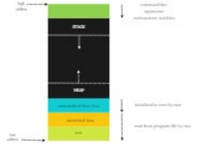
https://miro.com/app/board/uXjVO8fsSCY=/?share_link_id=911485536411

miro free Untitled

memória de programas em c

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack



1 text segment (instructions)

É uma das seções de um programa em um arquivo objeto ou na memória, que contém **instruções executáveis**.

Como uma região de memória, um segmento de texto pode ser colocado abaixo do heap ou pilha para **evitar que heaps e estouros de pilha o sobrescrevam**.

Normalmente, o segmento de texto é compartilhável, de modo que **apenas uma única cópia** precisa estar na memória para programas executados com frequência, como editores de texto, o compilador C, os shells e assim por diante.

Além disso, o segmento de texto **geralmente é somente leitura**, para **evitar que um programa modifique acidentalmente suas instruções.**

2 initialized data segment

É uma **parte do espaço de endereço virtual de um programa**, que contém as **variáveis globais** e as **variáveis estáticas** que são inicializadas.

Observe que o segmento de dados **não é somente leitura**, pois os valores das variáveis podem ser alterados em tempo de execução.

Por exemplo, a **string global** definida por

`char s[] = "hello world"`

e uma **instrução** C como

`int debug = 1`

fora do main (ou seja, global) seria armazenada na área de **leitura/gravação inicializada**.

E uma **instrução** C global como

`const char* string = "hello world"`

faz com que a string literal "hello world" seja armazenada na área de **leitura inicializada** e a variável de ponteiro de caractere string na área de **leitura-gravação inicializada.**

Exemplo:

`static int i = 10`

será armazenado no **segmento de dados** e global

`int i = 10`

também será armazenado no segmento de dados

3 Uninitialized data segment (bss)

Os dados neste segmento são **inicializados pelo kernel** para aritmética zero antes que o programa comece a executar.

Dados não inicializados começam no final do segmento de dados e **contêm todas as variáveis globais e variáveis estáticas** que são **inicializadas em zero** ou **não têm inicialização explícita** no código-fonte.

Por exemplo, uma variável declarada static **int i;** estaria contido no segmento BSS.

Por exemplo, uma variável global declarada **int j;** estaria contido no segmento BSS.

4 HEAP

Segmento onde normalmente ocorre a **alocação dinâmica de memória.**

A área de heap começa no final do segmento BSS e cresce para endereços maiores a partir daí.

A área de Heap é **gerenciada por malloc, realloc e free**, que podem usar as chamadas de sistema brk e sbrk para ajustar seu tamanho (observe que o uso de brk/sbrk e uma única "área de heap" não é necessário para cumprir o contrato de malloc/realloc/free; eles também podem ser implementados usando mmap para reservar regiões potencialmente não contíguas da memória virtual no espaço de endereço virtual do processo).

A área Heap é **compartilhada por todas as bibliotecas compartilhadas e módulos carregados dinamicamente em um processo.**

Exemplose

O comando **size()) informa os tamanhos (em bytes) dos segmentos de texto, dados e bss.**

```
#include <stdio.h>

int main(void)
{
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960    248 8 1216  4c0 memory-layout
```

2. Vamos adicionar uma **variável global** no programa, agora verifique o tamanho do bss (destacado em vermelho).

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960    248 12   1220  4c4 memory-layout
```

3. Vamos **adicionar uma variável estática** que também é armazenada em bss.

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
  static int i; /* Uninitialized static variable stored in bss */
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960    248 16   1224  4c8 memory-layout
```

4. Vamos **inicializar a variável estática** que será então armazenada no Segmento de Dados (DS)

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
  static int i = 100; /* Initialized static variable stored in DS*/
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960    252 12   1224  4c8 memory-layout
```

5. Vamos **inicializar a variável global** que será então armazenada no Segmento de Dados (DS)

```
#include <stdio.h>

int global = 10; /* Initialized global variable stored in DS*/

int main(void)
{
  static int i = 100; /* Initialized static variable stored in DS*/
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960    256 8    1224  4c8 memory-layout
```

5 STACK

A área de pilha contém a **pilha de programas**, uma estrutura **LIFO**, normalmente localizada nas **partes mais altas da memória.**

Na arquitetura de computador PC x86 padrão, ele **cresce em direção ao endereço zero**; em algumas outras arquiteturas, ela cresce na **direção oposta.** Um registrador "ponteiro de pilha" rastreia o topo da pilha; ele é ajustado cada vez que um valor é "empurrado" para a pilha. O conjunto de valores enviados para uma chamada de função é chamado de "quadro de pilha"; Um quadro de pilha consiste no mínimo de um endereço de retorno.

Stack, onde as variáveis automáticas são armazenadas, juntamente com as informações que são salvas cada vez que uma função é chamada.

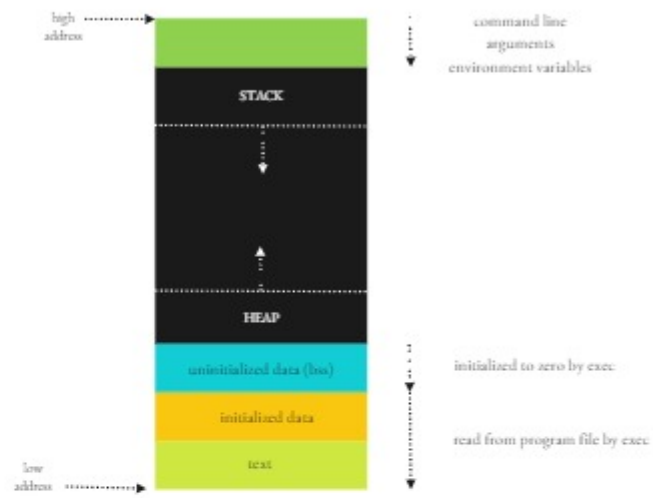
Cada vez que uma **função é chamada**, o endereço para onde retornar e certas informações sobre o ambiente do chamador, como alguns dos registros da máquina, **são salvos na pilha.**

A função recém-chamada então **aloca espaço na pilha para suas variáveis automáticas e temporárias.**

É assim que as **funções recursivas** em C podem funcionar. **Cada vez que uma função recursiva chama a si mesma, um novo quadro de pilha é usado, para que um conjunto de variáveis não interfira nas variáveis de outra instância da** função.

memória de programas em c

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack



1 text segment (instructions)

É uma das seções de um programa em um arquivo objeto ou na memória, que contém **instruções executáveis**.

Como uma região de memória, um segmento de texto pode ser colocado abaixo do heap ou pilha para **evitar que heaps e estouros de pilha o sobrescrevam**.

Normalmente, o segmento de texto é compartilhável, de modo que **apenas uma única cópia** precisa estar na memória para programas executados com frequência, como editores de texto, o compilador C, os shells e assim por diante.

Além disso, o segmento de texto **geralmente é somente leitura**, para **evitar que um programa modifique acidentalmente suas instruções**.

2 initialized data segment

É uma **parte do espaço de endereço virtual de um programa**, que contém as **variáveis globais** e as **variáveis estáticas** que são inicializadas.

Observe que o segmento de dados **não é somente leitura**, pois os valores das variáveis podem ser alterados em tempo de execução.

Por exemplo, a **string global** definida por

```
char s[] = "hello world"
```

e uma **instrução** C como

```
int debug = 1
```

fora do main (ou seja, global) seria armazenada na área de **leitura/gravação inicializada**.

E uma **instrução** C global como

```
const char* string = "hello world"
```

faz com que a string literal "hello world" seja armazenada na área de **leitura inicializada** e a variável de ponteiro de caractere string na área de **leitura-gravação inicializada**.

Exemplo:

```
static int i = 10
```

será armazenado no **segmento de dados** e global

```
int i = 10
```

também será armazenado no segmento de dados

3 Uninitialized data segment (bss)

Os dados neste segmento são **inicializados pelo kernel** para aritmética zero antes que o programa comece a executar.

Dados não inicializados começam no final do segmento de dados e **contêm todas as variáveis globais e variáveis estáticas** que são **inicializadas em zero** ou **não têm inicialização explícita** no código-fonte.

Por exemplo, uma variável declarada static **int i**; estaria contido no segmento BSS.

Por exemplo, uma variável global declarada **int j**; estaria contido no segmento BSS.

4 HEAP

Segmento onde normalmente ocorre a **alocação dinâmica de memória**.

A área de heap começa no final do segmento BSS e cresce para endereços maiores a partir daí.

A área de Heap é **gerenciada por malloc, realloc e free**, que podem usar as chamadas de sistema brk e sbrk para ajustar seu tamanho (observe que o uso de brk/sbrk e uma única "área de heap" não é necessário para cumprir o contrato de malloc/realloc/free; eles também podem ser implementados usando mmap para reservar regiões potencialmente não contíguas da memória virtual no espaço de endereço virtual do processo).

A área Heap é **compartilhada por todas as bibliotecas compartilhadas e módulos carregados dinamicamente em um processo**.

Exemplos:

O comando **size(1) informa os tamanhos (em bytes) dos segmentos de texto, dados e bss**.

```
#include <stdio.h>

int main(void)
{
 return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960   248 8 1216 4c0 memory-layout
```

2. Vamos adicionar uma **variável global** no programa, agora verifique o tamanho do bss (destacado em vermelho).

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
 960   248 12      1220 4c4 memory-layout
```

3. Vamos **adicionar uma variável estática** que também é armazenada em bss.

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
  static int i; /* Uninitialized static variable stored in bss */
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
 960   248 16      1224 4c8 memory-layout
```

4. Vamos **inicializar a variável estática** que será então armazenada no Segmento de Dados (DS)

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
  static int i = 100; /* Initialized static variable stored in DS*/
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960   252 12      1224 4c8 memory-layout
```

5 Vamos **inicializar a variável global** que será então armazenada no Segmento de Dados (DS)

```
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
  static int i = 100; /* Initialized static variable stored in DS*/
  return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ tamanho memory-layout
texto dados bss dec hex filename
960   256 8      1224 4c8 memory-layout
```

A área de pilha contém a **pilha de programas**, uma estrutura **LIFO**, normalmente localizada nas **partes mais altas da memória**.

Na arquitetura de computador PC x86 padrão, ele **cresce em direção ao endereço zero**; em algumas outras arquiteturas, ela cresce na **direção oposta**. Um registrador "ponteiro de pilha" rastreia o topo da pilha; ele é ajustado cada vez que um valor é "empurrado" para a pilha. O conjunto de valores enviados para uma chamada de função é chamado de "quadro de pilha"; Um quadro de pilha consiste no mínimo de um endereço de retorno.

Stack, onde as variáveis automáticas são armazenadas, juntamente com as informações que são salvas cada vez que uma função é chamada.

Cada vez que uma **função é chamada**, o endereço para onde retornar e certas informações sobre o ambiente do chamador, como alguns dos registros da máquina, **são salvos na pilha**.

A função recém-chamada então **aloca espaço na pilha para suas variáveis automáticas e temporárias**.

É assim que as **funções recursivas** em C podem funcionar. **Cada vez que uma função recursiva chama a si mesma, um novo quadro de pilha é usado, para que um conjunto de variáveis não interfira nas variáveis de outra instância da função**.