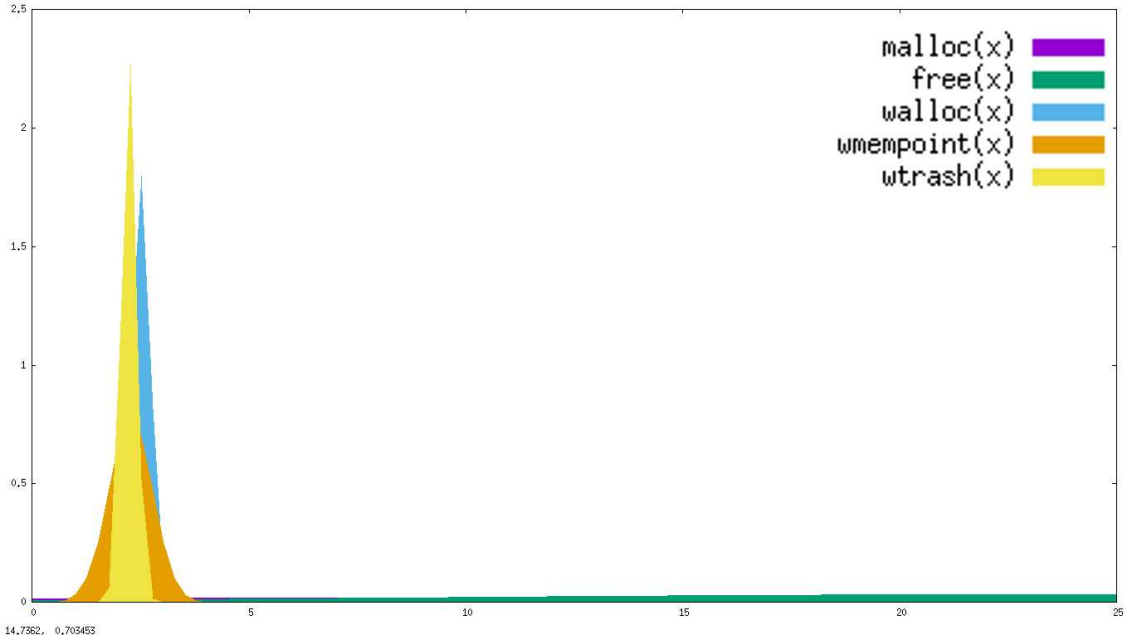
3.1. OpenBSD

In OpenBSD 6.4 the function `free` don't appear to run in constant time for different allocations size. The running time appears to grow linearlly when the allocaton size grows exponentially, which implies a logarithmic complexity. In the machine where the tests were performed, we got equivalent performance with our implementations only when the deallocated space had a size of about 100 bytes. With allocations of 5 KiB repeated a hundres thousand times, the result is shown below.

For small values, the function `malloc` shows a performance equivalent than `_Walloc`, indicating that the lesser performance shown in this table is because of the coast for asking more memory for the Operating System with a system call. Contrary to function `free`, in `malloc` we didn't find a growth in running time when we increased the allocated size.

OpenBSD Function	Mean (μs)	Standard Deviation (μs)
`malloc`	6,30530874	25,124535168
`free`	22,00978538	12,809965439
`_Walloc`	2,50560902	0,228082461
`_Wmempoint`	2,26598656	0,505975457
`_Wtrash`	2,22814472	0,166606876

We can observe the difference between the functions in the graphic below which shows the probability density for each funcion above running in a given interval in microseconds according with the above data:



Our functions almost always run in less than 3 microseconds while the native functions

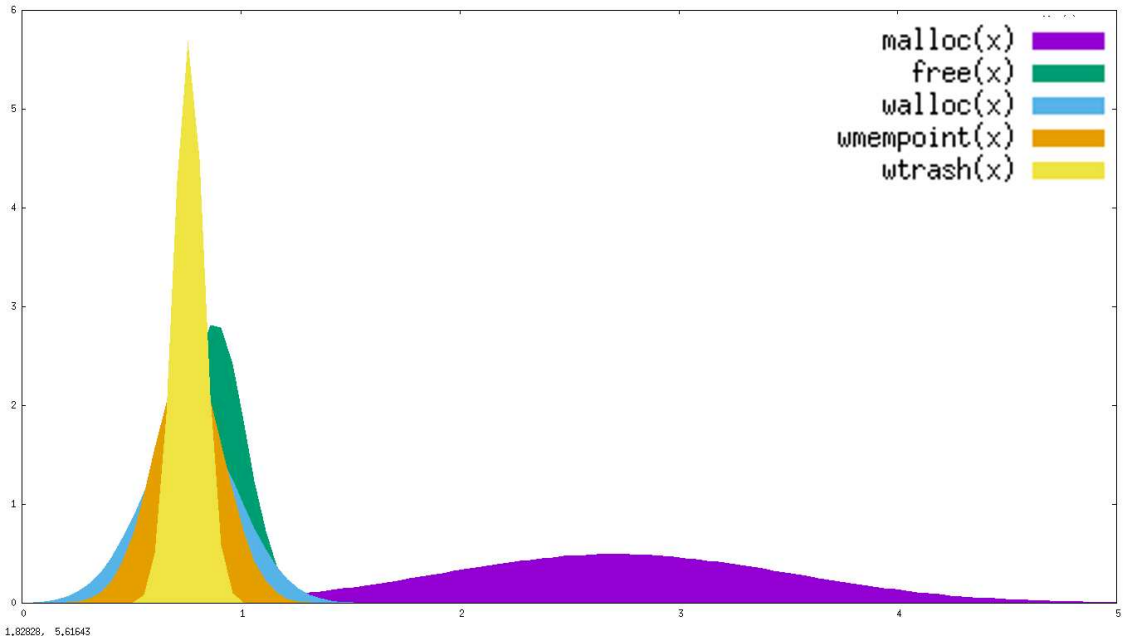
running time distribution is next to linear between 0 and a maximum value. The standard deviation here is relevant, as it interfere with the worst case observed and this should be taken in consideration evaluating the performance of this function.

3.2. Linux (Ubuntu 19.04)

Contrary to OpenBSD which implements additional measures in the allocator to randomize the obtained addresses and that always return freed memory to the kernel imediately, Linux implementations for `malloc` and `free` are more performance oriented, opting to use less security measures. All tested functions run in constant time. The data obtained are:

	Means (μs)	Standard Deviation (μs)
`malloc`	2,71246735	0,801445880
`free`	0,87757830	0,141186377
`_Walloc`	0,76750440	0,218803555
`_Wmempoint`	0,75614315	0,155982817
`_Wtrash`	0,75579410	0,066124620

All the functions have a very similar performance, running in about 1 microsecond. Only `malloc` function from standard library present a worst performance and with more deviation. Comparing the worst cases, our function runs about four times faster.

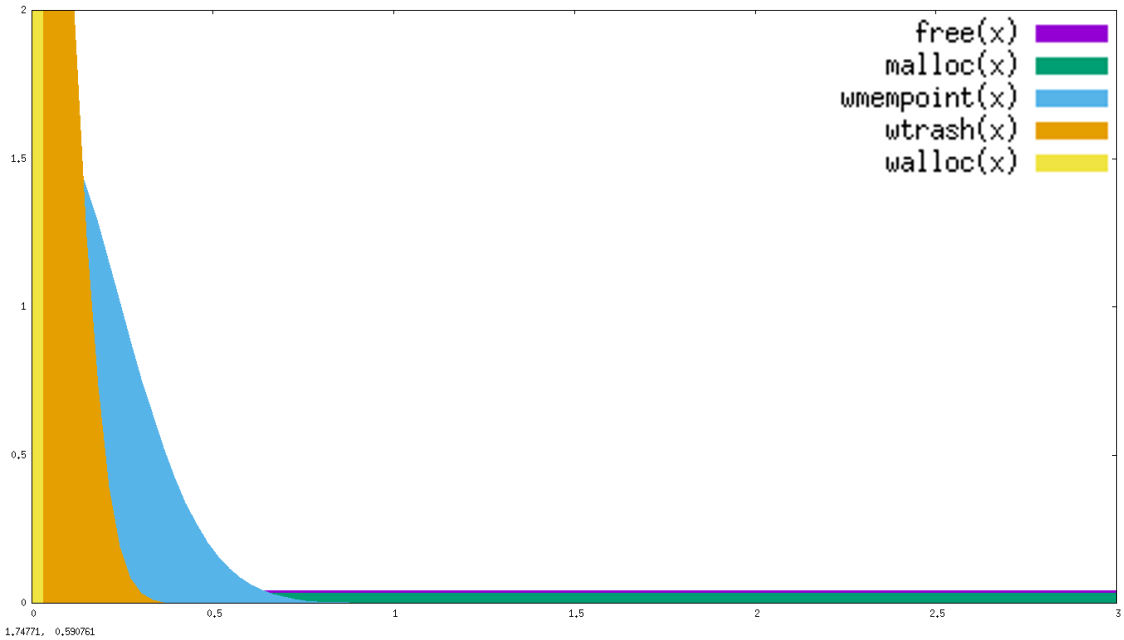


3.3. Windows 10

Windows 10 stands out for having very short execution times in average, but higher standard deviations. While the functions are fast, sometimes we can have bad luck needing to wait for longer periods before receiving the requested memory. Despite the fast native implementation compared with other systems, our implementation managed to achieve even shorter execution times with much smaller standard deviation.

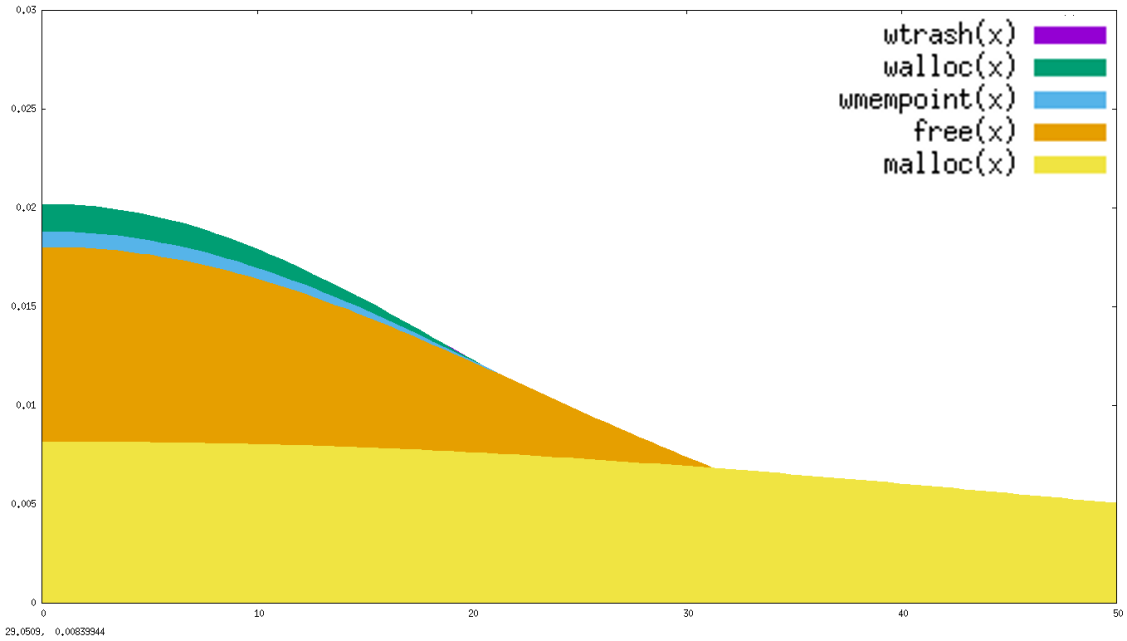
Windows Function	Mean (μs)	Standard Deviation (μs)
`malloc`	1,59881	11,466939064
`free`	0,57427	9.402517112
`_Walloc`	0,00003	0.005477171
`_Wmempoint`	0,00515	0,233332428
`_Wtrash`	0,00045	0,097825831

The defferent behaviour in Windws functions indicates the differen philosophies between Windows and Linus implementations. In Windows the priority appears to be achieving smaller averages for execution times, even if sometimes we need to deal with a rare worst case scenario where the performance becames terrible. In Linux the priority is avoiding unpleasant surprises with the worst case scenario, even if this makes our average case slower.



3.4. Web Assembly (Firefox Quantum 67.04)

Web Assembly (Firefox)	Mean (μs)	Standard Deviation (μs)
`malloc`	2,399475651	48.931141529
`free`	0,489213791	22,130980751
`_Walloc`	0,394548569	19,744325980
`_Wmempoint`	0,451152325	21,208728454
`_Wtrash`	0,398001614	19,744481718



While running Web Assembly environment in a web browser, predictably all our functions presented bigger standard deviations. The big surprise is that the average time sometimes managed to be smaller comparing with the Operating System (this test was done in Linux). Our functions also presented less standard deviation than functions from system libraries.

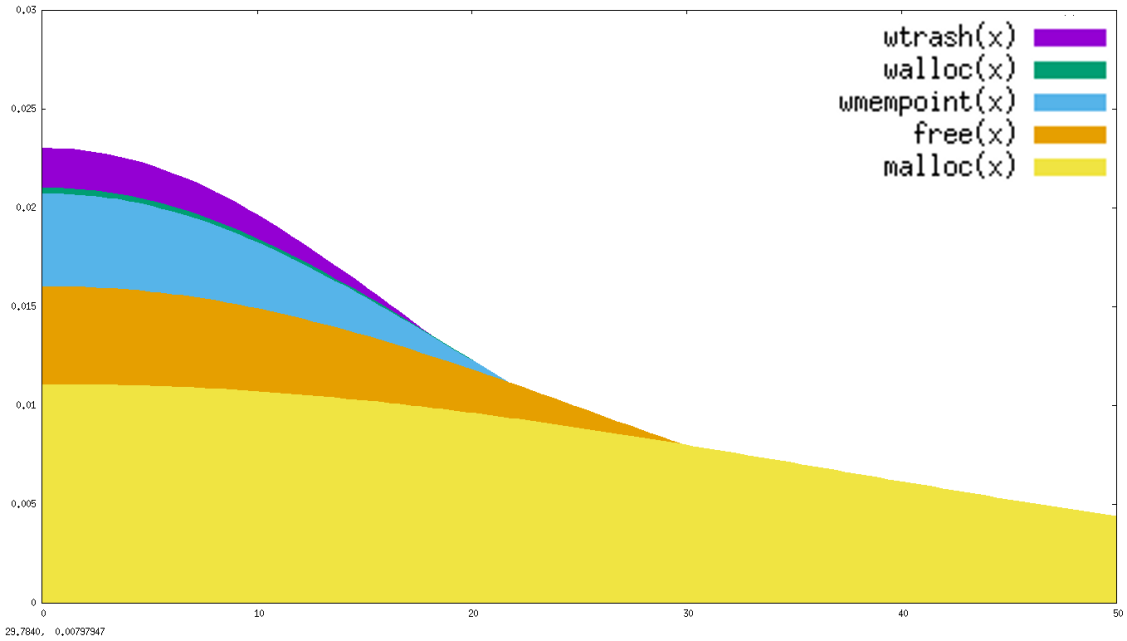
In the above graph, we can't distinguish the functions `_Walloc` and `_Wtrash`, because both achieved practically identical performance.

3.5. Web Assembly (Microsoft Edge 40.15254.369)

Running Web Assembly code in Microsoft Edge at Windows 10 we got a performance similar to Web Assembly running at Firefox. The difference is that here the functions spend a more predictably time running, with a smaller standard deviation. Here `malloc` running in the web browser had a better running time in average comparing with `malloc` running natively.

Web Assembly (Edge)	Mean (μs)	Standard Deviation (μs)
`malloc`	1,075476093	36,039479231
`free`	0,620922375	24,892216258
`_Walloc`	0,366360054	18,970295969
`_Wmempoint`	0,346548996	19,232215457
`_Wtrash`	0,302235565	17,317682053

The graph below uses the same scale than the previous one with Firefox results.



4. Conclusion

In all test cases, the functions developed here present a better performance and a more predictably running time comparing with native implementations. This is because in our implementation we don't need to use system calls asking for more memory. We use a single system call at initialization and after this our functions manage this single big chunk of memory.

This have the drawback of requiring us to specify at initialization what the maximum ammount of memory our program needs. The second drawback is that we don't have the function `free`. All deallocation must happen in blocks. In applications adaptable to this scheme, the performance gains are even bigger, because instead of successive calls to `free`, we can use a single `_Wtrash` call to free many allocations at once.

In some cases we can use in sequentially `_Wmempoint`, `_Walloc` and `_Wtrash` to simulate a call of `malloc` followed by `free` when we need a temporary and short lived allocation. In this case, the results show that calling the three functions is faster than calling the two (`malloc` and `free`) in all tested platforms.

The previous disadvantages are usually not a big problem in some applications as game development. In them we usually shouldn't spend more than a fixed and documented ammount of resources like memory. A game can easily be organized to fit in our stack-based allocation scheme. If so, the functions defined here bring benefits like less loading times thanks to faster allocations, which in a game usually happends during the initialization of a new scenario or stage.

References

Gregory, J. (2019) "Game Engine Architecture", CRC Press, third edition.

Zakai, A. (2011) "Emscripten: an LLVM-to-JavaScript compiler", Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, p. 301–312.

Knuth, D. E. (1984) "Literate Programming", The Computer Journal, volume 27, edio 2, p. 97–111

Ranck, S. (2000) "Game Programming Gems", Charles River Media, volume 1,

edition 1, p. 92–100