Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Introdução rápida ao gerenciamento de memória: ou como evitar o maldito *segmentation fault*

Moacir Ponti

2012 (v.1.1)

Sumário

1 Alocação de memória

1.1 Regiões de memória

Um programa em `C` compilado, quando executado pelo sistema operacional cria e utiliza quatro regiões de memória [1], que contém:

1. o código binário do programa
2. as variáveis globais e seus valores
3. a pilha: região de memória com usos diversos e de tamanho limitado[1], onde são armazenados os valores de variáveis locais, endereço de retorno de funções e parâmetros (argumentos) para funções, entre outros. A alocação de memória na pilha é feita de forma automática.
4. o *heap*: região de memória livre que o programa pode usar por meio de funções de alocação dinâmica. O programador é responsável pela alocação e liberação da memória nessa região.

1.2 Alocação automática

A alocação automática é feita quando declaramos variáveis locais ou globais, sejam elas variáveis simples, estruturas ou arranjos. Como no exemplo abaixo.

```
01. int varglobal_1;
02. char varglobal_2[5];
03.
04. void teste(char A) {
05.     char B = A;
06. }
07.
08. int main(int argc, char *argv[]) {
09.     float varlocal_1;
10.     int varlocal_2[2];
11.     teste('X');
12.     return 0;
13. }
```

As variáveis globais são alocadas numa região específica da memória. Elas são automaticamente liberadas quando o programa termina.

Os argumentos da função principal e do procedimento `teste()`, bem como suas variáveis locais são alocados automaticamente na *pilha*. O tipo da variável (`float`, `int`, etc.) vai definir quantos bytes serão alocados. As variáveis locais são liberadas após o retorno da função ou após a finalização do procedimento.

[1] comumente 4KB em sistemas 32bits e 8KB em sistemas 64bits

Na tabela abaixo está uma ilustração simplificada da pilha, incluindo endereço dos bytes alocados e nome da variável ou função relacionada. A tabela inclui o estado da pilha após a execução da linha 5 (o programa iniciou, a função teste foi chamada e o programa está "parado" antes do final da função)

	Pilha
0x000	`main()`
0x004	`argc`
0x008	`argv`
0x010	`varlocal_1`
0x014	`varlocal_2`
0x022	`teste()`
0x024	`A`
0x025	`B`

Após a execução da linha 11 (ou seja, o programa estaria parado na linha 12), as variáveis locais e argumentos de teste() são desalocados automaticamente, o ponteiro que controla a chamada à função não é mais necessário e a pilha ficaria na forma abaixo.

	Pilha
0x000	`main()`
0x004	`argc`
0x008	`argv`
0x010	`varlocal_1`
0x014	`varlocal_2`

Após uma variável ser alocada, o tamanho e local são fixos durante todo o tempo de execução do problema. No exemplo, a variável `varlocal_2` é um arranjo com 2 posições. Caso seja necessário aumentar o tamanho do arranjo isso não será possível, pois o espaço foi definido em **tempo de compilação**. Isso quer dizer que ao compilar foi fixado cada espaço a ser alocado. Apesar de podermos utilizar esse espaço livremente, durante a execução do programa não será possível reduzir ou aumentar a região alocada. A única forma seria alterar o código fonte e recompilar o programa.

Assim, o programador precisa saber de antemão o espaço necessário na memória para armazenar todos os dados do programa. Nem sempre isso acontece, sendo frequentemente preciso ajustar o armazenamento em **tempo de execução**. Além disso, a pilha pode ser limitada pelo sistema (frequentemente 4KB, conforme citado anteriormente), e caso o programa ultrapasse esse limite haverá "estouro de pilha" (*stack overflow*), causando muitas vezes a interrupção do programa e consequente perda dos dados.

1.3 Alocação dinâmica

Para obter espaço de armazenamento de forma livre é preciso alocar memória de forma dinâmica. Por meio desse método o programador pode alocar blocos de memória e associar esses blocos a

variáveis especiais conhecidas como ponteiros. Cada linguagem de programação com suporte a alocação dinâmica oferece diferentes ferramentas.

O padrão ANSI C especifica quatro funções para esse fim, todas definidas no arquivo de cabeçalho `stdlib.h`: `calloc()`, `malloc()`, `free()` e `realloc()`. O padrão C11 (Cx1) também inclui a função `aligned_alloc()`.

Um ponto importante dessa técnica é o uso de ponteiros. Uma variável ponteiro pode armazenar endereços de memória. O foco desse texto não é a explicação sobre ponteiros, portanto iremos assumir que o leitor está familiarizado com esse tipo de variável.

As funções `calloc()`, `malloc()`, `realloc()` e `aligned_alloc()` retornam um ponteiro caso a alocação dinâmica seja bem sucedida, ou nulo (`NULL`) caso não tenha conseguido alocar a memória. O tempo de vida de um objeto alocado vai desde a alocação até a liberação (desalocação explícida pela chamada da função `free()`. Veja exemplo a seguir.

```
01. int varglobal_1;
02.
03. int aloca_e_libera(int nbytes) {
04.     int sucesso = 0;
05.     char *pont;
06.     pont = calloc(nbytes, sizeof(char));
07.     if (pont != NULL)
08.         sucesso = 1;
09.     free(pont);
10.     return sucesso;
11. }
12.
13. int main(void) {
14.     int varlocal_1 = 3;
15.     aloca_e_libera(varlocal_1);
16.     return 0;
17. }
```

A função `calloc(a,b)` retorna um ponteiro para uma região da memória heap onde foram reservados `a` vezes `b` bytes, ou retorna nulo se houve problemas na alocação. A região alocada é ainda preenchida com zeros. A função `malloc()` por sua vez apenas aloca memória sem zerar os bits das regiões alocadas.

Após a execução da linha 08 (e antes da linha 09), a memória deverá estar no estado ilustrado abaixo, supondo que a alocação foi realizada com sucesso:

Pilha		
endereço	identificador	valor
0x000	`main()`	
0x004	`varlocal_1`	3
0x008	`aloca_e_libera()`	
0x012	`nbytes`	3
0x014	`sucesso`	1
0x018	`pont`	0x499
Heap		
0x499	*alocado*	'0'
0x500	*alocado*	'0'
0x501	*alocado*	'0'

1.4 Vazamento de memória

A alocação dinâmica requer um cuidado especial pois é preciso sempre liberar explicitamente a região de memória alocada. Do contrário, a região alocada permanece como "sendo utilizada" mesmo após o encerramento do programa. Essa região, caso não liberada dentro do programa, não poderá ser liberada posteriormente. Veja o exemplo abaixo:

```
int main(void) {
    int *A;
    A = (int *) calloc(10000, sizeof(int));
    return 0;
}
```

O programa declara um ponteiro para inteiro e faz a alocação de 100 vezes o tamanho de um inteiro. Considerando que `sizeof(int)` é 4 (quatro)[2], um bloco de 40000 bytes será alocado na memória heap. Como ele não é liberado de forma automática, pois foi alocado dinamicamente, esse bloco permanecerá em uso, ocorrendo o que é chamado de vazamento de memória (*memory leak*). Para que não haja memory leak, é preciso incluir `free(A)` que libera a memória:

```
int main(void) {
    int *A;
    A = (int *) calloc(10000, sizeof(int));
    free(A);
    return 0;
}
```

1.5 Acesso à memória *heap*

Quando alocamos um bloco de bytes na memória *heap*, o acesso é feito por meio de um ponteiro. No exemplo anterior o ponteiro para inteiro `A` armazena o endereço do primeiro byte alocado.

[2]pode variar com o sistema e a versão do compilador, podendo ser 8 em sistemas 64bits

Pilha		
endereço	identificador	valor
0x000	`main()`	
0x004	`A`	0x8a0
Heap		
0x8a0	*alocado*	42
0x8a4	*alocado*	0
0x8a8	*alocado*	0
0x8ac	*alocado*	0

Figura 1: Estado da memória na execução do Exemplo anterior

Quando desejamos acessar o primeiro bloco de 4 bytes (já que o ponteiro é do tipo `int *`), podemos **dereferenciar** o endereço utilizando o operador asterisco.

No exemplo anterior `A` retorna o endereço da memória alocada e `*A` retorna o valor naquele endereço. Veja o exemplo a seguir.

```
int main(void) {
    int *A;
    A = (int *) calloc(4, sizeof(int)); // aloca memoria
    *A = 42;  // atribui valor

    // imprime valores
    printf("%x\n", A); // %x indica valor em hexadecimal
    printf("%d\n", *A);
    free(A);
    return 0;
}
```

Note que ao dereferenciar o ponteiro, podemos tanto escrever (i.e. atribuir um valor) quanto ler (e.g. imprimir o valor contido).

Foram alocados 4 vezes o tamanho de um inteiro, assumindo 4 bytes por inteiro, a memória ficaria no estado ilustrado a seguir, na Figura 1.

Para acessar os outros blocos de 4 bytes, e assim armazenar outros valores inteiros, precisamos "percorrer" a região de memória alocada. Assim como `*A` corresponde ao primeiro inteiro alocado (referente ao endereço de memória `0x8a0`), podemos fazer `*(A+1)` para acessar o próximo bloco de memória, e dereferenciar o ponteiro no endereço `0x8a4`.

Mas desde quando $0 + 1 = 4$? A aritmética utilizada para operações sobre ponteiros é chamada aritmética de ponteiros e realiza as operações proporcionais ao tamanho do tipo do ponteiro. Nesse caso o ponteiro é `int*` e, como cada inteiro tem tamanho 4, `(A+1)` significa que queremos obter o próximo inteiro, ou seja, temos que avançar 4 bytes na memória. O exemplo abaixo ilustra esse conceito. Compile e execute o programa para ver a saída

```
int main(void) {
    int *A;
    A = (int *) calloc(4, sizeof(int));
    *A = 42;
    *(A+1) = 33;
    *(A+2) = 11;
    printf("%x\n", A);
    printf("%d\n", *A);
    printf("%d\n", *(A+1));
    free(A);
    return 0;
}
```

Para facilitar o acesso à memória heap, o operador `[]`, utilizado para acesso a arranjos, também pode ser usado para dereferenciar. Para isso apenas indique o deslocamento a ser feito nos blocos de memória. Por exemplo: `*A` equivale a `A[0]`, já que estamos dereferenciando a primeira posição, sem deslocamento, já `*(A+2)` equivale a `A[2]` pois primeiro deslocamos 2 blocos e depois dereferenciamos, obtendo o valor naquela posição.

2 Falha de segmentação: acesso indevido

Falha de segmentação ou *segmentation fault*[3] é um erro em tempo de execução (ou seja, não detectado durante a compilação). Quando você executa um programa e ele acusa um erro como esse, significa que seu programa tentou acessar (ou seja: ler ou escrever) uma região de memória para o qual não tem permissão de acesso. Ao tentar acessar, o sistema operacional detecta que seu programa não tem acesso àquela região de memória, e interrompe o programa.

As principais causas desse erro são:

1. Dereferenciar um ponteiro nulo (tanto escrita quanto leitura)

   Exemplo 1:

   ```
   int main(void) {
       double *x;
       x = (double *) calloc(2, sizeof(double));
       x = NULL;
       printf("%lf", x[0]); // leitura indevida
       return 0;
   }
   ```

2. Dereferenciar um ponteiro não inicializado (tanto escrita quanto leitura). Em geral quando declaramos uma variável não sabemos qual valor ela possui. Ao dereferenciar, o programa

[3] e também *segfault*, *bus error*, *access violation*, erro maldito, etc.

tenta acesso àquele valor, que pode ser 0 (zero) ou outro valor qualquer, frequentemente causando uma tentativa de acesso a um endereço não permitido.

Exemplo 2:

```
int main(void) {
    double *x;
    printf("%lf", *x); // leitura indevida
    return 0;
}
```

3. Dereferenciar um ponteiro que foi liberado (tanto escrita quanto leitura). Ao liberar um ponteiro, a região na memória antes alocada passa a ser de acesso "proibido". Como a variável mantém armazenado esse endereço, ao tentar o acesso o erro é gerado.

   Exemplo 3:

```
int main(void) {
    double *x;
    x = (double *) calloc(2, sizeof(double));
    free(x);
    x[1] = 0.99; // escrita indevida
    return 0;
}
```

Resumo dos exemplos:

```
char *p1 = NULL;       // inicializado como nulo: correto,
                       // ... mas *não* poderá ser dereferenciado

char *p2;              // não inicializado, *não* poderá ser dereferenciado

char *p3 = malloc(10); // incializado (alocado), pode ser dereferenciado

free(p3);              // liberado, *não* pode ser mais dereferenciado
```

2.1 Exemplos comuns de violação de memória

Tentativa de liberar região não alocada (dupla liberação, *double free*):

```
int **mat = (int **)malloc(sizeof(int *) * 3); // vetor de ponteiros
mat[0] = (int *)malloc(sizeof(int)*10); // aloca 10 ponteiros em mat[0]
for (i = 1; i < 3; i++) {
  free(mat[i]);
}
```

```
// apenas a posicao 0 de mat foi alocada as outras nao
// portanto esta liberando regiao nao alocada
// gerando violacao de memoria
```

Escrita em posição indevida (*invalid write*):

```
int B[5] = {5, 6, 7, 8, 9};
int N = 5;
int *A = malloc(N*sizeof(int));
int j = 0, i = 1; // 'j' inicializado em 0, 'i' inicializado em 1
while (j < N){
A[i] = B[j]; // 'j' e 'i' sao indices diferentes
j++;         // quando j = (N-1) i = (N-1)+1 e
i++;         // havera escrita indevida em A
}
```

3 Depurando falhas de segmentação e vazamento de memória

O programa `valgrind` é muito útil para depurar vazamento de memória e falha de segmentação. Primeiramente, é importante incluir na compilação do programa a opção `-g` para permitir a compilação com informações de depuração. Assim será possível saber as linhas de código onde possíveis erros ocorrem.

Exemplos de depuração

1. **Vazamento de memória**

```
int main(void) {
    double *x, *y;
    x = (double *) calloc(10, sizeof(double));
    y = (double *) calloc(10, sizeof(double));
    free(x);
    return 0;
}
```

Após compilar o programa, com o comando: `gcc -o programa1 programa1.c -g` e executar:

```
$ valgrind .\programa1
```

O valgrind irá monitorar a memória utilizada pelo seu programa. A saída será algo parecido com:

```
== Memcheck, a memory error detector
== Command: ./v1
==
== HEAP SUMMARY:
==     in use at exit: 80 bytes in 1 blocks
==   total heap usage: 2 allocs, 1 frees, 160 bytes allocated
==
== LEAK SUMMARY:
==    definitely lost: 80 bytes in 1 blocks
==    indirectly lost: 0 bytes in 0 blocks
==      possibly lost: 0 bytes in 0 blocks
==    still reachable: 0 bytes in 0 blocks
==         suppressed: 0 bytes in 0 blocks
== Rerun with --leak-check=full to see details of leaked memory
==
== For counts of detected and suppressed errors, rerun with: -v
== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
```

"HEAP SUMMARY" indica o uso de memória heap. Nesse caso foram detectados, em uso ao finalizar o programa 80 bytes em 1 bloco. Além disso o uso total da heap foi de 2 alocações e 1 liberação com total de 160 bytes alocados.

Em "LEAK SUMMARY" é possível ver os blocos de bytes perdidos. O programa detectou perda de 80 bytes em 1 bloco. Esses 80 bytes foram alocados com 10 espaços do tamanho `double` na variável `y`, e não liberado.

2. **Falha de segmentação**

```
01. int main(void) {
02.     int *A;
03.     A = (int *) calloc(3, sizeof(int));
04.     A[0] = 100;     A[1] = 101;     A[2] = 102;
05.     printf("%d %d %d\n", A[0], A[1], A[2]); // leitura indevida
06.     A = NULL;
07.     printf("%d", A[0]); // leitura indevida
08.     return 0;
09. }
```

Ao executar o programa acima, o resultado é:

```
$ ./programa2
100 101 102
Segmentation fault
```

Para tentar verificar o local da falha, executar:

```
$ valgrind ./programa2
```

A saída do programa, será parecida com:

```
== Memcheck, a memory error detector
== Command: ./v2
==
100 101 102
== Invalid read of size 4
==    at 0x4005BD: main (valgrind2.c:7)
==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==
== Process terminating with default action of signal 11 (SIGSEGV)
==  Access not within mapped region at address 0x0
==    at 0x4005BD: main (valgrind2.c:7)
==
== HEAP SUMMARY:
==     in use at exit: 12 bytes in 1 blocks
==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==
== LEAK SUMMARY:
==    definitely lost: 12 bytes in 1 blocks
==    indirectly lost: 0 bytes in 0 blocks
==      possibly lost: 0 bytes in 0 blocks
==    still reachable: 0 bytes in 0 blocks
==         suppressed: 0 bytes in 0 blocks
== Rerun with --leak-check=full to see details of leaked memory
==
== For counts of detected and suppressed errors, rerun with: -v
== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
Segmentation fault
```

A mensagem "Invalid read of size 4" indica leitura inválida de tamanho 4 na linha 7 do código. É feita uma leitura de um inteiro de 4 bytes, mas em um ponteiro nulo.

O programa ainda gera como saída: "Address 0x0 is not stack'd, malloc'd or (recently) free'd". Endereço `0x0` indica a leitura ou escrita em ponteiro nulo. Essa frase indica que esse local não está empilhado (na pilha da memória: *stack'd*), não foi alocado dinamicamente (*malloc'd*), ou então foi recentemente liberado (*free'd*). No exemplo nenhum dos três casos ocorreu, houve um erro do programador na linha 6, que atribuiu nulo à variável que continha o endereço da região de memória alocada dinamicamente.

Referências

[1] Schildt, H. *C completo e total*. Pearson, São Paulo, 3.ed, 1997.

[2] *Valgrind*: instrumentation framework for building dynamic analysis tools. `http://valgrind.org` v-3.8.1, 2012.

[3] Kernighan, B.; Ritchie, D. *C*: a linguagem de programação. Campus: São Paulo, 16.ed., 1986

[4] ISO/IEC. *Programming languages — C*: International Standard 9899:201x 3.ed., 2011