13

A Abstração: Espaços de Endereço

Nos primeiros dias, construir sistemas de computador era fácil. Porque você pergunta? Porque os usuários não esperavam muito. São esses malditos usuários com suas expectativas de "facilidade de uso", "alto desempenho", "confiabilidade", etc., que realmente causaram todas essas dores de cabeça. Da próxima vez que você encontrar um desses usuários de computador, agradeça-lhe por todos os problemas que causou.

13.1 Sistemas Iniciais

Do ponto de vista da memória, as primeiras máquinas não forneciam muita abstração aos usuários. Basicamente, a memória física da máquina se parecia com o que você vê na Figura 13.1 (página 2).

O sistema operacional era um conjunto de rotinas (uma biblioteca, na verdade) que ficavam na memória (começando no endereço físico 0 neste exemplo), e haveria um programa em execução (um processo) que atualmente ficava na memória física (começando no endereço físico 64k neste exemplo) e usou o restante da memória. Havia poucas ilusões aqui e o usuário não esperava muito do sistema operacional. A vida era fácil para os desenvolvedores de sistemas operacionais naquela época, não era?

13.2 Multiprogramação e compartilhamento de tempo

Depois de um tempo, como as máquinas eram caras, as pessoas começaram a compartilhá-las de forma mais eficaz. Assim a era do**multiprogramação**nasceu [DV66], no qual vários processos estavam prontos para serem executados em um determinado momento, e o sistema operacional alternava entre eles, por exemplo, quando alguém decidia realizar uma E/S. Fazer isso aumentou a eficácia **utilização**da CPU. Tais aumentos em**eficiência**eram particularmente importantes naquela época em que cada máquina custava centenas de milhares ou até milhões de dólares (e você achava que seu Mac era caro!).

Logo, porém, as pessoas começaram a exigir mais das máquinas, e a era da **compartilhamento de tempo**nasceu [S59, L60, M62, M83]. Especificamente, muitos perceberam as limitações da computação em lote, particularmente nos próprios programadores [CV65], que estavam cansados de longos (e, portanto, ineficazes)

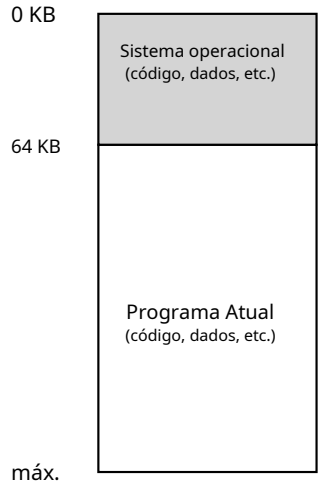

Figura 13.1:**Sistemas operacionais: os primeiros dias**

ativo) ciclos de depuração de programa. A noção de**interatividade**tornou-se importante, pois muitos usuários podem estar usando uma máquina simultaneamente, cada um aguardando (ou esperando) uma resposta oportuna de suas tarefas em execução no momento.

Uma maneira de implementar o compartilhamento de tempo seria executar um processo por um breve período, dando-lhe acesso total a toda a memória (Figura 13.1), depois interrompê-lo e salvar todo o seu estado em algum tipo de disco (incluindo toda a memória física). , carregue o estado de algum outro processo, execute-o por um tempo e, assim, implemente algum tipo de compartilhamento bruto da máquina [M+63].

Infelizmente, esta abordagem tem um grande problema: é muito lenta, especialmente à medida que a memória aumenta. Embora salvar e restaurar o estado do nível de registro (o PC, registros de uso geral, etc.) seja relativamente rápido, salvar todo o conteúdo da memória no disco é brutalmente ineficiente. Assim, o que preferimos fazer é deixar os processos na memória enquanto alternamos entre eles, permitindo que o sistema operacional implemente o compartilhamento de tempo de forma eficiente (como mostrado na Figura 13.2, página 3).

No diagrama, existem três processos (A, B e C) e cada um deles possui uma pequena parte da memória física de 512 KB reservada para eles. Assumindo uma única CPU, o sistema operacional escolhe executar um dos processos (digamos A), enquanto os outros (B e C) ficam na fila de prontidão aguardando para serem executados.

À medida que o compartilhamento de tempo se tornou mais popular, você provavelmente pode adivinhar que novas demandas foram impostas ao sistema operacional. Em particular, permitir que vários programas residam simultaneamente na memória torna**proteção**uma questão importante; você não quer que um processo seja capaz de ler, ou pior, escrever na memória de algum outro processo.

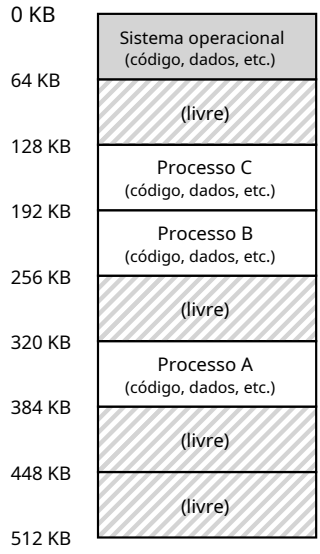

Figura 13.2:**Três Processos: Compartilhando Memória**

13.3 O Espaço de Endereço

No entanto, temos que manter esses usuários incômodos em mente, e isso exige que o sistema operacional crie um**fácil de usar**abstração da memória física. Chamamos essa abstração de**espaço de endereço**, e é a visão da memória do programa em execução no sistema. Compreender essa abstração fundamental da memória do sistema operacional é fundamental para entender como a memória é virtualizada.

O espaço de endereço de um processo contém todo o estado da memória do programa em execução. Por exemplo, o**código**do programa (as instruções) precisam estar em algum lugar da memória e, portanto, estão no espaço de endereço. O programa, enquanto está em execução, usa um**pilha**para acompanhar onde ela está na cadeia de chamadas de função, bem como para alocar variáveis locais e passar parâmetros e retornar valores de e para rotinas. finalmente, o **amontoar**é usado para memória alocada dinamicamente e gerenciada pelo usuário, como a que você pode receber de uma chamada paraMalloc()em C ounovoem uma linguagem orientada a objetos como C++ ou Java. Claro, há outras coisas lá também (por exemplo, variáveis inicializadas estaticamente), mas por enquanto vamos apenas assumir esses três componentes: código, pilha e heap.

No exemplo da Figura 13.3 (página 4), temos um pequeno espaço de endereço (apenas 16 KB)[1]. O código do programa fica no topo do espaço de endereço

[1]Freqüentemente usaremos pequenos exemplos como este porque (a) é difícil representar um espaço de endereço de 32 bits e (b) a matemática é mais difícil. Gostamos de matemática simples.

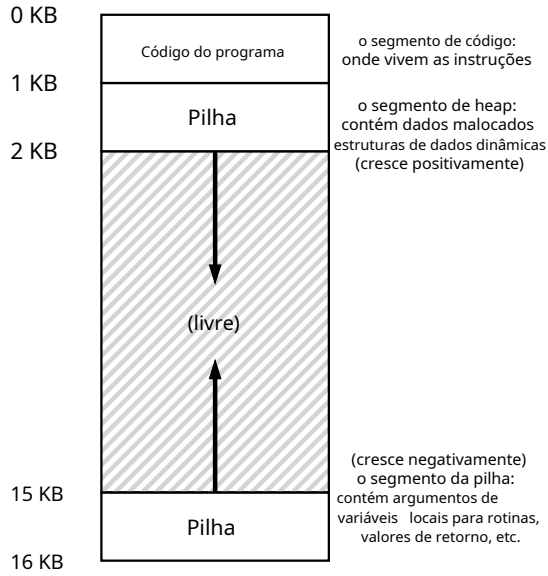

Figura 13.3:**Um exemplo de espaço de endereço**

(começando em 0 neste exemplo e é compactado no primeiro 1K do espaço de endereço). O código é estático (e, portanto, fácil de colocar na memória), então podemos colocá-lo no topo do espaço de endereço e saber que não precisará de mais espaço enquanto o programa for executado.

A seguir, temos as duas regiões do espaço de endereço que podem crescer (e diminuir) enquanto o programa é executado. Esses são o heap (no topo) e a pilha (no fundo). Nós os colocamos assim porque cada um deseja poder crescer, e ao colocá-los em extremos opostos do espaço de endereço, podemos permitir esse crescimento: eles apenas precisam crescer em direções opostas. O heap, portanto, começa logo após o código (em 1 KB) e cresce para baixo (digamos, quando um usuário solicita mais memória viamaloc());a pilha começa em 16 KB e cresce (digamos, quando um usuário faz uma chamada de procedimento). Entretanto, esse posicionamento de pilha e heap é apenas uma convenção; você pode organizar o espaço de endereço de uma maneira diferente, se desejar (como veremos mais tarde, quando vários**tópicos**coexistir em um espaço de endereço, infelizmente não há mais uma maneira legal de dividir o espaço de endereço assim).

É claro que, quando descrevemos o espaço de endereçamento, o que estamos descrevendo é o**abstração**que o sistema operacional está fornecendo ao programa em execução. O programa realmente não está na memória nos endereços físicos de 0 a 16 KB; em vez disso, ele é carregado em alguns endereços físicos arbitrários. Examine os processos A, B e C na Figura 13.2; lá você pode ver como cada processo é carregado na memória em um endereço diferente. E daí o problema:

TELECRUX:HAITÓVIRTUALIZARMEMÓRIA

Como o sistema operacional pode construir essa abstração de um ambiente privado e potencialmente grande?

espaço de endereço para vários processos em execução (todos compartilhando memória) em cima de uma única memória física?

Quando o sistema operacional faz isso, dizemos que o sistema operacional está**virtualizando memória**, porque o programa em execução pensa que está carregado na memória em um endereço específico (digamos 0) e tem um espaço de endereço potencialmente muito grande (digamos 32 bits ou 64 bits); A realidade é um pouco diferente.

Quando, por exemplo, o processo A da Figura 13.2 tenta realizar uma carga no endereço 0 (que chamaremos de**endereço virtual**), de alguma forma, o sistema operacional, em conjunto com algum suporte de hardware, terá que garantir que a carga não vá realmente para o endereço físico 0, mas sim para o endereço físico 320 KB (onde A é carregado na memória). Esta é a chave para a virtualização da memória, subjacente a todos os sistemas de computadores modernos do mundo.

13.4 Metas

Chegamos assim à função do SO neste conjunto de notas: virtualizar a memória. O sistema operacional não apenas virtualizará a memória; fará isso com estilo. Para garantir que o sistema operacional faça isso, precisamos de alguns objetivos para nos guiar. Já vimos esses objetivos antes (pense na Introdução) e os veremos novamente, mas certamente vale a pena repeti-los.

Um objetivo principal de um sistema de memória virtual (VM) é**transparência**[2]. O sistema operacional deve implementar a memória virtual de uma forma invisível para o programa em execução. Assim, o programa não deve estar ciente do fato de que a memória é virtualizada; em vez disso, o programa se comporta como se tivesse sua própria memória física privada. Nos bastidores, o sistema operacional (e o hardware) faz todo o trabalho para multiplexar a memória entre muitas tarefas diferentes e, portanto, implementa a ilusão.

Outro objetivo da VM é**eficiência**. O sistema operacional deve se esforçar para tornar a virtualização o mais**eficiente**possível, tanto em termos de tempo (ou seja, não fazer os programas rodarem muito mais lentamente) quanto de espaço (ou seja, não usar muita memória para estruturas necessárias para suportar a virtualização). Ao implementar a virtualização com eficiência de tempo, o sistema operacional terá que contar com suporte de hardware, incluindo recursos de hardware como TLBs (sobre os quais aprenderemos no devido tempo).

Finalmente, um terceiro objetivo da VM é**proteção**. O sistema operacional deve certificar-se de **proteger**processos uns dos outros, bem como o próprio sistema operacional de programas

[2]Este uso da transparência às vezes é confuso; alguns estudantes pensam que "ser transparente" significa manter tudo aberto, ou seja, como deveria ser o governo. Aqui, significa o contrário: que a ilusão fornecida pelo sistema operacional não deve ser visível para os aplicativos. Assim, no uso comum, um sistema transparente é aquele que é difícil de perceber, e não aquele que responde às solicitações estipuladas pela Lei de Liberdade de Informação.

TPI:TELEPPRINCÍPIOÓFEUSOLAÇÃO

O isolamento é um princípio fundamental na construção de sistemas confiáveis. Se duas entidades estiverem devidamente isoladas uma da outra, isto implica que uma pode falhar sem afectar a outra. Os sistemas operacionais se esforçam para isolar os processos uns dos outros e, dessa forma, evitar que um prejudique o outro. Usando isolamento de memória, o sistema operacional garante ainda que os programas em execução não possam afetar a operação do sistema operacional subjacente. Alguns sistemas operacionais modernos levam o isolamento ainda mais longe, isolando partes do sistema operacional de outras partes do sistema operacional. Tal**micronúcleos**[BH70, R+89, S+03] podem, portanto, fornecer maior confiabilidade do que projetos típicos de kernel monolítico.

cessações. Quando um processo executa um carregamento, um armazenamento ou uma busca de instrução, ele não deve ser capaz de acessar ou afetar de forma alguma o conteúdo da memória de qualquer outro processo ou do próprio sistema operacional (ou seja, qualquer coisa*fora*seu espaço de endereço). A proteção permite-nos assim entregar a propriedade de **isolamento** entre processos; cada processo deve ser executado em seu próprio casulo isolado, protegido da devastação de outros processos defeituosos ou mesmo maliciosos.

Nos próximos capítulos, concentraremos nossa exploração nos aspectos básicos **mecanismos**necessários para virtualizar a memória, incluindo suporte a hardware e sistemas operacionais. Também investigaremos alguns dos mais relevantes**políticas** que você encontrará em sistemas operacionais, incluindo como gerenciar o espaço livre e quais páginas retirar da memória quando houver pouco espaço. Ao fazer isso, aumentaremos sua compreensão de como um moderno

sistema de memória virtual realmente funciona[3].

13.5 Resumo

Vimos a introdução de um importante subsistema de sistema operacional: a memória virtual. O sistema VM é responsável por fornecer a ilusão de um espaço de endereço privado grande, esparso para cada programa em execução; cada espaço de endereço virtual contém todas as instruções e dados de um programa, que podem ser referenciados pelo programa por meio de endereços virtuais. O sistema operacional, com muita ajuda de hardware, pegará cada uma dessas referências de memória virtual e as transformará em endereços físicos, que podem ser apresentados à memória física para buscar ou atualizar as informações desejadas. O sistema operacional fornecerá esse serviço para muitos processos ao mesmo tempo, protegendo os programas uns dos outros, bem como protegendo o sistema operacional. Toda a abordagem requer uma grande quantidade de mecanismos (isto é, muitas máquinas de baixo nível), bem como algumas políticas críticas para funcionar; começaremos de baixo para cima, descrevendo primeiro os mecanismos críticos. E assim prosseguimos!

[3]Ou vamos convencê-lo a abandonar o curso. Mas espere; se você passar pela VM, provavelmente conseguirá chegar até o fim!

ALADO: EMUITOADDRESSSUOSEEEUSVIRTUAIS

Você já escreveu um programa C que imprime um ponteiro? O valor que você vê (algum número grande, geralmente impresso em hexadecimal), é um**endereço virtual**. Você já se perguntou onde o código do seu programa é encontrado? Você também pode imprimir e, sim, se puder imprimi-lo, também é um endereço virtual. Na verdade, qualquer endereço que você possa ver como programador de um programa de nível de usuário é um endereço virtual. É apenas o sistema operacional, por meio de suas técnicas complicadas de virtualização de memória, que sabe onde estão essas instruções e valores de dados na memória física da máquina. Portanto, nunca se esqueça: se você imprimir um endereço em um programa, trata-se de um endereço virtual, uma ilusão de como as coisas estão dispostas na memória; apenas o sistema operacional (e o hardware) conhece a verdade.

Aqui está um pequeno programa (va.c)que imprime as localizações dosprincipal() rotina (onde o código reside), o valor de um valor alocado no heap retornado de maloc(),e a localização de um número inteiro na pilha:

```
1 #inclui <stdio.h>
2 #inclui <stdlib.h>
3 int principal(int argc, char *argv[]) {
4     printf("localização do código: %p\n", principal); printf("localização
5     da pilha: %p\n", malloc(100e6)); int x = 3;
6
7     printf("localização da pilha: %p\n", &x); retornar x;
8
9 }
```

Quando executado em um Mac de 64 bits, obtemos a seguinte saída:

```
localização do código: 0x1095afe50
localização da pilha: 0x1096008c0 localização
da pilha: 0x7fff691aea64
```

A partir disso, você pode ver que o código vem primeiro no espaço de endereço, depois no heap, e a pilha fica na outra extremidade desse grande espaço virtual. Todos esses endereços são virtuais e serão traduzidos pelo sistema operacional e pelo hardware para buscar valores de suas verdadeiras localizações físicas.

Referências

[BH70] "O Núcleo de um Sistema Multiprogramação" por Per Brinch Hansen. Comunicações da ACM, 13:4, abril de 1970. *O primeiro artigo a sugerir que o sistema operacional, ou kernel, deveria ser um substrato mínimo e flexível para a construção de sistemas operacionais customizados; este tema é revisitado ao longo da história da pesquisa em OS.*

[CV65] "Introdução e Visão Geral do Sistema Multics" por FJ Corbato, VA Vyssotsky. Conferência Conjunta de Computação de Outono, 1965. *Um ótimo artigo inicial do Multics. Aqui está a grande citação sobre o compartilhamento de tempo: "O ímpeto para o compartilhamento de tempo surgiu primeiro dos programadores profissionais devido à sua constante frustração na depuração de programas em instalações de processamento em lote. Assim, o objetivo original era compartilhar computadores para permitir o acesso simultâneo de várias pessoas, dando a cada uma delas a ilusão de ter toda a máquina à sua disposição."*

[DV66] "Semântica de Programação para Computações Multiprogramadas" por Jack B. Dennis, Earl C. Van Horn. Comunicações da ACM, Volume 9, Número 3, março de 1966. *Um artigo inicial (mas não o primeiro) sobre multiprogramação.*

[L60] "Simbiose Homem-Computador" por JCR Licklider. Transações IRE sobre Fatores Humanos em Eletrônica, HFE-1:1, março de 1960. *Um artigo interessante sobre como os computadores e as pessoas entrarão em uma era simbiótica; claramente bem à frente de seu tempo, mas mesmo assim é uma leitura fascinante.*

[M62] "Sistemas de Computação de Compartilhamento de Tempo" por J. McCarthy. Gestão e o Computador do Futuro, MIT Press, Cambridge, MA, 1962. *Provavelmente o primeiro artigo registrado de McCarthy sobre compartilhamento de tempo. Em outro artigo [M83], ele afirma estar pensando na ideia desde 1957. McCarthy deixou a área de sistemas e se tornou um gigante em Inteligência Artificial em Stanford, incluindo a criação da linguagem de programação LISP. Veja a página inicial de McCarthy para mais informações: http://www-formal.stanford.edu/jmc/*

[M+63] "Um sistema de depuração de compartilhamento de tempo para um computador pequeno" por J. McCarthy, S. Boilen, E. Fredkin, JCR Licklider. AFIPS '63 (primavera), Nova York, NY, maio de 1963. *Um ótimo exemplo inicial de um sistema que trocava a memória do programa para o "tambor" quando o programa não estava em execução e depois de volta para a memória "central" quando estava prestes a ser executado.*

[M83] "Reminiscências sobre a história do tempo compartilhado", de John McCarthy. 1983. Disponível: http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html. *Uma excelente nota histórica sobre a origem da ideia de compartilhamento de tempo, incluindo algumas dúvidas para aqueles que citam o trabalho de Strachey [S59] como o trabalho pioneiro nesta área.*

[NS07] "Valgrind: Uma Estrutura para Instrumentação Binária Dinâmica Pesada" por N. Nethercote, J. Seward. PLDI 2007, San Diego, Califórnia, junho de 2007. *Valgrind é um programa que salva vidas para aqueles que usam linguagens inseguras como C. Leia este artigo para aprender sobre suas técnicas de instrumentação binária muito legais – é realmente impressionante.*

[R+89] "Mach: Um kernel de software de sistema" por R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones. COMPCON '89, fevereiro de 1989. *Embora não seja o primeiro projeto em micronúcleos em si, o projeto Mach na CMU era bem conhecido e influente; ainda hoje vive nas entranhas do Mac OS X.*

[S59] "Compartilhamento de tempo em computadores grandes e rápidos", por C. Strachey. Anais da Conferência Internacional sobre Processamento de Informação, UNESCO, junho de 1959. *Uma das primeiras referências sobre compartilhamento de tempo.*

[S+03] "Melhorando a confiabilidade dos sistemas operacionais de commodities" por MM Swift, BN Bershad, HM Levy. SOSP '03. *O primeiro artigo a mostrar como o pensamento semelhante ao microkernel pode melhorar a confiabilidade do sistema operacional.*

Lição de casa (código)

Neste trabalho de casa, aprenderemos apenas algumas ferramentas úteis para examinar o uso de memória virtual em sistemas baseados em Linux. Isto será apenas uma breve sugestão do que é possível; você terá que se aprofundar por conta própria para realmente se tornar um especialista (como sempre!).

Questões

1. A primeira ferramenta Linux que você deve verificar é a ferramenta muito simples livre.Primeiro, digitehomem livree leia toda a página do manual; é curto, não se preocupe!
2. Agora, corralivre,talvez usando alguns dos argumentos que podem ser úteis (por exemplo, -m,para exibir totais de memória em megabytes). Quanta memória há no seu sistema? Quanto é grátis? Esses números correspondem à sua intuição?
3. A seguir, crie um pequeno programa que utilize uma certa quantidade de memória, chamadousuário de memória.c.Este programa deve receber um argumento de linha de comando: o número de megabytes de memória que utilizará. Quando executado, ele deve alocar um array e transmitir constantemente através do array, tocando em cada entrada. O programa deve fazer isso indefinidamente ou, talvez, por um determinado período de tempo também especificado na linha de comando.
4. Agora, enquanto executa seuusuário de memóriaprograma, também (em uma janela de terminal diferente, mas na mesma máquina) execute olivre ferramenta. Como os totais de uso de memória mudam quando o programa está em execução? E quando você mata ousuário de memóriaprograma? Os números correspondem às suas expectativas? Tente isso para diferentes quantidades de uso de memória. O que acontece quando você usa grandes quantidades de memória?
5. Vamos tentar mais uma ferramenta, conhecida comopmap.Passe algum tempo e leia o mapapágina de manual em detalhes.
6. Para usarpmap,você tem que saber o**ID do processo**do processo no qual você está interessado. Portanto, primeiro executeps auxwpara ver uma lista de todos os processos; em seguida, escolha um interessante, como um navegador. Você também pode usar seuusuário de memóriaprograma neste caso (na verdade, você pode até fazer com que esse programa chamegetpid()e imprima seu PID para sua conveniência).
7. Agora corramapaem alguns desses processos, usando vários sinalizadores (como -X)para revelar muitos detalhes sobre o processo. O que você vê? Quantas entidades diferentes constituem um espaço de endereço moderno, em oposição à nossa concepção simples de código/pilha/heap?
8. Finalmente, vamos corrermapaNas suasusuário de memóriaprograma, com diferentes quantidades de memória usada. O que você vê aqui? A saída de mapacorresponde às suas expectativas?