Capítulo 16

Alocação de memória

As aplicações, utilitários e o próprio sistema operacional precisam de memória para executar. Como ocorre com os demais recursos de hardware, a memória disponível no sistema deve ser gerenciada pelo SO, para evitar conflitos entre aplicações e garantir justiça no seu uso. Fazendo uso dos mecanismos de hardware de memória apresentados no capítulo 14, o sistema operacional aloca e libera áreas de memória para os processos (ou para o próprio núcleo), conforme a necessidade. Este capítulo apresenta os principais conceitos relacionados à alocação de memória.

16.1 Alocadores de memória

Alocar memória significa reservar áreas de memória RAM que podem ser usadas por um processo, por um descritor de *socket* ou de arquivo no núcleo, por um cache de blocos de disco, etc. Ao final de seu uso, cada área de memória alocada é liberada pela entidade que a solicitou e colocada à disposição do sistema para novas alocações.

O mecanismo responsável pela alocação e liberação de áreas de memória é chamado um *alocador de memória*. Em linhas gerais, o alocador reserva ou libera partes da memória RAM, de acordo com o fluxo de solicitações que recebe (de processos ou do núcleo do sistema operacional). Para tal, o alocador deve manter um registro contínuo de quais áreas estão sendo usadas e quais estão livres. Para ser eficiente, ele deve realizar a alocações rapidamente e minimizar o desperdício de memória [Wilson et al., 1995].

Alocadores de memória podem existir em diversos contextos:

Alocador de memória física : organiza a memória física do computador, definindo as áreas que podem ser alocadas pelo núcleo e processos e as áreas reservadas (BIOS, DMA, etc.). Gerencia o uso das áreas alocáveis, atendendo requisições de alocação/liberação do núcleo e de processos.

Alocador de espaço de núcleo: o núcleo do SO continuamente cria e destrói muitas estruturas de dados relativamente pequenas, como descritores de arquivos abertos, de processos, *sockets* de rede, *pipes*, etc. O alocador de núcleo obtém áreas de memória do alocador físico e as utiliza para alocar essas estruturas para o núcleo.

Alocador de espaço de usuário: um processo pode solicitar blocos de memória para armazenar estruturas de dados dinâmicas, através de operações como `malloc` e `free`. O alocador de memória do processo geralmente é implementado por

bibliotecas providas pelo sistema operacional, como a `LibC`. Essas bibliotecas interagem com o núcleo para solicitar o redimensionamento da seção `HEAP` do processo quando necessário (Seção 15.2).

A Figura 16.1 apresenta uma visão geral dos mecanismos de alocação de memória em um sistema operacional típico. Na figura pode-se observar os três alocadores: de memória física, de núcleo e do espaço de usuário. O esquema apresentado nessa figura procura ser genérico, pois as implementações variam muito entre sistemas operacionais distintos.

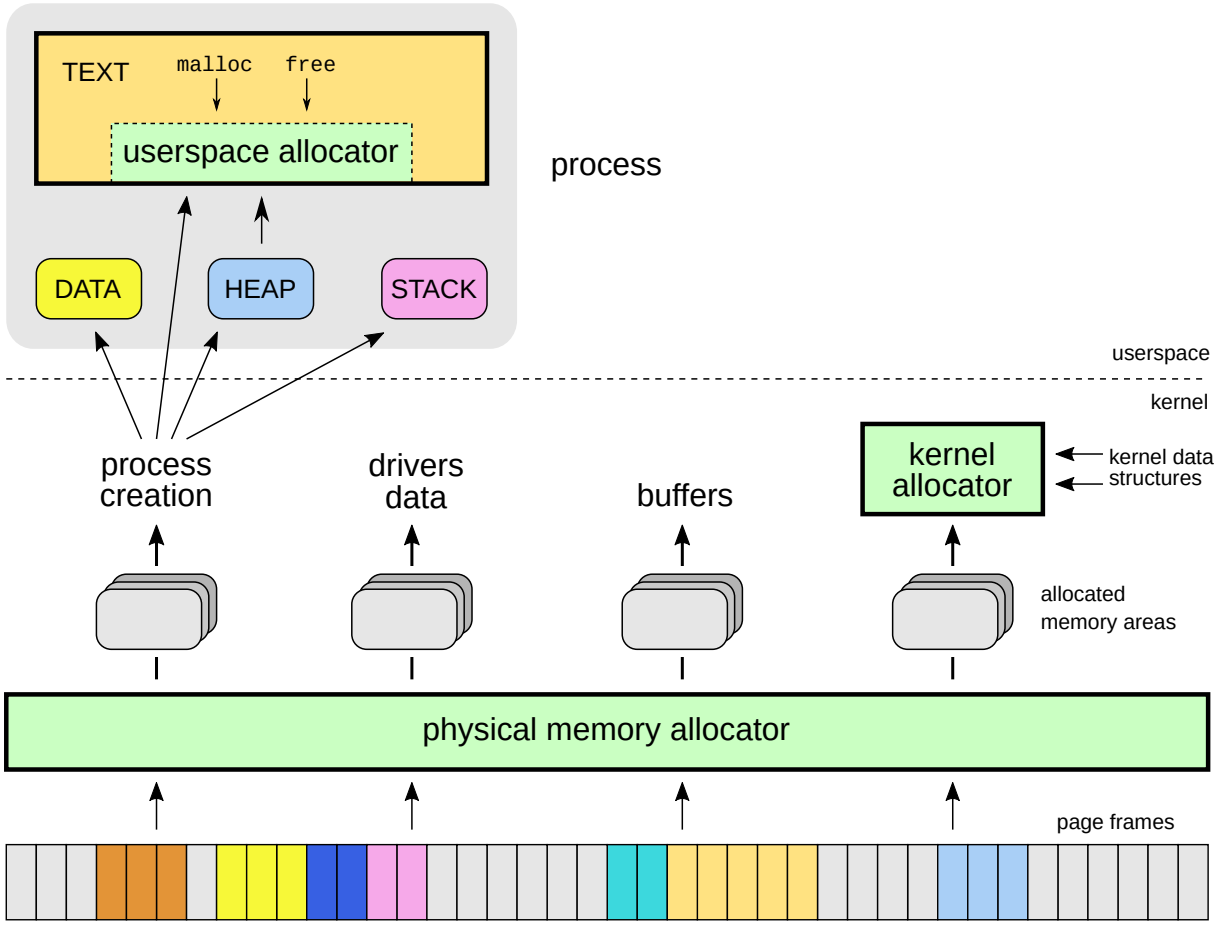

Figura 16.1: Mecanismos de alocação de memória.

16.2 Alocação básica

O problema básico de alocação consiste em manter uma grande área de memória RAM e atender um fluxo de requisições de alocação e liberação de partes dessa área para o sistema operacional ou para as aplicações. Essas requisições ocorrem o tempo todo, em função das tarefas em execução no sistema, e devem ser atendidas rapidamente.

Vejamos um exemplo simples: considere um sistema hipotético com 1 GB de memória RAM livre em uma área única[1]. O alocador de memória recebe a seguinte sequência de requisições:

[1]Sistemas reais, como os computadores PC, podem ter várias áreas de memória livre distintas e não-contíguas.

1. Aloca 200 MB (a_1)
2. Aloca 100 MB (a_2)
3. Aloca 100 MB (a_3)
4. Libera a_1
5. Aloca 300 MB (a_4)
6. Libera a_3

O alocador atende essas requisições em sequência, reservando e liberando áreas de memória conforme necessário. A Figura 16.2 apresenta uma evolução possível das áreas de memória com as ações do alocador.

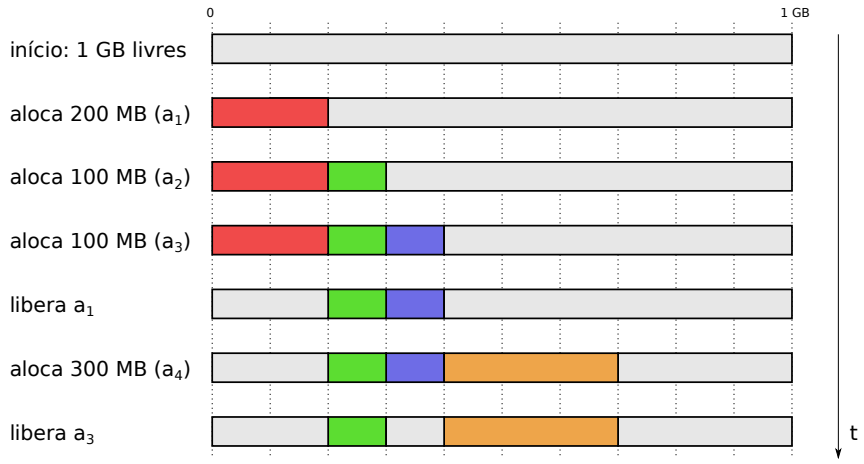

Figura 16.2: Sequência de alocações e liberações de memória.

Na Figura 16.2 pode-se observar que, como efeito das alocações e liberações, a área de memória inicialmente vazia se transforma em uma sequência de áreas ocupadas (alocadas) e áreas livres, que evolui a cada nova requisição. Essas informações são geralmente mantidas em uma ou mais listas duplamente encadeadas (ou árvores) de áreas de memória.

16.3 Fragmentação

Ao longo da vida de um sistema, áreas de memória são alocadas e liberadas continuamente. Com isso, podem surgir áreas livres ("buracos" na memória) entre as áreas alocadas. Por exemplo, na Figura 16.2 pode-se observar que o sistema ainda tem 600 MB de memória livre após a sequência de operações, mas somente requisições de alocação de até 300 MB pode ser aceitas, pois esse é o tamanho da maior área livre contínua disponível. Esse fenômeno se chama *fragmentação externa*, pois fragmenta a memória livre, fora das áreas alocadas.

A fragmentação externa é muito prejudicial, porque limita a capacidade de alocação de memória do sistema. Além disso, quanto mais fragmentada estiver a memória livre, maior o esforço necessário para gerenciá-la, pois mais longas serão as listas encadeadas de área de memória livres. Pode-se enfrentar o problema da

fragmentação externa de duas formas: *minimizando* sua ocorrência, através de estratégias de alocação, *desfragmentando* periodicamente a memória do sistema, ou permitindo a fragmentação interna.

16.3.1 Estratégias de alocação

Para minimizar a ocorrência de fragmentação externa, cada pedido de alocação pode ser analisado para encontrar a área de memória livre que melhor o atenda. Essa análise pode ser feita usando um dos seguintes critérios:

***First-fit* (primeiro encaixe):** consiste em escolher a primeira área livre que satisfaça o pedido de alocação; tem como vantagem a rapidez, sobretudo se a lista de áreas livres for muito longa. É a estratégia adotada na Figura 16.2.

***Best-fit* (melhor encaixe):** consiste em escolher a menor área possível que possa receber a alocação, minimizando o desperdício de memória. Contudo, algumas áreas livres podem ficar pequenas demais e com isso se tornarem inúteis.

***Worst-fit* (pior encaixe):** consiste em escolher sempre a maior área livre possível, de forma que a "sobra" seja grande o suficiente para ser usada em outras alocações.

***Next-fit* (próximo encaixe):** variante da estratégia *first-fit* que consiste em percorrer a lista de áreas a partir da última área alocada ou liberada, para que o uso das áreas livres seja distribuído de forma mais homogênea no espaço de memória.

Diversas pesquisas [Johnstone and Wilson, 1999] demonstraram que as abordagens mais eficientes são a *best-fit* e a *first-fit*, sendo esta última bem mais rápida. A Figura 16.3 ilustra essas estratégias na alocação de um bloco de 80 MB em uma memória de 1 GB parcialmente alocada.

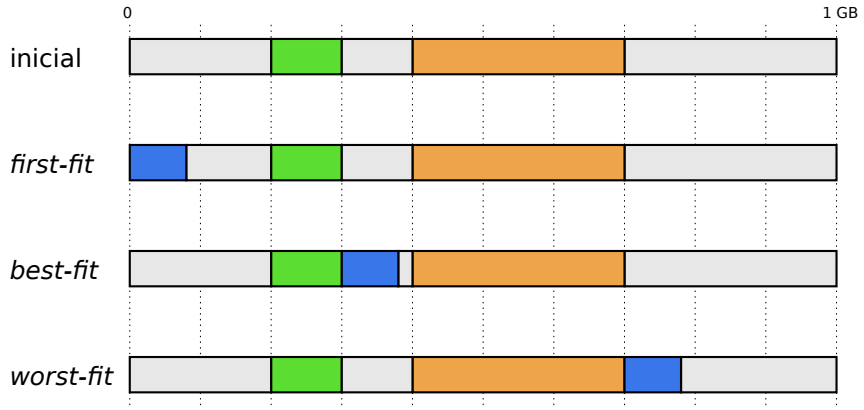

Figura 16.3: Estratégias para minimizar a fragmentação externa.

16.3.2 Desfragmentação

Outra forma de tratar a fragmentação externa consiste em *desfragmentar* a memória periodicamente. Para tal, as áreas de memória usadas pelos processos devem ser movidas na memória de forma a concatenar as áreas livres e assim diminuir a fragmentação. Ao mover um processo na memória, suas informações de endereçamento virtual

(registrador base/limite, tabela de segmentos ou de páginas) devem ser devidamente ajustadas para refletir a nova posição do processo na memória RAM. Por essa razão, a desfragmentação só pode ser aplicada a áreas de memória físicas, pois as mudanças de endereço das áreas de memória serão ocultadas pelo hardware. Ela não pode ser aplicada, por exemplo, para a gestão da seção *heap* de um processo.

Como as áreas de memória não podem ser acessadas durante a desfragmentação, é importante que esse procedimento seja executado rapidamente e com pouca frequência, para não interferir nas atividades normais do sistema. As possibilidades de movimentação de áreas podem ser muitas, portanto a desfragmentação deve ser tratada como um problema de otimização combinatória. A Figura 16.4 ilustra três possibilidades de desfragmentação de uma determinada situação de memória; as três alternativas produzem o mesmo resultado (uma área livre contínua com 450 MB), mas têm custos distintos.

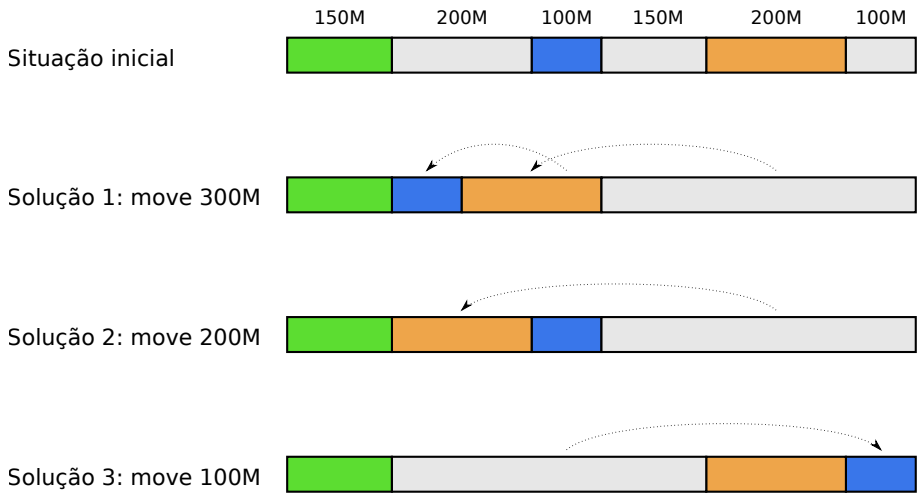

Figura 16.4: Possibilidades de desfragmentação.

16.3.3 Fragmentação interna

Uma alternativa para minimizar o impacto da fragmentação externa consiste em arredondar algumas requisições de alocação, para evitar sobras muito pequenas. Por exemplo, na alocação com *best-fit* da Figura 16.3, a área alocada poderia ser arredondada de 80 MB para 100 MB, evitando a sobra da área de 20 MB (Figura 16.5). Dessa forma, evita-se a geração de um fragmento de memória livre, mas a memória adicional alocada provavelmente não será usada por quem a requisitou. Esse desperdício de memória dentro da área alocada é denominado *fragmentação interna* (ao contrário da fragmentação externa, que ocorre nas áreas livres).

A fragmentação interna afeta todas as formas de organização de memória; as partições e segmentos sofrem menos com esse problema, pois o nível de arredondamento das áreas de memória pode ser decidido caso a caso. No caso da memória paginada, essa decisão não é possível, pois as alocações são sempre feitas em múltiplos inteiros de páginas. Assim, em um sistema com páginas de 4 KBytes (4.096 bytes), um processo que solicite 550.000 bytes (134,284 páginas) receberá 552.960 bytes (135 páginas), ou seja, 2.960 bytes a mais que o solicitado.

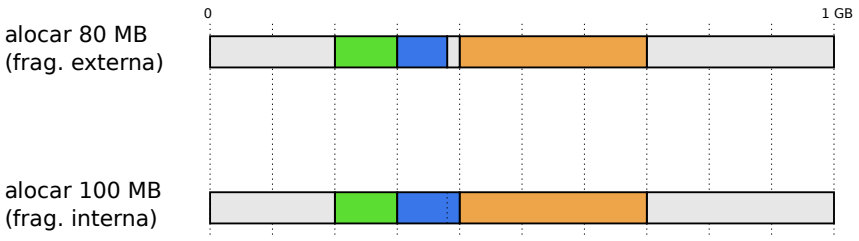

Figura 16.5: Fragmentação interna.

Em média, para cada processo haverá uma perda de 1/2 página de memória por fragmentação interna. Uma forma de minimizar a perda por fragmentação interna seria usar páginas de menor tamanho (2K, 1K, 512 bytes ou ainda menos). Todavia, essa abordagem implica em ter mais páginas por processo, o que geraria tabelas de páginas maiores e com maior custo de gerência.

16.4 O alocador Buddy

Existem estratégias de alocação mais sofisticadas e eficientes que as apresentadas na seção 16.3.1. Um algoritmo de alocação muito conhecido é o chamado *Buddy Allocator*, ou alocador por pares [Wilson et al., 1995], explicado a seguir. Em sua versão mais simples, a estratégia *Buddy* sempre aloca blocos de memória de tamanho 2^n, com n inteiro e ajustável. Por exemplo, para uma requisição de 85 KBytes será alocado um bloco de memória com 128 KBytes (2^7 KBytes ou 2^{17} bytes), e assim por diante. O uso de blocos de tamanho 2^n reduz a fragmentação externa, mas pode gerar muita fragmentação interna.

O valor de n pode variar entre os limites n_{min} e n_{max}, ou seja, $n_{min} < n < n_{max}$. n_{min} define o menor bloco que pode ser alocado, para evitar custo computacional e desperdício de espaço com a alocação de blocos muito pequenos. Valores entre 1 KByte e 64 KBytes são usuais. Por sua vez, n_{max} define o tamanho do maior bloco alocável, sendo limitado pela quantidade de memória RAM disponível. Em um sistema com 2.000 MBytes de memória RAM livre, o maior bloco alocável teria 1.024 MBytes (ou 2^{20} Bytes). Os 976 MBytes restantes também podem ser alocados, mas em blocos menores.

O funcionamento do alocador Buddy binário (blocos de 2^n bytes) é simples:

- Ao receber uma requisição de alocação de memória de tamanho 40 KBytes (por exemplo), o alocador procura um bloco livre com 64 KBytes (pois 64 KBytes é o menor bloco com tamanho 2^n que pode conter 40 KBytes). Caso não encontre um bloco com 64 KBytes, procura um bloco livre com 128 KBytes, o divide em dois blocos de 64 KBytes (os *buddies*) e usa um deles para a alocação. Caso não encontre um bloco livre com 128 KBytes, procura um bloco com 256 KBytes para dividir em dois, e assim sucessivamente.

- Ao liberar uma área de memória alocada, o alocador verifica se o par (*buddy*) do bloco liberado também está livre; se estiver, funde os dois em um bloco maior, analisa o novo bloco em relação ao seu par e continua as fusões de blocos, até encontrar um par ocupado ou chegar ao tamanho máximo de bloco permitido. A fusão entre dois blocos vizinhos também é chamada de *coalescência*.

A Figura 16.6 apresenta um exemplo de funcionamento do alocador *Buddy*. Nesse exemplo didático, o tamanho da memória é de 1 MBytes (1.024 KBytes) e o menor bloco alocável é de 64 KBytes. O alocador de memória recebe a seguinte sequência de requisições: *aloca 200 KB* (a_1), *aloca 100 KB* (a_2), *aloca 150 KB* (a_3), *libera* a_1 e *libera* a_2.

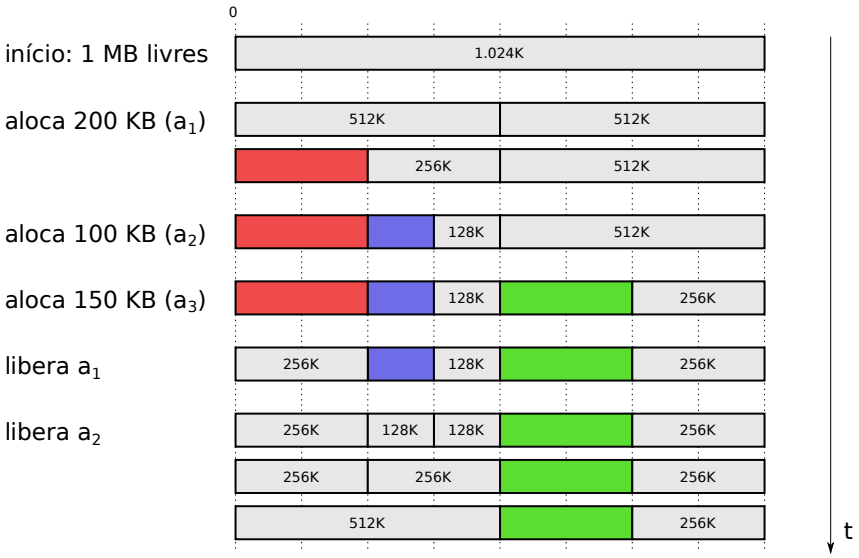

Figura 16.6: O alocador Buddy binário.

Além da estratégia binária apresentada aqui, existem variantes de alocador *Buddy* que usam outras formas de dividir blocos, como a estratégia Buddy com Fibonacci, na qual os tamanhos dos blocos seguem a sequência de Fibonacci[2] e a estratégia Buddy com pesos, na qual os blocos têm tamanhos 2^n mas são divididos em sub-blocos de tamanho distintos (por exemplo, um bloco de 64 KB seria dividido em dois blocos, de 48 KB e 16KB).

O alocador Buddy é usado em vários sistemas. Por exemplo, no núcleo Linux ele é usado para a alocação de memória física (*page frames*), entregando áreas de memória RAM para a criação de processos, para o alocador de objetos do núcleo e para outros subsistemas. O arquivo `/proc/buddyinfo` permite consultar informações das alocações existentes:

```
# cat /proc/buddyinfo

Chunk size           4K     8K    16K   32K 64K 128K 256K 512K 1M 2M   4M
Node 0, zone    DMA   1      0      1     2   3    2    0    0  1  1    3
Node 0, zone  DMA32  12      8      9     7   6   10    6    6  4  3  551
Node 0, zone Normal 237  15443  11954  3798 984  444  289  133 70 37  126
```

16.5 O alocador Slab

O alocador Slab foi inicialmente proposto para o núcleo do sistema operacional SunOS 5.4 [Bonwick, 1994]. Ele é especializado na alocação de "objetos de núcleo", ou seja, as pequenas estruturas de dados que são usadas para representar descritores de processos, de arquivos abertos, *sockets* de rede, *pipes*, etc. Esses objetos de núcleo

[2]Na sequência de Fibonacci, cada termo corresponde à soma dos dois termos anteriores: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc. Em termos matemáticos, $F_0 = 0, F_1 = 1, F_{n>1} = F_{n-1} + F_{n-2}$.

são continuamente criados e destruídos durante a operação do sistema, são pequenos (dezenas ou centenas de bytes) e têm tamanhos relativamente padronizados.

Alocar e liberar memória para objetos de núcleo usando um alocador básico ou Buddy implicaria em um custo computacional elevado, além de desperdício de memória em fragmentação. Por isso é necessário um alocador especializado, capaz de fornecer memória para esses objetos rapidamente e com baixo custo. Outra questão importante é a inicialização dos objetos de núcleo: pode-se economizar custos de inicialização se os objetos liberados forem mantidos na memória e reutilizados, ao invés daquela área de memória ser liberada.

O alocador Slab usa uma estratégia baseada no *caching* de objetos. É definido um *cache* para cada tipo de objeto usado pelo núcleo: descritor de processo, de arquivo, de *socket*, etc[3]. Cada cache é então dividido em *slabs* (placas ou lajes) que contêm objetos daquele tipo, portanto todos com o mesmo tamanho. Um slab pode estar **cheio**, quando todos os seus objetos estão em uso, **vazio**, quando todos os seus objetos estão livres, ou **parcial**. A Figura 16.7 ilustra a estrutura dos caches de objetos.

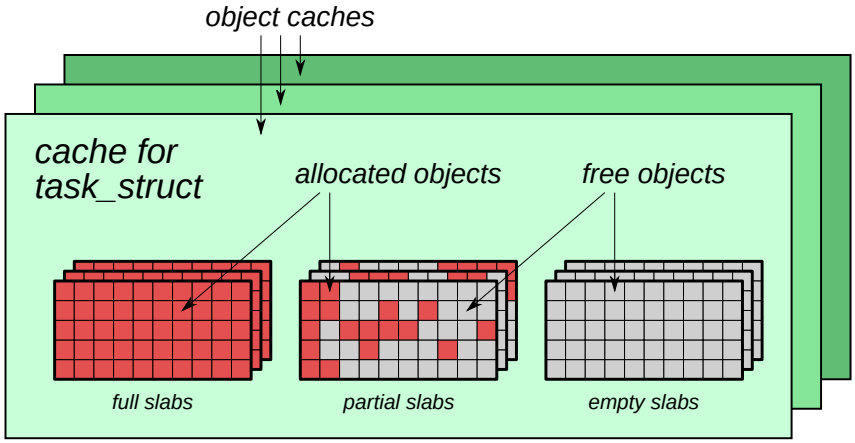

Figura 16.7: Estrutura de caches, slabs e objetos do alocador Slab.

A estratégia de alocação é a seguinte: quando um novo objeto de núcleo é requisitado, o alocador analisa o cache daquele tipo de objeto e entrega um objeto livre de um slab parcial; caso não hajam slabs parciais, entrega um objeto livre de um slab vazio (o que altera o status desse slab para parcial). Caso não existam slabs vazios, o alocador pede mais páginas de RAM ao alocador de memória física para criar um novo slab, inicializar seus objetos e marcá-los como livres. Quando um objeto é liberado, ele é marcado como livre; caso todos os objetos de um slab fiquem livres, este é marcado como vazio. Caso o sistema precise liberar memória para outros usos, o alocador pode descartar os slabs vazios, liberando suas áreas junto ao alocador de memória física.

O alocador Slab é usado para a gestão de objetos de núcleo em muitos sistemas operacionais, como Linux, Solaris, FreeBSD e Horizon (usado no console Nintendo Switch). No Linux, contadores de uso dos *slabs* em uso podem ser consultados no arquivo `/proc/slabinfo`:

[3]No núcleo Linux versão 5.4 há mais de 150 caches de objetos distintos.

```
# cat /proc/slabinfo

# name          <active_objs> <num_objs> <objsize> <obj/slab> <pg/slab>
RAWv6                 252         252       1152        28        8
UDPv6                 104         104       1216        26        8
fat_inode_cache        44          44        744        22        4
fat_cache               0           0         40       102        1
pid_namespace          76          76        208        19        1
posix_timers_cache    374         374        240        17        1
request_sock_TCP        0           0        304        26        2
TCP                   224         224       2048        16        8
sock_inode_cache     2783        2783        704        23        4
file_lock_cache       160         160        200        20        1
inode_cache         18902       18902        608        26        4
mm_struct             207         225       2112        15        8
files_cache           230         230        704        23        4
signal_cache          346         368       1024        16        4
task_struct          1141        1185       5824         5        8
...
```

16.6 Alocação no espaço de usuário

Da mesma forma que o núcleo, aplicações no espaço de usuário podem ter necessidade de alocar memória durante a execução para armazenar estruturas de dados dinâmicas (conforme discutido na Seção 15.3). Ao ser criado, cada processo recebe uma área para alocação dinâmica de variáveis, chamada `HEAP`. O tamanho dessa seção pode ser ajustado através de chamadas de sistema que modifiquem o ponteiro *Program Break* (vide Seção 15.2).

A gerência da seção `HEAP` pode ser bastante complexa, caso a aplicação use variáveis dinâmicas para construir listas, pilhas, árvores ou outras estruturas de dados mais sofisticadas. Por isso, ela usualmente fica a cargo de bibliotecas de sistema, como a biblioteca C padrão (*LibC*), que oferecem funções básicas de alocação de memória como `malloc` e `free`. A biblioteca então fica encarregada de alocar/liberar blocos de memória na seção `HEAP`, gerenciar quais blocos estão livres ou ocupados, e solicitar ao núcleo do SO o aumento ou redução dessa seção, conforme necessário. Se necessário, a biblioteca pode também requisitar ao núcleo áreas de memória fora do `HEAP` para finalidades específicas.

Existem várias implementações de alocadores de uso geral para o espaço de usuário. As implementações mais simples seguem o esquema apresentado na seção 16.2, com estratégia *best-fit*. Implementações mais sofisticadas, como a *DLmalloc* (*Doug Lea's malloc*) e *PTmalloc* (*Pthreads malloc*), usadas nos sistemas GNU/Linux, usam diversas técnicas para evitar a fragmentação e agilizar a alocação de blocos de memória.

Além dos alocadores de uso geral, podem ser desenvolvidos alocadores customizados para aplicações específicas. Uma técnica muito usada em sistemas de tempo real, por exemplo, é o *memory pool* (reserva de memória). Nessa técnica, um conjunto de blocos de mesmo tamanho é pré-alocado, constituindo um *pool*. A aplicação pode então obter e liberar blocos de memória desse *pool* com rapidez, pois o alocador só precisa registrar quais blocos estão livres ou ocupados.

Exercícios

1. Explique o que é *fragmentação externa*. Quais formas de alocação de memória estão livres desse problema?

2. Explique o que é *fragmentação interna*. Quais formas de alocação de memória estão livres desse problema?

3. Em que consistem as estratégias de alocação *first-fit*, *best-fit*, *worst-fit* e *next-fit*?

4. Considere um sistema com processos alocados de forma contígua na memória. Em um dado instante, a memória RAM possui os seguintes "buracos", em sequência e isolados entre si: 5K, 4K, 20K, 18K, 7K, 9K, 12K e 15K. Indique a situação final de cada buraco de memória após a seguinte sequência de alocações: $12K \rightarrow 10K \rightarrow 5K \rightarrow 8K \rightarrow 10K$. Considere as estratégias de alocação *first-fit*, *best-fit*, *worst-fit* e *next-fit*.

5. Considere um banco de memória com os seguintes "buracos" não-contíguos:

B1	B2	B3	B4	B5	B6
10MB	4MB	7MB	30MB	12MB	20MB

   Nesse banco de memória devem ser alocadas áreas de 5MB, 10MB e 2MB, nesta ordem, usando os algoritmos de alocação *First-fit*, *Best-fit* ou *Worst-fit*. Indique a alternativa correta:

   (a) Se usarmos *Best-fit*, o tamanho final do buraco B4 será de 6 Mbytes.
   (b) Se usarmos *Worst-fit*, o tamanho final do buraco B4 será de 15 Mbytes.
   (c) Se usarmos *First-fit*, o tamanho final do buraco B4 será de 24 Mbytes.
   (d) Se usarmos *Best-fit*, o tamanho final do buraco B5 será de 7 Mbytes.
   (e) Se usarmos *Worst-fit*, o tamanho final do buraco B4 será de 9 Mbytes.

6. Considere um alocador de memória do tipo *Buddy* binário. Dada uma área contínua de memória RAM com 1 GByte (1.024 MBytes), apresente a evolução da situação da memória para a sequência de alocações e liberações de memória indicadas a seguir.

   (a) Aloca A_1 200 MB
   (b) Aloca A_2 100 MB
   (c) Aloca A_3 150 MB
   (d) Libera A_2
   (e) Libera A_1
   (f) Aloca A_4 100 MB
   (g) Aloca A_5 40 MB
   (h) Aloca A_6 300 MB

Atividades

1. Construa um simulador de algoritmos básicos de alocação de memória. O simulador deve produzir aleatoriamente uma sequência de blocos de memória de tamanhos diferentes, simular sua alocação e gerar como saída o número de fragmentos livres de memória, os tamanhos do menor e do maior fragmentos e o tamanho médio dos fragmentos. Devem ser comparadas as estratégias de alocação *first-fit*, *next-fit*, *best-fit* e *worst-fit*.

Referências

J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Conference*, volume 16. Boston, MA, USA, 1994.

M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36, 1999.

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.