Live long and prosper, Mr Spock

Grep

Home

HEAP VS STACK

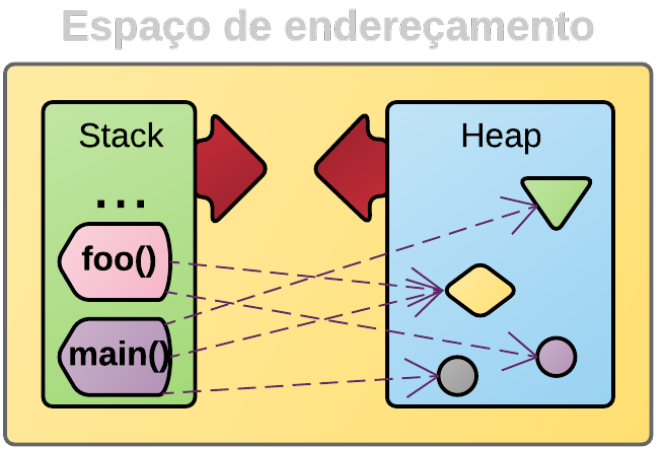
Qual a diferença entre alocação de memória feita no Heap ou na pilha de funções (stack)

Atualizado em 2017/06/28 13:35

Ativar fundo escuro

O sistema operacional ao carregar um programa na memória disponibiliza ao programa um espaço de endereçamento. Esse espaço é a memória disponível para aquele programa. O *Heap,* ou área de alocação dinâmica, é um espaço reservado para variáveis e dados criados durante a execução do programa (*runtime*). Vamos dizer que o *Heap* é a memória global do programa.

Já a pilha de funções (*stack*) é uma área da memória que aloca dados/variáveis ou ponteiros quando uma função é chamada e desalocada quando a função termina. Podemos dizer então que representa a memória local àquela função.



Um processo ao ser carregado na memória pelo sistema operacional tem outros dados armazenados em seu espaço de endereçamento além do *Heap* e da *Stack*. Esses dados são por exemplo:

. Texto - Código fonte do programa
. BSS (*Block Started by Symbol*) - Variáveis não inicializadas
. Dados - Dados de inicialização
. Heap - Espaço de alocação dinâmica
. Stack - Espaço para variáveis de funções ao serem chamadas

Iremos focar especificamente nas áreas de *Heap* e *Stack,* comparando seu funcionamento e avaliando vantagens e desvantagens.

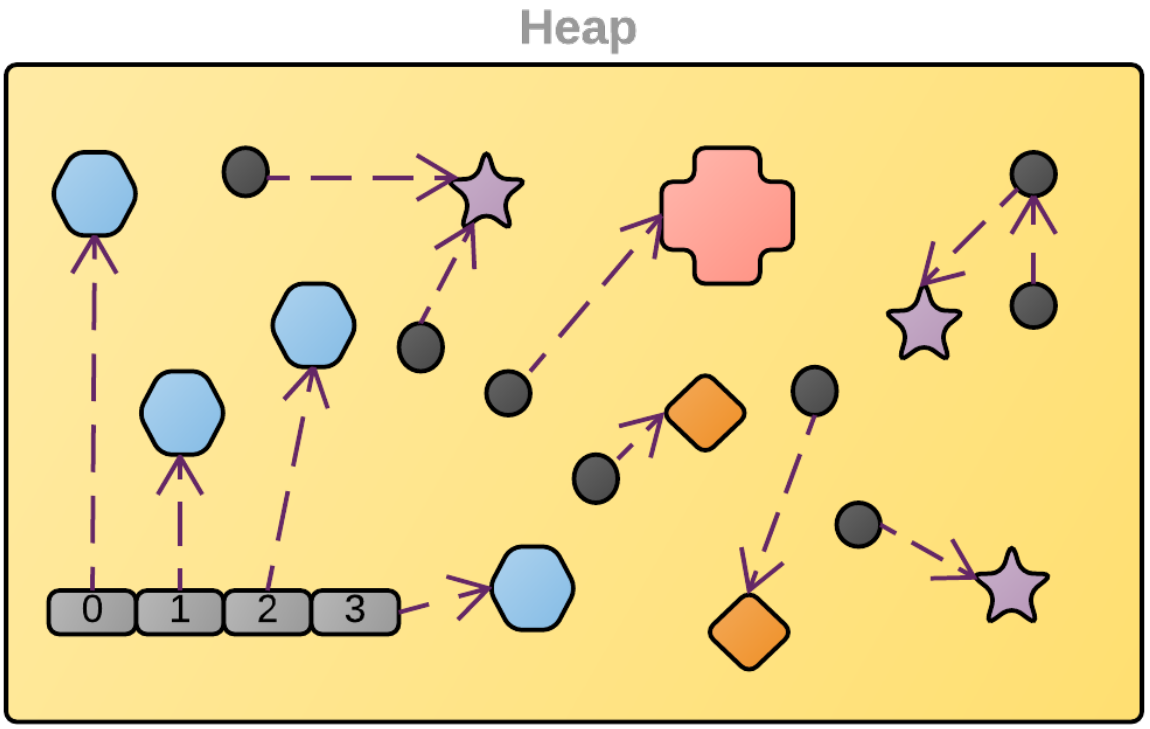
HEAP

O *heap* é uma área de alocação dinâmica de variáveis. Se um programa utiliza uma lista encadeada por exemplo, ele aloca essa estrutura que cresce dinamicamente no *Heap.* Na linguagem C/C++, para alocar memória no *Heap*, utilizamos as funções *malloc(),*

calloc(), realloc() e *new (*para C++*).* Para desalocar variáveis no *Heap* usamos as funções *free()* e *delete (*para C++*).*

Ao desalocar memória do *Heap* a área volta a estar disponível para novas alocações. À medida que muitas alocações/desalocações ocorrem no *Heap* ele sofre muita fragmentação. Isso gera impacto na performance e na eficiência de como o programa aloca memória. Memória alocadas no *Heap* permitem leitura e escrita.

Na imagem abaixo pode-se ter uma noção do que seria uma abstração de como acontece a alocação de variáveis no *Heap.* Existem áreas contíguas, grandes e pequenas sendo utilizadas. Na imagem as setas representam apontamentos feitos por ponteiros referenciando áreas de memória que contém dados ou estruturas e objetos.



STACK

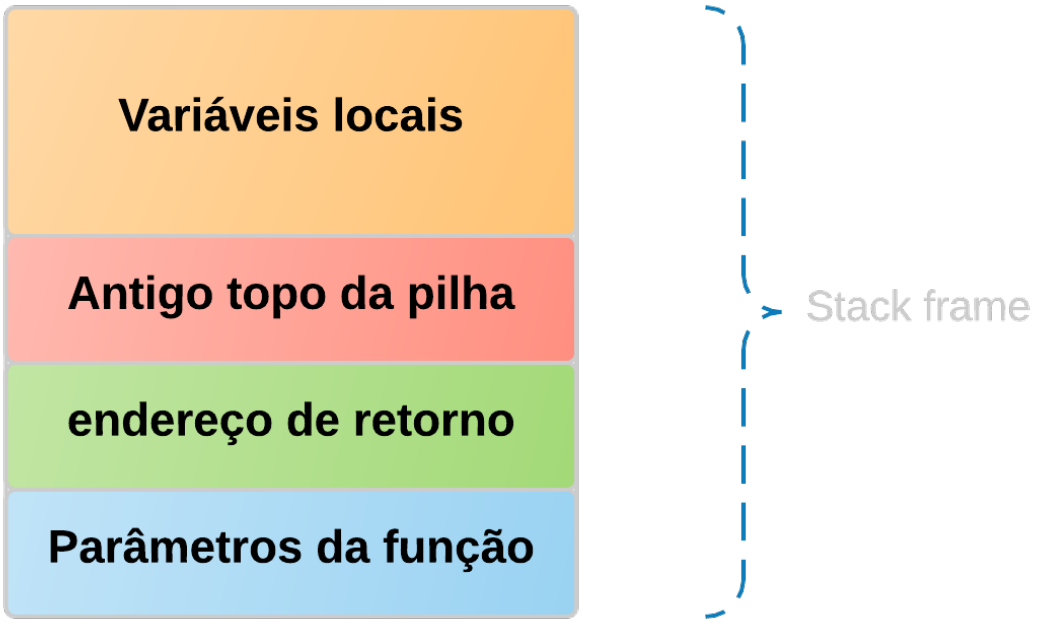
A pilha de funções também é uma área disponibilizada dentro do espaço de endereçamento

do processo. Essa área funciona como uma estrutura de dados LIFO (*last in first out*). Quando uma função é chamada durante a execução de um programa, um bloco de memória é empilhado no topo da pilha de funções. Nesse bloco existem referências para todas as variáveis criadas ou apontadas dentro da função chamada. Ao término da execução da função, esse bloco é desempilhado/desalocado.

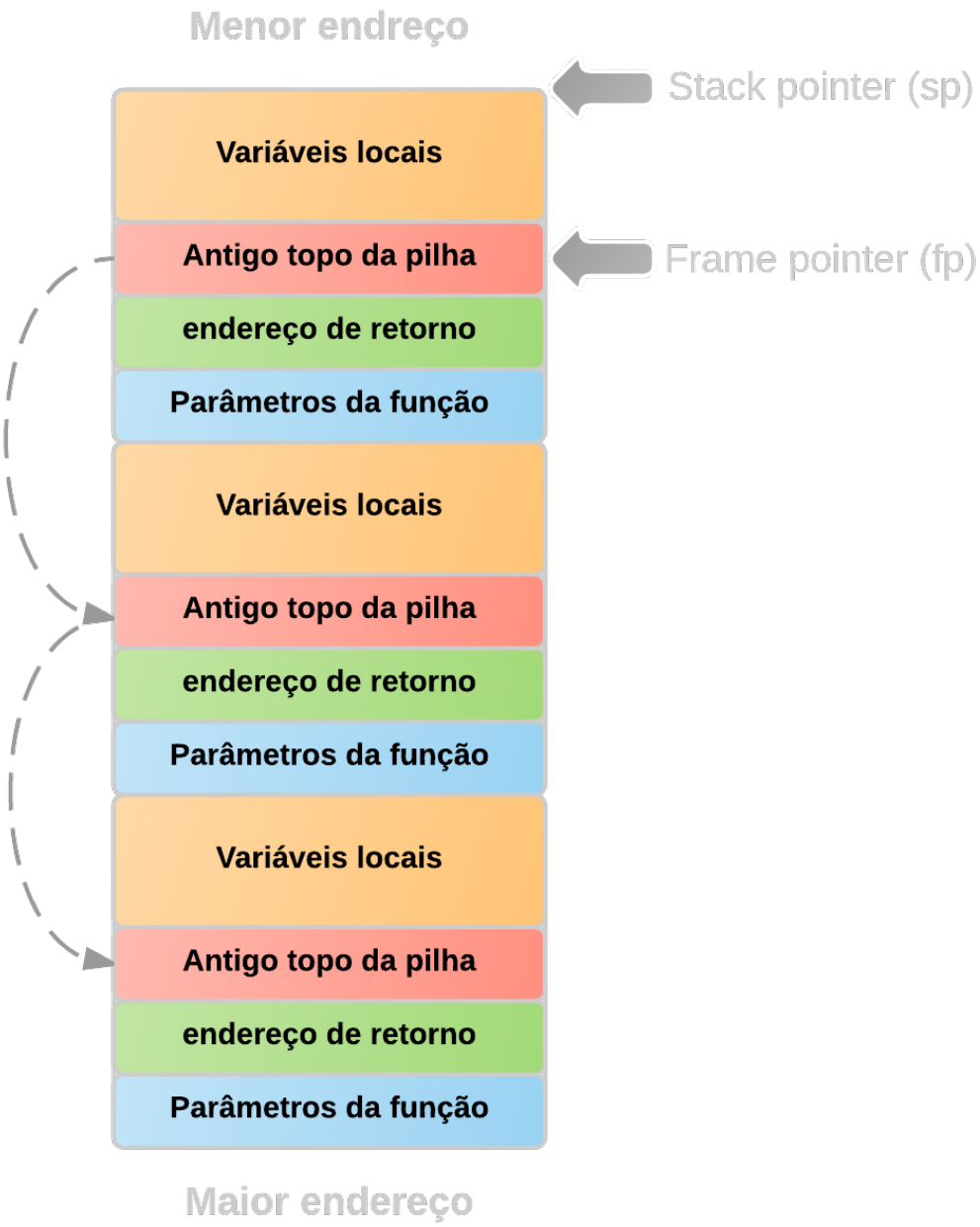
O *runtime* de cada linguagem junto ao sistema operacional irá controlar o tamanho máximo dessa pilha. Imagine um programa recursivo que faz uma chamada para uma função ***A*** dentro da própria função ***A***. Isso irá fazer inúmeros empilhamentos nessa pilha de funções. Normalmente os limites dessas pilhas são muito grandes e nem sempre é necessário se preocupar com elas.

Olhando esse funcionamento podemos concluir que o bloco alocado na pilha representa o conjunto de variáveis locais àquela função. Se eu criar uma variável dentro de uma função que seja um *array* de caracteres (*string*) e o retornar no final da função, quem tiver chamado ela não conseguirá acessar essa variável pois ela não está disponível no *Heap* (área para variáveis globais). Como a função retornou/terminou sua execução, seu bloco com referências para suas variáveis locais foram desalocados e não podem mais ser acessados. A única forma de fazer isso é se a variável for criada utilizando *malloc* dentro da função, pois o *malloc* irá garantir que a alocação acontecerá no *Heap*, logo a referência para a variável continuará sendo um endereço de memória válido para quem chamou a função.

Cada bloco empilhado é chamado de *Stack frame.* Dentro de cada frame/bloco temos variáveis para os parâmetros passados para a função, o endereço de retorno (para onde a instrução *return* aponta). Temos um ponteiro chamado *frame pointer* que representa onde o topo da pilha estava anteriormente e temos também uma área para variáveis criadas dentro da função (variáveis locais).



Temos então que para cada função chamada empilhamos um *stack frame* no topo da pilha de funções. O topo dessa pilha sempre contém o menor endereço de memória. Dessa forma se o topo da pilha for o endereço ***X***, qualquer endereço menor do que ***X*** é inválido e qualquer endereço maior do que ***X*** é um *stack frame* válido.



Na imagem acima temos então um exemplo de pilha de funções cuja altura é 3. É possível notar a variável *Stack pointer* (*sp*) que aponta para o topo da pilha. Durante a execução da função esse ponteiro pode ser atualizado, pois criamos ou utilizamos variáveis.

Para facilitar o deslocamento dentro do *stack frame*, existe outro ponteiro chamado *frame pointer*, responsável por guardar a última referência do topo da pilha (*stack pointer*) antes da chamada da função. Por sua posição dentro do *stack frame*, que é sempre no mesmo lugar, diferentemente do *stack pointer*, fica fácil deslocar até os parâmetros, até o endereço de retorno e até as variáveis locais. Como pode ser notado, a pilha sempre começa no maior endereço reservado no espaço de endereçamento para a pilha e vai crescendo no sentido dos

menores endereços. Isso garante que *stack frames* válidos sempre sejam os que possuem endereços menores ou iguais ao *stack pointer*.

COMPARAÇÃO

Vamos fazer uma comparação das duas regiões do espaço de endereçamento disponível para o programa em execução separando por tópico e fazendo uma breve avaliação.

1. Velocidade de acesso

O acesso a variáveis alocadas na *Stack* são extremamento rápidos. Como eles dependem apenas de um deslocamento de ponteiros, essa operação tem custo muito baixo. No caso do *Heap* o acesso é relativamente baixo e depende muito do *runtime (forma de* execução*)* da linguagem e da biblioteca que faz alocação. Além disso, quando a memória de um programa é muito grande e está muito fragmentada, acessar os endereços pode não ser tão rápido quanto deslocar um ponteiro.

2. Escopo

Variáveis alocadas dentro da pilha (*Stack*) são acessíveis apenas no escopo local à função responsável por aquele *stack frame*. Ao final da execução da função, ou seja, ao ser desempilhada, essas variáveis são desalocadas. Já no *Heap* temos que o escopo das variáveis é global. Tendo uma referência para o endereço da memória que contém o dado, é possível acessar essa variável dentro de qualquer função.

3. Desalocação

Variáveis alocadas no *Heap* somente são desalocadas através de uma instrução explícita do programa através de *free(), delete* no caso de C ou C++, respectivamente, ou ao final da execução do programa. Já variáveis alocadas na *Stack,* são desalocadas quando a função retorna, sendo assim desempilhada da *stack* de funções.

4. Gerenciamento de memória

Os espaços de memória utilizados pela *Stack* são eficientes e não fragmentados. O *Heap* por sua vez, fragmenta muito a memória, o que pode levar a um mal aproveitamento do espaço de endereçamento como consequência.

5. Limite

A *Stack* possui limite de crescimento. Esse limite varia de acordo com a linguagem à qual o programa foi escrito. Chamadas recursivas podem rapidamente estourar a pilha de funções de uma linguagem. No caso do *Heap*, seu crescimento é dinâmico. Quando não há mais espaço alocável no *Heap*, é solicitado ao sistema operacional através de uma chamada de sistema (*system call*) que mais memória seja disponibilizada no espaço de endereçamento do programa. Assim, o crescimento do *Heap* fica à cargo dos limites impostos e controlados pelo sistema operacional.

Vamos fazer um breve exemplo de código na linguagem C mostrando alocações na memória feitas tanto no *Heap* quanto com na *Stack*.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/**
 * Função que aloca uma área de memória no Heap. Em seu stack frame, essa
 * função não terá parâmetros. No entanto, existirão o endereço de retorno
 * da função, uma variável na pilha com o endereço do dado alocado no Heap
 * e o frame pointer que aponta para antigo topo da pilha, que nesse caso
 * será o stack frame da função main.
 */
char *
allocation_example ()
{
    /* Alocamos memória no Heap. 12 é o tamanho da string */
    char *my_string = malloc(13 * sizeof(char));

    memcpy(my_string, "Don't panic!", 12);
    my_string[12] = '\0';

    return my_string;
}


/**
 * Função principal do programa e primeira função a ser empilhada na Stack de
 * funções. Ela possui uma variável local, chamada answer, que armazena um
 * inteiro cujo valor é 42. Além disso, ela faz a chamada para a função
 * allocation_example que empilha um stack frame para essa função na Pilha
 * de funções. Como retorno da execução da função, temos apenas um ponteiro
 * para um endereço de memória no Heap que contém os dados da variável
 * new_string.
 *
 * Ao final imprimimos os valores de nossas variáveis criadas tanto na Stack
 * quanto no Heap. Além disso, ao chamar a função printf fazemos novamente
 * empilhamento na Stack de funções.
 */
int
main (int argc, char* argv[])
{
    int answer = 42;

    char *new_string = allocation_example();

    printf("%s\n", new_string);
    printf("%d\n", answer);

    return EXIT_SUCCESS;
```

```
}
```

Considerando que criamos um arquivo chamado *heap_vs_stack.c* com o código acima, para executar esse programa vamos utilizar o GNU GCC para compilá-lo da seguinte forma:

```
#
# Compila o código fonte para gerar um arquivo binário
#

$> gcc heap_vs_stack.c -o heap_vs_stack

#
# Executando o programa
#

$> ./heap_vs_stack
Don't panic!
42
```

Vimos então as principais diferenças entre o *Heap* e a *Stack*. Entendemos como eles funcionam e como eles podem sofrer mudanças de acordo com a linguagem de programação e com o sistema operacional. Cada um tem suas vantagens e desvantagens. É muito importante conhecer os limites, restrições e características de cada uma dessas áreas de alocação dentro do espaço de endereçamento de seu programa para poder fazer um melhor uso da memória e garantir maior controle sobre como armazenamos nossos dados em tempo de execução.

Assuntos relacionados

Heap | Stack | Espaço de endereçamento

Pilha de função | Pilha | Sistemas Operacionais

Memória | Gerência de memória | Alocação de memória

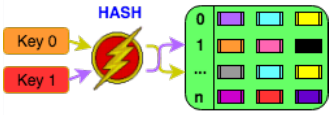
Compiladores

Gustavo Pantuza

Posts em português para fortalecer a comunidade brasileira de Ciência da computação. Caso tenha dúvidas, críticas ou sugestões de temas ou para o blog deixe nos comentários :)

Artigos recomendados

TIPOS ABSTRATOS DE DADOS - TABELA HASH



OPENVSWITCH - GUIA DE COMANDOS E CONFIGURAÇÃO

Social

Github

Linkedin

Twitter

Youtube

Facebook

Keybase

Conteúdo

Artigos

Tutoriais

Resumos

Projetos

Rss

Tags

Sobre

Sobre

UFMG

Palestras

Currículo

Lattes

Mendeley

Google Scholar

Research Gate