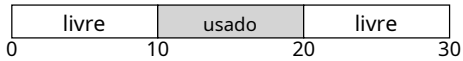
# 17

# Gerenciamento de espaço livre

Neste capítulo, faremos um pequeno desvio em nossa discussão sobre virtualização de memória para discutir um aspecto fundamental de qualquer sistema de gerenciamento de memória, seja ele uma biblioteca malloc (gerenciando páginas do heap de um processo) ou o próprio sistema operacional (gerenciando partes do endereço espaço de um processo). Especificamente, discutiremos as questões que envolvem**gerenciamento de espaço livre**.

Vamos tornar o problema mais específico. Gerenciar o espaço livre certamente pode ser fácil, como veremos quando discutirmos o conceito de**paginação**. É fácil quando o espaço que você gerencia está dividido em unidades de tamanho fixo; nesse caso, basta manter uma lista dessas unidades de tamanho fixo; quando um cliente solicitar um deles, retorne a primeira entrada.

Onde o gerenciamento do espaço livre se torna mais difícil (e interessante) é quando o espaço livre que você gerencia consiste em unidades de tamanho variável; isso surge em uma biblioteca de alocação de memória em nível de usuário (como emMalloc() e livre())e em um sistema operacional gerenciando memória física ao usar**segmentação** para implementar memória virtual. Em ambos os casos, o problema que existe é conhecido como**fragmentação externa**: o espaço livre é cortado em pequenos pedaços de diferentes tamanhos e assim fragmentado; as solicitações subsequentes podem falhar porque não há um único espaço contíguo que possa satisfazer a solicitação, mesmo que a quantidade total de espaço livre exceda o tamanho da solicitação.




A figura mostra um exemplo deste problema. Neste caso, o espaço livre total disponível é de 20 bytes; infelizmente, ele está fragmentado em dois pedaços de tamanho 10 cada. Como resultado, uma solicitação de 15 bytes falhará mesmo que haja 20 bytes livres. E assim chegamos ao problema abordado neste capítulo.

CRUX:HAITÓMUMA IDADEFREESRITMO
Como deve ser gerenciado o espaço livre, ao satisfazer necessidades de tamanho variável? missões? Que estratégias podem ser usadas para minimizar a fragmentação? Quais são as despesas gerais de tempo e espaço de abordagens alternativas?

## 17.1 Suposições

A maior parte desta discussão se concentrará na grande história dos alocadores encontrados nas bibliotecas de alocação de memória no nível do usuário. Baseamo-nos na excelente pesquisa de Wilson [W+95], mas encorajamos os leitores interessados a irem ao próprio documento de origem para mais detalhes[1].

Assumimos uma interface básica como a fornecida porMalloc()e livre(). Especificamente,void *malloc(tamanho t tamanho)leva um único parâmetro, tamanho,qual é o número de bytes solicitados pela aplicação; ele devolve um ponteiro (de nenhum tipo específico ou um**ponteiro vazio**na linguagem C) para uma região desse tamanho (ou superior). A rotina complementar vazio grátis (void *ptr)pega um ponteiro e libera o pedaço correspondente. Observe a implicação da interface: o usuário, ao liberar espaço, não informa à biblioteca seu tamanho; portanto, a biblioteca deve ser capaz de descobrir o tamanho de um pedaço de memória quando recebe apenas um ponteiro para ele. Discutiremos como fazer isso um pouco mais adiante neste capítulo.

O espaço que esta biblioteca administra é conhecido historicamente como opilha, e a estrutura de dados genérica usada para gerenciar o espaço livre no heap é algum tipo de**lista gratuita**. Essa estrutura contém referências a todos os pedaços de espaço livre na região gerenciada de memória. É claro que esta estrutura de dados não precisa ser uma lista*por si só*, mas apenas algum tipo de estrutura de dados para rastrear o espaço livre.

Assumimos ainda que estamos principalmente preocupados com **fragmentação externa**, como descrito acima. É claro que os alocadores também poderiam ter o problema de**fragmentação interna**; se um alocador distribuir pedaços de memória maiores do que o solicitado, qualquer espaço não solicitado (e, portanto, não utilizado) em tal pedaço será considerado*interno*fragmentação (porque o desperdício ocorre dentro da unidade alocada) e é outro exemplo de desperdício de espaço. Entretanto, por uma questão de simplicidade e por ser o mais interessante dos dois tipos de fragmentação, focaremos principalmente na fragmentação externa.

Também assumiremos que uma vez que a memória é entregue a um cliente, ela não pode ser realocada para outro local na memória. Por exemplo, se um programa chamaMalloc()e recebe um ponteiro para algum espaço dentro do heap, essa região de memória é essencialmente "de propriedade" do programa (e não pode ser movida pela biblioteca) até que o programa a retorne por meio de uma chamada correspondente paralivre().Assim, não**compactação**de espaço livre é possível, o que

[1]Tem quase 80 páginas; portanto, você realmente precisa estar interessado!

seria útil para combater a fragmentação[2]. A compactação poderia, no entanto, ser usada no sistema operacional para lidar com a fragmentação ao implementar **segmentação**(conforme discutido no referido capítulo sobre segmentação).

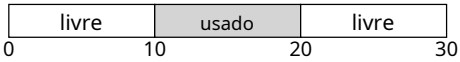
Finalmente, assumiremos que o alocador gerencia uma região contígua de bytes. Em alguns casos, um alocador poderia solicitar o crescimento daquela região; por exemplo, uma biblioteca de alocação de memória em nível de usuário pode chamar o kernel para aumentar o heap (por meio de uma chamada de sistema comosbrk)quando fica sem espaço. No entanto, para simplificar, assumiremos apenas que a região terá um único tamanho fixo durante toda a sua vida.

## 17.2 Mecanismos de Baixo Nível

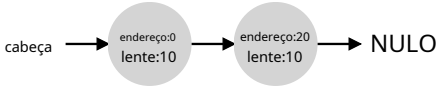
Antes de nos aprofundarmos em alguns detalhes da política, abordaremos primeiro alguns mecanismos comuns usados na maioria dos alocadores. Primeiro, discutiremos os conceitos básicos de divisão e coalescência, técnicas comuns em quase todos os alocadores. Em segundo lugar, mostraremos como é possível rastrear o tamanho das regiões alocadas com rapidez e relativa facilidade. Por fim, discutiremos como construir uma lista simples dentro do espaço livre para controlar o que é gratuito e o que não é.

### Divisão e Coalescência

Uma lista livre contém um conjunto de elementos que descrevem o espaço livre que ainda resta no heap. Portanto, suponha o seguinte heap de 30 bytes:




A lista gratuita para este heap teria dois elementos. Uma entrada descreve o primeiro segmento livre de 10 bytes (bytes 0 a 9) e uma entrada descreve o outro segmento livre (bytes 20 a 29):

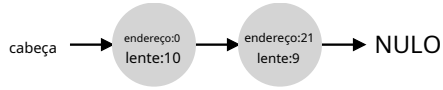



Conforme descrito acima, uma solicitação maior que 10 bytes falhará (retornando NULL); simplesmente não há um único pedaço contíguo de memória desse tamanho disponível. Uma solicitação exatamente desse tamanho (10 bytes) poderia ser facilmente atendida por qualquer um dos pedaços livres. Mas o que acontece se o pedido for para algo*menor*de 10 bytes?

Suponha que temos uma solicitação de apenas um único byte de memória. Neste caso, o alocador executará uma ação conhecida como**divisão**: ele encontrará

[2]Depois que você entrega um ponteiro para um pedaço de memória a um programa C, geralmente é difícil determinar todas as referências (ponteiros) para aquela região, que podem ser armazenadas em outras variáveis ou mesmo em registradores em um determinado ponto da execução. Este pode não ser o caso em linguagens mais fortemente tipadas e com coleta de lixo, o que permitiria, assim, a compactação como uma técnica para combater a fragmentação.

um pedaço livre de memória que pode satisfazer a solicitação e dividi-la em dois. O primeiro pedaço será retornado ao chamador; o segundo pedaço permanecerá na lista. Assim, em nosso exemplo acima, se uma solicitação de 1 byte fosse feita e o alocador decidisse usar o segundo dos dois elementos da lista para satisfazer a solicitação, a chamada para malloc() retornaria 20 (o endereço do região alocada de 1 byte) e a lista ficaria assim:




Na foto você pode ver que a lista permanece basicamente intacta; a única mudança é que a região livre agora começa em 21 em vez de 20, e a duração dessa região livre agora tem apenas 9[3]. Assim, a divisão é comumente usada em alocadores quando as solicitações são menores que o tamanho de qualquer pedaço livre específico.

Um mecanismo corolário encontrado em muitos alocadores é conhecido como **coalescendo**de espaço livre. Veja nosso exemplo acima mais uma vez (10 bytes livres, 10 bytes usados e outros 10 bytes livres).

Dado esse heap (minúsculo), o que acontece quando um aplicativo chama free(10), retornando assim o espaço no meio do heap? Se simplesmente adicionarmos esse espaço livre de volta à nossa lista sem pensar muito, poderemos acabar com uma lista parecida com esta:


Observe o problema: embora todo o heap esteja agora livre, ele está aparentemente dividido em três pedaços de 10 bytes cada. Assim, se um usuário solicitar 20 bytes, um simples percurso de lista não encontrará esse pedaço livre e retornará falha.

O que os alocadores fazem para evitar esse problema é unir o espaço livre quando um pedaço de memória é liberado. A ideia é simples: ao retornar um pedaço livre na memória, observe cuidadosamente os endereços do pedaço que você está retornando, bem como os pedaços próximos de espaço livre; se o espaço recém-liberado estiver próximo a um (ou dois, como neste exemplo) pedaços livres existentes, mescle-os em um único pedaço livre maior. Assim, com a coalescência, nossa lista final deverá ficar assim:


Na verdade, esta era a aparência inicial da lista de heap, antes de qualquer alocação ser feita. Com a coalescência, um alocador pode garantir melhor que grandes extensões livres estejam disponíveis para o aplicativo.

[3]Esta discussão pressupõe que não há cabeçalhos, uma suposição irrealista, mas simplificadora, que fazemos por enquanto.

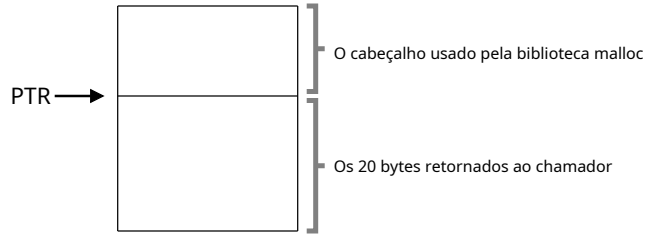



Figura 17.1:**Um cabeçalho de região Plus alocado**

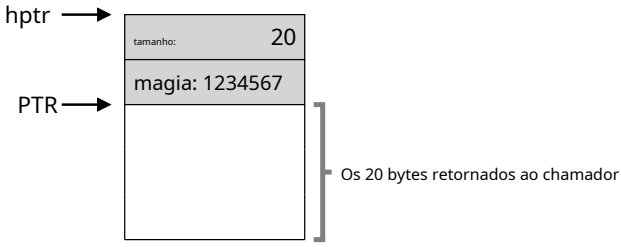



Figura 17.2:**Conteúdo específico do cabeçalho**

### Rastreando o tamanho das regiões alocadas

Você deve ter notado que a interface paragrátis(void *ptr)não aceita parâmetro de tamanho; portanto, presume-se que, dado um ponteiro, a biblioteca malloc pode determinar rapidamente o tamanho da região de memória que está sendo liberada e, assim, incorporar o espaço de volta à lista livre.

Para realizar esta tarefa, a maioria dos alocadores armazena um pouco de informação extra em um**cabeçalho**bloco que é mantido na memória, geralmente logo antes do pedaço de memória distribuído. Vejamos um exemplo novamente (Figura 17.1). Neste exemplo, estamos examinando um bloco alocado de tamanho 20 bytes, apontado porptr;imagine o usuário chamadoMalloc()e armazenou os resultados em ponto,por exemplo,ptr = malloc(20);.

O cabeçalho contém minimamente o tamanho da região alocada (neste caso, 20); também pode conter indicadores adicionais para acelerar a desalocação, um número mágico para fornecer verificação adicional de integridade e outras informações. Vamos supor um cabeçalho simples que contenha o tamanho da região e um número mágico, como este:

```
estrutura typedef {
    tamanho interno;
    magia interna;
} cabeçalho_t;
```

O exemplo acima seria parecido com o que você vê na Figura 17.2. Quando o usuário ligagrátis (ptr),a biblioteca então usa aritmética de ponteiro simples para descobrir onde o cabeçalho começa:

```
vazio livre(void *ptr) {
    cabeçalho_t *hptr = (cabeçalho_t *) ptr - 1; . . .
```

Depois de obter esse ponteiro para o cabeçalho, a biblioteca pode determinar facilmente se o número mágico corresponde ao valor esperado como uma verificação de integridade (assert(hptr->mágica == 1234567))e calcule o tamanho total da região recém-liberada por meio de matemática simples (ou seja, adicionando o tamanho do cabeçalho ao tamanho da região). Observe o detalhe pequeno, mas crítico, na última frase: o tamanho da região livre é o tamanho do cabeçalho mais o tamanho do espaço alocado ao usuário. Assim, quando um usuário solicitaNbytes de memória, a biblioteca não procura um pedaço livre de tamanhoN;em vez disso, ele procura um pedaço livre de tamanhoNmais o tamanho do cabeçalho.

#### Incorporando uma lista gratuita

Até agora tratamos a nossa lista livre simples como uma entidade conceitual; é apenas uma lista que descreve os pedaços livres de memória na pilha. Mas como construímos essa lista dentro do próprio espaço livre?

Em uma lista mais típica, ao alocar um novo nó, você simplesmente chamaria Malloc() quando você precisar de espaço para o nó. Infelizmente, dentro da biblioteca de alocação de memória, você não pode fazer isso! Em vez disso, você precisa construir a lista*dentro*o próprio espaço livre. Não se preocupe se isso parecer um pouco estranho; é, mas não tão estranho que você não consiga fazer isso!

Suponha que temos um pedaço de memória de 4.096 bytes para gerenciar (ou seja, o heap tem 4 KB). Para gerenciar isso como uma lista livre, primeiro temos que inicializar a referida lista; inicialmente, a lista deverá ter uma entrada, de tamanho 4096 (menos o tamanho do cabeçalho). Aqui está a descrição de um nó da lista:

```
typedef estrutura __node_t {
    interno                 tamanho;
    estrutura __node_t *próximo;
} nó_t;
```

Agora vamos dar uma olhada em algum código que inicializa o heap e coloca o primeiro elemento da lista livre dentro desse espaço. Estamos assumindo que o heap é construído dentro de algum espaço livre adquirido através de uma chamada para a chamada do sistema mmap(); esta não é a única maneira de construir tal pilha, mas nos serve bem neste exemplo. Aqui está o código:

```
// mmap() retorna um ponteiro para um pedaço de espaço livre node_t
*head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                        MAP_ANON|MAP_PRIVATE, -1, 0); =
cabeça->tamanho     4096 - tamanhode(nó_t); = NULO;
cabeça->próximo
```

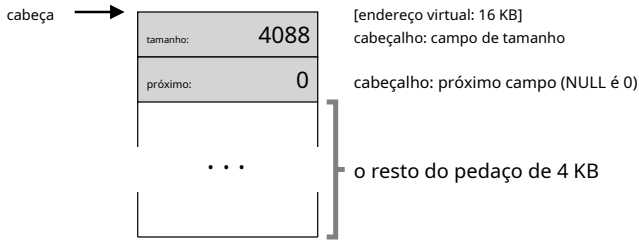



Figura 17.3:**Uma pilha com um pedaço grátis**

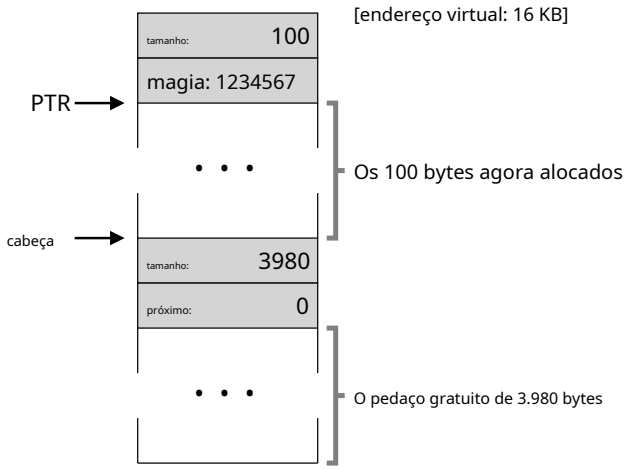



Figura 17.4:**Uma pilha: após uma alocação**

Depois de executar esse código, o status da lista é que ela possui uma única entrada, de tamanho 4088. Sim, é um heap minúsculo, mas serve como um bom exemplo para nós aqui. Ocabeçaponteiro contém o endereço inicial deste intervalo; vamos supor que seja 16 KB (embora qualquer endereço virtual sirva). Visualmente, o heap se parece com o que você vê na Figura 17.3.

Agora, vamos imaginar que um pedaço de memória seja solicitado, digamos, com tamanho de 100 bytes. Para atender a essa solicitação, a biblioteca primeiro encontrará um pedaço grande o suficiente para acomodar a solicitação; como há apenas um pedaço livre (tamanho: 4088), esse pedaço será escolhido. Então, o pedaço será **dividir** em dois: um pedaço grande o suficiente para atender a solicitação (e o cabeçalho, conforme descrito acima) e o pedaço livre restante. Assumindo um cabeçalho de 8 bytes (um tamanho inteiro e um número mágico inteiro), o espaço no heap agora se parece com o que você vê na Figura 17.4.

Assim, na solicitação de 100 bytes, a biblioteca alocou 108 bytes

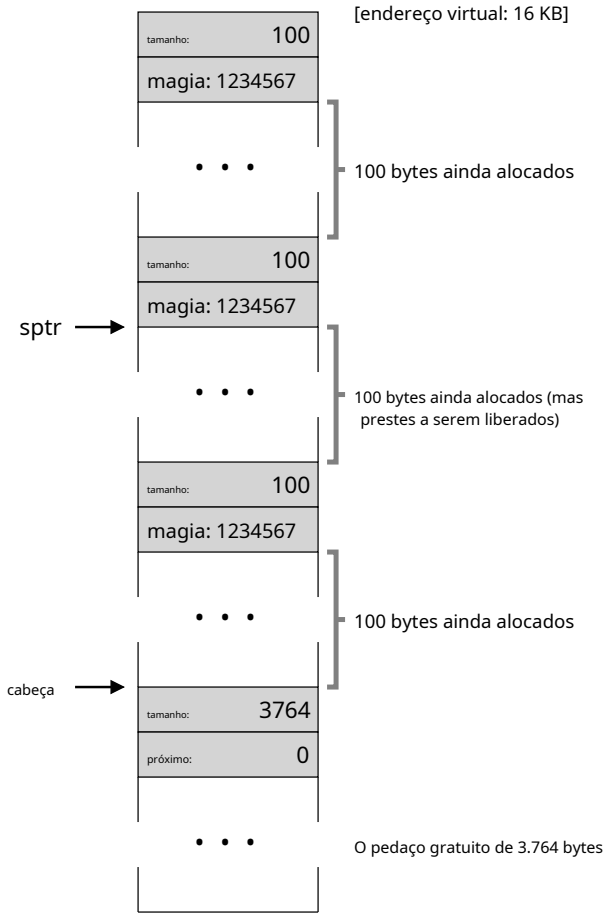



Figura 17.5:**Espaço livre com três pedaços alocados**

fora do pedaço livre existente, retorna um ponteiro (marcadoPTRna figura acima) para ele, armazena as informações do cabeçalho imediatamente antes do espaço alocado para uso posterior emlivre(),e reduz o único nó livre na lista para 3.980 bytes (4.088 menos 108).

Agora vamos dar uma olhada no heap quando há três regiões alocadas, cada uma com 100 bytes (ou 108 incluindo o cabeçalho). Uma visualização desse heap é mostrada na Figura 17.5.

Como você pode ver aqui, os primeiros 324 bytes do heap estão agora alocados e, portanto, vemos três cabeçalhos nesse espaço, bem como três regiões de 100 bytes sendo usadas pelo programa de chamada. A lista livre permanece desinteressante: apenas um único nó (apontado porcabeça),mas agora apenas 3764

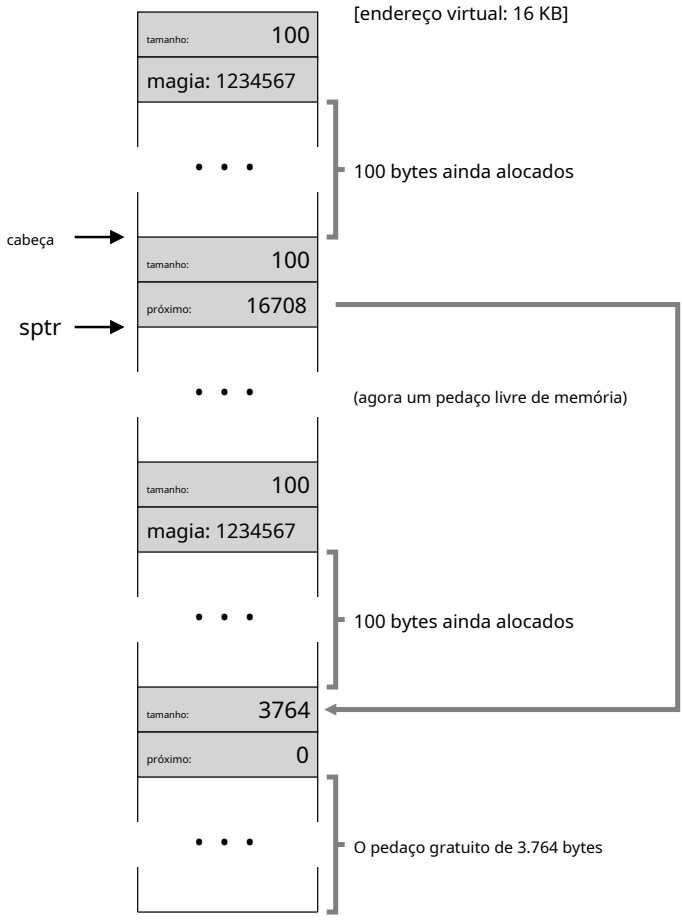



Figura 17.6:**Espaço livre com dois pedaços alocados**

bytes de tamanho após as três divisões. Mas o que acontece quando o programa chamador retorna alguma memória vialivre()?

Neste exemplo, o aplicativo retorna a parte intermediária da memória alocada, chamandográtis(16500) (o valor 16500 é obtido adicionando o início da região de memória, 16384, ao 108 do bloco anterior e aos 8 bytes do cabeçalho deste bloco). Este valor é mostrado no diagrama anterior pelo ponteirosptr.

A biblioteca descobre imediatamente o tamanho da região livre e, em seguida, adiciona o pedaço livre de volta à lista livre. Supondo que inserimos no topo da lista livre, o espaço agora fica assim (Figura 17.6).

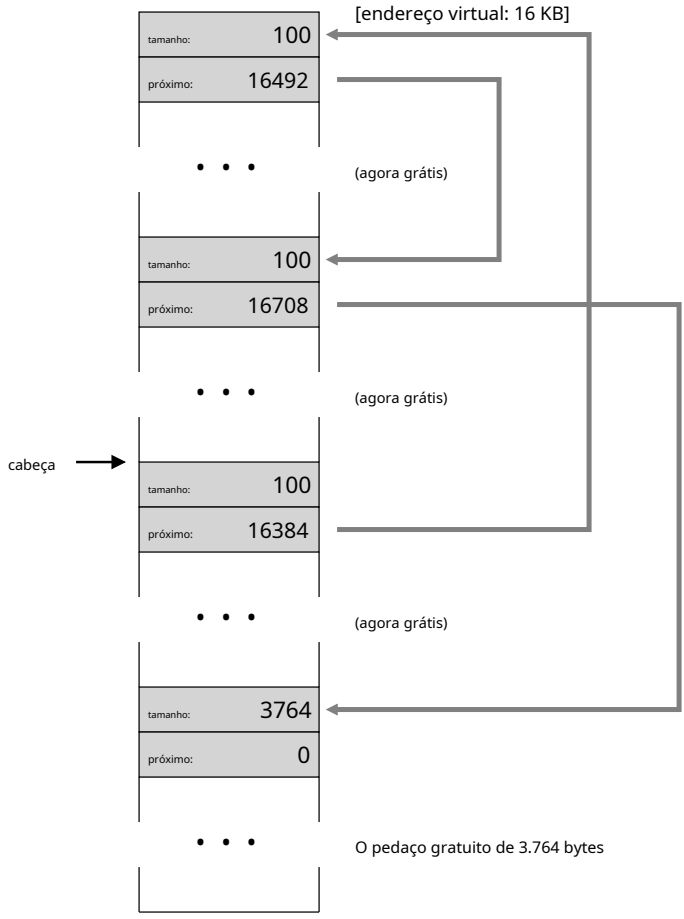



Figura 17.7:**Uma lista gratuita não coalescida**

Agora temos uma lista que começa com um pequeno pedaço livre (100 bytes, apontado pelo topo da lista) e um grande pedaço livre (3764 bytes). Nossa lista finalmente tem mais de um elemento! E sim, o espaço livre está fragmentado, uma ocorrência infeliz, mas comum.

Um último exemplo: vamos supor agora que os dois últimos pedaços em uso foram liberados. Sem coalescência, você acaba com fragmentação (Figura 17.7).

Como você pode ver na figura, agora temos uma grande bagunça! Por que? Simples, esquecemos de**coalescer**a lista. Embora toda a memória esteja livre, ela é cortada em pedaços, aparecendo assim como uma memória fragmentada, apesar de não o ser. A solução é simples: percorra a lista e**mesclar** pedaços vizinhos; quando terminar, o heap estará inteiro novamente.

## Crescendo a pilha

Deveríamos discutir um último mecanismo encontrado em muitas bibliotecas de alocação. Especificamente, o que você deve fazer se o heap ficar sem espaço? A abordagem mais simples é simplesmente falhar. Em alguns casos, esta é a única opção e, portanto, retornar NULL é uma abordagem honrosa. Não se sinta mal! Você tentou e, embora tenha falhado, lutou o bom combate.

A maioria dos alocadores tradicionais começa com um heap pequeno e solicita mais memória do sistema operacional quando acaba. Normalmente, isso significa que eles fazem algum tipo de chamada de sistema (por exemplo,sbrkna maioria dos UNIX sistemas) para aumentar o heap e, em seguida, alocar os novos pedaços a partir daí. Para atender osbrksolicitação, o sistema operacional encontra páginas físicas livres, mapeia-as no espaço de endereço do processo solicitante e, em seguida, retorna o valor do final do novo heap; nesse ponto, um heap maior estará disponível e a solicitação poderá ser atendida com êxito.

## 17.3 Estratégias Básicas

Agora que temos algumas máquinas sob controle, vamos examinar algumas estratégias básicas para gerenciar o espaço livre. Essas abordagens são baseadas principalmente em políticas bastante simples que você mesmo pode criar; experimente antes de ler e veja se você encontra todas as alternativas (ou talvez algumas novas!).

O alocador ideal é rápido e minimiza a fragmentação. Infelizmente, como o fluxo de alocação e de solicitações gratuitas pode ser arbitrário (afinal, são determinados pelo programador), qualquer estratégia específica pode ter um desempenho muito ruim, dado o conjunto errado de entradas. Assim, não descreveremos a "melhor" abordagem, mas sim falaremos sobre alguns princípios básicos e discutiremos seus prós e contras.

#### Melhor ajuste

O**melhor ajuste**A estratégia é bastante simples: primeiro, pesquise na lista livre e encontre pedaços de memória livre que sejam tão grandes ou maiores que o tamanho solicitado. Depois, retorne aquele que for menor naquele grupo de candidatos; este é o chamado pedaço de melhor ajuste (também pode ser chamado de menor ajuste). Uma passagem pela lista livre é suficiente para encontrar o bloco correto a ser retornado.

A intuição por trás do melhor ajuste é simples: ao retornar um bloco próximo ao que o usuário pede, o melhor ajuste tenta reduzir o desperdício de espaço. Contudo, há um custo; implementações ingênuas pagam uma pesada penalidade de desempenho ao realizar uma busca exaustiva pelo bloco livre correto.

#### Pior ajuste

O**pior ajuste**abordagem é o oposto do melhor ajuste; encontre o maior pedaço e retorne o valor solicitado; mantenha o pedaço restante (grande) na lista gratuita. O pior ajuste tenta deixar grandes pedaços livres em vez de muitos

pequenos pedaços que podem surgir de uma abordagem mais adequada. Mais uma vez, contudo, é necessária uma busca completa de espaço livre e, portanto, esta abordagem pode ser dispendiosa. Pior ainda, a maioria dos estudos mostra que ele tem um desempenho ruim, levando a uma fragmentação excessiva e ao mesmo tempo com altas despesas gerais.

#### Primeiro ajuste

O**fiprimeiro ajuste**O método simplesmente encontra o primeiro bloco que é grande o suficiente e retorna o valor solicitado ao usuário. Como antes, o espaço livre restante é mantido livre para solicitações subsequentes.

O primeiro ajuste tem a vantagem da velocidade — não é necessária uma busca exaustiva de todos os espaços livres — mas às vezes polui o início da lista livre com pequenos objetos. Assim, a forma como o alocador gerencia a ordem da lista livre torna-se um problema. Uma abordagem é usar**ordenação baseada em endereço**; ao manter a lista ordenada pelo endereço do espaço livre, a coalescência torna-se mais fácil e a fragmentação tende a ser reduzida.

#### Próximo ajuste

Em vez de sempre iniciar a busca de primeiro ajuste no início da lista, o**próximo ajuste**O algoritmo mantém um ponteiro extra para o local da lista onde alguém olhou pela última vez. A ideia é distribuir de maneira mais uniforme as buscas por espaço livre pela lista, evitando assim a fragmentação do início da lista. O desempenho de tal abordagem é bastante semelhante ao primeiro ajuste, pois uma busca exaustiva é mais uma vez evitada.

## Exemplos

Aqui estão alguns exemplos das estratégias acima. Visualize uma lista gratuita com três elementos, de tamanhos 10, 30 e 20 (ignoraremos os cabeçalhos e outros detalhes aqui, em vez disso nos concentraremos apenas em como as estratégias funcionam):


Suponha uma solicitação de alocação de tamanho 15. A abordagem mais adequada pesquisaria a lista inteira e descobriria que 20 era o mais adequado, pois é o menor espaço livre que pode acomodar a solicitação. A lista gratuita resultante:


Como acontece neste exemplo, e muitas vezes acontece com uma abordagem de melhor ajuste, resta agora um pequeno pedaço livre. Uma abordagem de pior ajuste é semelhante, mas em vez disso encontra o maior pedaço, neste exemplo 30. A lista resultante:

A estratégia de primeiro ajuste, neste exemplo, faz o mesmo que a de pior ajuste, encontrando também o primeiro bloco livre que pode satisfazer a solicitação. A diferença está no custo da busca; tanto o melhor quanto o pior ajuste examinam toda a lista; o first-fit examina apenas os pedaços livres até encontrar um que se encaixe, reduzindo assim o custo de pesquisa.

Estes exemplos apenas arranham a superfície das políticas de alocação. Análises mais detalhadas com cargas de trabalho reais e comportamentos mais complexos do alocador (por exemplo, coalescência) são necessárias para uma compreensão mais profunda. Talvez algo para uma seção de lição de casa, você diz?

## 17.4 Outras Abordagens

Além das abordagens básicas descritas acima, há uma série de técnicas e algoritmos sugeridos para melhorar de alguma forma a alocação de memória. Listamos alguns deles aqui para sua consideração (ou seja, para fazer você pensar um pouco mais do que apenas a alocação mais adequada).

### Listas Segregadas

Uma abordagem interessante que já existe há algum tempo é o uso de**listas segregadas**. A ideia básica é simples: se um aplicativo específico tiver uma (ou algumas) solicitações de tamanho popular, mantenha uma lista separada apenas para gerenciar objetos desse tamanho; todas as outras solicitações são encaminhadas para um alocador de memória mais geral.

Os benefícios de tal abordagem são óbvios. Por ter um pedaço de memória dedicado para um tamanho específico de solicitações, a fragmentação é uma preocupação muito menor; além disso, as solicitações de alocação e gratuitas podem ser atendidas rapidamente quando são do tamanho certo, já que não é necessária nenhuma pesquisa complicada em uma lista.

Assim como qualquer boa ideia, essa abordagem também introduz novas complicações no sistema. Por exemplo, quanta memória deve ser dedicada ao conjunto de memória que atende solicitações especializadas de um determinado tamanho, em oposição ao conjunto geral? Um alocador específico, o**alocador de laje**do superengenheiro Jeff Bonwick (que foi projetado para uso no kernel Solaris), lida com esse problema de uma maneira bastante agradável [B94].

Especificamente, quando o kernel é inicializado, ele aloca um número de**caches de objetos**para objetos do kernel que provavelmente serão solicitados com frequência (como bloqueios, inodes do sistema de arquivos, etc.); os caches de objetos, portanto, são listas livres segregadas de um determinado tamanho e atendem à alocação de memória e às solicitações gratuitas rapidamente. Quando um determinado cache está com pouco espaço livre, ele solicita alguns**lajes**de memória de um alocador de memória mais geral (sendo a quantidade total solicitada um múltiplo do tamanho da página e do objeto em questão). Por outro lado, quando todas as contagens de referência dos objetos dentro de um determinado bloco vão para zero, o alocador geral pode recuperá-los do alocador especializado, o que geralmente é feito quando o sistema VM precisa de mais memória.

ALADO:GREATEENGENHEIROSARÉRREALMENTEGREAT
Engenheiros como Jeff Bonwick (que não apenas escreveu o alocador de placas mencionado aqui, mas também foi o líder de um sistema de arquivos incrível, o ZFS) são o coração do Vale do Silício. Por trás de quase qualquer grande produto ou tecnologia está um ser humano (ou um pequeno grupo de seres humanos) que está muito acima da média em seus talentos, habilidades e dedicação. Como Mark Zuckerberg (do Facelivro) diz: "Alguém que é excepcional em sua função não é apenas um pouco melhor do que alguém que é muito bom. Eles são 100 vezes melhores." É por isso que, ainda hoje, uma ou duas pessoas podem abrir uma empresa que muda a face do mundo para sempre (pense no Google, na Apple ou no Facebook). Trabalhe duro e você poderá se tornar uma pessoa "100x" também! Se isso falhar, encontre uma maneira de trabalhar *com* tal pessoa; você aprenderá mais em um dia do que a maioria aprende em um mês.

O alocador de placas também vai além da maioria das abordagens de lista segregada, mantendo os objetos livres nas listas em um estado pré-inicializado. Bonwick mostra que a inicialização e a destruição de estruturas de dados são dispendiosas [B94]; ao manter os objetos liberados em uma lista específica em seu estado inicializado, o alocador de placas evita ciclos frequentes de inicialização e destruição por objeto e, assim, reduz visivelmente os custos indiretos.

### Alocação de amigos

Como a coalescência é crítica para um alocador, algumas abordagens foram projetadas para simplificar a coalescência. Um bom exemplo é encontrado no **alocador de amigos binário**[K65].

Em tal sistema, a memória livre é primeiramente pensada conceitualmente como uma grande espaço de tamanho $2^N$. Quando uma solicitação de memória é feita, a busca por espaço livre divide recursivamente o espaço livre por dois até que um bloco grande o suficiente para acomodar a solicitação seja encontrado (e uma divisão adicional em dois resultaria em um espaço muito pequeno). Neste ponto, o bloco solicitado é devolvido ao usuário. Aqui está um exemplo de um espaço livre de 64 KB sendo dividido na busca por um bloco de 7 KB (Figura 17.8, página 15).

No exemplo, o bloco de 8 KB mais à esquerda é alocado (conforme indicado pelo tom de cinza mais escuro) e retornado ao usuário; observe que este esquema pode sofrer **fragmentação interna**, já que você só pode distribuir blocos com potência de dois tamanhos.

A beleza da alocação de amigos está no que acontece quando esse bloco é liberado. Ao retornar o bloco de 8 KB para a lista livre, o alocador verifica se os 8 KB do "companheiro" estão livres; nesse caso, ele agrupa os dois blocos em um bloco de 16 KB. O alocador então verifica se o companheiro do bloco de 16KB ainda está livre; nesse caso, ele une esses dois blocos. Esse processo de coalescência recursiva continua subindo na árvore, seja restaurando todo o espaço livre ou parando quando um amigo estiver em uso.

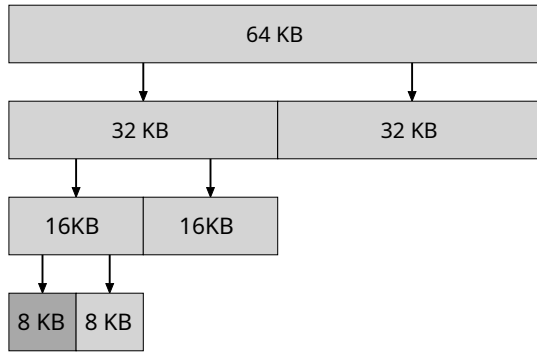



Figura 17.8:**Exemplo de heap gerenciado por amigos**

A razão pela qual a alocação de amigos funciona tão bem é que é simples determinar o amigo de um bloco específico. Como, você pergunta? Pense nos endereços dos blocos no espaço livre acima. Se você pensar com bastante cuidado, verá que o endereço de cada par de amigos difere apenas em um único bit; qual bit é determinado pelo nível na árvore de amigos. E assim você tem uma ideia básica de como funcionam os esquemas de alocação de amigos binários. Para mais detalhes, como sempre, veja a pesquisa de Wilson [W+95].

#### Outras ideias

Um grande problema com muitas das abordagens descritas acima é a falta de **dimensionamento**. Especificamente, a pesquisa em listas pode ser bastante lenta. Assim, os alocadores avançados utilizam estruturas de dados mais complexas para lidar com esses custos, trocando simplicidade por desempenho. Os exemplos incluem árvores binárias balanceadas, árvores espalhadas ou árvores parcialmente ordenadas [W+95].

Dado que os sistemas modernos geralmente têm múltiplos processadores e executam cargas de trabalho multithread (algo que você aprenderá detalhadamente na seção do livro sobre Simultaneidade), não é surpreendente que muito esforço tenha sido gasto para fazer os alocadores funcionarem bem. em sistemas baseados em multiprocessadores. Dois exemplos maravilhosos são encontrados em Berger et al. [B+00] e Evans [E06]; verifique-os para obter detalhes.

Estas são apenas duas das milhares de ideias que as pessoas tiveram ao longo do tempo sobre alocadores de memória; leia por conta própria se estiver curioso. Caso contrário, leia sobre como funciona o alocador glibc [S15], para ter uma ideia de como é o mundo real.

## 17.5 Resumo

Neste capítulo, discutimos as formas mais rudimentares de alocadores de memória. Esses alocadores existem em todos os lugares, vinculados a cada programa C que você escreve, bem como no sistema operacional subjacente que gerencia a memória para suas próprias estruturas de dados. Tal como acontece com muitos sistemas, existem muitos

compensações a serem feitas na construção de tal sistema, e quanto mais você souber sobre a carga de trabalho exata apresentada a um alocador, mais poderá fazer para ajustá-lo para funcionar melhor para essa carga de trabalho. Criar um alocador rápido, eficiente em termos de espaço e escalonável que funcione bem para uma ampla gama de cargas de trabalho continua sendo um desafio constante nos sistemas de computadores modernos.

## Referências

[B+00] "Hoard: um alocador de memória escalável para aplicativos multithreaded" por Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson. ASPLOS-IX, novembro de 2000. *Excelente alocador de Berger e companhia para sistemas multiprocessadores. Além de ser um papel divertido, também é utilizado na prática!*

[B94] "O alocador de Slab: um alocador de memória do kernel com cache de objetos", por Jeff Bonwick. USÊNIX '94. *Um artigo interessante sobre como construir um alocador para um kernel de sistema operacional e um ótimo exemplo de como se especializar para tamanhos de objetos comuns específicos.*

[E06] "Uma implementação escalável de malloc(3) simultâneo para FreeBSD" por Jason Evans. Abril de 2006. http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf. *Uma visão detalhada de como construir um alocador realmente moderno para uso em multiprocessadores. O alocador "jemalloc" é amplamente utilizado hoje, no FreeBSD, NetBSD, Mozilla Firefox e no Facebook.*

[K65] "Um alocador rápido de armazenamento" por Kenneth C. Knowlton. Comunicações da ACM, Volume 8:10, outubro de 1965. *A referência comum para alocação de amigos. Fato estranho e aleatório: Knuth dá crédito pela ideia não a Knowlton, mas a Harry Markowitz, um economista ganhador do Prêmio Nobel. Outro fato estranho: Knuth comunica todos os seus e-mails através de uma secretária; ele mesmo não envia e-mail, mas diz à secretária qual e-mail enviar e então a secretária faz o trabalho de enviar o e-mail.*

*Último fato de Knuth: ele criou o TeX, a ferramenta usada para compor este livro. É um software incrível*[4].

[S15] "Compreendendo o glibc malloc" por Sploitfun. Fevereiro de 2015. sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/. *Um mergulho profundo em como funciona o glibc malloc. Surpreendentemente detalhado e uma leitura muito legal.*

[W+95] "Alocação dinâmica de armazenamento: uma pesquisa e revisão crítica" por Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. Workshop Internacional sobre Gerenciamento de Memória, Escócia, Reino Unido, setembro de 1995. *Uma pesquisa excelente e abrangente sobre muitas facetas da alocação de memória. Muitos detalhes para entrar neste pequeno capítulo!*

[4] Na verdade usamos LaTeX, que é baseado nas adições de Lamport ao TeX, mas próximo o suficiente.

**Trabalho de casa (simulação)**

O programa,malloc.py,permite explorar o comportamento de um alocador de espaço livre simples, conforme descrito no capítulo. Consulte o README para obter detalhes de sua operação básica.

## Questões

1. Primeira corrida com as bandeiras -n 10 -H 0 -p MELHOR -s 0para gerar algumas alocações e liberações aleatórias. Você pode prever o que alloc()/free() retornará? Você consegue adivinhar o estado da lista gratuita após cada solicitação? O que você percebe na lista gratuita ao longo do tempo?
2. Como os resultados são diferentes ao usar uma política de pior ajuste para pesquisar a lista gratuita (-pior)?O que muda?
3. E quando usar FIRST fit (-p PRIMEIRO)?O que acelera quando você usa o primeiro ajuste?
4. Para as questões acima, a forma como a lista é mantida ordenada pode afetar o tempo que leva para encontrar um local gratuito para algumas das apólices. Use as diferentes ordenações de listas gratuitas (-lADDRSORT, -l SIZESORT+, - l SIZESORT-)para ver como as políticas e a ordenação das listas interagem.
5. A fusão de uma lista gratuita pode ser muito importante. Aumentar o número de alocações aleatórias (digamos -número 1000).O que acontece com solicitações de alocação maiores ao longo do tempo? Execute com e sem coalescência (ou seja, sem e com o -Cflag). Que diferenças no resultado você vê? Qual é o tamanho da lista gratuita ao longo do tempo em cada caso? A ordem da lista é importante neste caso?
6. O que acontece quando você altera a fração percentual alocada -P para mais de 50? O que acontece com as alocações quando se aproxima de 100? E quando a porcentagem se aproxima de 0?
7. Que tipos de solicitações específicas você pode fazer para gerar um espaço livre altamente fragmentado? Use o -Um flag para criar listas gratuitas fragmentadas e ver como diferentes políticas e opções alteram a organização da lista gratuita.