xv6:
シンプルでUnixライクな教育用オペレーティングシステム

ラス・コックスフランス・カアショクロバート

・モリス2019年10月27日

コンテンツ

序文と謝辞

これは、オペレーティングシステムのクラスのためのテキストの草稿です。xv6はDennis RitchieとKen ThompsonのUnix Version 6 (v6) [10]を再実装したものです。xv6はv6の構造とスタイルを緩やかに踏襲していますが、マルチコアのRISC-V [9]用にANSI C [5]で実装されています。

このテキストはxv6のソースコードと一緒に読んでください。これはJohn LionsのCommentary on UNIX 6th Edition [7]に触発されたアプローチです。xv6を使ったいくつかの実践的な宿題を含むv6とxv6のオンラインリソースへのポインタについては `https://pdos.csail.mit.edu/6.828` を参照してください。

私たちはこのテキストを、MITのオペレーティングシステムのクラスである6.828で使用しました。xv6に直接または間接的に貢献してくださった6.828の教授陣、ティーチングアシスタント、学生の皆さんに感謝します。特に、Austin ClementsとNickolai Zeldovichには感謝しています。最後に、本文のバグや改善点をメールで送ってくださった方々に感謝します。Abutalib Aghayev, Sebastian Boehm, Anton Burtsev, Raphael Carvalho, Tej Chajed, Rasit Eskicioglu, Color Fuzzy, Giuseppe, Tao Guo, Robert Hilderman, Wolfgang Keller, Austin Liew, Pavan Maddamsetti,Jacek Masiulaniec, Michael McConville, miguelgvieira, Mark Morrissey, Harry Pan, Askar Safin, Salman Shah, Ruslan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda, tzerbib, Xi Wang, and Zou Chang Wei.

間違いを発見された方、改善のためのご提案をお持ちの方は、Frans KaashoekとRobert Morris(kaashoek,rtm@csail.mit.edu)までメールをお送りください。

第1章

オペレーティングシステムのインターフェース

オペレーティングシステムの役割は、複数のプログラムでコンピュータを共有し、ハードウェアだけでサポートするよりも便利なサービスを提供することです。OSは低レベルのハードウェアを管理し、抽象化することで、例えばワープロは使用されているディスクハードウェアの種類を気にする必要がないようにします。また、複数のプログラム間でハードウェアを共有することで、それらのプログラムが同時に実行される（または実行されているように見える）ようにします。最後に、オペレーティングシステムは、データを共有したり、一緒に作業したりするために、プログラムの相互作用を制御する方法を提供します。

オペレーティングシステムは、インターフェイスを通じてユーザープログラムにサービスを提供します。良いインターフェイスを設計するのは難しいことです。一方では、インターフェイスはシンプルで狭いものであってほしいと思っています。なぜなら、その方が正しい実装を簡単に行うことができるからです。一方で、アプリケーションに多くの洗練された機能を提供したいという気持ちもあります。この問題を解決するためのコツは、いくつかのメカニズムに依存したインターフェースを設計し、それらを組み合わせることで多くの汎用性を持たせることです。本書では、オペレーティングシステムの概念を説明するための具体的な例として、1つのオペレーティングシステムを使用しています。xv6というオペレーティング・システムは、Ken ThompsonとDennis RitchieのUnixオペレーティング・システム[10]で紹介された基本的なインターフェースを提供し、Unixの内部設計を模倣しています。Unixは、メカニズムがうまく組み合わさった狭いインターフェースを提供し、驚くほどの一般性を提供しています。このインターフェイスは非常に成功しており、現代のオペレーティングシステムであるBSD、Linux、Mac OS X、Solaris、さらにはMicrosoft WindowsもUnixに似たインターフェイスを持っています。xv6を理解することは、これらのシステムやその他多くのシステムを理解するための良いスタートとなります。

図1.1に示すように、xv6は伝統的な*カーネル*の形をしており、実行中のプログラムにサービスを提供する特別なプログラムです。実行中のプログラムは*プロセス*と呼ばれ

、命令、データ、スタックを格納したメモリを持っています。命令は、プログラムの計算を実行します。データは、計算の対象となる変数です。スタックは、プログラムのプロシージャコールを整理します。

プロセスがカーネルサービスを呼び出す必要がある場合、プロセスはオペレーティングシステムのインターフェースのプロシージャコールを呼び出します。このような手続きを*システムコール*と呼びます。システムコールはカーネルに入り、カーネルはサービスを実行して戻ります。このようにして、プロセスは*ユーザー空間*と*カーネル空間*を交互に実行することになります。

カーネルは、CPUのハードウェア保護機構を利用して、ユーザースペースで実行される各プロセスが自分自身のメモリのみにアクセスできるようにします。カーネルは、これらの保護機能を実現するために必要なハードウェア特権を持って実行され、ユーザープログラムはその特権を持たずに実行されます。ユーザープログラムが

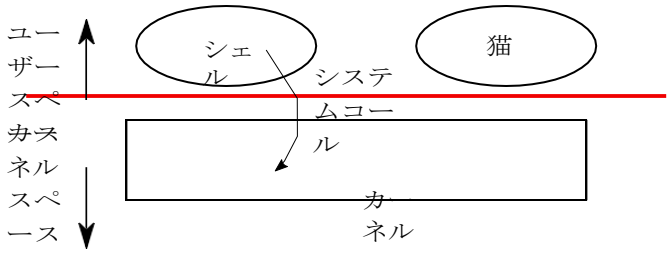


図1.1:カーネルと2つのユーザープロセス。

ユーザープログラムがシステムコールを呼び出すと、ハードウェアは特権レベルを上げ、カーネル内であらかじめ用意された機能の実行を開始します。

カーネルが提供するシステムコールの集合体が、ユーザープログラムが見ることのできるインターフェイスです。xv6カーネルは、Unixカーネルが従来提供してきたサービスやシステムコールのサブセットを提供しています。図1.2にxv6のシステムコールの一覧を示します。

本章の残りの部分では、xv6のサービス（プロセス、メモリ、ファイル記述子、パイプ、ファイルシステム）の概要を説明し、コードスニペットや、伝統的なUnix系システムの主要なユーザーインターフェースである*シェル*がどのようにそれらを使用しているかについて説明します。シェルがシステムコールを使用することで、システムコールがいかに注意深く設計されているかがわかります。

シェルは、ユーザーからのコマンドを読み取って実行する普通のプログラムです。シェルがカーネルの一部ではなくユーザープログラムであるという事実は、システムコールインターフェースの威力を示しています：シェルには特別なものは何もありません。その結果、現代のUnixシステムには、それぞれが独自のユーザーインターフェースとスクリプト機能を持つ、さまざまなシェルが用意されています。xv6シェルは、Unix Bourneシェルのエッセンスをシンプルに実装したものです。その実装は(user/sh.c:1)にあります。

1.1 プロセスとメモリー

xv6のプロセスは、ユーザー空間のメモリ（命令、データ、スタック）と、カーネルに非公開のプロセスごとの状態で構成されています。Xv6は*プロセスをタイムシェアする*ことができます。つまり、利用可能なCPUを、実行を待っているプロセスのセットの間で透過的に切り替えます。プロセスが実行されていないとき、xv6はそのCPUレジスタを保存し、次にそのプロセスを実行するときに復元します。カーネルは、各プロセスにプロセス識別子（`pid`）を関連付けます。

プロセスは、`fork`システムコールを使用して新しいプロセスを作成することができます。`Fork`は新しいプロセスを作成します。

を*子プロセス*と呼び、*親プロセスと呼ばれる*呼び出し側のプロセスと全く同じメモリ内容を持つ。Forkは親プロセスと子プロセスの両方で戻ります。親プロセスでは、forkは子プロセスのpidを返し、子プロセスでは0を返します。例えば，C言語[5]で書かれた次のようなプログラムを考えてみましょう.

```
int pid =
fork(); if(pid >
0){...
  printf("parent: child=%d\en",
  pid); pid = wait(0);
  printf("child %d is done toen", pid);
} else if(pid == 0){。
```

システム	コール説明
	fork()プロセスの作成
	exit(xstatus)成功か失敗かを示すxstatusで現在のプロセスを終了さ
せる	wait(*xstatus)子プロセスが終了するの
	を待ち、子プロセスの終了ステータスをxstatusにコピーする
	kill(pid)プロセスpidを　終了させる
	getpid()現在のプロセスのpid　を返す
	sleep(n)nクロック分の
スリープ exec(filename	, *argv)ファイルを
	ロードして実行する
	sbrk(n)プロセスのメモリをnバイト
増やす open(filename,	flags)ファイルをオープンする。
write(fd, buf,	n)開いているファイルにnバイト　　書き込む
	close(fd)リリース オープンファイル fd
	dup(fd)fdを複製 する
	pipe(p)パイプを 作成して、pのfdを返す
	chdir(dirname)カレントディレクトリの　　変更
	mkdir(dirname)新規
ディレクトリ	の作成 mknod(name,
major,	
	minor)デバイスファ
イルの	作成
	fstat(fd)開いているファイルの情報　を返す
link(f1,	f2)ファイルf1に別の名前(f2)　をつける
	unlink(ファイル名)ファイルの削除

図 1.2: Xv6 システムコール

```
    printf("child:
    exiting\en"); exit(0);
  } else {
    printf("fork error\")と表示されます。
  }
```

exitシステムコールは、呼び出したプロセスの実行を停止し、メモリやオープンファイルなどのリソースを解放します。exitは整数のステータス引数を取り、通常は0が成功を、1が失敗を表します。waitシステムコールは、現在のプロセスの終了した子のpidを返し、その子の終了ステータスをwaitに渡されたアドレスにコピーします。呼び出し元の子が誰も終了していない場合、waitはその子が終了するのを待ちます。親プロセスが子プロセスの終了状態を気にしない場合は、waitに0を渡すことができます。

この例では、出力ライン

```
親：child=1234 子：
exiting
```

は、親と子のどちらが先にprintfコールに到達するかによって、どちらの順番でも出力される可能性があります。子が終了した後、親のwaitが戻り、親は次のように出力します。

```
親：子1234は完了
```

子は初期状態では親と同じメモリ内容を持っていますが、親と子は異なるメモリと異なるレジスタで実行されているため、一方の変数を変更しても他方には影響しません。例えば、親プロセスのwaitの戻り値をpidに格納しても、子プロセスの変数pidは変化しません。子プロセスのpidの値は0のままです。

execシステムコールは、呼び出したプロセスのメモリを、ファイルシステムに格納されたファイルからロードされた新しいメモリイメージに置き換えます。ファイルは特定のフォーマットを持っていなければならず、ファイルのどの部分が命令を持ち、どの部分がデータで、どの命令から開始するかなどを指定します。execが成功すると、呼び出したプログラムに戻るのではなく、ファイルから読み込まれた命令が、ELFヘッダで宣言されたエントリポイントで実行を開始します。Execは2つの引数を取ります。実行ファイルを含むファイル名と、文字列引数の配列です。例えば、以下のようになります。

```
char *argv[3];

argv[0] =

"echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo",
argv); printf("exec
error\en") となります。
```

このフラグメントは、呼び出したプログラムを、引数リスト echo hello で実行されるプログラム /bin/echo のインスタンスに置き換えます。ほとんどのプログラムでは、最初の引数は無視されます。

xv6シェルは、上記のコールを使用してユーザーに代わってプログラムを実行します。シェルの主な構造は単純で，main (user/sh.c:145)を参照してください．mainループはgetcmdを使ってユーザからの入力を1行読みます。その後，fork を呼び出し，シェルプロセスのコピーを作成します．親はwaitを呼び出し，子はコマンドを実行します．例えば、ユーザーがシェルに「echo hello」と入力した場合、runcmdは「echo hello」を引数にして呼び出されます。echo hello」の場合は、exec (user/sh.c:78)が呼び出されます。execが成功した場合、子プロセスはruncmdではなくechoの命令を実行します。echoがexitを呼び出すと、親はmainのwait(user/sh.c:145)から復帰します。

プロセスの作成とプログラムのロードを別々の呼び出しで行うことで、シェルでのI/Oリダイレクションに巧妙な利用法があることは後述します。重複したプロセスを作成してすぐに置き換えるという無駄を避けるために、オペレーティング・カーネルはコピーオンライトなどの仮想メモリ技術を使用して、このユースケースに合わせてkkの実装を最適

化しています。

Xv6 では、ほとんどのユーザースペースのメモリが暗黙的に割り当てられます。fork は親のメモリを子がコピーするのに必要なメモリを割り当て、exec は実行ファイルを格納するのに十分なメモリを割り当てます。実行時により多くのメモリを必要とする（おそらく malloc のために）プロセスは、sbrk(n)を呼び出してデータメモリを n バイト増やすことができ、sbrk は新しいメモリの位置を返します。

Xv6 は、ユーザーの概念や、あるユーザーを別のユーザーから保護するという概念を提供していません。Unix の用語では、すべての xv6 プロセスは root として実行されます。

1.2 I/Oとファイルディスクリプター

*ファイル記述子*は，プロセスが読み書きできるカーネル管理オブジェクトを表す小さな整数である。プロセスは、ファイル、ディレクトリ、デバイスをオープンしたり、パイプを作成したり、既存の記述子を複製したりすることで、ファイル記述子を取得します。ファイル記述子が参照するオブジェクトを、わかりやすくするために「ファイル」と呼ぶことがあります。ファイル記述子のインターフェースは、ファイル、パイプ、デバイスの違いを抽象化し、それらをすべてバイトのストリームのように見せています。

xv6カーネルは内部的に、ファイル記述子をプロセスごとのテーブルへのインデックスとして使用しており、すべてのプロセスがゼロから始まるファイル記述子のプライベート空間を持つようになっています。慣習的に、プロセスはファイルディスクリプター0（標準入力）から読み込み、ファイルディスクリプター1（標準出力）に出力を書き込み、ファイルディスクリプター2（標準エラー）にエラーメッセージを書き込みます。これから説明するように、シェルはこの規約を利用して、I/Oリダイレクションやパイプラインを実装しています。シェルは常に3つのファイル記述子を開いていることを保証します（user/sh.c:151）。これらはデフォルトではコンソール用のファイル記述子です。

`read` and `write`システムコールは，ファイル記述子で指定されたオープンファイルからのバイトの読み取りと，ファイル記述子で指定されたオープンファイルへのバイトの書き込みを行う。`read(fd, buf, n)`は，ファイル記述子`fd`から最大で`n`バイトを読み込み，`buf`にコピーし，読み込んだバイト数を返す。ファイルを参照する各ファイル記述子には、それに関連するオフセットがあります。`Read`は、現在のファイルオフセットからデータを読み込み、そのオフセットを読み込んだバイト数だけ進めます。後続の読み込みでは、最初の読み込みで返されたバイトの次のバイトが返されます。以降の読み取りでは、最初の読み取りで返されたバイトの次のバイトが返されます。読み取るバイトがなくなると、ファイルの終わりを示す0が返されます。

`write(fd, buf, n)`は、`buf`からファイルディスクリプタ`fd`に`n`バイトを書き込み、書き込んだバイト数を返す。書き込まれたバイト数は、エラーが発生した場合にのみ、`n`バイト未満のデータが書き込まれます。`read`と同様に、`write`は、現在のファイルオフセットにデータを書き込み、その後、書き込んだバイト数だけオフセットを進めます。

次のプログラム片（`cat`というプログラムの本質を形成している）は、データを標準入力から標準出力にコピーします。エラーが発生した場合は、標準エラーにメッセージを書き込みます。

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

```
n;

for(;;){
  n = read(0, buf, sizeof
  buf); if(n == 0)
    break;
  if(n < 0){
    fprintf(2, "read
    error\en"); exit();
  }
  if(write(1, buf, n) != n){
    fprintf(2, "write
    error\en"); exit()となります。
  }
}
```

このコードフラグメントで重要なことは、catはファイルから読んでいるのか、コンソールから読んでいるのか、パイプから読んでいるのかを知らないということです。同様に、catは自分がコンソールやファイルなどに印刷しているかどうかを知りません。ファイル記述子の使用と、ファイル記述子 0 が入力、ファイル記述子 1 が出力という規約により、cat の単純な実装が可能になります。

closeシステムコールは、ファイル記述子を解放し、将来のopen、pipe、またはdupシステムコール（下記参照）で再利用できるようにします。新たに割り当てられたファイル記述子は、常に現在のプロセスで最も低い番号の未使用記述子になります。

ファイル記述子とforkが相互に作用することで、I/Oリダイレクションが簡単に実装できるようになります。Forkは親のファイルディスクリプターテーブルをメモリと一緒にコピーするので、子は親と全く同じファイルを開いた状態で起動します。システムコールのexecは、呼び出したプロセスのメモリを置き換えますが、そのファイルテーブルは保持します。この動作により、シェルは分岐し、選択されたファイル記述子を再オープンしてから新しいプログラムを実行することで、I/Oリダイレクションを実装することができます。以下は、cat < input.txtというコマンドに対してシェルが実行するコードの簡略版です。

```
char *argv[2];

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

子がファイルディスクリプター0を閉じた後、openは新たに開いたinput.txtにそのファイルディスクリプターを使用することが保証されます。0 は利用可能な最小のファイル記述子となります。Cat は 、ファイル記述子0（標準入力）でinput.txtを参照して実行します。

xv6シェルのI/Oリダイレクトのコードは、まさにこの方法で動作しています（user/sh.c:82）。このコードのこの時点で、シェルはすでに子シェルをフォークしており、runcmdがexecを呼び出して新しいプログラムをロードしていることを思い出してください。これで、forkとexecが別の呼び出しであることが良いアイデアである理由が明らかになったはずです。なぜなら、この2つの呼び出しが別々であれば、シェルは子シェルをフォークし、子シェルでopen、close、dupを使用して標準入力と出力のファイルディスクリプターを変更した後、exec すること ができるからです。実行されるプログラム（この例では cat ） への変更は必要ありません。もし、fork と exec が単一のシステムコールに統合されたとしたら、シェルが標準入出力をリダイレクトするために何か他の（おそらくもっと複雑な）スキームが必要になるか、プログラム自体が

I/O をリダイレクトする方法を理解しなければならないでしょう。

`fork` は
ファイル記述子テーブルをコピーしますが、基礎となる各ファイルオフセットは親と子で共有されます。この例を考えてみましょう。

```
if(fork() == 0) {...
  write(1, "hello ",
  6); exit(0);
} else {
  wait(0);
  write(1, "world\en", 6);
}
```

このフラグメントの最後には、ファイル記述子1に添付されたファイルにhello worldというデータが格納されます。親の書き込み（waitのおかげで、子の書き込みが終わってから実行されます）は、子の書き込みが終わったところを拾います。この動作は、(echo hello; echo world) >output.txtのように、一連のシェルコマンドから連続した出力を生成するのに役立ちます。

dupシステムコールは、既存のファイル記述子を複製し、同じ基礎となるI/Oオブジェクトを参照する新しいファイル記述子を返します。両方のファイル記述子は、forkによって複製されたファイル記述子と同様に、オフセットを共有します。これは hello world をファイルに書き込む別の方法です。

```
fd = dup(1)です。
write(1, "hello ", 6)を行います。
write(fd, "world\en", 6);
```

2つのファイル記述子が、同じオリジナルのファイル記述子からforkとdupの連続した呼び出しによって派生したものであれば、オフセットを共有する。それ以外のファイル記述子は，たとえ同じファイルに対するオープンコールから派生したものであっても，オフセットを共有することはありません。2>&1 は，シェルに対して，ディスクリプタ 1 の複製であるファイル ディスクリプタ 2 を与えることを指示します。既存のファイルの名前と、存在しないファイルのエラーメッセージの両方が、tmp1というファイルに表示されます。xv6シェルはエラーファイルディスクリプターのI/Oリダイレクトをサポートしていませんが、これで実装方法がわかりました。

ファイル記述子は、それが何に接続されているかの詳細を隠すことができるため、強力な抽象化です。ファイル記述子1に書き込むプロセスは、ファイル、コンソールなどのデバイス、またはパイプに書き込んでいる可能性があります。

1.3 パイプ

*パイプ*とは、ファイルディスクリプターのペアとしてプロセスに公開される小さなカーネルバッファのことで、1つは読み込み用、もう1つは書き込み用です。パイプの一端にデータを書き込むと、そのデータはパイプのもう一端から読めるようになります。パイプは、プロセスが通信するための手段を提供します。

次のコード例では、標準入力をパイプの読み込み側に接続して、プログラムwcを実行しています。

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

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

```
= 0;

pipe(p);
if(fork() == 0)
{...
  close(0);
  dup(p[0]);
  close(p[0])となります。
  close(p[1]);
  exec("/bin/wc",
  argv);
} else {
```

```
  close(p[0])となります。
  write(p[1], "hello world\en",
  12); close(p[1]);
}
```

プログラムがpipeを呼び出すと，pipeは新しいパイプを作成し，読み書きされたファイル記述子を配列pに記録する。forkの後，親も子もパイプを参照するファイル記述子を持つ。子は読み込み終了をファイルディスクリプタ0にダンプし，pのファイルディスクリプタをクローズしてwcを実行する。wcが標準入力から読み込むとき、パイプから読み込みます。親は、パイプの読み取り側を閉じ、パイプに書き込み、書き込み側を閉じます。

データが利用できない場合、パイプ上のreadは、データが書き込まれるか、書き込み側を参照するすべてのファイルディスクリプターが閉じられるかのいずれかを待ちます。後者の場合、データファイルの終端に達した場合と同様に、readは0を返します。readが新しいデータの到着が不可能になるまでブロックするという事実は、子プロセスが上記のwcを実行する前にパイプの書き込み側を閉じることが重要である理由の1つです：もしwcのファイル記述子の1つがパイプの書き込み側を参照していたら、wcはend-of-fileを見ることができません。

xv6シェルでは、上のコード（user/sh.c:100）と同様の方法で、grep fork sh.c | wc -lといったパイプラインを実装しています。子プロセスは、パイプラインの左端と右端をつなぐパイプを作成します。そして、パイプラインの左端のためにforkとruncmdを、右端のためにforkとruncmdを呼び出し、両方が終了するのを待ちます。パイプラインの右端は、それ自体がパイプを含むコマンド（例：a | b | c）である可能性があり、そのコマンド自体が2つの新しい子プロセス（b用とc用）をフォークします。このように、シェルはプロセスのツリーを作成することができます。このツリーの葉はコマンドで、内部のノードは左と右の子プロセスが完了するまで待つプロセスです。原理的には、内部ノードにパイプラインの左端を実行させることもできますが、それを正しく行うと実装が複雑になります。

パイプラインは、一時的なファイルと変わらないように見えるかもしれません。

```
echo hello world | wc
```

としてパイプなしで実装できました。

```
echo hello world >/tmp/xyz; wc </tmp/xyz
```

このような場合、パイプには一時ファイルに比べて少なくとも4つの利点があります。第1に、パイプは自動的に自分自身をクリーンアップします。ファイルのリダイレクションでは、シェルは終了時に/tmp/xyzを削除するように注意しなければなりません。第2に、パイプは任意の長さのデータストリームを渡すことができます。一方、ファイルリダイレクションでは、すべてのデータを保存するためにディスク上に十分な空き領域が必要です。第3に、パイプはパイプラインのステージを並行して実行することができますが

、ファイルのアプローチでは第2のプログラムが始まる前に第1のプログラムが終了する必要があります。第4に、プロセス間通信を実装する場合、パイプのブロッキングリード/ライトは、ファイルのノンブロッキングセマンティクスよりも効率的です。

1.4 ファイルシステム

xv6ファイルシステムでは、解釈されないバイト配列であるデータファイルと、データファイルや他のディレクトリへの名前付き参照を含むディレクトリが用意されています。ディレクトリは、次の場所から始まるツリー構造になっています。

は、*ルート*と呼ばれる特別なディレクトリを指します。a/b/cのような*パス*は、ルートディレクトリ/の中のaというディレクトリの中のbというディレクトリの中のcというファイルまたはディレクトリを指します。/で始まらないパスは、呼び出したプロセスの*カレントディレクトリ*からの相対的な評価となります。これらのコードフラグメントはどちらも同じファイルを開きます（関係するすべてのディレクトリが存在すると仮定します）。

```
chdir("/a")となります。
chdir("b")となります。
open("c", O_RDONLY);

open("/a/b/c", O_RDONLY)となります。
```

最初のフラグメントはプロセスのカレントディレクトリを/a/bに変更し、2番目のフラグメントはプロセスのカレントディレクトリを参照も変更もしません。

新しいファイルやディレクトリを作成するシステムコールは複数あります。mkdirは新しいディレクトリを作成し、open（O_CREATEフラグ付き）は新しいデータファイルを作成し、mknodは新しいデバイスファイルを作成します。この例では、この3つをすべて説明します。

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

Mknodは、ファイルシステムにファイルを作成しますが、そのファイルには中身がありません。その代わり、ファイルのメタデータがデバイスファイルとしてマークし、カーネルデバイスを一意に識別するメジャーデバイス番号とマイナーデバイス番号（mknodの2つの引数）を記録します。後でプロセスがそのファイルを開くと、カーネルは読み書きのシステムコールをファイルシステムに渡すのではなく、カーネルデバイスの実装に振り分けます。

Fstatは、ファイル記述子が参照しているオブジェクトの情報を取得します。stat.h（kernel/stat.h）で次のように定義されている構造体statに入力します。

```
#define T_DIR     1   // ディレクトリ
#define T_FILE    2   // ファイル
#define T_DEVICE  3   // デバイス

struct stat {
  int          dev; // ファイルシステムのディスクデバイス uint   ino;
  // Inode番号
  short type; // ファイルの種類
  short nlink; // ファイルへのリンクの数
  uint64 size; // ファイルのサイズ(バイト)
```

```
};
```

ファイルの名前は、ファイル自体とは別のもので、*inode*と呼ばれる同じ基本ファイルが、*リンク*と呼ばれる複数の名前を持つことができる。リンクシステムコールは、既存のファイルと同じinodeを参照する別のファイルシステム名を作成します。このフラグメントは、aとbの両方の名前を持つ新しいファイルを作成します。

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

各inodeは一意の*inode番号*で識別されます。上記の一連のコードの後、fstatの結果を見ることで、aとb が同じ基礎的なコンテンツを参照していることを判断することができます。どちらも同じinode番号（ino）を返し、nlinkカウントは2に設定されます。

unlinkシステムコールは、ファイルシステムから名前を削除する。ファイルのinodeとその内容を保持するディスク領域は、ファイルのリンクカウントが0になり、そのファイルを参照するファイル記述子がない場合にのみ解放される。したがって

```
unlink("a");
```

を最後のコードシーケンスに変更すると、inodeとファイルの内容がb.としてアクセス可能な状態になります。

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

は、一時的なinodeを作成するための慣用的な方法で、プロセスがfd を閉じるときにクリーンアップされます。
や出口があります。

ファイルシステムを操作するシェルコマンドは、mkdir, ln, rmなどのユーザーレベルプログラムとして実装されています。この設計では、新しいユーザーレベルのプログラムを追加するだけで、誰でも新しいユーザーコマンドでシェルを拡張することができます。今にして思えば、この計画は当然のことのように思えますが、Unixの時代に設計された他のシステムでは、このようなコマンドをシェルに組み込んでいる（シェルをカーネルに組み込んでいる）場合が多いのです。

例外として，シェルに組み込まれている cd があります（user/sh.c:160）．通常のコマンドとして cd を実行した場合，シェルは子プロセスをフォークし，子プロセスは cd を実行し，cd は*子プロセス*の作業ディレクトリを変更します．親（つまりシェル）の作業ディレクトリは変更されません。

1.5 リアルワールド

標準」のファイル記述子、パイプ、そしてそれらを操作するための便利なシェル構文を組み合わせたUnixは、汎用の再利用可能なプログラムを書く上で大きな進歩となりました。このアイデアは、Unixのパワーと人気の多くの原因となった「ソフトウェア・ツール」の文化全体に火をつけ、シェルは最初のいわゆる「スクリプト言語」となりました。Unixのシステムコールインターフェースは、BSD、Linux、Mac OS Xなどのシステムに受け継がれています。

Unixのシステムコールインタフェースは、POSIX（Portable Operating System Interface）規格によって標準化されています。Xv6はPOSIXに準拠して*いません*。システムコール（lseek の

ような基本的なものを含む）が欠落していたり、システムコールを部分的にしか実装していなかったり、その他にも様々な違いがあります。xv6　　の主な目標は、UNIXライクなシステムコールのインターフェースを提供しつつ、シンプルでわかりやすいものにすることです。基本的なUnixプログラムを実行するために、いくつかのシステムコールとシンプルなCライブラリでxv6を拡張した人もいます。しかし、最近のカーネルは、xv6よりも多くのシステムコールや、多くの種類のカーネルサービスを提供しています。例えば、ネットワーク、勝利のためのシステム、ユーザーレベルのスレッド、多くのデバイスのドライバなどをサポートしています。最近のカーネルは継続的かつ急速に進化しており、POSIXを超える多くの機能を提供しています。

最近のUnix系OSでは、上述のコンソールデバイスファイルのように、特別なファイルとしてデバイスを公開するという初期のUnixのモデルをほとんどの場合踏襲していません。そのため

これは、「リソースはファイルである」という概念を現代の設備に適用したもので、ネットワークやグラフィックス、その他のリソースをファイルやファイルツリーとして表現するものである。

ファイルシステムとファイルディスクリプターは強力な抽象化手段である。それでも、オペレーティングシステムのインターフェースには他のモデルがあります。Unixの前身であるMulticsは、ファイルストレージをメモリのように見せて抽象化し、まったく異なるテイストのインターフェースを実現しました。Multicsの設計の複雑さは、Unixの設計者に直接的な影響を与え、彼らはよりシンプルなものを作ろうとしました。

本書では、xv6がどのようにしてUnixライクなインターフェイスを実装しているかを検証していますが、そのアイデアやコンセプトはUnix以外にも応用できます。どのようなオペレーティング・システムであっても、プロセスを基礎となるハードウェアに多重化し、プロセス同士を分離し、プロセス間のコミュニケーションを制御するメカニズムを提供しなければなりません。xv6を学んだ後は、他のより複雑なオペレーティングシステムを見て、それらのシステムにもxv6の基礎となるコンセプトを見ることができるはずです。

1.6 エクササイズ

1. UNIXシステムコールを使用して、2つのプロセス間で1バイトを「ピンポン」するプログラムを作成します。このプログラムの性能を1秒あたりの変化量で測定しなさい。

第2章

オペレーティングシステムの構成

オペレーティングシステムの主な要件は、複数のアクティビティを同時にサポートすることです。例えば、第1章で説明したシステムコールインターフェースを使って、プロセスは`fork`で新しいプロセスを開始することができます。オペレーティングシステムは、これらのプロセス間でコンピュータのリソースを*タイムシェアする*必要があります。例えば、プロセスの数がハードウェアCPUの数よりも多くても、オペレーティングシステムはすべてのプロセスが確実に進行するようにしなければなりません。オペレーティング・システムは、プロセス間の*隔離*も行わなければなりません。つまり、あるプロセスにバグがあって失敗しても、失敗したプロセスに依存していないプロセスには影響を与えてはならないのです。しかし、完全な隔離では強すぎます。なぜなら、プロセスが意図的に相互作用することが可能でなければならないからです（パイプラインがその例です）。このように、オペレーティングシステムは、多重化、分離、相互作用という3つの要件を満たさなければなりません。

本章では、この3つの要件を満たすために、オペレーティングシステムがどのように構成されているかを概観します。いろいろな方法があることがわかりますが、このテキストでは、多くのUnixオペレーティング・システムで採用されている*モノリシック・カーネル*を中心とした主流のデザインに焦点を当てます。この章では、xv6の分離の単位であるxv6プロセスの概要と、xv6の実行開始時の最初のプロセスの生成についても説明します。

Xv6はマルチコアのRISC-Vマイクロプロセッサ上で動作し、その低レベルの機能（例えばプロセスの実装）の多くはRISC-Vに特有のものです。RISC-Vは64ビットのCPUで、xv6は「LP64」というC言語で書かれています。これは、C言語のロング（L）とポインタ（P）が64ビットであることを意味しますが、intは32ビットです。本書は、読者が何らかのアーキテクチャでマシンレベルのプログラミングを少しでも行ったことがあることを前提としており、RISC-V特有のアイデアを随時紹介していきます。RISC-Vに関する有用な参考文献として、"The RISC-V Reader:An Open Architecture Atlas"

[9]です。ユーザーレベルISA[2]と特権アーキテクチャ[1]が正式な仕様となっています。

本文中では、計算を実行するハードウェア要素を、Central Processing Unitの頭文字をとって、一般的に*CPUと表記しています*。他の文書（RISC-V仕様書など）では、CPUの代わりに、processor、core、hartという言葉が使われています。Xv6は、*マルチコアの*RISC-Vハードウェアを想定しています。つまり、複数のCPUがメモリを共有しながら、独立したプログラムを並列に実行することを想定しています。この文章では、マルチプロセッサという言葉をマルチコアの同義語として使うことがありますが、マルチプロセッサは、複数の異なるプロセッサ・チップを搭載したコンピュータ・ボードを指すこともあります。

完全なコンピュータのCPUはサポートハードウェアに囲まれており、その多くはI/Oインターフェースの形をしています。Xv6はqemuの「-machine virt」オプションでシミュレートされたサポートハードウェア用に書かれています。これには、RAM、ブートコードの入ったROM、ユーザーの鍵盤へのシリアル接続などが含まれます。

ボードやスクリーン、そして記録用のディスク。

2.1 物理的リソースの抽象化

オペレーティング・システムに出会ったときの最初の疑問は、なぜオペレーティング・システムが必要なのかということでしょう。つまり、図1.2のシステムコールをライブラリとして実装し、アプリケーションがそれにリンクするようにすればいいのです。この計画では、各アプリケーションが自分のニーズに合わせた独自のライブラリを持つこともできます。アプリケーションは、ハードウェア資源と直接対話し、その資源をアプリケーションにとって最適な方法で使用することができます（例えば、高い性能や予測可能な性能を実現することができます）。組込み機器やリアルタイムシステム用のOSの中には、このように構成されているものがあります。

このライブラリアプローチの欠点は、複数のアプリケーションが動作している場合、各アプリケーションがうまく動作しなければならないことです。例えば、他のアプリケーションが実行できるように、各アプリケーションは定期的にCPUを放棄しなければなりません。このような*協力的な*タイムシェアリング方式は、すべてのアプリケーションがお互いに信頼し合い、バグがなければ問題ないかもしれません。しかし、アプリケーション同士がお互いに信頼し合っていなかったり、バグがあったりすることの方が多いので、協調的なスキームが提供するよりも強力な分離が必要になることがよくあります。

強力な分離を実現するためには、アプリケーションが機密性の高いハードウェア・リソースに直接アクセスすることを禁止し、代わりにリソースをサービスとして抽象化することが有効です。例えば、アプリケーションは、生のディスク・セクタを読み書きするのではなく、オープン、リード、ライト、クローズのシステム・コールを通じてのみファイル・システムとやりとりします。これにより、アプリケーションにはパス名の利便性が提供され、オペレーティング・システムは（インターフェースの実装者として）ディスクの管理を行うことができます。

同様に、Unixではプロセス間でハードウェアCPUを透過的に切り替え、必要に応じてレジスタの状態を保存・復元しているため、アプリケーションが時間共有を意識する必要はありません。この透過性により、一部のアプリケーションが無限ループに陥っていても、OSはCPUを共有することができます。

他の例として、Unixのプロセスは、物理的なメモリと直接やりとりするのではなく、`exec`を使ってメモリイメージを構築します。これにより、オペレーティングシステムは、メモリ内のどこにプロセスを配置するかを決定することができます。メモリが不足して

いる場合には、オペレーティングシステムは、プロセスのデータの一部をディスクに保存することもできます。また、Execは、実行可能なプログラムイメージを保存するファイルシステムの利便性をユーザーに提供します。

Unixのプロセス間の相互作用の多くは、ファイル記述子を介して行われます。ファイル記述子は、多くの詳細（パイプやファイル内のデータがどこに格納されているかなど）を抽象化するだけでなく、相互作用を単純化する方法で定義されています。例えば、パイプラインの1つのアプリケーションが失敗した場合、カーネルはパイプラインの次のプロセスのためにend-of-fileシグナルを生成します。

ご覧のように、図1.2のシステム・コール・インターフェースは、プログラマの利便性と強力な分離の可能性の両方を提供するように注意深く設計されています。Unixのインターフェイスはリソースを抽象化する唯一の方法ではありませんが、非常に優れた方法であることが証明されています。

2.2 ユーザーモード、スーパーバイザーモード、システムコール

強力なアイソレーションには、アプリケーションとオペレーティング・システムの間に硬い境界が必要です。アプリケーションがミスをしても、オペレーティング・システムが故障したり、他のアプリケーションが故障したりすることは避けたいものです。その代わり、オペレーティング・システムは失敗したアプリケーションをクリーンアップし、他のアプリケーションを継続して実行できるようにしなければなりません。強力な分離を実現するためには、オペレーティングシステムは、アプリケーションがオペレーティングシステムのデータ構造や命令を変更できない(読めない)ように、また、アプリケーションが他のプロセスのメモリにアクセスできないように手配しなければなりません。

CPUは、強力なアイソレーションをハードウェアでサポートしています。例えば、RISC-Vでは、CPUが命令を実行できるモードとして、*マシンモード*、*スーパバイザモード*、*ユーザーモード*の3つがあります。マシンモードで実行される命令は完全な権限を持ち、CPUはマシンモードで起動します。マシンモードは、主にコンピュータの設定を目的としています。Xv6はマシンモードで数行実行した後、スーパバイザモードに移行します。

スーパバイザモードでは、CPUは特権*的な命令*を実行することができます。例えば、割り込みの有効化や無効化、ページテーブルのアドレスを保持するレジスタの読み書きなどが可能になります。ユーザーモードのアプリケーションが特権命令を実行しようとすると、CPUはその命令を実行せずにスーパバイザモードに切り替え、スーパバイザモードのコードが「やってはいけないことをやった」という理由でアプリケーションを終了させることができます。第1章の図1.1にこの構成を示します。アプリケーションは、ユーザーモードの命令（数字の足し算など）しか実行できず、*ユーザー空間*で動作しているといいます。一方、スーパバイザモードのソフトウェアは、特権的な命令も実行できるため、*カーネル空間*で動作しているといいます。カーネルスペース（またはスーパーバイザーモード）で動作しているソフトウェアを*カーネル*と呼ぶ。

カーネルの機能（例：xv6の`read`システムコール）を呼び出したいアプリケーションは、カーネルに移行する必要があります。CPUはユーザモードからスーパバイザモードに切り替える特別な命令を用意しており、カーネルが指定したエントリポイントからカーネルに入ります（RISC-Vではこのためにecall命令を用意しています）。(RISC-Vでは、この目的のために`ecall`命令が用意されています。)
CPUがスーパバイザモードに切り替わると、カーネルはシステムコールの引数を検証し、アプリケーションが要求された操作を実行してよいかどうかを判断し、拒否または実行することができます。アプリケーションがカーネルのエントリーポイントを決めることができれば、悪意のあるアプリケーションは、例えば、引数の検証がスキップされたポイントでカーネルに入ることができます。

2.3　カーネルの構成

設計上の重要な問題は、オペレーティングシステムのどの部分をスーパバイザモードで実行するかということです。1つの可能性として、オペレーティングシステム全体がカーネルに存在し、すべてのシステムコールの実装がスーパバイザモードで実行されるというものがあります。このような構成を*モノリシック・カーネル*と呼びます。

この組織では、OS全体が完全なハードウェア特権で動作します。この組織は、OS設計者がOSのどの部分にハードウェア特権を必要としないかを決定する必要がないので便利です。さらに、オペレーティング・システムのさまざまな部分が協力しやすくなります。例えば、オペレーティング・システムは、ファイル・システムと仮想メモリ・システムの両方で共有できるバッファ・キャッシュを持っています。

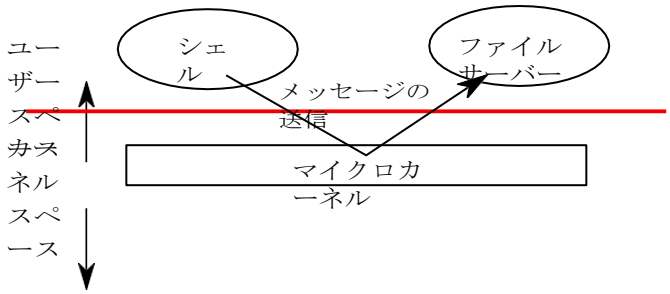


図2.1:ファイルシステム・サーバーを備えたマイクロカーネル

モノリシックな組織の欠点は、オペレーティング・システムのさまざまな部分の間のインターフェイスが複雑であることが多く（このテキストの残りの部分で説明します）、そのため、オペレーティング・システムの開発者がミスを犯しやすいことです。モノリシック・カーネルでは、スーパバイザ・モードでのエラーがカーネルの故障を引き起こすことが多いため、ミスは致命的です。カーネルが故障すると、コンピュータが動作しなくなり、すべてのアプリケーションも故障します。コンピュータは再起動しなければ再び動き出すことができません。

カーネルでのミスのリスクを減らすために、OSの設計者は、スーパバイザモードで実行されるオペレーティングシステムコードの量を最小限にし、オペレーティングシステムの大部分をユーザモードで実行することができます。このようなカーネルの構成を*マイクロカーネル*と呼びます。

このマイクロカーネルの設計を図2.1に示します。この図では、ファイルシステムがユーザレベルのプロセスとして動作しています。プロセスとして動作するOSサービスをサーバと呼ぶ。アプリケーションがファイルサーバとやりとりするために、カーネルはユーザモードのプロセスから別のプロセスにメッセージを送るプロセス間通信メカニズムを提供します。例えば、シェルのようなアプリケーションがファイルを読み書きしたい場合、ファイルサーバーにメッセージを送信し、応答を待ちます。

マイクロカーネルの場合、カーネルインターフェースは、アプリケーションの起動、メッセージの送信、デバイスハードウェアへのアクセスなどを行ういくつかの低レベル関数で構成されています。このような構成により、オペレーティングシステムの大部分がユーザーレベルのサーバーに存在するため、カーネルを比較的シンプルにすることができます。

Xv6 は、ほとんどの Unix オペレーティング・システムと同様に、モノリシック・カーネルとして実装されています。そのため、xv6のカーネル・インターフェースはオペレーティングシステム・インターフェースに対応しており、カーネルは完全なオペレーティングシステムを実装しています。xv6は多くのサービスを提供していないので、そのカーネルはいくつかのマイクロカーネルよりも小さくなっていますが、概念的にはxv6はモノリシック（一枚岩）です。

2.4　コード：xv6編成

xv6のカーネルソースは、`kernel/`サブディレクトリにあります。図2.2にファイルの一覧を示します。各モジュールのインターフェースは`defs.h`（kernel/defs.h）で定義されています。

2.5　プロセス概要

xv6(他のUnixオペレーティングシステムと同様)における隔離の単位は*プロセス*です。プロセス・アブストラクションは、あるプロセスが他のプロセスのメモリ、CPU、ファイル・システムを破壊したりスパイしたりすることを防ぎます。

ファイル	説明
bio.c	ファイルシステムのディスクブロックキャッシュ。
console.c	ユーザーのキーボードとスクリーンに接続します。
entry.S	一番最初の起動手順。
exec.c	exec()システムコールです。
file.c	ファイル記述子のサポート。
fs.c	ファイルシステム。
kalloc.c	物理ページアロケータ。
kernelvec.S	カーネルからのトラップやタイマー割り込みの処理。
log.c	ファイルシステムのロギングとクラッシュリカバリー
main.c	ブート時に他のモジュールの初期化を制御する。
pipe.c	パイプです。
plic.c	RISC-V割り込みコントローラ。
printf.c	コンソールへのフォーマットされた出力。
proc.c	プロセスとスケジューリング
sleeplock.c	CPUを獲得できるロック。
spinlock.c	CPUを得られないロック。
start.c	初期のマシンモードブートコード
string.c	Cの文字列とバイト配列のライブラリ
swtch.S	スレッド切り替え
syscall.c	システムコールをハンドリング機能にディスパッチする。
sysfile.c	ファイル関連のシステムコール。
sysproc.c	プロセス関連のシステムコール。
トランポリン.S	ユーザーとカーネルを切り替えるためのアセンブリコード。
トラップ.C	トラップや割り込みを処理して復帰するCコード
uart.c	シリアルポートのコンソールデバイスドライバです。
virtio_disk.c	ディスク装置のドライバ。
vm.c	ページテーブルとアドレス空間を管理する。

図2.2:Xv6カーネルのソースファイル

記述子などがあります。また、プロセスがカーネル自体を破壊することを防ぎ、プロセスがカーネルのアイソレーション・メカニズムを覆すことができないようにしています。バグのあるアプリケーションや悪意のあるアプリケーションがカーネルやハードウェアを騙して何か悪いこと(隔離を回避するなど)をさせる可能性があるので、カーネルはプロセスの抽象化を慎重に実装しなければなりません。カーネルがプロセスを実装するために使用するメカニズムには、ユーザー/スーパーバイザー・モード・フラグ、アドレス空間、スレッドのタイム・スライシングなどがあります。

分離を強化するために、プロセス抽象化は、プログラムが自分専用のマシンを持っているかのような錯覚を与えます。プロセスはプログラムに、他のプロセスが読み書きできないプライベートなメモリ・システムや*アドレス空間*のようなものを提供します。また、プロセスは、プログラムの命令を実行する独自のCPUのようなものをプログラムに提供します。

Xv6では、各プロセスに独自のアドドレス空間を与えるために、ハードウェアで実装されたページテーブルを使用しています。RISC-Vのページテーブルは、*仮想アドレス*（あるプロセスがそのプロセスのために使用するアドレス）を変換（またはマッピング）します。

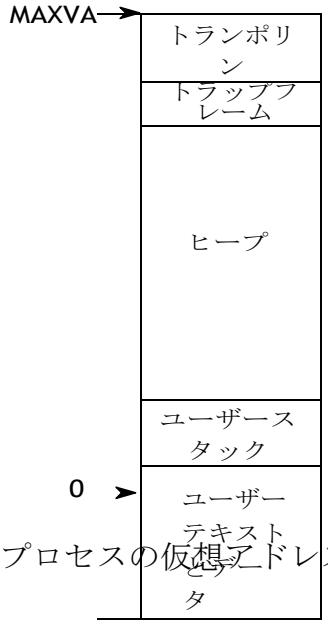


図2.3：ユーザープロセスの仮想アドレス空間のレイアウト

RISC-
V命令が操作する）を*物理アドレス*（CPUチップがメインメモリに送るアドレス）に変換します。

Xv6では、各プロセスごとに個別のページテーブルが用意され、そのプロセスのアドレス空間が定義されます。図2.3に示すように、アドレス空間には、仮想アドレス0から始まるプロセスの*ユーザーメモリ*が含まれます。最初に命令、次にグローバル変数、次にスタック、そして最後にプロセスが必要に応じて拡張できる「ヒープ」領域（malloc用）が続きます。Xv6は39ビットの仮想アドレスを持つRISC-
V上で動作しますが、38ビットしか使用しません。したがって、最大のアドレスは $2^{38}-1$ = 0x3fffffff であり、これは `MAXVA` である(kernel/riscv.h:349)。アドレス空間の最上部には、xv6が*トランポリン*用のページと、第4章で説明するように、カーネルに切り替えるためのプロセスの*トラップフレーム*をマッピングするページを確保しています。

xv6カーネルは、プロセスごとに多くの状態を保持し、それらを`proc`構造体（kernel/proc.h:86）にまとめます。プロセスのカーネル状態で最も重要な部分は、ページテーブル、カーネルスタック、ランステートです。ここでは、`proc`構造体の要素を参照するために`p->xxx`という表記を使用します。
前述のトラップフレームは`p->tf`です。

各プロセスには、そのプロセスの命令を実行する実行スレッド（略して*スレッド*）があります。スレッドは一時停止し、後で再開することができます。プロセス間を透過的に切り替えるために、カーネルは現在実行中のスレッドを一時停止し、他のプロセスのスレッドを再開します。スレッドの状態（ローカル変数、関数呼び出しのリターンアドレス）の多くは、スレッドのスタックに格納されています。各プロセスには、ユーザースタックとカーネルスタックの2つのスタックがあります（`p->`

>kstack）。プロセスがユーザ命令を実行しているときは、そのユーザスタックのみが使用されており、カーネルスタックは空です。プロセスが（システムコールや割り込みで）カーネルに入ると、カーネルコードはプロセスのカーネルスタック上で実行されます。プロセスがカーネル内にある間は、ユーザースタックには保存されたデータが残っていますが、実際には使用されません。プロセスのスレッドは、ユーザースタックとカーネルスタックを交互に使用します。カーネルスタックは独立しており（ユーザーコードから保護されています）、プロセスがユーザースタックを破壊した場合でも、カーネルは実行できます。

プロセスがシステムコールを行うには、RISC-Vのecall命令を実行します。この命令は

は、ハードウェアの特権レベルを上げ、プログラムカウンタをカーネルが定義したエントリーポイントに変更します。エントリポイントのコードは、カーネルスタックに切り替わり、システムコールを実装するカーネル命令を実行します。システムコールが完了すると、カーネルはユーザースタックに切り替わり、sret命令を呼び出してユーザー空間に戻ります。sret命令はハードウェアの特権レベルを下げ、システムコール命令の直後にユーザー命令の実行を再開します。プロセスのスレッドは、I/Oを待つためにカーネル内で「ブロック」することができ、I/Oが終了すると中断したところから再開することができます。

p->stateは、プロセスが割り当てられているか、実行準備ができているか、実行中か、I/O待ちか、終了しているかを示します。

p->pagetableには、RISC-Vハードウェアが示す形式のプロセスのページテーブルが格納されます。xv6では、ユーザ空間でプロセスを実行する際に、ページングハードウェアがプロセスのp->pagetableを使用するようになっています。プロセスのページテーブルは、そのプロセスのメモリを格納するために割り当てられた物理ページのアドレスの記録としても機能します。

2.6 コード：xv6の起動と最初のプロセス

xv6をより具体的にするために、カーネルがどのようにして最初のプロセスを開始・実行するのかを概説します。以降の章では、この概要に出てくるメカニズムをより詳しく説明します。

RISC-Vコンピュータの電源が入ると、自己を初期化し、読み取り専用メモリに格納されているブートローダを実行します。ブートローダはxv6カーネルをメモリにロードします。そして、マシンモードでは、CPUが_entry(kernel/entry.S:12)からxv6を実行します。Xv6はRISC-Vのページングハードウェアを無効にして起動し、仮想アドレスが物理アドレスに直接マッピングされます。

ローダーはxv6カーネルを物理アドレス0x80000000のメモリにロードします。カーネルを0x0ではなく0x80000000に配置するのは、アドレス範囲0x0:0x80000000にI/Oデバイスが含まれているからです。

_entryの命令は、xv6がCコードを実行できるようにスタックをセットアップします。Xv6はファイルstart.c(kernel/start.c:11)で初期スタックstack0のスペースを宣言しています。RISC-Vのスタックは下に向かって伸びていくので、_entryのコードはスタックポインタレジスタspにスタックの先頭であるstack0+4096のアドレスをロードします。これでスタックがで

きたので、_entryはstartのCコードを呼び出します（kernel/start.c:21）。

関数startは、マシンモードでのみ許可される設定を行い、その後、スーパバイザモードに切り替わります。スーパバイザモードに入るために、RISC-Vはmretという命令を用意しています。startはそのような呼び出しから戻るのではなく、あたかもそのような呼び出しがあったかのように設定します。前の特権モードをレジスタmstatusでsupervisorに設定し、リターンアドレスをレジスタmepcにmainのアドレスを書き込むことでmainに設定し、ページテーブルレジスタsatpに0を書き込むことでsupervisorモードでの仮想アドレス変換を無効にし、すべての割り込みと例外をsupervisorモードに委ねます。

スーパバイザモードに移行する前に、startはもう1つのタスクを実行します。それは、タイマ割り込みを生成するようにクロックチップをプログラムすることです。これらの作業を終えた後、startはmretを呼び出してスーパーバイザーモードに「戻り」ます。これにより、プログラムカウンタが main に変わります (kernel/main.c:11)。

`main` (kernel/main.c:11) はいくつかのデバイスやサブシステムを初期化した後、`userinit` (kernel/proc.c:197)
を呼び出して最初のプロセスを生成します。最初のプロセスは小さなプログラムである`initcode.S`
(user/initcode.S:1)を実行し、`exec`システムコールを呼び出してカーネルに再参入します。第1章で見たように
第1項の`exec` は
、現在のプロセスのメモリやレジスタを新しいプログラム（ここでは`/init`）に置き換えます。カーネルは`exec`を完了すると、ユーザースペースに戻り、`/init`を実行します。`Init` (user/init.c:11)
は、必要に応じて新しいコンソール・デバイス・ファイルを作成し、それをファイル記述子0、1、2としてオープンします。その後、ループしてコンソールシェルを起動し、シェルが終了するまで孤児となったゾンビを処理し、それを繰り返します。システムが立ち上がりました。

2.7 リアルワールド

現実の世界では、モノリシックなカーネルとマイクロカーネルの両方が存在します。多くのUnixカーネルはモノリシックです。例えば、Linuxはモノリシックカーネルですが、一部のOS機能はユーザーレベルのサーバーとして動作しています（例：ウィンドウシステム）。L4、Minix、QNXなどのカーネルは、サーバーを備えたマイクロカーネルとして構成されており、組み込み環境で広く展開されています。

ほとんどのOSはプロセスの概念を採用しており、ほとんどのプロセスはxv6のものと似ています。しかし、最近のオペレーティングシステムでは、1つのプロセスが複数のCPUを利用できるように、1つのプロセス内で複数のスレッドをサポートしています。プロセス内で複数のスレッドをサポートするには、xv6にはないかなりの機械が必要で、スレッドがプロセスのどの部分を共有するかを制御するために、潜在的なインターフェースの変更（Linuxの`clone`、`fork`の変形など）も含まれます。

2.8 エクササイズ

1. gdbを使って、最初のカーネルからユーザーへの移行を観察することができます。`make qemu-gdb` を実行してください。
   同じディレクトリにある別のウィンドウでgdbを実行します。gdbコマンド`break *0x3ffff07e`を入力してください。これは、ユーザースペースにジャンプするカーネルの`sret`命令にブレークポイントを設定します。タイプ
   gdbはブレークポイントで停止し、`sret`を実行しようとしています。`stepi`を入力してください。gdbは現在、`initcode.S`の開始点であるユーザースペースのアドレス0x4で実行していることを示すはずです。

第3章

ページテーブル

ページテーブルとは、OSが各プロセスに専用のアドレス空間とメモリを提供するための仕組みである。ページテーブルは、メモリアドレスの意味と、物理メモリのどの部分にアクセスできるかを決定します。ページテーブルによって、xv6は異なるプロセスのアドドレス空間を分離し、単一の物理メモリに多重化することができます。また、ページテーブルは、xv6がいくつかのトリックを実行するためのインダイレクトレベルを提供します。同じメモリ（トランポリンページ）を複数のアドレス空間にマッピングしたり、マッピングされていないページでカーネルスタックやユーザスタックをガードしたりします。本章の残りの部分では、RISC-Vハードウェアが提供するページテーブルと、xv6がそれらをどのように使用するかについて説明します。

3.1 ページングハードウェア

RISC-Vの命令（ユーザーとカーネルの両方）は、仮想アドレスを操作することを覚えておいてください。一方、マシンのRAM（物理メモリ）は、物理アドレスでインデックスされています。RISC-Vのページテーブルハードウェアは、それぞれの仮想アドレスを物理アドレスにマッピングすることで、この2種類のアドレスを結びつけます。

xv6は、39ビットの仮想アドレスを持つSv39 RISC-V上で動作します（図3.1参照）。64ビットの仮想アドレスのうち、上位25ビットは未使用です。このSv39の構成では、RISC-

Vのページテーブルは、論理的には2^{27}(134,217,728)個の*ページテーブルエントリ(PTE)*の配列となります。各PTEには、44ビットの物理ページ番号（PPN）といくつかのフラグが含まれています。ページングハードウェアは、39ビットのうち上位27ビットをページテーブルにインデックスしてPTEを探し、上位44ビットをPTEのPPNから、下位12ビットを元の仮想アドレスからコピーした56ビットの物理アドレスを作成して、仮想アドレスを変換します。このように、ページテーブルは、4096（2）^{12}バイトのアラインメントされたチャンクの粒度で、仮想アドレスから物理アドレスへの変換をオペレーティングシステムに制御させます。このような塊を「ページ」と呼びます。

Sv39 RISC-Vでは、仮想アドレスの上位25ビットは変換に使用されません。将来的には、RISC-Vはこれらのビットを使用して、より多くのレベルの変換を定義することができます。同様に、物理アドレスにも拡張の余地があります。Sv39では56ビットですが、64ビットに拡張することも可能です。

図3.2に示すように、実際の変換は3つのステップで行われます。ページテーブルは、3階層のツリーとして物理メモリに格納されています。ツリーのルートは、4096バイトのページテーブルページであり

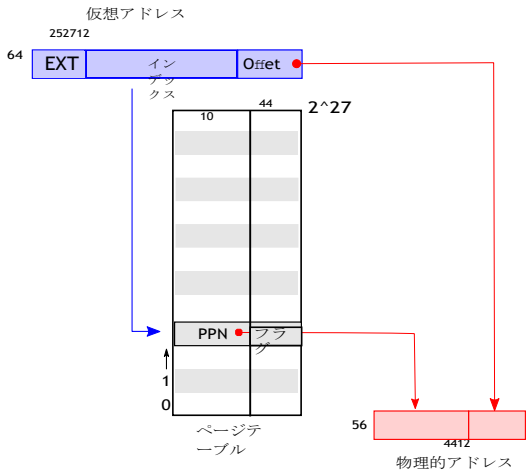


図3.1:RISC-Vの仮想アドレスと物理アドレス。

には512個のPTEが含まれており、これらのPTEには、ツリーの次のレベルのページテーブルページの物理アドレスが含まれています。それらのページにはそれぞれ、ツリーの最終レベルの512個のPTEが含まれています。ページングハードウェアは、27ビットのうち上位9ビットでルートのptable-tableページのPTEを選択し、中間の9ビットで次の階層のページテーブルページのPTEを選択し、下位の9ビットで最終的なPTEを選択します。

アドレスの変換に必要な3つのPTEのいずれかが存在しない場合、ページングハードウェアは障害を発生させます。この3階層構造により、大きな範囲の仮想アドレスがマッピングされていない場合に、ページテーブルのページ全体を省略することができます。

各PTEには、関連する仮想アドレスの使用許可をページングハードウェアに伝えるフラグビットが含まれています。PTE_V は 、PTEが存在するかどうかを示します。PTE_Vが設定されていない場合、そのページへの参照はフォールトになります（つまり、許可されません）。PTE_Rは、そのページへの読み出しを許可するかどうかを制御します。PTE_Wは、そのページへの書き込みを許可するかどうかを制御します。PTE_X は 、CPUがページの内容を命令として解釈して実行できるかどうかを制御する。PTE_U は 、ユーザーモードの命令がページにアクセスすることを許可するかどうかを制御します。PTE_Uが設定されていない場合、PTEはスーパバイザモードでのみ使用することができます。図3.2に、これらの仕組みを示します。フラグをはじめとするページハードウェア関連の構造体は，(kernel/riscv.h)で定義されています.

ハードウェアにページテーブルの使用を指示するために、カーネルはページテーブルのルートページの物理アドレスをsatpレジスタに書き込まなければなりません。各CPUはそれぞれ独自のsatpを持っています。CPUは、自身のsatp が

指すページテーブルを使用して、後続の命令が生成するすべてのアドレスを変換します。各CPUが独自のsatpを持つのは、異なるCPUが異なるプロセスを実行し、それぞれが独自のページテーブルで記述されたプライベートアドレス空間を持つことができるようにするためです。

用語についての注意点です。物理メモリとは、DRAMの記憶セルのこと。物理メモリの1バイトは、物理アドレスと呼ばれるアドレスを持つ。命令は仮想アドレスのみを使用し、ページングハードウェアが物理アドレスに変換した後、DRAMハードウェアに送り、記憶の読み書きを行う。仮想メモリとは、物理メモリと仮想アドレスを管理するためにカーネルが提供する抽象化とメカニズムの集合体を指す。

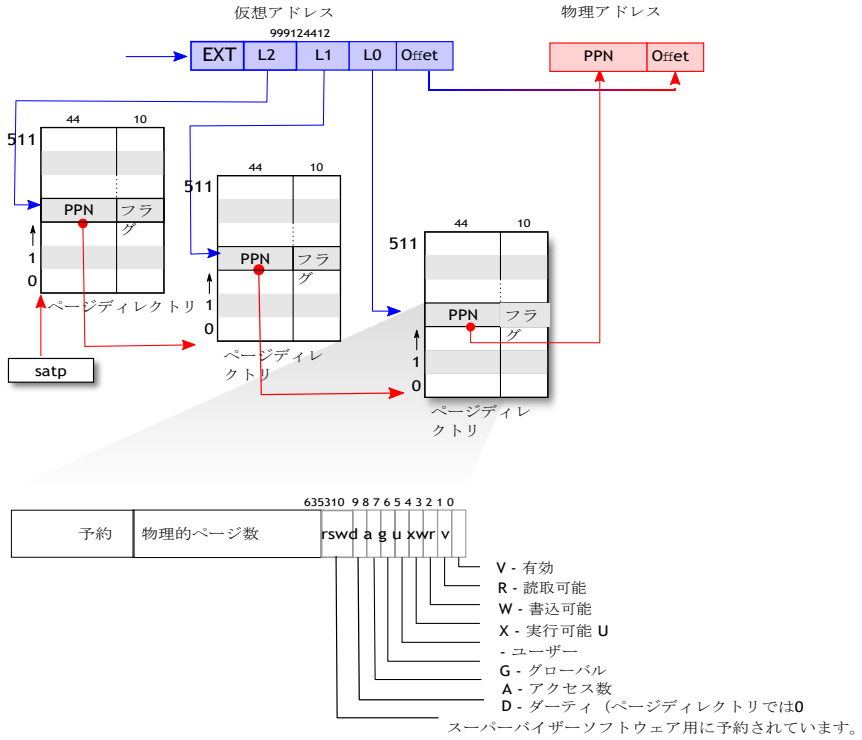


図3.2: RISC-Vのページテーブルハードウェア。

3.2 カーネルのアドレス空間

カーネルは独自のページテーブルを持っています。プロセスがカーネルに入ると、xv6はカーネルのページテーブルに切り替わり、カーネルがユーザー空間に戻ると、ユーザープロセスのページテーブルに切り替わります。カーネルのメモリはプライベートです。

図3.3は、カーネルのアドレス空間のレイアウトと、仮想アドレスから物理アドレスへのマッピングを示したものです。ファイル（kernel/memlayout.h）は，xv6のカーネルメモリレイアウトのための定数を宣言しています。

QEMUは、ディスクインターフェースなどのI/Oデバイスを含むコンピュータをシミュレートします。QEMUは、デバイスのインターフェースを、物理メモリの0x80000000以下に配置されたメモリマップド制御レジスタとしてソフトウェアに公開します。カーネルは、これらのメモリロケーションを読み書きすることで、デバイスと対話することができます。第4章では、xv6がどのようにデバイスと相互作用するかを説明します。

カーネルは、ほとんどの仮想アドレスにIDマッピングを使用しています。つまり、カーネルのアドレス空間のほとんどは「ダイレクトマッピング」されています。例えば、カーネル自身は、仮想アドレス空間でも物理メモリでもKERNBASEに位置しています。このダイレクトマッピングにより、（仮想アドレスを持つ）ページの読み書きと、（物理

アドレスを持つ）同じページを参照するPTEの操作の両方が必要なカーネルコードが簡素化されます。ダイレクトマッピングされていない仮想アドレスがいくつかあります。

- トランポリンページ。仮想アドレス空間の最上位にマッピングされており、ユーザーページテーブルも同様にマッピングされています。第4章では、トランポリンページの役割について説明しますが

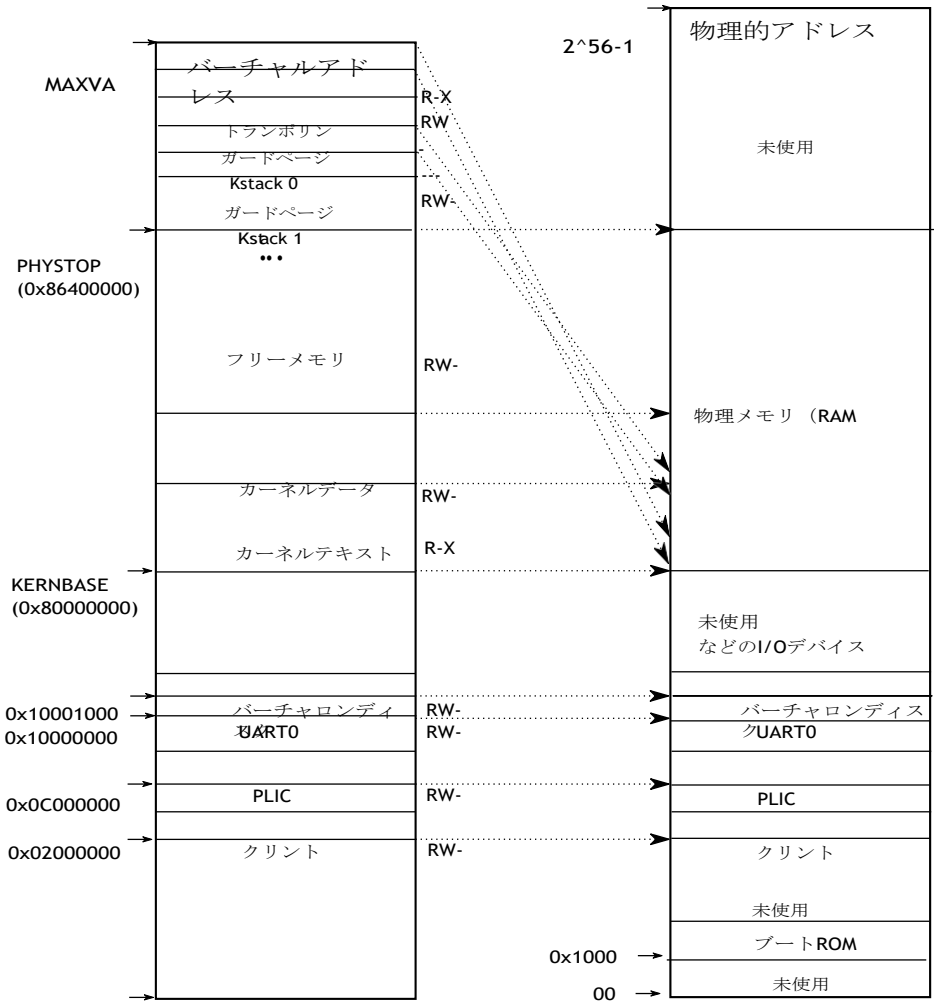


図3.3：
左側がxv6のカーネルアドレス空間。RWXはPTEの読み取り、書き込み、実行の許可を意味しています。右側は、xv6が期待するRISC-Vの物理アドレス空間。

物理ページ（トランポリンコードが格納されている）は、カーネルの仮想アドレス空間に2回マッピングされています。

- カーネルスタックのページ。各プロセスには独自のカーネルスタックがあり、その下にxv6がマッピングされていない*ガードページ*を残せるように高くマッピングされています。ガードページのPTEは無効（`PTE_V`が設定されていない）なので、カーネルがカーネルスタックをオーバーフローさせた場合、フォールトが発生してカーネルがパニックに陥る可能性が高くなります。ガードページがないと、スタックがオーバーフローした場合、他のカーネルメモリを上書きしてしまい、不正な動作になってしまいます。パニック・クラッシュの方が好ましい。

カーネルはスタックをハイメモリマッピングで使用しますが、スタックはダイレクトマッピングのアドレスでもカーネルにアクセスできます。別の設計では、ダイレクトマッピングだけになるかもしれません。

と、直接マッピングされたアドレスのスタックを使用することができます。しかし、この場合、ガードページを設けると、物理メモリを参照する仮想アドレスのマッピングが解除されてしまい、使いにくくなります。

カーネルは、トランポリン用のページとカーネルテキスト用のページをPTE_RとPTE_Xというパーミッションでマッピングします。カーネルはこれらのページから命令を読み取り、実行します。カーネルは他のページをPTE_RとPTE_Wという権限でマッピングし、これらのページのメモリを読み書きできるようにします。ガードページのマッピングは無効です。

3.3 コード：アドレス空間の作成

アドレス空間とページテーブルを操作するxv6のコードのほとんどはvm.c（kernel/vm.c:1）にあります。中心となるデータ構造はpagetable_tで、これは実際にはRISC-Vのルートページテーブルページへのポインタです。pagetable_tには、カーネルページテーブルまたはプロセスごとのページテーブルのいずれかを指定できます。中心となる関数は、仮想アドレスのPTEを見つけるwalkと、新しいマッピング用のPTEをインストールするmappagesです。kvmで始まる関数はカーネルのページテーブルを操作し、uvmで始まる関数はユーザのページテーブルを操作します。copyoutとcopyinは、システムコールの引数として与えられたユーザの仮想アドレスとの間でデータをコピーしますが、対応する物理メモリを見つけるためにそれらのアドレスを明示的に変換する必要があるため、vm.c にあります。

ブートシーケンスの初期に、mainはkvminit (kernel/vm.c:24)を呼び出し、カーネルのページテーブルを作成します。この呼び出しは、xv6がRISC-Vでページングを有効にする前に行われたので、アドレスは物理メモリを直接参照します。Kvminitはまず、ページテーブルのルートページを保持するために物理メモリのページを割り当てます。次にkvmmapを呼び出して、カーネルが必要とするトランスレーションをインストールします。トランスレーションには、カーネルの命令とデータ、PHYSTOPまでの物理メモリ、そして実際にデバイスとなるメモリ範囲が含まれます。

kvmmap (kernel/vm.c:120)はmappages (kernel/vm.c:151)を呼び出し、仮想アドレスの範囲とそれに対応する物理アドレスの範囲のマッピングをページテーブルにインストールします。この作業は、範囲内の各仮想アドレスに対して、ページ間隔で個別に行われます。マッピングされる各仮想アドレスに対して、mappagesはwalkを呼び出し、そのアドレスのPTEのアドレスを見つけます。そして、PTEを初期化して、関連する物理ページ番号、必要なパーミッション（PTE_W、PTE_X、および/またはPTE_R）、およびPTEを有効にするためのPTE_Vを保持します（kernel/vm.c:163）。

walk (kernel/vm.c:74) は RISC-V のページングハードウェアを模倣して、仮想アドレスの

PTE を調べます (図 3.2 参照)。 `walk` は 3 階層のページテーブルを 9 ビットずつ下ります。walkは各レベルの9ビットの仮想アドレスを使用して、次のレベルのページテーブルまたは最終ページのPTEを見つけます(kernel/vm.c:80)。`alloc`引数がセットされている場合、`walk`は新しいページテーブルページを割り当て、その物理アドレスをPTEに置きます。walkはツリーの最下層にあるPTEのアドレスを返します(kernel/vm.c:90)。

上記のコードは、物理メモリがカーネルの仮想アドドレス空間に直接マッピングされていることに依存しています。例えば、`walk` がページテーブルのレベルを下げる際には、次のレベルのページテーブルの（物理的な）アドレスをPTE（kernel/vm.c:82）から取り出し、そのアドレスを仮想アドレスとして使用して次のレベルのPTE（kernel/vm.c:80）を取り出します。

`main` は `kvminithart` (kernel/vm.c:55) を呼び出し、カーネルのページテーブルをインストールします。これは物理的な

ページテーブルのルートページの論理アドレスをレジスタ satp に入力します。この後、CPUはカーネルのページテーブルを使ってアドレスを変換します。カーネルはIDマッピングを使用しているので、次の命令の仮想アドレスは正しい物理メモリアドレスにマッピングされます。

main から呼び出される procinit (kernel/proc.c:24) は、各プロセスのためにカーネルスタックを割り当てます。kvmmap はマッピング PTE をカーネルページテーブルに追加し、kvminithart の呼び出しはカーネルページテーブルを satp にリロードして、ハードウェアが新しい PTE を認識できるようにします。

各RISC-Vコアは、ページテーブルのエントリを*TLB（Translation Look-aside Buffer）*にキャッシュしており、xv6がページテーブルを変更する際には、対応するキャッシュされたTLBエントリを無効にするようにCPUに指示しなければなりません。そうしないと、ある時点でTLBがキャッシュされた古いマッピングを使用して、その間に他のプロセスに割り当てられた物理ページを指し示す可能性があり、その結果、あるプロセスが他のプロセスのメモリに落書きできる可能性があるのです。RISC-Vには、現在のコアのTLBをフラッシュするsfence.vmaという命令があります。xv6では、satpレジスタを再ロードした後のkvminithartと、ユーザー空間に戻る前にユーザーページテーブルに切り替えるtrampolineコードでsfence.vmaを実行しています（kernel/trampoline.S:79）。

3.4 物理的なメモリの割り当て

カーネルは、ページテーブル、ユーザーメモリ、カーネルスタック、パイプバッファのために、ランタイムに物理メモリの割り当てと解放を行う必要があります。

xv6は、カーネルの終端とPHYSTOPの間の物理メモリをランタイムアロカションに使用します。4096バイトのページ全体を一度に割り当てたり解放したりします。どのページが空いているかは、ページ自体にリンクリストを通すことで把握しています。割り当てではリンクリストからページを削除し、解放では解放されたページをリンクリストに追加します。

3.5 コードです。物理メモリのアロケータ

アロケータはkalloc.c（kernel/kalloc.c:1）にあります。アロケータのデータ構造は、割り当てが可能な物理メモリページの*フリーリスト*です。各フリーページのリスト要素は struct run です(kernel/kalloc.c:17)。アロケータは、このデータ構造を保持するためのメモリをどこから調達しているのでしょうか。各フリーページの run 構造体は、フリーページ自体に格納されます。フリーリストはスピンロックで保護されて

います（kernel/kalloc.c:21-
24）。ロックが構造体のフィールドを保護していることを明確にするために、リストとロックは構造体でラップされています。第5章ではロックについて詳しく説明しますので、今のところはロックや獲得・解放の呼び出しは無視してください。

関数 `main` は `kinit` を呼び出してアロケータを初期化します (kernel/kalloc.c:27)。 `kinit` は カーネルの終わりから `PHYSTOP` までのすべてのページを保持するようにフリーリストを初期化します。`kinit` は `freerange` を呼び出し、`kfree` へのページごとの呼び出しによってフリーリストにメモリを追加します。PTEは4096バイト境界にアラインされた（4096の倍数である）物理アドレスしか参照できないので、`freerange`は`PGROUNDUP`を使用して、以下のことを確認します。

は、アラインドされた物理アドレスのみを解放します。アロケータはメモリがない状態で開始され、kfree
それを何とかしてあげてください。

アロケータは、アドレスを整数として扱って演算を行うこともあれば（例：フリーレンジで全ページを横断する）、アドレスをポインタとして使ってメモリを読み書きすることもあります（例：各ページに格納されているランを構造を操作する）。このようにアドレスを二重に使うことが、アロケータのコードにC型キャストが多い主な理由です。もう1つの理由は、解放と割り当てが本質的にメモリのタイプを変えることです。

関数 kfree (kernel/kalloc.c:47) は、まず、解放されるメモリのすべてのバイトに値 1 を設定します。これにより、メモリを解放した後にメモリを使用する（「ダングリング・リファレンス」を使用する）コードは、古い有効なコンテンツの代わりにゴミを読むことになります。うまくいけば、そのようなコードはより早く壊れることになります。kallocは、フリーリストの最初の要素を削除して返します。

3.6 プロセスのアドレス空間

各プロセスは個別のページテーブルを持っており，xv6
がプロセスを切り替える際にはページテーブルも変更されます．図2.3に示すように，プロセスのユーザメモリは仮想アドレス0から始まり，MAXVA（kernel/riscv.h:349）まで増やすことができるため，原理的には256ギガバイトのメモリをアドレスとして使用することができます.

プロセスがxv6にユーザーメモリの追加を要求すると、xv6はまずkallocを使用して物理ページを割り当てます。そして、新しい物理ページを指すPTEをプロセスのページテーブルに追加します。Xv6は、これらのPTEにPTE_W、PTE_X、PTE_R、PTE_U、PTE_Vのフラグを設定します。ほとんどのプロセスはユーザーアドレス空間全体を使用しないため、xv6は未使用のPTEではPTE_Vをクリアにしておきます。

ここでは、ページテーブルの使用例をいくつか紹介します。まず、各プロセスのページテーブルは、ユーザーアドレスを物理メモリの異なるページに変換し、各プロセスがプライベートなユーザーメモリを持つようにしています。第二に、各プロセスは自分のメモリをゼロから始まる連続した仮想アドレスと見なしますが、プロセスの物理メモリは非連続である可能性があります。第3に、カーネルはトランポリンコードを持つページをユーザーアドレス空間の最上位にマッピングするため、物理メモリの1ページがすべてのアドレス空間に表示されます。

図 3.4 は xv6
の実行中のプロセスのユーザメモリのレイアウトをより詳細に示しています。スタック

は 1 ページで、exec で作成された初期の内容を示しています。コマンドライン引数を含む文字列と、それらへのポインタの配列がスタックの一番上にあります。そのすぐ下には、あたかも関数main(argc, argv) が呼び出されたかのように、プログラムをmainで開始するための値があります。

ユーザースタックが割り当てられたスタックメモリをオーバーフローしたことを検出するために、xv6はスタックのすぐ下に無効なガードページを配置します。ユーザースタックがオーバーフローし、プロセスがスタックの下のアドレスを使おうとすると、マッピングが有効でないため、ハードウェアはページフォールト例外を生成します。現実のオペレーティングシステムでは、ユーザースタックがオーバーフローしたときに、代わりに自動的にユーザースタックのためのより多くのメモリを割り当てるかもしれません。

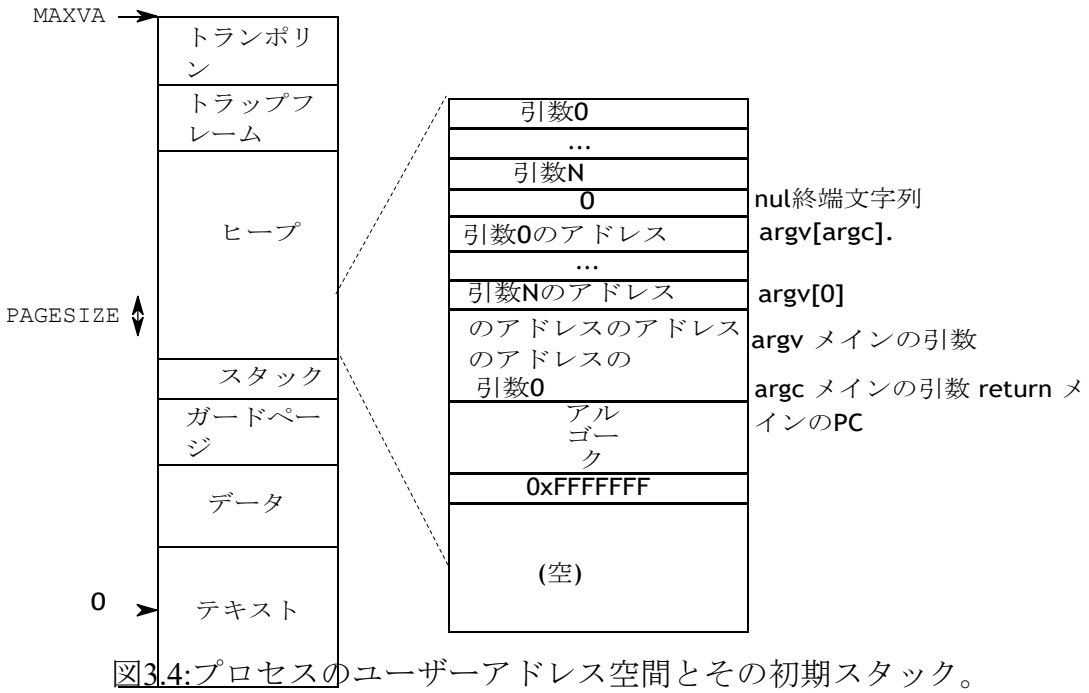


図3.4:プロセスのユーザーアドレス空間とその初期スタック。

3.7 コード: sbrk

Sbrk
は、プロセスがメモリを縮小または拡大するためのシステムコールです。このシステムコールは関数 growproc (kernel/proc.c:224) によって実装されています。 growproc は uvmalloc または uvmdealloc を呼び出しますが、その際の n の値が正か負かで決まります。uvmalloc (kernel/vm.c:236) は kalloc で物理メモリを割り当て、mappages でユーザページテーブルに PTE を追加します。uvmdealloc は uvmunmap (kernel/vm.c:176) を呼び出し、 walk を使用して PTE を見つけ、kfree を使用して参照する物理メモリを解放します。

xv6はプロセスのページテーブルを、ユーザーの仮想アドレスをどのようにマッピングするかをハードウェアに伝えるだけでなく、どの物理メモリページがそのプロセスに割り当てられているかを示す唯一の記録として使用します。これが、（uvmunmapで）ユーザーメモリを解放する際に、ユーザーページテーブルの調査が必要な理由です。

3.8 Code: exec

Exec は、アドレス空間のユーザ部分を作成するシステムコールです。ファイルシステムに格納されているファイルからアドレス空間のユーザ部分を初期化します。Exec (kernel/exec.c:13) は、第 7 章で説明した namei (kernel/exec.c:26) を用いて、名前付きのバイナリパスを開きます。そして、ELFヘッダを読み込みます。Xv6のアプリケーションは、(kernel/elf.h

)で定義されている、広く使われている*ELF フォーマット*で記述されています。ELFバイナリは、ELFヘッダ `struct elfhdr` (kernel/elf.h:6)と、それに続くプログラムセクションヘッダ `struct proghdr` (kernel/elf.h:25)のシーケンスで構成されています。各`proghdr`は、メモリにロードしなければならないアプリケーションのセクションを記述しています。xv6プログラムには1つのプログラム・セクション・ヘッダしかありませんが、他のシステムでは、命令用とデータ用に別々のセクションがあるかもしれません。

最初のステップは、そのファイルにELFバイナリが含まれているかどうかをチェックすることです。ELFバイナリ

は4バイトの「マジックナンバー」0x7F、.code 'E'、.code 'L'、.code 'F'、またはELF_MAGIC (kernel/elf.h:3)で始まります。ELFヘッダーが正しいマジックナンバーを持っていれば、exec はそのバイナリが整形されていると判断します。

Exec は proc_pagetable (kernel/exec.c:38) でユーザーマッピングのない新しいページテーブルを割り当て、uvmalloc (kernel/exec.c:52) で各 ELF セグメント用のメモリを割り当て、loadeg (kernel/exec.c:10) で各セグメントをメモリにロードします。loadeg は walkaddr を使って ELF セグメントの各ページを書き込む割り当てられたメモリの物理的なアドレスを見つけ、readi を使ってファイルから読み込みます。

execで作られた最初のユーザープログラムである/initのプログラムセクションのヘッダーは次のようになっています。

```
# objdump -p _init
                user/_init:ファイルフォーマット elf64-littleriscv

プログラムヘッダー。
    LOAD        off0x00000000000000b0 vaddr 0x0000000000000000
                                      paddr 0x0000000000000000 align
          2**3 filesz 0x0000000000000840 memsz 0x0000000000000858 flags
          rwx
   STACK        off0x00000000 vaddr 0x0000000000000000
                                      paddr 0x0000000000000000 align
          2**4 filesz 0x0000000000000000 memsz 0x0000000000000000 flags
          rw-.
```

プログラムセクションのヘッダーの filesz が memsz よりも小さい場合がありますが、これはファイルから読み込むのではなく、その間をゼロで埋めるべきであることを示しています (C のグローバル変数の場合)。/initの場合、filesz は2112バイト、memsz は2136バイトなので、uvmalloc は2136バイト分の物理メモリを確保しますが、ファイル/initからは2112バイトしか読み込みません。

ここで、exec はユーザスタックを割り当てて初期化します。スタックページを 1 つだけ確保します。exec は引数文字列を 1 つずつスタックの一番上にコピーし、それらへのポインタを ustack に記録します。また、main に渡される argv リストの最後に null ポインタを置きます。ustack の最初の 3 つのエントリは、偽のリターン PC、argc、および argv ポインタです。

Exec はスタックページのすぐ下にアクセスできないページを配置するので、1つ以上のページを使おうとするプログラムは失敗します。このアクセスできないページは、exec が大きすぎる引数を扱うことも可能にします。そのような状況では、exec が引数をスタックにコピーするために使用する copyout (kernel/vm.c:361) 関数は、宛先ページがアクセスできないことを認識し、1 を返します。

新しいメモリイメージの準備中に、exec は

無効なプログラムセグメントなどのエラーを検出すると、ラベルbadにジャンプし、新しいイメージを解放して1を返します。exec は 、システムコールが成功すると確信できるまで、古いイメージの解放を待たなければなりません。古いイメージがなくなってしまった場合、システムコールはそこに1を返すことができません。exec
の唯一のエラーケースは、イメージの作成中に起こります。イメージの作成が完了すると、exec
は新しいページテーブルにコミットし（kernel/exec.c:110）、古いページテーブルを解放します（kernel/exec.c:114）。

ExecはELFファイルからのバイトを ELFファイルで指定されたアドレスのメモリにロードします。ユーザーやプロセスは、ELFファイルに好きなアドレスを入れることができます。ELFファイルのアドレスは、偶然にも故意にもカーネルを参照する可能性があるため、execはリスクを伴います。不注意なカーネルの結果は、クラッシュから、カーネルの分離メカニズムの悪意ある破壊（すなわち、セキュリティ・エクスプロイト）まで多岐にわたります。例えば、if(ph.vaddr + ph.memsz < ph.vaddr)では、合計値が

64ビットの整数です。危険なのは、ユーザーが選んだアドレスを指すph.vaddrと、合計が0x1000にオーバーフローするほどの大きさのph.memszを持つELFバイナリを、ユーザーが有効な値のように見せかけて作ってしまうことです。ユーザーアドレス空間にカーネルも含まれていた（ただし、ユーザーモードでは読み書きできない）古いバージョンのxv6では、ユーザーはカーネルメモリに対応するアドレスを選ぶことができ、その結果、ELFバイナリのデータをカーネルにコピーすることができました。RISC-V版のxv6では、カーネルが独立したページテーブルを持っているため、このようなことは起こりません。loadegは、カーネルのページテーブルではなく、プロセスのページテーブルにロードします。

カーネルの開発者が重要なチェックを省略することは容易であり、実際のカーネルでは、ユーザー・プログラムがカーネルの権限を取得するためにチェックを省略してきた長い歴史があります。xv6はカーネルに供給されるユーザーレベルのデータの検証を完全には行っていない可能性があり、悪意のあるユーザープログラムがxv6の隔離を回避するために利用できるかもしれません。

3.9 リアルワールド

多くのオペレーティングシステムと同様に、xv6はメモリ保護とマッピングのためにページングハードウェアを使用します。例えば、xv6にはディスクからのデマンドページング、コピーオンライトフォーク、共有メモリ、レイジーに割り当てられたページ、自動的に拡張されるスタック、メモリマップされたファイルなどがありません。RISC-Vは物理アドレスレベルでの保護をサポートしていますが、xv6はその機能を使用していません。大容量のメモリを搭載したマシンでは、RISC-Vがサポートする"スーパーページ"を使用することが有効です。スモールページは、物理的なメモリが少ない場合に、細かい粒度でディスクへの割り当てとページアウトを可能にします。例えば、あるプログラムが8キロバイトのメモリしか使わない場合、4メガバイトの物理メモリ全体にスーパーページを与えるのは無駄なことです。より大きなページは次のような場合に有効です。大容量のRAMを搭載したマシンでは、ページテーブル操作のオーバーヘッドを減らすことができます。

xv6カーネルにはスモールオブジェクト用のメモリを提供するmallocのようなアロケータがないため、ダイナミックアロケーションを必要とするような高度なデータ構造を使用することができません。メモリの割り当ては一昔前に話題になったことがあり、限られたメモリを効率的に使用することと、将来の未知の要求に備えることが基本的な問題でした[6]。今日、人々はスペース効率よりもスピードを重視しています。さらに、より精巧なカーネルでは、（xv6のように）4096バイトのブロックだけではなく、さまざまなサイズの小さなブロックを多数割り当てることになるでしょう。

は、大規模なアロケーションだけでなく、小規模なアロケーションも扱う必要があります。

3.10 エクササイズ

1. RISC-
   Vのデバイスツリーを解析し、コンピュータに搭載されている物理メモリの量を調べます。

2. `sbrk(1)`を呼び出してアドレス空間を1バイト拡張するユーザープログラムを書いてください。このプログラムを実行し、`sbrk`
   を呼び出す前のプログラムと呼び出した後のプログラムのページテーブルを調べます。カーネルはどのくらいのスペースを割り当てましたか？新しいメモリのページテーブルには何が含まれていますか？

3. カーネルにスーパーページを使用するようにxv6を変更します。

4. xv6を修正して、ユーザープログラムがNULLポインタを参照したときに、フォールトを受け取るようにします。つまり、仮想アドレス0がユーザープログラム用にマッピングされないようにxv6を修正します。

5. Unixのexecの実装には、伝統的にシェルスクリプトに対する特別な処理が含まれている。実行するファイルが `#!` というテキストで始まっている場合、最初の行はそのファイルを解釈するために実行するプログラムとみなされる。例えば、`myprog arg1` を実行するために `exec` が呼び出され、`myprog` の最初の行が `#!/interp` であった場合、`exec` は `/interp myprog arg1` というコマンドラインで `/interp` を実行する。xv6でこの規約のサポートを実装する。

6. カーネルにアドレス空間のランダム化を実装。

第4章

トラップとデバイスドライバー

何らかのイベントによってCPUが通常の命令の逐次実行をやめ、そのイベントを処理する特別なコードに制御を移さなければならない状況が3つあります。1つはシステムコールで、ユーザープログラムが`ecall`命令を実行してカーネルに何かを依頼する場合です。もうひとつの状況は、*例外*です。ユーザーまたはカーネルの命令が、ゼロ除算や無効な仮想アドレスの使用など、不正な処理を行った場合です。3つ目の状況は、デバイスの*割り込みです*。これは、デバイスが注意を払う必要があることを通知するもので、例えば、ディスクのハードウェアが読み取りまたは書き込みの要求を終了したときなどに起こります。

本書では、このような状況の総称として*トラップ*を使用しています。通常、トラップが発生したときに実行していたコードは、後で再開する必要があり、何か特別なことが起こったことを意識する必要はありません。つまり、トラップは透過的であることが望まれます。これは、割り込みコードが通常期待していない割り込みの場合に特に重要です。したがって、通常のシーケンスは、トラップによって強制的に制御がカーネルに移され、カーネルがレジスタやその他の状態を保存して実行を再開できるようにし、カーネルが適切なハンドラコード（システムコールの実装やデバイスドライバなど）を実行し、カーネルが保存された状態を復元してトラップから戻り、元のコードが中断したところから再開するというものです。

xv6カーネルはすべてのトラップを処理します。これはシステムコールでは当然のことで、ユーザープロセスがデバイスを直接使用しないように隔離が要求されていることや、デバイス処理に必要な状態はカーネルだけが持っていることから、割り込みでは理にかなっています。また、xv6はユーザースペースからのすべての例外に対して問題のあるプログラムを殺すことで応答することから、例外でも理にかなっています。

Xv6のトラップ処理は、RISC-V CPUによるハードウェアアクション、カーネルCコードの準備を行うアセンブリ「ベクター」、トラップの処理を決定するCトラップハンドラ、システムコールまたはデバイ

スドライバサービスルーチンの4段階で行われます。3種類のトラップには共通性があるため、カーネルは単一のコードパスですべてのトラップを処理することができますが、カーネル空間からのトラップ、ユーザー空間からのトラップ、タイマー割り込みという3つの異なるケースには、別々のアセンブリベクターとCトラップハンドラを用意した方が便利であることがわかりました。

本章の最後に、デバイスドライバについて説明します。デバイス処理はトラップとは別のテーマですが、カーネルとデバイスハードウェアとのやりとりが割り込みによって行われることが多いため、ここでは取り上げます。

4.1 RISC-Vトラップマシン

RISC-V
はいくつかの制御レジスタをサポートしています。これらのレジスタはカーネルが書き込んで CPU
に割り込みの処理方法を指示したり、カーネルが読み込んで発生した割り込みについて調べたりします。`riscv.h` (kernel/riscv.h:1) には xv6
が使用する定義が含まれています．ここでは，最も重要なレジスタの概要を説明します
.

- **stvec**です。カーネルはここにトラップハンドラのアドレスを書き込み、RISC-Vはここにジャンプしてトラップを処理します。

- **sepc**です。トラップが発生すると、RISC-Vはここにプログラムカウンタを保存します（その後、`pc` は`stvec`に置き換えられるため）。`sret`（トラップからの復帰）命令は、`sepc` を`pc`にコピーします。カーネルは`sepc`に書き込み、`sret` の行き先を制御することができます。

- **scause**です。RISC-Vでは、ここにトラップの理由を記した数字を入れます。

- **sscratch**です。カーネルはここに、トラップハンドラの最初の段階で便利な値を配置します。

- **sstatus**です。SIEビットは、デバイス割り込みを有効にするかどうかを制御します。カーネルがSIEをクリアした場合、RISC-VはカーネルがSIEをセットするまでデバイス割り込みを延期します。SPPビットは、トラップがユーザモードから来たのか、スーパバイザモードから来たのかを示し、`sret`がどのモードに戻るかを制御します。

これらは、スーパバイザモードで処理される割り込みに関するもので、ユーザモードでは読み書きできません。マシンモードで処理される割り込みには、同等のコントロールレジスタ群がありますが、xv6ではタイマー割り込みという特殊なケースでのみ使用しています。

RISC-Vハードウェアは、強制的にトラップを発生させる必要がある場合、すべてのトラップタイプ(タイマー割り込みを除く)について以下の処理を行います。

1. トラップがデバイスの割り込みで、`sstatus` SIEビットがクリアされている場合は、以下のようなことはしないでください。

2. SIEをクリアすることで、割り込みを無効にします。

3. pcをsepcにコピーします。
4. 現在のモード（ユーザまたはスーパバイザ）をsstatusのSPPビットに保存します。
5. 割り込みの原因を反映させるためにスカイスを設定します。
6. モードをスーパーバイザーに設定します。
7. stvecをPCにコピーします。
8. 新しいPCで実行を開始します。

ここで重要なのは、CPUがこれらのステップを1つの動作として行うことです。例えば、CPUがプログラムカウンタの切り替えを行わないなど、これらのステップのうち1つが省略された場合を考えてみましょう。そうすると、トラップはユーザ命令を実行したままスーパバイザモードに切り替わる可能性があります。これらのユーザ命令は、例えば、物理メモリのすべてにアクセスできるページテーブルを指すようにsatpレジスタを変更するなどして、ユーザとカーネルの分離を破ることができます。そのため、トラップのエントリーポイントは、ユーザープログラムではなく、カーネルが指定することが重要です。

なお、CPUはカーネルのページテーブルに切り替えたり、カーネル内のスタックに切り替えたり、pc以外のレジスタを保存したりすることはありません。これらの作業は、必要に応じてカーネルが行う必要があります。トラップ中にCPUが最小限の作業しかしないのは、ソフトウェアに柔軟性を持たせるためです。例えば、ページテーブルの切り替えが必要ない状況もあり、それによってパフォーマンスが向上する場合もあります。

4.2 カーネルスペースからのトラップ

xv6カーネルがCPU上で実行されているとき、例外とデバイス割り込みの2種類のトラップが発生する可能性があります。前節では、このようなトラップに対するCPUの対応について説明しました。

カーネルが実行されると、stvecはkernelvecのアセンブリコードを指します（kernel/kernelvec.S:10）。xv6はすでにカーネル内にあるので、カーネルvecはsatpがカーネルのページテーブルに設定されていることと、スタックポインタが有効なカーネルスタックを参照していることに依存することができます。カーネルvecはすべてのレジスタを保存するので、中断されたコードを妨げることなく最終的に再開することができます。

kernelvecは、中断されたカーネルスレッドのスタック上にレジスタを保存しますが、これはレジスタ値がそのスレッドに属するからです。このことは、トラップによって別のスレッドに切り替わった場合に特に重要です。その場合、トラップは実際に新しいスレッドのスタック上に戻り、中断されたスレッドの保存されたレジスタはそのスタック上に安全に残されます。

kernelvecはレジスタを保存した後、kerneltrap (kernel/trap.c:134)にジャンプします。kerneltrapはデバイス割り込みと例外の2種類のトラップに備えています。devintr (kernel//-trap.c:177)を呼び出して前者をチェックし、処理します。トラップがデバイス割り込みでなければ、それは例外であり、それがカーネル内で発生した場合は常に致命的なエラーとなります。

タイマー割り込みによってkerneltrapが呼び出され、プロセスのカーネルスレッドが（スケジューラースレッドではなく）実行されている場合、kerneltrapは他のスレッドに

実行のチャンスを与えるためにyieldを呼び出します。ある時点で、それらのスレッドの1つが yield を行い、私たちのスレッドとその kerneltrap を再び再開させます。yieldで何が起こるかは第6章で説明します。

kerneltrapの作業が終了したら、トラップによって中断されたコードに戻る必要があります。sstatus に保存されていた sepc や保存されていた previous mode が yield によって乱された可能性があるため、kerneltrap は起動時にそれらを保存します。kernelvec は保存されたレジスタをスタックからポップして sret を実行し、sepc を pc にコピーして中断されたカーネルコードを再開します。

タイマー割り込みによってkerneltrapがyieldを呼び出した場合、トラップリターンがどのように起こるかを考えてみる価値はあるでしょう。

Xv6では、CPUがユーザ空間からカーネルに入ると、CPUのstvecがkernelvecに設定されます。これはusertrap(kernel/trap.c:29)で確認できます。これは usertrap (kernel/trap.c:29) で見ることができます。カーネルが実行されていても stvec が間違った値になっている時間帯があり、その間はデバイス割り込みを無効にすることが重要です。

のウィンドウに表示されます。幸運なことに、RISC-
Vはトラップを取り始めると常に割り込みを無効にし、xv6はstvecを設定するまで再び割り込みを有効にしません。

4.3 ユーザースペースからのトラップ

トラップは，ユーザ空間での実行中に，ユーザプログラムがシステムコール（ecall命令）を行ったり，不正な動作をしたり，デバイスが割り込みを行ったりした場合に発生する。ユーザー空間からのトラップの高レベルの経路は、uservec (kernel/trampoline.S:16)→usertrap (kernel/trap.c:37)の順になり、戻るときはusertrapret (kernel/trap.c:90)→userret (kernel/trampoline.S:16)となります。

ユーザーコードからのトラップは、カーネルからのトラップよりも難易度が高いです。なぜなら、satpはカーネルをマッピングしていないユーザーページテーブルを指しており、スタックポインタには無効な値、あるいは不正な値が含まれている可能性があるからです。

RISC-Vハードウェアはトラップ中にページテーブルを切り替えないので、ユーザーページテーブルにはstvecが指すトラップベクター命令のマッピングを含める必要があります。さらに、トラップベクターはカーネルページテーブルを指すようにsatpを切り替えなければならず、クラッシュを避けるためには、ベクター命令がカーネルページテーブルでユーザーページテーブルと同じアドレスにマッピングされている必要があります。Xv6はこれらの制約を、トラップベクターコードを含む*トランポリン*ページで満たします。Xv6は、トランポリンページを、カーネルページテーブルとすべてのユーザーページテーブルの同じ仮想アドレスにマッピングします。この仮想アドレスが「TRAMPOLINE」です（図2.3や図3.3で見たとおりです）。トランポリンの内容はtrampoline.Sに設定されており、（ユーザーコード実行時には）stvecがuservec (kernel/trampoline.S:16)。

uservecの起動時には、すべてのレジスタに割り込みコードが所有する値が入っています。しかし、uservecは、satpを設定し、レジスタを保存するアドレスを生成するために、いくつかのレジスタを変更することができる必要があります。RISC-Vは、sscratchレジスタという形で助け舟を出してくれます。uservecの最初にあるcsrrw命令は、a0とsscratchの内容を入れ替えます。これでユーザーコードのa0は保存され、uservecは1つのレジスタ(a0)で遊ぶことができ、a0にはカーネルが以前sscratchに入れた値が入ります。

uservecの次のタスクは、ユーザレジスタの保存です。ユーザー空間に入る前に、カーネルは、すべてのユーザーレジスタを保存するスペースを持つプロセスごとのトラップフレームを指すようにsscratchを設定します（kernel/proc.h:44）。satpはまだユーザーページテーブルを参照しているので、uservecはトラップフレームをユーザーアドレス空間にマッピングする必要があります。xv6は各プロセスを生成する際に、プロセスのトラップフレーム用のページを割り当て、常にTRAMPOLINEのすぐ下にあるユーザー仮

想アドレスTRAPFRAMEにマッピングされるように手配しています。プロセスのp->tfはトラップフレームを指していますが、その物理アドレスはカーネルのページテーブルを介してアクセスできるようになっています。

したがって、a0とsscratchをスワップした後のa0は、現在のプロセスのトラップフレームへのポインタを保持しています。

uservecは、sscratchから読み込まれたユーザーのa0を含む、すべてのユーザーレジスタをそこに保存するようになりました。

trapframeには，現在のプロセスのカーネルスタック，現在のCPUのhartid，usertrapのアドレス，カーネルページテーブルのアドレスへのポインタが含まれています．uservecはこれらの値を取得し，satpをカーネルページテーブルに切り替え，usertrapを呼び出します.

usertrapの仕事は、kerneltrapと同様に、トラップの原因を判断し、処理して返すことです(kernel/trap.c:37)。前述のように、まずカーネルモードからのトラップを処理するために、stvecをカーネルvecに変更します。プロセススイッチがあるかもしれないので、やはりsepcを保存します。

は、sepcの上書きを引き起こす可能性のあるusertrapに含まれています。トラップがシステムコールの場合はsyscallが、デバイス割り込みの場合はdevintrが、そうでない場合は例外が発生し、カーネルは障害を起こしたプロセスを殺します。RISC-Vでは、システムコールの場合、ecall命令を指すプログラムポインタを残しておくので、システムコールのパスは、保存されたユーザpcに4つ追加されます。途中、usertrapはプロセスが殺されたか、CPUをゆずるべきかをチェックします（このトラップがタイマー割り込みの場合）。

ユーザー空間に戻るための最初のステップは、usertrapret (kernel/trap.c:90)の呼び出しです。この関数は、ユーザー空間からの将来のトラップに備えてRISC-V制御レジスタを設定します。これには、stvec をuservecを参照するように変更すること、uservec が依存するtrapframeフィールドを準備すること、sepc を以前に保存されたユーザプログラムカウンタに設定することなどが含まれます。最後に、usertrapret は、ユーザとカーネルの両方のページテーブルにマッピングされたトランポリンページ上のuserretを呼び出します。これは、userret内のアセンブリコードがページテーブルを切り替えるためです。

usertrapretの呼び出しでは、a0にプロセスのユーザページテーブルへのポインタを、a1にTRAPFRAMEを渡しています(kernel/trampoline.S:88)。userretはsatpをプロセスのユーザページテーブルに切り替えます。ユーザーページテーブルは、トランポリンページとTRAPFRAMEの両方をマッピングしていますが、カーネルからの他の情報は何もないことを思い出してください。ここでも、トランポリンページがユーザーページテーブルとカーネルページテーブルで同じ仮想アドレスにマッピングされていることが、satpを変更した後もuservecの実行を継続させる要因となっています。

trapretは、後でTRAPFRAMEとスワップすることに備えて、トラップフレームの保存されたユーザーa0をsscratchにコピーします。この時点から、userretが使用できるデータは、レジスタの内容とトラップフレームの内容だけになります。次にuserretはトラップフレームから保存されたユーザーレジスタを復元し、ユーザーa0を復元して次のトラップのためにTRAPFRAMEを保存するために、a0とsscratchの最終的なスワップを行い、sretを使用してユーザー空間に戻ります。

4.4 タイマー割込み

Xv6はタイマー割り込みを使用してクロックを維持し、計算処理の切り替えを可能にしています。この切り替えはusertrapとkerneltrapのyieldコールによって行われます。タイマー割り込みは、RISC-Vの各CPUに取り付けられたクロックハードウェアから供給されます。Xv6はこのクロックハードウェアをプログラムし、各CPUに定期的に割り込みをかけます。

RISC-

Vでは、スーパーバイザーモードではなく、マシンモードでタイマー割り込みを行う必要があります。RISC-
Vのマシンモードは、ページングなしで実行され、別の制御レジスタのセットを使用するため、通常のxv6カーネルコードをマシンモードで実行することは実用的ではありません。そのため、xv6ではタイマー割り込みを上記のトラップ機構とは完全に分けて処理しています。

マシンモードで実行されるコードは、メインの前にあるstart.cで、タイマー割り込みを受け取るための設定を行います（kernel/start.c:56）。この仕事の一部は、CLINTハードウェア（コア・ローカル・インタラプタ）をプログラムして、一定の遅延の後に割り込みを発生させることです。もう一つの仕事は、タイマー割り込みハンドラがレジスタを保存するための、トラップフレームに似たスクラッチ領域を設定し、CLINTレジスタのアドレスを見つけることです。最後に、スタートはmtvecをtimervecに設定し、タイマー割り込みを有効にします。

タイマー割り込みは、ユーザーコードやカーネルコードの実行中にいつでも発生する可能性があり、カーネルが重要な動作中にタイマー割り込みを無効にする方法はありません。したがって、タイマ割込みハンドラは、割込まれたカーネルコードを妨害しないように仕事をしなければなりません。基本的な戦略は、タイマー割り込みがRISC-
Vに「ソフトウェア割り込み」を発生させるように要求し、直ちにリターンすることです。タイマー割り込みは

RISC-
Vでは、通常のトラップ機構でソフトウェア割り込みをカーネルに配信し、カーネルがそれを無効にすることができます。タイマー割り込みで発生するソフトウェア割り込みを処理するコードは、devintr（kernel/trap.c:197）で見ることができます。

マシンモードのタイマー割り込みベクターは timervec (kernel/kernelvec.S:93)です。timervecは、スタートで用意されたスクラッチ領域にいくつかのレジスタを保存し、CLINTに次のタイマー割り込みをいつ発生させるかを指示し、RISC-
Vにソフトウェア割り込みを発生させるように要求し、レジスタを復元して戻ります。はありません。
タイマー割り込みに関わるCコード。

4.5 コードシステムコールの呼び出し

第2章は、initcode.Sがexecシステムコールを呼び出すところで終わりました(user/initcode.S:11)。ユーザー・コールがカーネル内のexecシステム・コールの実装に至るまでの流れを見てみましょう。

ユーザコードは，execの引数をレジスタa0とa1に置き，システムコール番号をa7 に 置く
。システムコール番号は，関数ポインタのテーブルであるsyscalls配列のエントリと一致します（kernel/syscall.c:108）．ecall命令はカーネルにトラップして，上で見たようにuservec，usertrap，そしてsyscallを実行します.

Syscall (kernel/syscall.c:133) は、保存された a7 を含む trapframe からシステムコール番号をロードし、システムコールテーブルにインデックスを付けます。最初のシステムコールでは、a7 には SYS_exec (kernel/syscall.h:8) という値が含まれており、syscall はシステムコールテーブルの SYS_exec'th エントリを呼び出しますが、これは sys_exec を呼び出すことに相当します。

Syscallは、システムコール関数の戻り値をp->tf-
>a0に記録します。システムコールがユーザー空間に戻ると、userretはp-
>tfの値をマシンレジスタにロードし、sretを使ってユーザー空間に戻ります。したがって、execがユーザ空間に戻るときには、システムコールハンドラが返した値をa0に返すことになります(kernel/syscall.c:140)。システムコールは通常、エラーを示す負の数を返し、成功を示す0または正の数を返します。システムコール番号が無効な場合、syscallはエラーを表示し、-1を返します。

4.6 コードシステムコール引数

この後の章では、特定のシステムコールの実装について説明します。この章では、システムコールのメカニズムについて説明します。システムコールの引数を見つけるという

メカニズムが1つ残っています。

RISC-
VにおけるC言語の呼び出し規約では、引数をレジスタで渡すことが規定されています。システムコールの間、これらのレジスタ（保存されたユーザレジスタ）はトラップフレーム、p-
>tfで利用可能です。関数argint、argaddr、argfdは、*n*番目のシステムコール引数を、整数、ポインタ、ファイルディスクリプタのいずれかとして取得します。これらの関数はすべて、保存されたユーザレジスタの1つを取得するためにargrawを呼び出します(kernel/syscall.c:35)。

システムコールの中には、引数としてポインターを渡すものがあり、カーネルはそのポインターを使ってユーザーメモリを読み書きする必要があります。例えば、execシステムコールは、ユーザー空間の文字列引数を参照するポインタの配列をカーネルに渡します。これらのポインターには2つの課題があります。まず、ユーザ・プログラムがバグを持っていたり、悪意を持っていたりして、無効なポインタや、意図されたポインタをカーネルに渡してしまうことがあります。

を使ってカーネルを騙し、ユーザーメモリではなくカーネルメモリにアクセスさせることができます。次に、xv6のカーネルのページテーブルのマッピングは、ユーザーのページテーブルのマッピングと同じではないため、カーネルは通常の命令を使ってユーザーが提供したアドレスからロードやストアを行うことができません。

多くのカーネル関数は、ユーザー空間からの安全な読み込みを行う必要があります。`fetchstr`はその一例です（kernel/syscall.c:25）。`fetchstr`は`copyinstr`を呼び出し、ユーザーページテーブルの仮想アドレスを検索し、それをカーネルが使用できるアドレスに変換し、そのアドレスから文字列をカーネルにコピーします。

`copyinstr`（kernel/vm.c:412）は、ユーザページテーブル`pagetable`の仮想アドレス`srcva`から最大バイトを`dst`にコピーします。`srcva`の物理アドレス`pa0`を決定するために、`walkaddr`（`walk`を呼び出す）を使用してソフトウェアでページテーブルを探索します。`walkaddr`（kernel/vm.c:97）は、ユーザが提供した仮想アドレスがプロセスのユーザ・アドレス空間の一部であることをチェックするので、プログラムがカーネルを騙して他のメモリを読み込ませることはできません。同様の機能である`copyout`は，カーネルのデータをユーザが指定したアドレスにコピーします.

4.7 デバイスドライバー

*ドライバー*とは、特定のデバイスを管理するオペレーティングシステムのコードです。ドライバーは、デバイスのハードウェアに操作を実行するように指示し、実行時に割り込みを生成するようにデバイスを設定し、結果として生じる割り込みを処理し、デバイスからのI/Oを待っている可能性のあるプロセスと対話します。ドライバは、管理するデバイスと同時に実行されるため、ドライバコードは厄介なものです。さらに、ドライバはデバイスのハードウェアインターフェースを理解しなければなりませんが、そのインターフェースは複雑で文書化されていない場合があります。

オペレーティングシステムの注意を必要とするデバイスは、通常、トラップの一種である割込みを発生させるように設定できます。カーネルのトラップ処理コードは、デバイスが割り込みを発生させたことを認識し、ドライバの割り込みハンドラを呼び出すことができなければなりません。xv6では、このディスパッチは`devintr`で行われます（kernel/trap.c:177）。

多くのデバイスドライバーは、プロセスの一部として実行されるコードと、プロセスの一部として実行されるコードの2つに大別されます。

は割込み時間に実行されます。プロセスレベルのコードは、デバイスにI/Oを実行させる`read`や`write`などのシステムコールによって駆動されます。このコードは、ハードウェアに操作の開始を要求し（例えば、ディスクにブロックの読み取りを要求する）、コードは操作が完了するのを待ちます。最終的にデバイスは操作を完了し、割り込みを発生させます。ドライバの割り込みハンドラは、どのような操作が完了したかを把握し、必要

に応じて待機中のプロセスを起こし、おそらくハードウェアに待機中の次の操作の作業を開始するように指示します。

4.8　コードです。コンソールドライバー

ドライバの構造を簡単に説明すると、コンソールドライバです。コンソールドライバは、RISC-
Vに搭載された*UART*シリアルポートハードウェアを介して、人間が入力した文字を受け取ります。ドライバは一度に1行分の入力を蓄積し、バックスペースやcontrol-
uなどの特殊な入力文字を処理します。

シェルなどのユーザープロセスでは、readシステムコールを使用してコンソールから入力行を取得できます。QEMUでxv6に入力すると、QEMUのシミュレートされたUARTハードウェアを介してキーストロークがxv6に送信されます。

ドライバが通信するUARTハードウェアは、QEMUでエミュレートされた16550チップ[8]です。実際のコンピューターでは、16550はターミナルや他のコンピューターに接続するRS232シリアルリンクを管理します。QEMUを実行しているときは、キーボードとディスプレイに接続されています。

UARTハードウェアは、メモリマップされた制御レジスタのセットとしてソフトウェアに表示されます。つまり、RISC-
VハードウェアがUARTデバイスに接続する物理アドレスがいくつかあり、ロードやストアがRAMではなくデバイスのハードウェアと連動するようになっています。UARTのメモリマップドアドレスは0x10000000、つまりUART0です（kernel/memlayout.h:21）。UARTには、それぞれが1バイトの幅を持ついくつかの制御レジスタがあります。UART0からのオフセットは(kernel/uart.c:22)で定義されています。例えば、LSRレジスタには、入力された文字がソフトウェアによって読み取られるのを待っているかどうかを示すビットが含まれています。これらの文字（もしあれば）は、RHRレジスタから読み取ることができます。UARTハードウェアは、文字が読み込まれるたびに、待機中の文字の内部FIFOからその文字を削除し、FIFOが空になるとLSRの「ready」ビットをクリアします。

Xv6のmainはconsoleinit（kernel/console.c:189）を呼び出してUARTハードウェアを初期化し、入力割り込みを生成するようにUARTハードウェアを設定します（kernel/uart.c:34）。

xv6シェルは、init.c（user/init.c:15）で開いたファイルディスクリプターを経由してコンソールから読み込みます。
consoleread は、入力が (割り込み経由で) 到着して cons.buf にバッファリングされるのを待ち、入力をユーザ空間にコピーし、 (1行すべてが到着した後)
ユーザプロセスに戻ります。ユーザーがまだ全行を入力していない場合、読み取りプロセスはsleepコール（kernel/con-
sole.c:103）で待機します（sleepの詳細については第6章で説明します）。

ユーザーが文字を入力すると、UARTハードウェアはRISC-
Vに割り込みを発生させるよう要求します。RISC-
Vとxv6は上述のように割り込みを処理し、xv6のトラップ処理コードはdevintrを呼び出します（kernel/trap.c:177）。devintrはRISC-
Vのscauseレジスタを見て、割り込みが外部デバイスからのものであることを発見します。devintrはRISC-
Vのscauseレジスタを見て、割り込みが外部デバイスからのものであることを発見し、PLIC[1]と呼ばれるハードウェアユニットにどのデバイスが割り込んできたのかを問い合わせる（kernel/trap.c:186）。UARTだった場合、devintrはuartintrを呼び出します。

uartintr (kernel/uart.c:84) は、待っている入力文字を UART ハードウェアから読み取り、consoleintr に渡します。これ以上入力すると新しい割り込みが発生するため、文字を待つことはありません。consoleintrの仕事は、全行が到着するまでcons.bufに入力文字を蓄積することです。
consoleintrは、バックスペースやその他いくつかの文字を特別に扱います。改行が来ると、consoleintr は待機している consoleread を起こします (存在する場合)。

consolereadは、cons.bufの全行を観察し、それをユーザースペースにコピーして、（システムコール機構を介して）ユーザースペースに戻ります。

マルチコアマシンでは、割り込みはどのCPUにも届く可能性がありますが、その判断はPLICが行います。割り込みは、コンソールから読み込んだプロセスを実行しているCPUと同じCPUに届くこともあれば、まったく関係のないことをしているCPUに届くこともあります。したがって、割り込みハンドラは、割り込みをかけたプロセスやコードについて考えることはできません。

4.9 リアルワールド

すべてのプロセスのユーザーページテーブルにカーネルメモリをマッピングすれば、特別なトランポリンページの必要性はなくなります。そうすれば、ユーザー空間からカーネルにトラッピングする際に、ページテーブルを切り替える必要がなくなります。そうすれば、カーネル内のシステムコールの実装で、現在のプロセスのユーザーメモリがマッピングされていることを利用できるようになり、カーネルコードがユーザーポインタを直接参照解除できるようになります。多くのオペレーティングシステムは、効率を上げるためにこのようなアイデアを使用しています。Xv6では、ユーザーポインタの不用意な使用によるカーネルのセキュリティバグの可能性を減らすため、また、ユーザーとカーネルの仮想アドレスが重ならないようにするために必要な複雑さを軽減するために、これらのアイデアを避けています。

Xv6では、ユーザープログラムの実行時だけでなく、カーネルでの実行時にもデバイスやタイマーの割り込みを許可しています。タイマー割り込みは、カーネル実行中であっても、タイマー割り込みハンドラからのスレッドスイッチ（`yield`の呼び出し）を強制します。カーネルのスレッドがユーザースペースに戻らずに長時間計算することがある場合、カーネルのスレッド間でCPUを公平にタイムスライスする機能は有用です。しかし、カーネル・コードが（タイマー割り込みのために）一時停止し、後で別のCPUで再開するかもしれないことを念頭に置く必要があることが、xv6の複雑さの原因となっています。デバイスやタイマの割り込みがユーザ・コードの実行中にのみ発生するのであれば、カーネルはもう少しシンプルになるでしょう。

一般的なコンピュータに搭載されているすべてのデバイスを完全にサポートするのは大変な作業です。なぜなら、デバイスの数は多く、デバイスには多くの機能があり、デバイスとドライバの間のプロトコルは複雑で文書化されていない可能性があるからです。多くのオペレーティングシステムでは、ドライバがコアカーネルよりも多くのコードを占めています。

UARTドライバは、UART制御レジスタを読み込んで1バイトずつデータを取得します。データの移動はソフトウェアが行うため、このパターンは*プログラムされたI/O*と呼ばれます。プログラムされたI/Oはシンプルですが、高速データレートで使用するには遅すぎます。大量のデータを高速で移動させる必要がある機器では、一般的に*ダイレクトメモリアクセス（DMA）*を使用します。DMAデバイスのハードウェアは、受信データをRAMに直接書き込み、送信データをRAMから読み出す。最近のディスクやネットワーク機器はDMAを採用している。DMAデバイスのドライバは、RAMにデータを用意し、制御レジスタへの1回の書き込みで、用意したデータの処理をデバイスに指示する。

割り込みは、デバイスが予想外のタイミングで注意を必要とする場合に有効で、あまり頻繁ではありません。しかし、割り込みはCPUのオーバーヘッドが大きい。そこで

、ネットワークやディスク装置などの高速デバイスでは、割り込みの必要性を減らす工夫がなされています。一つは、送受信の要求をまとめて一つの割込みにする方法。また、ドライバが割り込みを完全に無効にして、デバイスに注意が必要かどうかを定期的にチェックするという方法もあります。この手法を*ポーリング*と呼びます。ポーリングは、デバイスが非常に高速に動作する場合には意味がありますが、デバイスがほとんどアイドル状態の場合にはCPU時間を浪費します。ドライバの中には、現在のデバイスの負荷に応じて、ポーリングと割り込みを動的に切り替えるものがあります。

UARTドライバは、受信データをまずカーネル内のバッファにコピーし、次にユーザースペースにコピーします。これは低いデータレートでは理にかなっていますが、このような二重コピーは、非常に速くデータを生成または消費するデバイスのパフォーマンスを著しく低下させます。オペレーティングシステムの中には、ユーザー空間のバッファとデバイスのハードウェアの間でデータを直接移動できるものがあり、多くの場合DMAを使用します。

4.10 エクササイズ

1. `uartputc` (kernel/uart.c:61) は、UART
   デバイスをポーリングして、前の出力文字で終了するのを待ちます。代わりに割り込みを使用するように変換します。

2. イーサネットカードのドライバを追加する。

第5章

ロッキング

xv6を含むほとんどのカーネルは、複数のアクティビティの実行をインターリーブしています。xv6のRISC-
Vのように、複数のCPUが独立して実行されるコンピュータのことです。複数のCPUは物理的なRAMを共有しており、xv6はこの共有を利用して、すべてのCPUが読み書き可能なデータ構造を保持します。この共有により、あるCPUがデータ構造を読み込んでいる間に別のCPUがデータ構造を更新したり、複数のCPUが同じデータを同時に更新したりする可能性があります。このような並列アクセスは、慎重に設計しなければ、誤った結果やデータ構造の破壊を招く可能性があります。また、ユニプロセッサであっても、カーネルがCPUを複数のスレッドに切り替えて実行することで、スレッド間のインターリーブが発生します。また、デバイスの割り込みハンドラが、割り込み可能なコードと同じデータを変更する場合、割り込みのタイミングが悪ければ、データが破損する可能性があります。*コンカレンシー*とは、マルチプロセッサの並列処理、スレッドの切り替え、割り込みなどにより、複数の命令ストリームがインターリーブされる状況を指す。

カーネルには、同時にアクセスされるデータがたくさんあります。たとえば、2つのCPUが同時に`kalloc`を呼び出して、同時にフリーリストの先頭からデータを取り出すことができます。カーネルの設計者は、並列処理による性能の向上や応答性の向上のために、多くの同時処理を可能にしたいと考えています。しかし、その結果、カーネル設計者は、そのような同時実行にもかかわらず、正しさを自分自身に納得させるために多くの努力を費やします。正しいコードを得るためには様々な方法がありますが、中には推論しやすい方法もあります。並行性の下での正しさを目的とした戦略と、それをサポートする抽象化は、*並行性制御*技術と呼ばれます。Xv6
では、状況に応じていくつかの同時実行制御技術を使用します。この章では、広く使われている技術である*「ロック」*に焦点を当てます。

ロックには相互排除の機能があり、一度に1つのCPUしかロックを保持できないよう

になっています。プログラマーが各共有データアイテムにロックを関連付け、コードがアイテムを使用する際に常に関連付けられたロックを保持していれば、そのアイテムは一度に1つのCPUだけが使用することができます。このような状況では、ロックがデータアイテムを保護していると言います。

この章の残りの部分では、xv6がロックを必要とする理由、xv6がロックをどのように実装し、どのように使用するかを説明します。

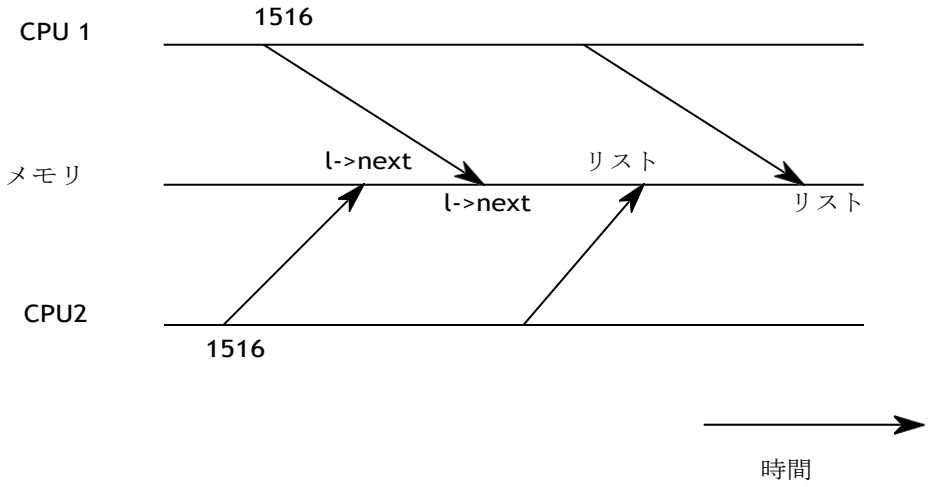


図 5.1: レースの例

5.1 レースコンディション

ロックが必要な理由の一例として、マルチプロセッサ上のどのCPUからでもアクセス可能なリンクリストを考えてみましょう。このリストはプッシュとポップの操作に対応しており、これらは同時に呼び出すことができます。kalloc() (kernel/kalloc.c:69) は空きページのリストからメモリのページをポップし、kfree() (kernel/kalloc.c:47) は空きページのリストにページをプッシュします。

同時リクエストがない場合、リストプッシュ操作を以下のように実装することができます。

```
1     struct element {.
2       intデータです。
3       struct element *next;
4     };
5
6     struct element *list = 0;
7
8     ボイド
9     push(int data)
10    {
11      struct element *l;
12
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
17    }
```

この実装は、分離して実行される場合は正しい。しかし、2つ以上のコピーが同時に

実行されると、このコードは正しくありません。2つのCPUが同時にpushを実行した場合、どちらかが16行目を実行する前に、両方が15行目を実行する可能性があります（図5.1参照）。このとき、listの前の値をnextに設定した2つのlist要素が存在することになります。listへの2つの代入が起こると

16行目では、2番目のものが1番目のものを上書きし、1番目の割り当てに関わる要素は失われます。

16行目の失われた更新は、*レースコンディションの一例*です。競合状態とは、あるメモリ位置が同時にアクセスされ、少なくとも1つのアクセスが書き込みである状態のことです。競合状態は多くの場合、バグの兆候であり、更新が失われた場合（アクセスが書き込みの場合）や、不完全に更新されたデータ構造を読み込んだ場合に起こります。レースの結果は、関係する2つのCPUの正確なタイミングや、メモリシステムによるメモリ操作の順序に左右されるため、レースによって引き起こされるエラーの再現やデバッグが困難になります。例えば、`push` のデバッグ中にprint文を追加すると、実行のタイミングが十分に変わり、レースが消滅することがあります。

レースを避けるためには、ロックを使うのが一般的です。ロックは*相互排除*を保証するもので、1つのCPUが一度に実行できるのは、プッシュの微妙な行であり、これでは上記のシナリオは不可能です。上記のコードを正しくロックしたバージョンでは、わずか数行が追加されています（黄色でハイライトされています）。

```
6     struct element *list = 0;
7     struct lock listlock;
8
9     ボイド
10    push(int data)
11    {
12      struct element *l;
13      l = malloc(sizeof *l);
14      l->data = data;
15
16      アクイジション(&listlock)です。
17      l->next = list;
18      list = l;
19      release(&listlock)となっています。
20    }
```

アクイジションからリリースまでの一連の命令を*クリティカルセクション*と呼ぶことがあります。ロックは通常、リストを保護していると言われています。

ロックがデータを保護すると言っても、実際には、ロックがデータに適用される不変量のコレクションを保護するということです。不変量とは、操作を行っても維持されるデータ構造の特性です。通常、操作の正しい動作は、操作の開始時に不変量が真であることに依存します。操作は一時的に不変量に違反することがありますが、終了する前に再び不変量を維持しなければなりません。例えば、リンクリストの場合、不変量は、`list`がリストの最初の要素を指し、各要素の`next`フィールドが次の要素を指すというものです

。17行目では`l`が次のリストの要素を指していますが、`list`はまだ`l`を指していません（18行目で再設定）。このように、一時的にリストの不変量に違反した状態で、第2のCPUがリストの不変量に依存したコードを実行したために、上述のような競合状態が発生したのです。ロックを適切に使用することで、一度に1つのCPUだけがクリティカルセクションのデータ構造を操作できるようになり、データ構造の不変量が成立していないときにデータ構造の操作を実行してしまうCPUがなくなるのです。

ロックとは、同時実行されるクリティカルセクションを1つずつ実行するように*直列化する*ことで、（クリティカルセクションが分離した状態で正しいと仮定して）不変性を維持することができると考えることができます。また、以下のことも可能です。

同じロックで保護されているクリティカルセクションは、互いにアトミックであると考えられます。これにより、各プロセスは以前のクリティカルセクションからの変更の完全なセットのみを見ることができ、部分的に完了した更新を見ることはありません。複数のプロセスが同時に同じロックを求めている場合、あるいはロックに*競合*が発生している場合を「*コンフリクト*」と呼びます。

なお、acquireをpushの前に移動させるのは正しいことです。例えば、acquireの呼び出しを13行目の前に移動しても構いません。この場合、malloc
の呼び出しも直列化されるため、並列性が低下する可能性があります。以下の「ロックの使用」のセクションでは、アクイジションとリリースの呼び出しをどこに挿入するかについてのガイドラインがあります。

5.2 コードロック

Xv6にはスピンロックとスリープロックという2種類のロックがあります。まず、スピンロックについて説明します。Xv6はスピンロックをスピンロック構造体（kernel/spinlock.h:2）として表現します。この構造体の重要なフィールドはlockedで、ロックが利用可能な場合はゼロ、ホールドされている場合は非ゼロのワードです。論理的には、xv6は次のようなコードを実行してロックを取得するはずです。

```
21    ボイド
22    acquire(struct spinlock *lk) // does not work!
23    {
24      for(;;) {
25        if(lk->locked == 0) {...
26          lk->locked = 1となります。
27          ブレークします。
28        }
29      }
30    }
```

残念ながら、この実装はマルチプロセッサ上での相互排除を保証するものではありません。2つのCPUが同時に25行目に到達し、lk-
>lockedが0であることを確認した後、26行目を実行して両方がロックを獲得するということが起こりえます。この時点で、2つの異なるCPUがロックを保持していることになり、相互排除の特性に反することになります。そこで、25行目と26行目を*アトミック*（不可分）なステップとして実行する方法が必要になります。

ロックは広く使用されているため、マルチコアプロセッサでは通常、25行目と26行目をアトミックに実装した命令が用意されています。RISC-
Vの場合、この命令は「amoswap r, a 」 で す 。amoswap は 、メモリアドレスaの値を読み出し、そのアドレスにレジスタrの内容を書き込み、読み出した値をrに入れます。この一連の動作はアトミックに行われ、読み出しと書き込みの間

に他のCPUがメモリアドレスを使用することを防ぐための特別なハードウェアを使用しています。

Xv6のアクイジション(kernel/spinlock.c:22)では、ポータブルCライブラリの`sync_lock_test_and_set`を使用していますが、これは`amoswap`命令に集約されます。`lk->locked`となります。`acquire`関数はスワップをループに巻き込み、ロックを獲得するまで再試行（スピン）します。反復するたびに`lk->locked`に1つずつスワップされ、前回の値を確認します。前回の値が0であればロックを獲得したことになり、スワップによって`lk->locked`が設定されます。

を1に変更しています。前の値が1であれば、他のCPUがロックを保持しており、lk->lockedに1をアトミックにスワップしても、その値は変わりませんでした。

ロックを取得すると、デバッグのために、ロックを取得したCPUを記録します。lk->cpuフィールドはロックによって保護されており、ロックを保持している間のみ変更することができます。

関数 release (kernel/spinlock.c:46) は acquire の逆で、lk->cpu フィールドをクリアしてからロックを解放します。概念的には、リリースは lk->locked にゼロを代入するだけです。C標準では、コンパイラは複数のストア命令で代入を実装することができますので C言語の代入は、同時実行コードに対してアトミックでない場合があります。代わりに、リリースはアトミックな割り当てを行うCライブラリ関数sync_lock_releaseを使用します。この関数は、RISC-Vのamoswap命令にも相当します。

5.3 コードロックの使用

Xv6では、競合状態を避けるために多くの場所でロックを使用しています。上記のプッシュとよく似た簡単な例を見るには、kalloc (kernel/kalloc.c:69) と free (kernel/kalloc.c:34) を見てください。練習問題1と2で、これらの関数がロックを省略するとどうなるかを試してみてください。誤った動作を引き起こすことは難しく，コードがロックエラーや競合から解放されているかどうかを確実にテストすることは困難であることがわかるでしょう．xv6にレースが存在する可能性は低くありません。

ロックを使用する際に難しいのは、いくつのロックを使用するか、また、それぞれのロックが保護すべきデータや不変量を決定することです。いくつかの基本原則があります。まず、あるCPUが変数を書き込み、同時に他のCPUがそれを読み書きできる場合は、ロックを使用して2つの操作が重ならないようにします。次に、ロックは不変量を保護するということを覚えておいてください。不変量が複数のメモリロケーションを含む場合、不変量を確実に維持するためには、通常、すべてのメモリロケーションを1つのロックで保護する必要があります。

上記のルールでは、ロックが必要な場合については述べられていますが、ロックが不必要な場合については何も述べられていません。また、ロックは並列性を低下させるため、ロックをかけすぎないことが効率化のために重要です。並列性が重要でなければ、1つのスレッドだけを持ち、ロックを気にしないようにすることができます。シンプルなカーネルであれば、マルチプロセッサ上でも、カーネルに入るときに取得し、カーネルを出るときに解放しなければならない単一のロックを持つことで、これを実現できます（ただし、パイプリードやウェイトなどのシステムコールは問題になります）。この方法は「ビッグカーネルロック」と呼ばれることもあり、多くのユニプロセッサOSがマルチプロセッサ上で動作するように変換されてきましたが、この方法は並列性を犠牲に

しています：一度にカーネル内で実行できるのは1つのCPUだけです。カーネルが重い計算を行う場合には、より細かいロックのセットを使用して、カーネルが複数のCPUで同時に実行できるようにした方が効率的です。

粗視化されたロックの例として，xv6 の `kalloc.c` アロケータでは，1つのロックで保護された 1つのフリーリストがあります．異なるCPU上の複数のプロセスが同時にページを割り当てようとすると、それぞれのプロセスはアクワイアで回転しながら順番を待つことになります。空回りは有用な作業ではないため、パフォーマンスが低下します。ロックの競合が CPU時間のかなりの部分を無駄にしている場合、アロケータの設計を変更して、それぞれが独自のロックを持つ複数のフリーリストを持つようにすれば、真の意味での並列割り当てが可能になり、パフォーマンスが向上するでしょう。

細かいロックの例として、xv6ではファイルごとに独立したロックを持っているので、異なるファイルを操作するプロセスでも、お互いのロックを待たずに進めることができます。ファイル

	ロックの説明
	bcache.lockブロックバッファキャッシュエントリの割り当てを保護します。
	cons.lockコンソール・ハードウェアへのアクセスをシリアル化し，出力の混在を回避
	ftable.lockファイルテーブル内の構造体ファイルの割り当てを シリアル化
	icache.lockinodeキャッシュエントリの割り当てを保護します。
	vdisk_lockディスクハードウェアへのアクセスとDMAディスクリプターのキューを シリアル化　kmem.lockメモリの割り当てを シリアル化
	log.lockトランザクションログパイプの pi->lock各パイプの操作を シリアライズ　する
	pid_locknext_pidの増分を シリアライズする
	プロックの p->lockプロセスの状態への変更をシリアライズ
	tickslockティックカウンタへの操作を シリアライズ
	inodeの ip->lockSerializesは、各inodeとそのコンテンツに対する操作を行います。 bufの b->lockSerializesは、各ブロックバッファに対する操作を行います。

図 5.2: xv6 のロック

プロセスが同じファイルの異なる領域を同時に書き込めるようにするには、ロックスキームをさらに細かく設定する必要があります。最終的にロックの粒度を決定するには、複雑さを考慮するだけでなく、パフォーマンスを測定する必要があります。

以降の章では、xv6の各部分を説明する際に、xv6がロックを使用して並行性に対処している例について言及します。予告として、図5.2にxv6のすべてのロックの一覧を示します。

5.4 デッドロックとロックオーダー

カーネルを経由するコードパスが同時に複数のロックを保持しなければならない場合、

すべてのコードパスが同じ順序でロックを取得することが重要です。そうしないと、*デッドロック*の危険性があります。例えば、xv6の2つのコードパスがロックAとBを必要としているが、コードパス1はA→Bの順でロックを取得し、もう一方のパスはB→Aの順でロックを取得するとします。スレッドT1がコードパス1を実行してロックAを取得し、スレッドT2がコードパス2を実行してロックBを取得したとします。このようなデッドロックを回避するには、すべてのコードパスが同じ順序でロックを取得する必要があります。グローバルなロック取得順序の必要性は、ロックが各関数の仕様の一部であることを意味します。

Xv6では、`sleep`の動作方法（第6章参照）により、プロセス単位のロック（各`struct proc`のロック）を含む長さ2のロック順序連鎖が多数存在します。例えば、`consoleintr` (kernel/console.c:143)は、入力された文字を処理する割り込みルーチンです。改行が発生すると、コンソール入力を待っているプロセスはすべて起こされます。これを行うために、`consoleintr`は`cons.lock`を保持しながら`wakeup`を呼び出し、wakeupは待ち受けているプロセスのロックを取得して、待ち受けているプロセスを起こします。そのため、グローバルなデッドロック回避のためのロック順序には、`cons.lock`はどのプロセスのロックよりも先に取得しなければならないというルールが含まれています。ファイルシステムのコードにはxv6で最も長いロックチェーンが含まれています。

例えば、ファイルを作成する際には、ディレクトリのロック、新規ファイルのinodeのロック、ディスクのブロックバッファのロック、ディスクドライバのvdisk_lock、呼び出し元のプロセスのp->lockを同時に保持する必要があります。デッドロックを避けるため、ファイルシステムのコードは常に前の文で述べた順序でロックを取得します。

グローバルなデッドロック回避順序を守ることは、意外と難しいものです。例えば、コードモジュールM1がモジュールM2を呼び出すが、ロック順序ではM2のロックをM1のロックよりも先に取得しなければならないといったように、ロック順序が論理的なプログラム構造と矛盾する場合があります。ロックのアイデンティティが事前にわからない場合もあります。これは、次に取得するロックのアイデンティティを知るために、あるロックを保持する必要があるためです。このような状況は、ファイルシステムがパス名に含まれる連続したコンポーネントを検索したり、waitやexitのコードがプロセスのテーブルを検索して子プロセスを探したりする際に発生します。最後に、デッドロックの危険性は、ロックの数が多ければ多いほどデッドロックの機会が増えることになるため、ロックの仕組みをどれだけ細かくできるかという制約になることがよくあります。デッドロックを回避する必要性は、しばしばカーネルの実装における主要な要因となります。

5.5 ロックと割込みハンドラ

xv6のスピンロックの中には、スレッドと割込みハンドラの両方で使用されるデータを保護するものがあります。例えば、clockintr
タイマー割り込みハンドラは、カーネルスレッドがsys_sleep (kernel/sysproc.c:64)のtickを読み取るのとほぼ同時にtickをインクリメントするかもしれません(kernel/trap.c:163)。ロックtickslockはこの2つのアクセスをシリアライズします。

スピンロックと割り込みの相互作用には潜在的な危険があります。例えば、sys_sleepがtickslockを保持していて、そのCPUがタイマ割込みによって中断されたとします。clockintrはtickslockを獲得しようとし、それが保持されていることを確認し、解放されるのを待つでしょう。この状況では、tickslockは決して解放されません：sys_sleepだけがそれを解放することができますが、sys_sleepはclockintrが戻るまで実行を続けません。そのため、CPUはデッドロックし、どちらかのロックを必要とするコードもフリーズしてしまいます。

このような状況を避けるためには、スピンロックが割り込みハンドラで使用されている場合、CPUは割り込みを有効にした状態でそのロックを保持してはいけません。Xv6はより保守的で、CPUが何らかのロックを取得すると、xv6は常にそのCPUの割り込みを無効にします。割り込みは他のCPUでも発生するので、割り込みの取得はスレッド

がスピンロックを解放するのを待つことができますが、同じCPUではできません。

xv6 は CPU がスピンロックを持っていないときに割り込みを再び有効にしますが、入れ子になったクリティカ ルセクションに対応するために、少し帳簿をつける必要があります。acquire は push_off (kernel/spinlock.c:87) を、release は pop_off (kernel/spinlock.c:98) を呼び出して、現在の CPU のロックの入れ子のレベルを追跡します。pop_off は 、そのカウントがゼロになると、一番外側のクリティカルセクションの開始時に存在していた割り込みイネーブルの状態を復元します。intr_offとintr_on関数は、それぞれ割り込みを無効にするRISC-V命令と割り込みを有効にするRISC-V命令を実行します。

アクイジションでは、lk->locked を設定する前に push_off を厳密に呼び出すことが重要です (kernel/spinlock.c:28)。もしこの2つが逆だったら、割り込みが有効な状態でロックが保持されている短い時間があり、不運なタイミングで割り込みが入るとシステムがデッドロックしてしまいます。同様に、リリースが pop_off を呼び出すのは、ロックを解放した後であることが重要です (kernel/spinlock.c:63)。

5.6 命令とメモリの順序

プログラムは、ソースコードの記述順に実行されると考えるのが自然です。しかし、コンパイラやCPUの中には、より高いパフォーマンスを得るために、コードを順不同に実行するものが少なくありません。ある命令が完了するのに何サイクルもかかる場合、CPUはその命令を早めに発行し、他の命令とオーバーラップさせてCPUのストールを回避することがあります。例えば、一連の命令の中で、AとBが互いに依存していないことにCPUが気付いたとします。コンパイラも同様に、ある文の命令を、その前にある文の命令よりも先に発行することで、順序を入れ替えることができます。

コンパイラやCPUは、正しく書かれた直列コードの結果を変えないように、再順序付けを行う際にルールに従っています。しかし、このルールでは、同時実行コードの結果を変更するような再順序付けが可能であり、マルチプロセッサ上では誤った動作を引き起こしやすくなります[2, 3]。CPUの順序規則は、*メモリモデル*と呼ばれています。

例えば、このpushのコードでは、コンパイラやCPUが4行目に対応するストアを6行目のリリースの後に移動させてしまうと大変なことになってしまいます。

```
1        l = malloc(sizeof *l);
2        l->data = data;
3        アクイジション(&listlock)です。
4        l->next = list;
5        list = l;
6        release(&listlock)となっています。
```

このような並び替えが発生した場合、他のCPUがロックを取得して更新されたリストを観測しても、初期化されていないlist->nextが表示されるウィンドウが発生します。

ハードウェアやコンパイラにこのような再順序付けを行わないように指示するために、xv6ではsync_synchronize()アクイジション（kernel/spinlock.c:22）とリリース（kernel/spinlock.c:46）の両方でsync_synchronize()は*メモリバリア*です。バリアーを越えてロードやストアの順番を変えないようにコンパイラやCPUに指示します。xv6では共有データへのアクセスにロックを使用しているため、xv6のアクイジションとリリースのバリアは、問題となるほとんどすべてのケースで順序を強制的に変更しています。第8章では、いくつかの例外について説明します。

5.7 スリープロック

xv6では、ロックを長時間保持する必要がある場合があります。例えば、ファイルシステム（第7章）では、ディスク上でファイルの内容を読み書きしている間、ファイルをロックしたままにしており、これらのディスク操作には数十ミリ秒かかることがあります。このようにスピンロックを長く保持すると、他のプロセスがスピンロックを取得しようとした場合、取得プロセスがスピンロック中に長時間CPUを浪費することになり、

無駄が生じます。もう一つの欠点は、スピンロックを保持したままCPUを降伏することができないことです。これは、ロックをかけたプロセスがディスクを待っている間に、他のプロセスがCPUを使用できるようにするためです。スピンロックを保持したまま降伏することは、2番目のスレッドがスピンロックを獲得しようとした場合にデッドロックを引き起こす可能性があるため、違法となります。また、ロックを保持したまま降伏することは、スピンロックを保持している間は割り込みをオフにしなければならないという要件にも違反します。したがって、次のようになります。

のように、獲得を待っている間はCPUを降伏させ、ロックを保持している間は降伏（と割り込み）を許可するタイプのロックです。

Xv6では、このようなロックを*sleep-lock*という形で提供しています。`acquiresleep`（kernel/sleeplock.c:22）は、第6章で説明する技術を用いて、待機中にCPUを解放します。高いレベルでは、スリープロックにはスピンロックで保護されたロックされたフィールドがあり、`acquiresleep`の`sleep`への呼び出しはアトミックにCPUを解放し、スピンロックを解除します。その結果、`acquiresleep`が待機している間に他のスレッドが実行できるようになります。

スリープロックは割り込みを有効にしたままなので、割り込みハンドラでは使用できません。また、スリープロックはスピンロッククリティカルセクション内では使用できません（スピンロックはスリープロッククリティカルセクション内で使用可能）。

スピンロックは短いクリティカルセクションに最適で、それを待つことはCPU時間を浪費するため、スリープロックは長い操作に適しています。

5.8 リアルワールド

ロックを使用したプログラミングは、並行処理プリミティブや並列処理に関する長年の研究にもかかわらず、依然として困難です。多くの場合、ロックは同期キューのような高レベルの構造の中に隠すのがベストですが、xv6ではそのようにはなっていません。ロックを使用してプログラミングする場合は、レースコンディションを特定するツールを使用することをお勧めします。

ほとんどのOSはPOSIXスレッド（Pthread）をサポートしており、ユーザープロセスが複数のスレッドを異なるCPU上で同時に実行できるようになっています。Pthreadには、ユーザーレベルのロックやバリアなどの機能があります。Pthreadをサポートするには、オペレーティング・システムのサポートが必要です。たとえば、あるpthreadがシステムコールでブロックした場合、同じプロセスの別のpthreadがそのCPUで実行できるようにする必要があります。別の例として、pthreadがプロセスのアドレス空間を変更した場合（例：メモリのマップやアンマップ）、カーネルは同じプロセスのスレッドを実行する他のCPUがハードウェアページテーブルを更新してアドレス空間の変更を反映するように手配しなければなりません。

アトミック命令を使わずにロックを実装することも可能ですが、コストがかかりますし、ほとんどのOSがアトミック命令を使用しています。

ロックは、多くのCPUが同時に同じロックを取得しようとするとコストがかかります。あるCPUがローカルキャッシュにロックをキャッシュしていて、他のCPUがそのロックを取得しなければならない場合、ロックを保持しているキャッシュラインを更新す

るアトミック命令は、あるCPUのキャッシュから他のCPUのキャッシュにそのラインを移動させなければならず、おそらくそのキャッシュラインの他のコピーを無効にしなければなりません。他のCPUのキャッシュからキャッシュラインをフェッチすることは、ローカルキャッシュからラインをフェッチすることに比べて何桁もコストがかかります。

ロックにかかる費用を回避するために、多くのオペレーティングシステムではロックフリーのデータ構造やアルゴリズムが採用されています。例えば、本章冒頭のリンクリストのように、リスト検索時にロックを必要とせず、リストにアイテムを挿入するためのアトミックな命令を1つだけ実装することも可能です。しかし、ロックフリーのプログラミングは、ロックを使ったプログラミングよりも複雑で、例えば、命令やメモリの並び替えを気にしなければなりません。ロックを使ったプログラミングはすでに難しいので、xv6ではロックフリープログラミングの複雑さを増すことを避けています。

5.9 エクササイズ

1. kalloc（kernel/kalloc.c:69）のアクイジションとリリースの呼び出しをコメントアウトします。これは、kallocを呼び出すカーネルコードに問題を起こすように思えますが、どのような症状が出ると予想されますか？xv6を実行すると、このような症状が現れますか?usertestsを実行したときはどうでしょうか?もし問題が見られなければ、なぜでしょう?kallocのクリティカルセクションにダミーのループを挿入することで、問題を引き起こすことができるかどうか見てみましょう。

2. 仮に、kfreeのロックをコメントアウトしたとしましょう(kallocのロックを復元した後に)。何が間違っているでしょうか?kfreeでのロックの欠如はkallocでのロックよりも害が少ないでしょうか?

3. 2つのCPUが同時にkallocを呼び出すと，片方がもう片方を待たなければならず，パフォーマンスが低下してしまいます．kalloc.c を改良して並列性を高め、異なるCPUから同時にkallocを呼び出しても、お互いに待つことなく処理を進められるようにします。

4. ほとんどのOSでサポートされているPOSIXスレッドを使って、並列プログラムを書いてみましょう。例えば，並列ハッシュテーブルを実装し，コア数の増加に伴ってput/getの回数が増加するかどうかを測定する．

5. xv6 で Pthreads のサブセットを実装する。つまり、ユーザーレベルのスレッド・ライブラリを実装して、ユーザー・プロセスが複数のスレッドを持ち、これらのスレッドが異なる CPU 上で並行して実行できるようにします。ブロッキングシステムコールを行うスレッドとその共有アドレス空間を変更するスレッドを正しく処理するデザインを作成します。

第6章

スケジューリング

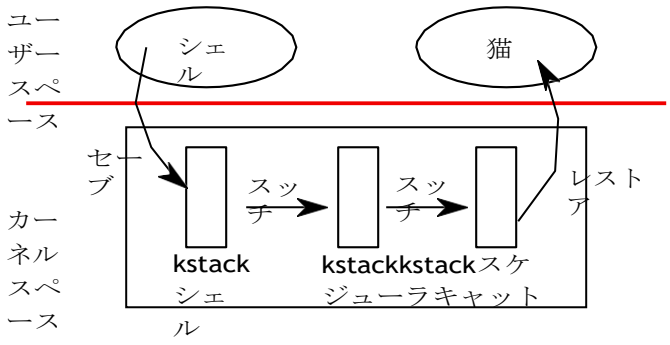
オペレーティングシステムでは、コンピュータのCPU数よりも多くのプロセスが実行される可能性があるため、プロセス間でCPUを時間的に共有する計画が必要です。理想的には、ユーザープロセスからはこの共有が透過的であることが望ましい。一般的なアプローチは、プロセスをハードウェアCPUに*多重化する*ことで、各プロセスに仮想的なCPUがあるかのような錯覚を与えることです。本章では、xv6がどのようにしてこの多重化を実現しているかを説明します。

6.1 マルチプレックス

Xv6では、2つの状況で各CPUをあるプロセスから別のプロセスに切り替えることで多重化しています。まず、xv6のスリープ・ウェイクアップ機構は、プロセスがデバイスやパイプのI/Oが完了するのを待ったり、子プロセスが終了するのを待ったり、スリープシステムコールで待ったりするときに切り替えます。第二に、xv6は定期的にスイッチを強制的に切り替えて、長期間スリープせずに計算を行うプロセスに対処します。この多重化は、xv6がメモリアロケータとハードウェアページテーブルを使って各プロセスが独自のメモリを持っているように錯覚させるのと同様に、各プロセスが独自のCPUを持っているように見せかけます。

多重化を実現するには、いくつかの課題があります。まず、あるプロセスから別のプロセスへの切り替え方法です。コンテキストスイッチのアイデアは単純ですが、その

実装はxv6の中でも最も不透明なコードの一つです。第二に、ユーザー・プロセスに透過的な方法でスイッチを強制するにはどうすればよいか？Xv6では、タイマー割り込みでコンテキストスイッチを駆動するという標準的な手法を採用しています。第3に、多くのCPUが同時にプロセスを切り替える可能性があり、レースを避けるためにロックプランが必要です。第4に、プロセスが終了する際には、プロセスのメモリやその他のリソースを解放しなければなりませんが、例えば、カーネルスタックを使用したまま解放することはできませんので、すべてを自分で行うことはできません。第5に、マルチコア・マシンの各コアは、システム・コールが正しいプロセスのカーネル状態に影響を与えるように、どのプロセスを実行しているかを覚えておく必要があります。最後に、スリープとウェイクアップは、あるプロセスがCPUを放棄してイベントを待つためにスリープしたり、別のプロセスが最初のプロセスを起こしたりすることを可能にします。wakeupの通知が失われるようなレースを避けるためには注意が必要です。Xv6では、これらの問題をできるだけシンプルに解決しようとしていますが、それでも結果的にはトリッキーなコードになっています。



カーネル

図6.1:あるユーザープロセスから別のユーザープロセスへの切り替え。この例では、xv6は1つのCPU（したがって1つのスケジューラースレッド）で動作します。

6.2 コードコンテキスト・スイッチング

図6.1は、あるユーザー・プロセスから別のユーザー・プロセスに切り替わる際の手順をまとめたものです。旧プロセスのカーネル・スレッドへのユーザー・カーネル移行（システム・コールまたは割り込み）、現在のCPUのスケジューラ・スレッドへのコンテキスト・スイッチ、新プロセスのカーネル・スレッドへのコンテキスト・スイッチ、そしてユーザー・レベル・プロセスへのトラップ・リターンです。xv6のスケジューラは、CPUごとに専用のスレッド（保存されたレジスタとスタック）を持っています。これは、スケジューラが古いプロセスのカーネル・スタック上で実行するのは安全ではないからです。他のコアがプロセスを起こして実行する可能性があり、2つの異なるコアで同じスタックを使用するのは災害になるからです。このセクションでは、カーネル・スレッドとスケジューラ・スレッドの切り替えの仕組みを検討します。

あるスレッドから別のスレッドに切り替える際には、古いスレッドのCPUレジスタを保存し、新しいスレッドの以前に保存されていたレジスタを復元します。スタックポインタとプログラムカウンタが保存・復元されるということは、CPUがスタックを切り替え、どのコードを実行しているかを切り替えることになります。

関数 `swtch` は、カーネルのスレッド切り替えのための保存と復元を行います。`swtch` はスレッドについては直接知らず、*コンテキスト*と呼ばれるレジスタセットの保存と復元を行います。プロセスがCPUを手放すときには、プロセスのカーネルスレッドが`swtch`を呼び出して自分のコンテキストを保存し、スケジューラのコンテキストに戻ります。各コンテキストは `struct context` (kernel/proc.h:2) に含まれており、それ自体はプロセスの `struct proc` や CPU の `struct cpu` に含まれています。`Swtch`は2つの引数を取ります: `struct context *old`と`struct context *new`です。現在のレジスタを`old`に保存し、`new`からレジスタをロードして返します。

`swtch`からschedulerまでのプロセスを追ってみましょう。第4章では、割り込みの終了時の可能性として、`usertrap`が`yield`を呼び出すことを見ました。`yield`は`sched`を呼び、`swt`

chはp->contextに現在のコンテキストを保存し、cpu->schedulerに保存されているスケジューラのコンテキストに切り替えます(kernel/proc.c:494)。

Swtch (kernel/swtch.S:3)は呼び出し側で保存されたレジスタのみを保存します。呼び出し側で保存されたレジスタは、呼び出したCコードによって（必要に応じて）スタックに保存されます。Swtchは、構造体コンテキスト内の各レジスタのフィールドのオフセットを知っています。swtchはプログラムカウンタを保存しません。その代わりに、swtchが呼び出されたときのリターンアドレスを保持するraレジスタを保存します。さて、swtchはレジスタを

は，以前の swtch によって保存されたレジスタの値を保持する新しいコンテキストである．swtchが戻るときには，復元されたraレジスタが指す命令，つまり，新しいスレッドが以前にswtchを呼び出した命令に戻ります。さらに、新しいスレッドのスタックにも戻ります。

この例では、sched が swtch を呼び出して、CPUごとのスケジューラコンテキストである cpu->scheduler に切り替えました。このコンテキストは、スケジューラのswtchへの呼び出しによって保存されていました（kernel/proc.c:460）。これまで追跡してきた swtch が戻ってきたとき、それは sched ではなく、scheduler に戻り、そのスタックポインタは現在の CPU のスケジューラスタックを指します。

6.3 コードスケジューリング

前節ではswtchの低レベルの詳細について見てきましたが、ここではswtchを前提として、あるプロセスのカーネルスレッドからスケジューラを経由して別のプロセスに切り替えることについて見ていきましょう。スケジューラは、CPUごとに専用のスレッドが存在し、それぞれがスケジューラ機能を実行しています。この機能は、次に実行するプロセスを選択する役割を担っています。CPUを手放したいプロセスは、自分のプロセスロックp->lockを取得し、保持している他のロックを解放し、自分の状態（p->state）を更新してから、sched.Yieldを呼び出す必要があります。Yield (kernel/proc.c:500) は、後述する sleep や exit と同様、この規則に従っています。schedはこれらの条件をダブルチェックし(kernel/proc.c:484-489)、さらにこれらの条件の暗黙の了解である、ロックが保持されているので、割り込みは無効にすべきだとしています。最後に sched は swtch を呼び、現在のコンテキストを p->context に保存し、cpu->scheduler のスケジューラコンテキストに切り替えます。swtchは、スケジューラのswtchが戻ってきたかのように、スケジューラのスタック上に戻ります（kernel/proc.c:460）。スケジューラはforループを続け、実行するプロセスを見つけ、それに切り替え、このサイクルを繰り返します。

xv6がswtchの呼び出しに関わらずp->lockを保持していることを確認しました。swtchの呼び出し元が既にロックを保持している必要があり、ロックの制御は切り替え先のコードに移ります。通常、ロックを取得したスレッドはロックの解放にも責任を負うので、正しさについての推論が容易になります。p->lock はプロセスの状態やコンテキストフィールドに関する不変量を保護しますが、swtch での実行時には真ではないため、コンテキストスイッチではこの慣習を破る必要があります。swtch中にp->lockが保持されていない場合に起こりうる問題の例として、yieldがプロセスの状態をRUNNABLEに設定した後、swtchがプロセスに自身のカーネルスタックの使用を停止させる前に、別のCPUがプロセスの実行を

決定する可能性があります。その結果、2つのCPUが同じスタック上で動作することになりますが、これは正しくありません。

カーネルスレッドは常にschedでCPUを放棄し、常にスケジューラの同じ場所に切り替わります。スケジューラは（ほぼ）常に以前にschedを呼び出したカーネルスレッドに切り替わります。したがって、xv6がスレッドを切り替える行番号を表示すると、次のような単純なパターンが表示されます。(kernel/proc.c:460), (kernel/proc.c:494), (kernel/proc.c:460), (ker-nel/proc.c:494) といった具合です。このような2つのスレッドの切り替えを行う手続きを*コルーチン*と呼ぶことがあります。この例では、schedとschedulerがお互いのコ・ルーティンになっています。

スケジューラのswtchへの呼び出しがschedで終わらない場合があります。新しいプロセスが最初にスケジューリングされるとき、forkretで開始されます(kernel/proc.c:512)。Forkretはp->lockを解放するために存在します。そうでなければ、新しいプロセスはusertrapretで開始される可能性があります。

スケジューラ(kernel/proc.c:442)は、実行するプロセスを見つけ、それが降参するまで実行するという単純なループを実行します。

を繰り返します。スケジューラは、プロセステーブルをループして、p->state == RUNNABLEとなっている実行可能なプロセスを探します。プロセスが見つかると、CPUごとの現在のプロセス変数c->procを設定し、そのプロセスをRUNNINGとマークして、swtchを呼び出して実行を開始します（kernel/proc.c:455-460）。

スケジューリングコードの構造を考える一つの方法は、各プロセスに対して一連の不変量を適用し、それらの不変量が真でないときは常にp->lockを保持するというものです。つまり、CPUのレジスタはプロセスのレジスタ値を保持していなければならず（つまりswtchはレジスタ値をコンテキストに移動していない）、c->procはプロセスを参照していなければなりません。もうひとつの不変性は、プロセスがRUNNABLEである場合、アイドル状態のCPUのスケジューラがそのプロセスを実行しても安全でなければならないということです。これは、p->contextにプロセスのレジスタが保持されていること(つまり、実際のレジスタには含まれていない)、プロセスのカーネルスタック上でCPUが実行されていないこと、どのCPUのc->procもプロセスを参照していないことを意味します。これらの特性は、p->lockが保持されている間は真でないことが多いことに注意してください。

上記の不変性を維持することが、xv6があるスレッドでp->lockを取得し、別のスレッドでそれを解放することが多い理由であり、例えばyieldで取得してschedulerで解放することがあります。yieldが実行中のプロセスの状態を変更してRUNNABLEにし始めたら、ロックは不変性が回復するまで保持されなければなりません。最も早い正しいリリースポイントは、（自身のスタック上で実行している）スケジューラがc->procをクリアした後です。同様に、スケジューラがRUNNABLEプロセスをRUNNINGに変換し始めると、カーネル・スレッドが完全に実行されるまで（例えばyieldの場合はswtchの後）ロックを解除することはできません。

p->lockは他にも保護しているものがあります。exitとwaitの相互作用、ウェイクアップの失敗を防ぐ仕組み（セクション6.5参照）、プロセスの終了と他のプロセスがその状態を読み書きすることによる競合の回避（exitシステムコールがp->pidを見てp->killedを設定することなど（kernel/proc.c:596））などです。p->lockのさまざまな機能を分割して、わかりやすく、そしておそらくパフォーマンスを向上させることができるかどうかを検討する価値があるかもしれません。

6.4 コード：mycpuとmyproc

Xv6はしばしば、現在のプロセスのproc構造へのポインタを必要とします。ユニプロセッサでは、現在のprocを指すグローバル変数を持つことができます。しかし、マルチコア・マシンでは、各コアが異なるプロセスを実行するため、これはうまくいきません。この

問題を解決する方法は、各コアが独自のレジスタセットを持っているという事実を利用することです。これらのレジスタの1つを利用して、コアごとの情報を見つけることができます。

Xv6 では、各 CPU の `struct cpu`（kernel/proc.h:22）を管理しています。この struct cpu には、現在その CPU で実行されているプロセス（存在する場合）、CPU のスケジューラースレッドの保存されたレジスタ、割り込み禁止の管理に必要な入れ子のスピンロックの数が記録されています。関数 `mycpu` (kernel/proc.c:58) は、現在の CPU の `struct cpu` へのポインタを返します。RISC-V は CPU に番号を付け、それぞれに *hartid* を与えます。Xv6 では、カーネル内で各 CPU の hartid がその CPU の `tp` レジスタに保存されるようになっています。これにより `mycpu` は `tp` を使用して `cpu` 構造体の配列をインデックスし、正しい構造体を見つけることができます。

CPU の `tp` が常に CPU の hartid を保持するようにするには、少し複雑な作業が必要です。`mstart` は、CPU のブートシーケンスの早い段階で、まだマシンモードの間に `tp` レジスタを設定します (kernel/start.c:45)。`usertrapret` は、ユーザプロセスが `tp` を変更する可能性があるため、`tp` をトランポリンページに保存します。最後に

uservec は 、ユーザー空間からカーネルに入るときに、保存されたtp を 復 元 します（kernel/trampoline.S:70）。コンパイラはtpレジスタを決して使用しないことを保証します。RISC-
Vがxv6に現在のhartidを直接読ませることができればもっと便利ですが、これはマシンモードでのみ許可されており、スーパバイザモードでは許可されていません。

cpuid と mycpu の戻り値は壊れやすくなっています。タイマーが割り込み、スレッドが降伏した後に別のCPU
に移動した場合、以前に返された値はもはや正しくありません。この問題を回避するために，xv6 では呼び出し側が割り込みを無効にし，返された struct cpu の使用を終えてから割り込みを有効にすることを要求しています.

関数 myproc (kernel/proc.c:66) は、現在の CPU で実行されているプロセスの struct proc ポインタを返します。myproc は、割り込みを無効にして mycpu を起動し、 struct cpu から現在のプロセスポインタ (c->proc) をフェッチした後、割り込みを有効にします。myproc
の戻り値は、割り込みが有効になっていても安全に使用できます。タイマー割り込みによって呼び出したプロセスが別の CPU に移動しても、その struct proc ポインタはそのままです。

6.5 睡眠と覚醒

スケジューリングやロックは、あるプロセスの存在を他のプロセスから隠すのに役立ちますが、これまでのところ、プロセスが意図的に相互作用するのを助ける抽象化はありません。この問題を解決するために多くのメカニズムが発明されてきました。Xv6ではsleep and wakeupと呼ばれるものを使用しています。これは、あるプロセスがイベントを待つためにスリープし、イベントが発生したら別のプロセスがそれを起こすというものです。スリープとウェイクアップは、しばしば*シーケンスコーディネーションや条件付き同期*メカニズムと呼ばれます。

ここでは、プロデューサーとコンシューマーを協調させる*セマフォ*[4]と呼ばれる同期メカニズムについて説明します。セマフォはカウントを保持し、2つの操作を提供します。V "オペレーション（プロデューサー用）はカウントをインクリメントします。P "操作（コンシューマ用）は、カウントがゼロでなくなるまで待ち、その後カウントをデクリメントして戻ります。プロデューサースレッドとコンシューマースレッドが1つずつあり、それらが異なるCPU上で実行され、コンパイラがあまり積極的に最適化しなければ、この実装は正しいでしょう。

```
100   構造体セマフォ {
101     struct spinlock lock;
102     int count;
```

```
103     };
104
105     ボイド
106     V(セマフォ構造体 *s)
107     {
108        acquire(&s->lock)しています。
109        s->count += 1;
110        release(&s->lock)となります。
111     }
112
113     ボイド
114     P(struct semaphore *s)
115     {
```

```
116       while(s->count == 0)
117         ;
118       acquire(&s->lock)しています。
119       s->count -= 1;
120       release(&s->lock)となります。
121   }
```

上記の実装はコストがかかります。プロデューサがめったに動作しない場合、コンシューマは `while` ループの中でゼロではないカウントを期待してほとんどの時間を費やすことになります。コンシューマのCPUは、`s->count`を繰り返し*ポーリングする*ことで、*ビジーウェイティング*よりも生産性の高い仕事を見つけることができます。ビジーウェイトを回避するには、コンシューマがCPUを降伏させ、Vがカウントを増加させた後にのみ再開する方法が必要です。

ここでは、そのための一歩を踏み出してみましたが、それだけでは不十分です。次のように動作する`sleep`と`wakeup`という一組のコールを想像してみましょう。`sleep(chan)`は、*待機チャネル*と呼ばれる任意の値`chan`をスリープさせます。`sleep`は呼び出したプロセスをスリープさせ、CPUを他の作業のために解放します。`Wakeup(chan)`は、`chan`でスリープしているすべてのプロセス（もしあれば）を起こし、そのスリープコールを返させます。`chan`上で待っているプロセスがない場合、`wakeup`は何もしません。セマフォの実装を変更して、`sleep`と`wakeup`を使うようにすることができます（黄色でハイライトした変更点）。

```
200   ボイド
201   V(セマフォ構造体 *s)
202       {
203       acquire(&s->lock)しています。
204       s->count += 1;
205       wakeup(s)です。
206       release(&s->lock)となります。
207   }
208
209   ボイド
210   P(struct semaphore *s)
211   {
212     while(s->count == 0)
213       睡眠
214     acquire(&s->lock)しています。
215     s->count -= 1;
216     release(&s->lock)となります。
217   }
```

`P`が回転する代わりにCPUを放棄するようになったのは良いことです。しかし、このインターフェイスを使って`sleep`と`wakeup`を設計することは、*ロストウェイクアップ*問題と呼ばれる問題に悩ま

されることなく、簡単ではないことがわかりました。P が 212行目でs->count == 0を見つけたとします。Pが212行目と213行目の間にいる間に、Vが別のCPUで実行され、s->count を0以外の値に変更してwakeupを呼び出しましたが、wakeupはプロセスが眠っていないことを発見したため、何もしませんでした。P は213行目で実行を続け、sleepを呼び出してスリープ状態になります。これは問題を引き起こします。Pは、すでに起こったV の呼び出しを待って眠っているのです。幸運にもプロデューサーが再びVを呼び出さない限り、カウントがゼロではないにもかかわらず、コンシューマは永遠に待つことになります。

この問題の根源は、「P は s->count == 0 のときだけスリープする」という不変量が、ちょうど悪いタイミングでVが実行されることで破られてしまうことにあります。この不変性を守るための間違った方法は、Pのロック取得（下の黄色でハイライトされている部分）を移動させ、カウントのチェックとsleepの呼び出しがアトミックになるようにすることです。

```
300     ボイド
301     V(セマフォ構造体 *s)
302     {
303       acquire(&s->lock)しています。
304       s->count += 1;
305       wakeup(s)です。
306       release(&s->lock)となります。
307     }
308
309     ボイド
310     P(struct semaphore *s)
311     {
312       acquire(&s->lock)しています。
313       while(s->count == 0)
314         睡眠
315       s->count -= 1;
316       release(&s->lock)となります。
317        }
```

このバージョンのPは、313行目と314行目の間でVが実行されるのをロックが阻止するので、ロストウェイクアップを回避できると期待できるかもしれません。しかし、それだけではなく、デッドロックが発生します。Pはスリープ中にロックを保持しているので、Vはロックを待って永遠にブロックしてしまいます。

呼び出し側はsleepに*コンディションロック*を渡さなければならず、呼び出し側のプロセスがスリープとしてマークされ、スリープチャネルで待機した後にロックを解放できるようにする。このロックにより、同時実行中のVはPが自分をスリープさせ終わるまで待つことになり、wakeupがスリープ中のコンシューマを見つけて起こすことができます。コンシューマーが再び目覚めたら、sleepはロックを再取得してから戻ります。私たちの新しい正しいスリープ/ウェイクアップ方式は、以下のように使用できます（黄色で強調された変更点）。

```
400     ボイド
401     V(セマフォ構造体 *s)
402        {
403       acquire(&s->lock);
404       s->count += 1;
405       wakeup(s)です。
406       release(&s->lock)です。
407     }
408
```

```
409   void
410   P(struct semaphore *s)
411      {
412     acquire(&s->lock)です。
```

```
413       while(s->count == 0)
414          sleep(s, &s->lock)
          を行います。
415       s->count -= 1;
416       release(&s->lock)となり
          ます。
417     }
```

Pがs->lockを保持していることで、Pがc->countを確認してからsleepを呼び出すまでの間にVがs->lockを起こそうとすることを防ぐことができます。しかし、s->lockをアトミックに解放し、消費プロセスをスリープさせるためにはsleepが必要であることに注意してください。

6.6 コードスリープ&ウェイクアップ

sleep (kernel/proc.c:533) と wakeup (kernel/proc.c:567) の実装を見てみましょう。基本的なアイデアは、sleep は現在のプロセスを SLEEPING とマークし、その後 sched を呼び出して CPU を再リースします。wakeup は与えられた待機チャネルでスリープしているプロセスを探し、それを RUNNABLE とマークします。sleepとwakeupの呼び出し側は、お互いに都合の良い数字をチャンネルとして使用することができます。Xv6では、待機中のカーネルデータ構造のアドレスがよく使われます。

sleepはp->lockを取得します(kernel/proc.c:544)。これで、スリープに移行するプロセスはp->lockとlkの両方を保持することになります。lkの保持は呼び出し元（この例ではP）に必要でした。他のプロセス（この例ではVを実行しているプロセス）がwakeup(chan)の呼び出しを開始できないようにするためです。他のプロセスがwakeup(chan)の呼び出しを開始するかもしれませんが、wakeupはp->lockの取得を待つため、sleepがプロセスをスリープさせ終わるまで待ち、wakeupがsleepを逃さないようにします。

ちょっと複雑なことがあります。LKがp->lockと同じロックである場合、sleepがp->lockを取得しようとすると自分自身でデッドロックしてしまいます。しかし、sleepを呼び出したプロセスがすでにp->lockを保持している場合は、同時起動のウェイクアップを逃さないためにそれ以上何もする必要はありません。このケースは、wait (kernel/proc.c:567)がp->lockを持ってsleepを呼び出すときに発生します。

sleepはp->lockを保持しており、他には保持していないので、sleepチャネルを記録し、プロセスの状態をSLEEPINGに変更し、schedを呼び出すことでプロセスをスリープさせることができます（kernel/proc.c:549-552）。プロセスがSLEEPINGと判定されるまでp->lockが（スケジューラによって）解放されないことがなぜ重要なのかは、すぐに明らかにな

るでしょう。

ある時点で、プロセスがコンディションロックを取得し、スリーパーが待っている条件を設定し、wakeup(chan)を呼び出します。条件ロックを保持したままwakeupが呼ばれることが重要です[1].wakeupはプロセステーブルをループします(kernel/proc.c:567)。これは、プロセスの状態を操作する可能性があるためと、p->lockによってsleepとwakeupがお互いを見逃さないようにするためです。wakeupは、SLEEPINGの状態にあるプロセスの中で、一致するチャンを見つけると、そのプロセスの状態をRUNNABLEに変更します。次にスケジューラが実行されたときに、そのプロセスが実行可能な状態であることがわかります。

なぜスリープとウェイクアップのロックルールは、スリープ中のプロセスがウェイクアップを逃さないようにしているのですか？スリープ中のプロセスは、条件ロックまたは自身のp->ロックのいずれか、またはその両方を

[1] 厳密に言えば、wakeup が 単にacquire の 後に続くだけで十分です（つまり、 リ リ ー ス 後にwakeupと呼ぶこともできます）。

条件をチェックする前の時点から、SLEEPINGとマークされた後の時点まで。wakeupを呼び出したプロセスは、wakeupのループでこれらのロックを*両方とも*保持します。したがって、wakerはconsumingスレッドが条件をチェックする前に条件をtrueにするか、wakerのwakeupがsleepingスレッドをSLEEPINGとマークされた後に厳密にチェックするかのどちらかになります。そうするとwakeupはスリープ中のプロセスを見て、（他のものが先に起こさない限り）それを起こします。

複数のプロセスが同じチャネルでスリープ状態になっていることがあります。例えば、複数のプロセスがパイプから読み込んでいる場合などです。wakeupを1回呼び出すと、これらすべてのプロセスが起動します。そのうちの1つが最初に実行され、sleepが呼び出されたときのロックを取得し、（パイプの場合）パイプで待機しているデータを読みます。他のプロセスは、起こされたにもかかわらず、読むべきデータがないことに気づきます。他のプロセスは、起こされたにもかかわらず、読むべきデータがないことに気づくでしょう。彼らから見れば、ウェイクアップは「偽装」であり、再びスリープしなければなりません。このような理由から、sleepは常に条件をチェックするループの中で呼び出されます。

2つのsleep/wakeupの使用者が誤って同じチャンネルを選択しても害はありません。偽のウェイクアップが発生しますが、上述のようにループすることでこの問題を許容します。sleep/wakeupの魅力の多くは、軽量であること（スリープチャンネルとして動作するための特別なデータ構造を作成する必要がない）と、インダイレクトの層を提供すること（呼び出し側はどのプロセスと対話しているかを知る必要がない）です。

6.7 コードパイプ

プロデューサとコンシューマを同期させるためにスリープとウェイクアップを使用する、より複雑な例として、xv6のパイプの実装があります。パイプの一方の端に書き込まれたバイトは、カーネル内のバッファにコピーされ、その後、パイプのもう一方の端から読み取ることができるという、パイプのインターフェースを第1章で見ました。パイプの一端に書き込まれたバイトは、カーネル内のバッファにコピーされ、パイプのもう一端から読み取ることができます。今後の章では、パイプを取り巻くファイル記述子のサポートについて検討しますが、ここではパイプライトとパイプリードの実装について見ていきましょう。

各パイプは、ロックとデータバッファを含むstruct pipeで表されます。フィールドnreadとnwriteは、バッファから読み込まれたバイト数とバッファに書き込まれたバイト数の合計を数えます。バッファは、buf[PIPESIZE-1]の次に書き込まれるバイトがbuf[0]となるように折り返します。カウントはラップしません。この規則により、実装はフルバッファ(nwrite == nread+PIPESIZE)と空のバッファ(nwrite == nread)を区別することができるが、バッファへのインデックスは単なるbuf[nread]ではなくbuf[nread % PIPESIZE]を使用しなければならないことになる(nwriteも同様)。

piperead と pipewrite の呼び出しが、2つの異なるCPU上で同時に行われたとします。pipewrite (kernel/pipe.c:77) はまずパイプのロックを取得します。このロックはカウント、データ、およびそれらに関連する不変性を保護します。次に Piperead (kernel/pipe.c:103)がロックを取得しようとしますが、できません。ロックを待つためにアクワイア(kernel/spinlock.c:22)で回転します。piperead が待っている間、pipewrite は書き込まれるバイト(addr[0..n-1])をループし、それぞれを順番にパイプに追加していきます(kernel/pipe.c:95)。このループの途中で、バッファがいっぱいになることがあります(kernel/pipe.c:85)。この場合、pipewrite は wakeup を呼んで、眠っている読者にバッファで待っているデータがあることを知らせ、その後、&pi->nwrite でスリープして、読者がバッファから何バイトか取り出すのを待ちます。スリープは、pipewrite のプロセスをスリープさせる一環として、pi->lock を解放します。

pi->lock が利用可能になったので、piperead はそれを獲得することに成功し、重要な局面に入ります：pi->nread != pi->nwrite を発見します (kernel/pipe.c:110) (pipewrite went to sleep be-...)

pi->nwrite == pi->nread+PIPESIZE (kernel/pipe.c:85))が発生すると、for ループに入り、パイプからデータをコピーし(kernel/pipe.c:117)、コピーしたバイト数だけnreadをインクリメントします。このバイト数が書き込み可能になったので、piperead は戻る前にwakeup (ker- nel/pipe.c:124)を呼び出し、眠っているライターを起こします。Wakeup は、&pi->nwriteでスリープしているプロセスを見つけます。これはpipewriteを実行していたが、バッファがいっぱいになったときに停止したプロセスです。そのプロセスを RUNNABLE としてマークします。

パイプコードでは、リーダとライタで別々のスリープチャンネルを使用しています（pi->nreadとpi->nwrite）。これは、万が一、同じパイプを待っているリーダとライタがたくさんいる場合に、システムをより効率的にするためです。パイプコードは、スリープ条件をチェックするループの中でスリープします。複数のリーダまたはライタがいる場合、最初に目覚めたプロセス以外は、条件がまだ偽であることを確認し、再びスリープします。

6.8 コードを待機、終了、キル

sleepとwakeupは様々な種類の待ち時間に利用できます。興味深い例として、第1章で紹介した、子供の終了と親のwaitの相互作用があります。後者の場合、後続のwaitの呼び出しは、exitを呼び出したずっと後に、子供の死を観察しなければなりません。xv6がwaitがそれを観察するまで子の死を記録する方法は、exitが呼び出し元をZOMBIE状態にし、親のwaitがそれに気づくまでそこに留まり、子の状態をUNUSEDに変更し、子の終了ステータスをコピーし、子のプロセスIDを親に返すというものです。親が子よりも先に終了した場合、親は子をinitプロセスに渡し、initプロセスは永久にwaitを呼び出し続けます。主な実装上の課題は、親と子のwaitとexit、およびexitとexitの間でレースやデッドロックが発生する可能性があることです。

Waitは、ウェイクアップの失敗を避けるために、呼び出したプロセスのp->lockを条件ロックとして使用し、開始時にそのロックを取得します(kernel/proc.c:383)。その後、プロセステーブルをスキャンします。ZOMBIE状態の子を見つけると、その子のリソースとproc構造体を解放し、その子の終了ステータスをwaitに与えられたアドレスにコピーし（0でない場合）、その子のプロセスIDを返します。waitは子プロセスを発見したが、どれも終了していない場合、sleepを呼び出して子プロセスの1つが終了するのを待ち(kernel/proc.c:430)、その後再びスキャンを行います。ここで、sleepで解放される条件ロックは、上述の特殊なケースである待機プロセスのp->ロックです。waitはしばしば2つのロックを保持していること、子のロックを取得しようとする前に自分のロックを取得していること、そしてデッドロックを避けるためにxv6のすべてが同じロック順序（親、子の順）に従わなければならないことに注意してください。

Wait はすべてのプロセスの np->parent

を見て子プロセスを探します。これは、共有変数はロックによって保護されなければならないという通常のルールに違反しています。np が現在のプロセスの祖先である可能性もありますが、その場合は np->lock を取得すると前述の順序に反してデッドロックが発生する可能性があります。プロセスの parent フィールドは親によってのみ変更されるので、np->parent==p が真であれば、現在のプロセスが変更しない限り値は変更できません。

Exit (kernel/proc.c:318) は、終了ステータスを記録し、いくつかのリソースを解放し、任意の子供を

initプロセスを実行し、待機中の親を起こし、呼び出し元をゾンビとしてマークし、CPUを恒久的に解放します。最後のシーケンスは少しトリッキーです。終了するプロセスは、親のロックを保持しながら、その状態をZOMBIEに設定し、親をウェイクアップしなければなりません。子は自分のp->ロックも保持していなければなりません。そうしないと、親は子の状態がZOMBIEであることを見て、子がまだ実行中に子を解放する可能性があるからです。ロックの取得順序はデッドロックを避けるために重要です。waitは親のロックを子のロックより先に取得するので、exitも同じ順序で取得しなければなりません。

Exitでは、特殊なウェイクアップ関数であるwakeup1が呼び出されます。この関数は、親がwaitでスリープしている場合に限り、親のみをウェイクアップさせます（kernel/proc.c:583）。wakeup1によって親が実行されることがありますが、wait内のループは、子のp->lockがスケジューラによって解放されるまで子を調べることができず、exitがその状態をZOMBIEに設定したかなり後までwaitはexitプロセスを調べることができません（kernel/proc.c:371）。

exitはプロセスの自己終了を可能にしますが、kill（kernel/proc.c:596）はあるプロセスが他のプロセスの終了を再要求します。犠牲者は別のCPUで実行されているかもしれないし、カーネルのデータ構造に対する繊細な一連の更新を行っている最中かもしれないので、killが犠牲者のプロセスを直接破壊するのは複雑すぎます。そのため、killはほとんど何もしません。犠牲者のp->killedを設定し、眠っている場合はそれを起こしているだけです。最終的に犠牲者はカーネルに入ったり出たりしますが、その時点で usertrap のコードは、p->killedが設定されていれば exitを呼び出します。犠牲者がユーザースペースで動作している場合、システムコールを実行するか、タイマー（または他のデバイス）の割り込みによって、すぐにカーネルに入ることになります。

犠牲者のプロセスがスリープ状態の場合、killのwakeup呼び出しにより犠牲者がスリープから復帰します。これは、待機中の条件が真でない可能性があるため、潜在的に危険です。しかし、xv6のsleepの呼び出しは常にwhileループに包まれており、sleepが戻った後に条件を再テストします。sleepの呼び出しの中には、ループの中でp->killedをテストし、それが設定されている場合は現在のアクティビティを放棄するものもあります。これは、放棄することが正しい場合にのみ行われます。例えば、パイプの読み書きコード（kernel/pipe.c:86）は、killedフラグが設定されている場合に戻りますが、最終的にコードはtrapに戻り、再びフラグをチェックして終了します。

xv6 のスリープループの中には p->killed をチェックしないものがありますが、これはコードがアトミックであるべき複数ステップのシステムコールの途中にあるためです。virtio ドライバ (kernel/virtio_disk.c:242) がその例で、ディスク操作がファイルシステムを正しい状態にするために必要な一連の書き込みのうちの一つである可能性があるため、p->killed をチェックしません。ディスクI/Oを待っている間にkillされたプロセスは、現在のシステムコールが完了し、usertrapがkillフラグを確認するまで終了しません。

6.9 リアルワールド

xv6のスケジューラは、各プロセスを順番に実行するシンプルなスケジューリングポリシーを実装しています。このポリシーは*ラウンドロビン*と呼ばれています。実際のOSでは、プロセスに優先度を持たせるなど、より高度なポリシーが実装されています。これは、実行可能な高い優先度のプロセスが、実行可能な低い優先度のプロセスよりもスケジューラによって優先されるという考え方です。このようなポリシーは、しばしば競合する目標があるため、すぐに複雑になってしまいます。例えば、運用側は、公平性と高いスループットを保証したいと思うかもしれません。また、複雑なポリシーを設定することで、不完全な

は、*優先順位の逆転*や*コンボイ*などの相互作用を傾向づけました。優先順位の逆転は、優先度の低いプロセスと優先度の高いプロセスがロックを共有している場合に起こります。このロックを優先度の低いプロセスが取得すると、優先度の高いプロセスの処理が妨げられます。また、多くの高優先度プロセスが、共有ロックを取得する低優先度プロセスを待っているときに、待ち行列が形成されることがあり、いったん待ち行列が形成されると、その状態が長く続くことがあります。このような問題を回避するために、高機能なスケジューラには追加のメカニズムが必要です。

スリープとウェイクアップはシンプルで効果的な同期方法ですが、他にもたくさんの方法があります。いずれの場合も、最初の課題は、この章の冒頭で見た「ウェイクアップの消失」という問題を避けることです。オリジナルのUnixカーネルのスリープは、単純に割り込みを無効にするもので、UnixがシングルCPUのシステム上で動作していたため、それで解決していました。xv6はマルチプロセッサ上で動作するため、スリープに明示的なロックを加えています。FreeBSDの`msleep`も同様のアプローチをとっています。Plan 9の`sleep`は、スリープする直前にスケジューリングロックを保持したまま実行されるコールバック関数を使用しています。この関数は、スリープ状態を最後にチェックして、ウェイクアップの失敗を防ぐ役割を果たします。Linuxカーネルのスリープでは、待機チャネルの代わりに、wait queueと呼ばれる明示的なプロセスキューを使用しています。

`wakeup`でプロセスリスト全体をスキャンして、マッチする`chan`を持つプロセスを探すのは非効率的です。より良い解決策は、`sleep`と`wakeup`の両方の`chan`を、Linuxのwait queueのような、その構造上でスリープしているプロセスのリストを保持するデータ構造に置き換えることです。Plan 9の`sleep`と`wakeup`では、その構造体をランデブー・ポイント（`Rendez`）と呼んでいます。多くのスレッドライブラリでは、同じ構造体を条件変数として参照しています。この文脈では、`sleep`と`wakeup`の操作は`wait`と`signal`と呼ばれています。これらのメカニズムには共通点があります。それは、スリープ状態が、スリープ中にアトミックに落とされるある種のロックによって保護されていることです。

`wakeup`の実装では、特定のチャンネルを待っているすべてのプロセスを起こしますが、その特定のチャンネルを多くのプロセスが待っている場合があります。オペレーティングシステムはこれらすべてのプロセスをスケジューリングし、スリープ状態をチェックするために競争します。このような動作をするプロセスは「雷の*ような群れ*」と呼ばれることがありますが、これは避けた方がよいでしょう。ほとんどの条件変数には、ウェイクアップのための2つのプリミティブがあります。1つのプロセスをウェイクアップさせるシグナルと、待っているすべてのプロセスをウェイクアップさせるブロードキャストです。

同期にはセマフォがよく使われます。このカウントは通常、パイプバッファで利用可能なバイト数や、プロセスが持つゾンビチルドレンの数などに相当します。明示的なカウントを抽象化の一部として使用することで、「ロスト・ウェイクアップ」問題を回避できます。つまり、発生したウェイクアップの数が明示的にカウントされます。また、カウントすることで、spurious wakeupやthundering herdの問題も回避できます。

xv6では、プロセスの終了とクリーンアップが非常に複雑です。例えば、犠牲者のプロセスはカーネルの奥深くで眠っている可能性があり、そのスタックをアンワインドするには多くの注意深いプログラミングが必要となるからです。多くのオペレーティングシステムでは、`longjmp`のような例外処理のための明示的なメカニズムを使用してスタックをアンワインドします。さらに、待っているイベントがまだ起こっていないにもかかわらず、スリープ中のプロセスを起こす原因となるイベントが他にもあります。たとえば、Unixのプロセスがスリープしているときに、他のプロセスがそのプロセスにシグナルを送ることがあります。この場合、プロセスは中断されたシステムコールから、値-1とエラーコードがEINTRに設定された状態で戻ります。アプリケーションはこれらの値をチェックして、何をすべきかを決めることができます。Xv6はシグナルをサポートしていないので、このような複雑さは生じません。

Xv6のkillサポートは完全に満足できるものではありません。p->killedをチェックすべきスリープループがあります。関連した問題として、p->killedをチェックするsleepループでも、sleepとkillの間で競合が発生します。犠牲者のループがp->killedをチェックした直後、sleepを呼び出す前に、killがp->killedを設定して犠牲者を起こそうとすることがあります。この問題が発生すると、被害者は待っていた条件が発生するまでp->killedに気づきません。これは、かなり後になるかもしれませんし(例えば、犠牲者が待っているディスクブロックをvirtioドライバが返したとき)、全くならないかもしれません(例えば、犠牲者がコンソールからの入力を待っているが、ユーザが何も入力しなかったとき)。

実際のオペレーティングシステムでは、allocprocの線形時間検索ではなく、明示的なfreeリストを用いて定数時間でfree proc構造を見つけるでしょう。

6.10 エクササイズ

1. Sleep はデッドロックを避けるために lk != &p->lock をチェックしなければなりません (kernel/proc.c:543-546)。を置き換えることで、この特殊なケースが解消されたとします。

```
if(lk != &p->lock){
  acquire(&p->lock);
  release(lk);
}
```

で

```
release(lk);
acquire(&p->lock);
```

これをやると睡眠が壊れる。どうやって？

2. ほとんどのプロセスのクリーンアップはexitかwaitのどちらかで行うことができました。しかし、開いているファイルを閉じるのはexitでなければならないことがわかった。なぜでしょう？その答えはパイプにあります。

3. xv6でsleepとwakeupを使わずにセマフォを実装する（ただし、スピンロックを使うのはOK）。xv6のsleepとwakeupの用途をセマフォに置き換えてみてください。結果を判断してください。

4. 犠牲者の sleep ループが p->killed をチェックした後、sleep を呼び出す前に kill を行うと、犠牲者は現在のシステムコールを放棄することになるので、前述の kill と sleep の間の競合を修正しました。

5. 例えば、virtioドライバのプロセスが他のプロセスに殺された場合、whileループから素早く戻ることができるように、すべてのsleepループが`p->killed`をチェックするようにプランを設計します。

6. xv6を修正して、あるプロセスのカーネル・スレッドから別のスレッドに切り替えるときに、スケジューラ・スレッドを介して切り替えるのではなく、1つのコンテキスト・スイッチだけを使用するようにしました。切り替え先のスレッドは、自分で次のスレッドを選択して`swtch`を呼び出す必要があります。課題は、複数のコアが誤って同じスレッドを実行しないようにすること、ロッキングを正しく行うこと、そしてデッドロックを避けることです。

7. 実行可能なプロセスがないときに、RISC-Vのwfi（割込み待ち）命令を使用するようにxv6のスケジューラを変更します。実行可能なプロセスが待機しているときはいつでも、wfiで一時停止しているコアがないことを確認してください。

8. p->lock ロックは多くの不変量を保護しており、p->lock で保護されている xv6 コードの特定の部分を見たときに、どの不変量が適用されているのかを把握するのは難しいでしょう。p->lock を複数のロックに分割することで、よりクリーンなプランを設計します。

第7章

ファイルシステム

ファイルシステムの目的は、データを整理して保存することです。ファイルシステムは通常、ユーザーやアプリケーション間でのデータの共有や、再起動後もデータを利用できるような*永続性*をサポートする。

xv6ファイルシステムは、Unixライクなファイル、ディレクトリ、パス名を提供し（第1章参照）、データをVirtioディスクに保存して永続性を持たせています（第4章参照）。このファイルシステムはいくつかの課題に対応しています。

- ファイルシステムでは、名前の付いたディレクトリやファイルのツリーを表現したり、各ファイルのコンテンツを格納するブロックの識別情報を記録したり、ディスクのどの領域が空いているかを記録したりするために、ディスク上のデータ構造が必要になります。
- ファイルシステムは、*クラッシュリカバリー*に対応している必要があります。つまり、クラッシュ（電源障害など）が発生しても、再起動後にファイルシステムが正しく動作する必要があります。リスクとしては、クラッシュによって一連の更新が中断され、ディスク上に不整合なデータ構造（例えば、ファイルで使用されていると同時にフリーとマークされているブロック）が残る可能性があります。
- 異なるプロセスが同時にファイルシステムを操作する可能性があるため、ファイルシステムのコードは不変性を維持するために調整する必要があります。
- ディスクへのアクセスはメモリへのアクセスに比べて桁違いに遅いため、ファイルシステムは人気のあるブロックをメモリ内にキャッシュしておく必要があります。

本章の残りの部分では、xv6がこれらの課題にどのように対処するかを説明します。

7.1 概要

xv6ファイルシステムの実装は、図7.1に示す7つの層で構成されています。ディスク層は、virtioのハードドライブ上のブロックを読み書きします。バッファ・キャッシュ層は、ディスク・ブロックをキャッシュし、アクセスを同期させることで、一度に1つのカーネル・プロセスだけが特定のブロックに格納されたデータを変更できるようにします。ロギング層では、上位層が複数のブロックの更新を*トランザクション*にまとめることができ、ブロックがフェースの中でアトミックに更新されることを保証します。

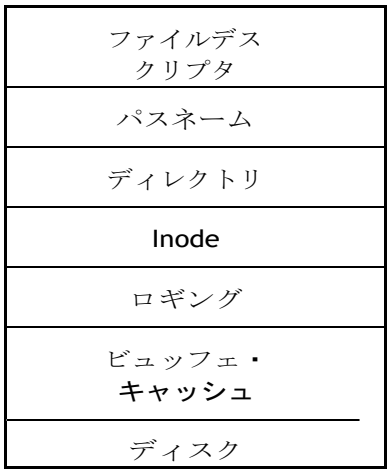


図7.1:xv6ファイルシステムのレイヤー。

のクラッシュ（つまり、すべてが更新されるか、1つも更新されないか）が発生します。inode層は個々のファイルを提供し、各ファイルは一意のi-numberを持つ*inode*として表現され、ファイルのデータを保持するいくつかのブロックを持つ。ディレクトリ層は、各ディレクトリを特殊なinodeとして実装し、その内容はディレクトリエントリのシーケンスであり、各エントリにはファイル名とi-番号が含まれる。パス名層は、`/usr/rtm/xv6/fs.c`のような階層的なパス名を提供し、再帰的なルックアップでそれらを解決します。ファイルディスクリプター層は、多くのUnixリソース（パイプ、デバイス、ファイルなど）をファイルシステムのインターフェイスを使って抽象化し、アプリケーションプログラマの生活を簡素化します。

ファイルシステムは、ディスクのどこにinodeやコンテンツブロックを格納するかの計画を立てなければなりません。そのために、xv6は図7.2に示すように、ディスクをいくつかのセクションに分割します。ファイルシステムでは、ブロック0は使用しません（ブートセクタが格納されています）。ブロック1は*スーパーブロック*と呼ばれ、ファイルシステムに関するメタデータ（ファイルシステムのサイズ（ブロック単位）、データブロック数、inode数、ログのブロック数）が格納されています。2から始まるブロックにはログが入っています。ログの後にはinodeがあり、ブロックごとに複数のinodeがあります。その後、どのデータブロックが使用されているかを追跡するビットマップブロックが続きます。残りのブロックはデータブロックで、それぞれがビットマップブロックでフリーとマークされているか、ファイルやディレクトリのコンテンツを保持しています。スーパーブロックは、`mkfs`と呼ばれる別のプログラムによって埋められ、初期のファイルシステムが構築されます。

この章の残りの部分では、バッファキャッシュから始めて、各層について説明します。この章では、下層の抽象化をうまく利用することで、上層の設計を容易にしている場面に注目してください。

7.2 バッファキャッシュ層

バッファキャッシュには2つの役割があります。(1)ディスクブロックへのアクセスを同期させ、ブロックのコピーがメモリ内に1つだけ存在し、一度に1つのカーネルスレッドだけがそのコピーを使用するようにする。(2)人気のあるブロックをキャッシュし、低速のディスクから再読込する必要がないようにする。コードは`bio.c`にあります。

バッファキャッシュが提供する主なインターフェースは、`bread`と`bwrite`で構成されていますが、前者は
は，メモリ上で読み書き可能なブロックのコピーを含む*buf*を取得し，後者は変更されたバッファをディスク上の適切なブロックに書き込みます。カーネルスレッドは、バッファの使用が終わったら、`brelse`を呼び出してバッファを解放しなければなりません。バッファキャッシュは、バッファごとのスリープロックを使用して、次のことを保証します。

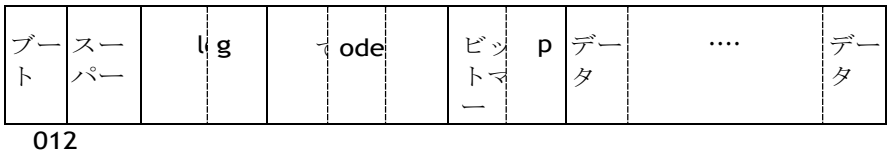


図7.2: xv6ファイルシステムの構造。

breadはロックされたバッファを返し、brelseはロックを解除します。

バッファキャッシュの話に戻ろう。バッファキャッシュは、ディスクブロックを保持するための固定数のバッファを持っています。つまり、ファイルシステムがまだキャッシュ内にないブロックを要求した場合、バッファキャッシュは、現在他のブロックを保持しているバッファをリサイクルしなければなりません。バッファキャッシュは、新しいブロックのために、最も最近使用されたバッファをリサイクルします。最近使用されたバッファとは、すぐに再び使用される可能性が最も低いバッファであると仮定しています。

7.3 コードバッファキャッシュ

バッファキャッシュは、バッファの二重リンクリストです。main (kernel/- main.c:27) から呼び出される関数 binit は、静的配列 buf の NBUF バッファでリストを初期化します (kernel/bio.c:42-51)。バッファキャッシュへの他のすべてのアクセスは、buf 配列ではなく bcache.head を通じてリンクリストを参照します。

バッファには、2つのステートフィールドが関連付けられています。フィールド valid は 、 バッファがブロックのコピーを保持していることを示します。フィールドdisk は 、バッファの内容がディスクに渡され、バッファが変更される可能性があることを示しています（例えば、ディスクからデータにデータを書き込むことができます）。

bread (kernel/bio.c:91) は bget を呼び出して、与えられたセクター (kernel/bio.c:95) のバッファを取得します。バッファがディスクから読み込まれる必要がある場合、bread はバッファを返す前に virtio_disk_rw を呼び出してそれを行います。

Bget (kernel/bio.c:58) は、バッファリストをスキャンして、指定されたデバイス番号とセクタ番号を持つバッファを探します (kernel/bio.c:64-72)。そのようなバッファがあれば、bget はそのバッファのスリープロックを取得する。Bget は 、ロックされたバッファを返します。

指定されたセクタにキャッシュされたバッファがない場合、bget は別のセクタを保持していたバッファを再利用してバッファを作成しなければなりません。Bget はバッファ・リストを 2 度目にスキャンし、使用されていないバッファを探します (b->refcnt = 0); そのようなバッファはすべて使用できます。Bget は、バッファのメタデータを編集して新しいデバイスとセクタの番号を記録し、スリープロックを取得します。なお、b->valid =

0を指定することで、bgetはバッファの以前の内容を誤って使用することなく、ディスクからブロックデータを確実に読み込むことができます。

キャッシュされたバッファは、ディスク・セクタごとに最大1つであることが重要です。これは、読者が書き込みを確認できるようにするためと、ファイル・システムが同期のためにバッファにロックをかけるためです。Bgetは、最初のループでブロックがキャッシュされているかどうかをチェックしてから、2番目のループでブロックがキャッシュされていることを宣言するまで（dev、blockno、refcntを設定するまで）、bache.lockを継続的に保持することで、この不変性を確保しています。これにより、ブロックの有無のチェックと（存在しない場合）ブロックを格納するバッファの指定がアトミックに行われます。

bgetがbcache.lockクリティカルセクションの外でバッファのスリープロックを取得しても安全です。これは、非ゼロのb->refcntによって、バッファが別の

ディスクブロックのスリープロックは、ブロックのバッファリングされたコンテンツの読み取りと書き込みを保護します。
bcache.lockは、どのブロックがキャッシュされているかという情報を保護します。

すべてのバッファがビジー状態の場合、ファイルシステムコールを同時に実行しているプロセスが多すぎるため、bgetはパニックに陥ります。より優雅な対応としては、バッファが空くまでスリープすることが考えられますが、その場合はデッドロックが発生する可能性があります。

breadが（必要に応じて）ディスクを読み込み、バッファを呼び出し元に返した後は、呼び出し元がバッファを独占的に使用し、データのバイトを読み書きできるようになります。呼び出し側がバッファを変更する場合は、バッファを解放する前にbwriteを呼び出して変更したデータをディスクに書き込まなければならない。Bwrite (kernel/bio.c:105)は、ディスクハードウェアと対話するために virtio_disk_rw を呼び出します。

呼び出し側は、バッファを使い終わったら、brelseを呼び出してバッファを解放しなければなりません。(b-releaseを短縮したbrelseという名前は暗号のようですが、覚える価値はあります。これはUnixで生まれ、BSD、Linux、Solarisでも使用されています)。Brelse (kernel/bio.c:115)はスリープロックを解除し、バッファをリンクリストの先頭に移動させます(kernel/bio.c:126-131)。バッファを移動させると、リストはバッファが使用された（解放されたという意味）日にち順になります。リストの最初のバッファが最も最近使用されたもので、最後のバッファが最も最近使用されたものです。bgetの2つのループはこれを利用しています。既存のバッファを探すスキャンは、最悪の場合、リスト全体を処理しなければなりませんが、最近使用されたバッファを最初にチェックすることで（bcache.headから始まり、次のポインタをたどる）、参照の位置関係が良好な場合にはスキャン時間を短縮することができます。再利用するバッファを選ぶためのスキャンでは、（prevポインタをたどって）逆方向にスキャンすることにより、最も最近使用されたバッファを選びます。

7.4 ロギング層

ファイルシステムの設計で最も興味深い問題の1つがクラッシュリカバリーです。この問題は、ファイルシステムの操作の多くがディスクへの複数回の書き込みを伴うため、書き込みの一部の後にクラッシュが発生すると、ディスク上のファイルシステムが不整合な状態になってしまう可能性があることから生じます。例えば、ファイルのトランケーション（ファイルの長さをゼロにしてコンテンツブロックを解放する）中にクラッシュが発生したとします。ディスク書き込みの順番によっては、クラッシュによって、空きとマークされたコンテンツブロックへの参照を持つinodeが残る場合と、割り当てられているが参照されていないコンテンツブロックが残る場合があります。

後者は比較的問題ありませんが、解放されたブロックを参照しているinodeは、再起

動後に深刻な問題を引き起こす可能性があります。再起動後、カーネルはそのブロックを別のファイルに割り当ててしまうかもしれず、2つの異なるファイルが意図せずに同じブロックを指していることになります。もしxv6が複数のユーザをサポートしていたら、この状況はセキュリティ上の問題になるかもしれません。なぜなら、古いファイルの所有者は、別のユーザが所有する新しいファイルのブロックを読み書きできるからです。

Xv6では、ファイルシステム操作中のクラッシュの問題を、シンプルな形のロギングで解決しています。xv6のシステムコールは、ディスク上のファイルシステムのデータ構造を直接書き込むことはありません。代わりに、ディスク上の*ログ*に希望するすべてのディスク書き込みの記述を配置します。システムコールがすべての書き込みをログに記録すると、ログに操作が完了したことを示す特別な*コミットレコード*をディスクに書き込みます。この時点で、システムコールは書き込みをディスク上のファイルシステムのデータ構造にコピーします。書き込みが完了した後、システムコールはディスク上のログを消去します。

システムがクラッシュして再起動した場合、ファイルシステムのコードは、プロセスを実行する前に以下のようにクラッシュから回復します。ログが完全な操作を含んでいるとマークされている場合には

復旧コードは、書き込みをディスク上のファイルシステムのあるべき場所にコピーします。ログに完全な操作が含まれているというマークがない場合、リカバリーコードはそのログを無視します。リカバリーコードは、ログを消去して終了します。

なぜxv6のログは、ファイルシステム操作中のクラッシュの問題を解決するのでしょうか？操作がコミットされる前にクラッシュが発生した場合、ディスク上のログは完了としてマークされず、リカバリーコードはそれを無視し、ディスクの状態は操作が開始されなかったかのようになります。操作がコミットした後にクラッシュが発生した場合、リカバリーは操作の書き込みをすべて再生し、ディスク上のデータ構造に書き込むために操作が開始されていた場合は、その書き込みを繰り返すことになります。いずれの場合でも、ログはクラッシュに対して操作をアトミックにします。つまり、リカバリー後に操作の書き込みがすべてディスクに現れるか、あるいはまったく現れないかのどちらかです。

7.5 ログデザイン

ログは、スーパーブロックで指定された既知の固定された場所に存在します。ログは、ヘッダーブロックに続いて、更新されたブロックコピー（「ログブロック」）のシーケンスで構成されています。ヘッダーブロックには、ログされたブロックごとのセクタ番号の配列と、ログされたブロックの数が含まれています。ディスク上のヘッダーブロックのカウントは、ログにトランザクションがないことを示すゼロか、ログに指定された数のログブロックを持つ完全なコミットされたトランザクションが含まれていることを示すゼロ以外のいずれかである。Xv6は、トランザクションのコミット時にヘッダーブロックを書き込みますが、それ以前には書き込みません。また、ログに記録されたブロックをファイルシステムにコピーした後、カウントをゼロに設定します。したがって、トランザクションの途中でクラッシュした場合、ログのヘッダブロックのカウントはゼロになり、コミット後にクラッシュした場合はゼロ以外のカウントになります。

各システムコールのコードは、クラッシュに対してアトミックでなければならない一連の書き込みの開始と終了を示しています。異なるプロセスによるファイルシステム操作の同時実行を可能にするために、ロギングシステムは複数のシステムコールの書き込みを1つのトランザクションに蓄積することができます。そのため、1つのコミットに複数の完全なシステムコールの書き込みが含まれることがあります。1つのシステムコールが複数のトランザクションにまたがることを避けるため、ロギングシステムはファイルシステムのシステムコールが進行していないときにのみコミットします。

複数のトランザクションをまとめてコミットする考え方を*グループコミット*といいます。グループコミットは、コミットにかかる固定費を複数の操作で償却するため、ディスク操作の回数を減らすことができます。また、グループコミットは、ディスクシステムに、より多くの同時書き込みを可能にし、おそらく、1回のディスクローテーション中にすべての書き込みを行うことができます。Xv6のvirtioドライバはこのような*バッチ処*

*理*をサポートしていませんが、xv6のファイルシステムの設計はそれを可能にします。

Xv6では、ログを格納するためにディスク上に一定量のスペースを確保します。トランザクションでシステムコールによって書き込まれたブロックの合計数が、そのスペースに収まらなければなりません。これは2つの結果をもたらします。1つのシステムコールが、ログにあるスペースよりも多くの個別ブロックを書き込むことはできません。これはほとんどのシステムコールでは問題になりませんが、多くのブロックを書き込む可能性のあるシステムコールが2つあります：書き込みとアンリンクです。大きなファイルの書き込みでは、多くのデータブロックと多くのビットマップブロック、そしてinodeブロックが書き込まれる可能性があります。大きなファイルのリンク解除では、多くのビットマップブロックとinodeが書き込まれる可能性があります。Xv6の書き込みシステムコールは、大きな書き込みをログに収まるような複数の小さな書き込みに分割します。また、xv6ファイルシステムは実際には1つのビットマップブロックしか使用しないため、アンリンクは問題を起こしません。ログスペースが限られていることのもう一つの結果は、ログシステムがシステムコールに

は、システムコールの書き込みがログの残りのスペースに収まることが確実でない限り、開始しません。

7.6 コード：ロギング

システムコールでのログの典型的な使い方は次のようなものです。

```
begin_op()です。
...
bp = bread(...)です。
bp->data[...] =
...; log_write(bp);
...
end_op()です。
```

begin_op (kernel/log.c:126)
は、ログシステムが現在コミットしておらず、この呼び出しによる書き込みを保持するのに十分な未予約のログスペースがあるまで待ちます。
は、ログスペースを予約しているシステムコールの数を数えます。予約されたスペースの合計は、log.extracted×MAXOPBLOCKSです。log.extractionを増加させることで、スペースを確保すると同時にコミュニケーションの悪化を防ぐことができます。このシステムコールの間に発生しないようにします。このコードでは、各システムコールが最大でMAXOPBLOCKS個のブロックを書き込むことを保守的に想定しています。

log_write (kernel/log.c:214) は、bwriteのプロキシとして動作します。ブロックのセクタ番号を記録します。
ブロックはコミットされるまでキャッシュに保持されます。そのブロックはコミットされるまでキャッシュに残っていなければなりません。コミットされるまでは、キャッシュされたコピーが変更の唯一の記録となり、コミットされるまではディスク上の元の場所に書き込まれることはなく、同じトランザクション内の他のリードが変更を確認しなければなりません。log_writeは、1つのトランザクション内で1つのブロックが複数回書き込まれたことを認識し、そのブロックにログの同じスロットを割り当てます。このような最適化は、しばしば*吸収*と呼ばれます。例えば、複数のファイルのinodeを含むディスクブロックが1つのトランザクション内で複数回書き込まれることはよくあることです。複数回のディスク書き込みを1回に吸収することで、ファイルシステムはログスペースを節約することができ、ディスクブロックのコピーを1つだけディスクに書き込む必要があるため、より良いパフォーマンスを得ることができます。

end_op
(kernel/log.c:146)は、まず未処理のシステムコールのカウントをデクリメントします。このカウントがゼロになった場合、commit()を呼び出して現在のトランザクションをコミットします。write_log() (kernel/log.c:178)
は、トランザクションで変更された各ブロックをバッファキャッシュからディスク上のログのスロットにコピーします。 write_head() (kernel/log.c:102)

は、ヘッダブロックをディスクに書き込みます。これは、次のトランザクションがログに記録されたブロックの書き込みを開始する前に行われなければならず、クラッシュしても、あるトランザクションのヘッダと次のトランザクションのログに記録されたブロックを使用してリカバリが行われないようにします。

`recover_from_log` (kernel/log.c:116) は、最初のユーザプロセスが実行される前のブート時に `fsinit`(kernel/fs.c:43) から呼び出される `initlog` (kernel/log.c:55) から呼び出される (kernel/proc.c:524) 。ログのヘッダを読み、ログがコミットされたトランザクションを含んでいることをヘッダが示していれば、`end_op`の動作を模倣します。

ログの使用例としては、filewrite（kernel/file.c:135）があります。トランザクションは次のようになります。

```
begin_op();
ilock(f->ip);
r = writei(f->ip,
...); iunlock(f->ip);
end_op()となります。
```

このコードは、ログがオーバーフローしないように、大きな書き込みを一度に数セクタの個々のトランザクションに分割するループに包まれています。writeiの呼び出しは、このトランザクションの一部として、ファイルのinode、1つまたは複数のビットマップブロック、およびいくつかのデータブロックなど、多くのブロックを書き込みます。

7.7 コードブロックアロケータ

xv6のブロックアロケータは、ディスク上にフリービットマップを保持しており、ブロックごとに1ビットが割り当てられています。0のビットは対応するブロックがフリーであることを示し、1のビットは使用中であることを示す。mkfsというプログラムは、ブート・セクタ、スーパーブロック、ログ・ブロック、inodeブロック、ビットマップ・ブロックに対応するビットを設定します。

ブロックアローケータには，新しいディスクブロックを割り当てる balloc と，ブロックを解放する bfree という2つの関数があります。balloc (kernel/fs.c:72) にある balloc のループは、ブロック 0 からファイルシステムのブロック数である sb.size までのすべてのブロックを考慮します。ballocは、ビットマップビットが0のブロックを探し、それがフリーであることを示します。balloc はそのようなブロックを見つけると、ビットマップを更新してブロックを返します。効率化のため、このループは2つに分かれています。外側のループは、ビットマップビットの各ブロックを読み取ります。内側のループは、1つのビットマップブロック内のすべてのBPBビットをチェックします。2つのプロセスが同時にブロックを割り当てようとした場合に発生する可能性のあるレースは、バッファキャッシュが一度に1つのビットマップブロックを使用できるのは1つのプロセスだけであるという事実によって防止されています。

Bfree (kernel/fs.c:91) は、正しいビットマップブロックを見つけ、正しいビットをクリアします。この場合も、breadとbrelseが示す排他的使用により、明示的なロックの必要性はありません。

本章の残りの部分で説明するコードの多くと同様に、balloc とbfreeはトランザクション内で呼び出す必要があります。

7.8 Inode層

*inode*という言葉には、2つの関連した意味があります。inode」は、ファイルのサイズと

データブロック番号のリストを含むディスク上のデータ構造を指す場合と、ディスク上のinodeのコピーとカーネル内で必要な追加情報を含むメモリ内inodeを指す場合があります。また、「inode」は、ディスク上のinodeのコピーと、カーネル内で必要な追加情報を含むメモリ内inodeを指す場合もあります。

ディスク上のinodeは、inodeブロックと呼ばれるディスクの連続した領域に詰め込まれています。すべてのinodeは同じ大きさなので、数nが与えられれば、ディスク上のn番目のinodeを見つけるのは簡単です。実際、inode番号またはi番号と呼ばれるこの数字nは、実装においてinodeを識別する方法です。ディスク上のinodeは、`dinode`構造体(kernel/fs.h:32)で定義されます。`type`フィールドは、ファイル、ディレクトリ、特殊ファイル（デバイス）を区別する。タイプがゼロの場合は、ディスク上の

ディスクのinodeが解放されたことを示します。nlinkフィールドは、ディスク上のinodeとそのデータブロックがいつ解放されるかを認識するために、このinodeを参照しているディレクトリ・エントリの数をカウントします。sizeフィールドは、ファイルのコンテンツのバイト数を記録します。addrs配列には、ファイルのコンテンツを格納しているディスクブロックのブロック番号が記録されている。

struct inode (kernel/file.h:17) は、ディスク上の struct dinode のメモリ内コピーです。カーネルは、inodeを参照するCポインタがある場合にのみ、inodeをメモリに格納します。refフィールドは、メモリ内のinodeを参照しているCポインタの数をカウントし、参照カウントがゼロになるとカーネルはinodeをメモリから破棄します。iget関数とiput関数は、inodeへのポインタの取得と解放を行い、参照カウントを変更します。inodeへのポインタは、ファイル記述子、現在の作業ディレクトリ、execなどの一時的なカーネルコードから得られます。

icache.lockは、inodeがキャッシュに一度しか存在しないという不変性と、キャッシュされたinodeのrefフィールドが、キャッシュされたinodeへのメモリ内ポインタの数をカウントするという不変性を保護します。各in-memory inodeはsleep-lockを含むlockフィールドを持ち、inodeのフィールド（ファイル長など）やinodeのファイルやディレクトリのコンテンツ・ブロックへの排他的なアクセスを保証します。inodeのrefが0より大きい場合、システムはそのinodeをキャッシュに保持し、別のinodeのためにキャッシュ・エントリを再利用しないようにする。最後に、各inodeには、ファイルを参照するディレクトリ・エントリの数をカウントするnlinkフィールド（ディスク上にあり、キャッシュされている場合はメモリにコピーされます）があります。

iget() が返す struct inode ポインタは、対応する iput() を呼び出すまで有効であることが保証されています。inode は削除されず、ポインタが参照するメモリは別の inode に再利用されることはありません。 iget() は inode への非排他的なアクセスを提供するため、同じ inode へのポインタが多数存在する可能性があります。ファイルシステムのコードの多くの部分は、inodeへの長期的な参照（オープンファイルやカレントディレクトリなど）を保持するために、また、複数のinodeを操作するコード（パス名検索など）においてデッドロックを回避しつつ競合を防ぐために、iget()のこの動作に依存しています。

igetが返すinode 構造体は、有用なコンテンツを持たない可能性があります。ディスク上のinodeのコピーを確実に保持するために、コードはilockを呼び出す必要があります。これはinodeをロックし（他のプロセスがilockできないように）、まだ読まれていなければディスクからinodeを読み込みます。inodeポインタの取得とロックを分離することで、ディレクトリ検索時など、いくつかの状況でデッドロックを回避することができます。複数のプロセスが iget

が返す inode への C ポインタを保持できますが、一度にロックできるのは 1 つのプロセスだけです。

inodeキャッシュは、カーネルコードやデータ構造がCポインタを保持するinodeのみをキャッシュします。このキャッシュの主な仕事は、複数のプロセスによるアクセスを同期させることであり、キャッシュは二の次です。inodeが頻繁に使用される場合、inodeキャッシュで保持されていなければ、おそらくバッファキャッシュがそのinodeをメモリに保持します。inodeキャッシュは*ライトスルー*です。つまり、キャッシュされたinodeを変更するコードは、`iupdate`で直ちにディスクに書き込まなければなりません。

7.9 コードInodes

新しいinodeを割り当てるために（例えば、ファイルを作成するときなど）、xv6はialloc
を呼び出します（kernel/fs.c:197）。Iallocはballoc に
似ています。ディスク上のinode構造体を1ブロックずつループして、フリーとマークされて
いるものを探します。見つけたら、新しい タ イ プ
をディスクに書き込むことでそれを要求し、その後、iget の
テールコールでinodeキャッシュからエントリを返します(kernel/fs.c:211)。ialloc が
正しく動作するかどうかは、一度に1つのプロセスしかbpへの参照を保持できないという事
実にかかっています。

Iget (kernel/fs.c:244) は、inode キャッシュを調べて、目的のデバイスと inode
番号を持つアクティブなエントリ (ip->ref > 0)
を探します。見つかれば、そのinodeへの新しい参照を返します(kernel/fs.c:253-257)。iget
が ス キ ャ ン す る 際には、最初に空いたスロットの位置を記録し（kernel/fs.c:258-
259）、キャッシュ・エントリを割り当てる必要がある場合にはこれを使用します。

コードは、inodeのメタデータやコンテンツを読み書きする前に、ilockを使ってinodeを
ロックする必要があります。Ilock (kernel/fs.c:290)
は、この目的のためにスリープ・ロックを使用します。ilockがinodeへの排他的なアクセス
を行うと、必要に応じてディスク（より正確にはバッファキャッシュ）からinodeを読み
込みます。関数 iunlock (kernel/fs.c:318)
はスリープロックを解除しますが、これにより、眠っているプロセスが起こされる可能
性があります。

Iput (kernel/fs.c:334) は、参照カウント (kernel/fs.c:357) をデクリメントすることで、inode
への C ポインタを解放します。これが最後の参照であれば、inode キャッシュのその
inode のスロットは空いており、別の inode に再利用することができます。

iputが，inodeに対するCポインタの参照がなく，inodeにリンクがない（どのディレクト
リにも存在しない）と判断した場合，inodeとそのデータブロックを解放しなければならない
。Iput は
itruncを呼び出してファイルを0バイトに切り詰め，データブロックを解放し，inodeの
タイプを0（未割り当て）に設定し，inodeをディスクに書き込みます（kernel/fs.c:339）
.

iputがinodeを解放する場合のロックプロトコルは、より詳細に検討する必要があります
。1つの危険性は、同時実行スレッドがこのinodeを使用するためにilockで待っているかも
しれず（例えば、ファイルを読んだり、ディレクトリをリストアップしたりするため）、
inodeがもはや割り当てられていないことに気づく準備ができないことです。これは、シ
ステムコールがキャッシュされたinodeへのリンクがなく、ip-
>refが1つである場合に、そのinodeへのポインタを取得する方法がないため、起こりえ
ません。その1つの参照とは、iputを呼び出したスレッドが所有する参照のことです。iput
がicache.lockクリティカルセクションの外で参照カウントが1であることをチェックして
いるのは事実ですが、その時点でリンクカウントは0であることがわかっているので、

どのスレッドも新しい参照を取得しようとはしません。もう1つの主な危険性は、ialloc の同時呼び出しが、iputが解放しているのと同じinodeを選択してしまうことです。これは 、iupdateがディスクを書き込み、inodeのタイプが0になった後にのみ起こります。割り当 てを行うスレッドは、inodeを読み書きする前に、inodeのスリープ・ロックを取得するの を丁寧に待ち、その時点でiputはそれを終了します。

iput()はディスクへの書き込みが可能です。これは、ファイルシステムを使用するシステムコール はすべて

ディスクへの書き込みは、システムコールがファイルへの参照を持つ最後のものである 可能性があるからです。read() の ように一見読み取り専用のように見えるコールでも、最終的にはiput()を呼び出すことに なります。つまり、読み取り専用のシステムコールであっても、ファイルシステムを使 用する場合は、トランザクションでラップしなければならないということです。

iput()とクラッシュの間には難しい相互作用があります。 iput()は、ファイルのリンクカウントがゼロになっても、すぐにはファイルを切り捨て ません。なぜなら、あるプロセスがまだメモリ内にinodeへの参照を保持しているかもし れないからです。しかし、最後のプロセスがファイルディスクリプターを閉じる前にク ラッシュが起こると

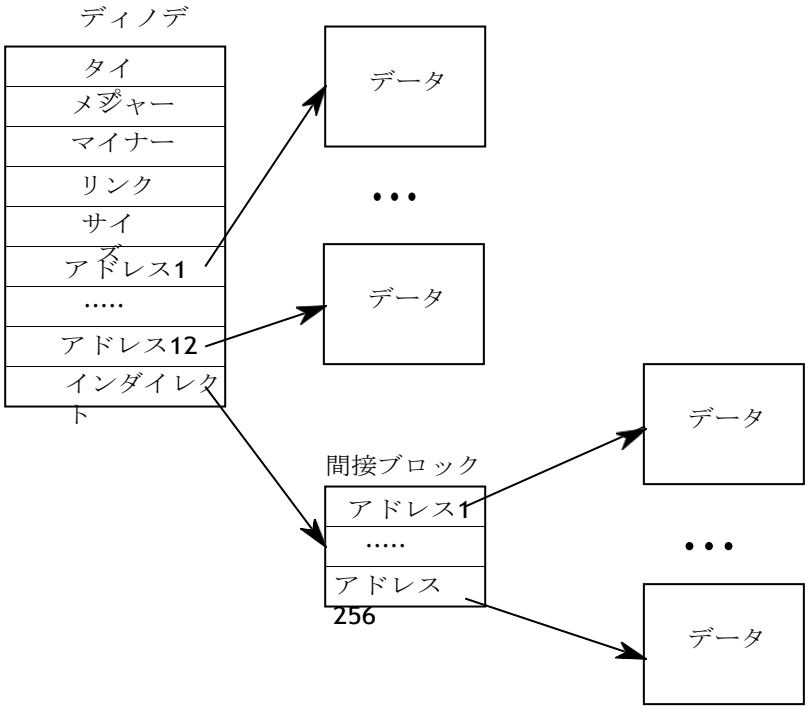


図7.3：ディスク上のファイルの表現。

の場合、そのファイルはディスク上に割り当てられていると表示されますが、そのファイルを指すディレクトリエントリはありません。

ファイルシステムはこのケースを2つの方法のいずれかで処理します。単純な方法は、再起動後のリカバリー時に、ファイルシステムがファイルシステム全体をスキャンして、割り当て済みのマークが付いているが、それを指すディレクトリエントリがないファイルを探します。そのようなファイルが存在すれば、それらのファイルを解放することができます。

2つ目の解決策は、ファイルシステムのスキャンを必要としません。この解決策では、ファイルシステムは、リンクカウントが0になったが参照カウントが0ではないファイルのinode
inumberをディスク（例えば、スーパーブロック）に記録します。参照カウントが0になったときにファイルシステムがそのファイルを削除した場合、そのinodeをリストから削除することでディスク上のリストを更新します。復旧時には、ファイルシステムはリスト内のすべてのファイルを解放する。

Xv6にはどちらの解決策も実装されていないため、inodeが使用されていないにもかかわらず、ディスク上に割り当てられていると表示されることがあります。つまり、時間が経つとxv6はディスクスペースを使い果たしてしまう危険性があるのです。

7.10　コードInodeの内容

ディスク上のinode構造体struct
dinodeには、サイズとブロック番号の配列が含まれています（図7.3参照）。inodeのデータは、dinodeのadrs配列に記載されたブロックにあります。データの最初のNDIRECTブロックは、配列の最初のNDIRECTエントリにリストされており、これらのブロックは

の*直接ブロック*です。次のNINDIRECTブロックのデータはinodeではなく、*間接ブロック*と呼ばれるデータブロックに記載されています。addrs配列の最後のエントリは、間接ブロックのアドレスを示しています。このように、ファイルの最初の12キロバイト（NDIRECT×BSIZE）はinodeに記載されているブロックから読み込むことができますが、次の256キロバイト（NINDIRECT×BSIZE）は間接ブロックを参照してからでないと読み込むことができません。これは、ディスク上では良い表現ですが、クライアントにとっては複雑な表現です。関数bmapは、後述するreadiやwriteiなどの高レベルのルーチンが表現を管理できるようにします。Bmapは、inode
ipのbn'thデータブロックのディスクブロック番号を返す。ipがまだそのようなブロックを持っていない場合、bmapはブロックを割り当てる。

関数bmap
(kernel/fs.c:379)は、最初のNDIRECTブロックがinode自体にリストアップされている(kernel/fs.c:384-388)という、簡単なケースを選ぶことから始めます。次のNINDIRECTブロックは、ip->addrs[NDIRECT]の間接ブロックに記載されています。Bmap は間接ブロックを読み込み(kernel/fs.c:395)、次にブロック内の右の位置からブロック番号を読み込む(kernel/fs.c:396)。ブロック番号がNDIRECT+NINDIRECTを超えると、bmapはパニックに陥ります。writeiには、これを防ぐためのチェックが入っています(kernel/fs.c:494)。

Bmapは、必要に応じてブロックを割り当てます。ip->addrs[]または間接エントリがゼロの場合、ブロックが割り当てられていないことを示す。bmap はゼロに遭遇すると、必要に応じて割り当てられる新しいブロックの番号に置き換える (kernel/fs.c:385-386) (kernel/fs.c:393-394)。

itruncは、ファイルのブロックを解放し、inodeのサイズをゼロにリセットします。Itrunc (kernel/fs.c:414) は、まず直接ブロックを解放し (kernel/fs.c:420-425)、次に間接ブロックに記載されているブロックを解放し (kernel/fs.c:430-433)、最後に間接ブロック自体を解放します (kernel/fs.c:435-436)。

Bmapでは、readiとwriteiがinodeのデータを簡単に取得できるようになっています。Readi (kernel/fs.c:460) は、オフセットとカウントがファイルの終端を超えていないことを確認することから始めます。ファイルの終わりを越えて開始した読み取りはエラーを返し（kernel/fs.c:465-466）、ファイルの終わりを越えて開始した読み取りは要求されたよりも少ないバイト数を返します（kernel/fs.c:467-468）。writei (kernel/fs.c:487) は readi と同じですが，次の 3
つの例外があります．すなわち，ファイルの終わりから始まるか，またはファイルの終わりを越える書き込みは，最大ファイルサイズまでファイルを成長させます (kernel/fs.c:494-495)．c:494-495），ループはデータを取り出す代わりにバッファにコピーする（kernel/fs.c:37），書き込みによってファイルが拡張された場合，writei
はそのサイズを更新しなければならない（kernel/fs.c:508-515）．

readi、writeiともに、ip->type ==

T_DEVのチェックから始まります。このケースは、データがファイルシステムに存在しない特殊なデバイスを扱うもので、ファイルディスクリプター層ではこのケースに戻ります。

関数 stati (kernel/fs.c:446) は、inode のメタデータを stat 構造体にコピーしますが、この stat 構造体は stat システムコールを介してユーザプログラムに公開されます。

7.11 コード：ディレクトリ・レイヤー

ディレクトリは、内部的にはファイルと同様に実装されています。ディレクトリのinodeはT_DIR型で，データは一連のディレクトリ・エントリです。各エントリは struct dirent (kernel/fs.h:56)で、名前とinode番号を含みます。名前は最大でもDIRSIZ(14)文字で、それより短い場合はNUL(0)バイトで終了します。inode番号が0のディレクトリエントリはフリーである。

関数 dirlookup (kernel/fs.c:531) は、ディレクトリを検索して、与えられた名前のエントリを探します。

見つかった場合は、対応するinodeへのポインタをロック解除して返し、呼び出し側が編集したい場合に備えて、ディレクトリ内のエントリのバイト・オフセットを*poff に設定します。dirlookup が正しい名前のエントリを見つけた場合、*poffを更新し、igetで取得したロックされていないinodeを返します。igetがロックされていないinodeを返すのは、Dirlookup のおかげです。呼び出し側はdpをロックしていますので、もしルックアップが.カレント・ディレクトリのエイリアスである
.を探していた場合、返す前にinodeをロックしようとすると、dp を再度ロックしようとし、デッドロックが発生します。(複数のプロセスや親ディレクトリのエイリアスである.を含む，より複雑なデッドロックのシナリオもありますが，問題は.だけではありません)呼び出し側は，dpをロック解除してからipをロックし，同時に1つのロックしか保持しないようにすることができます。

関数 dirlink (kernel/fs.c:558)
は、与えられた名前とインデックス番号を持つ新しいディレクトリエントリをディレクトリ
dp
に書き込みます。名前がすでに存在している場合、dirlinkはエラーを返します（kernel/fs.c:564-
568）。メインループは、ディレクトリエントリを読み込んで、未割り当てのエントリを探します。見つかった場合はループを早期に停止し（kernel/fs.c:542-
543）、offには利用可能なエントリのオフセットを設定します。そうでない場合は、offをdp->sizeに設定してループを終了します。いずれにしても、dirlinkはオフセットoffに書き込むことで、ディレクトリに新しいエントリを追加します(kernel/fs.c:578-581)。

7.12 コードパス名

パス名の検索には、パスの構成要素ごとにdirlookupを
連続して呼び出す必要があります。Namei (kernel/fs.c:665)
は、pathを評価し、対応するinodeを返します。関数nameiparentはその変種で，最後の要素の前で停止し，親ディレクトリのinodeを返し，それをコピーします.
は、最終的な要素を name
に変換します。どちらも一般化された関数namexを呼んで実際の作業を行います。

Namex (kernel/fs.c:630)
は、パスの評価がどこから始まるかを決めることから始めます。パスがスラッシュで始まっていればルートから、そうでなければカレントディレクトリから評価を開始します(kernel/fs.c:634-637)。次に、skipelemを使用して、パスの各要素を順に検討します（kernel/fs.c:639）。このループの各反復は、現在のinode ipでnameを検索しなければなりません。反復は、ipをロックし、それがディレクトリであるかどうかをチェックすることから始まります。もしそうでなければ、検索は失敗します(kernel/fs.c:640-644)。(ipをロックする必要があるのは、ip-

>typeが足元で変化する可能性があるからではなく、ilockが実行されるまで、ip->typeがディスクからロードされたことが保証されないからです)呼び出しが nameiparent であり、これが最後のパス要素である場合、ループは nameiparent の定義に従って早期に停止します。最後のパス要素は既に name にコピーされているので、namex はロックされていない ip を返すだけで済みます (kernel/fs.c:645-649)。最後に、ループはdirlookupを使ってパス要素を探し、ip = nextを設定して次の繰り返しに備えます（kernel/fs.c:650-655）。ループは、パス要素がなくなると、ipを返します。パス名で移動したディレクトリのinodeやディレクトリブロック（バッファキャッシュにない場合）を読み込むために、数回のディスク操作が必要になるかもしれません。Xv6では、あるカーネルスレッドによるnamexの呼び出しがディスクI/Oでブロックされても、別のパス名を検索する別のカーネルスレッドが同時に進行できるように慎重に設計されています。Namexはパス内の各ディレクトリを個別にロックしているので、異なるディレクトリを並行して進めることができます。

このような同時進行は、いくつかの課題をもたらします。たとえば、あるカーネルスレッドがパス名を検索している間に、別のカーネルスレッドがディレクトリのリンクを解除してディレクトリツリーを変更している場合があります。また、他のカーネルスレッドが削除したディレクトリを検索して、そのブロックが別のディレクトリやファイルに再利用されてしまう可能性もあります。

Xv6ではこのようなレースを回避しています。例えば、namexでdirlookup を実行すると、lookupスレッドがディレクトリのロックを保持しており、dirlookupはigetを使って取得したinodeを返します。Iget はinodeの参照カウントを増やします。dirlookupからinodeを受け取って初めて、namexはディレクトリのロックを解除する。ここで、別のスレッドがディレクトリからinodeをアンリンクしても、inodeの参照カウントがまだ0より大きいため、xv6はまだinodeを削除しません。

もう一つのリスクはデッドロックです。例えば、next は ". "を探すときにipと同じinodeを指しています。ipのロックを解除する前にnextをロックするとデッドロックになってしまいます。このデッドロックを避けるために、namexはnextのロックを取得する前にディレクトリのロックを解除します。ここでも、igetとilockの分離が重要であることがわかります。

7.13 ファイル記述子層

Unixインターフェースの素晴らしい点は、コンソールやパイプなどのデバイス、そしてもちろん実際のファイルを含め、Unixのほとんどのリソースがファイルとして表現されていることです。ファイル記述子層は、この統一性を実現する層です。

Xv6では、第1章で説明したように、各プロセスに独自のオープンファイルのテーブル、つまりファイルディスクリプターが与えられます。各オープンファイルは構造体ファイル（kernel/file.h:1）で表されます。構造体ファイルは、inodeまたはpipeのラッパーにI/Oオフセットを加えたものです。openを呼び出すたびに、新しいオープンファイル（新しい構造体ファイル）が作成されます。複数のプロセスが同じファイルを個別にオープンした場合、インスタンスごとにI/Oオフセットが異なります。一方、1つのオープン・ファイル（同一の構造体ファイル）が、1つのプロセスのファイル・テーブルに複数回表示されたり、複数のプロセスのファイル・テーブルに表示されたりすることがあります。これは、あるプロセスがopenを使ってファイルをオープンした後、dupを使ってエイリアスを作成したり、forkを使って子プロセスと共有した場合に起こります。参照カウントは、特定のオープンファイルへの参照の数を追跡します。ファイルは、読み取り用、書き込み用、またはその両方が可能です。readableおよびwritableフィールドはこれを追跡します。

システム内のすべてのオープンファイルは、グローバルファイルテーブルであるftableに保持されます。ファイルテーブルには，ファイルの割り当て（filealloc），複製参照の作成（filedup），参照の解放（fileclose），データの読み書き（fileread，filewrite）を行う関数があります。

最初の3つは、今ではおなじみの形式です。filealloc (kernel/file.c:30) は、参照されていないファイル (f->ref == 0) を求めてファイルテーブルをスキャンし、新しい参照を返します。 filedup (kernel/file.c:48) は参照カウントを増やし、fileclose (kernel/file.c:60) は減らします。ファイルの参照カウントが0になると、filecloseは、タイプに応じて、基

礎となるパイプまたはinodeを解放します。

関数 `filestat`、`fileread`、`filewrite` は、ファイルの `stat`、`read`、`write` 操作を実装します。`filestat` (kernel/file.c:88) は inode に対してのみ許可され、`stati` を呼び出します。`fileread`と`filewrite`は、操作がオープンモードで許可されているかどうかをチェックし、パイプまたはinodeの実装のいずれかにコールを通します。ファイルがinodeを表している場合、`fileread`と`filewrite`はI/Oオフセットを操作のためのオフセットとして使用し、それを進めます(kernel/file.c:122-123)(kernel/file.c:153-154)。パイプにはオフセットの概念がありません。inode関数では、呼び出し側がロックを処理する必要があることを思い出してください(kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166)。inodeのロックには、読み取りオフセットと書き込みオフセットがアトミックに更新されるという便利な副次効果があり、同じファイルに同時に複数の書き込みをしてもお互いのデータを上書きすることはありません。

彼らの書き込みが交錯してしまうことがある。

7.14 コードシステムコール

下位レイヤーが提供する機能を使えば、ほとんどのシステムコールの実装は簡単です（(kernel/sysfile.c)参照）。しかし、いくつかのコールについては、詳しく見ていく必要があります。

関数sys_linkおよびsys_unlinkは、ディレクトリを編集し、inodeへの参照を作成または削除します。これらは、トランザクションを使うことの利点を示すもう1つの良い例です。Sys_link (kernel/sysfile.c:120)はまず、引数である2つの文字列oldとnewを取得します(kernel/sysfile.c:125)。oldが存在し、かつディレクトリではないと仮定して (kernel/sysfile.c:129-132)、sys_linkはそのip->nlinkカウントをインクリメントします。次にsys_linkはnameiparentを呼んでnewの親ディレクトリと最終パス要素を見つけ(kernel/sysfile.c:145)、oldのinodeを指す新しいディレクトリエントリを作成します (kernel/sysfile.c:148)。新しい親ディレクトリは、既存のinodeと同じデバイス上に存在していなければなりません。inode番号は単一のディスク上でのみ一意に意味を持ちます。このようなエラーが発生した場合、sys_linkは過去に戻ってip->nlinkをデクリメントしなければなりません。

トランザクションは、複数のディスクブロックを更新する必要があるため、実装が簡素化されますが、実行する順番を気にする必要はありません。すべて成功するか、成功しないかのどちらかです。例えば、トランザクションがない場合、リンクを作成する前にip->nlinkを更新すると、ファイルシステムが一時的に安全でない状態になり、その間にクラッシュが発生すると大混乱に陥ります。トランザクションがあれば、このような心配をする必要はありません。

Sys_link は、既存の inode に対して新しい名前を作成します。関数create (kernel/sysfile.c:242)は、新しいinodeに新しい名前を付けます。O_CREATEフラグ付きのopenは新しい普通のファイルを作成し、mkdirは新しいディレクトリを作成し、mkdevは新しいデバイスファイルを作成するという、3つのファイル作成システムコールを一般化したものです。sys_linkと同様に、createは親ディレクトリのinodeを得るためにnameiparentを呼び出すことから始まります。次にdirlookupを呼び出して、その名前が既に存在するかどうかをチェックします(kernel/sysfile.c:252)。名前が存在する場合、createの動作はどのシステムコールに使用されているかに依存します：openはmkdirやmkdevとは異なるセマンティクスを持っています。openがmkdirやmkdevとは異なるセマンティクスを持っているからです。createがopenの代わりに使われていて(type == T_FILE)、存在する名前自体が通常のファイルであれば、openは成功として扱い、createも成功となります(kernel/sysfile.c:256)。それ以外の場合はエラーになります(kernel/sysfile.c:257-258)。名前がまだ存在していない場合、createはiallocで新しいinodeを割り当てます(kernel

/sysfile.c:261)。新しいinodeがディレクトリの場合、createはそれを.で初期化して
...
エントリが表示されます。最後に、データが正しく初期化されたので、createはそれを親ディレクトリにリンクすることができます(kernel/sysfile.c:274)。createはsys_linkと同様に、ipとdpという2つのinodeロックを同時に保持します。システム内の他のプロセスがipのロックを保持したままdpをロックしようとすることはないので、inode ipは新たに割り当てられたものであるため、デッドロックの可能性はありません。

createを使うと、sys_open、sys_mkdir、sys_mknodを簡単に実装することができます。Sys_open (kernel/sysfile.c:287)は、新しいファイルを作成することはその機能のほんの一部に過ぎないので、最も複雑です。open に O_CREATEフラグが渡された場合、create (kernel/sysfile.c:301)を呼び出します。そうでなければ、namei (kernel/sysfile.c:307)を呼び出します。createはロックされたinodeを返しますが、nameiはそうではないので、sys_openは自分でinodeをロックしなければなりません。これは、ディレクトリが書き込みではなく読み込みのためだけにオープンされているかどうかをチェックするのに便利な場所を提供します。一方的にinodeが得られたと仮定して、sys_open は
ファイルとファイルディスクリプタを確保し(kernel/sysfile.c:325)、ファイルを埋めます(kernel/sysfile.c:337-)。

342).なお，部分的に初期化されたファイルは，現在のプロセスのテーブルにしかないので，他のプロセスはアクセスできない。

第6章では、ファイルシステムができる前のパイプの実装について検討しました。関数sys_pipeは、パイプペアを作成する方法を提供することで、その実装をファイルシステムに接続します。その引数は、2つの整数を格納する空間へのポインタで、そこに2つの新しいファイルディスクリプタを記録します。そして、パイプを割り当て、ファイルディスクリプタを設置します。

7.15 リアルワールド

現実のオペレーティングシステムにおけるバッファキャッシュは、xv6のものよりもはるかに複雑ですが、キャッシュとディスクへのアクセスの同期という同じ2つの目的を果たしています。Xv6のバッファキャッシュは、V6と同様にシンプルなLRU（Lest Recently Used）退避ポリシーを採用していますが、より複雑なポリシーを実装することも可能で、それぞれがワークロードによっては適していますが、他のワークロードではあまり適していません。より効率的なLRUキャッシュでは、リンクリストを廃止し、ルックアップにはハッシュテーブルを、LRU退避にはヒープを使用します。最近のバッファキャッシュは、メモリマップドファイルをサポートするために仮想メモリシステムと統合されています。

Xv6のログシステムは非効率的です。コミットは、ファイルシステムのシステムテンコールと同時に行うことはできません。ブロック内の数バイトしか変更されていない場合でも、システムはブロック全体をログに記録します。ブロックごとに同期したログ書き込みを行いますが、それぞれのログ書き込みにはディスク全体の回転時間が必要になる可能性があります。本物のログシステムは、これらの問題をすべて解決します。

クラッシュリカバリーを提供する方法は、ロギングだけではありません。初期のファイルシステムでは、再起動時にスキャベンジャー（例えば、UNIXのfsckプログラム）を使用して、すべてのファイルとディレクトリ、ブロックとinodeの空きリストを調べ、不整合を探して解決していました。大規模なファイルシステムではスキャベンジングに数時間かかることもあり、元のシステムコールがアトミックになるような方法で不整合を解決できない場合もあります。ログからのリカバリははるかに速く、クラッシュ時にもシステムコールをアトミックにすることができます。

Xv6では、初期のUNIXと同じinodeとディレクトリの基本的なディスク上のレイアウトを採用しています。BSDのUFS/FFSとLinuxのext2/ext3は基本的に同じデータ構造を使用しています。ファイルシステムのレイアウトで最も非効率的な部分はディレクトリで、検索のたびにすべてのディスクブロックをリニアスキャンする必要があります。これは、ディレクトリが数個のディスクブロックしかない場合は合理的ですが、多くのファイ

ルを保持するディレクトリではコストがかかります。Microsoft WindowsのNTFS、Mac OS XのHFS、SolarisのZFSなどでは、ディレクトリをディスク上のバランスのとれたブロックツリーとして実装しています。これは複雑ですが、対数時間のディレクトリ検索を保証します。

Xv6はディスク障害に対してナイーブで、ディスク操作に失敗するとxv6はパニックに陥ります。これが妥当であるかどうかは、ハードウェアに依存します。オペレーティングシステムが、ディスク障害を隠すために冗長性を使用する特別なハードウェアの上に置かれている場合、おそらくオペレーティングシステムが障害を目にする頻度は非常に低く、パニックを起こしても問題ないでしょう。一方、普通のディスクを使用しているオペレーティングシステムでは、障害が発生することを想定し、1つのファイルのブロックが失われても、ファイルシステムの残りの部分の使用に影響を与えないように、より優雅に処理する必要があります。

Xv6では、ファイルシステムが1つのディスクデバイスに収まり、サイズが変わらないことが要求されています。大規模なデータベースやマルチメディアファイルの増加に伴い、ストレージの必要性がますます高まってきているため、オペレーティングシステムでは

1つのファイルシステムに1つのディスク」というボトルネックを解消する方法を模索しています。基本的なアプローチは、多くのディスクを1つの論理ディスクにまとめることです。RAIDのようなハードウェアソリューションが最も一般的ですが、最近のトレンドは、この論理をできるだけソフトウェアで実装することです。これらのソフトウェアは、ディスクを追加・削除することで、論理デバイスを拡張・縮小するなど、豊富な機能を備えているのが一般的だ。xv6で採用されている固定サイズのinodeブロックの配列は、このような環境ではうまく機能しません。xv6で採用されている固定サイズのinodeブロックの配列は、そのような環境ではうまく機能しません。ディスク管理とファイルシステムを分離することは、最もクリーンなデザインかもしれませんが、両者の間の複雑なインターフェースのために、SunのZFSのように両者を組み合わせるシステムもあります。

Xv6のファイルシステムには、スナップショットや増分バックアップのサポートなど、最新のファイルシステムにはない機能がたくさんあります。

現代のUnixシステムでは、名前付きパイプ、ネットワーク接続、リモートアクセスされたネットワークファイルシステム、/procなどの監視・制御インターフェイスなど、多くの種類のリソースをディスク上のストレージと同じシステムコールでアクセスすることができます。これらのシステムでは、xv6のfileadとfilewriteのif文の代わりに、一般的に各オープンファイルに操作ごとの関数ポインタのテーブルを与え、関数ポインタを呼び出して、そのinodeの呼び出しの実装を呼び出します。ネットワーク・ファイル・システムやユーザー・レベル・ファイル・システムは、これらの呼び出しをネットワークRPCに変える関数を提供し、応答を待ってから戻ります。

7.16 エクササイズ

1. なぜバロックではパニックが起きるのか？xv6は回復できるのか？

2. なぜiallocでパニックになるのか？xv6は回復できるのか？

3. なぜ filealloc はファイルが足りなくなってもパニックにならないのですか？なぜこの方が一般的で、扱う価値があるのでしょうか？

4. ipに対応するファイルが、sys_linkの間で他のプロセスによってアンリンクされたとします。のiunlock(ip)とdirlinkへの呼び出しを行っています。リンクは正しく作成されますか？どうしてでしょうか，それともどうしてでしょうか？

5. createは、成功するために必要な4つの関数呼び出し（iallocに1つ、dirlinkに3つ）を行います。成功しなければ，createはpanicを呼び出します．なぜこれが許されるのでしょうか？なぜ4つの呼び出しのどれもが失敗してはいけないのでしょうか

？

6. sys_chdirはiput(cp->cwd)の前にiunlock(ip)を呼んでいるので、 cp->cwd を ロックしようとするかもしれませんが、iunlock(ip) を iputの後まで延期してもデッドロックは起こりません。なぜだろう？

7. lseekシステムコールの実装。また、lseekをサポートするには lseekがf->ip->sizeを超えて設定した場合に、ファイルの穴を0で埋めるfilewrite。

8. open に O_TRUNC と O_APPEND を追加し、シェルで > と >> 演算子が動作するようにしました。

9. シンボリックリンクをサポートするようにファイルシステムを変更する。

10. 名前付きパイプをサポートするようにファイルシステムを変更する。

11. mmapをサポートするために、ファイルとVMシステムを変更する。

第8章

コンカレンシーの再

検討

並列性能が高く、同時実行にもかかわらず正しく、かつアンダースタンダブルなコードを同時に得ることは、カーネル設計の大きな課題です。ロックを単純に使用することが正しさへの最良の道ですが、常に可能とは限りません。この章では、xv6がロックを複雑な方法で使用することを余儀なくされた例と、xv6がロックに似た技術を使用するがロックを使用しない例を紹介します。

8.1 ロッキングパターン

キャッシュされたアイテムは、しばしばロックするのが難しいものです。例えば、ファイルシステムのブロックキャッシュ(kernel/bio.c:26)には、最大 NBUF
個のディスクブロックのコピーが保存されています。そうしないと、異なるプロセスが同じブロックの異なるコピーに対して矛盾した変更を行う可能性があるからです。キャッシュされた各ブロックは、buf 構造体に格納されます
(kernel/buf.h:1)。buf構造体にはロックフィールドがあり、一度に1つのディスクブロックを使用するのは1つのプロセスのみであることを保証します。しかし、ロックフィールドだけでは十分ではありません。あるブロックがキャッシュに全く存在しない場合に、2つのプロセスが同時に使用したいとしたらどうでしょうか。ブロックがまだキャッシュされていないので、構造体bufは存在せず、ロックするものもありません。Xv6 では、追加のロック
(bcache.lock) をキャッシュされたブロックの ID
セットに関連付けることで、この状況に対処しています。ブロックがキャッシュされているかどうかをチェックしたり (bget (kernel/bio.c:58)
など)、キャッシュされているブロックのセットを変更する必要があるコードは、bcache.lock

を保持する必要があります。コードが必要なブロックと構造体bufを見つけた後、bcache.lock
を解放して特定のブロックだけをロックすることができます。これはよくあるパターンで、アイテムのセットに対して1つのロック、さらにアイテムごとに1つのロックを行います。

通常、ロックを取得したのと同じ関数がロックを解放します。しかし、より正確な見方をすると、ロックは原子的に見える必要のあるシーケンスの開始時に取得され、そのシーケンスが終了したときに解放されるということになります。シーケンスの開始と終了が異なる関数、異なるスレッド、異なるCPUで行われる場合、ロックの取得と解放は同じように行われなければなりません。ロックの機能は、他の利用者を待たせることであり、データの一部を特定のエージェントに固定することではありません。その一例が yield の acquire (kernel/proc.c:500)で、これはアクワイヤリングプロセスではなく、スケジューラースレッドでリリースされます。別の例としては、ilock (kernel/fs.c:290) の acquiresleep があります。このコードは、ディスクの読み取り中にスリープすることが多く、別の CPU で目を覚ます可能性があるため、ロックの取得と解放が別の CPU で行われる可能性があります。

オブジェクトに埋め込まれたロックによって保護されているオブジェクトを解放することは、ロックを所有しているだけでは解放が正しく行われることを保証できないため、デリケートなビジネスです。問題となるのは、他のスレッドがそのオブジェクトの取得を待っている場合です。オブジェクトを解放すると、埋め込まれているロックも暗黙のうちに解放されてしまうため、待っているスレッドが誤動作してしまいます。一つの解決策は、オブジェクトへの参照がいくつあるかを追跡し、最後の参照がなくなったときにのみオブジェクトを解放することです。その例として、pipeclose (kernel/pipe.c:59)を参照してください。pi->readopenおよびpi->writeopenは、パイプが参照しているファイル記述子があるかどうかを追跡します。

8.2 ロック式パターン

xv6では、オブジェクトが割り当てられており、解放や再利用されるべきではないことを示すために、多くの場所で参照カウントやフラグを一種のソフトロックとして使用しています。プロセスのp->stateがこのように動作し、file、inode、buf構造体の参照カウントも同様です。いずれの場合も、ロックによってフラグや参照カウントが保護されますが、オブジェクトが早期に解放されるのを防ぐのは後者です。

ファイルシステムでは、コードが通常のロックを使用した場合に発生するデッドロックを回避するために、構造体のinode参照カウントを、複数のプロセスが保持できる一種の共有ロックとして使用しています。例えば、namexのループ（kernel/fs.c:630）は、パス名の各コンポーネントで指定されたディレクトリを順番にロックします。なぜなら、複数のロックを保持していると、パス名にドットが含まれている場合（例：a/./b）、自分自身とデッドロックを起こす可能性があるからです。また、ディレクトリと...を含む同時検索でもデッドロックする可能性があります。第7章で説明したように、この解決策は、ループがディレクトリのinodeを次の反復に持ち越すことで、その参照カウントを増加させますが、ロックはしません。

データアイテムの中には、異なる時期に異なるメカニズムで保護されているものがあります。また、明示的なロックではなく、xv6コードの構造によって暗黙的に同時アクセスから保護されている場合もあります。例えば、物理ページが空いているときは、kmem.lockによって保護されています（kernel/kalloc.c:24）。その後、そのページがパイプとして割り当てられると(kernel/pipe.c:23)、別のロック(埋め込まれたpi->lock)で保護されます。そのページが新しいプロセスのユーザーメモリに再割り当てされた場合、ロックではまったく保護されません。代わりに、アロケータがそのページ

を（解放されるまで）他のプロセスに渡さないという事実によって、同時アクセスから保護されます。新しいプロセスのメモリの所有権は複雑です。まず親がフォークでメモリを割り当てて操作し、次に子がそれを使用し、（子が終了した後）親が再びメモリを所有し、それを`kfree`に渡します。ここには2つの教訓があります。データ・オブジェクトは、その寿命の異なる時点で、異なる方法で同時実行から保護される可能性があり、その保護は、明示的なロックではなく、暗黙的な構造の形をとるかもしれません。ロックのような例としては、`mycpu()` (ker-nel/proc.c:66)を呼び出す際に割り込みを無効にする必要があることが挙げられます。割り込みを無効にすることで、呼び出したコードは、タイマのイン/アウトに対してアトミックになります。

テラプトは、コンテキストスイッチを強制的に行い、プロセスを別のCPUに移動させることができます。

8.3 ロックは一切なし

xv6では、ロックを一切使用せずに変更可能なデータを共有する箇所がいくつかあります。一つはスピンロックの実装で、RISC-Vのアトミック命令はロックに依存していると見ることもできます。

はハードウェアで実装されています。もう一つは、main.c (kernel/main.c:7)のstarted変数で、CPU zeroがxv6の初期化を終えるまで他のCPUが走らないようにするために使われています。このvolatile は、コンパイラが実際にロード命令とストア命令を生成することを保証します。3つ目の例は proc.c (kernel/proc.c:383) (kernel/proc.c:291) における p->parent の使用で、適切にロックすればデッドロックが可能ですが、他のプロセスが同時に p->parent を 変更できないことは明らかなようです。4つ目の例はp->killedで、これはp->lockを保持している間に設定されますが(kernel/proc.c:596)、保持していない状態でチェックされます(kernel/trap.c:56)。

Xv6には、あるCPUやスレッドがデータを書き込み、別のCPUやスレッドがそのデータを読み取るケースがありますが、そのデータを保護するための特定のロックはありません。例えば、forkでは、親が子のユーザーメモリページを書き込み、子（別のスレッド、おそらく別のCPU）がそのページを読みますが、そのページを明示的に保護するロックはありません。親の書き込みが終わるまで子は実行を開始しないので、これは厳密にはロックの問題ではありません。メモリバリアがなければ、あるCPUが他のCPUの書き込みを見ることができないからです。これは、潜在的なメモリ順序問題です（第5章参照）。しかし、親はロックを解放し、子は起動時にロックを取得しているので、アクワイアとリリースのメモリバリアにより、子のCPUが親の書き込みを見ることができます。

8.4 平行法

ロックは主に、正しさのために並列性を抑制するものです。性能も重要なので、カーネル設計者は、正しさと優れた並列性の両方を達成する方法でロックを使用する方法を考えなければならないことがよくあります。xv6はハイパフォーマンスのために体系的に設計されているわけではありませんが、xv6のどの操作が並列で実行でき、どの操作がロックで衝突するかを考慮する価値はあります。

xv6のパイプは、かなり優れた並列処理の例です。各パイプには独自のロックがあり、異なるプロセスが異なるCPUで異なるパイプを並行して読み書きできるようになっています。ただし、あるパイプについては、書き手と読み手がお互いにロックを解除するのを待たなければならず、同じパイプを同時に読み書きすることはできません。また、空のパイプからの読み出し（または満杯のパイプへの書き込み）はブロックしなければなりませんが、これはロック方式によるものではありません。

コンテキストスイッチングはより複雑な例です。それぞれの CPU で実行される 2 つのカーネルスレッドは、同時に yield、sched、swtch を呼び出すことができ、これらの呼び出しは並行して実行されます。スレッドはそれぞれロックを保持していますが、異なるロックなので、お互いに待つ必要はありません。し

かし、スケジューラに入ると、2つのCPUがプロセスのテーブルを検索してRUNNABLEなものを探している間に、ロックで衝突する可能性があります。つまり、xv6はコンテキストスイッチの際に複数のCPUからパフォーマンス上のメリットを得られる可能性がありますが、おそらくそれほどではないでしょう。

別の例として、異なるCPUの異なるプロセスからforkを同時に呼び出す場合があります。これらの呼び出しは、pid_lockやkmem.lock、そしてUNUSEDプロセスのプロセステーブル検索に必要なプロセスごとのロックのために、お互いに待たなければならないかもしれません。一方で、2つのフォークプロセスは、ユーザーメモリページのコピーやページテーブルページのフォーマットを完全に並行して行うことができます。

上記の各例のロック方式は、特定のケースで並列性能を犠牲にしています。いずれの場合も，より精巧な設計を行えば，より多くの並列性を得ることができます．その価値があるかどうかは，関連する操作がどのくらいの頻度で呼び出されているか，どのくらいの時間で

競合するロックが保持されている間にコードが費やす時間、同時にいくつのCPUが競合する操作を実行する可能性があるか、コードの他の部分がより制限的なボトルネックになっていないかなどです。あるロック方式がパフォーマンスの問題を引き起こす可能性があるかどうか、あるいは新しいデザインが著しく優れているかどうかを推測するのは難しいため、現実的なワークロードでの測定が必要になることがあります。

8.5 エクササイズ

1. xv6のパイプの実装を変更し、同じパイプへの読み込みと書き込みを異なるコア上で並行して実行できるようにしました。

2. xv6の`scheduler()`を変更し、異なるコアが同時に実行可能なプロセスを探している場合にロックの競合を減らしました。

3. xv6の`fork()`の直列化の一部を削除しました。

第9章

まとめ

このテキストでは、xv6という1つのオペレーティングシステムを1行ずつ学ぶことで、オペレーティングシステムの主要なアイデアを紹介しました。いくつかのコードラインは主要なアイデアの本質を体現しており(コンテキストスイッチング、ユーザとカーネルの境界、ロックなど)、それぞれのラインが重要です。他のコードラインは、特定のオペレーティングシステムのアイデアをどのように実装するかを説明しており、異なる方法で簡単に実現できます(スケジューリングのためのより良いアルゴリズム、ファイルを表現するためのより良いディスク上のデータ構造、同時トランザクションを可能にするためのより良いログの記録など。)。すべてのアイデアは、大成功を収めたシステムコールインターフェースであるUnixインターフェースの文脈で説明されましたが、これらのアイデアは他のオペレーティングシステムの設計にも引き継がれています。

参考文献

[1] The RISC-V instruction set manual: privileged architecture . https://content. riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10. pdf, 2017.

[2] The RISC-V instruction set manual: user-level ISA. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2017.

[3] ハンス-J・ボエームスレッドはライブラリとして実装できません。*ACM PLDI Conference*, 2005.

[4] Edsger Dijkstra.Cooperating sequential processes. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html, 1965.

[5] Brian W. Kernighan.*The C Programming Language*.Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[6] ドナルド・クヌース*Fundamental Algorithms.コンピュータ・プログラミングの極意。(第2版)*, 第1巻.1997.

[7] ジョン・ライオンズ*UNIXの解説 第6版*.Peer to Peer Communications, 2000.

[8] マーティン・マイケルとダニエル・デュリッチNS16550A:UARTのデザインとアプリケーションの考察」 http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0491.pdf, 1987.

[9] デビッド・パターソンとアンドリュー・ウォーターマン。*The RISC-V Reader: an open architecture Atlas*.ストロベリーキャニオン、2017年

[10] Dennis M. Ritchie, Ken Thompson.UNIXのタイムシェアリングシステム。*Commun.ACM*, 17(7):365-375, 1974年7月.

インデックス