

Capítulo 15

Uso da memória

Os mecanismos de hardware vistos no capítulo 14 constroem uma abstração de memória virtual, desvinculada da organização da memória física, que é usada pelos processos e pelo próprio núcleo do sistema operacional. Este capítulo apresenta como essa memória é vista pelos processos e como eles a usam.

15.1 Espaço de endereçamento virtual de um processo

Na maioria dos sistemas atuais, o espaço de endereços virtuais de cada processo é organizado da seguinte forma: a parte inicial dos endereços virtuais é reservada para uso do processo, enquanto a parte final é reservada para o núcleo do sistema operacional. A Figura 15.1 ilustra a organização usual do espaço de endereçamento de cada processo em sistemas Windows e Linux de 32 e 64 bits [Love, 2010; Russinovich et al., 2008]. No sistema Linux 32 bits, por exemplo, os endereços iniciais (faixa `0x00000000-0xbfffffff`, com 3 GB) são usados pelo processo e os endereços finais (faixa `0xc0000000-0xffffffff`, com 1 GB) são usados pelo núcleo do sistema.

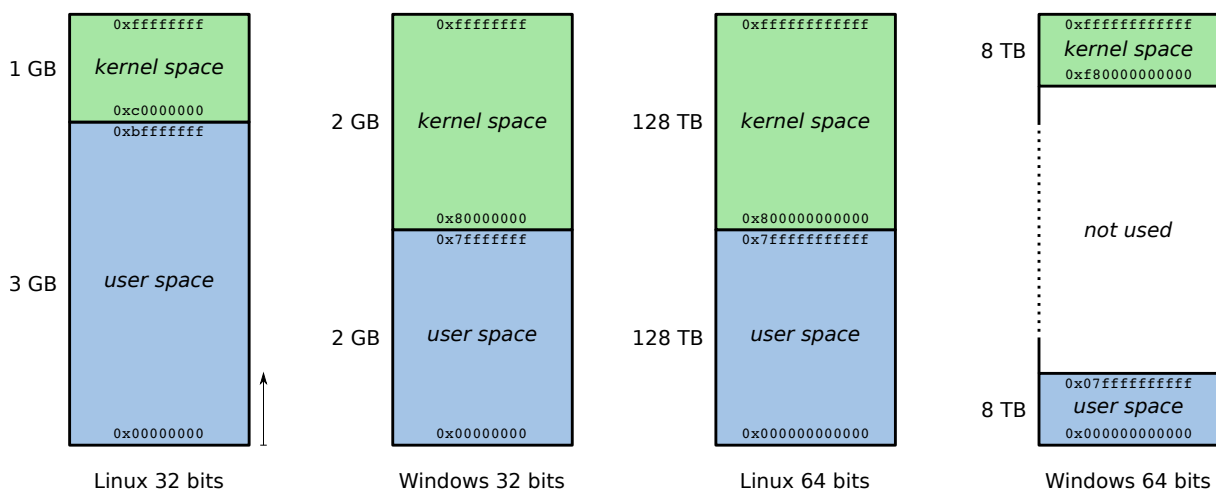


Figura 15.1: Organização do espaço de endereçamento em SOs atuais.

Observa-se que o núcleo faz parte do espaço de endereçamento de cada processo, mas não é acessível ao mesmo, por conta dos *flags* de controle da tabela de páginas vistos na seção 14.7.2. As páginas do processo são marcadas com o flag *user*, enquanto as

páginas do núcleo não o são. Com isso, o código no espaço do usuário não pode acessar as páginas do núcleo, mas o código do núcleo pode acessar as páginas do processo.

Com esse layout de endereços, as páginas do núcleo são mapeadas em todos os processos nos mesmos endereços virtuais, o que otimiza o uso do cache TLB, pois os endereços das páginas importantes do núcleo são sempre mantidos no TLB. Dessa forma, a execução de uma chamada de sistema não exige a mudança dos mapas de memória ativos nem o esvaziamento do cache TLB, o que contribui para um bom desempenho.

O bug de segurança *Meltdown*, descoberto em 2018 nos mecanismos de memória virtual dos processadores Intel e ARM [Lipp et al., 2018], levou à revisão da organização do espaço de endereçamento virtual dos processos nos sistemas operacionais mais usados. Esse bug permite a um código de usuário ignorar flags da tabela de páginas e ler partes da memória do núcleo, possivelmente expondo dados sensíveis lidos/escritos por outros processos, como senhas ou chaves de criptografia.

A solução encontrada para esse problema pelos principais sistemas operacionais consiste em retirar o núcleo do espaço de endereçamento dos processos. No Linux, a solução *Kernel Page-Table Isolation* (KPTI) mantém apenas um pequeno conjunto de páginas do núcleo no espaço de endereços do processo, para receber *syscalls* e interrupções; o restante da memória do núcleo usa uma tabela de páginas separada. Soluções similares foram adotadas pelos sistemas Microsoft Windows e Apple IOS.

15.2 A memória de um processo

Cada processo é implementado pelo sistema operacional como uma “cápsula” de memória isolada dos demais processos, ou seja, uma área de memória exclusiva que só o próprio processo e o núcleo do sistema podem acessar. A área de memória do processo contém as informações necessárias à sua execução: código binário, variáveis, bibliotecas, buffers, etc. Essa área é dividida em seções ou segmentos, que são intervalos de endereços que o processo pode acessar. A lista de seções de memória de cada processo é mantida pelo núcleo, no descritor do mesmo.

As principais seções de memória de um processo são:

TEXT: contém o código binário a ser executado pelo processo, gerado durante a compilação e a ligação com as bibliotecas e armazenado no arquivo executável. Esta seção se situa no início do espaço de endereçamento do processo e tem tamanho fixo, calculado durante a compilação.

DATA: esta seção contém as variáveis estáticas inicializadas, ou seja, variáveis que estão definidas do início ao fim da execução do processo e cujo valor inicial é declarado no código-fonte do programa. Esses valores iniciais são armazenados no arquivo do programa executável, sendo então carregados para esta seção de memória quando o processo inicia. Nesta seção são armazenadas tanto variáveis globais quanto variáveis locais estáticas (por exemplo, declaradas como `static` em C).

BSS: historicamente chamada de *Block Started by Symbol*, esta seção contém as variáveis estáticas não-inicializadas. Esta seção é separada da seção `DATA` porque as variáveis inicializadas precisam ter seu valor inicial armazenado no arquivo executável, o que não é necessário para as variáveis não-inicializadas. Com essa separação de variáveis, o arquivo executável fica menor.

HEAP: esta seção é usada para armazenar variáveis alocadas dinamicamente, usando operadores como `malloc()`, `new()` e similares. O final desta seção é definido por um ponteiro chamado *Program Break*, ou simplesmente *break*, que pode ser ajustado através de chamadas de sistema para aumentar ou diminuir o tamanho da mesma.

STACK: esta seção é usada para manter a pilha de execução do processo, ou seja, a estrutura responsável por gerenciar o fluxo de execução nas chamadas de função e também para armazenar os parâmetros, variáveis locais e o valor de retorno das funções. Geralmente a pilha cresce “para baixo”, ou seja, inicia em endereços maiores e cresce em direção aos endereços menores da memória. O tamanho total desta seção pode ser fixo ou variável, dependendo do sistema operacional.

Em programas com múltiplas *threads*, esta seção contém somente a pilha do programa principal. Como *threads* podem ser criadas e destruídas dinamicamente, a pilha de cada *thread* é mantida em uma seção de memória própria, alocada dinamicamente no *heap* ou em blocos de memória alocados na área livre para esse fim.

Cada uma dessas seções tem um conteúdo específico e por isso devem ser definidas permissões distintas para os acessos às mesmas. Por exemplo, a seção TEXT contém o código binário, que pode ser lido e executado, mas não deve ser modificado. As demais seções normalmente devem permitir acessos somente em leitura e escrita (sem execução), mas alguns programas que executam código interpretado ou com compilação dinâmica (como Java) podem gerar código dinamicamente e com isso necessitar de acessos em execução à pilha ou ao *heap*.

A Figura 15.2 apresenta a organização das seções de memória de um processo. Nela, observa-se que as duas seções de tamanho variável (*stack* e *heap*) estão dispostas em posições opostas e vizinhas à memória livre. Dessa forma, a memória livre disponível ao processo pode ser aproveitada da melhor forma possível, tanto pelo *heap* quanto pelo *stack*, ou por ambos.

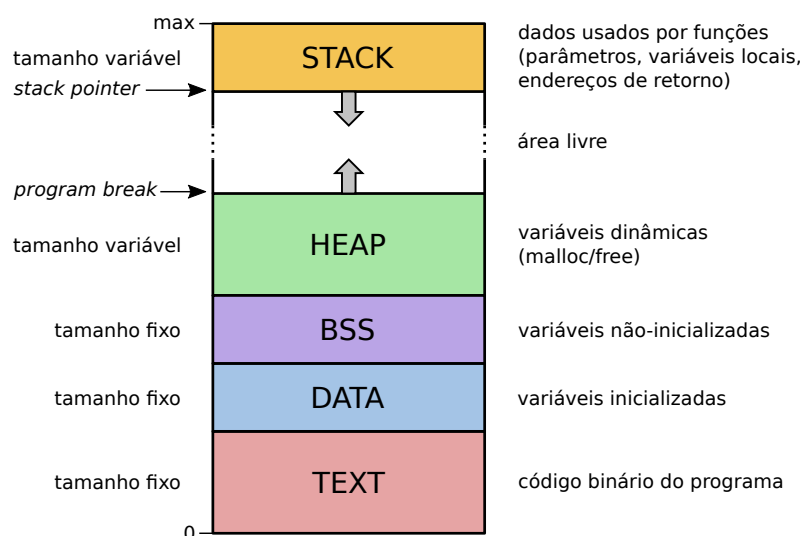


Figura 15.2: Organização da memória de um processo.

No sistema Linux, o comando `pmap` (*process map*) permite observar o mapa de memória de um processo, ou seja, a distribuição das seções de memória que o compõem. O exemplo a seguir ilustra o resultado (simplificado) desse comando aplicado a um processo que executa um simples `printf` ("Hello World\n") em linguagem C (27505 é o PID, identificador numérico do processo):

```

1 # pmap 27505
2
3 27505:  /usr/bin/hello
4 Address      Kbytes      Mode      Mapping
5 0000000000400000      808      r-x--    /usr/bin/hello      (TEXT)
6 000000000006c9000      12      rw---    /usr/bin/hello      (DATA)
7 000000000006cc000       8      rw---    [ anon ]             (BSS)
8 0000000000092e000      140     rw---    [ anon ]             (HEAP)
9 00007ffe6a5df000      132     rw---    [ stack ]            (STACK)

```

Na listagem acima, a coluna *Address* indica o endereço inicial da seção de memória, *Kbytes* indica seu tamanho, *Mode* indica suas permissões de acesso e *Mapping* indica o tipo de conteúdo da seção: `/usr/bin/hello` indica que o conteúdo da seção provém diretamente do arquivo indicado, `anon` indica uma seção *anônima* (que não está relacionada a nenhum arquivo em disco) e `stack` indica que a seção contém uma pilha.

Os endereços de cada seção de memória do processo podem variar de acordo com a estrutura do programa e o sistema operacional. Além disso, muitos sistemas operacionais implementam uma técnica chamada *Address Space Layout Randomization* (ASLR), que consiste em variar aleatoriamente a posição dessas áreas a cada vez que um processo é iniciado. Com isso, as posições das variáveis e funções de um programa na memória variam a cada execução, o que melhora a segurança do programa frente a ataques de estouro de *buffer* ou de pilha.

15.3 Alocação de variáveis

Um programa em execução armazena suas informações em variáveis, que são basicamente espaços de memória reservados e nomeados. Por exemplo, uma declaração `int soma` em linguagem C indica uma área de memória de 4 bytes que pode armazenar um número inteiro e cujo nome é `soma`. Em linguagens como C e C++, variáveis podem ser alocadas na memória usando três abordagens usuais: a alocação *estática*, a alocação *automática* e a alocação *dinâmica*. Estas formas de alocação serão descritas a seguir.

15.3.1 Alocação estática

Na alocação estática, o espaço necessário para a variável é definido durante a compilação do programa. O espaço correspondente em memória RAM é reservado no início da execução do processo e mantido até o encerramento deste. Variáveis com alocação estática são alocadas na seção de memória `DATA`, se forem inicializadas no código-fonte, ou na seção `BSS`, caso contrário.

Na linguagem C, esta forma de alocação é usada para variáveis globais ou variáveis locais estáticas¹, como a variável `soma` no exemplo a seguir:

¹Na linguagem C, variáveis locais estáticas são aquelas definidas como `static` dentro de uma função, que preservam seu valor entre duas invocações da mesma função.

```
1 #include <stdio.h>
2
3 int soma = 0 ;
4
5 int main ()
6 {
7     int i ;
8
9     for (i=0; i<1000; i++)
10        soma += i ;
11    printf ("Soma de inteiros até 1000: %d\n", soma) ;
12
13    return (0) ;
14 }
```

15.3.2 Alocação automática

Por default, as variáveis definidas dentro de uma função (variáveis locais e parâmetros) são alocadas de forma automática na pilha de execução do programa (seção STACK). O espaço usado para armazenar essas variáveis é alocado quando a função é invocada e liberado quando a função termina, de forma transparente para o programador. Isso é o que ocorre por exemplo com a variável *i* no código anterior.

Se uma função for chamada recursivamente, as variáveis locais e parâmetros serão novamente alocados na pilha, em áreas distintas para cada nível de recursão. Isso permite preservar os valores das mesmas em cada um dos níveis. O exemplo a seguir permite observar a existência de múltiplas instâncias de variáveis locais em chamadas recursivas:

```
1 #include <stdio.h>
2
3 long int fatorial (int n)
4 {
5     long int parcial ;
6
7     printf ("inicio: n = %d\n", n) ;
8     if (n < 2)
9         parcial = 1 ;
10    else
11        parcial = n * fatorial (n - 1) ;
12    printf ("final : n = %d, parcial = %ld\n", n, parcial) ;
13    return (parcial) ;
14 }
15
16 int main ()
17 {
18    printf ("Fatorial (4) = %ld\n", fatorial (4)) ;
19    return 0 ;
20 }
```

A execução do código acima gera o resultado apresentado na listagem a seguir. Pode-se observar claramente que, durante as chamadas recursivas à função *fatorial (n)*, vários valores distintos para as variáveis *n* e *parcial* são armazenados na memória:

```
1 inicio: n = 4
2 inicio: n = 3
3 inicio: n = 2
4 inicio: n = 1
5 final : n = 1, parcial = 1
6 final : n = 2, parcial = 2
7 final : n = 3, parcial = 6
8 final : n = 4, parcial = 24
9 Fatorial (4) = 24
```

15.3.3 Alocação dinâmica

Na alocação dinâmica de memória, o processo requisita explicitamente blocos de memória para armazenar dados, os utiliza e depois os libera, quando não forem mais necessários (ou quando o programa encerrar). Esses blocos de memória são alocados na seção HEAP, que pode aumentar de tamanho para acomodar mais alocações quando necessário.

A requisição de blocos de memória dinâmicos é feita através de funções específicas, que retornam uma referência (ou ponteiro) para o bloco de memória alocado. Um exemplo de alocação e liberação de memória dinâmica na linguagem C pode ser visto no trecho de código a seguir:

```
1 char * prt ; // ponteiro para caracteres
2
3 ptr = malloc (4096) ; // solicita um bloco de 4.096 bytes;
4 // ptr aponta para o início do bloco
5
6 if (!ptr) // se ptr for nulo, ocorreu um erro
7 abort () ; // e a área não foi alocada
8
9 ... // usa ptr para acessar o bloco alocado
10
11 free (ptr) ; // libera o bloco alocado na linha 3
```

Alocações dinâmicas são muito usadas para armazenar objetos em linguagens orientadas a objetos. O trecho de código a seguir ilustra a criação dinâmica de objetos em Java:

```
1 Rectangle rect1 = new Rectangle (10, 30) ;
2 Rectangle rect2 = new Rectangle (3, 2) ;
3 Triangle tr1 = new Triangle (3, 4, 5) ;
```

A memória alocada dinamicamente por um processo é automaticamente liberada quando sua execução encerra. Contudo, pode ser necessário liberar blocos de memória dinâmicos sem uso durante uma execução, sobretudo se ela for longa, como um servidor Web ou um gerenciador de ambiente *desktop*. Programas que só alocam memória e não a liberam podem acabar consumindo toda a memória disponível no sistema, impedindo os demais programas de funcionar.

A liberação dos blocos de memória dinâmicos durante a execução pode ser manual ou automática, dependendo da linguagem de programação usada. Em linguagens mais simples, como C e C++, a liberação dos blocos de memória alocados deve ser

feita pelo programador, usando funções como `free()` ou `delete()`. Linguagens mais sofisticadas, como Java, Python e C#, possuem mecanismos de “coleta de lixo” (*garbage collection*) que automaticamente varrem os blocos de memória alocados e liberam os que não estiverem mais em uso, usando diversas técnicas [Wilson et al., 1995].

15.4 Atribuição de endereços

Ao escrever um programa usando uma linguagem de programação, como C, C++ ou Java, o programador usa nomes para referenciar entidades abstratas como variáveis, funções, parâmetros e valores de retorno. Com esses nomes, não há necessidade do programador definir ou manipular endereços de memória explicitamente. O trecho de código em C a seguir (`soma.c`) ilustra esse conceito; nele, são usados símbolos para referenciar posições na memória contendo dados (`i` e `soma`) ou trechos de código (`main`, `printf` e `exit`):

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int soma = 0 ;
5
6 int main ()
7 {
8     int i ;
9
10    for (i=0; i< 10; i++)
11    {
12        soma += i ;
13        printf ("i vale %d e soma vale %d\n", i, soma) ;
14    }
15    exit(0) ;
16 }
```

Todavia, o processador do computador precisa acessar endereços de memória para buscar as instruções a executar e seus operandos e para escrever os resultados do processamento dessas instruções. Por isso, quando o programa `soma.c` for compilado, ligado a bibliotecas, carregado na memória e executado pelo processador, cada referência a uma variável, procedimento ou função no programa terá de ser transformada em um ou mais endereços específicos na área de memória do processo.

A listagem a seguir apresenta o código *Assembly* correspondente à compilação do programa em linguagem C `soma.c`. Nele, pode-se observar que não há mais referências a nomes simbólicos, apenas a endereços:

```

1 0000000000000000 <main>:
2   0: 55                push  %rbp
3   1: 48 89 e5           mov   %rsp,%rbp
4   4: 48 83 ec 10        sub   $0x10,%rsp
5   8: c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%rbp)
6   f: eb 2f             jmp   40 <main+0x40>
7  11: 8b 15 00 00 00 00  mov   0x0(%rip),%edx
8  17: 8b 45 fc           mov   -0x4(%rbp),%eax
9  1a: 01 d0             add   %edx,%eax
10 1c: 89 05 00 00 00 00  mov   %eax,0x0(%rip)
11 22: 8b 15 00 00 00 00  mov   0x0(%rip),%edx
12 28: 8b 45 fc           mov   -0x4(%rbp),%eax
13 2b: 89 c6             mov   %eax,%esi
14 2d: bf 00 00 00 00    mov   $0x0,%edi
15 32: b8 00 00 00 00    mov   $0x0,%eax
16 37: e8 00 00 00 00    callq 3c <main+0x3c>
17 3c: 83 45 fc 01       addl  $0x1,-0x4(%rbp)
18 40: 83 7d fc 04       cmpl  $0x4,-0x4(%rbp)
19 44: 7e cb             jle   11 <main+0x11>
20 46: bf 00 00 00 00    mov   $0x0,%edi
21 4b: e8 00 00 00 00    callq 50 <main+0x50>

```

Dessa forma, os nomes simbólicos das variáveis e blocos de código usados por um programa devem ser traduzidos em endereços de memória em algum momento entre a escrita do código pelo programador e sua execução pelo processador. A atribuição de endereços aos nomes simbólicos pode ocorrer em diversos momentos da vida do programa:

Na edição: o programador escolhe o endereço de cada uma das variáveis e do código do programa na memória. Esta abordagem normalmente só é usada na programação de sistemas embarcados simples, programados diretamente em *Assembly*.

Na compilação: ao traduzir o código-fonte, o compilador escolhe as posições das variáveis na memória. Para isso, todos os códigos-fontes necessários ao programa devem ser conhecidos no momento da compilação, para evitar conflitos de endereços entre variáveis em diferentes arquivos ou bibliotecas. Essa restrição impede o uso de bibliotecas precompiladas. Esta abordagem era usada em programas executáveis com extensão *.COM* do MS-DOS e Windows.

Na ligação: na fase de compilação, o compilador traduz o código fonte em código binário, mas não define os endereços das variáveis e funções, gerando como saída um *arquivo objeto (object file)*², que contém o código binário e uma *tabela de símbolos* descrevendo as variáveis e funções usadas, seus tipos, onde estão definidas e onde são usadas. A seguir, o ligador (*linker*) pega os arquivos objetos com suas tabelas de símbolos, define os endereços de memória dos símbolos e gera o programa executável [Levine, 2000].

Na carga: também é possível definir os endereços de variáveis e de funções durante a carga do código em memória para o lançamento de um novo processo. Nesse

²Os arquivos com extensão *.o* em UNIX ou *.obj* em Windows são exemplos de arquivos-objeto obtidos da compilação de arquivos em C ou outra linguagem compilada.

caso, um *carregador (loader)* é responsável por carregar o código do processo na memória e definir os endereços de memória que devem ser utilizados. O carregador pode ser parte do núcleo do sistema operacional ou uma biblioteca ligada ao executável, ou ambos. Esse mecanismo normalmente é usado na carga de bibliotecas dinâmicas (DLL - *Dynamic Linking Libraries*).

Na execução: os endereços emitidos pelo processador durante a execução do processo são analisados e convertidos nos endereços efetivos a serem acessados na memória real. Por exigir a análise e a conversão de cada endereço gerado pelo processador, este método só é viável com o auxílio do hardware.

A maioria dos sistemas operacionais atuais usa uma combinação de técnicas, envolvendo a tradução durante a ligação (para o código principal do programa e a construção de bibliotecas), durante a carga (para bibliotecas dinâmicas) e durante a execução (para todo o código). A tradução direta de endereço durante a edição ou compilação só é usada na programação de sistemas mais simples, como microcontroladores e sistemas embarcados. A Figura 15.3 ilustra os momentos de tradução de endereços acima descritos.

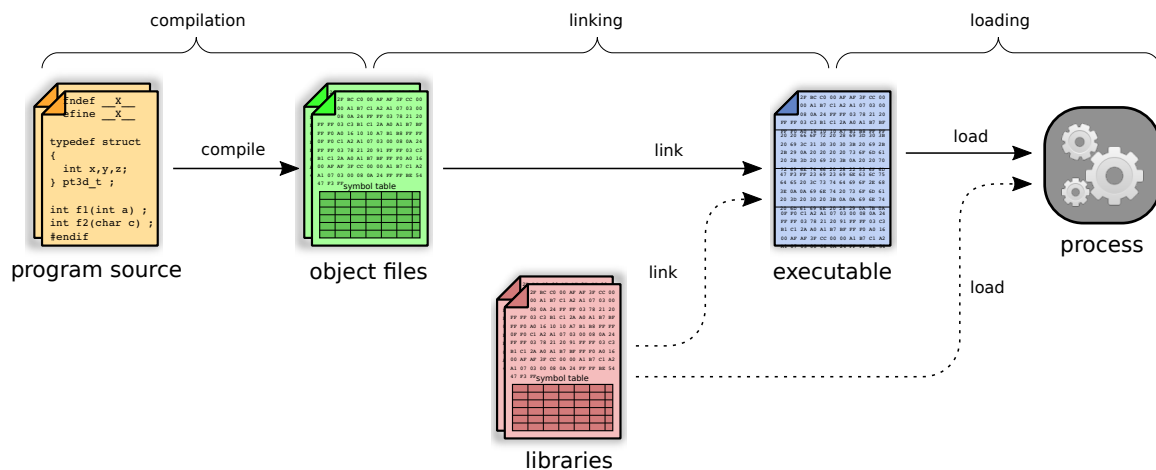


Figura 15.3: Momentos da tradução de endereços.

Exercícios

1. Como é organizado o espaço de memória de um processo?
2. Sobre as afirmações a seguir, relativas ao uso da memória RAM pelos processos, indique quais são incorretas, justificando sua resposta:
 - (a) A área de memória TEXT contém o código-fonte a ser compilado e executado pelo processo.
 - (b) A área de memória DATA é usada para armazenar todas as variáveis e constantes usadas pelo processo.
 - (c) A área de memória HEAP é usada para as alocações dinâmicas de memória, sendo usada através de funções como malloc e free.

- (d) A área de memória STACK contém as pilhas do programa principal e das demais *threads* do processo.
3. Explique as principais formas de alocação de variáveis.
 4. Explique em que consiste a resolução de endereços nos seguintes momentos: *codificação, compilação, ligação, carga e execução*.

Referências

- J. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.
- M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, 2018.
- R. Love. *Linux Kernel Development, Third Edition*. Addison-Wesley, 2010.
- M. Russinovich, D. Solomon, and A. Ionescu. *Microsoft Windows Internals, Fifth Edition*. Microsoft Press, 2008.
- P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.