Gerenciador de Memória Weaver

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.*

Resumo: *Este artigo descreve usando programação literária um gerenciador de memória escrito para o Motor de Jogos Weaver. Ele é voltado a programas onde a memória é alocada e liberada seguindo uma ordem baseada em pilha e a quantidade máxima de memória é conhecida. Ele fornece a possibilidade de criar marcações durante a execução do programa de modo que depois possam liberar toda a memória alocada desde a última marcação. Depois da criação do gerenciador de memória, também comparamos o desempenho dele com o malloc da biblioteca C padrão rodando em Linux, Windows e em um navegador de Internet usando Web Assembly.*

1. Introdução

1.1. Gerenciadores de Memória em Motores de Jogos

Muitos motores de jogos implementam seus próprios gerenciadores de memória ao invés de utilizar as funções da biblioteca de sistema para obter memória dinamicamente. Segundo [Gregory 2019], isso ocorre porque, a implementação de funções como *malloc* e *free* costumam ser relativamente lentas quando utilizadas em jogos quando comparados a gerenciadores feitos sob medida, além de poderem causar fragmentação de memória.

Gregory identifica cinco padrões de projeto comuns utilizados em gerenciadores de memória de jogos:

Alocadores Baseados em Pilha: Esses alocadores sempre alocam novas regiões de memória sequencialmente e toda desalocação deve ser feita em ordem inversa à alocação. A implementação é bastante simples e busca manter a localidade espacial das regiões de memória usadas. Toda a memória alocada à partir de qualquer ponto de execução pode ser desalocada muito rapidamente com uma única chamada de função.

Alocadores de Reservatório: Essa técnica pode ser usada quando sabemos que podemos precisar de até n tipos de elementos, todos eles com o mesmo tamanho. Pode-se então alocar com antecedência o espaço para todos os elementos e manter ele sempre à disposição usando variáveis para definir quando o elemento realmente existe e quando podemos considerar sua posição com livre.

Alocações Alinhadas: Diferentes tipos de dados e arquiteturas podem ter diferentes restrições de alinhamento de memória alocada. A plataforma de jogos

Playstation 3, por exemplo, requer que qualquer posição de memória a ser passada por meio de DMA (*Direct Memory Access*) tenha um alinhamento de 128 bits. Ou seja, o endereço precisa ser um múltiplo de 128 bits.

Alocações de Quadro Único e Duplo: Um motor de jogos divide sua computação em quadros, sendo que em cada quadro uma nova imagem é enviada para a tela. Podem existir então variáveis alocadas que devem ter um tempo de vida de um ou dois quadros somente, podendo ser desalocadas automaticamente depois disso.

Desfragmentação de Memória: Quando uma memória é alocada e liberada em ordem qua não pode ser controladas, podem se formar lacunas entre espaços de memória alocados, as quais podem ser muitas e ao mesmo tempo com um tamanho pequeno demais para serem reaproveitadas. Para evitar isso é comum usar ao invés de ponteiros para objetos alocados, índices de referência para os ponteiros em si. Desta forma, regiões de memória alocadas podem ser movidas incremental e periodicamente de modo a evitar a fragmentação.

O objetivo deste artigo será definir um alocador com alinhamento de memória definido pelo usuário baseado em pilha, o qual consegue armazenar duas pilhas diferentes na região de memória gerenciada por ele. Gerenciar qual das pilhas a ser usada será responsabilidae do usuário. Essa técnica é descrita em [Ranck, 2000] que mostra que ela foi usada no jogo *Hydro Thunder* da empresa *Midway*.

Alocadores de reservatório não serão tratados aqui, mas podem ser construídos sobre o alocador presente aqui como uma segunda camada de gerenciamento. O problema da desfragmentação também não será tratado, pois alocadores baseados em pilha não sofrem com o problema da fragmentação.

1.2. Ambientes de Execução

Neste artigo vamos nos preocupar em garantir que as funções definidas rodem com sucesso em quatro ambientes diferentes: Windows 10, macOS Sierra, Linux e Web Assembly. Os três primeiros são os três sistemas operacionais para computadores pessoais. O último é uma especificação de máquina virtual especializada em interpretar um subconjunto otimizado de Javascript, criada para permitir a execução de programas de computador completos dentro de ambientes como navegadores de Internet. Seu desenvolvimento partiu do método apresentado por [Zakai, 2011] que ofereceu um modo de compilar código em C e C++ para Javascript de maneira otimizada.

Para desenvolver de maneira portável nos quatro ambientes, serão usadas macros condicionais e as diferentes formas de obter memória no diferentes sistemas será comparada.

1.3. Programação Literária e Notação Usada no Artigo

Este artigo utiliza a técnica de "Programação Literária" para desenvolver o seu gerenciador de memória. Esta técnica foi apresentada em [Knuth, 1984] e consiste e uma filosofia de desenvolvimento de *software* na qual um programador desenvolve um programa escrevendo e explicando didaticamente o código necessário, se preocupando em deixar o seu funcionamento claro para as pessoas que lerem a explicação. Ferramentas automáticas são então utilizadas para extrair o código existente na expicação, mudar a ordem do código conforme for mais adequado para o compilador e produzir à partir do código extraído um programa executável.

Por exemplo, neste artigo serão definidos dois arquivos diferentes: `memory.c` e `memory.h`, os quais podem ser inseridos estaticamente em qualquer projeto, ou compilados como uma biblioteca compartilhada. O que colocaremos em `memory.h` será:

Arquivo: `src/memory.h`:

```
#ifndef WEAVER_MEMORY_MANAGER
#define WEAVER_MEMORY_MANAGER
#ifdef __cplusplus
extern "C" {
#endif
                <Seção a ser Inserida: Declarações de Memória>
#ifdef __cplusplus
}
#endif
#endif
```

As duas primeiras linhas assim como a última são macros de segurança que impedem que as funções e variáveis declaradas ali sejam declaradas redundantemente se alguém incluir mais de uma vez o arquivo em um código-fonte. As demais linhas contém macros que checam se estamos compilando usando C++ ao invés de C. Se for o caso, nós declaramos todas as funções que existem neste arquivo como funções do tipo C, para que o compilador C++ saiba que elas não serão modificadas por meio de sobrecarga de operadores e que por isso não é necessário armazenar informações adicionais além do nome da função para reconhecê-la.

No meio do código acima, deixamos indicado em letras vermelhas que iremos adicionais mais tarde ali um novo trecho de código chamado "Declarações de Memória", com a declaraçãol das funções que iremos usar. Folheando o artigo, você encontrará nas páginas seguintes um outro trecho de código cujo título não será `memory.h`, mas sim "Declarações de Memória". Será ali que o código que vai nesta parte do arquivo será encontrado. Pdem existir mais de um trecho de código com este título. Isso significa que para produzir o código funcional utilizado pelo compilador, devemos concatenar todos estes trechos com o mesmo título e colocar na parte indicada deste arquivo. Isso nos permite colocar a declaração de funções à medida que formos explicando elas no código, sem precisar declarar todas de uma vez só porque elas pertencem a um mesmo trecho de código.

1.4. Funções a serem Definidas

Nosso gerenciador de memória irá definir um total de 6 novas funções. A primeira recebe um tamanho em bytes e retorna uma região contínua de memória a ser gerenciada (que chamamos de "arena"):

Seção: Declarações de Memória:

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

A segunda recebe uma arena criada pela primeira e libera toda a memória reservada nela. Se haviam elementos não-desalocados na arena, retornamos false, caso contrário retornamos verdadeiro:

Seção: Declarações de Memória (continuação):

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

A terceira equivale a um `malloc` e recebe uma arena, um número que será uma potência de dois ou zero que representa o alinhamento em bits que a memória alocada precisará respeitar, um número que indica se queremos alocar da pilha esquerda (0) ou direita (1) da arena e um tamanho em bytes. A função sempre retornará um endereço múltiplo do alinhamento, assumindo que ele é uma potência de dois:

Seção: Declarações de Memória (continuação):

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

A quarta função serve para colocar uma marcação, a qual chamamos de "breakpoint" dentro de uma arena onde alocamos memória. Essa marcação pode ser usada para determinar que alocações ocorreram antes e quais ocorreram depois que ela foi feita. A flag indica se essa marcação deve ser colocada na região de alocação esquerda (0) ou direita (1). A função retorna se foi bem-sucedida ou não.

Seção: Declarações de Memória (continuação):

```
bool Wbreakpoint(void *arena, int regiao);
```

A última função usa a marcação criada pela função anterior e desaloca de uma só vez toda a memória alocada com `Walloc` desde que a última marcação foi criada. Ela recebe uma flag para saber se isso deve ser feito na região esquerda (0) ou direita (1):

Seção: Declarações de Memória (continuação):

```
void Wtrash(void *arena, int regiao);
```

2. Implementação

2.1. Obtendo uma Região de Memória Inicial

Em um gerenciador de memória de propósito geral nós não temos como saber qual a quantidade máxima de memória que iremos precisar. Mas em um gerenciador de memória usado em jogos, é essencial que estabeleçamos um limite máximo de uso de memória. Em tais casos, estamos menos interessados em obter resultados precisos de computação e mais interessados em garantir um desempenho contínuo, sem perda de performance súbita como quando esgotamos a memória principal e temos que usar memória *swap*. Nestes casos é interessante alocar de uma só vez a quantidade máxima de memória que nos comprometemos a usar e gerenciar esta mesma quantidade durante o tempo de execução do programa.

O modo que utilizaremos para obter essa quantidade de memória inicial varia dependendo do ambiente de execução. Pode-se usar até mesmo um `malloc` para fazer isso, embora aqui nós iremos preferir usar a alternativa que desperdice menos memória possível. Se possível uma que retorne exatamente o valor que queremos sem gastar qualquer valor adicional preenchendo informações e estruturas de dado adicionais.

Tanto em qualquer sistema Unix (Linux, OpenBSD, macOS) como quando compilamos para WebAssembly com o compilador Emscripten, a escolha mais econômica é usar o `mmap`. Ee é uma função bastante ampla que permite mapear para a memória qualquer coisa, de arquivos em disco até simplesmente alocar uma região nova de memória para um programa.

Para podermos usar `mmap`, só temos que inserir o cabeçalho adequado:

Seção: Incluir Cabeçalhos Necessários:

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

Tendo definido o cabeçalho, para usar a função para alocar uma região de tamanho M que não está associada a nenhum arquivo, somente à memória física e que pode tanto ser lida como escrita, invocamos ela da seguinte forma:

Seção: Alocar em 'arena' região de 'M' bytes:

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

Os argumentos que foram setados para nulo, zero ou -1 são apenas argumentos

que não são necessários no modo que estamos usando o `mmap`.

Assim como nós criamos uma nova região de memória para usarmos, depois vai ser necessário desfazer ela. Neste caso, usamos o `munmap`:

Seção: Desalocar 'arena' de tamanho 'M' bytes:

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

Em ambientes Windows, a função equivalente a isso é a a `CreateFileMapping`, com a diferença de que ela retorna um controlador que precisa de uma função adicional para por fim obter um ponteiro para a região alocada. Felizmente, segundo a documentação da API do Windows, é permitido fechar tal controlador com o `CloseHandle` antes de desalocar e desfazer a região para a qual o ponteiro aponta. Graças à isso, conseguimos manter a simetria entre o código Windows e das demais plataformas, pois como o controlador será fechado aqui, não precisamos memorizá-lo com uma estrutura adicional a ser encerrada futuramente.

Seção: Alocar em 'arena' região de 'M' bytes (continuação):

```
#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
```

Para desalocar a região alocada usaremos o `UnmapViewOfFile`:

Seção: Desalocar 'arena' de tamanho 'M' bytes (continuação):

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

Usar as funções acima requer os seguintes cabeçalhos:

Seção: Incluir Cabeçalhos Necessários (continuação):

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

2.2. Obtendo o Tamanho da Página

Em um computador real, em contraste com uma máquina virtual, quando um programa pede memória para o Sistema Operacional, ele sempre irá receber memória em múltiplos do tamanho da página usada internamente pela máquina. Tipicamente o tamanho de uma página é de 4 KiB. Sendo assim, não adianta pedirmos quantidade de memória que não seja múltiplo de 4 KiB. Se pedirmos apenas 2 KiB, continuaremos recebendo 4 KiB. Se pedirmos 5 KiB, receberemos 8 KiB.

É importante então que nestes ambientes, nosso gerenciador esteja ciente disso e que mesmo que um usuário peça uma quantidade de memória que não seja múltipla do tamanho da página, ele sempe irá ajustar o pedido para um valor múltiplo para

assim não desperdiçar memória.

Na maioria dos sistemas Unix, desde que compatíveis com o POSIX, podemos obter facilmente o tamanho de uma página por meio da função `sysconf`:

Seção: Obter tamanho de página 'p':

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

A documentação do macOS afirma que ele possui tal função que é compatível com o POSIX. Contudo, a documentação não menciona a opção de obter o tamanho da página por ela. Ao invés disso, a documentação menciona usar a função BSD `getpagesize` para obter tal informação. Para nos mantermos seguindo a documentação, faremos isso então:

Seção: Obter tamanho de página 'p' (continuação):

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

Tanto a função BSD acima como a função POSIX estão definidas no mesmo cabeçalho:

Seção: Incluir Cabeçalhos Necessários (continuação):

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

No Windows, o modo de obter o tamanho da página é por meio da função mais complexa `GetSystemInfo` que retorna também uma série de informações adicionais sobre o sistema que não precisaremos usar no momento.

Seção: Obter tamanho de página 'p' (continuação):

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

E para usar esta função, a documentação nos aconselha incluir diretamente o cabeçalho `windows.h`:

Seção: Incluir Cabeçalhos Necessários (continuação):

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

Por fim, o caso do ambiente WebAssembly. Neste ambiente, a memória dinâmica é obtida por meio de uma seção de memória linear que começa com um tamaho padrão e pode crescer por meio de um operador `grow_memory`, o qual também é configurado com um tamanho máximo. De qualquer forma, aqui também a quantidade de memória alocada é múltipla de uma página. O tamanho da página na máquina virtual WebAssembly é documentado como sendo sempre o mesmo:

Seção: Obter tamanho de página 'p' (continuação):

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

```
#endif
```

2.3. Sobre Execução em Diferentes Threads

É importante garantir que o código definido aqui não deixe de funcionar quando invocado simultaneamente por mais de um trecho de código. Para isso, iremos apenas garantir um *mutex* para que o nosso gerenciador de memória não tenha a mesma região de memória sendo acessada por mais de uma *thread*. Caso tenhamos um caso no qual várias threads precise alocar memória simultaneamente, pode ser mais vantajoso dar para cada uma delas a sua própria região de memória para evitar que cada uma delas bloqueie todas as outras durante sua alocação.

Nem todos os ambientes suportam threads. A máquina virtual Web Assembly, ao menos a versão implementada nos navegadores de Internet não oferece suporte à elas, exceto em versões experimentais. Sendo assim, definiremos código para nosos mutex somente para os demais ambientes. No caso do Windows, iremos preferir usar a API para "Seções Críticas" ao invés de usar diretamente um mutex. O principal motivo é evitar uma chamada de sistema para o Kernel caso ninguém esteja usando nosso Mutex. As seções críticas conseguem isso apenas trazendo a restrição de não poderem ser compartilhadas com outros programas. No mais, as seções críticas funcionam de maneira análoga a um mutex.

Em sistemas baseados em Unix, incluimos o cabeçalho da biblioteca POSIX "pthreads". No Windows, os cabeçalhos relevantes já foram incluídos no `windows.h`.

Seção: Incluir Cabeçalhos Necessários (continuação):

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

Um mutex é declarado da seguinte forma:

Seção: Declaração de Mutex:

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

Quando o mutex é inicializado, colocamos em uma variável booleana chamada `error` se ocorreu algum problema. No caso da biblioteca *pthreads*, sua função de inicialização já retorna valor não-nulo se ocorreu um problema. No Windows, o sistema garante que a função de inicialização nunca falhe. Nos dois casos assumimos que temos um ponteiro genérico para o nosso mutex.

Seção: Inicialização de '*mutex':

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

O código para destruir um mutex é o exposto abaixo. Neste caso não iremos nos preocupar com casos de erro.

Seção: Finaliza '*mutex':

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

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

A operação clássica de *wait* para requerer o uso do mutex:

Seção: '*mutex':WAIT():

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

E a função de *signal* para liberar um mutex que foi pedido pela thread:

Seção: '*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. Cabeçalho de Arenas de Memória

Logo após alocarmos uma região de memória (ou "arena"), a primeira coisa a fazer é reservar os seus bytes iniciais para armazenar informações gerais sobre ela. As informações que desejamos são: o tamanho total de espaço que temos (`remaining_space`), o tamanho total da arena (`total_size`) e ponteiros para o começo de uma região livre na pilha de memória esquerda e direita que iremos alocar (`left_free`, `right_free`).

Precisaremos também de um mutex como o que definimos para proteger a arena de ser manipulada simultaneamente por duas threads.

Outra informação que podemos precisar só em momentos de depuração é a quantidade mínima de memória que chegamos a ter ao longo do tempo de vida da arena. Isso é útil de armazenar porque após encerrarmos a arena e descobrirmos que alocamos para a arena muita memória que não foi usada, podemos querer diminuir o tamanho que demos a ela. Usaremos a variável `smallest_remaining_space` para armazenar isso e ela só estará definida se a macro `W_DEBUG_MEMORY` estiver definida.

O cabeçalho de nossa arena de memória terá então a seguinte forma:

Seção: Cabeçalho da Arena:

```
struct arena_header{
                <Seção a ser Inserida: Declaração de Mutex>
  void *left_free, *right_free;
  size_t remaining_space, total_size;
#if defined(W_DEBUG_MEMORY)
  size_t smallest_remaining_space;
#endif
};
```

Assumimos que nós temos um ponteiro para a arena de memória recém-obtida, o qual chamamos de `arena` e que nós temos o tamanho total em bytes que está alocado nela na variável `M`, então conseguimos inicializar o cabeçalho ali com o código abaixo. Para calcular as próximas posições onde iremos inserir, convertemos o ponteiro para o começo da arena para um ponteiro de caractere para podermos fazer nossos cálculos com aritmética de ponteiro com múltiplos de 1 byte. O próximo espaço que temos livre para a pilha de memória esquerda é o byte imediatamente

após o cabeçalho. Já a pilha direita fica com o último byte da arena.

Seção: Inicializa cabeçalho em 'arena' de tamanho '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 -> total_size = M;
#if defined(W_DEBUG_MEMORY)
  header ->  smallest_remaining_space = header -> remaining_space;
#endif
  { // Mutex initialization
    void *mutex = &(header -> mutex);
                <Seção a ser Inserida: Inicialização de '*mutex'>
  }
}
```

2.4. O Código de Alocação de Nova Arena

Com o código que definirmos já temos como definir a função `Wcreate_arena`. O que a função terá que fazer são as 6 operações abaixo:

1. Receber do usuário um tamanho `t` para alocar.
2. Descobrir qual o tamanho da página `p` no sistema atual.
3. Obter `M`, sendo igual ao menor múltiplo de `p` maior que `t`. Se `M` for menor que o espaço necessário para o cabeçalho da arena, ele se torna igual ao menor múltiplo de `p` que é maior que o tamanho do cabeçalho.
4. Alocamos uma nova arena de tamanho `M`.
5. Inicializamos o seu cabeçalho.
6. Retornamos o ponteiro para o começo da arena, onde está seu cabeçalho, ou `NULL` em caso de problemas.

O código para fazer isso será então:

Seção: Definição de '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:
              <Seção a ser Inserida: Obter tamanho de página 'p'>
  // Operation 3:
  M = (((t - 1) / p) + 1) * p;
  if(M < header_size)
    M = (((header_size - 1) / p) + 1) * p;
  // Operation 4:
          <Seção a ser Inserida: Alocar em 'arena' região de 'M' bytes>
  // Operation 5:
     <Seção a ser Inserida: Inicializa cabeçalho em 'arena' de tamanho 'M'>
  // Operation 6:
  if(error) return NULL;
  return arena;
}
```

2.5. A Função "Wdestroy_arena"

Uma vez que fornecemos uma forma de criar novas arenas, vamos definir também a função que irá finalizá-las. Esta é uma função mais simples que fará quatro coisas:

1. Destruirá o mutex associado à arena.
2. Imprimirá um aviso na tela se existe alguma coisa ainda alocada na arena.
3. Se estamos em modo de depuração, imprimirá a quantidade de memória que nunca chegou a ser usada pela arena.
4. Devolverá para o Sistema Operacional a memória que ele pediu para a arena.

Seção: Definição de '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;
                    <Seção a ser Inserida: Finaliza '*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
        <Seção a ser Inserida: Desalocar 'arena' de tamanho 'M' bytes>
  return ret;
}
```

Embora isso nos faça ter que incluir o cabeçalho das funções de entrada e saída padrão se estivermos em modo de depuração:

Seção: Incluir Cabeçalhos Necessários (continuação):

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

2.6. Mantendo o Alinhamento em Memória Alocada

Quando vamos alocar uma nova memória começamos com um endereço p que está disponível no qual iremos colocar nossa memória alocada. Mas devemos respeitar o alinhamento a, o qual é uma potência de 2 ou o número zero (que significa "qualquer alinhamento"). O modo de fazer isso depende se estamos na pilha esquerda (posição inicial da memória na arena) ou direita (posição final da memória na arena). Pois uma das pilhas de memória cresce para posições maiores e outra para menores.

No caso da memória da pilha esquerda, vamos querer colocar nossa alocação em um endereço p, mas podemos ter que deslocar p um pouco mais para os próximos endereços para poder alinhar a memória. Fazemos isso considerando que $a-1$ representa tanto o pior caso de quantidade de posições que vamos deslocar p como uma máscara de bits que devem ser nulos para que o endereço seja válido, pelo fato de a ser uma potência de dois. Sendo assim, nosso código de alinhamento é:

Seção: Alinha 'p' e marca 'offset' de acordo com 'a' (esquerda):

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

```
}
```

Já na pilha direita, vamos querer alocar em um endereço p, mas possivelmente teremos que dimminuir o endereço inicial para mantê-lo alinhado ao invés de aumentá-lo. Com isso não é necessário somar o endereço inicial com o valor de pior caso $a-1$, basta remover diretamente os bits finais que forem necessários para o alinhamento:

Seção: Alinha 'p' e marca 'offset' de acordo com 'a' (direita):

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

2.7. Alocando Memória

Antes de alocar memória, temos que verificar se existe espaço disponível. Fazemos isso checando os valores do cabeçalho da arena e comparando com o valor que temos que alocar, considerando o pior caso de alinhamento, onde precisaremos de um espaço adicional de $a-1$. Se não houver espaço, o valor p a ser retornado terá que ser `NULL`.

Se vamos alocar na pilha esquerda, o nosso endereço alocado será a próxima posição depois desse cabeçalho após passar por uma correção de alinhamento.

Se vamos alocar na pilha direita, obtemos o valor do endereço alocado indo para a próxima região livre marcada no cabeçalho da arena e subtraímos do endereço o tamanho que queremos alocar sem o cabeçalho, somando 1 ao resultado. Assim teremos à nossa direita a quantidade certa de memória que precisamos retornar ao usuário. Mas antes disso, novamente fazemos a correção necessária de alinhamento.

Uma vez que isso foi feito, basta atualizarmos o cabeçalho da arena e `p` está pronto para ser retornado.

Seção: Alocação de 'p', tamanho 't' em 'arena', alinhamento 'a':

```
{
  int offset;
  struct arena_header *header = (struct arena_header *) arena;
  if(header -> remaining_space >= t + ((a == 0)?(0):(a - 1))){
    if(right){
      p = ((char *) header -> right_free) - t + 1;
        <Seção a ser Inserida: Alinha 'p' e marca 'offset' de acordo com 'a'
(direita)>
    }
    else{
      p = ((char *) header -> right_free) + sizeof(struct alloc_header);
        <Seção a ser Inserida: Alinha 'p' e marca 'offset' de acordo com 'a'
(esquerda)>
    }
    header -> remaining_space -= (t + offset);
#if defined(W_DEBUG_MEMORY)
    if(header -> remaining_space < header -> smallest_remaining_space)
      header -> smallest_remaining_space = header -> remaining_space;
#endif
  }
```

2.8. A Função Walloc

Podemos agora juntar as peças para definir a função de alocação. Ela irá receber

`arena` (arena onde o usuário deseja alocar), `a` (o alinhamento), `right` (1 se desejamos alocar na pilha direita e 0 na esquerda) e `t` (tamanho que o usuário deseja alocar).

A primeira coisa a fazer é dar um "wait" no mutex da arena. Depois fazemos a alocação e então damos um "signal" no mutex. Será preciso termos também uma variável `p` que é o que iremos retornar e apontará para a região de memória requisitada pelo usuário.

Seção: Definição de 'Walloc':

```
void *Walloc(void *arena, unsigned a, int right, size_t t){
  struct arena_header *header = (struct arena_header *) arena;
  void *mutex = header -> mutex;
  void *p = NULL;
                    <Seção a ser Inserida: '*mutex':WAIT()>
   <Seção a ser Inserida: Alocação de 'p', tamanho 't' em 'arena', alinhamento
'a'>
                    <Seção a ser Inserida: '*mutex':SIGNAL()>
  return p;
}
```

2.?. Organização Final do Arquivo-Fonte

Salvaremos todo o código de definição de funções que fizemos no arquivo abaixo que poderá então ser compilado:

Arquivo: `src/memory.c`**:**

```
            <Seção a ser Inserida: Incluir Cabeçalhos Necessários>
#include "memory.h"
                  <Seção a ser Inserida: Cabeçalho da Arena>
              <Seção a ser Inserida: Definição de 'Wcreate"'arena'>
             <Seção a ser Inserida: Definição de 'Wdestroy"'arena'>
                  <Seção a ser Inserida: Definição de 'Walloc'>
```

Referências

Gregory, J. (2019) "Game Engine Architecture", CRC Press, terceira edição.

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, edição 2, p. 97–111

Ranck, S. (2000) "Game Programming Gems", Charles River Media, volume 1, edição 1, p. 92–100