xv6

um Unix-like sistema operacional simples, ensino

Russ Cox
Frans Kaashoek
Robert Morris

xv6-book@pdos.csail.mit.edu

Projecto a partir de 28 de agosto de 2012

Conteúdo

Page 5

Prefácio e agradecimentos

Este é um projecto de texto destinado a uma classe de sistemas operacionais. Ele explica o con- principal conceitos de sistemas operacionais, estudando um kernel de exemplo, xv6 nomeado. xv6 é uma re-implementação de Dennis Ritchie e Ken Thompson Unix da versão 6 (v6). xv6 loosely segue a estrutura e estilo de v6, mas é implementado em ANSI C para um x86-multiprocessador baseado.

O texto deve ser lido juntamente com o código fonte para xv6. Esta abordagem é inspirada por Comentário de John Lions on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1ª edição (14 de junho de 2000)). Veja http://pdos.csail.mit.edu/6.828 para ponteiros para recursos on-line para v6 e xv6.

Nós temos usado este texto em 6,828, a classe de sistema operacional no MIT. Agradecemos ao faculdade, TAS, e estudantes de 6,828 que contribuíram direta ou indiretamente para xv6. Em particular, gostaríamos de agradecer a Austin Clements e Nickolai Zeldovich.

Página 7

Capítulo 0

design de interface
núcleo
processo
chamadas de sistema
espaço do usuário
espaço de kernel

Interfaces do sistema operacional

O trabalho de um sistema operacional é compartilhar um computador entre vários programas e para fornecer um conjunto de serviços mais úteis do que os suportes de hardware sozinho. O sistema operacional gerencia e abstrai o hardware de baixo nível, de modo que, por exemplo, um processador de texto não precisa preocupar-se com que tipo de hardware do disco está sendo utilizado. Ele também multiplexes o hardware, permitindo que muitos programas para compartilhar o computador e executar (ou parecem ser executados) ao mesmo tempo. Finalmente, os sistemas operacionais fornecem con- maneiras controlada para programas para interagir, para que eles possam compartilhar dados ou trabalhar em conjunto.

Um sistema operacional fornece serviços para os programas do usuário através de uma interface. Projetando uma boa interface acaba por ser difícil. Por um lado, gostaríamos a interface para ser simples e estreita, porque isso faz com que seja mais fácil obter a imple- mentação direita. Por outro lado, podemos ser tentados a oferecer muitos sofisticado recursos para as aplicações. O truque para resolver esta tensão é projetar interfaces que contar com alguns mecanismos que podem ser combinadas para fornecer tanto generalidade.

Este livro utiliza um único sistema operacional como um exemplo concreto para ilustrar ope- conceitos de sistemas ating. Esse sistema operacional, xv6, fornece as interfaces básicas intro- produzida por sistema operacional Unix de Dennis Ritchie e Ken Thompson, bem como mim- design interno do Icking Unix. Unix fornece uma interface estreita cujos mecanismos combinam bem, oferecendo um surpreendente grau de generalidade. Essa interface foi tão sucesso que operacional moderno sistemas-BSD, Linux, Mac OS X, Solaris, e até mesmo, em menor escala, o Microsoft Windows-têm interfaces Unix. Entendimento xv6 é um bom começo para a compreensão de qualquer um desses sistemas e muitos outros.

Como mostrado na F igura 0-1, xv6 toma a forma tradicional de um kernel, um pro- especial grama que fornece serviços para os programas em execução. Cada programa em execução, chamado de processo, tem de memória contendo instruções, dados, e uma pilha. As instruções im- plementar computação do programa. Os dados são as variáveis em que o computacional ção actua. A pilha organiza chamadas de procedimento do programa.

Quando um processo precisa chamar um serviço de kernel, ele invoca uma chamada de procedimento em a interface de sistema operativo. Tais procedimentos são chamadas de sistema de chamada. O sistema

chamada entra o kernel; o kernel executa o serviço e retorna. Assim, um processo alternates entre a execução no espaço do usuário e espaço de kernel.

O kernel usa mecanismos de proteção de hardware da CPU para garantir que cada processo de execução no espaço do usuário pode acessar apenas a sua própria memória. Os executado no kernel com os privilégios de hardware necessários para implementar essas proteções; programas de usuário executar sem esses privilégios. Quando um programa usuário invoca uma chamada de sistema, o hardware eleva o nível de privilégio e começa a executar uma função pré-estabelecido no kernel.

A coleção de chamadas de sistema que fornece um kernel é a interface que pro- usuário gramas ver. O kernel xv6 proporciona um subconjunto dos serviços e chamadas de sistema que o Unix kernels tradicionalmente oferecer. As chamadas são:

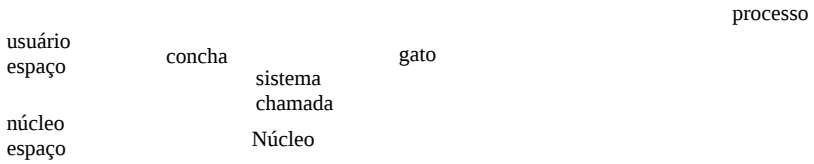


Figura 0-1. Um kernel e dois processos de usuário.

Chamada de sistema	Descrição
fork ()	Criar processo
saída ()	Finaliza o processo atual
wait ()	Espere por um processo filho para sair
matar (PID)	Finaliza o processo pid
getpid ()	Retorno id do processo atual
sono (n)	Dormir por n segundos
exec (filename, * argv)	Carregue um arquivo e executá-lo
sbrk (n)	Cresça a memória do processo por n bytes
open (filename, bandeiras)	Abra um arquivo; sinalizadores indicam leitura / gravação
read (fd, buf, n)	Leia n despedidas de um arquivo aberto em buf
escrever (fd, buf, n)	Escrever n bytes em um arquivo aberto
close (fd)	Solte aberto fd arquivo
dup (FD)	Duplicar fd
pipe (p)	Criar um tubo e retornar fd de em p
chdir (dirname)	Altere o diretório atual
mkdir (dirname)	Crie um novo diretório
mknod (nome, major, minor)	Criar um arquivo de dispositivo
fstat (FD)	Retornar informações sobre um arquivo aberto
ligação (f1, f2)	Criar um outro nome (F2) para o arquivo f1
desvincular (filename)	Remover um arquivo

O restante deste capítulo descreve serviços de processos de xv6, memória descritiva arquivo res, cachimbos e sistema de arquivo e ilustra-los com trechos de código e discussões de como o shell usa-los. O uso do shell de chamadas do sistema ilustra quão cuidadosamente eles foram concebidos.

O shell é um programa comum que lê comandos do usuário e execútis-los, e é a principal interface do usuário para sistemas Unix-like tradicionais. O fato que o shell é um programa do usuário, não faz parte do kernel, ilustra o poder do sisinterface de chamada TEM: não há nada de especial sobre o shell. Também significa que a casca é fácil de substituir; como resultado, os sistemas Unix modernos têm uma variedade de reservatórios para escolher , cada um com sua própria interface de usuário e de script recursos. O shell xv6 é uma simples implementação da essência do shell Unix Bourne. A sua implementação pode ser encontrado em linha (7890)

Processos e memória

Page 9

Um processo xv6 consiste em memória de espaço do usuário (instruções, dados e pilha) e estado por processo privado para o kernel. Xv6 fornece de compartilhamento de tempo: é transparente muda o CPUs disponível entre o conjunto de processos à espera de execução. Quando um processo não está em execução, xv6 salva seus registros de CPU, restaurá-los quando se próximas corridas o processo. O kernel associa um identificador de processo, ou pid, com cada processo.

pid + código
código garfo +
processo de criança
processo pai
código garfo +
exit + código
esperar + código
esperar + código
printf + código
esperar + código
código + exec
código + exec

Um processo pode criar um novo processo usando a chamada de sistema fork. Garfo cria um novo processo, chamado o processo de criança, com exatamente o mesmo conteúdo de memória como o processo de chamada, chamada de processo pai. Fork retorna em ambos os pais eo criança. Na controladora, fork retorna pid da criança; na criança, ela retorna zero. Por exem- amplo, considere o seguinte fragmento de programa:

```
int pid;

pid = fork ();
if (pid> 0) {
   printf ("parent: criança =% d \ n", pid);
   pid = wait ();
   printf ("% d criança é feito \ n", pid);
} Else if (pid == 0) {
   printf ("child: sair \ n");
   saída ();
} Else {
   printf ("erro fork \ n");
}
```

A chamada de sistema de saída faz com que o processo de chamada para parar a execução e para liberar re- fontes, tais como memória e arquivos abertos. A chamada de sistema espera retorna o pid de um filho do atual processo saiu; Se nenhum dos filhos do chamador saiu, aguarde aguarda um para fazê-lo. No exemplo, as linhas de saída

```
parent: criança = 1234
criança: sair
```

pode sair em qualquer ordem, dependendo se o pai ou filho chega ao seu printf chamar primeiro. Depois que a criança sai de espera retornos do pai, fazendo com que o pai impressão

```
parent: criança 1234 é feito
```

Note-se que o pai e filho estavam a executar com memória diferente e diferente registos de: modificação de uma variável num não afecta a outra.

A chamada de sistema exec substitui a memória do processo de chamada com uma nova memória imagem carregada de um arquivo armazenado no sistema de arquivos. O arquivo deve ter uma for- especial esteira, que especifica qual parte do arquivo mantém instruções, que é parte de dados, em que a instrução para iniciar, etc. xv6 usa o formato ELF, que Capítulo 2 discute em mais detalhes. Quando exec bem-sucedido, ele não retorna ao programa de chamada; em vez disso, as instruções carregadas do arquivo de iniciar a execução no ponto de entrada declarado na Cabeçalho ELF. Exec usa dois argumentos: o nome do arquivo que contém o executável e uma série de argumentos de cadeia. Por exemplo:

Page 10

```
char * argv [3];

argv [0] = "echo";
argv [1] = "Olá";
argv [2] = 0;
exec ("/ bin / echo", argv);
printf ("exec erro \ n");
```

getcmd + código
código garfo +
código + exec
código garfo +
código + exec
malloc + código
sbrk + código
descritor de arquivo

Este fragmento substitui o programa de chamada com uma instância do programa / Bin / echo correndo com a lista de argumentos echo Olá. A maioria dos programas de ignorar o primeiro argumento, que é convencionalmente o nome do programa.

O shell xv6 usa as chamadas acima para executar programas em nome dos usuários. O principal estrutura da concha é simples; veja principal (8001.)O loop principal lê a entrada no linha de comando usando getcmd. Em seguida, ele chama garfo, o que cria uma cópia do pro- shell

cesso. As chamadas shell pai espera, enquanto o processo de criança corre o comando. Por exem-
amplo, se o usuário tivesse digitado " echo Olá " no prompt, runcmd teria sido
chamado com " echo Olá " como argumento. runcmd (7906) executa o comando real.
Para " echo Olá ", seria chamar exec (7926). Se for bem-sucedido exec então a criança vai exe-
instruções bonitos de ecoar em vez de runcmd. Em alguns Echo Point irá chamar saída,
o que fará com o pai para retornar de espera na principal (8001). Você pode se perguntar
por garfo e exec não são combinadas em uma única chamada; veremos mais adiante que separam
apela para a criação de um processo e carregamento de um programa é um projeto inteligente.

Xv6 aloca mais memória do espaço do usuário implicitamente: garfo aloca a memória re-
cessário para a cópia da criança de memória do pai, e exec aloca memória suficiente
para armazenar o arquivo executável. Um processo que precisa de mais memória em tempo de execução (talvez
para malloc) pode chamar sbrk (n) a crescer sua memória de dados por n bytes; sbrk retorna o
localização da nova memória.

Não Xv6 não fornece uma noção de usuários ou de proteger um usuário de outro; em
Termos Unix, todos os processos xv6 executados como root.

I / O e descritores de arquivos

Um descritor de arquivo é um pequeno número inteiro que representa um objeto gerenciado-kernel que
um processo pode ler ou escrever. Um processo pode obter um descritor de arquivo por abertura
ing um arquivo, diretório ou dispositivo, ou através da criação de um tubo, ou através da duplicação de uma de- existente
scriptor. Por simplicidade, muitas vezes, vai se referir ao objeto um descritor de arquivo refere-se a como um
" Arquivo "; a interface de descritor de arquivo abstrai as diferenças entre os arquivos, tubulações,
e dispositivos, tornando-os todos se parecem com fluxos de bytes.

Internamente, o kernel xv6 usa o descritor de arquivo como um índice para um ta- por processo
ble, de modo que cada processo tem um espaço privado de descritores de arquivos a partir de zero. Por
convenção, um processo lê a partir de descritor de arquivo 0 (entrada padrão), escreve a saída para arquivo
descritor 1 (saída padrão), e escreve as mensagens de erro para o arquivo descritor 2 (standard
erro). Como veremos, o shell explora a convenção para implementar I O redirecionamento /
e oleodutos. O shell garante que ele sempre tem três descritores de arquivos abertos (8007)
que são de descritores de arquivos padrão para o console.

As chamadas de sistema de leitura e escrita ler e escrever bytes de bytes para abrir arquivos
nomeado por descritores de arquivos. A leitura chamada (fd, buf, n) lê no máximo n bytes do

arquivo descritor fd, copia-os em buf, e retorna o número de bytes lidos. Cada
descritor de arquivo que se refere a um arquivo tem um deslocamento associado a ele. Leia lê dados
a partir do arquivo atual de deslocamento e então avança que compensado pelo número de bytes ler:
uma leitura posterior irá retornar os bytes seguintes os retornados pela primeira leitura.
Quando não houver mais bytes para ler, ler retorna zero para sinalizar o fim do arquivo.

código garfo +
código + exec

A gravação de chamadas (fd, buf, n) escreve n bytes de buf ao fd descritor de arquivo e
retorna o número de bytes escritos. Menos de n bytes só são escritos quando um er-
ror ocorre. Gosta de leitura, gravação grava dados no arquivo atual de deslocamento e então avança
que compensado pelo número de bytes escrito: cada gravação começa onde o anterior
saiu fora.

O fragmento de programa seguinte (que forma a essência do gato) copia dados
a partir de sua entrada padrão para sua saída padrão. Se ocorrer um erro, ele grava uma mensagem
para o erro padrão.

```
de char buf [512];
int n;

para (;;) {
   n = ler (0, buf, buf sizeof);
   if (n == 0)
      break;
   se (n <0) {
      fprintf (2, "erro de leitura \ n");
      saída ();
   }
   se (write (1, buf, n)! = n) {
      fprintf (2, "erro de gravação \ n");
      saída ();
   }
}
```

A coisa importante a se notar no fragmento de código é que o gato não sabe se ele

é a leitura de um arquivo, console, ou um cachimbo. Da mesma forma gato não sei se é imprimindo para um console, um arquivo, ou o que quer. O uso de descritores de arquivos e o convencional ção que arquivo descritor 0 é a entrada e arquivo descritor 1 é saída permite uma imple- simples mentação de gato.

A chamada de sistema perto libera um descritor de arquivo, tornando-o livre para reutilização por um futura aberta, tubulação, ou chamada de sistema dup (veja abaixo). Um descritor de arquivo recém-alocado é alaspectos, o descritor não utilizado de menor número do processo atual.

Descritores de arquivos e interagir garfo para fazer eu O redirecionamento / fácil de implementar. Cópias Fork tabela descritor de arquivo do pai junto com sua memória, para que a criança começa com exatamente os mesmos arquivos abertos como o pai. A chamada de sistema exec substitui o chamando a memória do processo, mas preserva a sua tabela de arquivos. Esse comportamento permite que o shell para implementar I / O redirecionamento por bifurcação, reabrindo descritores de arquivos escolhidos, e, em seguida Execção do novo programa. Aqui é uma versão simplificada do código de um shell é executado pela comando cat <input.txt:


```
char * argv [2];

argv [0] = "gato";
argv [1] = 0;
if (fork () == 0) {
  close (0);
  open ("input.txt", O_RDONLY);
  exec ("gato", argv);
}
```

Depois que a criança fecha arquivo descritor 0, aberto é garantido para usar esse descritor de arquivo para o input.txt recém-inaugurado: 0 será o menor descritor de arquivo disponível. Cat seguida executa com arquivo descritor 0 (entrada padrão) referindo-se a input.txt.

O código para I / O redirecionamento no shell xv6 funciona exatamente dessa maneira (7930). Rechamar que neste momento no código do shell já bifurcada o shell criança e que runcmd chamará exec para carregar o novo programa. Agora deve ser evidente por que é um boa idéia que fork e exec são chamadas separadas. Esta separação permite que o shell para corrigir o processo de criança antes que a criança corre o programa pretendido.

Embora cópias garfo tabela descritor do arquivo, cada deslocamento de arquivos subjacente é compartilhada entre pai e filho. Veja este exemplo:

```
if (fork () == 0) {
  write (1, "Olá", 6);
  saída ();
} Else {
  wait ();
  write (1, "mundo \ n", 6);
}
```

No final deste fragmento, o arquivo anexado ao arquivo descritor 1 conterá os dados Olá mundo. A escrita do pai (que, graças à espera, é executado somente após a criança é feito) pega onde escrita da criança parou. Este comportamento ajuda a produzir saída sequencial a partir de sequências de comandos shell, como (echo Olá, mundo eco) > Output.txt.

A chamada de sistema dup duplica um descritor de arquivo existente, retornando um novo que refere-se ao mesmo objecto subjacente de I / O. Ambos os descritores de arquivos partes um deslocamento, assim como os descritores de arquivos duplicados por um garfo fazer. Esta é uma outra maneira de escrever Olá mundo em um arquivo:

```
fd = dup (1);
write (1, "Olá", 6);
escrever (fd, "mundo \ n", 6);
```

Dois descritores de arquivos partes um deslocamento se eles foram obtidos a partir do mesmo original arquivo descritor por uma seqüência de garfo e chamadas dup. Caso contrário, descritores de arquivos não partes offsets, mesmo que resultaram de concursos públicos para o mesmo arquivo. Dup permite conchas implementar comandos como este: ls-arquivo existente não-arquivo existente> tmp1 2> & 1. O 2> & 1 diz ao shell para dar o comando de um descritor de arquivo 2, que é uma duplicate de descritor 1. Tanto o nome do arquivo existente e a mensagem de erro para o arquivo não-existente vai aparecer no tmp1 arquivo. O shell xv6 não suporta I / O redi-

reção para o descritor de arquivo de erro, mas agora você sabe como implementá-lo.

Page 13

Descritores de arquivos são uma poderosa abstração, porque eles se escondem os detalhes do que tubo
eles estão ligados a: a escrita processo para o arquivo descritor de 1 pode ser escrita em um arquivo, para
um dispositivo como o console, ou a um tubo.

Pipes

Um tubo é um buffer de núcleo pequena exposta a processos como um par de descritores de arquivo,
um para leitura e escrita para uma. A escrita de dados para uma das extremidades do tubo que faz
dados disponíveis para leitura a partir da outra extremidade do tubo. Pipes fornecer uma maneira para
processos para se comunicar.

O seguinte exemplo de código corre o wc programa com entrada padrão conectado
para a extremidade de um tubo de ler.

```
int p [2];
char * argv [2];

argv [0] = "wc";
argv [1] = 0;

pipe (p);
if (fork () == 0) {
  close (0);
  dup (p [0]);
  close (p [0]);
  close (p [1]);
  exec ("/ bin / wc", argv);
} Else {
  escrever (p [1], "Olá mundo \ n", 12);
  close (p [0]);
  close (p [1]);
}
```

O programa chama tubo, o que cria um novo tubo e grava o arquivo ler e escrever
descritores na matriz p. Depois de garfo, pai e filho têm descritores de arquivos rência
anel ao tubo. A criança dups final para ler arquivo descritor 0, fecha o arquivo de-
scriptors em p, e executivos wc. Quando wc lê a partir de sua entrada padrão, ele lê a partir do
pipe. A mãe escreve para o final de gravação do tubo e depois fecha tanto do seu arquivo
descritores.

Se não houver dados disponíveis, a ler em uma tubulação de espera tanto para dados a serem escritos ou todos
descritores de arquivos referentes ao final de gravação a ser fechado; neste último caso, de leitura será re-
0 girar, tal como se o final de um ficheiro de dados foi atingido. O fato de que os blocos de leitura
até que é impossível para os novos dados para se chegar é uma razão que é importante para o
criança para fechar o final de gravação do tubo antes de executar wc acima: se um dos arquivos do wc
descritores que se refere ao final de gravação do tubo, wc nunca veria fim-de-arquivo.

O shell xv6 implementa pipelines como sh.c garfo grep | wc -l em um man-
ner semelhante ao código acima (7950) O processo de criança cria um pipe para conectar o
extremidade esquerda da conduta com a extremidade direita. Em seguida, ele chama runcmd para a extremidade esquerda do
gasoduto e runcmd para a extremidade direita, e aguarda a esquerda e a direita termina a concluir sua
ish, chamando espera duas vezes. A extremidade direita do oleoduto pode ser um comando que se

Page 14

inclui um tubo (por exemplo, a | b | c), que se bifurca dois novos processos filhos (um para b
e um para c). Assim, a concha pode criar uma árvore de processos. As folhas desta árvore
são comandos e os nós interiores são processos que esperar até que a esquerda e direita
crianças completa. Em princípio, você poderia ter os nós interiores executar a extremidade esquerda
um gasoduto, mas fazê-lo corretamente iria complicar a execução.

raiz
caminho
diretório atual

Pipes pode parecer nada mais poderoso do que os arquivos temporários: o gasoduto
echo Olá mundo | wc

poderia ser implementada sem tubos como

echo Olá mundo> / tmp / xyz; wc </ tmp / xyz

Há pelo menos três diferenças fundamentais entre tubos e arquivos temporários. Primeiro, os tubos limpar-se automaticamente; com o redirecionamento de arquivo, um shell teria que ser o cuidado de remover / tmp / xyz quando terminar. Em segundo lugar, os tubos podem passar arbitrariamente longo fluxos de dados, enquanto arquivo redirecionamento requer espaço livre suficiente em disco para armazenar todos os dados. Em terceiro lugar, para permitir que tubos de sincronização: dois processos podem usar um par de tubos para enviar mensagens para trás e para a frente um para o outro, com cada ler bloqueando a sua pro- chamada cesso até que o outro processo foi enviado dados com gravação.

Sistema de arquivo

O sistema de arquivos xv6 fornece arquivos de dados, que são matrizes de bytes não interpretados, e diretórios, que contêm nomeadas referências a arquivos de dados e outros diretórios. Xv6 implementa diretórios como um tipo especial de arquivo. Os diretórios de formar uma árvore, a partir de um diretório especial chamado de raiz. Um caminho como / a / b / c se refere ao arquivo ou diretório c nomeado dentro do diretório chamado b dentro do diretório nomeado na direção da raiz tory /. Caminhos que não começam com / são avaliadas em relação ao cur- do processo de chamada diretório aluguel, que pode ser alterado com a chamada de sistema chdir. Ambos estes código fragmentos de abrir o mesmo arquivo (supondo que todos os diretórios envolvidos existem):

```
chdir ("/ a");
chdir ("b");
open ("c", O_RDONLY);

open ("/ a / b / c", O_RDONLY);
```

O primeiro fragmento muda o diretório atual do processo para / a / b; a segunda nem refere-se a nem modifica o diretório atual do processo.

Há sistema de múltiplas chamadas para criar um novo arquivo ou diretório: mkdir cria um novo diretório, aberta com a bandeira O_CREATE cria um novo arquivo de dados, e mknod cria um novo arquivo de dispositivo. Este exemplo ilustra os três:

```
mkdir ("/ dir");
fd = open ("/ dir / arquivo", O_CREATE | O_WRONLY);
close (fd);
mknod ("/ consola", 1, 1);
```

Mknod cria um arquivo no sistema de arquivos, mas o arquivo não tem conteúdo. Em vez disso, o arquivo metadados marca como um arquivo de dispositivo e registra os maiores e menores números de dispositivos (Os dois argumentos para mknod), que o identificar um dispositivo de kernel. Quando um pro-

Página 15

cesso depois abre o arquivo, o kernel pode desviar ler e sistema de gravação chamadas para o kernel implementação dispositivo em vez de passá-los para o sistema de arquivos. inode Links

fstat recupera informações sobre o objeto de um descritor de arquivo se refere. Ele preenche uma Stat struct, definida em stat.h como:

```
#define T_DIR 1       // Diretório
#define T_FILE 2      // Arquivo
#define T_DEV 3       // Dispositivo

struct estatísticas {
  Tipo de curto; // Tipo de arquivo
  int dev;          // Dispositivo de disco do sistema de arquivos
  uint ino;         // Número Inode
  curto nlink; // Número de links para o arquivo
  tamanho uint;  // Tamanho do arquivo em bytes
};
```

O nome de um arquivo é diferente do arquivo em si; o mesmo arquivo subjacente, chamado de inode, pode ter vários nomes, chamados links. A chamada de sistema ligação cria outro arquivo nome do sistema referentes ao mesmo inode de um arquivo existente. Isto cria um fragmento novo arquivo chamado tanto a e b.

```
open ("a", O_CREATE | O_WRONLY);
ligação ("um", "b");
```

Ler ou escrever para um é o mesmo que ler ou escrever para b. Cada inode

é identificado por um número do nó original. Após a sequência de código de cima, é possível determinar que a e b se referem aos mesmos conteúdos subjacentes inspecionando o resultado de fstat: ambos irão devolver o mesmo número de inode (ino), ea contagem nlink será definido como 2.

A chamada de sistema unlink remove um nome do sistema de arquivos. Inode do arquivo e o espaço em disco, segurando o seu conteúdo só são liberados quando o link de contagem do arquivo é zero e não há descritores de arquivos se referem a ele. Assim, adicionando

```
desligar ("a");
```

para a última seqüência de código deixa o conteúdo inode e arquivos acessível quanto b. Além do mais,

```
fd = open ("/ tmp / xyz", O_CREATE | O_RDWR);
desvincular ("/ tmp / xyz");
```

é uma forma idiomática para criar um inode temporária que vai ser limpo quando o processo fecha fd ou saídas.

Comandos Xv6 para as operações do sistema de arquivos são implementadas como programas de nível de usuário como mkdir, ln, rm, etc. Este projeto permite que qualquer pessoa para estender o shell com novo usuário comandos. Na traseira vista este plano parece óbvio, mas outros sistemas concebidos no tempo de Unix muitas vezes construídos tais comandos no shell (e construiu o shell para o kernel).

Uma exceção é cd, que está embutido no shell (8016)cd deve alterar o curalugar diretório de trabalho do próprio reservatório. Se cd foram executados como um comando regular, em seguida, o shell iria desembolsar um processo filho, o processo de criança correria cd e cd faria alterar o diretório de trabalho da criança. (Ie, do shell) diretório de trabalho do pai

não mudaria.

Mundo real

Combinação de Unix da descritores " padrão " arquivo, tubulações, e shell conveniente sintaxe para as operações sobre eles foi um grande avanço na escrita de propósito geral programas reutilizáveis. A idéia provocou toda uma cultura de " " ferramentas de software que foi responsável por grande parte do poder e popularidade do Unix, eo shell foi a primeira chamada " Linguagem de script. " A interface chamada de sistema Unix persiste até hoje em sistemas como BSD, Linux e Mac OS X.

Kernels modernos fornecem muitos mais chamadas do sistema, e muitos outros tipos de semente serviços, que xv6. Para a maior parte, os sistemas operacionais Unix-derivados modernos têm não seguiu o modelo Unix início de expor dispositivos como arquivos especiais, como o console arquivo de dispositivo discutido acima. Os autores do Unix passou a construir Plan 9, que apaenchiam os " recursos são arquivos " conceito de instalações modernas, as redes que representam, gráficos e outros recursos como arquivos ou árvores de arquivos.

A abstração do sistema de arquivos tem sido uma idéia poderosa, mais recentemente aplicado a recursos de rede sob a forma de a World Wide Web. Mesmo assim, há outros moels para interfaces do sistema operacional. Multics, um antecessor do Unix, abstraído arquivo armazenamento idade de uma maneira que fez com que pareça memória, produzindo um sabor muito diferente de interrosto. A complexidade do projeto Multics teve uma influência direta sobre os designers de Unix, que tentou construir algo mais simples.

Este livro examina como xv6 implementa sua interface Unix-like, mas as idéias e conceitos se aplicam a mais do que apenas Unix. Qualquer sistema operacional deve multiplex processamento es para o hardware subjacente, isolar processos entre si, e fornecer nismo nismos para a comunicação inter-processo controlado. Depois de estudar xv6, você deve ser capaz de olhar para outros, sistemas operacionais mais complexos e veja os conceitos subjacentes xv6 nesses sistemas bem.

Capítulo 1

espaço de endereço
endereço virtual
endereço físico
memória do usuário
struct código proc +
p> xxx + código

O primeiro processo de

Este capítulo explica o que acontece quando xv6 primeiro começa a ser executado, por meio do criação do primeiro processo. Ao fazê-lo, o texto fornece um vislumbre da exe- tação de todos os principais abstrações que xv6 fornece, e como eles interagem. A maioria dos xv6 evita-invólucro especial o primeiro processo e, em vez reutiliza código que deve fornecer xv6 para a operação normal. Próximos capítulos irão explorar cada abstração em mais de- cauda.

Xv6 roda em Intel 80386 ou mais tarde (" x 86 ") processadores em uma plataforma PC, e muito de sua funcionalidade de baixo nível (por exemplo, a sua implementação do processo) é específico do x86. Este livro pressupõe que o leitor tem feito um pouco de programação em nível de máquina em algum arquitetura, e vai apresentar idéias específicas do x86 como eles vêm para cima. Apêndice A brevemente descreve a plataforma PC.

Visão geral do processo

Um processo é uma abstração que proporciona a ilusão de um programa que tem a sua própria máquina abstrata. Um processo fornece um programa com o que parece ser uma pri- sistema vate de memória ou espaço de endereço, que outros processos não podem ler ou escrever. Um processo também fornece o programa com o que parece ser o seu próprio processador para executar as instruções do programa.

Xv6 utiliza tabelas de páginas (que são implementados por hardware) para dar a cada processo o seu próprio espaço de endereço. A tabela de página x86 traduz (ou " mapas ") um endereço virtual (O endereço que uma instrução x86 manipula) para um endereço físico (um endereço que o chip do processador envia para a memória principal).

Xv6 mantém uma tabela de página separada para cada processo que define o processo de espaço de endereço. Como ilustrado na Figura 1-1 , um espaço de endereço inclui o processo de memória do usuário a partir de zero, o endereço virtual. Instruções de vir em primeiro lugar, seguido por glob- variáveis ai, então a pilha, e, finalmente, a área " pilha " (para malloc) que o processo pode expandir conforme necessário.

Espaço de endereçamento de cada processo mapeia instruções e dados do kernel, bem como a memória do programa do usuário. Quando um processo invoca uma chamada de sistema, a chamada de sistema exe- cútis nos mapeamentos do kernel do espaço de endereço do processo. Este arranjo existe para que o código de chamada do sistema do kernel pode referir-se diretamente para a memória do usuário. A fim de deixar espaço para a memória do usuário a crescer, espaços de endereços de xv6 mapear o kernel em alta ad- vestidos, começando em 0x80100000.

O kernel xv6 mantém muitas peças de estado para cada processo, que reúne em um proc struct (2103). Peças mais importantes do estado do kernel de um processo são a sua tabela de página, a sua pilha de kernel, e seu estado de execução. Usaremos a notação p> xxx para se referir a elementos da estrutura proc.

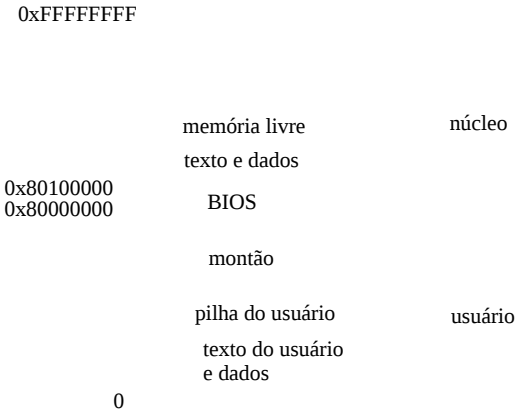


Figura 1-1. Esquema de um espaço de endereço virtual

fio
p> KStack + código
p> estado + código
p> PGDIR + código

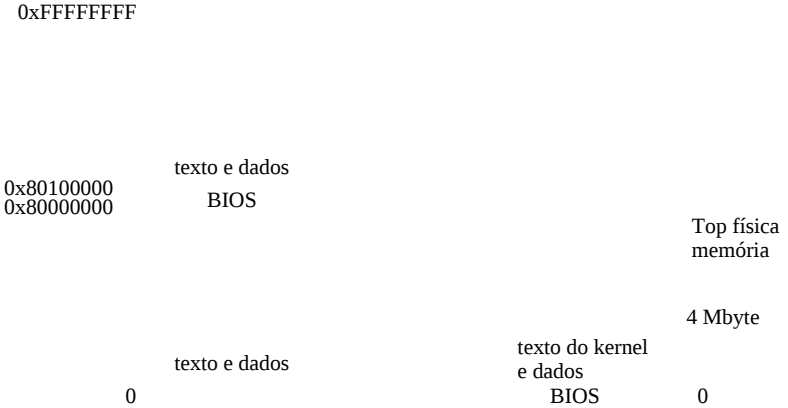
Cada processo tem um segmento de execução (ou thread para o short) que executa a instruções do processo. Um segmento pode ser suspenso e mais tarde retomado. Para alternar transparently entre os processos, o kernel suspende o segmento em execução no momento e resome o fio de outro processo. Grande parte do estado de uma rosca (variáveis locais, função chamar endereços de retorno) é armazenada em pilhas do thread. Cada processo tem duas pilhas: um pilha do usuário e uma pilha de kernel (p> KStack). Quando o processo é de execução in- utilizador truções, apenas a sua pilha do usuário estiver em uso, e sua pilha do kernel está vazio. Quando o processo entra no kernel (através de uma chamada de sistema ou de interrupção), o código do kernel é executado no pilha do kernel do processo; enquanto um processo está no kernel, sua pilha usuário ainda contém salvo de dados, mas não é utilizada activamente. Suplentes linha de uma processo entre usando ativamente o pilha do usuário ea pilha de kernel. A pilha de kernel é separado (e protegido de usuário código) para que o kernel pode executar, mesmo que um processo tenha destruído a sua pilha do usuário.

Quando um processo faz uma chamada de sistema, o processador muda para a pilha de kernel, aumenta o nível de privilégio hardware, e começa a executar as instruções do kernel que implementar a chamada de sistema. Quando a chamada do sistema for concluída, o kernel retorna ao usuário espaço: o hardware reduz o seu nível de privilégio, volta para a pilha do usuário, e resome a execução de instruções de utilização logo após a instrução de chamada do sistema. Um de processo thread pode " block " no kernel que esperar para I / O, e retomar de onde parou quando o I / O foi concluída.

p> estado indica se o processo é alocado, pronto para correr, correr, esperaing para I / O, ou sair.

p> PGDIR detém tabela de páginas do processo, no formato que o ex- hardware x86 pectos. xv6 faz com que o hardware de paginação para usar de um processo p> PGDIR ao executar esse processo. Tabela de páginas de um processo também serve como o registro dos endereços do páginas físicas alocada para armazenar a memória do processo.

Código: o primeiro espaço de endereço



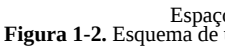


Figura 1-2. Esquema de um espaço de endereço virtual

Quando um PC em poderes, ele inicializa em si e carrega um carregador de inicialização do disco na memória e executa-lo. Apêndice B explica os detalhes. Cargas boot loader de Xv6 o kernel xv6 a partir do disco e executa-lo a partir de entrada (1040) O hardware x86 paginação Ware não está habilitado quando o kernel inicia; endereços virtuais mapear diretamente física endereços.

boot loader
código de entrada +
Código KERNBASE +
código de entrada +
entrypgdir + código

O carregador de inicialização carrega o kernel xv6 na memória durante 0x100000 endereço físico. A razão pela qual ele não carrega o kernel em 0x80100000, onde o kernel espera encontrar suas instruções e dados, é que não pode ser qualquer memória física em um tal de alta dirigir sobre uma pequena máquina. A razão pela qual ele coloca o kernel em vez de 0x100000 0x0 é porque a faixa de endereços 0xa0000: 0x100000 contém dispositivos de E / S.

Para permitir que o resto do kernel para executar, a entrada define uma tabela de páginas que mapeia virtualmente a partir de endereços al 0x80000000 (chamado KERNBASE (0207) Para endereços físicos partida ing em 0x0 (ver Figura 1-1). Criação de duas faixas de endereços virtuais que mapeiam para o mesmo intervalo de memória física é um uso comum de tabelas de páginas, e vamos ver mais exemplos como este.

A tabela de página de entrada é definida em main.c (1311) Nós olhamos para os detalhes da página tables no Capítulo 2, mas o conto é que a entrada 0 mapeia endereços virtuais 0: 0x400000 para endereços físicos 0: 0x400000. Este mapeamento é necessário enquanto entrada está sendo executado em endereços baixos, mas acabará por ser removida.

Entrada 960 mapeia endereços KERNBASE virtual: KERNBASE + 0x400000 para ad- física vestidos 0: 0x400000. Esta entrada será usado pelo kernel após a entrada tenha terminado; ele mapeia os altos endereços virtuais em que o kernel espera encontrar as suas instruções e dados aos baixos endereços físicos onde o carregador de inicialização carregados eles. Este mapeamento limita as instruções do kernel e dados para 4 Mbytes.

Retornando a entrada, ele carrega o endereço físico do entrypgdir em controle register% CR3. O hardware de paginação deve saber o endereço físico de entrypgdir, torna-

fazer com que ele não sabe como traduzir endereços virtuais ainda; não tem uma tabela de página Ainda. O símbolo entrypgdir refere-se a um endereço na memória alta, eo macro V2P_WO (0220) subtrai KERNBASE a fim de encontrar o endereço físico. Para permitir que o hardware de paginação, xv6 define o CR0_PG bandeira no registro de controle% CR0.

V2P_WO + código
Código CR0_PG +
código principal +
código principal +
código principal +
allocproc + código
EMBRIÃO + código
pid + código
forkret + código
trapret + código
p> contexto + código
forkret + código
trapret + código
forkret + código
trapret + código

O processador ainda está executando instruções em endereços baixos depois de paginação é enabled, que funciona desde entrypgdir mapeia endereços baixos. Se xv6 tinha omitido entrada 0 de entrypgdir, o computador teria caído ao tentar executar o intrução após o que permitiu aos paginação.

Agora entrada precisa transferir para o código do kernel C, e executá-lo em alta memória. Em primeiro lugar, faz com que o ponteiro de pilha, esp%, ponto de memória para ser utilizado como pilha (1054) Todos os símbolos têm endereços altos, incluindo pilha, de modo a pilha ainda será válida mesmo quando os mapeamentos de baixo são removidos. Finalmente entrada salta para principal, que é também um alta endereço. O salto indireta é necessária porque a montadora seria de outra forma gerar um salto direto relativo ao PC, que iria executar a versão de baixa memória de principal. Principal não pode retornar, uma vez que o não há retorno PC na pilha. Agora o kernel está sendo executado em endereços de alto no principal função (1217)

Código: criar o primeiro processo

Após principal inicializa vários dispositivos e subsistemas, cria-se o primeiro processo por chamando userinit (1239) Primeira ação do Userinit é chamar allocproc. O trabalho de allocproc (2205) é alocar um slot (a proc struct) na tabela de processos e rubricam ize as partes do estado do processo necessários para a sua lista de discussão do kernel para executar. Allocproc é chamado para cada novo processo, enquanto userinit é chamado apenas para o primeiro processo. Allocproc varre a tabela proc para um slot com o estado NÃO UTILIZADO (2211-2213) Quando ele encontra um slot não utilizado, allocproc define o estado de embrião para marcá-lo como usado e dá a processar um único pid (2201-2219) Em seguida, ele tenta alocar uma pilha do kernel para o processo de thread do kernel. Se a alocação de memória falhar, allocproc altera o estado de volta a UNUSADO e retorna zero para sinalizar o fracasso.

Agora allocproc deve configurar pilha do kernel do novo processo. allocproc é escrito de modo que ele pode ser usado por um garfo, bem como durante a criação do primeiro processo. allocproc

define-se o novo processo com uma pilha do kernel especialmente preparado e um conjunto de regis- do kernel
ters que fazem com que " return " para o espaço do usuário quando ele é executado primeiro. O layout do pré-
pilha do kernel pared será como mostrado na Figura 1- 3. allocproc faz parte deste trabalho
através da criação de valores do contador de programa de retorno que fará do kernel do novo processo
thread para executar no primeiro forkret e, em seguida, em trapret (2236-2241) O kernel discussão
começará a executar com conteúdo do registro copiados de p> contexto. Assim, definindo p
> Ao contexto> EIP para forkret fará com que a thread do kernel para executar no início do
forkret (2533) Esta função irá retornar para qualquer outro endereço é na parte inferior do
empilhar. O código de mudança de contexto (2708) define o ponteiro de pilha para apontar apenas para além da
final de p-> contexto. lugares allocproc p-> contexto na pilha, e coloca um ponteiro para
trapret logo acima dele; que é onde forkret vai voltar. trapret restaura regis- usuário
tros de valores armazenados no topo da pilha do núcleo e salta para o processo de (3027).
Esta configuração é a mesma para garfo comum e para a criação do primeiro processo, embora em
último caso, o processo começará a executar no local em espaço de usuário de zero em vez de

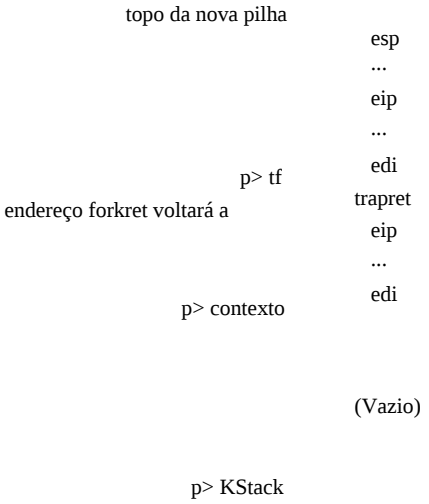


Figura 1-3 . A nova pilha de kernel.

um retorno de garfo.

Como veremos no capítulo 3, a maneira que controlar as transferências de software para usuário
o núcleo é através de um mecanismo de interrupção, que é usado pelo sistema de chamadas, interrompe e
exceções. Sempre que as transferências de controle no kernel enquanto um processo está em execução, e
hardware e xv6 armadilha código de entrada salvar usuário se registra na pilha do kernel do processo.
userinit escreve valores no topo da nova pilha que são parecidos com aqueles que
estar lá, se o processo tivesse entrado o kernel através de uma interrupção (2264-2270) De modo que o ou-
código dinary para o regresso a partir do kernel de volta para o código do usuário do processo funcionará.
Estes valores são uma trapframe struct que armazena os registros de usuário. Agora, o novo
pilha do núcleo do processo é completamente preparado como mostrado na Figura 1-3 .

O primeiro processo está indo para executar um pequeno programa (initcode.S; (7700). O
processo necessita de memória física no qual armazenar este programa, o programa precisa
ser copiados para a memória, e o processo necessita de uma tabela de página, que se refere a que mem-
ory.

userinit chama setupkvm (1737) para criar uma tabela de página para o processo com (a primeira)
mapeamentos apenas para a memória que o kernel usa. Vamos estudar essa função em detalhe
no Capítulo 2, mas a um nível elevado e setupkvm UserInit criar um espaço de endereço como
mostrado F IGURA 1-1 .

O conteúdo inicial da memória do primeiro processo são a forma compilada de init-
code.S; como parte do processo de construção do kernel, o vinculador que incorpora binário no kernel
e define dois símbolos especiais, _binary_initcode_start e _bina-
ry_initcode_size, indicando a localização e tamanho do binário. Cópias UserInit
que binário na memória do novo processo, chamando inituvm, qual aloca
página de memória física, mapeia endereço virtual zero a essa memória, e copia o bi-
insignificante para essa página (1805).

userinit + código
struct
trapframe + código
initcode.S + código
userinit + código
setupkvm + código
initcode.S + código
_binary_initcode_start + c
_binary_initcode_size + c
inituvm + código

Page 22

Então userinit configura o quadro de interceptação (0002) com o estado do modo de usuário inicial: o % Cs registo contém um seletor de segmento para segmento SEG_UCODE rodando a privi- DPL_USER nível lege (ou seja, o modo de usuário de modo não kernel), e similarmente% ds,% es, e ss% usar SEG_UDATA com privilégio DPL_USER. O bit% eflags FL_IF está definido para permitir hardware Ware interrompe; vamos reexaminar isso no Capítulo 3.

O ponteiro de pilha% esp está definido para maior endereço virtual válido do processo, p> sz. O ponteiro de instrução é definido como o ponto de entrada para o initcode, endereço 0.

A função userinit define p> nome para initcode principalmente para depuração. Cenário p> CWD define o diretório de trabalho atual do processo; examinaremos namei em detalhes no Capítulo 6.

Uma vez que o processo é inicializado, userinit marca-lo disponível para agendamento por setting p-> estado para RUNNABLE.

Código: A execução do primeiro processo

Agora que o primeiro estado do processo é preparado, é hora de executá-lo. Depois de chamadas principais userinit mpmain chama agendador para iniciar os processos em execução (1267) Scheduler (2458) procura por um processo com p> estado definido para RUNNABLE, e só há uma: initproc. Ele define a variável proc per-cpu para o processo que encontrou e chamadas switchuvm para contar a hardware para começar a usar tabela de páginas do processo de destino (1768) Alterando tabelas de páginas durante a execução no kernel funciona porque setupkvm provoca tabelas de páginas de todos os processos ter mapeamentos idênticos para o código do kernel e dados. switchuvm também configura uma tarefa SEG_TSS segmento estado que instrui o hardware executar chamadas de sistema e interrupções na pilha do kernel do processo. Vamos reexaminar o segmento estado tarefa no Capítulo 3.

agendador agora define p> estado de execução e chama swtch (2708) para realizar uma mudança de contexto para kernel discussão do processo de destino. swtch salva os registros atuais e carrega os registros salvos do kernel discussão alvo (cesso> contexto) para o x86 registros de hardware, incluindo o ponteiro da pilha e ponteiro de instrução. O actual contexto não é um processo, mas sim um contexto especial per-cpu scheduler, assim agendador informa swtch para salvar os registros de hardware atual no armazenamento por CPU (CPU> de horários uler) e não em contexto de discussão do kernel de qualquer processo. Vamos examinar swtch em mais detalhes no Capítulo 5. A instrução ret definitiva (2727) pops do processo de destino% eip a partir da pilha, terminando a mudança de contexto. Agora, o processador está sendo executado no kernel pilha de processo p.

Allocproc definir de initproc p> ao contexto> EIP para forkret, de modo que o ret começa executing forkret. Na primeira chamada (que é este), forkret (2533) corre inicialização funções que não podem ser executados a partir de principal, porque eles devem ser executados no contexto de um processo regular com a sua própria pilha de kernel. Então, forkret retornos. Allocproc arranjado para que a palavra do topo da pilha após p> contexto é exibido fora seria trapret, então agora trapret começa a executar, com esp% definido para p> tf. Trapret (3027) utiliza instruções pop para restaurar registros do quadro de interceptação (0602) assim como swtch fez com o contexto do kernel: Popal restaura os registos gerais, em seguida, a instrução popl ções restaurar% gs,% fs,% es, e% ds. Os saltos ADDL ao longo dos dois campos e trapno errcode. Por fim, a instrução iret aparece% cs,% eip, bandeiras%,% esp, e ss% de a pilha. O conteúdo do quadro armadilha pode ter sido transferido para o estado da CPU, então

Código SEG_UCODE + Código DPL_USER + Código SEG_UDATA + Código DPL_USER + FL_IF + código userinit + código p> nome + código p> CWD + código código namei + userinit + código Código + RUNNABLE mpmain + código código + agendador switchuvm + código setupkvm + código Código SEG_TSS + código + agendador swtch + código cpu- > Código + agendador swtch + código código + ret forkret + código código + ret forkret + código forkret + código código principal + p> contexto + código trapret + código swtch + código código Popal + popl + código código addl + código iret +

Page 23

o processador continua na EIP% especificado no quadro de interceptação. Para initproc, que significa zero endereço virtual, a primeira instrução de initcode.S.

Neste ponto,% eip detém zero e% esp detém 4096. Estes são os endereços virtuais no espaço de endereço do processo. Hardware paginação do processador traduz em endereços físicos. allocuvm configurar tabela de página do processo para que endereço virtual zero, refere-se à memória física alocada para este processo, e definir um sinalizador (PTE_U) que informa o hardware de paginação para permitir que o código de usuário para acessar a memória. O userinit (2264) configurar os baixos pedaços de% cs para executar o código de utilizador do processo na CPL = 3 significa que o código de utilizador só pode usar páginas com PTE_U set, e não pode modificar sensi-

initproc + código initcode.S + código allocuvm + código Código PTE_U + userinit + código código + exec SYS_exec + código T_SYSCALL + código código + exec exit + código / Init + código

registros de hardware tivas como% CR3. Assim, o processo é limitado à utilização de apenas o seu /initcode.asm código
própria memória.

A primeira chamada do sistema: exec

A primeira ação de initcode.S é invocar a chamada de sistema exec. Como vimos no Capítulo 0, exec substitui a memória e registros do processo atual com um novo programa, mas deixa os descritores de arquivos, identificação de processo e processo pai inalterado.

Initcode.S (7708) começa empurrando três valores no stack- $ argv, $ init, e $ 0 e, em seguida, define% eax para SYS_exec e executa int T_SYSCALL: ele está pedindo o kernel para executar a chamada de sistema exec. Se tudo correr bem, exec nunca retorna: começa running o programa denominado por $ init, que é um ponteiro para a string terminada em NUL / Nisso (7721-7723) Se o exec falhar e não voltar, initcode laços chamando a saída sischamada tem, o que definitivamente não deve retornar (7715-7719)

Os argumentos para a chamada do sistema exec é de US $ inicialização e argv $. O zero definitiva faz este olhar escrito à mão chamada de sistema como o sistema ordinário chama, como veremos no capítulo 3. Tal como antes, esta configuração especial evita-invólucro do primeiro processo (neste caso, sua primeira chamada do sistema) e, em vez reutiliza código que xv6 deve prever operação padrão ção.

Capítulo 2 irá cobrir a implementação de exec em detalhes, mas a um alto nível, irá substituir initcode com o binário de init /, carregado para fora do sistema de arquivos. Agora initcode (7700) é feito, e o processo será executado / init vez. Nisso (7810) cria um novo arquivo de dispositivo do console, se necessário e, em seguida, abre-o como arquivo descritores 0, 1 e 2. Zombies Então ele faz um loop, iniciando um shell console, alças órfãs até o shell saídas, e repete. O sistema é para cima.

Mundo real

A maioria dos sistemas operacionais têm adotado o conceito de processo, e a maioria dos processos semelhante ao do xv6. Um verdadeiro sistema operativo iria encontrar estruturas proc livres com um livre lista explícita em tempo constante em vez da busca em tempo linear em allocproc; xv6 usa a varredura linear (o primeiro de muitos) pela simplicidade.

layout do espaço de endereços do xv6 tem o defeito que ele não pode fazer uso de mais de 2 GB de RAM física. É possível corrigir isso, embora o melhor plano seria para mudar para uma máquina com endereços de 64 bits.

Exercícios

1. Defina um ponto de interrupção na swtch. Passo único com stepi do gdb através da ret para forkret, em seguida, usar o remate do gdb para avançar para trapret, então stepi até chegar ao initcode em zero endereço virtual.

2. KERNBASE limita a quantidade de memória de um único processo pode utilizar, que pode ser irritante em uma máquina com um total de 4 GB de RAM. Gostaria de levantar KERNBASE permitir uma processar a usar mais memória?

Capítulo 2

entradas de tabela de página (PTE)
página
diretório página
páginas da tabela de páginas
Código PTE_P +
Código PTE_W +

Tabelas de páginas

Tabelas de páginas são o mecanismo através do qual o sistema operacional controla o que endereços de memória significam. Eles permitem xv6 multiplex os espaços de endereços de diferente processos em uma única memória física, e para proteger as memórias de pro- diferente processos. O nível de engano fornecida por tabelas de páginas também é uma fonte para muitos puro truques. xv6 utiliza tabelas de páginas principalmente para multiplex espaços de endereço e para proteger os Estados ory. Ele também usa alguns truques simples na tabela de páginas: mapeando a mesma memória (o ker- nel) em vários espaços de endereçamento, mapeando a mesma memória mais de uma vez em um ad- espaço vestido (cada página do usuário também é mapeado em vista físico do núcleo de memória), e guardando uma pilha do usuário com uma página não mapeada. O resto deste capítulo explica as tabelas de páginas que o hardware x86 fornece e como xv6 utiliza-los.

Hardware de paginação

Como um lembrete, instruções x86 (tanto do usuário e kernel) manipular endereços virtuais. RAM da máquina, ou a memória física, é indexado com endereços físicos. O x86 hardware mesa página conecta esses dois tipos de endereços, mapeando cada virtual abordar a um endereço físico.

Uma tabela de página x86 é logicamente uma matriz de 2 ^ 20 (1.048.576) entradas de tabela de página (PTE). Cada PTE contém um 20-bit número de página física (PPN) e algumas bandeiras. O hardware paginação traduz um endereço virtual, usando suas melhores 20 bits para o índice no tabela de página para encontrar um PTE, e substituindo top 20 bits do endereço com o PPN na PTE. As cópias de hardware paginação as baixas 12 bits inalterados do virtual para o endereço físico traduzido. Assim, uma tabela de página dá o controle sobre o sistema operacional Virtual-to-físicos traduções de endereços na granularidade de pedaços alinhados de 4096 (2 ^ 12) bytes. Esse pedaço é chamado de página.

Tal como mostrado na F igura 2-1, a tradução real acontece em duas etapas. A tabela de página é armazenado na memória física como uma árvore de dois níveis. A raiz da árvore é um byte 4096 diretório página que contém 1.024 referências PTE-como nas páginas da tabela principal. Cada página da tabela de página é um array de 1024 PTEs de 32 bits. O hardware paginação usa o top 10 bits de um endereço virtual para selecionar uma entrada de diretório página. Se a entrada de diretório de página é Actualmente, a paginação do hardware utiliza os próximos 10 bits do endereço virtual para seleccionar um PTE a partir da página da tabela de página que a entrada de diretório página se refere a. Se tanto o página de entrada de diretório ou o PTE não estiver presente, o hardware de paginação levanta uma falha. Esta estrutura de dois níveis permite uma tabela de páginas para omitir páginas inteiras da tabela de páginas na com-

mon caso em que grandes faixas de endereços virtuais não têm mapeamentos.

Cada PTE contém bandeira bits que informam o hardware de paginação como o vir- associado
endereço tual é permitido para ser usado. PTE_P indica se o PTE está presente: se é
não definido, uma referência para a página faz com que uma falha (ou seja, não é permitido). Controles PTE_W

Page 26

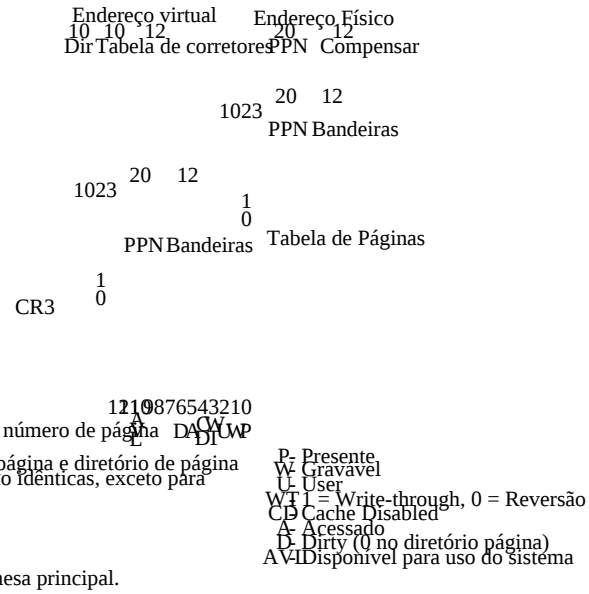


Figura 2-1 . x86 hardware mesa principal.

se as instruções estão autorizados a emitir escreve para a página; se não for definido, apenas lê e
fetches instrução são permitidos. PTE_U controla se os programas do usuário é permitido
usar a página; se claro, apenas o kernel é permitido usar a página. Figura 2 mostra como -1
tudo funciona. As bandeiras e todas as estruturas relacionadas com hardware outra página são definidos no
mmu.h (0200)

Código PTE_U + kvmalloc + código

Algumas notas sobre termos. Memória física refere-se a células de armazenamento em DRAM. A
byte de memória física tem um endereço, chamado de endereço físico. Instruções de uso
apenas endereços virtuais, que o hardware paginação traduz em endereços físicos, e
então envia para o hardware DRAM para ler ou escrever o armazenamento. Nesse nível de discussão
não existe tal coisa como memória virtual, apenas endereços virtuais.

Espaço de endereço de processo

A tabela Página criada por entrada tem mapeamentos suficiente para permitir que o kernel C
código para começar a correr. No entanto, a principal muda imediatamente para uma nova tabela de página por
chamando kvmalloc (1757). Porque kernel tem um plano mais elaborado para descrever pro-
espaços de endereço cesso.

Cada processo tem uma tabela de página separada, e xv6 diz o hardware tabela de página
tabelas de páginas interruptor quando interruptores xv6 entre processos. Como mostrado na Figura 2-2 , uma
memória do usuário do processo começa no endereço virtual zero e pode crescer até KERNBASE, al-
mugido um processo para lidar com até 2 GB de memória. Quando um processo pede para xv6
mais memória, xv6 primeiro encontra páginas físicas gratuitas para fornecer o armazenamento, e em seguida, adiciona

Page 27

Virtual
4 Gig

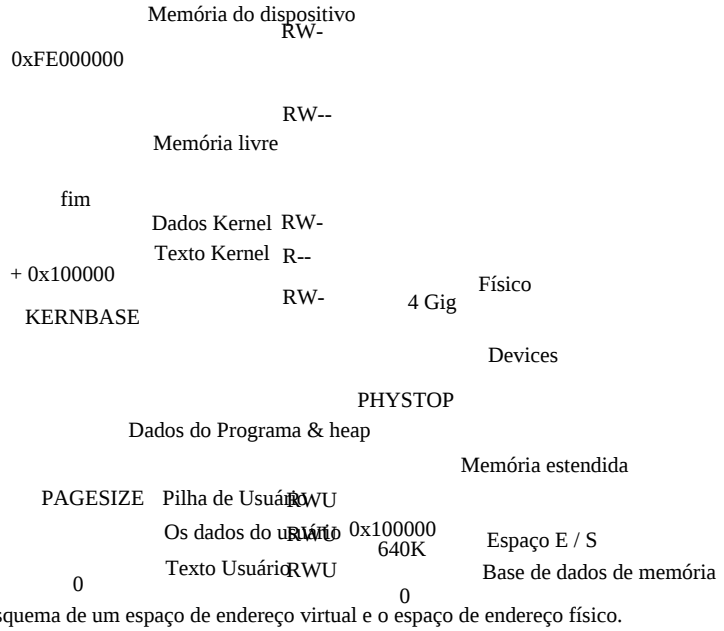


Figura 2-2 . Esquema de um espaço de endereço virtual e o espaço de endereço físico.

PTEs a tabela de páginas do processo que apontam para as novas páginas físicas. define o xv6 Código PTE_U +
PTE_U, PTE_W e PTE_P bandeiras nestes PTEs. A maioria dos processos não use o todo espaço de endereço do usuário; xv6 deixa PTE_P claro em não utilizado PTEs. Página 'diferentes processos mesas de traduzir os endereços de usuários para páginas diferentes de memória física, de modo que cada processo tem memória do usuário privada.

Xv6 inclui todos os mapeamentos necessários para o kernel para funcionar em todos os processos da página table; esses mapeamentos todos aparecem acima KERNBASE. Ele mapeia endereços virtuais KERNBASE: KERNBASE + PHYSTOP a 0: PHYSTOP. Uma razão para este mapeamento é de modo a que o kernel pode usar suas próprias instruções e dados. Outra razão é que o com- pletamente do kernel vezes necessita de ser capaz de escrever uma determinada página de memória física, por exemplo, quando criação de páginas da tabela de páginas; ter cada página física comparecer a uma previsível virtual endereço torna este conveniente. Um defeito desse arranjo é que xv6 não pode fazer utilização de mais de 2 GB de memória física. Alguns dispositivos que usam o mapeamento de memória I / O aparecer em endereços físicos a partir de 0xFE000000, assim tabelas de páginas, incluindo xv6 um mapeamento direto para eles. Não Xv6 não definir o sinalizador PTE_U no PTEs acima KERNBASE, portanto, apenas o kernel pode usá-los.

Tendo tabela de páginas de cada processo contém mapeamentos, tanto para a memória do usuário e todo o kernel é conveniente quando se muda de código de usuário para kernel código durante sistema chama e interrompe: essas opções não requerem chaves de tabela de página. Para o

maior parte do kernel não tem sua própria tabela de página; é quase sempre tomando emprestado tabela de página de algum processo. Código PTE_U + código principal + kvmalloc + código setupkvm + código mappages + código kmap + código

Para rever, xv6 assegura que cada processo só pode usar sua própria memória, e que cada processo vê a sua memória como tendo endereços virtuais contíguos a partir de zero. xv6 implementa a primeira, definindo o bit PTE_U apenas em PTEs de endereços virtuais que se referem à memória do próprio processo. Ele implementa o segundo usando a capacidade de tabelas de páginas para traduzir endereços virtuais sucessivas para qualquer páginas físicas acontecer para ser alocada para o processo. Código PHYSTOP + mappages + código código walkpgdir + código walkpgdir + Código PHYSTOP +

Código: a criação de um espaço de endereço

chamadas principais kvmalloc (1797) para criar e mudar para uma tabela de páginas com os mapeamentos acima KERNBASE necessário para o kernel para funcionar. A maior parte do trabalho acontece em setupkvm (1737). Ele primeiro aloca uma página de memória para armazenar o diretório de página. Em seguida, ele chama mappages para instalar as traduções que as necessidades de kernel, as quais estão descritas na kmap (1728) array. As traduções incluem instruções e dados do kernel, física de memória de até PHYSTOP, e faixas de memória, que são, na verdade, dispositivos de E / S. montar- kvm não instala os mapeamentos para a memória do usuário; isso vai acontecer mais tarde.

mappages (1679)instala mapeamentos em uma tabela de página para uma gama de endereços virtuais para uma gama correspondente de endereços físicos. Fá-lo separadamente para cada virtuais abordar na gama, em intervalos de página. Para cada endereço virtual a ser mapeada, map- páginas chama walkpgdir para encontrar o endereço do PTE para esse endereço. Em seguida, inicial- truir o PTE para manter o número da página física relevante, as permissões desejadas (PTE_W e / ou PTE_U) e PTE_P para marcar o PTE como válidos (1691)

walkpgdir (1654)imita as ações do hardware x86 paginação como ele olha para cima o TEP para um endereço virtual (ver Figura 2 -1). walkpgdir utiliza os 10 bits superiores de o endereço virtual para encontrar a entrada de diretório página (1659)Se a entrada de diretório de página não estiver presente, então a página da tabela de página necessário ainda não foi atribuída; se o alloc argumento é definido, walkpgdir aloca-lo e coloca o seu endereço físico na página de direção tory. Por fim, utiliza os próximos 10 bits do endereço virtual para encontrar o endereço do PTE na página de tabela de página (1672)

Alocação de memória física

O kernel precisa alocar e liberar memória física livre em tempo de execução para a página de ta- bles, memória usuário processo, as pilhas do kernel, e tampões de tubulação.

xv6 utiliza a memória física entre a extremidade do núcleo e para PHYSTOP run-time de alocação. Ele aloca e libera páginas inteiras de 4096 bytes de cada vez. Ele mantém o controle de quais páginas estão livres enfiando uma lista ligada através das próprias páginas. Alocação consiste em remover uma página da lista ligada; libertação consiste em adicionar a página liberado para a lista.

Há um problema de inicialização: tudo de memória física devem ser mapeadas a fim para o alocador para inicializar a lista livre, mas a criação de uma tabela de páginas com esses mapeamentos envolve a alocação de páginas na tabela de páginas. xv6 resolve esse problema usando uma página separada alocador durante a entrada, que aloca memória logo após o fim dos dados do kernel

segmento. Este alocador não suporta libertando e é limitado pelo mapeamento de 4 MB no entrypgdir, mas que é suficiente para afectar a primeira tabela de página do kernel.

Código: alocador de memória física

Estrutura de dados do alocador é uma lista livre de páginas de memória física que estão disponíveis capaz de destinação. Cada elemento da lista de páginas livres é uma corrida struct (2764)Onde o alocador de obter a memória para manter essa estrutura de dados? Ele armazena o funcionamento de cada página livre estrutura na própria página livre, já que não há mais nada lá armazenados. A lista é livre protegido por um bloqueio de rotação (2765- 2766)A lista eo bloqueio estão envoltos em uma estrutura para deixar claro que o bloqueio protege os campos na estrutura. Por agora, ignorar o bloqueio e as chamadas para adquirir e liberar; Capítulo 4 examinará bloqueio em detalhe.

A função de chamadas principais kinit1 e kinit2 para inicializar o alocador (2780)O razão para ter duas chamadas é que, para grande parte da principal não pode usar fechaduras ou memória acima de 4 megabytes. A chamada para kinit1 configura para alocação-lock menos na primeira 4 megabytes, ea chamada para kinit2 permite bloqueio e organiza para mais memória para ser allocatable. principal deve determinar a quantidade de memória física está disponível, mas este acaba por ser difícil no x86. Em vez disso, assume que a máquina tem 240 megabytes (PHYSTOP) de memória física, e usa toda a memória entre o final do kernel e PHYSTOP como o conjunto inicial de memória livre. kinit1 e chamada kinit2 freerange adicionar memória à lista livre via por página chama a kfree. A PTE só pode referem-se a um endereço físico que se encontra alinhado com um limite de 4096 bytes (é um múltiplo de 4096), assim freerange usa PGROUNDUP para garantir que ele libera apenas alinhado ad- física vestidos. O alocador começa sem memória; essas chamadas para kfree dar-lhe algum para gerenciar.

O alocador se refere às páginas físicas por seus endereços virtuais como mapeado em alta memória, não por seus endereços físicos, razão pela qual kinit usa p2v (PHYSTOP) para traduzir PHYSTOP (um endereço físico) para um endereço virtual. O alocador vezes trata endereços como números inteiros, a fim de efectuar operações aritméticas sobre eles (por exemplo, percorrendo todos páginas em kinit), e às vezes usa endereços como ponteiros para ler e escrever de memória (Por exemplo, manipular a estrutura de execução armazenados em cada página); esta dupla utilização de endereços é o principal motivo que o código alocador está cheio de tipo C casts. A outra razão é que a liberação e alocação inerentemente alterar o tipo de memória.

O kfree função (2815)começa por cada byte na memória de ser libertado

struct run + código
código principal +
kinit1 + código
kinit2 + código
Código PHYSTOP +
código freerange +
kfree + código
PGROUNDUP + código
Tipo de elenco
kalloc + código
sobre página livre

com o valor 1. Isto fará com que o código que usa memória após liberando-o ("usos" dangling referências ") para ler lixo em vez do velho válido conteúdo; espero que causará tal código para quebrar mais rapidamente. Então kfree lança v para um ponteiro para a estrutura de execução, registra a old início da lista livre em r-> seguinte, e define a lista livre igual a r. remove kalloc e retorna o primeiro elemento da lista livre.

Parte do usuário de um espaço de endereço

Figura 2-3 mostra o layout da memória do usuário de um processo de execução, de xv6. A pilha é acima da pilha para que ele possa expandir (com sbrk). A pilha é um único página, e é mostrado com o conteúdo inicial como criado por exec. Strings contendo os

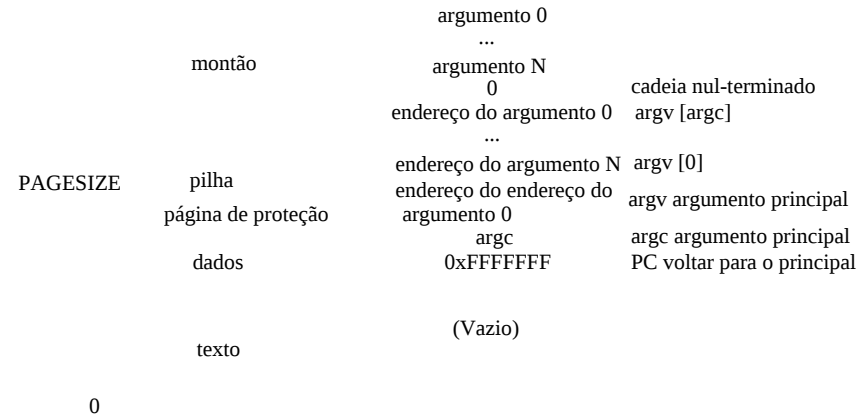


Figura 2-3 . Disposição da memória de um processo do usuário com a sua pilha inicial.

argumentos de linha de comando, assim como um conjunto de ponteiros para eles, estão no topo da pilha. Pouco menos de que são valores que permitem que um programa para iniciar a principal como se o função chamada main (argc, argv) tinha acabado de começar. Para proteger-se uma pilha crescente ao longo da página stack, xv6 coloca uma página de proteção logo abaixo da pilha. A página de proteção não é mapeados e por isso, se a pilha é executado fora da página stack, o hardware irá gerar um exception porque não pode traduzir o endereço da falha.

código namei + Formato ELF struct elfhdr + código Código ELF_MAGIC + setupkvm + código allocuvm + código

Código: exec

Exec é a chamada de sistema que cria a parte do usuário de um espaço de endereço. Ele inicializa o parte do usuário de um espaço de endereço a partir de um arquivo armazenado no sistema de arquivos. Exec (5910) binário o caminho nomeados usando namei (5920) O que é explicado no Capítulo 6. Em seguida, ele lê o cabeçalho da ELF. Xv6 aplicações estão descritas no formato ELF amplamente utilizado, definido em elf.h. Um binário ELF consiste em um cabeçalho ELF, struct elfhdr (0955) Seguindo- lowed por uma sequência de cabeçalhos de seção programa, proghdr struct (0974) Cada progh- dr descreve uma seção do aplicativo que deve ser carregado na memória; xv6 programas têm apenas um programa de cabeçalho de seção, mas outros sistemas pode ter separado seções para instruções e dados.

O primeiro passo é uma verificação rápida que o arquivo provavelmente contém um binário ELF. Um ELF binário começa com a de quatro bytes " número mágico " 0x7F, 'E', 'L', 'F', ou ELF_MAGIC (0952) Se o cabeçalho ELF tem o número mágico direito, exec assume que o binário é bem formada.

Exec aloca uma nova tabela de página com nenhum mapeamento de usuário com setupkvm (5931) Alo memória cados para cada segmento ELF com allocuvm (5943) e cargas cada segmento em

memória com loaduvm (5945) allocuvm verifica que os endereços virtuais solicitados é abaixo KERNBASE. loaduvm (1818) walkpgdir usa para encontrar o endereço físico do almemória localizada na qual a escrever cada página do segmento ELF, e Readi para ler a partir do arquivo.

loaduvm + código
loaduvm + código
código walkpgdir +
readi + código
/ Init + código
allocuvm + código
código + exec
ustack + código
argv + código
argc + código
ação copyout + código
seginit + código

O cabeçalho da seção de programas para / init, o primeiro programa de usuário criada com exec, se parece com isso:

```
# Objdump -p _init

_nisso:     formato de arquivo ELF32-i386

Programa Cabeçalho:
    Carga fora  0x00000054 0x00000000 vaddr paddr 0x00000000 alinhar 2 ** 2
         filesz 0x000008c0 memsz 0x000008cc bandeiras rwx
```

Filesz O cabeçalho de parte de programa pode ser menor do que o memsz, indicando que a distância entre eles deve ser preenchido com zeros (para C variáveis globais) em vez de ler o arquivo. Para / init, filesz é 2240 bytes e memsz é 2252 bytes, e, assim, allocuvm aloca memória física suficiente para manter 2.252 bytes, mas só lê 2240 bytes do arquivo / init.

Agora exec aloca e inicializa a pilha do usuário. Ele aloca apenas uma página pilha. Cópias Exec as cordas argumento para o topo da pilha, um por vez, registrando a ponteiros para eles em ustack. Coloca um ponteiro nulo no final da qual será o lista argv passado para principal. As primeiras três entradas na ustack são o retorno falso PC, argc e argv ponteiro.

Exec coloca uma página inacessível logo abaixo da página de pilha, de modo que os programas que tente usar mais de uma página irá falha. Esta página inacessível também permite a exec lidar com argumentos que são muito grandes; nessa situação, a função de ação copyout que exec usa para copiar argumentos para a pilha vai notar que a página de destino em não accessible, e irá retornar -1.

Durante a preparação da imagem de memória novo, se exec detecta um erro como um segmento do programa inválido, ele salta para o rótulo ruim, libera a nova imagem, e re-Acontece -1. Exec deve esperar para libertar a imagem antiga até que esteja certo de que a chamada do sistema será ter sucesso: se a imagem velha se foi, a chamada de sistema não pode retornar -1 a ele. A única error casos em exec acontecer durante a criação da imagem. Uma vez que a imagem é completa, exec pode instalar a nova imagem (5989) e sem o antigo (5990) Finalmente, exec retorna 0.

Mundo real

Como a maioria dos sistemas operacionais, xv6 usa o hardware de paginação para proteção de memória ção e mapeamento. A maioria dos sistemas operacionais fazer uso muito mais sofisticado de paginação que xv6; por exemplo, xv6 carece de paginação sob demanda a partir do disco, garfo copy-on-write, compartilhada memória, páginas preguiçosamente-atribuídas, bem como as pilhas que se estendem automaticamente. Os suportes 86 tradução de endereços usando a segmentação (ver Apêndice B), mas xv6 usa apenas segmentos para o truque comum de variáveis de execução por CPU, como proc que estão em um endereço fixo, mas têm valores diferentes em diferentes CPUs (ver seginit). Implementação

ções de (ou por thread) de armazenamento por CPU em arquiteturas não-segmento dedicaria a registar-se para segurando um ponteiro para a área de dados por CPU, mas o x86 tem tão poucos geal registra que o esforço extra necessário para usar a segmentação vale a pena.

Código CR_PSE +

Em máquinas com muita memória pode fazer sentido para usar 4 Mbyte do x86 " Páginas de super. " Pequenos páginas faz sentido quando a memória física é pequena, para permitir a atribuição cação e página para o disco com granularidade fina. Por exemplo, se um programa usa apenas 8 Kbyte de memória, dando-lhe um 4 Mbyte página física é um desperdício. Páginas maiores fazem sentido em máquinas com muita memória RAM, e pode reduzir a sobrecarga para a tabela de páginas manipulação. Xv6 usa páginas de super em um só lugar: a tabela de página inicial (1311) A matriz initialization define dois dos 1.024 PDE, em índices de zero e 960 (KERNBASE >> PDXSHIFT), deixando o outro PDEs zero. Xv6 define o bit PTE_PS nestes dois PDEs para marcá-los como páginas de super. O kernel também informa ao hardware de paginação para permitir super-páginas por set-

ting o bit CR_PSE (Página Tamanho Extension) em% cr4.

Xv6 deve determinar a configuração de RAM real, em vez de assumir 240 MB. No x 86, existem pelo menos três algoritmos comuns: o primeiro é a sondar a espaço de endereço físico procurando regiões que se comportam como memória, preservando a va- ues escritos para eles; o segundo é para ler o número de kilobytes de memória de um conhecido local 16-bit na RAM não volátil do PC; eo terceiro é olhar em BIOS memória para uma tabela de layout de memória deixou como parte das mesas de multiprocessadores. Leitura a tabela de layout de memória é complicado.

A alocação de memória foi um tema quente há muito tempo, os problemas básicos sendo cia uso ciente de memória limitada e se preparando para futuras solicitações desconhecidos; veja Knuth. Hoje as pessoas se preocupam mais com a velocidade que o espaço-eficiência. Além disso, uma mais elabo- Kernel taxa provavelmente alocar muitos tamanhos diferentes de pequenos blocos, em vez de (como em xv6) apenas blocos de 4096 bytes; um alocador verdadeira seria necessário para lidar com pequena alocação ções, bem como as grandes.

Exercícios

1. Olhe para os sistemas operacionais real para ver como eles o tamanho da memória.

2. Se xv6 não tinha usado páginas de super, o que seria a declaração de direito para en- trypgdir?

3. implementações Unix do exec tradicionalmente incluem tratamento especial para scripts shell. Se o arquivo para executar começa com o texto # !, em seguida, a primeira linha é levado para ser um pro- gram a correr para interpretar o arquivo. Por exemplo, se exec é chamado para executar myprog arg1 e primeira linha de myprog é #! / interp, então exec runs / interp com linha de comando / Arg1 myprog interp. Implementar suporte para esta convenção em xv6.

Capítulo 3

exceção
interromper

Armadilhas, interrupções e drivers

Ao executar um processo, uma CPU executa o ciclo normal de processador: ler um in- struction, avançar o contador de programa, executar a instrução, repita. Mas existem eventos em que o controle a partir de um programa do usuário deve transferido de volta para o kernel in- vez de executar a instrução seguinte. Estes eventos incluem um dispositivo de sinalização que ele quer atenção, um programa de usuário fazendo algo ilegal (por exemplo, faz referência a um ad- virtual vestido para o qual não há PTE), ou um programa de usuário pedir ao kernel para um serviço com uma chamada de sistema. Existem três principais desafios para o manejo desses eventos: 1) a kernel deve providenciar que um processador muda do modo de usuário para o modo kernel (e verso); 2) o kernel e os dispositivos devem coordenar as suas actividades paralelas; e 3) o kernel deve compreender a interface dos dispositivos bem. Enfrentar estes três desa- dificulda- requer conhecimento detalhado de hardware e programação cuidadosa, e pode resultar em código do kernel opaco. Este capítulo explica como xv6 aborda estes três desafios.

Chama de sistemas, as exceções e interrupções

Com um sistema de chamada de um programa de usuário pode pedir um serviço do sistema operacional, como vimos no fim do último capítulo. A exceção termo se refere a uma ação ilegal programa que gera uma interrupção. Exemplos de ações de programas ilegais incluem divisão por ze- ro, tenta aceder à memória para um TEP que não está presente, e assim por diante. O termo in- terrupt refere-se a um sinal gerado por um dispositivo de hardware, o que indica que ele precisa at-

lando do sistema operativo. Por exemplo, um chip de relógio pode gerar uma interrupção cada 100 ms para permitir que o kernel para implementar o compartilhamento de tempo. Como outro exemplo, quando o disco tenha lido um bloco de disco, ele gera uma interrupção para alertar o opera- sistema que o bloco está pronto para ser recuperado ing.

O kernel lida com todas as interrupções, ao invés de processos de manuseá-los, porque em maioria dos casos, apenas o kernel tem o privilégio e Estado requerido. Por exemplo, em ordem para o tempo-slice entre os processos em resposta as interrupções de relógio, o kernel deve ser in- vidos, mesmo que apenas para forçar os processos que não cooperam para produzir o processador.

Em todos os três casos, o projeto do sistema operacional deve providenciar o seguinte para acontecer. O sistema deve salvar registradores do processador para o futuro currículo transparente. O sistema deve ser configurado para execução no kernel. O sistema deve escolheu um lugar para o kernel para iniciar a execução. O kernel deve ser capaz de recuperar informações sobre o evento, por exemplo, os argumentos de chamada do sistema. Deve ser feito de forma segura; O sistema deve manter o isolamento de processos do usuário e do kernel.

Para atingir esse objetivo o sistema operacional deve estar ciente dos detalhes de como o hardware lida com chamadas de sistema, exceções e interrupções. Na maioria dos processadores estes três eventos são tratados por um único mecanismo de hardware. Por exemplo, nos x86, um

Page 34

programa invoca uma chamada de sistema, gerando uma interrupção com a instrução int.
Do mesmo modo, as excepções gerar uma interrupção também. Assim, se o sistema operativo tem um plano de tratamento de interrupção, em seguida, o sistema operacional pode lidar com as chamadas de sistema e exi percepções também.

int + código
manipulador de interrupção
armadilhas
modo kernel
modo de usuário

O plano de base é como se segue. Uma interrupção interrompe o circuito processador normal e começa a executar uma nova seqüência chamada um manipulador de interrupção. Antes de iniciar a in- manipulador terrupt, o processador salva seus registos, de modo a que o sistema operativo pode re- armazená-los quando ele retorna a partir da interrupção. Um desafio na transição para e a partir do manipulador de interrupção é que o processador deve alternar do modo de usuário para kernels nel modo, e de volta.

Uma palavra sobre a terminologia: Embora o termo oficial x86 é de interrupção, xv6 refere-se a todos estes como armadilhas, em grande parte porque era o termo usado pelo PDP11 / 40 e lá- fore é o termo Unix convencional. Este capítulo utiliza os termos armadilha e interromper in- terchangeably, mas é importante lembrar que as armadilhas são causadas pela actual processo de execução em um processador (por exemplo, o processo faz com que uma chamada de sistema e, como resultado gera uma armadilha), e as interrupções são causadas por dispositivos e pode não estar relacionada com a Atualmente processo em execução. Por exemplo, um disco pode gerar uma interrupção, quando é feita a recuperação de um bloco para um processo, mas no momento da interrupção alguma outra processo pode ser executado. Esta propriedade de interrupções faz pensar sobre as interrupções mais difícil do que pensar em armadilhas, porque as interrupções acontecem simultaneamente com outras atividades. Ambos dependem, no entanto, no mesmo mecanismo de hardware para transferir con- trolar entre o modo kernel do usuário e de forma segura, que discutiremos a seguir.

Proteção X86

O x86 tem 4 níveis de protecção, numeradas de 0 (mais de privilégio) a 3 (menor privilégio). Na prática, a maioria dos sistemas operacionais utilizam apenas 2 níveis: 0 e 3, que são então chamados modo kernel e modo de usuário, respectivamente. O nível de privilégio atual com a qual o x86 executa instruções são armazenados em% cs registo, na CPL campo.

No x86, manipuladores de interrupção estão definidos na tabela de descritores de interrupção (IDT). O IDT tem 256 entradas, cada um dando o cs% e% eip a ser usado no manuseio do interrupção correspondente.

Para fazer uma chamada de sistema no x86, um programa chama a instrução n int, onde n especifica o índice para a IDT. A instrução int executa os seguintes passos:

- Fetch o descritor n'th do IDT, onde n é o argumento de int.
- Verifique se CPL em% cs é <= DPL, onde DPL é o nível de privilégio no de- scriptor.
- Salve% esp e ss% em uma CPU-interno registros, mas apenas se o se- segmento-alvo PL <CPL do lector.
- Coloque% ss e esp% de um descritor de segmento tarefa.
- Empurre% ss.

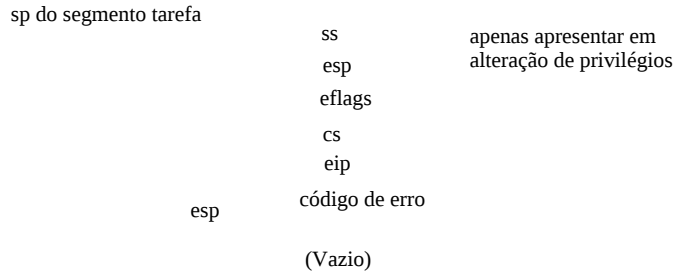


Figura 3-1 . Pilha Kernel após uma instrução int.

- Empurre% esp.
- Empurre% eflags.
- Empurre% cs.
- Empurre% eip.
- Desmarque alguns pedaços de eflags%.
- Definir% cs e eip% para os valores no descritor.

A instrução int é uma instrução complexa, e pode-se perguntar se tudo são necessárias essas ações. A CPL cheque <= DPL permite que o kernel para proibir sistems em alguns níveis de privilégio. Por exemplo, para um programa de utilizador para executar int instruction com sucesso, o DPL deve ser 3. Se o programa do usuário não tem a aproprivilégio caso disso, então a instrução int int irá resultar em 13, que é uma protecção geral culpa ção. Como outro exemplo, a instrução int não pode utilizar a pilha utilizador para guardar valores, porque o usuário não pode ter configurado uma pilha adequada, de modo que o hardware usa a pilha especificada nos segmentos de tarefas, que é configurado no modo kernel.

A Figura 3-1 mostra a pilha depois de uma instrução completa int e houve um mudança de nível de privilégio (o nível de privilégio no descritor é inferior CPL). Se o instrução int não exigiu uma mudança de nível de privilégio, o x86 não vai salvar% ss e % Esp. Após ambos os casos,% eip está apontando para o endereço especificado no descritor de table, ea instrução nesse endereço é a próxima instrução a ser executada e o primeira instrução do manipulador para int n. É o trabalho do sistema operacional para implementar esses manipuladores, e abaixo vamos ver o que xv6 faz.

Um sistema operacional pode usar a instrução iret a regressar de um instrução int ção. Ela mostra os valores salvos durante a instrução int da pilha, e currículos execução na% eip salvo.

Código: A primeira chamada de sistema

Capítulo 1 terminou com initcode.S invocar uma chamada de sistema. Vamos olhar para isso de novo

(7713) O processo levou os argumentos para uma chamada exec na pilha do processo, e colocar o número de chamada de sistema em% eax. Os números de chamada de sistema combina com as entradas no syscalls array, uma tabela de ponteiros de função (3350) Precisamos providenciar que o int instruction muda o processador do modo de usuário para o modo kernel, que o kernel do invokes a função kernel correto (ie, sys_exec), e que o kernel pode recuperar o

int + código
código iret +
int + código
initcode.S + código
código + exec
sys_exec + código
int + código
tvinit
código principal +
código idt +
vetores [i] + código

argumentos para sys_exec. As próximas subseções descreve como xv6 organiza este para chamadas do sistema, e, em seguida, vamos descobrir que podemos reutilizar o mesmo código para interrupções e exceções.

T_SYSCALL + código
Código DPL_USER +
Código DPL_USER +
switchuvm + código
alltraps + código

Código: manipuladores armadilha Assembleia

Xv6 deve configurar o hardware x86 para fazer algo sensível ao encontrar um instrução INT, o que faz com que o processador para gerar uma armadilha. O x86 permite 256 diferentes interrupções. Interrupções 0-31 são definidos para exceções de software, como a divisão er- Rors ou tentar acessar endereços de memória inválidos. Xv6 mapeia a inter- 32 hardware Rupts ao intervalo 32-63 e usos interromper 64 como a interrupção de chamadas do sistema.

Tvinit (3067) Chamado de principal, configura as 256 entradas no idt tabela. Interrupção i é tratado pelo código no endereço indicado no vectores [i]. Cada ponto de entrada é diferente, porque o x86 fornece não fornece o número armadilha para o manipulador de interrupção. Usando 256 manipuladores diferentes é a única maneira para distinguir os 256 casos.

Tvinit lida T_SYSCALL, o usuário chamada de sistema armadilha, especialmente: especifica que o portão é do tipo " armadilha ", passando um valor de 1 como segundo argumento. Portões Armadilha não fazer limpar o sinalizador FL, permitindo que outras interrupções durante o tratador de chamadas do sistema.

O kernel também estabelece a chamada de sistema privilégio portão para DPL_USER, o que permite uma programa do usuário para gerar a armadilha com uma instrução int explícito. não xv6 não permite processos para levantar outras interrupções (por exemplo, o dispositivo interrompe) com int; se tentarem, eles vai encontrar uma exceção de proteção geral, que vai para o vetor 13.

Ao alterar os níveis de proteção de usuário para o modo kernel, o kernel não deve utilizar a pilha do processo de utilizador, uma vez que não podem ser válidos. O processo usuário pode ser malicioso ou conter um erro que faz com que o usuário% esp para conter um endereço que não faz parte da memória do usuário do processo. Programas Xv6 o hardware x86 para executar um interruptor de pilha em uma armadilha através da criação de um descritor de segmento tarefa através do qual o hardware carrega um seletor de segmento de pilha e um novo valor para% esp. A função switchuvm (1773) armazena o endereço do topo da pilha do cerne do processo de utilizador no descritor segmento tarefa.

Quando ocorre uma armadilha, o hardware do processador faz a seguir. Se o processador estava sendo executado no modo de usuário, ele carrega% esp e ss% do descritor de segmento tarefa, empurra o velho e ss% esp usuário% para a nova pilha. Se o processador estava sendo executado no modo kernel, nenhuma das opções acima acontece. O processador então empurra os eflags%, % Cs, e registros% EIP. Para algumas armadilhas, o processador também empurra uma palavra erro. O processador carrega% eip e cs% a partir da entrada IDT relevante.

xv6 usa um script Perl (2950) para gerar os pontos de entrada que as entradas do IDT apontam a. Cada entrada empurra um código de erro se o processador não, empurra o núme- interrupção ber, e depois salta para alltraps.

Alltraps (3004) continua a salvar registradores do processador: ele empurra% ds,% es,% fs,

Page 37

CPU> ts.esp0
ss
esp
eflags
cs
eip
© ©
trapno
ds
es
fs
gs
eax
ECX
edx
EBX
OESP
ebp
esi
esp edi
(Vazio)

apenas apresentar em alteração de privilégios

p> KStack

Figura 3-2 . O trapframe na pilha do kernel

% gs, e os registradores de uso geral (3005-3010) O resultado deste esforço é que o
pilha do kernel agora contém um trapframe struct (0602) contendo pelo registo processador
ters no momento da armadilha (s ee Figura 3-2). O processador empurra% ss,% esp,
% eflags, cs% e% eip. O processador ou o vetor armadilha empurra um número de erro,
e alltraps empurra o resto. O quadro de interceptação contém todas as informações necessárias
para restaurar os registradores do processador do modo de usuário quando o kernel retorna à corrente
processo, de modo a que o processador possa continuar exactamente como foi iniciado quando a armadilha.
Lembre-se do Capítulo 2, que userinit construir uma trapframe com a mão para alcançar esta meta
(Ver Figura 1-3) .

alltraps + código
alltraps + código
Código SEG_KCPU +
alltraps + código
trap + código
alltraps + código

No caso da primeira chamada do sistema, a% eip salvo é o endereço da instrução
logo após a instrução int. % Cs é o selector de utilizador segmento de código. % eflags é a
conteúdo das eflags cadastre-se no ponto de executar a instrução int. Como parte de
salvar os registradores de uso geral, alltraps também economiza% eax, que contém o
número de chamada de sistema para o kernel para inspecionar mais tarde.

Agora que os registradores do processador modo de usuário são salvas, alltraps pode terminar set-
ting-se o processador para executar código do kernel C. O processador definir o seletores% cs e
% Ss antes de entrar no manipulador; alltraps define% ds e% es (3013-3015) Estabelece% fs e
% gs para apontar para o segmento de dados SEG_KCPU por CPU (3016-3018)

Uma vez que os segmentos estão definidas corretamente, alltraps pode chamar a armadilha C manipulador armadilha. Ele
empurra% esp, que aponta para o quadro de interceptação apenas construído, para a pilha como um
argumento para armadilha (3021) Em seguida, ele chama (3022) Após a armadilha retornos, alltraps pops

o argumento para fora da pilha por adição ao ponteiro de pilha (3023) e, em seguida, começa a exe-
ing o código na etiqueta trapret. Traçamos através deste código no Capítulo 2, quando o
primeiro processo de usuário ele correu para sair para o espaço do usuário. A mesma seqüência acontece aqui: pop-
ping entre o quadro de interceptação restaura os registradores de modo de usuário e, em seguida, saltos IRET
volta para o espaço do usuário.

trapret + código
código iret +
trap + código
TF-> trapno + código
T_SYSCALL + código
código + syscall
CP-> código + morto

A discussão até agora falou sobre as armadilhas que ocorrem no modo de usuário, mas armadilhas
Também pode acontecer enquanto o kernel está em execução. Nesse caso, o hardware não faz
trocar pilhas ou salvar o ponteiro de pilha ou pilha seletor de segmento; caso contrário, a mesma
etapas ocorrer como em armadilhas de modo de usuário, eo mesmo código de manipulação xv6 armadilha executa.
Quando iret depois restaura um cs modo kernel%, o processador continua a executar em
modo kernel.

trap + código
pânico + código
trap + código
código + syscall
SYS_exec + código
CP-> tf + código
código + syscall

Código: manipulador de interceptação C

Vimos na última seção que cada manipulador configura um quadro de interceptação e depois chama
a armadilha função C. Armadilha (3101) olha para o número hardware armadilha TF-> trapno para
decidir por isso que foi chamado e o que precisa ser feito. Se a armadilha é T_SYSCALL,
armadilha chama o manipulador syscall chamada de sistema. Vamos revisitar os dois CP-> matou cheques em
Capítulo 5.

Depois de verificar se uma chamada de sistema, armadilha procura por interrupções de hardware (que dis-
xingar abaixo). Para além dos dispositivos de hardware esperados, uma armadilha pode ser causada por uma
espúria interromper, uma interrupção hardware indesejado.

Se a armadilha não é uma chamada de sistema e não um dispositivo de hardware à procura de atenção,
trap supõe que ele foi causado por comportamento incorreto (por exemplo, dividir por zero), como parte do
código que estava sendo executado antes da armadilha. Se o código que causou a armadilha foi um utilizador
programa, gravuras xv6 detalhes e define CP-> matou para se lembrar de limpar o
processo de usuário. Vamos ver como xv6 faz essa limpeza no Capítulo 5.

Se fosse o kernel em execução, deve haver um bug do kernel: gravuras armadilha detalhes
sobre a surpresa e depois chama pânico.

As chamadas do sistema: Código

Para chamadas de sistema, armadilha invoca syscall (3375) Cargas SYSCALL a chamada de sistema
número do quadro armadilha, que contém o% eax guardado, e os índices na
mesas de chamada do sistema. Pela primeira chamada do sistema,% eax contém o valor SYS_exec (3207)
e syscall irá invocar a entrada SYS_exec'th da tabela de chamada de sistema, o que cor-
ponde a invocar sys_exec.

SYSCALL registra o valor de retorno da função de chamada de sistema em% eax. Quando o trap retorna ao espaço do usuário, ele irá carregar os valores da CP-> tf no regis- máquina ters. Assim, quando retorna exec, ele irá retornar o valor que o manipulador de re- chamada de sistema virado (3381.)Sistema chama convencionalmente retornar números negativos para indicar erros, números positivos para o sucesso. Se o número de chamada de sistema é inválida, syscall imprime uma erro e retorna -1.

Os últimos capítulos examinará a execução de determinadas chamadas do sistema. Este capítulo está relacionado com os mecanismos de chamadas do sistema. Há um pouco de mecanização

nismo deixou: encontrar os argumentos de chamada do sistema. As funções auxiliares Argint e argptr, argstr recuperar o argumento de chamada de sistema n'th, como um inteiro, ponteiro, ou uma string. Argint usa o espaço do usuário% esp cadastre-se para localizar o argumento n'th: pontos esp% em o endereço de retorno para o stub chamada de sistema. Os argumentos são à direita acima, em % Esp + 4. Em seguida, o argumento é a enésima% esp + 4 + 4 * n.

Argint + código
fetchint + código
p> sz + código
argptr + código
argstr + código

Argint chama fetchint para ler o valor nesse endereço de memória de usuário e escrevê-lo para * ip. fetchint pode simplesmente lançar o endereço para um ponteiro, porque o usuário eo kernel compartilham a mesma tabela de páginas, mas o kernel deve verificar se o ponteiro pelo usuário é de fato um ponteiro na parte do usuário do espaço de endereço. O kernel tem configurar o hardware de tabela de página para se certificar de que o processo não pode acessar a memória fora de sua memória privado local: se um programa usuário tenta ler ou escrever memória em um endereço de p-> sz ou acima, o processador irá causar uma armadilha de segmentação, e armadilha vai matar o processo, como vimos acima. Agora, porém, o kernel está em execução e que pode derefence qualquer endereço que o usuário pode ter passado, por isso deve verificar explicitamente que o endereço está abaixo p> sz

argptr é um propósito semelhante à Argint: ele interpreta o enésimo argumento chamada de sistema ment. chamadas argptr Argint para buscar o argumento como um inteiro e, em seguida, verifica se o inteiro como um ponteiro do usuário é de fato na parte do usuário do espaço de endereço. Note-se que dois cheques ocorrer durante uma chamada para código argptr. Em primeiro lugar, o ponteiro da pilha do usuário é verificada du- ing a busca do argumento. Em seguida, o argumento, ele próprio um ponteiro do usuário, está marcada.

argstr é o último membro da chamada de sistema argumento trio. Ele interpreta o enésimo argumento como um ponteiro. Ele garante que o ponteiro aponta em uma seqüência de NUL terminada e que a cadeia completa está localizado abaixo da extremidade da parte do utilizador do endereço espaço.

As implementações de chamadas de sistema (por exemplo, sysproc.c e sysfile.c) são tipicamente wrappers: eles decodificar os argumentos usando Argint, argptr e argstr e depois chamar as implementações reais. No capítulo 2, sys_exec usa essas funções para obter a sua ar- argumen-.

Código: Interrupções

Os dispositivos na placa-mãe pode gerar interrupções, e xv6 preciso configurar a hardware para lidar com essas interrupções. Sem o apoio dispositivo xv6 não seria utilizável; um usuário não pode digitar no teclado, um sistema de arquivos não pode armazenar dados em disco, etc. Fortu- lizmente, acrescentando interrupções e suporte para dispositivos simples não exige muito adicionalidade complexidade ai. Como veremos, as interrupções podem usar o mesmo código para chamadas de sistemas e exceções.

Interrupções são semelhantes às chamadas do sistema, exceto dispositivos gerá-los a qualquer momento. Há hardware na placa-mãe para sinalizar a CPU quando um dispositivo precisa de atenção ção (por exemplo, o usuário digitou um personagem no teclado). Devemos programar o de- vice-para gerar uma interrupção, e providenciar para que a CPU recebe a interrupção.

Vamos olhar para o dispositivo temporizador e interrupções. Gostaríamos que o hardware do temporizador utensílios para gerar uma interrupção, dizem, 100 vezes por segundo para que o kernel pode acompanhar a passagem do tempo e assim o kernel pode cronometrar-slice entre múltiplos processamento running es. A escolha de 100 vezes por segundo permite uma performance interativa decente

enquanto não inundando o processador com a manipulação de interrupções.

Como o próprio processador x86, placas-mãe têm evoluído, e da forma como inter-Rupts são fornecidos evoluiu muito. As placas de início teve uma in- programável simples controler terrupt (o chamado PIC), e você pode encontrar o código para gerenciá-lo na PI-cirq.c.

Com o advento das placas multiprocessador PC, uma nova maneira de lidar com as interrupções era necessário, porque cada CPU precisa de um controlador de interrupção para lidar com interrupções enviar a ela, e deve haver um método para o encaminhamento interrupções para processadores. Desta forma con-siste de duas partes: uma parte que está no sistema I / O (a IO APIC, ioapic.c), e uma parte que está ligado a cada processador (o APIC local, lapic.c). Xv6 é projetado para uma placa com múltiplos processadores, e cada processador deve ser programado para receber interrompe.

Para também funciona corretamente em uniprocessadores, programas Xv6 o inter- programável controler rupt (PIC) (6932) Cada PIC pode lidar com um máximo de 8 interrupções (ou seja, de-vícios) e multiplexar-los sobre o pino de interrupção do processador. Para permitir uma maior de 8 dispositivos, PICs pode ser em cascata e, normalmente, placas têm pelo menos dois. Utilização programas Xv6 INB e instruções outb o mestre para gerar IRQ de 0 a 7 e o escravo para gerar IRQ 8 a 16. Os programas Inicialmente xv6 o PIC para mascarar tudo in-terrupts. O código em timer.c define temporizador 1 e permite a interrupção temporizador no PIC (7574) Esta descrição omite alguns dos detalhes de programação do PIC. Estes Detalhes do PIC (e o IOAPIC e LAPIC) não são importantes para este texto, mas o leitor interessado pode consultar os manuais de cada dispositivo, que são referenciados no arquivos de origem.

Em multiprocessadores, xv6 deve programar o IOAPIC, eo LAPIC em cada pro-cessor. O IO APIC tem uma mesa e o processador pode programar entradas na tabela através de memória mapeada I / O, em vez de usar a INB e instruções outb. Durante inicialização, programas xv6 para mapear interrupção IRQ 0 a 0, e assim por diante, mas desactiva-los tudo. Dispositivos específicos permitir particulares interrupções e dizer a que o processador do inter-rupt deve ser encaminhado. Por exemplo, as rotas xv6 teclado interrompe para processador 0 (7516) Rotas Xv6 disco interrompe o processador para o número mais elevado no sistema, tão veremos a seguir.

O chip temporizador está dentro do LAPIC, de modo que cada processador pode receber in- temporizador terrupts independentemente. Xv6 puser em um lugar lapicinit (6651) A linha de chave é a que O temporizador de programa (6664) Esta linha de conta a LAPIC para gerar periodicamente em uma inter-rupt em IRQ_TIMER, que é IRQ 0. Linha (6693) permite que as interrupções no LAPIC uma CPU, o que fará com que a entregar as interrupções ao processador local.

Um processador pode controlar se quer receber interrupções através da bandeira IF no eflags registrar. A instrução cli desativa interrupções no processador limpando IF, e sti permite interrupções em um processador. Xv6 desativa interrupções durante a inicialização do a CPU principal (8412) e os outros processadores (1126) O programador em cada processador permite que as interrupções (2464) Para controlar o que determinado código fragmentos não são interrompidos, xv6 desativa interrupções durante estes fragmentos de código (por exemplo, ver switchuvm) (1773)

O temporizador interrompe através vector 32 (que xv6 escolheu para lidar com IRQ 0), que xv6 configuração em idtinit (1265) A única diferença entre o vector de 32 e 64 vector (o para um sistema de chamadas) é que é um vector de 32 porta de interrupção, em vez de uma porta armadilha. Inter-

in- programável
controler terrupt (PIC)
INB + código
outb + código
timer.c + código
lapicinit + código
IRQ_TIMER, + código
Se c código +
cli + código
sti + código
switchuvm + código
idtinit + código

portões Rupt limpa SE, de modo que o processador de interrupção não receber interrupções enquanto ele está a lidar com a interrupção atual. De agora em diante, até armadilha, interrompe seguir o mesmo caminho de código como chamadas e exceções do sistema, a criação de um quadro de interceptação.

Armadilha quando é chamado para uma interrupção de tempo, faz apenas duas coisas: incrementar o variável carrapatos (3063) E chamada de despertar. Este último, como veremos no capítulo 5, pode causar a interrupção para retornar em um processo diferente.

trap + código
despertador + código
motorista
bloco
setor
amortecedor
struct buf + código
B_VALID + código
Código B_DIRTY +
Código B_BUSY +
ideinit + código
código principal +
código + picenable
código + ioapicenable
IDE_IRQ + código

Drivers

Um driver é o pedaço de código em um sistema operacional que gerencia um dispositivo em particular: ele fornece manipuladores de interrupção para um dispositivo, faz com que um dispositivo para executar operações, diz-es um dispositivo para gerar interrupções, etc. código do driver pode ser complicado de escrever, porque a motorista executa simultaneamente com o dispositivo que ele gerencia. Além disso, o controlador deve compreender a interface do dispositivo (por exemplo, o que portas I / O fazer o que), e que inter-

cara pode ser complexo e mal documentadas.

O driver de disco é um bom exemplo em xv6. Os dados de cópias do disco de drivers de e para o disco. Hardware Disk apresenta tradicionalmente os dados no disco como uma seqüência numerada de blocos de 512 bytes (também chamados setores): sector 0 é o primeiro 512 bytes, o sector 1 é o seguinte, e assim por diante. Para representar setores de disco de um sistema operacional tem uma estrutura que corresponde a um sector. Os dados armazenados nesta estrutura é de- dez fora de sincronia com o disco: pode ainda não ter sido lido do disco (o disco é trabalhando nisso, mas não retornou o conteúdo do sector ainda), ou que poderia ter sido para cima datados, mas ainda não escrito. O condutor deve garantir que o resto do xv6 não recebe quando confundiu a estrutura está fora de sincronia com o disco.

Código: driver de disco

O dispositivo IDE fornece acesso a discos conectados ao IDE padrão PC con- troller. IDE agora está caindo fora de moda em favor de SCSI e SATA, mas a interface é simples e permite-nos concentrar-se sobre a estrutura geral de um condutor em vez de o de- caudas de uma determinada peça de hardware.

O driver de disco representam setores do disco com uma estrutura de dados chamada de buffer, struct buf (3500). Cada tampão representa o conteúdo de um sector de um determinado dispositivo de disco. Os campos dev e setoriais dar o número do setor dispositivo e e o campo de dados é uma cópia em memória do setor de disco.

As bandeiras acompanhar a relação entre memória e disco: a bandeira B_VALID significa que os dados tenham sido lidos em, ea bandeira B_DIRTY significa que os dados precisam ser escrito para fora. A bandeira B_BUSY é um pouco de bloqueio; isso indica que algum processo está a utilizar o tampão e outros processos não deve. Quando um buffer tem o sinalizador B_BUSY, dizemos o tampão é trancado.

O kernel inicializa o driver de disco no momento da inicialização chamando ideinit (3851) a partir de principal (1234). Ideinit chama picenable e ioapicenable para permitir que o IDE_IRQ inter- rupt (3856-3857). A chamada para picenable permite a interrupção em um único processador; ioapi- cenable permite a interrupção por um multiprocessador, mas apenas no último CPU (ncpu-1): num sistema de dois processadores, um processador trata interrupções de disco.

Em seguida, ideinit investiga o hardware do disco. Ele começa chamando idewait (3858) para esperar que o disco seja capaz de aceitar comandos. A placa-mãe PC apresenta a esta- bits de tus do hardware de disco em I / O port 0x1f7. Idewait (3833) polls os bits de estado até a pouco ocupado (IDE_BSY) é clara e o bit de pronto (IDE_DRDY) está definido.

Agora que o controlador de disco está pronto, ideinit pode verificar quantos discos são presente. Assume-se que o disco 0 está presente, porque o carregador de boot eo kernel foram ambos carregados do disco 0, mas deve verificar se há disco 1. Ele escreve a porta I / O para 0x1f6 selecione o disco 1 e, em seguida, aguarda um tempo para o bit de status para mostrar que o disco está pronto (3860-3867). Se não, ideinit assume o disco está ausente.

Após ideinit, o disco não é usado novamente até que o buffer cache chama iderw, que actualiza um tampão trancada, como indicado pelas bandeiras. Se B_DIRTY está definido, iderw escreve o tampão para o disco; se B_VALID não está definido, iderw lê o tampão do disco.

ideinit + código
idewait + código
Código IDE_BSY +
Código IDE_DRDY +
iderw + código
Código B_DIRTY +
B_VALID + código
iderw + código
votação
espera ocupada
iderw + código
idestart + código
iderw + código
iderw + código
trap + código
ideintr + código
código inSL +

Disk acessos normalmente levam milissegundos, um longo tempo para um processador. A bota loader emite disco comandos de leitura e lê os bits de status várias vezes até que os dados são pronto. Esta votação ou espera ocupada é bom em um carregador de inicialização, o que não tem nada me- ter que fazer. Em um sistema operativo, no entanto, é mais eficiente para deixar um outro processo executado na CPU e organizar para receber uma interrupção quando a operação de disco tem com- efectuados. Iderw leva essa última abordagem, mantendo a lista de pedidos de disco pendentes em um fila e usando interrupções para saber quando cada solicitação foi concluída. Embora iderw mantém uma fila de pedidos, o controlador de disco IDE simples só pode lidar com uma operação de cada vez. O driver de disco mantém o invariante que enviou o tampão na frente da fila para o hardware de disco; os outros estão simplesmente à espera a sua vez.

Iderw (3954) adiciona o tampão b para o fim da fila (3967-3971). Se o tampão é na frente da fila, iderw devem enviá-lo para o hardware do disco ligando idestart (3924-3926). Caso contrário, o tampão irá ser iniciado quando os tampões à frente dele são tomadas de cuidados.

Idestart (3875) questões ou a ler ou a escrever para o dispositivo do tampão e do setor, de acordo com os sinalizadores. Se a operação é uma gravação, idestart deve fornecer os dados agora (3889) ea interrupção será sinal de que os dados foram gravados no disco. Se a operação

ção é uma leitura, a interrupção será sinal de que os dados estão prontos, eo manipulador lerá lo. Note-se que iderw tem conhecimento detalhado sobre o dispositivo IDE, e escreve-se o direito valores com as portas direitas. Se qualquer uma dessas declarações outb é errado, o IDE irá fazer algo diferente do que o que nós queremos. Obter estes detalhes à direita é uma das razões por isso que escrever drivers de dispositivo é um desafio.

Após ter adicionado o pedido para a fila e começou a se necessário, deve iderw esperar pelo resultado. Como discutido acima, a votação não faz uso eficiente do CPU. Em vez disso, iderw dorme, esperando que o manipulador de interrupção para registro no buffer de bandeiras que a operação é feita (3978-3979) Embora este processo está dormindo, vai xv6 agendar outros processos para manter a CPU ocupada.

Eventualmente, o disco vai terminar o seu funcionamento e provocar uma interrupção. trap vontade chamar ideintr para lidar com isso (3924) Ideintr (3902) consulta o primeiro tampão na fila para descobrir qual operação estava acontecendo. Se o buffer estava sendo lido eo disco controlador tem espera de dados, ideintr lê os dados para o buffer com inSL (3915-

Page 43

3917) Agora o buffer está pronto: ideintr define B_VALID, limpa B_DIRTY, e acorda qualquer processo de dormir no tampão (3919-3922) Finalmente, ideintr deve passar a próxima espera- ção tampão para o disco (3924-3926)

B_VALID + código Código B_DIRTY +

Mundo real

Apoiar todos os dispositivos em uma placa-mãe PC em toda a sua glória é muito trabalho, torna- causa existem muitos dispositivos, os dispositivos têm muitos recursos, e o protocolo torna- dispositivo tween e motorista pode ser complexa. Em muitos sistemas operacionais, os motoristas to- gether conta para mais de código no sistema operacional do que o kernel do núcleo.

Drivers de dispositivos reais são muito mais complexos do que o driver de disco neste capítulo, mas as idéias básicas são as mesmas: tipicamente dispositivos são mais lentos do que CPU, de modo que o hardware ware utiliza interrupções para notificar o sistema operacional de mudanças de status. Disco Modern controladores normalmente aceitar várias solicitações de disco em aberto em um tempo e até mesmo re- encomendá-los a fazer uso mais eficiente do braço do disco. Quando os discos eram mais simples, op- sistema operacio- frequentemente reordenadas o pedido fila si.

Muitos sistemas operacionais têm drivers para discos de estado sólido, porque eles fornecem acesso muito mais rápido aos dados. Mas, apesar de um estado sólido funciona de maneira muito diferente de um rígido mecânico tradicional, ambos os dispositivos oferecem interfaces baseados em blocos e leitura ing / escrita dos blocos em um disco de estado sólido é ainda mais caro do que a leitura / gravação RAM.

Outro hardware é surpreendentemente similar à discos: buffers de dispositivo de rede segurar Pack- ets, buffers de dispositivo de áudio conter amostras de som, placa de vídeo buffers armazenar dados de vídeo e sequências de comandos. Dispositivos-discos de alta largura de banda, placas gráficas e de rede cartões-costumam usar acesso direto à memória (DMA) em vez do I explícito / O (inSL, saída sl) neste driver. DMA permite que o disco ou outros controladores de acesso directo à física memória. O controlador fornece o endereço físico do dispositivo de campo de dados do buffer e as cópias de dispositivos diretamente de ou para a memória principal, interrompendo uma vez que a cópia é com- pleta. Usando DMA significa que a CPU não está envolvido em tudo na transferência, o que pode ser mais eficiente e é menos desgastante para caches de memória da CPU.

A maioria dos dispositivos deste capítulo utilizado I instruções de E / S para programá-los, o que reflecte a natureza destes dispositivos mais antigos. Todos os dispositivos modernos são programados usando de memória mapeada I / O.

Alguns motoristas de mudar dinamicamente entre votação e interrupções, pois o uso de interrupções pode ser caro, mas usando polling pode introduzir atraso até que o pro- motorista cessos um evento. Por exemplo, para um controlador de rede que recebe uma explosão de pacotes, pode mudar de interrupções para polling, uma vez que sabe que mais pacotes devem ser pro- cessed e é menos caro para processá-los usando polling. Uma vez que não há mais pacotes precisarem de ser processados, o condutor pode voltar para interrupções, de modo que será alertando ed imediatamente quando um novo pacote chega.

O driver IDE encaminhado interrupções estaticamente para um processador particular. Alguns motoristas tem um algoritmo sofisticado para encaminhamento interrompe a processador de modo que a carga de pacotes de processamento é bem equilibrado, mas boa localização é conseguido também. Por exemplo, uma driver de rede pode organizar para entregar interrupções para os pacotes de uma rede de conexão ção para o processador que está gerenciando este respeito, embora interrupções para pacotes de

outra conexão são entregues a outro transformador. Este caminho pode ficar bastante so-
phisticated; por exemplo, se algumas conexões de rede são de curta duração, enquanto outros são
longa vida e do sistema operacional quer manter todos os processadores ocupado para alcançar alta
taxa de transferência.

Se o processo de usuário lê um arquivo, os dados para esse arquivo é copiado duas vezes. Em primeiro lugar, é copiado do disco para a memória do kernel pelo condutor, e, posteriormente, ele é copiado a partir do kernel espaço para o espaço do usuário pela chamada de sistema de leitura. Se o processo de usuário, em seguida, envia os dados na rede, em seguida, os dados são copiados novamente duas vezes: uma vez no espaço de usuário para o kernel espaço e do espaço do kernel para o dispositivo de rede. Para apoiar os pedidos que baixa latência é importante (por exemplo, um servidor Web de páginas estáticas da Web), sistemas operacionais usam caminhos de código especiais para evitar estas muitas cópias. Como um exemplo, no mundo real opera-
sistemas ing, buffers normalmente correspondem ao tamanho de hardware, de modo que somente leitura cópias pode ser mapeado para o espaço de endereço de um processo usando o hardware de paginação, sem qualquer cópia.

Exercícios

1. Defina um ponto de interrupção na primeira instrução de syscall () para pegar a primeira chamada de sistema (Por exemplo, br syscall). Que valores estão na pilha neste momento? Explique a saída de x / 37x $ esp no que breakpoint com cada valor rotulado como para o que é (por exemplo,% salvo ebp por armadilha, trapframe.eip, espaço de rascunho, etc.).

2. Adicione uma nova chamada de sistema

3. Adicione um driver de rede

Capítulo 4

trancar

Bloqueando

Xv6 roda em multiprocessadores, computadores com várias CPUs executando código inde-
dentemente. Essas múltiplas CPUs operar em um único espaço de endereço físico e share

estruturas de dados; xv6 deve introduzir um mecanismo de coordenação para mantê-los a partir de interfering uns com os outros. Mesmo em um único processador, xv6 deve usar algum mecanismo para manter manipuladores de interrupção de interferir com código não-interrupção. Xv6 usa a mesma conceito de baixo nível para tanto: a fechadura. Um bloqueio fornece exclusão mútua, garantindo que apenas uma CPU de cada vez pode manter o bloqueio. Se xv6 só acessa uma estrutura de dados ao mesmo tempo mantendo um bloqueio em particular, então xv6 pode ter certeza de que apenas uma CPU de cada vez é acessoing a estrutura de dados. Nesta situação, dizemos que o bloqueio protege os dados trutura ture.

O resto deste capítulo explica por que xv6 precisa fechaduras, como xv6 implementa-las, e como ele usa-los. Uma observação importante será que, se você olhar para uma linha de código em xv6, você deve estar se perguntando se há um outro processador que poderia mudar o intendiam comportamento da linha (por exemplo, porque um outro processador também está executando essa linha ou outra linha de código que modifica uma variável compartilhada) e o que aconteceria se um manipulador de interrupção correu. Em ambos os casos você tem que ter em mente que cada linha de C pode haver várias instruções de máquina e, portanto, outro processador ou uma interrupção pode zombar em torno do meio de uma instrução C. Você não pode presumir que as linhas de código no página são executadas sequencialmente, nem você pode supor que uma única instrução C será exeatomicamente bonito. Simultaneidade faz raciocínio sobre a correção muito mais dicil.

Condições de corrida

Como um exemplo de por que precisamos de fechaduras, considerar vários processadores compartilhando um único disco, tal como o disco IDE em xv6. O driver de disco mantém uma lista vinculada da saída pé solicitações de disco (3821) e os processadores podem adicionar novos pedidos à lista concomitanatualmente (3954) Se não houvesse pedidos simultâneos, você pode implementar a lista ligada do seguinte modo:

```
1   lista struct {
2      dados int;
3      lista struct * próximo;
4   };
5
6   struct lista * list = 0;
7
8   vazio
9   inserir (dados int)
10  {
11     lista struct * l;
```

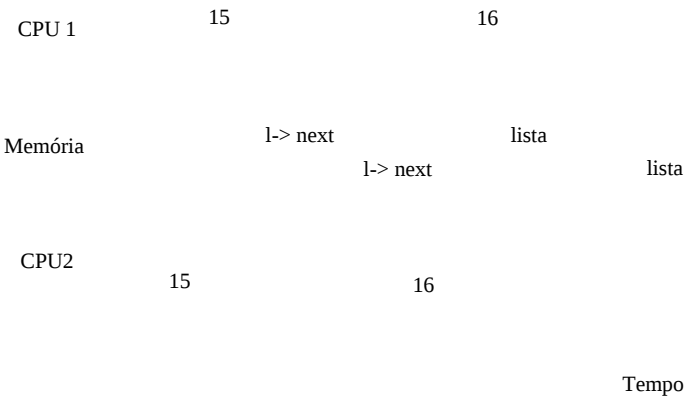



Figura 4-1 . Exemplo corrida

```
12                                                    condição de corrida
13     l = malloc (sizeof * l);
14     l-> Dados = dados;
15     l-> next = lista;
16     list = l;
17  }
```

Provando essa implementação correta é um exercício típico de uma estrutura de dados e algoclasse ritmos. Mesmo que esta aplicação pode ser provada correta, não é, pelo menos,

não em um multiprocessador. Se dois processadores diferentes executar inserção, ao mesmo tempo, ele poderia acontecer que tanto executar a linha 15, antes ou executa 16 (ver F IGURA 4-1) . Se isso acontecer, haverá agora dois nós da lista com jogo seguinte ao antigo valor de lista. Quando as duas atribuições a lista acontecer na linha 16, o segundo irá sobre- escrever o primeiro; o nó envolvido na primeira missão será perdido. Este tipo de problema é chamado de uma condição de corrida. O problema com raças é que eles dependem o momento exato das duas CPUs envolvidos e como as suas operações de memória são or- pe- o sistema de memória, e são, consequentemente, difícil de reproduzir. Por exem- ple, acrescentando instruções de impressão durante a depuração inserção pode alterar o calendário do execução o suficiente para fazer a corrida desaparecer.

A maneira típica de evitar corridas é a utilização de uma fechadura. Locks garantir a exclusão mútua, de modo que apenas uma CPU pode executar inserção de cada vez; isso faz com que o cenário acima im- possível. A versão corretamente travado do código acima adiciona apenas algumas linhas (não numerada):

```
6     struct lista * list = 0;
      bloqueio listlock struct;
7
8     vazio
9     inserir (dados int)
10    {
11       lista struct * l;
```



```
12                                                    struct
      adquirir (& listlock);                            código spinlock +
13    l = malloc (sizeof * l);                        adquirir código +
14    l-> Dados = dados;                              atômico
15    l-> next = lista;
16    list = l;
      release (& listlock);
17    }
```

Quando dizemos que um bloqueio protege os dados, que realmente queremos dizer que o bloqueio protege alguns coleção de invariantes que se aplicam aos dados. Invariantes são propriedades de trutura de dados turas que são mantidos em todas as operações. Normalmente, uma operação é o comportamento correto depende das invariantes sendo verdadeiro quando a operação começa. A operação pode violar temporariamente os invariantes, mas deve restabelecer-los antes de terminar. Por exem- amplo, no caso lista ligada, o invariável é que os pontos de lista no primeiro nó no lista e que os próximos pontos de campo de cada nó no próximo nó. A implementação de insert viole esta invariante temporariamente: linha 13 cria um novo elemento da lista l com o intenção de que l seja o primeiro nó na lista, mas a próxima ponteiro l's não aponta para o próximo nó na lista ainda (restabelecida na linha 15) e lista não aponta para l ainda (Restabelecida na linha 16). A condição de corrida que vimos anteriormente aconteceu porque um segundo CPU executado código que dependia das invariantes lista, enquanto eles estavam (tem- riamente) violados. O uso adequado de um bloqueio garante que apenas um CPU de cada vez pode opera- comeu na estrutura de dados, de modo que nenhum CPU executar uma operação de estrutura de dados quando invariantes da estrutura de dados não segure.

Código: Locks

Xv6 de representa um bloqueio como um spinlock struct (1401). O campo crítico na estrutura está bloqueado, uma palavra que é zero quando o bloqueio está disponível e diferente de zero quando é realizada. Logicamente, xv6 deve adquirir um bloqueio ao executar um código como

```
21    vazio
22    adquirir (struct spinlock * lk)
23    {
24       para (;;) {
25          if (! lk-> bloqueado) {
26             lk-> bloqueado = 1;
27             break;
28          }
29       }
30    }
```

Infelizmente, esta aplicação não garante a exclusão mútua em um moderno multiprocessador. Pode acontecer que dois (ou mais) CPUs atingir simultaneamente linha 25, veja que lk-> bloqueado é zero, e em seguida, ambos pegar o bloqueio por executar as linhas 26 e

27. Neste ponto, duas CPUs diferentes manter o bloqueio, o que viola a exclusão mútua propriedade Sion. Em vez de ajudar-nos a evitar condições de corrida, esta implementação de adquirem tem a sua própria condição de corrida. O problema aqui é que as linhas 25 e 26 de exe- ed ações como separados. Para que a rotina acima para ser correto, as linhas 25 e 26 deve executar em um atômica passo (ou seja, indivisíveis).

Page 48

Para executar essas duas linhas atomicamente, xv6 depende de um hardware especial 386 in- struction, xchg (0569). Em uma operação atômica, xchg troca uma palavra na memória com o conteúdo de um registrador. O adquirem função (1474) repete esta instrução xchg em um laço; cada iteração lê lk-> bloqueado e atomicamente define-o como 1 (1483). Se o bloqueio é held, lk-> bloqueado já será 1, de modo que o xchg retorna 1 e o ciclo continua. Se os XCHG retorna 0, no entanto, adquirir adquiriu com sucesso o lock-bloqueado foi 0 e é agora um-para que o loop pode parar. Uma vez que o bloqueio é adquirido, adquirir registros, para depuração, o CPU e rastreamento de pilha que adquiriu o bloqueio. Quando um processo adquire bloquear e se esqueça de liberá-lo, esta informação pode ajudar a identificar o culpado. Estes campos de depuração são protegidos pela fechadura e só deve ser editado, mantendo o bloquear.

xchg + código
adquirir código +
+ código de liberação
bloqueios recursiva
iderw + código

A liberação função (1502) é o oposto de adquirir: ele limpa a depuração campos e, em seguida, libera o bloqueio.

Fechaduras modularidade e recursiva

O projeto do sistema se esforça para, abstrações modulares limpas: o melhor é quando um chamador faz não precisa saber como um receptor especial implementa a funcionalidade. Locks interferir com esta modularidade. Por exemplo, se uma CPU mantém um bloqueio particular, não pode chamar qualquer f função que vai tentar readquirir que lock: desde o chamador não pode liberar o bloqueio un- til f retornos, se f tenta adquirir o mesmo bloqueio, ele irá girar para sempre, ou impasse.

Não há soluções transparentes que permitem que o chamador e receptor para esconder que bloqueios que eles usam. Uma solução comum, transparente, mas insatisfatória é recursiva fechaduras, que permitem que um receptor para readquirir um bloqueio já realizada pelo seu interlocutor. O problema com esta solução é que os bloqueios recursivas não pode ser usado para proteger invariantes. Depois de in- sert chamado adquirir (e listlock) acima, pode-se supor que nenhuma outra função detém o bloqueio, que nenhuma outra função é no meio de uma operação de lista, e mais impor- tantly que todas as invariantes lista de espera. Em um sistema com bloqueios recursiva, inserção pode as- sume nada depois que ele chama de adquirir: talvez adquirir conseguido apenas porque um dos chamador de inserção já realizou o bloqueio e foi no meio de editar os dados da lista estrutura. Talvez os invariantes segurar ou talvez não. A lista já não protege eles. Bloqueios são tão importantes para proteger os chamadores e chamados um do outro como eles são para proteger diferentes processadores entre si; bloqueios recursiva desistir desse propriedade.

Como não há solução transparente ideal, devemos considerar fechaduras parte do especificação da função. O programador deve providenciar que a função não invoca um função f mantendo um bloqueio que as necessidades de f. Locks forçar-se em nossa abstração ções.

Código: Usando bloqueios

Xv6 é cuidadosamente programado com bloqueios para evitar condições de corrida. Um exemplo simples é no controlador IDE (3800). Conforme mencionado no início do capítulo, iderw (3954) tem uma fila de pedidos e processadores de disco pode adicionar novos pedidos à lista concomitan- atualmente (3969). Para proteger esta lista e outros invariantes no driver, iderw adquire o

Page 49

idelock (3965) e liberta no final da função. Exercício 1 explora como desenca- ger a condição de corrida que vimos no início do capítulo, movendo o ac-

idelock + código

quire para depois da manipulação fila. Vale a pena experimentar o exercício porque vai deixar claro que não é assim tão fácil de acionar a raça, o que sugere que é difícil para encontrar corrida-condições bugs. Não é improvável que xv6 tem algumas raças.

A parte mais difícil sobre o uso de bloqueios é decidir quantos bloqueios de usar e quais os dados e invariantes cada bloqueio protege. Existem alguns princípios básicos. Em primeiro lugar, qualquer um tempo variável pode ser escrita por um processador central, ao mesmo tempo que um outro processador pode ler ou escrever, um bloqueio deve ser introduzido para manter as duas operações a partir de sobreposição. Em segundo lugar, lembre-se que os bloqueios proteger invariantes: se um invariante envolve múltiplos dados estruturas, tipicamente todas as estruturas necessitam de ser protegidos por um único bloqueio para garantir o invariante é mantida.

As regras acima dizer quando são necessários bloqueios, mas não dizem nada sobre quando os bloqueios são desnecessárias, e é importante para a eficiência não para bloquear muito, porque os bloqueios reduzir paralelismo. Se a eficiência não era importante, então pode-se utilizar um processador único computador e nenhuma preocupação de todo sobre bloqueios. Para proteger as estruturas de dados do kernel, ele seria suficiente para criar um único bloqueio que deve ser adquirido ao entrar no kernel e lançado em sair do kernel. Muitos sistemas operacionais monoprocessados ter sido con- convertidos para rodar em multiprocessadores usando essa abordagem, às vezes chamado de " kernel do gigante bloquear ", mas a abordagem de prejudicar verdadeira concorrência: apenas uma CPU pode executar no núcleo de cada vez. Se o kernel faz qualquer computação pesada, seria mais eficiente a utilização de um conjunto maior de mais bloqueios de grão fino, de modo que o kernel possa executar em mul- tiple CPUs simultaneamente.

Em última análise, a escolha de granularidade do bloqueio é um exercício de programação paralela. Xv6 usa uma estrutura de dados grosseiro alguns bloqueios específicos; por exemplo, xv6 utiliza uma única fechadura proteger a tabela de processos e seus invariantes, que são descritos no Capítulo 5. Um abordagem mais granulação fina seria ter um bloqueio por entrada na tabela de processos de modo threads que trabalham em diferentes entradas na tabela de processos podem ocorrer em paralelo. No entanto, ele complica as operações que têm invariantes sobre a mesa processo todo, uma vez que pode ter que tirar vários bloqueios. Felizmente, os exemplos de xv6 vontade ajudar a transmitir como usar fechaduras.

Ordenação de bloqueio

Se um caminho de código através do kernel deve tirar vários bloqueios, é importante que todos caminhos de código adquirir os bloqueios na mesma ordem. Se não o fizerem, há um risco de mortos- bloquear. Digamos que dois caminhos de código em xv6 precisa fechaduras A e B, mas o caminho do código 1 adquire bloqueios na ordem A e B, e o outro código adquire-los na ordem B e A. Esta situação pode resultar em um impasse, porque o caminho do código 1 pode adquirir a trava A e antes que ele adquire bloqueio B, caminho de código 2 poderia adquirir a trava B. Agora nem caminho de código pode prosseguir, porque o caminho do código 1 precisa bloquear B, que o caminho do código 2 detém, e caminho de código 2 necessidades bloquear A, que detém caminho de código 1. Para evitar esses impasses, todos os caminhos de código deve adquirir bloqueios na mesma ordem. Evitar impasse é outro exemplo que ilustra por que travas devem ser parte da especificação de uma função: o chamador deve invocar funções em uma ordem consistente de modo a que as funções de aquisição de bloqueios na mesma ordem.

Page 50

Porque xv6 usa bloqueios de granulação grossa e xv6 é simples, xv6 tem poucos lock-fim cadeias. A cadeia mais longa é de apenas dois de profundidade. Por exemplo, ideintr mantém o bloqueio ide ao chamar de despertar, que adquire o bloqueio ptable. Há um número de outros exemplos envolvendo o sono e despertar. Estas ordenações aconteceu porque o sono e excitação tem uma invariante complicado, como discutido no Capítulo 5. No sistema de arquivo há uma série de exemplos de cadeias de dois, porque o sistema de arquivos deve, por exem- amplo, adquirir um bloqueio em um diretório e o bloqueio em um arquivo no diretório para desvincular a arquivo de seu diretório pai corretamente. Xv6 sempre adquire os bloqueios na ordem primeiro diretório pai e, em seguida, o arquivo.

ideintr + código
despertador + código
código + ptable
+ código do sono
despertador + código
código de carrapatos +
sys_sleep + código
tickslock + código
iderw + código
idelock + código
ideintr + código
pushcli + código
popcli + código
cli + código
pushcli + código
popcli + código
popcli + código
cli + código
sti + código
adquirir código +
xchg + código
+ código de liberação
popcli + código
xchg + código

Manipuladores de interrupção

Xv6 usa bloqueios para proteger manipuladores de interrupção rodando em uma CPU de não-interrupção código de acesso aos mesmos dados em outra CPU. Por exemplo, a interrupção de temporizador ament- Dler (3114) incrementos de carrapatos, mas pode ser outra CPU no sys_sleep ao mesmo tempo, utilizando a variável (3473) O tickslock bloqueio sincroniza o acesso pelos dois CPUs para a variável única.

As interrupções podem causar concorrência, mesmo em um único processador: se as interrupções idêntica + código
abled, código do kernel pode ser interrompido a qualquer momento para executar um manipulador de interrupção em seu lugar.
Suponha iderw segurou o idelock e depois foi interrompido para executar ideintr. Ideintr
tentaria bloquear idelock, vê-lo se realizou, e esperar por ele para ser lançado. Neste situa-
ção, idelock nunca será liberado somente iderw pode liberá-lo, e não vai iderw
continuar a correr até ideintr retorna-so o processador, e, eventualmente, o todo
sistema, vai impasse.

Para evitar esta situação, se um bloqueio é usado por um manipulador de interrupção, um processador deve
nunca segure que trava com as interrupções habilitadas. Xv6 é mais conservador: nunca detém
qualquer fechadura com interrupções habilitado. Ele usa pushcli (1555) e popcli (1566) para gerenciar uma
pilha de " operações 'desabilitar interrupções' (CLI é a instrução x86 que desativa inter-
Rupts. Adquirir chamadas pushcli antes de tentar adquirir um bloqueio (1476). E solte chamadas
popcli após a liberação do bloqueio (1521). Pushcli (1555) e popcli (1566) são mais de
apenas wrappers em torno cli e sti: eles são contados, de modo que é preciso duas chamadas de pop
cli para desfazer duas chamadas para pushcli; Desta forma, se o código adquire duas fechaduras diferentes, inter-
Rupts não será reativada até que ambos os bloqueios foram liberados.

É importante que adquirir chamada pushcli antes do xchg que possam adquirir a
tranca (1483). Se os dois foram invertidos, haveria alguns ciclos de instrução quando o
bloqueio foi realizado com as interrupções habilitado, e uma interrupção infelizmente cronometrado faria
impasse do sistema. Da mesma forma, é importante que a chamada libertação popcli apenas após o
xchg que libera o bloqueio (1483)

A interação entre os manipuladores de interrupção e código não-interrupção fornece uma
bom exemplo porque bloqueios recursiva são problemáticas. Se xv6 utilizadas fechaduras recursiva (a secção
ond adquirir em uma CPU é permitida se a primeira aquisição aconteceu naquela CPU também), então
manipuladores de interrupção poderia correr enquanto o código não-interrupção é em uma seção crítica. Este
poderia criar o caos, já que quando o manipulador de interrupção é executado, invariantes que o manipulador
invoca pode ser violada temporariamente. Por exemplo, ideintr (3902) assume que o
lista ligada com pedidos pendentes é bem formado. Se xv6 teria usado recursiva

fechaduras, em seguida, pode ser executado enquanto ideintr iderw está no meio de manipular o iderw + código
lista ligada, e a lista ligada vai acabar em um estado incorreto. + código de liberação

Ordenação de Memória

Este capítulo tem assumido que os processadores iniciar e concluir as instruções no
ordem em que aparecem no programa. Muitos processadores, no entanto, executar in-
truções fora de ordem para alcançar um maior desempenho. Se uma instrução leva muitos
ciclos para completar, um processador pode querer emitir a instrução inicial para que ele possa
sobreposição com outras instruções e evitar barracas de processador. Por exemplo, um processador
pode observar que em uma seqüência de série de instrução A e B não são dependentes de cada
e começar outra instrução B antes de A de modo que ele irá ser completada quando o processador
conclui A. Concorrência, no entanto, pode expor este reordenamento de software, o que
levar a um comportamento incorreto.

Por exemplo, pode-se perguntar o que acontece se a liberação apenas atribuído 0 a lk-
> Bloqueado, em vez de usar xchg. A resposta a esta questão não é clara, porque diferença
ent gerações de processadores x86 fazer diferentes garantias sobre a ordenação de memória.
Se lk-> bloqueado = 0, foram autorizados a ser re-ordenada dizer depois popcli, do que adquirir pode
quebrar, porque para outro segmento interrupções seria ativado antes de um bloqueio é liberado.
Para evitar depender de especificações do processador pouco claras sobre encomenda de memória, xv6 leva
nenhum risco e usa xchg, que os transformadores devem garantir não para reordenar.

Mundo real

Primitivas de concorrência e programação paralela são áreas ativas de pesquisa, torna-
causar programação com fechaduras ainda é um desafio. É melhor usar bloqueios como a base
para construções de alto nível, como filas sincronizados, embora xv6 não faz isso. Se
você programa com bloqueios, é aconselhável usar uma ferramenta que tenta identificar raça condi-
ções, porque é fácil de se perder uma invariante que requer o bloqueio.

Programas em nível de usuário precisa bloqueios também, mas em xv6 aplicações têm um segmento de ex-
execuç~ao e processos não compartilham memória, e por isso não há necessidade de bloqueios em xv6
aplicações.

É possível implementar fechaduras sem instruções atómicas, mas é caro,

ea maioria dos sistemas operacionais utilizam instruções atômicas.

Instruções atômicas não são livres ou quando um bloqueio está contente. Se um processador tem um bloqueio em cache em seu cache local, e um outro processador deve adquirir o bloqueio, em seguida, a instrução atômica para atualizar a linha que mantém o bloqueio deve mover a linha de o cache do processador de um para o cache do outro processador e, talvez, a anulação de quaisquer outras cópias da linha de cache. Buscando uma linha de cache de cache do outro processador podem ser ordens de magnitude mais caro do que buscar uma linha de um cache local.

Para evitar as despesas associadas com fechaduras, muitos sistemas operacionais utilizam livre-lock estruturas de dados e algoritmos, e tentar evitar operações atômicas nesses algoritmos. Por exemplo, é possível implementada uma lista de ligação como o de início de o capítulo que não requer bloqueios durante a lista de pesquisas, e uma instrução atômica para inserir um item em uma lista.

Exercícios

1. livrar-se fora do xchg em adquirir. explicar o que acontece quando você executa xv6?

2. mover a adquirir em iderw para antes de dormir. há uma corrida? por que você não observá-lo quando arrancar xv6 e executar stressfs? aumentar a seção crítica com um laço manequim; oque você vê agora? explicar.

3. fazer questão homework postou.

4. Definir um pouco nas bandeiras de um buffer não é uma operação atômica: o processador faz um cópia de bandeiras num registo, edita o registo, e escreve-lo de volta. Assim, é importante que os dois processos não estão escrevendo para bandeiras, ao mesmo tempo. xv6 edita o bit B_BUSY apenas mantendo buflock mas edita as bandeiras B_VALID e B_WRITE sem segurar quaisquer bloqueios. Por que isso é seguro?

Page 53

múltiplo

Capítulo 5

Scheduling

Qualquer sistema operacional é susceptível de ser executado com mais processos do que o computador tem processadores, e assim por algum plano é necessário para compartilhar o tempo os processadores entre a pro- processos. Um plano ideal é transparente para os processos do usuário. Uma abordagem comum é a pro- vide cada processo com a ilusão de que ele tem seu próprio processador virtual, e ter a sistema operacional múltiplos processadores virtuais multiplex em um único processador físico. Neste capítulo Como xv6 multiplexes um processador entre vários processos.

Multiplexing

Xv6 adota essa abordagem multiplexação. Quando um processo está esperando por re- disco quest, xv6 coloca para dormir, e horários outro processo para ser executado. Além disso, xv6 US- ing temporizador interrompe para forçar um processo de parar de correr em um processador após um fixo quantidade de tempo (100 ms), de modo que ele pode marcar um outro processo no processador. Este multiplexing cria a ilusão de que cada processo tem sua própria CPU, assim como xv6 utilizado o alocador de memória e de página hardware tabelas para criar a ilusão de que cada processo tem sua própria memória.

Multiplexing Implementando tem alguns desafios. Em primeiro lugar, como mudar de um processo para o outro? Xv6 usa o mecanismo padrão de troca de contexto; embora a idéia é simples, o código para implementar é tipicamente entre o código mais opaca em um sistema operativo. Em segundo lugar, como fazer troca de contexto de forma transparente? Xv6 utiliza o técnica padrão de usar o manipulador de interrupção do timer para conduzir mudanças de contexto. Terceiro, muitas CPUs pode estar trocando entre processos em simultâneo, e um plano de bloqueio é necessário para evitar corridas. Em quarto lugar, quando um processo que já saiu da sua memória e outros recursos devem ser liberados, mas não pode fazer tudo isso em si, porque (por exemplo) que não pode libertar a sua própria pilha de kernel enquanto ainda usá-lo. Xv6 tenta resolver estes problemas como sim- ply quanto possível, mas, no entanto, o código resultante é complicado.

xv6 deve proporcionar meios para processos de coordenar entre si. Por exem- ple, um processo pai pode ter de esperar por um de seus filhos para sair, ou um processo lendo em uma tubulação pode ter de esperar por algum outro processo para escrever o pipe. Bastante do que fazer a CPU resíduos processo de espera, verificando repetidamente se o desejar evento aconteceu, xv6 permite que um processo desistir da CPU e dormir à espera de uma evento, e permite que outro processo para acordar o primeiro processo up. É necessário cuidado para evitar raças que resultam na perda de notificações de eventos. Como um exemplo destes problemas e sua solução, este capítulo analisa a implementação de tubos.

Código: Contexto comutação

Tal como mostrado na F igura 5-1, para alternar entre os processos, xv6 executa dois tipos de

Page 54

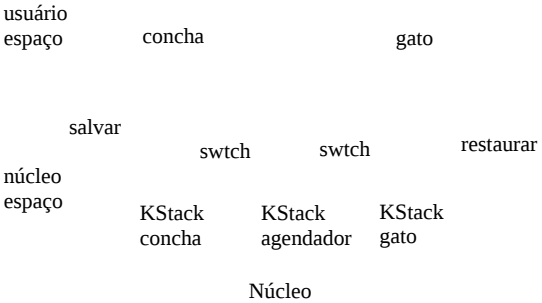


Figura 5.1 A mudança de um processo de usuário para outro. Neste exemplo, xv6 corre com uma CPU (e assim, um segmento agendador).

contexto muda em um nível baixo: a partir de kernel discussão de um processo para a atual CPU do rosca scheduler, e do segmento agendador para thread do kernel de um processo. xv6 nev- er muda diretamente a partir de um processo de espaço do usuário para outro; isto acontece por meio de um transição user-kernel (chamada de sistema ou de interrupção), a mudança de contexto para o programador, e um mudança de contexto para kernel discussão de um novo processo, e um retorno armadilha. Nesta seção, vamos exemplo a mecânica de alternar entre uma lista de discussão do kernel e um fio de programador.

Todo processo xv6 tem sua própria pilha de kernel e registrar definido, como vimos no capítulo 2. Cada CPU tem uma rosca agenda separada para uso quando ele está executando o de horários uler em vez de kernel discussão de qualquer processo. Mudar de um segmento para outro in- volve salvar registros de CPU do velho de rosca, e restaurando registros previamente salvos de o novo segmento; o fato de que esp% e% eip são salvos e restaurados significa que o CPU irá mudar pilhas e mudar o que o código está sendo executado.

swtch não conhece diretamente sobre threads; ele só salva e restaura conjuntos de registros, chamados contextos. Quando é tempo para o processo de desistir da CPU, ker- do processo rosca nel chamará swtch para salvar seu próprio contexto e retornar ao contexto planejador. Cada contexto é representado por uma estrutura de contexto *, um ponteiro para uma estrutura armazenado em a pilha do núcleo envolvido. Swtch leva dois argumentos: a estrutura de contexto ** velho e contexto struct * novo. Ele empurra o registro atual CPU para a pilha e salva o stack pointer in * idade. Cópias Então swtch novo para% esp, pops salvo anteriormente regis- ters, e retorna.

Em vez de seguir o programador em swtch, vamos sim seguir o nosso processo de usuário de volta. Vimos no Capítulo 3 que uma possibilidade, no final de cada interrupção é que armadilha chama rendimento. Rendimento em chamadas de volta sched, que chama swtch para salvar o atual contexto em cesso> contexto e mude para o contexto agendador salvo anteriormente em CPU> agendador (2516)

Swtch (2702) começa por carregar seus argumentos na pilha para os registradores% EAX e% edx (2709-2710) swtch deve fazer isso antes de mudar o ponteiro da pilha e não pode mais acessar os argumentos via% esp. Então swtch empurra o estado registo, criando um estrutura de contexto na pilha atual. Só o receptor-save registros precisam ser salvos; a Convenção sobre a x86 é que estes são% ebp,% ebx,% esi,% ebp e% esp.

Swtch empurra os quatro primeiros explicitamente (2713-2716) ele salva o último implicitamente como a contexto struct * escrito para * velho (2719) Há um registo mais importante: a contador de programa% eip foi salva pela instrução de chamada que invocou swtch e é na pilha logo acima% ebp. Após ter salvo o contexto de idade, swtch está pronto para restaurar o novo. Ele move o ponteiro para o novo contexto para o ponteiro da pilha (2720) A nova pilha tem a mesma forma que o antigo que swtch apenas deixou-a nova pilha era o antigo em uma chamada anterior a swtch-so swtch pode inverter a seqüência de re- armazenar o novo contexto. Ela mostra os valores para edi%,% esi,% ebx, e% ebp e depois retornos (2723-2727) Porque swtch mudou o ponteiro da pilha, os valores restaurados e o endereço de instrução retornou ao são os do novo contexto.

No nosso exemplo, Sched chamado swtch para mudar para CPU> scheduler, a CPU per- contexto planejador. Esse contexto foi salvo com o telefonema de programador para swtch (2478) Quando o swtch fomos traçando retorna, ele retorna para não sched mas de horários uler, e sua stack pointer pontos no agendador pilha da CPU atual, não initproc das pilha de kernel.

swtch + código
contextos
contexto struct + código
trap + código
código + rendimento
código + sched
swtch + código
cpu-
> Código + agendador
swtch + código
swtch + código
código + sched
swtch + código
cpu-
> Código + agendador
swtch + código
código + agendador
swtch + código
ptable.lock + código
código + sched
+ código do sono
exit + código
código + sched
swtch + código
cpu-
> Código + agendador
código + agendador
ptable.lock + código
swtch + código
ptable.lock + código
swtch + código
código + rendimento

Código: Scheduling

A última seção olhou os detalhes de baixo nível de swtch; Agora vamos dar swtch como um dada e examinar as convenções envolvidos na mudança do processo para o Agendador e de volta a processar. Um processo que pretende dar-se a CPU deve adquirir a pro- cesso bloqueio de tabela ptable.lock, liberar quaisquer outros bloqueios que está segurando, atualize seu próprio estado (Cesso> estado), e em seguida, chamar sched. Rendimento (2922) segue esta convenção, assim como o sono e de saída, que examinaremos mais tarde. Sched double-checks essas condições (2507- 2512) e, em seguida, uma implicação destas condições: desde um bloqueio é mantido, a CPU deve estar em execução com interrupções desabilitadas. Finalmente, as chamadas SCHED swtch para salvar o atual contexto em cesso> contexto e mude para o contexto do planejador em CPU> planejador.

Retornos swtch na pilha do programador como se swtch do programador haviam retornado (2478)O programador continua o loop for, encontra um processo a correr, muda para isso, e o ciclo repete-se.

Acabamos de ver que xv6 detém ptable.lock entre as chamadas para swtch: o chamador swtch já deve manter o bloqueio, e controle do bloqueio passa para a comutação de a código. Esta convenção é incomum com fechaduras; a convenção típico é o fio que adquire um bloqueio também é responsável de libertar o bloqueio, o que faz com que seja mais fácil para raciocinar cerca de correção. Para troca de contexto é necessário quebrar a convenção típica porque ptable.lock protege invariantes em campos de estado e de contexto do processo que não são verdadeiras durante a execução em swtch. Um exemplo de um problema que pode surgir se ptable.lock não foram realizadas durante swtch: a CPU diferente pode decidir executar a processo depois rendimento tinha definido seu estado para RUNNABLE, mas antes swtch causou a parar usando sua própria pilha kernel. O resultado seria duas CPUs em execução na mesma pilha, que não pode estar certo.

A lista de discussão do kernel sempre desiste de seu processador em sched e sempre muda para o mesmo local no programador, que (quase) sempre muda para um processo no sched. Assim, se fosse para imprimir os números de linha onde xv6 interruptores tópicos, seria de observar o seguinte padrão simples: (2478) (2516) (2478) (2516)E assim por diante. O mento

Page 56

mentos em que esta mudança estilizado entre duas threads acontece às vezes são re- referidos como co-rotinas; Neste exemplo, sched e planejador são co-rotinas de entre si.

coroutines
código + sched
código + agendador
swtch + código
código + sched
forkret + código
ptable.lock + código
código + agendador
ptable.lock + código
Código + RUNNABLE
switchuvm + código
swtch + código
ptable.lock + código
código + rendimento
Código + RUNNABLE
código + agendador
ptable.lock + código
ptable.lock + código
exit + código
esperar + código

Há um caso quando swtch do programador para um novo processo não acabar em sched. Vimos neste caso no Capítulo 2: quando um novo processo está prevista em primeiro lugar, tornar- gins na forkret (2533)Forkret só existe para honrar esta convenção, liberando a ptable.lock; de outra forma, o novo processo pode começar a trapret.

Scheduler (2458)executa um loop simples: encontrar um processo a correr, executá-lo até que ele pare, repita. planejador detém ptable.lock para a maioria de suas ações, mas libera o bloqueio (E permite explicitamente interrupções), uma vez em cada iteração do seu ciclo exterior. Este é im- portante para o caso especial em que este CPU está ocioso (pode encontrar nenhum processo RUNNABLE). Se um programador de marcha lenta em loop com o bloqueio realizado continuamente, nenhuma outra CPU que foi a execução de um processo jamais poderia realizar uma troca de contexto ou qualquer sistema relacionada com processo chamada, e, em particular, nunca poderia marcar um processo como RUNNABLE de modo a quebrar o marcha lenta CPU fora do seu ciclo de programação. A razão para permitir interrupções periodicamente sobre uma CPU em marcha lenta é que pode haver nenhum processo RUNNABLE porque os processos (por exemplo, o shell) estão à espera de I / O; Se o planejador deixou interrupções desativado todo o tempo, o I / O nunca iria chegar.

O planejador faz um loop sobre a tabela de processos em busca de um processo executável, um que tem p> estado == RUNNABLE. Uma vez que encontra um processo, ele define o atual por CPU processo proc variável, muda para a tabela de páginas do processo com switchuvm, marca o processo como sendo executado, e em seguida, chama swtch para começar a executá-lo (2472-2478)

Uma maneira de pensar sobre a estrutura do código de programação é que ele arranja para impor um conjunto de invariantes sobre cada processo, e mantém ptable.lock sempre que os invariantes não são verdadeiras. Uma invariante é que, se um processo está em execução, as coisas devem ser configurado para que o rendimento de um interrupção do timer pode alternar corretamente afastado do processo; isso significa que os registros de CPU deve manter os valores de registro do processo (ou seja, não são, na verdade, em um contexto),% CR3 deve se referir a pagetable do processo,% esp deve re- fer a pilha do kernel do processo para que swtch pode empurrar registros corretamente, e proc deve se referir a proc ranhura do processo []. Outra invariante é que, se um processo está Executável, as coisas devem ser configurado para que agendador de uma CPU ociosa pode executá-lo; este significa que p> contexto deve manter as variáveis de segmento do kernel do processo, que não CPU é executar na pilha do kernel do processo, que não CR3% da CPU se refere ao processo de tabela de página, e que não proc de CPU refere-se ao processo.

Manter as invariantes acima é a razão pela qual xv6 adquire ptable.lock em uma thread (muitas vezes em rendimento) e libera o bloqueio em um segmento diferente (o programador segmento ou outro próximo segmento kernel). Uma vez que o código começou a modificar a corrida estado do processo para torná-lo RUNNABLE, deve manter o bloqueio até que tenha terminado a restauração as invariantes: o ponto de lançamento correcto mais antigo é depois agendador pára de usar o pro- mesa e página de cesso limpa proc. Da mesma forma, uma vez planejador começa a converter um processo executável para execução, o bloqueio não pode ser liberado até o fio do kernel é completamente corrida (após a swtch, por exemplo, em rendimento).

ptable.lock protege outras coisas também: atribuição de IDs e livre de processo ranhuras tabela processo, a interação entre a saída e esperar, a máquina para evitar perdida

wakeups (ver secção seguinte) e, provavelmente, outras coisas também. Talvez valha a pena pensar

sobre se as diferentes funções do ptable.lock poderia ser dividido, certamente para clareza e talvez para o desempenho.

seqüência
coordenação
condicional
sincronização

Sono e despertar

Fechaduras ajudar CPUs e processos de evitar a interferência com o outro, e programação ajuda processos compartilham uma CPU, mas até agora não temos abstrações que tornam mais fácil para processos para se comunicar. Sono e wakeup preencher esse vazio, permitindo um processo para sono à espera de um evento e outro processo para despertá-lo uma vez que o evento tem hap- ceu. Sono e despertar são freqüentemente chamados de coordenação em seqüência ou condicional mecanismos de sincronização, e há muitos outros desses mecanismos na op- literatura sistemas Microsoft®.

Para ilustrar o que queremos dizer, vamos considerar uma simples fila de produtor / consumidor. Esta fila é semelhante ao utilizado pelo condutor IDE para sincronizar um processador e driver de dispositivo (ver Capítulo 3), mas abstrai todos os códigos específicos de IDE de distância. A fila permite um processo para enviar um ponteiro com um outro processo diferente de zero. Assumindo que existe apenas um emissor e um receptor e eles executam em diferentes CPUs, este exe- tação é correta:

```
100     struct q {
101        void * ptr;
102     };
103
104     void *
105     enviar (struct q * q, void * p)
106     {
107        while (q-> ptr! = 0)
108           ;
109        q-> ptr = p;
110     }
111
112     void *
113     recv (struct q * q)
114     {
115        void * p;
116
117        while ((p = q-> ptr) == 0)
118           ;
119        q-> ptr = 0;
120        retornar p;
121     }
```

Enviar voltas até que a fila está vazia (ptr == 0) e, em seguida, coloca o ponteiro p no fila. Receb voltas até que a fila não é vazio e leva o ponteiro para fora. Quando executado em diferentes processos, enviar e recv tanto editar q-> ptr, mas enviar apenas escreve para o ponteiro quando é zero e recv só escreve para o ponteiro quando é diferente de zero, então eles não pisar em outro.

A implementação acima pode ser correta, mas é caro. Se o remetente envia raramente, o receptor vai passar a maior parte do seu tempo a girar no circuito hop- enquanto ção de um apontador. CPU do pôde encontrar trabalho mais produtivo se houvesse um

215 216
teste dorme esperar para despertar para sempre
recv

Tempo

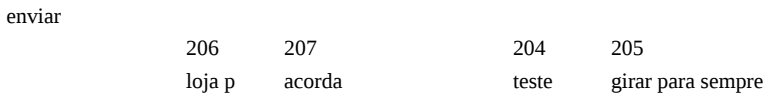


Figura 5-2 . Exemplo perdido problema de despertar

caminho para o receptor para ser notificado quando o envio tinha entregue um ponteiro.

+ código do sono
despertador + código
código chan +
esperar canal

Vamos imaginar um par de chamadas, dormir e despertar, que o trabalho da seguinte forma. Sleep (chan) dorme no valor arbitrário chan, denominado canal de espera. Puts sono O processo de chamada para dormir, liberando a CPU para outras tarefas. Despertar (chan) acorda todos os processos que dormem no chan (se houver), fazendo com que seu sono chama para voltar. Se não pro- processos estão à espera de chan, despertador não faz nada. Nós podemos mudar o cação fila mentação de usar o sono e despertar:

```
201   void *
202   enviar (struct q * q, void * p)
203   {
204      while (q-> ptr! = 0)
205         ;
206      q-> ptr = p;
207      wakeup (q); / * Wake recv * /
208   }
209
210   void *
211   recv (struct q * q)
212   {
213      void * p;
214
215      while ((p = q-> ptr) == 0)
216         sono (q);
217      q-> ptr = 0;
218      retornar p;
219   }
```

Receb agora dá-se a CPU em vez de fiação, o que é bom. No entanto, verifica-se por não ser fácil de projetar sono e wakeup com esta interface sem sofrendo do que é conhecido como o "wake up perdido " problema '(ver Figura 5-2). Supor que recv descobre que q-> ptr == 0 na linha 215 e decide chamar o sono. Antes recv pode dormir (por exemplo, o seu processador recebeu uma interrupção e o processador está rodando o in- manipulador terrupt, atrasando temporariamente a chamada para o sono), enviar corre em outra CPU: ele muda q-> ptr para ser diferente de zero e chamadas de despertar, que não encontra os processos de dormir e, portanto, não faz nada. Agora recv continua executando na linha 216: ele chama o sono e vai dormir. Isso causa um problema: recv está dormindo à espera de um ponteiro que tem al-

pronto chegou. O próximo envio vai dormir à espera de recv para consumir o ponteiro na a fila, altura em que o sistema será bloqueada.

num impasse
sono. + código
+ código do sono
+ código do sono

A raiz do problema é que a invariante que recv só dorme quando q-> ptr == 0 é violada por send correr apenas no momento errado. Uma maneira incorreta de proteger o invariante seria modificar o código para RECV como se segue:

```
300   struct q {
301      struct bloqueio spinlock;
302      void * ptr;
303   };
304
305   void *
306   enviar (struct q * q, void * p)
307   {
308      adquirir (e q-> Bloqueio);
309      while (q-> ptr! = 0)
310         ;
311      q-> ptr = p;
312      wakeup (q);
313      release (& q-> Bloqueio);
314   }
315
316   void *
317   recv (struct q * q)
```

```
318     {  void * p;
319
320
321        adquirir (e q-> Bloqueio);
322        while ((p = q-> ptr) == 0)
323           sono (q);
324        q-> ptr = 0;
325        release (& q-> Bloqueio);
326        retornar p;
327     }
```

Esta solução protege o invariante, porque quando se vai chamar o sono o processo ainda detém o q-> Bloqueio e enviar adquire que travam antes de chamar de despertar. sono não vai perca a despertar. No entanto, esta solução tem um impasse: quando recv vai dormir ele segura a fechadura q-> bloqueio, e o remetente irá bloquear ao tentar adquirir esse bloquear.

Esta aplicação incorreta deixa claro que, para proteger a invariante, devemos alterar a interface de sono. O sono deve ter como argumento o bloqueio que o sono pode liberação somente após o processo de chamada está dormindo; isso evita o despertar perdeu na ex- amplo acima. Uma vez que o processo de chamada é acordado de novo dormir readquire o bloqueio antes retornando. Nós gostaríamos de ser capaz de ter o seguinte código:

```
400     struct q {
401        struct bloqueio spinlock;
402        void * ptr;
403     };
404
405     void *
406     enviar (struct q * q, void * p)
```


Page 60

```
407     {                                                    + código do sono
408        adquirir (e q-> Bloqueio);                        despertador + código
409        while (q-> ptr! = 0)                              SONO + código
410           ;                                              código + sched
411        q-> ptr = p;                                      Código + RUNNABLE
412        wakeup (q);                                       ptable.lock + código
413        release (& q-> Bloqueio);                         despertador + código
414     }                                                    ptable.lock + código
415
416     void *
417     recv (struct q * q)
418     {
419        void * p;
420
421        adquirir (e q-> Bloqueio);
422        while ((p = q-> ptr) == 0)
423           sono (q, e q-> Bloqueio);
424        q-> ptr = 0;
425        release (& q-> Bloqueio);
426        retornar p;
427     }
```

O fato de que recv detém q-> bloqueio impede enviar de tentar acordá-lo torna- verificação tween de recv de q-> ptr e seu apelo para dormir. Naturalmente, o processo de recebimento melhor não realizar q-> Bloqueio enquanto ele está dormindo, já que impediria o remetente de acordando-o, e conduzir a um impasse. Então, o que nós queremos é que o sono atomicamente liberar q-> Bloqueio e colocar o processo de recebimento para dormir.

A implementação completa emissor / receptor também iria dormir no envio quando espera- ção para um receptor para consumir o valor de um envio anterior.

Código: Sono e despertar

Vamos olhar para a implementação de sono e wakeup em xv6. A idéia básica é para ter um sono marcar o processo atual como dormir e depois chamar sched para liberar o processador; despertar procura por um processo de dormir no ponteiro dado e marca como RUNNABLE.

Dorme (2553) começa com algumas checagens: deve haver um processo atual (2555) e dormir deve ter sido passado um bloqueio (2558-2559) Então dormir adquire PT- able.lock (2568) Agora, o processo de ir dormir detém tanto ptable.lock e lk. Segurando lk foi necessário o chamador (no exemplo, recv): ele garantiu que nenhuma outra processo (no exemplo, um envio de corrida) poderia iniciar uma chamada de despertar (chan). Agora isso

sono detém ptable.lock, é seguro para liberar lk: algum outro processo pode iniciar uma chamada à excitação (chan), mas de despertar não será executado até que ele possa adquirir ptable.lock, por isso deve aguardar até que o sono tenha acabado de colocar o processo de dormir, mantendo a despertar de perder o sono.

Há uma complicação menor: se lk é igual a & ptable.lock, e depois dormir faria impasse tentar adquiri-lo como & ptable.lock e depois liberá-lo como lk. Neste caso, sono considera a adquirir e liberar a anular-se mutuamente e ignora-los entirely (2567). Por exemplo, esperar (2403) chamadas dormir com & ptable.lock.

Page 61

Agora que o sono detém ptable.lock e nenhum outro, ele pode colocar o processo para dormir por gravar o canal de sono, alterando o estado do processo, e chamando sched (2573-2575).

Em algum momento depois, um processo será chamada de despertar (chan). Acorda adquire PTable.lock e chama wakeup1, que faz o trabalho real. É importante que a reactivação segurar o ptable.lock tanto porque está a manipular estados do processo e porque, tal como acabamos de ver, ptable.lock garante que o sono e despertar não falta um do outro. Wakeup1 é uma função separada, porque às vezes o programador precisa para executar um despertar quando ele já detém a ptable.lock; vamos ver um exemplo disso mais tarde. Wakeup1 (2603) varre a tabela de processos. Quando ele encontra um processo em dormir estado com um chan correspondente, ele muda de estado desse processo para RUNNABLE. A próxima vez que o Scheduler é executado, ele vai ver que o processo está pronto para ser executado.

(2603)

ptable.lock + código
wakeup1 + código
despertador + código
SONO + código
código chan +
Código + RUNNABLE
pipewrite + código
piperead + código

Despertar sempre devem ser chamados, mantendo um bloqueio que impede a observação de seja qual for a condição de ativação estiver; No exemplo acima, esse bloqueio é q-> bloqueio. O argumento completo para por que o processo de dormir não vai perder a excitação é que em tudo tempos de antes de verificar a condição até depois que ele está dormindo, ele possua ou o bloqueio com a condição de ou o ptable.lock ou ambos. Desde executa despertar enquanto segura ambos os bloqueios, a ativação deve executar, quer perante os potenciais cheques dorminhoco a condição, ou após o dormente potencial tenha completado colocando-se a dormir.

Às vezes, é o caso de que vários processos estão dormindo no mesmo canal; por exemplo, mais do que um processo de tentativa de ler a partir de um tubo. A única chamada para Wake se vai acordá-los todos. Um deles será executado primeiro e adquirir o bloqueio que dormir foi chamado com, e (no caso de tubos) ler o que está à espera de dados no tubo. Os outros processos vai descobrir que, apesar de estar acordado, não há dados para serem lidos. Do seu ponto de vista, o despertar foi " espúria ", e eles devem dormir novamente. Para esta razão sono é sempre chamado dentro de um loop que verifica a condição.

Chamadores de sono e despertar pode usar qualquer número mutuamente conveniente como o canal; na prática xv6 muitas vezes usa o endereço de uma estrutura de dados do kernel envolvido em de espera, tal como um tampão de disco. Nenhum dano é feito se duas utilizações do sono / wakeup accidentally escolher o mesmo canal: eles vão ver wakeups espúrias, mas looping como descrito acima vai tolerar esse problema. Muito do charme de sono / despertar é que ele é leve (sem necessidade de criar estruturas de dados especiais para atuarem como canais de sono) e fornece uma camada de engano (chamadores não precisa saber o que eles processo específico estão interagindo com).

Código: Pipes

A fila simples que usamos no início deste capítulo era um brinquedo, mas xv6 contém dois reais filas que usa o sono e despertar para sincronizar os leitores e escritores. Uma é no Driver IDE: processos adicionar solicitações de disco para uma fila e, em seguida, chama o sono. O intermanipulador rupt usa despertador para alertar o processo que o seu pedido foi concluído.

Um exemplo mais complexo é a implantação de tubos. Vimos a interface para tubos no Capítulo 0: bytes escritos para uma extremidade de um tubo são copiados em um no-kernel tampão e, em seguida, pode ser lida para fora da outra extremidade do tubo. Capítulos futuros examina o apoio do sistema de arquivos em torno de tubos, mas vamos olhar agora para a implementação ções de pipewrite e piperead.

Page 62

Cada tubo é representado por uma estrutura de tubo, que contém um bloqueio e um conjunto de dados tubo struct + código
tampão. Os campos nLeia e nwrite contar o número de bytes lidos e gravados Código + RUNNABLE
para o tampão. O buffer envolve: o próximo byte escrito após buf [PIPESIZE-1] esperar + código
é buf [0], mas as contagens não enrole. Esta convenção permite que o display implementação Código ZOMBIE +
tinguir um buffer cheio (nwrite == nLeia + PIPESIZE) a partir de um nwrite buffer vazio == struct código proc +
nLeia), mas isso significa que a indexação para o buffer deve usar buf [nLeia% PIPESIZE] p> pai + código
em vez de apenas buf [nLeia] (e da mesma forma para nwrite). Vamos supor que as chamadas para ptable.lock + código
piperead e pipewrite acontecer simultaneamente em dois processadores diferentes.

Pipewrite (6080) começa com a aquisição de fechamento do tubo, o que protege as contagens,
os dados, e seus invariantes associados. Piperead (6101) em seguida, tenta adquirir o bloqueio
também, mas não pode. Ele gira em adquirir (1474) aguardando o bloqueio. Enquanto piperead esperas,
pipewrite laços ao longo dos bytes sendo escrito-addr [0], addr [1], ..., addr [n-1] -
adição de cada um para o tubo, por sua vez (6094) Durante este ciclo, pode acontecer que o
preenchimento do buffer (6086) Neste caso, as chamadas pipewrite alerta para alertar qualquer leitores para dormir
o fato de que não há espera de dados no buffer e depois dorme em & p> nwrite que esperar
para um leitor a tomar alguns bytes para fora do buffer. Lançamentos sono p-> bloqueio como parte de
colocando processo de pipewrite para dormir.

Agora que p> Bloqueio está disponível, piperead consegue adquiri-lo e começar a correr
a sério: ele descobre que p> nLeia = p> nwrite! (6106) (Pipewrite fui dormir torna-
causar p> nwrite == p> nLeia + PIPESIZE (6086) Para que ele desce até o loop for,
cópias de dados para fora do tubo (6113-6117) E incrementos nLeia pelo número de bytes
copiado. Que muitos bytes estão agora disponíveis para a escrita, assim chamadas piperead wakeup (6118)
para acordar algum escritor de dormir antes de retornar ao seu interlocutor. Despertar encontra um processo
dormir em & p> nwrite, o processo que estava em execução pipewrite mas parou quando
o buffer preenchido. Ele marca esse processo como RUNNABLE.

O código de tubulação usa canais de sono separados para leitor e escritor (p> nLeia e
p-> nwrite); isso pode tornar o sistema mais eficiente no caso improvável de que há
muitos leitores e escritores de espera para o mesmo tubo. O código de tubulação dorme dentro
um loop de verificar o estado de sono; se houver vários leitores ou escritores, mas a todos os
primeiro processo de acordar vai ver a condição ainda é falsa e dormir novamente.

Código: Aguarde, de saída, e matar

O sono e despertar pode ser usado em vários tipos de situações envolvendo uma condição que
pode ser verificado precisa ser esperado. Como vimos no Capítulo 0, um processo pai pode
Chamada em espera para aguardar uma criança para sair. Em xv6, quando uma criança sai, ele não morre ime-
tamente. Em vez disso, ele muda para o estado do processo ZOMBIE até que as chamadas de pais esperar para
saber da saída. O pai é então responsável por liberar a memória associada
com o processo de preparação e o proc estrutura para reutilização. Cada estrutura de processo
mantém um ponteiro para seu pai na p> pai. Se as saídas dos pais antes da criança, a ini-
cial processo de inicialização adota a criança e espera que ele. Esta etapa é necessária para certificar-se
que algum processo limpa após a criança quando ele sai. Todas as estruturas de processo são
protegido por ptable.lock.

Wait começa com a aquisição ptable.lock. Em seguida, ele verifica a tabela de processos procura
para crianças. Se espera conclui que o processo atual tem filhos, mas que nenhum deles

ter saído, ele chama o sono de esperar por um dos filhos para sair (2439) e loops. exit + código
Aqui, o bloqueio está sendo lançado em sono é ptable.lock, o caso especial vimos acima. código + sched
Sair adquire ptable.lock e então acorda pai do atual processo (2376) p> KStack + código
Isto pode parecer prematuro, já que a saída não tem marcado o atual processo como um zumbi p> PGDIR + código
ainda, mas é seguro: embora o pai está agora marcado como RUNNABLE, o loop na espera swtch + código
não pode ser executado até lançamentos de saída ptable.lock chamando sched para entrar no scheduler,
assim que esperar não pode olhar para o processo de sair até que o estado foi definido para ZOMBIE
(2388) Antes de sair reprograma, ele reparents todos os filhos do processo de saída, passando
los para a initproc (2378-2385) Finalmente, as chamadas de saída sched a abandonar a CPU.

Agora, o planejador pode optar por executar o pai do processo de saída, que está dormindo
em espera (2439) A chamada para dormir retornos segurando ptable.lock; esperar rescans o pro-
mesa cesso e encontra o filho terminou com o estado == ZOMBIE. (2382) Ele registra a criança de
pid e depois limpa a proc struct, liberar a memória associada ao pro-
cesso (2418-2426)

O processo de criança poderia ter feito a maior parte da limpeza durante a saída, mas é importante que o processo pai ser o único a liberar p> KStack e p-> PGDIR. quando o criança corre saída, sua pilha fica na memória alocada como p> KStack e ele usa seu próprio pagetable. Eles só podem ser liberados após o processo filho terminar a execução de a última vez chamando swtch (via sched). Esta é uma das razões que o procedimento previsto no agendador dure é executado em sua própria pilha ao invés de na pilha do thread que chamou sched.

Exit permite que um aplicativo para encerrar em si; matar (2625) permite que um aplicativo encerrar outro processo. Matança de execução tem dois desafios: 1) o a-ser-morto processo pode ser executado em outro processador e deve desligar a sua stack para a sua planejador do processador antes xv6 pode denunciá-lo; 2) o a-ser-morto pode estar em sono, segurando recursos do kernel. Para enfrentar esses desafios, matar não faz muito: ele é executado através da tabela de processos e conjuntos p-> mortos para o processo a ser eliminado e, se for dormir, ele acorda-lo. Se o processo a-ser-morto está sendo executado em outro processador, ele será entrar no kernel em algum momento em breve: ou porque chama a chamada de sistema ou um interrupt ocorre (por exemplo, a interrupção do timer). Quando o processo a-ser-morto deixa o kernel novamente, cheques armadilha se p> mortos está definido, e em seguida, o processo exige saída, terminando eleself.

Se o processo a-ser-morto está em repouso, a chamada à excitação fará com que a torna-toprocesso a correr e retornar de sono morto. Isto é potencialmente perigoso porque o processo retorna de sono, mesmo que a condição está esperando pode não ser verdade. Xv6 é cuidadosamente programado para usar um loop while em torno de cada chamada para dormir, e testes nesse loop while se p> mortos está definido, e, em caso afirmativo, retorna ao seu interlocutor. O chamador deve também verificar se p> mortos está definido, e deve retornar ao seu chamador se definir, e assim por diante. Mostraria finalmente aliado o processo se desenrola a sua stack para armadilha, e armadilha irá verificar p> mortos. Se ele estiver definido, o processo chama saída, que encerra em si. Vemos um exemplo do processo de controlo das p-> Morto em um loop while em torno de sono na implementação de tubos (6087).

Há um loop while que não vá para p> mortos. O driver ide (3979) invoca mediatamente dormir novamente. A razão é que ele é garantido para ser acordado enfazer com que ele está à espera de uma interrupção de disco. E, se ele não espera para a interrupção de disco, xv6 pode ficar confuso. Se um segundo processo chama iderw antes da inter- excelente rupt acontece, então ideintr vai acordar que segundo processo, em vez de o processo

Página 64

que foi originalmente espera para a interrupção. Esse segundo processo vai acreditar que tem recebeu os dados que ela estava esperando, mas que recebeu os dados que o primeiro processo estava a ler!

round robin
inversão de prioridades
comboios
rebanho trovejante

Mundo real

O planejador xv6 implementa uma política de escalonamento simples, que funciona cada processo, por sua vez. Esta política é chamado de round robin. Sistemas operacionais real implementar políticas mais sofisticadas que, por exemplo, permitem que os processos de ter prioridades. O idéia é que um processo de alta prioridade executável será preferido pelo programador durante um segmento de baixa prioridade executável. Essas políticas podem se tornar complexo rapidamente porque lá são muitas vezes competindo objetivos: por exemplo, a operação pode também querer garantee equidade e de alto rendimento. Além disso, as políticas de complexos pode levar a unintendinterações ed como inversão de prioridade e comboios. Inversão de prioridade pode acontecer quando uma baixa prioridade e de alta prioridade do processo de share um bloqueio, que quando accessário pelo processo de baixa prioridade pode fazer com que o processo de alta prioridade para não correr. A longo comboio pode formar quando muitos processos de alta prioridade estão à espera de uma low-prioriprocesso ty que adquire um bloqueio compartilhado; uma vez que um comboio formou eles podem persistir por longo período de tempo. Para evitar esses tipos de problemas de mecanismos adicionais são necessário em programadores sofisticados.

Sono e despertar são um método de sincronização simples e eficaz, mas há muitos outros. O primeiro desafio em todos eles é evitar os " " ativamentos perdidas problema que vimos no início do capítulo. O sono do Unix kernel original simplesmente desativada interrupções, que foram suficientes porque Unix correu em um sistema com uma única CPU. Porque xv6 roda em multiprocessadores, ele adiciona um bloqueio explícito para dormir. FreeBSD de msleep leva a mesma abordagem. Sono Plan 9 de usa uma função callback que é executado com o bloqueio do agendamento realizado pouco antes de ir dormir; a função serve como uma última verificação minuto da condição de sono, para evitar wakeups perdidas. O kernel do Linux de sono utiliza uma fila de processo explícito, em vez de um canal de espera; a fila tem seu próprio bloqueio interno.

Digitalização de toda a lista de processos em alerta para processos com uma correspondência chan é ineficiente. A melhor solução é substituir o chan tanto no sono e wakeup com um estrutura de dados que contém uma lista de processos que dormem em que a estrutura. Plano 9 do sono Hora de Acordar e que estrutura um ponto de encontro ou Rendez. Muitas bibliotecas de rosca refer para a mesma estrutura que uma variável de estado; nesse contexto, as operações de dormir e despertar são chamados de espera e de sinal. Todos estes mecanismos compartilham a mesma flavor: a condição do sono é protegido por algum tipo de bloqueio caiu durante atomicamente dorme.

A implementação de despertar acorda todos os processos que estão aguardando em uma parcanal espe-, e que poderia ser o caso de que muitos processos estão à espera de que parcanal espe-. O sistema operacional irá horários todos estes processos e eles vão corrida para verificar o estado de sono. Processos que se comportam desta forma são, por vezes, chamado um rebanho de trovão, e é melhor evitar. A maioria das variáveis de condição tem dois primitivas para despertar: sinal, que acorda um processo, e transmissão, que acorda todos os processos em espera.

Página 65

Semáforos são outro mecanismo de coordenação comum. Um semáforo é uma valor inteiro com duas operações, incremento e decremento (ou para cima e para baixo). Isto é aways possível para incrementar um semáforo, mas o valor do semáforo não é permitido cair abaixo de zero: um decréscimo de um semáforo de zero dorme até que outro processo incrementos o semáforo, e, em seguida, essas duas operações cancelam. O valor inteiro corresponde tipicamente a uma contagem real, tal como o número de bytes disponíveis em um tubo tampão ou o número de crianças que um zombie processo tem. Usando uma contagem explícita como parte da abstração evita o " despertar perdeu " problema: há uma contagem explícita do número de ativamentos que tenham ocorrido. A contagem também evita o espúrio problemas de ativação e Thundering Herd.

sinal de + código

Finalizando processos e limpá-los introduz muita complexidade em xv6. Na maioria dos sistemas operativos é ainda mais complexa, porque, por exemplo, o a-enprocesso de mortos pode ser bem dentro do sono kernel, e descontrair sua pilha requer muita programação cuidadosa. Muitos sistema operacional desanuviar a pilha usando explícita mecanismos de manipulação de exceção, como longjmp. Além disso, existem outros eventos que podem causar um processo de dormir para ser acordado, embora os eventos é esperando por ainda não aconteceu. Por exemplo, quando um processo está a dormir, outra processo pode enviar um processo it.signalto retornará da chamada de sistema interrompida com o valor -1 e com o código de erro definido para EINTR. O aplicativo pode verificar se há esses valores e decidir o que fazer. Não Xv6 não suporta sinais e essa complexidade não se coloca.

Exercícios

1. O sono tem de verificar lk! = & Ptable.lock para evitar um impasse (2567-2570). Pode eliminate o caso especial pela substituição

```
if (lk! = & ptable.lock) {
   adquirir (& ptable.lock);
   release (lk);
}
```

com

```
release (lk);
adquirir (& ptable.lock);
```

Fazer isso seria quebrar o sono. Como?

2. A maioria limpeza processo pode ser feito por qualquer saída ou esperar, mas já vimos que saída não deve p livre> pilha. Acontece que a saída deve ser o único a fechar o arquivos abertos. Por quê? A resposta envolve tubos.

3. Implementar semáforos em xv6. Você pode usar mutexes mas não use o sono e acorda. Substitua os usos de sono e wakeup em xv6 com semáforos. Julgue o resultado.

Página 66

Página 67

Capítulo 6

persistência
recuperação de falhas
transação
inode
superbloco

Sistema de arquivo

O objetivo de um sistema de arquivos é organizar e armazenar dados. Os sistemas de arquivos tipicamente

compartilhamento de apoio de dados entre usuários e aplicações, bem como a persistência de modo que dados ainda está disponível depois de um reboot.

O sistema de arquivos xv6 fornece arquivos Unix-like, diretórios e caminhos (ver capíter 0), e armazena seus dados em um disco IDE para persistência (ver Capítulo 3). O sis- arquivo TEM aborda vários desafios:

- O sistema de arquivos precisa de estruturas de dados em disco para representar a árvore de di- nomeado reitorias e arquivos, para gravar as identidades dos blocos que mantêm con- de cada arquivo tenda, e para registrar quais áreas do disco são gratuitos.
- O sistema de arquivos deve apoiar a recuperação de falhas. Isto é, se um acidente (por exemplo, poder falha) ocorre, o sistema de arquivos ainda deve funcionar corretamente após uma reinicialização. O risco é que um acidente pode interromper uma seqüência de atualizações e deixar inconsistente em disco estruturas de dados (por exemplo, um bloco que é usado em um arquivo e marcado livre).
- Diferentes processos podem funcionar no sistema de arquivo ao mesmo tempo, e deve co- coordenar a manter invariantes.
- Acessando um disco é ordens de magnitude mais lento do que o acesso de memória, portanto o arquivo sistema deve manter um cache de memória de blocos populares.

O restante deste capítulo explica como xv6 enfrenta esses desafios.

Visão global

A implementação do sistema de arquivo xv6 é organizado em camadas 6, como mostrado na figura 6-1 . A camada mais baixa lê e escreve os blocos no disco IDE através do tampão cache, que sincroniza o acesso aos blocos de disco, certificando-se de que somente um kernel de processo em um momento pode editar os dados do sistema de arquivos armazenados em qualquer bloco particular. O segundo camada permite que as camadas mais altas para embrulhar atualizações para diversos blocos em uma transação, a en- Certifique-se de que os blocos são actualizados atomicamente (ou seja, todos eles são actualizados ou nenhum). A terceira camada fornece arquivos sem nome, cada um representado usando um inode e uma sequência de blocos que prendem os dados do arquivo. A quarta camada implementa diretórios como um tipo especial de inode cujo conteúdo é uma seqüência de entradas de diretório, cada um dos quais contém um nome e uma referência ao inode do arquivo chamado. A quinta camada fornece hinomes de caminho erarchical como /usr/rtm/xv6/fs.c, utilizando pesquisa recursiva. O lay-finais er abstrai muitos recursos Unix (por exemplo, tubulações, equipamentos, arquivos, etc.) utilizando o sistema de arquivos interface, simplificando a vida dos programadores de aplicativos.

O sistema de arquivos deve ter um plano para onde armazena inodes e blocos de conteúdo em o disco. Para fazê-lo, xv6 divide o disco em várias secções, conforme mostrado na Figura 6-2 . O sistema de arquivos não usa bloco 0 (que detém o setor de inicialização). O bloco 1 é chamado o superbloco; ela contém metadados sobre o sistema de arquivos (o tamanho do sistema de arquivos em blocos,

Page 68

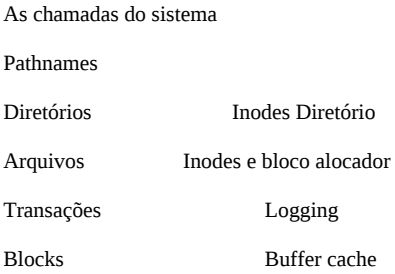


Figura 6-1 . Camadas do sistema de arquivos xv6.

o número de blocos de dados, o número de inodos, e o número de blocos na log). Blocos a partir de 2 inodes espera, com vários inodes por bloco. Após aqueles vêm blocos bitmap rastreamento que blocos de dados em uso. A maioria dos blocos restantes são blocos de dados, que possuem conteúdo de arquivos e diretórios. Os blocos no final do segurar um disco de registo que faz parte da camada de transacção.

código + pão
bwrite + código
amortecedor
brelse + código

O restante deste capítulo discute cada camada, a partir do fundo. Olhe para situações onde abstrações bem escolhidos em camadas inferiores facilitar o projeto de maior queridos.

Buffer de cache Camada

O cache de buffer tem dois empregos: (1) o acesso de sincronização de blocos de disco para garantir que apenas uma cópia de um bloco de memória e é em que apenas um segmento do kernel de cada vez usa essa cópia; (2) colocar em cache blocos populares, de modo que eles não sejam re-lido a partir do lento disco. O código está em bio.c.

A interface principal exportado pelo cache de buffer é composto de pão e bwrite; o ex-obtém um tampão contendo uma cópia de um bloco que pode ser lida ou modificada cadas na memória, e o último escreve um tampão modificado para o bloco apropriado na disco. A lista de discussão do kernel deve liberar um buffer chamando brelse quando é feito com ele.

O cache de buffer sincroniza o acesso a cada bloco, permitindo no máximo um ker- nel linha para ter uma referência para o buffer do bloco. Se uma thread do kernel tenha obtido uma referência a um tampão, mas ainda não divulgou, chamadas de outros segmentos para o pão para o mesmo bloco vai esperar. Camadas do sistema de arquivos maiores dependem de bloco sychro- do buffer cache nização para ajudá-los a manter invariantes.

O cache de buffer tem um número fixo de buffers para manter blocos de disco, o que significa que, se o sistema de arquivos pede um bloco que já não está no cache, o cache de buffer deve reciclar um buffer que actualmente detém algum outro bloco. O cache de buffer recicla o menos utilizado recentemente tampão para o novo bloco. O pressuposto é que a menos recentemente tampão utilizado é o menos provável de ser utilizado novamente em breve.

inicialização super- inodes bit mapa dados .... dados logar

0 1 2 (3030)

Figura 6-2 . Estrutura do sistema de arquivos xv6. O fs.h cabeçalho contém os dados constantes e estruturas que descrevem o layout exato do sistema de arquivos.

Código: cache de buffer

O cache de buffer é uma lista duplamente vinculada de buffers. O BINIT função, chamada por principal (1231), Inicializa a lista com os buffers NBUF no buf matriz estática (4050-4059). Todos os outros acessos ao buffer cache refere-se à lista ligada via bcache.head, não o buf array.

Um buffer tem três bits de estado associados. B_VALID indica que o buffer contém uma cópia válida do bloco. B_DIRTY indica que o conteúdo foi tampão modificada e precisa ser escrito para o disco. B_BUSY indica que alguns núcleo segmento tem uma referência a esse buffer e ainda não liberou.

Pão (4102) chama bget para obter uma reserva para o sector dado (4106). Se o tampão precisa ser lido a partir do disco, as chamadas pão iderw fazer isso antes de retornar o buffer.

Bget (4066) varre a lista de buffer para um buffer com o dispositivo dado e sector núme- bros (4073-4084). Se houver uma tal tampão, e o tampão não está ocupado, ajusta o bget Flag B_BUSY e retornos (4076-4083). Se o buffer já está em uso, bget dorme no tampão de esperar por sua liberação. Quando retorna do sono, bget não pode assumir que o buffer já está disponível. De fato, desde o sono solto e readquirido buf_table_lock, há é nenhuma garantia de que b ainda é o tampão direita: talvez ele tenha sido reutilizado para um diferente setor de disco. Bget não tem escolha a não ser começar de novo (4082). Na esperança de que o resultado será ser diferente desta vez.

Se bget não tinha a instrução goto, então a corrida na Figura 6-3 poderia ocorrer. O primeiro processo tem um tampão e carregou sector 3 nele. Agora, dois outros processos vir. O primeiro faz um get para tampão 3 e dorme no circuito de cache blocos. O segundo faz um get para tampão 4, e pode dormir no mesmo tampão mas no circuito de blocos alocados recém porque não existem tampões livres eo tampão que mantém o 3 é uma na frente da lista e é seleccionado para reutilização. O primeiro processo libera o buffer e despertar acontece para agendar processo de 3 em primeiro lugar, e vai pegar o setor de buffer e carga de 4 nele. Quando isso for feito ele vai liberar o buffer (Contendo sector 4) e despertar processo 2. Sem o processo de instrução goto 2 vai marcar o buffer ocupado, e retornar de bget, mas o buffer contém setor 4, in- vez de 3. Este erro pode resultar em todos os tipos de estragos, porque setores 3 e 4 têm conteúdo diferente; xv6 os usa para armazenar inodes.

Se não houver um tampão para o sector dado, bget deve fazer um, possivelmente, reutilizando uma

código BINIT +
código principal +
Código NBUF +
bcache.head + código
B_VALID + código
Código B_DIRTY +
Código B_BUSY +
bget + código
iderw + código
bget + código
Código B_BUSY +
+ código do sono
buf_table_lock + código
bget + código
bget + código
Código B_BUSY +
B_VALID + código
Código B_DIRTY +

tampão que realizou um sector diferente. Ele varre a lista tampão uma segunda vez, à procura de um bloco que não está ocupado: qualquer bloco pode ser usado. Bget edita os metadados bloco para gravar o novo número sector dispositivo e e marcar o bloco ocupado antes de voltar o bloco (4091-4093) Note-se que a atribuição de bandeiras não só define o bit B_BUSY mas também limpa os bits B_VALID e B_DIRTY, certificando-se que o pão vai atualizar a

Página 70

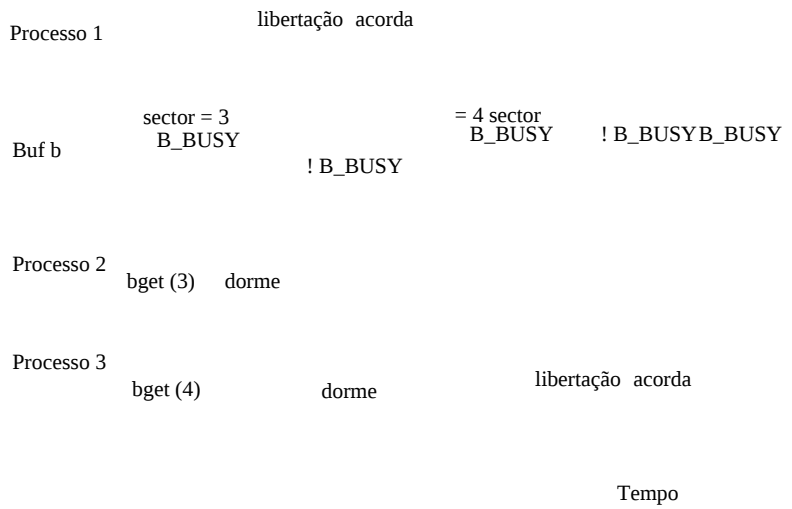


Figura 6-3 . Uma corrida processo resultando em 3 de receber um tampão contendo bloco 4, mesmo que pediu o bloco 3.

dados do buffer de disco em vez de usar o conteúdo do bloco anterior.

Uma vez que o cache de tampão é utilizado para a sincronização, é importante que haja sempre apenas um buffer para um setor de disco particular. As atribuições (4089-4091) são apenas seguro porque primeiro ciclo do bget determinou que nenhum buffer já existia para esse sector, e bget não desistiu buf_table_lock desde então.

Se todos os buffers estão ocupados, algo deu errado: pânicos bget. A mais graciosa resposta poderia ser a de dormir até que um tampão tornou-se livre, embora não faria em seguida, ser uma possibilidade de impasse.

Uma vez que o pão voltou um tampão para o seu chamador, o chamador tem uso exclusivo do tampão e pode ler ou escrever os bytes de dados. Se o chamador não gravar os dados, ele deve chamar bwrite para escrever os dados modificados para o disco antes de liberar o buffer. Bwrite (4114) define o sinalizador B_DIRTY e chama iderw para escrever o buffer para o disco.

Quando o chamador é feito com um buffer, ele deve chamar brelse para liberá-lo. (O nome brelse, um encurtamento de liberação b, é a aprendizagem enigmática, mas vale a pena: ele se originou em Unix e é usado em BSD, Linux, Solaris e também.) Brelse (4125) move-se o tampão da sua posição na lista ligada à parte da frente da lista (4132-4137) limpa o bit B_BUSY, e acorda todos os processos que dormem no buffer. Movendo o tampão tem o efeito de que o buffers são ordenados por quão recentemente eles foram usados (que significa liberado): o primeiro tampão na lista é o usado mais recentemente, eo último é o menos utilizado recentemente. Os dois loops em bget aproveitar essa: a verificação de um tampão existente deve processar o lista inteira na pior das hipóteses, mas verificando os buffers mais recentemente usado pela primeira vez (a partir em bcache.head e seguindo próximos ponteiros) irá reduzir o tempo de verificação quando há boa localidade de referência. A varredura para escolher um tampão para reutilizar picks a menos recentemente bloco usado por varredura para trás (na sequência de ponteiros prev).

código + pão
bwrite + código
Código B_DIRTY +
iderw + código
brelse + código
Código B_BUSY +

Page 71

logar
cometer

Camada Logging

Um dos problemas mais interessantes em design de sistema de arquivos é a recuperação de falhas. O problema surge porque muitas operações do sistema de arquivos envolvem múltiplas gravações para o disco, e um acidente depois de um subconjunto das gravações pode deixar o sistema de arquivos em disco em um incon- estado persistente. Por exemplo, dependendo da ordem do disco escreve, um acidente durante arquivo deleção pode tanto deixar uma entrada apontando diretório para um inode livre, ou pode deixar uma alocada, mas inode unreferenced. O último é relativamente benigna, mas uma entrada de diretório que se refere a um inode libertado é susceptível de causar sérios problemas após uma reinicialização.

Xv6 resolve o problema de acidentes durante as operações do sistema de arquivos com uma simples ver- sion da exploração madeireira. Uma chamada de sistema xv6 não escrever diretamente os dados do sistema de arquivos do disco on- estruturas. Em vez disso, ele coloca uma descrição de todo o disco escreve que pretende fazer em um log no disco. Uma vez que a chamada de sistema tem registrado todas as suas gravações, ele grava um especial cometer recorde para o disco, indicando que o log contém uma operação completa. Às Nesse ponto, a chamada de sistema copia as gravações para as estruturas de dados do sistema de arquivos em disco. Após essas gravações tenham concluído, a chamada de sistema apaga o log no disco.

Se o sistema falhar e reiniciar, o código do sistema de arquivos recupera da acidente da seguinte forma, antes de executar quaisquer processos. Se o log está marcado como um operação completa, em seguida, as cópias do código de recuperação das gravações para onde eles pertencem o sistema on-disco do arquivo. Se o registro não está marcado como contendo uma operação completa, o código de recuperação ignora o log. O código de recuperação terminar, apagando o log.

Por que o log de xv6 resolver o problema de acidentes durante as operações do sistema de arquivos? Se o acidente ocorre antes da operação compromete, então o log no disco não será marcada como concluída, o código de recuperação irá ignorá-lo, e o estado do disco será como se a operação não tinha sequer começado. Se o acidente ocorre após a operação com- MITS, em seguida, a recuperação vai repetir tudo de gravações da operação, talvez repetindo-se a operação tinha começado a escrevê-los à estrutura de dados em disco. Em ambos os casos, o log faz operações atômicas no que diz respeito a falhas: após a recuperação, quer toda a gravações da operação aparecem no disco, ou nenhuma delas aparece.

Log projeto

O log reside num local fixo conhecido no final do disco. É constituída por um bloco de cabeçalho seguido por uma sequência de blocos de dados. O bloco de cabeçalho contém um ar- ray de números do setor, uma para cada um dos blocos de dados registrados. O bloco de cabeçalho também contém a contagem de blocos registrados. Xv6 escreve o bloco cabeçalho quando uma transação compromete-se, mas não antes, e define a contagem a zero depois de copiar os blocos registrados para o sistema de arquivos. Assim, um meio caminho acidente através de uma operação resultará em uma contagem de zero no bloco do cabeçalho do log; um acidente após uma consolidação irá resultar em uma contagem diferente de zero.

Cada código de chamada do sistema indica o início eo fim da sequência de gravações que deve ser atômica; vamos chamar uma tal sequência de uma transação, embora seja muito mais simples de um banco de dados de transação. Apenas uma chamada de sistema podem estar em uma transação em qualquer uma tempo: outros processos devem esperar até que qualquer operação em curso tenha terminado. Assim, o log detém, no máximo, uma transação ao mesmo tempo.

Página 72

escrever + código
desvincular código +
begin_trans + código
log_write + código
bwrite + código
commit_trans + código
install_trans + código
código recover_from_log +
initlog + código

O Xv6 não permitir transações simultâneas, a fim de evitar o seguinte tipo de problema. Transação X Suponha que tem escrito uma modificação para um inode no log. Concurrent transação Y, em seguida, lê um inode diferente no mesmo bloco, atualizações que inode, escreve o bloco inode para o log, e compromete. Isso é um desastre: o inode bloco que Y de cometer grava no disco contém modificações por X, o que não tem cometido. Um acidente e recuperação neste momento iria expor uma das modificações do X mas não todos, quebrando assim a promessa de que as transações são atômicas. Existem sofisticação cado maneiras de resolver este problema; xv6 resolve proibindo transações concorrentes.

Xv6 permite somente leitura chamadas do sistema para executar em simultâneo com uma transação. In- fechaduras ode causar a transação a aparecer atômica para a chamada do sistema somente leitura.

Xv6 dedica uma quantidade fixa de espaço no disco para armazenar o log. Nenhum sistema chamada pode ser permitido escrever blocos mais distintas do que há espaço no log. Isto é não é um problema para a maioria das chamadas do sistema, mas dois deles podem potencialmente escrever muitos blocos: escrever e desvincular. Um grande write arquivo pode escrever muitos blocos de dados e muitos blocos de bitmap, bem como um bloco de inode; desvinculando um arquivo grande pode escrever muitos blocos de bitmap e um inode. Chamada de sistema de gravação de Xv6 rompe grandes gravações em multi- ple gravações menores que se encaixam no log e desvincule não causar problemas, pois em praticar o sistema de arquivos xv6 usa apenas um bloco de bitmap.

Código: logging

Um uso típico do log em uma chamada de sistema parecido com este:

```
begin_trans ();
...
bp = pão (...);
BP> data [...] = ...;
log_write (pb);
...
commit_trans ();
```

begin_trans (4277) espera até que ele obtém uso exclusivo do log e depois retorna.

log_write (4325) atua como um proxy para bwrite; ele acrescenta um novo conteúdo do bloco de o log no disco e registra número do setor do bloco na memória. folhas log_write o bloco modificado no cache de buffer de memória, de modo que as leituras subsequentes do bloquear durante a transação irá produzir o bloco modificado. avisos log_write quando um bloco é escrito várias vezes durante uma única transação, e substitui o de bloco cópia anterior no log.

commit_trans (4301) primeiro escreve bloco do cabeçalho do log para o disco, de modo que um acidente após Neste ponto fará com que a recuperação de re-escrever os blocos no log. commit_trans seguida chama install_trans (4221) para ler cada quarteirão do log e escrevê-lo para o bom colocar no sistema de arquivos. Finalmente commit_trans escreve o cabeçalho de log com uma contagem de zero, de modo que uma falha após a próxima transacção começa resultará no código de recuperação IG- Noring o log.

recover_from_log (4268) é chamado de initlog (4205) Que é chamada durante inicialização antes do primeiro processo de usuário é executado. Ele lê o cabeçalho de log, e imita os (2944) ações de commit_trans se o cabeçalho indica que o log contém um comprometido

transação.

Um exemplo de uso do log ocorre em FILEWRITE (5352) A transação parece este:

```
begin_trans ();
ilock (f-> ip);
r = writei (f-> ip, ...);
iUnlock (f-> ip);
commit_trans ();
```

FILEWRITE + código
código writei +
ilock + código
begin_trans + código
balloc + código
código BFree +
readsb + código
inode

Este código é envolto em um loop que rompe com grandes gravações em transações individuais de apenas alguns setores de cada vez, para evitar transbordamento do log. A chamada para writei escreve muitos blocos como parte dessa transação: inode do arquivo, um ou mais blocos de bitmap, e alguns blocos de dados. A chamada para ilock ocorre após os begin_trans como parte de um estratégia global para evitar impasse: uma vez que existe efetivamente um bloqueio ao redor de cada trans- ação, a regra de ordenação lock-evitando impasse é transação antes de inode.

Código: Bloco alocador

Conteúdo de arquivos e diretórios são armazenados em blocos de disco, que deve ser alocados a partir de um piscina livre. bloco alocador de xv6 mantém um bitmap livre em disco, com um bit por bloco. Um pouco de zero indica que o bloco correspondente é livre; um bit que indica ele está em uso. Os bits correspondentes ao setor de boot, superbloco, blocos de inode, e blocos de bitmap são sempre definido.

O alocador de blocos fornece duas funções: balloc aloca um novo bloco de disco, e BFree libera um bloco. Balloc (4454) começa por chamar readsb para ler o superbloco a partir do disco (ou buffer cache) em sb. balloc decide quais blocos de armazenar os dados bloquear bitmap livre calculando quantos blocos são consumidos pelo setor de boot, o superbloco, e os inodes (usando BBLOCK). O loop (4462) considera cada bloco, começando no bloco 0 até sb.size, o número de blocos do sistema de ficheiros. Ele procura por um bloco cujo bitmap bit é zero, o que indica que ele é livre. Se balloc encontrar tal bloco, ele atualiza o bitmap e retorna o bloco. Para uma maior eficiência, o circuito é dividido em dois pedaços. O loop externo lê cada bloco de bits de bitmap. As verificações de loop interno tudo Bocados BPB em um único bloco de bitmap. A corrida que pode ocorrer se dois processos tentar alocar um bloco, ao mesmo tempo é evitada pelo facto de que a cache da memória intermédia apenas

permite que um processo de usar um bloco de cada vez.

BFree (4481) encontra o bloco de bitmap direita e limpa o bit direita. Novamente a exclusão uso sive implícito pão e brelse evita a necessidade de bloqueios explícitos.

Inodes

O termo inode pode ter um dos dois significados relacionados. Pode se referir ao on-estrutura de dados disco que contém o tamanho de um arquivo e uma lista de números de blocos de dados. Ou " inode " pode se referir a um inode in-memory, que contém uma cópia do inode no disco como bem como informação adicional necessária dentro do kernel.

Todos os inodes em disco são embalados em uma área contígua de disco chamado de in-blocos do ODE. Cada inodo é o mesmo tamanho, de modo que é fácil, dado um número n, para encontrar o

inodo enésimo no disco. Na verdade, este número n, ligou para o número inode ou i-número, é como inodes são identificados na implementação.

struct dinode + código
struct inode + código
código iget +
iput + código
ilock + código
ialloc + código
balloc + código
código iget +

O inode no disco é definido por um dinode struct (3676) O distinguiu tipo de campo guishes entre arquivos, diretórios e arquivos especiais (dispositivos). Um tipo de zero indica que um inode no disco é gratuito. O campo nlink conta o número de entradas de diretório que se referem a este inode, a fim de reconhecer quando o inode deve ser liberado. O campo de tamanho registra o número de bytes do conteúdo do arquivo. Os registros de matriz addrs os números dos blocos dos blocos de disco que prendem o conteúdo do arquivo.

O kernel mantém o conjunto de inodes ativos na memória; struct inode (3762) é o cópia na memória de um dinode struct no disco. O kernel armazena um inode na memória somente se houver ponteiros C referentes àquele inode. O campo ref conta o número de Os ponteiros em C referindo-se ao inode in-memory, eo kernel descarta o inode de memória se a contagem chegar a zero. As funções iGet e iput adquirir e solte ponteiros para um inode, modificando a contagem de referência. Ponteiros para um inode pode vir de descritores de arquivos, diretórios de trabalho atual e código do kernel transitória como exec.

Segurando um ponteiro para um inode retornado por iget () garante que o inode vontade ficar em cache e não serão excluídos (e, em particular, não serão reutilizados para um arquivo diferente). Assim, um ponteiro retornado por iget () é uma forma fraca de bloqueio, embora não dá direito ao titular de realmente olhar para o inode. Muitas partes do sistema de arquivos código dependem este comportamento de iget (), tanto para conter referências a longo prazo para inodes (Como abrir arquivos e diretórios atuais) e para evitar corridas evitando impasse no o código que manipula vários inodes (tais como pesquisa caminho).

O inode struct que iGet retornos podem não ter qualquer conteúdo útil. Em ordem para garantir que ele detém uma cópia do inode no disco, o código deve chamar ilock. Isto bloqueia o inode (para que nenhum outro processo pode ILOCK-lo) e lê o inode do disco, se ele já não foi lido. iUnlock libera o bloqueio no inode. Separando aqui-sição de ponteiros inode de bloqueio ajuda a evitar impasse em algumas situações, por exem-amplo durante a pesquisa de diretório. Vários processos podem segurar um ponteiro C a um inode retornado por iget, mas apenas um processo pode bloquear a inodo de cada vez.

O cache inode só armazena inodes para que o código do kernel ou estruturas de dados detêm Ponteiros C. Sua principal tarefa é realmente sincronizar o acesso por vários processos, não caching. Se um inode é usado com freqüência, o buffer cache provavelmente irá mantê-lo em mem-ory se não for mantido pelo cache inode.

Código: Inodes

Para alocar um novo inode (por exemplo, ao criar um arquivo), xv6 chama ialloc (4603) Ialloc é semelhante ao balloc: ele varre as estruturas de inodo no disco, uma bloquear ao mesmo tempo, procurando um que é marcado livre. Quando se encontra um, ele afirma que por escrevendo o novo tipo de disco e, em seguida, retorna uma entrada do cache inode com a chamada cauda para iget (4620) A correcta operação do ialloc depende do facto apenas um processo por vez pode ser mantendo uma referência a bp: ialloc pode ter certeza que algum outro processo não vê simultaneamente que o inode está disponível e tentar reivindicá-lo.

Eu Recebos olha através do cache inode para uma entrada ativa (IP-> ref> 0) com o número do dispositivo e inode desejado. Se encontrar um, ele retorna uma nova referência para que inode. Como scans iGet, ele registra a posição do primeiro slot vazio Que ele usa, se ele precisa alocar uma entrada de cache.

A chamada deve bloquear o inode usando ilock antes de ler ou escrever seus metadados ou conteúdo. ILOCK utiliza um ciclo de sono agora familiar de esperar por IP-> da bandeira I_BUSY bit ser claro e, em seguida, define- Uma vez ilock tem acesso exclusivo ao inode, ele pode carregar os metadados inode do disco (mais provável, o cache de buffer), se necessário. O iUnlock função limpa o bit I_BUSY e acorda todos os processos que dormem na ilock.

Eu Colocar libera um ponteiro C a um inode por diminuindo a contagem de referência Se esta é a última referência, slot do inode no cache inode agora está livre e pode ser re-utilizado por um inodo diferente.

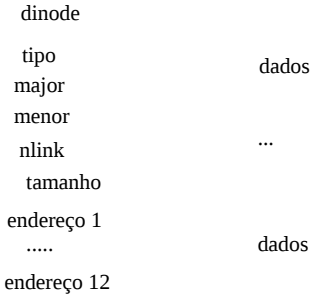
Se iput vê que não há referências ponteiro C para um inode e que o inode não tem links para ele (ocorre em nenhum diretório), então o inode e seus blocos de dados deve ser libertada. Iput bloqueia novamente o inode; chama itrunc para truncar o arquivo para zero bytes, liberando os blocos de dados; define o tipo inode a 0 (não alocado); escreve a mudança no disco; e finalmente desbloqueia o inode.

O protocolo de bloqueio em iput merece um olhar mais atento. A primeira parte pena exem- ining é que quando o bloqueio ip, iput simplesmente assumido que iria ser desbloqueado, em vez utilizando uma ansa de sono. Este deve ser o caso, porque o chamador é necessária para desbloquear ip antes de chamar iput, e o chamador tem a única referência a ele (IP-> ref == 1). O segunda parte, convém examinar que iput libera temporariamente e readquire o bloqueio de cache. Isso é necessário porque itrunc e iupdate vai dormir durante i disco / o, mas devemos considerar o que pode acontecer quando o bloqueio não é realizada. Especificação camente, uma vez iupdate termina, a estrutura em disco é marcado como disponível para uso, e um chamada simultânea para ialloc pode encontrá-lo e realocá-lo antes iput pode terminar. Ial- loc retornará uma referência ao bloco de ligando iget, que vai encontrar no ip cache, ver que a sua bandeira I_BUSY está definido, e dormir. Agora, o inode no interior do núcleo está fora de sincronia em comparação com o disco: ialloc reinicializado a versão em disco, mas depende do chamador carregá-lo na memória durante ilock. A fim de se certificar de que isso acontece, iput deve limpar não só I_BUSY mas também I_VALID antes de liberar o bloqueio inode. Ele faz este por bandeiras zeragem.

código iget + ilock + código
Código I_BUSY + ilock + código
iUnlock + código
Código I_BUSY + iput + código
itrunc + código
iput + código
iput + código
itrunc + código
iupdate + código
ialloc + código
código iget +
Código I_BUSY + I_VALID + código
struct dinode + código
Código NDIRECT + blocos diretos
Código NINDIRECT + bloco indireto
Código bsize +

Código: conteúdo Inode

A estrutura em disco inodo, dinode estrutura, contém um tamanho e uma disposição de números de bloco (ver F igura 6-4). Os dados de inodo é encontrado nos blocos enumerados no matriz addrs de dinode. Os primeiros blocos de dados NDIRECT estão listados no primeiro NDIRECT entradas na matriz; Esses blocos são chamados blocos diretos. A próxima NINDIRECT os blocos de dados não estão listados no inodo mas num bloco de dados chamado o indirecta bloco. A última entrada na matriz addrs dá o endereço do bloco indireto. Assim os primeiros 6 kB (NDIRECT × bsize) bytes de um arquivo pode ser carregado a partir de blocos listados na inode, enquanto os próximos 64 kB (NINDIRECT × bsize) bytes só pode ser carregado após ter consultado ing o bloco indireto. Esta é uma boa representação no disco, mas um complexo para



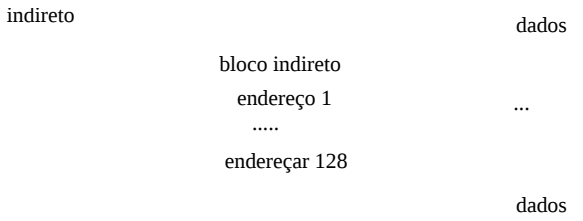


Figura 6-4 . A representação de um arquivo no disco.

clientes. O bmap função gerencia a representação para que as rotinas de nível superior como readi e writei, que veremos em breve. Bmap devolve o bloco de disco número do bloco de dados bn'th para o ip inode. Se ip não tem um tal bloco, no entanto, bmap aloca um.

código bmap +

O bmap função (4810) começa por escolher fora o caso fácil: o primeiro NDIRECT blocos estão listados na própria inodo (4815-4819) Os seguintes blocos NINDIRECT estão listados na o bloco indireto no IP-> addrs [NDIRECT]. Bmap lê o bloco indireto (4826) e em seguida, lê um número de bloco a partir da posição correta dentro do bloco (4827) Se o bloco número excede NDIRECT + NINDIRECT, pânicos BMAP: chamadores são responsáveis por não pedir- ing sobre números de blocos fora de alcance.

readi + código
código writei +
código bmap +
Código NDIRECT +
Código NINDIRECT +
itrunc + código
readi + código
código writei +

Bmap aloca bloco, conforme necessário. Blocos não alocados são indicados por um núme- bloco ber de zero. Como bmap encontra zeros, ele substitui-los com os números de fresco blocos, alocada sob demanda. (4816-4817, 4824-4825)

Bmap aloca blocos sob demanda como o inode cresce; itrunc liberta-los, a redefinição tamanho do inodo a zero. Itrunc (4856) começa por liberando os blocos diretos (4862-4867) em seguida, os listados no bloco indireto (4872-4875) E, finalmente, o próprio bloco indirecta (4877-4878)

Bmap facilita para escrever funções para acessar fluxo de dados do inode, como readi e writei. Readi (4902) lê os dados do inode. Ele começa a fazer-se de que o off- definir e contar não estão lendo além do final do arquivo. Lê que start para além do final do arquivo retornará um erro (4913-4914) enquanto lê-se que no início ou atravessar o final de o arquivo de retornar menos bytes do que o solicitado (4915-4916) Os principais processos de loop cada

Página 77

bloco do arquivo, a cópia de dados do buffer em dst (4918-4923) A função writei (4952) é idêntico ao Readi, com três exceções: escreve que começam em ou cross o fim do ficheiro de aumentar o arquivo, até o tamanho máximo (4965-4966) as cópias de loop dados para os buffers em vez de fora (4971) e se a gravação foi prorrogado o arquivo, writei deve atualizar o seu tamanho (4976-4979)

código writei +
readi + código
código writei +
readi + código
código writei +
Código T_DEV +
código stati +
+ código de estatísticas

Ambos readi e writei começar verificando IP-> tipo == T_DEV. Este caso lida com dispositivos especiais cujos dados não vive no sistema de arquivos; vamos voltar a Neste caso, na camada de descritor de arquivo.

O stati função (4423) cópias inodo metadados na estrutura estatística, que é expostos a programas do usuário através da chamada de sistema stat.

Código T_DIR +
struct dirent + código
Código DIRSIZ +
dirlookup + código
código iget +
. + Código
.. + Código
dirlink + código
dirlookup + código
nameiparent + código
namex + código

Código: camada de diretório

Um diretório é implementado internamente muito parecido com um arquivo. Sua inode tem tipo T_DIR e seus dados é uma seqüência de entradas de diretório. Cada entrada é um dirent struct (3700) que contém um nome e um número de inode. O nome é no máximo DIRSIZ (14) char- acters; se for menor, é denunciado por uma NUL (0) byte. As entradas do diretório com inode número zero são gratuitos.

O dirlookup função (5011) procura em um diretório para uma entrada com o dado nome. Se encontrar um, ele retorna um ponteiro para o inode correspondente, desbloqueado, e define * poff para o deslocamento da entrada dentro do diretório byte, caso os desejos de chamadas editá-lo. Se dirlookup encontra uma entrada com o nome certo, ele atualiza * poff, lançamentos o bloco, e retorna um inode desbloqueado obtido via iget. Dirlookup é a razão que iget retorna inodes desbloqueadas. O chamador bloqueou dp, por isso, se a pesquisa era para ., Um alias para o diretório atual, na tentativa de bloquear o inode antes de voltar iria tentar re-lock dp e impasse. (Existem bloqueio mais complicado scenar- ios envolvendo múltiplos processos e .., um alias para o diretório pai; . não é a

único problema.) A pessoa pode desbloquear dp e trave ip, garantindo que ele só mantém um bloqueio de cada vez.

O dirlink função (5052) escreve uma nova entrada de diretório com o nome dado e número de inode para o dp diretório. Se o nome já existe, retornos dirlink um erro (5058-5062) O loop principal lê entradas de diretório procurando uma não alocado entrada. Quando se encontra um, ele pára a malha mais cedo (5022-5023) Com off definido para a compensação de a entrada disponível. Caso contrário, o ciclo termina com off definido para DP> tamanho. De qualquer jeito, dirlink em seguida, adiciona uma nova entrada para o diretório, escrevendo no deslocamento off (3072-3078)

Os nomes de caminho: Código

Nome do caminho de pesquisa envolve uma sucessão de chamadas para dirlookup, um para cada caminho componente. Namei (5189) avalia caminho e retorna o inode correspondente. O função nameiparent é uma variante: pára antes do último elemento, retornando o inode do diretório pai e copiar o elemento final no nome. Ambos chamar o geral-zada namex função para fazer o trabalho real.

Namex (5154) começa por decidir onde a avaliação caminho começa. Se o caminho en-gins com uma barra, a avaliação começa na raiz; caso contrário, o diretório atual (5158-

Página 78

5161) Em seguida, ele usa skipelem considerar cada elemento do caminho, por sua vez (5163) Cada iteração do loop deve pesquisar o nome no atual ip inode. A iteração começa travando ip e verificar que se trata de um diretório. Se não, a pesquisa falhar (5164-5168) (Ip bloqueio é necessário não porque IP-> tipo pode mudar sob os pés-não-pode-lo, mas porque até ilock corre, IP-> tipo não é garantido que foram carregados do disco.) Se a chamada for nameiparent e este é o último elemento do caminho, o ciclo pára cedo, como por a definição de nameiparent; o elemento caminho final já foi copiado para nome, então namex só precisa devolver o ip desbloqueado (5169-5173) Finalmente, o loop procura o elemento caminho usando dirlookup e se prepara para a próxima iteração, definindo ip = next (5174-5179) Quando o loop é executado a partir de elementos de caminho, ele retorna ip.

skipelem + código
ilock + código
nameiparent + código
namex + código
dirlookup + código
arquivo struct + código
+ código aberto
código + dup
código garfo +
código + ftable
fileallocar + código
filedup + código
FileClose + código
FILEREAD + código
FILEWRITE + código
filedup + código
FileClose + código
código filestat +
FILEREAD + código
FILEWRITE + código
+ código de estatísticas
leia + código
escrever + código
código stati +

Camada de descritor de arquivo

Um do aspecto legal da interface Unix é que a maioria dos recursos em Unix são representado como um arquivo, incluindo dispositivos como o console, tubos, e, claro, o Real arquivos. A camada de descritor de arquivo é a camada que atinge essa uniformidade.

Xv6 dá a cada processo de sua própria tabela de arquivos abertos, ou descritores de arquivos, como vimos no Capítulo 0. Cada arquivo aberto é representado por um arquivo struct (3750) Que é um invólucro em torno de qualquer um inode ou um tubo, além de uma i / o offset. Cada chamada para abrir cria um novo arquivo aberto (um novo arquivo struct): se vários processos de abrir o mesmo arquivo de forma independente, os diferentes casos terá deslocamentos diferentes de E / S. Por outro lado, um único aberto arquivo (o mesmo arquivo struct) pode aparecer várias vezes na tabela de arquivo de um processo e também nas mesas de múltiplos processos de arquivo. Isso aconteceria se um processo utilizado aberto para abrir o arquivo e, em seguida, criou aliases usando dup ou compartilhado com a criança que usa garfo. A contagem de referência acompanha o número de referências a um arquivo aberto particular. A arquivo pode ser aberto para leitura ou escrita ou ambos. Os campos legível e gravável controlar isso.

Todos os arquivos abertos no sistema são mantidos em uma tabela de arquivo global, o ftable. O arquivo tabela tem uma função de alocar um arquivo (filealloc), criar uma referência duplicado (Filedup), solte uma referência (FileClose), e ler e gravar dados (FILEREAD e FILEWRITE).

Os três primeiros seguem a forma agora familiar. Filealloc (5225) verifica a tabela de arquivos para um arquivo não referenciado (f-> ref == 0) e retorna uma nova referência; filedup (5252) in-crements a contagem de referência; e FileClose (5264) o diminui. Quando rência de um arquivo contagem cia atinge zero, FileClose libera o tubo subjacente ou inode, de acordo com o tipo.

O filestat funções, FILEREAD e FILEWRITE implementar o status, ler, e escrever operações em arquivos. Filestat (5302) só é permitido em inodes e chamadas stati. FILEREAD e verificação FILEWRITE que a operação é permitido pela aberto modo e, em seguida, passar a chamada através de um ou outro tubo ou implementação inode. Se o arquivo representa um inode, FILEREAD e FILEWRITE usar o i / o deslocamento como o deslocamento para a operação e depois avance- (5325-5326, 5365-5366) Pipes não têm noção do off-definido. Lembre-se que as funções de inode exigir o chamador para lidar com bloqueio (5305-5307, 5324-

5327, 5364-5378) O bloqueio inode tem o efeito colateral conveniente que o ler e escrever
offsets são atualizados atomicamente, de modo que a escrita múltipla para o mesmo arquivo simultaneamente

Página 79

não pode substituir os dados de cada um, embora seus gravações pode acabar entrelaçado.

sys_link + código
sys_unlink + código
nameiparent + código
sys_link + código
criar + código
+ código aberto
O_CREATE + código
mkdir + código
mkdev + código
sys_link + código
criar + código
nameiparent + código
dirlookup + código
mkdir + código
mkdev + código
Código T_FILE +
ialloc + código
. + Código
.. + Código
criar + código
sys_link + código
código + sys_open
sys_mkdir + código
sys_mknod + código
+ código aberto
O_CREATE + código
código namei +
código + sys_open

As chamadas do sistema: Código

Com as funções que as camadas inferiores proporcionam a aplicação de mais siste-
chamadas TEM é trivial (ver sysfile.c). Existem algumas chamadas que merecem um olhar mais atento.

As funções sys_link e diretórios editar sys_unlink, criando ou removendo
referências a inodes. Eles são outro bom exemplo do poder da utilização de transação
ções. Sys_link (5513) começa por ir buscar os seus argumentos, duas cordas velho eo novo (5518)
Assumindo velho existe e não é um diretório (5520-5530) Sys_link incrementa seu IP
> Contagem nlink. Então sys_link chama nameiparent para encontrar o diretório pai e
elemento caminho final de novo (5536) e cria uma nova entrada de diretório apontando para in- do velho
ode (5539) O novo diretório pai deve existir e estar no mesmo dispositivo como o ex-
no existentes inode: números de inode só tem um sentido único em um único disco. Se um erro
como isso ocorre, sys_link deve voltar e decrement IP-> nlink.

Transações simplificar a execução, porque ela exige a atualização múltipla
blocos de disco, mas não temos que se preocupar com a ordem em que as fazemos. Eles EI
ther vai tudo dar certo ou nenhum. Por exemplo, sem operações, atualizando IP-> nlink
antes de criar um link, iria colocar o sistema de arquivos temporariamente em um estado inseguro, e um
acidente entre poderia resultar em estragos. Com as transações que não tem que se preocupar
sobre isso.

Sys_link cria um novo nome para um inode existente. A função de criar (5657)
cria um novo nome para um novo inode. É uma generalização da criação de três arquivo
sistema chama: aberto com a bandeira O_CREATE faz um novo arquivo comum, mkdir faz uma
nova directoryy e mkdev faz um novo arquivo de dispositivo. Como sys_link, criar começa por
caling nameiparent para obter o inode do diretório pai. Em seguida, chama dirlookup
para verificar se o nome já existe (5667) Se o nome existir, crie de ser-
havior depende do sistema de chamar ele está sendo usado para: aberto tem semânticas diferentes
de mkdir e mkdev. Se criar está sendo usada em nome de aberto (tipo == T_FILE)
eo nome que existe é um arquivo regular, em seguida, trata abertos que como um sucesso, por isso,
criar faz também (5671) Caso contrário, é um erro (5672-5673) Se o nome não al-
pronto existir, crie agora aloca um novo inode com ialloc (5676) Se o novo inode é
um diretório, criar inicializa com. e .. entradas. Finalmente, agora que os dados estejam
inicializado corretamente, criar pode vinculá-lo para o diretório pai (5689) Criar, como
sys_link, detém dois bloqueios inode simultaneamente: ip e dp. Não há possibilidade de
impasse porque o ip inode é recém-alocado: nenhum outro processo no sistema será
manter o bloqueio de IP e, em seguida, tentar bloquear dp.

Usando criar, é fácil de implementar sys_open, sys_mkdir e sys_mknod.
Sys_open (5701) é o mais complexo, porque a criação de um novo arquivo é apenas uma pequena parte da
o que ele pode fazer. Se aberto é passado a bandeira O_CREATE, ele chama criar (5712) Caso contrário,
ele chama namei (5717) Criar retornos um inode trancada, mas namei não, então sys_open
deve bloquear o próprio inode. Isso fornece um local conveniente para verificar se os diretórios
só são abertas para a leitura, não escrever. Assumindo que o inode foi obtido de uma forma
ou de outra, sys_open aloca um arquivo e um descritor de arquivo (5726) e, em seguida, preenche o
arquivo (5734-5738) Note-se que nenhum outro processo pode acessar o arquivo parcialmente inicializado, uma vez que

Página 80

é só na mesa do processo atual.

sys_pipe + código
código + fsck

Capítulo 5 examinou a aplicação de tubos antes mesmo de nós tinha um arquivo sis-
TEM. O sys_pipe função conecta que a implementação do sistema de arquivos por pro-
necer uma maneira de criar um par de tubos. Seu argumento é um ponteiro para o espaço para dois inteiros,

onde ele irá gravar os dois novos descritores de arquivos. Em seguida, ele atribui o tubo e in-
bana os descritores de arquivos.

Mundo real

O cache de buffer em um sistema operacional do mundo real é muito mais complexo
que xv6 do, mas serve os mesmos dois propósitos: caching e sincronizar o acesso a
o disco. Buffer cache do Xv6, como V6 de, usa um usado menos recentemente (LRU) despejo simples
política; há muitos mais políticas complexas que podem ser implementadas, cada bom para
algumas cargas de trabalho e não tão bom para outros. Um cache LRU mais eficiente seria eli-
nate a lista ligada, em vez de usar uma tabela hash para pesquisas e um montão de LRU evic-
ções. Caches de buffer modernos são normalmente integrado com o sistema de memória virtual
para suportar arquivos de memória mapeada.

Sistema de registro de Xv6 é terrivelmente ineficiente. Ele não permite que a atualização simultânea
chamadas do sistema, mesmo quando as chamadas do sistema operar completamente diferentes partes do arquivo
sistema. Ele registra blocos inteiros, mesmo que apenas alguns bytes em um bloco são alteradas. Ele per-
formas log síncrona escreve: um bloco de cada vez, cada um dos quais é provável que exigem uma
tempo inteiro rotação do disco. Sistemas de registro de reais resolver todos esses problemas.

Logging não é a única forma de proporcionar a recuperação de falhas. Sistemas de arquivos adiantados usaram um
limpador durante reboot (por exemplo, o UNIX programa fsck) para examinar cada arquivo
e diretório e do bloco e inode listas livres, procurando e resolver inconsistências
cies. Scavenging pode levar horas para sistemas de arquivos grandes, e há situações em que
não é possível adivinhar a resolução correta de uma inconsistência. A recuperação de um log
é muito mais rápido e é correcta.

Xv6 usa o mesmo layout básico em disco de inodes e diretórios como o início de UNIX;
este regime tem sido notavelmente persistente ao longo dos anos. Do BSD UFS / FFS e Linux de
ext2 / ext3 uso essencialmente as mesmas estruturas de dados. A parte mais ineficiente do arquivo
layout do sistema é o diretório, o que exige uma varredura linear sobre todos os blocos do disco du-
ing cada pesquisa. Isso é razoável quando diretórios são apenas alguns blocos de disco, mas é
caro para os diretórios que prendem muitos arquivos. NTFS, Mac OS X de Windows da Microsoft
HFS e ZFS do Solaris, apenas para citar alguns, implementar um diretório como um brado em disco
árvore ANCED de blocos. Este complicado, mas garante diretório em tempo logarítmica
pesquisas.

Xv6 é ingênuo sobre falhas de disco: se uma operação de disco falhar, pânicos xv6. Se este
é razoável depende do hardware: se um sistema operacional fica no topo de hardware especial
ware que utiliza redundância para mascarar as falhas do disco, talvez o sistema operacional vê
falhas tão pouco frequente que em pânico está bem. Por outro lado, os sistemas operativos
usando discos lisos devem esperar falhas e tratá-los de forma mais elegante, para que o
perda de um bloco de um arquivo não afecta a utilização do resto do sistema de ficheiros.

Xv6 exige que o sistema de arquivos caber em um dispositivo de disco e não mudam de tamanho. Como
grandes bancos de dados e arquivos multimídia conduzir os requisitos de armazenamento cada vez mais elevados, operacional

Page 81

sistemas estão desenvolvendo maneiras de eliminar o " um disco por sistema de arquivos " gargalo. O
abordagem básica é combinar diversos discos em um único disco lógico. Soluções de hardware
tais como RAID ainda são os mais populares, mas a tendência atual está se movendo em direção a im-
plementar o máximo dessa lógica em software quanto possível. Estes implementação software
ções normalmente permitindo a funcionalidade rica como crescendo ou diminuindo o dispositivo lógico
adicionando ou removendo discos na mosca. É claro, uma camada de armazenagem que pode crescer ou
encolher na mosca requer um sistema de arquivos que podem fazer o mesmo: a matriz de tamanho fixo de in-
blocos ode utilizados pelos sistemas de arquivos Unix não funciona bem em tais ambientes. Separação
classificação de gerenciamento de disco do sistema de arquivos pode ser o design mais limpo, mas a com-
interface de plex entre os dois levou alguns sistemas, como ZFS da Sun, para combiná-las.

FILEREAD + código
FILEWRITE + código

Sistema de arquivos do Xv6 carece de muitas outras características em sistemas de arquivos de hoje; por exemplo, ele
não tem suporte para instantâneos e backup incremental.

Xv6 tem duas implementações diferentes de arquivo: Canos e inodes. Modern sis- Unix
tems têm muitas: tubos, conexões de rede e inodes de muitos tipos diferentes de
sistemas de arquivos, incluindo sistemas de arquivos de rede. Em vez de se as declarações em FILEREAD
e FILEWRITE, esses sistemas normalmente dão cada arquivo aberto uma tabela de ponteiros de função,
um por operação, e chamar o ponteiro de função para chamar a implementação desse inode
da chamada. Sistemas de arquivos de rede e sistemas de arquivos em nível de usuário fornecer funções que transformam
essas chamadas em RPCs de rede e esperar a resposta antes de retornar.

Exercícios

1. Por que motivo de pânico em balloc? Podemos recuperar?
2. Por pânico em ialloc? Podemos recuperar?
3. Números de geração de inode.
4. Por que não filealloc pânico quando se esgota de arquivos? Por que isso é mais comum e, portanto, vale a manipulação?
5. Suponha que o arquivo correspondente a ip fica desvinculada por outro processo entre chamadas de sys_link para iUnlock (ip) e dirlink. Será que o link seja criada corretamente? Por que ou por que não?
6. criar faz quatro chamadas de função (uma para ialloc e três para dirlink) que ele necessita para ter sucesso. Se algum não, criar chamadas pânico. Por que isso é aceitável? Por que qualquer um desses quatro chamadas não pode falhar?
7. sys_chdir chama iUnlock (ip) antes iput (CP-> CWD), o que poderá tentar bloquear CP-> CWD, ainda adiar iUnlock (ip) até depois da iput não causaria conflitos. Por que não?

Página 83

Um apêndice

contador de programa

Hardware PC

Este apêndice descreve computador pessoal (PC) de hardware, a plataforma sobre a qual xv6 executado.

A PC é um computador que adere a vários padrões da indústria, com o objetivo de que uma determinada peça de software pode ser executado em PCs vendidos por vários fornecedores. Estas normas evoluem ao longo do tempo e de um PC a partir da década de 1990 não se parece com um PC agora.

Do lado de fora de um PC é uma caixa com um teclado, um ecrã, e vários dispositivos (Por exemplo, CD-ROM, etc.). Dentro da caixa é uma placa de circuito (a " mãe ") com CPU batatas fritas, chips de memória, chips gráficos, I / O, chips controladores e ônibus por meio do qual o chips de comunicar. Os ônibus aderir a protocolos padrão (por exemplo, PCI e USB) a fim que os dispositivos vão trabalhar com PCs de vários fornecedores.

Do nosso ponto de vista, podemos abstrair o PC em três componentes: CPU, memória e de entrada / saída (I / O) dispositivos. A CPU executa o cálculo, o memory contém instruções e dados para que a computação, e dispositivos de permitir que a CPU interagir com o hardware para o armazenamento, a comunicação, e outras funções.

Você pode pensar em memória principal, ligado à CPU com um conjunto de fios, ou linhas, alguns para bits de endereço, alguns deles para bits de dados, e alguns para sinalizadores de controle. Para ler uma valor da memória principal, a CPU envia tensões altas ou baixas, representando 1 ou 0 bits sobre as linhas de endereço e um 1 na "linha" "ler" por uma quantidade de tempo prescrito e em seguida, lê de volta o valor, interpretando as tensões nas linhas de dados. Para escrever um valor para a memória principal, a CPU envia bits apropriados no endereço e linhas de dados e um 1 na "linha" "escrever" para uma quantidade de tempo prescrito. Interfaces de memória real são mais complexas do que isso, mas os detalhes só são importantes se você precisa para alcançar alto desempenho.

Processador e memória

CPU de um computador (unidade central de processamento, ou processador) executa um conceitualmente simlaço ple: ele consulta um endereço em um registo chamado o contador de programa, lê uma mainstrução chine desse endereço na memória, avança o contador de programa após o instrução, e executa as instruções. Repita. Se a execução da instrução

não modifica o contador de programa, este ciclo vai interpretar a memória apontada para
por o contador do programa, como uma sequência de instruções de máquina para executar um após o
outro. Instruções que fazem mudar o contador de programa incluem ramos e função
chama.

O mecanismo de execução é inútil sem a capacidade de armazenar e modificar programa
dados. O armazenamento mais rápido para os dados são fornecidos por registo conjunto do processador. Um registo
é uma célula de armazenamento dentro do próprio processador, capaz de manter uma máquina palavra-sized
valor (normalmente 16, 32 ou 64 bits). Os dados armazenados em registros normalmente pode ser lido ou

Página 84

escrito rapidamente, num único ciclo de CPU.

ponteiro de instrução
registradores de controle
registradores de segmento

PCs tem um processador que implementa o conjunto de instruções x86, que foi original-
ly definido pela Intel e tornou-se um padrão. Vários fabricantes produzir transformação
sors que implementam o conjunto de instruções. Como todos os outros padrões de PC, este padrão é
também evoluindo, mas normas mais recentes são compatíveis com os padrões do passado. O
carregador de boot tem de lidar com um pouco dessa evolução, porque cada processador de PC começa
simulando um processador Intel 8088, o chip CPU no PC original da IBM lançado em 1981.
No entanto, para a maioria de xv6 você vai estar preocupado com o moderno conjunto de instruções x86.

O x86 moderno oferece oito uso geral de 32 bits registros.-% eax,% ebx,
Ecx%,% edx,% edi, esi%,% ebp, e esp% e um contador de programa% eip (o "in-
ponteiro struction). O prefixo comum e representa prolongado, uma vez que estes são de 32 bits
extensões de 16 bits registros% de machado,% BX, CX%,% dx, di%,% de Si,% pb, sp% e% IP. O
dois conjuntos de registros são alias, de modo que, por exemplo,% machado é a metade inferior de% eax: escrito
ING% machado altera o valor armazenado em% eax e vice-versa. Os quatro primeiros registros
também têm nomes para o fundo dois bytes de 8 bits:% al% e ah denotar a baixa e alta
8 bits de machado%; Bl%,% bh, cl%,% ch, dl% e% dh continuar o padrão. Além de
esses registros, o x86 tem oito de 80 bits registradores de ponto flutuante, bem como um punhado de
special-purpose registra como a registros de controle% CR0,% CR2, CR3% e% cr4; o
debug registra% DR0,% dr1, dr2% e% dr3; os registos do segmento% cs,% ds,% es,
% fs,% gs, e ss%; ea mesa descritor global e local pseudo-registros% GDTR
e ldtr%. Os registros de controle e registros de segmento são importantes para qualquer operação
sistema. Os registros de ponto flutuante e de depuração são menos interessantes e não usado por
xv6.

Registros são rápidos, mas caro. A maioria dos processadores fornecer no máximo algumas dezenas de
registradores de uso geral. O próximo nível conceptual de armazenamento é a principal random-ac-
cesso de memória (RAM). A memória principal é 10-100x mais lento do que um registro, mas é muito mais
mais barata, de modo que não pode haver mais do mesmo. Uma razão a memória principal é relativamente lento é que
é fisicamente separado do chip do processador. Um processador x86 tem algumas dezenas
registros, mas um PC típico hoje tem gigabytes de memória principal. Devido à enor-
mous diferenças em termos de velocidade de acesso e de tamanho entre os registos e memória principal,
a maioria dos processadores, incluindo o x86, armazenar cópias de seções recentemente acessados de principal
memória em memória cache on-chip. A memória cache serve como um meio termo en-
registros de interpolação e de memória, tanto em tempo de acesso e de tamanho. Processadores x86 de hoje
normalmente têm dois níveis de cache um pequeno cache de primeiro nível, com tempos de acesso relativamente
próximo a velocidade do clock do processador e um cache maior de segundo nível, com os tempos de acesso em
entre o cache de primeiro nível e memória principal. Esta tabela mostra os números reais para
um sistema Intel Core 2 Duo:

Intel Core 2 Duo E7200 de 2,53 GHz
TODO: Conecte em números não-made-up!

armazenamento	tempo de acesso	tamanho
registrar	0,6 ns	64 bytes
Cache L1	0,5 ns	64 kilobytes
Cache L2	10 ns	4 megabytes
memória principal	100 ns	4 gigabytes

Página 85

Portas de E / S
mapeamento de memória
I / O

Para a maior parte, os processadores x86 esconder o cache do sistema operativo, portanto, pode pensar o processador como tendo apenas dois tipos de armazenamento-registos e memo- ry-e não se preocupar com as distinções entre os diferentes níveis de memória hierarquia.

I / O

Processadores deve se comunicar com dispositivos, bem como a memória. O processador x86 fornece especial dentro e fora instruções que lêem e escrevem os valores de ad- dispositivo vestidos chamado portas I / O. A implementação destas instruções de hardware é essencialmente o mesmo que ler e escrever de memória. Os primeiros processadores x86 teve uma ad- adicional linha de vestido: 0 significava leitura / gravação de uma porta de I / O e 1 significava leitura / escrita da principal memória. Cada dispositivo de hardware monitora estas linhas para lê e escreve a sua asgama assinado de portas I / O. Portas de um dispositivo que o software configurar o dispositivo, examine seu status, e fazer com que o dispositivo para tomar ações; por exemplo, o software pode usar I / O port lê e escreve para fazer com que o hardware da interface do disco para ler e escrever em setores o disco.

Muitas arquiteturas de computadores não têm instruções de acesso dispositivo separado. Em vez os dispositivos de ter corrigido endereços de memória e do processador se comunica com o dispositivo (a mando do sistema operacional) por ler e escrever valores para aqueles advestidos. Na verdade, arquiteturas modernas x86 usar esta técnica, chamada memória mapeada I / O, para a maioria dos dispositivos de alta velocidade, tais como rede, disco e controladores gráficos. Para razões de compatibilidade com versões anteriores, porém, o velho dentro e fora instruções demorar, como fazer dispositivos de hardware legado que os utilizam, como o controlador de disco IDE, que xv6 usa.

Página 87

endereço lógico
endereço linear
endereço físico

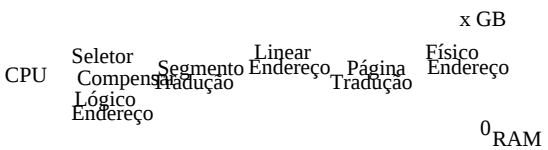


Figura 1-B . A relação entre o lógico, linear, e endereços físicos.

boot loader
boot loader
modo real

Apêndice B

O gerenciador de inicialização

Quando um botas x86 PC, ele começa a executar um programa chamado BIOS, que é armazenados na memória não-volátil na placa-mãe. O trabalho do BIOS é para preparar o hardware e, em seguida, transferir o controle para o sistema operacional. Especificamente, ele transfere controle de código carregado do setor de boot, o primeiro setor de 512 bytes do disco de boot. O setor de inicialização contém o carregador de boot: instruções que carregam o kernel na mem- ory. O BIOS carrega o setor de inicialização no endereço de memória 0x7c00 e depois salta (conjuntos % ip do processador) para esse endereço. Quando o carregador de boot começa a executar, a pro- cessor está simulando um processador Intel 8088, e o trabalho do carregador é colocar o processador em um modo de operação mais moderno, para carregar o kernel xv6 do disco na memória, e em seguida, para transferir o controle para o kernel. O gerenciador de inicialização xv6 compreende dois arquivos de origem, um escrito em uma combinação de (bootasm.S de 16 bits e de montagem x86 de 32 bits; (8400) e um escrito em C (bootmain.c; (8500).

Código: inicialização Assembleia

A primeira instrução no carregador de boot é cli (8412) Que desativa in- processador terrupts. Interrupções são uma maneira para que os dispositivos de hardware para invocar sistema operacional função ções chamados manipuladores de interrupção. O BIOS é um pequeno sistema operacional, e que poderia ter criar seus próprios manipuladores de interrupção, como parte da inicialização do hardware. Mas o BIOS não está mais rodando-o carregador de boot é-para que ele não é adequado ou seguro para lidar com as interrupções de dispositivos de hardware. Quando xv6 está pronto (no Capítulo 3), ele irá reativar interrupções.

O processador é em modo real, em que simula um Intel 8088. Em modo real há oito de 16 bits registradores de uso geral, mas o processador envia 20 bits de ad- vestir a memória. O segmento registra% cs, ds%,% es, e% ss fornecer o adicional bits necessários para gerar endereços de memória de 20 bits a partir de registros de 16 bits. Quando um pro- g refere-se a um endereço de memória, o processador adiciona automaticamente 16 vezes o valor de um dos registradores de segmento; Esses registros são de 16 bits de largura. Quais regis- segmento ter é geralmente implícita no tipo de referência à memória: a instrução busca usar% cs, dados lê e escreve usar% ds, e pilha lê e escreve uso% ss.

Página 88

Xv6 finge que uma instrução x86 usa um endereço virtual para a sua memória operandos, mas uma instrução x86 realmente usa um endereço lógico (ver Figura B-1) . A endereço lógico é composto por um seletor de segmento e um deslocamento, e às vezes é escrito como segmento: offset. Mais frequentemente, o segmento é implícita e o programa só diretamente manipula o deslocamento. O hardware segmentação realiza a tradução descrito acima para gerar um endereço linear. Se o hardware de paginação está habilitado (ver Capítulo 2), que traduz endereços lineares para endereços físicos; caso contrário, o processador usa lin- endereços de ouvido como endereços físicos.

boot loader
endereço lógico
endereço linear
endereço virtual
Modo protegido
descritor de segmento
tabela
código gdt +

O carregador não tem que permitir que o hardware de paginação; os endereços lógicos que ele usos são convertidos para endereços lineares pelo harware segmentação, e então utilizado di- tamente como endereços físicos. Xv6 configura o hardware segmentação de traduzir endereços lógico linear sem alteração, de modo que são sempre iguais. Para histórica razões que têm utilizado o endereço virtual termo para se referir aos endereços manipulados por programas; um endereço virtual xv6 é o mesmo que um endereço lógico x 86, e é igual a o endereço linear para que o hardware segmentação mapeia. Depois de paginação é en- abled, o único endereço de mapeamento interessante no sistema será linear para físico.

O BIOS não garante nada sobre o conteúdo do ds%,%,% es ss, de modo primeira ordem do negócio depois de desativar as interrupções é definir% machado para zero e, em seguida, copiar que o valor zero% ds,% es, e ss% (8415-8418)

Um segmento virtual: compensado pode render um endereço físico de 21 bits, mas o Intel 8088 só poderia resolver 20 bits de memória, por isso descartou a top bit: 0xFFFF0 + 0xffff = 0x10ffef, mas 0xffff endereço virtual: 0xffff no 8088 referido endereço físico 0x0ffef. Alguns softwares início contou com o hardware ignorando o bit endereço 21, de modo quando a Intel lançou processadores com mais de 20 bits do endereço físico, IBM pro- DESDE um hack de compatibilidade que é um requisito para hardware compatível com PC. Se o segundo bit de porta de saída do controlador de teclado é baixa, a 21 bit endereço físico é sempre apagada; se for alta, o bit 21 actua normalmente. O gerenciador de inicialização deve habilitar o 21 bit endereço usando I / O para o controlador do teclado em portas 0x64 e 0x60 (8420- 8436).

16-bit de uso geral e de segmento registros de modo real torná-lo um pouco estranho para um

programa para usar mais de 65.536 bytes de memória, e impossível de utilizar mais do que um megabyte. processadores x86 desde a 80286 tem um modo protegido, o que permite phys- endereços iCal para ter muitos mais bits, e no modo (uma vez que o 80386) a "32 de bits' 'que provoca registos, endereços virtuais, e mais aritmética inteira de ser levada a cabo com 32 bits, em vez de 16. A seqüência de inicialização xv6 permite modo protegido e 32-bit o modo como se segue.

No modo protegido, um registrador de segmento é um índice para um descritor de segmento mesa (ver Figura B-2) . Cada entrada da tabela especifica um endereço físico base, um máximo endereço virtual chamado de limite, e bits de permissão para o segmento. Estes permissível sões são a proteção em modo protegido: o kernel pode usá-los para garantir que um programa usa apenas sua própria memória.

xv6 quase não faz uso de segmentos; que utiliza o hardware paginação em vez disso, como O capítulo 2 descreve. O gerenciador de inicialização configura o gdt tabela descritor segmento (8482-8485) de modo que todos os segmentos têm um endereço de base igual a zero e o limite máximo possível (quatro gigabytes). A tabela tem uma entrada nula, uma entrada para o código executável, e um en-

Página 89

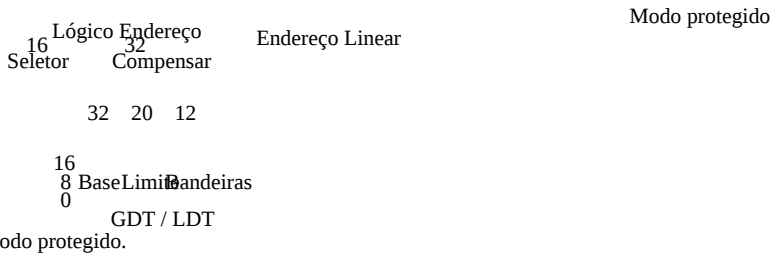


Figura B-2 . Segmentos em modo protegido.

tente de dados. O descritor de segmento de código tem um conjunto indicador que indica que o código deve ser executado no modo de 32 bits. Com esta configuração, quando o carregador de boot entra prote- modo ed, endereços lógicos mapear one-to-one para endereços físicos.

O gerenciador de inicialização executa uma instrução lgdt (8441) para carregar o processador de mundial tabela descritor (GDT) registrar com o valor gdtdesc (8487-8489) O que aponta para o gdt tabela.

Depois de ter carregado o registro GDT, o carregador de boot permite o modo protegido por definindo o 1 bit (CR0_PE) no registrador% CR0 (8442-8444) Ativando o modo protegido faz não mudar imediatamente como o processador traduz lógico endereços físicos; it is somente quando se carrega um novo valor em um registrador de segmento que o processador lê o GDT e muda suas configurações de segmentação interna. Não se pode modificar diretamente% cs, assim, em vez do código executa um (salto agora) instrução LJMP (8453) Que permite que um código seletor de segmento a ser especificado. O salto continua a execução na linha seguinte (8456) mas ao fazê-lo define% cs para se referir à entrada do descritor código em gdt. Esse descritor descreve um segmento de código de 32 bits, de modo que o processador comuta para o modo de 32 bits. A bota loader tem nutrido o processador através de uma evolução a partir de 8088 por meio de 80.286 para 80386.

A primeira ação da carregador de boot no modo de 32 bits é inicializar o segmento de dados reg- isters com SEG_KDATA (8458-8461) Endereço lógico agora mapear diretamente para ad- física vestidos. O único passo à esquerda antes de executar o código C é a criação de uma pilha em um conjunto de tomadas região de memória. A memória de 0xa0000 para 0x100000 normalmente é cheio de regiões memória do dispositivo, e o kernel xv6 espera para ser colocado em 0x100000. O próprio carregador de inicialização está em 0x7c00 através 0x7d00. Essencialmente qualquer outra seção do memo- ry seria um ótimo local para a pilha. O gerenciador de inicialização escolhe 0x7c00 (conhecido em este ficheiro como começar $) como o topo da pilha; a pilha vai crescer para baixo de lá, to- ala 0x0000, longe do carregador de boot.

Finalmente, o carregador de inicialização chama a função bootmain (8463) O trabalho de Bootmain é carregar e executar o kernel. Ele só retorna se algo deu errado. Neste caso, o código envia algumas palavras de saída na porta 0x8a00 (8470-8476) Em hardware real, há há nenhum dispositivo conectado a essa porta, de modo que este código não faz nada. Se o carregador de boot é correndo dentro de um simulador de PC, 0x8a00 porta está ligado ao próprio simulador e pode transferir o controle de volta para o simulador. Simulator ou não, o código, em seguida, executa uma loop infinito (8477-8478) Um carregador de inicialização real, pode tentar imprimir uma mensagem de erro primeiro.

boot loader
descritor mundial
tabela
gdtdesc + código
código gdt +
Código CR0_PE +
código gdt +
Código SEG_KDATA +
bootmain + código

Página 90

Código: C de bootstrap

código readseg +
código stosb +
_start + código
código de entrada +

A parte C do carregador de boot, bootmain.c (8500) Espera-se encontrar uma cópia do executável do kernel no disco a partir do segundo sector. O kernel é um ELF formato binário, como vimos no Capítulo 2. Para ter acesso aos cabeçalhos ELF, bootmain carrega os primeiros 4096 bytes do binário ELF (8514) Ele coloca a cópia na memória a endereços 0x10000.

O próximo passo é uma verificação rápida que isso provavelmente é um binário ELF, e não um disco não inicializado. Bootmain lê da seção de conteúdo a partir da localização de disco off bytes após o início do cabeçalho ELF, e escreve para a memória a partir do endereço paddr. Bootmain chama readseg para carregar dados de disco (8538) e chamadas STOSB a zero o restante do segmento (8540) Stosb (0492) usa a instrução x86 rep stosb para inicializar cada byte de um bloco de memória.

O kernel foi compilado e ligado para que ele espera encontrar-se em virtual a partir de endereços 0x80100000. Ou seja, as instruções de chamada de função mencionar destino endereços ção que se parecem com 0xf01xxxxx; você pode ver exemplos no kernel.asm. Este endereço é configurado em kernel.ld. 0x80100000 é um endereço relativamente elevado, no sentido a fim de o espaço de endereço de 32 bits; Capítulo 2 explica as razões para esta escolha. Pode não haver qualquer memória física em um endereço tão elevado. Uma vez iniciada a do kernel execução, ele irá configurar o hardware de paginação para mapear endereços virtuais a partir de 0x80100000 para endereços físicos a partir de 0x00100000; o kernel assume que há memória física neste endereço inferior. Neste ponto do processo de inicialização, contudo nunca, paginação não está habilitado. Em vez disso, kernel.ld especifica que o ELF paddr começam em 0x00100000, o que faz com que o gerenciador de inicialização para copiar o kernel para o baixo ad- física vestidos para que o hardware de paginação serão eventualmente apontar.

Etapa final do boot loader é chamar ponto de entrada do kernel, que é a instrução ção em que o kernel espera para iniciar a execução. Para xv6 o endereço de entrada é 0x10000c:

```
# Objdump do kernel -f

do kernel:     formato de arquivo ELF32-i386
Arquitetura: i386, bandeiras 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
endereço inicial 0x0010000c
```

Por convenção, o símbolo _start especifica o ponto de entrada ELF, que é definido nas os entry.S arquivo (1036) Desde xv6 ainda não definiu a memória virtual ainda, ponto de entrada de xv6 é o endereço físico de entrada (1040)

Mundo real

O gerenciador de inicialização descrito neste apêndice compila para cerca de 470 bytes de macódigo de chine, dependendo das otimizações utilizadas na elaboração do código C. Em ouder para caber em que a pequena quantidade de espaço, o carregador de boot xv6 faz um grande simplificando ing hipótese, que o kernel foi escrito para o disco de inicialização de forma contígua começando no sector 1. Mais comumente, kernels são armazenados em sistemas de arquivos comuns, onde eles não podem ser contíguos, ou são carregados através de uma rede. Estas complicações requerem a

Página 91

gerenciador de inicialização para ser capaz de conduzir uma variedade de controladores e de disco e rede compreensão ficar vários sistemas de arquivos e protocolos de rede. Em outras palavras, o próprio carregador de inicialização deve ser um pequeno sistema operativo. Uma vez que tais gestores de início complicado certamente não será caber em 512 bytes, a maioria dos sistemas operacionais de PC usam um processo de inicialização de duas etapas. Em primeiro lugar, uma simboot loader ple como aquele neste apêndice carrega um boot-loader cheio de recursos a partir de um localização do disco conhecido, muitas vezes contando com a menos BIOS com restrição de espaço para acesso ao disco em vez de tentar conduzir o disco propriamente dito. Em seguida, o carregador cheio, aliviado do 512-byte limite, pode implementar a complexidade necessária para localizar, carregar e executar o desejado

kernel. Talvez um design mais moderno teria o BIOS ler diretamente um maior
carregador de inicialização do disco (e iniciá-lo em modo protegido e 32-bit).

Este apêndice é escrito como se a única coisa que acontece entre a energia e
a execução do gerenciador de inicialização é que o BIOS carrega o setor de inicialização. Na verdade, a
BIOS faz uma enorme quantidade de inicialização, a fim de tornar o complexo de hardware
um olhar moderno computador como um padrão PC tradicional.

Exercícios

1. Devido à granularidade sector, a chamada para readseg no texto é equivalente a leitura-
seg ((uchar *) 0x100000, 0xb500, 0x1000). Na prática, esse comportamento desleixado transforma
por não ser um problema Por que não o desleixado causar problemas readsect?
2. algo sobre BIOS com duração + problemas de segurança
3. Suponha que você queria bootmain () para carregar o kernel em vez de 0x200000
0x100000, e você fez isso, modificando bootmain () para adicionar 0x100000 à va de cada
Seção ELF. Algo iria dar errado. O Quê?
4. Parece potencialmente perigoso para o carregador de boot para copiar o cabeçalho ELF para os Estados
ory no local arbitrário 0x10000. Por que não chamar malloc obter o memo-
ry que precisa?

Página 93

Índice

Página 94

Página 95

Página 96