14

Interlúdio: API de memória

Neste interlúdio, discutimos as interfaces de alocação de memória em UNIX sistemas. As interfaces fornecidas são bastante simples e, portanto, o capítulo é curto e direto ao ponto[1]. O principal problema que abordamos é este:

CRUX:HAITÓALOCALIZARADEMUMA IDADEMEMÓRIA
Em vocêNIXProgramas /C, entendendo como alocar e gerenciar a memória é crítica na construção de software robusto e confiável. Quais interfaces são comumente usadas? Que erros devem ser evitados?

14.1 Tipos de memória

Ao executar um programa C, existem dois tipos de memória alocadas. O primeiro é chamado**pilha**memória, e suas alocações e desalocações são gerenciadas*implicitamente*pelo compilador para você, o programador; por esse motivo às vezes é chamado**automático**memória.

Declarar memória na pilha em C é fácil. Por exemplo, digamos que você precise de algum espaço em uma funçãofunção()para um número inteiro, chamadox.Para declarar tal pedaço de memória, basta fazer algo assim:

```
função vazia() {
    interno x; //declara um inteiro na pilha. . .

}
```

O compilador faz o resto, certificando-se de liberar espaço na pilha quando você chamafunção().Quando você retorna da função, o compilador desaloca a memória para você; portanto, se você quiser que alguma informação permaneça além da invocação da chamada, é melhor não deixar essa informação na pilha.

[1]Na verdade, esperamos que todos os capítulos sejam! Mas este é mais curto e pontudo, pensamos.

É esta necessidade de memória de longa duração que nos leva ao segundo tipo de memória, chamada**amontoar**memória, onde todas as alocações e desalocações são *explicitamente*gerenciado por você, o programador. Uma responsabilidade pesada, sem dúvida! E certamente a causa de muitos bugs. Mas se você for cuidadoso e prestar atenção, você usará essas interfaces corretamente e sem muitos problemas. Aqui está um exemplo de como alguém pode alocar um número inteiro no heap:

```
função vazia() {
   int *x = (int *) malloc(tamanho(int)); . . .

}
```

Algumas notas sobre este pequeno trecho de código. Primeiro, você pode notar que a alocação de pilha e heap ocorre nesta linha: primeiro, o compilador sabe que deve abrir espaço para um ponteiro para um número inteiro quando vê sua declaração desse ponteiro (interno*x);posteriormente, quando o programa chamamaloc(),solicita espaço para um número inteiro no heap; a rotina retorna o endereço de tal número inteiro (em caso de sucesso, ouNULOem caso de falha), que é então armazenado na pilha para uso do programa.

Devido à sua natureza explícita e ao seu uso mais variado, a memória heap apresenta mais desafios para usuários e sistemas. Portanto, é o foco do restante de nossa discussão.

14.2 OMalloc()Chamar

O**Malloc()**chamada é bastante simples: você passa um tamanho solicitando algum espaço no heap, e ela é bem-sucedida e retorna um ponteiro para o espaço recém-alocado ou falha e retornaNULO[2].

A página de manual mostra o que você precisa fazer para usar o malloc; tipo homem malocna linha de comando e você verá:

```
#inclui <stdlib.h>
. . .
void *malloc(tamanho_t tamanho);
```

A partir dessas informações, você pode ver que tudo o que você precisa fazer é incluir o arquivo de cabeçalhostdlib.hpara usar malloc. Na verdade, você nem precisa fazer isso, já que a biblioteca C, à qual todos os programas C estão vinculados por padrão, tem o código paraMalloc()dentro dele; adicionar o cabeçalho apenas permite que o compilador verifique se você está chamandoMalloc()corretamente (por exemplo, passando o número certo de argumentos para ele, do tipo certo).

O único parâmetroMalloc()leva é do tipotamanho tque simplesmente descreve quantos bytes você precisa. Entretanto, a maioria dos programadores não digita um número aqui diretamente (como 10); na verdade, seria considerado falta de educação fazê-lo. Em vez disso, várias rotinas e macros são

[2]Observe queNULOem C não há nada de especial, apenas uma macro para o valor zero.

TPI: CGALINHAEUNDDÚVIDA, TRYEUTÓEUA

Se você não tem certeza de como alguma rotina ou operador que você está usando se comporta, não há substituto para simplesmente experimentá-lo e certificar-se de que ele se comporta conforme o esperado. Ao ler as páginas de manual ou outra documentação é útil, como funciona na prática é o que importa. Escreva algum código e teste-o! Essa é sem dúvida a melhor maneira de garantir que seu código se comporte conforme desejado. Na verdade, foi isso que fizemos para verificar novamente o que dizíamos sobretamanho de()eram realmente verdade!

utilizado. Por exemplo, para alocar espaço para um valor de ponto flutuante de precisão dupla, basta fazer o seguinte:

```
duplo *d = (duplo *) malloc(tamanho(duplo));
```

Uau, isso é muitodobro-eu! Esta invocação deMalloc()usa o tamanho de()operador solicite a quantidade certa de espaço; em C, isso geralmente é pensado como um*tempo de compilação*operador, o que significa que o tamanho real é conhecido em*tempo de compilação*e, portanto, um número (neste caso, 8, para um duplo) é substituído como argumento paramaloc().Por esta razão, tamanho de()é corretamente considerado um operador e não uma chamada de função (uma chamada de função ocorreria em tempo de execução).

Você também pode passar o nome de uma variável (e não apenas um tipo) para tamanho de(),mas em alguns casos você pode não obter os resultados desejados, então tome cuidado. Por exemplo, vejamos o seguinte trecho de código:

```
int *x = malloc(10 * tamanhode(int));
printf("%d\n",tamanho(x));
```

Na primeira linha, declaramos espaço para um array de 10 inteiros, o que é ótimo e elegante. Porém, quando usamostamanho de()na próxima linha, retorna um valor pequeno, como 4 (em máquinas de 32 bits) ou 8 (em máquinas de 64 bits). A razão é que neste caso,tamanho de()pensa que estamos simplesmente perguntando quão grande é*ponteiro*para um número inteiro não é a quantidade de memória alocada dinamicamente. No entanto, às vezestamanho de()funciona como você poderia esperar:

```
interno x[10];
printf("%d\n",tamanho(x));
```

Neste caso, há informação estática suficiente para o compilador saber que 40 bytes foram alocados.

Outro lugar para ter cuidado é com as cordas. Ao declarar espaço para uma string, use a seguinte expressão:malloc(strlen(s) + 1),que obtém o comprimento da string usando a funçãostrlen(),e adiciona 1 a ele para abrir espaço para o caractere de final de string. Usandotamanho de() pode causar problemas aqui.

Você também pode notar queMalloc()retorna um ponteiro para digitarvazio. Fazer isso é apenas a maneira em C de devolver um endereço e deixar o programador decidir o que fazer com ele. O programador ajuda ainda mais usando o que é chamado de**elenco**; em nosso exemplo acima, o programador lança o tipo de retorno deMalloc()para um ponteiro para umdobro.A conversão realmente não realiza nada, a não ser dizer ao compilador e a outros programadores que possam estar lendo seu código: “sim, eu sei o que estou fazendo”. Ao lançar o resultado demaloc(),o programador está apenas dando alguma garantia; o elenco não é necessário para a correção.

14.3 Olivre()Chamar

Acontece que alocar memória é a parte fácil da equação; saber quando, como e até mesmo se liberar memória é a parte difícil. Para liberar memória heap que não está mais em uso, os programadores simplesmente chamam**livre()**:

```
int *x = malloc(10 * tamanhode(int)); . . .

grátis(x);
```

A rotina recebe um argumento, um ponteiro retornado pormaloc(). Assim, você pode notar que o tamanho da região alocada não é passado pelo usuário e deve ser rastreado pela própria biblioteca de alocação de memória.

14.4 Erros Comuns

Existem vários erros comuns que surgem no uso deMalloc() elivre().Aqui estão alguns que vimos repetidamente no ensino do curso de graduação em sistemas operacionais. Todos esses exemplos são compilados e executados sem nenhum ruído do compilador; embora compilar um programa C seja necessário para construir um programa C correto, está longe de ser suficiente, como você aprenderá (muitas vezes da maneira mais difícil).

O gerenciamento correto de memória tem sido um problema tão grande que muitas linguagens mais recentes têm suporte para**gerenciamento automático de memória**. Nessas línguas, enquanto você chama algo semelhante aMalloc() para alocar memória (geralmente**novo**ou algo semelhante para alocar um novo objeto), você nunca precisa chamar algo para liberar espaço; em vez disso, um **coletor de lixo**é executado e descobre a qual memória você não tem mais referências e a libera para você.

Esquecendo de alocar memória

Muitas rotinas esperam que a memória seja alocada antes de serem chamadas. Por exemplo, a rotinastrcpy(dst,src)copia uma string de um ponteiro de origem para um ponteiro de destino. No entanto, se você não tomar cuidado, você pode fazer o seguinte:

```
char *src = "olá"; char *dst;
                                // opa! não alocado
strcpy(dst,src); // segfault e morre
```

TPI: EUTCOMPILADOÓREUTRUM6=EUTEUSCORRETO

Só porque um programa foi compilado (!) ou mesmo executado uma ou várias vezes corretamente, não significa que o programa esteja correto. Muitos eventos podem ter conspirado para levá-lo a um ponto em que você acredita que funciona, mas então alguns a coisa muda e para. Uma reação comum dos alunos é dizer (ou gritar) "Mas funcionou antes!" e então culpar o compilador, o sistema operacional, o hardware ou até mesmo (ousamos dizer) o professor. Mas o problema geralmente está exatamente onde você pensa que estaria, no seu código. Comece a trabalhar e depure-o antes de culpar os outros componentes.

Quando você executa este código, provavelmente levará a um**falha de segmentação**[3], que é um termo sofisticado para**VOCÊ FEZ ALGO ERRADO COM A MEMÓRIA SEU PROGRAMADOR TOLO E ESTOU IRRITADO.**

Nesse caso, o código adequado pode ser assim:

```
char *src = "olá";
char *dst = (char *) malloc(strlen(src) + 1); strcpy(dst,src); //
trabalhe corretamente
```

Alternativamente, você poderia usarstrdup()e tornar sua vida ainda mais fácil. Leia o fortepágina de manual para obter mais informações.

Não alocando memória suficiente

Um erro relacionado é não alocar memória suficiente, às vezes chamado de **estouro de buffer**. No exemplo acima, um erro comum é fazer*quase*espaço suficiente para o buffer de destino.

```
char *src = "olá";
char *dst = (char *) malloc(strlen(src)); // muito pequeno! strcpy(dst,src); //
trabalhe corretamente
```

Curiosamente, dependendo de como o malloc é implementado e de muitos outros detalhes, este programa muitas vezes será executado aparentemente corretamente. Em alguns casos, quando a cópia da string é executada, ela grava um byte além do final do espaço alocado, mas em alguns casos isso é inofensivo, talvez sobrescrevendo uma variável que não é mais usada. Em alguns casos, esses overflows podem ser extremamente prejudiciais e, na verdade, são a fonte de muitas vulnerabilidades de segurança nos sistemas [W06]. Em outros casos, a biblioteca malloc alocou um pouco de espaço extra de qualquer maneira e, portanto, seu programa na verdade não rabisca o valor de alguma outra variável e funciona muito bem. Mesmo em outros casos, o programa irá de fato falhar e travar. E assim aprendemos outra lição valiosa: mesmo que tenha funcionado corretamente uma vez, não significa que esteja correto.

[3]Embora pareça misterioso, você logo aprenderá por que esse acesso ilegal à memória é chamado de falha de segmentação; se isso não é um incentivo para continuar lendo, o que é?

Esquecendo de inicializar a memória alocada

Com este erro, você chamaMalloc()corretamente, mas esqueça de preencher alguns valores no tipo de dados recém-alocado. Não faça isso! Se você esquecer, seu programa eventualmente encontrará um**leitura não inicializada**, onde lê do heap alguns dados de valor desconhecido. Quem sabe o que pode estar lá? Se tiver sorte, algum valor para que o programa ainda funcione (por exemplo, zero). Se você não tiver sorte, algo aleatório e prejudicial.

Esquecendo de liberar memória

Outro erro comum é conhecido como**vazamento de memória**, e ocorre quando você se esquece de liberar memória. Em aplicativos ou sistemas de longa execução (como o próprio sistema operacional), isso é um grande problema, pois o vazamento lento de memória eventualmente leva à falta de memória, momento em que é necessário reiniciar. Assim, em geral, quando você terminar com um pedaço de memória, certifique-se de liberá-lo. Observe que usar uma linguagem com coleta de lixo não ajuda aqui: se você ainda tiver uma referência a algum pedaço de memória, nenhum coletor de lixo jamais irá liberá-lo e, portanto, vazamentos de memória continuam sendo um problema mesmo em linguagens mais modernas.

Em alguns casos, pode parecer que não está ligandolivre()é razoável. Por exemplo, seu programa tem vida curta e será encerrado em breve; neste caso, quando o processo terminar, o sistema operacional limpará todas as páginas alocadas e, portanto, nenhum vazamento de memória ocorrerá por si só. Embora isto certamente "funcione" (veja o aparte na página 7), é provavelmente um mau hábito a desenvolver, por isso tenha cuidado ao escolher tal estratégia. No longo prazo, um dos seus objetivos como programador é desenvolver bons hábitos; um desses hábitos é entender como você está gerenciando a memória e (em linguagens como C), liberando os blocos alocados. Mesmo que você consiga não fazer isso, provavelmente é bom adquirir o hábito de liberar cada byte que você aloca explicitamente.

Liberando memória antes de terminar

Às vezes, um programa libera memória antes de terminar de usá-lo; tal erro é chamado de**ponteiro pendurado**, e isso, como você pode imaginar, também é uma coisa ruim. O uso subsequente pode travar o programa ou sobrescrever a memória válida (por exemplo, você chamoulivre(),mas depois ligouMalloc()novamente para alocar outra coisa, que então recicla a memória liberada erroneamente).

Liberando memória repetidamente

Às vezes, os programas também liberam memória mais de uma vez; isso é conhecido como**duplo grátis**. O resultado disso é indefinido. Como você pode imaginar, a biblioteca de alocação de memória pode ficar confusa e fazer todo tipo de coisas estranhas; acidentes são um resultado comum.

ALADO: COINÓMEMÓRIAEUSEUCOMIDOÓNCESNOSSOPROCESSOESAÍDAS

Ao escrever um programa de curta duração, você pode alocar algum espaço usandomaloc().O programa é executado e está prestes a ser concluído: é necessário chamarlivre()várias vezes antes de sair? Embora pareça errado não fazê-lo, nenhuma memória será "perdida" em qualquer sentido real. A razão é simples: existem realmente dois níveis de gerenciamento de memória no sistema.

O primeiro nível de gerenciamento de memória é executado pelo sistema operacional, que distribui memória aos processos quando eles são executados e a recupera quando os processos terminam (ou morrem). O segundo nível de gestão é*dentro de*cada processo, por exemplo, dentro do heap quando você chama Malloc()elivre().Mesmo se você não ligarlivre() (e, portanto, vazar memória no heap), o sistema operacional irá recuperar*todos*a memória de o processo (incluindo as páginas de código, pilha e, conforme relevante aqui, heap) quando a execução do programa termina. Não importa qual seja o estado do seu heap no espaço de endereço, o sistema operacional recupera todas essas páginas quando o processo termina, garantindo assim que nenhuma memória seja perdida, apesar de você não tê-la liberado.

Assim, para programas de curta duração, o vazamento de memória geralmente não causa nenhum problema operacional (embora possa ser considerado uma forma inadequada). Quando você escreve um servidor de longa execução (como um servidor web ou sistema de gerenciamento de banco de dados, que nunca sai), o vazamento de memória é um problema muito maior e eventualmente levará a uma falha quando o aplicativo ficar sem memória. E, claro, o vazamento de memória é um problema ainda maior dentro de um programa específico: o próprio sistema operacional. Nos mostrando mais uma vez: quem escreve o código do kernel tem o trabalho mais difícil de todos...

Chamandolivre()Incorretamente

Um último problema que discutimos é o chamado delivre()incorretamente. Afinal, livre()espera que você apenas passe para ele uma das dicas que recebeu de Malloc()mais cedo. Quando você passa algum outro valor, coisas ruins podem (e acontecem). Assim, tal**liberações inválidas**são perigosos e, claro, também devem ser evitados.

Resumo

Como você pode ver, existem muitas maneiras de abusar da memória. Devido a erros frequentes de memória, toda uma ecosfera de ferramentas foi desenvolvida para ajudar a encontrar esses problemas em seu código. Confira ambos**purificar**[HJ92] e **valgrind**[SN05]; ambos são excelentes para ajudá-lo a localizar a origem dos seus problemas relacionados à memória. Depois de se acostumar a usar essas ferramentas poderosas, você se perguntará como sobreviveu sem elas.

14.5 Suporte de SO subjacente

Você deve ter notado que não falamos sobre chamadas de sistema ao discutir Malloc()elivre().A razão para isso é simples: não são chamadas de sistema, mas sim chamadas de biblioteca. Assim, a biblioteca malloc gerencia o espaço dentro do seu espaço de endereço virtual, mas ela mesma é construída sobre algumas chamadas do sistema que chamam o sistema operacional para solicitar mais memória ou liberar alguma de volta para o sistema.

Uma dessas chamadas de sistema é chamadacara,que é usado para alterar a localização do programa**quebrar**: a localização do final do heap. Ele recebe um argumento (o endereço da nova quebra) e, portanto, aumenta ou diminui o tamanho do heap com base no fato de a nova quebra ser maior ou menor que a quebra atual. Uma chamada adicionalsbrkrecebe um incremento, mas serve a um propósito semelhante.

Observe que você nunca deve ligar diretamente para nenhum dos doisfreioousbrk. Eles são usados pela biblioteca de alocação de memória; se você tentar usá-los, provavelmente fará com que algo dê (terrivelmente) errado. Atenha-seMalloc()e livre() em vez de.

Finalmente, você também pode obter memória do sistema operacional através do mmap()chamar. Ao passar os argumentos corretos,mmap()pode criar um **anônimo**região de memória dentro do seu programa - uma região que não está associada a nenhum arquivo específico, mas sim a**espaço de troca**, algo que discutiremos em detalhes mais tarde na memória virtual. Essa memória também pode ser tratada como um heap e gerenciada como tal. Leia a página de manual dommap()para mais detalhes.

14.6 Outras Chamadas

Existem algumas outras chamadas suportadas pela biblioteca de alocação de memória. Por exemplo,chamadaoc()aloca memória e também a zera antes de retornar; isso evita alguns erros onde você assume que a memória está zerada e esquece de inicializá-la (veja o parágrafo sobre “leituras não inicializadas” acima). A rotinarealocar()também pode ser útil quando você alocou espaço para algo (digamos, um array) e precisa adicionar algo a ele:realocar()cria uma nova região maior de memória, copia a região antiga nela e retorna o ponteiro para a nova região.

14.7 Resumo

Apresentamos algumas das APIs que tratam da alocação de memória. Como sempre, acabamos de abordar o básico; mais detalhes estão disponíveis em outro lugar. Leia o livro C [KR88] e Stevens [SR05] (Capítulo 7) para obter mais informações. Para um artigo moderno e interessante sobre como detectar e corrigir muitos desses problemas automaticamente, consulte Novark et al. [N+07]; este documento também contém um bom resumo de problemas comuns e algumas idéias interessantes sobre como localizá-los e corrigi-los.

Referências

[HJ92] "Purify: detecção rápida de vazamentos de memória e erros de acesso" por R. Hastings, B. Joyce. USENIX Inverno '92. *O papel por trás da ferramenta Purify, agora um produto comercial.*

[KR88] "A linguagem de programação C" por Brian Kernighan, Dennis Ritchie. Prentice-Hall 1988. *O livro C, dos desenvolvedores de C. Leia-o uma vez, programe um pouco, depois leia-o novamente e mantenha-o perto de sua mesa ou onde quer que você programe.*

[N+07] "Exterminador: Correção automática de erros de memória com alta probabilidade" por G. Novark, ED Berger, BG Zorn. PLDI 2007, San Diego, Califórnia. *Um artigo interessante sobre como encontrar e corrigir erros de memória automaticamente e uma excelente visão geral de muitos erros comuns em programas C e C++. Uma versão estendida deste documento está disponível no CACM (Volume 51, Edição 12, dezembro de 2008).*

[SN05] "Usando Valgrind para detectar erros de valores indefinidos com precisão de bits" por J. Seward, N. Nethercote. USÊNIX '05. *Como usar o valgrind para encontrar certos tipos de erros.*

[SR05] "Programação Avançada na UNIXMeio Ambiente" por W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *Já dissemos, diremos novamente: leia este livro muitas vezes e use-o como referência sempre que tiver dúvidas. Os autores sempre ficam surpresos ao ver como cada vez que leem algo neste livro, aprendem algo novo, mesmo depois de muitos anos de programação C.*

[W06] "Pesquisa sobre ataques e contramedidas de buffer overflow" por T. Werthman. Disponível: www.nds.rub.de/lehre/seminar/SS06/Werthmann BufferOverflow.pdf. *Uma boa pesquisa sobre buffer overflows e alguns dos problemas de segurança que eles causam. Refere-se a muitas das façanhas famosas.*

Lição de casa (código)

Neste trabalho de casa, você ganhará alguma familiaridade com a alocação de memória. Primeiro, você escreverá alguns programas com bugs (divertido!). Em seguida, você usará algumas ferramentas para ajudá-lo a encontrar os bugs inseridos. Então, você perceberá o quão incríveis são essas ferramentas e as utilizará no futuro, tornando-se mais feliz e produtivo. As ferramentas são o depurador (por exemplo, gdb) e um detector de bug de memória chamadovalgrind [SN05].

Questões

1. Primeiro, escreva um programa simples chamadonulo.cque cria um ponteiro para um número inteiro, define-o comoNULO,e então tenta desreferencia-lo. Compile isso em um executável chamadonulo.O que acontece quando você executa este programa?

2. Em seguida, compile este programa com informações de símbolo incluídas (com o -gflag). Fazendo isso, vamos colocar mais informações no executável, permitindo que o depurador acesse informações mais úteis sobre nomes de variáveis e coisas do gênero. Execute o programa no depurador digitandogdb nuloe então, uma vezgdbestá correndo, digitando correr.O que o gdb mostra para você?

3. Finalmente, use ovalgrindferramenta neste programa. Usaremos overificação de memória ferramenta que faz partevalgrindpara analisar o que acontece. Execute isso digitando o seguinte:valgrind --leak-check=sim nulo.O que acontece quando você executa isso? Você pode interpretar a saída da ferramenta?

4. Escreva um programa simples que aloque memória usandoMalloc()mas esquece de liberá-lo antes de sair. O que acontece quando este programa é executado? Você pode usargdbencontrar algum problema com isso? Que tal valgrind (novamente com o -verificação de vazamento = sim flag)?

5. Escreva um programa que crie um array de inteiros chamadodadosdo tamanho 100 usandomaloc;então, definadados[100]para zero. O que acontece quando você executa este programa? O que acontece quando você executa este programa usandovalgrind?O programa está correto?

6. Crie um programa que aloque um array de inteiros (como acima), os libere e então tente imprimir o valor de um dos elementos do array. O programa é executado? O que acontece quando você usa valgrindnele?

7. Agora passe um valor engraçado para free (por exemplo, um ponteiro no meio do array que você alocou acima). O que acontece? Você precisa de ferramentas para encontrar esse tipo de problema?

8. Experimente algumas das outras interfaces para alocação de memória. Por exemplo, crie uma estrutura de dados simples semelhante a um vetor e rotinas relacionadas que usamrealocar()para gerenciar o vetor. Use um array para armazenar os elementos dos vetores; quando um usuário adiciona uma entrada ao vetor, userealocar()para alocar mais espaço para isso. Qual é o desempenho desse vetor? Como isso se compara a uma lista vinculada? Usar valgrindpara ajudá-lo a encontrar bugs.

9. Passe mais tempo lendo sobre como usargdbevalgrind.Conhecer suas ferramentas é fundamental; gaste o tempo e aprenda como se tornar um depurador especialista nos EUANIXe ambiente C.