xv6: um sistema operacional de ensino simples, semelhante ao Unix

Russ Cox Frans Kaashoek Robert Morris

5 de setembro de 2022

Conteúdo

Prefácio e agradecimentos

Este é um rascunho de texto destinado a uma aula sobre sistemas operacionais. Ele explica os principais conceitos de sistemas operacionais por meio do estudo de um kernel de exemplo, denominado xv6. O Xv6 é modelado no Unix Versão 6 (v6) de Dennis Ritchie e Ken Thompson [17]. O Xv6 segue vagamente a estrutura e o estilo do v6, mas é implementado em ANSI C [7] para um RISC-V multi-core [15].

Este texto deve ser lido em conjunto com o código-fonte para xv6, uma abordagem inspirada no Comentário de John Lions sobre o UNIX 6ª Edição [11]. Consulte https://pdos.csail.mit.edu/6.1810 para obter referências a recursos online para v6 e xv6, incluindo diversas tarefas de laboratório usando xv6 .

Utilizamos este texto nas disciplinas 6.828 e 6.1810, disciplinas de sistemas operacionais do MIT. Agradecemos ao corpo docente, aos assistentes de ensino e aos alunos dessas disciplinas que contribuíram direta ou indiretamente para o xv6. Em particular, gostaríamos de agradecer a Adam Belay, Austin Clements e Nickolai Zeldovich. Por fim, gostaríamos de agradecer às pessoas que nos enviaram e-mails com bugs no texto ou sugestões de melhorias: Abutalib Aghayev, Sebastian Boehm, brandb97, Anton Burtsev, Raphael Carvalho, Tej Chajed, Rasit Eskicioglu, Color Fuzzy, Wojciech Gac, Giuseppe, Tao Guo, Haibo Hao, Naoki Hayama, Chris Henderson, Robert Hilderman, Eden Hochbaum, Wolfgang Keller, Henry Laih, Jin Li, Austin Liew, Pavan Maddamsetti, Jacek Masiulaniec, Michael McConville, m3hm00d, miguelgvieira, Mark Morrissey, Muhammed Mourad, Harry Pan, Harry Porter, Siyuan Qian, Askar Safin, Salman Shah, Huang Sha, Vikram Shenoy, Adeodato Simó, Ruslan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda, tzerbib, Vanush Vaswani, Xi Wang e Zou Chang Wei, Sam Whitlock, LucyShawYang e Meng Zhou

Se você detectar erros ou tiver sugestões de melhoria, envie um e-mail para Frans Kaashoek e Robert Morris (kaashoek,rtm@csail.mit.edu).

Capítulo 1

Interfaces do sistema operacional

A função de um sistema operacional é compartilhar um computador entre vários programas e fornecer um conjunto de serviços mais útil do que o hardware suporta sozinho. Um sistema operacional gerencia e abstrai o hardware de baixo nível, de modo que, por exemplo, um processador de texto não precise se preocupar com o tipo de hardware de disco que está sendo usado. Um sistema operacional compartilha o hardware entre vários programas para que eles sejam executados (ou pareçam ser executados) ao mesmo tempo. Por fim, os sistemas operacionais fornecem maneiras controladas para os programas interagirem, para que possam compartilhar dados ou trabalhar em conjunto.

Um sistema operacional fornece serviços aos programas do usuário por meio de uma interface. Projetar uma boa interface acaba sendo difícil. Por um lado, gostaríamos que a interface fosse simples e concisa, pois isso facilita a implementação correta. Por outro lado, podemos ser tentados a oferecer muitos recursos sofisticados aos aplicativos. O segredo para resolver essa tensão é projetar interfaces que dependam de alguns mecanismos que possam ser combinados para fornecer muita generalidade.

Este livro utiliza um único sistema operacional como exemplo concreto para ilustrar conceitos de sistema operacional. Esse sistema operacional, o xv6, fornece as interfaces básicas introduzidas pelo sistema operacional Unix de Ken Thompson e Dennis Ritchie [17], além de imitar o design interno do Unix.
O Unix oferece uma interface estreita cujos mecanismos se combinam bem, oferecendo um grau surpreendente de generalidade. Essa interface tem sido tão bem-sucedida que os sistemas operacionais modernos — BSD, Linux, macOS, Solaris e até mesmo, em menor grau, o Microsoft Windows — possuem interfaces semelhantes às do Unix. Entender o xv6 é um bom começo para entender qualquer um desses sistemas e muitos outros.

Como mostra a Figura 1.1, o xv6 assume a forma tradicional de um kernel, um programa especial que fornece serviços aos programas em execução. Cada programa em execução, chamado de processo, possui memória contendo instruções, dados e uma pilha. As instruções implementam a computação do programa. Os dados são as variáveis sobre as quais a computação atua. A pilha organiza as chamadas de procedimento do programa.
Um determinado computador normalmente tem muitos processos, mas apenas um único kernel.

Quando um processo precisa invocar um serviço do kernel, ele invoca uma chamada de sistema, uma das chamadas na interface do sistema operacional. A chamada de sistema entra no kernel; o kernel executa o serviço e retorna. Assim, um processo alterna entre a execução no espaço do usuário e no espaço do kernel.

Conforme descrito em detalhes nos capítulos subsequentes, o kernel usa os mecanismos de proteção de hardware fornecidos por uma CPU1 para garantir que cada processo em execução no espaço do usuário possa acessar apenas

1Este texto geralmente se refere ao elemento de hardware que executa uma computação com o termo CPU, uma sigla

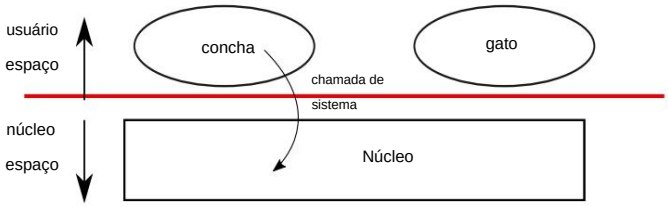

Figura 1.1: Um kernel e dois processos de usuário.

sua própria memória. O kernel executa com os privilégios de hardware necessários para implementar essas proteções; os programas do usuário são executados sem esses privilégios. Quando um programa do usuário invoca uma chamada de sistema, o hardware aumenta o nível de privilégio e começa a executar uma função pré-definida no kernel.

O conjunto de chamadas de sistema que um kernel fornece é a interface que os programas do usuário veem. O kernel xv6 fornece um subconjunto dos serviços e chamadas de sistema que os kernels Unix tradicionalmente oferecem.
A Figura 1.2 lista todas as chamadas de sistema do xv6.

O restante deste capítulo descreve os serviços do xv6 — processos, memória, descritores de arquivo, pipes e um sistema de arquivos — e os ilustra com trechos de código e discussões sobre como o shell, a interface de linha de comando do Unix, os utiliza. O uso de chamadas de sistema pelo shell ilustra o cuidado com que foram projetados.

O shell é um programa comum que lê comandos do usuário e os executa. O fato de o shell ser um programa do usuário, e não parte do kernel, ilustra o poder da interface de chamada de sistema: não há nada de especial no shell. Isso também significa que o shell é fácil de substituir; como resultado, os sistemas Unix modernos têm uma variedade de shells para escolher, cada um com sua própria interface de usuário e recursos de script. O shell xv6 é uma implementação simples da essência do shell Bourne do Unix. Sua implementação pode ser encontrada em (user/sh.c:1).

1.1 Processos e memória

Um processo xv6 consiste em memória de espaço do usuário (instruções, dados e pilha) e estado por processo, privado do kernel. O Xv6 compartilha o tempo dos processos: ele alterna de forma transparente as CPUs disponíveis entre o conjunto de processos aguardando execução. Quando um processo não está em execução, o xv6 salva os registradores de CPU do processo, restaurando-os na próxima execução. O kernel associa um identificador de processo, ou PID, a cada processo.

Um processo pode criar um novo processo usando a chamada de sistema fork . Fork fornece ao novo processo uma cópia exata da memória do processo que fez a chamada, tanto instruções quanto dados. Fork retorna tanto no processo original quanto no novo. No processo original, fork retorna o PID do novo processo. No novo processo, fork retorna zero. Os processos original e novo são frequentemente chamados de pai e filho.

---

para unidade central de processamento. Outra documentação (por exemplo, a especificação RISC-V) também utiliza as palavras processador, núcleo e hart em vez de CPU.

Chamada de sistema	Descrição
int fork() int	Crie um processo e retorne o PID do filho.
exit(int status) int	Encerra o processo atual; status reportado para wait(). Sem retorno.
wait(int *status) int kill(int	Aguarde a saída de uma criança; saia do status em *status; retorna o PID da criança.
pid) int getpid() int	Encerra o PID do processo. Retorna 0 ou -1 em caso de erro.
sleep(int n) int	Retorna o PID do processo atual.
exec(char *file, char	Pausa por n tiques de relógio.
*argv[]) Carrega um arquivo e o executa com argumentos; retorna somente se houver erro.	
char *sbrk(int n) int	Aumenta a memória do processo em n bytes. Retorna o início da nova memória.
open(char *file, int flags) int write(int	Abre um arquivo; sinalizadores indicam leitura/gravação; retorna um fd (descritor de arquivo).
fd, char *buf, int n) Grava n bytes de buf no descritor de arquivo fd; retorna n.	
int read(int fd, char *buf, int n) Lê n bytes em buf; retorna o número lido; ou 0 se for o fim do arquivo.	
int fechar(int fd)	Libere o arquivo aberto fd.
int dup(int fd)	Retorna um novo descritor de arquivo referindo-se ao mesmo arquivo que fd.
int pipe(int p[])	Crie um pipe, coloque descritores de arquivo de leitura/gravação em p[0] e p[1].
int chdir(char *dir)	Altera o diretório atual.
int mkdir(char *dir)	Crie um novo diretório.
int mknod(char *arquivo, int, int)	Crie um arquivo de dispositivo.
int fstat(int fd, struct stat *st)	Coloque informações sobre um arquivo aberto em *st.
int stat(char *file, struct stat *st) Coloque informações sobre um arquivo nomeado em *st.	
int link(char *file1, char *file2) Cria outro nome (file2) para o arquivo file1.	
int unlink(char *arquivo)	Remover um arquivo.

Figura 1.2: Chamadas de sistema Xv6. Se não indicado de outra forma, essas chamadas retornam 0 para nenhum erro e -1 se há um erro.

Por exemplo, considere o seguinte fragmento de programa escrito na linguagem de programação C [7]:

```
int pid = garfo();
se(pid > 0){
    printf("pai: filho=%d\n", pid);
    pid = esperar((int *) 0);
    printf("criança %d terminou\n", pid);
} senão se(pid == 0){
    printf("filho: saindo\n");
    saída(0);
} outro {
    printf("erro de bifurcação\n");
}
```

A chamada de sistema exit faz com que o processo de chamada pare de executar e libere recursos como memória e arquivos abertos. Exit recebe um argumento de status inteiro, convencionalmente 0 para indicar sucesso e 1 para indicar falha. A chamada de sistema wait retorna o PID de um filho saído (ou eliminado) de

o processo atual e copia o status de saída do filho para o endereço passado para wait; se nenhum dos filhos do chamador tiver saído, wait aguarda que um o faça. Se o chamador não tiver filhos, wait retorna imediatamente -1. Se o pai não se importar com o status de saída de um filho, ele pode passar um endereço 0 para wait.

No exemplo, as linhas de saída

```
pai: filho=1234 filho: saindo
```

pode sair em qualquer ordem (ou até mesmo misturados), dependendo se o pai ou o filho chega primeiro à sua chamada printf . Após a saída do filho, a espera do pai retorna, fazendo com que o pai imprima

```
pai: filho 1234 está pronto
```

Embora o processo filho tenha inicialmente o mesmo conteúdo de memória que o processo pai, ambos estão executando com memória e registradores separados: alterar uma variável em um não afeta o outro. Por exemplo, quando o valor de retorno de wait é armazenado em pid no processo pai, isso não altera a variável pid no processo filho. O valor de pid no processo filho ainda será zero.

A chamada de sistema exec substitui a memória do processo que fez a chamada por uma nova imagem de memória carregada de um arquivo armazenado no sistema de arquivos. O arquivo deve ter um formato específico, que especifica qual parte do arquivo contém instruções, qual parte são dados, em qual instrução iniciar, etc. O Xv6 usa o formato ELF, que o Capítulo 3 discute em mais detalhes. Normalmente, o arquivo é o resultado da compilação do código-fonte de um programa. Quando exec é bem-sucedido, ele não retorna ao programa que fez a chamada; em vez disso, as instruções carregadas do arquivo começam a ser executadas no ponto de entrada declarado no cabeçalho ELF. exec recebe dois argumentos: o nome do arquivo que contém o executável e uma matriz de argumentos de string. Por exemplo:

```
char *argv[3];

argv[0] = "eco"; argv[1] =
"olá"; argv[2] = 0; exec("/bin/
eco", argv);
printf("erro de execução\n");
```

Este fragmento substitui o programa chamador por uma instância do programa /bin/echo em execução com a lista de argumentos echo hello. A maioria dos programas ignora o primeiro elemento da matriz de argumentos, que é convencionalmente o nome do programa.

O shell xv6 usa as chamadas acima para executar programas em nome dos usuários. A estrutura principal do shell é simples; veja main (user/sh.c:146). O loop principal lê uma linha de entrada do usuário com getcmd. Em seguida, chama fork, que cria uma cópia do processo do shell. O processo pai chama wait, enquanto o processo filho executa o comando. Por exemplo, se o usuário tivesse digitado "echo hello" no shell, runcmd teria sido chamado com "echo hello" como argumento. runcmd (user/sh.c:55) executa o comando real. Para "echo hello", ele chamaria exec (user/sh.c:79). Se exec for bem-sucedido, o filho executará instruções de echo em vez de runcmd. Em algum momento, echo chamará exit, o que fará com que o pai retorne de wait em main (user/sh.c:146).

Você pode se perguntar por que fork e exec não são combinados em uma única chamada; veremos mais adiante que o shell explora essa separação em sua implementação de redirecionamento de E/S. Para evitar o desperdício de criar um processo duplicado e substituí-lo imediatamente (por exec), os kernels operacionais otimizam a implementação de fork para esse caso de uso usando técnicas de memória virtual, como cópia na gravação (consulte a Seção 4.6).

O Xv6 aloca a maior parte da memória do espaço do usuário implicitamente: fork aloca a memória necessária para a cópia filha da 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) para aumentar sua memória de dados em n bytes; sbrk retorna a localização da nova memória.

1.2 E/S e descritores de arquivo

Um descritor de arquivo é um pequeno inteiro que representa um objeto gerenciado pelo kernel do qual um processo pode ler ou gravar. Um processo pode obter um descritor de arquivo abrindo um arquivo, diretório ou dispositivo, criando um pipe ou duplicando um descritor existente. Para simplificar, frequentemente nos referimos ao objeto ao qual um descritor de arquivo se refere como "arquivo"; a interface do descritor de arquivo abstrai as diferenças entre arquivos, pipes e dispositivos, fazendo com que todos pareçam fluxos de bytes. Nos referiremos à entrada e à saída como E/S.

Internamente, o kernel xv6 usa o descritor de arquivo como um índice em uma tabela por processo, de modo que cada processo tenha um espaço privado de descritores de arquivo começando em zero. Por convenção, um processo lê do descritor de arquivo 0 (entrada padrão), grava a saída no descritor de arquivo 1 (saída padrão) e grava mensagens de erro no descritor de arquivo 2 (erro padrão). Como veremos, o shell explora a convenção para implementar redirecionamentos de E/S e pipelines. O shell garante que sempre tenha três descritores de arquivo abertos (user/sh.c:152). que são, por padrão, descritores de arquivo para o console.

O sistema de leitura e gravação chama bytes de leitura e bytes de gravação para abrir arquivos nomeados por descritores de arquivo. A chamada read(fd, buf, n) lê no máximo n bytes do descritor de arquivo fd, copia-os para buf e retorna o número de bytes lidos. Cada descritor de arquivo que se refere a um arquivo possui um deslocamento associado. read lê os dados do deslocamento do arquivo atual e, em seguida, avança esse deslocamento pelo número de bytes lidos: uma leitura subsequente retornará os bytes seguintes aos retornados pela primeira leitura. Quando não há mais bytes para ler, read retorna zero para indicar o fim do arquivo.

A chamada write(fd, buf, n) grava n bytes de buf no descritor de arquivo fd e retorna o número de bytes gravados. Menos de n bytes são gravados somente quando ocorre um erro. Assim como read, write grava dados no deslocamento do arquivo atual e, em seguida, avança esse deslocamento pelo número de bytes gravados: cada gravação continua de onde a anterior parou.

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

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


para(;;){
```

```
    n = ler(0, buf, tamanho de buf); se(n == 0) quebrar;
    se(n < 0)
        { fprintf(2,
    "erro de
        leitura\n"); sair(1);


    } se(escrever(1, buf, n) != n){
        fprintf(2, "erro de gravação\n"); exit(1);

    }
}
```

O importante a ser observado no fragmento de código é que cat não sabe se está lendo de um arquivo, console ou pipe. Da mesma forma, cat não sabe se está imprimindo para um console, um arquivo ou qualquer outro lugar. O uso de descritores de arquivo e a convenção de que o descritor de arquivo 0 é a entrada e o descritor de arquivo 1 é a saída permitem uma implementação simples de cat.

A chamada de sistema close libera um descritor de arquivo, tornando-o livre para reutilização em futuras chamadas de sistema open, pipe ou dup (veja abaixo). Um descritor de arquivo recém-alocado é sempre o descritor não utilizado de menor numeração do processo atual.

Descritores de arquivo e fork interagem para facilitar a implementação do redirecionamento de E/S. O fork copia a tabela de descritores de arquivo do processo pai juntamente com sua memória, de modo que o processo filho comece com exatamente os mesmos arquivos abertos que o processo pai. A chamada de sistema exec substitui a memória do processo que fez a chamada, mas preserva sua tabela de arquivos. Esse comportamento permite que o shell implemente o redirecionamento de E/S por meio de fork, reabrindo os descritores de arquivo escolhidos no processo filho e, em seguida, chamando exec para executar o novo programa. Aqui está uma versão simplificada do código que um shell executa para o comando cat < input.txt:

```
char *argv[2];

argv[0] = "gato"; argv[1] =
0; if(fork() == 0)
{ fechar(0); abrir("input.txt",
    O_RDONLY);
    exec("gato", argv);

}
```

Após o processo filho fechar o descritor de arquivo 0, open garante que usará esse descritor de arquivo para o arquivo input.txt recém-aberto: 0 será o menor descritor de arquivo disponível. cat então executa com o descritor de arquivo 0 (entrada padrão) referindo-se ao arquivo input.txt. Os descritores de arquivo do processo pai não são alterados por esta sequência, pois ela modifica apenas os descritores do processo filho.

O código para redirecionamento de E/S no shell xv6 funciona exatamente dessa maneira (user/sh.c:83). Lembre-se de que neste ponto do código o shell já bifurcou o shell filho e que runcmd chamará exec para carregar o novo programa.

O segundo argumento para open consiste em um conjunto de sinalizadores, expressos como bits, que controlam o que open faz. Os valores possíveis são definidos no cabeçalho de controle de arquivo (fcntl) (kernel/fcntl.h:1-5):

O_RDONLY, O_WRONLY, O_RDWR, O_CREATE e O_TRUNC, que instruem open a abrir o arquivo para leitura, ou para gravação, ou para leitura e gravação, para criar o arquivo se ele não existir e para truncá-lo para comprimento zero.

Agora deve ficar claro por que é útil que fork e exec sejam chamadas separadas: entre as duas, o shell tem a chance de redirecionar a E/S do shell filho sem perturbar a configuração de E/S do shell principal. Pode-se, em vez disso, imaginar uma hipotética chamada de sistema combinada forkexec , mas as opções para realizar o redirecionamento de E/S com tal chamada parecem estranhas. O shell poderia modificar sua própria configuração de E/S antes de chamar forkexec (e então desfazer essas modificações); ou forkexec poderia receber instruções de redirecionamento de E/S como argumentos; ou (o que é menos atraente) todo programa, como cat, poderia ser ensinado a fazer seu próprio redirecionamento de E/S.

Embora o fork copie a tabela do descritor de arquivo, cada deslocamento de arquivo subjacente é compartilhado entre Pais e filhos. Considere este exemplo:

```
if(fork() == 0) { write(1, "olá
    ", 6); exit(0); } else { wait(0); write(1,
    "mundo\n",
6);


}
```

No final deste fragmento, o arquivo anexado ao descritor de arquivo 1 conterá os dados hello world.
A gravação no pai (que, graças à espera, só é executada após a conclusão do filho) continua de onde a gravação do filho parou. Esse comportamento ajuda a produzir uma saída sequencial a partir de sequências de comandos de shell, como (echo hello; echo world) >output.txt.

A chamada de sistema dup duplica um descritor de arquivo existente, retornando um novo que se refere ao mesmo objeto de E/S subjacente. Ambos os descritores de arquivo compartilham um deslocamento, assim como os descritores de arquivo duplicados por fork . Esta é outra maneira de escrever hello world em um arquivo:

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

Dois descritores de arquivo compartilham um deslocamento se forem derivados do mesmo descritor de arquivo original por uma sequência de chamadas fork e dup . Caso contrário, os descritores de arquivo não compartilham deslocamentos, mesmo que tenham resultado de chamadas open para o mesmo arquivo. O dup permite que shells implementem comandos como este: ls existing-file non-existing-file > tmp1 2>&1. O 2>&1 informa ao shell para atribuir ao comando um descritor de arquivo 2, que é uma duplicata do descritor 1. Tanto o nome do arquivo existente quanto a mensagem de erro para o arquivo inexistente serão exibidos no arquivo tmp1. O shell xv6 não suporta redirecionamento de E/S para o descritor de arquivo de erro, mas agora você sabe como implementá-lo.

Os descritores de arquivo são uma abstração poderosa, porque eles escondem os detalhes do que estão conectados: um processo escrevendo no descritor de arquivo 1 pode estar escrevendo em um arquivo, em um dispositivo como o console ou em um pipe.

1.3 Tubos

Um pipe é um pequeno buffer de kernel exposto aos processos como um par de descritores de arquivo, um para leitura e outro para escrita. A gravação de dados em uma extremidade do pipe torna esses dados disponíveis para leitura na outra extremidade. Pipes fornecem uma maneira para os processos se comunicarem.

O código de exemplo a seguir executa o programa wc com a entrada padrão conectada à extremidade de leitura de um pipe.

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

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

pipe(p);
if(fork() == 0) { fechar(0);
    dup(p[0]);
    fechar(p[0]);
    fechar(p[1]); exec("/
    bin/wc", argv); } else
    { fechar(p[0]); escrever(p[1], "olá
 mundo\n",
    12); fechar(p[1]);


}
```

O programa chama pipe, que cria um novo pipe e registra os descritores de arquivo de leitura e gravação no array p. Após a bifurcação, tanto o pai quanto o filho possuem descritores de arquivo referentes ao pipe. O filho chama close e dup para fazer com que o descritor de arquivo zero se refira à extremidade de leitura do pipe, fecha os descritores de arquivo em p e chama exec para executar wc. Quando wc lê de sua entrada padrão, ele lê do pipe. O pai fecha o lado de leitura do pipe, escreve no pipe e, em seguida, fecha o lado de gravação.

Se não houver dados disponíveis, uma leitura em um pipe aguarda que os dados sejam gravados ou que todos os descritores de arquivo referentes à extremidade de gravação sejam fechados; neste último caso, a leitura retornará 0, como se o fim de um arquivo de dados tivesse sido alcançado. O fato de a leitura bloquear até que seja impossível a chegada de novos dados é um dos motivos pelos quais é importante que o filho feche a extremidade de gravação do pipe antes de executar o comando wc acima: se um dos descritores de arquivo de wc se referisse à extremidade de gravação do pipe, wc nunca veria o fim do arquivo.

O shell xv6 implementa pipelines como grep fork sh.c | wc -l de maneira semelhante ao código acima (usuário/sh.c:101). O processo filho cria um pipe para conectar a extremidade esquerda do pipeline com a extremidade direita. Em seguida, ele chama fork e runcmd para a extremidade esquerda do pipeline e fork e runcmd para a extremidade direita, e aguarda a conclusão de ambos. A extremidade direita do pipeline pode ser um comando que inclui um pipe (por exemplo, a | b | c), que bifurca dois novos processos filhos (um para b e outro para c). Assim, o shell pode criar uma árvore de processos. As folhas

dessa árvore são comandos e os nós internos são processos que esperam até que os filhos esquerdo e direito sejam concluídos.

Os pipes podem não parecer mais poderosos do que os arquivos temporários: o pipeline

```
eco olá mundo | wc
```

poderia ser implementado sem tubos como

```
eco olá mundo >/tmp/xyz; wc </tmp/xyz
```

Pipes têm pelo menos três vantagens sobre arquivos temporários nessa situação. Primeiro, pipes se autolimpam; com o redirecionamento de arquivos, um shell teria que ter o cuidado de remover /tmp/xyz ao terminar. Segundo, pipes podem passar fluxos de dados arbitrariamente longos, enquanto o redirecionamento de arquivos requer espaço livre suficiente em disco para armazenar todos os dados. Terceiro, pipes permitem a execução paralela de etapas do pipeline, enquanto a abordagem de arquivo exige que o primeiro programa termine antes do segundo iniciar.

1.4 Sistema de arquivos

O sistema de arquivos xv6 fornece arquivos de dados, que contêm matrizes de bytes não interpretadas, e diretórios, que contêm referências nomeadas a arquivos de dados e outros diretórios. Os diretórios formam uma árvore, começando em um diretório especial chamado raiz. Um caminho como /a/b/c refere-se ao arquivo ou diretório chamado c dentro do diretório chamado b dentro do diretório chamado a no diretório raiz /. Caminhos que não começam com / são avaliados em relação ao diretório atual do processo chamador, que pode ser alterado com a chamada de sistema chdir . Ambos os fragmentos de código abrem o mesmo arquivo (assumindo que todos os diretórios envolvidos existam):

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

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

O primeiro fragmento altera o diretório atual do processo para /a/b; o segundo não faz referência nem altera o diretório atual do processo.

Existem chamadas de sistema para criar novos arquivos e diretórios: mkdir cria um novo diretório, open com o sinalizador O_CREATE cria um novo arquivo de dados e mknod cria um novo arquivo de dispositivo. Este exemplo ilustra todos os três:

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

O mknod cria um arquivo especial que se refere a um dispositivo. Associados a um arquivo de dispositivo estão os números de dispositivo principal e secundário (os dois argumentos do mknod), que identificam exclusivamente um dispositivo do kernel. Quando um processo abre posteriormente um arquivo de dispositivo, o kernel desvia chamadas de leitura e gravação do sistema para a implementação do dispositivo do kernel em vez de passá-las para o sistema de arquivos.

O nome de um arquivo é diferente do próprio arquivo; o mesmo arquivo subjacente, chamado inode, pode ter vários nomes, chamados links. Cada link consiste em uma entrada em um diretório; a entrada contém um nome de arquivo e uma referência a um inode. Um inode contém metadados sobre um arquivo, incluindo seu tipo (arquivo, diretório ou dispositivo), seu comprimento, a localização do conteúdo do arquivo no disco e o número de links para um arquivo.

A chamada de sistema fstat recupera informações do inode ao qual um descritor de arquivo se refere.
preenche uma estrutura stat, definida em stat.h (kernel/stat.h) como:

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

estrutura stat {
  int dev;              // Dispositivo de disco do sistema de arquivos
  ino; short            // Número do inode uint
  type; // Tipo de arquivo short nlink; // Número
  de links para o arquivo uint64 size; // Tamanho do arquivo em
  bytes };
```

A chamada do sistema de link cria outro nome de sistema de arquivo que se refere ao mesmo inode que um existente arquivo ing. Este fragmento cria um novo arquivo chamado a e b.

```
abrir("a", O_CREATE|O_WRONLY); link("a", "b");
```

Ler ou escrever em a é o mesmo que ler ou escrever em b. Cada inode é identificado por um número de inode exclusivo. Após a sequência de código acima, é possível determinar que a e b se referem ao mesmo conteúdo subjacente inspecionando o resultado de fstat: ambos retornarão o mesmo número de inode (ino) e a contagem de nlinks será definida como 2.

A chamada de sistema unlink remove um nome do sistema de arquivos. O inode do arquivo e o espaço em disco que contém seu conteúdo só são liberados quando a contagem de links do arquivo é zero e nenhum descritor de arquivo faz referência a ele. Adicionando assim

```
desvincular("a");
```

para a última sequência de código deixa o inode e o conteúdo do arquivo acessíveis como b. Além disso,

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

é uma maneira idiomática de criar um inode temporário sem nome que será limpo quando o processo fechar o fd ou sair.

O Unix fornece utilitários de arquivo que podem ser chamados a partir do shell como programas de nível de usuário, por exemplo , mkdir, ln e rm. Esse design permite que qualquer pessoa estenda a interface de linha de comando adicionando novos programas de nível de usuário. Em retrospectiva, esse plano parece óbvio, mas outros sistemas projetados na época do Unix frequentemente incorporavam esses comandos ao shell (e o shell ao kernel).

Uma exceção é o cd, que é incorporado ao shell (user/sh.c:161). cd deve alterar o diretório de trabalho atual do próprio shell. Se cd fosse executado como um comando regular, o shell

bifurcar um processo filho, o processo filho executaria cd, e cd alteraria o diretório de trabalho do filho. O diretório de trabalho do pai (ou seja, do shell) não mudaria.

1.5 Mundo real

A combinação de descritores de arquivo "padrão", pipes e sintaxe shell conveniente do Unix para operações sobre eles foi um grande avanço na escrita de programas reutilizáveis de uso geral. A ideia deu início a uma cultura de "ferramentas de software" que foi responsável por grande parte do poder e popularidade do Unix, e o shell foi a primeira chamada "linguagem de script". A interface de chamada de sistema do Unix persiste até hoje em sistemas como BSD, Linux e macOS.

A interface de chamada de sistema do Unix foi padronizada pelo padrão POSIX (Portable Operating System Interface). O Xv6 não é compatível com POSIX: faltam muitas chamadas de sistema (incluindo as básicas, como lseek), e muitas das chamadas de sistema que ele fornece diferem do padrão. Nossos principais objetivos para o xv6 são simplicidade e clareza, ao mesmo tempo em que fornecemos uma interface de chamada de sistema simples, semelhante ao UNIX. Várias pessoas estenderam o xv6 com algumas chamadas de sistema a mais e uma biblioteca C simples para executar programas Unix básicos. Os kernels modernos, no entanto, fornecem muito mais chamadas de sistema e muitos mais tipos de serviços de kernel do que o xv6. Por exemplo, eles oferecem suporte a redes, sistemas de janelas, threads em nível de usuário, drivers para muitos dispositivos e assim por diante. Os kernels modernos evoluem contínua e rapidamente e oferecem muitos recursos além do POSIX.

O Unix unificou o acesso a múltiplos tipos de recursos (arquivos, diretórios e dispositivos) com um único conjunto de interfaces de nome e descritor de arquivo. Essa ideia pode ser estendida a mais tipos de recursos; um bom exemplo é o Plan 9 [16], que aplicou o conceito de “recursos são arquivos” a redes, gráficos e outros. No entanto, a maioria dos sistemas operacionais derivados do Unix não seguiu esse caminho.

O sistema de arquivos e os descritores de arquivos têm sido abstrações poderosas. Mesmo assim, existem outros modelos para interfaces de sistemas operacionais. O Multics, um predecessor do Unix, abstraiu o armazenamento de arquivos de uma forma que o fez parecer memória, produzindo uma interface muito diferente. A complexidade do design do Multics teve uma influência direta nos designers do Unix, que buscavam construir algo mais simples.

O Xv6 não fornece uma noção de usuários ou de proteção de um usuário de outro; em termos Unix, todos os processos xv6 são executados como root.

Este livro examina como o xv6 implementa sua interface semelhante à do Unix, mas as ideias e conceitos se aplicam a mais do que apenas o Unix. Qualquer sistema operacional deve multiplexar processos no hardware subjacente, isolar processos uns dos outros e fornecer mecanismos para comunicação controlada entre processos. Depois de estudar o xv6, você deverá ser capaz de analisar outros sistemas operacionais mais complexos e entender os conceitos subjacentes ao xv6 nesses sistemas também.

1.6 Exercícios

1. Escreva um programa que utilize chamadas de sistema UNIX para "pingar" um byte entre dois processos através de um par de pipes, um para cada direção. Meça o desempenho do programa em trocas por segundo.

Capítulo 2

Organização do sistema operacional

Um requisito fundamental para um sistema operacional é suportar diversas atividades simultaneamente. Por exemplo, usando a interface de chamada de sistema descrita no Capítulo 1, um processo pode iniciar novos processos com fork. O sistema operacional deve compartilhar o tempo dos recursos do computador entre esses processos. Por exemplo, mesmo que haja mais processos do que CPUs de hardware, o sistema operacional deve garantir que todos os processos tenham a chance de executar. O sistema operacional também deve providenciar o isolamento entre os processos. Ou seja, se um processo tiver um bug e apresentar mau funcionamento, isso não deve afetar os processos que não dependem do processo com bug. O isolamento completo, no entanto, é muito forte, pois deve ser possível que os processos interajam intencionalmente; pipelines são um exemplo. Portanto, um sistema operacional deve atender a três requisitos: multiplexação, isolamento e interação.

Este capítulo fornece uma visão geral de como os sistemas operacionais são organizados para atender a esses três requisitos. Existem muitas maneiras de fazer isso, mas este texto se concentra em projetos tradicionais centrados em um kernel monolítico, usado por muitos sistemas operacionais Unix. Este capítulo também fornece uma visão geral de um processo xv6, que é a unidade de isolamento no xv6, e da criação do primeiro processo quando o xv6 é iniciado.

O Xv6 é executado em um microprocessador RISC-V multi-core1 , e grande parte de sua funcionalidade de baixo nível (por exemplo, sua implementação de processo) é específica para RISC-V. RISC-V é uma CPU de 64 bits, e o xv6 é escrito em C "LP64", o que significa que long (L) e ponteiros (P) na linguagem de programação C são de 64 bits, mas int é de 32 bits. Este livro pressupõe que o leitor tenha feito um pouco de programação em nível de máquina em alguma arquitetura, e apresentará ideias específicas do RISC-V à medida que surgirem. Uma referência útil para RISC-V é "The RISC-V Reader: An Open Architecture Atlas" [15]. O ISA em nível de usuário [2] e a arquitetura privilegiada [3] são as especificações oficiais.

A CPU de um computador completo é cercada por hardware de suporte, grande parte dele na forma de interfaces de E/S. O Xv6 foi escrito para o hardware de suporte simulado pela opção "-machine virt" do qemu. Isso inclui RAM, uma ROM contendo o código de inicialização, uma conexão serial com o teclado/tela do usuário e um disco para armazenamento.

1Por "multinúcleo", este texto se refere a múltiplas CPUs que compartilham memória, mas executam em paralelo, cada uma com seu próprio conjunto de registradores. Este texto às vezes usa o termo multiprocessador como sinônimo de multinúcleo, embora multiprocessador também possa se referir mais especificamente a um computador com vários chips de processador distintos.

2.1 Abstraindo recursos físicos

A primeira pergunta que se pode fazer ao se deparar com um sistema operacional é: por que tê-lo? Ou seja, seria possível implementar as chamadas de sistema da Figura 1.2 como uma biblioteca, com a qual os aplicativos se conectam. Nesse plano, cada aplicativo poderia até ter sua própria biblioteca, adaptada às suas necessidades. Os aplicativos poderiam interagir diretamente com os recursos de hardware e utilizá-los da melhor maneira possível para o aplicativo (por exemplo, para atingir um desempenho alto ou previsível). Alguns sistemas operacionais para dispositivos embarcados ou sistemas de tempo real são organizados dessa maneira.

A desvantagem dessa abordagem de biblioteca é que, se houver mais de um aplicativo em execução, eles precisam se comportar bem. Por exemplo, cada aplicativo precisa liberar a CPU periodicamente para que outros aplicativos possam ser executados. Esse esquema cooperativo de compartilhamento de tempo pode ser aceitável se todos os aplicativos confiarem uns nos outros e não apresentarem bugs. É mais comum que aplicativos não confiem uns nos outros e apresentem bugs, portanto, muitas vezes, deseja-se um isolamento mais forte do que o proporcionado por um esquema cooperativo.

Para obter um isolamento forte, é útil proibir que aplicativos acessem diretamente recursos de hardware sensíveis e, em vez disso, abstrair os recursos em serviços. Por exemplo, aplicativos Unix interagem com o armazenamento apenas por meio das chamadas de sistema de abertura, leitura, gravação e fechamento do sistema de arquivos, em vez de ler e gravar diretamente no disco. Isso proporciona ao aplicativo a conveniência de nomes de caminho e permite que o sistema operacional (como implementador da interface) gerencie o disco. Mesmo que o isolamento não seja uma preocupação, programas que interagem intencionalmente (ou apenas desejam se manter fora do caminho uns dos outros) provavelmente considerarão um sistema de arquivos uma abstração mais conveniente do que o uso direto do disco.

Da mesma forma, o Unix alterna de forma transparente as CPUs de hardware entre os processos, salvando e restaurando o estado dos registradores conforme necessário, para que os aplicativos não precisem se preocupar com o compartilhamento de tempo. Essa transparência permite que o sistema operacional compartilhe CPUs mesmo que alguns aplicativos estejam em loops infinitos.

Como outro exemplo, os processos Unix usam o comando exec para construir sua imagem de memória, em vez de interagir diretamente com a memória física. Isso permite que o sistema operacional decida onde colocar um processo na memória; se a memória estiver limitada, o sistema operacional pode até armazenar alguns dados de um processo em disco. O comando exec também oferece aos usuários a conveniência de um sistema de arquivos para armazenar imagens de programas executáveis.

Muitas formas de interação entre processos Unix ocorrem por meio de descritores de arquivo. Os descritores de arquivo não apenas abstraem muitos detalhes (por exemplo, onde os dados em um pipe ou arquivo são armazenados), como também são definidos de forma a simplificar a interação. Por exemplo, se uma aplicação em um pipeline falhar, o kernel gera um sinal de fim de arquivo para o próximo processo no pipeline.

A interface de chamada de sistema na Figura 1.2 foi cuidadosamente projetada para proporcionar conveniência ao programador e a possibilidade de forte isolamento. A interface Unix não é a única maneira de abstrair recursos, mas provou ser uma ótima maneira.

2.2 Modo de usuário, modo supervisor e chamadas de sistema

O isolamento forte requer uma fronteira rígida entre os aplicativos e o sistema operacional. Se o aplicativo cometer um erro, não queremos que o sistema operacional falhe ou que outros aplicativos sejam afetados.

falha. Em vez disso, o sistema operacional deve ser capaz de limpar o aplicativo com falha e continuar executando outros aplicativos. Para alcançar um isolamento forte, o sistema operacional deve garantir que os aplicativos não possam modificar (ou mesmo ler) suas estruturas de dados e instruções, e que os aplicativos não possam acessar a memória de outros processos.

As CPUs fornecem suporte de hardware para isolamento robusto. Por exemplo, o RISC-V possui três modos nos quais a CPU pode executar instruções: modo máquina, modo supervisor e modo usuário. Instruções de operação executadas em modo máquina têm privilégio total; uma CPU inicia em modo máquina. O modo máquina destina-se principalmente à configuração de um computador. O Xv6 executa algumas linhas em modo máquina e depois muda para o modo supervisor.

No modo supervisor, a CPU tem permissão para executar instruções privilegiadas: por exemplo, habilitar e desabilitar interrupções, ler e escrever no registrador que contém o endereço de uma tabela de páginas, etc. Se um aplicativo no modo usuário tentar executar uma instrução privilegiada, a CPU não executa a instrução, mas alterna para o modo supervisor para que o código do modo supervisor possa encerrar o aplicativo, porque ele fez algo que não deveria. A Figura 1.1 no Capítulo 1 ilustra essa organização. Um aplicativo pode executar apenas instruções do modo usuário (por exemplo, somar números, etc.) e é dito estar em execução no espaço do usuário, enquanto o software no modo supervisor também pode executar instruções privilegiadas e é dito estar em execução no espaço do kernel. O software em execução no espaço do kernel (ou no modo supervisor) é chamado de kernel.

Um aplicativo que deseja invocar uma função do kernel (por exemplo, a chamada de sistema read no xv6) deve fazer a transição para o kernel; um aplicativo não pode invocar uma função do kernel diretamente. As CPUs fornecem uma instrução especial que alterna a CPU do modo usuário para o modo supervisor e entra no kernel em um ponto de entrada especificado pelo kernel. (O RISC-V fornece a instrução ecall para esse propósito.)
Após a CPU alternar para o modo supervisor, o kernel pode validar os argumentos da chamada de sistema (por exemplo, verificar se o endereço passado para a chamada de sistema faz parte da memória do aplicativo), decidir se o aplicativo tem permissão para executar a operação solicitada (por exemplo, verificar se o aplicativo tem permissão para gravar o arquivo especificado) e, em seguida, negá-la ou executá-la. É importante que o kernel controle o ponto de entrada para transições para o modo supervisor; se o aplicativo pudesse decidir o ponto de entrada do kernel, um aplicativo malicioso poderia, por exemplo, entrar no kernel em um ponto onde a validação dos argumentos é ignorada.

2.3 Organização do kernel

Uma questão fundamental de projeto é qual parte do sistema operacional deve ser executada no modo supervisor. Uma possibilidade é que todo o sistema operacional resida no kernel, de modo que as implementações de todas as chamadas de sistema sejam executadas no modo supervisor. Essa organização é chamada de kernel monolítico.

Nesta organização, todo o sistema operacional é executado com privilégios totais de hardware. Essa organização é conveniente porque o projetista do sistema operacional não precisa decidir qual parte do sistema operacional não precisa de privilégios totais de hardware. Além disso, facilita a cooperação entre diferentes partes do sistema operacional. Por exemplo, um sistema operacional pode ter um cache de buffer que pode ser compartilhado tanto pelo sistema de arquivos quanto pelo sistema de memória virtual.

Uma desvantagem da organização monolítica é que as interfaces entre as diferentes partes do sistema operacional são frequentemente complexas (como veremos no restante deste texto) e, portanto, é

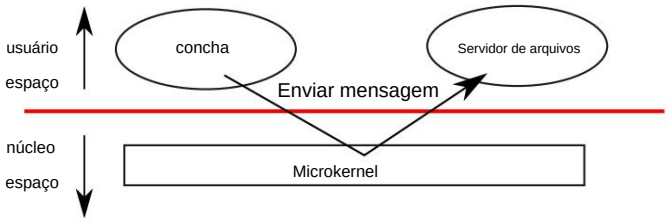

Figura 2.1: Um microkernel com um servidor de sistema de arquivos

É fácil para um desenvolvedor de sistema operacional cometer um erro. Em um kernel monolítico, um erro é fatal, pois um erro no modo supervisor frequentemente causa falha no kernel. Se o kernel falhar, o computador para de funcionar e, portanto, todos os aplicativos também falham. O computador precisa ser reiniciado para iniciar novamente.

Para reduzir o risco de erros no kernel, os projetistas de sistemas operacionais podem minimizar a quantidade de código do sistema operacional executado no modo supervisor e executar a maior parte do sistema operacional no modo usuário. Essa organização do kernel é chamada de microkernel.

A Figura 2.1 ilustra este projeto de microkernel. Na figura, o sistema de arquivos é executado como um processo em nível de usuário. Os serviços do sistema operacional executados como processos são chamados de servidores. Para permitir que os aplicativos interajam com o servidor de arquivos, o kernel fornece um mecanismo de comunicação entre processos para enviar mensagens de um processo em modo de usuário para outro. Por exemplo, se um aplicativo, como o shell, deseja ler ou gravar um arquivo, ele envia uma mensagem ao servidor de arquivos e aguarda uma resposta.

Em um microkernel, a interface do kernel consiste em algumas funções de baixo nível para iniciar aplicativos, enviar mensagens, acessar hardware do dispositivo, etc. Essa organização permite que o kernel seja relativamente simples, já que a maior parte do sistema operacional reside em servidores de nível de usuário.

No mundo real, tanto kernels monolíticos quanto microkernels são populares. Muitos kernels Unix são monolíticos. Por exemplo, o Linux possui um kernel monolítico, embora algumas funções do sistema operacional sejam executadas como servidores em nível de usuário (por exemplo, o sistema de janelas). O Linux oferece alto desempenho para aplicativos com uso intensivo de sistema operacional, em parte porque os subsistemas do kernel podem ser totalmente integrados.

Sistemas operacionais como Minix, L4 e QNX são organizados como um microkernel com servidores e têm sido amplamente implantados em ambientes embarcados. Uma variante do L4, seL4, é pequena o suficiente para ter sua segurança de memória e outras propriedades de segurança verificadas [8].

Há muito debate entre desenvolvedores de sistemas operacionais sobre qual organização é melhor, e não há evidências conclusivas de um lado ou de outro. Além disso, depende muito do que "melhor" significa: desempenho mais rápido, código menor, confiabilidade do kernel, confiabilidade de todo o sistema operacional (incluindo serviços em nível de usuário), etc.

Há também considerações práticas que podem ser mais importantes do que a questão de qual organização. Alguns sistemas operacionais possuem um microkernel, mas executam alguns dos serviços de nível de usuário no espaço do kernel por questões de desempenho. Alguns sistemas operacionais possuem kernels monolíticos porque foi assim que começaram, e há pouco incentivo para migrar para uma organização puramente de microkernel, pois novos recursos podem ser mais importantes do que reescrever o sistema operacional existente para se adequar a um design de microkernel.

Da perspectiva deste livro, microkernel e sistemas operacionais monolíticos compartilham muitas ideias-chave. Eles implementam chamadas de sistema, usam tabelas de páginas, lidam com interrupções e oferecem suporte

Arquivo	Descrição
bio.c	Cache de bloco de disco para o sistema de arquivos.
console.c	Conecte-se ao teclado e à tela do usuário.
entrada.S	Primeiras instruções de inicialização.
exec.c	chamada de sistema exec().
arquivo.c fs.c	Suporte ao descritor de arquivo.
kalloc.c	Sistema de arquivos.
kernelvec.S	Alocador de páginas físicas.
log.c	Lida com armadilhas do kernel e interrupções de temporizador.
main.c	Registro do sistema de arquivos e recuperação de falhas.
pipe.c	Controla a inicialização de outros módulos durante a inicialização.
plic.c	Tubos.
printf.c	Controlador de interrupção RISC-V.
proc.c	Saída formatada para o console.
sleeplock.c	Processos e agendamento.
spinlock.c	Bloqueios que produzem a CPU.
start.c	Bloqueios que não cedem a CPU.
string.c	Código de inicialização inicial do modo máquina.
swtch.S	Biblioteca de strings e matrizes de bytes em C.
syscall.c	Troca de thread.
sysfile.c	Despachar chamadas do sistema para a função de manuseio.
sysproc.c	Chamadas de sistema relacionadas a arquivos.
trampoline.S	Chamadas de sistema relacionadas ao processo.
trap.c	Código de montagem para alternar entre usuário e kernel.
uart.c	Código C para manipular e retornar armadilhas e interrupções.
virtio_disk.c	Driver de dispositivo de console de porta serial.
vm.c	Driver de dispositivo de disco.
	Gerenciar tabelas de páginas e espaços de endereço.

Figura 2.2: Arquivos de origem do kernel Xv6.

processos, eles usam bloqueios para controle de simultaneidade, eles implementam um sistema de arquivos, etc. Este livro concentra-se nessas ideias principais.

O Xv6 é implementado como um kernel monolítico, como a maioria dos sistemas operacionais Unix. Assim, o xv6 A interface do kernel corresponde à interface do sistema operacional, e o kernel implementa o sistema operacional completo. Como o xv6 não fornece muitos serviços, seu kernel é menor do que alguns microkernels, mas conceitualmente o xv6 é monolítico.

2.4 Código: organização xv6

O código-fonte do kernel xv6 está no subdiretório kernel/. O código-fonte é dividido em arquivos, a seguir uma noção aproximada de modularidade; a Figura 2.2 lista os arquivos. As interfaces entre módulos são definidas em

Figura 2.3: Layout do espaço de endereço virtual de um processo

defs.h (kernel/defs.h).

2.5 Visão geral do processo

A unidade de isolamento no xv6 (assim como em outros sistemas operacionais Unix) é um processo. A abstração de processo impede que um processo destrua ou espione a memória, a CPU, os descritores de arquivo, etc. de outro processo. Também impede que um processo destrua o próprio kernel, impedindo que um processo subverta os mecanismos de isolamento do kernel. O kernel deve implementar a abstração de processo com cuidado, pois um aplicativo malicioso ou com bugs pode induzir o kernel ou o hardware a fazer algo errado (por exemplo, burlar o isolamento). Os mecanismos usados pelo kernel para implementar processos incluem o sinalizador de modo usuário/supervisor, espaços de endereço e fatiamento de tempo de threads.

Para ajudar a reforçar o isolamento, a abstração do processo dá a um programa a ilusão de que ele possui sua própria máquina privada. Um processo fornece ao programa o que parece ser um sistema de memória privado, ou espaço de endereço, que outros processos não conseguem ler ou escrever. Um processo também fornece ao programa o que parece ser sua própria CPU para executar as instruções do programa.

O Xv6 utiliza tabelas de páginas (implementadas por hardware) para dar a cada processo seu próprio espaço de endereço. A tabela de páginas RISC-V traduz (ou "mapeia") um endereço virtual (o endereço que uma instrução RISC-V manipula) para um endereço físico (um endereço que o chip da CPU envia para a memória principal).

O Xv6 mantém uma tabela de páginas separada para cada processo que define o espaço de endereço desse processo. Conforme ilustrado na Figura 2.3, um espaço de endereço inclui a memória do usuário do processo, começando no endereço virtual zero. As instruções vêm primeiro, seguidas pelas variáveis globais, depois pela pilha e, finalmente, por uma área de "heap" (para malloc) que o processo pode expandir conforme necessário. Há uma série de fatores que limitam o tamanho máximo do espaço de endereço de um processo: ponteiros no RISC-V têm 64 bits de largura; o hardware usa apenas os 39 bits mais baixos ao procurar endereços virtuais em tabelas de páginas; e o xv6 usa apenas 38 desses 39 bits. Portanto, o endereço máximo é 2 ÿ 1 = 0x3fffffffff, que é

MAXVA (kernel/riscv.h:363). No topo do espaço de endereço, o xv6 reserva uma página para um trampolim e uma página que mapeia o trapframe do processo. O xv6 usa essas duas páginas para fazer a transição de entrada e saída do kernel; a página do trampolim contém o código para a transição de entrada e saída do kernel, e o mapeamento do trapframe é necessário para salvar/restaurar o estado do processo do usuário, como explicaremos no Capítulo 4.

O kernel xv6 mantém muitas partes de estado para cada processo, que ele reúne em uma estrutura proc (kernel/proc.h:85). As partes mais importantes do estado do kernel de um processo são sua tabela de páginas, sua pilha do kernel e seu estado de execução. Usaremos a notação p->xxx para nos referirmos aos elementos da estrutura proc ; por exemplo, p->pagetable é um ponteiro para a tabela de páginas do processo.

Cada processo possui uma thread de execução (ou thread, para abreviar) que executa as instruções do processo. Uma thread pode ser suspensa e posteriormente retomada. Para alternar entre processos de forma transparente, o kernel suspende a thread em execução e retoma a thread de outro processo. Grande parte do estado de uma thread (variáveis locais, endereços de retorno de chamadas de função) é armazenada nas pilhas da thread.
Cada processo possui duas pilhas: uma pilha do usuário e uma pilha do kernel (p->kstack). Quando o processo está executando instruções do usuário, apenas sua pilha do usuário está em uso, e sua pilha do kernel está vazia. Quando o processo entra no kernel (para uma chamada de sistema ou interrupção), o código do kernel é executado na pilha do kernel do processo; enquanto um processo está no kernel, sua pilha do usuário ainda contém dados salvos, mas não é usada ativamente. A thread de um processo alterna entre usar ativamente sua pilha do usuário e sua pilha do kernel.
A pilha do kernel é separada (e protegida do código do usuário) para que o kernel possa ser executado mesmo que um processo tenha destruído sua pilha do usuário.

Um processo pode fazer uma chamada de sistema executando a instrução ecall do RISC-V . Essa instrução aumenta o nível de privilégio de hardware e altera o contador de programa para um ponto de entrada definido pelo kernel.
O código no ponto de entrada alterna para uma pilha do kernel e executa as instruções do kernel que implementam a chamada do sistema. Quando a chamada do sistema é concluída, o kernel retorna para a pilha do usuário e retorna ao espaço do usuário chamando a instrução sret , que reduz o nível de privilégio de hardware e retoma a execução das instruções do usuário logo após a instrução de chamada do sistema. A thread de um processo pode "bloquear" no kernel para aguardar a E/S e retomar de onde parou quando a E/S terminar.

p->state indica se o processo está alocado, pronto para execução, em execução, aguardando E/S ou encerrando. p-
>pagetable contém a tabela de páginas do processo, no formato esperado pelo hardware RISC-V. O Xv6 faz com que o hardware de paginação utilize a p->pagetable de um processo ao executá-lo no espaço do usuário. A tabela de páginas de um processo também serve como registro dos endereços das páginas físicas alocadas para armazenar a memória do processo.

Em resumo, um processo agrupa duas ideias de design: um espaço de endereço para dar ao processo a ilusão de ter sua própria memória e uma thread para dar ao processo a ilusão de ter sua própria CPU. No xv6, um processo consiste em um espaço de endereço e uma thread. Em sistemas operacionais reais, um processo pode ter mais de uma thread para aproveitar múltiplas CPUs.

2.6 Código: iniciando o xv6, o primeiro processo e chamada do sistema

Para tornar o xv6 mais concreto, descreveremos como o kernel inicia e executa o primeiro processo. Os capítulos subsequentes descreverão os mecanismos que aparecem nesta visão geral com mais detalhes.

Quando o computador RISC-V é ligado, ele se inicializa e executa um carregador de boot armazenado na memória somente leitura. O carregador de boot carrega o kernel xv6 na memória. Em seguida, no modo máquina, a CPU executa o xv6 a partir de _entry (kernel/entry.S:7). O RISC-V começa com o hardware de paginação desabilitado: endereços virtuais são mapeados diretamente para endereços físicos.

O carregador carrega o kernel xv6 na memória no endereço físico 0x80000000. O motivo pelo qual ele coloca o kernel em 0x80000000 em vez de 0x0 é porque o intervalo de endereços 0x0:0x80000000 contém dispositivos de E/S.

As instruções em _entry configuram uma pilha para que o xv6 possa executar código C. O xv6 declara espaço para uma pilha inicial, stack0, no arquivo start.c (kernel/start.c:11). O código em _entry carrega o registrador de ponteiro de pilha sp com o endereço stack0+4096, o topo da pilha, porque a pilha no RISC-V cresce para baixo. Agora que o kernel tem uma pilha, _entry chama o código C no início (kernel/start.c:21).

A função start realiza algumas configurações permitidas apenas no modo máquina e, em seguida, alterna para o modo supervisor. Para entrar no modo supervisor, o RISC-V fornece a instrução mret. Essa instrução é mais frequentemente usada para retornar de uma chamada anterior do modo supervisor para o modo máquina. start não retorna de tal chamada e, em vez disso, configura tudo como se houvesse uma: define o modo de privilégio anterior como supervisor no registrador mstatus, define o endereço de retorno como main escrevendo o endereço de main no registrador mepc, desabilita a tradução de endereços virtuais no modo supervisor escrevendo 0 no registrador de tabela de páginas satp e delega todas as interrupções e exceções ao modo supervisor.

Antes de entrar no modo supervisor, start executa mais uma tarefa: programa o chip do relógio para gerar interrupções de tempo. Com essa tarefa concluída, start "retorna" ao modo supervisor chamando mret. Isso faz com que o contador de programa mude para o modo principal (kernel/main.c:11).

Após o principal (kernel/main.c:11) inicializa vários dispositivos e subsistemas, ele cria o primeiro processo chamando userinit (kernel/proc.c:233). O primeiro processo executa um pequeno programa escrito em assembly RISC-V, que faz a primeira chamada de sistema no xv6. initcode.S (user/initcode.S:3) carrega o número para a chamada do sistema exec , SYS_EXEC (kernel/syscall.h:8), no registro a7 e, em seguida, chama ecall para entrar novamente no kernel.

O kernel usa o número no registro a7 em syscall (kernel/syscall.c:132) para chamar a chamada de sistema desejada. A tabela de chamadas de sistema (kernel/syscall.c:107) mapeia SYS_EXEC para sys_exec, que o kernel invoca. Como vimos no Capítulo 1, exec substitui a memória e os registradores do processo atual por um novo programa (neste caso, /init).

Depois que o kernel conclui o exec, ele retorna ao espaço do usuário no processo /init . init (user/init.c:15) Cria um novo arquivo de dispositivo de console, se necessário, e o abre como descritores de arquivo 0, 1 e 2. Em seguida, inicia um shell no console. O sistema está funcionando.

2.7 Modelo de Segurança

Você pode se perguntar como o sistema operacional lida com códigos maliciosos ou com bugs. Como lidar com códigos maliciosos é mais difícil do que lidar com bugs acidentais, é razoável considerar este tópico como relacionado à segurança. Aqui está uma visão geral das premissas e objetivos típicos de segurança no design de sistemas operacionais.

O sistema operacional deve assumir que o código de nível de usuário de um processo fará o possível para destruir o kernel ou outros processos. O código do usuário pode tentar desreferenciar ponteiros fora de seu espaço de endereço permitido; pode tentar executar quaisquer instruções RISC-V, mesmo aquelas não destinadas ao código do usuário; pode tentar ler e escrever qualquer registrador de controle RISC-V; pode tentar acessar diretamente o hardware do dispositivo; e pode passar valores inteligentes para chamadas de sistema na tentativa de enganar o kernel para travar ou fazer algo estúpido. O objetivo do kernel é restringir cada processo do usuário para que tudo o que ele possa fazer seja ler/escrever/executar sua própria memória de usuário, usar os 32 registradores RISC-V de uso geral e afetar o kernel e outros processos das maneiras que as chamadas de sistema devem permitir. O kernel deve impedir quaisquer outras ações. Isso normalmente é um requisito absoluto no projeto do kernel.

As expectativas para o código do kernel são bem diferentes. O código do kernel é assumido como escrito por programadores bem-intencionados e cuidadosos. Espera-se que o código do kernel esteja livre de bugs e certamente não contenha nada malicioso. Essa suposição afeta a forma como analisamos o código do kernel. Por exemplo, existem muitas funções internas do kernel (por exemplo, os bloqueios de spin) que causariam sérios problemas se o código do kernel as utilizasse incorretamente. Ao examinar qualquer trecho específico do código do kernel, queremos nos convencer de que ele se comporta corretamente. Assumimos, no entanto, que o código do kernel em geral está escrito corretamente e segue todas as regras sobre o uso das funções e estruturas de dados do kernel. No nível de hardware, presume-se que a CPU, a RAM, o disco etc. do RISC-V operem conforme anunciado na documentação, sem bugs de hardware.

É claro que na vida real as coisas não são tão simples. É difícil evitar que um código de usuário inteligente torne um sistema inutilizável (ou o faça entrar em pânico) consumindo recursos protegidos pelo kernel – espaço em disco, tempo de CPU, slots na tabela de processos, etc. Geralmente é impossível escrever código livre de bugs ou projetar hardware livre de bugs; se os escritores de código de usuário malicioso estiverem cientes de bugs no kernel ou no hardware, eles os explorarão. Mesmo em kernels maduros e amplamente utilizados, como o Linux, as pessoas descobrem novas vulnerabilidades continuamente [1]. Vale a pena projetar salvaguardas no kernel contra a possibilidade de ele ter bugs: asserções, verificação de tipos, páginas de proteção de pilha, etc. Finalmente, a distinção entre código de usuário e kernel às vezes é confusa: alguns processos privilegiados em nível de usuário podem fornecer serviços essenciais e efetivamente fazer parte do sistema operacional e, em alguns sistemas operacionais, o código de usuário privilegiado pode inserir novo código no kernel (como com os módulos carregáveis do kernel do Linux).

2.8 Mundo real

A maioria dos sistemas operacionais adotou o conceito de processo, e a maioria dos processos se assemelha aos do xv6. Os sistemas operacionais modernos, no entanto, suportam várias threads dentro de um processo, permitindo que um único processo explore múltiplas CPUs. Suportar múltiplas threads em um processo envolve uma série de mecanismos que o xv6 não possui, incluindo possíveis mudanças na interface (por exemplo, o clone do Linux, uma variante do fork) para controlar quais aspectos de um processo as threads compartilham.

2.9 Exercícios

1. Adicione uma chamada de sistema ao xv6 que retorne a quantidade de memória livre disponível.

Capítulo 3

Tabelas de páginas

Tabelas de páginas são o mecanismo mais popular pelo qual o sistema operacional fornece a cada processo seu próprio espaço de endereço privado e memória. Tabelas de páginas determinam o significado dos endereços de memória e quais partes da memória física podem ser acessadas. Elas permitem que o xv6 isole os espaços de endereço de diferentes processos e os multiplexe em uma única memória física. Tabelas de páginas são um design popular porque fornecem um nível de indireção que permite aos sistemas operacionais realizarem muitos truques. O xv6 realiza alguns truques: mapeia a mesma memória (uma página de trampolim) em vários espaços de endereço e protege as pilhas do kernel e do usuário com uma página não mapeada. O restante deste capítulo explica as tabelas de páginas que o hardware RISC-V fornece e como o xv6 as utiliza.

3.1 Hardware de paginação

Vale lembrar que as instruções RISC-V (tanto de usuário quanto de kernel) manipulam endereços virtuais. A RAM da máquina, ou memória física, é indexada com endereços físicos. O hardware da tabela de páginas RISC-V conecta esses dois tipos de endereços, mapeando cada endereço virtual a um endereço físico.

O Xv6 roda em Sv39 RISC-V, o que significa que apenas os 39 bits inferiores de um endereço virtual de 64 bits são usados; os 25 bits superiores não são usados. Nesta configuração Sv39, uma tabela de páginas RISC-V é logicamente uma matriz de 2^{27} (134.217.728) entradas de tabela de páginas (PTEs). Cada PTE contém um número de página física (PPN) de 44 bits e alguns sinalizadores. O hardware de paginação traduz um endereço virtual usando os 27 bits superiores dos 39 bits para indexar na tabela de páginas para encontrar um PTE, criando um endereço físico de 56 bits cujos 44 bits superiores vêm do PPN no PTE e cujos 12 bits inferiores são copiados do endereço virtual original. A Figura 3.1 mostra esse processo com uma visão lógica da tabela de páginas como uma matriz simples de PTEs (veja a Figura 3.2 para uma história mais completa). Uma tabela de páginas fornece ao sistema operacional controle sobre as traduções de endereços virtuais para físicos na granularidade de blocos alinhados de 4096 (2^{12}) bytes. Esse bloco é chamado de página.

No Sv39 RISC-V, os 25 bits superiores de um endereço virtual não são usados para tradução. O endereço físico também tem espaço para crescimento: no formato PTE, há espaço para que o número de páginas físicas aumente em mais 10 bits. Os projetistas do RISC-V escolheram esses números com base em previsões tecnológicas. 2 bytes equivalem a 512 GB, o que 2^{39} deve ser espaço de endereço suficiente para aplicativos que executam

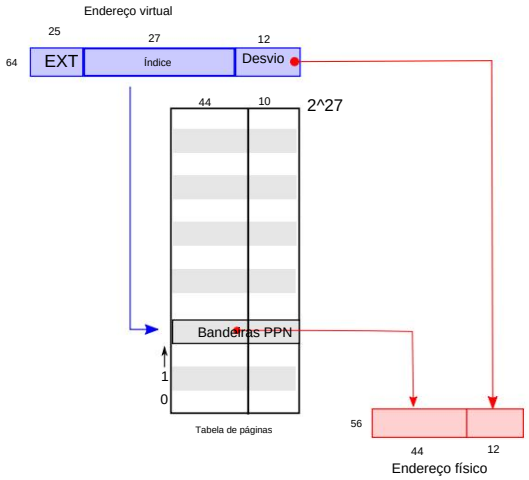

Figura 3.1: Endereços virtuais e físicos RISC-V, com uma tabela de páginas lógicas simplificada.

em computadores RISC-V. 2 é espaço de memória física suficiente para, em um futuro próximo, acomodar muitos dispositivos de E/S e chips DRAM. Se mais for necessário, os projetistas do RISC-V definiram Sv48 com endereços virtuais de 48 bits [3].

Como mostra a Figura 3.2, uma CPU RISC-V traduz um endereço virtual em um físico em três etapas.
Uma tabela de páginas é armazenada na memória física como uma árvore de três níveis. A raiz da árvore é uma página de tabela de páginas de 4.096 bytes que contém 512 PTEs, que contêm os endereços físicos das páginas da tabela de páginas no próximo nível da árvore. Cada uma dessas páginas contém 512 PTEs para o nível final da árvore.
O hardware de paginação usa os 9 bits superiores dos 27 bits para selecionar uma PTE na página da tabela de páginas raiz, os 9 bits do meio para selecionar uma PTE em uma página da tabela de páginas no próximo nível da árvore e os 9 bits inferiores para selecionar a PTE final. (No Sv48 RISC-V, uma tabela de páginas tem quatro níveis, e os bits 39 a 47 de um índice de endereço virtual no nível superior.)

Se qualquer um dos três PTEs necessários para traduzir um endereço não estiver presente, o hardware de paginação gera uma exceção de falha de página, deixando para o kernel lidar com a exceção (veja Capítulo 4).

A estrutura de três níveis da Figura 3.2 permite uma maneira eficiente de gravar PTEs em termos de memória, em comparação com o design de nível único da Figura 3.1. No caso comum em que grandes intervalos de endereços virtuais não têm mapeamentos, a estrutura de três níveis pode omitir diretórios de páginas inteiros. Por exemplo, se um aplicativo usa apenas algumas páginas começando no endereço zero, as entradas de 1 a 511 do diretório de páginas de nível superior são inválidas e o kernel não precisa alocar páginas para os 511 diretórios de páginas intermediárias. Além disso, o kernel também não precisa alocar páginas para os diretórios de páginas de nível inferior para esses 511 diretórios de páginas intermediárias. Portanto, neste exemplo, o design de três níveis economiza 511 páginas para diretórios de páginas intermediárias e 511 × 512 páginas para diretórios de páginas de nível inferior.

Embora uma CPU percorra a estrutura de três níveis no hardware como parte da execução de uma instrução de carregamento ou armazenamento, uma desvantagem potencial dos três níveis é que a CPU deve carregar três PTEs da memória para executar a tradução do endereço virtual na instrução de carregamento/armazenamento para um endereço físico. Para evitar o custo de carregamento de PTEs da memória física, uma CPU RISC-V armazena em cache as entradas da tabela de páginas em um Translation Look-aside Buffer (TLB).

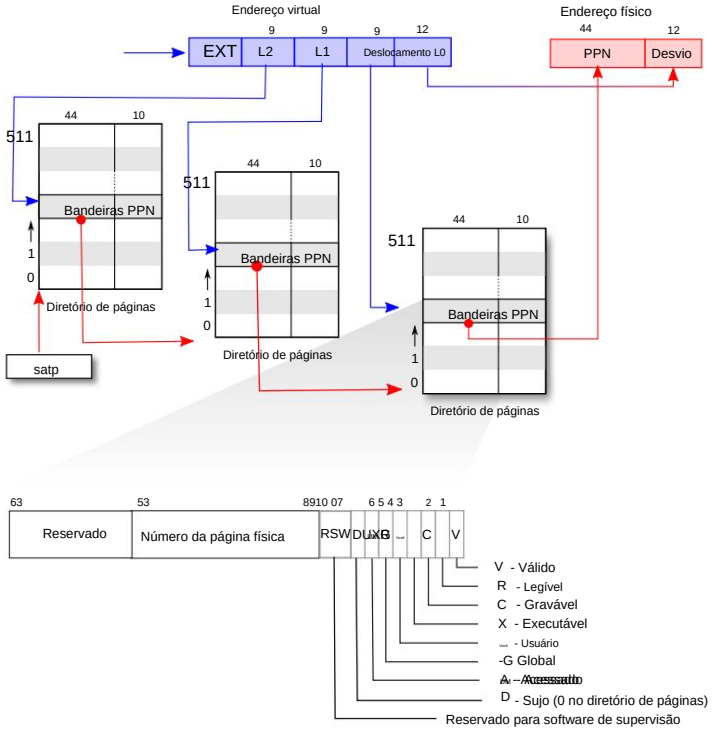

Figura 3.2: Detalhes da tradução de endereços RISC-V.

Cada PTE contém bits de sinalização que informam ao hardware de paginação como o endereço virtual associado pode ser usado. PTE_V indica se a PTE está presente: se não estiver definida, uma referência à página causa uma exceção (ou seja, não é permitida). PTE_R controla se as instruções podem ler na página. PTE_W controla se as instruções podem gravar na página. PTE_X controla se a CPU pode interpretar o conteúdo da página como instruções e executá-las.
PTE_U controla se instruções em modo usuário têm permissão para acessar a página; se PTE_U não estiver definido, o PTE só poderá ser usado em modo supervisor. A Figura 3.2 mostra como tudo funciona. Os sinalizadores e todas as outras estruturas relacionadas ao hardware da página são definidos em (kernel/riscv.h).

Para instruir uma CPU a usar uma tabela de páginas, o kernel precisa gravar o endereço físico da página raiz da tabela de páginas no registrador satp. Uma CPU traduzirá todos os endereços gerados por instruções subsequentes usando a tabela de páginas apontada por seu próprio satp. Cada CPU possui seu próprio satp, permitindo que diferentes CPUs executem processos diferentes, cada um com um espaço de endereço privado descrito por sua própria tabela de páginas.

Normalmente, um kernel mapeia toda a memória física em sua tabela de páginas para que possa ler e escrever em qualquer local da memória física usando instruções de carregamento/armazenamento. Como os diretórios de páginas estão na memória física, o kernel pode programar o conteúdo de uma PTE em um diretório de páginas escrevendo no endereço virtual da PTE usando uma instrução de armazenamento padrão.

Algumas notas sobre termos. Memória física refere-se às células de armazenamento na DRAM. Um byte de memória física possui um endereço, chamado endereço físico. As instruções usam apenas endereços virtuais, que o hardware de paginação traduz em endereços físicos e, em seguida, envia ao hardware DRAM para leitura.

ou armazenamento de gravação. Ao contrário da memória física e dos endereços virtuais, a memória virtual não é um espaço físico objeto, mas se refere à coleção de abstrações e mecanismos que o kernel fornece para gerenciar
memória física e endereços virtuais.

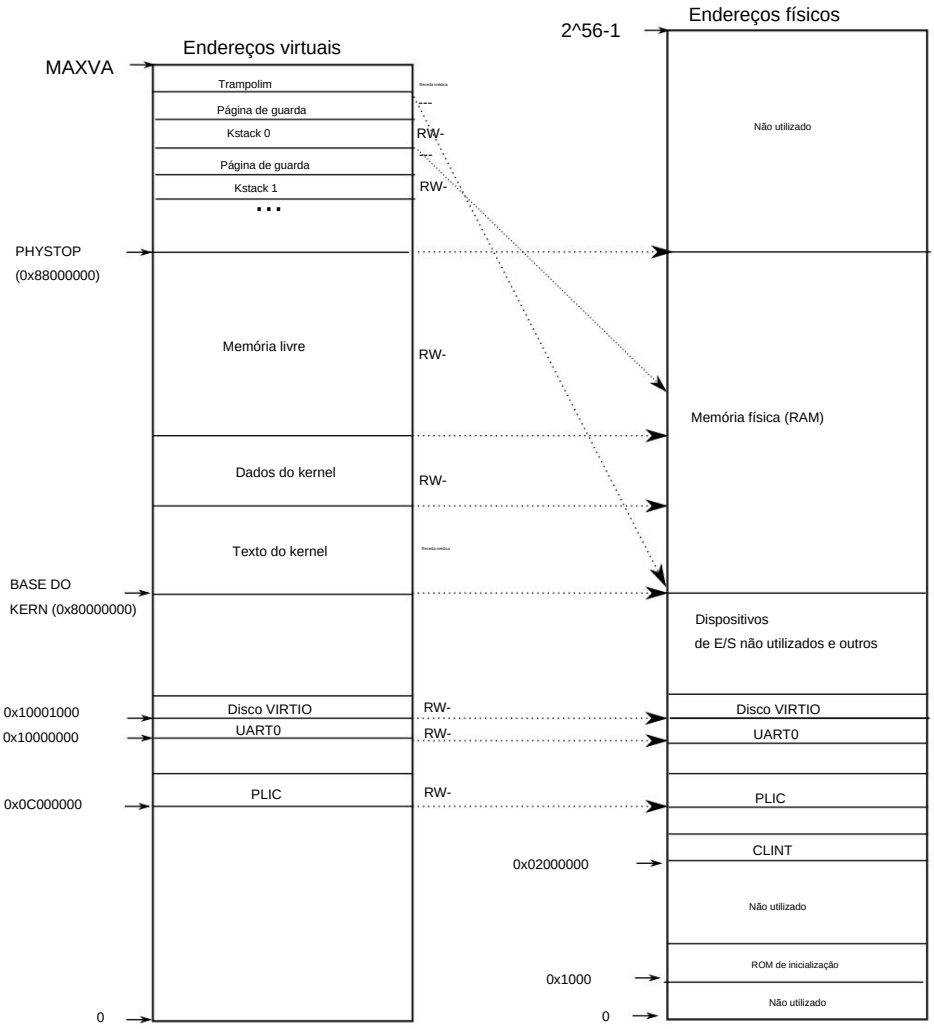

Figura 3.3: À esquerda, o espaço de endereço do kernel do xv6. RWX refere-se a leitura, gravação e execução de PTE permissões. À direita, o espaço de endereço físico RISC-V que o xv6 espera ver.

3.2 Espaço de endereço do kernel

O Xv6 mantém uma tabela de páginas por processo, descrevendo o espaço de endereço do usuário de cada processo, além de uma tabela de página única que descreve o espaço de endereço do kernel. O kernel configura o layout do seu espaço de endereço para obter acesso à memória física e a vários recursos de hardware em intervalos previsíveis.
Endereços virtuais. A Figura 3.3 mostra como este layout mapeia endereços virtuais do kernel para endereços físicos. O arquivo (kernel/memlayout.h) declara as constantes para o layout de memória do kernel do xv6.

O QEMU simula um computador que inclui RAM (memória física) começando no endereço físico 0x80000000 e continuando até pelo menos 0x88000000, que o xv6 chama de PHYSTOP.
A simulação do QEMU também inclui dispositivos de E/S, como uma interface de disco. O QEMU expõe as interfaces do dispositivo ao software como registradores de controle mapeados na memória, localizados abaixo de 0x80000000 no espaço de endereço físico. O kernel pode interagir com os dispositivos lendo/escrevendo esses endereços físicos especiais; tais leituras e gravações se comunicam com o hardware do dispositivo em vez da RAM. O Capítulo 4 explica como o xv6 interage com os dispositivos.

O kernel acessa a RAM e os registradores de dispositivos mapeados na memória usando "mapeamento direto", ou seja, mapeando os recursos em endereços virtuais iguais ao endereço físico. Por exemplo, o próprio kernel está localizado em KERNBASE=0x80000000 tanto no espaço de endereço virtual quanto na memória física. O mapeamento direto simplifica o código do kernel que lê ou grava na memória física. Por exemplo, quando o fork aloca memória do usuário para o processo filho, o alocador retorna o endereço físico dessa memória; o fork usa esse endereço diretamente como um endereço virtual ao copiar a memória do usuário do processo pai para o filho.

Há alguns endereços virtuais do kernel que não são mapeados diretamente:

- A página do trampolim. Ela é mapeada no topo do espaço de endereço virtual; as tabelas de páginas do usuário têm esse mesmo mapeamento. O Capítulo 4 discute o papel da página do trampolim, mas vemos aqui um caso de uso interessante de tabelas de páginas; uma página física (contendo o código do trampolim) é mapeada duas vezes no espaço de endereço virtual do kernel: uma vez no topo do espaço de endereço virtual e outra com um mapeamento direto.

- As páginas da pilha do kernel. Cada processo tem sua própria pilha do kernel, que é mapeada em uma posição alta para que, abaixo dela, o xv6 possa deixar uma página de guarda não mapeada. A PTE da página de guarda é inválida (ou seja, PTE_V não está definida), de modo que, se o kernel transbordar uma pilha do kernel, provavelmente causará uma exceção e o kernel entrará em pânico. Sem uma página de guarda, uma pilha transbordando sobrescreveria outra memória do kernel, resultando em operação incorreta. Uma falha de pânico é preferível.

Embora o kernel utilize suas pilhas por meio de mapeamentos de alta memória, elas também são acessíveis ao kernel por meio de um endereço mapeado diretamente. Um projeto alternativo poderia ter apenas o mapeamento direto e usar as pilhas no endereço mapeado diretamente. Nesse arranjo, no entanto, fornecer páginas de guarda envolveria o desmapeamento de endereços virtuais que, de outra forma, se refeririam à memória física, o que dificultaria o uso.

O kernel mapeia as páginas do trampolim e o texto do kernel com as permissões PTE_R e PTE_X. O kernel lê e executa instruções dessas páginas. O kernel mapeia as outras páginas com as permissões PTE_R e PTE_W, para que possa ler e gravar na memória dessas páginas. Os mapeamentos para as páginas de guarda são inválidos.

3.3 Código: criando um espaço de endereço

A maior parte do código xv6 para manipular espaços de endereço e tabelas de páginas reside em vm.c (kernel/vm.c:1).
A estrutura de dados central é pagetable_t, que na verdade é um ponteiro para um RISC-V

Tabela de páginas raiz; uma pagetable_t pode ser a tabela de páginas do kernel ou uma das tabelas de páginas por processo. As funções centrais são walk, que encontra o PTE para um endereço virtual, e mappages, que instala PTEs para novos mapeamentos. Funções que começam com kvm manipulam a tabela de páginas do kernel; funções que começam com uvm manipulam uma tabela de páginas do usuário; outras funções são usadas para ambos. copyout e copyin copiam dados de e para endereços virtuais do usuário fornecidos como argumentos de chamada de sistema; elas estão em vm.c porque precisam traduzir explicitamente esses endereços para encontrar a memória física correspondente.

No início da sequência de inicialização, o principal chama kvminit (kernel/vm.c:54) para criar a tabela de páginas do kernel usando kvmmake (kernel/vm.c:20). Esta chamada ocorre antes que o xv6 habilite a paginação no RISC-V, portanto, os endereços se referem diretamente à memória física. O kvmmake primeiro aloca uma página de memória física para armazenar a página da tabela de páginas raiz. Em seguida, ele chama o kvmmap para instalar as traduções necessárias ao kernel. As traduções incluem as instruções e os dados do kernel, a memória física até o PHYSTOP e os intervalos de memória que, na verdade, são dispositivos. proc_mapstacks (kernel/proc.c:33) aloca uma pilha do kernel para cada processo. Ele chama o kvmmap para mapear cada pilha no endereço virtual gerado pelo KSTACK, o que deixa espaço para as páginas de proteção de pilha inválidas.

kvmmap (kernel/vm.c:132) chama mappages (kernel/vm.c:143), que instala mapeamentos em uma tabela de páginas para um intervalo de endereços virtuais em um intervalo correspondente de endereços físicos. Ele faz isso separadamente para cada endereço virtual no intervalo, em intervalos de página. Para cada endereço virtual a ser mapeado, mappages chama walk para encontrar o endereço da PTE correspondente a esse endereço. Em seguida, ele inicializa a PTE para conter o número da página física relevante, as permissões desejadas (PTE_W, PTE_X e/ou PTE_R) e PTE_V para marcar a PTE como válida (kernel/vm.c:158).

andar (kernel/vm.c:86) imita o hardware de paginação RISC-V enquanto procura no PTE por um endereço virtual (veja a Figura 3.2). O comando walk desce a tabela de páginas de 3 níveis 9 bits por vez. Ele usa os 9 bits de endereço virtual de cada nível para encontrar o PTE da tabela de páginas do próximo nível ou da página final (kernel/vm.c:92). Se o PTE não for válido, a página necessária ainda não foi alocada; se o argumento alloc estiver definido, walk aloca uma nova página da tabela de páginas e insere seu endereço físico no PTE. Ele retorna o endereço do PTE na camada mais baixa da árvore (kernel/vm.c:102).

O código acima depende da memória física ser mapeada diretamente no espaço de endereço virtual do kernel. Por exemplo, à medida que o walk desce níveis da tabela de páginas, ele extrai o endereço (físico) da tabela de páginas do nível seguinte de uma PTE (kernel/vm.c:94). e então usa esse endereço como um endereço virtual para buscar o PTE no próximo nível abaixo (kernel/vm.c:92).

chamadas principais kvminithart (kernel/vm.c:62) para instalar a tabela de páginas do kernel. Ele grava o endereço físico da página raiz da tabela de páginas no registrador satp. Depois disso, a CPU traduzirá os endereços usando a tabela de páginas do kernel. Como o kernel usa um mapeamento de identidade, o endereço virtual da próxima instrução será mapeado para o endereço de memória física correto.

Cada CPU RISC-V armazena em cache as entradas da tabela de páginas em um Translation Look-aside Buffer (TLB) e, quando o xv6 altera uma tabela de páginas, deve informar à CPU para invalidar as entradas TLB em cache correspondentes. Caso contrário, em algum momento posterior, o TLB poderá usar um mapeamento em cache antigo, apontando para uma página física que, nesse meio tempo, foi alocada a outro processo e, como resultado, um processo poderá rabiscar na memória de outro processo. O RISC-V possui uma instrução sfence.vma que limpa o TLB da CPU atual. O xv6 executa sfence.vma em kvminithart após recarregar o registrador satp e no código trampoline que alterna para um

tabela de página do usuário antes de retornar ao espaço do usuário (kernel/trampoline.S:89).

Também é necessário emitir sfence.vma antes de alterar o satp, para aguardar a conclusão de todos os carregamentos e armazenamentos pendentes. Essa espera garante que as atualizações anteriores na tabela de páginas tenham sido concluídas e que os carregamentos e armazenamentos anteriores usem a tabela de páginas antiga, não a nova.
um.

Para evitar a limpeza completa do TLB, as CPUs RISC-V podem suportar identificadores de espaço de endereço (ASIDs) [3]. O kernel pode então limpar apenas as entradas TLB para um espaço de endereço específico. O Xv6 não utiliza esse recurso.

3.4 Alocação de memória física

O kernel deve alocar e liberar memória física em tempo de execução para tabelas de páginas, memória do usuário, pilhas do kernel e buffers de pipe.

O Xv6 utiliza a memória física entre o final do kernel e o PHYSTOP para alocação em tempo de execução. Ele aloca e libera páginas inteiras de 4.096 bytes por vez. Ele rastreia quais páginas estão livres, encadeando uma lista encadeada entre as próprias páginas. A alocação consiste em remover uma página da lista encadeada; a liberação consiste em adicionar a página liberada à lista.

3.5 Código: Alocador de memória física

O alocador reside em kalloc.c (kernel/kalloc.c:1). A estrutura de dados do alocador é uma lista livre de páginas de memória física disponíveis para alocação. Cada elemento da lista de páginas livres é uma execução de struct (kernel/kalloc.c:17). De onde o alocador obtém a memória para armazenar essa estrutura de dados? Ele armazena a estrutura de execução de cada página livre na própria página livre, já que não há mais nada armazenado lá. A lista livre é protegida por um bloqueio de spin (kernel/kalloc.c:21-24). A lista e o bloqueio são encapsulados em uma struct para deixar claro que o bloqueio protege os campos na struct. Por enquanto, ignore o bloqueio e as chamadas para acquire e release; o Capítulo 6 examinará o bloqueio em detalhes.

A função main chama kinit para inicializar o alocador (kernel/kalloc.c:27). O kinit inicializa a lista de páginas livres para armazenar todas as páginas entre o final do kernel e o PHYSTOP. O Xv6 deve determinar quanta memória física está disponível analisando as informações de configuração fornecidas pelo hardware. Em vez disso, o xv6 assume que a máquina possui 128 megabytes de RAM. O kinit chama o freerange para adicionar memória à lista de páginas livres por meio de chamadas por página ao kfree. Uma PTE só pode se referir a um endereço físico alinhado em um limite de 4096 bytes (é um múltiplo de 4096), portanto, o freerange usa PGROUNDUP para garantir que libere apenas os endereços físicos alinhados. O alocador inicia sem memória; essas chamadas ao kfree lhe dão algum trabalho para gerenciar.

O alocador às vezes trata endereços como inteiros para realizar operações aritméticas com eles (por exemplo, percorrendo todas as páginas em modo livre) e, às vezes, usa endereços como ponteiros para ler e escrever na memória (por exemplo, manipulando a estrutura de execução armazenada em cada página); esse uso duplo de endereços é o principal motivo pelo qual o código do alocador está cheio de conversões de tipo em C. O outro motivo é que a liberação e a alocação alteram inerentemente o tipo de memória.

A função kfree (kernel/kalloc.c:47) começa definindo cada byte na memória que está sendo liberada para o valor 1. Isso fará com que o código que usa memória após liberá-la (usa “referências pendentes”) leia lixo em vez do antigo conteúdo válido; esperançosamente, isso fará com que esse código quebre mais rápido.
Em seguida, kfree anexa a página à lista livre: ele converte pa para um ponteiro para struct run, registra o início antigo da lista livre em r->next e define a lista livre como igual a r. kalloc remove e retorna o primeiro elemento na lista livre.

3.6 Espaço de endereço do processo

Cada processo possui uma tabela de páginas separada e, quando o xv6 alterna entre processos, também altera as tabelas de páginas. A Figura 3.4 mostra o espaço de endereço de um processo com mais detalhes do que a Figura 2.3. A memória de usuário de um processo começa no endereço virtual zero e pode crescer até MAXVA (kernel/riscv.h:360). permitindo que um processo enderece em princípio 256 Gigabytes de memória.

O espaço de endereço de um processo consiste em páginas que contêm o texto do programa (que o xv6 mapeia com as permissões PTE_R, PTE_X e PTE_U), páginas que contêm os dados pré-inicializados do programa, uma página para a pilha e páginas para o heap. O xv6 mapeia os dados, a pilha e o heap com as permissões PTE_R, PTE_W e PTE_U.

Usar permissões dentro de um espaço de endereço de usuário é uma técnica comum para proteger um processo de usuário. Se o texto fosse mapeado com PTE_W, um processo poderia modificar acidentalmente seu próprio programa; por exemplo, um erro de programação pode fazer com que o programa escreva em um ponteiro nulo, modificando instruções no endereço 0, e então continue em execução, talvez causando mais confusão. Para detectar esses erros imediatamente, o xv6 mapeia o texto sem PTE_W; se um programa tentar acidentalmente armazenar no endereço 0, o hardware se recusará a executar o armazenamento e gerará uma falha de página (consulte a Seção 4.6).
O kernel então mata o processo e imprime uma mensagem informativa para que o desenvolvedor possa rastrear o problema.

Da mesma forma, ao mapear dados sem PTE_X, um programa de usuário não pode pular acidentalmente para um endereço nos dados do programa e começar a executar nesse endereço.

No mundo real, fortalecer um processo por meio da definição cuidadosa de permissões também auxilia na defesa contra ataques de segurança. Um adversário pode fornecer uma entrada cuidadosamente construída a um programa (por exemplo, um servidor web) que acione um bug no programa na esperança de transformá-lo em um exploit [14].
Definir permissões cuidadosamente e outras técnicas, como a randomização do layout do espaço de endereço do usuário, tornam esses ataques mais difíceis.

A pilha é uma única página e é exibida com o conteúdo inicial criado por exec. Strings contendo os argumentos da linha de comando, bem como um array de ponteiros para eles, ficam no topo da pilha. Logo abaixo, estão os valores que permitem que um programa inicie em main como se a função main(argc, argv) tivesse acabado de ser chamada.

Para detectar que uma pilha de usuário está transbordando a memória alocada, o xv6 coloca uma página de guarda inacessível logo abaixo da pilha, limpando o sinalizador PTE_U . Se a pilha de usuário transbordar e o processo tentar usar um endereço abaixo da pilha, o hardware gerará uma exceção de falha de página porque a página de guarda está inacessível para um programa em execução no modo de usuário. Um sistema operacional real pode, em vez disso, alocar automaticamente mais memória para a pilha de usuário quando ela transborda.

Quando um processo solicita ao xv6 mais memória do usuário, o xv6 aumenta o heap do processo. O xv6 usa primeiro

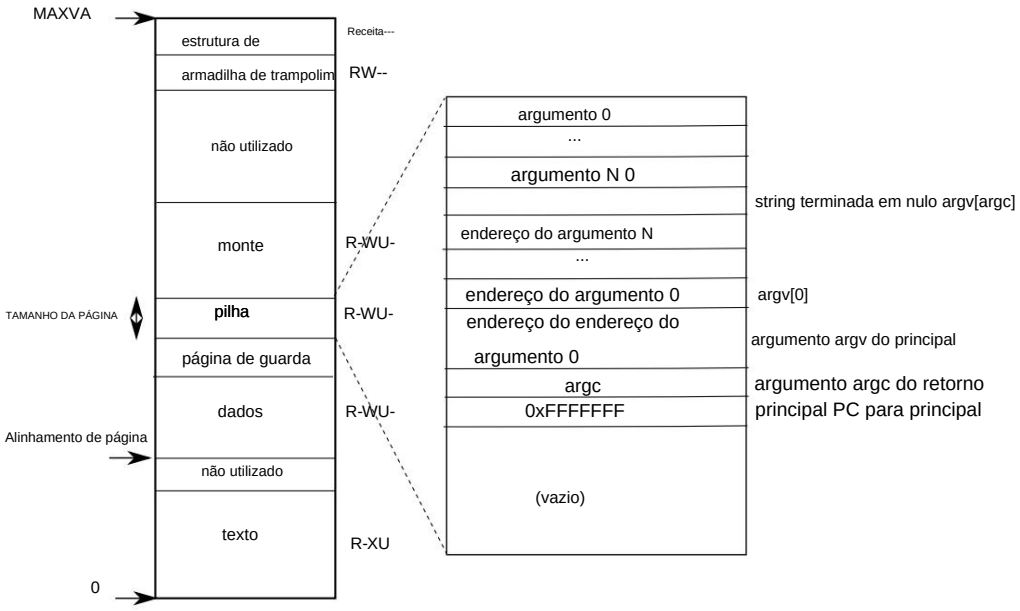

Figura 3.4: Espaço de endereço de usuário de um processo, com sua pilha inicial.

kalloc para alocar páginas físicas. Em seguida, ele adiciona PTEs à tabela de páginas do processo que apontam para as novas páginas físicas. O Xv6 define os sinalizadores PTE_W, PTE_R, PTE_U e PTE_V nesses PTEs. A maioria dos processos não utiliza todo o espaço de endereço do usuário; o Xv6 deixa PTE_V livre em PTEs não utilizados.

Vemos aqui alguns bons exemplos do uso de tabelas de páginas. Primeiro, as tabelas de páginas de diferentes processos traduzem endereços de usuários para diferentes páginas de memória física, de modo que cada processo tenha memória de usuário privada. Segundo, cada processo vê sua memória como tendo endereços virtuais contíguos começando em zero, enquanto a memória física do processo pode ser não contígua. Terceiro, o kernel mapeia uma página com código trampolim no topo do espaço de endereço do usuário (sem PTE_U), portanto, uma única página de memória física aparece em todos os espaços de endereço, mas pode ser usada apenas pelo kernel.

3.7 Código: sbrk

sbrk é a chamada de sistema para que um processo reduza ou aumente sua memória. A chamada de sistema é implementada pela função growproc (kernel/proc.c:260). growproc chama uvmalloc ou uvmdealloc, dependendo se n é positivo ou negativo. uvmalloc (kernel/vm.c:226) aloca memória física com kalloc e adiciona PTEs à tabela de páginas do usuário com mappages. uvmdealloc chama uvmunmap (kernel/vm.c:171), que usa walk para encontrar PTEs e kfree para liberar a memória física à qual se referem.

O Xv6 usa a tabela de páginas de um processo não apenas para informar ao hardware como mapear os endereços virtuais do usuário, mas também como o único registro de quais páginas de memória física são alocadas para aquele processo. É por isso que a liberação de memória do usuário (no uvmunmap) requer a análise da tabela de páginas do usuário.

3.8 Código: exec

exec é uma chamada de sistema que substitui o espaço de endereço do usuário de um processo por dados lidos de um arquivo, chamado de arquivo binário ou executável. Um binário normalmente é a saída do compilador e do vinculador e contém instruções de máquina e dados do programa. exec (kernel/exec.c:23) abre o caminho binário nomeado usando namei (kernel/exec.c:36), que é explicado no Capítulo 8. Em seguida, ele lê o cabeçalho ELF. Os binários Xv6 são formatados no formato ELF amplamente utilizado, definido em (kernel/elf.h). Um binário ELF consiste em um cabeçalho ELF, struct elfhdr (kernel/elf.h:6), seguido por uma sequência de cabeçalhos de seção do programa, struct proghdr (kernel/elf.h:25). Cada progvhdr descreve uma seção do aplicativo que deve ser carregada na memória; os programas xv6 têm dois cabeçalhos de seção de programa: um para instruções e um para dados.

O primeiro passo é uma verificação rápida para verificar se o arquivo provavelmente contém um binário ELF. Um binário ELF começa com o "número mágico" de quatro bytes 0x7F, 'E', 'L', 'F' ou ELF_MAGIC (kernel/elf.h:3). Se o cabeçalho ELF tiver o número mágico correto, exec assume que o binário está bem formado.

exec aloca uma nova tabela de páginas sem mapeamentos de usuários com proc_pagetable (kernel/exec.c:49), aloca memória para cada segmento ELF com uvmalloc (kernel/exec.c:65), e carrega cada segmento na memória com loadseg (kernel/exec.c:10). loadseg usa walkaddr para encontrar o endereço físico da memória alocada na qual gravar cada página do segmento ELF e readi para ler o arquivo.

O cabeçalho da seção do programa para /init, o primeiro programa de usuário criado com exec, se parece com isto:

```
# objdump -p usuário/_init

usuário/_init:                formato de arquivo elf64-little

Cabeçalho do programa:
0x70000003 off                0x000000000000006bb0 vaddr 0x0000000000000000
                                                                paddr 0x0000000000000000 align 2**0 filesz 0x000000000000004a
                memsz 0x0000000000000000 sinalizadores r--
    CARREGAR desligado        0x0000000000001000 vaddr 0x0000000000000000
                                                                paddr 0x0000000000000000 alinhar 2**12 arquivosz 0x0000000000001000
                memsz 0x0000000000001000 sinalizadores rx
    CARREGAR desligado        0x0000000000002000 vaddr 0x0000000000001000
                                                                paddr 0x0000000000001000 alinhar 2**12 arquivosz 0x0000000000000010
                memsz 0x0000000000000030 sinalizadores rw-
EMPILHAMENTO desligado        0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000
                                                                alinhar 2**4 arquivosz 0x0000000000000000 memsz
                0x0000000000000000 sinalizadores rw-
```

Vemos que o segmento de texto deve ser carregado no endereço virtual 0x0 na memória (sem permissões de gravação) a partir do conteúdo no deslocamento 0x1000 do arquivo. Também vemos que os dados devem ser carregados no endereço 0x1000, que está no limite de uma página, e sem permissões de execução.

O valor "filesz" do cabeçalho de uma seção de programa pode ser menor que o valor "memsz", indicando que o espaço entre eles deve ser preenchido com zeros (para variáveis globais em C) em vez de lido do arquivo. Para /init, o valor "filesz" de dados é 0x10 bytes e o valor "memsz" é 0x30 bytes, e, portanto, uvmalloc aloca memória física suficiente para armazenar 0x30 bytes, mas lê apenas 0x10 bytes do arquivo /init.

Agora, exec aloca e inicializa a pilha do usuário. Ele aloca apenas uma página de pilha. exec copia as strings de argumentos para o topo da pilha, uma de cada vez, registrando os ponteiros para elas em ustack.
Ele coloca um ponteiro nulo no final do que será a lista argv passada para main. As três primeiras entradas em ustack são o contador de programa de retorno falso, argc e o ponteiro argv .

O comando exec coloca uma página inacessível logo abaixo da página da pilha, de modo que programas que tentarem usar mais de uma página apresentarão falhas. Essa página inacessível também permite que o comando exec trate de argumentos muito grandes; nessa situação, a cópia (kernel/vm.c:352) a função que exec usa para copiar argumentos para a pilha notará que a página de destino não está acessível e retornará -1.

Durante a preparação da nova imagem de memória, se exec detectar um erro, como um segmento de programa inválido, ele salta para o rótulo "bad", libera a nova imagem e retorna -1. exec deve aguardar para liberar a imagem antiga até ter certeza de que a chamada de sistema será bem-sucedida: se a imagem antiga desaparecer, a chamada de sistema não poderá retornar -1 para ela. Os únicos casos de erro em exec ocorrem durante a criação da imagem. Assim que a imagem estiver concluída, exec pode confirmar a nova tabela de páginas (kernel/exec.c:125). e liberar o antigo (kernel/exec.c:129).

exec carrega bytes do arquivo ELF para a memória em endereços especificados pelo arquivo ELF. Usuários ou processos podem colocar quaisquer endereços que desejarem em um arquivo ELF. Portanto, exec é arriscado, pois os endereços no arquivo ELF podem se referir ao kernel, acidentalmente ou propositalmente. As consequências para um kernel desavisado podem variar de uma falha a uma subversão maliciosa dos mecanismos de isolamento do kernel (ou seja, uma falha de segurança). O Xv6 realiza uma série de verificações para evitar esses riscos. Por exemplo, if(ph.vaddr + ph.memsz < ph.vaddr) verifica se a soma excede um inteiro de 64 bits. O perigo é que um usuário possa construir um binário ELF com um ph.vaddr que aponte para um endereço escolhido pelo usuário e um ph.memsz grande o suficiente para que a soma exceda para 0x1000, o que parecerá um valor válido. Em uma versão mais antiga do xv6, na qual o espaço de endereço do usuário também continha o kernel (mas não era legível/gravável no modo usuário), o usuário podia escolher um endereço que correspondesse à memória do kernel e, assim, copiar os dados do binário ELF para o kernel. Na versão RISC-V do xv6, isso não acontece, pois o kernel tem sua própria tabela de páginas; loadseg carrega na tabela de páginas do processo, não na tabela de páginas do kernel.

É fácil para um desenvolvedor de kernel omitir uma verificação crucial, e kernels do mundo real têm um longo histórico de verificações ausentes, cuja ausência pode ser explorada por programas de usuário para obter privilégios de kernel. É provável que o xv6 não faça um trabalho completo de validação dos dados de nível de usuário fornecidos ao kernel, o que um programa de usuário malicioso pode explorar para contornar o isolamento do xv6.

3.9 Mundo real

Como a maioria dos sistemas operacionais, o xv6 usa o hardware de paginação para proteção e mapeamento de memória. A maioria dos sistemas operacionais faz uso muito mais sofisticado de paginação do que o xv6, combinando exceções de paginação e falha de página, que discutiremos no Capítulo 4.

O Xv6 é simplificado pelo uso, pelo kernel, de um mapeamento direto entre endereços virtuais e físicos, e pela suposição de que há RAM física no endereço 0x8000000, onde o kernel espera ser carregado. Isso funciona com o QEMU, mas em hardware real acaba sendo uma má ideia; o hardware real coloca RAM e dispositivos em endereços físicos imprevisíveis, de modo que (por exemplo) pode não haver RAM em 0x8000000, onde o xv6 espera poder armazenar o kernel. Um kernel mais sério

os projetos exploram a tabela de páginas para transformar layouts arbitrários de memória física de hardware em layouts previsíveis de endereços virtuais do kernel.

O RISC-V suporta proteção no nível de endereços físicos, mas o xv6 não usa esse recurso.

Em máquinas com muita memória, pode fazer sentido usar o suporte do RISC-V para "superpáginas". Páginas pequenas fazem sentido quando a memória física é pequena, para permitir alocação e paginação para o disco com granularidade fina. Por exemplo, se um programa usa apenas 8 quilobytes de memória, atribuir a ele uma superpágina inteira de 4 megabytes de memória física é um desperdício. Páginas maiores fazem sentido em máquinas com muita RAM e podem reduzir a sobrecarga na manipulação da tabela de páginas.

A falta de um alocador do tipo malloc no kernel xv6 que possa fornecer memória para objetos pequenos impede que o kernel use estruturas de dados sofisticadas que exigiriam alocação dinâmica.
Um kernel mais elaborado provavelmente alocaria muitos tamanhos diferentes de blocos pequenos, em vez de (como no xv6) apenas blocos de 4.096 bytes; um alocador de kernel real precisaria lidar com alocações pequenas e também grandes.

A alocação de memória é um tópico recorrente e polêmico, cujos problemas básicos são o uso eficiente de memória limitada e a preparação para solicitações futuras desconhecidas [9]. Hoje em dia, as pessoas se preocupam mais com velocidade do que com eficiência de espaço.

3.10 Exercícios

1. Analise a árvore de dispositivos do RISC-V para descobrir a quantidade de memória física que o computador tem.

2. Escreva um programa de usuário que aumente seu espaço de endereço em um byte chamando sbrk(1). Execute o programa e investigue a tabela de páginas do programa antes e depois da chamada de sbrk . Quanto espaço o kernel alocou? O que contém o PTE para a nova memória?

3. Modifique o xv6 para usar super páginas para o kernel.

4. As implementações Unix de exec tradicionalmente incluem tratamento especial para scripts de shell. Se o arquivo a ser executado começar com o texto #!, a primeira linha será considerada um programa a ser executado para interpretar o arquivo. Por exemplo, se exec for chamado para executar myprog arg1 e a primeira linha de myprog for #!/interp, então exec executará /interp com a linha de comando /interp myprog arg1.
   Implemente suporte para esta convenção no xv6.

5. Implemente a randomização do layout do espaço de endereço para o kernel.

Capítulo 4

Armadilhas e chamadas de sistema

Existem três tipos de eventos que fazem com que a CPU interrompa a execução normal de instruções e force a transferência de controle para um código especial que lida com o evento. Uma situação é uma chamada de sistema, quando um programa de usuário executa a instrução ecall para solicitar que o kernel faça algo por ele. Outra situação é uma exceção: uma instrução (de usuário ou kernel) faz algo ilegal, como dividir por zero ou usar um endereço virtual inválido. A terceira situação é uma interrupção de dispositivo, quando um dispositivo sinaliza que precisa de atenção, por exemplo, quando o hardware do disco conclui uma solicitação de leitura ou gravação.

Este livro usa "trap" como um termo genérico para essas situações. Normalmente, qualquer código que estivesse em execução no momento da trap precisará ser retomado posteriormente e não precisaria estar ciente de que algo especial aconteceu. Ou seja, frequentemente queremos que as traps sejam transparentes; isso é particularmente importante para interrupções de dispositivo, que o código interrompido normalmente não espera. A sequência usual é que uma trap força uma transferência de controle para o kernel; o kernel salva registradores e outros estados para que a execução possa ser retomada; o kernel executa o código manipulador apropriado (por exemplo, uma implementação de chamada de sistema ou driver de dispositivo); o kernel restaura o estado salvo e retorna da trap; e o código original continua de onde parou.

O Xv6 lida com todas as armadilhas no kernel; as armadilhas não são entregues ao código do usuário. Lidar com armadilhas no kernel é natural para chamadas de sistema. Faz sentido para interrupções, já que o isolamento exige que apenas o kernel tenha permissão para usar dispositivos e porque o kernel é um mecanismo conveniente para compartilhar dispositivos entre vários processos. Também faz sentido para exceções, já que o Xv6 responde a todas as exceções do espaço do usuário eliminando o programa ofensivo.

O tratamento de traps Xv6 ocorre em quatro etapas: ações de hardware executadas pela CPU RISC-V, algumas instruções de montagem que preparam o caminho para o código C do kernel, uma função C que decide o que fazer com o trap e a chamada de sistema ou rotina de serviço do driver de dispositivo. Embora a semelhança entre os três tipos de traps sugira que um kernel poderia lidar com todos os traps com um único caminho de código, torna-se conveniente ter código separado para três casos distintos: traps do espaço do usuário, traps do espaço do kernel e interrupções de temporizador. O código do kernel (assembler ou C) que processa um trap é frequentemente chamado de manipulador; as primeiras instruções do manipulador são geralmente escritas em assembler (em vez de C) e, às vezes, são chamadas de vetor.

4.1 Máquinas de armadilha RISC-V

Cada CPU RISC-V possui um conjunto de registradores de controle que o kernel escreve para informar à CPU como lidar com traps, e que o kernel pode ler para descobrir se um trap ocorreu. Os documentos RISC-V contêm a história completa [3]. riscv.h (kernel/riscv.h:1) contém definições que o xv6 usa. Aqui está um esboço dos registros mais importantes:

- stvec: O kernel escreve o endereço do seu manipulador de trap aqui; o RISC-V salta para o endereço em stvec para manipular uma armadilha.
- sepc: Quando ocorre uma trap, o RISC-V salva o contador de programa aqui (já que o pc é então sobrescrito pelo valor em stvec). A instrução sret (retorno da trap) copia o sepc para o pc. O kernel pode escrever sepc para controlar para onde o sret vai.
- causa: o RISC-V coloca um número aqui que descreve o motivo da armadilha.
- sscratch: O código do manipulador de traps usa sscratch para ajudar a evitar a substituição de registros de usuários antes de salvá-los.
- sstatus: O bit SIE em sstatus controla se as interrupções do dispositivo estão habilitadas. Se o kernel limpar o SIE, o RISC-V adiará as interrupções do dispositivo até que o kernel configure o SIE. O bit SPP indica se uma interceptação veio do modo usuário ou do modo supervisor e controla para qual modo sret retorna.

Os registradores acima referem-se a traps manipulados no modo supervisor e não podem ser lidos ou escritos no modo usuário. Existe um conjunto semelhante de registradores de controle para traps manipulados no modo máquina; o xv6 os utiliza apenas para o caso especial de interrupções de temporizador.

Cada CPU em um chip multinúcleo tem seu próprio conjunto desses registradores, e mais de uma CPU pode estar manipulando uma armadilha a qualquer momento.

Quando precisa forçar uma armadilha, o hardware RISC-V faz o seguinte para todos os tipos de armadilha (exceto interrupções de temporizador):

1. Se a armadilha for uma interrupção de dispositivo e o bit SIE sstatus estiver limpo, não faça nada seguindo.
2. Desabilite as interrupções limpando o bit SIE em sstatus.
3. Copie o pc para sepc.
4. Salve o modo atual (usuário ou supervisor) no bit SPP em sstatus.
5. Defina scause para refletir a causa da armadilha.
6. Defina o modo como supervisor.
7. Copie stvec para o pc.

8. Comece a executar no novo PC.

Observe que a CPU não alterna para a tabela de páginas do kernel, não alterna para uma pilha no kernel e não salva nenhum registrador além do pc. O software do kernel deve executar essas tarefas.
Um motivo pelo qual a CPU realiza um trabalho mínimo durante uma armadilha é para fornecer flexibilidade ao software; por exemplo, alguns sistemas operacionais omitem uma troca de tabela de páginas em algumas situações para aumentar o desempenho da armadilha.

Vale a pena considerar se algum dos passos listados acima poderia ser omitido, talvez em busca de armadilhas mais rápidas. Embora existam situações em que uma sequência mais simples possa funcionar, muitas das etapas seriam perigosas se omitidas em geral. Por exemplo, suponha que a CPU não alterne os contadores de programa. Então, uma armadilha do espaço do usuário poderia alternar para o modo supervisor enquanto ainda executa instruções do usuário. Essas instruções do usuário poderiam quebrar o isolamento usuário/kernel, por exemplo, modificando o registrador satp para apontar para uma tabela de páginas que permitisse o acesso a toda a memória física.
Portanto, é importante que a CPU alterne para um endereço de instrução especificado pelo kernel, ou seja, stvec.

4.2 Armadilhas do espaço do usuário

O Xv6 lida com armadilhas de forma diferente, dependendo se elas ocorrem durante a execução no kernel ou no código do usuário. Aqui está a história das armadilhas do código do usuário; a Seção 4.5 descreve as armadilhas do código do kernel.

Uma armadilha pode ocorrer durante a execução no espaço do usuário se o programa do usuário fizer uma chamada de sistema (instrução ecall), ou algo ilegal, ou se um dispositivo interromper. O caminho de alto nível de uma armadilha do espaço do usuário é uservec (kernel/trampoline.S:21). então usertrap (kernel/trap.c:37); e ao retornar, usertrapret (kernel/trap.c:90) e então userret (kernel/trampoline.S:101).

Uma grande limitação no design do tratamento de traps do xv6 é o fato de que o hardware RISC-V não alterna entre tabelas de páginas ao forçar um trap. Isso significa que o endereço do manipulador de traps em stvec deve ter um mapeamento válido na tabela de páginas do usuário, já que essa é a tabela de páginas em vigor quando o código de tratamento de traps começa a ser executado. Além disso, o código de tratamento de traps do xv6 precisa alternar para a tabela de páginas do kernel; para poder continuar a execução após essa troca, a tabela de páginas do kernel também deve ter um mapeamento para o manipulador apontado por stvec.

O Xv6 atende a esses requisitos usando uma página de trampolim. A página de trampolim contém uservec,
O código de manipulação de traps xv6 para o qual stvec aponta. A página do trampolim é mapeada na tabela de páginas de cada processo no endereço TRAMPOLINE, que está no topo do espaço de endereço virtual, de modo que estará acima da memória que os programas usam para si mesmos. A página do trampolim também é mapeada no endereço TRAMPOLINE na tabela de páginas do kernel. Veja as Figuras 2.3 e 3.3. Como a página do trampolim é mapeada na tabela de páginas do usuário, sem o sinalizador PTE_U, as traps podem começar a ser executadas lá no modo supervisor. Como a página do trampolim é mapeada no mesmo endereço no espaço de endereço do kernel, o manipulador de traps pode continuar a execução após alternar para a tabela de páginas do kernel.

O código para o manipulador de armadilhas uservec está em trampoline.S (kernel/trampoline.S:21). Quando o uservec inicia, todos os 32 registradores contêm valores pertencentes ao código de usuário interrompido. Esses 32 valores precisam ser salvos em algum lugar na memória, para que possam ser restaurados quando a armadilha retornar ao espaço do usuário. Armazenar na memória requer o uso de um registrador para armazenar o endereço, mas neste ponto não há

Não há registradores de uso geral disponíveis! Felizmente, o RISC-V oferece uma ajuda na forma do registrador sscratch. A instrução csrw no início de uservec salva a0 em sscratch.
Agora o uservec tem um registrador (a0) para brincar.

A próxima tarefa do uservec é salvar os 32 registradores de usuário. O kernel aloca, para cada processo, uma página de memória para uma estrutura trapframe que (entre outras coisas) tem espaço para salvar os 32 registradores de usuário (kernel/proc.h:43). Como satp ainda se refere à tabela de páginas do usuário, uservec precisa que o trapframe seja mapeado no espaço de endereço do usuário. Xv6 mapeia o trapframe de cada processo no endereço virtual TRAPFRAME na tabela de páginas do usuário desse processo; TRAPFRAME está logo abaixo de TRAMPOLINE. O p->trapframe do processo também aponta para o trapframe, embora em seu endereço físico, para que o kernel possa usá-lo por meio da tabela de páginas do kernel.

Assim, uservec carrega o endereço TRAPFRAME em a0 e salva todos os registros do usuário lá, incluindo o a0 do usuário, lidos do zero.

O trapframe contém o endereço da pilha do kernel do processo atual, o hartid da CPU atual, o endereço da função usertrap e o endereço da tabela de páginas do kernel. uservec recupera esses valores, alterna satp para a tabela de páginas do kernel e chama usertrap.

O trabalho do usertrap é determinar a causa da armadilha, processá-la e retornar (kernel/- trap.c:37). Primeiro, ele altera stvec para que uma trap enquanto estiver no kernel seja manipulada por kernelvec em vez de uservec. Ele salva o registrador sepc (o contador de programa do usuário salvo), porque usertrap pode chamar yield para alternar para a thread do kernel de outro processo, e esse processo pode retornar ao espaço do usuário, no processo do qual ele modificará sepc. Se a trap for uma chamada de sistema, usertrap chama syscall para manipulá-la; se for uma interrupção de dispositivo, devintr; caso contrário, é uma ex-ception, e o kernel mata o processo com falha. O caminho da chamada de sistema adiciona quatro ao contador de programa do usuário salvo porque o RISC-V, no caso de uma chamada de sistema, deixa o ponteiro do programa apontando para a instrução ecall, mas o código do usuário precisa retomar a execução na instrução subsequente.
Na saída, o usertrap verifica se o processo foi encerrado ou se deve render a CPU (se essa armadilha for uma interrupção de timer).

O primeiro passo para retornar ao espaço do usuário é a chamada para usertrapret (kernel/trap.c:90). Esta função configura os registradores de controle RISC-V para se prepararem para uma futura captura a partir do espaço do usuário. Isso envolve alterar stvec para se referir a uservec, preparar os campos do trapframe dos quais uservec depende e definir sepc para o contador de programa do usuário salvo anteriormente. No final, usertrapret chama userret na página do trampolim mapeada nas tabelas de páginas do usuário e do kernel; o motivo é que o código assembly em userret alternará as tabelas de páginas.

A chamada de usertrapret para userret passa um ponteiro para a tabela de páginas de usuário do processo em a0 (kernel/trampoline.S:101). userret alterna o satp para a tabela de páginas do usuário do processo. Lembre-se de que a tabela de páginas do usuário mapeia tanto a página do trampolim quanto a TRAPFRAME, mas nada mais do kernel. O mapeamento da página do trampolim no mesmo endereço virtual nas tabelas de páginas do usuário e do kernel permite que o userret continue executando após a alteração do satp. A partir deste ponto, os únicos dados que o userret pode usar são o conteúdo do registrador e o conteúdo do trapframe. O userret carrega o endereço TRAPFRAME em a0, restaura os registradores de usuário salvos do trapframe via a0, restaura o usuário a0 salvo e executa sret para retornar ao espaço do usuário.

4.3 Código: Chamando chamadas de sistema

O capítulo 2 terminou com initcode.S invocando a chamada de sistema exec (user/initcode.S:11). Vamos ver como a chamada do usuário chega à implementação da chamada do sistema exec no kernel.

initcode.S coloca os argumentos para exec nos registradores a0 e a1, e coloca o número da chamada do sistema em a7. Os números da chamada do sistema correspondem às entradas no array syscalls, uma tabela de ponteiros de função (kernel/syscall.c:107). A instrução ecall é capturada no kernel e faz com que uservec, usertrap e então syscall sejam executados, como vimos acima.

chamada de sistema (kernel/syscall.c:132) recupera o número de chamada do sistema do a7 salvo no trapframe e o utiliza para indexar chamadas de sistema. Para a primeira chamada de sistema, a7 contém SYS_exec (kernel /syscall.h:8), resultando em uma chamada para a função de implementação de chamada do sistema sys_exec.

Quando sys_exec retorna, syscall registra seu valor de retorno em p->trapframe->a0. Isso fará com que a chamada original do espaço do usuário para exec() retorne esse valor, já que a convenção de chamada C no RISC-V coloca os valores de retorno em a0. Chamadas de sistema convencionalmente retornam números negativos para indicar erros e zero ou números positivos para sucesso. Se o número da chamada de sistema for inválido, syscall imprime um erro e retorna -1.

4.4 Código: Argumentos de chamada de sistema

Implementações de chamadas de sistema no kernel precisam encontrar os argumentos passados pelo código do usuário. Como o código do usuário chama funções wrapper de chamadas de sistema, os argumentos estão inicialmente onde a convenção de chamada RISC-VC os coloca: em registradores. O código de trap do kernel salva os registradores do usuário no quadro de trap do processo atual, onde o código do kernel pode encontrá-los. As funções do kernel argint, argaddr e argfd recuperam o n 'ésimo argumento de chamada de sistema do quadro de trap como um inteiro, um ponteiro ou um descritor de arquivo. Todas elas chamam argraw para recuperar o registrador de usuário salvo apropriado (kernel/syscall.c:34).

Algumas chamadas de sistema passam ponteiros como argumentos, e o kernel deve usar esses ponteiros para ler ou escrever na memória do usuário. A chamada de sistema exec, por exemplo, passa ao kernel um conjunto de ponteiros referentes a argumentos de string no espaço do usuário. Esses ponteiros apresentam dois desafios. Primeiro, o programa do usuário pode estar com bugs ou ser malicioso, e pode passar ao kernel um ponteiro inválido ou um ponteiro com a intenção de induzir o kernel a acessar a memória do kernel em vez da memória do usuário. Segundo, os mapeamentos de tabelas de páginas do kernel xv6 não são os mesmos que os mapeamentos de tabelas de páginas do usuário, portanto, o kernel não pode usar instruções comuns para carregar ou armazenar a partir de endereços fornecidos pelo usuário.

O kernel implementa funções que transferem dados com segurança de e para endereços fornecidos pelo usuário. fetchstr é um exemplo (kernel/syscall.c:25). Chamadas de sistema de arquivos como exec usam fetchstr para recuperar argumentos de nome de arquivo de string do espaço do usuário. fetchstr chama copyinstr para fazer o trabalho pesado. copyinstr (kernel/vm.c:403) Copia até max bytes para dst do endereço virtual srcva na tabela de páginas do usuário pagetable. Como pagetable não é a tabela de páginas atual, copyinstr usa walkaddr (que chama walk) para procurar srcva na pagetable, retornando o endereço físico pa0. O kernel mapeia cada endereço físico de RAM para o endereço virtual do kernel correspondente, de modo que copyinstr pode copiar diretamente bytes de string de pa0 para dst. walkaddr (kernel/vm.c:109) verifica se o endereço virtual fornecido pelo usuário faz parte do espaço de endereço do usuário do processo, para que os programas

não consegue enganar o kernel para que ele leia outra memória. Uma função semelhante, copyout, copia dados do kernel para um endereço fornecido pelo usuário.

4.5 Armadilhas do espaço do kernel

O Xv6 configura os registradores de trap da CPU de forma um pouco diferente, dependendo se o código do usuário ou do kernel está em execução. Quando o kernel está em execução em uma CPU, o kernel aponta stvec para o código assembly em kernelvec (kernel/kernelvec.S:12). Como o xv6 já está no kernel, o kernelvec pode contar com satp sendo definido na tabela de páginas do kernel e com o ponteiro da pilha se referindo a uma pilha do kernel válida. O kernelvec envia todos os 32 registradores para a pilha, de onde ele os restaurará posteriormente para que o código do kernel interrompido possa ser retomado sem perturbações.

kernelvec salva os registradores na pilha da thread do kernel interrompida, o que faz sentido porque os valores dos registradores pertencem a essa thread. Isso é particularmente importante se a armadilha causar uma troca para uma thread diferente – nesse caso, a armadilha retornará da pilha da nova thread, deixando os registradores salvos da thread interrompida em segurança em sua pilha.

kernelvec salta para kerneltrap (kernel/trap.c:135) Após salvar os registradores, o kerneltrap está preparado para dois tipos de armadilhas: interrupções de dispositivo e exceções. Ele chama devintr (kernel/ -trap.c:178) para verificar e lidar com o primeiro. Se a armadilha não for uma interrupção de dispositivo, deve ser uma exceção, e isso é sempre um erro fatal se ocorrer no kernel xv6; o kernel dispara pânico e interrompe a execução.

Se o kerneltrap foi chamado devido a uma interrupção do temporizador e uma thread do kernel de um processo está em execução (em oposição a uma thread do escalonador), o kerneltrap chama o método yield para dar a outras threads a chance de executar. Em algum momento, uma dessas threads irá executar o método yield, permitindo que nossa thread e sua kerneltrap sejam retomadas. O Capítulo 7 explica o que acontece no método yield.

Quando o trabalho do kerneltrap termina, ele precisa retornar ao código que foi interrompido pela armadilha. Como um yield pode ter perturbado o sepc e o modo anterior em sstatus, o kerneltrap os salva ao iniciar. Agora, ele restaura esses registradores de controle e retorna ao kernelvec (kernel/kernelvec.S:50). kernelvec retira os registradores salvos da pilha e executa sret, que copia sepc para pc e retoma o código do kernel interrompido.

Vale a pena pensar em como o retorno do trap acontece se o kerneltrap chamar yield devido a uma interrupção do temporizador.

O Xv6 define o stvec de uma CPU como kernelvec quando essa CPU entra no kernel pelo espaço do usuário; você pode ver isso em usertrap (kernel/trap.c:29). Há uma janela de tempo em que o kernel inicia a execução, mas stvec ainda está definido como uservec, e é crucial que nenhuma interrupção de dispositivo ocorra durante essa janela. Felizmente, o RISC-V sempre desabilita as interrupções quando começa a receber uma captura, e o xv6 não as habilita novamente até que stvec seja definido.

4.6 Exceções de falha de página

A resposta do Xv6 a exceções é bastante tediosa: se uma exceção acontece no espaço do usuário, o kernel encerra o processo com falha. Se uma exceção acontece no kernel, o kernel entra em pânico. Operação real

os sistemas geralmente respondem de maneiras muito mais interessantes.

Como exemplo, muitos kernels usam falhas de página para implementar bifurcações de cópia na escrita (COW). Para explicar a bifurcação de cópia na escrita, considere a bifurcação do xv6, descrita no Capítulo 3. A bifurcação faz com que o conteúdo de memória inicial do filho seja o mesmo do pai no momento da bifurcação. O Xv6 implementa a bifurcação com uvmcopy (kernel/vm.c:306). que aloca memória física para o filho e copia a memória do pai para ela. Seria mais eficiente se o filho e o pai pudessem compartilhar a memória física do pai. Uma implementação direta disso não funcionaria, no entanto, pois faria com que o pai e o filho interrompessem a execução um do outro com suas gravações na pilha e no heap compartilhados.

Pai e filho podem compartilhar memória física com segurança por meio do uso apropriado de permissões de tabela de páginas e falhas de página. A CPU gera uma exceção de falha de página quando um endereço virtual é usado sem mapeamento na tabela de páginas, ou tem um mapeamento cujo sinalizador PTE_V está limpo, ou um mapeamento cujos bits de permissão (PTE_R, PTE_W, PTE_X, PTE_U) proíbem a operação que está sendo tentada. O RISC-V distingue três tipos de falha de página: falhas de página de carregamento (quando uma instrução de carregamento não consegue traduzir seu endereço virtual), falhas de página de armazenamento (quando uma instrução de armazenamento não consegue traduzir seu endereço virtual) e falhas de página de instrução (quando o endereço no contador de programa não traduz). O registrador scause indica o tipo de falha de página e o registrador stval contém o endereço que não pôde ser traduzido.

O plano básico em uma bifurcação COW é que o pai e o filho compartilhem inicialmente todas as páginas físicas, mas cada um mapeie-as como somente leitura (com o sinalizador PTE_W limpo). Pai e filho podem ler a partir da memória física compartilhada. Se qualquer um deles gravar uma determinada página, a CPU RISC-V gera uma exceção de falha de página. O manipulador de traps do kernel responde alocando uma nova página de memória física e copiando para ela a página física para a qual o endereço com falha mapeia. O kernel altera a PTE relevante na tabela de páginas do processo com falha para apontar para a cópia e permitir gravações, bem como leituras, e então retoma o processo com falha na instrução que causou a falha. Como a PTE permite gravações, a instrução reexecutada agora será executada sem falhas. A cópia na gravação requer contabilidade para ajudar a decidir quando as páginas físicas podem ser liberadas, uma vez que cada página pode ser referenciada por um número variável de tabelas de páginas, dependendo do histórico de bifurcações, falhas de página, execuções e saídas.

Essa contabilidade permite uma otimização importante: se um processo incorrer em uma falha de página de armazenamento e a página física for referenciada apenas pela tabela de páginas desse processo, nenhuma cópia será necessária.

A cópia na gravação torna a bifurcação mais rápida, já que a bifurcação não precisa copiar memória. Parte da memória precisará ser copiada posteriormente, quando escrita, mas frequentemente a maior parte da memória nunca precisa ser copiada. Um exemplo comum é a bifurcação seguida por um comando exec: algumas páginas podem ser escritas após a bifurcação, mas o comando exec da filha libera a maior parte da memória herdada da mãe.
A bifurcação copy-on-write elimina a necessidade de copiar essa memória. Além disso, a bifurcação COW é transparente: nenhuma modificação nos aplicativos é necessária para que eles se beneficiem.

A combinação de tabelas de páginas e falhas de página abre uma ampla gama de possibilidades interessantes, além da bifurcação COW. Outro recurso amplamente utilizado é a alocação preguiçosa, que possui duas partes. Primeiro, quando um aplicativo solicita mais memória chamando sbrk, o kernel percebe o aumento de tamanho, mas não aloca memória física e não cria PTEs para o novo intervalo de endereços virtuais. Segundo, em uma falha de página em um desses novos endereços, o kernel aloca uma página de memória física e a mapeia na tabela de páginas. Assim como a bifurcação COW, o kernel pode implementar

alocação preguiçosa de forma transparente para aplicativos.

Como os aplicativos frequentemente solicitam mais memória do que precisam, a alocação preguiçosa é vantajosa: o kernel não precisa realizar nenhum trabalho para páginas que o aplicativo nunca usa. Além disso, se o aplicativo solicitar um aumento significativo no espaço de endereço, o sbrk sem alocação preguiçosa é caro: se um aplicativo solicitar um gigabyte de memória, o kernel precisa alocar e zerar 262.144 páginas de 4.096 bytes. A alocação preguiçosa permite que esse custo seja distribuído ao longo do tempo. Por outro lado, a alocação preguiçosa incorre na sobrecarga extra de falhas de página, que envolvem uma transição kernel/usuário. Os sistemas operacionais podem reduzir esse custo alocando um lote de páginas consecutivas por falha de página em vez de uma página e especializando o código de entrada/saída do kernel para tais falhas de página.

Outro recurso amplamente utilizado que explora falhas de página é a paginação sob demanda. Em exec, o xv6 carrega todo o texto e os dados de um aplicativo avidamente na memória. Como os aplicativos podem ser grandes e a leitura do disco é cara, esse custo inicial pode ser perceptível para os usuários: quando o usuário inicia um aplicativo grande a partir do shell, pode levar muito tempo até que ele veja uma resposta. Para melhorar o tempo de resposta, um kernel moderno cria a tabela de páginas para o espaço de endereço do usuário, mas marca as PTEs das páginas como inválidas. Em uma falha de página, o kernel lê o conteúdo da página do disco e o mapeia para o espaço de endereço do usuário. Assim como o COW fork e a alocação preguiçosa, o kernel pode implementar esse recurso de forma transparente para os aplicativos.

Os programas em execução em um computador podem precisar de mais memória do que a RAM disponível no computador. Para lidar com a situação de forma elegante, o sistema operacional pode implementar a paginação em disco. A ideia é armazenar apenas uma fração das páginas do usuário na RAM e armazenar o restante em disco, em uma área de paginação. O kernel marca as PTEs que correspondem à memória armazenada na área de paginação (e, portanto, não na RAM) como inválidas. Se um aplicativo tentar usar uma das páginas que foram paginadas para o disco, o aplicativo incorrerá em uma falha de página e a página deverá ser paginada: o manipulador de traps do kernel alocará uma página de RAM física, lerá a página do disco para a RAM e modificará a PTE relevante para apontar para a RAM.

O que acontece se uma página precisar ser paginada, mas não houver RAM física livre? Nesse caso, o kernel deve primeiro liberar uma página física, paginando-a para fora ou removendo-a para a área de paginação no disco e marcando as PTEs que se referem a essa página física como inválidas. A remoção é dispendiosa, portanto, a paginação tem melhor desempenho se for pouco frequente: se os aplicativos usarem apenas um subconjunto de suas páginas de memória e a união dos subconjuntos couber na RAM. Essa propriedade é frequentemente chamada de boa localidade de referência. Assim como acontece com muitas técnicas de memória virtual, os kernels geralmente implementam a paginação para o disco de forma transparente para os aplicativos.

Os computadores frequentemente operam com pouca ou nenhuma memória física livre, independentemente da quantidade de RAM fornecida pelo hardware. Por exemplo, provedores de nuvem multiplexam muitos clientes em uma única máquina para usar seu hardware de forma econômica. Outro exemplo: usuários executam muitos aplicativos em smartphones com uma pequena quantidade de memória física. Em tais cenários, a alocação de uma página pode exigir a remoção prévia de uma página existente. Portanto, quando a memória física livre é escassa, a alocação é dispendiosa.

A alocação lenta e a paginação sob demanda são particularmente vantajosas quando a memória livre é escassa.
A alocação avida de memória em sbrk ou exec incorre no custo extra de remoção para disponibilizar memória. Além disso, existe o risco de que o trabalho avidamente realizado seja desperdiçado, pois antes que o aplicativo use a página, o sistema operacional pode tê-la removido.

Outros recursos que combinam exceções de paginação e falha de página incluem extensão automática
pilhas e arquivos mapeados na memória.

4.7 Mundo real

O trampolim e o trapframe podem parecer excessivamente complexos. Uma força motriz é que o RISC-V intencionalmente faz o mínimo possível ao forçar uma armadilha, para permitir a possibilidade de um tratamento muito rápido da armadilha, o que acaba sendo importante. Como resultado, as primeiras instruções do manipulador de armadilhas do kernel precisam ser executadas efetivamente no ambiente do usuário: a tabela de páginas do usuário e o conteúdo do registrador do usuário. E o manipulador de armadilhas inicialmente ignora fatos úteis, como a identidade do processo em execução ou o endereço da tabela de páginas do kernel. Uma solução é possível porque o RISC-V fornece locais protegidos nos quais o kernel pode armazenar informações antes de entrar no espaço do usuário: o registrador sscratch e as entradas da tabela de páginas do usuário que apontam para a memória do kernel, mas são protegidas pela ausência de PTE_U. O trampolim e o trapframe do Xv6 exploram esses recursos do RISC-V.

A necessidade de páginas especiais de trampolim poderia ser eliminada se a memória do kernel fosse mapeada na tabela de páginas de usuário de cada processo (com os sinalizadores de permissão PTE apropriados). Isso também eliminaria a necessidade de uma troca de tabela de páginas ao capturar do espaço do usuário para o kernel. Isso, por sua vez, permitiria que implementações de chamadas de sistema no kernel aproveitassem o mapeamento da memória de usuário do processo atual, permitindo que o código do kernel desreferenciasse diretamente os ponteiros do usuário. Muitos sistemas operacionais têm usado essas ideias para aumentar a eficiência. O Xv6 as evita para reduzir as chances de bugs de segurança no kernel devido ao uso inadvertido de ponteiros de usuário e para reduzir um pouco da complexidade necessária para garantir que os endereços virtuais do usuário e do kernel não se sobreponham.

Sistemas operacionais de produção implementam bifurcação de cópia na gravação, alocação preguiçosa, paginação sob demanda, paginação em disco, arquivos mapeados na memória, etc. Além disso, sistemas operacionais de produção tentarão usar toda a memória física, seja para aplicativos ou caches (por exemplo, o cache de buffer do sistema de arquivos, que abordaremos mais adiante na Seção 8.2). O Xv6 é ingênuo nesse aspecto: você quer que seu sistema operacional use a memória física pela qual você pagou, mas o xv6 não. Além disso, se o xv6 ficar sem memória, ele retorna um erro para o aplicativo em execução ou o encerra, em vez de, por exemplo, remover uma página de outro aplicativo.

4.8 Exercícios

1. As funções copyin e copyinstr percorrem a tabela de páginas do usuário no software. Configure a tabela de páginas do kernel de forma que o kernel tenha o programa do usuário mapeado, e copyin e copyinstr possam usar memcpy para copiar argumentos de chamada de sistema para o espaço do kernel, contando com o hardware para percorrer a tabela de páginas.

2. Implemente alocação de memória preguiçosa.

3. Implemente o fork COW.

4. Existe uma maneira de eliminar o mapeamento especial de páginas TRAPFRAME em cada espaço de endereço de usuário? Por exemplo, o uservec poderia ser modificado para simplesmente inserir os 32 registradores de usuário na pilha do kernel ou armazená-los na estrutura proc?

5. O xv6 poderia ser modificado para eliminar o mapeamento especial da página TRAMPOLINE?

Capítulo 5

Interrupções e drivers de dispositivo

Um driver é o código em um sistema operacional que gerencia um dispositivo específico: ele configura o hardware do dispositivo, instrui o dispositivo a executar operações, lida com as interrupções resultantes e interage com processos que podem estar aguardando E/S do dispositivo. O código do driver pode ser complicado, pois ele é executado simultaneamente com o dispositivo que gerencia. Além disso, o driver precisa entender a interface de hardware do dispositivo, que pode ser complexa e mal documentada.

Dispositivos que precisam da atenção do sistema operacional geralmente podem ser configurados para gerar interrupções, que são um tipo de armadilha. O código de tratamento de armadilhas do kernel reconhece quando um dispositivo gera uma interrupção e chama o manipulador de interrupções do driver; no xv6, esse despacho ocorre em devintr (kernel/trap.c:178).

Muitos drivers de dispositivo executam código em dois contextos: uma metade superior, que é executada na thread do kernel de um processo, e uma metade inferior, que é executada em tempo de interrupção. A metade superior é chamada por meio de chamadas de sistema, como leitura e gravação, que exigem que o dispositivo execute E/S. Esse código pode solicitar ao hardware que inicie uma operação (por exemplo, solicitar ao disco que leia um bloco); em seguida, o código aguarda a conclusão da operação. Por fim, o dispositivo conclui a operação e gera uma interrupção. O manipulador de interrupção do driver, atuando como a metade inferior, descobre qual operação foi concluída, ativa um processo em espera, se apropriado, e informa ao hardware para iniciar a execução de qualquer próxima operação em espera.

5.1 Código: Entrada do console

O driver do console (kernel/console.c) é uma ilustração simples da estrutura do driver. O driver do console aceita caracteres digitados por um humano, por meio do hardware de porta serial UART conectado ao RISC-V.
O driver do console acumula uma linha de entrada por vez, processando caracteres de entrada especiais, como backspace e control-u. Processos do usuário, como o shell, usam a chamada de sistema read para buscar linhas de entrada do console. Quando você digita uma entrada para o xv6 no QEMU, suas teclas digitadas são enviadas ao xv6 por meio do hardware UART simulado do QEMU.

O hardware UART com o qual o driver se comunica é um chip 16550 [13] emulado pelo QEMU. Em um computador real, um 16550 gerenciaria um link serial RS232 conectado a um terminal ou outro computador.
Ao executar o QEMU, ele é conectado ao seu teclado e monitor.

O hardware UART parece, para o software, um conjunto de registradores de controle mapeados na memória.

Ou seja, existem alguns endereços físicos que o hardware RISC-V conecta ao dispositivo UART, de modo que cargas e armazenamentos interagem com o hardware do dispositivo em vez da RAM. Os endereços mapeados na memória para a UART começam em 0x10000000, ou UART0 (kernel/memlayout.h:21). Existem vários registradores de controle UART, cada um com a largura de um byte. Seus deslocamentos em relação à UART0 são definidos em (kernel/uart.c:22). Por exemplo, o registrador LSR contém bits que indicam se os caracteres de entrada estão aguardando para serem lidos pelo software. Esses caracteres (se houver) estão disponíveis para leitura no registrador RHR. Cada vez que um caractere é lido, o hardware da UART o exclui de um FIFO interno de caracteres em espera e limpa o bit "pronto" no LSR quando o FIFO está vazio. O hardware de transmissão da UART é amplamente independente do hardware de recepção; se o software escreve um byte no THR, a UART transmite esse byte.

As principais chamadas do Xv6 consoleinit (kernel/console.c:182) para inicializar o hardware da UART. Este código configura a UART para gerar uma interrupção de recebimento quando a UART recebe cada byte de entrada e uma interrupção de transmissão completa sempre que a UART termina de enviar um byte de saída (kernel/uart.c:53).

O shell xv6 lê o console por meio de um descritor de arquivo aberto por init.c (usuário/init.c:19). As chamadas para o sistema read passam pelo kernel até o consoleread (kernel/ console.c:80). consoleread aguarda a chegada da entrada (por meio de interrupções) e seu armazenamento em buffer em cons.buf, copia a entrada para o espaço do usuário e (após a chegada de uma linha inteira) retorna ao processo do usuário. Se o usuário ainda não tiver digitado uma linha inteira, qualquer processo de leitura aguardará na chamada sleep (kernel/ console.c:96). (O Capítulo 7 explica os detalhes do sono).

Quando o usuário digita um caractere, o hardware UART solicita ao RISC-V que lance uma interrupção, que ativa o manipulador de traps do xv6. O manipulador de traps chama devintr (kernel/trap.c:178). que analisa o registrador RISC-V Scause para descobrir se a interrupção é de um dispositivo externo. Em seguida, solicita a uma unidade de hardware chamada PLIC [3] que informe qual dispositivo interrompeu (kernel/trap.c:187). Se fosse o UART, devintr chamaria uartintr.

uartintr (kernel/uart.c:176) lê quaisquer caracteres de entrada em espera do hardware UART e os entrega ao consoleintr (kernel/console.c:136); Ele não espera por caracteres, pois entradas futuras gerarão uma nova interrupção. A função do consoleintr é acumular caracteres de entrada em cons.buf até que uma linha inteira chegue. O consoleintr trata o backspace e alguns outros caracteres de forma especial. Quando uma nova linha chega, o consoleintr desperta um consoleread em espera (se houver).

Uma vez ativado, o consoleread observará uma linha completa em cons.buf, a copiará para o espaço do usuário e retornará (por meio do mecanismo de chamada do sistema) para o espaço do usuário.

5.2 Código: Saída do console

Uma chamada de sistema de gravação em um descritor de arquivo conectado ao console eventualmente chega em uartputc (kernel/uart.c:87). O driver do dispositivo mantém um buffer de saída (uart_tx_buf) para que os processos de escrita não precisem esperar o UART terminar de enviar; em vez disso, o uartputc anexa cada caractere ao buffer, chama o uartstart para iniciar a transmissão do dispositivo (se ainda não estiver) e retorna. A única situação em que o uartputc aguarda é se o buffer já estiver cheio.

Cada vez que a UART termina de enviar um byte, ela gera uma interrupção. uartintr chama uartstart,

que verifica se o dispositivo realmente concluiu o envio e entrega ao dispositivo o próximo caractere de saída armazenado em buffer. Assim, se um processo gravar vários bytes no console, normalmente o primeiro byte será enviado pela chamada de uartputc para uartstart, e os bytes restantes armazenados em buffer serão enviados por chamadas de uartstart de uartintr à medida que as interrupções de transmissão completa chegam.

Um padrão geral a ser observado é o desacoplamento da atividade do dispositivo da atividade do processo por meio de buffer e interrupções. O driver do console pode processar a entrada mesmo quando nenhum processo está aguardando para lê-la; uma leitura subsequente verá a entrada. Da mesma forma, os processos podem enviar a saída sem precisar esperar pelo dispositivo. Esse desacoplamento pode aumentar o desempenho, permitindo que os processos sejam executados simultaneamente com a E/S do dispositivo, e é particularmente importante quando o dispositivo está lento (como no caso da UART) ou precisa de atenção imediata (como no caso de caracteres digitados ecoando). Essa ideia às vezes é chamada de E/S. simultaneidade.

5.3 Concorrência em drivers

Você deve ter notado chamadas para acquire em consoleread e em consoleintr. Essas chamadas adquirem um bloqueio, que protege as estruturas de dados do driver de console contra acesso concorrente. Há três perigos de simultaneidade aqui: dois processos em CPUs diferentes podem chamar consoleread ao mesmo tempo; o hardware pode solicitar que uma CPU envie uma interrupção de console (na verdade, UART) enquanto essa CPU já estiver em execução dentro de consoleread; e o hardware pode enviar uma interrupção de console em uma CPU diferente enquanto consoleread estiver em execução. Esses perigos podem resultar em corridas ou deadlocks. O Capítulo 6 explora esses problemas e como os bloqueios podem resolvê-los.

Outra maneira pela qual a concorrência requer cuidado nos drivers é que um processo pode estar aguardando a entrada de um dispositivo, mas a chegada da sinalização de interrupção da entrada pode ocorrer quando um processo diferente (ou nenhum processo) estiver em execução. Assim, os manipuladores de interrupção não podem pensar sobre o processo ou código que interromperam. Por exemplo, um manipulador de interrupção não pode chamar copyout com segurança com a tabela de páginas do processo atual. Os manipuladores de interrupção normalmente realizam relativamente pouco trabalho (por exemplo, apenas copiam os dados de entrada para um buffer) e ativam o código da metade superior para fazer o resto.

5.4 Interrupções do temporizador

O Xv6 utiliza interrupções de temporizador para manter seu relógio e permitir a alternância entre processos computacionais; as chamadas yield em usertrap e kerneltrap causam essa alternância. As interrupções de temporizador vêm do hardware de relógio conectado a cada CPU RISC-V. O Xv6 programa esse hardware de relógio para interromper cada CPU periodicamente.

O RISC-V exige que as interrupções de timer sejam executadas no modo máquina, não no modo supervisor. O modo máquina RISC-V executa sem paginação e com um conjunto separado de registradores de controle, portanto, não é prático executar código kernel xv6 comum no modo máquina. Como resultado, o xv6 lida com interrupções de timer completamente separadamente do mecanismo de captura descrito acima.

O código executado no modo de máquina em start.c, antes do main, é configurado para receber interrupções do temporizador (kernel/start.c:63). Parte do trabalho consiste em programar o hardware CLINT (interruptor local do núcleo) para gerar uma interrupção após um determinado atraso. Outra parte consiste em configurar uma área de risco, análoga à

trapframe, para ajudar o manipulador de interrupção do temporizador a salvar registradores e o endereço dos registradores CLINT. Por fim, start define mtvec como timervec e habilita interrupções de timer.

Uma interrupção de temporizador pode ocorrer a qualquer momento durante a execução de código do usuário ou do kernel; não há como o kernel desabilitar interrupções de temporizador durante operações críticas. Portanto, o manipulador de interrupções de temporizador deve executar sua função de forma a garantir que não perturbe o código do kernel interrompido. A estratégia básica é que o manipulador solicite ao RISC-V que gere uma "interrupção de software" e retorne imediatamente. O RISC-V entrega interrupções de software ao kernel com o mecanismo de captura comum e permite que o kernel as desabilite. O código para lidar com a interrupção de software gerada por uma interrupção de temporizador pode ser visto em devintr (kernel/trap.c:205).

O manipulador de interrupção do temporizador no modo máquina é timervec (kernel/kernelvec.S:95). Ele salva alguns registradores na área de rascunho preparada por start, informa ao CLINT quando gerar a próxima interrupção do temporizador, solicita ao RISC-V que lance uma interrupção de software, restaura os registradores e retorna. Não há código C no manipulador de interrupção do temporizador.

5.5 Mundo real

O Xv6 permite interrupções de dispositivo e temporizador durante a execução no kernel, bem como durante a execução de programas do usuário. As interrupções de temporizador forçam uma troca de thread (uma chamada para yield) a partir do manipulador de interrupções de temporizador, mesmo durante a execução no kernel. A capacidade de dividir o tempo da CPU de forma justa entre as threads do kernel é útil se as threads do kernel às vezes gastam muito tempo computando, sem retornar ao espaço do usuário. No entanto, a necessidade de o código do kernel estar ciente de que pode ser suspenso (devido a uma interrupção de temporizador) e posteriormente retomado em uma CPU diferente é a fonte de alguma complexidade no xv6 (consulte a Seção 6.6). O kernel poderia ser um pouco mais simples se as interrupções de dispositivo e temporizador ocorressem apenas durante a execução do código do usuário.

Oferecer suporte a todos os dispositivos em um computador típico em todo o seu esplendor dá muito trabalho, pois há muitos dispositivos, os dispositivos têm muitos recursos e o protocolo entre o dispositivo e o driver pode ser complexo e mal documentado. Em muitos sistemas operacionais, os drivers ocupam mais código do que o kernel principal.

O driver UART recupera dados um byte por vez, lendo os registradores de controle UART; esse padrão é chamado de E/S programada, pois o software controla a movimentação dos dados. A E/S programada é simples, mas lenta demais para ser usada em altas taxas de dados. Dispositivos que precisam mover grandes quantidades de dados em alta velocidade normalmente usam acesso direto à memória (DMA). O hardware do dispositivo DMA grava os dados de entrada diretamente na RAM e lê os dados de saída da RAM. Dispositivos de disco e rede modernos usam DMA.
Um driver para um dispositivo DMA prepararia dados na RAM e, então, usaria uma única gravação em um registro de controle para dizer ao dispositivo para processar os dados preparados.

Interrupções fazem sentido quando um dispositivo precisa de atenção em momentos imprevisíveis e não com muita frequência. Mas as interrupções têm alta sobrecarga de CPU. Assim, dispositivos de alta velocidade, como controladores de rede e de disco, usam truques que reduzem a necessidade de interrupções. Um truque é gerar uma única interrupção para um lote inteiro de solicitações de entrada ou saída. Outro truque é o driver desabilitar completamente as interrupções e verificar o dispositivo periodicamente para ver se ele precisa de atenção. Essa técnica é chamada de polling. O polling faz sentido se o dispositivo executa operações muito rapidamente, mas desperdiça tempo de CPU se o dispositivo estiver ocioso na maior parte do tempo. Alguns drivers alternam dinamicamente entre polling e interrupções.

dependendo da carga atual do dispositivo.

O driver UART copia os dados recebidos primeiro para um buffer no kernel e depois para o espaço do usuário. Isso faz sentido em baixas taxas de dados, mas essa cópia dupla pode reduzir significativamente o desempenho de dispositivos que geram ou consomem dados muito rapidamente. Alguns sistemas operacionais conseguem mover dados diretamente entre os buffers do espaço do usuário e o hardware do dispositivo, geralmente com DMA.

Conforme mencionado no Capítulo 1, o console aparece para os aplicativos como um arquivo comum, e os aplicativos leem a entrada e escrevem a saída usando as chamadas de sistema de leitura e escrita . Os aplicativos podem querer controlar aspectos de um dispositivo que não podem ser expressos pelas chamadas de sistema de arquivos padrão (por exemplo, habilitar/desabilitar o buffer de linha no driver do console). Os sistemas operacionais Unix suportam a chamada de sistema ioctl para esses casos.

Alguns usos de computadores exigem que o sistema responda em um tempo limitado. Por exemplo, em sistemas críticos de segurança, perder um prazo pode levar a desastres. O Xv6 não é adequado para configurações de tempo real rígido. Sistemas operacionais para tempo real rígido tendem a usar bibliotecas que se conectam ao aplicativo de forma a permitir uma análise para determinar o pior tempo de resposta. O Xv6 também não é adequado para aplicativos de tempo real flexível, quando perder um prazo ocasionalmente é aceitável, porque o escalonador do Xv6 é muito simplista e possui um caminho de código do kernel onde as interrupções são desabilitadas por um longo período.

5.6 Exercícios

1. Modifique uart.c para não usar interrupções. Talvez seja necessário modificar console.c também.
2. Adicione um driver para uma placa Ethernet.

Capítulo 6

Bloqueio

A maioria dos kernels, incluindo o xv6, intercala a execução de múltiplas atividades. Uma fonte de intercalação é o hardware multiprocessador: computadores com múltiplas CPUs executando independentemente, como o RISC-V do xv6. Essas múltiplas CPUs compartilham RAM física, e o xv6 explora esse compartilhamento para manter estruturas de dados que todas as CPUs leem e gravam. Esse compartilhamento aumenta a possibilidade de uma CPU ler uma estrutura de dados enquanto outra CPU está no meio da atualização, ou mesmo múltiplas CPUs atualizando os mesmos dados simultaneamente; sem um projeto cuidadoso, esse acesso paralelo provavelmente produzirá resultados incorretos ou uma estrutura de dados quebrada. Mesmo em um uniprocessador, o kernel pode alternar a CPU entre várias threads, fazendo com que sua execução seja intercalada. Por fim, um manipulador de interrupção de dispositivo que modifica os mesmos dados como algum código interrompível pode danificar os dados se a interrupção ocorrer no momento errado. O termo simultaneidade se refere a situações em que múltiplos fluxos de instruções são intercalados, devido ao paralelismo do multiprocessador, troca de threads ou interrupções.

Os kernels estão repletos de dados acessados concorrentemente. Por exemplo, duas CPUs poderiam chamar kalloc simultaneamente, saindo simultaneamente do topo da lista de acessos livres. Os projetistas de kernel gostam de permitir muita concorrência, pois isso pode gerar maior desempenho por meio do paralelismo e maior capacidade de resposta. No entanto, como resultado, os projetistas de kernel precisam se convencer da correção, apesar dessa concorrência. Existem muitas maneiras de se chegar a um código correto, algumas mais fáceis de raciocinar do que outras. Estratégias que visam à correção sob concorrência, e abstrações que as suportam, são chamadas de técnicas de controle de concorrência.

O Xv6 utiliza diversas técnicas de controle de concorrência, dependendo da situação; muitas outras são possíveis. Este capítulo se concentra em uma técnica amplamente utilizada: o bloqueio. Um bloqueio proporciona exclusão mútua, garantindo que apenas uma CPU por vez possa manter o bloqueio. Se o programador associar um bloqueio a cada item de dados compartilhado, e o código sempre mantiver o bloqueio associado ao usar um item, então o item será usado por apenas uma CPU por vez. Nessa situação, dizemos que o bloqueio protege o item de dados. Embora os bloqueios sejam um mecanismo de controle de concorrência fácil de entender, a desvantagem dos bloqueios é que eles podem limitar o desempenho, pois serializam operações simultâneas.

O restante deste capítulo explica por que o xv6 precisa de bloqueios, como o xv6 os implementa e como os utiliza.

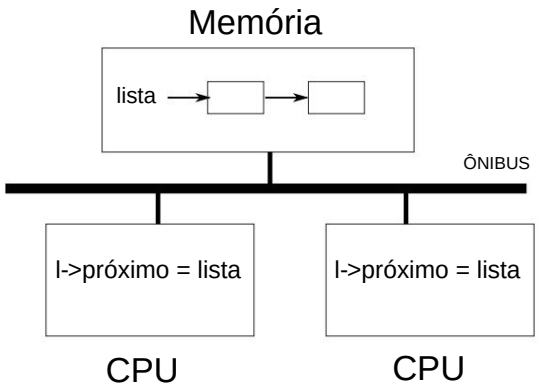

Figura 6.1: Arquitetura SMP simplificada

6.1 Corridas

Como exemplo de por que precisamos de bloqueios, considere dois processos com filhos que saíram chamando wait em duas CPUs diferentes. wait libera a memória da criança. Assim, em cada CPU, o kernel chamará kfree para liberar as páginas de memória das crianças. O alocador do kernel mantém uma lista encadeada: kalloc() (kernel /kalloc.c:69) extrai uma página de memória de uma lista de páginas livres e kfree() (kernel/kalloc.c:47) coloca uma página na lista de livres. Para melhor desempenho, podemos esperar que os kfrees da dois processos pais seriam executados em paralelo sem que nenhum deles tivesse que esperar pelo outro, mas isso não seria correto dada a implementação kfree do xv6.

A Figura 6.1 ilustra a configuração com mais detalhes: a lista vinculada de páginas livres está na memória que é compartilhado pelas duas CPUs, que manipulam a lista usando instruções de carga e armazenamento. (Na realidade, o os processadores têm caches, mas conceitualmente os sistemas multiprocessadores se comportam como se houvesse um único, memória compartilhada.) Se não houver solicitações simultâneas, você pode implementar uma operação de envio de lista do seguinte modo:

```
1   elemento struct {
2       dados int;
3       struct elemento *próximo;
4   };
5
6   struct elemento *lista = 0;
7
8   vazio
9   push(int dados)
10  {
11      struct elemento *l;
12
13      l = malloc(tamanho de *l);
14      l->dados = dados;
15      l->next = lista;
16      lista = l;
```

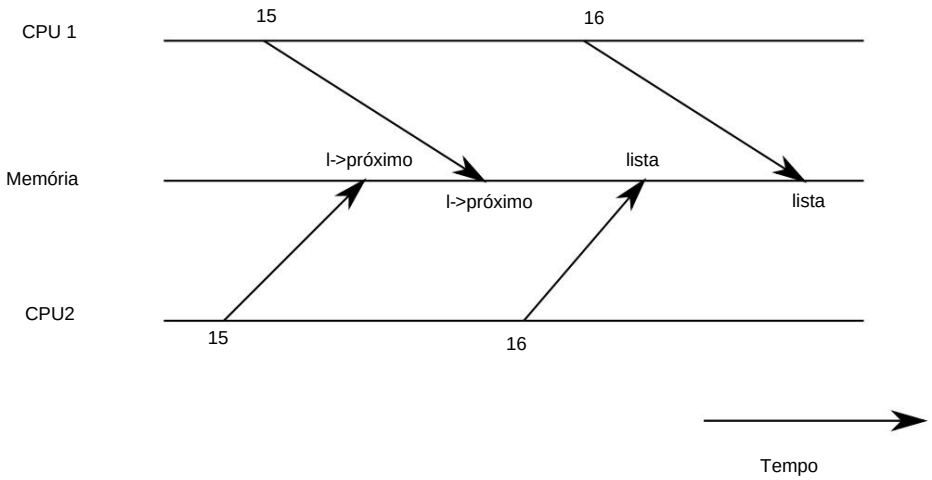

Figura 6.2: Exemplo de corrida

```
17          }
```

Esta implementação está correta se executada isoladamente. No entanto, o código não está correto se mais
mais de uma cópia é executada simultaneamente. Se duas CPUs executarem push ao mesmo tempo, ambas podem
execute a linha 15 conforme mostrado na Fig. 6.1, antes de executar a linha 16, o que resulta em um erro
resultado conforme ilustrado pela Figura 6.2. Haveria então dois elementos de lista com o próximo conjunto para o
valor anterior da lista. Quando as duas atribuições à lista acontecem na linha 16, a segunda será
substituir o primeiro; o elemento envolvido na primeira atribuição será perdido.

A atualização perdida na linha 16 é um exemplo de corrida. Uma corrida é uma situação em que uma memória
o local é acessado simultaneamente e pelo menos um acesso é de gravação. Uma corrida geralmente é um sinal de bug,
uma atualização perdida (se os acessos forem gravações) ou uma leitura de uma estrutura de dados atualizada de forma incompleta.
O resultado de uma corrida depende do código de máquina gerado pelo compilador, do tempo de
as duas CPUs envolvidas e como suas operações de memória são ordenadas pelo sistema de memória,
o que pode dificultar a reprodução e a depuração de erros induzidos por corrida. Por exemplo, adicionar print
declarações durante a depuração do push podem alterar o tempo de execução o suficiente para fazer o
raça desaparece.

A maneira usual de evitar corridas é usar um cadeado. Os cadeados garantem a exclusão mútua, de modo que apenas um
A CPU pode executar as linhas sensíveis do push por vez ; isso torna o cenário acima impossível.
A versão corretamente bloqueada do código acima adiciona apenas algumas linhas (destacadas em amarelo):

```
6           struct elemento *lista = 0;
7           struct lock lista de bloqueio;
8
            vazio
9 10        push(int dados)
11          {
12              struct elemento *l;
13              l = malloc(tamanho de *l);
14              l->dados = dados;
```

```
15
16          adquirir(&listlock); l->próximo =
17          lista; lista = l;
18          liberar(&listlock);
19
20      }
```

A sequência de instruções entre aquisição e liberação é frequentemente chamada de seção crítica.
Geralmente, diz-se que o bloqueio está protegendo a lista.

Quando dizemos que um bloqueio protege dados, na verdade queremos dizer que o bloqueio protege alguma coleção de invariantes que se aplicam aos dados. Invariantes são propriedades de estruturas de dados que são mantidas entre operações. Normalmente, o comportamento correto de uma operação depende de as invariantes serem verdadeiras quando a operação começa. A operação pode violar temporariamente as invariantes, mas deve restabelecê-las antes de terminar. Por exemplo, no caso da lista encadeada, a invariante é que a lista aponta para o primeiro elemento da lista e que o próximo campo de cada elemento aponta para o próximo elemento. A implementação de push viola essa invariante temporariamente: na linha 17, l aponta para o próximo elemento da lista, mas list ainda não aponta para l (restabelecido na linha 18). A corrida que examinamos acima ocorreu porque uma segunda CPU executou código que dependia das invariantes da lista enquanto elas estavam (temporariamente) violadas. O uso adequado de um bloqueio garante que apenas uma CPU por vez possa operar na estrutura de dados na seção crítica, de modo que nenhuma CPU executará uma operação na estrutura de dados quando as invariantes da estrutura de dados não forem válidas.

Você pode pensar em um bloqueio como a serialização de seções críticas simultâneas para que sejam executadas uma de cada vez, preservando assim as invariantes (assumindo que as seções críticas estejam corretas isoladamente). Você também pode pensar em seções críticas protegidas pelo mesmo bloqueio como atômicas entre si, de modo que cada uma veja apenas o conjunto completo de alterações das seções críticas anteriores e nunca veja atualizações parcialmente concluídas.

Embora úteis para a correção, os bloqueios limitam inerentemente o desempenho. Por exemplo, se dois processos chamarem kfree simultaneamente, os bloqueios serializarão as duas seções críticas, de modo que não há benefício em executá-los em CPUs diferentes. Dizemos que múltiplos processos entram em conflito se desejam o mesmo bloqueio ao mesmo tempo ou se o bloqueio sofre contenção. Um grande desafio no projeto do kernel é evitar a contenção de bloqueios em busca do paralelismo. O Xv6 faz pouco disso, mas kernels sofisticados organizam estruturas de dados e algoritmos especificamente para evitar a contenção de bloqueios.
No exemplo da lista, um kernel pode manter uma lista livre separada por CPU e somente tocar na lista livre de outra CPU se a lista da CPU atual estiver vazia e ele precisar roubar memória de outra CPU.
Outros casos de uso podem exigir designs mais complicados.

O posicionamento dos bloqueios também é importante para o desempenho. Por exemplo, seria correto mover a aquisição para o início do push, antes da linha 13. Mas isso provavelmente reduziria o desempenho, pois as chamadas para malloc seriam serializadas. A seção "Usando bloqueios" abaixo fornece algumas diretrizes sobre onde inserir as invocações acquire e release .

6.2 Código: Fechaduras

Xv6 possui dois tipos de travas: spinlocks e sleep-locks. Começaremos com spinlocks. Xv6 representa um spinlock como uma estrutura spinlock (kernel/spinlock.h:2). O campo importante na estrutura é bloqueado, uma palavra que é zero quando o bloqueio está disponível e diferente de zero quando é mantido. Logicamente, xv6 deve adquirir um bloqueio executando um código como

```
21  vazio
22  acquire(struct spinlock *lk) // não funciona!
23  {
24      para(;;) {
25          se(lk->bloqueado == 0) {
26              lk->bloqueado = 1;
27              quebrar;
28          }
29      }
30  }
```

Infelizmente, esta implementação não garante a exclusão mútua em um multiprocessador. pode acontecer que duas CPUs cheguem simultaneamente na linha 25, veja que lk->locked é zero, e então ambos pegam o bloqueio executando a linha 26. Neste ponto, duas CPUs diferentes seguram o bloqueio, o que viola a propriedade de exclusão mútua. O que precisamos é de uma maneira de fazer as linhas 25 e 26 serem executadas como uma etapa atômica (ou seja, indivisível).

Como os bloqueios são amplamente utilizados, os processadores multi-core geralmente fornecem instruções que implementam uma versão atômica das linhas 25 e 26. No RISC-V, esta instrução é amoswap r, a. amoswap lê o valor no endereço de memória a, escreve o conteúdo do registrador r nesse endereço, e coloca o valor lido em r. Ou seja, ele troca o conteúdo do registrador e da memória endereço. Ele executa essa sequência atomicamente, usando hardware especial para evitar qualquer outra CPU de usar o endereço de memória entre a leitura e a gravação.

Aquisição do Xv6 (kernel/spinlock.c:22) usa a chamada de biblioteca C portátil __sync_lock_test_and_set, que se resume à instrução amoswap ; o valor de retorno é o conteúdo antigo (trocado) de lk->bloqueado. A função acquire envolve a troca em um loop, tentando novamente (girando) até que tenha adquiriu o bloqueio. Cada iteração troca um para lk->locked e verifica o valor anterior; se o valor anterior é zero, então adquirimos o bloqueio e a troca terá definido lk->locked para um. Se o valor anterior for um, então alguma outra CPU detém o bloqueio, e o fato de que troquei um atomicamente para lk->locked e não alterei seu valor.

Depois que o bloqueio é adquirido, o acquire registra, para depuração, a CPU que adquiriu o bloqueio. O campo lk->cpu é protegido pelo bloqueio e só deve ser alterado enquanto o bloqueio estiver pressionado.

A função release (kernel/spinlock.c:47) é o oposto de adquirir: limpa o lk->cpu campo e então libera o bloqueio. Conceitualmente, a liberação requer apenas atribuir zero a lk->locked. O padrão C permite que os compiladores implementem uma atribuição com várias instruções de armazenamento, portanto, um A atribuição em C pode não ser atômica em relação ao código concorrente. Em vez disso, a liberação usa o C função de biblioteca __sync_lock_release que realiza uma atribuição atômica. Esta função também resume-se a uma instrução RISC-V amoswap .

6.3 Código: Usando bloqueios

O Xv6 usa travas em muitos lugares para evitar corridas. Como descrito acima, kalloc (kernel/kalloc.c:69) e kfree (kernel/kalloc.c:47) formam um bom exemplo. Experimente os Exercícios 1 e 2 para ver o que acontece se essas funções omitirem os bloqueios. Você provavelmente descobrirá que é difícil desencadear comportamentos incorretos, o que sugere que é difícil testar de forma confiável se o código está livre de erros de bloqueio e corridas. O Xv6 pode muito bem ter corridas ainda não descobertas.

Uma parte complexa do uso de bloqueios é decidir quantos bloqueios usar e quais dados e invariantes cada bloqueio deve proteger. Existem alguns princípios básicos. Primeiro, sempre que uma variável puder ser escrita por uma CPU ao mesmo tempo em que outra CPU pode lê-la ou escrevê-la, um bloqueio deve ser usado para evitar que as duas operações se sobreponham. Segundo, lembre-se de que os bloqueios protegem invariantes: se uma invariante envolve vários locais de memória, normalmente todos eles precisam ser protegidos por um único bloqueio para garantir que a invariante seja mantida.

As regras acima dizem quando os bloqueios são necessários, mas não dizem nada sobre quando eles são desnecessários, e é importante para a eficiência não bloquear demais, porque os bloqueios reduzem o paralelismo. Se o paralelismo não for importante, pode-se organizar para ter apenas uma única thread e não se preocupar com bloqueios. Um kernel simples pode fazer isso em um multiprocessador tendo um único bloqueio que deve ser adquirido ao entrar no kernel e liberado ao sair do kernel (embora bloquear chamadas de sistema, como leituras de pipe ou espera , representasse um problema). Muitos sistemas operacionais uniprocessadores foram convertidos para rodar em multiprocessadores usando essa abordagem, às vezes chamada de "bloqueio de kernel grande", mas a abordagem sacrifica o paralelismo: apenas uma CPU pode executar no kernel por vez. Se o kernel fizer alguma computação pesada, seria mais eficiente usar um conjunto maior de bloqueios mais refinados, para que o kernel pudesse executar em várias CPUs simultaneamente.

Como exemplo de bloqueio de granulação grossa, o alocador kalloc.c do xv6 possui uma única lista de páginas livres protegida por um único bloqueio. Se vários processos em CPUs diferentes tentarem alocar páginas simultaneamente, cada um terá que aguardar sua vez, girando em aquisição. Girar desperdiça tempo de CPU, pois não é um trabalho útil. Se a contenção pelo bloqueio desperdiçasse uma fração significativa do tempo de CPU, talvez o desempenho pudesse ser melhorado alterando o design do alocador para ter várias listas de páginas livres, cada uma com seu próprio bloqueio, para permitir uma alocação verdadeiramente paralela.

Como exemplo de bloqueio de granularidade fina, o xv6 possui um bloqueio separado para cada arquivo, de modo que processos que manipulam arquivos diferentes podem frequentemente prosseguir sem esperar pelos bloqueios uns dos outros. O esquema de bloqueio de arquivos poderia ser ainda mais granular se fosse necessário permitir que processos gravassem simultaneamente diferentes áreas do mesmo arquivo. Em última análise, as decisões sobre a granularidade do bloqueio precisam ser orientadas por medições de desempenho, bem como por considerações de complexidade.

À medida que os capítulos subsequentes explicam cada parte do xv6, eles mencionarão exemplos do uso de xv6 de bloqueios para lidar com a simultaneidade. Como prévia, a Figura 6.3 lista todos os bloqueios no xv6.

6.4 Deadlock e ordenação de bloqueio

Se um caminho de código no kernel precisar conter vários bloqueios simultaneamente, é importante que todos os caminhos de código adquiram esses bloqueios na mesma ordem. Caso contrário, há risco de deadlock. Digamos que dois caminhos de código no xv6 precisem dos bloqueios A e B, mas o caminho de código 1 adquire bloqueios na ordem A e B.

Trancar	Descrição
bcache.lock	Protege a alocação de entradas de cache do buffer de bloco
cons.lock	Serializa o acesso ao hardware do console, evita saídas misturadas
ftable.lock	Serializa a alocação de um arquivo struct na tabela de arquivos
itable.lock	Protege a alocação de entradas de inode na memória
vdisk_lock	Serializa o acesso ao hardware do disco e à fila de descritores DMA
kmem.lock	Serializa a alocação de memória
log.lock	Serializa operações no log de transações
pi->lock do pipe	Serializa operações em cada pipe
pid_lock	Serializa incrementos de next_pid
proc's p->lock	Serializa as alterações no estado do processo
wait_lock	Ajuda a evitar despertares perdidos
tickslock	Serializa operações no contador de ticks
ip->lock do inode	Serializa operações em cada inode e seu conteúdo
bloqueio de buf	Serializa as operações em cada buffer de bloco

Figura 6.3: Bloqueios em xv6

B, e o outro caminho os adquire na ordem B e depois A. Suponha que o thread T1 execute o caminho de código 1 e adquire o bloqueio A, e o thread T2 executa o caminho de código 2 e adquire o bloqueio B. O próximo T1 tentará adquirir o bloqueio B, e T2 tentará adquirir o bloqueio A. Ambos os adquirentes bloquearão indefinidamente, porque em em ambos os casos, a outra thread mantém o bloqueio necessário e não o liberará até que sua aquisição retorne. Para evitar tais impasses, todos os caminhos de código devem adquirir bloqueios na mesma ordem. A necessidade de uma estrutura global A ordem de aquisição de bloqueios significa que os bloqueios são efetivamente parte da especificação de cada função: chamadores deve invocar funções de forma que os bloqueios sejam adquiridos na ordem acordada.

O Xv6 possui muitas cadeias de ordem de bloqueio de comprimento dois envolvendo bloqueios por processo (o bloqueio em cada struct proc) devido à maneira como o sleep funciona (veja o Capítulo 7). Por exemplo, consoleintr (kernel/console.c:136) é a rotina de interrupção que lida com caracteres digitados. Quando uma nova linha chega, qualquer processo que esteja aguardando uma entrada do console deve ser ativado. Para fazer isso, consoleintr mantém cons.lock enquanto chama wakeup, que adquire o bloqueio do processo de espera para Acorde-o. Consequentemente, a ordem de bloqueio global para evitar deadlock inclui a regra de que cons.lock deve ser adquirido antes de qualquer bloqueio de processo. O código do sistema de arquivos contém as cadeias de bloqueio mais longas do xv6. Por exemplo, a criação de um arquivo requer a manutenção simultânea de um bloqueio no diretório e um bloqueio no diretório. inode do novo arquivo, um bloqueio em um buffer de bloco de disco, o vdisk_lock do driver de disco e o p->lock do processo de chamada . Para evitar deadlock, o código do sistema de arquivos sempre adquire bloqueios na ordem mencionada na frase anterior.

Cumprir uma ordem global de prevenção de impasses pode ser surpreendentemente difícil. Às vezes, o bloqueio conflitos de ordem com a estrutura lógica do programa, por exemplo, talvez o módulo de código M1 chame o módulo M2, mas a ordem de bloqueio exige que um bloqueio em M2 seja adquirido antes de um bloqueio em M1. Às vezes, as identidades de fechaduras não são conhecidas com antecedência, talvez porque uma fechadura deva ser mantida para descobrir a identidade do bloqueio a ser adquirido em seguida. Esse tipo de situação surge no sistema de arquivos quando ele procura componentes sucessivos em um nome de caminho e no código para esperar e sair enquanto pesquisam na tabela

de processos procurando por processos filhos. Por fim, o perigo de deadlock costuma ser uma restrição à precisão de um esquema de bloqueio, já que mais bloqueios geralmente significam mais oportunidades de deadlock. A necessidade de evitar deadlocks costuma ser um fator importante na implementação do kernel.

6.5 Eclusas reentrantes

Pode parecer que alguns deadlocks e desafios de ordenação de bloqueios poderiam ser evitados com o uso de bloqueios reentrantes, também chamados de bloqueios recursivos. A ideia é que, se o bloqueio for mantido por um processo e esse processo tentar obtê-lo novamente, o kernel poderia simplesmente permitir isso (já que o processo já possui o bloqueio), em vez de acionar o pânico, como faz o kernel xv6.

Acontece, no entanto, que bloqueios reentrantes dificultam o raciocínio sobre concorrência: eles quebram a intuição de que bloqueios fazem com que seções críticas sejam atômicas em relação a outras seções críticas. Considere as duas funções a seguir, f e g:

```
struct spinlock lock; int data = 0; //
protegido por bloqueio

f()
   { adquirir(&bloquear);
    se(dados == 0)
        { chamar_uma vez();
        h();
        dados = 1;

    } liberação(&bloqueio);
}

g()
    { adquirir(&lock); if(dados
    == 0){ chamar_uma
        vez(); dados = 1;


    } liberação(&bloqueio);
}
```

Olhando para este fragmento de código, a intuição é que call_once será chamado apenas uma vez: por f, ou por g, mas não por ambos.

Mas se bloqueios reentrantes forem permitidos, e h chamar g, call_once será chamado duas vezes.

Se bloqueios reentrantes não forem permitidos, então h chamando g resulta em um deadlock, o que também não é bom. Mas, supondo que chamar call_once seja um erro grave , um deadlock é preferível. O desenvolvedor do kernel observará o deadlock (o kernel entra em pânico) e pode corrigir o código para evitá-lo, enquanto chamar call_once duas vezes pode resultar silenciosamente em um erro difícil de rastrear.

Por esse motivo, o xv6 usa bloqueios não reentrantes, mais fáceis de entender. No entanto, desde que os programadores mantenham as regras de bloqueio em mente, qualquer uma das abordagens pode funcionar. Se o xv6 fosse...

Para usar bloqueios reentrantes, seria necessário modificar a aquisição para perceber que o bloqueio está atualmente sendo mantido pela thread que fez a chamada. Também seria necessário adicionar uma contagem de aquisições aninhadas à struct spinlock, de forma semelhante a push_off, que será discutido a seguir.

6.6 Bloqueios e manipuladores de interrupção

Alguns spinlocks xv6 protegem dados usados tanto por threads quanto por manipuladores de interrupção. Por exemplo, o manipulador de interrupção do timer clockintr pode incrementar ticks (kernel/trap.c:164). aproximadamente ao mesmo tempo em que um thread do kernel lê os ticks em sys_sleep (kernel/sysproc.c:59). O bloqueio tickslock serializa os dois acessos.

A interação de spinlocks e interrupções levanta um perigo potencial. Suponha que sys_sleep esteja com o tickslock bloqueado e sua CPU seja interrompida por uma interrupção de timer. clockintr tentaria obter o tickslock, verificar se ele estava bloqueado e aguardar sua liberação. Nessa situação, o tickslock nunca será liberado: somente sys_sleep pode liberá-lo, mas sys_sleep não continuará em execução até que clockintr retorne. Portanto, a CPU entrará em deadlock e qualquer código que precise de qualquer um dos bloqueios também travará.

Para evitar essa situação, se um spinlock for usado por um manipulador de interrupção, uma CPU nunca deve manter esse bloqueio com interrupções habilitadas. O Xv6 é mais conservador: quando uma CPU adquire qualquer bloqueio, o xv6 sempre desabilita as interrupções nessa CPU. Interrupções ainda podem ocorrer em outras CPUs, portanto, a aquisição de uma interrupção pode esperar que uma thread libere um spinlock; mas não na mesma CPU.

O Xv6 reativa interrupções quando uma CPU não possui spinlocks; ele deve fazer uma pequena contabilidade para lidar com seções críticas aninhadas. acquire chama push_off (kernel/spinlock.c:89) e libera chamadas pop_off (kernel/spinlock.c:100) para rastrear o nível de aninhamento de bloqueios na CPU atual. Quando essa contagem chega a zero, pop_off restaura o estado de ativação de interrupção que existia no início da seção crítica mais externa. As funções intr_off e intr_on executam instruções RISC-V para ativar e desativar interrupções, respectivamente.

É importante adquirir a chamada push_off estritamente antes de definir lk->locked (kernel/spin- lock.c:28). Se as duas opções fossem invertidas, haveria uma breve janela de tempo em que o bloqueio seria mantido com as interrupções habilitadas, e uma interrupção com tempo de execução indesejável causaria um deadlock no sistema. Da mesma forma, é importante liberar a chamada pop_off somente após a liberação do bloqueio (kernel/spinlock.c:66).

6.7 Instruções e ordenação de memória

É natural pensar em programas sendo executados na ordem em que as instruções do código-fonte aparecem.
Este é um modelo mental razoável para código single-threaded, mas é incorreto quando múltiplas threads interagem através da memória compartilhada [2, 4]. Um motivo é que os compiladores emitem instruções de carregamento e armazenamento em ordens diferentes daquelas implícitas no código-fonte, podendo omiti-las completamente (por exemplo, armazenando dados em cache em registradores). Outro motivo é que a CPU pode executar instruções fora de ordem para aumentar o desempenho.
Por exemplo, uma CPU pode perceber que, em uma sequência serial de instruções, A e B não são dependentes uma da outra. A CPU pode iniciar a instrução B primeiro, seja porque suas entradas estão prontas antes das entradas de A, seja para sobrepor a execução de A e B.

Como exemplo do que poderia dar errado, neste código para push, seria um desastre se o
o compilador ou a CPU moveu o armazenamento correspondente à linha 4 para um ponto após o lançamento na linha 6:

```
1    l = malloc(sizeof *l); l->data = dados;
2    adquirir(&listlock); l-
3    >next = lista; lista = l;
4    liberar(&listlock);

5 6
```

Se tal reordenação ocorresse, haveria uma janela durante a qual outra CPU poderia adquirir o bloqueio e observar a lista atualizada, mas veria uma lista não inicializada->próximo.

A boa notícia é que compiladores e CPUs ajudam programadores simultâneos seguindo um conjunto de regras chamado modelo de memória e fornecendo alguns primitivos para ajudar os programadores a controlar a reordenação.

Para informar ao hardware e ao compilador para não reordenar, o xv6 usa __sync_synchronize() em ambos os processos de aquisição (kernel/spinlock.c:22) e liberar (kernel/spinlock.c:47). __sync_synchronize() é uma barreira de memória: ela informa ao compilador e à CPU para não reordenarem cargas ou armazenamentos através da barreira. As barreiras na aquisição e liberação do xv6 forçam a ordem em quase todos os casos em que isso importa, já que o xv6 usa bloqueios em torno de acessos a dados compartilhados. O Capítulo 9 discute algumas exceções.

6.8 Bloqueios de sono

Às vezes, o xv6 precisa manter um bloqueio por muito tempo. Por exemplo, o sistema de arquivos (Capítulo 8) mantém um arquivo bloqueado enquanto lê e grava seu conteúdo no disco, e essas operações de disco podem levar dezenas de milissegundos. Manter um bloqueio de spin por tanto tempo levaria a desperdício se outro processo quisesse adquiri-lo, já que o processo de aquisição consumiria CPU por um longo tempo enquanto girava. Outra desvantagem dos bloqueios de spin é que um processo não pode ceder a CPU enquanto mantém um bloqueio de spin; gostaríamos de fazer isso para que outros processos possam usar a CPU enquanto o processo com o bloqueio aguarda o disco.
Ceder enquanto mantém um spinlock é ilegal porque pode levar a um deadlock se uma segunda thread tentar adquirir o spinlock; como a aquisição não cede a CPU, a rotação da segunda thread pode impedir a primeira thread de executar e liberar o bloqueio. Ceder enquanto mantém um bloqueio também violaria o requisito de que as interrupções devem estar desativadas enquanto um spinlock é mantido. Portanto, gostaríamos de um tipo de bloqueio que ceda a CPU enquanto aguarda para adquirir e permita cedes (e interrupções) enquanto o bloqueio é mantido.

O Xv6 fornece tais bloqueios na forma de bloqueios de sono. acquiresleep (kernel/sleeplock.c:22) cede a CPU enquanto aguarda, usando técnicas que serão explicadas no Capítulo 7. Em um nível mais alto, um sleep-lock possui um campo bloqueado que é protegido por um spinlock, e a chamada de acquiresleep para sleep cede atomicamente a CPU e libera o spinlock. O resultado é que outras threads podem executar enquanto acquiresleep aguarda.

Como os bloqueios de suspensão deixam as interrupções habilitadas, eles não podem ser usados em manipuladores de interrupção. Como acquiresleep pode render a CPU, os bloqueios de suspensão não podem ser usados dentro de seções críticas de spinlock (embora spinlocks possam ser usados dentro de seções críticas de sleep-lock).

Os spin-locks são mais adequados para seções críticas curtas, pois esperar por eles desperdiça tempo de CPU; Os bloqueios de sono funcionam bem para operações demoradas.

6.9 Mundo real

Programar com travas continua desafiador, apesar de anos de pesquisa sobre primitivas de concorrência e paralelismo. Muitas vezes, é melhor ocultar travas em construções de nível superior, como filas sincronizadas, embora o xv6 não faça isso. Se você programa com travas, é aconselhável usar uma ferramenta que tente identificar corridas, pois é fácil perder uma invariante que exija uma trava.

A maioria dos sistemas operacionais suporta threads POSIX (Pthreads), que permitem que um processo de usuário tenha várias threads em execução simultaneamente em diferentes CPUs. Pthreads suportam bloqueios, barreiras, etc. em nível de usuário. Pthreads também permitem que um programador especifique opcionalmente que um bloqueio deve ser redefinido. participante.

O suporte a Pthreads no nível do usuário requer suporte do sistema operacional. Por exemplo, se uma pthread for bloqueada em uma chamada de sistema, outra pthread do mesmo processo poderá ser executada naquela CPU. Outro exemplo: se uma pthread altera o espaço de endereço do seu processo (por exemplo, mapeia ou desmapeia memória), o kernel deve providenciar para que outras CPUs que executam threads do mesmo processo atualizem suas tabelas de páginas de hardware para refletir a alteração no espaço de endereço.

É possível implementar bloqueios sem instruções atômicas [10], mas é caro e a maioria sistemas operacionais usam instruções atômicas.

Bloqueios podem ser caros se muitas CPUs tentarem adquirir o mesmo bloqueio ao mesmo tempo. Se uma CPU tiver um bloqueio armazenado em cache em seu cache local e outra CPU precisar adquiri-lo, a instrução atômica para atualizar a linha de cache que contém o bloqueio deve mover a linha do cache de uma CPU para o cache da outra CPU, e talvez invalidar quaisquer outras cópias da linha de cache. Buscar uma linha de cache do cache de outra CPU pode ser muito mais caro do que buscar uma linha de um cache local.

Para evitar os custos associados a bloqueios, muitos sistemas operacionais utilizam estruturas de dados e algoritmos livres de bloqueios [6, 12]. Por exemplo, é possível implementar uma lista encadeada como a apresentada no início do capítulo, que não requer bloqueios durante as buscas na lista e uma instrução atômica para inserir um item em uma lista. A programação livre de bloqueios é mais complicada, no entanto, do que programar bloqueios; por exemplo, é preciso se preocupar com a reordenação de instruções e memória. Programar com bloqueios já é difícil, portanto, o xv6 evita a complexidade adicional da programação livre de bloqueios.

6.10 Exercícios

1. Comente as chamadas para adquirir e liberar em kalloc (kernel/kalloc.c:69). Isso parece causar problemas para o código do kernel que chama kalloc; quais sintomas você espera ver? Ao executar o xv6, você vê esses sintomas? E ao executar testes de usuário? Se não vê problema, por que não? Veja se consegue provocar um problema inserindo loops fictícios na seção crítica de kalloc.

2. Suponha que você tenha comentado o bloqueio em kfree (após restaurar o bloqueio em kalloc). O que pode dar errado agora? A ausência de bloqueios em kfree é menos prejudicial do que em kalloc?

3. Se duas CPUs chamarem kalloc ao mesmo tempo, uma terá que esperar pela outra, o que prejudica o desempenho. Modifique kalloc.c para ter mais paralelismo, de modo que chamadas simultâneas a kalloc de diferentes CPUs possam prosseguir sem esperar uma pela outra.

4. Escreva um programa paralelo usando threads POSIX, que são suportadas pela maioria dos sistemas operacionais. Por exemplo, implemente uma tabela de hash paralela e meça se o número de puts/gets aumenta com o aumento do número de núcleos.

5. Implemente um subconjunto de Pthreads no xv6. Ou seja, implemente uma biblioteca de threads em nível de usuário para que um processo de usuário possa ter mais de uma thread e organize para que essas threads possam ser executadas em paralelo em CPUs diferentes. Crie um projeto que trate corretamente uma thread que faz uma chamada de sistema de bloqueio e altera seu espaço de endereço compartilhado.

Capítulo 7

Agendamento

Qualquer sistema operacional provavelmente executará com mais processos do que a capacidade de CPU do computador, portanto, é necessário um plano para compartilhar o tempo das CPUs entre os processos. Idealmente, o compartilhamento seria transparente para os processos do usuário. Uma abordagem comum é dar a cada processo a ilusão de que possui sua própria CPU virtual, multiplexando os processos nas CPUs de hardware. Este capítulo explica como o xv6 realiza essa multiplexação.

7.1 Multiplexação

O Xv6 multiplexa alternando cada CPU de um processo para outro em duas situações. Primeiro, o mecanismo de suspensão e ativação do xv6 alterna quando um processo aguarda a conclusão da E/S do dispositivo ou do pipe, ou aguarda a saída de um processo filho, ou aguarda na chamada de sistema de suspensão . Segundo, o xv6 força periodicamente uma troca para lidar com processos que computam por longos períodos sem suspensão. Essa multiplexação cria a ilusão de que cada processo possui sua própria CPU, assim como o xv6 utiliza o alocador de memória e as tabelas de páginas de hardware para criar a ilusão de que cada processo possui sua própria memória.

A implementação da multiplexação apresenta alguns desafios. Primeiro, como alternar de um processo para outro? Embora a ideia de troca de contexto seja simples, a implementação é um dos códigos mais opacos do Xv6. Segundo, como forçar trocas de forma transparente para os processos do usuário? O Xv6 usa a técnica padrão na qual as interrupções de um temporizador de hardware controlam as trocas de contexto. Terceiro, todas as CPUs alternam entre o mesmo conjunto compartilhado de processos, e um plano de bloqueio é necessário para evitar corridas. Quarto, a memória e outros recursos de um processo devem ser liberados quando o processo termina, mas ele não pode fazer tudo isso sozinho porque (por exemplo) não pode liberar sua própria pilha do kernel enquanto ainda a utiliza. Quinto, cada núcleo de uma máquina multinúcleo deve se lembrar de qual processo está executando para que as chamadas de sistema afetem o estado correto do kernel do processo. Finalmente, a suspensão e a ativação permitem que um processo desista da CPU e espere ser ativado por outro processo ou interrupção. É necessário cuidado para evitar corridas que resultem na perda de notificações de ativação. O Xv6 tenta resolver esses problemas da forma mais simples possível, mas mesmo assim o código resultante é complicado.

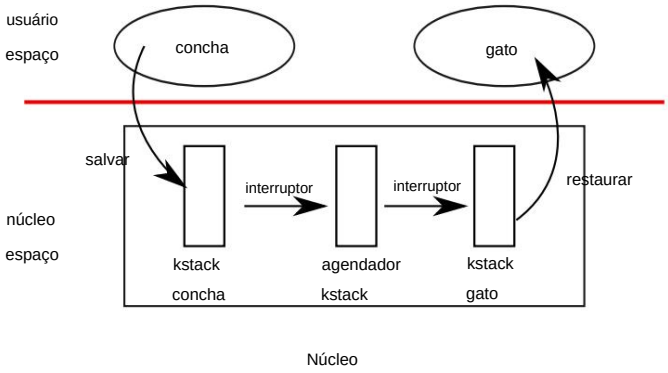

Figura 7.1: Troca de um processo de usuário para outro. Neste exemplo, o xv6 é executado com uma CPU (e, portanto, uma thread do escalonador).

7.2 Código: Troca de contexto

A Figura 7.1 descreve as etapas envolvidas na alternância de um processo de usuário para outro: uma transição usuário-kernel (chamada de sistema ou interrupção) para a thread do kernel do processo antigo, uma troca de contexto para a thread do escalonador da CPU atual, uma troca de contexto para a thread do kernel de um novo processo e um retorno de trap para o processo em nível de usuário. O escalonador xv6 possui uma thread dedicada (registradores e pilha salvos) por CPU, pois não é seguro para o escalonador executar na pilha do kernel do processo antigo: algum outro núcleo pode despertar o processo e executá-lo, e seria um desastre usar a mesma pilha em dois núcleos diferentes. Nesta seção, examinaremos a mecânica da alternância entre uma thread do kernel e uma thread do escalonador.

Alternar de um thread para outro envolve salvar os registradores da CPU do thread antigo e restaurar os registradores salvos anteriormente do novo thread; o fato de o ponteiro de pilha e o contador de programa serem salvos e restaurados significa que a CPU alternará as pilhas e o código que está executando.

A função swtch executa salvamentos e restaurações para uma troca de thread do kernel. swtch não sabe diretamente sobre threads; ele apenas salva e restaura conjuntos de 32 registradores RISC-V, chamados contextos.
Quando chega a hora de um processo liberar a CPU, a thread do kernel do processo chama swtch para salvar seu próprio contexto e retornar ao contexto do escalonador. Cada contexto está contido em uma estrutura context (kernel/proc.h:2).
contido na struct proc de um processo ou na struct cpu de uma CPU . swtch recebe dois argumentos: struct context *old e struct context *new. Ele salva os registradores atuais em old, carrega os registradores de new e retorna.

Vamos acompanhar um processo através de swtch até o escalonador. Vimos no Capítulo 4 que uma possibilidade ao final de uma interrupção é que a usertrap chame yield. O yield, por sua vez, chama sched, que chama swtch para salvar o contexto atual em p->context e alternar para o contexto do escalonador salvo anteriormente em cpu->context (kernel/proc.c:497).

interruptor (kernel/swtch.S:3) Salva apenas os registradores salvos pelo chamador; o compilador C gera código no chamador para salvar os registradores salvos pelo chamador na pilha. O swtch sabe o deslocamento do campo de cada registrador no contexto da estrutura. Ele não salva o contador do programa. Em vez disso, o swtch salva o registrador ra , que contém o endereço de retorno de onde o swtch foi chamado. Agora, o swtch restaura os registradores de

O novo contexto, que contém valores de registradores salvos por um swtch anterior. Quando swtch retorna, ele retorna às instruções apontadas pelo registrador ra restaurado , ou seja, a instrução da qual a nova thread chamou swtch anteriormente. Além disso, ele retorna para a pilha da nova thread, pois é para lá que o sp restaurado aponta.

Em nosso exemplo, o sched chamou swtch para alternar para cpu->context, o contexto do escalonador por CPU. Esse contexto foi salvo no momento em que o escalonador chamou swtch (kernel /proc.c:463). para alternar para o processo que está desistindo da CPU. Quando o switch que estávamos rastreando retorna, ele não retorna para sched , mas para scheduler, com o ponteiro de pilha na pilha do scheduler da CPU atual.

7.3 Código: Agendamento

A última seção analisou os detalhes de baixo nível do swtch; agora, vamos considerar o swtch como um dado e examinar a comutação de uma thread do kernel de um processo para outro, através do escalonador. O escalonador existe na forma de uma thread especial por CPU, cada uma executando a função de escalonador . Esta função é responsável por escolher qual processo executar em seguida. Um processo que deseja abrir mão da CPU deve adquirir seu próprio bloqueio de processo p->lock, liberar quaisquer outros bloqueios que esteja mantendo, atualizar seu próprio estado (p->state) e, em seguida, chamar sched. Você pode ver esta sequência em yield (kernel/proc.c:503). dormir e sair. sched verifica novamente alguns desses requisitos (kernel/proc.c:487-492) e então verifica uma implicação: como um bloqueio é mantido, as interrupções devem ser desabilitadas. Por fim, sched chama swtch para salvar o contexto atual em p->context e alternar para o contexto do escalonador em cpu->context. swtch retorna na pilha do escalonador como se o swtch do escalonador tivesse retornado (kernel/proc.c:463). O agendador continua seu loop for , encontra um processo para executar, alterna para ele e o ciclo se repete.

Acabamos de ver que o xv6 mantém p->lock em chamadas para swtch: o chamador de swtch já deve manter o bloqueio, e o controle do bloqueio passa para o código comutado. Essa convenção é incomum com bloqueios; geralmente, a thread que adquire um bloqueio também é responsável por liberá-lo, o que facilita o raciocínio sobre a correção. Para troca de contexto, é necessário quebrar essa convenção porque p->lock protege invariantes nos campos de estado e contexto do processo que não são verdadeiros durante a execução em swtch. Um exemplo de um problema que poderia surgir se p->lock não fosse mantido durante swtch: uma CPU diferente poderia decidir executar o processo após yield ter definido seu estado como RUNNABLE, mas antes de swtch fazer com que ele parasse de usar sua própria pilha do kernel. O resultado seriam duas CPUs rodando na mesma pilha, o que causaria o caos.

O único lugar onde uma thread do kernel cede sua CPU é no sched, e ela sempre alterna para o mesmo local no escalonador, que (quase) sempre alterna para alguma thread do kernel que anteriormente chamava o sched. Portanto, se alguém imprimisse os números das linhas onde o xv6 alterna as threads, observaria o seguinte padrão simples: (kernel/proc.c:463), (kernel/proc.c:497), (kernel/proc.c:463), (kernel/proc.c:497), e assim por diante. Procedimentos que intencionalmente transferem o controle entre si por meio de troca de threads são às vezes chamados de corrotinas; neste exemplo, sched e scheduler são corrotinas uma da outra.

Há um caso em que a chamada do planejador para swtch não termina em sched. allocproc define o registro ra de contexto de um novo processo para forkret (kernel/proc.c:515), para que sua primeira troca

“retorna” ao início dessa função. forkret existe para liberar o p->lock; caso contrário, como o novo processo precisa retornar ao espaço do usuário como se estivesse retornando de fork, ele poderia começar em usertrapret. scheduler (kernel/proc.c:445) executa

um loop: encontre um processo para executar, execute-o até que ele ceda e repita.

O escalonador percorre a tabela de processos em busca de um processo executável, que tenha p->state == RUNNABLE. Ao encontrar um processo, ele define a variável de processo atual por CPU c->proc, marca o processo como EM EXECUÇÃO e, em seguida, chama swtch para iniciar sua execução (kernel/proc.c:458-463).

Uma maneira de pensar sobre a estrutura do código de escalonamento é que ele impõe um conjunto de invariantes sobre cada processo e mantém p->lock sempre que essas invariantes não forem verdadeiras. Uma invariante é que, se um processo estiver EM EXECUÇÃO, o yield de uma interrupção de temporizador deve ser capaz de alternar com segurança para longe do processo; isso significa que os registradores da CPU devem conter os valores de registrador do processo (ou seja, swtch não os moveu para um contexto) e c->proc deve se referir ao processo. Outra invariante é que, se um processo for EXECUTÁVEL, deve ser seguro para o escalonador de uma CPU ociosa executá-lo; isso significa que p->context deve conter os registradores do processo (ou seja, eles não estão realmente nos registradores reais), que nenhuma CPU está executando na pilha do kernel do processo e que nenhum c->proc da CPU se refere ao processo. Observe que essas propriedades geralmente não são verdadeiras enquanto p->lock é mantido.

Manter as invariantes acima é o motivo pelo qual o xv6 frequentemente adquire p->lock em uma thread e o libera em outra, por exemplo, adquirindo em yield e liberando em scheduler. Uma vez que yield tenha começado a modificar o estado de um processo em execução para torná-lo RUNNABLE, o bloqueio deve permanecer mantido até que as invariantes sejam restauradas: o primeiro ponto de liberação correto é após o scheduler (executando em sua própria pilha) limpar c->proc. Da mesma forma, uma vez que o scheduler começa a converter um processo RUNNABLE para RUNNING, o bloqueio não pode ser liberado até que a thread do kernel esteja completamente em execução (após a troca, por exemplo em yield).

7.4 Código: mycpu e myproc

O Xv6 frequentemente precisa de um ponteiro para a estrutura proc do processo atual . Em um processador único, pode-se ter uma variável global apontando para o proc atual. Isso não funciona em uma máquina com vários núcleos, já que cada núcleo executa um processo diferente. A maneira de resolver esse problema é explorar o fato de que cada núcleo tem seu próprio conjunto de registradores; podemos usar um desses registradores para ajudar a encontrar informações por núcleo.

O Xv6 mantém uma estrutura cpu para cada CPU (kernel/proc.h:22), que registra o processo em execução naquela CPU (se houver), os registradores salvos para a thread do escalonador da CPU e a contagem de spinlocks aninhados necessários para gerenciar a desabilitação de interrupções. A função mycpu (kernel/proc.c:74) O Xv6 retorna um ponteiro para a estrutura cpu da CPU atual. O RISC-V numera suas CPUs, atribuindo a cada uma um hartid. O Xv6 garante que o hartid de cada CPU seja armazenado no registrador tp dessa CPU enquanto estiver no kernel.
Isso permite que o mycpu use o tp para indexar uma matriz de estruturas de CPU para encontrar a correta.

Garantir que o tp de uma CPU sempre mantenha o hartid da CPU é um pouco complicado. start define o registrador tp no início da sequência de inicialização da CPU, enquanto ainda está no modo de máquina (kernel/start.c:51). usertrapret salva tp na página do trampolim, pois o processo do usuário pode modificá-lo. Por fim, uservec restaura esse tp salvo ao entrar no kernel a partir do espaço do usuário (kernel/trampoline.S:77).
O compilador garante nunca usar o registrador tp . Seria mais conveniente se o xv6 pudesse

pergunte ao hardware RISC-V sobre o hartid atual sempre que necessário, mas o RISC-V permite isso apenas em modo máquina, não no modo supervisor.

Os valores de retorno de cpuid e mycpu são frágeis: se o temporizador interrompesse e causasse o thread para render e então mover para uma CPU diferente, um valor retornado anteriormente não seria mais correto. Para evitar esse problema, o xv6 requer que os chamadores desabilitem as interrupções e as habilitem apenas depois que eles terminam de usar a estrutura retornada cpu.

A função myproc (kernel/proc.c:83) retorna o ponteiro struct proc para o processo que está sendo executado na CPU atual. myproc desabilita interrupções, invoca mycpu, busca a CPU atual ponteiro de processo (c->proc) para fora da estrutura da CPU e, em seguida, habilita interrupções. O valor de retorno de myproc é seguro de usar mesmo se as interrupções estiverem habilitadas: se uma interrupção do temporizador move o processo de chamada para uma CPU diferente, seu ponteiro struct proc permanecerá o mesmo.

7.5 Sono e despertar

Agendamentos e bloqueios ajudam a ocultar as ações de uma thread de outra, mas também precisamos de abstrações que ajudem as threads a interagir intencionalmente. Por exemplo, o leitor de um pipe em xv6 pode precisa esperar que um processo de escrita produza dados; a chamada de um pai para esperar pode precisar esperar por um criança para sair; e um processo que lê o disco precisa esperar que o hardware do disco conclua a leitura. O kernel xv6 usa um mecanismo chamado suspensão e ativação nessas situações (e em muitas outras). O modo de suspensão permite que uma thread do kernel aguarde um evento específico; outra thread pode chamar o modo de ativação para indicar que as threads que aguardam um evento devem ser retomadas. O modo de suspensão e a ativação são frequentemente chamados de sequências. mecanismos de coordenação ou sincronização condicional.

O sono e a vigília fornecem uma interface de sincronização de nível relativamente baixo. Para motivar o como eles funcionam no xv6, nós os usaremos para construir um mecanismo de sincronização de nível superior chamado um semáforo [5] que coordena produtores e consumidores (xv6 não usa semáforos). A O semáforo mantém uma contagem e fornece duas operações. A operação “V” (para o produtor) incrementa a contagem. A operação “P” (para o consumidor) espera até que a contagem seja diferente de zero, e então decrementa e retorna. Se houvesse apenas um thread produtor e um consumidor thread, e eles foram executados em CPUs diferentes, e o compilador não otimizou de forma muito agressiva, esta implementação estaria correta:

```
100 estrutura semáforo {
101 estrutura de bloqueio spinlock;
102             contagem de int;
103         };
104
105 vazio
106 V(estrutura semáforo *s)
107 {
108             adquirir(&s->bloquear);
109             s->contagem += 1;
110             liberar(&s->bloquear);
111         }
```

```
112
113 vazio
114 P(struct semáforo *s)
115 {
116             enquanto(s->contagem == 0)
117                 ;
118             adquirir(&s->bloquear);
119             s->contagem -= 1;
120             liberar(&s->bloquear);
121         }
```

A implementação acima é cara. Se o produtor agir raramente, o consumidor gastará a maior parte do tempo girando no loop while esperando por uma contagem diferente de zero. A CPU do consumidor provavelmente conseguiria encontrar trabalho mais produtivo do que ficar ocupado esperando, consultando repetidamente s->count. Evitar a espera ocupada requer uma maneira para o consumidor ceder a CPU e retomar somente após V incrementa a contagem.

Aqui está um passo nessa direção, embora, como veremos, não seja suficiente. Imaginemos um par de chamadas, sleep e wakeup, que funcionam da seguinte maneira. sleep(chan) dorme no valor arbitrário chan, chamado de canal de espera. sleep coloca o processo de chamada em hibernação, liberando a CPU para outros trabalho. wakeup(chan) desperta todos os processos que estão dormindo no chan (se houver), fazendo com que suas chamadas de sono sejam retornar. Se não houver processos aguardando o canal, o wakeup não faz nada. Podemos alterar o semáforo implementação para usar sleep e wakeup (mudanças destacadas em amarelo):

```
200 vazio
201 V(struct semáforo *s)
202 {
203             adquirir(&s->bloquear);
204             s->contagem += 1;
205             despertar(es);
206             liberar(&s->bloquear);
207         }
208
209 nulo
210 P(estrutura semáforo *s)
211 {
212             enquanto(s->contagem == 0)
213                 sono(s);
214             adquirir(&s->bloquear);
215             s->contagem -= 1;
216             liberar(&s->bloquear);
217         }
```

P agora desiste da CPU em vez de girar, o que é bom. No entanto, não é fácil de projetar o sono e o despertar com esta interface sem sofrer com o que é conhecido como o problema do despertar perdido. Suponha que P encontre s->count == 0 na linha 212. Enquanto P está entre as linhas 212 e 213, V roda em outra CPU: ele muda s->count para ser diferente de zero e

chama wakeup, que não encontra nenhum processo dormindo e, portanto, não faz nada. Agora P continua executando na linha 213: ele chama sleep e entra em modo sleep. Isso causa um problema: P está dormindo esperando uma chamada de V isso já aconteceu. A menos que tenhamos sorte e o produtor ligue para V novamente, o consumidor irá esperar para sempre, mesmo que a contagem seja diferente de zero.

A raiz deste problema é que a invariante de que P só dorme quando s->count == 0 é violada por V correndo no momento errado. Uma maneira incorreta de proteger a invariante seria mover a aquisição do bloqueio (destacada em amarelo abaixo) em P para que sua verificação da contagem e sua chamada para o sono é atômico:

```
300 vazio
301 V(struct semáforo *s)
302 {
303             adquirir(&s->bloquear);
304             s->contagem += 1;
305             despertar(es);
306             liberar(&s->bloquear);
307         }
308
309 nulo
310 P(estrutura semáforo *s)
311 {
312             adquirir(&s->bloquear);
313             enquanto(s->contagem == 0)
314                 sono(s);
315             s->contagem -= 1;
316             liberar(&s->bloquear);
317         }
```

Poderíamos esperar que esta versão de P evitasse o despertar perdido porque o bloqueio impede V de executar entre as linhas 313 e 314. Ele faz isso, mas também gera um deadlock: P mantém o bloqueio enquanto ele dorme, então V ficará bloqueado para sempre esperando o bloqueio.

Vamos corrigir o esquema anterior alterando a interface do sleep : o chamador deve passar o bloqueio de condição para sleep para que ele possa liberar o bloqueio depois que o processo de chamada for marcado como sleep e aguardando no canal de suspensão. O bloqueio forçará um V concorrente a esperar até que P termine de colocar para dormir, para que o despertador encontre o consumidor adormecido e o acorde. Assim que o consumidor acorda novamente, o sono recupera o bloqueio antes de retornar. Nosso novo modo correto de sono/despertar o esquema pode ser utilizado da seguinte forma (alteração destacada em amarelo):

```
400 vazio
401 V(struct semáforo *s)
402 {
403             adquirir(&s->bloquear);
404             s->contagem += 1;
405             despertar(es);
406             liberar(&s->bloquear);
407         }
```

```
408
409 nulo
410 P(struct semáforo *s) 411 {

412             adquirir(&s->bloquear);
413             enquanto(s->contagem == 0)
414                     dormir(s, &s->bloqueio); s-
415             >contagem -= 1; liberar(&s-
416             >bloqueio);
417         }
```

O fato de P conter s->lock impede que V tente acordá-lo entre a verificação de s->count por P e sua chamada para dormir. Observe, no entanto, que precisamos de dormir para liberar atomicamente s->lock e colocar o processo consumidor em sono, a fim de evitar despertares perdidos.

7.6 Código: Sono e despertar

Suspensão do Xv6 (kernel/proc.c:536) e despertar (kernel/proc.c:567) Fornece a interface mostrada no último exemplo acima, e sua implementação (além das regras de como usá-las) garante que não haja despertares perdidos. A ideia básica é fazer com que o sleep marque o processo atual como SLEEPING e, em seguida, chamar sched para liberar a CPU; o wakeup procura um processo dormindo no canal de espera fornecido e o marca como RUNNABLE. Chamadores de sleep e wakeup podem usar qualquer número mutuamente conveniente como canal. O Xv6 frequentemente usa o endereço de uma estrutura de dados do kernel envolvida na espera.

sleep adquire p->lock (kernel/proc.c:547). Agora o processo que vai dormir mantém ambos p->lock e lk. Manter lk era necessário no chamador (no exemplo, P): isso garantia que nenhum outro processo (no exemplo, um V em execução) pudesse iniciar uma chamada para wakeup(chan). Agora que sleep mantém p->lock, é seguro liberar lk: algum outro processo pode iniciar uma chamada para wakeup(chan), mas wakeup aguardará para adquirir p->lock e, portanto, aguardará até que sleep termine de colocar o processo para dormir, evitando que wakeup perca o sleep.

Agora que o sleep contém p->lock e nenhum outro, ele pode colocar o processo para dormir gravando o canal de sono, alterando o estado do processo para SLEEPING e chamando sched (kernel/proc.c:551-554). Em breve ficará claro por que é essencial que p->lock não seja liberado (pelo agendador) até que o processo seja marcado como SLEEPING.

Em algum momento, um processo adquirirá o bloqueio de condição, definirá a condição que o sleeper está aguardando e chamará wakeup(chan). É importante que wakeup seja chamado enquanto a condição lock1 estiver sendo mantida . O wakeup percorre a tabela de processos (kernel/proc.c:567). Ele adquire o p->lock de cada processo que inspeciona, tanto porque pode manipular o estado desse processo quanto porque o p->lock garante que o modo de suspensão e o modo de ativação não se percam. Quando o modo de ativação encontra um processo no estado SLEEPING com um canal correspondente, ele altera o estado desse processo para RUNNABLE. Na próxima vez que o agendador for executado, ele verá que o processo está pronto para ser executado.

1A rigor, é suficiente que o despertar siga apenas a aquisição (ou seja, pode-se chamar o despertar após a liberação).

Por que as regras de bloqueio para sleep e wakeup garantem que um processo em modo sleep não perca um wakeup? O processo em modo sleep mantém o bloqueio de condição ou seu próprio p->lock , ou ambos, de um ponto antes de verificar a condição até um ponto depois de ser marcado como SLEEPING. O processo que chama o wakeup mantém ambos os bloqueios no loop do wakeup. Assim, o processo que o ativa torna a condição verdadeira antes que a thread consumidora a verifique; ou o processo que o ativa examina a thread em modo sleep estritamente após ela ter sido marcada como SLEEPING. Então, o processo em modo wakeup verá o processo em modo sleep e o ativará (a menos que algo o ative primeiro).

Às vezes, vários processos estão dormindo no mesmo canal; por exemplo, mais de um processo lendo de um pipe. Uma única chamada para wakeup despertará todos eles. Um deles será executado primeiro e obterá o bloqueio com o qual sleep foi chamado e (no caso de pipes) lerá quaisquer dados que estejam aguardando no pipe. Os outros processos descobrirão que, apesar de terem sido despertados, não há dados para serem lidos. Do ponto de vista deles, o wakeup foi "espúrio" e eles devem dormir novamente. Por esse motivo, sleep é sempre chamado dentro de um loop que verifica a condição.

Não há problema algum se dois usuários de sleep/wakeup escolherem acidentalmente o mesmo canal: eles verão despertares espúrios, mas o looping descrito acima tolerará esse problema. Grande parte do charme de sleep/wakeup reside em sua leveza (não há necessidade de criar estruturas de dados especiais para atuar como canais de sleep) e em fornecer uma camada de indireção (os chamadores não precisam saber com qual processo específico estão interagindo).

7.7 Código: Pipes

Um exemplo mais complexo que usa sleep e wakeup para sincronizar produtores e consumidores é a implementação de pipes do xv6. Vimos a interface para pipes no Capítulo 1: bytes gravados em uma extremidade de um pipe são copiados para um buffer no kernel e, em seguida, podem ser lidos da outra extremidade do pipe. Os próximos capítulos examinarão o suporte ao descritor de arquivo em torno dos pipes, mas vamos agora dar uma olhada nas implementações de pipewrite e piperead.

Cada pipe é representado por um pipe struct, que contém um bloqueio e um buffer de dados .
Os campos nread e nwrite contam o número total de bytes lidos e gravados no buffer. O buffer é encapsulado: o próximo byte escrito após buf[PIPESIZE-1] é buf[0]. As contagens não são encapsuladas. Essa convenção permite que a implementação distinga um buffer cheio (nwrite == nread+PIPESIZE) de um buffer vazio (nwrite == nread), mas significa que a indexação no buffer deve usar buf[nread % PIPESIZE] em vez de apenas buf[nread] (e similarmente para nwrite).

Vamos supor que as chamadas para piperead e pipewrite aconteçam simultaneamente em duas CPUs diferentes. pipewrite (kernel/pipe.c:77) começa adquirindo o bloqueio do pipe, que protege as contagens, os dados e seus invariantes associados. piperead (kernel/pipe.c:106) Então tenta adquirir o bloqueio também, mas não consegue. Ele gira em adquirir (kernel/spinlock.c:22) aguardando o bloqueio. Enquanto piperead aguarda, pipewrite percorre os bytes que estão sendo escritos (addr[0..n-1]), adicionando cada um ao pipe por vez (kernel/pipe.c:95). Durante esse loop, pode acontecer que o buffer fique cheio (kernel/pipe.c:88). Neste caso, o pipewrite chama wakeup para alertar os leitores adormecidos sobre o fato de que há dados aguardando no buffer e, então, entra em modo sleep em &pi->nwrite para esperar que um leitor retire alguns bytes do buffer. O sleep libera pi->lock como parte do processo de colocar o processo do pipewrite em modo sleep.

Agora que pi->lock está disponível, piperead consegue adquiri-lo e entra em sua seção crítica: ele descobre que pi->nread != pi->nwrite (kernel/pipe.c:113) (pipewrite entrou em modo de espera porque pi->nwrite == pi->nread+PIPESIZE (kernel/pipe.c:88)), então ele cai no loop for , copia os dados do pipe (kernel/pipe.c:120), e incrementa nread pelo número de bytes copiados. Essa quantidade de bytes agora está disponível para escrita, então piperead chama wakeup (kernel/pipe.c:127) para despertar quaisquer escritores adormecidos antes que ele retorne. O comando wakeup encontra um processo adormecido em &pi->nwrite, o processo que estava executando o pipewrite, mas parou quando o buffer ficou cheio. Ele marca esse processo como EXECUTÁVEL.

O código do pipe usa canais de suspensão separados para leitor e gravador (pi->nread e pi->nwrite); Isso pode tornar o sistema mais eficiente no caso improvável de haver muitos leitores e escritores aguardando o mesmo pipe. O código do pipe dorme dentro de um loop que verifica a condição de sono; se houver vários leitores ou escritores, todos os processos, exceto o primeiro a acordar, verão que a condição ainda é falsa e dormirão novamente.

7.8 Código: Espere, saia e mate

sleep e wakeup podem ser usados para muitos tipos de espera. Um exemplo interessante, apresentado no Capítulo 1, é a interação entre a saída de um filho e a espera de seu pai . No momento da morte do filho, o pai pode já estar dormindo em espera ou fazendo outra coisa; neste último caso, uma chamada subsequente para wait deve observar a morte do filho, talvez muito tempo depois de chamar exit. A maneira como o xv6 registra a morte do filho até que wait a observe é para exit colocar o chamador no estado ZOMBIE , onde permanece até que wait do pai perceba, altere o estado do filho para UNUSED, copie o status de saída do filho e retorne o ID do processo do filho ao pai. Se o pai sair antes do filho, o pai entrega o filho ao processo init , que chama wait perpetuamente; assim, cada filho tem um pai para limpar depois dele. Um desafio é evitar disputas e deadlocks entre esperas e saídas simultâneas de pai e filho, bem como saídas e saídas simultâneas.

wait começa adquirindo wait_lock (kernel/proc.c:391). O motivo é que wait_lock atua como um bloqueio de condição que ajuda a garantir que o pai não perca um despertar de um filho existente.
Em seguida, wait varre a tabela de processos. Se encontrar um filho no estado ZOMBIE, libera os recursos desse filho e sua estrutura proc , copia o status de saída do filho para o endereço fornecido para wait (se não for 0) e retorna o ID do processo do filho. Se wait encontrar filhos, mas nenhum tiver saído, chama sleep para aguardar que algum deles saia (kernel/proc.c:433). então verifica novamente. wait geralmente contém dois bloqueios, wait_lock e pp->lock de algum processo ; a ordem para evitar deadlock é primeiro wait_lock e depois pp->lock. exit (kernel/proc.c:347) Registra o status de saída, libera

alguns recursos, chama reparent para entregar seus filhos ao processo init , desperta o pai caso esteja em espera, marca o chamador como zumbi e libera a CPU permanentemente. exit mantém wait_lock e p->lock durante esta sequência. Ele mantém wait_lock porque é o bloqueio condicional para o wakeup(p->parent), evitando que um pai em espera perca o wakeup. exit também deve manter p->lock para esta sequência, para evitar que um pai em espera veja que o filho está no estado ZOMBIE antes que o filho finalmente chame swtch. exit adquire esses bloqueios na mesma ordem que wait para evitar deadlock.

Pode parecer incorreto que exit acorde o pai antes de definir seu estado como ZOMBIE, mas isso é seguro: embora wakeup possa fazer com que o pai seja executado, o loop em wait não pode examinar o filho até que o p->lock do filho seja liberado pelo planejador, então wait não pode examinar o processo de saída até bem depois que exit tenha definido seu estado como ZOMBIE (kernel/proc.c:379).

Enquanto exit permite que um processo se encerre, kill (kernel/proc.c:586) permite que um processo solicite o término de outro. Seria muito complexo para kill destruir diretamente o processo da vítima, já que a vítima pode estar executando em outra CPU, talvez no meio de uma sequência sensível de atualizações nas estruturas de dados do kernel. Portanto, kill faz muito pouco: apenas define o p->killed da vítima e, se estiver em modo de espera, a desperta. Eventualmente, a vítima entrará ou sairá do kernel, momento em que o código em usertrap chamará exit se p->killed estiver definido (ele verifica chamando killed (kernel/proc.c:615)). Se a vítima estiver sendo executada no espaço do usuário, ela logo entrará no kernel fazendo uma chamada de sistema ou porque o timer (ou algum outro dispositivo) interrompe.

Se o processo da vítima estiver em modo de espera, a chamada de ativação do comando kill fará com que a vítima retorne do modo de espera. Isso é potencialmente perigoso, pois a condição que está sendo aguardada pode não ser verdadeira. No entanto, chamadas xv6 para sleep são sempre encapsuladas em um loop while que testa novamente a condição após o retorno do sleep . Algumas chamadas para sleep também testam p->killed no loop e abandonam a atividade atual se ela estiver definida. Isso só é feito quando tal abandono for correto. Por exemplo, o código de leitura e gravação do pipe retorna se o sinalizador killed estiver definido; eventualmente, o código retornará para trap, que verificará novamente p->killed e sairá.

Alguns loops de suspensão xv6 não verificam p->killed porque o código está no meio de uma chamada de sistema multietapas que deveria ser atômica. O driver virtio (kernel/virtio_disk.c:285) Um exemplo: ele não verifica p->killed porque uma operação de disco pode ser uma de um conjunto de gravações necessárias para que o sistema de arquivos permaneça em um estado correto. Um processo que é encerrado enquanto aguarda E/S de disco não será encerrado até concluir a chamada de sistema atual e o usertrap visualizar o sinalizador de encerramento.

7.9 Bloqueio de Processo

O bloqueio associado a cada processo (p->lock) é o bloqueio mais complexo no xv6. Uma maneira simples de pensar em p->lock é que ele deve ser mantido durante a leitura ou gravação de qualquer um dos seguintes campos struct proc : p->state, p->chan, p->killed, p->xstate e p->pid. Esses campos podem ser usados por outros processos ou por threads do escalonador em outros núcleos, portanto, é natural que sejam protegidos por um bloqueio.

Entretanto, a maioria dos usos de p->lock protegem aspectos de nível superior da estrutura de dados do processo xv6. estruturas e algoritmos. Aqui está o conjunto completo de coisas que p->lock faz:

- Junto com p->state, ele evita corridas na alocação de slots proc[] para novos processos.
- Ele oculta um processo da vista enquanto ele está sendo criado ou destruído.
- Impede que a espera de um pai colete um processo que definiu seu estado como ZOMBIE, mas tem ainda não rendeu a CPU.
- Impede que o planejador de outro núcleo decida executar um processo de produção após definir seu estado para RUNNABLE , mas antes de terminar, alterne.

- Garante que apenas o agendador de um núcleo decida executar um processo RUNNABLE .
- Evita que uma interrupção do temporizador faça com que um processo ceda enquanto estiver em comutação.
- Junto com o bloqueio de condição, ajuda a evitar que o despertar ignore um processo que está chamando sleep mas não terminou de renderizar a CPU.
- Impede que o processo de morte da vítima saia e talvez seja realocado entre verificação de p->pid e configuração de p->killed.
- Torna a verificação e gravação de kill de p->state atômicas.

O campo p->parent é protegido pelo bloqueio global wait_lock em vez de p->lock. Somente o processo pai modifica p->parent, embora o campo seja lido tanto pelo próprio processo quanto por outros processos que buscam seus filhos. O propósito de wait_lock é atuar como o bloqueio condicional quando wait está em modo sleep, aguardando a saída de qualquer filho. Um filho que está saindo mantém wait_lock ou p->lock até que seu estado seja definido como ZOMBIE, seu pai tenha despertado e a CPU tenha sido liberada. wait_lock também serializa saídas simultâneas de um pai e um filho, de modo que o processo init (que herda o filho) tenha a garantia de ser despertado de sua espera. wait_lock é um bloqueio global, e não um bloqueio por processo em cada pai, porque, até que um processo o adquira, ele não pode saber quem é seu pai.

7.10 Mundo real

O escalonador xv6 implementa uma política de escalonamento simples, que executa cada processo por vez. Essa política é chamada de round robin. Sistemas operacionais reais implementam políticas mais sofisticadas que, por exemplo, permitem que os processos tenham prioridades. A ideia é que um processo executável de alta prioridade seja preferido pelo escalonador a um processo executável de baixa prioridade. Essas políticas podem se tornar complexas rapidamente porque frequentemente há objetivos conflitantes: por exemplo, o sistema operacional também pode querer garantir justiça e alto rendimento. Além disso, políticas complexas podem levar a interações não intencionais, como inversão de prioridade e comboios. A inversão de prioridade pode ocorrer quando um processo de baixa prioridade e um de alta prioridade usam um bloqueio específico, que, quando adquirido pelo processo de baixa prioridade, pode impedir o processo de alta prioridade de progredir. Um longo comboio de processos em espera pode se formar quando muitos processos de alta prioridade aguardam por um processo de baixa prioridade que adquire um bloqueio compartilhado; uma vez formado, o comboio pode persistir por um longo tempo. Para evitar esses tipos de problemas, mecanismos adicionais são necessários em planejadores sofisticados.

Sleep e Wakeup são métodos de sincronização simples e eficazes, mas existem muitos outros. O primeiro desafio em todos eles é evitar o problema de "perdas de ativação" que vimos no início do capítulo. A função sleep do kernel Unix original simplesmente desabilitava interrupções, o que era suficiente porque o Unix rodava em um sistema com uma única CPU. Como o xv6 roda em multiprocessadores, ele adiciona um bloqueio explícito à função sleep. O msleep do FreeBSD adota a mesma abordagem. A função sleep do Plan 9 usa uma função de retorno de chamada que é executada com o bloqueio de agendamento mantido pouco antes de entrar em modo sleep; a função serve como uma verificação de última hora da condição de sleep, para evitar perdas de ativação. O kernel Linux

sleep usa uma fila de processos explícita, chamada fila de espera, em vez de um canal de espera; a fila tem seu próprio bloqueio interno.

Varrer todo o conjunto de processos durante o processo de despertar é ineficiente. Uma solução melhor é substituir o canal tanto no modo de suspensão quanto no de despertar por uma estrutura de dados que contenha uma lista de processos em suspensão nessa estrutura, como a fila de espera do Linux. O modo de suspensão e o modo de despertar do Plan 9 chamam essa estrutura de ponto de encontro. Muitas bibliotecas de threads se referem à mesma estrutura como uma variável de condição; nesse contexto, as operações de suspensão e despertar são chamadas de espera e sinalização. Todos esses mecanismos compartilham o mesmo princípio: a condição de suspensão é protegida por algum tipo de bloqueio que é descartado atomicamente durante o processo de suspensão.

A implementação do wakeup ativa todos os processos que aguardam em um canal específico, e pode ser que muitos processos estejam aguardando esse canal específico. O sistema operacional agendará todos esses processos e eles verificarão rapidamente a condição de suspensão.
Processos que se comportam dessa maneira são às vezes chamados de "manada estrondosa" e é melhor evitá-los.
A maioria das variáveis de condição tem duas primitivas para ativação: sinal, que ativa um processo, e transmissão, que ativa todos os processos em espera.

Semáforos são frequentemente usados para sincronização. A contagem normalmente corresponde a algo como o número de bytes disponíveis em um buffer de pipe ou o número de filhos zumbis que um processo possui. Usar uma contagem explícita como parte da abstração evita o problema do "despertar perdido": há uma contagem explícita do número de despertares ocorridos. A contagem também evita os problemas de despertares espúrios e de manada estrondosa.

Terminar processos e limpá-los introduz muita complexidade no xv6. Na maioria dos sistemas operacionais, é ainda mais complexo, porque, por exemplo, o processo vítima pode estar dormindo profundamente dentro do kernel, e desenrolar sua pilha requer cuidado, já que cada função na pilha de chamadas pode precisar fazer alguma limpeza. Algumas linguagens ajudam fornecendo um mecanismo de exceção, mas não C. Além disso, existem outros eventos que podem fazer com que um processo em espera seja despertado, mesmo que o evento que ele está aguardando ainda não tenha ocorrido. Por exemplo, quando um processo Unix está em espera, outro processo pode enviar um sinal para ele. Nesse caso, o processo retornará da chamada de sistema interrompida com o valor -1 e com o código de erro definido como EINTR. O aplicativo pode verificar esses valores e decidir o que fazer. O Xv6 não suporta sinais e essa complexidade não surge.

O suporte do Xv6 para kill não é totalmente satisfatório: existem loops de sleep que provavelmente deveriam verificar se há p->killed. Um problema relacionado é que, mesmo para loops de sleep que verificam p->killed, há uma corrida entre sleep e kill; este último pode ativar p->killed e tentar acordar a vítima logo após o loop da vítima verificar p->killed, mas antes de chamar sleep. Se esse problema ocorrer, a vítima não notará o p->killed até que a condição que está aguardando ocorra. Isso pode ocorrer um pouco mais tarde ou até mesmo nunca (por exemplo, se a vítima estiver aguardando uma entrada do console, mas o usuário não digitar nenhuma entrada).

Um sistema operacional real encontraria estruturas proc livres com uma lista livre explícita em constante tempo em vez da pesquisa de tempo linear em allocproc; xv6 usa a varredura linear para simplificar.

7.11 Exercícios

1. O sono tem que verificar lk != &p->lock para evitar um deadlock. Suponha que o caso especial fosse eliminado pela substituição

```
se(lk != &p->lock){
    adquirir(&p->bloquear);
    liberar(lk);
}
```

com

```
liberar(lk);
adquirir(&p->bloquear);
```

Fazer isso interromperia o sono. Como?

2. Implemente semáforos em xv6 sem usar sleep e wakeup (mas é aceitável usar bloqueios de spin). Substitua os usos de sleep e wakeup em xv6 por semáforos. Avalie o resultado.

3. Corrija a corrida mencionada acima entre kill e sleep, de modo que uma kill que ocorra depois que o loop sleep da vítima verificar p->killed, mas antes de chamar sleep, resulte no abandono da chamada de sistema atual pela vítima.

4. Crie um plano para que cada loop de suspensão verifique p->killed para que, por exemplo, um processo que está no driver virtio possa retornar rapidamente do loop while se for morto por outro processo.

5. Modifique o xv6 para usar apenas uma troca de contexto ao alternar entre as threads do kernel de um processo, em vez de alternar através da thread do escalonador. A thread que está rendendo o processo precisará selecionar a próxima thread e chamar swtch. Os desafios serão evitar que múltiplos núcleos executem a mesma thread acidentalmente; acertar o bloqueio; e evitar deadlocks.

6. Modifique o escalonador do xv6 para usar a instrução RISC-V WFI (wait for interrupt) quando nenhum processo estiver executável. Tente garantir que, sempre que houver processos executáveis aguardando para serem executados, nenhum núcleo esteja pausado no WFI.

Capítulo 8

Sistema de arquivos

O objetivo de um sistema de arquivos é organizar e armazenar dados. Sistemas de arquivos normalmente oferecem suporte ao compartilhamento de dados entre usuários e aplicativos, bem como à persistência, para que os dados permaneçam disponíveis após uma reinicialização.

O sistema de arquivos xv6 fornece arquivos, diretórios e nomes de caminho semelhantes aos do Unix (consulte o Capítulo 1) e armazena seus dados em um disco virtio para persistência. O sistema de arquivos aborda vários desafios:

- O sistema de arquivos precisa de estruturas de dados no disco para representar a árvore de diretórios e arquivos nomeados, para registrar as identidades dos blocos que contêm o conteúdo de cada arquivo e para registrar quais áreas do disco estão livres.

- O sistema de arquivos deve suportar recuperação de falhas. Ou seja, se ocorrer uma falha (por exemplo, falta de energia), o sistema de arquivos ainda deve funcionar corretamente após uma reinicialização. O risco é que uma falha interrompa uma sequência de atualizações e deixe estruturas de dados inconsistentes no disco (por exemplo, um bloco que é usado em um arquivo e marcado como livre).

- Diferentes processos podem operar no sistema de arquivos ao mesmo tempo, portanto, o código do sistema de arquivos deve coordenar para manter invariantes.

- O acesso a um disco é muito mais lento do que o acesso à memória, portanto, o sistema de arquivos deve manter um cache na memória de blocos populares.

O restante deste capítulo explica como o xv6 aborda esses desafios.

8.1 Visão geral

A implementação do sistema de arquivos xv6 é organizada em sete camadas, mostradas na Figura 8.1. A camada de disco lê e grava blocos em um disco rígido virtio. A camada de cache de buffer armazena em cache os blocos de disco e sincroniza o acesso a eles, garantindo que apenas um processo do kernel por vez possa modificar os dados armazenados em um bloco específico. A camada de registro permite que camadas superiores encapsulem atualizações para vários blocos em uma transação e garante que os blocos sejam atualizados atomicamente em caso de falhas (ou seja, todos eles são atualizados ou nenhum). A camada de inode fornece arquivos individuais, cada um

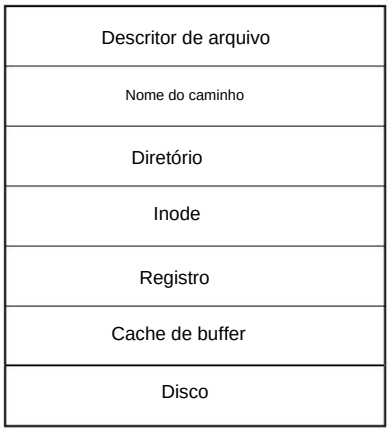

Figura 8.1: Camadas do sistema de arquivos xv6.

Representado como um inode com um número i único e alguns blocos contendo os dados do arquivo. A camada de diretório implementa cada diretório como um tipo especial de inode cujo conteúdo é uma sequência de entradas de diretório, cada uma contendo o nome e o número i de um arquivo. A camada de nome de caminho fornece nomes de caminho hierárquicos, como /usr/rtm/xv6/fs.c, e os resolve com consulta recursiva. A camada de descritor de arquivo abstrai muitos recursos Unix (por exemplo, pipes, dispositivos, arquivos, etc.) usando a interface do sistema de arquivos, simplificando a vida dos programadores de aplicativos.

O hardware de disco tradicionalmente apresenta os dados no disco como uma sequência numerada de blocos de 512 bytes (também chamados de setores): o setor 0 corresponde aos primeiros 512 bytes, o setor 1 aos próximos e assim por diante. O tamanho do bloco que um sistema operacional usa para seu sistema de arquivos pode ser diferente do tamanho do setor que um disco usa, mas normalmente o tamanho do bloco é um múltiplo do tamanho do setor. O Xv6 armazena cópias dos blocos que ele leu na memória em objetos do tipo struct buf (kernel/buf.h:1). Os dados armazenados nessa estrutura às vezes estão fora de sincronia com o disco: eles podem ainda não ter sido lidos do disco (o disco está trabalhando nisso, mas ainda não retornou o conteúdo do setor) ou podem ter sido atualizados pelo software, mas ainda não foram gravados no disco.

O sistema de arquivos deve ter um plano para onde ele armazena inodes e blocos de conteúdo no disco. Para fazer isso, o xv6 divide o disco em várias seções, como mostra a Figura 8.2. O sistema de arquivos não usa o bloco 0 (ele contém o setor de boot). O bloco 1 é chamado de superbloco; ele contém metadados sobre o sistema de arquivos (o tamanho do sistema de arquivos em blocos, o número de blocos de dados, o número de inodes e o número de blocos no log). Os blocos que começam em 2 armazenam o log. Após o log estão os inodes, com vários inodes por bloco. Depois deles, vêm os blocos de bitmap que rastreiam quais blocos de dados estão em uso. Os blocos restantes são blocos de dados; cada um é marcado como livre no bloco de bitmap ou contém conteúdo para um arquivo ou diretório. O superbloco é preenchido por um programa separado, chamado mkfs, que constrói um sistema de arquivos inicial.

O restante deste capítulo discute cada camada, começando pelo cache do buffer. Fique atento a situações
situações em que abstrações bem escolhidas em camadas inferiores facilitam o design de camadas superiores.

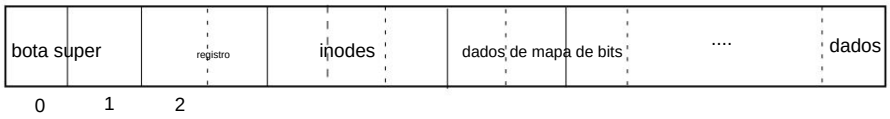

Figura 8.2: Estrutura do sistema de arquivos xv6.

8.2 Camada de cache de buffer

O cache de buffer tem duas funções: (1) sincronizar o acesso aos blocos do disco para garantir que apenas uma cópia de um bloco esteja na memória e que apenas uma thread do kernel por vez utilize essa cópia; (2) armazenar em cache os blocos populares para que não precisem ser relidos do disco lento. O código está em bio.c.

A interface principal exportada pelo cache de buffer consiste em bread e bwrite; o primeiro obtém um buf contendo uma cópia de um bloco que pode ser lido ou modificado na memória, e o segundo grava um buffer modificado no bloco apropriado no disco. Uma thread do kernel deve liberar um buffer chamando brelse ao terminar de usá-lo. O cache de buffer usa um bloqueio de suspensão por buffer para garantir que apenas uma thread por vez use cada buffer (e, portanto, cada bloco do disco); bread retorna um buffer bloqueado e brelse libera o bloqueio.

Vamos retornar ao cache de buffer. O cache de buffer tem um número fixo de buffers para armazenar dados de disco. blocos, o que significa que, se o sistema de arquivos solicitar um bloco que ainda não esteja no cache, o cache de buffer deve reciclar um buffer que atualmente contém outro bloco. O cache de buffer recicla o buffer menos usado recentemente para o novo bloco. A suposição é que o buffer menos usado recentemente é o menos provável de ser usado novamente em breve.

8.3 Código: Cache de buffer

O cache de buffer é uma lista duplamente encadeada de buffers. A função binit, chamada por main (kernel/ -main.c:27), inicializa a lista com os buffers NBUF no array estático buf (kernel/bio.c:43-52). Todos os outros acessos ao cache do buffer referem-se à lista encadeada via bcache.head, não ao array buf .

Um buffer possui dois campos de estado associados. O campo válido indica que o buffer contém uma cópia do bloco. O campo disco indica que o conteúdo do buffer foi entregue ao disco, o que pode alterar o buffer (por exemplo, gravar dados do disco em dados).

pão (kernel/bio.c:93) chama bget para obter um buffer para o setor fornecido (kernel/bio.c:97). Se o buffer precisar ser lido do disco, o bread chama virtio_disk_rw para fazer isso antes de retornar o buffer. bget (kernel/bio.c:59) verifica a lista de

buffers em busca de um buffer com os números de dispositivo e setor fornecidos (kernel/bio.c:65-73). Se houver tal buffer, o bget adquire o bloqueio de suspensão para o buffer. O bget então retorna o buffer bloqueado.

Se não houver um buffer em cache para o setor fornecido, o bget deve criar um, possivelmente reutilizando um buffer que continha um setor diferente. Ele verifica a lista de buffers uma segunda vez, procurando por um buffer que não esteja em uso (b->refcnt = 0); qualquer buffer desse tipo pode ser usado. O bget edita os metadados do buffer para registrar o novo dispositivo e número do setor e obtém seu bloqueio de suspensão. Observe que a atribuição b->valid = 0 garante que o bread lerá os dados do bloco do disco em vez de usar incorretamente os dados do buffer.

conteúdos anteriores.

É importante que haja no máximo um buffer em cache por setor do disco, para garantir que os leitores vejam as gravações e porque o sistema de arquivos usa bloqueios nos buffers para sincronização. O bget garante essa invariante mantendo o comando bache.lock continuamente desde a verificação do primeiro loop, se o bloco está em cache, até a declaração do segundo loop, de que o bloco agora está em cache (definindo dev, blockno e refcnt). Isso faz com que a verificação da presença de um bloco e (se não estiver presente) a designação de um buffer para conter o bloco seja atômica.

É seguro para o bget adquirir o sleep-lock do buffer fora da seção crítica bcache.lock , já que o b->refcnt diferente de zero impede que o buffer seja reutilizado para um bloco de disco diferente. O sleep-lock protege leituras e gravações do conteúdo armazenado em buffer do bloco, enquanto o bcache.lock protege informações sobre quais blocos estão armazenados em cache.

Se todos os buffers estiverem ocupados, muitos processos estarão executando chamadas ao sistema de arquivos simultaneamente; o bget entra em pânico. Uma resposta mais elegante seria suspender o processamento até que um buffer fique livre, embora houvesse a possibilidade de um deadlock.

Após o bread ler o disco (se necessário) e retornar o buffer ao seu chamador, este último terá uso exclusivo do buffer e poderá ler ou gravar os bytes de dados. Se o chamador modificar o buffer, deverá chamar bwrite para gravar os dados alterados no disco antes de liberar o buffer. bwrite (kernel/bio.c:107) chama virtio_disk_rw para falar com o hardware do disco.

Quando o chamador termina de usar um buffer, ele deve chamar brelse para liberá-lo. (O nome brelse, uma abreviação de b-release, é enigmático, mas vale a pena aprender: ele se originou no Unix e também é usado no BSD, Linux e Solaris.) brelse (kernel/bio.c:117) libera o bloqueio de suspensão e move o buffer para a frente da lista encadeada (kernel/ bio.c:128-133). Mover o buffer faz com que a lista seja ordenada pela data de uso recente dos buffers (ou seja, liberação): o primeiro buffer da lista é o mais recentemente usado e o último é o menos recentemente usado. Os dois loops em bget aproveitam isso: a busca por um buffer existente deve processar a lista inteira no pior caso, mas verificar primeiro os buffers usados mais recentemente (começando em bcache.head e seguindo os ponteiros "next ") reduzirá o tempo de busca quando houver uma boa localidade de referência. A busca para selecionar um buffer para reutilização seleciona o buffer menos recentemente usado, varrendo para trás (seguindo os ponteiros "prev").

8.4 Camada de registro

Um dos problemas mais interessantes no projeto de sistemas de arquivos é a recuperação de falhas. O problema surge porque muitas operações do sistema de arquivos envolvem múltiplas gravações no disco, e uma falha após um subconjunto das gravações pode deixar o sistema de arquivos em disco em um estado inconsistente. Por exemplo, suponha que ocorra uma falha durante o truncamento de um arquivo (definindo o tamanho de um arquivo como zero e liberando seus blocos de conteúdo). Dependendo da ordem das gravações em disco, a falha pode deixar um inode com uma referência a um bloco de conteúdo marcado como livre ou pode deixar um bloco de conteúdo alocado, mas não referenciado.

Este último é relativamente benigno, mas um inode que se refere a um bloco liberado provavelmente causará sérios problemas após uma reinicialização. Após a reinicialização, o kernel pode alocar esse bloco para outro arquivo, e agora temos dois arquivos diferentes apontando involuntariamente para o mesmo bloco. Se o xv6 suportasse múltiplos usuários, essa situação poderia ser um problema de segurança, já que o proprietário do arquivo antigo seria capaz de ler

e escrever blocos no novo arquivo, de propriedade de um usuário diferente.

O Xv6 resolve o problema de travamentos durante operações do sistema de arquivos com uma forma simples de registro. Uma chamada de sistema xv6 não grava diretamente as estruturas de dados do sistema de arquivos em disco. Em vez disso, ela registra em um log no disco uma descrição de todas as gravações que deseja realizar. Após registrar todas as suas gravações, a chamada de sistema grava um registro de confirmação especial no disco, indicando que o log contém uma operação concluída. Nesse ponto, a chamada de sistema copia as gravações para as estruturas de dados do sistema de arquivos em disco. Após a conclusão dessas gravações, a chamada de sistema apaga o log no disco.

Se o sistema travar e reinicializar, o código do sistema de arquivos se recupera da falha da seguinte maneira, antes de executar qualquer processo. Se o log for marcado como contendo uma operação concluída, o código de recuperação copia as gravações para o local correspondente no sistema de arquivos em disco. Se o log não for marcado como contendo uma operação concluída, o código de recuperação ignora o log. O código de recuperação finaliza apagando o log.

Por que o log do xv6 resolve o problema de travamentos durante operações do sistema de arquivos? Se o travamento ocorrer antes da operação ser confirmada, o log no disco não será marcado como concluído, o código de recuperação o ignorará e o estado do disco será como se a operação nem tivesse sido iniciada. Se o travamento ocorrer após a operação ser confirmada, a recuperação reproduzirá todas as gravações da operação, talvez repetindo-as se a operação tivesse começado a gravá-las na estrutura de dados do disco. Em ambos os casos, o log torna as operações atômicas em relação a travamentos: após a recuperação, todas as gravações da operação aparecem no disco ou nenhuma delas aparece.

8.5 Projeto de log

O log reside em um local fixo conhecido, especificado no superbloco. Ele consiste em um bloco de cabeçalho seguido por uma sequência de cópias de bloco atualizadas ("blocos registrados"). O bloco de cabeçalho contém uma matriz de números de setor, um para cada bloco registrado, e a contagem de blocos de log. A contagem no bloco de cabeçalho no disco é zero, indicando que não há transação no log, ou diferente de zero, indicando que o log contém uma transação confirmada completa com o número indicado de blocos registrados. O Xv6 grava o bloco de cabeçalho quando uma transação é confirmada, mas não antes, e define a contagem como zero após copiar os blocos registrados para o sistema de arquivos. Portanto, uma falha no meio de uma transação resultará em uma contagem zero no bloco de cabeçalho do log; uma falha após uma confirmação resultará em uma contagem diferente de zero.

O código de cada chamada de sistema indica o início e o fim da sequência de gravações que deve ser atômica em relação a travamentos. Para permitir a execução simultânea de operações do sistema de arquivos por diferentes processos, o sistema de registro pode acumular as gravações de múltiplas chamadas de sistema em uma única transação.
Assim, um único commit pode envolver a gravação de várias chamadas de sistema completas. Para evitar a divisão de uma chamada de sistema entre transações, o sistema de registro só realiza o commit quando não há chamadas de sistema de arquivos em andamento.

A ideia de confirmar várias transações juntas é conhecida como confirmação em grupo. A confirmação em grupo reduz o número de operações de disco porque amortiza o custo fixo de uma confirmação em múltiplas operações. A confirmação em grupo também transfere ao sistema de disco mais gravações simultâneas, talvez permitindo que o disco grave todas elas durante uma única rotação. O driver virtio do Xv6 não suporta esse tipo de processamento em lote, mas o design do sistema de arquivos do Xv6 permite.

O Xv6 dedica uma quantidade fixa de espaço em disco para armazenar o log. O número total de blocos gravados pelas chamadas de sistema em uma transação deve caber nesse espaço. Isso tem duas consequências.
Nenhuma chamada de sistema pode gravar mais blocos distintos do que o espaço disponível no log.
Isso não é um problema para a maioria das chamadas de sistema, mas duas delas podem potencialmente gravar muitos blocos: write e unlink. Uma gravação em um arquivo grande pode gravar muitos blocos de dados e muitos blocos de bitmap, bem como um bloco de inode; desvincular um arquivo grande pode gravar muitos blocos de bitmap e um inode. A chamada de sistema write do Xv6 divide gravações grandes em várias gravações menores que cabem no log, e unlink não causa problemas porque, na prática, o sistema de arquivos xv6 usa apenas um bloco de bitmap. A outra consequência do espaço limitado no log é que o sistema de log não pode permitir que uma chamada de sistema seja iniciada a menos que tenha certeza de que as gravações da chamada de sistema caberão no espaço restante no log.

8.6 Código: registro

Um uso típico do log em uma chamada de sistema se parece com isto:

```
begin_op();
 ...
bp = pão(...); bp-
>dados[...] = ...; log_write(bp);

 ...
fim_op();
```

begin_op (kernel/log.c:127) aguarda até que o sistema de registro não esteja realizando commits no momento e até que haja espaço de log não reservado suficiente para armazenar as gravações desta chamada. log.outstanding conta o número de chamadas de sistema que reservaram espaço de log; o espaço total reservado é log.outstanding vezes MAXOPBLOCKS. Incrementar log.outstanding reserva espaço e impede que um commit ocorra durante esta chamada de sistema. O código, conservadoramente, assume que cada chamada de sistema pode gravar até MAXOPBLOCKS blocos distintos.

log_write (kernel/log.c:215) atua como um proxy para bwrite. Ele registra o número do setor do bloco na memória, reservando um slot no log no disco, e fixa o buffer no cache do bloco para evitar que o cache do bloco o despeje. O bloco deve permanecer no cache até ser confirmado: até então, a cópia em cache é o único registro da modificação; ele não pode ser gravado em seu lugar no disco até depois do commit; e outras leituras na mesma transação devem ver as modificações. log_write percebe quando um bloco é gravado várias vezes durante uma única transação e aloca esse bloco no mesmo slot no log. Essa otimização é frequentemente chamada de absorção. É comum que, por exemplo, o bloco de disco contendo inodes de vários arquivos seja gravado várias vezes dentro de uma transação. Ao absorver várias gravações de disco em uma, o sistema de arquivos pode economizar espaço de log e obter melhor desempenho porque apenas uma cópia do bloco de disco deve ser gravada no disco.

fim_op (kernel/log.c:147) Primeiro, diminui a contagem de chamadas de sistema pendentes. Se a contagem for zero, ele confirma a transação atual chamando commit(). Há quatro etapas neste processo. write_log() (kernel/log.c:179) copia cada bloco modificado na transação do cache do buffer para seu slot no log no disco. write_head() (kernel/log.c:103) escreve o bloco de cabeçalho no disco: este é o ponto de confirmação e uma falha após a gravação resultará na recuperação da reprodução do

gravações de transações do log. install_trans (kernel/log.c:69) lê cada bloco do log e o grava no local apropriado no sistema de arquivos. Por fim, end_op grava o cabeçalho do log com contagem zero; isso precisa acontecer antes que a próxima transação comece a gravar os blocos registrados, para que uma falha não resulte na recuperação usando o cabeçalho de uma transação com os blocos registrados da transação subsequente. recover_from_log (kernel/log.c:117) é chamado de initlog (kernel/log.c:55), que é chamado de

fsinit(kernel/fs.c:42) durante a inicialização antes da execução do primeiro processo do usuário (kernel/proc.c:527). Ele lê o cabeçalho do log e imita as ações do end_op se o cabeçalho indicar que o log contém uma transação confirmada.

Um exemplo de uso do log ocorre em filewrite (kernel/file.c:135). A transação se parece com isso:

```
começar_op();
ilock(f->ip); r =
escreveri(f->ip, ...); iunlock(f->ip);
end_op();
```

Este código é encapsulado em um loop que divide gravações grandes em transações individuais de apenas alguns setores por vez, para evitar o estouro do log. A chamada a writei grava muitos blocos como parte dessa transação: o inode do arquivo, um ou mais blocos de bitmap e alguns blocos de dados.

8.7 Código: Alocador de bloco

O conteúdo de arquivos e diretórios é armazenado em blocos de disco, que devem ser alocados de um pool livre. O alocador de blocos do Xv6 mantém um bitmap livre no disco, com um bit por bloco. Um bit zero indica que o bloco correspondente está livre; um bit um indica que está em uso. O programa mkfs define os bits correspondentes ao setor de boot, superbloco, blocos de log, blocos de inode e blocos de bitmap.

O alocador de blocos fornece duas funções: balloc aloca um novo bloco de disco e bfree libera um bloco. balloc O loop em balloc em (kernel/fs.c:72) Considera cada bloco, começando no bloco 0 até sb.size, o número de blocos no sistema de arquivos. Ele procura um bloco cujo bit de bitmap seja zero, indicando que está livre. Se balloc encontrar tal bloco, ele atualiza o bitmap e o retorna. Para maior eficiência, o loop é dividido em duas partes. O loop externo lê cada bloco de bits de bitmap. O loop interno verifica todos os bits de Bits por Bloco (BPB) em um único bloco de bitmap. A corrida que pode ocorrer se dois processos tentarem alocar um bloco ao mesmo tempo é evitada pelo fato de que o cache de buffer permite que apenas um processo use um bloco de bitmap por vez. bfree (kernel/fs.c:92) encontra o bloco de bitmap correto e limpa o bit correto. Novamente o exclusivo

o uso implícito de bread e brelse evita a necessidade de bloqueio explícito.

Como acontece com grande parte do código descrito no restante deste capítulo, balloc e bfree devem ser chamado dentro de uma transação.

8.8 Camada Inode

O termo inode pode ter um de dois significados relacionados. Pode se referir à estrutura de dados em disco que contém o tamanho de um arquivo e a lista de números de blocos de dados. Ou "inode" pode se referir a um inode na memória, que contém uma cópia do inode em disco, bem como informações extras necessárias dentro do kernel.

Os inodes no disco são compactados em uma área contígua do disco chamada de blocos de inode. Cada inode tem o mesmo tamanho, então é fácil, dado um número n, encontrar o enésimo inode no disco. Na verdade, esse número n, chamado de número de inode ou número i, é como os inodes são identificados na implementação.

O inode no disco é definido por um dinode struct (kernel/fs.h:32). O campo type distingue entre arquivos, diretórios e arquivos especiais (dispositivos). Um tipo zero indica que um inode em disco está livre. O campo nlink conta o número de entradas de diretório que se referem a esse inode, a fim de reconhecer quando o inode em disco e seus blocos de dados devem ser liberados. O campo size registra o número de bytes de conteúdo no arquivo. O array addrs registra os números dos blocos de disco que contêm o conteúdo do arquivo.

O kernel mantém o conjunto de inodes ativos na memória em uma tabela chamada itable; struct inode (kernel/file.h:17) é a cópia em memória de um dinode struct no disco. O kernel armazena um inode na memória somente se houver ponteiros C referentes a esse inode. O campo ref conta o número de ponteiros C referentes ao inode na memória, e o kernel descarta o inode da memória se a contagem de referências cair para zero. As funções iget e iput adquirem e liberam ponteiros para um inode, modificando a contagem de referências. Ponteiros para um inode podem vir de descritores de arquivo, diretórios de trabalho atuais e código transitório do kernel, como exec.

Existem quatro mecanismos de bloqueio ou similares no código de inode do xv6. itable.lock protege a invariante de que um inode está presente na tabela de inodes no máximo uma vez, e a invariante de que um campo ref de um inode na memória conta o número de ponteiros na memória para o inode. Cada inode na memória possui um campo de bloqueio contendo um sleep-lock, que garante acesso exclusivo aos campos do inode (como o tamanho do arquivo), bem como aos blocos de conteúdo de arquivo ou diretório do inode. A ref de um inode, se for maior que zero, faz com que o sistema mantenha o inode na tabela e não reutilize a entrada da tabela para um inode diferente. Por fim, cada inode contém um campo nlink (no disco e copiado na memória, se estiver na memória) que conta o número de entradas de diretório que se referem a um arquivo; o xv6 não libera um inode se sua contagem de links for maior que zero.

Um ponteiro de inode de struct retornado por iget() tem a garantia de ser válido até a chamada correspondente a iput(); o inode não será excluído e a memória referenciada pelo ponteiro não será reutilizada para um inode diferente. iget() fornece acesso não exclusivo a um inode, permitindo que haja vários ponteiros para o mesmo inode. Muitas partes do código do sistema de arquivos dependem desse comportamento de iget(), tanto para manter referências de longo prazo a inodes (como arquivos abertos e diretórios atuais) quanto para evitar disputas, evitando deadlocks em código que manipula múltiplos inodes (como consultas de caminho).

O inode da estrutura que iget retorna pode não ter nenhum conteúdo útil. Para garantir que ele contenha uma cópia do inode no disco, o código deve chamar ilock. Isso bloqueia o inode (para que nenhum outro processo possa bloqueá- lo) e lê o inode do disco, caso ainda não tenha sido lido. iunlock libera o bloqueio do inode. Separar a aquisição de ponteiros de inode do bloqueio ajuda a evitar deadlocks em algumas situações, por exemplo, durante a consulta de diretórios. Vários processos podem conter um

Ponteiro C para um inode retornado por iget, mas somente um processo pode bloquear o inode por vez.

A tabela de inodes armazena apenas inodes para os quais o código do kernel ou as estruturas de dados contêm ponteiros C. Sua principal função é sincronizar o acesso por múltiplos processos. A tabela de inodes também armazena em cache inodes usados com frequência, mas o cache é secundário; se um inode for usado com frequência, o cache de buffer provavelmente o manterá na memória. O código que modifica um inode na memória o grava em disco com iupdate.

8.9 Código: Inodes

Para alocar um novo inode (por exemplo, ao criar um arquivo), o xv6 chama ialloc (kernel/fs.c:199). ialloc é semelhante a balloc: ele percorre as estruturas de inode no disco, um bloco de cada vez, procurando por uma que esteja marcada como livre. Ao encontrar uma, ele a reivindica gravando o novo tipo no disco e, em seguida, retorna uma entrada da tabela de inode com a chamada final para iget (kernel/fs.c:213). A operação correta do ialloc depende do fato de que apenas um processo por vez pode estar mantendo uma referência a bp: o ialloc pode ter certeza de que algum outro processo não verá simultaneamente que o inode está disponível e tentará reivindicá-lo. iget (kernel/fs.c:247) procura na tabela de inodes por uma entrada ativa (ip->ref > 0) com o

dispositivo e número de inode desejados. Se encontrar um, retorna uma nova referência para esse inode (kernel/fs.c:256-260). À medida que o iget faz a varredura, ele registra a posição do primeiro slot vazio (kernel/fs.c:261-262), que ele usa se precisar alocar uma entrada de tabela.

O código deve bloquear o inode usando ilock antes de ler ou gravar seus metadados ou conteúdo. ilock (kernel/fs.c:293) usa um sleep-lock para esse propósito. Assim que o ilock tiver acesso exclusivo ao inode, ele lê o inode do disco (mais provavelmente, do cache do buffer), se necessário. A função iunlock (kernel/fs.c:321) libera o bloqueio de suspensão, o que pode fazer com que quaisquer processos em suspensão sejam ativados.

iput (kernel/fs.c:337) libera um ponteiro C para um inode diminuindo a contagem de referência (kernel/fs.c:360). Se esta for a última referência, o slot do inode na tabela de inodes agora estará livre e poderá ser reutilizado para um inode diferente.

Se o iput perceber que não há referências de ponteiro C para um inode e que o inode não tem links para ele (não ocorre em nenhum diretório), então o inode e seus blocos de dados devem ser liberados. O iput chama o itrunc para truncar o arquivo para zero bytes, liberando os blocos de dados; define o tipo de inode como 0 (não alocado); e grava o inode no disco (kernel/fs.c:342).

O protocolo de bloqueio em iput, no caso em que libera o inode, merece uma análise mais detalhada. Um perigo é que uma thread concorrente possa estar aguardando em ilock para usar este inode (por exemplo, para ler um arquivo ou listar um diretório) e não esteja preparada para descobrir que o inode não está mais alocado. Isso não pode acontecer porque não há como uma chamada de sistema obter um ponteiro para um inode na memória se não houver links para ele e ip->ref for um. Essa referência é a referência pertencente à thread que chama iput. É verdade que iput verifica se a contagem de referências é um fora de sua seção crítica itable.lock , mas nesse ponto a contagem de links é conhecida como zero, então nenhuma thread tentará adquirir uma nova referência. O outro perigo principal é que uma chamada simultânea para ialloc pode escolher o mesmo inode que iput está liberando. Isso só pode acontecer depois que iupdate grava no disco de forma que o inode tenha o tipo zero. Essa corrida é benigna; o thread de alocação esperará educadamente para adquirir o bloqueio de espera do inode antes de ler ou escrever no inode, momento em que o iput termina com ele.

iput() pode gravar no disco. Isso significa que qualquer chamada de sistema que utilize o sistema de arquivos pode gravar no disco, pois a chamada de sistema pode ser a última a ter uma referência ao arquivo. Mesmo chamadas como read() , que parecem ser somente leitura, podem acabar chamando iput(). Isso, por sua vez, significa que mesmo chamadas de sistema somente leitura devem ser encapsuladas em transações se utilizarem o sistema de arquivos.

Há uma interação desafiadora entre iput() e travamentos. iput() não trunca um arquivo imediatamente quando a contagem de links para o arquivo cai para zero, porque algum processo pode ainda conter uma referência ao inode na memória: um processo pode ainda estar lendo e gravando no arquivo, pois o abriu com sucesso. Mas, se ocorrer um travamento antes que o último processo feche o descritor de arquivo para o arquivo, o arquivo será marcado como alocado em disco, mas nenhuma entrada de diretório apontará para ele.

Os sistemas de arquivos lidam com esse caso de duas maneiras. A solução mais simples é que, na recuperação, após a reinicialização, o sistema de arquivos varre todo o sistema de arquivos em busca de arquivos marcados como alocados, mas sem nenhuma entrada de diretório apontando para eles. Se houver algum arquivo desse tipo, ele poderá liberá-lo.

A segunda solução não requer a varredura do sistema de arquivos. Nesta solução, o sistema de arquivos registra em disco (por exemplo, no superbloco) o i-número de inode de um arquivo cuja contagem de links cai para zero, mas cuja contagem de referências não é zero. Se o sistema de arquivos remover o arquivo quando sua contagem de referências chegar a 0, ele atualizará a lista em disco removendo esse inode da lista. Na recuperação, o sistema de arquivos libera qualquer arquivo da lista.

O Xv6 não implementa nenhuma das soluções, o que significa que os inodes podem ser marcados como alocados no disco, mesmo que não estejam mais em uso. Isso significa que, com o tempo, o Xv6 corre o risco de ficar sem espaço em disco.

8.10 Código: Conteúdo Inode

A estrutura do inode no disco, struct dinode, contém um tamanho e uma matriz de números de bloco (veja a Figura 8.3). Os dados do inode são encontrados nos blocos listados na matriz addrs do dinode . Os primeiros blocos de dados NDIRECT são listados nas primeiras entradas NDIRECT da matriz; esses blocos são chamados de blocos diretos. Os próximos blocos de dados NINDIRECT são listados não no inode, mas em um bloco de dados chamado bloco indireto. A última entrada na matriz addrs fornece o endereço do bloco indireto.

Assim, os primeiros 12 kB (NDIRECT x BSIZE) bytes de um arquivo podem ser carregados a partir de blocos listados no inode, enquanto os próximos 256 kB (NINDIRECT x BSIZE) bytes só podem ser carregados após consultar o bloco indireto. Esta é uma boa representação em disco, mas complexa para clientes. A função bmap gerencia a representação para que rotinas de nível superior, como readi e writei, que veremos em breve, não precisem gerenciar essa complexidade. bmap retorna o número do bloco de disco do bn'ésimo bloco de dados para o inode ip. Se ip ainda não tiver um bloco desse tipo, bmap aloca um.

A função bmap (kernel/fs.c:383) começa escolhendo o caso fácil: os primeiros blocos NDIRECT são listados no próprio inode (kernel/fs.c:388-396). Os próximos blocos NINDIRECT são listados no bloco indireto em ip->addrs[NDIRECT]. bmap lê o bloco indireto (kernel/fs.c:407) e então lê um número de bloco da posição correta dentro do bloco (kernel/fs.c:408). Se o número de blocos exceder NDIRECT+NINDIRECT, o bmap entra em pânico; writei contém a verificação que impede que isso aconteça (kernel/fs.c:513). O bmap aloca blocos conforme necessário. Um comando ip->addrs[] ou uma entrada indireta de zero indica que

nenhum bloco foi alocado. À medida que o bmap encontra zeros, ele os substitui pelo número de blocos novos,

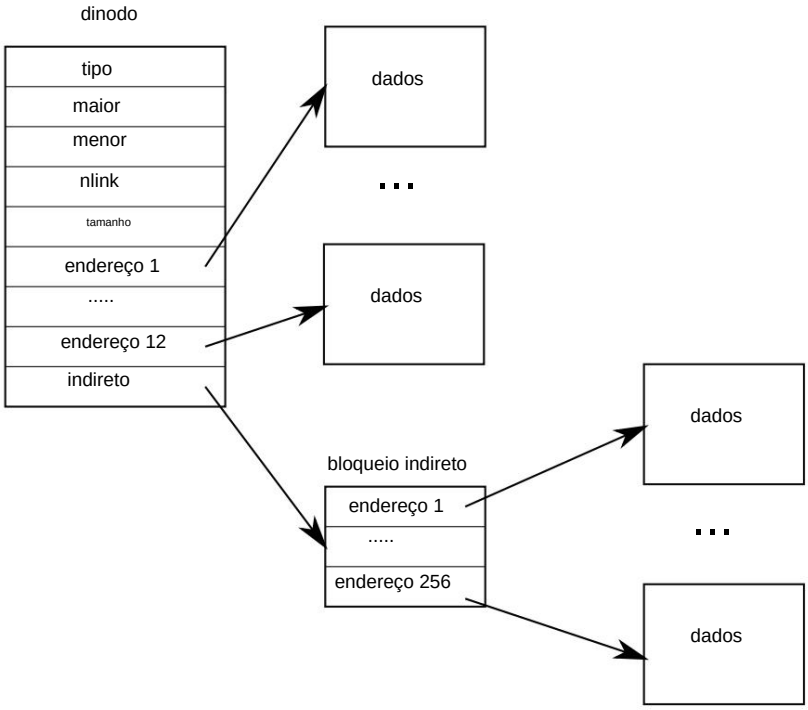

Figura 8.3: A representação de um arquivo no disco.

alocado sob demanda (kernel/fs.c:389-390) (kernel/fs.c:401-402).

itrunc libera os blocos de um arquivo, redefinindo o tamanho do inode para zero. itrunc (kernel/fs.c:426) começa liberando os blocos diretos (kernel/fs.c:432-437), então os listados no bloco indireto (kernel/fs.c:442-445), e finalmente o próprio bloco indireto (kernel/fs.c:447-448).

O bmap facilita para readi e writei obterem os dados de um inode. readi (kernel/fs.c:472) começa certificando-se de que o deslocamento e a contagem não ultrapassem o final do arquivo. Leituras que começam além do final do arquivo retornam um erro (kernel/fs.c:477-478) enquanto leituras que começam ou cruzam o final do arquivo retornam menos bytes do que o solicitado (kernel/fs.c:479-480). O loop principal processa cada bloco do arquivo, copiando dados do buffer para dst (kernel/fs.c:482-494). escrever (kernel/fs.c:506) é idêntico ao readi, com três exceções: gravações que começam ou cruzam o final do arquivo aumentam o arquivo até o tamanho máximo do arquivo (kernel/fs.c:513-514); o loop copia dados para os buffers em vez de para fora (kernel/fs.c:36); e se a gravação estendeu o arquivo, o writei deve atualizar seu tamanho

A função stati (kernel/fs.c:458) copia metadados do inode para a estrutura stat , que é exposta aos programas do usuário por meio da chamada de sistema stat .

8.11 Código: camada de diretório

Um diretório é implementado internamente de forma muito semelhante a um arquivo. Seu inode é do tipo T_DIR e seus dados são uma sequência de entradas de diretório. Cada entrada é uma struct dirent (kernel/fs.h:56). que contém um nome e um número de inode. O nome tem no máximo DIRSIZ (14) caracteres; se for menor, é terminado por um byte NULL (0). Entradas de diretório com número de inode zero são livres.

A função dirlookup (kernel/fs.c:552) pesquisa em um diretório por uma entrada com o nome fornecido. Se encontrar um, ele retorna um ponteiro para o inode correspondente, desbloqueado, e define *poff como o deslocamento de bytes da entrada dentro do diretório, caso o chamador deseje editá-la. Se dirlookup encontrar uma entrada com o nome correto, ele atualiza *poff e retorna um inode desbloqueado obtido via iget. dirlookup é o motivo pelo qual iget retorna inodes desbloqueados. O chamador bloqueou dp, então se a busca fosse por um alias para o diretório atual, tentar bloquear o inode antes de retornar tentaria bloquear dp novamente e causar um deadlock. (Existem cenários de deadlock mais complicados envolvendo múltiplos processos e este não é o único problema.) O chamador pode desbloquear dp e então bloquear ip, garantindo que ele mantenha apenas um bloqueio por vez.

A função dirlink (kernel/fs.c:580) Grava uma nova entrada de diretório com o nome e o número de entrada fornecidos no diretório dp. Se o nome já existir, dirlink retornará um erro (kernel/fs.c:586-590). O loop principal lê entradas de diretório em busca de uma entrada não alocada. Ao encontrar uma, ele interrompe o loop antecipadamente (kernel/fs.c:563-564). com offset definido como o deslocamento da entrada disponível. Caso contrário, o loop termina com offset definido como dp->size. De qualquer forma, dirlink adiciona uma nova entrada ao diretório escrevendo no deslocamento off

8.12 Código: Nomes de caminhos

A consulta de nome de caminho envolve uma sucessão de chamadas para dirlookup, uma para cada componente de caminho. namei (kernel/fs.c:687) avalia o caminho e retorna o inode correspondente. A função nameiparent é uma variante: ela para antes do último elemento, retornando o inode do diretório pai e copiando o elemento final para name. Ambas chamam a função generalizada namex para fazer o trabalho real.

namex (kernel/fs.c:652) começa decidindo onde a avaliação do caminho começa. Se o caminho começar com uma barra, a avaliação começa na raiz; caso contrário, no diretório atual (kernel/fs.c:656-659).
Em seguida, ele usa skipelem para considerar cada elemento do caminho por vez (kernel/fs.c:661). Cada iteração do loop deve procurar o nome no IP do inode atual. A iteração começa bloqueando o IP e verificando se ele é um diretório. Caso contrário, a busca falha (kernel/fs.c:662-666). (O bloqueio de ip é necessário não porque ip->type possa mudar rapidamente — ele não pode — mas porque, até que ilock seja executado, não há garantia de que ip->type tenha sido carregado do disco.) Se a chamada for nameiparent e este for o último elemento do caminho, o loop para mais cedo, conforme a definição de nameiparent; o elemento final do caminho já foi copiado para name, então namex precisa retornar apenas o ip desbloqueado (kernel/fs.c:667-671).
Por fim, o loop procura o elemento do caminho usando dirlookup e se prepara para a próxima iteração definindo ip = next (kernel/fs.c:672-677). Quando o loop fica sem elementos de caminho, ele retorna ip.

O procedimento namex pode levar muito tempo para ser concluído: pode envolver várias operações de disco para ler inodes e blocos de diretório para os diretórios percorridos no caminho (se não estiverem no cache de buffer). O Xv6 foi cuidadosamente projetado para que, se uma invocação de namex por um kernel

Se o thread estiver bloqueado em uma E/S de disco, outro thread do kernel pesquisando um caminho diferente poderá prosseguir simultaneamente. O namex bloqueia cada diretório no caminho separadamente para que as pesquisas em diretórios diferentes possam prosseguir em paralelo.

Essa simultaneidade apresenta alguns desafios. Por exemplo, enquanto uma thread do kernel está procurando um caminho, outra thread do kernel pode estar alterando a árvore de diretórios desvinculando um diretório.
Um risco potencial é que uma pesquisa pode estar pesquisando um diretório que foi excluído por outro thread do kernel e seus blocos foram reutilizados para outro diretório ou arquivo.

O Xv6 evita tais corridas. Por exemplo, ao executar dirlookup no namex, a thread de pesquisa mantém o bloqueio no diretório e o dirlookup retorna um inode obtido usando o iget. O iget aumenta a contagem de referências do inode. Somente após receber o inode do dirlookup o namex libera o bloqueio no diretório. Agora, outra thread pode desvincular o inode do diretório, mas o xv6 não excluirá o inode ainda, porque a contagem de referências do inode ainda é maior que zero.

Outro risco é o deadlock. Por exemplo, next aponta para o mesmo inode que ip ao procurar por ".". Bloquear next antes de liberar o bloqueio em ip resultaria em um deadlock. Para evitar esse deadlock, o namex desbloqueia o diretório antes de obter um bloqueio em next. Aqui, novamente, vemos por que a separação entre iget e ilock é importante.

8.13 Camada de descritor de arquivo

Um aspecto interessante da interface Unix é que a maioria dos recursos no Unix são representados como arquivos, incluindo dispositivos como o console, pipes e, claro, arquivos reais. A camada do descritor de arquivo é a camada que garante essa uniformidade.

O Xv6 dá a cada processo sua própria tabela de arquivos abertos, ou descritores de arquivos, como vimos no Capítulo 1. Cada arquivo aberto é representado por um arquivo struct (kernel/file.h:1), que é um wrapper em torno de um inode ou um pipe, mais um deslocamento de E/S. Cada chamada para open cria um novo arquivo aberto (um novo arquivo struct): se vários processos abrirem o mesmo arquivo independentemente, as diferentes instâncias terão diferentes deslocamentos de E/S. Por outro lado, um único arquivo aberto (o mesmo arquivo struct) pode aparecer várias vezes na tabela de arquivos de um processo e também nas tabelas de arquivos de vários processos. Isso aconteceria se um processo usasse open para abrir o arquivo e, em seguida, criasse aliases usando dup ou o compartilhasse com um filho usando fork. Uma contagem de referência rastreia o número de referências a um arquivo aberto específico. Um arquivo pode ser aberto para leitura ou escrita, ou ambos. Os campos readable e writable rastreiam isso.

Todos os arquivos abertos no sistema são mantidos em uma tabela de arquivos global, a ftable. A tabela de arquivos possui funções para alocar um arquivo (filealloc), criar uma referência duplicada (filedup), liberar uma referência (fileclose) e ler e gravar dados (fileread e filewrite).

Os três primeiros seguem o formato agora familiar: filealloc (kernel/file.c:30) verifica a tabela de arquivos em busca de um arquivo não referenciado (f->ref == 0) e retorna uma nova referência; filedup (kernel/file.c:48) incrementa a contagem de referência; e fileclose (kernel/file.c:60) decrementa-o. Quando a contagem de referências de um arquivo chega a zero, fileclose libera o pipe ou inode subjacente, de acordo com o tipo.

As funções filestat, fileread e filewrite implementam as operações stat, read e write em arquivos. filestat (kernel/file.c:88) é permitido apenas em inodes e chama stati. fileread

e filewrite verificam se a operação é permitida pelo modo aberto e, em seguida, passam a chamada para a implementação do pipe ou do inode. Se o arquivo representar um inode, fileread e filewrite usam o deslocamento de E/S como deslocamento para a operação e, em seguida, o avançam (kernel/file.c:122-123) (kernel/ file.c:153-154). Pipes não têm o conceito de offset. Lembre-se de que as funções inode exigem que o chamador trate do bloqueio (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166). O bloqueio in-ode tem o efeito colateral conveniente de que os deslocamentos de leitura e gravação são atualizados atomicamente, de modo que várias gravações no mesmo arquivo simultaneamente não podem substituir os dados umas das outras, embora suas gravações possam acabar entrelaçadas.

8.14 Código: Chamadas de sistema

Com as funções que as camadas inferiores fornecem, a implementação da maioria das chamadas de sistema é trivial (veja (kernel/sysfile.c)). Há algumas chamadas que merecem uma análise mais detalhada.

As funções sys_link e sys_unlink editam diretórios, criando ou removendo referências a inodes. Elas são outro bom exemplo do poder do uso de transações. sys_link (kernel/sys- file.c:124) começa buscando seus argumentos, duas strings old e new (kernel/sysfile.c:129). Supondo que old exista e não seja um diretório (kernel/sysfile.c:133-136), sys_link incrementa sua contagem ip->nlink . Em seguida, sys_link chama nameiparent para encontrar o diretório pai e o elemento de caminho final do novo (kernel/sysfile.c:149). e cria uma nova entrada de diretório apontando para o inode do antigo (kernel/sys- file.c:152). O novo diretório pai deve existir e estar no mesmo dispositivo que o inode existente: os números de inode só têm um significado único em um único disco. Se ocorrer um erro como esse, o sys_link deve retornar e decrementar ip->nlink.

As transações simplificam a implementação porque exigem a atualização de vários blocos de disco, mas não precisamos nos preocupar com a ordem em que as executamos. Ou todas serão bem-sucedidas ou nenhuma. Por exemplo, sem transações, atualizar ip->nlink antes de criar um link colocaria o sistema de arquivos temporariamente em um estado inseguro, e uma falha no meio do processo poderia causar estragos. Com transações, não precisamos nos preocupar com isso.

sys_link cria um novo nome para um inode existente. A função create (kernel/sysfile.c:246) Cria um novo nome para um novo inode. É uma generalização das três chamadas de sistema de criação de arquivos: open com o sinalizador O_CREATE cria um novo arquivo comum, mkdir cria um novo diretório e mkdev cria um novo arquivo de dispositivo. Assim como sys_link, create começa chamando nameiparent para obter o inode do diretório pai. Em seguida, chama dirlookup para verificar se o nome já existe (kernel/sysfile.c:256). Se o nome existir, o comportamento de create depende da chamada de sistema para a qual ele está sendo usado: open tem semântica diferente de mkdir e mkdev. Se create estiver sendo usado em nome de open (tipo == T_FILE) e o nome existente for um arquivo comum, open o trata como um sucesso, assim como create (kernel/sysfile.c:260). Caso contrário, é um erro (kernel/sysfile.c:261-262). Se o nome ainda não existir, o create agora aloca um novo inode com ialloc (kernel/sysfile.c:265). Se o novo inode for um diretório, o create o inicializa com . e entradas. Finalmente, agora que os dados foram inicializados corretamente, o create pode vinculá-los ao diretório pai (kernel/sysfile.c:278). O comando create, assim como o sys_link, mantém dois bloqueios de inode simultaneamente: ip e dp. Não há possibilidade de deadlock porque o ip do inode é recém-alocado: nenhum outro processo no sistema manterá o bloqueio de ip e tentará bloquear dp.

..

Usando o create, é fácil implementar sys_open, sys_mkdir e sys_mknod. sys_open (kernel/sysfile.c:305) é o mais complexo, pois criar um novo arquivo é apenas uma pequena parte do que ele pode fazer. Se o comando open receber o sinalizador O_CREATE , ele chama create (kernel/sysfile.c:320). Caso contrário, ele chama namei (kernel/sysfile.c:326). create retorna um inode bloqueado, mas namei não, então sys_open deve bloquear o próprio inode. Isso fornece um local conveniente para verificar se os diretórios estão abertos apenas para leitura, não para escrita. Supondo que o inode tenha sido obtido de uma forma ou de outra, sys_open aloca um arquivo e um descritor de arquivo (kernel/sysfile.c:344). e então preenche o arquivo (kernel/sysfile.c:356- 361). Observe que nenhum outro processo pode acessar o arquivo parcialmente inicializado, pois ele está apenas na tabela do processo atual.

O Capítulo 7 examinou a implementação de pipes antes mesmo de termos um sistema de arquivos. A função sys_pipe conecta essa implementação ao sistema de arquivos, fornecendo uma maneira de criar um par de pipes.
Seu argumento é um ponteiro para o espaço de dois inteiros, onde ele registrará os dois novos descritores de arquivo.
Em seguida, ele aloca o pipe e instala os descritores de arquivo.

8.15 Mundo real

O cache de buffer em um sistema operacional do mundo real é significativamente mais complexo que o do xv6, mas atende aos mesmos dois propósitos: armazenar em cache e sincronizar o acesso ao disco. O cache de buffer do Xv6, assim como o do V6, usa uma política simples de remoção de dados menos usados recentemente (LRU); há muitas políticas mais complexas que podem ser implementadas, cada uma adequada para algumas cargas de trabalho e não tão adequada para outras. Um cache LRU mais eficiente eliminaria a lista encadeada, usando, em vez disso, uma tabela de hash para consultas e um heap para remoção de LRU. Os caches de buffer modernos são normalmente integrados ao sistema de memória virtual para suportar arquivos mapeados na memória.

O sistema de registro do Xv6 é ineficiente. Uma confirmação não pode ocorrer simultaneamente com chamadas de sistema de arquivos. O sistema registra blocos inteiros, mesmo que apenas alguns bytes em um bloco sejam alterados. Ele realiza gravações de registro síncronas, um bloco por vez, cada uma das quais provavelmente exigirá um tempo inteiro de rotação do disco. Sistemas de registro reais resolvem todos esses problemas.

O registro em log não é a única maneira de fornecer recuperação de falhas. Os primeiros sistemas de arquivos utilizavam um scavenger durante a reinicialização (por exemplo, o programa fsck do UNIX) para examinar cada arquivo e diretório, bem como as listas de blocos e inodes livres, procurando e resolvendo inconsistências. A varredura pode levar horas para sistemas de arquivos grandes, e há situações em que não é possível resolver inconsistências de forma que as chamadas de sistema originais sejam atômicas. A recuperação a partir de um log é muito mais rápida e faz com que as chamadas de sistema sejam atômicas em caso de falhas.

O Xv6 usa o mesmo layout básico de inodes e diretórios em disco do UNIX antigo; esse esquema tem se mantido notavelmente persistente ao longo dos anos. O UFS/FFS do BSD e o ext2/ext3 do Linux usam essencialmente as mesmas estruturas de dados. A parte mais ineficiente do layout do sistema de arquivos é o diretório, que requer uma varredura linear em todos os blocos do disco durante cada consulta. Isso é razoável quando os diretórios são apenas alguns blocos de disco, mas é caro para diretórios que contêm muitos arquivos. O NTFS do Microsoft Windows, o HFS do macOS e o ZFS do Solaris, apenas para citar alguns, implementam um diretório como uma árvore balanceada de blocos em disco. Isso é complicado, mas garante consultas de diretório em tempo logarítmico.

O Xv6 é ingênuo em relação a falhas de disco: se uma operação de disco falhar, o Xv6 entra em pânico. Se isso é razoável

Depende do hardware: se um sistema operacional estiver instalado em um hardware especial que usa redundância para mascarar falhas de disco, talvez o sistema operacional detecte falhas com tão pouca frequência que entrar em pânico seja aceitável. Por outro lado, sistemas operacionais que usam discos comuns devem esperar falhas e tratá-las com mais cuidado, para que a perda de um bloco em um arquivo não afete o uso do restante do sistema de arquivos.

O Xv6 exige que o sistema de arquivos caiba em um único dispositivo de disco e não mude de tamanho. À medida que grandes bancos de dados e arquivos multimídia aumentam cada vez mais os requisitos de armazenamento, os sistemas operacionais estão desenvolvendo maneiras de eliminar o gargalo de "um disco por sistema de arquivos". A abordagem básica é combinar vários discos em um único disco lógico. Soluções de hardware como RAID ainda são as mais populares, mas a tendência atual é implementar o máximo possível dessa lógica em software. Essas implementações de software normalmente permitem funcionalidades avançadas, como aumentar ou diminuir o dispositivo lógico adicionando ou removendo discos dinamicamente. É claro que uma camada de armazenamento que pode aumentar ou diminuir dinamicamente requer um sistema de arquivos que faça o mesmo: a matriz de blocos inode de tamanho fixo usada pelo xv6 não funcionaria bem em tais ambientes. Separar o gerenciamento de disco do sistema de arquivos pode ser o design mais limpo, mas a interface complexa entre os dois levou alguns sistemas, como o ZFS da Sun, a combiná-los.

O sistema de arquivos do Xv6 não possui muitos outros recursos dos sistemas de arquivos modernos; por exemplo, não possui suporte porta para snapshots e backup incremental.

Os sistemas Unix modernos permitem que muitos tipos de recursos sejam acessados com as mesmas chamadas de sistema usadas no armazenamento em disco: pipes nomeados, conexões de rede, sistemas de arquivos de rede acessados remotamente e interfaces de monitoramento e controle, como /proc. Em vez das instruções if do xv6 em fileread e filewrite, esses sistemas normalmente fornecem a cada arquivo aberto uma tabela de ponteiros de função, um por operação, e chamam o ponteiro de função para invocar a implementação da chamada naquele inode. Sistemas de arquivos de rede e sistemas de arquivos em nível de usuário fornecem funções que transformam essas chamadas em RPCs de rede e aguardam a resposta antes de retornar.

8.16 Exercícios

1. Por que entrar em pânico no balloc ? O xv6 pode se recuperar?

2. Por que entrar em pânico no ialloc ? O xv6 pode se recuperar?

3. Por que o filealloc não entra em pânico quando fica sem arquivos? Por que isso é mais comum e, portanto, vale a pena lidar com isso?

4. Suponha que o arquivo correspondente a ip seja desvinculado por outro processo entre as chamadas de sys_link para iunlock(ip) e dirlink. O link será criado corretamente? Por quê?

5. create faz quatro chamadas de função (uma para ialloc e três para dirlink) necessárias para ter sucesso. Se alguma não for bem-sucedida, create chama panic. Por que isso é aceitável? Por que nenhuma dessas quatro chamadas pode falhar?

6. sys_chdir chama iunlock(ip) antes de iput(cp->cwd), o que pode tentar bloquear cp->cwd, mas adiar iunlock(ip) para depois de iput não causaria deadlocks. Por que não?

7. Implemente a chamada de sistema lseek . O suporte a lseek também exigirá que você modifique filewrite para preencher lacunas no arquivo com zero se lseek ultrapassar f->ip->size .
8. Adicione O_TRUNC e O_APPEND para abrir, para que os operadores > e >> funcionem no shell.
9. Modifique o sistema de arquivos para suportar links simbólicos.
10. Modifique o sistema de arquivos para suportar pipes nomeados.
11. Modifique o sistema de arquivos e VM para suportar arquivos mapeados na memória.

Capítulo 9

Concorrência revisitada

Obter simultaneamente um bom desempenho paralelo, correção apesar da concorrência e código compreensível é um grande desafio no projeto do kernel. O uso direto de bloqueios é o melhor caminho para a correção, mas nem sempre é possível. Este capítulo destaca exemplos em que o xv6 é forçado a usar bloqueios de forma complexa e exemplos em que o xv6 usa técnicas semelhantes a bloqueios, mas não bloqueios.

9.1 Padrões de bloqueio

Itens em cache costumam ser um desafio para bloquear. Por exemplo, o cache de blocos do sistema de arquivos (kernel /bio.c:26) armazena cópias de até NBUF de blocos de disco. É vital que um determinado bloco de disco tenha no máximo uma cópia no cache; caso contrário, processos diferentes podem fazer alterações conflitantes em cópias diferentes do que deveria ser o mesmo bloco. Cada bloco em cache é armazenado em uma struct buf (kernel/buf.h:1). Uma struct buf possui um campo de bloqueio que ajuda a garantir que apenas um processo use um determinado bloco de disco por vez. No entanto, esse bloqueio não é suficiente: e se um bloco não estiver presente no cache e dois processos quiserem usá-lo ao mesmo tempo? Não há struct buf (já que o bloco ainda não está armazenado em cache) e, portanto, não há nada para bloquear. O Xv6 lida com essa situação associando um bloqueio adicional (bcache.lock) ao conjunto de identidades dos blocos armazenados em cache.
Código que precisa verificar se um bloco está armazenado em cache (por exemplo, bget (kernel/bio.c:59)), ou alterar o conjunto de blocos em cache, deve conter bcache.lock; depois que o código encontrar o bloco e a estrutura buf necessários, ele pode liberar bcache.lock e bloquear apenas o bloco específico. Este é um padrão comum: um bloqueio para o conjunto de itens, mais um bloqueio por item.

Normalmente, a mesma função que adquire um bloqueio o libera. Mas uma maneira mais precisa de ver as coisas é que um bloqueio é adquirido no início de uma sequência que deve parecer atômica e liberado quando essa sequência termina. Se a sequência começa e termina em funções diferentes, ou threads diferentes, ou em CPUs diferentes, então a aquisição e a liberação do bloqueio devem fazer o mesmo. A função do bloqueio é forçar outros usuários a esperar, não fixar um dado a um agente específico. Um exemplo é a aquisição em yield (kernel/proc.c:503). que é liberado na thread do agendador e não no processo de aquisição. Outro exemplo é o acquiresleep em ilock (kernel/fs.c:293); esse código geralmente fica em modo de espera durante a leitura do disco; ele pode acordar em uma CPU diferente, o que significa que o bloqueio pode ser adquirido e liberado em CPUs diferentes.

Liberar um objeto protegido por uma trava embutida no objeto é uma tarefa delicada, pois possuir a trava não é suficiente para garantir que a liberação seja correta. O caso problemático surge quando alguma outra thread está aguardando na aquisição para usar o objeto; liberar o objeto implicitamente libera a trava embutida, o que causará o mau funcionamento da thread em espera. Uma solução é rastrear quantas referências ao objeto existem, de modo que ele só seja liberado quando a última referência desaparecer. Veja pipeclose (kernel/pipe.c:59). por exemplo; pi->readopen e pi->writeopen rastreiam se o pipe tem descritores de arquivo que se referem a ele.

Geralmente, vemos bloqueios em torno de sequências de leituras e gravações em conjuntos de itens relacionados; os bloqueios garantem que outros threads vejam apenas sequências completas de atualizações (desde que eles também sejam bloqueados). E quanto às situações em que a atualização é uma simples gravação em uma única variável compartilhada? Por exemplo, setkilled e killed (kernel/proc.c:607). bloqueio em torno de seus usos simples de p->killed. Se não houvesse bloqueio, uma thread poderia escrever p->killed ao mesmo tempo em que outra thread o lê. Isso é uma corrida, e a especificação da linguagem C diz que uma corrida produz um comportamento indefinido, o que significa que o programa pode travar ou gerar resultados incorretos . Os bloqueios impedem a corrida e evitam o comportamento indefinido.

Um dos motivos pelos quais corridas podem interromper programas é que, se não houver travas ou construções equivalentes, o compilador pode gerar código de máquina que lê e grava na memória de maneiras bem diferentes do código C original. Por exemplo, o código de máquina de uma thread que chama killed poderia copiar p->killed para um registrador e ler apenas esse valor armazenado em cache; isso significaria que a thread poderia nunca ver nenhuma gravação em p->killed. As travas impedem esse armazenamento em cache.

9.2 Padrões semelhantes a fechaduras

Em muitos casos, o xv6 usa uma contagem de referências ou um sinalizador, semelhante a um bloqueio, para indicar que um objeto está alocado e não deve ser liberado ou reutilizado. O p->state de um processo atua dessa maneira, assim como as contagens de referências em estruturas de arquivo, inode e buf. Embora em cada caso um bloqueio proteja o sinalizador ou a contagem de referências, é este último que impede que o objeto seja liberado prematuramente.

O sistema de arquivos usa contagens de referência de inode de estrutura como um tipo de bloqueio compartilhado que pode ser mantido por vários processos, a fim de evitar deadlocks que ocorreriam se o código usasse bloqueios comuns. Por exemplo, o loop em namex (kernel/fs.c:652) bloqueia o diretório nomeado por cada componente do caminho por vez. No entanto, o namex deve liberar cada bloqueio ao final do loop, pois, se ele contivesse vários bloqueios, poderia entrar em deadlock consigo mesmo se o caminho incluísse um ponto (por exemplo, a/./b). Também poderia entrar em deadlock com uma consulta simultânea envolvendo o diretório e, como explica o Capítulo 8, a solução é que o loop carregue o inode do diretório para a próxima iteração com sua contagem de referências incrementada, mas não bloqueada.

Alguns itens de dados são protegidos por mecanismos diferentes em momentos diferentes e podem, às vezes, ser protegidos contra acesso simultâneo implicitamente pela estrutura do código xv6, em vez de por bloqueios explícitos. Por exemplo, quando uma página física está livre, ela é protegida por kmem.lock (kernel/kalloc.c:24). Se a página for então alocada como um pipe (kernel/pipe.c:23), é protegida por um bloqueio diferente (o pi->lock incorporado). Se a página for realocada para a memória do usuário de um novo processo, ela não será protegida

[1] “Threads e corridas de dados” em https://en.cppreference.com/w/c/language/memory_model

por um bloqueio. Em vez disso, o fato de o alocador não ceder essa página a nenhum outro processo (até que ela seja liberada) a protege do acesso concorrente. A propriedade da memória de um novo processo é complexa: primeiro, o pai a aloca e manipula em um fork, depois o filho a utiliza e (após a saída do filho) o pai novamente se torna dono da memória e a passa para o kfree. Há duas lições aqui: um objeto de dados pode ser protegido da concorrência de diferentes maneiras em diferentes momentos de sua vida útil, e a proteção pode assumir a forma de uma estrutura implícita em vez de bloqueios explícitos.

Um exemplo final de bloqueio é a necessidade de desabilitar interrupções em torno de chamadas para mycpu() (kernel /proc.c:83). Desabilitar interrupções faz com que o código de chamada seja atômico em relação às interrupções do temporizador, o que poderia forçar uma troca de contexto e, assim, mover o processo para uma CPU diferente.

9.3 Nenhuma fechadura

Existem alguns lugares onde o xv6 compartilha dados mutáveis sem nenhum bloqueio. Um deles é na implementação de spinlocks, embora se possa considerar as instruções atômicas do RISC-V como dependentes de bloqueios implementados em hardware. Outro é a variável iniciada em main.c (kernel/main.c:7). usado para impedir que outras CPUs sejam executadas até que a CPU zero tenha concluído a inicialização do xv6; o volátil garante que o compilador realmente gere instruções de carregamento e armazenamento.

Xv6 contém casos em que uma CPU ou thread grava alguns dados e outra CPU ou thread lê os dados, mas não há um bloqueio específico dedicado à proteção desses dados. Por exemplo, em um fork, o pai grava as páginas de memória do usuário do filho, e o filho (uma thread diferente, talvez em uma CPU diferente) lê essas páginas; nenhum bloqueio protege explicitamente essas páginas. Isso não é estritamente um problema de bloqueio, já que o filho não inicia a execução até que o pai termine de escrever. É um potencial problema de ordenação de memória (consulte o Capítulo 6), pois sem uma barreira de memória não há razão para esperar que uma CPU veja as escritas de outra CPU. No entanto, como o pai libera bloqueios e o filho adquire bloqueios ao iniciar, as barreiras de memória em acquire e release garantem que a CPU do filho veja as escritas do pai.

9.4 Paralelismo

O bloqueio visa principalmente suprimir o paralelismo em prol da correção. Como o desempenho também é importante, os projetistas do kernel frequentemente precisam pensar em como usar bloqueios de forma a atingir a correção e permitir o paralelismo. Embora o xv6 não seja sistematicamente projetado para alto desempenho, ainda vale a pena considerar quais operações do xv6 podem ser executadas em paralelo e quais podem entrar em conflito com os bloqueios.

Pipes no xv6 são um exemplo de paralelismo bastante bom. Cada pipe possui seu próprio bloqueio, permitindo que diferentes processos leiam e escrevam em pipes diferentes em paralelo em CPUs diferentes. Para um determinado pipe, no entanto, o escritor e o leitor devem esperar que um do outro libere o bloqueio; eles não podem ler/escrever no mesmo pipe ao mesmo tempo. Também é possível que uma leitura de um pipe vazio (ou uma escrita em um pipe cheio) seja bloqueada, mas isso não se deve ao esquema de bloqueio.

A troca de contexto é um exemplo mais complexo. Duas threads do kernel, cada uma executando em sua própria CPU, podem chamar yield, sched e swtch simultaneamente, e as chamadas serão executadas em paralelo.

Cada thread segura uma trava, mas são travas diferentes, então não precisam esperar uma pela outra.
No entanto, uma vez no agendador, as duas CPUs podem entrar em conflito com bloqueios ao procurar na tabela de processos por uma que seja EXECUTÁVEL. Ou seja, o xv6 provavelmente obterá um benefício de desempenho com múltiplas CPUs durante a troca de contexto, mas talvez não tanto quanto poderia.

Outro exemplo são chamadas simultâneas para bifurcação de processos diferentes em CPUs diferentes. As chamadas podem ter que esperar uma pela outra por pid_lock e kmem.lock, e por bloqueios por processo necessários para pesquisar na tabela de processos por um processo NÃO UTILIZADO. Por outro lado, os dois processos de bifurcação podem copiar páginas de memória do usuário e formatar páginas da tabela de páginas totalmente em paralelo.

O esquema de bloqueio em cada um dos exemplos acima sacrifica o desempenho paralelo em certos casos. Em cada caso, é possível obter mais paralelismo usando um design mais elaborado. Se vale a pena depende de detalhes: com que frequência as operações relevantes são invocadas, quanto tempo o código passa com um bloqueio contencioso mantido, quantas CPUs podem estar executando operações conflitantes ao mesmo tempo, se outras partes do código são gargalos mais restritivos. Pode ser difícil adivinhar se um determinado esquema de bloqueio pode causar problemas de desempenho ou se um novo design é significativamente melhor, portanto, a medição em cargas de trabalho realistas é frequentemente necessária.

9.5 Exercícios

1. Modifique a implementação do pipe do xv6 para permitir que uma leitura e uma gravação no mesmo pipe ocorram em paralelo em núcleos diferentes.

2. Modifique o scheduler() do xv6 para reduzir a contenção de bloqueio quando diferentes núcleos estiverem procurando por processos executáveis ao mesmo tempo.

3. Elimine parte da serialização no fork() do xv6.

Capítulo 10

Resumo

Este texto apresentou as principais ideias em sistemas operacionais, estudando um sistema operacional, o xv6, linha por linha. Algumas linhas de código incorporam a essência das ideias principais (por exemplo, troca de contexto, limite usuário/kernel, bloqueios, etc.) e cada linha é importante; outras linhas de código ilustram como implementar uma ideia específica de sistema operacional e poderiam ser facilmente implementadas de diferentes maneiras (por exemplo, um algoritmo melhor para escalonamento, melhores estruturas de dados em disco para representar arquivos, melhor registro para permitir transações simultâneas, etc.). Todas as ideias foram ilustradas no contexto de uma interface de chamada de sistema específica e muito bem-sucedida, a interface Unix, mas essas ideias são transferidas para o projeto de outros sistemas operacionais.

Bibliografia

[1] Vulnerabilidades e exposições comuns do Linux (CVEs). https://cve.mitre.org/cgi-bin/cvekey.cgi?palavra-chave=linux.

[2] Manual do conjunto de instruções RISC-V Volume I: ISA sem privilégios. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf , 2019 .

[3] Manual do conjunto de instruções RISC-V Volume II: arquitetura privilegiada. https: //github.com/riscv/riscv-isa-manual/releases/download/ Priv-v1.12/riscv-privileged-20211203.pdf, 2021.

[4] Hans-J Boehm. Threads não podem ser implementadas como uma biblioteca. Conferência ACM PLDI, 2005.

[5] Edsger Dijkstra. Processos sequenciais cooperativos. https://www.cs.utexas.edu/usuários/EWD/transcrições/EWD01xx/EWD123.html, 1965.

[6] Maurice Herlihy e Nir Shavit. A Arte da Programação Multiprocessador, Reimpressão Revisada. 2012.

[7] Brian W. Kernighan. A Linguagem de Programação C. Prentice Hall Técnico Profissional Referência, 2ª edição, 1988.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch e Simon Winwood. Sel4: Verificação formal de um kernel de sistema operacional. Em Anais do 22º Simpósio ACM SIGOPS sobre Princípios de Sistemas Operacionais, páginas 207–220, 2009.

[9] Donald Knuth. Algoritmos Fundamentais. A Arte da Programação de Computadores. (Segunda ed.), volume 1. 1997.

[10] L Lamport. Uma nova solução para o problema de programação concorrente de Dijkstra. Comunicações da ACM, 1974.

[11] John Lions. Comentário sobre UNIX 6ª edição. Comunicação ponto a ponto, 2000.

[12] Paul E. Mckenney, Silas Boyd-wickizer e Jonathan Walpole. Uso de RCU no Linux kernel: Uma década depois, 2013.

[13] Martin Michael e Daniel Durich. NS16550A: considerações sobre o projeto e a aplicação da UART. http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0491.pdf , 1987.

[14] Aleph One. Quebrando a pilha por diversão e lucro. http://phrack.org/issues/49/14.html#artigo.

[15] David Patterson e Andrew Waterman. O leitor RISC-V: um atlas de arquitetura aberta. Strawberry Canyon, 2017.

[16] Dave Presotto, Rob Pike, Ken Thompson e Howard Trickey. Plan 9, um sistema distribuído. Em Anais da Conferência EurOpen da Primavera de 1991, páginas 43–50, 1991.

[17] Dennis M. Ritchie e Ken Thompson. O sistema de compartilhamento de tempo UNIX. Comun. ACM, 17(7):365–375, julho de 1974.

Índice