xv6: シンプルで Unix 風な
教育用オペレーティングシステム

Russ Cox　　Frans Kaashoek　　Robert Morris

菅原 健（訳）

2020 年 7 月 23 日

目次

序文と謝辞

この文書は，オペレーティングシステムの講義用テキストのドラフトです．xv6 というカーネルを例題にして，オペレーティングシステムの主なコンセプトを説明します．xv6 は，Dennis Ritchie と Ken Thompson による Unix Version 6 (v6) [10] の再実装です．xv6 は，v6 の構造とスタイルをゆるく踏襲しますが，ANSI C [5] で記述され，マルチコアの RISC-V [9] で動作します．

このテキストは，xv6 のソースコードと一緒に読むことを想定しています．“John Lions’s Commentary on UNIX 6th Edition” [7] に触発されたアプローチです．xv6 手を動かす実習課題を含む v6 と xv6 オンライン教材へのリンクは，`https://pdos.csail.mit.edu/6.828` を見てください．

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講義である 6.828 で利用しました．xv6 に直接的・間接的に貢献してくれた教員，ティーチングアシスタント，そして 6.828 の受講生に感謝します．Austin Clements と Nickolai Zeldovich には特に感謝をします．最後に，テキストのバグや改善のための助言をメールを送ってくれた以下の人たちに感謝します: 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.

間違いを見つけたり，改善のための提案があれば，Frans Kaashoek と Robert Morris (kaashoek,rtm@csail.mit.edu) にメールしてください．

第 1 章

オペレーティングシステムのインタフェース

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 1 台のコンピュータを共有することと，ハードウェア単独よりも便利なサービスを提供することにあります．オペレーティングシステムは下層にあるハードウェアを管理・抽象化します．そうすることで，たとえば，ワードプロセッサがディスクハードウェアの種類を気にしなくてよくなります．また，複数のプログラムでハードウェアを共有することで，それらを同時に実行することができます．また，プログラム同士がデータを共有したり一緒に働いたりするための，管理された方法を提供します．

オペレーティングシステムは，あるインタフェースを介して，ユーザープログラムにサービスを提供します．良いインタフェースを設計するのは難しいことが分かっています．単純かつ限定的なインタフェースは実装のしやすさで優れていますが，洗練されたたくさんの機能を提供したいという誘惑もあります．両立するためのトリックは，少数のメカニズムにしか依存しないインタフェースを設計し，それらの組み合わせにより多様性を実現することです．

この本は，オペレーティングシステムの概念を学ぶための実例として，あるオペレーティングシステムを用います．それは xv6 と言い，Ken Thompson と Dennis Ritchie による Unix オペレーティングシステム [10] の基本的なインタフェースを提供するとともに，Unix の内部設計を模倣しています．Unix は限定されたインタフェースを提供しますが，それらは巧みに組み合わせることができ，驚くほどの多様さを実現することができます．このインタフェースは大成功しており，現代的なオペレーティングシステムである BSD, Linux, Mac OS X, Solaris, また程度の差はありますが Microsoft Windows までもが Unix 的なインタフェースを持っています．xv6 を理解することは，そのようなオペレーティングシステムを理解するよいスタート地点になるはずです．

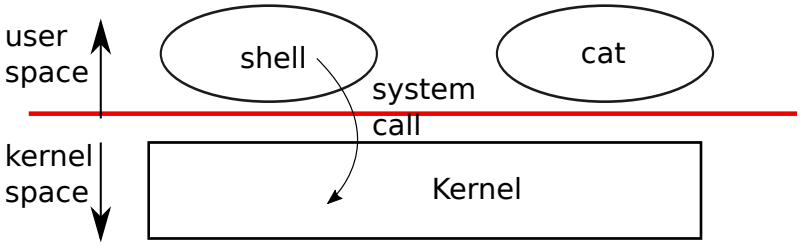

図 1.1　カーネルと 2 つのユーザプロセス

図 1.1 に示すように，xv6 は古典的なカーネル，すなわち実行中のプログラムにサービスを提供する特殊なプログラムのかたちをとります．実行中の各プログラム（プロセスと呼びます）は命令列を含むメモリ，データ，およびスタックを持ちます．命令列は，プログラムの計算を実現します．データは，計算が対象とする変数のことです．スタックはプログラムのプロシージャ呼び出し (procedure call) を管理します．

プロセスがカーネルのサービスを呼び出すときは，オペレーティングシステムのインタフェースを介してプロシージャ呼び出しを行います．そのようなプロシージャはシステムコールと呼ばれます．システムコールはカーネルに入り，サービスを実行してリターンします．すなわちプロセスは，ユーザ空間とカーネル空間を行ったり来たりします．

カーネルは，CPU が提供するハードウェア的な保護メカニズムを利用して，ユーザ空間で実行する各プロセスが，固有のメモリにしかアクセスできないようにします．そのような保護を実現するには，ハードウェア的な特権を持つカーネルが必要です．ユーザプログラムにはそのような特権がありません．ユーザプログラムがシステムコールを発行すると，ハードウェアが特権レベルを昇格させ，あらかじめ設定しておいたカーネルの関数を実行します．

カーネルが提供するシステムコールが，ユーザプログラムから見えるインタフェースです．xv6 カーネルは Unix カーネルの伝統的なシステムコールの一部を提供します．図 1.2 は xv6 の全てのシステムコールです．

この章の続きでは，xv6 の提供するサービスの概要を述べます．すなわち，プロセス，メモリ，ファイルディスクリプタ，パイプ，およびファイルシステムです．それらをコードと合わせて説明するとともに，Unix 的システムにおける重要なユーザインタフェースであるシェルが，どのようにそれらを利用するか議論します．シェルがシステムコールを利用する様は，システムコールがいかに繊細に設計されたかをよく示しています．

シェルはありふれたプログラムで，ユーザが入力したコマンドを読み込み，それを実行します．シェルがユーザプログラムであり，カーネルの一部ではないという

システムコール	説明
`fork()`	プロセスを作る.
`exit(xstatus)`	現在のプロセスを終了. `xstatus` で成否を示す.
`wait(*xstatus)`	子プロセスの終了を待ち，終了コードを `xstatus` にコピーする.
`kill(pid)`	`pid` 番のプロセスを終了する.
`getpid()`	現在のプロセスのプロセス ID (PID) をリターンする.
`sleep(n)`	`n` クロック刻みのあいだスリープする.
`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)`	ファイルを開く. `flags` は読み書きを示す.
`read(fd, buf, n)`	開いているファイルから `n` バイト読み，`buf` に入れる.
`write(fd, buf, n)`	開いているファイルに `n` バイト書き込む.
`close(fd)`	開いているファイルを表す `fd` を解放する.
`dup(fd)`	`fd` を複製する.
`pipe(p)`	パイプを作り，そのファイルディスクリプタ (fd) を `p` に入れる.
`chdir(dirname)`	カレントディレクトリを変更する.
`mkdir(dirname)`	新しいディレクトリを作る.
`mknod(name, major, minor)`	デバイスファイルを作る.
`fstat(fd)`	開いているファイルに関する情報をリターンする.
`link(f1, f2)`	ファイル `f1` の別の名前 (`f2`) を作る.
`unlink(filename)`	ファイルを消去する.

図 1.2　xv6 のシステムコール

事実は，システムコールインターフェースの力を示します．シェルは決して特別なプログラムでは無いのです．つまり，シェルは簡単に取替えることができます．現代的な Unix システムでは，さまざまなシェルを選ぶことができて，それらは独自のユーザインタフェースとスクリプティング機能を持っています．xv6 シェルは Unix Bourne シェルのエッセンスのみを取り出した単純な実装です．実装は(user/sh.c:1) にあります.

1.1　プロセスとメモリ

xv6 のプロセスは，ユーザ空間のメモリ（命令列，データ，およびスタック）と，カーネルだけが見ることができるプロセス固有の状態を持ちます．xv6 はプロセス間でタイムシェア (*Time share*) を行います．すなわち，実行待ちをしているプロセスを，利用可能な（複数の）CPU に対して透過的に切り替えます．プロセスが実行状態でないとき，xv6 は CPU レジスタを退避しておき，実行するときに復元します．カーネルは各プロセスに対し，プロセス ID （あるいは `pid`）を割り当てます.

プロセスは，`fork` システムコールを利用することで，新しいプロセスを生むことができます．`fork` は呼び出し元（親プロセスと呼びます）とまったく同一のメモリ

を持つ新しいプロセス（子プロセスと呼びます）を作ります．fork は，親プロセスと子プロセスの両方にリターンします．親プロセスにおいて，fork の返値は子プロセスの pid です．一方，子プロセスにおける fork の返値は 0 です．たとえば，C 言語で書かれた次のプログラムを考えてみましょう [5]:

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

exit システムコールは呼んだプロセスの実行を中止し，メモリや開いている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ます．慣習的に，0 が成功, 1 が失敗を表します．wait システムコールは，終了した子プロセスの pid をリターンするとともに，子プロセスの終了ステータスを，wait に渡されたアドレスへコピーします．もし，どの子プロセスも終了しない場合，wait はどれかが終了するまで待ちます．もし，親プロセスが子プロセスの終了コードに興味がない場合は，引数に 0 を渡します．

上記の例において，

```
parent: child=1234
child: exiting
```

のような出力が順不同で現れます．親プロセスと子プロセスのどちらが先に printf を呼んだかによって順序が決まります．子プロセスが終了したあと，親プロセスの wait はリターンし，その結果親プロセスは次を出力します:

```
parent: child 1234 is done
```

初期状態の子プロセスは，親プロセスとメモリ内容は同じですが，別のメモリとレジスタを使って実行を行います．すなわち，一方で加えた変更は，もう一方には伝わりません．たとえば，親プロセスにおいて wait の返値が pid 変数に代入されたとしても，それは子プロセスの pid 変数は変更しません．子プロセスの pid 変数は変わらず 0 を保持します．

exec システムコールは，呼び出し元プロセスのメモリを，ファイルシステムから

読み出したファイルをロードしたメモリイメージに取り替えます．そのファイルは，ファイルのどの部分が命令列か，どの部分がデータか，またどの命令列から実行を開始するか，などの情報を持つ特定のフォーマットに従っている必要があります．xv6 が利用する ELF フォーマットについては 3 章で詳しく説明します．成功した `exec` は，呼び出し元プログラムにはリターンしません．そのかわり，そのファイルから読み出された命令列の中で，ELF ヘッダにおいてエントリポイントと指定された命令にリターンします．`exec` は 2 つの引数を取ります．実行可能ファイルのファイル名と，文字列引数の配列です．以下が例です．

```
char *argv[3];

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

このコード片は，呼び出し元プログラムを，`echo hello` という引数で実行する `/bin/echo` というプログラムに取り替えます．最初の引数には慣習上プログラム名を指定しますが，多くのプログラムはこれを無視します．

xv6 のシェルは，上記のシステムコールを使うことで，ユーザにかわってプログラムを実行します．(user/sh.c:145) の `main` から分かるように，シェルの主構造は単純です．メインループは `getcmd` を用いて，ユーザ入力を 1 行読み込みます．そのあと，シェルは `fork` を呼んでシェルプロセスのコピーを作ります．子プロセスがコマンドを実行する間，親プロセスは `wait` を呼びます．たとえば，ユーザが入力したコマンドが “`echo hello`” であった場合，`runcmd` は “`echo hello`” を引数として呼び出されます．実際にコマンドを実行するのは `runcmd` (user/sh.c:58) です．“`echo hello`” コマンドに対しては，`exec` を呼び出します (user/sh.c:78)．ある時点で `echo` は `exit` を呼び，それによって親プロセスは `wait` から `main` にリターンします (user/sh.c:145)．

なぜ `fork` と `exec` を合わせて 1 つのシステムコールにしないのか疑問に思うかもしれません．のちほど，プロセス生成とプログラムのロードを分けて呼ぶことが，シェルの I/O リダイレクションで賢く使われている例を見ます．上記の `fork` の利用例において，重複したプロセスを作る無駄を避けるために，カーネルはコピーオンライト (copy-on-write) などの仮想メモリのテクニックを用いて `fork` の実装を最適化しています．

xv6 は，ユーザ空間のメモリを暗黙のうちにアロケート (allocate) します．`fork` は子プロセスが親プロセスのメモリのコピーを得るのに必要なメモリをアロケートし，

exec は実行可能ファイルを保持するのに必要なメモリをアロケートします．実行時に（多くの場合 malloc によって）さらにメモリが必要になったプロセスは sbrk(n) を呼ぶことで，データメモリを n バイト増加させることができます．sbrk は新しいメモリの場所をリターンします．

xv6 は，ユーザや，あるユーザから別のユーザの保護といった概念を提供しません．Unix の言葉で言えば，xv6 の全てのプロセスは root として実行します．

1.2 I/O とファイルディスクリプタ

ファイルディスクリプタは，カーネルが管理するオブジェクトで，プロセスが読み書きするものを表す小さい整数です．プロセスがファイルディスクリプタを得るのは，ファイル，ディレクトリ，あるいはデバイスを開いたときや，あるいはパイプを作ったり，既存のディスクリプタをコピーしたときです．簡単のため，ファイルディスクリプタが指し示すオブジェクトを，単に「ファイル」と呼びます．ファイルディスクリプタインタフェースは，ファイルや，パイプや，デバイスなどの差異を抽象化して，どれもバイトのストリームのように見せます．

xv6 カーネルは内部的に，プロセス固有のテーブルのインデックスとしてファイルディスクリプタを利用します．どのプロセスも，そのプロセスに固有な 0 から始まるファイルディスクリプタを持っています．慣習上，プロセスは 0 番ファイルディスクリプタ（標準入力）から読み込み，1 番ファイルディスクリプタ（標準出力）に書き込み，2 番ファイルディスクリプタ（標準エラー出力）にエラーメッセージを書き込みます．のちほど見るように，シェルはこの慣習を利用して I/O リダイレクションとパイプラインを実装します．シェルは，全てのプロセスがこれら 3 つのファイルディスクリプタが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のファイルディスクリプタです．

read と write システムコールは，ファイルディスクリプタが指すファイルに対し，バイト列を読み出したり書き込んだりします．read(fd, buf, n) は，ファイルディスクリプタ fd から最大で n バイトを読み出し，それを buf にコピーした上で，読みだしたバイト数をリターンします．ファイルに関連づいたファイルディスクリプタにはオフセットがあります．read は，現在のオフセットを起点に読み込みを行い，読んだ分だけオフセットを増加させます．そのあとに呼び出される read は，先の read が読んだものに続くバイト列を返します．もしそれ以上読み込むバイトが無いとき，read は 0 をリターンして，ファイルの終端であることを示します．

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リターンします．返値が n より少なくなるのはエラー

のときだけです．read と同様に，write はそのファイルの現在のオフセットに書き込みを行い，書き込んだバイト数分オフセットを進めます．続く write は，以前の write が終えた場所から始めます．

次のコード片は cat の本質的な部分であり，標準入力のデータを標準出力にコピーします．もしエラーが生じたら，それを標準エラー出力へ書き込みます．

```
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();
  }
}
```

ここで重要なのは，このコード片において，読み込み元がファイル，コンソール，およびパイプなどのいずれであるか，cat は関知しないということです．同様に，cat は書き込んでいる先がコンソールなのか，ファイルなのか，あるいは何か別のものなのかにも関知しません．ファイルディスクリプタを使うことと，0 番ファイルディスクリプタは入力・1 番ファイルディスクリプタは出力であるという慣習が，cat の実装を単純化しています．

close システムコールはファイルディスクリプタを解放して，将来の open, pipe, あるいは dup システムコール（後述します）が再利用できるようにします．新たにアロケートされたファイルディスクリプタは常に，そのプロセスの未使用ファイルディスクリプタのうち，最小の値を取ります．

ファイルディスクリプタと fork を使うことで，I/O リダイレクションを簡単に実装することができます．fork は，メモリとともに，親プロセスのファイルディスクリプタテーブルをコピーします．そのため，子プロセスは親プロセスと全く同じファイルを開いた状態で開始します．exec システムコールは，呼び出し元プロセスのメモリを取り替えますが，ファイルテーブルはそのままにしておきます．そのことを利用して，

シェルは I/O リダイレクションを実現します: fork を呼ぶ，選択したファイルディスクリプタを再度開く，新しいプログラムを exec する．以下は，cat < input.txt に対してシェルが実行するコードを単純化したものです:

```
char *argv[2];

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

子プロセスはファイルディスクリプタ 0 番を閉じたあと，open システムコールで input.txt を開きます．それにより得るファイルディスクリプタは 0 番です．0 が未使用ディスクリプタの中で最小だからです．cat はそのあと，input.txt に関連づいたファイルディスクリプタ 0 番を用いてプログラムを実行します．

xv6 シェルにおける I/O リダイレクションは，まさにその方法で動いています (user/sh.c:82)．この時点でシェルはすでに子プロセスを fork しており，runcmd がこれから exec を読んで新しいプログラムをロードするのでした．ここで，fork と exec を分けておくのが良いアイディアであることが明らかになるはずです．それらが分かれていることにより，シェルは子プロセスを fork したあと，子プロセスにおいて open, close, および dup を用いて標準入出力のファイルディスクリプタを変更し，そのあとに exec を呼ぶことができるのです．exec で呼ばれる側 (例では cat) の変更は不要です．もし fork と exec が合わせて 1 つのシステムコールであったら，シェルが標準入出力をリダイレクトするのにそのほかの（たぶんもっと複雑な）方法を使うか，もしくは呼ばれるプログラム自身が I/O をどうやってリダイレクトするか理解する必要があります．

fork はファイルディスクリプタテーブルをコピーしますが，対応するファイルのオフセットは親プロセスと子プロセスで共有されています．次の例を考えてみます:

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

このコード片の最後において，ファイルディスクリプタ 1 番に対応するファイルには，hello world というデータが入っているはずです．親プロセスにおける write (wait のおかげで必ず子プロセスのあとに実行される) は，子プロセスの write が終わった箇所から始めることができます．この挙動は，シェルコマンドの列から，出力を順番に取り出すのに役に立ちます．たとえば，(echo hello; echo world)>output.txt です．

dup システムコールは，既存のファイルディスクリプタを複製し，元と同じ I/O オブジェクトを参照する新しいファイルディスクリプタをリターンします．複製されたファイルディスクリプタは，元のディスクリプタとオフセットを共有します．fork で複製された場合と同様です．以下は，hello world をファイルに書き込む別の方法です．

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\en", 6);
```

fork の後に dup を呼ぶことで得た（同一のファイルディスクリプタから複製された）2 つのファイルディスクリプタはオフセットを共有します．それ以外の場合では，たとえ同一のファイルを open して得たファイルディスクリプタであっても，オフセットを共有することはありません．

dup により，シェルは次のようなコマンドを実現できます．ls existing-file non-existing-file > tmp1 2>&1. ここで 2>&1 は，ファイルディスクリプタ 2 番は 1 番の複製であるとシェルに伝えます．存在するファイル名と存在しないファイルに対するエラーメッセージの両方が tmp1 に現れることになります．xv6 のシェルはエラーファイルディスクリプタの I/O リダイレクションはサポートしませんが，これでどうやって実装すればよいか分かったはずです．ファイルディスクリプタはつながっている先の詳細を隠してくれる強力な抽象化です．あるプロセスがファイルディスクリプタ 1 番に書き込みを行うとき，実際の書き込み先はファイル，コンソールなどのデバイス，およびパイプのいずれでもありえます．

1.3 パイプ

パイプとはカーネルの小さいバッファで，プロセスに対しては一対のファイルディスクリプタに見えます: 片方は読み出し，もう片方は書き込み用です．データをパイプの一方に書き込むと，そのデータはパイプのもう一方から読み出すことができるようになります．パイプを用いると，プロセス間で通信を行うことができます．

次のプログラムは，wc プログラムを，パイプの読み出し側を標準入力に接続して実

行します.

```
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]);
  close(p[1]);
  exec("/bin/wc", argv);
} else {
  close(p[0]);
  write(p[1], "hello world\en", 12);
  close(p[1]);
}
```

このプログラムは pipe を呼んで新しいパイプを作り，読み書き用のファイルディスクリプタを配列 p に記録します．fork のあと，親プロセスと子プロセスはそのパイプに関連づいたファイルディスクリプタを持ちます．子プロセスはパイプの読み出し側をファイルディスクリプタ 0 番に dup で複製し，p のファイルディスクリプタを閉じ，そのあとで wc を exec します．wc が標準入力から読み出しを行うと，パイプから読み込みが行われます．親プロセスはパイプの読み出し側を閉じ，パイプに対して書き込みを行い，そして最後にパイプの書き込み側を閉じます.

もしデータが来ていない場合，パイプへの read は新しいデータが到来するか，パイプの書き込み側のファイルディスクリプタが全て閉じられるまで待ちます．後者の場合，read は，あたかもファイルの終端に達したかのように 0 をリターンします．新データの到来がありえなくなるまで read がブロックするすることは，子プロセスにおいてパイプの書き込み側を閉じなくてはいけない理由の 1 つです．もし wc のファイルディスクリプタの 1 つがパイプの書き込み側を指していた場合，wc は決してファイル終端にたどり着けません.

xv6 シェルは grep fork sh.c | wc -l のようなパイプラインを，上記のコードのような方法で実装しています (user/sh.c:100)．子プロセスはパイプラインの左側から右側までをつなぐためにパイプを作ります．そのあと，左側のコマンドのために

fork と runcmd を呼び，そのあと，右側のコマンドのために fork と runcmd を呼びます．最後に，両方が終わるまで待ちます．パイプラインの右側は，さらにパイプを含むことができます（たとえば a | b | c）．それはさらに 2 つの新しい子プロセスに fork します (すなわち b と c です)，そのようにして，シェルはプロセスのツリーを作ることができます．ツリーの葉はコマンドで，内側のノードは左右の子プロセスの完了を待つためのプロセスとなります．原理的には，内側のノードにパイプラインの左側の処理を割り当てることも可能ですが，正しく実行しようとすると実装が複雑になってしまうでしょう．

パイプをテンポラリファイルと比べた利点は分かりづらいかもしれません．次のパイプライン

```
echo hello world | wc
```

は，パイプラインを使わずに次のようにしても実装できるからです．

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

この使い方において，パイプのテンポラリファイルに対する利点は少なくとも 4 つあります．第一に，パイプは自動で消えます．ファイルのリダイレクションを使う場合，シェルは終了後に /tmp/xyz を消すように気をつけなくてはいけません．第二に，パイプは任意に長いデータのストリームを渡すことができます．それに対しリダイレクションでは，データ全体を保存するための領域がディスクに必要です．第三に，パイプでは，パイプラインの各ステージを並列に実行させることができます．ファイルを使うアプローチでは，先のプログラムが完了したあとで無いと，次のプログラムを開始できません．第四に，プロセス間通信に利用する場合，パイプのブロック付きの読み書きは，ファイルへのブロックなし読み書きよりもずっと効率的です．

1.4 ファイルシステム

xv6 のファイルシステムは単なるバイト配列であるデータファイルと，ディレクトリを提供します．ディレクトリは，データファイル・ディレクトリの名前とその場所を提供します．ディレクトリはツリーを構成します．その起点はルートと呼ばれる特別なディレクトリです．/a/b/c のようなパスは，c という名のファイルまたはディレクトリが，b というディレクトリに入っていて，それは更に a というルートディレクトリに入っているディレクトリに入っていることを意味します．/ から始まらないパスは，呼び出し元プロセスのカレントディレクトリから相対的に評価されます．以下のコード片はいずれも同じファイルを開きます（関係するディレクトリは存在するものとします）：

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

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

1 つ目コード片は，そのプロセスのディレクトリを /a/b に移動します．それに対して 2 つ目のコード片は，カレントディレクトリを参照することも変更することもしません．

新しいファイルやディレクトリを作るためのシステムコールはいくつもあります．mkdir は新しいディレクトリを作ります．O_CREATE フラグ付きで open すると新しいファイルを作ります．mknod は新しいデバイスファイルを作ります．以下は，それら 3 つ全ての例です:

```
mkdir("/dir");
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．そのかわり，そのファイルのメタデータにデバイスファイルであるマークがついており，メジャーおよびマイナーデバイス番号を記録しています（mknod の 2 つの引数で指定します）．プロセスがそのファイルを開くと，カーネルは read や write をファイルシステムではなく，カーネル内のデバイスの実装に誘導します．

fstat は，ファイルディスクリプタが指し示すオブジェクトの情報を取得します．結果は，stat.h (kernel/stat.h) で定義される stat 構造体に入ります:

```
#define T_DIR     1   // Directory
#define T_FILE    2   // File
#define T_DEVICE  3   // Device

struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

ファイル名はファイルそのものとは独立です．*inode* と呼ばれるファイルの内部表現は複数の名前を持つことができ，それはリンクと呼ばれます．link システムコー

ルは，既存ファイルの inode を指し示す別のファイルを作成します．次のコード片は，a と b という 2 つ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");
```

a に読み書きすることは，b に読み書きすることと等価です．各 inode は，固有の inode 番号で識別します．上記のコードを実行したあと，a と b が同じ内容を示すことは，fstat で確認できます．両者は同じ inode 番号 (ino) を持ち，また nlink は 2 になっているはずです．

unlink システムコールは，ファイルシステムから名前を削除します．そのファイルの inode およびファイル内容を記憶するディスク上の領域は，ファイルのリンク数が 0 になり，かつ参照しているファイルディスクリプタが無くなったときに限り解放されます．よって，先のコードに

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

と付け足したとしても，ファイルの inode とその内容には，b としてアクセスできます．さらに，

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

とすることは，そのプロセスが fd を閉じたり，あるいは終了したときに消える一時ファイルを作成するときのイディオムです．

ファイルシステムの操作のためのシェルのコマンドは，mkdir, ln, rm などのユーザレベルプログラムとして実装されます．そのようにすることで，新しいユーザレベルプログラムを足すだけで，シェルを拡張して新しいユーザコマンドを追加できます．今となっては当たり前に見えるかもしれませんが，Unix が設計された時代，他のシステムはそのようなコマンドをシェルに内蔵していました（そして，シェルをカーネルに内蔵していました)．

ひとつの例外は cd で，それはシェルに内蔵されています (user/sh.c:160)．cd はシェル自体のカレントディレクトリを変更しなくてはいけません．もし cd が通常コマンドとして実行されると，シェルは子プロセスを生成し，その子プロセスが cd を実行することになります．そうすると，子プロセスの作業ディレクトリは変化しますが，親プロセスの（すなわちシェルの）作業ディレクトリは変化しません．

1.5 世の中のオペレーティングシステム

Unix において,「標準」のファイルディスクリプタ，パイプ，およびそれらを使うための便利なシェルの構文が組み合わさることが，再利用性の高い汎用プログラムを書くことの大きな利点でした．このアイディアは，Unix のパワーと人気の源泉である「ソフトウェアツール」の文化に火をつけ，シェルはいわゆる最初の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，今日においても，BSD, Linux, および Mac OS X などのシステムで維持されています.

Unix のシステムコールインターフェースは，Portable Operating System Interface (POSIX) として標準化されました．xv6 は POSIX 準拠ではありません．xv6 には足りないシステムコールがあったり（ごく基本的な `lseek` を含みます），部分的にしか実装されていないシステムコールがあるなどの違いがあるためです．Unix 風のシステムコールインターフェースを持ちますが，xv6 の主なゴールは単純さと明快さにあります．新しいシステムコールや，基本的な Unix プログラムを実行するための簡単な C ライブラリで xv6 を拡張する人もいます．しかしながら，現代的なカーネルは，xv6 と比べて圧倒的に多くのシステムコールやサービスを提供しています．たとえば，ネットワーク機能，ウィンドウシステム，ユーザレベルスレッド，多くのデバイスドライバなどを備えています．現代的なカーネルは，持続的かつ急速に発展しており，POSIX を超える多くの機能を提供します.

Unix から派生した現代的なオペレーティングシステムの多くは, 先に見た `console` デバイスファイルのような，デバイスを特殊ファイルとして扱う古き Unix モデルを採用しませんでした．Unix の作者たちは Plan 9 に移り，「リソースはファイルである」というコンセプトを，より現代的な設備であるネットワークやグラフィクスなどの資源に適用しました.

ファイルシステムとファイルディスクリプタは強力な抽象化でしたが，オペレーティングシステムには別のインターフェースもありえました．Unix の先祖である Multics は，ファイルストレージをメモリのように抽象化することで，全く異なるインタフェースを作り上げました．Multics の設計の複雑さは，Unix の設計者たちに直接的に影響を与え，シンプルを求める原動力となりました.

この本は，xv6 がどのようにして Unix 風のインタフェースを実装するかを説明しますが，そのアイディアとコンセプトは Unix に限らず適用できるものです．オペレーティングシステムであれば，どのようなものであっても，プロセスをハードウェアに対して切り替えたり，プロセス同士を分離したり，プロセス間通信のためのメカニズムを提供したりします．xv6 を学んだあとにより複雑なオペレーティングシステムを

見れば，xv6 と同様のコンセプトを見つけるはずです.

1.6 練習問題

1. 2 つのプロセスを（各方向からなる）1 組のパイプで接続し，ある 1 バイトのデータをプロセス間で卓球のように行き来させるプログラムを，Unix システムコールを用いて書いてください．また，1 秒あたりの行き来の回数を性能評価してください.

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にとって重要な要求は，いろいろな活動を一度にできることです．たとえば，1章で述べたシステムコールインターフェースにおいて，プロセスは `fork` を使って新しいプロセスを生むことができます．オペレーティングシステムは，多数のプロセス間で，コンピュータのリソースをタイムシェア（時間的に共有）しなくてはいけません．たとえば，物理的な CPU よりも多数のプロセスがあったとしても，オペレーティングシステムはそれらの進行を保証しなくてはいけません．オペレーティングシステムは加えて，プロセス間の分離も行わなくてはなりません．あるプロセスがバグでフェイル (fail) したとしても，別のプロセスに影響があってはいけません．しかし，完全な分離は強すぎます．プロセス間で，意図的に相互通信したい場合もあるからです．パイプラインはその一例です．オペレーティングシステムは，次の 3 つの要求を満たす必要があります．すなわち，マルチプレクス (multiplexing), 分離 (isolation), および相互通信 (interaction) です．

この章では，3 つの要求を満たすために，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説します．さまざまな実現法がありえることが明らかになりますが，この本ではモノリシックカーネルを中心とする主流の設計を述べます．この章はまた，xv6 の分離の単位であるプロセスと，xv6 の起動時における最初のプロセスの生成についても概説します．

xv6 は，マルチコアの RISC-V マイクロプロセッサで動作するので，多くの低レベル機能（たとえばプロセスの実装）は RISC-V 固有です．RISC-V は 64 ビット CPU です．xv6 は特に “LP64” C, すなわち long 型とポインタ (P) 型が 64 ビットである一方，int 型は 32 ビットという C 言語で記述されています．読者に（何らかのアーキテクチャでの）マシンレベルのプログラミングの経験が少しあると想定し，RISC-V 固有のアイディアが出てきたらそれを紹介します．RISC-V に関する便利な参考文献は “The RISC-V Reader: An Open Architecture Atlas” [9] です．ユーザレベルの命令

セットアーキテクチャと権限付きアーキテクチャの公式の仕様は [2] と [1] です．

この文章は計算を実行する要素のことを基本的に *CPU* と呼びます．Central Processing Unit の略です．他の文章（たとえば RISC-V の仕様書）は，CPU のかわりに，プロセッサ，コア，および hart という用語を使うこともあります．xv6 はマルチコアの RISC-V のハードウェアを想定します．それはすなわち，複数の CPU がメモリを共有しながら，独立のプログラムを並列で実行するということです．この文章はときどき，マルチコアと同じ意味でマルチプロセッサという用語を使います．厳密には，マルチプロセッサとは，1 つの基板に複数のプロセッサチップが載ったものを表すこともあります．

完成品のコンピュータにおける CPU は，それを助けるためのハードウェアに囲まれており，そのほとんどは I/O インタフェースのためにあります．xv6 は，qemu に`-machine virt` オプションをつけてシミュレートできるハードウェアをサポートします．それは，RAM, ブートコードが入った ROM, ユーザのキーボードや画面につながったシリアル通信，およびディスクストレージを含みます．

2.1　物理的なリソースの抽象化

オペレーティングシステムに初めて出会ったときに最初に思い浮かぶ疑問は，なぜそんなものが必要なのか，ということかもしれません．図 1.2 のシステムコールは，アプリケーションプログラムをリンクするライブラリとして実現することもできます．その場合，各アプリケーションは，そのアプリケーション専用のライブラリを使うことができます．アプリケーションは，ハードウェアリソースに直接アクセスすることができ，そのアプリケーションに最適な方法で使うことができます（そうすることで，高速化や，予測可能な性能を達成できます）．実際，組込機器向けのオペレーティングシステムやリアルタイム向けシステムには，そのような方法で構成されたものもあります．

ライブラリを用いたアプローチの欠点は，2 つ以上のアプリケーションが走っているとき，それらのアプリケーションは行儀よくしなくてはいけないということです．たとえば，各アプリケーションは，他のアプリケーションが実行できるように定期的に CPU を手放さなくてはなりません．各アプリケーションがお互いを信用しており，かつバグが無いならば，そのような協調的タイムシェリングはうまくいきます．しかし，アプリケーション同士は信頼しあっておらず，かつバグがあるのがより一般的な状況です．そのため，協調的な方法よりも，より強い分離が欲しいときがあります．

強い分離を実現するよい方法は，アプリケーションに，繊細なハードウェアリソースを直接触らせないことです．そのかわり，リソースをサービスとして抽象化するの

です．たとえば，アプリケーションが生のディスクのセクタを直接読み書きするかわりに，`open`, `read`, `write`, および `close` システムコールを介してファイルシステムにアクセスさせるのです．そのようにすることで，アプリケーションはパスや名前でアクセスができて便利になるとともに，オペレーティングシステムは（そのインタフェースの実装者として）ディスクを管理することができます．

同様に，Unix は複数のプロセスに対してハードウェアの CPU を透過的に切り替えます．レジスタ状態を必要に応じて退避することで，アプリケーションがタイムシェアリングであることを気にしなくてよいようにします．この透過性により，たとえ無限ループに陥るアプリケーションがいたとしても，オペレーティングシステムは CPU を共有することができます．

Unix のプロセスが，物理メモリと直接やり取りするかわりに，`exec` を使ってメモリイメージを構築するのは別の例です．そうすることで，オペレーティングシステムはメモリのどこにそのプロセスを置くか決めることができるようになります．もしメモリが足りなかったら，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．`exec` により，実行可能なプログラムイメージを，ファイルシステムに保存するという利便性もあります．

Unix プロセス間の相互通信の多くは，ファイルディスクリプタを経由して行います．ファイルディスクリプタは，抽象化によって詳細（たとえばパイプの中のデータやファイルはどこに保存されているか，など）を気にしなくてよくなるだけでなく，相互通信を単純化する方法も規定します．たとえば，パイプライン中のあるアプリケーションがフェイルした場合，カーネルはファイル終端シグナルをパイプラインの次のプロセスに送ることができます．

ここで見たように，図 1.2 のシステムコールは緻密に設計されており，プログラマの便利さだけでなく，強い分離を行う機会を提供します．Unix インタフェー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，とても良いものであると知られています．

2.2 ユーザモード，スーパーバイザモード，およびシステムコール

強い分離をするには，アプリケーションとオペレーティングシステムの間に明確な境界線が必要です．もしあるアプリケーションがフェイルしても，オペレーティングシステムや別のアプリケーションにはフェイルが伝搬しないで欲しいです．オペレーティングシステムはフェイルしたアプリケーションを片づけ，他のアプリケーションを実行させ続けなくてはいけません．そのような強い分離を実現するには，あるアプ

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列を変更できなくする（あるいは見ることすらできなくする）とともに，別のアプリケーションのメモリにアクセスできなくする必要があります．

多くの CPU には，強い分離を行うためのハードウェア的な支援機能があります．たとえば，RISC-V は CPU が命令列を実行するのに 3 つのモードがあります．マシンモード，スーパーバイザモード，およびユーザモードです．マシンモードで実行する命令列は無制限の権限を持ちます．起動直後の CPU はマシンモードです．マシンモードは，ほとんどの場合，コンピュータのコンフィギュレーションで使います．xv6 がマシンモードで実行するのはわずか数行で，そのあとはスーパーバイザモードに切り替わります．

スーパーバイザモードにおいて，CPU は特権命令を実行することができます．特権命令を用いると，たとえば，割込の有効化・無効化や，ページテーブルのアドレスを指定するレジスタの読み書きなどができます．もしユーザ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，かわりに，スーパーバイザモードに切り替わります．そして，するべきでないことをしようとしたアプリケーションを強制終了します．1 章の図 1.1 はそのような構成を示しています．アプリケーションはユーザモードの命令（たとえば足し算など）のみを実行することができ，それを「ユーザ空間で実行している」と言います．それに対し，スーパーバイザモードでは特権命令も実行することができ，「カーネル空間で実行している」と言います．カーネルとは，カーネル空有で実行する（あるいはスーパーバイザモードで実行する）ソフトウェアのことです．

あるアプリケーションが，カーネルの関数（たとえば xv6 における `read` システムコール）を呼びたいと思ったら，カーネルに切り替えなくてはいけません．CPU が提供する特殊な命令を実行すると，ユーザモードからスーパーバイザモードが切り替わった上で，カーネルが指定するエントリポイントに飛びます (RISC-V では `ecall` 命令を使います)．CPU がスーパー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，そのアプリケーションが実行して良い要求なのか判断し，実行するか拒否します．スーパーバイザへの切り替えにおけるエントリポイントをカーネルが支配するのは重要です．もしアプリケーションがカーネルのエントリポイントを選べたとしたら，悪意を持ったアプリケーションは，たとえば，引数の検証を回避できる場所を選ぶことができてしまいます．

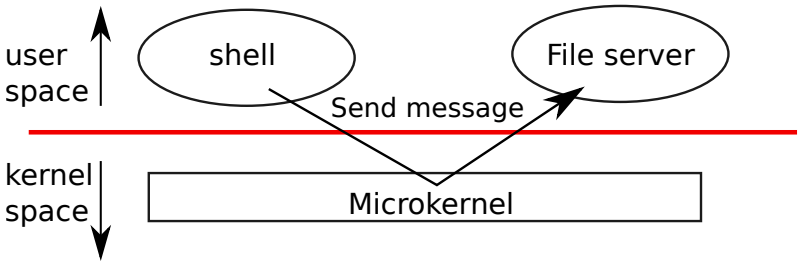

図 2.1　マイクロカーネルとファイルサーバ

2.3　カーネルの構成

オペレーティングシステムのどの部分をスーパーバイザモードで実行するかが，設計上の重要な問いです．ひとつの方法は，オペレーティングシステム全体をカーネルに入れてしまい，システムコールの実装全体をスーパーバイザモードで実行させることです．この構成をモノリシックカーネルと呼びます．

この構成では，オペレーティングシステム全体が無制限のハードウェア特権を持って実行されます．そうすると，ハードウェア特権が必要な場所とそうでない場所を判断しなくてよくなるので，OS の設計者にとって楽です．さらに，オペレーティングシステムの異なる部分同士が簡単に協調できるようになります．たとえば，ファイルシステムと仮想メモリシステムの間に，共有のバッファキャッシュを用意する場合などです．

モノリシックカーネルの欠点は，オペレーティングシステムの異なる部分同士のインタフェースが複雑になりがちなことです（本書のあとの方で詳しく見ます）．そうすると，オペレーティングシステムの開発者が間違いを犯しやすくなります．モノリシックカーネルにおいて間違いは致命的です．スーパーバイザモードでは，たとえ小さな間違いであってもカーネルをフェイルさせうるからです．カーネルがフェイルするとコンピュータは止まってしまい，アプリケーションも当然フェイルします．復帰するには，コンピュータを再起動しなくてはいけません．

カーネルにおける間違いを減らすために，スーパーバイザモードで実行するコードを厳選し，オペレーティングシステムの大部分をユーザモードで走らせることもできます．そのようなカーネルの構成法をマイクロカーネルと呼びます．

図 2.1 にマイクロカーネルを図示します．図において，ファイルシステムはユーザレベルのプロセスとして走ります．プロセスとして走る OS のサービスをサーバと呼びます．アプリケーションがファイルサーバと相互通信できるように，カーネルはプロセス間通信のメカニズムを提供することで，あるユーザモードプロセスから別の

ユーザモードプロセスにメッセージを送れるようにします．たとえば，シェルのようなアプリケーションがファイルの読み書きをしたいと思ったら，そのアプリケーションはファイルサーバにメッセージを送り，その応答を待ちます．

マイクロカーネルにおけるカーネルのインタフェースは，アプリケーションを開始する，メッセージを送る，デバイスのハードウェアにアクセスするなどのごく少ない低レベル関数からなります．そのような構成を取ると，オペレーティングシステムの大部分はユーザレベルのサーバとなるため，カーネルを比較的単純にすることができます．

xv6 は，多くの 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します．xv6 のカーネルインタフェースはオペレーティングシステムのインタフェースと同一であり，カーネルがオペレーティングシステム全体を実装します．xv6 はそこまで多くのサービスは提供しないため，ある種のマイクロカーネルよりも小さいのですが，コンセプト上，xv6 はモノリシックです．

2.4 Code: xv6 の構成

xv6 のカーネルのソースは `kernel/` サブディレクトリに入っています．ソースは，荒くモジュール化され，ファイル分けされています．図 2.2 はファイルの一覧です．それぞれのモジュールとのインタフェースは，`defs.h` (kernel/defs.h) で定義されています．

2.5 プロセスの概要

xv6 における分離の単位は（他の Unix オペレーティングシステムと同様に）プロセスです．プロセスによる抽象化を行うことで，あるプロセスが別のプロセスのメモリ, CPU, およびファイルディスクリプタなどを壊したり覗いたりできないようになります．同様に，プロセスがカーネルを破壊できないようにすることで，分離メカニズムを迂回できないようにします．カーネルは，プロセスの抽象化を注意深く実装しなくてはいけません．さもなくば，バグっていたり悪意を持ったアプリケーションが，カーネルやハードウェアをだまして何か悪いことを引き起こすからです（例: 分離の迂回）．カーネルがプロセスを実装するために用いるメカニズムには，ユーザモード・スーパーバイザモードのフラグ，アドレス空間，スレッドのタイムスライスなどがあります．

分離の強制力を高めるために，プロセスを抽象化することで，あるプログラムは，あたかもマシンを専有しているかのように見せます．プロセスはプログラムに対し，

ファイル名	説明
bio.c	ファイルシステムのためのディスクブロックキャッシュ.
console.c	ユーザのキーボードと画面への接続.
entry.S	起動時の一番最初の命令列.
exec.c	exec() システムコール.
file.c	ファイルディスクリプタのサポート.
fs.c	ファイルシステム.
kalloc.c	物理ページのアロケータ.
kernelvec.S	カーネルおよびタイマ割込からのトラップの処理.
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.
main.c	ブート時の他モジュールの初期化を制御.
pipe.c	パイプ.
plic.c	RISC-V の割込コントローラ.
printf.c	画面へのフォーマット付き出力.
proc.c	プロセスとそのスケジューリング.
sleeplock.c	CPU を手放すロック.
spinlock.c	CPU を手放さないロック.
start.c	初期のマシンモードでのブートコード.
string.c	C 文字列とバイト配列のためのライブラリ.
swtch.S	スレッド切り替え.
syscall.c	実際に処理をする関数にシステムコールをディスパッチする.
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.
trampoline.S	ユーザとカーネルを切り替えるためのアセンブリコード.
trap.c	トラップと割込を処理してリターンするための C コード.
uart.c	シリアルポートコンソールのデバイスドライバ.
virtio_disk.c	ディスクのデバイスドライバ.
vm.c	ページテーブルとアドレス空間の管理.

図 2.2　Xv6 kernel source files.

他のプロセスが読み書きすることができない，そのプロセス固有に見えるメモリシステム（あるいはアドレス空間）を提供します．加えて，そのプログラムの命令列を専属で実行するかのように見える CPU を提供します．

xv6 はページテーブル（ハードウェア的に実装されるもの）を用いて，各プロセスに固有のアドレス空間を与えます．RISC-V のページテーブルは，仮想アドレス（RISC-V 命令が扱うアドレス）を，物理アドレス（CPU がメインメモリに送るアドレス）に翻訳（あるいはマップ）します．

ページテーブルはそのプロセスのアドレス空間を決めるもので，xv6 は，プロセスごとに別のページテーブルを用意します．図 2.3 に示すように，アドレス空間には，そのプロセスのユーザメモリが，仮想アドレス 0 番から入っています．最初に命令列，続いてグローバル変数，次にスタック，そして最後にヒープ領域（malloc のためのもので，プロセスが必要に応じて増やせるもの）があります．xv6 は，仮想アドレス 39 ビットの RISC-V で動きますが，そのうち 38 ビットしか使いません．よって，

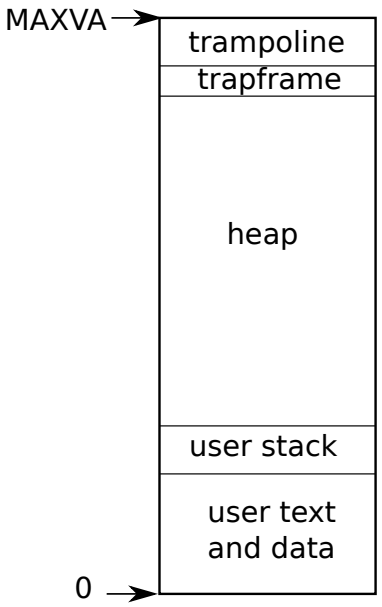

図 2.3　ユーザプロセスの仮想アドレス空間のレイアウト

最大長は $2^{38} - 1$ = `0x3fffffffff` で，これは `MAXVA` (kernel/riscv.h:349) です．4 章で詳しく見るように，アドレス空間の先頭にはトランポリンのための 1 ページと，プロセスのトラップフレームがカーネルに切り替わるための 1 ページがあります．

xv6 カーネルは，プロセスごとにさまざまな状態を保持しており，それらは `proc` 構造体 (kernel/proc.h:86) に収まっています．プロセスにとってもっとも重要なカーネルの状態は，ページテーブル，カーネルスタック，および実行状態です．以降は，`proc` 構造体の各要素を示すために `p->xxx` という記法を使います．たとえば，上述したトラップフレームは `p->tf` です．

各プロセスは，そのプロセスの命令列を実行する実行スレッド（あるいは単純にスレッド）を持ちます．スレッドは中断して，あとで再開できます．プロセスを透過的に切り替えるときには，カーネルは現在走っているスレッドを中断し，別のプロセスのスレッドを再開します．スレッド内の状態（ローカル変数，関数コールにおけるリターンアドレス）は，そのスレッド固有のスタックに記憶されます．プロセスは 2 つのスタックを持ちます．1 つはユーザスタック，もう 1 つはカーネルスタック (`p->kstack`) です．プロセスがユーザの命令列を実行しているときはユーザスタックのみが利用され，カーネルスタックは空です．（システムコールや割込により）プロセスがカーネルに入ると，カーネルのコードはそのプロセスのカーネルスタックを用いて実行を行います．あるプロセスがカーネルに入っているとき，ユーザスタックは記憶されたデータを保持し続けますが，積極的には利用はされません．プロセスのスレッドは，ユーザスタックとカーネルスタックを行ったり来たりします．カーネルス

タックは分離されているため（そしてユーザコードからは保護されているため），あるプロセスが自身のスタックを壊してしまったとしても，カーネルは実行を継続できます．

プロセスは，システムコールを呼ぶために RISC-V の `ecall` 命令を実行します．この命令は，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決めたエントリポイントに移します．システムコールが完了すると，カーネルはスタックをユーザスタックに切り替え，それから `sret` 命令を呼んでユーザ空間にリターンします．`sret` 命令は，ハードウェア特権レベルを下げ，システムコール命令の直後の命令から実行を再開します．プロセスのスレッドは，I/O を待つために，カーネルで「ブロック」することがあります．その場合，I/O が完了すると実行を再開します．

`p->state` は次のうちいずれかを表します: プロセスがアロケート済みである, 実行待ちである，実行中である，I/O を待っている，終了中である．

`p->pagetable` は, そのプロセスのページテーブルです．ページテーブルは, RISC-V が指定するフォーマットで表現されます．あるプロセスをユーザ空間で実行し始めるとき，xv6 そのプロセスの `p->pagetable` を使うようにページングハードウェアに指示します．プロセスのページテーブルは，そのプロセスのためにアロケートされた物理ページのアドレスの一覧にもなっています．

2.6 Code: xv6 の起動と最初のプロセス

xv6 をより具体化するために，カーネルがどのように開始し，そしてどうやって最初のプロセスを起動するか概説します．続く章では，ここで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オンになると, CPU は自身を初期化して, リードオンリーメモリに入った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．そのあと，マシンモード状態の CPU は `_entry` (kernel/entry.S:12) から実行を始めます．xv6 が開始したとき，RISC-V のページングハードウェアはまだ無効であり，仮想アドレスと物理アドレスは直接マップされています．

ローダは，xv6 カーネルを物理アドレス `0x80000000` にロードします．`0x0` ではなく `0x80000000` から始めるのは，`0x0`〜`0x80000000` のアドレス範囲には I/O デバイスが割りあたっているからです．

`_entry` にある命令列は，スタックを設定することで，xv6 が C コードを実行できるようにします．xv6 は，最初のスタックである `stack0` を `start.c` で宣言しています (kernel/start.c:11)．`_entry` のコードは，スタックポインタレジスタ `sp` に，スタックアドレス `stack0+4096` を代入します．RISC-V のスタックは下向きに育つの

で，このアドレスがスタックの先頭となります．スタックが準備できたら，_entry は start の C コードに入ります (kernel/start.c:21).

その関数（たち）はマシンモードでしか許されていない各種コンフィギュレーションを行い，そのあとスーパーバイザモードに切り替えます．スーパーバイザモードに入るために，RISC-V には mret 命令があります．この命令は，ほとんどの場合，どこかでスーパーバイザモードからマシンモードへの呼び出しが行われたあと，元の場所にリターンするために使います．しかし，start はそのような呼び出しからリターンするわけではありませんので，あたかもそうであったかのように準備をします．そのために次を行います: (i) 前の特権がスーパーバイザであることを mstatus レジスタにセットする，(ii) mepc レジスタに main 関数のアドレスを書き込むことで，返り先を main に指定する, (iii) ページテーブル用レジスタ satp に 0 を書き込むことで，仮想アドレスへの変換を無効化する，(iv) 全ての割込と例外をスーパーバイザモードに移譲する．

スーパーバイザモードにジャンプする前に，もう 1 つやらなくてはいけないことがあります．タイマ割込を行うように，クロックチップに設定を書き込むことです．以上の面倒を見終わったら，starn は mret を呼ぶことでスーパーバイザモードへ「リターン」します．それにより，プログラムカウンタは main (kernel/main.c:11) に移ります．

main (kernel/main.c:11) は，さまざまなデバイスとサブシステムの初期化を追えると，userinit を呼んで最初のプロセスを生成します (kernel/proc.c:197)．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) を実行します．それは，exec システムコールを呼んで再びカーネルに入ります．1 章で見たように，exec はそのプロセスのメモリやレジスタを，新しいプログラム（今回は /init）に取り替えます．カーネルが exec を実行し終えると，そのプロセスはユーザ空間にリターンして /init の実行が始まります．init (user/init.c:11) は，必要に応じてコンソール用の新しいデバイスファイルを生成し，それをファイルディスクリプタ 0, 1, および 2 番として開きます．そのあと，コンソール用シェルを起動し，孤立したゾンビプロセスの面倒を見るためのループに入ります．こうしてシステムは起動しました．

2.7 世の中のオペレーティングシステム

現実の世界には，モノリシックカーネルとマイクロカーネルの両方があります．多くの Unix カーネルはモノリシックです．たとえば，Linux はモノリシックカーネルです．ただし，一部の機能 (たとえばウィンドウシステム) はユーザレベルサーバとして動きます．L4, Minix, および QNX のようなカーネルはサーバを用いるマイクロ

カーネルの構成を持ち，組込用途で広く使われています．

多くのオペレーティングシステムが，プロセスというコンセプトを採用しており，それは xv6 のプロセスと似たものです．ただし，現代的なオペレーティングシステムは，1 つのプロセス内に複数のスレッドを持つことをサポートします．そうすることで，1 つのプロセスが複数の CPU を使うことができるようになります．1 つのプロセス内で複数のスレッドを動かせるようにするには，xv6 にはない多くのしかけ（たとえば，Linux は `clone` という `fork` の派生版を使います）を用いることで，プロセス内のスレッドが何を共有するか制御する必要があります．

2.8 練習問題

1. gdb を使うと，カーネルからユーザへの，一番最初の切り替えを観察することができます．`make qemu-gdb` を実行してください．別のウィンドウにおいて，同じディレクトリで gdb を実行してください．gdb のコマンド `break *0x3ffffff07e` を実行することで，ユーザ空間にジャンプする `sret` 命令にブレークポイントを設定してください．gdb で `continue` コマンドを入力すると，gdb はそのブレークポイントにおいて，`sret` を実行する直前で停止します．`stepi` を入力すると，アドレス `0x4` 番に実行が移ることが，gdb で確認できます．このアドレスはユーザ空間における，`initcode.S` の先頭を指しています．

第 3 章

ページテーブル

ページテーブルは，オペレーティングシステムが各プロセスに対し，そのプロセス固有のアドレス空間とメモリを提供するためのメカニズムです．ページテーブルは，メモリアドレスの意味と，物理メモリのどこにアクセスできるかを決めます．xv6 はページテーブルを用いることで，異なるプロセス間のアドレス空間を分離するとともに，１つしかない物理メモリをマルチプレクスします．ページテーブルが提供する間接的なアクセスは，次のようなトリックを実現するのにも役立ちます．同一のメモリ（トランポリンページ）を複数のアドレス空間にマップするトリックや，未割り当てページを作ることでカーネルスタックとユーザスタックの間をガードするトリックです．本章の残りは，RISC-V が提供するページテーブルと，それを xv6 がどのように使うかを説明します．

3.1　ページングハードウェア

RISC-V の命令列は（ユーザもカーネルもどちらも）仮想アドレスを使います．それに対してマシンの RAM, すなわち物理メモリは物理アドレスを使います．RISC-V のページテーブル用ハードウェアは，仮想アドレスを物理アドレスにマップすることで，それら 2 種類のアドレスを結びつけます．

xv6 が動作するのは Sv39 RISC-V, すなわち 39 ビットの仮想アドレスを持つものです（図 3.1 を見てください）．64 ビットの仮想アドレスのうち，上位 25 ビットは使用しません．Sv39 のコンフィギュレーションにおいて，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: Page Table Entries) からなる配列です．各 PTE は，44 ビットの物理ページ番号 (PPN: Physical Page Number) と，各種フラグを含みます．ページングハードウェアは，39 ビット中の上位 27 ビットをインデックスとして用いてページテーブルから PTE を探し，PPN

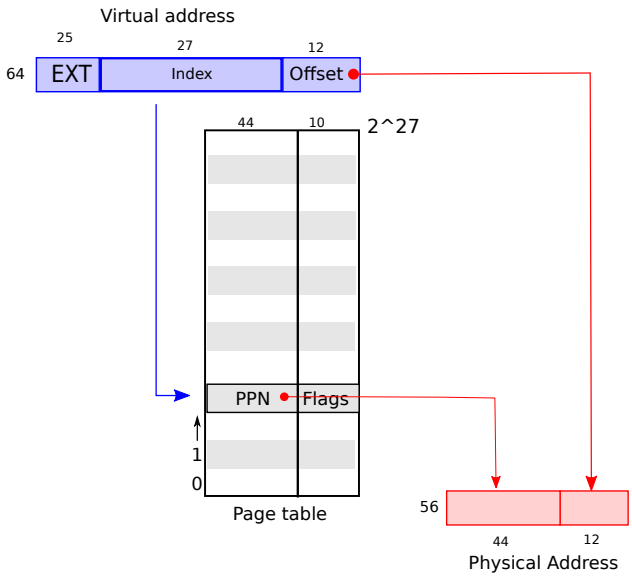

図 3.1　RISC-V の仮想アドレスと物理アドレス

の 44 ビットと，仮想アドレス下位 12 ビットを連結して 56 ビットの物理アドレスを作ります．すなわち，ページテーブルは，4096 (2^{12}) バイトに整列した塊を粒度として，仮想・物理アドレスの変換を制御することができます．そのような塊をページと呼びます．

Sv39 RISC-V において，仮想アドレスの上位 25 ビットは，アドレス変換には利用しません．将来的に，RISC-V はそれらのビットを，さらなるアドレス変換の階層のために利用するかもしれません．同様に，物理アドレスも増やす余地があります．Sv39 では 56 ビットですが，64 ビットまで増やすことができます．

図 3.2 に示すように，実際のアドレス変換は 3 ステップからなります．ページテーブルは 3 階層のツリーとして，物理メモリに保存されています．ツリーのルートは，4096 バイトのページテーブル用ページ (page-table page) です．このページは 512 個の PTE を含み，各エントリは次の階層のページテーブル用ページの物理アドレスを指します．ツリーの最後の階層に至るまで，どのページも 512 個の PTE を含みます．ページテーブルのハードウェアは，ルートにあるページテーブル用ページから PTE を選択するインデックスとして，27 ビットのうち最上位 9 ビットを用います．その次の階層では中間の 9 ビットを，最後の階層では最後の 9 ビットをそれぞれ用います．

もし，アドレス変換に必要な 3 つ PTE のうち 1 つでも足りなければ，ページング用ハードウェアはフォールト (fault) を発生させます．一般的な用途において，仮想アドレスの大半は未割り当てです．上記のような 3 階層の構造を用いることで，未割り当ての範囲に対応するページテーブル用ページ全体を省略し，メモリを節約できます．

各 PTE は，フラグビットを含んでおり，対応する仮想アドレスを使用してよいかど

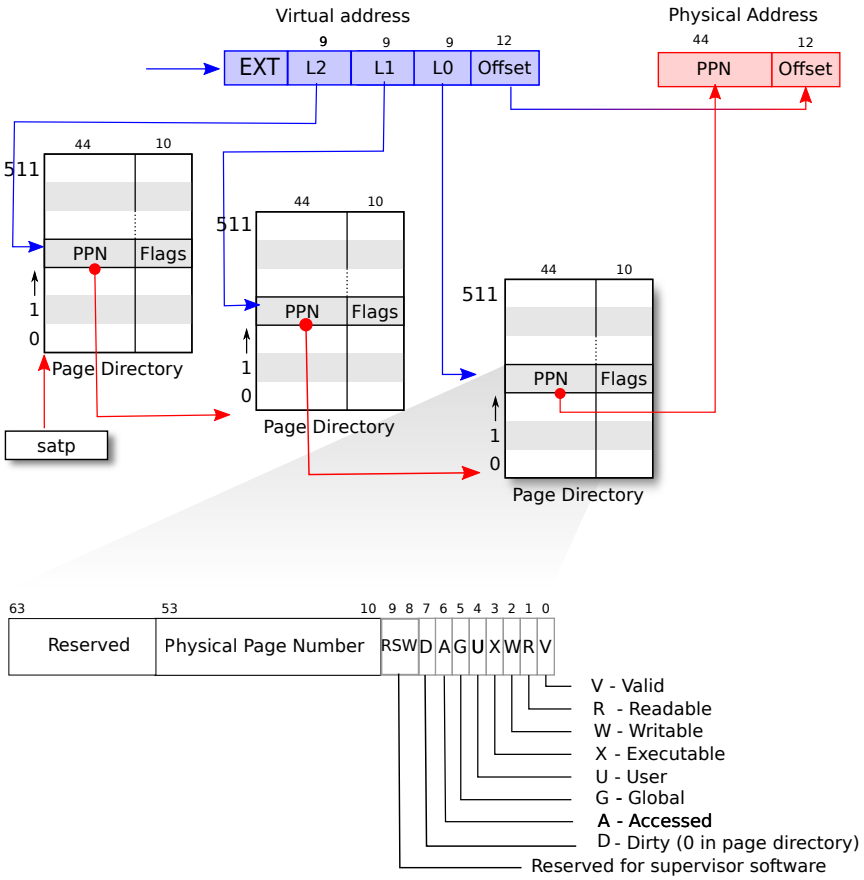

図 3.2　RISC-V のページテーブルのハードウェア

うかをページテーブル用ハードウェアに伝えるために利用されます．PTE_V は，PTE が存在するかどうかを示します．もしフラグがセットされていなければ，そのページへの参照はフォールトを生じます（すなわち許可されません）．PTE_R は，命令列が，そのページを読んでよいかどうかを指定します．また，PTE_W は，命令列が，そのページに書き込んでよいかどうかを指定します．PTE_X は，そのページの中身を命令として解釈し，実行して良いかどうかを指定します．PTE_U は，ユーザモードで実行する命令列が，そのページアクセスできるかどうかを指定します．もし PTE_U がセットされていなければ，その PTE はスーパーバイザモードでのみ使うことができます．図 3.2 に，以上がどのように動くか示します．各フラグと，ページング用ハードウェアに関連した構造体は (kernel/riscv.h) で定義されています．

あるページテーブルを使うようにハードウェアに指示するためには，ルートとなるページテーブル用ページの物理アドレスを，satp レジスタに書き込みます．各 CPU は，固有の satp レジスタを持ちます．CPU は，続く命令列が生成するアドレスを，その CPU 固有の satp が指すページテーブルを用いてアドレス変換します．各 CPU が固有の satp を持つので，各 CPU はそれぞれ別のプロセスを実行できます．プロ

セスは，ページテーブルが規定する固有のアドレス空間を持っています．

用語について注意をしておきます．物理メモリとは，DRAM 内部の記憶セルを意味します．物理メモリ内のバイトはアドレスを持ち，それを物理アドレスと呼びます．命令は仮想アドレスしか扱いません．仮想アドレスはページング用ハードウェアによって物理アドレスに変換され，その上で DRAM に送られて読み書きに利用されます．仮想アドレスと物理アドレスの間を取り持つ，カーネルが提供する抽象化のための機能の集まりのことを仮想メモリと呼びます．

3.2 カーネルのアドレス空間

カーネルは，自身のためのページテーブルを持ちます．プロセスがカーネルに入ると，xv6 はカーネル用のページテーブルに切り替えます．そのあと，カーネルがユーザ空間へ返るときは，xv6 はユーザプロセス用のページテーブルに切り替え直します．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図示しています．あるファイル (kernel/memlayout.h) に，xv6 のカーネルのメモリレイアウトを規定する定数が宣言されています．

QEMU は，I/O デバイス（たとえばディスクインタフェース）を含むコンピュータをシミュレートします．QEMU は，デバイスのインタフェースを，メモリマップされた制御レジスタとしてソフトウェアに見せます．それらは，物理メモリの `0x80000000` 未満にマップされます．カーネルはメモリ領域を読み書きすることで，それらのデバイスとやりとりできます．4 章では，xv6 がどのようにデバイスとやりとりするかを説明します．

カーネルは，仮想空間の大部分を，そのままマップします．すなわち，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．たとえば，カーネル自身は，仮想アドレスでも物理アドレスでも `KERNBASE` にあります．カーネルは通常の（仮想アドレスを用いた）ページの読み書きと，（物理アドレスを用いた）PTE 操作の両方を行う必要がありますが，ダイレクトマッピングを用いることでそのコードを簡単にすることができます．ただし，ダイレクトマッピングされていない仮想アドレスもいくつかあります．

- トランポリンページ．仮想アドレス空間の先頭にマップされます．どのユーザのページテーブルでも同じマッピングを持ちます．4 章においてこのトランポリンページの役割を説明しますが，この部分はページテーブルの面白い利用例になっています．（トランポリンのコードを含む）物理ページは，カーネルの

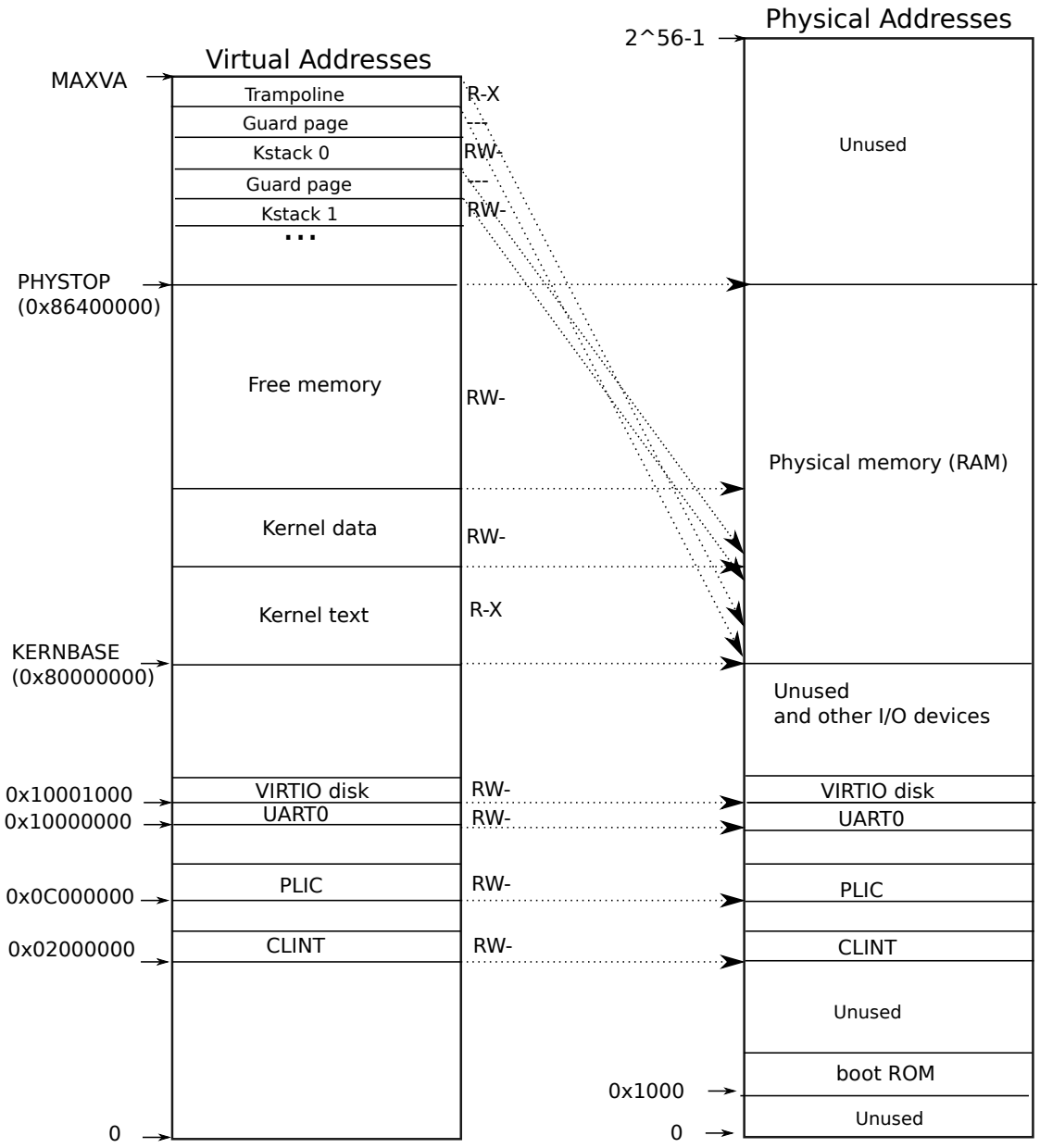

図 3.3　（左）xv6 のカーネルのアドレス空間．`RWX` は PTE の読み書き実行 (Read, Write, Execute) のパーミッションを示す．（右）xv6 が想定する RISC-V の物理アドレス空間.

仮想アドレス空間に 2 つマップされます．1 つは仮想アドレス空間の先頭であり，もう 1 つはカーネルの text 領域です.

- カーネルのスタックのページ．各プロセスは, 固有のカーネルスタックを持ち, 高位アドレス側にマップします．そうすることで，下位側に未割り当てのガードページを置くことができます．ガードページの PTE は無効です（すなわち `PTE_V` はセットされていません）．そうすることにより，カーネルがカーネル

のスタックをオーバーフローさせてしまった場合，高い確率でフォールトを起こし，カーネルがパニックを起こして止まることができます．もしガードページがなければ，オーバーフローはカーネルのメモリを上書きし，不正な処理をしてしまう可能性があります．それならば，パニックでクラッシュする方がマシです．

カーネルは上位メモリへのマッピングを介してスタックを利用しますが，ダイレクトマッピングされたアドレスを用いてもアクセスできます．ダイレクトマッピングだけを用意し，スタックをダイレクトマッピングされたアドレスで使うという設計もありえました．そのようにしてしまうと，ガードページを用意するには仮想アドレスのアンマップが必要となって使いづらくなります．明示的にアンマップしない限り，物理メモリを参照するためです．

カーネルはトランポリンのページとカーネルの text 領域を `PTE_R` と `PTE_X` のパーミッションでマップします．カーネルは，それらのページを読むことと実行ができます．カーネルは，それ以外のページは `PTE_R` と `PTE_W` でマップすることで，読み書きの両方ができるようにします．ガードページは無効としてマップされます．

3.3 Code: アドレス空間を作る

アドレス空間とページテーブルの操作に関するコードのほとんどは，`vm.c` (kernel/vm.c:1) に入っています．中心となる構造体は `pagetable_t` で，実はルートのページテーブル用ページへのポインタです．`pagetable_t` は，カーネルのページテーブルか，プロセス固有のページテーブルかのいずれかです．中心となる関数は，所与の仮想アドレスに対応する PTE を見つけるための `walk` と，新しいマッピングのために PTE を設置する `mappages` です．カーネルのページテーブルを操作する関数は `kvm` から, ユーザのページテーブルを操作する関数は `uvm` から始まる関数を使います．どちらでもない関数は両方に使います．`copyout` と `copyin` は，システムコールの引数として渡されたユーザの仮想アドレスを用いてデータを読み書きするために使います．それらにカーネルからアクセスするには，対応する物理アドレスを見つけるために明示的なアドレス変換が必要なので，`vm.c` に入っています．

ブートシーケンスの初期で，`main` は `kvminit` (kernel/vm.c:24) を呼ぶことでカーネルのページテーブルを作ります．これは，xv6 が RISC-V のページングを有効化する前ですので，アドレスは物理アドレスを直接参照します．`kvminit` はまず，ルートのページテーブル用のページをアロケートします．そのあと，`kvmmap` を呼ぶことで，カーネルが必要とするアドレス変換をインストールします．ここで言うアドレス変換

は次の領域を含みます: カーネルの命令列とデータ，PHYSTOP までの物理メモリ，およびデバイスに対応するメモリ領域.

kvmmap (kernel/vm.c:120) は mappages (kernel/vm.c:151) を呼び，ある仮想アドレスの範囲から，対応する物理アドレスの範囲へのマッピングをページテーブルにインストールします．この関数は，範囲に含まれる仮想アドレスに対し，ページの周期ごとに，個々に処理を行います．mappages は割り当てる仮想アドレスごとに walk を呼び，対応する PTE のアドレスを見つけます．続いて，対応する PTE を初期化します．すなわち，対応する物理ページ番号，所望のパーミッション (PTE_W, PTE_X, および PTE_R) を設定するとともに，PTE_V をセットしてその PTE を有効化します (kernel/vm.c:163).

walk (kernel/vm.c:74) は，RISC-V のページング用ハードウェアと同じ方法で，仮想アドレスに対応する PTE を見つけます (図 3.2 を見てください)．walk は，9 ビットずつ使って 3 階層のページテーブルを下っていきます．各階層では，仮想アドレスの 9 ビットを使って，次の階層の，もしくは最終的なページの PTE を見つけます (kernel/vm.c:80)．もし PTE が無効だった場合は，必要なページがアロケートされていないということです．もし alloc 引数がセットされていたら，walk は新しいページテーブル用ページをアロケートして，その物理アドレスを PTE に入れます．walk は，ツリーの最下層の PTE へ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レス空間にダイレクトマッピングされているという前提で動きます．たとえば，walk はページテーブルの階層を下っていくとき，次のページテーブルの（物理）アドレスを PTE から見つけ (kernel/vm.c:82)，そのアドレスを仮想アドレスとして用いて次の階層の PTE を得ます (kernel/vm.c:80).

main は kvminithart (kernel/vm.c:55) を呼んでカーネルのページテーブルをインストールします．それにより，ルートのページテーブル用ページの物理アドレスが，satp レジスタに書き込まれます．そのあと，CPU はページテーブルを用いてアドレス変換ができるようになります．カーネルはダイレクトマッピングを使っているので，次に実行する命令の仮想アドレスは，正しい物理メモリのアドレスを指しています.

procinit (kernel/proc.c:24) は，main から呼び出されるもので，プロセスごとのカーネルスタックをアロケートします．各スタックを，スタックのガードページの余裕を残しながら，KSTACK で生成する仮想アドレスにマップします．kvmmap は，新たにマッピングする PTE をカーネルページテーブルに追加したあと，kvminithart を呼んでそのカーネルページテーブルを satp にリロードすることで，ハードウェアに新しい PTE を知らせます.

RISC-V の各コアは，最近のページテーブルエントリを *Translation Look-aside Buffer* (*TLB*) にキャッシュするため，もしキャッシュされた TLB のエントリが期限切れになったときはそのことを教える必要があります．そうしないと，TLB に残っていた古いマッピングを利用してしまう可能性があります．古いマッピングが指している物理ページは，すでに別プロセスがアロケートして使っているかもしれません．そのようなことが起きると，あるプロセスが別プロセスのメモリを上書きできてしまう問題を引き起こします．RISC-V は `sfence.vma` 命令により，そのコアの TLB をフラッシュ (flush) することができます．xv6 は，`kvminithart` において `satp` レジスタをリロードしたあと，ユーザ空間に戻る前に，ユーザページテーブルに切り替えるためのトランポリンコードの内部で `sfence.vma` 命令を実行します (kernel/trampoline.S:79).

3.4　物理メモリのアロケート

カーネルは，実行時に，ページテーブル，ユーザメモリ，カーネルスタック，およびパイプバッファのためにメモリのアロケートと解放を行わなくてはなりません．

xv6 は，カーネルの終端から `PHYSTOP` の間の領域を，実行時のメモリアロケーションに利用します．4096 バイトのページ単位でアロケート・解放を行います．どのページが使用可能であるか追跡するために，ページ自体をリンクリストでつなぎます．ページをアロケートするときは，このリンクリストからページを 1 つ取り除きます．一方，ページの解放は，このリンクリストに解放されたページをつなぎ加えます．

3.5　Code: 物理メモリのアロケータ

ページを確保するアロケータ (allocator) は，`kalloc.c` (kernel/kalloc.c:1) に入っています．アロケータのためのデータ構造は，割り当てることができる物理メモリページの自由リスト (free list) です．リストの要素は `run` 構造体です (kernel/kalloc.c:17). アロケータは，そのデータ構造を入れるためのメモリを，どこから入手したらよいでしょうか？ `run` 構造体は，その自由なページの中に記憶します．そのページには他に記憶するものが無いので，`run` 構造体を入れておいてもよいのです．自由リストはスピンロックで保護されます (kernel/kalloc.c:21-24)．リストとロックは構造体でくるまれており，どのロックがどの領域を守っていることが分かるようになっています．当面は，ロックと `lock` および `acquire` の呼び出しのことは気にしないでください．ロックは 5 章で詳しく説明をします．

`main` 関数は，`kinit` を呼ぶことで，アロケータを初期化します (kernel/kalloc.c:27).

kinit は，自由リストにカーネルの終端から PHYSTOP までの全ページを入れて初期化します．本来ならば，コンフィギュレーション情報を解析することで，どれだけ物理メモリがあるのか調べなくてはいけません．xv6 はそうするかわりに，そのマシンは 128 MB の RAM を持つと決め打ちします．kinit は，freerange を呼ぶことで，自由リストにメモリを追加します．これは，内部的には，ページごとに kfree を呼んで実現します．PTE は，4096 バイト境界に整列した物理アドレスだけを指すことができます．freerange は，PGROUNDUP を用いることで，整列した物理アドレスだけを解放することを保証します．アロケータはメモリが空の状態で開始しますが，kfree により，ある程度がその管理に入ります．

アロケータは，アドレスを整数として扱って算術を行うことがあります (例：freerange における全ページの横断)．一方で，アドレスをポインタとして扱うことで，メモリへの読み書きを行うこともあります（例：各ページに入っている run 構造体の操作）．このように，アドレスを 2 つの意味で利用することが，アロケータのコードに C の型キャストがたくさん含まれる理由です．また，もう 1 つの理由は，メモリの解放・アロケートが，原理的にメモリの型を変更するためです．

kfree 関数 (kernel/kalloc.c:47) は，まず，対応するメモリの全バイトを 1 で埋めます．あるコードが解放済みのメモリを読み込んだ場合 (すなわち“dangling references”が置きた場合) に，過去のまともな内容ではなく，ゴミを読み込むようなります．そうすることで，問題のあるコードが早期に破綻することが期待できます．kfree は，対象のページを自由リストの先頭に付け加えるために次のようにします: pa を run 構造体へのポインタにキャストし，自由リストの古い先頭を r->next に記録し，そして自由リストを r にセットします．kalloc は自由リストの先頭の要素を取り除き，そしてその要素をリターンします．

3.6 プロセスのアドレス空間

各プロセスは，独立したページテーブルを持ちます．xv6 がプロセスを切り替えるときは，ページテーブルも合わせて切り替えます．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 番から開始し，MAXVA (kernel/riscv.h:349) まで増やすことができます．そうすることで，プロセスは原理的には 256 ギガバイトまでのメモリを使うことができます．

プロセスが xv6 にさらなるメモリを要求したら，xv6 はまず kalloc を使って物理ページをアロケートします．そのあと，プロセスのページテーブルに PTE を追加して，新たな物理ページを指し示すようにします．xv6 は，それらの PTE に対し PTE_W, PTE_X, PTE_R, PTE_U, および PTE_V をセットします．ほとんどのプロセスは，ユー

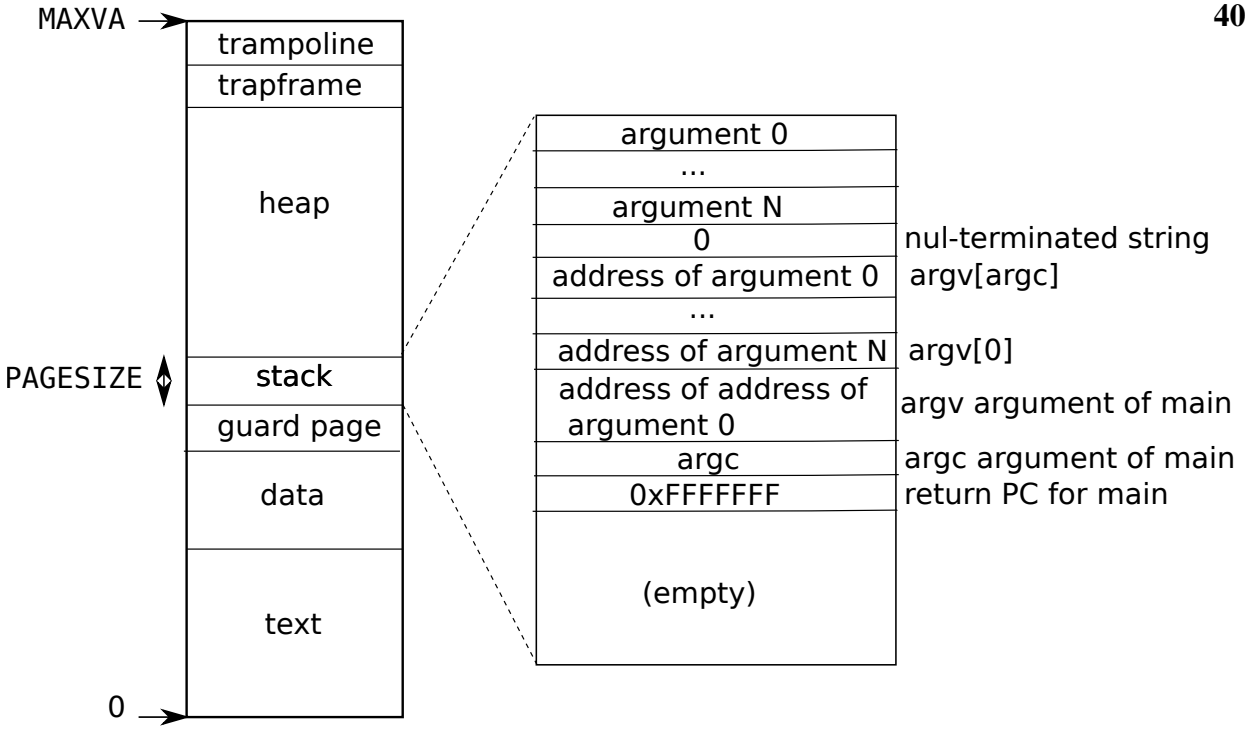

図 3.4　あるプロセスにおけるユーザのアドレス空間と，そのスタックの初期状態

ザのアドレス空間を使い切ったりはしません．xv6 は，未使用の PTE に対しては，`PTE_V` をクリア（ゼロ）のままにしておきます．

ページテーブルの使い方の良い例をいくつか見てみることにします．第一に，異なるプロセスのページテーブルは，ユーザのアドレスを異なる物理メモリのページに変換するので，各プロセスはプライベートなメモリを持つことができます．第二に，各プロセスは，実際の物理メモリは不連続であったとしても，0 から始まる連続した仮想アドレスを持つことができます．第三に，カーネルは，トランポリンコードをユーザのアドレス空間の先頭にマップします．そうすることで，物理メモリのあるページを，全てのアドレス空間に出現させることができます．

図 3.4 は xv6 において実行中のプロセスのユーザメモリのレイアウトを，詳しく絵にしたものです．スタックは 1 ページで，`exec` で作成された初期の内容を表示しています．コマンドライン引数の文字列と，それらへのポインタの配列がスタックの先頭にあります．そのすぐ下に置いてある値は，`main` から開始したプログラムが，あたかも `main(argc, argv)` として呼び出されたかのように見せるためにあります．

ユーザスタックが，アロケートしたスタックメモリからオーバーフローしたら気づけるように，xv6 は無効なガードページをスタックのすぐ下に置いておきます．もしスタックがあふれてスタックの下のアドレスを使おうとしたら，無効なマッピングへのアクセスがハードウェアにページフォルトを生じさせます．現実世界のオペレー

ティングシステムには，あふれる前に，ユーザスタックのメモリを自動でアロケートするものもあります．

3.7 Code: sbrk

`sbrk` はプロセスがメモリを増やしたり減らしたりするためのシステムコールです．これは，`growproc` 関数 (kernel/proc.c:224) で実装されています．`growproc` は，`n` が正か負かに応じて，`uvmalloc` もしくは `uvmdealloc` を呼び出します．`uvmalloc` (kernel/vm.c:236) は，`kalloc` で物理メモリをアロケートし，`mappages` によりユーザページテーブルに PTE を足します．`uvmdealloc` は `uvmunmap` (kernel/vm.c:176) を呼びます．それは，`walk` を用いて PTE を見つけ，`kfree` により対応する物理メモリを解放します．

プロセスのページテーブルは，ハードウェアに対して仮想アドレスのマッピング方法を指示するために利用しますが，そのプロセスにどの物理ページがアロケートされているかを示す唯一の記録でもあります．`uvmunmap` においてユーザメモリを解放するときに，ユーザページテーブルを見に行かなくてはいけないのはそのためです．

3.8 Code: exec

`exec` はアドレス空間の，ユーザに属する部分を作るためのシステムコールです．その部分を，ファイルシステムに記録されたファイルを用いて初期化します．`exec` (kernel/exec.c:13) は，7 章で説明するように，`namei` (kernel/exec.c:26) により名前付きのバイナリのパスを開きます．そのあと，ELF ヘッダを読み込みます．xv6 のアプリケーションは，広く使われる *ELF* フォーマットで記述されており，それは (kernel/elf.h) で定義されています．ELF バイナリは，ELF ヘッダである `elfhdr` 構造体 (kernel/elf.h:6) と，プログラムのセクションヘッダ `proghdr` 構造体 (kernel/elf.h:25) の列からなります．各 `proghdr` は，メモリにロードされるべきプログラムの 1 セクションを記述します．xv6 のプログラムには，プログラムセクションヘッダが 1 つしかありませんが，命令列やデータに対して別のセクションを割り当てるシステムもあります．

最初のステップは，ファイルが ELF バイナリであるかどうかの簡単なチェックです．ELF バイナリの先頭は，4 バイトのマジックナンバー `0x7F`, .code 'E', .code 'L', .code 'F', あるいは `ELF_MAGIC` です(kernel/elf.h:3)．もし ELF ヘッダが正しいマジックナンバーを持つならば，`exec` はそのバイナリが正しいフォーマットを持つと考えます．

exec は proc_pagetable (kernel/exec.c:38) により，ユーザによる割当のない新しいページテーブルを得ます．そのあと，各 ELF セグメントのメモリを uvmalloc (kernel/exec.c:52) でアロケートし，loadseg (kernel/exec.c:10) によりセグメントの内容をメモリにロードします．loadseg は，walkaddr を用いてアロケートしたメモリの物理アドレスを調べ，ELF セグメントの各ページを書きこみます．また，raedi により，ファイルの読み込みを行います．
exec が作る最初のプログラムである/init のプログラムセクションヘッダは以下の通りです．

```
# objdump -p _init
user/_init:     file format elf64-littleriscv

Program Header:
    LOAD off    0x00000000000000b0 vaddr 0x0000000000000000
                                   paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000840 memsz 0x0000000000000858 flags rwx
   STACK off    0x0000000000000000 vaddr 0x0000000000000000
                                   paddr 0x0000000000000000 align 2**4
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-
```

プログラムのセクションヘッダにおける filesz は，memsz よりも小さいときがあり，その差の領域はファイルから読むこむかわりに，ゼロで埋めなくてはいけません（C のグローバル変数のためです）．/init においては，filesz は 2112 バイトで，memsz は 2136 バイトです．よって，uvmalloc は 2136 バイトが入るのに十分な物理メモリを確保しますが，/init ファイルからは 2112 バイトしか読みません．

続いて，exec はユーザスタックを初期化します．そのために，1 ページだけ確保します．exec は，引数の文字列をスタックの先頭に 1 つずつコピーし，それらへのポインタを ustack により記録します．さらに，argv リストとして main に渡されるものの最後には，ナルポインタを置きます．ustack の最初の 3 エントリはフェイクのリターンアドレス，argc，および argv へのポインタです．

exec は，アクセス不可能なページを，先のスタックページのすぐ下に置きます．そうすることで，1 ページ以上使用しようとしたプログラムはフォルトを起こすようにします．このアクセス不可能ページは，長すぎる引数にも対処します．そのようなとき，exec が引数をスタックへコピーするのに使う copyout 関数 (kernel/vm.c:361) は，コピー先のページがアクセス不可能であることに気づいて 1 をリターンします．新しいメモリイメージを準備するあいだ，もし exec が不正なプログラムセグメントなどのエラーを見つけたら，bad というラベルにジャンプし，新しいイメージを解放し，そして 1 をリターンします．exec は，そのシステムコールが成功すると確認できるまで，古いイメージの解放は待たなくてはいけません．もし古いイメージが無くなっ

てしまうと，システムコールは 1 を返すことができないからです．exec のエラーが起きるのは，イメージを作る途中に限ります．もしイメージが完成したら，exec は新しいページテーブルを手に入れ (kernel/exec.c:110)，古いものを解放することができます (kernel/exec.c:114).

exec は ELF ファイルに入っているバイト列を，ELF ファイルが指定するアドレスに従ってメモリにロードします．ユーザもしくはプロセスは, 好きなアドレスを ELF ファイルに書くことができます．ELF ファイルは事故もしくは悪意によってカーネルを指すことがありうるので，exec には危険性があります．カーネルの不注意は，クラッシュから悪意を持ったカーネルの分離メカニズムの迂回（すなわちセキュリティのエクスプロイト）にいたるまでさまざまな結果をもたらします．xv6 は，いくつものチェックを行うことで，そのリスクを避けます．たとえば，`if(ph.vaddr + ph.memsz < ph.vaddr)` は，加算結果が 64 ビット整数からあふれるかどうかをチェックします．ありうる危険は次のようなものです: ユーザが選んだアドレスとして ph.vaddr が入った ELF バイナリを作ることができて，かつ ph.memsz が十分大きくて加算結果があふれて 0x1000 になってしまう場合，あふれた結果が正しい値に見えてしまう．xv6 の古いバージョンでは，ユーザのアドレス空間にはカーネルが（ユーザモードでは読み書きできない状態で）マップされていたので，ユーザはカーネルのメモリに対応するアドレスを選ぶことで， ELF バイナリのデータからカーネルへデータをコピーできました．この RISC-V のバージョンの xv6 では，そのようなことは起きません．カーネルは独自のページテーブルを持っており，loadseg はカーネルではなくプロセスのページテーブルに対するロードを行うためです.

カーネル開発者は，重要なチェックを怠ってしまうことがあります．そして，現実世界のカーネルでは，カーネル特権の奪取につながるチェックの不在に長い歴史があります．xv6 にも，ユーザレベルプログラムからカーネルに届くデータの検証が完全でない部分があり，悪意あるユーザプログラムが xv6 の分離を迂回するエクスプロイトを作ることができるかもしれません.

3.9 世の中のオペレーティングシステム

ほとんどのオペレーティングシステムと同じく，xv6 はページングハードウェアをメモリ保護とマッピングに使います．ほとんどオペレーティングシステムは，xv6 よりはるかに洗練されたページングの使い方をします．たとえば, xv6 にはディスクからのデマンドページング，fork におけるコピーオンライト，ページの遅延確保（lazily-allocated pages）, 自動で伸びるスタック，メモリマップドファイル (memory-mapped

file) などがありません.

RISC-V は物理アドレスのレベルでの保護もサポートしますが，xv6 はその機能を使いません.

メモリが大量にあるマシンにおいては，RISC-V のスーパーページ（サイズの大きいページ）にご利益があるかもしれません．一方，物理メモリが小さいときは，小さいページを使うほうが有利です．ページ確保やディスクへのページアウトを細かい粒度で行うことができるからです．たとえば，メモリを 8 キロバイト使うプログラムがあった場合，そのようなプログラムに，4 メガのスーパーページ全体を割り当てるのは無駄です．大きなページは，RAM をたくさん積んだマシンにおいてご利益があり，ページテーブル操作のオーバーヘッドを減らせる可能性があります.

xv6 カーネルには，`malloc` のような小さなオブジェクト用のメモリアロケータがありません．そのためカーネルは，動的なメモリ確保が必要な，洗練されたデータ構造を使うことができません.

メモリ確保は，大昔のホットトピックでした．予測できない将来のリクエストに備えて，限られたメモリを効率的に使いたい，というのが基本的な問題でした [6]．しかし今日では，空間的な効率性よりも速度が優先されます．また，より発展したカーネルは，xv6 のような 4096 バイトブロック単位ではなく，サイズの異なる小さなメモリブロックを大量に確保します．実世界におけるカーネルのアロケータは，大きなメモリだけでなく，小さなメモリのアロケートの面倒も見る必要があるのです.

3.10 練習問題

1. RISC-V のデバイスツリーを解析し，そのコンピュータが持つ物理メモリの量を求めてください.
2. `sbrk(1)` を呼ぶことで，アドレス空間を 1 バイト増加させるユーザプログラムを書いてください．そのプログラムを実行し，`sbrk` 呼び出しの前後におけるページテーブルを調査してください．カーネルがアロケートした空間はどれだけでしょうか？新しいメモリに対応する `pte` の中身はどうなっているでしょうか？
3. xv6 を改造して，カーネルがスーパーページを使えるようにしてください.
4. xv6 を改造して，ユーザプログラムがナルポインタをデリファレンスしたときにフォールトを受け取るようにしてください．すなわち，全てのプログラムにおいて，仮想アドレス 0 番がマップされないようにしてください.
5. Unix における `exec` は，伝統的に，シェルスクリプトを特別扱いします．実行するファイルが `\#!` から始まる場合，その 1 行目はファイルを解釈する

ためのプログラムと認識されます．たとえば，`myprog arg1` を実行するために `exec` が呼ばれ，`myprog` の 1 行目が `\#!/interp` であったら，`exec` は `/interp myprog arg1` というコマンドで `/interp` を実行します．xv6 に，以上の機能を実装してください．

6. カーネルに，アドレス空間のランダム化を実装してください．

第 4 章

トラップとデバイスドライバ

あるイベントをきっかけに，CPU がいつもの逐次的な命令列の実行を中断し，そのイベントを処理するための特別なコードに制御移すというシチュエーションが 3 つあります．1 つ目はシステムコールで，ユーザプログラムが `ecall` 命令を実行してカーネルに何かしてほしいと依頼する場合です．2 つ目は例外です．（ユーザもしくはカーネルの）コードが何か不正なことをしてしまった場合です．たとえばゼロで割り算をした場合や，無効な仮想アドレスを使用した場合です．3 つ目はデバイスによる割込で，デバイスが何か気づいて欲しいことがある場合です．たとえば，ディスクのハードウェアが，リクエストされていた読み込みや書き込み処理を完了したときなどです．

この本では，これらの 3 つのシチュエーションをトラップと総称します．通常，トラップ発生時にもともと実行していたコードは，あとから再開できなくてはならず，また，何か特別なことが生じたと気づくことがあってはなりません．すなわち，トラップは透過的であることを望みます．透過的であることは，割込について特に重要です．割り込まれることは，通常，割り込まれたコードにとって予想外のはずだからです．よって，通常の処理は次のようなものになります：トラップが強制的に処理をカーネルに移し，カーネルが再開に必要なレジスタやそのほかの状態を退避し，カーネルが適切なハンドラのコード（すなわちシステムコールやデバイスドライバの実装）を呼び出し，それが終わったら，退避しておいた状態をカーネルが復元し，トラップからリターンする．そうすることで，元のコードは中断したところから再開することができます．

xv6 カーネルは全てのトラップを処理します．システムコールを処理するのは当然でしょう．割込を扱うのも納得できます．分離をしなくてはいけないので，ユーザプロセスがデバイスを直接使えてはいけないからです．また，デバイスを制御するための状態は，カーネルだけが持っているためです．例外についても納得できると思いま

す．ユーザ空間届くどんな例外も，xv6 はその犯人を殺すことで対処するからです．

xv6 のトラップの処理プロセスは 4 ステージからなります．すなわち，(i) RISC-V CPU が行うハードウェア的なアクション，(ii) カーネルの C コードを指すアセンブリで書かれた「ベクタ」，(iii) そのトラップに対して何をするか決める C で書かれたトラップハンドラ，(iv) システムコールもしくはデバイスドライバのサービスルーチンです．3 つのトラップの種類の共通性を考えると，1 つのコードパスで全てのトラップを扱うこともできそうですが，3 つのケースごとにアセンブリのベクタや C のトラップハンドラを用意しておくと便利だということがわかります．3 つのケースとは，カーネル空間からのトラップ，ユーザ空間からのトラップ，そしてタイマ割込です．

この章の最後ではデバイスドライバについて議論します．デバイスの制御はトラップとは独立のトピックですが，カーネルとデバイスの相互通信は，割込によって行われることが多いので，この章に置いてあります．

4.1 RISC-V のトラップ機構

RISC-V にはいくつもの制御レジスタがあり，カーネルがそれらに書き込むことで CPU にどのように割込を扱うか指示したり，読むことで発生した割込について知ることができます．RISC-V のドキュメントには完全な内容が書いてあります [1]. `riscv.h` (kernel/riscv.h:1) には，xv6 が利用する定義が入っています．以下が，重要なレジスタの概要です．

- **stvec**: カーネルはトラップハンドラのアドレスをここに書き込みます．RISC-V はトラップが来たらそのアドレスにジャンプします．
- **sepc**: トラップが発生したら，RISC-V はプログラムカウンタをここに退避します (`pc` は，直後に `stvec` の値で上書きされてしまうからです)．`sret` (return from trap) 命令は `sepc` を `pc` にコピーします．カーネルは，`sepc` に書き込むことで，`sret` の飛び先を制御します．
- **scause**: RISC-V はトラップの理由を表す番号をここに置きます．
- **sscratch**: カーネルは，トラップハンドラの一番最初で便利になる値をここに置きます．
- **sstatus**: SIE ビットが，デバイス割込が有効かどうかを制御します．RISC-V は SIE がセットされるまで，デバイス割込を遅延させます．SPP ビットは，そのトラップがユーザモードから来たのかスーパーバイザモードから来たのかを示します．そして，`sret` が返るときのモードを制御します．

上記は，スーパーバイザモードで処理される割込に関連しており，それらはユーザ

モードで読み書きできてはいけません．マシンモードで割込を処理するために，また別の制御レジスタ一式があります．xv6 はそれらを，タイマ割込の特別な場合でのみ利用します．

トラップを強制しなくてはいけないとき，RISC-V はどのトラップに対しても次のことをします（ただしタイマ割込を除きます）

1. トラップがデバイスの割込で, `sstatus` の SIE ビットがゼロだったら，続く処理は行わない．
2. SIE をゼロにして割込を無効化する．
3. `pc` を `sepc` にコピーする．
4. 現在のモード（ユーザもしくはスーパーバイザ）を，`sstatus` の SPP ビットにセーブする．
5. 割込の原因を表すように `scause` をセットする．
6. スーパーバイザモードにする．
7. `stvec` を `pc` にコピーする．
8. 新しい `pc` で実行を開始する．

CPU が，以上のステップを単一のオペレーションとして行うのが重要です．以上のうち 1 つを取り除いた場合を考えてみてください．たとえば，CPU がプログラムカウンタを切り替えなかった場合などです．その場合，ユーザの命令列を実行しているにも関わらず，トラップがスーパーバイザモードへの切り替えを行ってしまう可能性があります．それは，ユーザとカーネルの分離を壊す可能性があります．たとえば，誰でもアクセスできる物理メモリの領域を指すように `satp` レジスタを書き換えてしまう場合などです．よって，ユーザプログラムではなく，カーネルがトラップのエントリポイントを指定することが重要なのです．

CPU は次のことはしてくれません: カーネルページテーブルの切り替え，カーネルのスタックへの切り替え，`pc` 以外のレジスタの退避．よって，カーネルが，必要に応じて上記の処理を行わなくてはいけません．トラップ時に CPU が最小限のことしか行わないのは，ソフトウェアに柔軟性を与えるためです．ページテーブルの切り替えが不要なシチュエーションもあり，そのときに性能を向上させることができます．

4.2 カーネル空間からのトラップ

xv6 カーネルが CPU 上で実行しているとき，2 種類のトラップが発生する可能性があります．すなわち，例外とデバイス割込です．それらのトラップに対して CPU がどのように応答するかは，前セクションで説明しました．

カーネルが実行中のとき，stvec は kernelvec (kernel/kernelvec.S:10) にあるアセンブリコードを指しています．xv6 はすでにカーネルにいますので，kernelvec は，satp はすでにカーネルページテーブルを指しており，スタックポインタは正しくカーネルスタックを指していると想定できます．kernelvec は，割り込まれたコードが問題なくあとから再開できるよう，全てのレジスタを退避します．

kernelvec によるレジスタの退避は，割り込まれたカーネルスレッドのカーネルスタックに対して行います．退避するレジスタは，そのスレッドに属しているためです．それは，トラップが別のスレッドへの切り替えを引き起こすときに特に重要です．その場合トラップは，切り替え先スレッドのスタックを用いてリターンを行います．そのあいだ，退避されたレジスタは，割り込まれたスレッドのスタックに安全に保存されます．

kernelvec は，レジスタを退避したあと，kerneltrap (kernel/trap.c:134) にジャンプします．kerneltrap はデバイス割込と例外のためにあります．kerneltrap は，デバイス割込のためのチェックと処理を，devintr (kernel/trap.c:177) を呼ぶことで行います．もし，そのトラップがデバイス割込でなかったとき，すなわち例外であったとき，それがカーネル内で生じるということはいつでも致命的エラーを意味します．

kerneltrap がタイマ割込により呼び出された場合で, かつあるプロセスのカーネルスレッド（つまりスケジューラではないスレッド）が実行中だったら, kerneltrap は yield を呼び出して，他のスレッドに実行を譲ります．いずれ，どれかのスレッドが yield を呼び出すことで，私達が今注目しているスレッドとその kerneltrap が再開できます．yield が何をするかは，6 章で説明を行います．

kerneltrap のしごとが終わったら，トラップによって割り込まれたコードにリターンしなくてはいけません．yield は，退避してあった sepc と sstatus の前モードを書き換えたかもしれないので，kerneltrap 開始時にまずそれらを復元します．kerneltrap はそれらの制御ジスタを復元してから，kernelvec (kernel/kernelvec.S:48) にリターンします．kernelvec は，スタックに退避されていたレジスタを取り出して復元し，sret を実行します．それにより，sepc が pc にコピーされ，割り込まれていたカーネルコードが再開します．

タイマ割込時に，kerneltrap が yield を呼んでいた場合，トラップからどのようにリターンするか考えてみるのはよいことです．

xv6 は，CPU がカーネル空間からユーザ空間に入るとき，CPU の stvec を kernelvec にセットします．これは，usertrap (kernel/trap.c:29) に書いてあります．すでにカーネルが実行をしているにもかかわらず，stvec が誤った値を保持している瞬間が存在するため，その間はデバイス割込を無効にしなくてはいけません．

便利なことに，RISC-V はトラップの処理を開始するときは常に割込を無効にします．そのため，xv6 が割込を再度有効にするのは，stvec に適切な値をセットしたあとです．

4.3 ユーザ空間からのトラップ

ユーザ空間にいるとき，次のような原因でトラップが発生する可能性があります: ユーザプログラムが (ecall 命令により) システムコールを呼んだとき，なにか不正なことをしたとき，あるいはデバイスが割込を行ったとき．ユーザ空間でトラップが生じたときの実行パスは，uservec (kernel/trampoline.S:16), から usertrap (kernel/trap.c:37); に続きます．一方，リターンするときは, usertrapret (kernel/trap.c:90) のあと userret (kernel/trampoline.S:16) となります．

ユーザ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しいです．なぜなら，satp がカーネルではなくユーザのページテーブルを指しており，またスタックポインタには不正（場合よっては悪意を持った）な値が入っているかもしれないからです．

RISC-V のハードウェアは，トラップ中にページテーブルを切り替えないので，stvec が指すトラップベクタの命令列は，ユーザページテーブルに入っている必要があります．さらにトラップベクタは，satp をカーネルのページテーブルに切り替えるので，その際にクラッシュしないように，トラップベクタの命令列は，カーネルのページテーブルとユーザのページテーブルで，同じアドレスにマップされていなくてはいけません．

xv6 は，上記の制約を，トラップベクタのコードを含むトランポリンページを使うことで解決します．xv6 はトランポリンページを，カーネルのページテーブルと全てのユーザのページテーブルにおいて，共通の仮想アドレスにマップします．この仮想アドレスは（図 2.3 と 図 3.3 で見たように）TRAMPOLINE です．トランポリンの中身は，trampoline.S でセットされます．そして，（ユーザコードを実行しているあいだは）stvec は uservec (kernel/trampoline.S:16) を指しています．

uservec が開始すると，全てのレジスタは割り込むコードの値を持つことになります．しかし，uservec は，satp をセットするためにいくつかのレジスタを書き換える必要があるとともに，レジスタを退避する先のアドレスを生成する必要があります．RISC-V は，sscratch レジスタによりそれを手伝ってくれます．uservec の先頭にある csrrw 命令は，a0 レジスタの中身を sscratch レジスタと入れ替えます．そうすることで，ユーザコードにおける a0 レジスタが退避されるとともに，uservec は a0 レジスタが使えるようになります．a0 には，カーネルが以前に

sscratch にセットした何らかの値が入っています.

uservec の次のしごとは，ユーザが使っていたレジスタを退避することです. ユーザ空間に入るまえに，カーネルは sscratch にプロセス固有の trapframe (kernel/proc.h:44) をセットしており，そこには（ほかのいろいろなものに加え）ユーザレジスタを退避する場所があります．satp はまだユーザのカーネルページを指しているので，uservec が動くためには，トラップフレームはユーザのアドレス空間にマップされていなくてはいけません．各プロセスを作るとき，xv6 はそのプロセス用のトラップフレーム用のページを 1 つ確保し，それが常にユーザの仮想アドレス TRAPFRAME にマップされるように調整します．そのアドレスは，TRAMPOLINE のすぐ下にあります．プロセスの p->tf はトラップフレームを指しています．ただし，それは物理アドレスであり，カーネルページテーブルを介してのみアクセスできます.

以上の準備をしておいたので，a0 と sscratch を取り替えたあと，a0 レジスタにはそのプロセスのトラップフレームへのポインタが入ります．uservec は，全てのユーザレジスタをそこに退避します．sscratch に移動された a0 レジスタの元の内容も退避します.

trapframe は次のような情報を保存しています: 現在のプロセスのカーネルスタックへのポインタ，現在の CPU の hartid, usertrap のアドレス，カーネルのページテーブルのアドレス．uservec はそれらの情報を利用して satp をカーネルページテーブルに切り替え，そして usertrap を呼び出します.

usertrap のしごとは kerneltrap と似ており，トラップの原因を調べ，処理し，そしてリターンすることです (kernel/trap.c:37)．上で述べたように，usertrap はまず，カーネルモードの kernelvec から届くトラップを処理するために，stvec を書き換えます．また sepc を再び退避します．usertrap においてプロセス切り替えが生じ，その結果 sepc が書き換わった可能性があるためです．もしトラップがシステムコールだった場合，syscall がそれを処理します．もしデバイス割込だった場合は devintr です．いずれでもないときは例外であるため，カーネルはその間違いを犯したプロセスを殺します．システムコールを処理するパスのときは，ユーザの pc に 4 を足します．なぜなら，システムコール呼び出しにおいて，pc は ecall 命令を指したままになっているからです．そのあと，usertrap は，プロセスが殺されたのか，あるいは CPU を yield しなくてはいけないのか（タイマ割込の場合）をチェックします.

ユーザ空間にリターンするための最初のステップは，usertrapret を呼び出すことです(kernel/trap.c:90)．この関数は，将来に起きるユーザ空間からのトラップに備えて RISC-V の制御レジスタを設定します．そのために，stvec を uservec を指すようにする，uservec が利用するトラップフレームのフィールドを用意する，sepc

を以前退避しておいたプログラムカウンタの値にセットする，などを行います．最終的に，usertrapret は（ユーザとカーネルのページテーブルの両方にマップされた）トランポリンページにある userret を呼び出します．そのようにするのは，userret 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 が userret を呼ぶとき，そのプロセスのユーザページテーブルを a0 レジスタ，TRAPFRAME を a1 レジスタに入れて渡します(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 の両方をマップしているのでした．しかし，カーネル空間の他のものはもうありません．繰り返しの説明となりますが，トランポリンページは，ユーザ空間とカーネル空間で同じアドレスにマップされているので，satp を切り替えたあとも，uservec は実行を続けることができます．trapret は，トラップフレームに退避されていた a0 レジスタの中身を sscratch にコピーして，近々 TRAPFRAME と取り替える処理に備えます．この時点から，userret が使うことができるのは，レジスタの中身と，トラップフレームの中身だけになります．続いて，userret は退避されていたユーザレジスタをトラップフレームから復元します．また，a0 と sscratch の最後の交換を行い，a0 レジスタの中身を復元するとともに，次のトラップのために TRAPFRAME を保存します．最後に，sret を用いてユーザ空間にリターンします．

4.4 タイマ割込

xv6 は，時計の管理と，compute-bound プロセスの切り替えのためにタイマ割込を用います．usertrap と kerneltrap における yield の呼び出しが，その切り替えを発生させます．タイマ割込は，各 RISC-V CPU に付属するクロックハードウェアからやってきます．xv6 は，CPU に周期的に割込をするように，このクロックハードウェアをプログラムします．

RISC-V では，タイマ割込をスーパーバイザモードではなく，マシンモードで受ける必要があります．RISC-V のマシンモードにはページングが無く，独立した制御レジスタを持っています．よって，通常の xv6 カーネルのコードをマシンモードで実行するのは現実的ではありません．その結果，xv6 は，タイマ割込を，上述のトラップメカニズムとは完全に別に扱います．

main が始まるまえ，start.c においてマシンモードでコードで実行するとき，タイマ割込を定期的に受けるように設定を行います(kernel/start.c:56)．そのためにやらなくてはいけないことの一部は，CLINT ハードウェア（コア固有の割込発生機）をプログラムして，ある遅延のあと割込を発生するようにします．やらなくてはならない別

のことは，スクラッチ領域 (scratch area) を設定することです．それはトラップフレームに対応するもので，タイマ割込のハンドラがレジスタを退避したり，CLINT レジスタのアドレスを見つけるのに利用するものです．最後に，start は timervec に mtvec を設定して，タイマ割込を有効にします．

タイマ割込は，ユーザやカーネルが実行中の，どんなタイミングでも発生することがあります．どんなにクリティカルな処理の途中であっても，カーネルがタイマ割込を止める方法はありません．そのため，タイマ割込のハンドラは，割り込まれたカーネルのコードの邪魔をしないことを保証しながらしごとをする必要があります．タイマ割込が RISC-V の「ソフトウェア割込」を発生させ，即座にリターンするというのが基本的な戦略です．RISC-V は，通常のトラップメカニズムに加えてソフトウェア割込を提供しており，それはカーネルが無効化することができます．タイマ割込が発生するソフトウェア割込のハンドラのコードは devintr (kernel/trap.c:197) にあります．

マシンモードでのタイマ割込のベクタは timervec (kernel/kernelvec.S:93) です．それは，いくつかのレジスタを (start が用意した) スクラッチ領域に退避し，CLINT に次にタイマ割込を発生するタイミングを指示した上で，RISC-V にソフトウェア割込を発生させます．そのあと，レジスタを復元してリターンします．タイマ割込に関わる C コードはありません．

4.5 Code: システムコールを呼ぶ

2 章では initcode.S が exec システムコールを呼ぶところまでを説明しました (user/initcode.S:11)．ユーザから呼び出したとき，カーネルの exec システムコールの実装にどうやってたどり着くのか見てみましょう．

ユーザコードは，exec の引数を a0 と a1 レジスタに入れるとともに，a7 レジスタにシステムコール番号を入れます．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 配列の添字となっています(kernel/syscall.c:108)．ecall 命令はトラップを発生させてカーネルに入り，これまで見てきたように，まず uservec と usertrap を実行し，そのあと syscall を呼びます．

syscall (kernel/syscall.c:133) は，トラップフレームから, かつて a7 レジスタに入っていたシステムコール番号を読み出し，システムコールテーブルを表引きします．最初のシステムコールでは，a7 レジスタには SYS_exec (kernel/syscall.h:8) が入っており，syscall はシステムコールテーブルにおける SYS_exec に対応するエントリである sys_exec を呼び出します．

syscall は，システムコール関数からの返値を p->tf->a0 に保存します．システ

ムコールがユーザ空間にリターンするとき，userret は p->tf に入っている値をマシンのレジスタにロードした上で，sret 命令によりユーザ空間へリターンします．その中で，システムコールハンドラの返値が，a0 レジスタへ入ります(kernel/syscall.c:140)．システムコールからの返値は，エラー時は負の数，正常時はゼロとするのが慣習です．システムコール番号が不正であった場合，syscall はエラーを表示するとともに，−1 を返します．

4.6 Code: システムコールの引数

あとの章では，特定のシステムコールの実装を見ます．この章は，システムコールそのもののメカニズムに興味があります．まだ説明していないメカニズムが 1 つあります．それは，システムコールの引数の見つけ方です．

RISC-V の C の呼び出し規約は，引数をレジスタで渡すことを決めています．システムコールの途中で，これらのレジスタ（すなわち，退避されたユーザレジスタ）は p->tf のトラップフレームでアクセスできます．argint, argaddr, および argfd 関数は，*n* 番目のシステムコール引数を，それぞれ整数，ポインタ，およびファイルディスクリプタとして返します．それらはいずれも，argraw を呼び出すことで，退避されたユーザレジスタを取得します(kernel/syscall.c:35)．

ポインタを引数として要求するシステムコールでは，カーネルはそれらのポインタを用いてメモリの読み書きをする必要があります．たとえば，exec システムコールは，ポインタ配列を引数として渡します．そのポインタは，ユーザ空間にある文字列引数を指しています．このようなポインタの取り扱いには 2 つの課題があります．第一に，バグっていたり，悪意を持っているユーザプログラムが，不正なポインタを渡すことでカーネルをだまし，ユーザメモリではなくカーネルメモリにアクセスをさせようとするかもしれません．第二に，xv6 のカーネルページのマッピングは，ユーザーページテーブルとは異なりますので，カーネルがユーザが渡したアドレスに読み書きをするのに，通常の命令を使うことができません．

ユーザ空間からの読み込みを安全に行わなくてはいけないカーネル関数がたくさんあります．fetchstr はその一例です(kernel/syscall.c:25)．exec のようなファイルのシステムコールは，fetchstr を利用してユーザ空間から文字列引数を取得します．fetchstr が呼ぶ copyinstr は，ユーザページテーブルから仮想アドレスを探し，カーネルが使えるアドレスに変換し，そしてそのアドレスにある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 にある仮想アドレス srcva から，最大で max バイトを読み出し，dst にコピーします．その

関数は，walkaddr (内部的に walk を呼びます) を用いてソフトウェア的にページテーブルを参照し，仮想アドレス srcva に対応する物理アドレス pa0 を得ます．カーネルは物理 RAM 全体をそのカーネル仮想アドレスにマップしているので，copyinstr は pa0 から dst へ，文字列を直接コピーすることができます．walkaddr (kernel/vm.c:97) は，ユーザが指定した仮想アドレスが，そのプロセスのアドレス空間に含まれることを確認することで，そのプログラムがカーネルをだまして別のメモリを読むことができないようにします．似た関数 copyout は，カーネルからユーザが指定したアドレスに対してデータをコピーします．

4.7 デバイスドライバ

ドライバとは，オペレーティングシステムに含まれるコードで，特定のデバイスを管理するためのものです．ドライバはデバイスハードウェアに処理をさせたり，それが終わったら割込をかけるように設定したり，結果として生じた割込を処理したり，また，そのデバイスの I/O を待っているプロセスとやりとりをしたりします．管理しているデバイスと並行して実行するので，ドライバのコードはトリッキーになりがちです．また，ドライバは，複雑だったりろくにドキュメント化されていなかったりする，デバイスのハードウェアインタフェースを理解できなくてはなりません．

オペレーティングシステムから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を発生するように設定します．カーネルのトラップを処理するコードは，デバイスがいつ割込を発生したのか認識し，そのドライバの割込ハンドラを呼び出さなくてはいけません．xv6 では，このような呼び出しは，devintr (kernel/trap.c:177) で行われます．

多くのデバイスドライバには，2 つの重要な部分があります．プロセスの一部として動くコードと，割込時に動くコードです．プロセスレベルのコードは，デバイスに対して I/O を行う，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，ハードウェアに処理の開始を指示することがあります（例: ディスクに対し，あるブロックの読み出しを要求する）．そのあと，そのコードは処理の完了を待ちます．いずれ，そのデバイスを処理を完了し，割込を発生させます．そのドライバの割込ハンドラは，（もしあるなら）どのオペレーションが完了したのか解明し，適切なものであれば待っていたプロセスを起床させます．場合によっては，順番待ちになっていた次のしごとをハードウェアに開始させます．

4.8 Code: コンソールドライバ

コンソールドライバは，ドライバ構造を端的に示す例です．コンソールドライバは，RISC-V に付属の *UART* シリアルポートのハードウェアから，人間が打鍵した文字列を受け取ります．ドライバは，1 行ごとに入力を蓄積するとともに，特殊な入力文字（バックスペースや Control-u）を処理します．シェルなどのユーザプロセスは，`read` システムコールを用いることで，コンソールからの入力を行単位で取得できます．QEMU において xv6 に対して打鍵すると，あなたの打鍵は QEMU がシミュレートする UART ハードウェアによって xv6 へ届きます．

ドライバが相手にするのは QEMU がエミュレートする 16550 チップです [8]．実際のコンピュータにおいて，16550 は，コンピュータもしくは端末に接続された RS232 シリアルリンクを管理します．QEMU を実行しているとき，16550 はあなたのキーボードとディスプレイにつながっています．

UART ハードウェアは，ソフトウェアにとって，メモリにマップされた制御レジスタに見えます．すなわち，ある物理アドレスが UART デバイスにつながっており，そのアドレスに読み書きをすると，（RAM ではなく）デバイスのハードウェアとやりとりできます．UART がマップされたアドレスは `0x10000000` もしくは `UART0` です (kernel/uart.c:22)．UART の制御レジスタはいくつもあり，いずれも 1 バイト長です．それらの，`UART0` からのオフセットは (kernel/uart.c:22) で定義されています．たとえば `LSR` レジスタには，ソフトウェアが読み出すことができる入力文字列があるかどうかを示すビット列が含まれます．それらの文字列（ある場合）は，`RHR` レジスタから読むことができます．読むたびに，UART ハードウェアは待機中の文字が入る内部の FIFO からそれを消去し，もし FIFO が空になれば `LSR` レジスタの “ready” ビットをゼロにします．

xv6 の `main` は，`consoleinit` (kernel/console.c:189) を呼ぶことで UART ハードウェアを初期化するとともに，入力時に割込を生成するように設定します (kernel/uart.c:34)．

xv6 のシェルは，`init.c` (user/init.c:15) が開いたファイルディスクリプタを介してコンソールから読み込みを行います．`read` システムコールの呼び出すと，カーネルは `consoleread` (kernel/console.c:87) を呼び出します．`consoleread` は（割込により）入力が来て `cons.buf` にバッファされるまで待ち，その入力をユーザ空間にコピーし，（行全体が届いたあとで）ユーザプロセスにリターンします．ユーザが行全体をまだ入力し終えていないあいだ，あらゆる読み込みプロセスは `sleep` を呼んで待ちます(kernel/console.c:103)．`sleep` については，6 章で説明をします．

ユーザが文字列を打鍵したら，UART ハードウェアは RISC-V に対して割込をするように要求します．RISC-V と xv6 はこれまで述べた方法で割込を処理し，xv6 のトラップ処理コードが `devintr` (kernel/trap.c:177) を呼びます．`devintr` は，RISC-V の `scause` レジスタを見て，その割込が外部デバイスから来たものだと気づきます．そのあと，`devintr` は，PLIC と呼ばれるハードウェアユニット [1] に問い合わせて，どのデバイスが割込をしたのか教えてもらいます(kernel/trap.c:186). もしそれが UART であった場合，`devintr` は `uartintr` を呼び出します．

`uartintr` (kernel/uart.c:84) は，UART ハードウェアから届いた待ち中の入力を全て読み出し，それを `consoleintr` に渡します．`uartintr` は，次の文字列が来るのは待ちません．将来の入力はまた別の割込を発生させるからです．`consoleintr` のしごとは，行全体が届くまで，入力文字列を `cons.buf` に蓄積することです．`consoleintr` は，バックスペースなどの特殊文字列の処理を行います．改行文字が届いたら，`consoleintr` は（もしあるなら）待っていた `consoleread` を起床させます．

起床後，`consoleread` は，`cons.buf` に 1 行分のデータを見つけ，それをユーザ空間にコピーし，（システムコールの機構を用いて）ユーザ空間へリターンします．

マルチコアのマシンにおいて，割込はどの CPU に届くこともありえます．どれにするかは，PLIC が管理しています．割込は，コンソールからの読み込みを待っているプロセスを実行している CPU に届くかもしれませんし，まったく関係ないことをやっている CPU に届くこともあります．割込ハンドラは，割り込まれたプロセスやコードについて考える必要はありません．

4.9　世の中のオペレーティングシステム

カーネルのメモリが全プロセスのユーザページテーブルにマップしてあれば，特別なトランポリンページは省略できます．その場合さらに，ユーザ空間からカーネル空間へのとトラップ時に，ページテーブルを切り替える必要もなくなります．その逆に，カーネル内のシステムコール実装は，ユーザメモリがマップされていることを利用して，ユーザポインタを直接デリファレンスできます．多くのオペレーティングシステムが，効率化のために，このアイディアを採用しています．xv6 がそうしないのは，ユーザから届いたポインタを不注意に扱うことによるセキュリティに関わるバグの可能性を減らすためです．また，ユーザとカーネルの仮想アドレスが重なってしまわないように管理する手間を減らすという利点もあります．

xv6 は，カーネル実行時でも，ユーザプログラムの実行時と同じように，デバイス割込とタイマ割込を受け付けます．タイマ割込は，たとえカーネルが実行中であって

も，タイマ割込ハンドラにより，スレッドを強制的に切り替えます（`yield` の呼び出し）．カーネルスレッドはたくさん計算をすることがありますので，カーネルスレッドに対しても公平にタイムスライスを行う能力は便利です．その一方，カーネルのコードが，タイマ割込によって中断され，そのあとに再開するかもしれないことは，xv6をいくらか複雑にしています．デバイスおよびタイマ割込をユーザコードの実行中しか受け付けないように限定すれば，カーネルをあるていど簡単にすることができます．

特定のコンピュータに搭載された全てのデバイスの全ての能力を引き出すことは大変です．デバイスはたくさんありますし，各デバイスにはたくさんの機能があるからです．また，デバイスとドライバ間のプロトコルは，複雑であったり，よくドキュメント化されていないこともあります．多くのオペレーティングシステムにおいて，ドライバは，カーネルのコアよりも多くのコードを占めます．

UART ドライバは，UART 制御レジスタを見ることで，データを 1 バイトずつ取得します．これは，ソフトウェアがデータの移動を駆動しているので，プログラムド *I/O* と呼ばれるパターンです．プログラムド I/O はシンプルですが，高いデータレートで使うには遅すぎます．たくさんのデータを高スピードで動かすデバイスは，通常，ダイレクトメモリアクセス *(DMA)* を使います．DMA デバイスハードウェアは，CPU に届くデータを直接 RAM に書き込んだり，CPU から出ているデータを RAM から直接読んだりできます．現代的なディスクやネットワークデバイスは DMA を使います．DMA デバイスのドライバは，RAM にデータを用意し，一度だけ制御レジスタに書き込みを行い，そのデータの処理を指示します．

デバイスが予測できないタイミングで，かつそこまで高くない頻度で CPU の注意を必要とするとき，割込は理にかなった方法です．しかし割込には大きな CPU オーバーヘッドがあります．よって，ネットワークやディスクのコントローラなどの高速なデバイスでは，割込を減らすテクニックが求められます．1 つのテクニックは，読み出し・書き込みリクエストの大きな塊ごとに，1 回だけ割込を行うというものです．また別のテクニックは，割込を完全に止めてしまい，そのデバイスが注意を求めていないか定期的にチェックするというものです．これは，ポーリングと呼ばれるテクニックです．ポーリングはデバイスの処理がすごく速いときに理にかなった方法ですが，そのデバイスが大部分の時間を待機している場合は，CPU の浪費となります．現在のデバイスの負荷に応じて，ポーリングと割込を動的に切り替えるドライバもあります．

UART ドライバは，届いたデータをユーザ空間にコピーする前に，まずカーネル内のバッファにコピーします．これは，データレートが低いときは理にかなった方法です．しかし，高速でデータを生成・消費するデバイスでは，そのような 2 重コピーが，劇的にパフォーマンスを低下させることがあります．オペレーティングシステム

によっては，ユーザ空間にあるバッファとデバイスハードウェアが直接やりとりできます．これは多くの場合， DMA を用いて行います．

4.10 練習問題

1. `uartputc` (kernel/uart.c:61) は，UART デバイスが先の出力文字を処理し終わるまでポーリングして待ちます．割込を使うように書き換えてください．
2. Ethernet カードのドライバを追加してください．

第 5 章

ロック

xv6 を含むほとんどのカーネルは，複数の活動をインタリーブ（訳注: またいで行ったり来たりすること）します．インタリーブが生じる原因の 1 つはマルチプロセッサのハードウェアです: xv6 の RISC-V がそうであるように，複数の CPU は独立に実行を行います．複数の CPU は物理的な RAM を共有しており，xv6 はそのことを利用して，全 CPU が読み書きできるデータ構造を管理します．そのような共有が行われることで，ある CPU が更新中のデータを，また別の CPU が読もうとすることがありえます．あるいは，複数の CPU が同じデータを同時に更新しようとすることすらありえます．注意深く設計を行わない限り，そのような並列アクセスはデータ構造の間違いや破壊を引き起こします．プロセッサが 1 つしかない場合であっても，カーネルは大量のスレッドを切り替えるので，それらの実行がインタリーブされることがあります．その結果，デバイスの割り込みハンドラが，割り込まれたコードと同じデータを変更しようとすることが，まずいタイミングで起きると，データの破壊を引き起こすことがあります．並行性 (concurrency) という言葉は，マルチプロセッサの並列実行，スレッドの切り替え，および割込により，複数の命令の流れがインタリーブされることを表します．

カーネルには，並行アクセスされるデータが山ほどあります．たとえば，2 つの CPU が同時に `kalloc` を呼び，並行して自由リストの先頭を取り出そうとすることがありえます．カーネル設計者は，並列処理で性能を上げたり，応答性を良くするために並行度を上げたいと望みます．その結果として，並行処理の正しさを検証するために，たくさんの労力を使うことになります．正しいコードにたどり着く方法はたくさんありますが，中でも理由付けが簡単なものがあります．並行処理における正しさを目的とした戦略と，それを支える抽象化は，並行性制御テクニック (concurrency control techniques) と呼ばれます．xv6 はシチュエーションに応じて，たくさんの並行性制御テクニックを利用しますが，さらにたくさんの可能性があります．この章は，

広く使われているテクニックであるロックについて述べます.

ロックは，相互排除を行うことで，1 度に 1 つの CPU だけがそのロックを取得できるように保証します．プログラマがロックと共有データアイテムを関連付け，コードがそのアイテムを使うときは必ず関連づいたロックを取得することにすれば，そのアイテムは 1 度に 1 つの CPU のみが使用することになります．これを，ロックがそのデータアイテムを保護する (protect)，と言います.

この章の残りは，xv6 にロックが必要な理由，xv6 がどうやってロックを実装しているか，そしてどうやってロックを使うかを説明します.

5.1 競合状態

なぜロックが必要か説明するための例として，マルチプロセッサにおいて，どの CPU からもアクセスできるリンクリストを考えましょう．このリストには，プッシュとポップを行うことができ，並行的に呼ばれることもありえます．xv6 のメモリアロケータは，実際そのように動いています．`kalloc()` (kernel/kalloc.c:69) は自由なページのリストからページをポップし，`kfree()` (kernel/kalloc.c:47) はその自由リストに新しいページをプッシュします.

並行するリクエストがなければ，リストの `push` 処理を次のように実装するかもしれません.

```
1       struct element {
2         int data;
3         struct element *next;
4       };
5
6       struct element *list = 0;
7
8       void
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，孤立して実行する限りは正しいです．しかし，複数のコピーが並行実

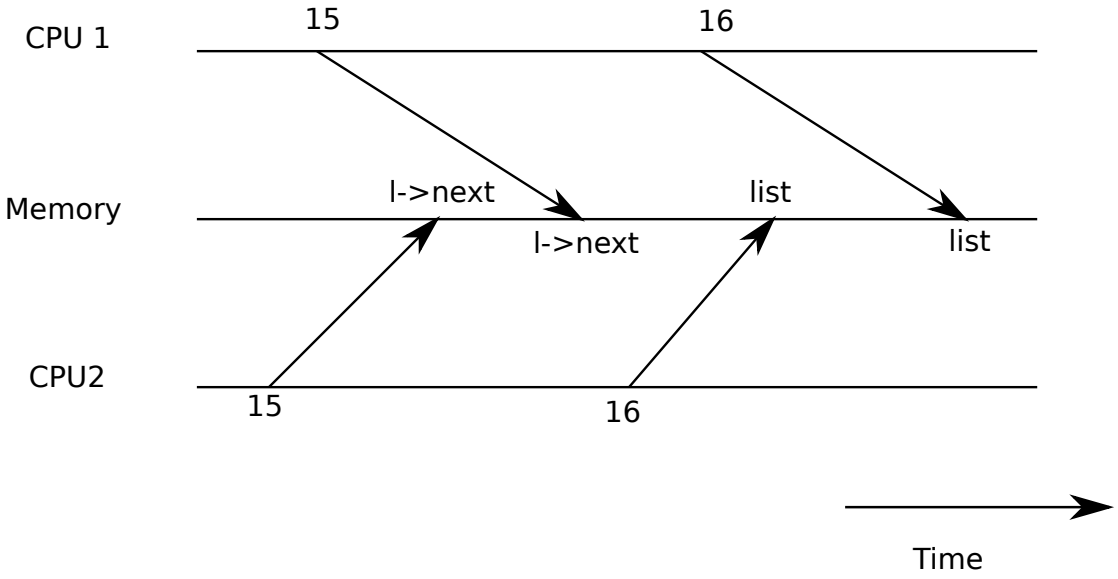

図 5.1　競合の例

行するとき，このコードは正しくありません．2 つの CPU が同時にプッシュをするとき，どちらかが 16 行目を実行する前に，両者が 15 行目を実行してしまうかもしれません (図 5.1 を見てください)．すると，両方のリスト要素の `next` が `list` の以前の値を指します．もし 16 行目で `list` への書き込みがリスト 2 分生じると，2 度目の書き込みが 1 度目を上書きしてしまいます．1 度目に代入した要素は喪失します．

16 行目における喪失は，競合状態の例です．競合状態とは，あるメモリの場所が並行的にアクセスされ，その少なくとも一方が書き込みであるときに生じます．競合はバグの兆候であり，更新したはずの内容の喪失（両方が書き込みだった場合）か，更新し終わっていないデータ構造の読み込みを起こします．競合の結末は関連する CPU がアクセスする正確なタイミングや，メモリシステムがそのメモリアクセスをどのように並べ替えるかに依存するので，競合に由来するエラーは再現してデバッグするのが難しいことがあります．たとえば，`push` をデバッグするために途中結果を出力しようとすると，実行のタイミングが変化して，競合が再現しなくなったりします．

競合を避けるための通常の方法はロックを使うことです．ロックは相互排除を保証して，`push` のセンシティブな行を，1 度に 1 つの CPU しか実行できないようにします．そうすることで，上記のシナリオは起きなくなります．上記のコードの正しくロックするバージョンは，数行を加えたものです（黄色でハイライトしてあります）．

```
6     struct element *list = 0;
7     struct lock listlock;
8
9     void
```

```
10      push(int data)
11      {
12        struct element *l;
13        l = malloc(sizeof *l);
14        l->data = data;
15
16        acquire(&listlock);
17        l->next = list;
18        list = l;
19        release(&listlock);
20      }
```

acquire と release で囲まれた領域をクリティカルセクションと呼びます．また，このロックは list を守っている，という言い方をします．

「あるロックがあるデータを守る」という言葉が実際に意味するのは，そのロックが，そのデータに関わるある不変量を守っているということです．不変量とは，あるオペレーション間で保たれるデータ構造の性質を表します．通常，あるオペレーションの正しさは，オペレーション開始時に不変量が正しいことを前提にします．そのオペレーションの途中で一時的に不変量が破れることがあるかもしれませんが，オペレーション完了時には不変量は再度保たれていなくてはいけません．たとえば，リンクリストのケースでの不変量は，「list はリストの最初の要素を指すこと」そして「各要素の next は，その次の要素を指すこと」です．push の実装は，一時的に不変量を破ります．17 行目において，l は次の要素を指しますが，list はまだ l を指していません（18 行目で直ります）．先ほど検討した競合状態が生じたのは，不変量が（一時的に）破れているときに，第二の CPU がリストの不変量に依存するコードを実行したためです．ロックを正しく使うことで，1 度に 1 つの CPU だけがクリティカルセクション内でデータ構造を処理することができるようになり，そのデータ構造の不変量が破れているときに，他の CPU がデータ構造のオペレーションを行うことはなくなりました．

ロックのことを，次のように考えることもできます：1 度に 1 つだけ実行するように，並行処理するクリティカルセクションをシリアル化 (Serialization) することで，不変量を保つもの（ただし，そのクリティカルセクションは，孤立して実行するときに正しいものとします）．あるいは，次のように考えることもできます: 同じロックで守ら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り，あるクリティカルセクションによる完全な変更のみが見える・不完全な変更は見えないようになっている．同じロックを同時に取ろうとすることを，複数のプロセスがコンフリクト (conflict) している，もしくは，ロックが競合している (*contension*) と言います．

なお，acquire を push の前方に動かしても 正しさは保たれます．たとえば，acquire の呼び出しを，13 行目の前に動かしても大丈夫です．そうした場合，malloc の呼び出しもシリアル化されるため，並列性が落ちます．あとの「ロックを使う」セクションでは，acquire と release の呼び出しをどこにいれるべきかのガイドラインを説明します．

5.2 Code: ロック

xv6 には 2 種類のロックがあります．スピンロックとスリープロックです．まずはスピンロックから説明します．xv6 は，spinlock 構造体でスピンロックを表現します(kernel/spinlock.h:2)．この構造体で重要なフィールドは locked で，ロックが使用可能ならばゼロ，使用中ならば非ゼロの値を持ちます．xv6 では，次のようなコードを実行してロックを取得します:

```
21    void
22    acquire(struct spinlock *lk) // does not work!
23    {
24      for(;;) {
25        if(lk->locked == 0) {
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

残念ながら，この実装ではマルチプロセッサにおける相互排除は保証できません．2 つの CPU が同時に 212 行目をにたどり着き，lk->locked がゼロだと気づき，26 行目を実行して 2 つの CPU とも ロックを取得することがありえます．その時点で，2 つの CPU がロックを取得したので，相互排除の性質が破れています．私たちがしなくてはいけないのは，212 行目と 26 行目をアトミックな（すなわち分離不可能な）ステップとして実行することです．

ロックは広く使われているので，マルチコアプロセッサは通常，212 と 26 行目をアトミックに実行する命令を提供します．RISC-V では，amoswap r, a です．amoswap 命令はメモリアドレス a の値を読み，レジスタ r の中身をそのアドレスに書き込みます．つまり，amoswap 命令は，それらのレジスタとメモリアドレスの中身を取り替えるのです．RISC-V は以上の流れを自動で行うとともに，特別なハードウェアを用いることで，読み書きを行うあいだに他の CPU がそのメモリアドレスを使わないように防ぎます．

Xv6 の acquire (kernel/spinlock.c:22) は，ポータブルな C ライブラリを使って，__sync_lock_test_and_set を呼び出します．この関数は，内部的に amoswap 命令を使います．返値は，古い（取り替えたあとの）lk->locked の値です．acquire 関数は，上記の取り替えをループでくるむことで，ロックが取得できるまで再試行（スピン）します．各くり返しにおいて，lk->locked に 1 を取り替えて入れるとともに，取り替える前の値を確認します．もし前の値が 1 だったら，他の CPU がロックを持っているということであり，また，amoswap が，lk->locked の値を書き換えなかったということでもあります．

ロックが取得できたら，acquire はデバッグのために，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で守られており，ロックを取得している状態でしか書き換えてはいけません．

release 関数(kernel/spinlock.c:46) は，acquire の逆のことをします．すなわち，lk->cpu の値をクリアし，ロックを解放します．概念上，release するには lk->locked にゼロを書き込めば十分です．しかし，C の標準は，1 つの代入を，複数のストア命令に翻訳することを許しています．よって， C における代入は，並行処理のコードという意味において，アトミックでない可能性があります．そのかわり，release は C のライブラリ関数である__sync_lock_release を呼び出します．これは，アトミックな代入を行うものです．この関数も，内部的には RISC-V の amoswap 命令を使います．

5.3 Code: ロックを使う

xv6 は競合状態を避けるために，さまざまな場所でロックを使います．すでに見た push とよく似たシンプルな例は kalloc (kernel/kalloc.c:69) と free (kernel/kalloc.c:34). です．これらの関数がロックを省略したら何が起きるかは，練習問題の 1 と 2 を試してみてください．おそらく，不正な挙動を引き起こすのは難しいのではないかと思います．ロックのエラーや競合が無いかを安定して試験はするのは難しいのです．xv6 に競合が残されていても不思議ではありません．

ロックを使う難しさは，ロックを何個使うかと，どのデータ・不変量をそのロックで保護するかを決めることにあります．基本的な原理がいくつかあります．第一に，ある CPU から書き込まれ，また別の CPU から読み書きされる変数があったら，両者が重ならないようにいつでもロックをつかうべきです．第二に，ロックは不変量を守るのだということを忘れないでください．もしその不変量が複数のメモリの場所を含むなら，通常，その全てを 1 つのロックで保護して，不変量が保たれるようにします．

上記のルールは，いつロックが必要かを述べていますが，ロックが不要な場合につ

いては何も言っていません．しかし，効率のためには，ロックしすぎないことが重要なのです．ロックは並列度を減らしてしまうからです．もし並列性が重要で無いならば，スレッドを 1 つだけにして，ロックのことは忘れてしまえばよいのです．ごく単純なカーネルでは，カーネルに入るためのロックを 1 つだけ持ち，カーネルから出るときに解放することで，マルチプロセッサでも同様のことを行うことができます（ただし，パイプの読み込みや `wait` は問題を起こすかもしれません）．もともとは単一プロセッサ用だったオペレーティングシステムの多くが，このアプローチ（「大きいカーネルロック」と呼ばれることがあります）でマルチプロセッサに対応しました．しかし，このアプローチは並列性を犠牲にしています．1 度に 1 つの CPU しかカーネルを実行できないのです．もしカーネルが何か重い計算をすることがあるならば，より粒度の細かいロックをたくさん使うことで，カーネルが同時に複数の CPU で実行できるようにするほうがより効率的です．

粒度の荒いロックの例として，xv6 の `kalloc.c` アロケータは 1 つの自由リストを 1 つのロックで保護しています．もし異なる CPU にある複数のプロセスが，同時にページをアロケートしようとしたら，どちらかは `acquire` 内でスピンして待たなくてはいけません．スピンは意味ある仕事ではないので，パフォーマンスを低下させます．もしロックの競合が CPU 時間の無視できない部分を閉めるならば，アロケータの設計を見直したら性能が改善するかもしれません．複数の自由リストをもち，それぞれにロックを持たせることで，真に並列してアロケーションを行えるようにするのです．

粒度の細かいロックの例として，xv6 はファイルごとに異なるロックを使います．そうすることで，異なるファイルを操作する複数のプロセスは，お互いのロックを待つことなく仕事を進めることができます．複数のプロセスが，同じファイルの異なる部分に対して同時に書き込みをしたいのであれば，ファイルをロックするスキームの粒度をさらに細かくすることもありえます．ロックの粒度の決定は，究極的には性能を測定するとともに，複雑さを考慮して行う必要があります．

続く章では，xv6 のいろいろな部分の説明をしますが，並行処理のために，xv6 がロックを使う例について述べることがあります．図 5.2 に，xv6 の全てのロックをあらかじめ見せておきます．

5.4　デッドロックとロック順序

カーネルにおいて，いくつかのロックを同時に取得するコードの実行パスがあるとき，全ての実行パスで同じ順序でロックを取得することが重要です．さもないと，デッドロックの危険性があります．xv6 において，ロック A と B を必要する 2 つの

ロック名	保護する対象
bcache.lock	ブロックバッファのキャッシュエントリのアロケーション
cons.lock	出力が混ざるのを避けるためのコンソールハードウェアへのアクセス.
ftable.lock	ファイルテーブルの file 構造体のアロケーション.
icache.lock	inode のキャッシュエントリのアロケーション.
vdisk_lock	ディスクハードウェアおよび DMA ディスクリプタのキューへのアクセス.
kmem.lock	メモリアロケーション.
log.lock	トランザクションログのオペレーション.
pipe の pi->lock	各パイプのオペレーション.
pid_lock	nexd_pid のインクリメント.
proc p->lock	プロセス状態の変更.
tickslock	ticks カウンタのオペレーション.
inode の ip->lock	各 inode とその中身に関するオペレーション.
buf の b->lock	各ブロックバッファのオペレーション

図 5.2 xv6 のロック

実行パスがあって，実行パス 1 は A から B という順序，実行パス 2 は B から A という順序でロックを取得するとしましょう．また，スレッド T1 が実行パス 1 を実行してロック A を取得し，スレッド T2 が実行パス 2 を実行してロック B を取得したとします．続いて T1 は ロック B を，T2 はロック A を取得しようとします．2 つのスレッドのロック取得は，永久にブロックします．なぜなら，どちらのスレッドにとっても他方のスレッドが必要なロックを取得しており，ロック取得がリターンするまでそのロックを解放しないからです．そのようなデッドロックを避けるには，全ての実行パスは，同じ順序でロックを取得する必要があります．ロックの取得順序をグローバルに統一しなくてはいけないことは，実質的に，ロックは各関数の仕様の一部であるということです．関数を使うプログラムは，ロックが指定された順序で呼出されるように，正しい順序で関数を呼び出さなくてはなりません．

xv6 には，2 つのロックを連鎖的に取得する場所がたくさんあります．プロセスごとのロック（各 proc 構造体に入っているロック）はその一例で，それは sleep の動作原理によります (6 章をみてください)．たとえば，consoleintr (kernel/console.c:143) は打鍵された文字を扱う割込ルーチンです．もし改行文字が来たら，そのコンソールの入力を待っている全てのプロセスを起床させなくてはいけません．そのために，consoleintr は cons.lock を取得して wakeup を呼び，待っているプロセスのロックを順に取得しては起床させます．その結果，デッドロックを避けるためのグローバルなロック順序として，各プロセスのロックを取得する前に cons.lock を取得するというルールがあります．ファイルシステムのコードに，xv6 における最長のロックの連鎖があります．たとえば，ファイルを作成するときは，次のようなたくさんのロックが必要です: そのディレクトリ，新しいファイルの inode, そのディスクの

ブロックバッファ，ディスクドライバの vdisk_lock, および呼び出し元プロセスの p->lock．デッドロックを避けるために，ファイルシステムのコードは常に前述の順序でデッドロックを取得します．

デッドロックを回避するためにロック順序を守ることは，驚くほど難しいことです．ロックの順序が，プログラムの論理的な構造と矛盾することすらあります．コードモジュール M1 からモジュール M2 を呼びたいのですが，モジュール M2 の中で使われるロックを，モジュール M1 の中で使われるロックよりも先に取得しなくてはいけない，というような場合です．どのロックを取得するか，事前にはわからない場合もあります．たとえば，次にどのロックを取得するか調べるために，まずあるロックを取得しなくてはいけない，ということがあります．そのようなシチュエーションは，ファイルシステムにおいてパス名の各部分を順次見ていくときに置きます．あるいは，wait や exit において，プロセスのテーブルから子プロセスを探すときも同様です．デッドロックの危険性は，粒度の細かいロックを使ってロックのスキームを組み立てるときの制約になります．ロックがたくさんあるということは，それだけデッドロックの可能性が高まることを意味するからです．デッドロックを避ける必要性は，カーネルの実装における主要な課題になることすらあります．

5.5　ロックと割込ハンドラ

xv6 のスピンロックには，スレッドと割込ハンドラで共有するデータを保護しているものもあります．たとえば，clockintr タイマ割込ハンドラは，カーネルスレッドが sys_sleep (kernel/sysproc.c:64) から読みだした ticks の分だけ，ticks (kernel/sysproc.c:64) を増やすことがあります．tickslock ロックが，2 つのアクセスをシリアル化しています．

スピンロックと割込が関わることには，潜在的な危険があります．sys_sleep が tickslock を取得しているときに，その CPU がタイマ割込に割り込まれたとします．clockintr は tickslock を取得しようとして，誰かが使っていることに気づき，解放されるまで待ちます．このシチュエーションでは，tickslock は永久に解放されません．解放できるのは sys_sleep だけですが，sys_sleep は clockintr がリターンするまで実行できないからです．よって CPU はデッドロックして，いずれかのロックを必要とするコードもまたフリーズします．

そのようなシチュエーションを避けるために，もしあるスピンロックが割込ハンドラで使われるならば，ある割込が有効な状態でそのロックを取得してはいけません．xv6 はさらに保守的で，CPU がロックを取得するときはいつも，その CPU の割込を無効にします．他の CPU で割込が発生し，その割込の acquire がスピンロックを解

放するのを待つことはありえますが，それは別の CPU でのことです．

xv6 は，その CPU がスピンロックを全て手放したら，割込を再び有効化します．入れ子になったクリティカルセクションを取り扱うためには，ロックの個数を覚えておく必要があります．`acquire` は `push_off` (kernel/spinlock.c:87) を，`release` は `pop_off` (kernel/spinlock.c:98) をそれぞれ呼び出すことで，その CPU におけるロックの入れ子の深さを調べます．もしその数がゼロになったら，一番外側のクリティカルセクションでの割込の有効状態を `pop_off` が復元します．`intr_off` と `intr_on` 関数は，RISC-V の命令を実行することで割込の無効化と有効化をそれぞれ行います．

`acquire` による `push_off` の呼び出しを，必ず `lk->locked` (kernel/spinlock.c:28) をセットする前に行うことは重要です．もし順序が入れ替わってしまうと，割込が有効化されたままロックを所持する瞬間ができてしまいます．その不幸なタイミングに割込が重なるとシステムがデッドロックします．同様に，`release` が `pop_off` を呼ぶのは，ロック(kernel/spinlock.c:63) を解放したあとに限ることが重要です．

5.6 命令とメモリの順序

ソースコードに書いてある順番でプログラムが実行されると考えるのが自然です．しかし，多くのコンパイラや CPU は，性能を上げるためにコードの順序を変更します．完了に何サイクルもかかる命令があったら，CPU その命令を先に実行することで，他の命令とオーバーラップして実行させ，CPU がストールしないようにします．たとえば，CPU は連続する命令 A と B の間に依存性が無いことに気づき，B の入力が A の入力よりも先に使用可能になったり，命令 A と B の実行をオーバーラップさせようとすると，命令 B を先に実行することがあります．コンパイラは，同様の並べ替えを行って，ソースコード上では先行する文の手前に命令列を出力することがあります．

コンパイラと CPU の並べ替えはいくつものルールを守って行われるので，書かれた順序で実行したときと結果は変わりません．しかし，そのルールは，並行処理するコードの結果が変わってしまうような並べ替えは許しており，マルチプロセッサにおいて誤動作を容易に引き起こします [2, 3]．CPU の並べ替えルールは，メモリモデルと呼ばれます．

たとえば，次の `push` のコードにおいて，コンパイラか CPU が 4 行目に対応するストア命令を，6 行目にある `release` のあとに動かしてしまうと大変なことになります．

```
1         l = malloc(sizeof *l);
2         l->data = data;
```

```
3         acquire(&listlock);
4         l->next = list;
5         list = l;
6         release(&listlock);
```

そのような並べ替えが起きると, 別の CPU がロックを取得して更新済み `list` を見れるが，`list->next` はまだ初期化されていないという瞬間ができてしまいます.

ハードウェアやコンパイラに並べ替えをしてほしくないことを伝えるために，xv6 は `acquire` (kernel/spinlock.c:22) と `release` `release` (kernel/spinlock.c:46) の両方で `__sync_synchronize()` を使っています．`__sync_synchronize()` はメモリバリアであり，そのバリアを超えてロードやストアを行わないようにコンパイラや CPU に伝えます．xv6 の `acquire` や `release` のバリアは，それが問題となるほとんどの場合でうまくいきます．xv6 は，共有されたデータの近くでロックを使用するためです．いくつかの例外は 8 章で説明します.

5.7　スリープロック

xv6 はロックを長く持ち続けなくてはいけないことがあります．たとえばファイルシステム（7 章）は，ディスクの中身を読み書きするあいだずっと，そのファイルをロックし続けます．そのようなディスクへのアクセスには何十ミリ秒もかかることもあります．長いあいだスピンロックを持ち続けると，別のプロセスがロックを取得しようとしていた場合にムダとなります．取得を試みるプロセスが長い間スピンして CPU を浪費するからです．スピンロックの別の欠点は，スピンロックを持っている間，そのプロセスは CPU を手放すことができないということです．もし手放すことができれば，そのロックがディスクを待っている間，別のプロセスが CPU を使うことができます．スピンロックを持っている間に CPU を手放すことが不正なのは，第二のスレッドがそのスピンロックを取得しようとするとデッドロックを起こすからです．`acquire` は CPU を手放さないので，第二のスレッドがスピンすると，第一のスレッドがロックを解放できなくなります．ロックを所持しながら CPU を手放すことは，スピンロックを所持する間は割込をオフすべしという要求にも違反します．以上のことを考えると，取得を待つあいだ CPU を手放すことができて，かつロックを持っているあいだでも CPU を手放せる（また割込を受け付ける）ようなロックが欲しくなります.

xv6 は，そのようなロックをスリープロックとして提供します．`acquiresleep` (kernel/sleeplock.c:22) は，6 章で述べる方法を用いて，待っている間 CPU を手放します．おおまかに言うと，スリープロックはスピンロックで保護されたフィールドを

持っており，acquiresleep が sleep を呼ぶと自動で CPU を手放すとともにスピンロックを解放します．その結果，acquiresleep を待っているあいだ，他のスレッドが実行を行うことができます．

スリープロックは割込を有効にしたままにするので，割り込みハンドラ内で使うことができます．acquiresleep は CPU を手放すかもしれないので，スリープロックはスピンロックで守られたクリティカルセクションの内部で使うことができません（その逆に，スリープロックで守られたクリティカルセクションの内部でスピンロックを使うことはできます）．

待ち時間が CPU を浪費するので，スピンロックは短いクリティカルセクションには最適です．それに対し，スリープロックは長く続く処理に向いています．

5.8　世の中のオペレーティングシステム

同期プリミティブや並行処理について長い間研究されてきましたが，ロックを用いたプログラミングは今も難しいままです．xv6 では採用していませんが，より高級な部品である同期キューなどで内部のロックを隠蔽するのが最適なことがあります．もしロックを使うプログラミングをするならば，競合状態を検出するツールを使うのは賢い選択です．ロックを必要とする不変量は簡単に見逃してしまうからです．

ほとんどのオペレーティングシステムは，POSIX スレッド (Pthread) をサポートしています．それを使うと，ユーザプロセスを，複数のスレッドを異なる CPU で並行実行できるようになります．Pthread はユーザレベルのロックやバリアなどをサポートします．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，あるスレッドがシステムコールでブロックしたときでも，同じプロセスの別スレッドは，その CPU で実行できなくてはいけません．また別の例として，あるスレッドが（たとえばメモリのマップやアンマップをして）そのプロセスのアドレス空間を変更したとき，同じプロセスの別スレッドが，ほかの CPU で実行している可能性があります．その場合，アドレス空間の変更を反映できるように，カーネルが調整を行う必要があります．

アトミックな命令を用いずにロックを実装することもできますが，それは高コストですので，ほとんどのオペレーティングシステムはアトミックな命令を使います．

たくさんの CPU が 1 つのロックを同時に取得しようとすると，ロックの処理は高価になります．もしある CPU のローカルキャッシュにロックがキャッシュされており，他の CPU がそのロックを使用しなくてはいけないとき，ロックが入ったキャッシュラインを書き換えようとするアトミック命令は，CPU をまたいでキャッシュのラインを移動させなくてはいけません．それと同時に，そのキャッシュラインの別のコ

ピーも期限切れ (invalidate) にするかもしれません．別の CPU のキャッシュからデータを取得することは，ローカルキャッシュから取得する場合とくらべて，桁違いに高コストです．

ロックに関連する浪費を避けるために，多くのオペレーティングシステムにはロック無しデータ構造やアルゴリズムがあります．たとえば，この章の最初に見せたようなリンクリストを，ロック無しでリストの探索ができ，アトミック命令 1 つでリストにアイテムを挿入するように実装することができます．ただし，ロック無しプログラミングは，ロックを使ったプログラミングよりも複雑です．ロックを使ったプログラミングがすでにもう難しいので，xv6 はロック無しプログラミングでさらに複雑にすることはしていません．

5.9 練習問題

1. `kalloc` (kernel/kalloc.c:69) において，`acquire` と `release` をコメントアウトしてください．`kalloc` を呼ぶカーネルのコードで問題が起きそうですが，どんな症状が起きるでしょうか？実際に xv6 を実行したとき，その予想した症状は観察できるでしょうか？ `usertests` を実行しているときではどうでしょうか？もし問題が起きないとしたら，それはなぜでしょうか？ `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，問題が起こりやすくなるか試してみてください．
2. （前の問題の `kalloc` のロックを元に戻したあと）かわりに `kfree` のロックをコメントアウトしたとします．今回はどんなまずいことが起きるでしょうか？ `kfree` のロックがないことは，`kalloc` のロックがない場合と比べて害が少ないといえるでしょうか？
3. もし 2 つの CPU が同時に `kalloc` を呼ぶと，一方はもう一方を待たなくてはならないはずで，これはパフォーマンス上よろしくありません．`kalloc.c` を改造して並列性を上げ，異なる CPU が同時に `kalloc` を呼んだときでも，お互いを待たずに進めるようにしてください．
4. ほとんどのオペレーティングシステムがサポートする POSIX スレッドを用いて，並列プログラムを書いてください．たとえば，並列化したハッシュテーブルを実装し，puts/gets がコア数の増加に対してスケールするか計測してください．
5. xv6 に，Pthreads のサブセットを実装してください．すなわち，ユーザレベルのスレッドライブラリを実装することで，ユーザプロセスが 2 つ以上のスレッドを持てるようにし，かつそれらのスレッドが異なる CPU で並列に実行でき

るようにしてください．ブロックするシステムコールを呼んだり，仮想アドレス空間を変更するスレッドを扱える設計を考えてください．

第 6 章

スケジューリング

ほぼ全てのオペレーティングシステムは，コンピュータに搭載された CPU よりも多い数のプロセスを走らせます．そのため，プロセス間で CPU をタイムシェアするための計画が必要です．CPU の共有は，ユーザプロセスにとって透過的であることが理想です．一般的なアプローチは，ハードウェアの CPU に対してプロセスをマルチプレクスする（切り替える）ことで，各プロセスが専用の CPU をもっているかのようにみせることです．この章では，xv6 が，どのようにしてそのようなマルチプレクスを行うのかを説明します．

6.1　マルチプレクス

あるプロセスから別のプロセスへと切り替えることでマルチプレクスするシチュエーションは 2 つあります．第一に，xv6 が `sleep` や `wakeup` 機能により切り替えを行う場合があります．プロセスがデバイスやパイプの I/O の完了を待つとき，子プロセスの完了を待つときや，`sleep` システムコールで待つときなどが例です．第二に，スリープせずに長時間計算を行うプロセスに対応するため，xv6 が周期的に強制的な切り替えを行います．このようにマルチプレクスを行うことで，各プロセスが専用の CPU を持っているようにみせます．これは，メモリアロケータとハードウェアのページテーブルを使うことで，各プロセスが専用のメモリを持っているようにみせることに似ています．

マルチプレクスの実装には，いくつかの課題があります．第一に，どうやってプロセスから別のプロセスに切り替えを行うかということです．コンテキストスイッチのアイディアは単純ですが，その実装は xv6 の中でもっとも不透明なものの 1 つです．第二に，透過性を保ちながら強制的に切り替えを行うにはどうしたらよいかということです．xv6 は，コンテキストスイッチを駆動する標準的な方法であるタイマ割込に

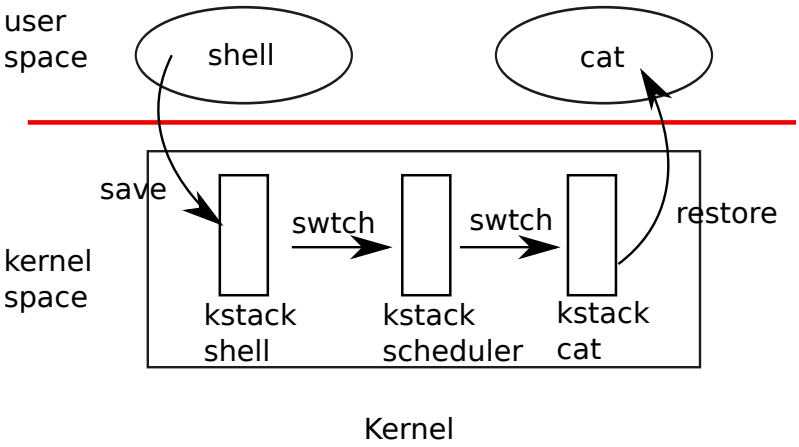

図 6.1　あるユーザプロセスから別のユーザプロセスへの切り替え．単独の CPU で xv6 が動作する場合（すなわちスケジューラスレッドが 1 つしか無い場合）

よる切り替えを採用します．第三に，たくさんの CPU がプロセスの間を並行的にスイッチするので，競合を防ぐためにロックの計画が必要だということです．第四に，プロセスから出るときにはメモリなどの資源を解放する必要がありますが，それをプロセス自身が行うことはできないということです．なぜなら（たとえば），プロセスは（まだ使っているので）自身のカーネルスタックを解放することができません．第五に，システムコールが正しいプロセスのカーネルの状態に影響を与えることができるように，マルチコアマシンの各コアが，どのプロセスを実行中か覚えておかなくてはいけないということです．最後に，`sleep` と `wakeup` を使うことで，プロセスが自発的に CPU を手放してスリープしてイベントを待ち，いずれ別のプロセスがそのプロセスを起床させます．そのとき，競合により起床の通知を見逃してしまわないように注意が必要です．xv6 は，それらの課題をできる限り単純な方法で解決しますが，その結果できたコードはいずれにせよトリッキーです．

6.2　Code: コンテキストスイッチ

図 6.1 に，あるユーザプロセスから別のユーザプロセスへの入り替えに関わるステップの概略を示します．すなわち，(i) ユーザからカーネルへの切り替え（システムコールあるいは割込をきっかけに，古いプロセスのカーネルスレッドに入る），(ii) 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, (iii) 新しい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，(iv) ユーザレベルプロセスへのトラップからのリターンというステップで進みます．xv6 のスケジューラには，CPU ごとに専用のスレッド（レジスタとスタックが保存される）があります．古いプロセスのカーネルスタックでスケジューラを実行するのは安全ではないことがあるからで

す．そのような例を exit で見ます．このセクションでは，カーネルスレッドからスケジューラスレッドへの切り替えの仕組みを説明します．

あるスレッドから別のスレッドに切り替えるには，古いスレッドの CPU レジスタの退避と，新しいスレッドのかつて退避されたレジスタの復元が必要です．スタックポインタとプログラムカウンタが退避され復元されるということは，スタックと実行しているコードの両方が切り替わるということです．

swtch 関数がカーネルスレッドの切り替えに関わる退避と復元を行います．swtch は, スレッドのことを直接は知らず, 単にレジスタの集まり（コンテキストと呼ばれる）の退避と復元を行います．プロセスが CPU を手放すときがくると，そのプロセスのカーネルスレッドは swtch を呼んでコンテキストを保存し，それからスケジューラのコンテキストにリターンします．コンテキストは, context 構造体(kernel/proc.h:2) に入っており，それ自体がプロセスの proc 構造体もしくは CPU の cpu 構造体に入っています．swtch は 2 つの引数をとります．struct context *old と struct context *new です．swtch は現在のレジスタを old に退避し，new からレジスタをロードしてリターンします．

swtch がスケジューラにいたるまでの道のりを追いかけてみましょう．4 章において，割込の結果として usertrap が yield を呼ぶことがあることを見ました．yield は続いて sched を呼び，これがさらに swtch を呼んで現在のコンテキストを p->context に退避するとともに，以前 cpu->scheduler (kernel/proc.c:494) に退避されたスケジューラのコンテキストへ切り替わります．

swtch (kernel/swtch.S:3) は呼び出された側が退避すべきレジスタ (callee-saved register) だけを退避します．呼び出し側が退避すべきレジスタ (caller-saved register) は，呼び出しを行う C のコードが，必要に応じてスタックに退避します．swtch は，context における各レジスタのフィールドへのオフセットを知っています．swtch は，プログラムカウンタは退避しません．そのかわり，swtch は，swtch 関数からのリターンアドレスが入った ra レジスタを保存します．swtch がリターンすると，復元された ra レジスタに入っていたアドレスにリターンします．それは，新しいスレッドが以前 swtch を呼び出した命令を指しています．さらに，swtch は新しいスレッドのスタックを用いてリターンします．

私たちの例では，sched が swtch を呼び出すことで，各 CPU のスケジューラのコンテキストである cpu->scheduler に切り替わっていました．このコンテキストは，スケジューラが swtch (kernel/proc.c:460) を呼んだときに退避されたものです．これまで私たちが追いかけてきた swtch がリターンすると，それは sched ではなく scheduler にリターンします．そしてそのスタックポインタは，現在の CPU のスケジューラのスタックを指しています．

6.3 Code: スケジューリング

前のセクションでは，swtch の低レベルの詳細を見ました．ここからは，swtch はあるものとして，あるプロセスのカーネルスレッドが，スケジューラを経由して別プロセスに切り替わる様子を見てみましょう．スケジューラは，CPU ごとの特殊なスレッドとして存在し，それぞれがスケジュールのための関数を実行しています．この関数は，次に実行するプロセスを選ぶ責任があります．CPU を手放そうとしているプロセスは，自身のプロセスロックである p->lock を取得し，ほかのロックを全て解放し，自身の状態 (p->state) を更新し，そして sched を呼びます．続いて，sleep や exit と同様に yield (kernel/proc.c:500) を呼びます．それについては，あとで詳しく説明します．sched はそれらの条件と (kernel/proc.c:484-489), その条件が意味するところをダブルチェックします．ロックを所持しているので，割込は無効になっているはずです．最後に，sched は swtch を呼んで現在のコンテキストを p->context に退避するとともに，cpu->scheduler に入ったスケジューラのコンテキストに切り替わります．swtch は，あたかも scheduler が呼んだ swtch からリターンしたかのように，スケジューラのスタックを用いてリターンします(kernel/proc.c:460)．スケジューラは for ループを進み，実行すべきプロセスを見つけ，それに切り替わります．そして以上をくり返します．

いま，swtch の呼び出しにおいて，xv6 は p->lock を所持していました．swtch の呼び出し元は，その時点でロックを所持していなくてはならず，そのロックは切り替え先のコードに持ち越されます．これは，ロックにおいて普通ではないことです．普通であれば，ロックを取得したのと同じスレッドが，その解放にも責任を持つことで，正しさの理由付けを簡単にしています．コンテキストスイッチでは，その慣習を破る必要があります．なぜなら p->lock はプロセスの state と context フィールドに関する不変量を保護しており，それは swtch の実行中は正しくないからです．swtch 時に p->lock を取得していなかったとしたら次のような問題が生じます: yield が状態を RUNNABLE にしたあと，しかし swtch がそのカーネルスタックの使用を止めるまえに，他の CPU がそのプロセスを実行しようとする可能性があります．そうすると，2 つの CPU が同じスタックで実行を行うことになり，それはまずいことです．

カーネルスレッドは，いつも sched で CPU を手放し，スケジューラの同じ場所に切り替わります．そしてそのスケジューラは（ほぼ）いつでも，以前 sched を呼んだカーネルスレッドに切り替わります．よって，もし xv6 がスレッドを切り替える箇所を表示すると，単純なパターンを見ることができます．(kernel/proc.c:460), (kernel/proc.c:494) などです． 2 つのスレッド間のこのようなスタイルの定まった切り

替えは，コルーチン (coroutine) と呼ばれることがあります．この例では，sched と scheduler はお互いのコルーチンです．

スケジューラの swtch の呼び出しが sched にたどり着かないケースが 1 つあります．新しいプロセスが最初にスケジュールされるとき，それは forkret (kernel/proc.c:512) から開始します．forkret は，p->lock を解放するためにあります．それとは別に，その新しいプロセスが usertrapret から開始する場合もありえます．

scheduler (kernel/proc.c:442) は単純なループを実行します．実行するプロセスを見つける，それが CPU を手放すまで実行する，繰り返す．スケジューラはプロセステーブルをループして，実行可能なプロセス，すなわち p->state == RUNNABLE であるものを探します．もしそのようなプロセスを見つけたら，CPU 固有の現在のプロセスを表す変数である c->proc に値をセットし，そのプロセスを RUNNING にし，そして swtch を呼んで実行を開始します (kernel/proc.c:455-460)．

スケジュール用のコードの構造は，プロセスに関するいくつかの不変量を強制するものであり，もし不変量が敗れるときは常に p->lock を取得するものであると考えることができます．不変量の 1 つは，そのプロセスが RUNNING であるならば，タイマ割込による yield で安全にそのプロセスから抜けることができるというものです．それはつまり，CPU レジスタがそのプロセスのレジスタの値をとっており（つまり swtch が context に移動しておらず），c->proc がそのプロセスを指していることを意味します．さらに別の不変量は，プロセスが RUNNABLE であれば，空いた CPU のスケジューラが実行しても安全であるというものです．それは，p->context がそのプロセスのレジスタを保存していること (本物のレジスタには入っていないこと)，そのプロセスのカーネルスタックを使っている CPU が無いこと，そしてどの CPU でも c->proc はそのプロセスを指していないことを意味します．p->lock を所持するあいだ，それらの性質が満たされない瞬間があることを思い出してみてください．

上記の不変量を保つことが，xv6 があるスレッドで p->lock を取得し，別のスレッドで解放することを頻繁に行う理由です．たとえば，yield におけるロック取得や，scheduler における解放です．yield がプロセスの状態を RUNNABLE に変更しようとし始めたら，不変量が回復するまでロックを保持し続けなくてはいけません．正しく解放できるもっとも近いポイントは，(自身のスタックで実行する) scheduler が c->proc をクリアしたあとです．同様に，スケジューラが runnable なプロセスを running に変更しはじめたら，(swtch のあと，yield の中など) でカーネルスレッドが完全に実行されるまでロックを解放することはできません．

p->lock は他のものも保護しています．exit と wait の相互作用は，起床の見逃しを防ぐしくみであり (6.5 節を見てください)，抜けようとしているプロセスと，そ

の状態の読み書きを行う別プロセス（たとえば，exit システムコールが p->pid を見て p->killed を書く場合 (kernel/proc.c:596)) の競合を避ける仕組みでもあります．分かりやすさや性能のために，p->lock の別の機能が分割できないかどうか考えてみると良いかもしれません．

6.4 Code: mycpu と myproc

xv6 では，現在プロセスの proc 構造体へのポインタがよく必要になります．単一プロセッサにおいては，現在の proc 構造体を指すグローバル変数を持っておくこともできますが，マルチコアのマシンではできません．各コアが異なるプロセスを実行するからです．各コアが固有に持つレジスタを使うことで，その問題を解決することができます．そのようなレジスタの 1 つを使うことで，コアごとの情報を効率的に見つけることができるようになります．

xv6 は，各 CPU ごとに cpu 構造体を管理します(kernel/proc.h:22). この構造体には次のような情報が記録されます: その CPU で現在実行しているプロセス（もしあるなら），その CPU のスケジューラスレッドの退避したレジスタ，（割込の無効化に必要な）入れ子になったスピンロックの数．mycpu 関数 (kernel/proc.c:58) は，現在の CPU の cpu 構造体へのポインタを返します．RISC-V はそれぞれの CPU に識別番号として hartid を与えます．xv6 は，カーネルにいるあいだは，tp レジスタにその CPU の hartid を入れておきます．mycpu は tp レジスタの値をインデックスとして cpu 構造体の配列を引くことで，自身のための cpu 構造体を見つけることができます．

CPU の tp が常にその CPU の hartid を持つように保証するのは少し入り組んでいます．CPU の起動シーケンスの早い段階，まだマシンモードにいるあいだに，mstart が tp レジスタを設定します(kernel/start.c:45). usertrapret が，トランポリンページにおいて tp を退避します．ユーザプロセスが tp を書き換えてしまうかもしれないからです．最終的に，ユーザ空間からカーネル空間に入るとき，uservec が退避しておいた tp を復元します(kernel/trampoline.S:70)．xv6 が直接 hartid を読み取れればよいのですが，それは（スーパーバイザモードではなく）マシンモードでしかできません．

cpuid と mycpu の返値は賞味期限が短いです．もしタイマが割り込んでそのスレッドが CPU を手放し，そしてそのあと別の CPU に移動したとすると，以前に取得した返値はもう正しくありません．そのような問題を避けるために，それらの関数を呼び出すときは割込を無効化し，返った cpu 構造体を使い終わるまでは有効化してはいけません．

myproc 関数 (kernel/proc.c:66) は，その CPU で現在実行中のプロセスの proc 構造

体へのポインタを返します．myproc は割込を無効化し，mycpu を呼び，cpu 構造体から現在のプロセスへのポインタ (c->proc) を取得し，そして割込を再度有効化します．myproc の返値は，割込が有効なときでも安全に使うことができます．もしタイマ割込がそのプロセスを別の CPU に移動させた場合でも，proc 構造体へのポインタは変わらないからです．

6.5 sleep と wakeup

スケジューリングとロックは，あるプロセスを別のプロセスから隠す手伝いをしますが，プロセス間が意図的に相互通信をするところは抽象化されていませんでした．その問題を解決するために，たくさんのメカニズムが発明されました．xv6 はそのうち，”sleep and wakeup” という方法を用います．これを使うと，あるプロセスはイベントを待つために眠り，そのイベントが起きたら別のプロセスが起床させることができます．”sleep and wakeup” はよく，*sequence coordination* または *conditional synchronization* メカニズムと呼ばれます．

例示のために，生産者と消費者の調整をする，セマフォ [4] と呼ばれる同期メカニズムを考えてみましょう．セマフォは，ある数を管理しており，2 つのオペレーションを提供します．（生産者のための）V オペレーションは，その数を 1 つ増やします．（消費者のための）P オペレーションは，その数が非ゼロになるまで待ち，そのあとで数字を 1 つ減らしてリターンします．もし生産者と消費者のスレッドが 1 つずつしかなく，それぞれが別の CPU で実行しており，かつ CPU がアグレッシブな最適化をしないならば，以下が正しい実装です．

```
100   struct semaphore {
101     struct spinlock lock;
102     int count;
103   };
104
105   void
106   V(struct semaphore *s)
107   {
108     acquire(&s->lock);
109     s->count += 1;
110     release(&s->lock);
111   }
112
113   void
114   P(struct semaphore *s)
```

```
115     {
116         while(s->count == 0)
117           ;
118         acquire(&s->lock);
119         s->count -= 1;
120         release(&s->lock);
121     }
```

上記の実装は高コストです．生産者が滅多に動かないとき，消費者はほとんどの時間を，while ループのなかで非ゼロの値を期待しながらスピンするのに浪費してしまいます．消費者の CPU は，s->count を何度もポーリングしてビジーウェイトするよりも，もっと生産的なことをすべきです．ビジーウェイトを避けるには，消費者が CPU を手放し，V が数を増やしたあとにかぎり再開する方法が必要です．

そのような方向にすすめるためのステップは次の通りです（ただし後ほど，これでは不十分だと分かります）．sleep と wakeup という一対の関数呼び出しを考えましょう．これらは，次のように動きます．sleep(chan) は，ウェイトチャネル と呼ばれる任意の値 chan に対して眠ります．sleep は呼び出したプロセスを眠った状態にし，他の仕事のために CPU を手放します．wakeup(chan) は，chan に対して眠っている全てのプロセス（もしあれば）を起床させ，それらの sleep 呼び出しからリターンさせます．もしその chan を待つプロセスが居なかったら，wakeup は何もしません．この sleep と wakeup を使うと，セマフォの実装は次のように変えることができます（変更部分を黄色でハイライトしています）．

```
200     void
201     V(struct semaphore *s)
202     {
203         acquire(&s->lock);
204         s->count += 1;
205         wakeup(s);
206         release(&s->lock);
207     }
208
209     void
210     P(struct semaphore *s)
211     {
212       while(s->count == 0)
213         sleep(s);
214       acquire(&s->lock);
215       s->count -= 1;
```

```
216       release(&s->lock);
217     }
```

P は，スピンするかわりに CPU を手放すようになりました．これは良いことです．しかし，このようなインタフェースで sleep と wakeup を実装しようとすると，起床の見逃し問題 (lost wake-up problem) と呼ばれる問題を避けるのは容易ではありません．212 行目において，P が s->count == 0 であることに気づいたとしましょう．P が 212 行目と 213 行目の間にいるあいだ，別の CPU で実行している V が s->count を非ゼロに変更し，wakeup を呼んだとします．この wakeup は眠っているプロセスを見つけることはできず，よって何もしません．続いて P は続いて 213 行目を実行し，sleep を呼んで眠ります．これは問題です．P が，すでに終わってしまった V の呼び出しを待っているからです．運良く生産者が V をもう一度呼ばない限り，消費者は（数が非ゼロであるにもかかわらず）永久に待つことになります（デッドロックしてしまいました）．

この問題の根源は，「p は s->count == 0 のときに限り眠る」という不変量が，まずいタイミングでの V の実行により破れたことにあります．不変量を保護するための間違った方法は，P におけるロックの取得を移動し（黄色でハイライトしています），数のチェックと sleep の呼び出しをアトミックにすることです．

```
300   void
301   V(struct semaphore *s)
302   {
303     acquire(&s->lock);
304     s->count += 1;
305     wakeup(s);
306     release(&s->lock);
307   }
308
309   void
310   P(struct semaphore *s)
311   {
312     acquire(&s->lock);
313     while(s->count == 0)
314       sleep(s);
315     s->count -= 1;
316     release(&s->lock);
317   }
```

ロックにより V が 313 行目と 314 行目の間で実行できなくなるので，P による wakeup

の見逃しががなくなると期待するかもしれません．それはその通りなのですが，これはデッドロックです．P は眠る間ロックを所持し続けているので，V はそのロックを永久に待つことになります．

sleep のインタフェースを変更することで，先の方法の問題を解決できます．呼び出し側が sleep にロックを渡すようにします．そうすることで，呼び出し元プロセスが sleep のチャネルを待ちながら眠ったあとに，sleep は渡されたロックを解放できます．そのロックは，P が眠るまで，並行する V を待たせます．そうすることで，wakeup は眠ってる消費者をみつけ，それを起床させることができるようになります．消費者が再び起床したら，sleep はリターン前にそのロックを再取得します．以上の正しい sleep/wakeup の方法は，次のように使うことができます（変更点を黄色でハイライトしています）．

```
400   void
401   V(struct semaphore *s)
402   {
403     acquire(&s->lock);
404     s->count += 1;
405     wakeup(s);
406     release(&s->lock);
407   }
408
409   void
410   P(struct semaphore *s)
411   {
412     acquire(&s->lock);
413     while(s->count == 0)
414       sleep(s, &s->lock);
415     s->count -= 1;
416     release(&s->lock);
417   }
```

P が s->lock を持つことで，P による c->count のチェックと sleep 呼び出しの合間に，V が起床しようとすることはなくなります．ただし，sleep は，s->lock の解放と，消費者プロセスを眠らせることをアトミックに行わなくてはいけない点に注意が必要です．

6.6 Code: Sleep と wakeup

sleep (kernel/proc.c:533) と wakeup (kernel/proc.c:567). の実装を見てみましょう．基本的なアイディアは次のようなものです．sleep は現在のプロセスを SLEEPING にマークし，そのあとで sched を呼んで CPU を手放します．一方，wakeup は，与えられたウェイトチャネルで待つプロセスを検索し，それらを RUNNABLE にマークします．sleep と wakeup の呼び出し元は，お互いに都合がよいどのような数値もチャネルとして使うことができます．xv6 は，待機に関わるカーネルのデータ構造のアドレスを，そのための値としてよく用います．

sleep は p->lock を取得します(kernel/proc.c:544). いままさに眠りにつこうとしているプロセスは，p->lock と lk の両方を所持します. lk は呼び出し元（例では P）で取得する必要があります．lk は，他のプロセス（例では V）が wakeup(chan) の呼び出しを行わないことを保証します．sleep が p->lock 取得するところまで来たら，lk を解放しても安全です．他のプロセスが wakeup(chan) の呼び出しを始めるかもしれませんが，wakeup は p->lock を取得するために待つからです．wakeup は sleep がプロセスを眠らせるまで待つようになるので，wakeup が sleep を見逃すことがなくなります．

少し混乱しやすいところがあります．lk が p->lock と同じものだと，sleep は p->lock を取得しようとしたときに，自身に対してデッドロックします．しかし，そもそも sleep を呼ぶプロセスがすでに p->lock を取得しているならば，それ以上何もしなくても wakeup の見逃しは起きません．このケースは，wait (kernel/proc.c:567) が p->lock を引数として sleep を呼ぶときに生じます．

ここまで来たら，sleep は p->lock のみを所持します．sleep は対象のプロセスを眠らせるために，sleep のチャネルを記録し，プロセスの状態を変更し，そして sched (kernel/proc.c:549-552) を呼びます．

その後のある時点で，プロセスは wakeup(chan) を呼びます(kernel/proc.c:567). wakeup は，プロセステーブルに対してループします．その際，wakeup は，検査する対象のプロセスの p->lock を取得します．なぜなら，プロセスの状態を操作する可能性があるとともに，p->lock は sleep と wakeup がお互いを見逃さないことを保証してくれるからです．所与の chan で待つ SLEEPING 状態のプロセスを見つけると，wakeup は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したとき，そのプロセスは実行する準備ができています．

xv6 のコードは，条件ロックを持っているときは常に wakeup を呼びます．セマフォの例では s->lock がロックでした．厳密に言えば，acquire の次に必ず wakeup

があれば十分です（つまり，release のあとに wakeup を呼ぶということです）．上記の sleep と wakeup に関するロックのルールが，眠っているプロセスが wakeup の見逃しを防ぐのはなぜでしょうか？眠りにつこうとするプロセスは，条件をチェックしてから状態が SLEEPING にマークされるまでのあいだ，条件ロック，自身の p->lock，もしくは両方を所持します．wakeup を呼ぶプロセスは，wakeup のループにおいて，それらのロックを両方を所持します．よって，起こす役目のプロセスは，消費者スレッドが条件をチェックする前に条件を真にするか，あるいは SLEEPING にマークされたスレッドを wakeup で見つけるかのいずれかしかありえません．よって，wakeup は眠っているプロセスを見つけ，起床をさせることができます（誰かが先に起床をさせていない場合に限ります）．

複数のプロセスが，同じチャネルに対して待つケースがあります．たとえば，2 つ以上のプロセスが 1 つのパイプから読み込みを行う場合です．wakeup を 1 度呼べば，それら全てを起床させます．そのうちの 1 つが最初に実行を行い，sleep 呼び出しに使ったロックを取得し，そして（パイプの場合であれば）パイプに入っている何らかのデータを読みます．他のプロセスは，せっかく起床しましたが，読み込むデータが無いことに気づきます．それらのプロセスの視点で，これは「偽の」起床であり，再び眠りにつきます．sleep を，常に条件をチェックするループの中で呼び出すのはそのためです．

二者が偶然同じチャネルを選んで sleep/wakeup を呼んだとしても害はありません．偽の起床は発生しますが，上記のようにループを繰り返すのでこの問題は許容できます．sleep/wakeup の魅力の大部分は，両方とも軽量であること（sleep のチャネルとして動く特別なデータ構造を作らなくてもよい）と，間接的なアクセスの層を提供すること（呼び出し元は，相手がどのプロセスなのか知らなくてもよい）ことにあります．

6.7 Code: パイプ

sleep と wakeup により生産者・消費者を同期化するより複雑な例は，xv6 のパイプの実装です．パイプのインタフェースは 1 章で見ました．パイプの片方の端にバイト列を書き込むと，それはカーネル内のバッファにコピーされ，そしてパイプのもう片方の端から読み出せる，というものでした．続く章では，ファイルディスクリプタがパイプをどのようにサポートするか説明しますが，ここでは pipewrite と piperead の実装を見てみることにします．

各パイプは，pipe 構造体で表現されます．各構造体には，ロックとデータバッファが含まれます．nread と nwrite フィールドは，バッファに読み書きされたデータ

の総数をカウントします．バッファは循環しており，buf[PIPESIZE-1] の次のバイトは buf[0] です．それに対し，カウントした数値は循環しません．そのように決めることで，バッファが満杯の状態 (nwrite == nread+PIPESIZE) と空の状態 (nwrite ==nread) を区別できます．しかし，これは，バッファをインデックスするには，buf[nread] ではなく，buf[nread % PIPESIZE] を使わなくてはいけないということです（nwrite でも同様です）．

別の CPU において，piperead と pipewrite が同時に起きたとしましょう．pipewrite (kernel/pipe.c:77) は，まずそのパイプのロックを取得します．そのロックは，カウント数，データ，および関連する不変量を保護しています．piperead (kernel/pipe.c:103) もまた，そのロックを取得しようとしますが失敗し，acquire でロックを待ってスピンします(kernel/spinlock.c:22)．piperead が待つあいだ，pipewrite は書き込むデータに対してループを行い (addr[0..n-1])，それぞれをパイプに加えていきます(kernel/pipe.c:95)．このループにおいて，バッファが満杯になることがありえます(kernel/pipe.c:85)．その場合，pipewrite は wakeup を呼ぶことで，眠っているプロセスに対してデータが待っていることを伝え，それから&pi->nwrite に対してスリープすることで，他のプロセスがバッファからバイト列を取り除いてくれることを待ちます．sleep は，pipewrite のプロセスを sleep させる処理の一環で pi->lock を解放します．

pi->lock が使えるようになったので，piperead はそれを取得してクリティカルセクションに入り，pi->nread != pi->nwrite であることに気づきます (kernel/pipe.c:110) （pipewrite は pi->nwrite == pi->nread+PIPESIZE なので眠りについたのでした(kernel/pipe.c:85))．よって，piperead は for ループに進み，パイプからデータをコピーし(kernel/pipe.c:117)，そして読んだバイト数だけ nread を増加させます．バッファには書き込みの余裕がたくさんできたので，piperead は wakeup を呼んで眠っている書き込み用プロセスを起床させ(kernel/pipe.c:124)，そしてリターンします．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，先ほど pipewrite を実行して，バッファが満杯なので眠ったプロセスです．wakeup は，そのプロセスを RUNNABLE にマークします．

パイプのコードは，読み込みを行うプロセスと書き込みを行うプロセスに対し，別の sleep のチャネルを使います (pi->nread と pi->nwrite です)．そうすることで，（あまりありませんが）読み込み・書き込みプロセスが大量にいるときに効率的になります．パイプのコードは，条件をチェックするループの内側で眠ります．一番最初に起床したプロセス以外は，条件が偽のままであることに気づき，再び眠ります．

6.8 Code: wait, exit, および kill

sleep と wakeup はいろいろなものを待つのに使えます．おもしろい例は，1 章で紹介した，子プロセスの exit と親プロセスの wait に関するやりとりです. 子が死んだとき，親は wait ですでに眠っているかもしれませんし，なにか別のことをしているかもしれません．後者の場合，あとから呼んだ wait が子供が死んだことに気づけなくてはいけません．wait が観察するときまで，子供の死を記録しておくために，exit は呼び出し元プロセスを ZOMBIE 状態にします．その子プロセスは，親プロセスの wait が気づくまで，その状態のままでいます．wait は，子プロセスの状態を UNUSED にし，子プロセスの終了ステータスをコピーし，そして親プロセスに対して子プロセスのプロセス ID を返します．もし親プロセスが子プロセスよりも先に終了する場合，親プロセスはその子プロセスを，永久に wait を呼び続けている init プロセスに渡します．よって，どんな子プロセスにも，終了後に片付けをしてくれる親プロセスがいることになります．実装上の大きな課題は，親と子の wait と exit，もしくは exit と exit に関わる競合とデッドロックです.

wait は，呼び出し元プロセスの p->lock をロックとして用いることで，wakeup の見逃しを防ぎますが，ロックの取得は start (kernel/proc.c:383) で行います．そのあと，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，wait は子の資源と proc 構造体を解放し，wait の引数のアドレス (0 でなければ) に子の終了コードをコピーし，そして子のプロセス ID をリターンします．子をみつけたがまだ終了していなかったとき，wait は sleep を呼んでどれかが終了するまで待ち (kernel/proc.c:430)，そして再びスキャンします．このとき，sleep が解放する条件ロックは待っているプロセスの p->lock であり，上で述べた特殊例にあたります．wait は 2 つのロックを取得することがよくありますが，子のロックを取得しようとするまえに，まず自身のロックを取得します．そうすることで，xv6 全体が同じロック順序（まず親，続いて子）になってデッドロックを避けることができます.

wait は, 各プロセスの np->parent を見て子を探します．そのとき, np->lock を取得せずに np->parent を使います．これは，共有されている変数はロックで守らなくてはいけないという通常のルールを破っています．そうしてもよいのは，np が現在のプロセスの先祖だからであり，np->lock を取得すると上で述べたロックの取得順序に違反してデッドロックを起こすからです．ロックを取得せずに np->parent を調べるのは，このケースでは安全なはずです．プロセスの parent フィールドは，親のみが変更するので，np->parent==p が成り立つならば，現在のプロセスが変更しない限り，その値は変わらないはずだからです.

exit (kernel/proc.c:318) は終了ステータスを記録し，資源を解放し，全ての子を init プロセスに渡し，待っている場合は親を起床させ，呼び出し元を ZOMBIE とマークし，そして永久に CPU を手放します．最後のシーケンスは少しトリッキーです．終了しつつあるプロセスは，その状態を ZOMBIE にして親を起床させるあいだ，親のロックを所持しなくてはなりません．なぜなら，親のロックは wakeup の見逃しをガードする条件ロックだからです．また，そうしないと，ZOMBIE 状態をみた親が，まだ実行中の子プロセスを解放してしまうかもしれないからです．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 は子のロックの前に親のロックを取得するので，exit も同じ順序を守らなくてはなりません．

exit は特殊な起床用の関数である wakeup1 を呼びます．これは，親だけを，かつ wait で待っているときに限り起床させるものです (kernel/proc.c:583)．子が，自身の状態を ZOMBIE にする前に親を起床させるのは誤りに見えますが，これは安全です．wakeup1 は親を実行させる可能性がありますが，wait のループは，スケジューラが子の p->lock を解放するまで子を調べることができません．よって，wait は，ずっとあとに exit がその状態を ZOMBIE にセットするまで，終了しつつある子プロセスを見ることが出来ません (kernel/proc.c:371).

exit がプロセス自ら停止するためにあるのに対し，kill (kernel/proc.c:596) は，あるプロセスが別のプロセスを停止させることができます．kill が被害者プロセスを直接壊すのは複雑すぎます．なぜなら，被害者プロセスは別の CPU で実行中かもしれず，カーネルのデータ構造を更新する重要な処理の途中かもしれないからです．よって，kill はほとんど何もしません．kill は被害者の p->killed をセットし，もしそれが眠っているならば起床させます．いずれ，その被害者がカーネルに入るか出るかする時点において，p->killed がセットされていれば usertrap が exit を呼び出します．もし被害者がユーザ空間で実行中であれば，システムコールを呼ぶか，タイマ（もしくは他のデバイスによる）割込により，近いうちにカーネルに入るはずです．

被害者プロセスが眠っている場合，kill は wakeup を呼んで被害者を起床させます．条件が満たされていないのに起床させることは潜在的に危険です．しかし，xv6 の sleep 呼び出しは常に while ループにくるまれていて，起床するとすぐに条件を再検証します．sleep の呼び出しの中には，ループ内で p->killed を検証し，もしセットされていれば現在の活動を放棄するものもあります．それは，そのような放棄が正しいときに限り行われます．たとえば，パイプの読み書きのコード (kernel/pipe.c:86) は killed フラグがセットされているとリターンします．そのコードはいずれトラップにもどり，フラグを再びチェックして exit します．

xv6 の sleep のループには， p->killed をチェックしないものもあります．ア

トミックでなくてはならない複数ステップからなるシステムコールの途中にいる場合です．virtio ドライバ(kernel/virtio_disk.c:242) がその例です．`p->killed` をチェックしないのは，そのディスク操作が，ファイルシステムを正しく保つために順番どおりに完遂しなくてはいけない書き込みの途中かもしれないからです．ディスク I/O を待っているあいだに殺されたプロセスは，現在のシステムコールが完了し，`usertrap` が `killed` フラグを見るまでは終了しません．

6.9 世の中のオペレーティングシステム

xv6 のスケジューラは単純なスケジューリングポリシーを実装します．すなわち，全てのプロセスを順に実行するというものです．これは，ラウンドロビンと呼ばれるポリシーです．世の中のオペレーティングシステムは，より洗練されたポリシーを実装しており，たとえば，プロセス間に優先度を設定することができます．これはすなわち，スカジューラが，実行可能な高優先度のプロセスを，低優先度プロセスよりも優遇するということです．そのようなポリシーはすぐに複雑になります．なぜなら，目標が相容れない場合があるからです．たとえば，そのオペレーティングシステムは，公平性や高スループットもまた保証したいと思うかもしれません．さらに，複雑なポリシーは，優先度の逆転 (priority inversion) や，コンボイ (convoy) などの意図しない相互作用を生じることがあります．優先度の逆転は，高優先度と低優先度のプロセスがロックを共有しているときに起きます．低優先度のプロセスがロックをつかむと，高優先度のプロセスは処理を進めることができなくなってしまいます．たくさんの高優先度プロセスが，低優先度プロセスが取得したロックを待つと，待っているプロセスの長いコンボイができる可能性があります．一度コンボイができると，それは長いあいだ維持されます．そのような問題を避けるには，追加のメカニズムを備えた洗練されたスケジューラが必要です．

`sleep` と `wakeup` は単純で効果的な同期方法ですが，別の方法もありえます．それらに共通する第一の課題は，この章の最初の方で見た起床の見逃し問題です．オリジナルの Unix カーネルの `sleep` は，単に割込を無効化していました．Unix は単一プロセッサで実行していたのでそれで十分だったのです．xv6 はマルチプロセッサで動作するので，`sleep` に明示的なロックが必要です．FreeBSD の `msleep` は同じアプローチをとります．Plan 9 の `sleep` は，休眠する直前に，スケジューラのロックを持った状態で実行するコールバック関数を使うことができます．この関数は，`sleep` する最後の瞬間に，スリープ条件をチェックする役目があり，それにより `wakeup` の見逃しを防ぎます．Linux カーネルの `sleep` は，ウェイトチャネルのかわりに，ウェイトキューと呼ばれる明示的なプロセスキューを使います．そのキューには，内部的

なロックがあります.

wakeup がプロセスリスト全体を調べて chan が一致するか確かめるのは非効率的です．より良い方法は sleep と wakeup で使う chan を構造体に置き換え，そこに待っているプロセスのリストを入れておくことです．Linux のウェイトキューがその例です．Plan 9 の sleep と wakeup はその構造体のことをランデブーポイントもしくは Rendez と呼びます．多くのスレッドは，その構造体のことを条件変数と呼びます．この文脈において，sleep と wakeup は，それぞれ wait と signal と呼ばれます．それらのメカニズムはいずれも似た雰囲気を持ちます．スリープする条件はある種のロックによって守られており，スリープ時にそのロックは自動で解放されます.

wakeup の実装は，特定のチャネルで待つプロセスを全て起床させていました．また，たくさんのプロセスが特定のチャネルを待つことがありました．オペレーティングシステムがそれらのプロセスのスケジュールを行い，それらは競ってスリープ条件をチェックします．プロセスのこのような振る舞いは，*thundering herd* と呼ばれることがあり，避けた方がよいものです．多くの条件変数は wakeup のために 2 つのプリミティブを持ちます．1 つのプロセスだけを起床させる signal と，待っているプロセス全てを起床させる broadcast です.

同期化のためにセマフォが使われることもあります．セマフォのカウントの値は，通常，パイプバッファ内のバイト数や，そのプロセスが持つゾンビの子プロセス数などとして利用します．抽象化において明示的なカウントを導入すると，起床の見逃し問題を防ぐことができます．発生した wakeup の数を明示的にカウントするのです．このカウントは，偽の起床や thundering herd 問題を避けるのにも役立ちます.

プロセスを終了してそれを掃除することは xv6 をだいぶ複雑にしています．多くのオペレーティングシステムにおいて，それはさらに複雑です．なぜなら，たとえば，被害者プロセスが眠っているカーネルの奥深くにいるかもしれず，そのカーネルスタックの巻き戻すには慎重なプログラミングが必要です．多くのオペレーティングシステムでは，スタックの巻き戻しを，割込ハンドリングにおける明示的なメカニズムとして用いています．たとえば longjmp です．また，待っているイベントがまだ生じていないのにも関わらず，眠っているプロセス起床させうるイベントがあります．たとえば，ある Unix プロセスが眠っているとき，別プロセスが signal を送ってくる可能性があります．その場合，そのプロセスは割り込まれたシステムコールから返値 -1 でリターンし，エラーコードを EINTR にセットします．アプリケーションは，これらの値をチェックしてどうするか決めることができます．xv6 はシグナルをサポートしないので，その複雑さが問題になることはありません.

xv6 による kill のサポートは不完全です．スリープのループで，p->killed をチェックしなくてはいけないかもしれないものがあります．関連する問題は，

p->killed をチェックするスリープのループであっても，sleep と kill のあいだで競合があることです．被害者のループが p->killed をチェックしたが sleep を呼ぶ前のタイミングで，kill が p->killed をセットして被害者を起床させる可能性があります．この問題が起きると，待っている条件が発生するまで，被害者は p->killed に気づきません．これは，かなりあとになったり（例: 被害者の待っていたディスクブロックを，virtio ドライバが返したとき），場合によっては永久に来ません（被害者がコンソール入力を待っているが，ユーザが何も入力しない場合）．

現実世界のオペレーティングシステムでは，利用可能な proc 構造体を取得するとき，allocproc を線形に探索するかわりに，自由リストを用いて定数時間で見つけるという方法を使うことがあります．xv6 は，単純さのために線形探索を用いています．

6.10 練習問題

1. sleep は，デッドロックを避けるために lk != &p->lock のチェックをする必要があります(kernel/proc.c:543-546)．このコード

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

   を，次のコードに置き換えることで特殊ケースを消してしまったとします:

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

   こうすると sleep は壊れてしまうのですが，どう壊れるでしょうか？
2. ほとんどのプロセスのクリーンアップは，exit もしくは wait のいずれかで行われます．結果として，開いているファイルは exit が閉じなくてはならないのですが，それはなぜでしょうか？回答には，パイプが関連しています．
3. xv6 にセマフォを実装してください．ただし，sleep と wakeup は使わないでください（スピンロックは使っても構いません）．続いて，xv6 の sleep と wakeup をセマフォに置き換えてください．この結果の良し悪しを判断してください．
4. 上記の kill と sleep の競合を解決してください．すなわち，被害者のスリープループが p->killed をチェックしたあとでかつ sleep を呼ぶ前に kill が呼ばれたら，被害者が現在のシステムコールを廃棄するようにしてください．

5. スリープループが毎回 p->killed をチェックするような計画を設計してください．そうすることで，たとえば，virtio ドライバにいるプロセスが他のプロセスから kill されたら，即座に例の while ループからリターンできるようにしてください．
6. あるプロセスのカーネルスレッドから別へ切り替わるときに，スケジュールスレッドを経由するのではなく，コンテキストスイッチ 1 回で済むように xv6 を改造してください．CPU を手放そうとしているスレッドが自ら次のスレッドを選び，swtch を呼ぶことになるでしょう．複数のコアが間違って同じスレッドを実行するのを防ぐところが難しいはずです．すなわち，正しくロックをしてデッドロックを回避するところが難しいはずです．
7. xv6 の scheduler を改造して，実行可能なプロセスが無いときは RISC-V の WFI (wait for interrupt) 命令を使うようにしてください．実行可能プロセスが待っているときは，WFI で停止するコアが無いことを保証してみてください．
8. p->lock は多くの不変量を保護しているため，xv6 の p->lock で保護されたある特定のコードを見ただけでは，どの不変量が保たれているのか判断が難しいことがあります．p->lock を複数のロックに分割することで，よりきれいな計画を設計してください．

第 7 章

ファイルシステム

ファイルシステムはデータを整理して保存するためにあります．ファイルシステムは通常，ユーザやアプリケーション間のデータの共有や，再起動後にもデータを使うための永続性を提供します．

xv6 のファイルシステムは，Unix 風のファイル，ディレクトリ，パス名（1 章を見てください），`virtio` ディスクへの永続的な保存（4 章をみてください）などを提供します．ファイルシステムは，いくつかの課題を解決する必要があります．

- ファイルシステムは，ディスク上に名前付きディレクトリとファイルのツリーを表現するためのデータ構造を必要とします．それを用いて，どのブロックがどのファイルの中身を保持するかや，ディスクのどの領域が空いているかを記録します．
- ファイルシステムはクラッシュリカバリをサポートしなくてはいけません．すなわち，クラッシュ（たとえば電源不良）が起きたとしても，ファイルシステムは再起動後に正しく動かなくてはいけません．危険なのは，クラッシュが更新の列に割り込むことで，ディスク上のデータ構造から一貫性が失われることです（例：あるブロックがあるファイルで使われているのに，未使用のマークがついてしまう）．
- 複数のプロセスが，同時にファイルシステムの操作をすることがありえます．そのため，ファイルシステムのコードが調整をして，不変量を管理しなくてはいけません．
- ディスクへのアクセスは，メモリへのアクセスと比べてケタ違いに低速です．そのため，ファイルシステムはよく使われるブロックのキャッシュをメモリ上に作り，管理する必要があります．

この章の残りは，xv6 がどのように上記の課題を解決するかを説明します．

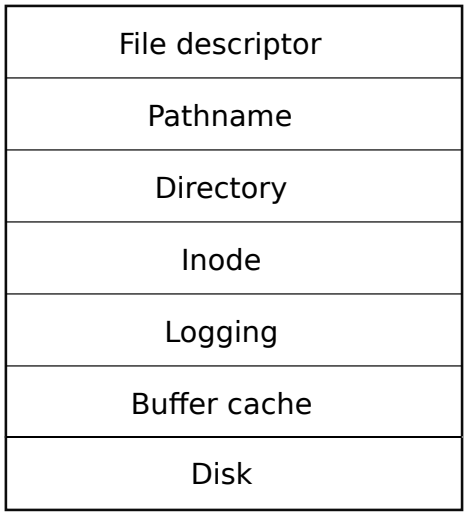

図 7.1　xv6 ファイルシステムの層

7.1　概要

xv6 のファイルシステムの実装は，図 7.1 に示す 7 層に整理できます．ディスク層は，`virtio` ハードドライブ上のブロックの読み書きを行います．バッファキャッシュ層は，ディスク上のブロックをキャッシュするとともに，そのアクセスを同期化して，ある特定のブロックに入ったデータを変更できるのが 1 度に 1 プロセスだけになるようにします．ロギング層は，より上位の層のために，複数ブロックにまたがる更新をトランザクションにまとめます．そうすることで，クラッシュしたとき，それらの更新がアトミックになる（すなわち，全て更新するかまったくしないかのどちらかになる）ようにします．inode 層はファイルを提供します．ファイルは，ユニークな i-number を持つ *inode* と，そのファイルの情報を持ついくつかのブロックで表現されます．ディレクトリ層は，ディレクトリを提供します．ディレクトリとは特別な種類の inode のことで，その中身はディレクトリエントリの列で，それぞれがそのファイルの名前と `i` 番号を含んでいます．パス名層は，`/usr/rtm/xv6/fs.c` のような階層化されたパス名を提供し，パス名を解決するための再帰的な参照を行います．ファイルディスクリプタ層は，たくさんの Unix 資源を，ファイルシステムのインタフェースで抽象化します（例：パイプ，デバイス，ファイルなど）．そうすることで，アプリケーションプログラマの仕事を単純化します．

ファイルシステムには，inode やファイルの中身を記録するブロックを，ディスク上のどこに記録するかという計画が必要です．そのために，xv6 はディスクを，図 7.2 に示すように複数のセクションに分割します．ファイルシステムは 0 番ブロックは使

いません（そこには，ブートセクタが入っています）．1 番ブロックはスーパーブロックと呼ばれます．そこには，ファイルシステムに関するメタデータが入っています（ブロック数でのファイルサイズ，データブロックの個数，inode の個数，ログに入っているブロックの数）．2 番以降のブロックにはログが入ります．ログの後ろにあるのは inode で，1 つのブロックに複数の inode が入っています．そのあとに来るのはビットマップブロックで，どのブロックが使用中かを追跡するのに使います．残りのブロックはデータブロックで，ビットマップブロックにおいて未使用とマークされているか，あるいはファイルやディレクトリの中身を保持しています．スーパーブロックは，`mkfs` と呼ばれる独立したプログラムによって書き込まれます．それが，ファイルシステムを初期化します．

この章の残りは，バッファキャッシュからはじめて，各層の説明をしていきます．低層でよく考えられた抽象化を選ぶことで，上層の設計が楽になるとうシチュエーションを見ることができます．

7.2 バッファキャッシュ層

バッファキャッシュには 2 つのしごとがあります．(1) ディスクブロックへのアクセスを同期化することで，唯一のコピーのみがメモリに入り，1 度に 1 つのカーネルスレッドだけがそのコピーを使うようにすること．(2) よく使われるブロックをキャッシュすることで，遅いディスクから何度も読まなくてよくすること．コードは `bio.c` です．

バッファキャッシュが提供する主なインタフェースは `bread` と `bwrite` です．前者はあるブロックのコピーをメモリ上で読み書きできる `buf` を取得します．後者は，更新されたバッファをディスク上の適切なブロックに書き戻します．カーネルスレッドは，使い終わったバッファを `brelse` で解放しなくてはいけません．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うことで，1 度に 1 つのスレッドだけが各バッファ（すなわち各ディスクブロック）を使えるようにします．`bread` はロックされたバッファをリターンし，`brelse` はロックを解放します．

バッファキャッシュの話にもどりましょう．バッファキャッシュは，ディスク上のブロックをたくわえるバッファを，ある決まった数だけ持っています．それはつまり，ファイルシステムがまだキャッシュに入っていないブロックを要求したら，今は別のブロックのために使われているバッファを再利用しなくてはいけないことを意味します．バッファキャッシュは，新しいブロックのために，最近使用頻度が低かった (least recently used) バッファを再利用します．最近使用頻度が低かったバッファは，将来使われる可能性がもっとも低いだろうと考えるわけです．

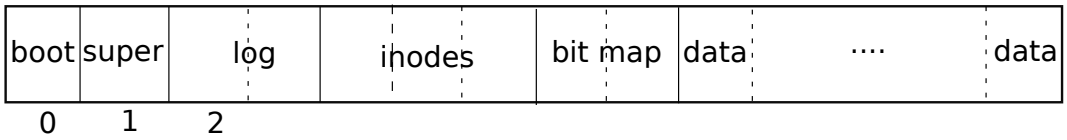

図 7.2　xv6 ファイルシステムの構造

7.3　Code: バッファキャッシュ

バッファキャッシュはバッファの 2 重リンクリストです．main 関数が呼ぶ binit 関数が(kernel/main.c:27)，静的配列 buf を使って，要素数 NBUF の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それ以外のアクセスは，buf 配列ではなく，bcache.head を介してリンクリストとして行います．

バッファには 2 つの状態フィールドが関連付けられています．valid フィールドは，そのバッファがあるブロックのコピーを保持していることを表します．disk フィールドは，バッファの中身がディスクに受け渡されたことを表します．これは，バッファを変える可能性があります（例：ディスクから読んだデータを data に書き込む）．

bread (kernel/bio.c:91) は bget を呼んで，所望のセクタに対応するバッファを取得します(kernel/bio.c:95)．もし，バッファをディスクから読まなくてはいけない場合には，bread はバッファを返す前に virtio_disk_rw により読み込みを行います．

bget (kernel/bio.c:58) はバッファリストをスキャンし，与えられたデバイスとセクタ番号を探します(kernel/bio.c:64-72)．もし見つかったら，bget はそのバッファのスリープロックを取得します．bget は，そのあとでロック済みバッファを返します．

与えられたセクタに対応するバッファが無かったら作る必要があります．そのためには，別のセクタのために使われているバッファの再利用が必要かもしれません．bget はバッファリストを再度スキャンし，使われていないバッファ (b->refcnt = 0) を探します．そのようにして使えるバッファが見つかったら，bget は新しいデバイスとセクタ番号を記録するためにバッファのメタデータを変更し，そしてスリープロックを取得します．b->valid = 0 と代入することで，bread はバッファの古い内容を使わずに，ディスクからそのブロックのデータを読み出します．

各ディスクのセクタごとに，キャッシュされたバッファは最大でも 1 個であることが重要です．そうすることで，バッファを読む人が書き込みに気づくことができます．また，ファイルシステムがバッファへのロックにより同期化を行っていることも理由の 1 つです．bget は，この不変量を保証するため，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調べるところから，2 つ目のループで（dev,

backno, および refcnt をセットして）ブロックがキャッシュされたと宣言されるまで，bache.lock を保持し続けます．そうすることで，ブロックが存在するか確認して，もし存在しないならばそのブロックが入るバッファを取得するという一連の処理がアトミックになります．

bget は，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．なぜなら，b->refcnt が非ゼロでであることにより，バッファが別のディスクブロックのために再利用されることはないからです．スリープロックは，ブロックのバッファされた内容の読み書きを保護します．それに対し，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．

もし全てのバッファが使用中だったら，同時にファイルのシステムコールを実行しているプロセスが多すぎるということです．bget はパニックを起こします．より穏やかな対応として，バッファが使用可能になるまでスリープする方法もありえますが，デッドロックの可能性があります．

bread が（必要に応じて）ディスクからの読み込みを完了し，バッファを呼び出し元にリターンしたら，その呼び出し元はバッファを排他的に利用して読み書きができます．もし呼び出し元がバッファを書き換えたら，バッファを解放する前に，bwrite を呼んでデータをディスクに書きもどす必要があります．bwrite (kernel/bio.c:105) は，virtio_disk_rw を呼んでディスクのハードウェアとやり取りをします．

バッファを使い終わったら brelse を呼んで解放しなくてはなりません．(brelse は “b-release” を短縮したものであり，暗号じみていますが学ぶ価値があります．Unix に起源があり，BSD, Linux, および Solaris でも使われています)．brelse (kernel/bio.c:115) はスリープロックを解放し，バッファをリンクリストの先頭に付け加えます(kernel/bio.c:126-131)．バッファを付け加えると，リストは，使われた（すなわち解放された）タイミングが新しい順にバッファを整列します．リストの先頭バッファはもっとも最近使われたもので，最後尾はもっとも使われていないものです．bget の 2 つのループがその性質を利用します．既存のバッファを探そうとすると，最悪の場合ではリスト全体を見る必要があります．しかし，最近使われたバッファから順に調べれば（すなわち bcache.head からはじめて順にリストをたどれば)，参照によい局所性があれば，スキャン時間を減らすことができます．一方，再利用するバッファを探すときは，リストを逆向きにたどれば，最近もっとも使われていないバッファを見つけることができます．

7.4 ロギング層

ファイルシステム設計でもっともおもしろい問題の 1 つはクラッシュリカバリです．ファイルシステムのオペレーションの多くは，ディスクへの複数回のアクセスが必要となるので，途中でクラッシュすると，ディスク上のファイルシステムが一貫性のない状態になってしまうことがあります．たとえば，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，中身を保存していたブロックを解放すること）の途中でクラッシュが起きた場合を考えてみましょう．ディスクへの書き込み順序によりますが，クラッシュにより，解放済みとマークされたブロックを使い続ける inode ができてしまうか，もしくはアロケートされているがどこからも参照されていないコンテンツ用ブロックができてしまいます．

後者は比較的マシですが，解放済みブロックを参照する 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のためにアロケートする可能性があります．そうすると，2 つの異なるファイルが，意図せず同じブロックを指すことになります．もし xv6 が複数ユーザをサポートしていたとしたら，この状況はセキュリティ上の問題になります．なぜなら，古いファイルの所有者は，別のユーザが持つ新しいファイルのブロックに対して読み書きができてしまうからです．

xv6 はファイルシステムのオペレーション途中でのクラッシュの問題を，シンプルなロギングで解決します．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読み書きすることはありません．そのかわり，ディスク上のログに記入するのです．システムコールは，全ての書き込み処理をログに書き終えたら，特別なコミット記録をディスクに書き込むことで，ログが完全なオペレーションを含むことマークをつけます．それから，書き込み内容をディスク上のファイルシステムのデータ構造へコピーします．書き込みが終わったら，システムコールはディスク上のログを消去します．

もしシステムがクラッシュして再起動した場合，ファイルシステムは次のようにしてクラッシュから復帰します．もしログに完全なオペレーションが入っているとマークされていたら，復帰のためのコードが動き，ディスク上のファイルシステムの適切な場所へデータをコピーします．一方，オペレーションが完全であるとマークされていないログは無視します．復帰のためのコードは，最後にログを消去します．

なぜ，xv6 のログを使うと，ファイルシステムのオペレーション途中のクラッシュ問題が解決するのでしょうか？もしオペレーションのコミット前にクラッシュが起きたら，ディスク上のログは完全とマークされないので，復帰用のコードはそれを無視

します．その結果，ディスクは，そのオペレーションが始まる前の状態にもどります．コミットのあとにクラッシュが起きたら，復帰用のコードが書き込みを再度行います．その際，クラッシュ前に行った書き込みを繰り返すこともありえます．いずれの場合でも，ログはオペレーションを（クラッシュに関して）アトミックにします．復帰のあと，オペレーションによる書き込みは，全体がディスク上にあるか，もしくはまったく無いかのいずれかになります．

7.5 ログの設計

ログは，スーパーブロックで指定する定位置に置かれます．ログはヘッダブロックと，更新されたブロックのコピー (ログブロック) の列からなります．ヘッダブロックには，各ログブロックのセクタ番号の配列と，ログブロックのカウント数が入っています．ディスク上のヘッダブロックのカウント数がゼロであるとき，ログにはトランザクションが記録されていません．一方，カウント数が非ゼロであるとき，ログはコミットされた完全なトランザクションを保存しており，それはカウント数分のブロックからなります．xv6 は，トランザクションをコミットするときにヘッダブロックへ書き込みを行い，ログブロックをファイルシステムへコピーしたあとにカウント数をゼロに戻します．トランザクションの途中でクラッシュが起きると，ヘッダブロックのカウント数をゼロのままになります．一方，コミット後にクラッシュが起きると，カウント数は非ゼロとなります．

各システムコールのコードが，クラッシュに関してアトミックにすべき書き込み列の範囲を決めます．複数のプロセスによるオペレーションを並行処理するため，ロギングシステムは複数のシステムコールによる書き込みを 1 つのトランザクションに固めることがあります．よって，1 つのコミットは，複数のシステムコールの書き込みを含むことがあります．システムコールがトランザクションをまたがないように，他のファイルシステムのシステムコールが動いていないときに限りコミットを行います．

複数のトランザクションをまとめてコミットするアイディアは，グループコミット (group commit) と呼ばれます．グループコミットは，ディスクオペレーションの数を減らす効果があります．コミットのための固定コストが，複数のオペレーションで割り勘になるからです．グループコミットにより，ディスクシステムはよりたくさんの書き込みを並行して行えるようになります．たとえば，ディスクが一回転するあいだに全てを書き込むような場合です．xv6 の `virtio` はそのようなバッチ処理は行いませんが，xv6 のファイルシステムの設計では原理的にそのようなことが可能です．

xv6 は，固定長のディスク領域を，ログを保存するために使います．あるトランザクションが書き込むのブロック数は，その領域に収まる必要があります．そのために

次の 2 つの制限が導かれます．1 つ目の制限は，どのシステムコールも，ログよりも大きいサイズの書き込みはできないというものです．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，write と unlink の 2 つは，潜在的にたくさんのブロックを書き込む可能性があります．大きなファイルの書き込みは，データブロックだけでなく，ビットマップブロックや inode への大量の書き込みを生じるからです．同様に，大きなファイルのアンリンクは，ビットマップブロックや inode への大量の書き込みを生じます．xv6 の write システムコールは，大きい書き込みを，ログに収まる小さい書き込みに分解します．xv6 では unlink は問題になりません．なぜなら，xv6 はビットマップブロックを 1 つしか使わないからです．ログのためのスペースが限られることの 2 つ目の制限は，ある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場合でなければ，そのシステムコールを始めることができないというものです．

7.6 Code: ロギング

システムコールにおいてログを使用する典型例は以下のようなものです．

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

begin_op (kernel/log.c:126) は，ロギングシステムがコミットしておらず，かつそのシステムコールによる書き込みが入るログ領域が空くまで待ちます．log.outstanding は，ログ領域を予約したシステムコール数をカウントします．使用されているスペースは log.outstanding×MAXOPBLOCKS です．log.outstanding を増加させることで，スペースを予約するとともに，そのシステムコールの間にコミットが発生しないようにします．コードは，各システムコールのコードが MAXOPBLOCKS ブロックを使用するかもしれないと保守的に想定します．

log_write (kernel/log.c:214) は，bwrite のプロキシとして動作します．それは，ブロックのセクタ番号をメモリに記録し，ディスク上のログにスロットを予約し，ブロックキャッシュ内のバッファをピン留めしてキャッシュが退去 (evict) されないようにします．ブロックは，コミットが済むまでキャッシュに留まる必要があります．キャッシュされたコピーが，行われた変更の唯一の記録だからです．コミットするま

で，それらをディスク上には書き込めません．一方，同じトランザクションの他の読み込みは，その変更を見ることができなくてはいけません．log_write は，トランザクション内の特定のブロックに何度も書き込みが行われたらそのことに気づき，ログの同じスロットにあるブロックをアロケートします．この最適化は，よく吸収 (absorption) と呼ばれます．たとえば，さまざまなファイルの inode を含むディスクブロックは，1 つのトランザクションの中で何度も書き込まれることが一般的です．複数のディスク書き込みを 1 つに吸収することで，ファイルシステムはログのスペースを節約するとともに，1 つのコピーだけがディスクに書き込まれるようにすることで性能向上を達成できます．

end_op (kernel/log.c:146) はまず，未解決な (outstanding な) システムコール数を減らします．その数がゼロになったら，commit() を呼んでそのトランザクションをコミットします．このプロセスには 4 つのステージがあります．write_log() (kernel/log.c:178) はトランザクションで変更される各ブロックを，バッファキャッシュからディスク上のログのスロットへコピーします．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．これがコミットする瞬間です．この書き込みが終わったあとにクラッシュが起きると，ログの中身を再度実行する復元が起きます．install_trans (kernel/log.c:69) はログの各ブロックを読み，ファイルシステムの適切な場所に書き込みます．最後に，end_op がカウント数がゼロのログヘッダを書き込みます．この処理は，次のトランザクションがログブロックに書き込みを行う前に行う必要があります．そうしないと，今回のログヘッダと，次のトランザクションのログブロックという誤った組み合わせでクラッシュ後の復元が行われてしまうからです．

recover_from_log (kernel/log.c:116) は，initlog (kernel/log.c:55), から呼び出されます．initlog は, 起動時に最初のプロセスが実行する前に (kernel/proc.c:524), fsinit (kernel/fs.c:43) が呼び出します．recover_from_log はログヘッダを読み，もしそれがコミット済みのトランザクションを含むのであれば end_op のマネをします．

ログを使用する 1 つの例は，filewrite (kernel/file.c:135). にあります．そのトランザクションは次のようなものです．

```
begin_op();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

このコードは，書き込みを，数セクタていどの小さなトランザクションに分解する

ループにつつまれています．そうすることでログを圧迫しないようにしています．writei を呼ぶと，そのトランザクションの一部として，たくさんのブロックを書き込みます．ファイルの inode，1 つもしくは複数のビットマップブロック，そしていくつかのデータブロックです．

7.7 Code: ブロックアロケータ

ファイルとディレクトリの中身はディスクブロックに保存されますが，そのためのディスクブロックは，使用可能なプールからアロケートする必要があります．xv6 のブロックアロケータは，使用可能なディスクを，1 ビットが 1 ブロックに対応するビットマップとして管理します．ビットが 0 であることは対応するブロックが使用可能であること，ビットが 1 であることは使用中であることをそれぞれ意味します．mkfs というプログラムが，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，およびビットマップブロックに対応するビットをセットします．

ブロックアロケータは 2 つの関数を提供します．新たなディスクブロックをアロケートする balloc と，ブロックを解放する bfree です．(kernel/fs.c:72) にある balloc のループは，0 番から（そのファイルシステムに入っているブロックの総数である）sb.size 番のブロックに向けて順番に調べていきます．balloc は対応するビットマップ上のビットがゼロ，すなわち未使用のブロックを探します．そのようなブロックを見つけると，balloc はビットマップを更新し，そのブロックを返します．効率のために，ループは 2 重に分かれています．外側のループはビットマップのビットを含むブロックを順に読みます．続く内側のループは，あるビットマップブロックの，全ての BPB ビットをチェックします．あるプロセスが，1 度に 1 つのビットマップブロックしか使えないようにバッファキャッシュが制御することで，2 つのプロセスが同時にブロックをアロケートしようとして生じる競合を防ぎます．

bfree (kernel/fs.c:91) は所望のビットマップブロックを見つけ，その中の所望のビットをクリアします．くり返しになりますが，bread と brelse による排他的な取り扱いにより，明示的なロックは不要です．

本章の残りで記述するコードと同様に，balloc と bfree はトランザクションの内部で呼ばれる必要があります．

7.8 inode 層

inode という用語は，2 つの関連した意味を持ちます．一方で，ファイルサイズや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ことがあります．

またもう一方で，メモリ上の inode を指すこともあります．これは，ディスク上の inode をコピーし，カーネル向けの追加情報を付け加えたものです．

ディスク上の inode は，inode ブロックと呼ばれるディスク上の連続した領域に格納されています．全ての inode は同じサイズであるため，ある数字 n が与えられたとき，容易に n 番目の inode をディスク上で見つけることができます．実際，この番号 n は inode 番号，もしくは i-number と呼ばれ，inode を識別するのに用います．

ディスク上の inode は，`dinode` 構造体(kernel/fs.h:32) が定義します．`type` フィールドにより，ファイル，ディレクトリ，および特殊ファイル（デバイス）を区別します．`type` が 0 であることは，そのディスク上の inode が使用可能であることを示します．`nlink` フィールドは，その inode を参照するディレクトリの個数をカウントしており，ディスク上の inode とデータブロックをいつ解放すべきかを知るために使います．`size` フィールドは，そのファイルの中身のバイト数を記録します．`addr` 配列は，そのファイルの中身を保存するブロックのブロック番号を記録します．

カーネルは，よく使う inode をメモリにいれておきます．`inode` 構造体(kernel/file.h:17) が，`dinode` のメモリ上のコピーです．カーネルがメモリ上に inode を置いておくのは，それを指す C のポインタが存在するときにかぎります．`ref` フィールドは，メモリ上 inode を参照する C ポインタの数をカウントするためにあります．もし参照カウントがゼロになったら，カーネルはその inode をメモリから破棄します．`iget` と `iput` 関数は，inode へのポインタを取得し，参照カウントを変更します．inode を指すポインタを使うのは，ファイルディスクリプタ，カレントディレクトリ，および `exec` などの一時的なカーネルのコードなどです．

xv6 の inode のコードには，4 つのロックもしくはロックに類似したメカニズムがあります．`icache.lock` は，「inode はキャッシュ内に最大でも 1 個しか存在しない」および「キャッシュされた inode の `ref` フィールドは，キャッシュされた inode を指すメモリ内ポインタの数を示す」という不変量を保護します．各メモリ内 inode は，スリープロックのためのフィールドを持ち，inode の各フィールド（たとえばファイルの長さ）や，その inode 内のファイル・ディレクトリのコンテンツブロックが排他的にアクセスされることを保証します．inode の `ref` がゼロより大きければ，システムはその inode のキャッシュを維持します．すなわち，そのキャッシュのエントリを別の inode のために再利用しません．最後に，各 inode は `nlink` フィールド（ディスク上，およびキャッシュされているならメモリ上のコピー）を持ち，その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します．xv6 は，`nlink` がゼロよりも大きい限りは，その inode を解放しません．

`iget()` で取得した `inode` 構造体を指すポインタは，対応する `iput` を呼ぶまでは有効です．すなわち，その inode は削除されず，そのポインタが参照するメモリが

別の inode のため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します．ファイルシステムの多くの部分が，この iget() の動作に頼ることで，inode に対して（開いたファイルとカレントディレクトリとして）長期的な参照を持ったり，複数の inode を操作するとき（たとえばパス名の解決）にデッドロックを避けながら競合を防いだりします．

iget が返す inode 構造体は，意味ある中身を持っていないかもしれません．それが，ディスク上の inode のコピーであることを保証するには ilock を呼ばなくてなりません．すると，inode をロックし（他のプロセスが ilock できないようにしてから），もしまだ読まれていなければディスクから inode を読み出します．iunlock は inode のロックを解放します．inode へのポインタの取得とロックを分離することは，ある状況においてデッドロックを避けるのに役立ちます．たとえば，ディレクトリを検索しているときです．複数のプロセスが iget が返した inode へのポインタを持つ可能性がありますが，inode をロックできるプロセスは 1 度に 1 つだけです．

カーネルのコードやデータ構造の C ポインタが指す inode のみがキャッシュされます．その主な仕事は，実は複数プロセスからのアクセスを同期化することにあり，キャッシュすること自体は副次的です．もしある inode が頻繁に使われるならば，たとえ inode キャッシュに入っていなかったとしても，バッファキャッシュがメモリ上に置いておくはずだからです．inode キャッシュはライトスルー (write through) です．すなわち，キャッシュされた inode を書き換えたコードは，iupdate を用いて即座にディスクへ書き込まなくてはなりません．

7.9 Code: inode

新しい inode をアロケートするとき（たとえばファイルを作るとき），xv6 は ialloc (kernel/fs.c:197) を呼びます．ialloc は balloc とよく似ており，ディスク上の inode 構造体を 1 ブロックずつスキャンし，未使用とマークされたものを探します．見つけたら，ialloc は新しい type をディスクに書き込むことでそれを確保し，最後に iget を呼んで得た inode キャッシュのエントリをリターンします (kernel/fs.c:211)．ialloc が正しく動作するには，1 度に 1 つのプロセスだけが bp への参照を持つという事実が必要です．ialloc は，他のプロセスが同じ inode を同時に確保しようとすることは無いと確信しています．

iget (kernel/fs.c:244) は inode キャッシュをスキャンし，所与のデバイスと inode 番号を持つ使用中エントリ ((ip->ref > 0)) を探します．見つけたら，その inode への新しい参照をリターンします (kernel/fs.c:253-257)．iget はスキャンするあいだ，最初の空きスロットの場所を記録しておき，新しいキャッシュのエントリをアロケー

トしなくてはならなくなったときに使います (kernel/fs.c:258-259).

コードは，inode のメタデータや中身を読み書きする前に，ilock を用いてその inode をロックします．ilock (kernel/fs.c:290) はそのためにスリープロックを使います．ilock がその inode への排他的アクセスを取得したら，必要に応じて inode をディスクから（多くの場合はバッファキャッシュから）読みます．iunlock 関数 (kernel/fs.c:318) はスリープロックを解放し，必要に応じて眠っていたプロセスを起床させます.

iput (kernel/fs.c:334) は，参照カウントを減らすことで，ある inode への C ポインタを解放します (kernel/fs.c:357)．もしそれが最後の参照であれば，inode キャッシュにおけるその inode のスロットは未使用となり，別の inode のために再利用できるようになります.

もしある inode を指す C ポインタが無く，またリンクも無い（どのディレクトリにも入っていなかったら）ことに iput が気づいたら，inode とそのデータブロックを解放しなくてはなりません．iput は itrunc を呼び出してファイルを 0 バイトにトランケートすることで，データブロックを解放します．また，inode の type を 0 (未使用) にセットし，それをディスクに書き込みます (kernel/fs.c:339).

iput が inode を解放するときのロックのプロトロルは詳しく見る価値があります．危険かもしれないのは，その inode を使うために（たとえばファイルやディレクトリの一覧を見るために）ilock で待っている別のスレッドがいるかもしれないことです．そのスレッドは，inode が解放されてしまうことを想定していません．しかしそのようなことは決して起きません．リンクが無く，かつ ip->ref が 1 であるならば，システムコールは，そのキャッシュされた inode へのポインタを得ることはできないはずだからです．なお，その 1 つの参照は，iput を呼んだスレッドによるものです．iput は，icache.lock クリティカルセクションの外で参照カウントが 1 であることを確認しますが，その時点においてリンク数は 0 であることがわかっているので，他のスレッドが新しいリファレンスを取得しようとすることはありません．他にありうるかもしれない危険は，並行する ialloc の呼び出しが，iput が解放しようとしているのと同じ inode を選んでしまうことです．それが生じるとしたら，inode の type が 0 になるように iupdate がディスクに書き込んだあとです．この競合は良性です．アロケートしようとしているスレッドは，inode に読み書きを行うまえに，inode のスリープロックをおとなしく待ちます．それまでには，iput は仕事を終えています.

iput() はディスクへの書き込みを行うことがあります．すなわち，ファイルシステムを使うシステムコールはどんなものでもディスクへの書き込みを行うことがあるということです．なぜなら，そのシステムコールが持つ参照が，そのファイルへの最

後の参照かもしれないからです．read() は読み込みしかしないように見えますが，最終的に iput() を呼ぶことがありえます．よって，ファイルシステムを使うならば，たとえ読み込みしか行わないシステムコールであっても，トランザクションでくるむ必要があります．

iput() とクラッシュには難しい関係があります．iput() は，そのファイルへのリンク数がゼロになったとき，すぐにトランケートするわけではありません．他のプロセスが，そのメモリ上 inode への参照を持っているかもしれないからです．まだそのファイルへの読み書きをしているプロセスもあるかもしれません．一方，全てのプロセスがそのファイルへのファイルディスクリプタを閉じる前にクラッシュが起きたら，そのファイルはディスク上で使用中にもかかわらず，それを指すディレクトリエントリが全くないという状況になってしまいます．

ファイルシステムがその問題を処理する方法は 2 通りあります．簡単な解決法は，再起動後の復帰時に，ファイルシステム全体をスキャンして，そのような（使用中だがそれを指すディレクトリエントリが無い）ファイルを探すというものです．もしそのようなファイルを見つけたら解放します．

2 つ目の解決法は，ファイルシステムをスキャンしなくても良い方法です．その解決法は，リンク数がゼロになったが，参照カウントがゼロになっていない inode の i-number を，ディスク上に（たとえばスーパーブロックに）記録しておくというものです．もし，参照カウントがゼロになってそのファイルを削除したら，ファイルシステムは inode をそのディスク上のリストから削除します．復帰時には，そのリストに載っているファイルを解放します．

xv6 は，どちらの解放も実装しません．すなわち，ディスク上で使用中になっているにもかかわらず，もう使われていないものがあるかもしれないということです．それにより，xv6 には，時間が経つとともにディスクのスペースを使い切ってしまうというリスクがあります．

7.10 Code: inode の中身

ディスク上の inode である dinode 構造体は，サイズとブロック番号の配列を持ちます（図 7.3 を見てください）．その inode のデータは，addrs 配列が指すブロックに入っています．データの先頭 NDIRECT ブロックは，その配列の先頭 NDIRECT 要素に入っています．それらは, 直接ブロック (direct block) と呼ばれます．次の NINDIRECT ブロックは，inode そのものではなく，間接ブロック (indirect block) と呼ばれるデータブロックに入っています．addrs の最後のエントリには，間接ブロックのアドレスが入っています．よって，ファイルの先頭 12 kB (NDIRECT×BSIZE) は inode 内の

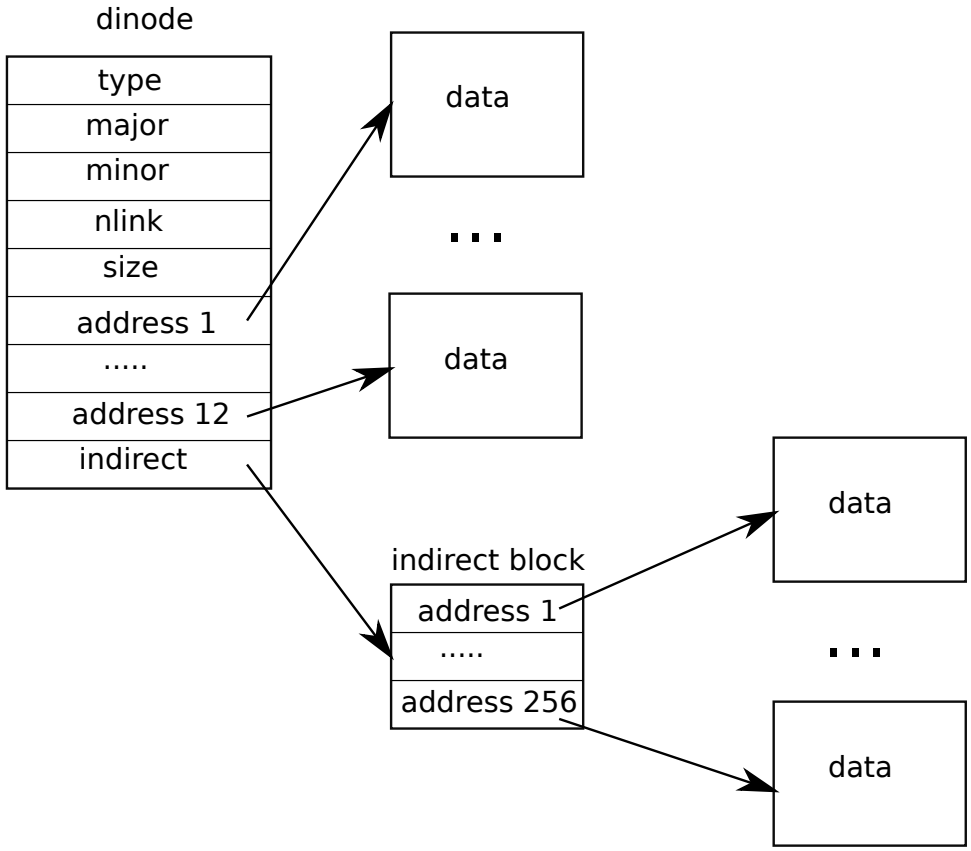

図 7.3　ディスク上でのファイルの表現

リストからロードすることができ，その次の 256 kB (NINDIRECT×BSIZE) は，間接ブロックを調べたあとでロードする必要があります．これは，ディスク上の表現としては優れていますが，利用者にとっては複雑です．bmap という関数が，すぐあとに出てくる readi や writei などの高レベルのルーチンのために，その表現を管理をします．bmap は，inode を表す ip に対し，bn 番目のデータブロックのブロック番号をリターンします．もし ip がまだそのブロックを持たなければ，bmap がアロケートします．

bmap 関数 (kernel/fs.c:379) は簡単なケースからはじめます．すなわち，inode 自体に記録されている最初の NDIRECT ブロックです (kernel/fs.c:384-388)．次の NINDIRECT ブロックは，ip->addrs[NDIRECT] の間接ブロックにあります．bmap は間接ブロックを読み (kernel/fs.c:395)，ブロック内の右側からブロック番号を読みます (kernel/fs.c:396)．もしブロック番号が NDIRECT+NINDIRECT を超えたら bmap はパニックを起こします．writei には，それを防ぐためのチェックがあります

(kernel/fs.c:494).

bmap は必要に応じてブロックをアロケートします．ip->addrs[] もしくは間接エントリがゼロであることが，ブロックがアロケートされていないことを示します．bmap はゼロを，demand (kernel/fs.c:385-386) (kernel/fs.c:393-394) を用いて新しくアロケートしたブロックの番号に置き換えます．

itrunc はファイルが含むブロックの解放を行い，その inode のサイズをゼロに戻します．itrunc (kernel/fs.c:414) はまず直接ブロックを解放し (kernel/fs.c:420-425)，続いて間接ブロックに記載されたブロックを解放し (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 のおかげで，readi と writei は簡単に inode のデータを取得できます．readi (kernel/fs.c:460) はまず，オフセットとカウント数がファイルの終端を超えないことを確認します．終端の先を開始点とする読み込みにはエラーをリターンします (kernel/fs.c:465-466)．一方，終端をまたぐ読み込みに対しては，要求よりも少ないバイト数をリターンします (kernel/fs.c:465-466)．メインループはファイルの各ブロックをスキャンし，バッファの各データを dst へコピーします (kernel/fs.c:470-478)．writei (kernel/fs.c:487) は readi と同じですが，3 つ例外があります．終端をまたぐ書き込みでは（最大サイズになるまでは）ファイルサイズを拡張します (kernel/fs.c:494-495)．ループでは，out ではなくバッファにデータをコピーします (kernel/fs.c:37)．もし書き込みがファイルを拡張したら，writei はそのサイズを更新します (kernel/fs.c:508-515).

readi と writei は，まず ip->type == T_DEV のチェックをします．これは，データがファイルシステム内には無い特殊ファイルの場合です．このケースについては，ファイルディスクリプタ層のところでまた説明します．

starti 関数 (kernel/fs.c:446) は inode のメタデータを stat 構造体へコピーします．ユーザプログラムは stat システムコールを呼ぶことで，この構造体を見ることができます．

7.11　Code: ディレクトリ層

ディレクトリは，内部的にはほとんどファイルです．ディレクトリの inode は T_DIR というタイプを持ち，その中身はディレクトリエントリが並んだものです．各エントリは dirent 構造体 (kernel/fs.h:56) で，名前と inode 番号を含みます．名前は最大で DIRSIZ (14) 文字までです．それより短い場合は NUL (0) で終端します．inode 番号がゼロのディレクトリは未使用です．

dirlookup 関数 (kernel/fs.c:531) は，指定された名前のファイルをディレクトリから探します．見つかったら，対応する（ロックされていない） inode へのポインタを

返します．また，呼び出し側が編集することに備えて，*poff に，そのディレクトリ内のエントリへのバイトオフセットをセットします．dirlookup が正しい名前のエントリを見つけたら，*poff を更新して，iget で得たロックされていない inode をリターンします．iget が inode をロックしないのは dirlookup のためです．呼び出し元がロックした dp を持っているので，もし探す対象がカレントディレクトリの別名である (.) であった場合，リターンする前に inode をロックしようとすると dp をさらにロックしようとしてデッドロックします（マルチプロセスと親ディレクトリ（..）に関連して，より入り組んだデッドロックのシナリオがあります．カレントディレクトリだけが問題なのではありません）．呼び出し元は dp をアンロックしてから ip をロックすることで，1 度に 1 つのロックしか取得しないことを保証できます．

dirlink 関数 (kernel/fs.c:558) は，ディレクトリ dp に対し，名前と与えられた inode 番号を持つ新しいディレクトリエントリを作ります．もしその名前がすでに存在したら，dirlink はエラーをリターンします (kernel/fs.c:564-568)．メインループはディレクトリエントリをスキャンして，未使用のエントリを探します．もし見つけたら，off にそのエントリのオフセットを記入してループを抜けます (kernel/fs.c:542-543)．そうでない場合は，off に dp->size をセットしてループを終えます．いずれの場合でも，dirlink は off が指す場所に新しいエントリを付け足します (kernel/fs.c:578-581)．

7.12 Code: パス名

パス名を探索するには，パスの各項目に対して何度も dirlookup を呼ぶ必要があります．namei (kernel/fs.c:665) はパスを評価し，対応する inode をリターンします．nameiparent 関数はその派生版で，最後の要素の一歩前で止め，親ディレクトリの inode をリターンするとともに，最後の要素を name に入れます．どちらの場合でも，より一般的な namex 関数が実際の処理を行います．

namex (kernel/fs.c:630) はまず，パス評価の起点を決めます．もしパスの先頭がスラッシュなら，評価はルートディレクトリから始めます．そうでない場合はカレントディレクトリです (kernel/fs.c:634-637)．namex は次に，skipelem を用いて，パスの各要素を順に調べていきます (kernel/fs.c:639)．ループの各くり返しでは，今調べている inode である ip から name を探します．探すにはまず ip をロックし，それがディレクトリかどうかを調べます．そうでなければ探索失敗です (kernel/fs.c:640-644)（ip をロックする必要があるのは，ip->type を変更するからではありません（変更できません）．ilock が走るまで，ip->type がディスクからロードされているとは限らないからです）．もしそれが nameiparent でパスの最後の要素であれば，nameiparent の定義の通り，そこでループを抜けます．最後の要素はすでに name

にコピーされているので，namex は単にロックを解除した ip (kernel/fs.c:645-649) を返します．そのループは最後に，その要素を dirlookup で探し，ip = next (kernel/fs.c:650-655) とすることで次のくり返しの準備をします．ループがパス名の最後までいったら，namex は ip をリターンします．

namex には長い時間がかかることがあります．パス名を解決するために訪れる複数のディレクトリに対し，inode とディレクリブロックを取得するために，（バッファキャッシュに入っていない場合）何度もディスクのオペレーションを行う必要があるからです．xv6 は注意深く設計されており，もし，あるカーネルスレッドの namex がディスク I/O でブロックしていたら，別のパス名を探している別のカーネルスレッドが並行して進むことができます．namex はパス上にあるディレクトリを別々にロックするので，別ディレクトリの探索は並行して行うことができるのです．

そのような並列処理ができることで，いくつかの新たな課題が生じます．たとえば，あるカーネルスレッドがパス名を探索しているあいだ，別のカーネルスレッドがあるディレクトリをアンリンクすることで，ディレクトリツリーが変わってしまうことがありえます．それにより，すでに消去されたディレクトリを探してしまう潜在的な危険性があります．その消去されたディレクトリのブロックは，別にディレクトリやファイルに再利用されている可能性があります．

xv6 ではそのような競合は起きません．たとえば，namex が dirlookup を実行しているあいだ，探索をしているスレッドはそのディレクトリのロックを所持するとともに，dirlookup は iget で取得した inode をリターンします．iget は，inode の参照カウントを増加させます．namex がそのディレクトリのロックを解放するのは，dirlookup から inode を受け取ったあとに限ります．別スレッドがその inode をディレクトリからアンリンクする可能性がありますが，その inode の参照カウントはゼロより大きいので，xv6 がその inode を削除することはありません．

別のリスクはデッドロックです．たとえば，「.」を探索するとき，next は ip と同じ inode を指す可能性があります．ip のロックを解放する前に next をロックしようとするとデッドロックが起きます．このデッドロックを避けるために，namex は next のロックの取得前にそのディレクトリをアンロックします．それもまた，iget と ilock を分けておく重要な理由です．

7.13 ファイルディスクリプタ層

Unix のインタフェースのかっこいいところは，ほぼ全てのリソースがファイルとして表現されるところです．コンソールなどのデバイス，パイプ，そしてもちろん本物のファイルなどです．その一貫性をもたらすのがファイルディスクリプタ層です．

1 章で見たように, xv6 は, プロセスごとに開いたファイルのテーブル, すなわちファイルディスクリプタを与えます．開いたファイルは，それぞれ `file` (kernel/file.h:1) で表現されます．この構造体は，inode もしくはパイプをくるんで I/O オフセットを付け加えたものです．`open` を呼びだすと，そのたびに開いたファイル（新しい `file` 構造体）が作られます．複数のプロセスが独立に同じファイルを `open` すると，それぞれが異なる I/O オフセットを持ちます．一方，同一の開いたファイル（すなわち同一の `file` 構造体）は，ある 1 つのプロセスのファイルテーブルに複数回現れたり，複数のプロセスに現れることができます．そのようなことが起きるのは，あるプロセスが `open` を用いてファイルを開いたあとで `dup` で別名を作ったり，`fork` で子プロセスを作って共有したりした場合です．参照カウントが，特定の開いたファイルへの参照数を管理します．ファイルは，読み，書き，あるは両方のために開くことができます．それは `readable` と `writable` フィールドが管理します．

システム内のオープン済みファイルは, グローバルファイルテーブル `ftable` に入ります．このファイルテーブルはファイルをアロケートするための関数 (`filealloc`), 参照を作ったり複製する関数 (`filedup`)，参照を解放するための関数 (`fileclose`), および読み書きするための関数 (`fileread` と `filewrite`) があります．

先の 3 つは，今や見慣れたかたちをしています．`filealloc` (kernel/file.c:30) はファイルテーブルをスキャンして参照のないファイル (`f->ref == 0`) を探し，新しい参照をリターンします．`filedup` (kernel/file.c:48) は参照カウントを増加させ，`fileclose` (kernel/file.c:60) は減少させます．ファイルの参照カウントがゼロに達したら，`fileclose` は `type` に応じてパイプもしくは inode を解放します．

`filestat`, `fileread` および `filewrite` 関数が, ファイルへの `stat`, `read`, および `write` の処理をそれぞれ実装します．`filestat` (kernel/file.c:88) は inode に対してのみ使用可能で，`starti` を呼び出します．`fileread` と `filewrite` は，その処理が `open` 時のモードによって許可されているかどうかをチェックし，パイプあるいは inode 用の実装につなぎます．もしファイルが inode を表すのであれば，`fileread` と `filewrite` は I/O オフセットをその処理のオフセットとして用い，そのあとに増加させます (kernel/file.c:122-123) (kernel/file.c:153-154)．パイプにはオフセットという概念はありません．inode の関数では，呼び出し元がロックを管理しなくてはいけないのでした (kernel/file.c:94-96) (kernel/file.c:121-124) (kernel/file.c:163-166)．inode のロックには便利な副作用があり，読み書きのオフセットが自動で更新されます．そのため，同一ファイルへの同時書きこみがお互いを上書きすることはありません．ただし，両者は混ざってしまいます．

7.14 Code: システムコール

低レイヤが提供する関数のおかげで，ほとんどシステムコールの実装は単純です（(kernel/sysfile.c) を見てください）．ただし，いくつかのシステムコールはよく見る価値があります．

sys_link 関数と sys_unlink 関数はディレクトリを編集することで，inode への参照を作ったり削除したりします．これも，トランザクションの威力が分かる良い例です．sys_link (kernel/sysfile.c:120) はまず，2 つ引数である文字列 old と new を取得します (kernel/sysfile.c:125)．old は存在して，かつディレクトリでは無いものとします (kernel/sysfile.c:129-132)．sys_link は ip->nlink のカウント値を増加させます．sys_link は続いて nameiparent を呼び，new の親ディレクトリとパスの最後の要素を得ます (kernel/sysfile.c:145)．そのあと，old の inode を指す新しいディレクトリエントリを作ります (kernel/sysfile.c:148)．新しい親ディレクトリは存在しなくてはならず，かつ既存の inode と同じデバイス上になくてはなりません．inode 番号は，同一のデバイス上においてのみユニークだからです．そのようなエラーが起きたら, sys_link は逆戻りして ip->nlink を減少させます．

複数ブロックにまたがる更新をするときに，その実行順序を気にしなくてよくなるという意味で，トランザクションは実装を単純にします．トランザクションは，全部が成功するか，まったく成功しないかのいずれかです．もしトランザクションがなければ，リンクを作るまえに ip->nlink を更新すると，ファイルシステムは一時的に安全ではない状態になります．その間にクラッシュが起きたら大惨事です．そのことを気にしなくてもよいのは，トランザクションがあるからです．

sys_link は，既存の inode に新しい名前をつけます．create 関数 (kernel/sysfile.c:242) は，新しい inode に新しい名前をつけます．これは，ファイルを作る 3 つのシステムコールを一般化したものです．すなわち，新しい通常ファイルを作る O_CREATE フラグ付きの open, 新しいディレクトリを作る mkdir，および新しいデバイスファイルを作る mkdev です．sys_link と同様に，create はまず nameiparent を呼び，親ディレクトリの inode を得ます．create は続いて dirlookup を呼び，その名前がすでに存在するかどうかをチェックします (kernel/sysfile.c:252)．もし存在したときの処理は，create を呼び出したシステムコールによります．open は，mkdir・mkdev とは異なる意味を持ちます．もしその create が open のために呼び出されたものであり (type == T_FILE)，かつ既存のファイルが通常のファイルであれば，open はこれを成功とみなすので，create も同様にします (kernel/sysfile.c:256)．それ以外では，重複するファイル名は

エラーです (kernel/sysfile.c:257-258). もしその名前が使用済みでなければ，create は ialloc (kernel/sysfile.c:261) を用いて新しい inode をアロケートします．もし新しい inode がディレクトリであれば，. と .. のエントリで初期化します．ここまででで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します (kernel/sysfile.c:274)．create は，sys_link と同様に，2 つの inode のロック ip と dp を同時に取得しますが，デッドロックの危険はありません．ip は今まさに取得したものであるため，他のプロセスが ip のロックを取得した上で dp をロックしようとすることはありえないからです．

create を使うと，sys_open, sys_mkdir, および sys_mknod は簡単に実装できます．sys_open (kernel/sysfile.c:287) がもっとも複雑なのは，ファイルを作る以外の仕事もするからです．O_CREATE フラグ付きで open が呼び出されると，open は create を呼び出します (kernel/sysfile.c:301)．そうでない場合，open は namei を呼びます (kernel/sysfile.c:307)．create はロックされた inode をリターンしますが，namei はそうではありません．そのため，inode 自体のロックは sys_open が行わなくてはなりません．sys_open は，ディレクトリが書き込みでなく読み込み用に開かれることチェックする良い場所です．inode を何らかの方法で得たら，sys_open はファイルおよびファイルディスクリプタをアロケートし (kernel/sysfile.c:325)，ファイルの中身を書き込みます (kernel/sysfile.c:337-342)．なお，初期化途中のファイルは現在のプロセスのテーブルにしか入っていないので，他のプロセスがアクセスすることはありえません．

6 章では，ファイルシステムに知る前にパイプの実装を見ました．sys_pipe 関数は，パイプのペアをつくることで，その実装とファイルシステムをつなげます．引数は整数 2 つ分の領域へのポインタです．sys_pipe 関数はパイプをアロケートし，ファイルディスクリプタを引数の領域に書き込みます．

7.15 世の中のオペレーティングシステム

世の中のオペレーティングシステムの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．しかし目的は同じで，ディスクアクセスをキャッシュすることと同期化することです．xv6 のバッファキャッシュは V6 と同じで，単純な LRU の立ち退きポリシーを採用しています．より複雑なポリシーがたくさんありえ，ある種の負荷に特化した性能を持ちます．より効率的な LRU キャッシュは，最も使われていないものを探し，かつヒープを維持するために，リンクリストではなくハッシュテーブルを使います．現代的なバッファキャッシュは，メモリにマップされた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 のロギングシステムは非効率です．まず，ファイルシステム系のシステムコールと並行してコミットを行うことができません．また，あるブロックにおける変更が数バイトしか無い場合でも，システムはブロック全体のログをとります．1 ブロックずつ同期的にログ書き込みを行いますので，それぞれごとに，ディスク 1 周分の時間がかかる可能性があります．世の中のロギングシステムは以上の問題を解決しています．

ロギングだけがクラッシュリカバリの方法ではありません．初期のファイルシステムは，再起動時にスカベンジング（ゴミ漁り，たとえば，Unix の `fsck`）をして，全ファイル・ディレクトリ・ブロック・inode の自由リストを調べ，一貫性がない部分を解決していました．大きいファイルシステムでは，スカベンジングには何時間もかかり，かつ一貫性のない箇所を（システムコールがアトミックであるかのようには）修正できない場合がありました．ログを用いた復帰ははるかに速く，クラッシュ時にシステムコールをアトミックにできます．

xv6 は，inode とディレクトリを配置するのに，初期の Unix と同様の基本的なディスク上のレイアウトを用います．この方法は何年ものあいだ，驚くほど維持されてきました．BSD の UFS/FFS および Linux の ext2/ext3 も，本質的には同じデータ構造を持ちます．ファイルシステムのレイアウトにおいてもっとも非効率な部分はディレクトリです．各探索において，全てのディスクブロックを線形にスキャンしなくてはならないからです．ディレクトリが数ブロックしかないときはリーズナブルですが，大量のファイルを持つディレクトリでは高コストになりえます．少なくとも，Microsoft Windows の NTFS，Mac OS X の HFS，そして Solaris の ZFS などは，ディスク上のバランスしたブロックのツリーとしてディレクトリを実装します．これは複雑な方法ですが，ディレクトリ探索が対数時間で終わることを保証してくれます．

xv6 はディスクのフェイルに対してナイーブです．もしディスクのオペレーションがフェイルしたら xv6 はパニックになります．これがリーズナブルかどうかはハードウェアによります．もし，オペレーティングシステムが乗っているのが，ディスクのフェイルを吸収できる冗長性を備えた特別なハードウェアであれば，オペレーティングシステムがフェイルに出会うのは稀なので，パニックになっても大丈夫でしょう．一方で，通常のディスクを使うオペレーティングシステムは，フェイルを見越してより穏やかな方法で処理し，ファイルの 1 ブロックの欠けがシステム全体に波及しないようにすべきです．

xv6 は，ファイルシステムが単一のディスクデバイスに収まっており，かつサイズが変わらないと想定します．大きなデータベースやマルチメディアにおける大容量化への要求は増すばかりなので，オペレーティングシステムは「1 ディスク 1 ファイルシステム」というボトルネックを解消するように発展しています．基本的なアプローチ

は，複数のディスクを組み合わせて単一の論理的なディスクにすることです．RAID などのハードウェア的な方法が現在でも一般的ですが，そのための機能をできるだけソフトウェアで実現する方向にトレンドは向かっています．そのようなソフトウェア実装は，通常，ディスクをその場で足したり減らしたりすることで，論理的なデバイスを大きくしたり小さくしたりするリッチな機能を持ちます．ストレージ層がその場で大きくなったり小さくなったりするのであれば，ファイルシステムも同じことができなくてはなりません．固定長配列に inode ブロックを入れる xv6 の方法は，そのような環境では使うことができません．ディスクの管理とファイルシステムを分離することがもっともクリアな設計ですが，それらのインタフェースが複雑なので，Sun の ZFS のように両者を組み合わせることもあります．

xv6 のファイルシステムは，現代的なファイルシステムにはあるたくさんの機能が欠けています．たとえば，スナップショットの作成や，インクリメンタルなバックアップをすることがでません．

現代的な Unix システムでは，ディスク上のストレージと同じシステムコールを用いて，さまざまな種類のリソースにアクセスすることができます．名前付きパイプ，ネットワーク接続，遠隔からアクセスするネットワークファイルシステム，および `/proc` のような監視・制御用のインタフェースです．xv6 では， `fileread` や `filewrite` の中の `if` 文で条件分岐をしています．それに対して，現代的なシステムでは，開いたファイルごとにオペレーションに対応する関数ポインタのテーブルを与えることで，inode ごとに特有の実装を呼び出せるようになっています．ネットワークファイルシステムとユーザレベルファイルシステムが提供する関数には，それらの呼び出しをネットワークごしの RPC (remote procedure call) に変換し，レスポンスが得られるまで待つ機能があります．

7.16 練習問題

1. `balloc` がパニックを起こす可能性があるのはなぜでしょうか？ xv6 はそこから復帰できるでしょうか？
2. `ialloc` がパニックを起こす可能性があるのはなぜでしょうか？ xv6 はそこから復帰できるでしょうか？
3. ファイルを使い切ったときに `filealloc` がパニックを起こさないのはなぜでしょうか？それがより一般的で，かつ処理する価値があるのはなぜでしょうか？
4. `sys_link` において，`iunlock(ip)` と `dirlink` の間で，`ip` に対応するファイルが別プロセスによってアンリンクされたとします．リンクは正しく作

られるでしょうか？それはなぜでしょうか？

5. create は進むために必要な関数呼び出しを 4 回行います (そのうち 1 つは ialloc で，残り 3 つは dirlink)．どれか 1 つでも欠ければ create は panic を呼びます．それでも良いのはなぜでしょうか？これらの 4 つの関数コールがどれも失敗しないのはなぜでしょうか？
6. sys_chdir は，p->cwd をロックするかもしれない iput(cp->cwd) の前に，iunlock(ip) を呼びます．しかし，iunlock(ip) を iput のあとまで延期してもデッドロックは起きません．それはなぜでしょうか？
7. lseek システムコールを実装してください．lseek をサポートするには，filewrite も改造する必要があります．lseek が f->ip->size を超えた場合，ファイルのその領域をゼロで埋める必要があるためです．
8. open に，O_TRUNC と O_APPEND を追加して，シェルで > および » オペレータが動くようにしてください．
9. ファイルシステムを改造して，シンボリックリンクをサポートしてください．
10. ファイルシステムを改造して，名前付きパイプをサポートしてください．
11. ファイルシステムと仮想メモリシステムを改造して，mmap をサポートしてください．

第 8 章

並行性・再び

良い並列性能，並列処理のおける正しさ，および理解しやすいコードの 3 つを満たすことは，カーネル設計における大きな課題です．素直なロックの使用はよい方法ですが，いつもできるわけではありません．この章では，xv6 がややこしい方法でロックを使わなくてはならなかった例と，ロック的だがロックではない方法を使う例を紹介します．

8.1 ロックのパターン

キャッシュされたアイテムをロックするのは難しいことがあります．たとえば，ファイルシステムのブロックキャッシュ(kernel/bio.c:26) は，ディスクブロックのコピーを NBUF 個まで持つことができます．あるディスクブロックに対し，キャッシュ内のコピーが最大でも 1 個であることは非常に重要です．そうしないと，同じブロックを示す異なるキャッシュが食い違うかもしれないからです．キャッシュされたブロックは buf 構造体 (kernel/buf.h:1) に入ります．buf 構造体にあるロック用のフィールドが，あるブロックを一度にひとつのプロセスしか使えないことを保証します．しかし，ロックだけでは足りません．ブロックがそもそもキャッシュに入ってなかったらどうなるでしょうか？（そのブロックはキャッシュされていないので）buf 構造体は存在せず，よってロックそのものが存在しません．xv6 はそのような状況を取り扱うために，キャッシュブロックの素性と追加のロック (bcache.lock) を関連づけます．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なくてはならないコード（たとえば bget (kernel/bio.c:58)）や，複数のキャッシュブロックを書き換えるコードは bcache.lock を取得しなくてはなりません．必要なブロックや buf 構造体を見つけたら，そのコードは bcache.lock を解放し，それからブロック固有のロックを取得します．アイテムの集合に対するロック 1 つと，各アイテムごとにロックの組

み合わせは頻出のパターンです.

通常，ロックを取得した関数がその解放も行います．しかし，より正確な理解は，アトミックに見えなくてはならないシーケンスの始まりでロックを取得し，そのシーケンスの終わりにロックを解放する，というものです．もしシーケンスの始まりと終わりが別の関数，別のスレッド，もしくは別の CPU にあるならば，ロックの取得・解放もそうしなくてはなりません．ロックの機能は他の利用者を待たせることであって，あるデータを特定のエージェントに紐付けすることではありません．1 つの例は `yield` (kernel/proc.c:500) における `acquire` で，`acquire` を呼んだプロセスではなく，スケジューラスレッドで解放されます．さらに別の例は `ilock` (kernel/fs.c:290) における `acquiresleep` です．このコードはディスクを読みながらスリープすることがよくあり，すると別の CPU で起床します．すなわち，ロックの解放が別の CPU で行われます.

オブジェクトには，そのオブジェクト自身に埋め込んであるロックによって保護されているものがあり，そのようなオブジェクトを解放（消去) するのはデリケートな仕事です．正しく解放するには，ロックだけでは足りないからです．問題が起きるのは，別のスレッドがそのオブジェクトを使用するために `acquire` で待っていた場合です．オブジェクトの解放は，ロックの解放を意味するので，待っていたスレッドが誤動作してしまいます．解決法の 1 つは，そのオブジェクトへの参照を数えておき，最後の参照が消滅した時点でのみ解放を行うというものです．`pipeclose` (kernel/pipe.c:59) がその例です．`pi->readopen` と `pi->writeopen` が，そのパイプを参照するファイルディスクリプタが存在するかどうかを追跡しています.

8.2 ロック的なパターン

多くの場面で，xv6 は参照カウントまたはフラグをソフトロック的なものとして利用することで，そのオブジェクトがアロケート済みで，解放したり再利用してはならないことを表します．プロセスの `p->state` は，ファイル, `inode`, および `buf` 構造体の参照カウントと同様に動きます．いずれの場合でも，ロックが存在してフラグ・参照カウントを保護しますが，オブジェクトの予期せぬ解放を防ぐのは後者です.

ファイルシステムは `inode` 構造体の参照カウントを複数プロセスが共有するロックのようなものとして使うことで，通常のロックならば生じてしまうデッドロックを回避します．たとえば，`namex` (kernel/fs.c:630) のループは，パス名に含まれる各部位に対応する名前のディレクトリをロックします．しかし `namex` は，各ループの最後でそれぞれのロックを解放しなくてはなりません．複数のロックを持つと，ドットを含むパス名（たとえば `a/./b`）のときに自身とデッドロックするからです．そのディ

レクトリと .. に並行アクセスすることによるデッドロックの可能性もあります．この問題は，7 章で説明したように，ロックはせずに，しかし参照カウントは増やしたままで inode 構造体を次のループに持ち越すことで解決できます．

その時々で異なるメカニズムで保護され，場合によっては明示的なロックではなく，xv6 のコード構造による暗黙的な並行アクセスによって保護されるデータもあります．たとえば，ある物理ページが使用可能なとき，それは kmem.lock (kernel/kalloc.c:24) によって保護されます．もしそのページがそのあとパイプとしてアロケートされたら (kernel/pipe.c:23), 今度は別のロック (埋め込まれた pi->lock) で保護されます．また，そのページが新しいプロセスのユーザメモリとしてさらに再アロケートされたら，ロックによる保護は無くなります．そのかわり，「アロケータはそのページを（解放されるまで）別のプロセスに渡さない」という事実が，そのページを並行アクセスから保護します．新しいプロセスのメモリの所有権は複雑です．まず，fork では親プロセスがアロケートと操作を行い，そのあとで子プロセスが利用します．いずれ子プロセスが終了したら，親プロセスが再びそのメモリを所有して kfree に渡します．このことから学ぶべきことが 2 つあります．第一に，ライフタイムのときどきにおいて，異なる方法で並行処理から保護されるデータオブジェクトがある．第二に，明示的なロックではなく，その暗黙的な構造によって保護がなされることがある．

最後のロック的なものの例は，mycpu() (kernel/proc.c:66) 呼び出しの周辺で割込を無効化することです．割込を無効化すると，呼び出し元のコードは，コンテキストスイッチを引き起こすタイマ割込に対してアトミックになります．そうすることで，プロセスを別 CPU に移すことができるようになります．

8.3 ロックしない

xv6 は，まったくロックを使わずに，書き換え可能なデータを共有する箇所も少しあります．一例はスピンロックの実装ですが，RISC-V のアトミック命令を，ハードウェア実装されたロックだと解釈することもできます．別の例は main.c の started 変数(kernel/main.c:7) で，0 番 CPU が xv6 の初期を終えるまで，他の CPU が走るのを防ぐためのものです．ここでは，volatile を使用することで，コンパイラがたしかにロード命令とストア命令を生成することを保証しています．最後の例は proc.c において p->parent を利用する箇所のいくつかです (kernel/proc.c:383) (kernel/proc.c:291)．そこで正しくロックを行うとデッドロックを引き起こす可能性があり，かつ他のプロセスが同時に p->parent を書き換えることができないのはほとんど明らかです．第四の例は p->killed で，p->lock を保持する間にセットされますが (kernel/proc.c:596)，ロックを取得することなくチェックされます (kernel/trap.c:56).

ある 1 つの CPU またはスレッドがデータを書き，また別の CPU またはスレッドが読むが，そのデータのための専用のロックが無いという箇所が xv6 にはあります．一例は `fork` で，親プロセスが子プロセスのユーザメモリページに書き込みを行い，子プロセス（別のスレッド，場合によっては別の CPU）がそれを読みますが，それらのページを明示的に保護するロックはありません．親プロセスが書き終えるまで子プロセスは実行を開始しませんので，厳密に言えば，これはロックが必要な箇所ではありません．しかし，メモリバリアが無い限り，ある CPU による書き込みが別 CPU に伝わる保証はないので，潜在的なメモリの順序問題（5 章を見てください）があります．しかし，親プロセスがロックを解放するとともに，子プロセスは開始時にロックを取得するので，`acquire` と `release` にあるメモリバリアが，親プロセスが書いた内容を子プロセスが見れることを保証します．

8.4 並列性

ロックとは，突き詰めれば，正しさのために並列性を抑制することです．パフォーマンスもまた重要なので，カーネル設計者は，正しさと並列性を両立するためのロックの使い方についてよく考えます．xv6 は性能のためにシステマティックに設計されたわけではありませんが，xv6 のオペレーションのうち並列処理できるものと，ロックで競合するものについて考えることには価値があります．

xv6 のパイプは，極めてよい並列性を持つものの例です．各パイプには固有のロックがあるので，異なるプロセスは（異なる CPU で），異なるパイプを並列で読み書きできます．ただし，同じパイプを読み書きするプロセス同士は，ロックの解放をお互いに待たなくてはなりません．すなわち，同一のパイプに対する同時読み書きはできません．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）についても同様ですが，これはロックに関連することではありません．

コンテキストスイッチはさらに複雑な例です．それぞれ別の CPU で実行している 2 つのカーネルスレッドは，`yield`, `sched`, および `swtch` を同時に呼び，並列で実行することがありえます．どちらのスレッドもロックを取得しますが，それらは別のロックなので，お互いを待つことをしません．ただしスケジューラ内では，テーブルから `RUNNABLE` なプロセスを探すループにおいて，2 つの CPU はコンフリクトするかもしれません．すなわち，xv6 は，コンテキストスイッチ時に CPU が複数あることのご利益をうけることができますが，おそらく改善の余地があります．

別の例は，別の CPU で実行する別のプロセスが，並行して `fork` を呼び出した場合です．それらは，`pid_lock`, `kmem.lock`, およびプロセステーブルから `UNUSED` プロセスを探すためのプロセス固有のロックのためにお互いを待つ可能性がありま

す．一方，2 つの fork は，ユーザページのコピーとページテーブル用ページの初期化を完全に並列で処理できます．

上記の例で示したロック方式は，並列性を犠牲にする場合があります．いずれのケースでも，より練られた設計をすることで，並列度を改善することができます．やる価値があるかどうかは，その詳細によります．すなわち，そのオペレーションが呼ばれる頻度，コードがロックの取り合いに費やす時間，それらのコードに別のボトルネックは無いか，などです．あるロック方式がパフォーマンスの問題を生じているかどうか，また新しい方式により大きな改善を得られるかどうかを推定するのは難しいことがあります．よって，現実的な負荷での計測が必要になることがしばしばあります．

8.5 練習問題

1. xv6 のパイプ実装を改造して，異なるコアから同一パイプへの読み込みと書き出しが，同時に実行できるようにしてください．
2. xv6 の scheduler() を改造して，異なるコアが同時に実行可能プロセスを探すときに生じるロックの取り合いを減らしてください．
3. xv6 の fork() におけるシリアル化をいくつか減らしてください．

第 9 章

まとめ

このテキストは，オペレーティングシステムの 1 つである xv6 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ィアを紹介しました．数行のコードが主要なアイディアの本質を体現することがあるので (例: コンテキストスイッチ，ユーザとカーネルの境界，ロックなど)，どの行も重要です．また，数行のコードは，オペレーティングシステムのある特定のアイディアをどうやって実装するかを表しており，かつそれらは簡単に別の方法に取り替えることができます（例: より良いスケジューリングアルゴリズム，より良いファイルを表すディスク上のデータ構造，並行するトランザクションを可能とするより良いロギングなど)．それらのアイディアの説明を，とても成功しているシステムコールインタフェースである Unix インタフェースの文脈で行いました．しかし，それらのアイディアは，他のオペレーティングシステムでも役に立つはずです．

参考文献

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

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

[3] Hans-J Boehm. Threads cannot be implemented as a library. *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] Donald Knuth. *Fundamental Algorithms. The Art of Computer Programming. (Second ed.)*, volume 1. 1997.

[7] John Lions. *Commentary on UNIX 6th Edition*. Peer to Peer Communications, 2000.

[8] Martin Michael and Daniel Durich. The NS16550A: UART design and application considerations. http://bitsavers.trailing-edge.com/components/national/_appNotes/AN-0 1987.

[9] David Patterson and Andrew Waterman. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[10] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

索引