Think OS

オペレーティングシステムの概要

バージョン0.7.4

Think OS

オペレーティングシステムの概要

バージョン0.7.4

アレン・B・ダウニー
グリーン ティー・プレス
ニーダム、マサチューセッツ州

Copyright © 2015 Allen B. Downey.

Green Tea Press
9 Washburn Ave

ニーダム MA 02492

クリエイティブ・コモンズ表示-非営利-継承の条件の下で、この文書をコピー、配布、および/または変更することが許可されています。

http://creativecommons. org/licenses/by-nc-sa/4.0/で公開されている4.0 International Licenseを使用しています。
本書のLATEXソースは、http://greenteapress.com/ thinkosから入手できます。

Preface

多くのコンピュータサイエンスプログラムにおいて、「オペレーティングシステム」は上級者向けのトピックです。このクラスを履修する頃には、学生はC言語でのプログラミング方法を知っていますし、コンピュータ・アーキテクチャのクラスも履修しているでしょう。通常、このクラスの目的は、学生がオペレーティングシステムの設計と実装に触れることであり、この分野の研究をしたり、OSの一部を書いたりする学生が出てくることを暗に想定しています。

この本は異なる読者を対象としており、目標も異なります。Olin CollegeのSoftware Systemsというクラスのために開発したものです。

このクラスを受講する学生のほとんどはPythonでプログラミングを学んでいますので、C言語を習得させることが目的のひとつです。その部分では、オライリーメディアの『Griffiths and Grif- fiths, Head First C』を使用しています。この本は、それを補完するためのものです。

私の生徒の中でオペレーティングシステムを書く人はほとんどいませんが、多くの生徒はC言語で低レベルのアプリケーションを書いたり、組み込みシステムを担当したりします。私の授業では、オペレーティングシステム、ネットワーク、データベース、組み込みシステムなどの教材を使用していますが、プログラマーが知っておくべきトピックに重点を置いています。

この本は、あなたがコンピュータ・アーキテクチャーを学んだことを前提としていません。進めていくうちに、必要なことを説明していきます。

この本が成功すれば、プログラムが動作するときに何が起こっているのか、そしてプログラムをより良く、より速く動作させるためには何をすればよいのかを理解することができるようになるはずです。

第1章では、コンパイル言語とインタプリタ言語の違いを、コンパイラの

仕組みを交えて説明しています。おすすめの読み物です。Head First C 第1章

第2章では、オペレーティングシステムがプロセスを使用して、実行中のプログラムが互いに干渉しないように保護する仕組みについて説明します。第3章では、仮想メモリとアドレス変換について説明しています。読むことをお勧めします。Head First C 第2章

第4章は、ファイルシステムとデータストリームについてです。 推薦図書です。

Head First C Chapter 3.

第5章では、数字や文字などの値がどのようにエンコードされるかを説明し、ビット演算子を紹介しています。

第6章では、ダイナミック・メモリー・マネージメントの使用方法とその

仕組みについて説明しています。おすすめの読み物です。Head First C 第

6章

第7章では、キャッシングとメモリ階層について。
第8章は、マルチタスクとスケジューリングについてです。

第9章では、POSIXのスレッドとミューテックスについて説明しています。推奨される読み物 Head First C 第12章とLittle Book of Semaphores 第1章と第2章です。

第10章では、POSIXの条件変数とproducer/consumer問題について説明しています。推奨される読み物 Little Book of Semaphores 第3章および第4章。
第11章では、POSIXセマフォの使用とC言語でのセマフォの実装について説明します。

今回のドラフトの注意点

この本の現在のバージョンは初期のドラフトです。文章を書いている間は、まだ図を入れていません。そのため、図ができたときに説明が大幅に改善されると思われる箇所がいくつかあります。

0.1 Using the code

本書のサンプルコードは、https://github.com/ AllenDowney/ThinkOSから入手できます。Git は、プロジェクトを構成するファイルを管理するた

めのバージョン管理システムです。Git の管理下にあるファイルの集合体をリポジトリと呼びます。GitHub は、Git リポジトリのストレージと便利な Web インターフェイスを提供するホスティングサービスです。
私のリポジトリのGitHubホームページには、コードを操作するためのいくつかの方法が用意されています。

- You can create a copy of my repository on GitHub by pressing the Fork button. If you don't already have a GitHub account, you'll need to create one. After forking, you'll have your own repository on GitHub

- このリポジトリは、本書の執筆中に書いたコードを記録するために使用できます。そして、そのレポをクローンします。つまり、ファイルを自分のコンピュータにコピーするのです。
- Or you could clone my repository. You don't need a GitHub account to do this, but you won't be able to write your changes back to GitHub.
- If you don't want to use Git at all, you can download the files in a Zip file using the button in the lower-right corner of the GitHub page.

Contributor List

ご提案や修正がありましたら、downey@allendowney.com までメールをお送りください。　いただいたご意見をもとに変更を加えた場合は、コントリビューターリストに追加させていただきます（省略を希望された場合を除く）。
エラーが表示されている文章の少なくとも一部を含めていただければ、私が検索しやすくなります。ページ番号やセクション番号でも構いませんが、作業性はあまり良くありません。ありがとうございました。

- I am grateful to the students in Software Systems at Olin College, who tested an early draft of this book in Spring 2014. They corrected many errors and made many helpful suggestions. I appreciate their pioneering spirit!
- James P Giannoules spotted a copy-and-paste error.
- Andy Engle knows the difference between GB and GiB.
- Aashish Karki noted some broken syntax.

他にも、Jim Tyson、Donald Robertson、Jeremy Vermast、Yuzhong Huang、Ian Hillなどが誤字・脱字を発見しています。

コンテンツ

1.1 第1章 編纂

1.2 Compiled and interpreted languages

プログラミング言語には、よく「コンパイル型」と「インタープリテッド型」があると言われる。「コンパイル」とは、プログラムを機械語に翻訳してハードウェアで実行することを意味し、「インタープリタ」とは、プログラムをソフトウェアのインタープリタで読み込んで実行することを意味する。通常、C言語はコンパイル型言語、Pythonはインタープリタ型言語とされています。しかし、この区別は必ずしも明確なものではありません。

まず、多くの言語にはコンパイルとインタープリタがあります。例えば、C言語のインタープリターやPythonのコンパイラーなどがあります。次に、Javaのように、プログラムを中間言語にコンパイルし、翻訳されたプログラムをインタプリタで実行するというハイブリッドな方法をとる言語があります。Javaでは、機械語に似たJavaバイトコードという中間言語を使用しますが、これを実行するのはソフトウェアインタプリタであるJava仮想マシン（JVM）です。

しかし、コンパイル言語とインタープリタ言語には、一般的な違いがあります。

1.3 Static types

多くのインタプリタ型言語は動的型をサポートしていますが、コンパイル型言語は通常、静的型に限定されます。静的型付けされている言語では

各変数がどのような型を指しているかは、プログラムを見て判断します。動的型付け言語では、プログラムが実行されるまで、変数の型がわからないことがあります。一般に、静的とはコンパイル時（プログラムがコンパイルされている間）、動的とはランタイム時（プログラムが実行されている間）に起こることを指します。

例えば、Pythonでは次のような関数を書くことができます。

```
def add(x, y):
return x + y
```

このコードを見ると、実行時にxとyがどのような型を指すのかがわかりません。この関数は何度も呼び出され、その度に異なる型の値が使われるかもしれません。加算演算子をサポートしている値であれば動作しますが、それ以外の型では例外や実行時エラーが発生します。

C言語では同じ関数を次のように書きます。

```
int add(int x, int y)
{ return x + y;
}
```

関数の1行目には，パラメータと戻り値の型宣言があります．xとyは整数と宣言されているので，コンパイル時にこの型に対して加算演算子が有効かどうかを確認できます（有効です）．また，戻り値も整数であることが宣言されています.

これらの宣言のおかげで、プログラムの他の場所でこの関数が呼ばれたとき、コンパイラは与えられた引数が正しい型であるかどうか、戻り値が正しく使われているかどうかをチェックすることができます。

これらのチェックは、プログラムが実行を開始する前に行われるので、エラーはより早く発見されます。さらに言えば、一度も実行されていないプログラムの一部にもエラーが発見される可能性があります。さらに、これらのチェックは実行時に行う必要がないため、コンパイル言語がインタープリタ言語よりも一般的に高速に動作する理由の一つとなっています。

また、コンパイル時に型を宣言することで、スペースを節約することができます。動的言語では，プログラムが実行されている間，変数名はメモリ

に保存され，プログラムからは十中八九アクセスできます．例えば，
Pythonの組み込み関数localsは，変数名とその値を含む辞書を返します．
以下は，Pythonインタプリタでの例です．

```
>>> x = 5
>>> print locals()
{'x': 5, 'ビルトイン': <モジュール 'ビルトイン'（ビルトイン）>,
'名前': 'メイン', 'ドック': なし, 'パッケージ': なし}。
```

これは、プログラムが実行されている間、変数の名前が（デフォルトの実行環境の一部である他の値とともに）メモリに保存されていることを示しています。
コンパイル言語では、変数名はコンパイル時には存在するが、ランタイムには存在しない。コンパイラは各変数の位置を決定し、その位置をコンパイルされたプログラムの一部として記録する1。実行時には、各変数の値はそのアドレスに格納されるが、変数名はまったく格納されない（ただし、デバッグのためにコンパイラが追加した場合は別）。

1.4 The compilation process

プログラマーであれば、コンパイル時に何が起こるのかというメンタルモデルを持っているはずです。このプロセスを理解していれば、エラーメッセージの解釈、コードのデバッグ、よくある落とし穴の回避などに役立ちます。
コンパイルの手順は

1. Preprocessing: C is one of several languages that include **preprocessing directives** that take effect before the program is compiled. For example, the #include directive causes the source code from another file to be inserted at the location of the directive.

2. Parsing: During parsing, the compiler reads the source code and builds an internal representation of the program, called an **abstract syntax tree**. Errors detected during this step are generally syntax errors.

3. Static checking: The compiler checks whether variables and values have the right type, whether functions are called with the right number and type of arguments, etc. Errors detected during this step are sometimes called **static semantic** errors.

4. Code generation: The compiler reads the internal representation of the program and generates machine code or byte code.

5. Linking: If the program uses values and functions defined in a library, the compiler has to find the appropriate library and include the required code.

1これは単純化したもので、詳細は後述します。

6. Optimization: At several points in the process, the compiler can transform the program to generate code that runs faster or uses less space. Most optimizations are simple changes that eliminate obvious waste, but some compilers perform sophisticated analyses and transformations.

通常、gccを実行すると、これらのステップがすべて実行され、EXCEL形式のファイルが生成されます。例えば、最小限のCプログラムを紹介しましょう。

```
#include <stdio.h>
int main()
{
    printf("Hello World¥n")。
}
```

このコードをhello.cというファイルに保存しておけば、次のようにコンパイルして実行することができます。

```
gcc hello.c
$ ./a.out
```

デフォルトでは、gccは実行コードをa.out（元々は「アセンブラ出力」の略）というファイルに保存します。2 行目は実行ファイルを実行します。接頭辞 ./ は、シェルにカレントディレクトリから探すように指示します。

通常、-oフラグを使用して実行ファイルのより良い名前を提供することは良いアイデアです。

```
$ gcc hello.c -o hello
$ ./hello
```

1.5 Object code

c」フラグは、プログラムのコンパイルとマシンコードの生成は行うが、リンクや実行ファイルの生成は行わないことをgccに指示します。

```
$ gcc hello.c -c
```

その結果、hello.oというファイルができます。oはオブジェクトコードの略で、コンパイルされたプログラムのことです。オブジェクトコードは実行可能ではありませんが、実行可能なプログラムにリンクすることができます。

UNIXのコマンド「nm」は、オブジェクトファイルを読み込んで、定義・使用する名前の情報を生成する。例えば、以下のようなものです。

```
$ nm hello.o
0000000000000000 T main
                 Uプット
```

この出力は、hello.oがmainという名前を定義し、putsという名前の関数を使用していることを示しています（put stringの略）。この例では、gccは、大きくて複雑な関数であるprintfを、比較的単純なputsに置き換えて最適化を行っています。
gccがどの程度最適化を行うかは、-Oフラグで制御できます。デフォルトではほとんど最適化されませんので、デバッグが容易になります。オプション

-O1は最も一般的で安全な最適化を行います。数字を大きくすると、コンパイルに時間がかかる追加の最適化が行われます。
理論的には、最適化によってプログラムの動作が変更されることはなく、スピードアップ以外の効果はありません。しかし、プログラムに微妙なバグがある場合、最適化によってそのバグが現れたり消えたりすることがあります。新しいコードを開発している間は、通常、最適化をオフにしておくのがよいでしょう。プログラムが動作し、適切なテストに合格したら、最適化をオンにして、テストに合格していることを確認します。

1.6 Assembly code

Sフラグは、-cフラグと同様に、プログラムをコンパイルしてアセンブリコードを生成するようgccに指示します。アセンブリコードとは、基本的にマシンコードを人間が読めるようにしたものです。

$ gcc hello.c -S

その結果、hello.sというファイルができ、以下のようになります。

```
        .file   "hello.c"
        .section    .rodata
.LC0:                   す。
                        .LFB0:
```

のメインとなりま

et 6, -16
.string "Hello World"
movq %rsp, %rb
.テキスト
p
.globl main
.type main, @function
.cfi_def_cfa_register 6
movl $.LC0, %edi
コールプット

.
c
f
i
_
s
t
a
r
t
p
r
o
c

p
u
s
h
q

%
r
b
p
.cfi_def_cfa_

offset 16

.

c

f

i

_

o

f

f

s

```
        movl $0, %eax
        popq %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:

        .size       main, .-main
        .ident      "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section    .note.GNU-stack,"",@progbits
```

gccは通常、実行しているマシン用のコードを生成するように設定されています。私の場合はx86アセンブリ言語を生成しますが、これはIntelやAMDなどのさまざまなプロセッサで動作します。私の場合はx86アセンブリ言語を生成しています。

1.7 Preprocessing

さらに、コンパイルの過程をさかのぼって、-Eフラグを使ってプリプロセッサだけを実行することもできます。

$ gcc hello.c -E

その結果、プリプロセッサからの出力が得られます。この例では、stdio.hからインクルードされたコード、stdio.hからインクルードされたすべてのファイル、それらのファイルからインクルードされたすべてのファイル、などが含まれています。私のマシンでは、合計で800行以上のコードが含まれています。ほとんどのCプログラムはstdio.hを含んでいるので、この800行のコードは何度もコンパイルされます。多くのCプログラムのようにstdlib.hもインクルードしている場合は、1800行以上のコードになります。

1.8 Understanding errors

コンパイルプロセスの手順がわかったので、エラーメッセージを理解するのが容易になりました。例えば、#includeディレクティブにエラーがあった場合、プリプロセッサからメッセージが表示されます。

hello.c:1:20: fatal error: stdioo.h: No such file or directory
compilation terminated.

構文エラーがあると、コンパイラからのメッセージが表示されます。

hello.c: 関数'main'内。

hello.c:6:1: error: expected ';' before '}' token

標準ライブラリで定義されていない関数を使用すると、リンカーからメッセージが表示されます。

/tmp/cc7iAUbN.o: In function `main':
hello.c:(.text+0xf): undefined reference to `printff'
collect2: error: ld returned 1 exit status

ldは、UNIXのリンカーの名前で、「ロード」がリンクと密接に関連したコンパイルプロセスのもう一つのステップであることから、この名前が付けられました。
プログラムが開始されると、Cは実行時のチェックをほとんど行わないので、実行時に見られる可能性のあるエラーはほんのわずかです。ゼロで割ったり、不正な浮動小数点演算を行ったりすると、"浮動小数点例外 "が発生します。また、メモリ上の不正な場所を読み書きしようとすると、"Segmentation fault "が発生します。

2.1 第2章 プロセス

2.2 Abstraction and virtualization

プロセスの話をする前に、いくつかの言葉を定義したいと思います。

- Abstraction: An abstraction is a simplified representation of something complicated. For example, if you drive a car, you understand that when you turn the wheel left, the car goes left, and vice versa. Of course, the steering wheel is connected to a sequence of mechanical and (often) hydraulic systems that turn the wheels, and the wheels interact with the road in ways that can be complex, but as a driver, you normally don't have to think about any of those details. You can get along very well with a simple mental model of steering. Your mental model is an abstraction.

  同じように、Webブラウザを使っている人は、リンクをクリックすると、そのリンク先のページが表示されることを理解していると思います。それを可能にしているソフトやネットワーク通信は複雑ですが、ユーザーとしてはその詳細を知る必要はありません。

  ソフトウェアエンジニアリングの大部分は、このような抽象化を設計することで、ユーザーや他のプログラマーが、強力で複雑なシステムを、その実装の詳細を知らなくても使えるようにすることです。

- Virtualization: An important kind of abstraction is virtualization, which is the process of creating a desirable illusion.
  例えば、多くの公共図書館は図書館間協力に参加しており、相互に本を借りることができます。私がリクエストすると

本を購入する際、地元の図書館の棚にある場合もあれば、他の蔵書から転送しなければならない場合もあります。どちらにしても、受け取り可能になると通知が来ます。その本がどこから来たのかを知る必要はないし、自分の図書館にどの本があるのかを知る必要もない。全体として、このシステムは、あたかも自分の図書館に世界中のすべての本が揃っているかのような錯覚をもたらします。

私の地元の図書館に物理的に置かれているコレクションは小さいかもしれませんが、私が利用できるコレクションには、図書館間の協力関係にあるすべての書籍が事実上含まれています。

また、ほとんどのコンピュータは1つのネットワークにしか接続されていませんが、そのネットワークは他のネットワークにも接続されているなど、様々な形で接続されています。インターネットとは、ネットワークの集合体であり、あるネットワークから次のネットワークにパケットを転送するためのプロトコルの集合体です。ユーザーやプログラマーの視点では、インターネット上のすべてのコンピュータが他のすべてのコンピュータに接続されているかのようにシステムが動作します。物理的な接続の数は少ないが、仮想的な接続の数は非常に多い。

仮想」という言葉は、仮想マシンの文脈でよく使われます。仮想マシンとは、特定のオペレーティングシステムを実行している専用のコンピューターであるかのような錯覚を起こすソフトウェアですが、実際には、仮想マシンは他の多くの仮想マシンとともに、別のオペレーティングシステムを実行しているコンピューター上で実行されている可能性があります。
仮想化の文脈では、実際に起こっていることを "物理的"、仮想的に起こっていることを "論理的 "または "抽象的 "と呼ぶことがあります。

2.3 Isolation

エンジニアリングの最も重要な原則の一つに「分離」があります。複数のコンポーネントからなるシステムを設計する場合、あるコンポーネントの変更が他のコンポーネントに望ましくない影響を与えないように、通常は

それらを互いに分離するのが良いとされています。

オペレーティングシステムの最も重要な目的の一つは、実行中の各プログラムを他のプログラムから分離し、プログラマーがあらゆる可能性のある相互作用について考えなくて済むようにすることです。この分離を実現するソフトウェアオブジェクトがプロセスです。
プロセスとは、実行中のプログラムを表すソフトウェアオブジェクトのことです。ここでいうソフトウェア・オブジェクトとは、オブジェクト指向プログラミングの意味で、一般的にはデータを含み、そのデータを操作するメソッドを提供するものを指します。プロセスは、以下のデータを含むオブジェクトである。

- The text of the program, usually a sequence of machine language instructions.
- Data associated with the program, including static data (allocated at compile time) and dynamic data (allocated at run time).
- The state of any pending input/output operations. For example, if the process is waiting for data to be read from disk or for a packet to arrive on a network, the status of these operations is part of the process.
- The hardware state of the program, which includes data stored in registers, status information, and the program counter, which indicates which instruction is currently executing.

通常、1つのプロセスが1つのプログラムを実行しますが、1つのプロセスが新しいプログラムをロードして実行することも可能です。

また、同じプログラムを複数のプロセスで実行することも可能であり、一般的に行われています。その場合、各プロセスは同じプログラムテキストを共有しますが、通常は異なるデータとハードウェアの状態を持ちます。ほとんどのOSは、プロセスを相互に分離するための基本的な機能を備えています。

- Multitasking: Most operating systems have the ability to interrupt a running process at almost any time, save its hardware state, and then resume the process later. In general, programmers don't have to think about these interruptions. The program behaves as if it is running continuously on a dedicated processor, except that the time between instructions is unpredictable.
- Virtual memory: Most operating systems create the illusion that each process has its own chunk of memory, isolated from all other processes. Again, programmers generally don't have to think about how virtual memory works; they can proceed as if every program has a dedicated chunk of memory.
- Device abstraction: Processes running on the same computer share the disk drive, the network interface, the graphics card, and other hardware. If processes interacted with this hardware directly, without coordination, chaos would ensue. For example, network data intended for one process might be read by another. Or multiple processes might try to store data in the same location on a hard drive. It is up to the operating system to maintain order by providing appropriate abstractions.

プログラマーとしては、これらの機能がどのように実装されているかを知る必要はあまりありません。しかし、もしあなたが好奇心を持っているならば、比喩的なフードの下で行われている多くの興味深いことを見つけるでしょう。そして、何が起こっているのかを知れば、より良いプログラマーになることができるでしょう。

2.4 UNIX processes

この本を書いている間、私が最も意識しているプロセスは、テキストエディターのemacsです。時々、ターミナル・ウィンドウに切り替えます。ターミナル・ウィンドウとは、コマンドライン・インターフェースを提供するUNIXシェルを実行するウィンドウのことです。

マウスを動かすと、ウィンドウマネージャが起動し、マウスがターミナルウィンドウの上にあることを確認し、ターミナルを起動します。ターミナルはシェルを起こします。シェルでmakeと入力すると、Makeを実行するための新しいプロセスが作成され、LaTeXを実行するための別のプロセスが作成され、さらに結果を表示するための別のプロセスが作成されます。

何かを調べる必要があるときは、別のデスクトップに切り替えることがありますが、そのときは再びウィンドウマネージャが起動します。Webブラウザのアイコンをクリックすると、ウィンドウマネージャはWebブラウザを実行するためのプロセスを作成します。Chromeのように、ウィンドウやタブごとに新しいプロセスを作成するブラウザもあります。

これは、私が認識しているプロセスだけです。 同時に、バックグラウンドでは他の多くのプロセスが実行されています。その多くは、OSに関連する処理を行っています。

UNIXのコマンドpsは、実行中のプロセスに関する情報を表示します。ターミナルで実行すると、以下のように表示されます。

```
  PID TTY          TIME CMD
 2687 pts/1    00:00:00 bash
 2801 pts/1    00:01:24 emacs
24762 pts/1    00:00:00 ps
```

1列目は、ユニークな数字のプロセスIDです。TTY」はテレタイプライターの略で、元々は機械式の端末でした。
3列目は、プロセスが使用したプロセッサの総時間を時、分、秒で表したものです。最後の列は、実行中のプログラムの名前です。この例では、bashは私がターミナルで入力したコマンドを解釈するシェルの名前、emacsはテキストエディタ、psはこの出力を生成するプログラムです。

デフォルトでは、ps は現在の端末に関連するプロセスのみをリストアップします。e フラグを使用すると、すべてのプロセスが表示されます（他のユーザに属するプロセスも含まれます。これは私の意見ではセキュリティ上の欠陥です)。

私のシステムでは、現在233のプロセスがあります。そのうちのいくつかを紹介します。

```
  PID TTY          TIME CMD
    1 ?        00:00:17 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:02 ksoftirqd/0
    4 ?        00:00:00 kworker/0:0
    8 ?        00:00:00 migration/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:16 rcu_sched
   47 ?        00:00:00 cpuset
   48 ?        00:00:00 khelper
   49 ?        00:00:00 kdevtmpfs
   50 ?        00:00:00 netns
   51 ?        00:00:00 bdi-default
   52 ?        00:00:00 kintegrityd
   53 ?        00:00:00 kblockd
   54 ?        00:00:00 ata_sff
   55 ?        00:00:00 khubd
   56 ?        00:00:00 md
   57 ?        00:00:00 devfreq_wq
```

initは、OSの起動時に最初に生成されるプロセスです。initは他の多くのプロセスを生成し、生成したプロセスが終了するまで待機しています。

kthreadd は、OS が新しいスレッドを作成するために使用するプロセスです。スレッドについては後で詳しく説明しますが、今のところ、スレッドは一種のプロセスと考えてください。頭のkはkernel（カーネル）の略で、スレッド作成などの中核機能を担うオペレーティングシステムの一部です。最後のdはdaemon（デーモン）を意味します。daemonは、バックグラウンドで動作し、オペレーティングシステムのサービスを提供する、このようなプロセスの別名です。この文脈では、"daemon "は助けてくれる精神という意味で使われており、邪悪な意味合いはありません。

ksoftirqdは、その名前から、カーネルデーモンであることが推測できます。

具体的には、ソフトウェア割り込み要求、つまり「ソフトIRQ」を処理します。
kworkerは、カーネルのために何らかの処理を行うためにカーネルが作成したワーカープロセスです。

これらのカーネルサービスを実行するプロセスは、しばしば複数存在します。私のシステムでは、現在、8つのksoftirqdプロセスと35のkworkerプロセスがあります。
他のプロセスについてはこれ以上詳しく説明しませんが、興味があれば検索してみてください。あなたのシステムでpsを実行して、私の結果と比較してみてください。

Chapter 3

Virtual memory

3.1 A bit of information theory

ビットとは2進数のことで、情報の単位でもあります。1ビットの場合、2つの可能性のうちの1つを指定することができ、通常は0と1と書きます。2ビットの場合、00、01、10、11の4つの可能な組み合わせがあります。一般的には、bビットの場合、2b個の値のいずれかを指定することができます。1バイトは8ビットなので、256の値のうちの1つを示すことができます。

逆に、アルファベットの1文字を格納したいとします。アルファベットは26文字ありますが、何ビット必要でしょうか？4ビットでは16個の値の中から1つを指定するだけなので、それでは足りません。5ビットなら32個の値を指定できるので、すべての文字に対応でき、数個の値を残せば十分です。

一般に，N個の値のうちの1つを指定したい場合は，$2^b \geq N$となるような最小のbの値を選ぶべきである。両辺の対数基数2をとると，$b \geqq \log_2 N$となる。

私がコインを投げて、その結果をあなたに伝えるとします。私はあなたに1ビットの情報を与えました。もし私が6面体のダイスを振ってその結果を伝えるなら、私はあなたに$\log 26$ビットの情報を与えたことになります。そして一般的に、結果の確率がN分の1であれば、その結果には$\log_2 N$ビットの情報が含まれていることになります。
同様に、結果の確率をpとすると、情報コンテントは$-\log_2 p$となる。この量は、結果の自己情報と呼ばれます。これは、結果がどれだけ意外なものであるかを測るもので、そのためsur-prisalとも呼ばれています。もしあなた

の馬が16分の1の確率で勝ち、その馬が勝った場合、あなたは4ビットの情報を得ることになります（配当金も一緒に）。しかし、人気のある馬が75%の確率で勝った場合、勝利のニュースは0.42ビットしか含まれていません。

直感的に、予想外のニュースは多くの情報を含んでいます。逆に、すでに確信していたことがあれば、それを確認しても情報量は少なくて済みます。本書のいくつかのトピックでは、ビット数bと、ビットがエンコードできる値の数N = 2bとの間で変換することに慣れておく必要があります。

3.2 Memory and storage

プロセスが実行されている間、そのデータのほとんどはメインメモリに保持されています。メインメモリは通常、何らかのランダムアクセスメモリ（RAM）です。現在のコンピュータの多くは、メインメモリが揮発性であるため、コンピュータがシャットダウンするとメインメモリの内容は失われる。一般的なデスクトップパソコンには、2〜8GiBのメモリが搭載されています。GiBとは「ジビバイト」の略で、230バイトのことである。

プロセスがファイルを読み書きする場合、それらのファイルは通常、ハードディスクドライブ（HDD）またはソリッドステートドライブ（SSD）に保存されます。これらのストレージデバイスは不揮発性であるため、長期的な保存に使用されます。現在、一般的なデスクトップコンピューターには、500GBから2TBの容量のHDDが搭載されています。GBは「ギガバイト」の略で、109バイトのことです。TBは「テラバイト」の略で、1012バイトである。

メインメモリのサイズには2進法のGiBを、HDDのサイズには10進法のGBとTBを使ったことにお気づきでしょうか。歴史的・技術的な理由から、メモリは2進法で、ディスクドライブは10進法で測定されています。本書では、2進法と10進法の単位を区別するように気をつけますが、「ギガバイト」という言葉やGBという略語は、しばしば曖昧に使われることがあるので注意が必要です。
カジュアルな使い方では、RAMだけでなく、HDDやSSDも含めて「メモリー」と呼ぶことがありますが、これらのデバイスの特性は大きく異なるので、区別する必要があるでしょう。ここでは、HDDやSSDのことをストレージと呼ぶことにします。

3.3 Address spaces

メインメモリの各バイトは、整数の物理アドレスで指定される。有効な物理アドレスの集合は、物理アドレス空間と呼ばれる。通常、0からN - 1までの範囲で構成され、Nはメインメモリのサイズです。1GiBの物理メモリを持つシステムでは、最も有効なアドレスは230 - 1で、10進法では1,073,741,823、16進法では0x3fff ffffとなります（接頭辞0xは16進法の数値を表します）。

しかし、ほとんどのOSでは仮想メモリが提供されており、プログラムは物理的なアドレスを扱うことはなく、物理的なメモリの容量を知る必要もありません。

その代わりに、プログラムは、0からM - 1までの番号が付けられた仮想アドレスで動作します。仮想アドレス空間の大きさは、オペレーティングシステムとその上で動作するハードウェアによって決定されます。

皆さんは、32ビットシステムや64ビットシステムという言葉を聞いたことがあると思います。これらの用語は、レジスタのサイズを示しており、通常、仮想アドレスのサイズでもあります。32ビットシステムでは、仮想アドレスは32ビットであり、仮想アドレス空間は0から0xffff ffffまでとなります。このアドレス空間のサイズは232バイト、つまり4GiBである。

64ビットシステムの場合、仮想アドレス空間のサイズは264バイト、つまり24-10246バイトです。これは16エクビバイトで、現在の物理メモリの約10億倍の大きさです。仮想アドレス空間が物理メモリよりもはるかに大きいというのは不思議な感じがするかもしれませんが、その仕組みはすぐにわかるでしょう。

プログラムがメモリ上の値を読み書きする際には、仮想的なアド・ドレスが生成されます。ハードウェアは、OSの助けを借りて、メインメモリーにアクセスする前に物理アドレスに変換する。この変換はプロセスごとに行われるため、2つのプロセスが同じ仮想アドレスを生成したとしても、物理メモリ上では異なる位置にマッピングされることになる。
このように、仮想メモリは、オペレーティングシステムがプロセスを相互に隔離する重要な手段の1つです。一般に、プロセスは他のプロセスのデータにアクセスすることができません。なぜなら、他のプロセスに割り当てられた物理メモリに対応する仮想アドレスを生成できないからです。

3.4 Memory segments

実行中のプロセスのデータは、5つのセグメントに整理されています。

- The **code segment** contains the program text; that is, the machine language instructions that make up the program.

- The **static segment** contains immutable values, like string literals. For example, if your program contains the string "Hello, World", those characters will be stored in the static segment.
- The **global segment** contains global variables and local variables that are declared static.

- The **heap segment** contains chunks of memory allocated at run time, most often by calling the C library function malloc.
- The **stack segment** contains the call stack, which is a sequence of stack frames. Each time a function is called, a stack frame is allocated to contain the parameters and local variables of the function. When the function completes, its stack frame is removed from the stack.

これらのセグメントの配置は、一部はコンパイラによって、一部はオペレーティングシステムによって決定されます。詳細はシステムごとに異なりますが、最も一般的な配置では

- The text segment is near the “bottom” of memory, that is, at addresses near 0.
- The static segment is often just above the text segment, that is, at higher addresses.
- The global segment is often just above the static segment.
- The heap is often above the global segment. As it expands, it grows up toward larger addresses.
- The stack is near the top of memory; that is, near the highest addresses in the virtual address space. As the stack expands, it grows down toward smaller addresses.

あなたのシステムでこれらのセグメントのレイアウトを決定するために、このプログラムを実行してみてください。このプログラムは、本書のリポジトリにあるaspace.cの中にあります（セクション0.1参照）。

```
#include <stdio.h>

#include <stdlib.h>.
int
global;
int main
()
{
    int local = 5;
    void *p = malloc(128);
    char *s = "Hello,
    World";

    printf ("Address of main is %p¥n",
```

main）；printf（"Address of global
is %p¥n", &global）；printf（"Address
of local is %p¥n", &local）。

```
    printf ("p points to %p¥n",
    p); printf ("s points
    to %p¥n", s);
}
```

mainは関数の名前です。変数として使われる場合は、mainの最初の機械語命令のアドレスを参照しており、テキストセグメントにあると予想されます。

globalはグローバル変数なので、グローバルセグメントにあることを期待しています
local
はローカル変数なので、スタック上にあることが予想されます。

sは「文字列リテラル」を意味します。これはプログラムの一部として現れる文字列です（ファイルから読み込まれる文字列やユーザーが入力する文字列とは異なります）。文字列の位置はスタティックセグメント内にあると考えられます（ローカル変数であるポインタsとは異なります）。

pには、ヒープ内のスペースを確保するmallocが返すアドレスが含まれています。「malloc "は "memory allocate "の略です。

フォーマットシーケンス%pは、printfに各アドレスを「ポインタ」としてフォーマットするよう指示し、結果を16進数で表示します。

このプログラムを実行すると、出力は次のようになります（読みやすくするためにスペースを入れました）。

```
Address of main  is    0x         40057d
Address  of  global is 0x         60104c
Address of local is  0x7ffe6085443c
p points to                0x        16c3010
s points to                0x         4006a4
```

予想通り、mainのアドレスが一番低く、次に文字列リテラルの位置が続きます。次にglobalの位置があり、次にpが指すアドレスがあります。localのアドレスはもっと大きいです。

最大のアドレスは、16進数で12桁。16進数の各桁は4ビットに対応するので、48ビットのアドレスとなります。つまり、仮想アドレス空間の使用可能な部分は248バイトということになります。

練習として、あなたのコンピュータでこのプログラムを実行し、私の結果と比較してみてください。mallocの2回目の呼び出しを追加して、システムのヒープが上に向かって（より大きなアドレスに向かって）成長するかどうかを確認してください。ローカル変数のアドレスを表示する関数を追加して、スタックが下がるかどうかを確認してください。

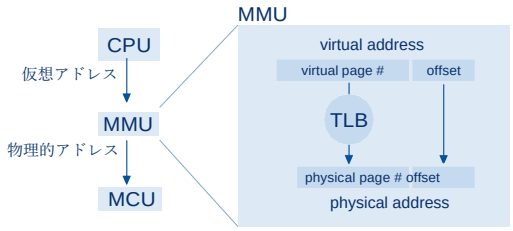

図3.1: アドレス変換処理のイメージ図

3.5 Static local variables

スタック上のローカル変数は、関数が呼ばれたときに自動的に割り当てられ、関数が戻ったときに自動的に解放されることから、自動変数と呼ばれることもあります。

C言語には、静的変数と呼ばれる別の種類のローカル変数があり、これはグローバルセグメントに割り当てられます。これはプログラムの開始時に初期化され、ある関数の呼び出しから次の関数の呼び出しまでその値を維持します。

例えば、次の関数は、自分が何回呼び出されたかを記録しています。

```
int times_called()
{
    static int counter = 0;
    counter++;
    カウンターを返す。
}
```

キーワード static は、counter が静的なローカル変数であることを示しています。初期化は、プログラムの開始時に一度だけ行われます。
この関数をaspace.cに追加すると、カウンタがスタックではなく、グローバル変数と一緒にグローバルセグメントに割り当てられることが確認できます。

3.6 Address translation

仮想アドレス（VA）はどのようにして物理アドレス（PA）に変換されるのでしょうか？基本的な仕組みは単純ですが、単純な実装では速度が遅す

ぎたり、スペースが大きすぎたりします。そのため、実際の実装はもう少し複雑になっています。

ほとんどのプロセッサには、CPUとメインメモリの間にメモリマネジメントユニット（MMU）が搭載されています。MMUは、VAとPAの間の高速変換を行います。

1. When a program reads or writes a variable, the CPU generates a VA.
2. The MMU splits the VA into two parts, called the page number and the offset. A “page” is a chunk of memory; the size of a page depends on the operating system and the hardware, but common sizes are 1–4 KiB.
3. The MMU looks up the page number in the translation lookaside buffer (TLB) and gets the corresponding physical page number. Then it combines the physical page number with the offset to produce a PA.
4. The PA is passed to main memory, which reads or writes the given location.

TLBには、（カーネルメモリに格納されている）ページテーブルからのデータのコピーがキャッシュされています。ページテーブルには、仮想ページ番号から物理ページ番号へのマッピングが含まれています。各プロセスはそれぞれ独自のページテーブルを持っているので、TLBは実行中のプロセスのページテーブルからのエントリーのみを使用するようにしなければなりません。
図3.1にこの処理のイメージ図を示します。どういう仕組みになっているかというと、VAが32ビットで、物理メモリが1GiBで、1KiBのページに分割されているとします。

- Since 1 GiB is 2^{30} bytes and 1 KiB is 2^{10} bytes, there are 2^{20} physical pages, sometimes called “frames.”
- The size of the virtual address space is 2^{32} B and the size of a page is 2^{10} B, so there are 2^{22} virtual pages.
- The size of the offset is determined by the page size. In this example the page size is 2^{10} B, so it takes 10 bits to specify a byte on a page.
- If a VA is 32 bits and the offset is 10 bits, the remaining 22 bits make up the virtual page number.
- Since there are 2^{20} physical pages, each physical page number is 20 bits. Adding in the 10 bit offset, the resulting PAs are 30 bits.

ここまでは、実現可能なことのように思えます。しかし、ページテーブル

の大きさはどのくらいになるのでしょうか。ページテーブルの最も単純な実装は、各仮想ページに1つのエントリを持つ配列です。各エントリには、物理ページ番号が含まれます。

この例では20ビットで、これに各フレームに関する追加情報を加えたものです。そのため、1エントリあたり3〜4バイトを想定しています。しかし、222の仮想ページを持つ場合、ページテーブルには224バイト、つまり16MiBが必要となる。

プロセスごとにページテーブルが必要なので、256のプロセスを実行するシステムでは、ページテーブルのためだけに232バイト、つまり4GiBが必要になります。これは32ビットの仮想アドレスでの話です。48ビットや64ビットの仮想アドレスでは、とんでもない数字になります。

幸いなことに、ほとんどのプロセスは仮想アドレス空間のほんの一部も使用しないので、実際にはそれほど大きなスペースは必要ありません。また、プロセスが仮想ページを使用しない場合は、ページテーブルにエントリを設ける必要はありません。

同じことを別の言い方で言うと、ページテーブルは「疎」であるということになります。これは、ページテーブルエントリの配列という単純な実装が良くないことを意味しています。幸いなことに、疎な配列のための良い実装がいくつかあります。

1つの選択肢は、Linuxを含む多くのオペレーティングシステムが採用しているマルチレベルページテーブルです。もう一つの方法は、各エントリに仮想ページ番号と物理ページ番号の両方を含める連想表です。ソフトウェアでは連想表の検索に時間がかかりますが、ハードウェアでは表全体を並行して検索することができるため、TLB内のページテーブルのエントリを表すのに連想配列がよく使われます。

これらの実装については、http://en.wikipedia. org/wiki/Page_tableに詳しく書かれています。しかし、面白いのは、ページテーブルは疎なので、疎な配列に適した実装を選択しなければならないということです。
先ほど、オペレーティングシステムは、実行中のプロセスを中断し、その状態を保存してから、別のプロセスを実行することができると述べました。この仕組みを「コンテキストスイッチ」と呼びます。各プロセスは独自のページテーブルを持っているので、OSはMMUと協力して各プロセスが

正しいページテーブルを取得するようにしなければなりません。古いマシンでは、コンテクストスイッチのたびにMMUのページテーブル情報を交換しなければならず、コストがかかりました。最近のシステムでは、MMUの各ページテーブルエントリにプロセスIDが含まれているため、複数のプロセスのページテーブルを同時にMMUに置くことができます。

Chapter 4

ファイルとファイルシステム

プロセスが終了（またはクラッシュ）すると、メインメモリーに保存されていたデータは失われます。しかし、ハードディスクドライブ（HDD）やソリッドステートドライブ（SSD）に保存されたデータは、プロセスが終了した後も、コンピューターがシャットダウンしても存続する「パーシステント（永続的）」なものである。

ハードディスクドライブは複雑です。データは、プラッター上に同心円状に配置されたトラックを構成するセクタに並べられたブロックに格納されている。

ソリッド・ステート・ドライブは、ブロックに秒単位で番号が振られているため、ある意味ではシンプルですが、ブロックごとに書き込める回数が限られているため、信頼性に欠けるという問題があります。

プログラマーとしては、このような複雑な問題には対処したくありません。必要なのは、永続的なストレージのハードウェアを適切に抽象化することです。最も一般的な抽象化は「ファイルシステム」と呼ばれるものです。抽象的ですが。

- A “file system” is a mapping from each file’s name to its contents. If you think of the names as keys, and the contents as values, a file system is a kind of key-value database (see https://en.wikipedia.org/wiki/Key-value_database).
- A “file” is a sequence of bytes.

ファイル名は通常、文字列であり、通常は「階層的」である。つまり、文字列は、最上位のディレクトリ（またはフォルダ）から、一連のサブディレクトリを経て、特定のファイルに至るパスを指定する。

抽象化と基本的なメカニズムの主な違いは、ファイルがバイトベースであることと、永続的なストレージがブロックベースであることです。オペレーティングシステムは、Cライブラリのバイトベースのファイル操作を、ストレージデバイスのブロックベースの操作に変換します。一般的なブロックサイズは1〜8KiBです。

例えば、次のコードでは、ファイルを開き、最初のバイトを読み取ります。

```
FILE *fp = fopen("/home/downey/file.txt",
"r"); char c = fgetc(fp);
```

fclose(fp)です。

このコードが実行されると

1. fopen uses the filename to find the top-level directory, called /, the subdirectory home, and the sub-subdirectory downey.

2. It finds the file named file.txt and "opens" it for reading, which means it creates a data structure that represents the file being read. Among other things, this data structure keeps track of how much of the file has been read, called the "file position".
   DOSでは、このデータ構造はファイルコントロールブロックと呼ばれていますが、UNIXでは別の意味になるので、この言葉は避けたいと思います。UNIXでは、良い名前がないようです。これはオープンファイルテーブルのエントリーなので、OpenFileTableEntryと呼ぶことにします。

3. When we call fgetc, the operating system checks whether the next character of the file is already in memory. If so, it reads the next character, advances the file position, and returns the result.

4. If the next character is not in memory, the operating system issues an I/O request to get the next block. Disk drives are slow, so a process waiting for a block from disk is usually interrupted so another process can run until the data arrives.

5. When the I/O operation is complete, the new block of data is stored in memory, and the process resumes. It reads the first character and stores it as a local variable.

6. When the process closes the file, the operating system completes or cancels any pending operations, removes data stored in memory, and frees the OpenFileTableEntry.

ファイルの書き込みのプロセスも同様ですが、いくつかの追加ステップがあります。ここでは、書き込み用のファイルを開き、最初の文字を変更する例を示します。

```
FILE *fp = fopen("/home/downey/file.txt", "w");
fputc('b', fp);
```

fclose(fp)です。

このコードが実行されると

1. Again, fopen uses the filename to find the file. If it does not already exist, it creates a new file and adds an entry in the parent directory, /home/downey.

2. The operating system creates an OpenFileTableEntry that indicates that the file is open for writing, and sets the file position to 0.

3. fputc attempts to write (or re-write) the first byte of the file. If the file already exists, the operating system has to load the first block into memory. Otherwise it allocates a new block in memory and requests a new block on disk.

4. After the block in memory is modified, it might not be copied back to the disk right away. In general, data written to a file is "buffered", which means it is stored in memory and only written to disk when there is at least one block to write.

5. When the file is closed, any buffered data is written to disk and the OpenFileTableEntry is freed.

要約すると、Cライブラリは、ファイル名からバイトのストリームにマッピングするファイルシステムの抽象化を提供しています。この抽象化は、実際にブロックで構成されているストレージデバイスの上に構築されています。

4.1 Disk performance

先ほど、ディスクドライブは遅いと言いました。現在のHDDでは、ディスクからメモリにブロックを読み出すのに平均5〜25msかかることがあります（https://en.wikipedia.org/wiki/Hard_disk_drive_performance_ 特性参照）。SSDはより高速で、4KiBのブロックを読むのに25 μ s、書き込むのに250 μ sかかります（http://en.wikipedia.org/wiki/Ssd#Controller 参照）。

この数字を、CPUのクロックサイクルと比較してみましょう。クロックレート2GHzのプロセッサは、次のように1回のクロックサイクルを実行します。0.5nsです。メモリからCPUに1バイトが届くまでの時間は、通常100ns程

度です。仮にプロセッサがクロックサイクルごとに1つの命令を完了するとすると、メモリからのバイトを待っている間に200の命令を完了することになる。

1マイクロ秒の間に2000個の命令を完了させるので、25秒待っている間に*SSDからのバイトを1μsで処理すると、5万回完了します。*

1ミリ秒で200万回の命令を実行しますから、HDDからのバイトを20ミリ秒待っている間に4000万回の命令を実行することになります。もし、CPUが待っている間に何もすることがなければ、それはアイドル状態である。OSがディスクからのデータを待っている間に別のプロセスに切り替えるのはそのためである。

- 主記憶装置と永続的記憶装置の間の性能差は、コンピュータシステム設計の大きな課題の1つです。オペレーティングシステムやハードウェアは、このギャップを「埋める」ためにいくつかの機能を提供しています。
    - Block transfers: The time it takes to load a single byte from disk is 5–25 ms. By comparison, the additional time to load an 8 KiB block is negligible. So systems generally try to read large blocks each time they access the disk.
    - Prefetching: Sometimes the operating system can predict that a process will read a block and start loading it before it is requested. For example, if you open a file and read the first block, there is a good chance you will go on to read the second block. The operating system might start loading additional blocks before they are requested.
    - Buffering: As I mentioned, when you write a file, the operating system stores the data in memory and only writes it to disk later. If you modify the block several times while it is in memory, the system only has to write it to disk once.
    - Caching: If a process has used a block recently, it is likely to use it again soon. If the operating system keeps a copy of the block in memory, it can handle future requests at memory speed.

  これらの機能の中には、ハードウェアで実装されているものもあります。例えば、ディスクドライブの中には、最近使用したブロックを保存するキャッシュを備えているものがあります。また、多くのディスクドライブでは、1つのブロックしか要求されていなくても、一度に複数のブロックを読み取ることができます。
  これらのメカニズムは、一般的にプログラムのパフォーマンスを向上させますが、動作を変えるものではありません。通常、プログラマーはこれらについて考える必要はありませんが、2つの例外があります。(1)プログラムのパフォーマンスが予想外に悪かった場合、問題を診断するためにこれ

らのメカニズムについて何か知っておく必要があるかもしれません。例えば、あるプログラムが値を表示した後にクラッシュした場合、その値はバッファの中にあるかもしれないので表示されないかもしれません。同様に、プログラムがデータをディスクに書き込んだ後にコンピュータの電源が切れた場合、データがキャッシュに入っていてまだディスクに入っていなければ、データが失われる可能性があります。

4.2 Disk metadata

ファイルを構成するブロックは、ディスク上に連続して配置されている場合があり、ファイルシステムのパフォーマンスは一般的に連続している方が優れていますが、ほとんどのオペレーティングシステムでは、連続した割り当てを必要としません。ディスク上のどこにでも自由にブロックを配置することができ、様々なデータ構造を用いてブロックを管理しています。

多くのUNIXファイルシステムでは、このデータ構造を「inode」と呼び、これは「index node」の略である。より一般的には、ファイルのブロックの位置など、ファイルに関する情報を「メタデータ」と呼ぶ。（ファイルの内容はデータなので、ファイルに関する情報はデータに関するデータであり、それゆえに「メタ」なのである）。

inodeは他のデータと一緒にディスク上に存在するため、ディスクブロックにきれいに収まるように設計されている。UNIXのinodeには、ファイルの所有者であるユーザーID、読み書きや実行が許可されている人を示す許可フラグ、最後に変更された時刻やアクセスされた時刻を示すタイムスタンプなど、ファイルに関する情報が含まれている。さらに、ファイルを構成する最初の12ブロックのブロック番号も含まれている。

ブロックサイズが8KiBの場合、最初の12ブロックで96KiBを構成します。ほとんどのシステムでは、大多数のファイルには十分な大きさですが、すべてのファイルに対応できる大きさではありません。そのため、inodeには「インダイレクトブロック」へのポインタも含まれており、このブロックには他のブロックへのポインタしか含まれていません。

インダイレクトブロック内のポインターの数は、ブロックのサイズとブロック番号に依存しますが、多くの場合、1024個です。1024のブロック番号と8KiBのブロックで、1つのインダイレクトブロックは8MiBを扱うことができます。これは最大級のファイルを除いては十分な大きさですが、それでもすべてのファイルを扱うには十分ではありません。

そのため、inodeには、インダイレクトブロックへのポインタを含む「ダブルインダイレクトブロック」へのポインタも含まれています。1024個

のインダイレクトブロックがあれば、8GiBをアドレスにすることができます。

それでも足りない場合は、ダブルインダイレクトブロックへのポインタを含むトリプルインダイレクトブロックがあり、最大ファイルサイズは8 TiBになります（最終的に）。UNIXのinodeが設計されたときは、これで十分な大きさだと思っていました。しかし、それはずっと昔の話だ。
インダイレクトブロックの代わりに、FATのようないくつかのファイルシステムでは、各ブロックに1つのエントリを含むファイルアロケーションテーブルを使用します。ルート・ディレクトリは、各ファイルの最初のクラスタへのポインタを含みます。各クラスタのFATエントリは、リンクリストのようにファイル内の次のクラスタを指します。詳細は、http://en.wikipedia.org/wiki/File_ Allocation_Tableを参照してください。

4.3 Block allocation

ファイルシステムは、どのブロックが各ファイルに属しているかを追跡する必要があり、また、どのブロックが使用可能であるかを追跡する必要もあります。新しいファイルが作成されると、ファイルシステムは使用可能なブロックを見つけてそれを割り当てます。また、ファイルが削除されると、ファイルシステムはそのブロックを再割り当てできるようにします。ブロック配分システムの目標は

- Speed: Allocating and freeing blocks should be fast.
- Minimal space overhead: The data structures used by the allocator should be small, leaving as much space as possible for data.
- Minimal fragmentation: If some blocks are left unused, or some are only partially used, the unused space is called “fragmentation”.
- Maximum contiguity: Data that is likely to be used at the same time should be physically contiguous, if possible, to improve performance.

特に、ファイルシステムの性能は、ファイルサイズやアクセスパターンなどの「ワークロードの特性」に依存するため、これらの目標をすべて達成するファイルシステムを設計することは困難です。ある作業負荷に対してよく調整されたファイルシステムでも、別の作業負荷ではあまり性能が良くないかもしれません。
そのため、ほとんどのOSが数種類のファイルシステムをサポートしており、ファイルシステムの設計は活発に研究開発が行われています。この10年間で、Linuxシステムは、従来のUNIXファイルシステムであるext2から、速度と連続性の向上を目的とした「ジャーナリング」ファイルシステムであるext3、そして最近では、より大きなファイルやファイルシステムを扱えるext4へと移行してきた。今後数年以内に、B-treeファイルシステムであるBtrfsへの移行が行われるかもしれない。

4.4 Everything is a file?

ファイルの抽象化は、実際には「バイトのストリーム」の抽象化であり、これはファイルシステムだけでなく、多くのものに有用であることがわか

りました。
その一例がUNIXパイプであり、これはプロセス間通信の簡単な形式である。あるプロセスの出力を別のプロセスの入力とするように、プロセスを設定することができます。最初のプロセスでは、パイプは書き込み可能なファイルのように動作するので、fputsやfprintfなどのCライブラリ関数を使用することができます。

2番目のプロセスでは、パイプは読み取り用に開かれたファイルのように動作するので

fgets と fscanf です。

ネットワーク通信にも、バイトのストリームという抽象化が使われている。
UNIXソケットは、異なるコンピュータ（通常）上のプロセス間の通信チャネルを表すデータ構造である。この場合も、プロセスは「ファイル」処理機能を使ってソケットからデータを読み取ったり、ソケットにデータを書き込んだりすることができる。
ファイルの抽象化を再利用することで、プログラマーは1つのAPI（アプリケーション・プログラム・インターフェース）を覚えればよいので、作業が楽になります。また、ファイルを扱うプログラムが、パイプやその他のソースから送られてくるデータも扱うことができるので、プログラムの汎用性が高まります。

第5章

その他のビット&バイト

5.1 Representing integers

コンピュータが数字を2進法で表現していることはご存じでしょう。正の数の場合、2進法での表現は簡単で、例えば510はb101となります。

負の数の場合、最もわかりやすい表現は、符号ビットを使って数値が正であるか負であるかを示すものです。しかし、「2の補数」と呼ばれる別の表現もあり、ハードウェアでの取り扱いが容易なため、こちらの方が一般的です。

負の数-xの2の補数を求めるには、xの2進表現を求め、すべてのビットを反転させて1を加えます。例えば、-510を表現するには、510の表現から始めます。すべてのビットを反転させて1を加えると、b11111011となります。

2の補数では、左端のビットが符号ビットのように機能し、正の数では0、負の数では1となります。

8ビットの数字を16ビットに変換するには、正の数の場合は0を増やし、負の数の場合は1を追加しなければなりません。つまり、符号ビットを新しいビットにコピーする必要があるのです。この作業を「符号拡張」といいます。

Cでは、符号なしと宣言しない限り、すべての整数型は符号付き（正負の数を表すことができる）です。その違いと、この宣言が重要な理由は、符号なし整数に対する演算では符号拡張が使われないことです。

5.2 Bitwise operators

C言語を学ぶ人は、ビット演算子の&と

|. これらの演算子は，整数をビットベクターとして扱い，対応するビットに対する論理演算を行います。

例えば、&は、両方のオペランドが1であれば1、そうでなければ0になるAND演算を計算します。以下は、2つの4ビットの数字に&を適用した例です。

```
  1100
& 1010
  -----
  1000
```

C言語では、12 & 10という式が8という値を持つことを意味します。

同様に、｜はOR演算を行い、オペランドのどちらかが1であれば1、そうでなければ0が得られます。

```
  1100
     |
1010
  -----
  1110
```

つまり、12｜10という式は、14という値を持つことになります。

最後に、^はXOR演算を行い、オペランドのどちらかが1であれば1が得られるが、両方は得られない。

```
  1100
^ 1010
  -----
  0110
```

つまり、12 ^ 10という式は、6という値を持っています。

一般的に、&はビットベクターからビットを消去するために、｜はビットをセットするために、^はビットを反転（トグル）するために使用されます。詳細は以下の通りです。

ビットをクリアする。任意の値xに対して、x&0は0、x&1はxとなります。

したがって、ベクターに3をANDすると、右端の2ビットだけが選択され、残りは0になります。

```
xxxx &
0011
  -----
  00xx
```

この文脈では、値3はいくつかのビットを選択し、残りのビットをマスクすることから、「マスク」と呼ばれています。
ビットを設定する。同様に、任意のxに対して、x|0はx、x|1は1となります。したがって、3でベクトルをORすると、右端のビットが設定され、残りは放置されます。

```
  xxxx
| OO11
  -----
  xx11
```

ビットをトグル。最後に、ベクトルを3とXORすると、右端のビットが反転し、残りのビットはそのままになります。練習として、^を使って12の2の補数を計算できるか見てみましょう。 ヒント：-1の2の補数表現は？

Cには、ビットを左右にシフトするシフト演算子<<と>>もあります。左にシフトすると数値は2倍になり、5 << 1は10、5 << 2は20となります。右にシフトするごとに2で割る（切り捨てる）ので、5 >> 1は2、2 >> 1は1となります。

5.3 Representing floating-point numbers

浮動小数点数は、科学的記数法の2進法で表される。10進法では、大きな数字は、係数と10を指数にしたものの積で書きます。例えば、光の速さ（m/s）は約2.998 - 108です。

ほとんどのコンピュータは、浮動小数点演算にIEEE規格を採用しています。C言語のfloatは通常32ビットのIEEE規格に対応し、doubleは通常64ビットの規格に対応しています。
32ビット規格では、左端のビットが符号ビット（s）、次の8ビットが指数（q）、最後の23ビットが係数（c）となります。

(-1)sc - 2q

これはほぼ正しいのですが、もう一つ問題があります。浮動小数点数は通常、ポイントの前に1桁の数字があるように正規化されます。例えば、基数10の場合、2998 - 105やその他の同等の表現ではなく、2.998 - 108を使用し

ます。基数2では、正規化された数字は常に2進法のポイントの前に1の数字があります。この位置の桁は常に1であるため、表現から除外することでスペースを節約できます。

例えば、1310の整数表現はb1101です。浮動小数点では、1.101 - 23なので、指数は3、格納される係数の部分は101（その後に0が20個続く）となります。

ほぼ正解なのですが、もうひとつ不思議なことがあります。指数は「バイアス」をかけて格納されます。32ビット規格では、バイアスは127なので、指数3は130として格納されます。

C言語で浮動小数点数をパックしたりアンパックしたりするには、ユニオン演算やビットワイズ演算を使います。以下にその例を示します。

```
union {
    float f;
    unsigned int
    u;
} p;

P.F = -13.0
符号付整数 sign = (p.u >> 31) & 1; 符
号付整数 exp = (p.u >> 23) & 0xff;



printf("%d¥n", sign);
printf("%d¥n", exp);
printf("0x%x¥n",
coef);
```

このコードは、本書のリポジトリにある float.c にあります（0.1 節参照）。

この組合わせにより、p.fを使って浮動小数点値を格納し、p.uを使ってそれを符号なし整数として読み出すことができます。

符号ビットを得るためには、ビットを右に31個シフトし、1ビットのマスクを使って右端のビットだけを選択します。

指数を求めるためには、ビットを23桁ずらし、右端の8ビットを選択します（16進数の0xffは1が8個）。

係数を得るためには、右端の23ビットを取り出し、残りのビットを無視する

必要があります。そのためには、右端の23ビットに1を、左端に0を配置したマスクを作ります。最も簡単な方法は、1を左に23個ずらして、1を引くことです。

このプログラムの出力は

1

130

0x500000

予想通り、負の数の符号ビットは1で、指数はバイアスを含めて130です。そして、16進数で表示した係数は、101の後に0が20個続きます。
練習として、64ビット規格を採用しているdoubleを組み立てたり、分解したりしてみてください。
http://en.wikipedia.org/wiki/IEEE_floating_point をご覧ください。

5.4 Unions and memory errors

C言語のユニオンには2つの一般的な使い方があります。1つは、前のセクションで見たように、データのバイナリ表現にアクセスすることです。もう1つは、異種のデータを格納することです。例えば、整数、浮動小数点、複素数、有理数などの数値を表現するのにユニオンを使用することができます。

しかし、ユニオンにはエラーがつきものです。ユニオンの中にどのような種類のデータが入っているかを把握するのは、プログラマーであるあなた次第です。浮動小数点の値を書いて、それを整数として解釈した場合、結果はたいてい無意味なものになります。

実は、メモリ上の位置を直角に読んでも同じことが起こります。その一つが、配列の終端を越えて読み込んだ場合です。

まず、スタック上に配列を確保し、0から99までの数字を入れる関数を作ってみます。

```
void f1() {
    int i;
    int array[100]。

        for (i=0; i<100;
        i++) { array[i] =
        i;
    }
}
```

次に、より小さな配列を作成し、開始前と終了後の要素に意図的にアクセ

スする関数を定義します。

```
void f2() {
    int x = 17;
    int
    array[10];
    int y = 123;

    printf("%d¥n", array[-2])となります。
```

printf("%d¥n", array[-1])で表示されます。
printf("%d¥n", array[10])となります。
printf("%d¥n", array[11])となります。
}

f1を呼び出してからf2を呼び出すと、次のような結果になります。

```
17
123
98
99
```

ここでの詳細は、スタック上に変数を配置するコンパイラに依存します。この結果から、コンパイラはxとyを隣り合わせにして、配列の「下」（低いアドレス）に配置したと推察できます。そして、配列の先を読むと、前の関数呼び出しによってスタックに残された値を取得しているように見えます。

この例では、すべての変数が整数なので、何が起こっているのかを理解するのは比較的簡単です。しかし、一般的に配列の境界を越えて読み取る場合、読み取った値はどのような型でもよいのです。たとえば、f1を変更して浮動小数点数の配列にすると、結果は次のようになります。

```
17
123
1120141312

1120272384
```

後者の2つの値は、浮動小数点の値を整数として解釈した場合に得られるものです。デバッグ中にこの出力に遭遇したら、何が起こっているのかを理解するのに苦労するでしょう。

5.5 Representing strings

文字列についても同様の問題が発生します。まず、C言語の文字列はヌル終端であることを覚えておいてください。文字列のスペースを確保する際には、末尾の1バイトを忘れないようにしましょう。
また、C言語の文字列に含まれる文字や数字はASCIIでエンコードされています。数字の「0」から「9」までのASCIIコードは、「0」から「9」ではなく「48」から「57」です。ASCIIコードの0は、文字列の終わりを示すNUL文字です。また、ASCIIコード1〜9は、一部の通信プロトコルで使用される特殊文字です。ASCIIコード7はベルで、一部の端末ではこれを印字すると音が出る。

アルファベット「A」のASCIIコードは65、「a」のコードは97です。これらのコードを2進法で表すとこうなります。

65 = b0100 0001

97 = b0110 0001

注意深く観察すると、この2つの文字が1ビット違うことに気づくだろう。このパターンは残りの文字にも当てはまります。右から数えて6ビット目は「ケースビット」として機能し、大文字では0、小文字では1となります。

練習問題として、文字列を受け取り、6ビット目を反転させることで、小文字から大文字に変換する関数を書いてみましょう。課題としては、文字列を1文字ずつではなく、32ビットまたは64ビットずつ読み込むことで、より高速なバージョンを作ることができます。文字列の長さが4バイトまたは8バイトの倍数であれば、この最適化は容易になります。

文字列の終わりを過ぎて読むと、おかしな文字が出てくる可能性があります。逆に、文字列を書いた後に、誤ってintやfloatとして読んでしまうと、結果が解釈しづらくなります。

例えば、実行した場合。

```
char array[] = "allen";
float *p = array;
printf("%f¥n", *p);
```

私の名前の最初の8文字をASCIIで表現し、倍精度の浮動小数点数として解釈すると、6977971387880058545766 4であることがわかります。

第6章

メモリ管理

C言語には、動的なメモリ割り当てのための4つの機能があります。

- malloc, which takes an integer size, in bytes, and returns a pointer to a newly-allocated chunk of memory with (at least) the given size. If it can't satisfy the request, it returns the special pointer value NULL.
- calloc, which is the same as malloc except that it also clears the newly allocated chunk; that is, it sets all bytes in the chunk to 0.
- free, which takes a pointer to a previously allocated chunk and deallocates it; that is, it makes the space available for future allocation.
- realloc, which takes a pointer to a previously allocated chunk and a new size. It allocates a chunk of memory with the new size, copies data from the old chunk to the new, frees the old chunk, and returns a pointer to the new chunk.

このAPIは、エラーが発生しやすいことで知られており、厳しいものです。メモリ管理は、大規模なソフトウェアシステムを設計する上で最も困難な部分の一つです。そのため、最近のほとんどの言語は、ガベージコレクションのような高レベルのメモリ管理機能を備えています。

6.1 Memory errors

C言語のメモリ管理APIは、テレビアニメ「ザ・シンプソンズ」のマイナーキャラクターであるジャスパー・ビアードリーに似ています。ジャスパー・ビアードリーは、厳格な臨時教師として登場し、すべての違反行為に対して「パドリン」という体罰を与えます。

ここでは、プログラムができることのうち、パドリングに値するものを紹介します。

- If you access (read or write) any chunk that has not been allocated, that's a paddling.
- If you free an allocated chunk and then access it, that's a paddling.
- If you try to free a chunk that has not been allocated, that's a paddling.
- If you free the same chunk more than once, that's a paddling.
- If you call realloc with a chunk that was not allocated, or was allocated and then freed, that's a paddling.

しかし、大規模なプログラムでは、1つのメモリがプログラムのある部分で割り当てられ、他のいくつかの部分で使用され、さらに別の部分で解放されることがあります。そのため、プログラムのある部分を変更すると、他の多くの部分を変更しなければならないことがあります。

また、プログラムのさまざまな部分で、割り当てられた同じチャンクへの参照、つまりエイリアスがたくさんあるかもしれません。そのチャンクは、そのチャンクへの参照がすべて使用されなくなるまで、解放されるべきではありません。これを正しく行うには、プログラムのすべての部分を注意深く分析する必要がありますが、これは困難であり、優れたソフトウェアエンジニアリングの基本原則に反するものです。

理想的には、メモリを確保するすべての関数は、ドキュメント化されたインターフェイスの一部として、そのメモリをどのように解放するかという情報を含むべきです。しかし、現実の世界では、ソフトウェアエンジニアリングの実践は、しばしばこの理想には及びません。
さらに悪いことに、メモリエラーは、その症状が予測できないため、見つけるのが難しい場合があります。例えば、以下のようなものです。

- If you read a value from an unallocated chunk, the system *might* detect the error, trigger a runtime error called a "segmentation fault", and stop the program. Or, the program might read unallocated memory without detecting the error; in that case, the value it gets is whatever happened to be stored at the accessed location, which is unpredictable, and might be different each time the program runs.

- If you write a value to an unallocated chunk, and don't get a segmentation fault, things are even worse. After you write a value to an invalid location, a long time might pass before it is read and causes problems. At that point it will be very difficult to find the source of the problem.

また、それ以上に悪いこともあります。Cスタイルのメモリ管理で最もよくある問題の1つは、mallocとfreeの実装に使用されるデータ構造（これについてはすぐに説明します）が、しばしば割り当てられたチャンクと一緒に保存されることです。そのため、動的に割り当てられたチャンクの終わりを超えて誤って書き込んでしまうと、これらのデータ構造が破壊されてしまいます。システムは通常、後になってmallocやfreeを呼び出し、それらの関数が不可解な方法で失敗するまで問題を検出しません。

ここから導き出される一つの結論は、安全なメモリ管理には設計と規律が必要だということです。メモリを確保するライブラリやモジュールを作成する場合は、メモリを解放するためのインターフェイスも提供する必要があり、メモリ管理は最初からAPI設計の一部であるべきです。

メモリを割り当てるライブラリを使用する場合は、APIの使用方法に気をつける必要があります。例えば、ライブラリがストレージの割り当てと解放を行う関数を提供している場合、それらの関数を使用し、例えば、mallocしていないチャンクに対してfreeを呼び出してはいけません。また、プログラムの異なる部分で同じチャンクへの複数の参照を維持することも避けるべきです。
多くの場合、安全なメモリ管理とパフォーマンスはトレードオフの関係にあります。例えば、メモリエラーの最も一般的な原因は、配列の境界を超えて書き込むことです。この問題を解決するには、境界チェックが必要です。つまり、配列にアクセスするたびに、インデックスが境界を越えていないかどうかをチェックする必要があります。配列のような構造を提供する高レベルのライブラリは，通常，境界チェックを行います。しかし，C言語の配列やほとんどの低レベルライブラリは境界チェックを行いません。

6.2 Memory leaks

もう1つのメモリエラーがありますが、これはパドリングに値するかどうかわかりません。メモリの塊を割り当てて、それを解放しなかった場合、それは「メモリリーク」です。

プログラムによっては、メモリリークをしても問題ないものもあります。例

えば、プログラムがメモリを確保し、その上で計算を行い、終了した場合、確保したメモリを解放する必要はないでしょう。プログラムが終了すると、そのプログラムのすべてのメモリはオペレーティングシステムによって解放されます。プログラムが終了する直前にメモリを解放することは、責任を感じさせるかもしれませんが、ほとんどの場合、時間の無駄です。
しかし、プログラムが長時間動作してメモリをリークしてしまうと、そのプログラムの総メモリ使用量は無限に増えてしまいます。その時、いくつかのことが起こるかもしれません。

- At some point, the system runs out of physical memory. On systems without virtual memory, the next call to `malloc` will fail, returning NULL.

- On systems with virtual memory, the operating system can move another process's pages from memory to disk and then allocate more space to the leaking process. I explain this mechanism in Section 7.8.
- There might be a limit on the amount of space a single process can allocate; beyond that, malloc returns NULL.
- Eventually, a process might fill its virtual address space (or the usable part). After that, there are no more addresses to allocate, so malloc returns NULL.

mallocがNULLを返したにもかかわらず、割り当てたと思われるチャンクに持続してアクセスした場合、セグメンテーション・フォールトが発生します。このような理由から、malloc を使用する前にその結果を確認するのが良いスタイルだと考えられています。ひとつの方法として、mallocを呼び出すたびに、次のような条件を追加することができます。

```
void *p = malloc(size);
if (p == NULL) {...
    perror("malloc
    failed"); exit(-1)とな
    ります。
}
```

perrorはstdio.hで宣言されており、エラーメッセージと、最後に発生したエラーについての追加情報を表示します。

stdlib.hで宣言されているexitは、プロセスを終了させます。引数には、プロセスがどのように終了したかを示すステータスコードを指定します。慣習上、ステータスコード0は正常終了を、-1はエラー状態を示します。異なるエラー状態を示すために他のコードが使用されることもあります。

エラーチェックコードは、プログラムを読みにくくするだけでなく、煩わしいものです。このような問題は，ライブラリの関数呼び出しとそのエラーチェックコードを自分の関数でラップすることで軽減することができます。例えば，戻り値をチェックする malloc のラッパーを以下に示します。

```
void *check_malloc(int size)
{
    void *p = malloc (size);
```

```
    if (p == NULL) {...
        perror("malloc
        failed"); exit(-1) とな
        ります。
    }
    pを返す。
}
```

メモリ管理は非常に難しいため、ウェブブラウザのような大規模なプログラムのほとんどは、メモリをリークします。システム上のどのプログラムが最も多くのメモリを使用しているかを確認するには、UNIXユーティリティーのpsとtopを使用します。

6.3 Implementation

プロセスが開始されると、システムは、テキストセグメントと静的に割り当てられたデータのためのスペース、スタックのためのスペース、そして動的に割り当てられたデータを含むヒープのためのスペースを割り当てます。

すべてのプログラムがデータを動的に割り当てるわけではないので、ヒープの初期サイズは小さいかゼロかもしれません。初期のヒープには1つのフリーチャンクだけが含まれています。

mallocが呼ばれると、十分な大きさの空きチャンクがあるかどうかをチェックします。もし見つからなければ、システムにもっとメモリを要求しなければなりません。それを行う関数が sbrk で、「プログラムブレーク」を設定します。これはヒープの終端へのポインタと考えることができます。

sbrkが呼ばれると、OSは物理メモリの新しいページを割り当て、プロセスのページテーブルを更新し、プログラムのブレークを設定します。

理論的には、プログラムは（mallocを使わずに）直接sbrkを呼び出し、ヒープを自分で管理することができます。しかし、mallocの方が使いやすく、ほとんどのメモリ使用パターンにおいて、高速に動作し、メモリを効率的に使用することができます。

ほとんどのLinuxシステムでは、メモリ管理API（malloc、free、calloc、reallocの各関数）を実装するために、Doug Lea氏が開発したdlmallocをベースにしたptmallocを使用しています。この実装の重要な要素を説明した短い論文が http://gee.cs.oswego.edu/ dl/html/malloc.html にあります。プログラマーにとって、最も重要な要素は、意識することです。

- The run time of malloc does not usually depend on the size of the chunk,

but might depend on how many free chunks there are. free is usually fast, regardless of the number of free chunks. Because calloc clears every byte in the chunk, the run time depends on chunk size (as well as the number of free chunks).

reallocは，新しいサイズが現在のサイズよりも小さい場合や，既存のチャンクを拡張するためのスペースがある場合には，高速に実行できることがあります。そうでない場合は、古いチャンクから新しいチャンクにデータをコピーしなければなりません。この場合、実行時間は古いチャンクのサイズに依存します。

- Boundary tags: When malloc allocates a chunk, it adds space at the beginning and end to store information about the chunk, including its size and the state (allocated or free). These bits of data are called "boundary tags". Using these tags, malloc can get from any chunk to the previous chunk and the next chunk in memory. In addition, free chunks are chained into a doubly-linked list; each free chunk contains pointers to the next and previous chunks in the "free list".

- mallocの内部データ構造を構成しているのは、バウンダリタグとフリーリストのポインターです。これらのデータ構造は、プログラムデータと混在しているため、プログラムのエラーで簡単に破損してしまいます。

- Space overhead: Boundary tags and free list pointers take up space. The minimum chunk size on most systems is 16 bytes. So for very small chunks, malloc is not space efficient. If your program requires large numbers of small structures, it might be more efficient to allocate them in arrays.

- Fragmentation: If you allocate and free chunks with varied sizes, the heap will tend to become fragmented. That is, the free space might be broken into many small pieces. Fragmentation wastes space; it also slows the program down by making memory caches less effective.

- Binning and caching: The free list is sorted by size into bins, so when malloc searches for a chunk with a particular size, it knows what bin to search in. If you free a chunk and then immediately allocate a chunk with the same size, malloc will usually be fast.

Chapter 7

Caching

7.1 How programs run

キャッシングを理解するためには、コンピュータがどのようにプログラムを実行するかを理解する必要があります。このテーマを深く理解するためには、コンピュータアーキテクチャを学ぶ必要があります。この章の目標は、プログラム実行の簡単なモデルを提供することです。

プログラムが起動するとき、コード（またはテキスト）は通常、ハードディスクやソリッドステートドライブ上にあります。オペレーティングシステムは、プログラムを実行するための新しいプロセスを作成し、「ローダー」がストレージからメインメモリにテキストをコピーし、mainを呼び出してプログラムを開始します。
プログラムが実行されている間、そのデータのほとんどはメインメモリーに格納されていますが、一部のデータは、CPU上の小さなメモリー単位であるレジスタに格納されています。これらのレジスターには

- The program counter, or PC, which contains the address (in memory) of the next instruction in the program.
- The instruction register, or IR, which contains the machine code instruction currently executing.
- The stack pointer, or SP, which contains the address of the stack frame for the current function, which contains its parameters and local variables.
- General-purpose registers that hold the data the program is currently

working with.

- A status register, or flag register, that contains information about the current computation. For example, the flag register usually contains a bit that is set if the result of the previous operation was zero.

プログラムが実行されているとき、CPUは「命令サイクル」と呼ばれる次のステップを実行します。

- Fetch: The next instruction is fetched from memory and stored in the instruction register.
- Decode: Part of the CPU, called the “control unit”, decodes the instruction and sends signals to the other parts of the CPU.
- Execute: Signals from the control unit cause the appropriate computation to occur.

ほとんどのコンピュータは、「命令セット」と呼ばれる数百種類の命令を実行することができます。しかし、ほとんどの命令は、いくつかの一般的なカテゴリーに分類されます。

- Load: Transfers a value from memory to a register.
- Arithmetic/logic: Loads operands from registers, performs a mathematical operation, and stores the result in a register.
- Store: Transfers a value from a register to memory.
- Jump/branch: Changes the program counter, causing the flow of execution to jump to another location in the program. Branches are usually conditional, which means that they check a flag in the flag register and jump only if it is set.

x86をはじめとするいくつかの命令セットでは、ロードと算術演算を組み合わせた命令が用意されている。

各命令サイクルでは、プログラムテキストから1つの命令が読み込まれる。また、一般的なプログラムでは、命令の約半分がデータのロードまたはストアを行います。ここに、コンピュータ・アーキテクチャの基本的な問題の1つである「メモリ・ボトルネック」が存在する。
現在のコンピューターでは、一般的なコアは1ns以下で命令を実行することができます。しかし、メモリとの間でデータを転送するのにかかる時間は約100ns。もしCPUが次の命令を取得するのに100ns、データを読み込むのに100ns待たなければならないとしたら、理論上可能な速度の200倍

も遅い命令を実行することになります。多くの計算では、CPUではなく、メモリが速度を制限する要因となります。

7.2 Cache performance

この問題の解決策、あるいは少なくとも部分的な解決策として、キャッシングがあります。キャッシュ」とは、CPUに物理的に近い場所にある小型で高速なメモリのことで、通常は同じチップ上にあります。

実際、現在のコンピュータにはいくつかのレベルのキャッシュが搭載されています。最も小型で高速なレベル1のキャッシュは1〜2MiBでアクセス時間は1ナノ秒程度、レベル2のキャッシュは4ナノ秒程度、レベル3のキャッシュは16ナノ秒程度となっています。

CPUは、メモリから値をロードすると、そのコピーをキャッシュに保存します。同じ値が再び読み込まれると、CPUはキャッシュされたコピーを取得するので、メモリを待つ必要がありません。

最終的にはキャッシュがいっぱいになります。そうなると、新しいものを入れるためには、何かを追い出さなければなりません。つまり、CPUがある値をロードした後、かなり後になってから再びロードすると、その値はもうキャッシュにはないかもしれないのです。

多くのプログラムのパフォーマンスは、キャッシュの有効性によって制限されます。CPUが必要とする命令やデータがたいていキャッシュに入っていれば、プログラムはCPUのフルスピードに近い状態で動作することができます。CPUがキャッシュにないデータを頻繁に必要とする場合、プログラムはメモリの速度によって制限されます。

キャッシュの「ヒット率」（h）は、メモリアクセスのうち、キャッシュ内のデータを見つけた割合であり、「ミス率」（m）は、メモリアクセスのうち、メモリに行かなければならなかった割合である。キャッシュのヒットを処理する時間をTh、キャッシュのミスを処理する時間をTmとすると、各メモリアクセスの平均時間は

$$hTh + mTm$$

同様に、「ミスペナルティ」をキャッシュミスの処理にかかる余分な時間、Tp = Tm - Thと定義することができます。すると、平均アクセスタイムは

$$Th + mTp$$

ミス率が低いと、平均アクセスタイムがThに近くなることがあります。つまり、あたかもメモリがキャッシュスピードで動いているかのようなパフォーマンスを発揮することができるのです。

7.3 Locality

プログラムが初めてバイトを読み込むとき、キャッシュは通常、要求されたバイトとその一部を含むデータの「ブロック」または「ライン」を読み込みます。

neighborsです。もし、プログラムが隣人の1つを読み進めても、それはすでにキャッシュされています。

例えば、ブロックサイズが64Bで、長さ64の文字列を読み込んだときに、その文字列の1バイト目がたまたまブロックの先頭にあったとします。最初のバイトを読み込んだときにはミスペナルティが発生しますが、その後は文字列の残りの部分がキャッシュに入ります。文字列全体を読み込んだ後のヒット率は63/64となり、約98%となります。文字列が2つのブロックにまたがっている場合は、2つのミスペナルティが発生します。しかし、それでもヒット率は62/64、つまりほぼ97%になります。その後、もう一度同じ文字列を読むと、命中率は100%になります。

逆に、プログラムが予測不能に飛び回り、メモリ上の散らばった場所からデータを読み出し、同じ場所に2度アクセスすることがほとんどない場合、キャッシュのパフォーマンスは低下します。
プログラムが同じデータを複数回使用する傾向を「時間的局所性」と呼ぶ。また、近くのデータを使う傾向を「空間的局所性」といいます。幸いなことに、多くのプログラムは両方のローカリティを自然に表示します。

- Most programs contain blocks of code with no jumps or branches. Within these blocks, instructions run sequentially, so the access pattern has spatial locality.
- In a loop, programs execute the same instructions many times, so the access pattern has temporal locality.
- The result of one instruction is often used immediately as an operand of the next instruction, so the data access pattern has temporal locality.
- When a program executes a function, its parameters and local variables are stored together on the stack; accessing these values has spatial locality.
- One of the most common processing patterns is to read or write the elements of an array sequentially; this pattern also has spatial locality.

次のセクションでは、プログラムのアクセスパターンとキャッシュのパフォーマンスの関係について説明します。

7.4 Measuring cache performance

カリフォルニア大学バークレー校の大学院生だった頃、ブライアン・ハーベイのコンピュータ・アーキテクチャのティーチング・アシスタントをしていました。私の好きな練習問題のひとつ

は，配列を繰り返し処理して，要素の読み書きにかかる平均時間を測定するプログラムを開発した。配列の大きさを変えることで、キャッシュの大きさやブロックの大きさなどの属性を推測することができます。

このプログラムの私の修正版は、この本のリポジトリのキャッシュディレクトリにあります（0.1節参照）。

このプログラムで重要なのは、このループです。

```
iters =
0; do {
    sec0 = get_seconds();

        for (index = 0; index < limit; index += stride)
        array[index] = array[index] + 1;

    iters = iters + 1;
    sec = sec + (get_seconds() - sec0);

} while (sec < 0.1);
```

内側のforループは，配列を走査します．limitは配列をどれだけ走査するかを決め，strideは走査する要素の数を決めます．例えば，limitが16，strideが4の場合，ループは要素0，4，8，12にアクセスすることになります．

sec は，内側のループが使用した合計 CPU 時間を記録する．外側のループは sec が 0.1 秒を超えるまで実行されますが，これは十分な精度で平均時間を計算できる長さです．

get_secondsは、システムコールclock_gettimeを使用し、秒数に変換し、結果をdoubleで返します。

```
double
get_seconds(){ struct
timespec ts;
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
return ts.tv_sec + ts.tv_nsec / 1e9;
}
```

配列の要素にアクセスする時間を分離するために、プログラムは2つ目のループを実行します。このループは、内側のループが配列に触れず、常に同じ変数をインクリメントするという点を除いて、ほとんど同じです。

```
iters2 =
0; do {
    sec0 = get_seconds();
```

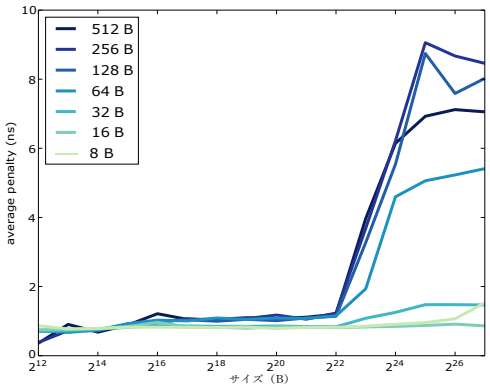

図 7.1: 配列サイズとストライドの関数としての平均ミスペナルティ。

```
        for (index = 0; index < limit; index +=
        stride) temp = temp + index;

    iters2 = iters2 + 1となります。
    sec = sec - (get_seconds() - sec0);

} while (iters2 < iters);
```

2つ目のループは、1つ目のループと同じ回数の繰り返しを行います。各反復の後、secから経過時間を差し引きます。ループが完了すると，secには，すべての配列アクセスにかかった時間からtempの増分にかかった時間を差し引いた時間が入ります．この差が，すべてのアクセスで発生したミスペナルティの合計となります．最後に，アクセス数で割って，アクセスごとの平均ミスペナルティをns単位で求めます．

sec * 1e9 / iters / limit * stride

cache.cをコンパイルして実行すると、次のような出力が得られます。

```
Size:      4096  Stride:        8  read+write:  0.8633  ns
Size:      4096  Stride:       16  read+write:  0.7023  ns
Size:      4096  Stride:       32  read+write:  0.7105  ns
Size:      4096  Stride:       64  read+write:  0.7058  ns
```

Pythonとmatplotlibがインストールされていれば、graph_data.pyを使って結果をグラフ化することができます。図7.1は、Dell Optiplex 7010で実行したときの結果です。配列のサイズとストライドは、配列の要素数ではなく、バイト数で表示されていることに注意してください。

このグラフを見て、キャッシュについてどのようなことが推測できるか考えてみましょう。ここでは、いくつかのことを考えてみましょう。

- The program reads through the array many times, so it has plenty of temporal locality. If the entire array fits in cache, we expect the average miss penalty to be near 0.
- When the stride is 4 bytes, we read every element of the array, so the program has plenty of spatial locality. If the block size is big enough to contain 64 elements, for example, the hit rate would be 63/64, even if the array does not fit in cache.
- If the stride is equal to the block size (or greater), the spatial locality is effectively zero, because each time we read a block, we only access one element. In that case we expect to see the maximum miss penalty.

要約すると，配列がキャッシュサイズよりも小さい場合や，ストライドがブロックサイズよりも小さい場合には，良好なキャッシュ性能が期待できる。配列がキャッシュよりも大きく、ストライドが大きい場合にのみ、パフォーマンスが低下します。

図7.1では、アレイが222B以下であれば、どのストライドでもキャッシュ性能は良好です。このことから、キャッシュサイズは4MiBに近いと推測されますが、実際にはスペック上は3MiBです。

ストライドが8、16、32Bの場合、キャッシュ性能は良好です。ストライドが8、16、32Bの場合、キャッシュ性能は良好ですが、64Bになると劣化し始め、ストライドが大きくなると平均ミスペナルティは約9nsになります。このことから、ブロックサイズが128B付近にあると推測されます。多くのプロセッサでは、小さくて高速なキャッシュと、大きくて低速なキャッシュを含む「マルチレベルキャッシュ」が採用されています。この例では、配列サイズが214Bより大きくなるとミスペナルティが少し大きくなるようなので、このプロセッサにはアクセスタイムが1ns以下の16KBのキャッシュも搭載されている可能性があります。

7.5 Programming for cache performance

メモリキャッシュはハードウェアで実装されているので、ほとんどの場合、プログラマーはキャッシュについてあまり知る必要はありません。しかし、キャッシュの仕組みを知っていれば、キャッシュをより効果的に使うプログラムを書くことができます。

例えば、大きな配列を扱う場合、配列を何度もトラバースするよりも、一度だけ配列をトラバースして、各要素に対していくつかの演算を実行した方が高速に処理できる場合があります。
2次元の配列を扱う場合、それは行の配列として格納されているかもしれません。要素をトラバースする場合，ストライドを行の長さに合わせて列ごとに行うよりも，ストライドを要素の大きさに合わせて行ごとに行った方が速くなります.

リンクされたデータ構造は、ノードが必ずしもメモリ上で連続していないため、常に空間的な位置関係を示すわけではありません。しかし、多くのノードを同時に割り当てた場合、それらは通常、ヒープ内で同位置に配置されます。さらに言えば、ノードの配列を一度に割り当てれば、それらが連続していることがわかります。

mergesortのような再帰的な戦略は、大きな配列をより小さな断片に分割し、その断片を使って作業するため、キャッシュの動作が良好な場合が多い。これらのアルゴリズムは、キャッシュの挙動を利用するように調整できる場合があります。
性能が重視されるアプリケーションでは、キャッシュのサイズやブロックサイズなど、ハードウェアの特性に合わせてアルゴリズムを設計することが可能です。このようなアルゴリズムを「キャッシュアウェア」と呼びます。キャッシュを意識したアルゴリズムの欠点は、ハードウェア固有のものであるということです。

7.6 The memory hierarchy

この章のどこかで、次のような疑問が浮かんだかもしれません。"キャッシュがメインメモリよりもはるかに高速であるならば、本当に大きなキャッシュを作って、メモリのことは忘れてしまえばいいのではないか？"

コンピュータアーキテクチャの話は抜きにして、エレクロニクスと経済性の2つの理由がある。キャッシュが高速なのは、小さくてCPUの近くにあるため、容量や信号の伝搬による遅延が最小限に抑えられるからです。キャッシュを大きくすれば遅くなります。

また、キャッシュはプロセッサチップ上のスペースを占有し、大きなチップは高価になる。メインメモリは通常、DRAM（ダイナミック・ランダム・アクセス・メモリ）で、1ビットあたり1つのトランジスタと1つのコンデンサしか使わないため、同じスペースに多くのメモリを詰め込むことができる。しかし、このメモリの実装方法は、キャッシュの実装方法よりも遅い。

また、メインメモリは通常、16個以上のチップを搭載したDIMM（Dual In-line Memory Module）にパッケージされている。1つの大きなチップよりも、いくつかの小さなチップの方が安価である。

速度、サイズ、コストの間のトレードオフが、キャッシュの基本的な理由です。速くて、大きくて、安いメモリ技術が一つあれば、他のものは必要ありません。
メモリだけでなく、ストレージにも同じ原理があります。SSD（ソリッド・ステート・ドライブ）は高速ですが、HDD（ハード・ドライブ）よりも高価なため、小型化される傾向にあります。テープドライブはハードディスクよりもさらに低速ですが、大容量のデータを比較的安価に保存することができます。

以下の表は、それぞれの技術の典型的なアクセス時間、サイズ、コストを示したものです。

Device	Access time	Typical size	Cost
Register	0.5 ns	256 B	?
Cache	1 ns	2 MiB	?
DRAM	100 ns	4 GiB	$10 / GiB
SSD	10 μs	100 GiB	$1 / GiB
HDD	5 ms	500 GiB	$0.25 / GiB
Tape	minutes	1–2 TiB	$0.02 / GiB

レジスターの数やサイズは、アーキテクチャの詳細に依存する。現在のコンピュータには約32本の汎用レジスタがあり、それぞれが1つの「ワード」を格納しています。32ビットコンピュータの場合、1ワードは32ビット（4B）、64ビットコンピュータの場合、1ワードは64ビット（8B）となり、レジスタファイルの総容量は100〜300Bとなる。

レジスターやキャッシュのコストは定量化しにくいものです。レジスターやキャッシュは、それらが搭載されているチップのコストに貢献していますが、消費者はそのコストを直接目にすることはありません。

表中のその他の数字は、オンラインのコンピュータハードウェアショップで販売されている代表的なハードウェアの仕様を調べたものです。あなたがこの記事を読む頃には、これらの数字は古くなっているでしょうが、ある時点での性能とコストのギャップを知ることができます。
これらの技術は「メモリ階層」を構成している（ここでの「メモリ」にはストレージも含まれていることに注意）。階層の各レベルは、上のレベルよりも大きく、遅くなっています。また、ある意味では、各階層は下の階層のキャッシュとして機能している。メインメモリは、SSDやHHDに永久保存されているプログラムやデータのキャッシュと考えることができます。また、テープに保存された非常に大きなデータセットを扱う場合は、ハードドライブを使ってデータのサブセットを一度にキャッシュすることができます。

7.7 Caching policy

メモリ階層は、キャッシングを考えるためのフレームワークを示唆してい

ます。階層の各レベルで、キャッシングの4つの基本的な問題に対処しなければなりません。

- Who moves data up and down the hierarchy? At the top of the hierarchy, register allocation is usually done by the compiler. Hardware on the CPU handles the memory cache. Users implicitly move data from storage to

- プログラムを実行したり、ファイルを開いたりする際には、メモリを使用します。しかし、オペレーティングシステムは、メモリとストレージの間でデータを行き来させます。階層の最下層では、管理者がディスクとテープの間で明示的にデータを移動させる。

- What gets moved? In general, block sizes are small at the top of the hierarchy and bigger at the bottom. In a memory cache, a typical block size is 128 B. Pages in memory might be 4 KiB, but when the operating system reads a file from disk, it might read 10s or 100s of blocks at a time.

- When does data get moved? In the most basic cache, data gets moved into cache when it is used for the first time. But many caches use some kind of "prefetching", meaning that data is loaded before it is explicitly requested. We have already seen one form of prefetching: loading an entire block when only part of it is requested.

- Where in the cache does the data go? When the cache is full, we can't bring anything in without kicking something out. Ideally, we want to keep data that will be used again soon and replace data that won't.

これらの質問に対する答えが「キャッシュポリシー」を構成する。階層の上の方では、キャッシュポリシーは高速である必要があり、ハードウェアで実装されているため、単純なものになりがちです。階層の下の方では、意思決定に時間がかかるため、適切に設計されたポリシーが大きな違いを生むことになります。
ほとんどのキャッシュポリシーは、「歴史は繰り返す」という原則に基づいています。最近の過去に関する情報があれば、それを使って直近の未来を予測することができます。例えば、あるデータブロックが最近使用されたならば、すぐにまた使用されることが予想されます。この原理を利用して、最近使用されていないデータブロックをキャッシュから削除する「最近使用されたものでないもの」（LRU）という交換ポリシーが提案されています。このトピックの詳細については、http://en.wikipedia.org/wiki/Cache_algorithms を参照してください。

7.8 Paging

仮想メモリを搭載したシステムでは、OSがメモリとストレージの間でペ

ージを行き来させることができます。6.2項で述べたように、この仕組みは「ページング」と呼ばれ、「スワッピング」と呼ばれることもあります。ここでは、その流れをご紹介します。

1. Suppose Process A calls malloc to allocate a chunk. If there is no free space in the heap with the requested size, malloc calls sbrk to ask the operating system for more memory.

2. If there is a free page in physical memory, the operating system adds it to the page table for Process A, creating a new range of valid virtual addresses.

3. If there are no free pages, the paging system chooses a "victim page" belonging to Process B. It copies the contents of the victim page from memory to disk, then it modifies the page table for Process B to indicate that this page is "swapped out".

4. Once the data from Process B is written, the page can be reallocated to Process A. To prevent Process A from reading Process B's data, the page should be cleared.

5. At this point the call to sbrk can return, giving malloc additional space in the heap. Then malloc allocates the requested chunk and returns. Process A can resume.

6. When Process A completes, or is interrupted, the scheduler might allow Process B to resume. When Process B accesses a page that has been swapped out, the memory management unit notices that the page is "invalid" and causes an interrupt.

7. When the operating system handles the interrupt, it sees that the page is swapped out, so it transfers the page back from disk to memory.

8. Once the page is swapped in, Process B can resume.

ページングがうまく機能すると、物理メモリの使用率が大幅に向上し、より少ないスペースでより多くのプロセスを実行できるようになります。その理由は以下の通りです。

- Most processes don't use all of their allocated memory. Many parts of the text segment are never executed, or execute once and never again. Those pages can be swapped out without causing any problems.

- If a program leaks memory, it might leave allocated space behind and never access it again. By swapping those pages out, the operating system can effectively plug the leak.

- On most systems, there are processes like daemons that sit idle most of the time and only occasionally "wake up" to respond to events. While they are idle, these processes can be swapped out.

- A user might have many windows open, but only a few are active at a time. The inactive processes can be swapped out.
- Also, there might be many processes running the same program. These processes can share the same text and static segments, avoiding the need to keep multiple copies in physical memory.

すべてのプロセスに割り当てられたメモリを合計すると、物理メモリのサイズを大幅に超えてしまうことがありますが、それでもシステムは正常に動作しています。

ある程度までは。

プロセスがスワップアウトされたページにアクセスすると、ディスクからデータを取り戻さなければならず、これには数ミリ秒かかることがあります。そのため、数ミリ秒かかることがあります。この遅延はしばしば目につきます。長時間アイドル状態のウィンドウを放置した後、再びウィンドウに戻ると、ウィンドウの起動が遅くなり、ページがスワップインされる間、ディスクドライブが動作する音が聞こえるかもしれません。

たまにはそのような遅延があってもいいかもしれませんが、あまりにも多くのプロセスが多くのスペースを使用していると、お互いに干渉し始めます。プロセスAが実行されると、プロセスBが必要とするページを退避させます。そして、Bが実行されると、Aが必要とするページを退避させます。このような状況になると、両方のプロセスの処理速度が低下し、システムが反応しなくなることがあります。このような状況を「スラッシング」と呼びます。
理論的には、オペレーティングシステムは、ページングの増加を検出し、システムが再び反応するまでプロセスをブロックまたはキルすることでスラッシングを回避することができます。しかし、私の知る限り、ほとんどのシステムはこの機能を備えておらず、また、うまく機能していません。そのため、物理メモリの使用を制限するか、スラッシングが発生したときに回復しようとするかは、ユーザーに委ねられることが多いのです。

Chapter 8

Multitasking

現在の多くのシステムでは、CPUに複数のコアが搭載されており、複数のプロセスを同時に実行することができます。さらに、各コアには「マルチタスク」機能があり、あるプロセスから別のプロセスに素早く切り替えることで、多くのプロセスが同時に実行されているように見せかけることができます。

オペレーティングシステムの中でマルチタスクを実現する部分が「カーネル」です。木の実や種の場合、カーネルは殻に囲まれた最も内側の部分です。オペレーティングシステムでは、カーネルは最下層のソフトウェアであり、その周りを "シェル "と呼ばれるインターフェースを含むいくつかの層が取り囲んでいます。コンピュータ科学者は、拡張されたメタファーが大好きです。

カーネルの最も基本的な仕事は、割り込みの処理です。割り込み」とは、通常の命令サイクルを停止し、実行の流れを「割り込みハンドラ」と呼ばれるコードの特別なセクションにジャンプさせるイベントのことです。

ハードウェア割り込みは、あるデバイスがCPUに信号を送ることで発生します。例えば、ネットワークインターフェースでは、データのパケットが到着すると割り込みが発生し、ディスクドライブでは、データの転送が完了すると割り込みが発生します。また、多くのシステムでは、一定の間隔や時間が経過したときに割り込みを起こすタイマーを備えています。

ソフトウェア割り込みは、実行中のプログラムによって引き起こされます。

例えば、ある命令が何らかの理由で完了できない場合、割り込みが発生し、その状態をOSが処理できるようになります。ゼロ除算などの浮動小数点エラーは、割り込みを使って処理されます。
プログラムがハードウェアデバイスにアクセスする必要がある場合、プログラムはシステムコールを行います。システムコールは関数呼び出しに似ていますが、関数の先頭にジャンプするのではなく、システムコールのトリガーとなる特別な命令を実行する点が異なります。

割り込みが発生し、実行の流れがカーネルにジャンプします。カーネルはシステムコールのパラメータを読み取り、要求された処理を実行した後、中断していた処理を再開します。

8.1 Hardware state

割り込みを処理するには、ハードウェアとソフトウェアの協力が必要です。割り込みが発生するときには、CPU上で複数の命令が実行されていたり、レジスタにデータが格納されていたりと、ハードウェアの状態が変化しています。

通常、ハードウェアはCPUを一貫した状態にする責任があります。例えば、すべての命令は完了するか、または開始されなかったかのように動作しなければなりません。例えば、すべての命令は完了するか、あるいは開始されなかったかのように動作するかのいずれかでなければなりません。また、ハードウェアはプログラムカウンタ(PC)を保存し、カーネルがどこで再開するかを知る責任があります。

そして、通常、割り込みハンドラの責任として、ハードウェアの残りの状態を変更する可能性があることを行う前に保存し、中断されたプロセスが再開する前に保存された状態を復元します。
この一連の流れの概要をご紹介します。

1. When the interrupt occurs, the hardware saves the program counter in a special register and jumps to the appropriate interrupt handler.
2. The interrupt handler stores the program counter and the status register in memory, along with the contents of any data registers it plans to use.
3. The interrupt handler runs whatever code is needed to handle the interrupt.
4. Then it restores the contents of the saved registers. Finally, it restores the program counter of the interrupted process, which has the effect of jumping back to the interrupted instruction.

この仕組みが正しく機能していれば、命令間の時間変化を検知しない限り、割り込みがあったことを割り込みプロセスが知る方法は一般的にありません。

8.2 Context switching

割り込みハンドラは、ハードウェア全体の状態を保存する必要がなく、使用する予定のレジスタだけを保存すればよいので、高速に処理できます。

しかし、割り込みが発生した場合、カーネルは必ずしも割り込みを受けたプロセスを再開するとは限りません。別のプロセスに切り替えるという選択肢があるのです。この仕組みを「コンテキストスイッチ」と呼びます。

一般的に、カーネルはプロセスがどのレジスタを使用するかわからないので、すべてのレジスタを保存しなければなりません。また、新しいプロセスに切り替わるときには、メモリ管理ユニットに保存されているデータをクリアしなければならないこともあります（3.6節参照）。また、コンテキストスイッチの後、新しいプロセスがキャッシュにデータをロードするのに時間がかかることがあります。このような理由から、コンテキストスイッチは数千サイクル、あるいは数マイクロ秒という比較的遅い時間で行われます。
マルチタスクシステムでは、各プロセスは「タイムスライス」または「量子」と呼ばれる短い期間の実行が許可されています。コンテキストスイッチの際、カーネルはハードウェアタイマーを設定し、タイムスライスの終了時に割り込みを発生させます。割り込みが発生すると、カーネルは別のプロセスに切り替えたり、中断していたプロセスの再開を許可したりします。この決定を行うオペレーティングシステムの部分が「スケジューラ」です。

8.3 The process life cycle

プロセスが作成されると、オペレーティングシステムは、「プロセスコントロールブロック」またはPCBと呼ばれる、プロセスに関する情報を含むデータ構造を割り当てます。とりわけ、PCBはプロセスの状態を追跡し、それは次のいずれかです。

- Running, if the process is currently running on a core.
- Ready, if the process could be running, but isn't, usually because there are more runnable processes than cores.
- Blocked, if the process cannot run because it is waiting for a future event like network communication or a disk read.
- Done, if the process has completed, but has exit status information that has not been read yet.

ここでは、プロセスがある状態から別の状態へと移行する原因となるイベントを紹介します。

- A process is created when the running program executes a system call like fork. At the end of the system call, the new process is usually ready. Then the scheduler might resume the original process (the “parent”) or start the new process (the “child”).

- When a process is started or resumed by the scheduler, its state changes from ready to running.
- When a process is interrupted and the scheduler chooses not to let it resume, its state changes from running to ready.
- If a process executes a system call that cannot complete immediately, like a disk request, it becomes blocked and the scheduler usually chooses another process.
- When an operation like a disk request completes, it causes an interrupt. The interrupt handler figures out which process was waiting for the request and switches its state from blocked to ready. Then the scheduler may or may not choose to resume the unblocked process.
- When a process calls exit, the interrupt handler stores the exit code in the PCB and changes the process's state to done.

8.4 Scheduling

セクション2.3で見たように、コンピュータには何百ものプロセスがあるかもしれませんが、通常はそのほとんどがブロックされています。ほとんどの場合、準備ができているプロセスや実行中のプロセスは数個しかありません。割り込みが発生すると、スケジューラはどのプロセスを開始または再開するかを決定します。

ワークステーションやラップトップでは、スケジューラの第一の目標は応答時間を最小にすることです。つまり、ユーザのアクションに対してコンピュータが迅速に応答する必要があります。サーバーでも応答時間は重要ですが、それに加えてスケジューラーは、単位時間あたりに完了するリクエストの数であるスループットを最大化しようとするかもしれません。
通常、スケジューラーはプロセスが何をしているかについてあまり情報を持っていないので、いくつかのヒューリスティックな方法に基づいて決定します。

- Processes might be limited by different resources. A process that does a lot of computation is probably CPU-bound, which means that its run time depends on how much CPU time it gets. A process that reads data from a network or disk might be I/O-bound, which means that it would run faster if data input and output went faster, but would not run faster with more CPU time. Finally, a process that interacts with the user is probably blocked, most of the time, waiting for user actions.

オペレーティングシステムは、過去の動作に基づいてプロセスを分類し、それに応じてスケジュールを組むことがあります。例えば、あるプロセスが

- 対話型プロセスがブロックされていない場合、ユーザーはおそらく返信を待っているので、すぐに実行すべきでしょう。一方、CPUに拘束されているプロセスで、長時間実行されているものは、時間的な制約が少ないかもしれません。

- If a process is likely to run for a short time and then make a blocking request, it should probably run immediately, for two reasons: (1) if the request takes some time to complete, we should start it as soon as possible, and (2) it is better for a long-running process to wait for a short one, rather than the other way around.

  例えとして、アップルパイを作るとします。クラストの準備には5分かかりますが、その後30分ほど冷やさなければなりません。一方、フィリングの準備には20分かかります。先にクラストを準備しておけば、クラストを冷やしている間にフィリングを準備することができ、35分でパイを完成させることができます。先にフィリングを用意した場合は55分かかります。

ほとんどのスケジューラは、何らかの形で優先順位ベースのスケジューリングを採用しており、各プロセスは時間の経過とともに上下に調整可能な優先順位を持っています。スケジューラは実行時に、最も高い優先度を持つ実行可能なプロセスを選択します。
ここでは、プロセスの優先順位を決定する要素をご紹介します。

- A process usually starts with a relatively high priority so it starts running quickly.
- If a process makes a request and blocks before its time slice is complete, it is more likely to be interactive or I/O-bound, so its priority should go up.
- If a process runs for an entire time slice, it is more likely to be long-running and CPU-bound, so its priority should go down.
- If a task blocks for a long time and then becomes ready, it should get a priority boost so it can respond to whatever it was waiting for.
- If process A is blocked waiting for process B, for example if they are connected by a pipe, the priority of process B should go up.
- The system call nice allows a process to decrease (but not increase) its own priority, allowing programmers to pass explicit information to the scheduler.

通常のワークロードを実行しているほとんどのシステムでは、スケジューリングアルゴリズムがパフォーマンスに大きな影響を与えることはありません。シンプルなスケジューリングポリシーで十分なのです。

8.5 Real-time scheduling

しかし、現実の世界と対話するプログラムでは、スケジューリングが非常に重要になります。例えば、センサーからのデータを読み取ってモーターを制御するプログラムでは、繰り返し行われるタスクをある最小の時間で完了させ、外部のイベントにはある最大の応答時間で対応しなければなりません。このような要求は、多くの場合、「期限」までに完了しなければならない「タスク」という言葉で表現されます。
締め切りに間に合うようにタスクをスケジューリングすることを「リアルタイムスケジューリング」といいます。アプリケーションによっては、Linuxなどの汎用OSを改造して、リアルタイムスケジューリングに対応させることができます。このような改造には次のようなものがあります。

- Providing richer APIs for controlling task priorities.
- Modifying the scheduler to guarantee that the process with highest priority runs within a fixed amount of time.
- Reorganizing interrupt handlers to guarantee a maximum completion time.
- Modifying locks and other synchronization mechanisms (coming up in the next chapter) to allow a high-priority task to preempt a lower-priority task.
- Choosing an implementation of dynamic memory allocation that guarantees a maximum completion time.

より要求の高いアプリケーション、特にリアルタイムでの応答が生死に関わるような領域では、「リアルタイムOS」が特殊な機能を提供し、多くの場合、汎用OSよりもはるかにシンプルな設計になっています。

Chapter 9

Threads

2.3節でスレッドについて触れたとき、私は「スレッドはプロセスの一種である」と言いました。ここでは、もう少し丁寧に説明します。

プロセスを作成すると、オペレーティングシステムは、テキストセグメント、スタティックセグメント、ヒープを含む新しいアドレス空間を作成します。また、プログラムカウンタやその他のハードウェアの状態、コールスタックを含む新しい「実行スレッド」を作成します。

これまで見てきたプロセスは、各アドレス空間で1つのスレッドのみが実行される「シングルスレッド」でした。本章では、同じアドレス空間で複数のスレッドが実行される「マルチスレッド」のプロセスについて学びます。

1つのプロセスの中では、すべてのスレッドが同じテキストセグメントを共有しているため、同じコードを実行します。しかし、異なるスレッドがコードの異なる部分を実行することがよくあります。

また、同じスタティックセグメントを共有しているので、あるスレッドがグローバル変数を変更すると、他のスレッドもその変更を見ることができます。また、ヒープも共有しているので、動的に割り当てられたチャンクをスレッドで共有することができます。

しかし、各スレッドは独自のスタックを持っているので、スレッド同士が干渉することなく関数を呼び出すことができます。通常、スレッドはお互いのローカル変数にアクセスしません（アクセスできない場合もありま

す）。

この章のサンプルコードは、本書のリポジトリにある counter という名前のディレクトリにあります。このコードをダウンロードする方法については，セクション0.1を参照してください。

9.1 Creating threads

C 言語で使用される最も一般的なスレッド規格は POSIX Threads（略して Pthreads）です。POSIX 規格では、スレッドモデルと、スレッドを作成および制御するためのインターフェイスが定義されています。UNIXのほとんどのバージョンがPthreadの実装を提供しています。
Pthread を使用することは、ほとんどの C ライブラリを使用することと同じです。

- You include headers files at the beginning of your program.
- You write code that calls functions defined by Pthreads.
- When you compile the program, you link it with the Pthread library.

私の例では、以下のようなヘッダーを入れています。

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
となります。

最初の 2 つは標準的なもので、3 つ目は Pthread 用、4 つ目はセマフォ用のものです。gcc で Pthread ライブラリを使用してコンパイルするには、コマンドラインで -l オプションを使用します。

gcc -g -O2 -o array array.c -lpthread
array.c という名前のソースファイルにデバッグ情報とオプティマイゼーションを加えてコンパイルし、Pthread ライブラリとリンクし、array という名前の実行ファイルを生成します。

9.2 Creating threads

スレッドを作成する Pthread 関数は pthread_create と呼ばれています。以下の関数では、その使い方を紹介しています。

```
pthread_t make_thread(void *(*entry)(void *), Shared *shared)
{
    int n;
    pthread_t スレッド。
```

n = pthread_create(&thread, NULL, entry, (void
*)shared); if (n != 0) {.
perror("pthread_create failed")が発生しました。

exit(-1)です。

}

スレッドを返す。

}

make_thread は、pthread_create を使いやすくするために書いたラッパーで、エラーチェックを行います。

pthread_create の戻り値の型は pthread_t で、これは新しいスレッドのID または「ハンドル」と考えることができます。

pthread create が成功した場合は 0 を返し、make_thread は新しいスレッドのハンドルを返す。エラーが発生した場合は、pthread create がエラーコードを返し、make_thread がエラーメッセージを表示して終了する。

make_threadのパラメータについては、少し説明が必要です。まず最初に、Shared はスレッド間で共有される値を格納するために定義した構造体です。以下の typedef ステートメントで新しい型を作成します。

```
typedef
struct { int
counter;
} Shared;
```

make shared は、Shared 構造体の領域を確保し、その内容を初期化します。

```
Shared *make_shared()
{
    Shared *shared = check_malloc(sizeof (Shared));
    shared->counter = 0;
    return shared;
}
```

さて、共有データ構造ができたところで、make_thread に戻りましょう。最初のパラメータは、void ポインタを受け取り、void ポインタを返す関数へのポインタです。この型を宣言する構文で目が血走ってしまうのは、あなただけではありません。いずれにしても、このパラメータの目的は、新しいスレッドの実行を開始する関数を指定することです。慣習上、この関数はentryと名付けられています。

```
void *entry(void *arg)
{
```

```
    Shared *shared = (Shared *) arg;
    child_code(shared);
    pthread_exit(NULL);
}
```

エントリのパラメータはvoidポインタとして宣言しなければなりませんが、このプログラムでは、実際にはShared構造体へのポインタであることがわかっているので、それに合わせて型キャストし、実際の作業を行う子コードに渡すことができるのです。

簡単な例として、child_codeは、共有カウンタの値を印刷し、それをインクリメントします。

```
void child_code(Shared *shared)
{
    printf("counter = %d_n", shared->counter);
    shared->counter++;
}
```

子コードが戻ると、entryはpthread_exitを呼び出し、このスレッドに加わるスレッドに値を渡すために使用することができます。この場合、子は何も言うことがないので、NULLを渡します。

最後に、子スレッドを作成するコードを紹介します。

```
    int i;
```

pthread_t child[NUM_CHILDREN]です。

```
    Shared *shared = make_shared(1000000);
    for (i=0; i<NUM_CHILDREN; i++) {。
        child[i] = make_thread(entry, shared);
    }
```

NUM_CHILDRENは、子スレッドの数を決定するコンパイル時の定数です。childはスレッドハンドルの配列です。

9.3 Joining threads

あるスレッドが他のスレッドの完了を待ちたいとき、スレッドは

pthread join. 以下は、私が作成したpthread joinのラッパーです： void join_thread(pthread_t thread)

```
{
    int ret = pthread_join(thread,
    NULL); if (ret == -1) {...
      perror("pthread_join
      failed"); exit(-1);
    }
}
```

パラメータには、待機させたいスレッドのハンドルを指定します。ラッパーが行うのは、pthread join を呼び出して結果を確認するだけです。

どのスレッドも他のスレッドに参加することができますが、最も一般的なパターンでは、親スレッドがすべての子スレッドを作成して参加します。前節の例の続きで、子を待つコードを紹介します。

```
for (i=0; i<NUM_CHILDREN; i++) {。
    join_thread(child[i]);
}
```

このループは、作成された順に1つずつ子を待ちます。子のスレッドがその順番通りに完了する保証はありませんが、このループはそうでなくても正しく動作します。子機の1つが遅れた場合、ループは待たなければならず、その間に他の子機が完了するかもしれません。しかし、それに関わらず、このループはすべての子が完了したときにのみ終了します。

本書のリポジトリをダウンロードした場合(セクション0.1を参照)、counter/counter.cにこの例があります。この例は、次のようにコンパイルして実行できます。

```
$ make counter
gcc -Wall counter.c -o counter -lpthread

$ .com/counter
```

5人の子供で実行したところ、以下のような出力が得られました。

```
counter = 0
counter = 0
counter = 1
counter = 0
counter = 3
```

走らせてみると、きっと違う結果になるでしょう。そして、もう一度実行すれば、毎回違う結果になるかもしれません。何が起こっているのか？

9.4 Synchronization errors

先ほどのプログラムの問題点は、子供たちが同期なしに共有変数counterにアクセスするため、複数のスレッドがcounterの同じ値を読み取っても、どのスレッドもcounterをインクリメントする前に読み取ってしまうこと

です。

ここでは、前節の出力を説明することができる一連の出来事を紹介します。

A子さんは0を読む

B子さんは0を読む

```
Child C reads 0
Child  A  prints     0
Child  B  prints     0
```

子Aはカウンタ=1を設

定 子Dは1を読み出す

```
Child  D  prints     1
Child  C  prints     0
Child A sets counter=1
Child B sets counter=2
Child C sets counter=3
Child E reads 3
Child  E  prints     3
Child D sets counter=4
Child E sets  counter=5
```

プログラムを実行するたびに、スレッドが異なるポイントで中断されたり、スケジューラーが異なるスレッドを選択して実行したりするので、イベントの順序や結果は異なります。

何らかの秩序を持たせたいとします。例えば、各スレッドが異なるcounterの値を読み込んでそれを増加させ、counterの値がchild_codeを実行したスレッドの数を反映するようにしたいとします。

これは、コードブロックの「相互排除」を保証するオブジェクトで、一度に1つのスレッドしかブロックを実行できません。

私は、ミューテックス・オブジェクトを提供する mutex.c という小さなモジュールを書きました。まず、使い方を説明し、その後、どのように機能するかを説明します。

以下は、スレッドの同期にミューテックスを使用したバージョンのchild_codeです。

```
void child_code(Shared *shared)
{
    mutex_lock(shared->mutex)です。
    printf("counter = %d_n", shared->counter);
    shared->counter++;
    mutex_unlock(shared->mutex)。
}
```

どのスレッドもカウンタにアクセスする前に、ミューテックスを「ロック」しなければなりませんが、これは他のすべてのスレッドを禁止する効果が

あります。スレッド A がミューテックスをロックし、child_code の途中にいるとします。もしスレッド B が到着して mutex_lock を実行すると、ブロックされます。
スレッドAが終了すると、mutex_unlockを実行し、スレッドBが続行できるようになります。事実上、スレッドは子コードを一度に一つずつ実行するために並んでいます。

互いに干渉しないようにしています。このコードを5人の子供で実行すると、次のようになります。

```
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
```

これで要件を満たすことができました。このソリューションが機能するためには、Shared構造体にMutexを追加する必要があります。

```
typedef
struct { int
counter;
Mutex
*mutex;
} Shared;
```

そして、make_shared Shared *make_shared(int end)で初期化します。

```
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->counter = 0;
    shared->mutex = make_mutex();   //-- this line is new
    return shared;
}
```

このセクションのコードは counter_mutex.c にあります。Mutex の定義は次のとおりです次のセクションで説明するmutex.cです。

9.5 Mutex

私が定義したMutexは、POSIXスレッドAPIで定義されているpthread_mutex_tという型のラッパーです。
POSIX ミューテックスを作成するには、pthread_mutex_t のための領域を割り当てる必要があります。

タイプを選択し、pthread_mutex_initを呼び出します。

この API の問題点の 1 つは、pthread_mutex_t が構造体のように動作するため、これを引数として渡すとコピーが作成され、ミュー テックスの

動作がおかしくなることです。これを避けるためには、pthread_mutex_t をアドレスで渡さなければなりません。
私のコードでは、それを簡単に実現することができます。これは単に pthread_mutex_t という読みやすい名前で、Mutex という型を定義しています。

```
#include <pthread.h>

typedef pthread_mutex_t Mutex;
```

そして、make_mutexを定義します。make_mutexはスペースを確保し、ミューテックスを初期化します。

```
ミューテックス *make_mutex()
{
    Mutex *mutex =
    check_malloc(sizeof(Mutex)); int n =
    pthread_mutex_init(mutex, NULL);
    if (n != 0)
    perror_exit("make_lock
    failed"); return mutex;
}
```

戻り値はポインターで、不要なコピーを引き起こすことなく、引数として渡すことができます。

ミューテックスをロックしたりアンロックしたりする関数は、POSIX関数のシンプルなラッパーです。

```
void mutex_lock(Mutex *mutex)
{
    int n = pthread_mutex_lock(mutex);
    if (n != 0) perror_exit("lock failed") となります。
}

void mutex_unlock(Mutex *mutex)
{
    int n = pthread_mutex_unlock(mutex);
    if (n != 0) perror_exit("unlocking failed") となります
}
```

このコードは、mutex.cとヘッダーファイルのmutex.hにあります。

Chapter 10

Condition variables

前章で示したように、多くの単純な同期問題はミューテックスを使って解決することができます。本章では、より大きな課題である、よく知られた「Producer-Consumer問題」と、それを解決するための新しいツールである「条件変数」を紹介します。

10.1 The work queue

マルチスレッドのプログラムでは、異なるタスクを実行するためにスレッドが編成されることがあります。多くの場合、これらのスレッドはキューを使って相互に通信します。「プロデューサー」と呼ばれるいくつかのスレッドがデータをキューに入れ、「コンシューマー」と呼ばれる他のスレッドがデータを取り出します。

たとえば、グラフィカル・ユーザー・インターフェースを備えたアプリケーションでは、GUIを実行してユーザーのイベントに応答するスレッドと、ユーザーのリクエストを処理するスレッドがあるかもしれません。このような場合、GUIスレッドがリクエストをキューに入れ、「バックエンド」スレッドがリクエストを取り出して処理することがあります。

この組織をサポートするためには、「スレッドセーフ」なキューの実装が必要です。これは、両方のスレッド（または2つ以上のスレッド）が同時にキューにアクセスできることを意味します。また、キューが空の場合や、キューのサイズが制限されている場合には、キューが満杯になった場合などの特殊なケースを処理する必要があります。
まず、スレッドセーフではないシンプルなキューから始めて、何が問

題なのかを見て、それを修正していきます。この例のコードは、この本のリポジトリの queue というフォルダにあります。ファイル queue.c には、サーキュラーバッファーの基本的な実装が含まれています。サーキュラーバッファーについては、https://en.wikipedia.org/wiki/Circular_buffer をご覧ください。

ここでは、構造の定義を説明します。

```
typedef
struct { int
*array; int
length; int
next_in; int
next_out;
} Queue;
```

array は、キューの要素を格納する配列です。この例では、要素は整数ですが、より一般的には、ユーザーイベントや作業項目などを含む構造体になります。

next_in は，次の要素を追加すべき場所を示す配列のインデックスで，同様に next_out は，次に削除すべき要素のインデックスである．

make_queueは、この構造体のスペースを確保し、フィールドを初期化します。

```
キュー *make_queue(int length)
{
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length + 1;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    return queue;
}
```

next_outの初期値については、少し説明が必要です。最初はキューが空なので、削除すべき次の要素がなく、next_out は無効です。next_out == next_in を設定することは、キューが空であることを示す特殊なケースなので、書くことができます。

```
int queue_empty(Queue *queue)
{
    return (queue->next_in == queue->next_out);
}
void queue_push(Queue *queue, int item) {.
        if
        (queue_full(queue)
        )
        { perror_exit("queu
```

```
    e is full");
}
```

queue->array[queue->next_in] = item;

```
    tail->next_in = tail_incr(tail, tail->next_in);
}
```

If the queue is full, queue_push prints an error message and exits. I will explain queue_full soon.

キューが満杯でない場合、queue_pushは新しい要素を挿入し、queue_incrを用いてnext_inをインクリメントします。

```
int queue_incr(Queue *queue, int i)
{
    return (i+1) % queue->length;
}
```

添字のiが配列の最後に到達すると、0に回り込みます。ここで、厄介なことが起こります。キューに要素を追加し続けると、最終的にはnext_inが折り返してnext_outに追いつきます。しかし、もしnext_in == next_outであれば、キューは空であると誤って結論づけてしまいます。

それを避けるために、キューが満杯であることを示す別の特別なケースを定義します。

```
int queue_full(Queue *tail)
{
    return (queue_incr(queue, queue->next_in) == queue->next_out);
}
```

next_inをインクリメントしてnext_outに着地した場合、キューを空っぽにすることなく別の要素を追加できないことを意味します。そこで、「終わり」の1つ前で要素を停止します（キューの終わりはどこでもよく、配列の終わりとは限らないことに留意してください）。

これで、キューから次の要素を削除して返す queue_pop が書けるようになりました。

```
int queue_pop(Queue
*queue) { if
(queue_empty(queue)) { 。
    perror_exit("queue is empty");
}

int item = queue->array[queue->next_out];
queue->next_out = queue_incr(queue, queue->next_out);
return item;
}
```

空のキューからポップしようとした場合、queue_pop はエラーメッセー

ジを表示して終了します。

10.2 Producers and consumers

では、このキューにアクセスするスレッドをいくつか作ってみましょう。これがプロデューサー・コードです。

```
void *producer_entry(void *arg)
{ Shared *shared = (Shared *)
arg;

    for (int i=0;
    i<QUEUE_LENGTH-1; i++)
    { printf("adding item %d¥n",
    i); queue_push(shared-
    >queue, i);
}
pthread_exit(NULL)です。
}
```

これが消費者コードです。

```
void *consumer_entry(void
*arg) { int item;
Shared *shared = (Shared *) arg;

    for (int i=0;
    i<QUEUE_LENGTH-1; i++)
    { item = queue_pop(shared-
    >queue); printf("consuming
    item %d¥n", item);
}
pthread_exit(NULL)です。
}
```

以下は、スレッドを開始して待機する親コードです。

```
pthread_t child[NUM_CHILDREN];
Shared *shared = make_shared();
child[0] = make_thread(producer_entry, shared);
child[1] = make_thread(consumer_entry,
shared);

    for (int i=0;
    i<NUM_CHILDREN; i++)
    { join_thread(child[i]);
}
```

そして最後に、キューを格納する共有構造です。

```
        typedef struct
        { Queue *queue;
    } Shared;

    Shared *make_shared()
```

```
{
    Shared *shared = check_malloc(sizeof(Shared));
    shared->queue = make_queue(QUEUE_LENGTH);
    return shared;
}
```

- これまでのコードは良い出発点ではありますが、いくつかの問題があります。
  - Access to the queue is not thread safe. Different threads could access array, next_in, and next_out at the same time and leave the queue in a broken, “inconsistent” state.
  - If the consumer is scheduled first, it finds the queue empty, print an error message, and exits. We would rather have the consumer block until the queue is not empty. Similarly, we would like the producer to block if the queue is full.

次のセクションでは、ミューテックスを使って最初の問題を解決します。

次のセクションでは、条件変数を使って2番目の問題を解決します。

10.3 Mutual exclusion

ミューテックスを使ってキューのスレッドセーフを実現します。このバージョンのコードは queue_mutex.c にあります。

まず、キューの構造体にMutexのポインタを追加します。

```
typedef
struct { int
*array; int
length; int
next_in; int
next_out;
Mutex *mutex;              //-- this line is new
} Queue;
```

そして、make_queueでMutexを初期化します。

```
Queue *make_queue(int length) {.
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_mutex();    //-- new
    return queue;
}
```

次に、queue_pushに同期のコードを追加します。

```
void queue_push(Queue *queue, int item)
    { mutex_lock(queue->mutex); //-- new
    if (queue_full(queue)) {
      mutex_unlock(queue->mutex);     //-- new
      perror_exit("queue is full");
    }

    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
    mutex_unlock(queue->mutex);     //-- new
}
```

キューがいっぱいになったかどうかをチェックする前に、Mutexをロックしなければなりません。そうしないと、スレッドはMutexをロックしたままになり、他のスレッドが先に進めなくなってしまいます。

queue_popの同期コードも同様です。

```
int        queue_pop(Queue
*queue)
{          mutex_lock(queue-
>mutex);                    if
(queue_empty(queue)) {  。
  mutex_unlock(queue->mutex);
  perror_exit("queue is empty");
}

int item = queue->array[queue->next_out];
queue->next_out = queue_incr(queue, queue->next_out);
mutex_unlock(queue->mutex);
リターンアイテム
}
```

他のQueue関数、queue_full、queue_empty、およびqueue_incrは、ミューテックスをロックしようとはしないことに注意してください。これらの関数を呼び出すスレッドは、最初にミューテックスをロックすることが要求されます。この要求は、これらの関数のドキュメント化されたインターフェースの一部です。

この追加コードにより、キューはスレッドセーフとなり、実行しても同期エラーは発生しないはずです。しかし、どこかの時点で、キューが空になったためにコンシューマが終了するか、キューが満杯になったためにプロデューサが終了するか、あるいはその両方が発生する可能性があります。

次のステップは、条件変数の追加です。

10.4 Condition variables

条件変数は、条件に関連付けられたデータ構造で、条件が真になるまでスレッドをブロックすることができます。例えば、thread_pop は、キューが空であるかどうかをチェックし、空であれば、「queue not empty」のような条件を待つことができます。

同様に、thread_push は、キューが一杯になっているかどうかをチェックし、一杯になっている場合は、一杯でなくなるまでブロックしたいと思うかもしれません。

ここでは1つ目の条件を扱いますが、2つ目の条件については練習として扱う機会があります。

まず、Queue構造体に条件変数を追加します。

```
typedef
struct { int
*array; int
length; int
next_in; int
next_out;
Mutex
*mutex;
Cond *nonempty;    //-- new
} Queue;
```

そして、make_queueで初期化します。

```
キュー *make_queue(int length)
{
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_mutex();
    queue->nonempty = make_cond();    //-- new
    return queue;
}
```

さて、queue_popでは、キューが空になっても終了せずに、条件変数を使ってブロックしています。

```
int queue_pop(Queue
*queue) { mutex_lock(queue-
>mutex); while
```

```
(queue_empty(queue)) { 。
   cond_wait(queue->nonempty,  queue->mutex);    //-- new
}
```

```
  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);
  mutex_unlock(queue->mutex);
  cond_signal(queue->nonfull);     //-- new
  return item;
}
```

cond_waitは複雑なので、ゆっくり説明しましょう。最初の引数は条件変数で、ここでは「キューが空でない」という条件を待ちます。第2引数は、キューを保護するミューテックスです。

ミューテックスをロックしたスレッドが cond_wait を呼び出すと、ミューテックスのロックを解除してからブロックします。これは重要なことです。もし cond_wait がブロックする前に mutex のロックを解除しなかった場合、他のスレッドはキューにアクセスすることができず、アイテムを追加することもできず、キューは常に空になってしまいます。

そのため、コンシューマーが nonempty でブロックされている間、プロデューサーは実行することができます。では、producerがqueue_pushを実行するとどうなるか見てみましょう。

```
void queue_push(Queue *queue, int item)
    { mutex_lock(queue->mutex);
        if
        (queue_full(queue
        ))
        { mutex_unlock(qu
        eue->mutex);
        perror_exit("queue
        is full");
    }
    queue->array[queue->next_in] = item;
    queue->next_in = queue_incr(queue, queue->next_in);
    mutex_unlock(queue->mutex);
    cond_signal(queue->nonempty);     //-- new
}
```

先ほどと同様に、queue_pushはMutexをロックし、キューが満杯であるかどうかをチェックします。満杯ではないと仮定して、queue_pushは新しい要素をキューに追加し、その後Mutexのロックを解除します。
しかし、戻る前にもう一つ、条件変数の「シグナル」を出します。

non-emptyです。

条件変数にシグナルを送ることは、通常、その条件が真であることを示します。条件変数を待っているスレッドがない場合、シグナルは何の効果もありません。

条件変数を待っているスレッドがあれば、そのうちの1つがブロックされずにcond_waitの実行を再開します。しかし、目覚めたスレッドがcond_waitから戻る前に、再びMutexを待ってロックしなければなりません。
さて、queue_pop に戻って、スレッドが cond_wait から戻ったときに何が起こるかを見てみましょう。whileループの先頭に戻ってループして

の条件を再度確認します。理由はすぐに説明しますが、ここでは、条件が真である、つまり、キューが空ではないと仮定しましょう。

consumer スレッドが while ループを終了したとき、私たちは 2 つのことを知っています。(1) 条件が真なので、キューに少なくとも1つのアイテムがあること、(2) Mutexがロックされているので、キューにアクセスしても安全であること。

アイテムを削除した後、queue_pop は mutex のロックを解除して戻ります。

次のセクションでは、私のCondコードがどのように動作するかを紹介しますが、その前に、よくある2つの質問に答えたいと思います。

- Why is cond_wait inside a while loop rather than an if statement; that is, why do we have to check the condition again after returning from cond_wait?
  条件を再確認しなければならない第一の理由は、信号が傍受される可能性があるからです。例えば、スレッドAがnonemptyで待機しているとします。スレッドBがキューにアイテムを追加し、nonemptyのシグナルを出しました。スレッドAは目を覚ましてミューテックスをロックしようとしますが、その機会を得る前に、邪悪なスレッドCが急襲してミューテックスをロックし、キューからアイテムをポップして、ミューテックスをアンロックします。これでキューは再び空になりましたが、スレッドAはそれ以上ブロックされていません。スレッドAはミューテックスをロックしてcond_waitから戻ることができました。スレッドAが条件を再度確認しないと、空のキューから要素をポップしようとし、おそらくエラーが発生します。

- The other question that comes up when people learn about condition variables is “How does the condition variable know what condition it is associated with?”

  この質問が理解できるのは、Condの構造とそれが関連する条件との間には明確な関連性がないからです。使われ方によっては暗黙の了解となっています。

  Condに関連する条件とは、cond_waitを呼び出したときにfalse、cond_signalを呼び出したときにtrueになるものです。

スレッドは cond_wait から戻るときに条件をチェックしなければならないので、条件が真のときだけ cond_signal を呼び出すことは厳密には必要ではない。もし、条件が真であるかもしれないと思う理由があれば、今がチェックするのに良い時期であることを示唆するためにcond_signalを呼ぶことができる。

10.5 Condition variable implementation

前のセクションで使用した Cond 構造体は、POSIX スレッド API で定義されている pthread_cond_t という型のラッパーです。これは、pthread_mutex_t のラッパーである Mutex と非常によく似ています。どちらのラッパーもutils.cとutils.hで定義されています。

これがその型定義です。

typedef pthread_cond_t Cond;

make_condは，空間を確保し，条件変数を初期化して，ポインタを返します。

```
Cond *make_cond() {
    Cond *cond = check_malloc(sizeof(Cond));
    int n = pthread_cond_init(cond, NULL);
    if (n != 0) perror_exit("make_cond failed")となります

    condを返す。
}
```

そして、cond_waitとcond_signalのラッパーです。 void cond_wait(Cond *cond, Mutex *mutex) {。

```
    int n = pthread_cond_wait(cond, mutex);
    if (n != 0) perror_exit("cond_wait failed")となります
}

void cond_signal(Cond *cond) {...
    int n = pthread_cond_signal(cond);
    if (n != 0) perror_exit("cond_signal failed")となりま
    す。
}
```

この時点では、何も驚くことはないでしょう。

Chapter 11

Semaphores inC

セマフォは同期について学ぶのに適していますが、ミューテックスや条件変数のように実際に広く使われているわけではありません。

しかし、同期問題の中には、セマフォを使って簡単に解決でき、より実証的に正しい解決策を得られるものもあります。

この章では、セマフォを扱うためのC言語のAPIと、それを簡単に扱えるようにするための私のコードを紹介します。また、最後の課題として、ミューテックスと条件変数を使ったセマフォの実装を書くことができるかどうかを紹介します。
この章のコードは、本書のリポジトリのディレクトリ・セマフォにあります（0.1節参照）。

11.1 POSIX Semaphores

セマフォは、スレッドがお互いに干渉せずに協調して動作するためのデータ構造です。

POSIX 標準ではセマフォのインタフェースが規定されており、これは Pthread には含まれていないが、Pthread を実装しているほとんどの UNIX はセマフォも提供している。
POSIXのセマフォはsem tという型を持っています。いつものように、sem tにラッパーをつけます。

を使用しています。インターフェイスはsem.hで定義されています： typedef sem_t Semaphore;
Semaphore *make_semaphore(int value);
void semaphore_wait(Semapore *sem);
void semaphore_signal(Semapore *sem);

Semaphoreはsem_tの同義語ですが、私はこちらの方が読みやすいと思いますし、大文字はオブジェクトのように扱ってポインタで渡すことを思い出させてくれます。

これらの関数の実装は、sem.cにあります。セマ

フォ *make_semaphore(int value)

```
{
    Semaphore *sem = check_malloc(sizeof(Semaphore));
    int n = sem_init(sem, 0, value);
    if (n != 0)
    perror_exit("sem_init failed");
    return sem;
}
```

make semaphore は、パラメータとしてセマフォの初期値を受け取ります。セマフォ用のスペースを確保し、初期化し、セマフォへのポインタを返します。

sem initは、成功すると0を、何か問題があると-1を返します。ラッパー関数を使うことの良い点は、エラーチェックのコードをカプセル化できるので、これらの関数を使うコードがより読みやすくなることです。

以下は，semaphore_waitの実装です： void

semaphore_wait(Semaphore *sem)

```
{
    int n = sem_wait(sem);
    if (n != 0) perror_exit("sem_wait failed")となります
}
```

そして、semaphore_signalです。

void semaphore_signal(Semaphore *sem)

```
{
    int n = sem_post(sem)です。
    if (n != 0) perror_exit("sem_post failed")となります
}
```

私はこの操作を「ポスト」ではなく「シグナル」と呼びたいのですが、どちらも一般的な用語です。

ここでは、セマフォをミューテックスとして使用する例を紹介します。

セマフォ *mutex = make_semaphore(1);

semaphore_wait(mutex)です。

```
// 保護されたコードはここに入る
semaphore_signal(mutex);
```

セマフォをミューテックスとして使用する場合、通常はセマフォを1に初期化して、ミューテックスがアンロックされていることを示します。つまり、1つのスレッドがブロックすることなくセマフォを渡すことができます。

ここでは、セマフォがミューテックスとして使用されていることを示すために、変数名 mutex を使用しています。しかし、セマフォの動作はPthreadのミューテックスとは異なることを覚えておいてください。

11.2 Producers and consumers with semaphores

これらのセマフォのラッパー関数を使って、10.2節のProducer-Consumer問題の解決策を書くことができます。この節のコードはqueue_sem.cにあります。

これはQueueの新しい定義で、mutexとcondition変数をsemaphoresに置き換えています。

```
typedef
struct { int
*array; int
length; int
next_in; int
next_out;
Semaphore *mutex;       //-- new
Semaphore *items;       //-- new
Semaphore *spaces;      //-- new
} Queue;
```

そして、新しいバージョンのmake_queue

です。キュー *make_queue(int length)

```
{
    Queue *queue = (Queue *) malloc(sizeof(Queue));
    queue->length = length;
    queue->array = (int *) malloc(length * sizeof(int));
    queue->next_in = 0;
    queue->next_out = 0;
    queue->mutex = make_semaphore(1);
```

```
    queue->items = make_semaphore(0)
```

となります。

```
    queue->spaces = make_semaphore(length-1);
    return queue;
}
```

mutexは、キューへの排他的なアクセスを保証するために使用されます。初期値は1なので、ミューテックスは最初はアンロックされています。

items は、キューに入っているアイテムの数で、これは queue_pop をブロックせずに実行できるコンサマースレッドの数でもあります。初期状態では、キューにアイテムはありません。

spacesは、キュー内の空のスペースの数で、これは、ブロックせずにqueue_pushを実行できるプロデューサー・スレッドの数です。初期状態では、space数はキューの容量であり、セクション10.1で説明したように、length-1となります。

ここでは、プロデューサー・スレッドによって実行される新しいバージョンのqueue_pushを紹介します。

```
void queue_push(Queue *queue, int
item) { semaphore_wait(queue->spaces);
semaphore_wait(queue->mutex);

queue->array[queue->next_in] = item;
tail->next_in = tail_incr(tail, tail->next_in);

semaphore_signal(queue->mutex);
semaphore_signal(queue->items);
}
```

代わりに、セマフォが利用可能なスペースの数を追跡し、キューが満杯になった場合にプロデューサをブロックします。

queue_popの新バージョンです。

```
int queue_pop(Queue
*queue)
{ semaphore_wait(queue-
>items);
semaphore_wait(queue-
>mutex);

int item = queue->array[queue->next_out];
queue->next_out = queue_incr(queue, queue->next_out)。
```

```
    semaphore_signal(queue->mutex);
    semaphore_signal(queue->spaces);

    リターンアイテム
}
```

この解決策は、The Little Book of Semaphoresの第4章で、疑似コードを使って説明されています。

本書のリポジトリにあるコードを使えば、このソリューションを次のようにコンパイルして実行できるはずです。

```
$ make queue_sem
$ ./queue_sem
```

11.3 Make your own semaphores

セマフォで解決できる問題は、条件変数やミューテックスでも解決できます。条件変数とミューテックスを使ってセマフォを実装することで、それが正しいことを証明できます。

先に進む前に、練習としてこれをやってみるといいかもしれません。条件変数とミューテックスを使って、sem.h のセマフォ API を実装する関数を書いてみましょう。この本のリポジトリには、mysem_soln.c と mysem_soln.h に私のソリューションがあります。

難しい場合は、私のソリューションにある以下の構造定義をヒントにしてみてください。

```
typedef struct {
  int値、ウェイク
  アップ
  ミューテックス
  *mutex;
  コンド *cond;
} Semaphore。
```

はセマフォの値です。 wakeups は保留中のシグナルの数を数えます。wakeupsが必要な理由は、The Little Book of Semaphoresで説明されているProperty 3をセマフォが持っていることを確認するためです。

mutexは値とウェイクアップへの排他的アクセスを提供し、condはセマフォを待つ場合にスレッドが待つ条件変数です。

この構造体の初期化コードを紹介します。

```
セマフォ *make_semaphore(int value)
{
  セマフォ *semaphore = check_malloc(sizeof(Semaphore));
```

```
    semaphore->value = value;
    semaphore->wakeups = 0;
    semaphore->mutex = make_mutex();
    semaphore->cond = make_cond();
    return semaphore;
}
```

11.3.1 Semaphore implementation

ここでは、POSIXのミューテックスと条件変数を使ったセマフォの実装を紹介します。

```
void semaphore_wait(Semaphore *semaphore)
{
  mutex_lock(semaphore->mutex);
  semaphore->value--。

    if (semaphore-
    >value < 0) { do
    {...
      cond_wait(semaphore->cond, semaphore->mutex);
    } while (semaphore->wakeups
    < 1); semaphore->wakeups--。
  }
  mutex_unlock(semaphore->mutex)。
}
```

スレッドがセマフォを待つときは、値をデクリメントする前にミューテックスをロックする必要があります。セマフォの値が負になると、スレッドは「ウェイクアップ」が可能になるまでブロックします。ブロックされている間、ミューテックスのロックは解除されているので、別のスレッドがシグナルを送ることができます。

以下はsemaphore_signalのコードです。

```
void semaphore_signal(Semaphore *semaphore)
{
  mutex_lock(semaphore->mutex);
  semaphore->value++;

    if (semaphore-
    >value <= 0)
    { semaphore-
    >wakeups++;
    cond_signal(semap
    hore->cond);
  }
  mutex_unlock(semaphore->mutex)。
}
```

この場合も、スレッドは値をインクリメントする前にミューテックスをロ

ックしなければなりません。セマフォが負の値だった場合は、スレッドが待っていることを意味するので、シグナリング・スレッドがwakeupをインクリメントし、条件変数にシグナルを送ります。

この時点で、待機中のスレッドの1つが目覚めるかもしれませんが、ミューテックスは、シグナリング・スレッドがロックを解除するまでロックされたままです。
その時点で、待機中のスレッドの1つがcond_waitから戻り、チェックします。

は、ウェイクアップがまだ可能かどうかを判断します。もしそうでなければ、ループして再び条件変数を待ちます。そうであれば、wakeupを減らし、ミューテックスのロックを解除して終了します。

このソリューションでは、do...whileループが使用されていることが明らかではないかもしれません。なぜ、従来のwhileループではないのか、わかりますか？何がいけないのでしょうか？

問題は、whileループを使った場合、この実装ではProperty 3が成立しないことです。スレッドがシグナルを発した後、走り回って自分のシグナルをキャッチすることが可能になります。
do...whileループでは、あるスレッドがシグナルを発すると、待ち受けているスレッドの1つがシグナルを受け取ることが保証されています1。たとえ、待ち受けているスレッドの1つが再開する前にシグナルを発するスレッドが走り回ってミューテックスを取得したとしてもです。

[1]Well, almost. It turns out that a well-timed spurious wakeup (see http://en.wikipedia.org/wiki/Spurious_wakeup）はこの保証に違反する可能性があります。