CS 241: programação do sistema

- atribuições
- Quizzes
- Socorro!
- Cronograma
- Honras
- Funcionários
- Livro didático
- Links

Malloc

- Alocadores de Memória
  - Introdução
  - API de alocação de memória C
    - Heaps e sbrk
  - Introdução à alocação
    - Estratégias de Posicionamento
    - Estratégia de posicionamento prós e contras
  - Tutorial de alocador de memória
    - Implementando malloc
    - Alinhamento e considerações de arredondamento
    - Implementando gratuitamente
    - atuação
    - Alocadores de listas livres explícitas
  - Estudo de caso: Buddy Allocator, um exemplo de uma lista segregada
  - Leitura Adicional
  - Tópicos
  - Perguntas / Exercícios

Memória em toda parte, mas não uma alocação a ser feita - Uma pilha realmente fragmentada

Introdução

Alocação de memória é muito importante! Alocar e desalocar memória heap é uma das operações mais comuns em qualquer aplicativo. O heap no nível do sistema é uma série de endereços contíguos que o programa pode expandir ou contrair e usar como acordo próprio (“Overview of Malloc” # ref-mallocinternals). No POSIX, isso é chamado de quebra do sistema. Usamos o sbrk para mover a quebra do sistema. A maioria dos programas não interage diretamente com essa chamada, eles usam um sistema de alocação de memória em torno dele para lidar com chunking e manter o controle de qual memória é alocada e qual é liberada.

Estaremos principalmente procurando alocadores simples. Só sei que existem outras maneiras de dividir a memória como com o mmap ou outros esquemas de alocação e métodos como o jemalloc .

API de alocação de memória C

- malloc(size_t bytes) é uma chamada da biblioteca C e é usada para reservar um bloco contíguo de memória que pode ser não inicializado (Jones # ref-jones2010wg14 P. 348). Ao contrário da memória de pilha, a memória permanece alocada até que seja free com o mesmo ponteiro. Se malloc pode retornar um ponteiro para pelo menos o espaço livre solicitado ou NULL . Isso significa que malloc pode retornar NULL mesmo que haja alguns espaços. Programas robustos devem verificar o valor de retorno. Se o seu código presumir que o malloc sucesso, e isso não acontecer, o programa provavelmente falhará (segfault) quando tentar gravar no endereço 0. Além disso, o malloc pode não zerar a memória devido ao desempenho - verifique seu código para ter certeza de que você não estão usando valores não inicializados.
- realloc(void *space, size_t bytes) permite que você redimensione uma alocação de memória existente que foi anteriormente alocada no heap (via malloc, calloc ou realloc) (Jones # ref-jones2010wg14 P. 349). O uso mais comum de realloc é redimensionar a memória usada para manter uma matriz de valores. Existem duas armadilhas com realloc. Um, um novo ponteiro pode ser retornado. Dois, pode falhar. Uma versão ingênua mas legível do realloc é sugerida abaixo com o uso da amostra.
  void * realloc (void * ptr , size_t newsize) { // Simple implementation always reserves more memory // and has no error checking void * result = malloc (newsize); size_t oldsize = ... //(depends on allocator's internal data structure) if (ptr) memcpy (result , ptr , newsize < oldsize ? newsize : oldsize); free (ptr); return result ; } int main () { // 1 int * array = malloc (sizeof (int) * 2); array [0] = 10 ; array [1] = 20 ; // Oops need a bigger array - so use realloc.. array = realloc (array , 3 * sizeof (int)); array [2] = 30 ; }
  O código acima não é robusto. Se realloc falhar, o array é um vazamento de memória. Código robusto verifica o valor de retorno e reatribui somente o ponteiro original se não for nulo.
  int main () { // 1 int * array = malloc (sizeof (int) * 2); array [0] = 10 ; array [1] = 20 ; void * tmp = realloc (array , 3 * sizeof (int)); if (tmp == NULL) { // Nothing I can do here } else if (tmp == array) { // realloc returned same space array [2] = 30 ; } else { // realloc returned different space array = tmp ; array [2] = 30 ; } }
- calloc(size_t nmemb, size_t size) inicializa o conteúdo da memória para zero e também recebe dois argumentos: o número de itens e o tamanho em bytes de cada item. Uma discussão avançada sobre essas limitações é http://locklessinc.com/articles/calloc/ . Os programadores costumam usar calloc vez de chamar explicitamente memset após malloc , para definir o conteúdo da memória para zero porque certas considerações de desempenho são levadas em conta. Nota calloc(x,y) é idêntico ao calloc(y,x) , mas você deve seguir as convenções do manual. Uma implementação ingênua do calloc está abaixo.

void * calloc (size_t n , size_t size) { size_t total = n * size ; // Does not check for overflow! void * result = malloc (total); if (! result) return NULL ; // If we're using new memory pages // just allocated from the system by calling sbrk // then they will be zero so zero-ing out is unnecessary, // We will be non-robust and memset either way. return memset (result , 0 , total); }

- free leva um ponteiro para o início de uma parte da memória e a torna disponível para uso nas chamadas subsequentes para as outras funções de alocação. Isso é importante porque não queremos que todo processo em nosso espaço de endereçamento tenha uma enorme quantidade de memória. Quando terminarmos de usar a memória, paramos de usá-la gratuitamente. Um uso simples está abaixo.
  int * ptr = malloc (sizeof (* ptr)); do_something (ptr); free (ptr);
  Se você usar um pedaço de memória depois que ele for liberado - isso é um comportamento indefinido.

Heaps e sbrk

O heap faz parte da memória do processo e não possui um tamanho fixo. Alocação de memória heap é executada pela biblioteca C quando você chama malloc (calloc , realloc) e free . Ao chamar sbrk a biblioteca C pode aumentar o tamanho do heap, já que seu programa exige mais memória heap. À medida que o heap e a pilha precisam crescer, os colocamos em extremidades opostas do espaço de endereço. As pilhas não crescem como um monte, novas partes da pilha são alocadas para novos segmentos. Assim, para arquiteturas típicas, o heap crescerá para cima e a pilha crescerá para baixo.
Atualmente, os alocadores modernos de memória do sistema operacional não precisam mais do sbrk - em vez disso, eles podem solicitar regiões independentes de memória virtual e manter várias regiões de memória. Por exemplo, solicitações de gibibyte podem ser colocadas em uma região de memória diferente de solicitações de alocação pequenas. No entanto, esse detalhe é uma complexidade indesejada.
Os programas não precisam chamar brk ou sbrk normalmente, embora chamar sbrk(0) possa ser interessante porque informa onde seu heap termina atualmente. Em vez disso, os programas usam malloc , calloc , realloc e free que fazem parte da biblioteca C. A implementação interna dessas funções pode chamar sbrk quando a memória heap adicional é necessária.
void * top_of_heap = sbrk (0); malloc (16384); void * top_of_heap2 = sbrk (0); printf ("The top of heap went from %p to %p \n " , top_of_heap , top_of_heap2); // Example output: The top of heap went from 0x4000 to 0xa000
Um fato importante que acabamos de descrever anteriormente é que a memória que foi recentemente obtida pelo sistema operacional deve ser zerada. Se o sistema operacional não zerou o conteúdo da RAM física, talvez seja possível para um processo aprender sobre a memória de outro processo que havia usado a memória anteriormente. Isso seria um vazamento de segurança. Infelizmente, isso significa que, para solicitações de malloc antes que qualquer memória tenha sido liberada, *muitas vezes* é zero. Isso é lamentável porque, muitos programadores confundem programas C escritos que assumem que a memória malloc será *sempre* zero.
char * ptr = malloc (300); // contents is probably zero because we get brand new memory // so beginner programs appear to work! // strcpy(ptr, "Some data"); // work with the data free (

ptr); // later char * ptr2 = malloc (300); // Contents might now contain existing data and is probably not zero

Introdução à Alocação

Vamos tentar escrever Malloc. Aqui está uma primeira tentativa - a versão ingênua.
void * malloc (size_t size) // Ask the system for more bytes by extending the heap space. // sbrk Returns -1 on failure void * p = sbrk (size); if (p == (void *) - 1) return NULL ; // No space left return p ; } void free () { /* Do nothing */ }
Acima está a implementação mais simples do malloc, mas existem algumas desvantagens.

- As chamadas do sistema são lentas em comparação com as chamadas da biblioteca. Devemos reservar uma grande quantidade de memória e só ocasionalmente pedir mais do sistema.
- Nenhuma reutilização de memória liberada. Nosso programa nunca reutiliza memória de heap - ele apenas pede uma pilha maior.

Se esse alocador fosse usado em um programa típico, o processo esgotaria rapidamente toda a memória disponível. Em vez disso, precisamos de um alocador que possa usar com eficiência o espaço do heap e só pedir mais memória quando necessário. Existem programas que podem usar esse tipo de alocador. Considere um videogame alocando objetos para carregar a próxima cena. É consideravelmente mais rápido fazer o que foi mencionado acima e simplesmente jogar fora todo o bloco de memória do que fazer as seguintes estratégias de posicionamento.

Estratégias de Posicionamento

Durante a execução do programa, a memória é alocada e desalocada, portanto, haverá espaço na memória heap que pode ser reutilizada para futuras solicitações de memória. O alocador de memória precisa controlar quais partes do heap estão atualmente alocadas e quais partes estão disponíveis. Suponha que nosso tamanho de heap atual seja 64K, embora nem todo ele esteja em uso, porque alguma memória malloc anterior já foi liberada pelo programa. Digamos que nosso heap se parece com a tabela a seguir.

16KiB	**10KiB**	**1 KiB**	**1 KiB**	**30 KiB**	**4KiB**	**2 KiB**
livre	alocado	livre	alocado	livre	alocado	livre

Se uma nova requisição de malloc para 2KiB é executada (malloc(2048)), onde malloc deve reservar a memória? Poderia usar o último buraco de 2KiB, que é o tamanho perfeito! Ou poderia dividir um dos outros dois buracos livres. Essas escolhas representam diferentes estratégias de veiculação. Qualquer que seja o buraco escolhido, o alocador precisará dividir o buraco em dois. O espaço recém-alocado, que será devolvido ao programa e um buraco menor, se houver espaço sobrando. Uma estratégia de ajuste perfeito encontra o menor buraco que é de tamanho suficiente (pelo menos 2 KiB):

16KiB	**10KiB**	**1 KiB**	**1 KiB**	**30 KiB**	**4KiB**	**2 KiB**
livre	alocado	livre	alocado	livre	alocado	HERE!

Uma estratégia de pior ajuste encontra o maior buraco que é de tamanho suficiente, então divida o buraco de 30 KiB em dois:

16KiB	**10KiB**	**1 KiB**	**1 KiB**	**2 KiB**	**28 KiB**	**4KiB**	**2 KiB**
livre	alocado	livre	alocado	HERE!	livre	alocado	livre

Uma estratégia de primeiro ajuste encontra o primeiro buraco disponível que é de tamanho suficiente, então divida o buraco de 16 KiB em dois:

2 KiB	**14KiB**	**10KiB**	**1 KiB**	**1 KiB**	**30 KiB**	**4KiB**	**2 KiB**
HERE!	livre	alocado	livre	alocado	livre	alocado	livre

Para introduzir outro conceito, a fragmentação externa é que, embora tenhamos memória suficiente no heap, ela pode ser dividida de uma forma que o desgaste não é capaz de fornecer o valor total. No exemplo abaixo, dos 64 KiB de memória heap, 17 KiB é alocado e 47 KiB é gratuito. No entanto, o maior bloco disponível é de apenas 30 KiB porque a nossa memória heap não alocada disponível é fragmentada em partes menores.

16KiB	**10KiB**	**1 KiB**	**1 KiB**	**30 KiB**	**4KiB**	**2 KiB**
livre	alocado	livre	alocado	livre	alocado	livre

Estratégia de posicionamento prós e contras

Os desafios de escrever um alocador de heap são

- Precisa minimizar a fragmentação (ou seja, maximizar a utilização da memória)
- Precisa de alto desempenho
- Implementação complicada - muita manipulação de ponteiros usando listas vinculadas e aritmética de ponteiros.
- Tanto a fragmentação quanto o desempenho dependem do perfil de alocação do aplicativo, que pode ser avaliado, mas não previsto e, na prática, sob condições de uso específicas, um alocador de finalidades especiais pode frequentemente superar uma implementação de propósito geral.
- O alocador não conhece as solicitações de alocação de memória do programa antecipadamente. Mesmo se o fizermos, este é o http://en.wikipedia.org/wiki/Knapsack_problem que é conhecido por ser NP difícil!

Estratégias diferentes afetam a fragmentação da memória heap de maneiras não óbvias, que são descobertas apenas por análise matemática ou por simulações cuidadosas sob condições do mundo real (por exemplo, simulando as solicitações de alocação de memória de um banco de dados ou servidor da web).
Primeiro, teremos uma abordagem mais matemática e única para cada um desses algoritmos (Garey, Graham e Ullman # ref-Garey: 1972: WAM: 800152.804907). O documento descreve um cenário em que você tem um certo número de posições e um certo número de alocações, e você está tentando encaixar as alocações em tão poucas caixas quanto possível, portanto, usando o mínimo de memória possível. O artigo discute as implicações teóricas e coloca um bom limite nas relações a longo prazo entre o uso de memória ideal e o uso real da memória. Para aqueles que estão interessados, o artigo

conclui que o uso de memória real sobre o uso de memória ideal à medida que o número de posições aumenta - as caixas podem ter qualquer distribuição - é de aproximadamente 1,7 para ajuste inicial e inferior para 1,7 para melhor ajuste. O problema com essa análise é que pouquíssimos aplicativos do mundo real precisam desse tipo de alocação única. As alocações de objetos de videogame geralmente designam um sub-pico diferente para cada nível e preenchem esse sub-pico se precisarem de um esquema de alocação de memória rápida que possam ser descartadas.
Na prática, usaremos o resultado de uma pesquisa mais rigorosa realizada em 2005 (Wilson et al. # Ref-10.1007 / 3-540-60368-9_19). O documento certifica-se de que a alocação de memória é um alvo em movimento. Um bom esquema de alocação para um programa pode não ser um bom esquema de alocação para outro programa. Os programas não seguem uniformemente a distribuição de alocações, muito mesmo. A pesquisa fala sobre todos os esquemas de alocação que nós introduzimos, bem como alguns extras. Aqui estão alguns tópicos resumidos

1. O melhor ajuste pode ter problemas quando um bloco é escolhido com o tamanho certo, e o espaço restante é tão pequeno que um programa provavelmente não o usará. Uma maneira de contornar isso poderia ser definir um limite para divisão. Essa pequena divisão não é observada com freqüência sob carga de trabalho regular. Além disso, o pior comportamento do melhor ajuste é ruim, mas geralmente não acontece p. 43
2. O artigo também fala sobre uma importante distinção do primeiro ajuste. Existem várias noções de primeiro. O primeiro pode ser encomendado em termos de tempo livre, ou pode ser encomendado através dos endereços do início do bloco, ou pode ser encomendado no momento da última liberação - sendo o primeiro menos usado recentemente. A pesquisa não aprofundou muito o desempenho de cada uma, mas fez uma observação de que o endereço solicitado e as listas usadas menos recentemente acabaram tendo um desempenho melhor que o usado pela primeira vez mais recentemente.
3. O artigo conclui, primeiramente, dizendo que sob cargas de trabalho aleatórias simuladas (assumindo uniformes ao acaso), o melhor ajuste e o primeiro ajuste também o fazem. Mesmo na prática, tanto o melhor como o endereço ordenado pela primeira vez se ajustam tão bem quanto com um limiar de divisão e coalescência. As razões pelas quais não são totalmente conhecidas.

Algumas notas adicionais que fazemos

1. O melhor ajuste pode não exigir uma varredura completa do heap. Quando um bloco de tamanho perfeito ou de tamanho perfeito dentro de um limite é encontrado, ele pode ser retornado, dependendo da política de caso de borda que você possui.
2. Pior ajuste pode não precisar escanear todo o heap também. Dependendo de como você configura as estruturas de dados, seu heap pode representar o max-heap da estrutura de dados e cada chamada de alocação simplesmente precisa estourar o topo, re-embeber e possivelmente inserir um bloco de memória dividida. Usando pilhas de Fibonacci, isso pode ser extremamente ineficiente.
3. O primeiro ajuste precisa ter algum tipo de pedido. Na maioria das vezes, as pessoas padronizarão as listas vinculadas, o que é uma boa escolha. Não há muitas

melhorias que você possa fazer com a política de lista vinculada usada menos recentemente e mais recentemente, mas com listas vinculadas ordenadas por endereço você pode acelerar a inserção de O (n) para O (log (n)) usando um random-skip-list em conjunto com a sua lista unicamente ligada. Basicamente, uma inserção usaria a lista de pulos como atalhos para encontrar o lugar certo para inserir o bloco e uma remoção passaria pela lista normalmente.

4. Existem muitas estratégias de posicionamento sobre as quais não falamos, uma é a próxima, que é ajustada em primeiro lugar no próximo bloco de ajuste. Isso adiciona uma espécie de aleatoriedade determinista. Não se espera que você conheça esse algoritmo, apenas saiba que ao implementar um alocador de memória como parte de um problema de máquina, existem mais que isso.

Tutorial de alocador de memória

Um alocador de memória precisa controlar quais bytes estão atualmente alocados e quais estão disponíveis para uso. Esta seção apresenta a implementação e os detalhes conceituais da construção de um alocador ou o código real que implementa malloc e free . Conceitualmente, estamos pensando em criar listas e listas de blocos vinculados! Por favor, aproveite a seguinte arte ASCII. bt é curto para tag de limite.
| I am one block | I am another | Tiny block | |----------------|----------------|------------| | meta | space |bt| meta | space|bt| meta |spc|bt| |----------------|----------------|------------|
Teremos ponteiros implícitos em nosso próximo bloco, o que significa que podemos obter de um bloco para outro usando a adição. Isso está em contraste com um metadata *next campo de metadata *next explícito metadata *next em nosso meta-bloco.
|----------------| p | |----------------|----------------|------------| | meta | space |bt| meta | space|bt| meta |spc|bt| |----------------|----------------|------------| If p is char pointer to the start of our block, to get to the next block would be p + sizeof(meta) + p->space + sizeof(boundary_tag)
O espaçamento real pode ser diferente. Os metadados podem conter coisas diferentes. Um metadado mínimo teria simplesmente o tamanho do bloco.
Como escrevemos inteiros e ponteiros na memória que já controlamos, mais tarde podemos consistentemente pular de um endereço para o outro. Esta informação interna representa alguma sobrecarga. Ou seja, mesmo se tivéssemos solicitado 1024 KiB de memória contígua do sistema, não poderíamos fornecer tudo isso para o programa em execução.
Nossa memória heap é uma lista de blocos onde cada bloco é alocado ou não alocado. Portanto, conceitualmente, há uma lista de blocos livres, mas está implícita na forma de informações de tamanho de bloco que armazenamos como parte de cada bloco. Vamos pensar nisso em termos de uma implementação simples.
typedef struct { size_t block_size ; char data [0]; } block ; block * p = sbrk (100); p -> size = 100 - sizeof (* p) - sizeof (boundary_tag); // Other block allocations
Poderíamos navegar de um bloco para o próximo bloco apenas adicionando o tamanho do bloco.
p + sizeof (meta) + p -> block_size + sizeof (boundary_tag)
Certifique-se de obter o seu casting certo! Caso contrário, você irá mover uma quantidade extrema de bytes.
O programa de chamada nunca vê esses valores. Eles são internos à implementação do alocador de memória. Como exemplo, suponha que seu alocador seja solicitado a reservar

80 bytes (malloc(80)) e requer 8 bytes de dados de cabeçalho internos. O alocador precisaria encontrar um espaço não alocado de pelo menos 88 bytes. Depois de atualizar os dados do heap, ele retornaria um ponteiro para o bloco. No entanto, o ponteiro retornado não aponta para o início do bloco porque é onde os dados de tamanho interno são armazenados! Em vez disso, retornaríamos o início do bloco + 8 bytes. Na implementação, lembre-se de que a aritmética do ponteiro depende do tipo. Por exemplo, p += 8 adiciona 8 sizeof(p) , não necessariamente 8 bytes!

Implementando malloc

A implementação mais simples usa o primeiro ajuste. Comece no primeiro bloco, assumindo que ele existe, e itere até encontrar um bloco que represente espaço não alocado de tamanho suficiente, ou verificamos todos os blocos. Se nenhum bloco adequado for encontrado, é hora de chamar novamente sbrk() para estender suficientemente o tamanho do heap. Para os propósitos desta classe, tentaremos atender todas as solicitações de memória até que o sistema operacional nos diga que ficaremos sem espaço no heap. Outros aplicativos podem limitar-se a um determinado tamanho de heap e fazer com que os pedidos falhem intermitentemente. Além disso, uma implementação rápida pode estender uma quantidade significativa para que não seja necessário solicitar mais memória heap no futuro próximo.
Quando um bloco livre é encontrado, ele pode ser maior que o espaço que precisamos. Em caso afirmativo, criaremos duas entradas em nossa lista implícita. A primeira entrada é o bloco alocado, a segunda entrada é o espaço restante. Existem maneiras de fazer isso se você quiser manter a sobrecarga pequena. Recomendamos primeiro para ir com legibilidade.
typedef struct { size_t block_size ; int is_free ; char data [0]; } block ; block * p = sbrk (100); p -> size = 100 - sizeof (* p) - sizeof (boundary_tag); // Other block allocations
Se você quiser que certos bits contenham diferentes informações, use campos de bits!
typedef struct { unsigned int block_size : 7 ; unsigned int is_free : 1 ; } size_free ; typedef struct { size_free info ; char data [0]; } block ;
O compilador irá lidar com isso para você. Depois de configurar seus campos, ele simplesmente passa por cada um dos blocos e verifica os campos apropriados
Aqui está uma representação visual do que acontece. Quanto a mais arte ascii, se assumirmos que temos um bloco que se parece com isso, queremos cuspir se a alocação for digamos 16 bytes
Check if this block is free |--| | meta | 40 bytes ---------------------------->|bt| |--|
Aqui está o que a pilha vai ficar depois
|------------------------|----------------------| | meta | 16 bytes ---->|bt| meta | 12 bytes -->|bt| |------------------------|----------------------|
Isso é antes das preocupações de alinhamento também.

Alinhamento e considerações de arredondamento

Muitas arquiteturas esperam que primitivos de múltiplos bytes sejam alinhados a um múltiplo de 2 (4, 16, etc.). Por exemplo, é comum exigir que os tipos de 4 bytes sejam alinhados aos limites de 4 bytes e aos tipos de 8 bytes nos limites de 8 bytes. Se as

primitivas de byte múltiplo não forem armazenadas em um limite razoável, por exemplo, iniciando em um endereço ímpar, o desempenho poderá ser significativamente impactado, pois pode exigir duas solicitações de leitura de memória, em vez de uma. Em algumas arquiteturas, a penalidade é ainda maior - o programa irá travar com um http://en.wikipedia.org/wiki/Bus_error#Unaligned_access . A maioria de vocês experimentou isso em suas classes de arquitetura, se não houvesse proteção de memória.
Como malloc não sabe como o usuário utilizará a memória alocada, o ponteiro retornado para o programa precisa ser alinhado para o pior caso, o que é dependente da arquitetura. Da documentação da glibc, o malloc da glibc usa a seguinte heurística ("Alocação de memória virtual e paginação" # ref-vma_paging)
O bloco que o malloc oferece é garantido para ser alinhado para que ele possa conter qualquer tipo de dado. Em sistemas GNU, o endereço é sempre um múltiplo de oito na maioria dos sistemas e um múltiplo de 16 em sistemas de 64 bits. ”Por exemplo, se você precisar calcular quantas unidades de 16 bytes são necessárias, não se esqueça de arredondar acima.
É assim que a matemática seria em C.
int s = (requested_bytes + tag_overhead_bytes + 15) / 16
A constante adicional garante que unidades incompletas sejam arredondadas. Note que o código real tem maior probabilidade de ter tamanhos de símbolos, por exemplo, sizeof(x) - 1 , em vez de uma constante numérica de codificação 15.
http://www.ibm.com/developerworks/library/pa-dalign/
Outro efeito adicional pode ser a fragmentação interna quando o bloco que você dá é maior do que o tamanho da alocação. Digamos que tenhamos um bloco livre de tamanho 16B (não incluindo metadados). Se eles alocarem 7 bytes, você pode querer arredondar até 16B e retornar o bloco inteiro. Isso fica muito sinistro quando você implementa coalescência e divisão. Se você não implementar, então você pode acabar retornando um bloco de tamanho 64B para uma alocação de 7B! Há *muita* sobrecarga para essa alocação, que é o que estamos tentando evitar.

Implementando gratuitamente

Quando o free é chamado, precisamos reaplicar o offset para voltar ao início 'real' do bloco - para onde armazenamos as informações de tamanho. Uma implementação ingênua seria simplesmente marcar o bloco como não utilizado. Se estivermos armazenando o status de alocação de bloco em um campo de bits, precisamos apenas limpar o bit:
p -> info . is_free = 0 ;
No entanto, temos um pouco mais de trabalho a fazer. Se o bloco atual e o bloco seguinte (se existirem) estiverem livres, precisamos aglutinar esses blocos em um único bloco. Da mesma forma, também precisamos verificar o bloco anterior. Se isso existe e representa uma memória não alocada, então precisamos unir os blocos em um único bloco grande. Para ser capaz de unir um bloco livre com um bloco livre anterior, também precisaremos encontrar o bloco anterior, então também armazenamos o tamanho do bloco no final do bloco. Essas são chamadas de “tags de limite” (Knuth # ref-knuth1973art). Estas são as soluções de Knuth para o problema da coalescência nos dois sentidos. Como os blocos são contíguos, o final de um bloco fica ao lado do início do próximo bloco. Assim, o bloco atual (além do primeiro) pode parecer alguns bytes mais atrás para procurar o tamanho do bloco anterior. Com esta informação você pode agora saltar para trás!

Tomemos por exemplo um duplo coalescente. Se quisermos liberar o bloco do meio, precisamos transformar os blocos ao redor em blocos grandes
----------------|----------------|------------|	free	space |bt| meta	space|bt| free |spc|bt
Torna-se um único bloco livre como abaixo
---|	free	space ---------------------------->|bt

atuação

Com a descrição acima, é possível construir um alocador de memória. Sua principal vantagem é a simplicidade - pelo menos simples em comparação com outros alocadores! A alocação de memória é uma operação de tempo linear no pior caso - procura listas encadeadas para um bloco livre suficientemente grande. A desalocação é tempo constante. Não mais que 3 blocos precisarão ser reunidos em um único bloco, e usando um esquema de blocos usado mais recentemente, apenas uma entrada de lista vinculada.
Usando esse alocador, é possível experimentar diferentes estratégias de posicionamento. Por exemplo, você pode começar a pesquisar desde o último bloco gratuito ou onde foi alocado pela última vez. Se você armazenar ponteiros em blocos, você precisa ter muito cuidado para que eles permaneçam sempre válidos. Particularmente quando você malloc, free, calloc, realloc, coalescer, split, etc.

Alocadores de listas livres explícitas

Melhor desempenho pode ser alcançado através da implementação de uma lista explícita duplamente ligada de nós livres. Nesse caso, podemos atravessar imediatamente para o próximo bloco livre e para o bloco livre anterior. Isso pode reduzir o tempo de pesquisa, porque a lista vinculada inclui apenas blocos não alocados. Uma segunda vantagem é que agora temos algum controle sobre a ordenação da lista encadeada. Por exemplo, quando um bloco é liberado, podemos optar por inseri-lo no início da lista encadeada em vez de sempre entre seus vizinhos. Podemos atualizar nossa estrutura para ficar assim
typedef struct { size_t info ; struct block * next ; char data [0]; } block ;
Aqui está como isso ficaria junto com nossa lista de links implícitos
Free list pointers above and below >--++ ++----------> \/ /\
|----------------|----------------|------------| | free | space |bt| meta | space|bt| free |spc|bt|
|----------------|----------------|------------| \/ /\ ++-------------------------------++
Mais uma vez desculpas pela arte ASCII, somos realmente programadores de sistemas e não pessoas de computação gráfica.
Onde armazenamos os ponteiros da nossa lista vinculada? Um truque simples é perceber que o bloco em si não está sendo usado e armazenar os ponteiros seguinte e anterior como parte do bloco, embora agora você tenha que garantir que os blocos livres sejam sempre suficientemente grandes para conter dois ponteiros. Ainda precisamos implementar Tags de limite, para que possamos liberar blocos corretamente e juntá-los com seus dois vizinhos. Consequentemente, listas livres explícitas requerem mais código e complexidade. Com listas vinculadas explícitas, é usado um algoritmo rápido e simples de 'Find-First' para encontrar o primeiro link suficientemente grande. No entanto, como a ordem dos links pode ser modificada, isso corresponde a diferentes estratégias de veiculação. Por exemplo, se os

links forem mantidos do maior para o menor, isso produzirá uma estratégia de posicionamento de "Pior ajuste".

Política de inserção de lista vinculada explícita

O bloco recém-liberado pode ser inserido facilmente em duas posições possíveis: no início ou na ordem de endereço. Inserir no início cria uma política LIFO (last-in, first-out). Os espaços free'd mais recentes serão reutilizados. Estudos sugerem que a fragmentação é pior do que usar a ordem de endereços (Wilson et al. # Ref-10.1007 / 3-540-60368-9_19). Inserir em ordem de endereço ("Política ordenada por endereço") insere blocos livres para que os blocos sejam visitados em ordem de endereço crescente. Essa política exigiu mais tempo para liberar um bloco, pois as tags de limite (dados de tamanho) devem ser usadas para localizar os blocos não alocados próximos e anteriores. No entanto, há menos fragmentação.

Estudo de caso: Buddy Allocator, um exemplo de uma lista segregada

Um alocador segregado é aquele que divide o heap em áreas diferentes que são manipuladas por subalocadores diferentes, dependendo do tamanho da solicitação de alocação. Os tamanhos são agrupados em potências de dois e cada tamanho é manipulado por um sub-alocador diferente e cada tamanho mantém sua própria lista livre.
Um alocador bem conhecido desse tipo é o alocador de amigos (Rangan, Raman e Ramanujam # ref-rangan1999foundations P. 85). Discutiremos o alocador de amigos binários que divide a alocação em blocos de tamanho
2^n;n=1,2,3,...
vezes algum número de unidade base de bytes, mas outros também existem como divisão de Fibonacci onde a alocação é arredondada para o próximo número de Fibonacci. O conceito básico é simples: se não houver blocos livres de tamanho
2^n
, vá para o próximo nível e roube esse bloco e divida-o em dois. Se dois blocos vizinhos do mesmo tamanho se tornarem não alocados, eles podem ser reunidos novamente em um único bloco grande com o dobro do tamanho.
Os alocadores de amigos são rápidos porque os blocos vizinhos a serem agrupados podem ser calculados a partir do endereço do bloco livre, em vez de percorrer as marcas de tamanho. O desempenho final geralmente requer uma pequena quantidade de código assembler para usar uma instrução de CPU especializada para encontrar o bit não-zero mais baixo.
A principal desvantagem do alocador Buddy é que eles sofrem *fragmentação interna* , porque as alocações são arredondadas para o tamanho de bloco mais próximo. Por exemplo, uma alocação de 68 bytes exigirá um bloco de 128 bytes.

Leitura adicional

Existem muitos outros esquemas de alocação. Um dos três alocadores usados internamente pelo kernel do Linux. Veja http://man7.org/linux/man-pages/man3/malloc.3.html !

- http://en.wikipedia.org/wiki/SLUB_%28software%29 (wikipedia)
- http://en.wikipedia.org/wiki/Buddy_memory_allocation

Tópicos

- Melhor ajuste
- Pior ajuste
- Primeiro ajuste
- Alocador de amigos
- Fragmentação Interna
- Fragmentação Externa
- sbrk
- Alinhamento Natural
- Tag de limite
- Coalescente
- Divisão
- Alocação de Slab / Pool de Memória

Perguntas / Exercícios

- O que é Fragmentação Interna? Quando isso se torna um problema?
- O que é Fragmentação Externa? Quando isso se torna um problema?
- Qual é a estratégia de posicionamento do Best Fit? Como é isso com fragmentação externa? Complexidade do Tempo?
- O que é uma estratégia de posicionamento de pior ajuste? É melhor com Fragmentação Externa? Complexidade do Tempo?
- Qual é a estratégia de colocação em primeiro lugar? É um pouco melhor com Fragmentação, certo? Complexidade prevista no tempo?
- Digamos que estamos usando um alocador de amigos com uma nova placa de 64kb. Como funciona a alocação de 1.5kb?
- Quando é que a implementação do malloc de 5 linhas tem um uso?
- O que é alinhamento natural?
- O que é Coalescing / Splitting? Como eles aumentam / diminuem a fragmentação? Quando você pode se unir ou dividir?
- Como as tags de limite funcionam? Como eles podem ser usados para coalescer ou dividir?

Garey, MR, RL Graham e JD Ullman. 1972. "Análise do pior caso de algoritmos de alocação de memória." In * Proceedings of the Fourth Acm Simpósio sobre Teoria da Computação *, 143-50. STOC '72. Nova York, NY, EUA: ACM. https://doi.org/10.1145/800152.804907 .
Jones, Larry. 2010. “Rascunho do comitê WG14 N1539 Iso / Iec 9899: 201x.” International Standards Organization.
Knuth, DE 1973. * A Arte da Programação de Computadores: Algoritmos Fundamentais *. Série Addison-Wesley em Ciência da Computação e Processamento de Informação, v. 1-2. Addison-Wesley. https://books.google.com/books?id=dC05RwAACAAJ .
“Visão geral de Malloc.” 2018. * MallocInternals - Glibc Wiki *. Fundação Software Livre. https://sourceware.org/glibc/wiki/MallocInternals .
Rangan, CP, V. Raman e R. Ramanujam. 1999. * Fundações de Tecnologia de Software e Ciência da Computação Teórica: 19ª Conferência, Chennai, Índia, 13 a 15 de dezembro de 1999 Procedimentos *. FOUNDATIONS of Computer software Technology and Theoretical Computer Science. Springer https://books.google.com/books?id=0uHME7EfjQEC .
“Virtual Memory Allocation and Paging.” 2001. *The GNU C Library - Virtual Memory Allocation and Paging*. Free Software Foundation. https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_3.html .
Wilson, Paul R., Mark S. Johnstone, Michael Neely, and David Boles. 1995. “Dynamic Storage Allocation: A Survey and Critical Review.” In *Memory Management*, edited by Henry G. Baler, 1–116. Berlin, Heidelberg: Springer Berlin Heidelberg.