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

Green Tea Press
9 Washburn Ave
ニーダム MA 02492

本書のLATEXソースは、http://greenteapress.com/ thinkosから入手できます。

巻頭言

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黙の前提としています。

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、プログラマが知っておかなければならないこと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 コードの使い方

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。

- Forkボタンを押すと、GitHub上に私のリポジトリのコピーを作成することができます。まだGitHubのアカウントをお持ちでない方は、アカウントを作成する必要があります。フォークする

と、GitHub上に自分のリポジトリができあがります。

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。

- あるいは、私のリポジトリをクローンすることもできます。これにはGitHubのアカウントは必要ありませんが、変更内容をGitHubに書き戻すことはできません。
- もし、Gitを全く使いたくない場合は、GitHubのページの右下にあるボタンを使って、ファイルをZipファイルでダウンロードすることができます。

投稿者リスト

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。

- 2014年春に本書の初期のドラフトをテストしてくれたOlin CollegeのSoftware Systemsの学生たちに感謝しています。彼らは多くの誤りを修正し、多くの有益な提案をしてくれました。彼らのパイオニア精神に感謝しています。
- James P Giannoules氏がコピー＆ペーストのミスを発見しました。
- アンディ・エングルは、GBとGiBの違いを知っています。
- Aashish Karkiは、いくつかの壊れた構文を指摘しました。

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

第1章 編纂

1.1　コンパイル言語とインタプリタ言語

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。

1.2　静止画タイプ

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ため、エラーを早期に発見することができます。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選択し、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]変数の位置を**アドレス**と呼ぶ。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を除く）。

1.3 編集作業の様子

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. 前処理。C言語は、プログラムがコンパイルされる前に効果を発揮する**前処理ディレクティブ**を含むいくつかの言語の1つです。例えば、#include指令は、他のファイルのソースコードを指令の位置に挿入します。

2. 構文解析を行う。構文解析では、コンパイラがソースコードを読み、**抽象構文木**と呼ばれるプログラムの内部表現を構築します。この段階で検出されるエラーは一般的に構文エラーです。

3. 静的チェック。コンパイラは、変数や値が正しい型であるか、関数が正しい数と型の引数で呼び出されているかなどをチェックします。この段階で検出されたエラーは、**静的セマンティック**エラーと呼ばれることもあります。

4. コード生成。コンパイラは、プログラムの内部表現を読み取って、マシンコードまたはバイトコードを生成する。

5. リンクする。プログラムがライブラリで定義された値や関数を使用する場合、コンパイラは適切なライブラリを探し、必要なコードをインクルードする必要があります。

[1]これは単純化したもので、詳細は後ほど説明します。

6. 最適化。プロセスのいくつかのポイントで、コンパイラはプログラムを変換して、より高速に動作するコードやより少ないスペースで使用できるコードを生成することができます。ほとんどの最適化は、明らかな無駄を省く単純な変更ですが、コンパイラによっては高度な分析や変換を行うものもあります。

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
```

1.4 オブジェクトコード

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てください。

1.5 アセンブリコード

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。

```
                .ファイル"hello.c"
                .セクション.ロダータ
.LC0:
        .文字列     "Hello World"
        .テキスト
                .globlmain
        .typ        emain, @function
のメイン
となり
ます。
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
.LFB0:  movq %rsp, %rbp
        .cfi_def_cfa_register 6
        movl $.LC0, %edi
        コールプット
```

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

1.6 前処理

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。

1.7 エラーの理解

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。

第2章 プロ

セス

2.1　抽象化と仮想化

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。

- 抽象化。抽象化とは、複雑なものを単純化して表現したものです。例えば、車を運転する人は、ハンドルを左に回すと車が左に行き、逆に左に回すと車が右に行くことを理解していると思います。もちろん、ハンドルは車輪を回転させる一連の機械的および（多くの場合）油圧システムに接続されており、車輪は複雑な方法で道路と相互作用していますが、ドライバーとしては通常、そのような詳細について考える必要はありません。ステアリングに関するシンプルなメンタルモデルで十分にやっていけるのです。あなたのメンタルモデルは抽象化されています。

  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。

- 仮想化。　抽象化の重要な一種である仮想化とは、望ましい幻想を作り出すプロセスです。

  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場合があります。

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。

2.2 孤立

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オブジェクトを指します。プロセスは、以下のデータを含むオブジェクトである。

- プログラムのテキストで、通常は機械語の構造体のシーケンスである。
- プログラムに関連するデータで、スタティックデータ（コンパイル時に割り当てられたもの）とダイナミックデータ（ランタイム時に割り当てられたもの）があります。
- 保留中の入力/出力操作の状態。例えば、プロセスがディスクからのデータの読み取りやネットワーク上のパケットの到着を待っている場合、これらの操作の状態はプロセスの一部となる。
- レジスターに格納されているデータやステータス情報、現在どの命令を実行しているかを示すプログラムカウンターなど、プログラムのハードウェア状態を表します。

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。

- マルチタスキング。ほとんどのOSは、実行中のプロセスをいつでも中断してハードウェアの状態を保存し、後でプロセスを再開する機能を持っています。一般的に、プログラマーはこの割り込みについて考える必要はありません。プログラムはあたかも専用のプロセッサ上で連続して実行されているかのように動作しますが、割り込みの時間は予測できません。
- 仮想メモリ。ほとんどのOSでは、各プロセスが他のプロセスから隔離された独自のメモリの塊を持っているかのような錯覚に陥ります。プログラマーは通常、仮想メモリの仕組みを考える必要はなく、すべてのプログラムが専用のメモリを持っているかのように作業を進めることができます。
- デバイスの抽象化。同じコンピュータ上で動作するプロセスは、ディスクドライブ、ネットワークインターフェース、グラフィックカードなどのハードウェアを共有しています。もし、プロセスがこれらのハードウェアと協調せずに直接やりとりすると、混乱が生じてしまいます。例えば、あるプロセスが意図したネットワークデータを別のプロセスが読んでしまうことがあります。　また、複数のプロセスがハードディスクの同じ場所にデータを保存しようとするかもしれません。適切な抽象化を行うことで秩序を保つのがOSの役目です。

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。

2.3 UNIXプロセス

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、次のように表示されます。

```
  pid ttytime 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          タイム 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 移行/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:16 rcu-sched
   47 ?        00:00:00 cpuset
   48 ?        00:00:00 クヘルパ
                        ー
   49 ?        00:00:00 kdevtmpfs
   50 ?        00:00:00 ネット
   51 ?        00:00:00 bdi-default
   52 ?        00:00:00 キンテグ
                        リット
   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。

第3章 仮想記憶

3.1 情報理論の話

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、2つの^{b}値のいずれかを示すことができます。1**バイト**は8ビットなので、256個の値を格納することができます。

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つを指定する場合は，$2^b \geqq N$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，$b \geq log_2 N$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教えたとしたら，私はあなたに*対数*6ビットの情報を与えたことになります・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$-log_2 p$となります。この量は、結果の**自己情報**と呼ばれます。この量は、結果がどれだけ驚くべき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。

この本のいくつかのトピックでは、ビット数bと、ビットが符号化できる値の数$N = 2^b$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 メモリとストレージ

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、2^{30}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、10^9バイトのことです。TBは「テラバイト」の略で、10^{12}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 アドレス空間

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、Nはメインメモリのサイズを表します。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$2^{30} - 1$であり、10進数では1,073,741,823、16進数では0x3fff ffffとなります（接頭辞0xは16進数を表します）。

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2^{32}バイト、つまり4GiBである。

64ビットシステムでは、仮想アドレス空間のサイズは2^{64}バイト、つまり$2^{4}1024^{6}$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に思われ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　メモリセグメント

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。

- **コードセグメント**には、プログラムを構成する機械語の命令であるプログラムテキストが含まれています。
- **静的セグメント**には、文字列リテラルのような不変的な値が格納されます。例えば、プログラムに "Hello, World "という文字列が含まれていた場合、その文字はスタティックセグメントに格納されます。
- **グローバルセグメント**には、グローバル変数と、スタティック宣言されたローカル変数が含まれます。

- **ヒープセグメント**には、実行時に割り当てられるメモリの塊が含まれており、多くの場合、Cライブラリの関数mallocを呼び出すことで割り当てられます。
- **スタックセグメント**には、スタックフレームの並びであるコールスタックがあります。関数が呼び出されるたびに、その関数のパラメータやローカル変数を格納するスタックフレームが割り当てられます。関数が完了すると、そのスタックフレームはスタックから削除されます。

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

- テキストセグメントは、メモリの "底 "に近い場所、つまり0に近いアドレスにあります。
- スタティックセグメントは、テキストセグメントのすぐ上、つまり上位のアドレスにあることが多い。
- グローバルセグメントは、しばしばスタティックセグメントのすぐ上にあります。
- ヒープは多くの場合、グローバルセグメントの上にあります。拡張すると、より大きなアドレスに向かって伸びていきます。
- スタックは、メモリの最上部、つまり仮想アドレス空間の最上位アドレスの近くにあります。スタックが拡張すると、より小さなアドレスに向かって成長していきます。

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）。

```
メインのアドレスは0x      40057d
グローバルのアドレスは0x
                          60104c
ローカルのアドレスは
0x7ffe6085443cです。
pのポイント       0x      16c3010
は
のポイントは      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2^{48}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て、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。

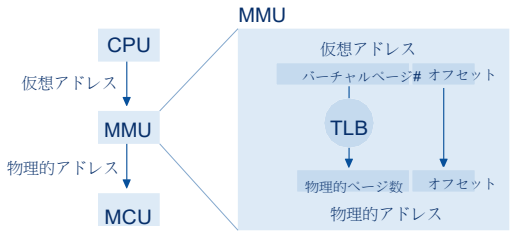

図3.1:アドレス変換処理のイメージ図

3.5 静的なローカル変数

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 アドレス変換

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. プログラムが変数を読み書きするとき、CPUはVAを生成します。
2. MMUは、VAをページ番号とオフセットと呼ばれる2つの部分に分割します。ページ」とはメモリの塊のことで、ページのサイズはOSやハードウェアによって異なりますが、一般的なサイズは1〜4KiBです。
3. MMUは、TLB（Translation Lookaside Buffer）でページ番号を検索し、対応する物理ページ番号を取得します。そして、物理ページ番号とオフセットを結合してPAを生成します。
4. PAはメインメモリに渡され、メインメモリは指定された場所を読み書きします。

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。

- 1GiBは2^{30}バイト、1KiBは2バイトなので、^{1020}物理的なページは2つあり、"フレーム "と呼ばれることもあります。
- 仮想アドレス空間のサイズは2^{32}B、1ページのサイズは2^{10}Bなので、2つの^{22}仮想ページが存在することになります。
- オフセットの大きさは、ページサイズによって決まります。この例では、ページサイズが2^{10}Bなので、ページ上の1バイトを指定するには10ビットが必要です。
- VAが32ビット、オフセットが10ビットの場合、残りの22ビットで仮想ページ番号を構成します。
- ^{20}物理ページは2つあるので、各物理ページ番号は20ビット。これに10ビットのオフセットを加えると、PAは30ビットになります。

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、2つの^{22}仮想ページがあれば、ページテーブルは2^{24}バイト、つまり16MiB必要になります。

さらに、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2^{32}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。

第4章

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。

- ファイルシステム」とは、各ファイルの名前からその内容までのマッピングのことです。名前をキー、内容を値と考えると、ファイルシステムはキー・バリュー・データベースの一種である（https://en.wikipedia.org/wiki/ Key-value_database参照）。
- ファイル」とはバイト列のことです。

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はファイル名から、/というトップレベルのディレクトリ、サブディレクトリのhome、サブサブディレクトリのdowneyを探します。

2. file.txtという名前のファイルを見つけ、それを読めるように「開く」。つまり、読まれるファイルを表すデータ構造を作るのです。 このデータ構造は、「ファイルポジション」と呼ばれる、ファイルがどれだけ読み込まれたかを記録しています。

   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. fgetcを呼び出すと、オペレーティングシステムは、ファイルの次の文字がすでにメモリ内にあるかどうかをチェックします。ファイルの次の文字がすでにメモリ内にあるかどうかをチェックし、ある場合は次の文字を読み込んでファイルの位置を進め、結果を返します。

4. 次の文字がメモリにない場合、OSは次のブロックを取得するためにI/Oリクエストを発行する。ディスクドライブは遅いので、通常、ディスクからのブロックを待つプロセスは中断され、データが到着するまで他のプロセスが実行される。

5. I/O操作が完了すると、新しいデータブロックがメモリに格納され、処理が再開されます。最初の文字を読み込んで、ローカル変数に格納します。

6. プロセスがファイルを閉じると、オペレーティングシステムは、保留中の操作を完了またはキャンセルし、メモリに保存されているデータを削除し、OpenFileTableEntryを解放します。

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. ここでもfopenはファイル名を使ってファイルを探します。まだ存在していなければ，新しいファイルを作成し，親ディレクトリにエントリを追加します。
   /home/downey.
2. オペレーティングシステムは、ファイルが書き込み可能な状態であることを示すOpenFileTableEntryを作成し、ファイルポジションを0に設定します。
3. fputcは、ファイルの最初のバイトを書き込もうとします（または再書き込もうとします）。ファイルがすでに存在している場合、オペレーティングシステムは最初のブロックをメモリにロードする必要があります。そうでない場合は、メモリに新しいブロックを割り当て、ディスクに新しいブロックを要求します。
4. メモリ上のブロックが変更された後、すぐにディスクにコピーバックされない場合があります。一般的に、ファイルに書き込まれるデータは「バッファリング」されており、メモリに格納され、書き込むべきブロックが少なくとも1つあるときにのみディスクに書き込まれる。
5. ファイルが閉じられると、バッファリングされていたデータがディスクに書き込まれ、OpenFileTableEntryが解放されます。

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 ディスクの性能

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25μs、書き込むのに250μs 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するので、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す。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。

- ブロック転送。ディスクから1バイトをロードするのにかかる時間は5〜25msです。それに比べて、8KiBのブロックを読み込むのにかかる時間はごくわずかです。そのため、システムはディスクにアクセスするたびに大きなブロックを読み込もうとします。

- プリフェッチ：オペレーティング・システムは、あるプロセスがブロックを読み、要求される前にロードを開始することを予測できる場合があります。例えば、ファイルを開いて最初のブロックを読んだ場合、2番目のブロックを読み進める可能性が高いと考えられます。オペレーティングシステムは、要求される前に追加のブロックの読み込みを開始するかもしれません。

- バッファリング。先に述べたように、ファイルを書き込むとき、OSはデータをメモリに保存し、後でディスクに書き込むだけです。メモリ上にあるブロックを何度か変更しても、ディスクへの書き込みは1回で済みます。

- キャッシング：あるプロセスが最近ブロックを使用した場合、すぐにまたそのブロックを使用する可能性があります。オペレーティングシステムがそのブロックのコピーをメモリに保持していれば、将来の要求をメモリの速度で処理することができます。

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 ディスクのメタデータ

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 ブロック配分

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

- スピード。ブロックの割り当てと解放は高速でなければなりません。
- スペースオーバーヘッドを最小限に抑える。アロケータが使用するデータ構造は小さく、データのためにできるだけ多くのスペースを確保する必要があります。
- フラグメンテーションは最小限に。一部のブロックが使用されずに残っていたり、一部しか使用されていない場合、その未使用領域は「フラグメンテーション」と呼ばれます。
- 最大限の連続性。パフォーマンスを向上させるために、同時に使用される可能性のあるデータは、可能であれば物理的に連続していることが望ましい。

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 すべてがファイル？

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。

また、ネットワーク通信では、バイトのストリームという抽象化が用いられる。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　整数の表現

コンピュータが数字を2進法で表現していることは、皆さんもご存知でしょう。正の数の場合、2進法での表現は簡単で、例えば、5の$_{10}$表現はb101です。

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、-5$_{10}$を表現するには、5の$_{10}$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 ビット演算子

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
| 0011
  -----
  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 浮動小数点数の表現

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表記します。　例えば、光の速さ（m/s）は約2.998 - 10です[8]。

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。

$$(-1)^s c \cdot 2^q$$

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 - 10^5やその他の同等の表現ではなく、2.998 - 10を使用します。[8]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なので、表現から外すことでスペースを節約できます。

例えば、13の整数表現$_{10}$はb1101です。浮動小数点では「$1.101 \cdot 2^3$」なので、指数は3、格納される係数の部分は101（その後にゼロが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;

unsigned int coef_mask = (1 << 23) - 1;
unsigned int coef = p.u & coef_mask;

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 ユニオンとメモリーエラー

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）に配置したと推測できます。　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　文字列の表現

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 は，整数のサイズをバイト単位で受け取り，新たに割り当てられた（少なくとも）指定されたサイズのメモリチャンクへのポインタを返します。要求を満たすことができない場合は，特別なポインタ値NULLを返します。
- callocは、mallocと同じですが、新たに割り当てられたチャンクをクリアします（チャンク内のすべてのバイトを0にします）。
- free は，以前に割り当てられたチャンクへのポインタを受け取り，そのチャンクを処理します（つまり，将来の割り当てのためにスペースを利用可能にします）。
- realloc は，以前に割り当てられたチャンクへのポインタと，新しいサイズを受け取ります。新しいサイズのメモリチャンクを確保し，古いチャンクから新しいチャンクにデータをコピーし，古いチャンクを解放し，新しいチャンクへのポインタを返します。

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 メモリエラー

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。

- 割り当てられていないチャンクにアクセス（読み書き）すると、それはパドリングになります。
- 割り当てられたチャンクを解放してからアクセスすると、それはパドリングになります。
- 割り当てられていないチャンクを解放しようとすると、それはパドリングになります。
- 同じチャンクを2回以上フリーにするとパドリングになります。
- 割り当てられていないチャンクや、割り当てられた後に解放されたチャンクでreallocを呼び出した場合、それはパドリングです。

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。

- 未割り当てのチャンクから値を読み出した場合、システムがエラーを検出し、「セグメンテーション・フォールト」と呼ばれるランタイムエラーが発生し、プログラムが停止することがあります。この場合、アクセスされた場所にたまたま保存されていた値が取得されますが、この値は予測できず、プログラムを実行するたびに異なる可能性があります。

- 割り当てられていないチャンクに値を書き込み、セグメンテーション・フォールトが発生しなかった場合、事態はさらに悪化します。無効な場所に値を書き込んだ後、それが読み込まれて問題が発生するまでに長い時間がかかるかもしれません。その時点で、問題の原因を見つけるのは非常に難しいでしょう。

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のための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 メモリリーク

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。

- ある時点で、システムは物理メモリを使い果たしてしまいます。仮想メモリを持たないシステムでは、次のmallocの呼び出しに失敗し、NULLが返されます。

- 仮想メモリを備えたシステムでは、オペレーティングシステムは、他のプロセスのページをメモリからディスクに移動させ、リークしたプロセスにより多くのスペースを割り当てることができます。このメカニズムについては7.8節で説明します。

- 1つのプロセスが割り当てられる容量には制限があるかもしれませんが、それを超えるとmallocはNULLを返します。

- 最終的に、プロセスはその仮想アドレス空間（または使用可能な部分）を埋め尽くすかもしれません。そうなると、割り当てられるアドレスがなくなってしまうので、mallocはNULLを返します。

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 インプリメンテーション

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。

- mallocの実行時間は、通常、チャンクのサイズには依存しませんが、空きチャンクの数に依存する場合があります。freeは、空きチャンクの数に関係なく、通常は高速です。callocはチャンク内のすべてのバイトをクリアするため、実行時間はチャンクのサイズ（およびフリーチャンクの数）に依存します。

  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。

- バウンダリータグ。mallocがチャンクを割り当てる際、チャンクのサイズや状態（割り当て済みか空きか）など、チャンクに関する情報を格納するためのスペースを最初と最後に追加します。これらのデータを「境界タグ」と呼びます。これらのタグを使用して、mallocは任意のチャンクからメモリ内の前のチャンクと次のチャンクに移動することができます。さらに、フリーチャンクは二重にリンクされたリストに連結されており、各フリーチャンクには「フリーリスト」の次のチャンクと前のチャンクへのポインタが含まれています。

  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。

- スペースのオーバーヘッド。バウンダリータグやフリーリストのポインターはスペースを取ります。ほとんどのシステムでは、チャンクの最小サイズは16バイトです。そのため、非常に小さなチャンクの場合、malloc はスペース効率が悪くなります。小さな構造体を大量に必要とするプログラムの場合は、配列で割り当てたほうが効率的かもしれません。

- フラグメンテーション。さまざまなサイズのチャンクを割り当てたり解放したりすると、ヒープはフラグメント化する傾向があります。つまり、空き領域が多くの小さな断片に分割されてしまうのです。フラグメント化はスペースを浪費し、メモリキャッシュの効果を低下させるため、プログラムの速度を低下させます。

- ビン化とキャッシング: フリーリストはサイズごとにビンに分類されているので、mallocが特定のサイズのチャンクを検索するときに、どのビンで検索すればよいかがわかります。チャンクを解放した後、すぐに同じサイズのチャンクを割り当てる場合、malloc は通常、高速に動作します。

第7章 キャ

ッシング

7.1　プログラムの動き

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

- プログラムカウンタ（PC）は、プログラムの次の命令のアドレス（メモリ内）を含んでいます。
- 現在実行中の機械コードの命令を格納した命令レジスタ（IR）。
- スタックポインタ（SP）は、現在の関数のスタックフレームのアドレスを表し、パラメータやローカル変数が含まれています。
- プログラムが現在扱っているデータを保持する汎用レジスター。

- 現在の演算に関する情報を格納するステータスレジスタ（フラグレジスタ）のこと。例えば、フラグレジスタには、前回の演算結果がゼロの場合にセットされるビットが通常含まれています。

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。

- フェッチする。次の命令をメモリからフェッチし、命令レジスタに格納する。
- デコード。コントロールユニット」と呼ばれるCPUの一部は、指示された内容をデコードし、CPUの他の部分に信号を送ります。
- 実行します。コントロールユニットからの信号により、適切な計算が行われる。

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。

- ロード：メモリからレジスタに値を転送する。
- 算術/論理。レジスタからオペランドを読み込み、数学的な演算を行い、その結果をレジスタに格納する。
- ストア。レジスタの値をメモリに転送する。
- jump/branch。プログラムカウンタを変更し、実行の流れをプログラムの別の場所にジャンプさせます。分岐は通常、条件付きで行われます。つまり、フラグレジスタのフラグをチェックし、それがセットされている場合にのみジャンプするのです。

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 キャッシュパフォーマンス

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T_h、キャッシュのミスを処理する時間をTとする$_m$と、各メモリアクセスの平均時間は次のようになります。

$$hT_h + mT_m$$

同じように、「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、$T_p = T_m - T$ と$_h$定義することができます。すると、平均アクセスタイムは

$$T_h + mT_p$$

ミス率が低ければ、平均アクセスタイムはTに$_h$近い値になります。つまり、メモリがキャッシュスピードで動作しているかのようにプログラムを実行することができるのである。

7.3 地域性

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位置していたとします。最初のバイトを読み込む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%となります。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。

- ほとんどのプログラムは、ジャンプや分岐のないコードブロックで構成されています。このブロックの中では、命令が順次実行されるため、アクセスパターンには空間的な局所性があります。
- ループでは、プログラムが同じ命令を何度も実行するため、アクセスパターンには時間的な局所性があります。
- ある命令の結果は、次の命令のオペランドとしてすぐに使われることが多いので、データアクセスパターンには時間的なロカリティがあります。
- プログラムが関数を実行するとき、そのパラメータやローカル変数はスタックにまとめて格納され、これらの値にアクセスすると空間的なローカル性を持つことになります。
- 最も一般的な処理パターンは、配列の要素を順番に読み書きすることで、これは空間的な局所性を持っています。

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　キャッシュパフォーマンスの測定

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

っ

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，これは平均時間を十分な精度で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();
```

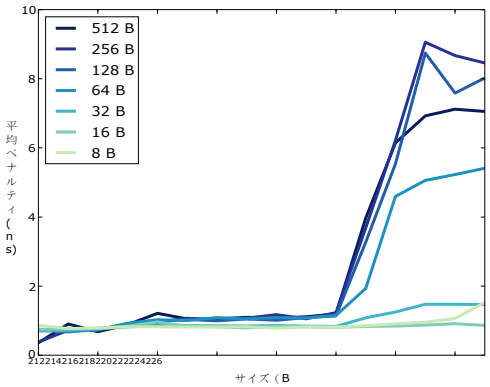

図 7.1:配列サイズとストライドの関数としての平均ミスペナルティ。

```
        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。

```
サイズ 4096 ストライドです。 8 read+write 0.8633 ns です。
サイズ 4096 ストライドです。 16 read+write 0.7023 ns です。
サイズ 4096 ストライドです。 32 read+write 0.7105 ns です。
サイズ 4096 ストライドです。 64 read+write 0.7058 ns です。
```

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。

- プログラムは配列を何度も読み込んでいるので、時間的なロカリティは十分にあります。配列全体がキャッシュに収まっていれば、平均ミスペナルティはほぼ0になると予想されます。
- ストライドが4バイトの場合は、配列のすべての要素を読み取るので、プログラムには十分な空間的なロカリティがあります。例えば、ブロックサイズが64個の要素を含むほど大きければ、配列がキャッシュに収まらなくても、ヒット率は63/64となります。
- ストライドがブロックサイズと同じ（またはそれ以上）の場合、ブロックを読み込むたびに1つの要素にしかアクセスできないため、空間的なロカリティは事実上ゼロになります。この場合、最大のミスペナルティが発生することになります。

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2^{22}B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、実際の仕様では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2^{14}B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のキャッシュが搭載されている可能性があります。

7.5 キャッシュパフォーマンスのためのプログラミング

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ピースに分割し、そのピース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 メモリ階層

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。

デバイス	アクセス時間	典型的なサイズ	コスト
登録	0.5ns	256 B	?
キャッシュ	1ナノ秒	2 MiB	?
DRAM	100 ns	4GiB	10ドル/GiB
SSD	10 μs	100GiB	1ドル/GiB
HDD	5 ms	500GiB	0.25ドル/GiB
テープ	分	1-2 TiB	0.02ドル/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、これらの数値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　キャッシングポリシー

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。

- 階層の上下にデータを移動させるのは誰ですか？階層の一番上では、レジスタの割り当ては通常、コンパイラが行います。CPUのハードウェアはメモリキャッシュを扱います。ユーザーは暗黙のうちにデータをストレージから

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。

- 何が移動するのか？一般的にブロックサイズは、階層の上の方が小さく、下の方が大きくなっています。メモリのページは4KiBかもしれませんが、OSがディスクからファイルを読み出すときには、一度に10ブロック、100ブロックを読み出すこともあります。

- データはいつ移動されるのですか？最も基本的なキャッシュでは、データが最初に使用されたときにキャッシュに移動します。しかし、多くのキャッシュはある種の「プリフェッチ」を使用しています。つまり、明示的に要求される前にデータが読み込まれるのです。プリフェッチの一例として、ブロックの一部しか要求されていないのに、ブロック全体を読み込む方法を見てみました。

- データはキャッシュのどこに入るのか？キャッシュがいっぱいになると、何かを追い出さないと何も入ってこなくなります。　理想的には、すぐに再利用されるデータを残し、そうでないデータを入れ替えることです。

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み出します。

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　ページング

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. プロセスAがmallocを呼び出してチャンクを割り当てたとします。ヒープ内に要求されたサイズの空き領域がない場合、mallocはsbrkを呼び出し、オペレーティングシステムにメモリの追加を要求します。

2. 物理メモリに空きページがあれば、OSはそのページをプロセスAのページテーブルに追加し、有効な仮想アドレスの範囲を新たに作成します。

3. 空きページがない場合、ページングシステムはプロセスBに属する「犠牲者ページ」を選択します。犠牲者ページの内容をメモリからディスクにコピーした後、プロセスBのページテーブルを変更して、このページが「スワップアウト」されたことを示します。

4. プロセスBのデータが書き込まれたら、そのページをプロセスAに再割り当てすることができます。プロセスAがプロセスBのデータを読み取れないようにするには、そのページをクリアする必要があります。

5. この時点で sbrk の呼び出しは戻り、malloc にヒープ内の追加スペースを与えることができます。その後、malloc は要求されたチャンクを割り当てて戻ります。プロセスAは再開できます。

6. プロセスAが完了したり、中断されたりすると、スケジューラーはプロセスBの再開を許可することがあります。プロセスBがスワップアウトされたページにアクセスすると、メモリ管理ユニットはそのページが「無効」であることに気づき、割り込みを発生させます。

7. オペレーティングシステムが割り込みを処理する際に、ページがスワップアウトされたことを確認し、そのページをディスクからメモリに戻します。

8. ページを交換すると、プロセスBが再開されます。

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。

- ほとんどのプロセスは、割り当てられたメモリのすべてを使用しません。テキストセグメントの多くの部分は、一度も実行されないか、一度だけ実行されて二度と実行されない。これらのページは、問題を起こさずにスワップアウトすることができます。

- プログラムがメモリをリークした場合、確保した領域を残して二度とアクセスしない可能性があります。このようなページをスワップすることで、OSは効果的にリークを防ぐことができます。

- ほとんどのシステムでは、デーモンのように、ほとんどの時間をアイドル状態で過ごし、時々イベントに対応するために「目覚める」だけのプロセスがあります。アイドル状態のプロセスは、スワップアウトすることができます。

- ユーザーは多くのウィンドウを開いているかもしれませんが、一度にアクティブなのは数個だけです。アクティブでないプロセスはスワップアウトすることができます。
- また、同じプログラムを実行するプロセスが多数存在する場合もあります。これらのプロセスは、同じテキストやスタティックセグメントを共有することができるので、物理メモリに複数のコピーを保持する必要がありません。

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。

第8章 マルチタスキング

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 ハードウェアの状態

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. 割り込みが発生すると、ハードウェアはプログラムカウンタを特別なレジスタに保存し、適切な割り込みハンドラにジャンプします。
2. 割り込みハンドラは、プログラムカウンタとステータスレジスタをメモリに格納し、使用する予定のデータレジスタの内容も格納します。
3. 割り込みハンドラは、その割り込みを処理するために必要なコードを実行します。
4. 次に、保存されたレジスタの内容を復元します。最後に、中断されたプロセスのプログラムカウンタを復元することで、中断された命令にジャンプバックする効果があります。

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 コンテキストの切り替え

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 プロセスのライフサイクル

プロセスが作成されると、オペレーティングシステムは、「プロセス制御ブロック」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, プロセスが現在コア上で実行されている場合。
- Ready」は、プロセスが実行可能であるにもかかわらず実行されていない場合で、通常は実行可能なプロセスの数がコアの数よりも多いために発生します。
- ブロック：ネットワーク通信やディスクの読み取りなど、将来のイベントを待っているためにプロセスが実行できない場合。
- プロセスは完了したが、まだ読み込まれていない終了ステータス情報がある場合、Done。

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。

- プロセスは、実行中のプログラムがforkなどのシステムコールを実行したときに生成されます。システムコールの終了時には、通常、新しいプロセスの準備ができています。その後、

スケジューラは元のプロセス（「親」）を再開するか、新しいプロセス（「子」）を開始するかを選択します。

- スケジューラーによってプロセスが開始または再開されると、その状態はreadyからrunningに変わります。
- プロセスが中断され、スケジューラーが再開させないことを選択すると、そのプロセスの状態は実行中から準備完了に変わります。
- プロセスがディスク要求などのすぐに完了できないシステムコールを実行した場合、そのプロセスはブロックされ、スケジューラーは通常、別のプロセスを選択します。
- ディスクリクエストのような操作が完了すると、割り込みが発生します。割り込みハンドラは、どのプロセスがリクエストを待っていたかを把握し、そのプロセスの状態をブロックからレディに切り替えます。その後、スケジューラはブロックされていないプロセスの再開を選択するかどうかを決めます。
- プロセスがexitを呼び出すと、割り込みハンドラはexitコードをPCBに格納し、プロセスの状態をdoneに変更します。

8.4 スケジューリング

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。

- プロセスはさまざまなリソースによって制限されることがあります。多くの計算を行うプロセスは、おそらくCPUに依存しており、その実行時間はCPU時間に依存しています。ネットワークやディスクからデータを読み取るプロセスは、I/Oに依存しているかもしれません。これは、データの入出力が速くなれば実行速度も速くなりますが、CPU時間が増えても実行速度は速くなりません。最後に、ユーザーと対話するプロセスは、おそらくほとんどの時間、ユーザ

ーのアクションを待つためにブロックされています。

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。

- もし、あるプロセスが短時間だけ実行され、その後ブロック化されたリクエストを行う可能性があるならば、2つの理由から、そのプロセスはおそらくすぐに実行されるべきです。(1) リクエストが完了するまでに時間がかかる場合、可能な限り早く開始すべきである、(2) 長時間実行しているプロセスが短時間のリクエストを待つことは、その逆よりも良いことである。

  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。

- プロセスは通常、比較的高い優先度で開始され、素早く実行されます。

- あるプロセスがリクエストを行い、そのタイムスライスが完了する前にブロックした場合、そのプロセスはインタラクティブまたはI/Oバウンドである可能性が高いため、その優先度を上げる必要があります。

- タイムスライス全体でプロセスが実行されている場合、長時間実行されていてCPUに負荷がかかっている可能性が高いため、その優先度を下げる必要があります。

- タスクが長い間ブロックしていて、その後準備ができたら、待っていたものに対応できるように優先度を上げるべきです。

- プロセスAがプロセスBを待つためにブロックされている場合、例えばパイプでつながっている場合は、プロセスBの優先度が上がるはずです。

- システムコールNiceは、プロセスが自身の優先度を下げる（上げることはできない）ことを可能にし、プログラマーがスケジューラーに明示的な情報を渡すことを可能にします。

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 リアルタイム・スケジューリング

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。

- タスクの優先度をコントロールするためのリッチなAPIを提供。
- 最も優先度の高いプロセスが一定時間内に実行されることを保証するためのスケジューラの変更。
- 割り込みハンドラを再編成し、最大完了時間を保証する。
- ロックやその他の同期メカニズム（次の章で説明します）を変更して、優先度の高いタスクが優先度の低いタスクを先取りできるようにする。
- 最大完了時間を保証する動的メモリ割り当ての実装を選択する。

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。

第9章 スレッド

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 スレッドの作成

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。

- プログラムの最初にヘッダーファイルを入れます。
- Pthreadsで定義された関数を呼び出すコードを書きます。
- プログラムをコンパイルする際には、Pthreadライブラリとリンクさせます。

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 スレッドの作成

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 スレッドの結合

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.ここでは、pthread joinのための私のラッパーを紹介します。

```
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。

```
カウン = 0
ター
カウン = 0
ター
カウン = 1
ター
カウン = 0
ター
カウン = 3
ター
```

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 シンクロエラー

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

C子が読む0 A子
が印刷する 0
B子が印刷する 0
子Aはカウンタ=1を
設定 子Dは1を読み
出す
子D が
print1 子C が
print0 子Aがcounter=1
子Bがcounter=2 子C
がcounter=3 子Eが3を
読む
子Eが プリ
ントする3 子Dがカ
ウンター＝4を設定
子Eがカウンター＝5
を設定
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。

カウン = 0
ター
カウン = 1
ター
カウン = 2
ター
カウン = 3
ター
カウン = 4
ター

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        (); //--この行は新しい 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 ミューテックス

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。

第10章 条件変数

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 ワークキュー

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、構造の定義について説明します。

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

}キューです。

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("queue is full")。
    }

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

```
    queue->next_in = queue_incr(queue, queue->next_in);
}
```

キューがいっぱいになった場合、queue_pushはエラーメッセージを表示して終了します。

queue_fullについては、近々説明します。

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を使用して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 *queue)
{
    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 生産者と消費者

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。

- キューへのアクセスはスレッドセーフではありません。異なるスレッドが同時に array, next_in, next_out にアクセスしてしまい、キューが壊れた「矛盾した」状態になってしまう可能性があります。
- コンシューマが最初にスケジューリングされた場合、キューが空であることを発見し、エラーメッセージを出力して終了します。私たちはむしろ、キューが空でなくなるまでコンシューマをブロックさせたいと考えています。同様に、producer はキューが満杯になるまでブロックするようにしたいと思います。

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 相互の排除

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                ; //--この行は新しいです 。
```

}キューです。

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          (); //--
    新しいリターンの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、ある時点で、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 条件変数

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      ; //--新しい
```

}キューです。

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        ); //--
  新しいリターンアイテム。
}
```

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(queue->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。

- なぜcond_waitはif文ではなくwhileループの中にあるのか、つまりcond_waitから戻ってきた後に再度条件を確認する必要があるのか。

  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す。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。

- 条件変数について学ぶときに出てくるもう一つの疑問は、"条件変数はどのようにして自分が関連している条件を知るのか？"ということです。

  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 条件変数の実装

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，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。

第11章 C言語のセ

マフォ

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セマフォ

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。このインターフェースは、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 セマフォ を使った プロデューサーと コンシューマー

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;
    セマフォ *mutex          ; //--新しい
    Semaphore *items         ; //--
    new Semaphore *spaces    ; //--
    new
```

}キューです。

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;
   queue->next_in = queue_incr(queue, queue->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 自分でセマフォを作る

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 value, wakeups;
  Mutex *mutex;
  コンド *cond;
}セマフォ。
```

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の実装

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(semaphore->cond);
  }
  mutex_unlock(semaphore->mutex)。
}
```

ここでも、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する必要があります。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したとき、待っているスレッドの1つがシグナルを受け取ることが保証[1]ています。

[1]まあ、ほとんどそうですね。タイミングよく 行われるスプリアスウェイクアップ（http://en. wikipedia.org/wiki/Spurious_wakeup参照）がこの保証を破ることができることがわかっています。