

# OS開発の小さな本

Erik Helin, Adam Renberg

2015-01-19 | Commit: [fe83e27dab3c39930354d2dea83f6d4ee2928212](#)



# コンテンツ

<b>1 Introduction</b>	<b>7</b>
About the Book.....	7
The Reader.....	8
Credits, Thanks and Acknowledgements.....	8
Contributors .....	8
Changes and Corrections .....	8
Issues and where to get help.....	9
License.....	9
 <b>2 First Steps</b>	 <b>11</b>
Tools .....	11
Quick Setup.....	11
Programming Languages.....	11
Host Operating System .....	12
Build System .....	12
Virtual Machine.....	12
Booting .....	12
BIOS.....	12
The Bootloader.....	13
The Operating System .....	13
Hello Cafebabe.....	13
Compiling the Operating System .....	13
Linking the Kernel .....	14
Obtaining GRUB.....	15
Building an ISO Image.....	15
Running Bochs.....	16

Further Reading .....	17
<b>3 Getting to C</b> .....	<b>19</b>
Setting Up a Stack .....	19
Calling C Code From Assembly.....	20
Packing Structs.....	20
Compiling C Code .....	21
Build Tools.....	21
Further Reading .....	22
<b>4 Output</b> .....	<b>23</b>
Interacting with the Hardware .....	23
The Framebuffer .....	23
Writing Text .....	23
Moving the Cursor.....	25
The Driver .....	26
The Serial Ports.....	26
Configuring the Serial Port.....	27
Configuring the Line .....	27
Configuring the Buffers.....	29
Configuring the Modem.....	29
Writing Data to the Serial Port.....	30
Configuring Bochs.....	31
The Driver .....	31
Further Reading .....	31
<b>5 Segmentation</b> .....	<b>33</b>
Accessing Memory .....	33
The Global Descriptor Table (GDT) .....	35
Loading the GDT.....	36
Further Reading .....	37
<b>6 Interrupts and Input</b> .....	<b>39</b>
Interrupts Handlers.....	39
Creating an Entry in the IDT .....	39
Handling an Interrupt .....	40
Creating a Generic Interrupt Handler .....	41

Loading the IDT .....	43
Programmable Interrupt Controller (PIC).....	43
Reading Input from the Keyboard .....	44
Further Reading .....	45
<b>7 The Road to User Mode .....</b>	<b>47</b>
Loading an External Program .....	47
GRUB Modules .....	47
Executing a Program.....	48
A Very Simple Program .....	48
Compiling .....	48
Finding the Program in Memory.....	49
Jumping to the Code .....	49
The Beginning of User Mode .....	49
<b>8 A Short Introduction to Virtual Memory .....</b>	<b>51</b>
Virtual Memory Through Segmentation? .....	51
Further Reading .....	51
<b>9 Paging .....</b>	<b>53</b>
Why Paging? .....	53
Paging in x86.....	53
Identity Paging .....	55
Enabling Paging.....	55
A Few Details .....	55
Paging and the Kernel .....	55
Reasons to Not Identity Map the Kernel .....	56
The Virtual Address for the Kernel .....	56
Placing the Kernel at 0xC0000000.....	56
Higher-half Linker Script .....	57
Entering the Higher Half .....	57
Running in the Higher Half .....	58
Virtual Memory Through Paging .....	58

Further Reading .....	59
<b>10 Page Frame Allocation</b> .....	<b>61</b>
Managing Available Memory .....	61
How Much Memory is There? .....	61
Managing Available Memory .....	63
How Can We Access a Page Frame? .....	63
A Kernel Heap .....	63
Further reading .....	63
<b>11 User Mode</b> .....	<b>65</b>
Segments for User Mode .....	65
Setting Up For User Mode .....	65
Entering User Mode .....	66
Using C for User Mode Programs .....	67
A C Library .....	68
Further Reading .....	68
<b>12 File Systems</b> .....	<b>69</b>
Why a File System? .....	69
A Simple Read-Only File System .....	69
Inodes and Writable File Systems .....	70
A Virtual File System .....	70
Further Reading .....	70
<b>13 System Calls</b> .....	<b>71</b>
Designing System Calls .....	71
Implementing System Calls .....	71
Further Reading .....	72
<b>14 Multitasking</b> .....	<b>73</b>
Creating New Processes .....	73
Cooperative Scheduling with Yielding .....	73
Preemptive Scheduling with Interrupts .....	74
Programmable Interval Timer .....	74
Separate Kernel Stacks for Processes .....	74
Difficulties with Preemptive Scheduling .....	75
Further Reading .....	75



# 第1章

## はじめに

このテキストは、独自のx86オペレーティングシステムを作成するための実用的なガイドです。技術的な詳細については十分なサポートを提供する一方で、サンプルやコードの抜粋によって多くを明らかにしないように設計されています。私たちは、ウェブやその他の方法で入手可能な膨大な（そしてしばしば素晴らしい）資料やチュートリアルの一部を集め、私たちが遭遇した問題や苦労した点について私たち自身の洞察を加えることを試みました。

この本は、オペレーティングシステムの理論や、特定のオペレーティングシステム（OS）がどのように動作するかについて書かれたものではありません。OSの理論については、Andrew Tanenbaum著の「Modern Operating Systems」をお勧めします[1]。現在のオペレーティングシステムの一覧や詳細は、インターネットで入手できます。

序盤の章では、すぐにコーディングができるように、かなり詳細かつ明確に説明しています。後半の章では、必要なことの概要を説明していますが、実装や設計の多くは、カーネル開発の世界に慣れている読者に任されています。いくつかの章の終わりには、興味深く、取り上げたトピックをより深く理解できるような、さらなる読み物へのリンクがあります。

第2章と第3章では、開発環境を整え、仮想マシンでOSカーネルを起動し、最終的にC言語でコードを書き始めます。第4章では、画面やシリアルポートへの書き込みを行い、第5章ではセグメンテーション、第6章では割り込みや入力にまで踏み込んでいきます。

この後、機能的ではあるが、素の状態のOSカーネルができあがります。第7章では、ページングによる仮想メモリ（第8章と第9章）、メモリの割り当て（第10章）、そして第11章ではユーザーアプリケーションの実行と、ユーザーモードのアプリケーションへの道が始まります。

最後の3章では、ファイルシステム（第12章）、システムコール（第13章）、マルチタスク（第14章）といった、より高度なトピックについて説明します。

## About the Book

OSカーネルと本書は、ストックホルムの王立工科大学[2]の上級個人コースの一環として制作されました。



著者はそれまでにOSの理論に関するコースを受講していましたが、OSカーネルの開発についてはわずかな実務経験しかありませんでした。以前のOSコースで学んだ理論が実際にどのように機能するのかをより深く理解するために、著者は小さなOSの開発に焦点を当てた新しいコースを作ることになりました。このコースのもう一つの目標は、小さなOSを基本的にゼロから開発する方法についての完全なチュートリアルを書くことであり、この短い本はその結果である。

x86アーキテクチャは、最も一般的なハードウェア・アーキテクチャの一つであり、昔からそうでした。OSのターゲットとしてx86アーキテクチャを採用することは難しい選択ではありませんでしたが、その理由は、大規模なコミュニティがあり、広範な

参考資料と成熟したエミュレータ 私たちが作業しなければならなかったハードウェアの詳細にまつわる文書や情報は、アーキテクチャの古さにもかかわらず（あるいはそのせいかもしれませんが）、必ずしも簡単に見つかり理解したりできるものではありませんでした。

OSの開発は約6週間のフルタイムで行われました。実装は小さなステップをいくつも繰り返し、各ステップの後には手動でOSのテストを行いました。このように段階的かつ反復的に開発を進めることで、コードのごく一部しか変更されていないため、バグが発生しても簡単に見つけることができました。読者の皆様にも同様の方法で作業を行っていただきたいと思います。

6週間の開発期間中、ほぼすべてのコードを著者が一緒に書きました（この作業方法はペアプログラミングとも呼ばれます）。このような開発スタイルにより、多くのバグを回避できたと考えていますが、科学的に証明することは困難です。

## The Reader

この本の読者は、UNIX/Linux、システムプログラミング、C言語、コンピュータシステム全般（16進法[3]など）に慣れている必要があります。この本は、それらの学習を始めるための手段になるかもしれませんが、より難しくなるでしょうし、オペレーティングシステムの開発は、それだけでもすでに困難です。行き詰まったときには、検索エンジンや他のチュートリアルが役に立つことが多いです。

## Credits, Thanks and Acknowledgements

OSDevコミュニティ[4]の素晴らしいWikiと親切なメンバーに感謝します。また、James Malloy氏の著名なカーネル開発チュートリアル[5]にも感謝します。また、指導教官のTorbjörn Granlund氏には、洞察に満ちた質問と興味深い議論をしていただきました。

この本のCSSフォーマットのほとんどは、Scott Chacon氏が『Pro Git, <http://progit.org/>』で行った作業に基づいています。

## Contributors

私たちは、人々が私たちに送ってくれるパッチにとっても感謝しています。以下のユーザーの皆さんが、この本に貢献してくださいました。

- alexschneider
- Avidanborisov
- nirs
- kedarmhaswade
- vamaea
- ansjob

## Changes and Corrections

この本はGithubでホストされています。もし提案、コメント、修正があれば、この本をフォークし、変更点を書き込んで、プルリクエストを送ってください。本書をより良いものにするために、私たちは喜ん

で何でも取り入れます。

## Issues and where to get help

本を読む際に問題が発生した場合は、Github上のissuesを確認してください。 <https://github.com/littleosbook/littleosbook/issues>.

## License

すべてのコンテンツは、Creative Commons Attribution Non Commercial Share Alike 3.0 license, <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> に基づいています。

コードサンプルはパブリックドメインですので、好きなようにお使いください。この本への言及は常に暖かく迎えられます。



## 第2章

# ファーストステップ

オペレーティング・システム（OS）の開発は簡単な作業ではありません。「この問題を解決するにはどうしたらよいのでしょうか」という質問は、プロジェクトの進行中にさまざまな問題に対して何度も出てくることでしょう。この章では、開発環境を整え、非常に小さな（そして原始的な）OSを起動することができます。

## Tools

### Quick Setup

私たち（著者）は、OS開発を行うためのOSとしてUbuntu [6]を使用し、物理的にも仮想的（仮想マシンVirtualBox [7]を使用）にも実行しています。これらのツールが本書で提供されているサンプルで動作することがわかっているため、すべてを起動して実行するための手っ取り早い方法は、私たちが行ったのと同じセットアップを使用することです。

物理的にも仮想的にもUbuntuがインストールされたら、apt-getを使って以下のパッケージをインストールしてください。

### Programming Languages

OSの開発には、C言語[8][9]を用い、GCC[10]を使用します。C言語を使うのは、OSを開発するには、生成されるコードを非常に精密に制御し、メモリに直接アクセスする必要があるからです。同様の機能を持つ他の言語を使用することもできますが、本書ではC言語のみを取り上げます。

このコードでは、GCCに特有の1つのタイプ属性を利用します。

属性 ((パック))

この属性を使用すると、コードで定義したとおりにコンパイラが構造体のメモリレイアウトを使用することができます。これについては、次の章で詳しく説明します。

この属性のため、GCC以外のCコンパイラではサンプルコードをコンパイルするのが難しいかもしれません。

アセンブリコードの記述には、アセンブラとしてNASM[11]を選択しました。これは、GNU AssemblerよりもNASMの構文の方が好きだからです。

この本ではスクリプト言語としてBash [12]を使用します。

## Host Operating System

すべてのコード例は、UNIX系OSでコンパイルされていることを前提としています。すべてのコード例は、Ubuntu [6]のバージョン11.04および11.10を使用してコンパイルに成功しています。

## Build System

Makefileの例を作成する際にはMake [13]を使用しています。

## Virtual Machine

OSを開発するには、物理的なコンピュータではなく、仮想マシン上でコードを実行できると非常に便利です。仮想マシン上でOSを起動する方が、OSを物理的な媒体にインストールしてから物理的なマシン上で実行するよりもはるかに速いからです。Bochs [14]はx86(IA-32)プラットフォーム用のエミュレータで、デバッグ機能を備えているため、OSの開発に適しています。他にもQEMU[15]やVirtualBox[7]がよく使われています。本書ではBochsを使用しています。

仮想マシンを使用すると、OSが実際の物理的なハードウェア上で動作することを保証できません。しかし、仮想マシンの環境は物理的なものと非常によく似ており、実行ファイルをCDにコピーして適当なマシンを探すだけで、OSをテストすることができます。

## Booting

オペレーティングシステムの起動は、小さなプログラムの連鎖に沿って制御を移すことで構成されています。それぞれのプログラムは前のプログラムよりも「強力」で、オペレーティングシステムは最後の「プログラム」です。次の図は、ブートプロセスの例です。

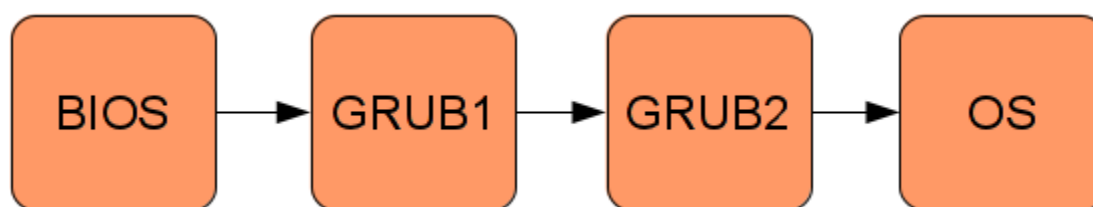


図2.1: ブートプロセスの一例。各ボックスはプログラムです。

## BIOS

PCの電源を入れると、コンピュータはBIOS (Basic Input Output System) [16]規格に準拠した小さなプログラムを起動します。このプログラムは通常、PCのマザーボード上の読み取り専用メモリチップに格納されている。BIOSプログラムの本来の役割は、画面への印刷やキーボード入力の読み取りなど、いくつかのライブラリ機能をエクスポートすることでした。最近のOSでは、BIOSの機能は使われておらず、



以下のようになっています。

BIOSは、ハードウェアと直接対話するドライバーで、BIOSをバイパスします。現在、BIOSは主に初期診断（パワーオンセルフテスト）を行い、その後、ブートローダーに制御を移す。

## The Bootloader

BIOSプログラムは、PCの制御をブートローダと呼ばれるプログラムに移します。ブートローダの仕事は、OS開発者である私たちに制御権を移し、私たちのコードを入力することです。しかし、ハードウェアの制約1や下位互換性のために、ブートローダは2つの部分に分かれていることが多く、ブートローダの第1の部分が第2の部分に制御を移し、最終的にOSにPCの制御を委ねることになります。

ブートローダを書くには、BIOS と連動する多くの低レベルコードを書く必要があります。そのため、既存のブートローダである GNU GRand Unified Bootloader (GRUB) [17]を使用します。

GRUBを使用すると、オペレーティングシステムを通常のELF[18]実行ファイルとして構築することができます、GRUBによって正しいメモリ位置にロードされます。カーネルのコンパイルには、コードを特定の方法でメモリ内に配置する必要があります（カーネルのコンパイル方法については、本章で後述します）。

## The Operating System

GRUBは、メモリ上のある位置にジャンプすることで、制御をOSに移します。ジャンプの前に、GRUBは、ランダムなコードではなく実際にOSにジャンプしていることを確認するために、マジックナンバーを探します。このマジックナンバーは、GRUBが準拠しているマルチブート仕様[19]の一部です。GRUBがジャンプすると、OSはコンピュータを完全に制御できるようになります。

## Hello Cafebabe

このセクションでは、GRUBと組み合わせて使用できる最小のOSの実装方法を説明します。このOSが行うことは、eaxレジスタに0xCAFE BABEを書き込むことです（ほとんどの人は、これをOSとは呼ばないでしょう）。

## Compiling the Operating System

C言語にはスタックがありませんので、この部分はアセンブリコードで記述する必要があります（スタックの設定方法は「C言語入門」の章で説明しています）。以下のコードを loader.s というファイルに保存します。

```
global loader                ; the entry symbol for ELF

MAGIC_NUMBER equ 0x1BADB002 ; define the magic number constant
FLAGS         equ 0x0        ; multiboot flags
CHECKSUM      equ -MAGIC_NUMBER ; calculate the checksum

; (マジックナンバー+チェックサム+フラグが0になるはず)
```

```
section .text:           ; start of the text (code) section
align 4                 ; the code must be 4 byte aligned
    dd MAGIC_NUMBER      ; write the magic number to the machine code,
    dd FLAGS             ; the flags,
```

---

1ブートローダは、512バイトしかないハードディスクのMBR（マスターブートレコード）ブートセクタに収まらなければなりません。

```

        dd CHECKSUM                ; and the checksum

loader:                ; the loader label (defined as entry point in linker script)
        mov eax, 0xCAFEBAE        ; place the number 0xCAFEBAE in the register eax
        .ループです。
        jmp .loop                ; loop forever

```

このOSが行うことは、eaxレジスタに0xCAFEBAEという非常に特殊な数字を書き込むことです。これは非常にOSが0xCAFEBAEという数字をeaxレジスタに入れていなければ、0xCAFEBAEという数字がeaxレジスタに入っている可能性はありません。

loader.sというファイルは、次のコマンドで32ビットのELF[18]オブジェクトファイルにコンパイルできます。

```
nasm -f elf32 loader.s
```

## Linking the Kernel

このコードをリンクして実行ファイルを作成する必要がありますが、一般的なプログラムをリンクする場合と比較して、若干の検討が必要です。GRUBは0x00100000（1メガバイト（MB））以上のメモリアドレスにカーネルをロードしたいのですが、1MB以下のアドレスはGRUB自身やBIOS、メモリマップドI/Oで使用されるためです。そのため、以下のようなリンカースクリプトが必要となります（GNU LD[20]用に作成）。

```

ENTRY(loader)          /* the name of the entry label */

SECTIONS {
    . = 0x00100000;      /* the code should be loaded at 1 MB */

    .text ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.text)         /* all text sections from all files */
    }

    .rodata ALIGN (0x1000) : /* 4KBでのアラインメント */。
    {
        *(.rodata*)      /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.data)         /* all data sections from all files */
    }

    .bss ALIGN (0x1000) : /* align at 4 KB */
    {
        *(COMMON)        /* all COMMON sections from all files */
    }
}

```

```
        *(.bss)                /* all bss sections from all files */  
    }  
}
```

リンカースクリプトをlink.ldというファイルに保存します。これで、実行ファイルを次のコマンドでリンクすることができます。

```
ld -T link.ld -melf_i386 loader.o -o kernel.elf
```

最終的な実行ファイルの名前はkernel.elfとなります。

## Obtaining GRUB

GRUB Legacy と GRUB 2 の両方を使用しているシステムで OS の ISO イメージを作成することができるため、使用する GRUB のバージョンは GRUB Legacy とします。このファイルは GRUB 0.97 から ftp でソースをダウンロードしてビルドできます。

[//alpha.gnu.org/gnu/grub/grub-0.97.tar.gz](http://alpha.gnu.org/gnu/grub/grub-0.97.tar.gz). ただし、configureスクリプトはUbuntu[21]ではうまく動作しないので、バイナリファイルは[http://littleosbook.github.com/files/stage2\\_eltorito](http://littleosbook.github.com/files/stage2_eltorito) からダウンロードしてください。ファイル stage2\_eltorito を、すでに loader.s と link.ld が入っているフォルダにコピーします。

## Building an ISO Image

実行ファイルは、仮想マシンや物理マシンで読み込み可能なメディアに配置する必要があります。本書では、ISO [22] イメージファイルをメディアとして使用しますが、仮想マシンや物理マシンがサポートするものに依じて、フロッピーイメージを使用することもできます。

ここではgenisoimageというプログラムを使ってカーネルのISOイメージを作成します。まず、ISOイメージに含まれるファイルを含むフォルダを作成する必要があります。以下のコマンドは、フォルダを作成し、ファイルを適切な場所にコピーします。

```
mkdir -p iso/boot/grub      # create the folder structure
cp stage2_eltorito iso/boot/grub/ # copy the bootloader
cp kernel.elf iso/boot/      # copy the kernel
```

GRUB用の設定ファイルmenu.lstを作成する必要があります。このファイルは、GRUBにカーネルの場所を伝え、いくつかのオプションを設定します。

```
default=0
timeout=
0
タイトルOS
カーネル /boot/kernel.elf
```

ファイルmenu.lstをiso/boot/grub/のフォルダに入れます。これで、isoフォルダの中身が下図のようになるはずです。

```
iso
|-- ブート
|-- grub
```

```
|-- menu.lst  
|-- stage2_eltorito  
|-- kernel.elf
```

そして、以下のコマンドでISOイメージを生成することができます。

```

genisoimage -R \
             -b boot/grub/stage2_eltorito \
             -no-emul-boot \
             -boot-load-size 4 \
             -A os \
             -input-charset utf8 \
             -quiet \
             -boot-info-table \
             -o os.iso \
             iso

```

コマンドで使用されるフラグの詳細については、genisoimageのマニュアルを参照してください。

ISOイメージos.isoには、カーネル実行ファイル、GRUBブートローダ、設定ファイルが含まれています。

## Running Bochs

これで、os.isoのISOイメージを使って、BochsエミュレータでOSを動かすことができます。Bochsの起動には設定ファイルが必要で、簡単な設定ファイルの例を以下に示します。

```

megs:          32
ディスプレイライブラリ : sdl
romimage:      file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:   file=/usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:   type=cdrom, path=os.iso, status=inserted
boot:          cdrom
log:           bochslog.txt
clock:         sync=realtime, time0=local
cpu:           count=1, ips=1000000

```

Bochsのインストール方法によっては、romimageとvgaromimageのパスを変更する必要があるかもしれません。Bochsのコンフィグファイルについての詳細は、Bochのウェブサイト[23]にあります。

設定をbochsrc.txtというファイルに保存しておけば、以下のコマンドでBochsを実行できます。

```
bochs -f bochsrc.txt -q
```

フラグ -f は与えられた設定ファイルを使用するように、フラグ -q は対話式スタートメニューをスキップするように指示します。これでBochsが起動し、GRUBからの情報を含むコンソールが表示されるはずです。

Bochsを終了した後、Bochが作成したログを表示します。

```
cat bochslog.txt
```



BochsがシミュレートしたCPUのレジスタの内容が、出力のどこかに表示されているはずです。  
RAX=00000000CAFEBABEまたはEAX=CAFEBABE（Bochsが64ビットサポート付きかなしによって異なる）が出力されていれば、あなたのOSは正常に起動したことになります。

## Further Reading

Gustavo Duertes氏は、x86コンピュータの起動時に実際に何が起こるかについて、詳細な記事を書いています。 <http://duartes.org/gustavo/blog/post/how-computers-boot-up>

グスタボは、<http://duartes.org/gustavo/blog/post/kernel-boot-process>で、カーネルが非常に初期の段階で何をしているかを説明し続けています。

OSDevのwikiにはx86コンピュータの起動に関する素晴らしい記事があります：

[http://wiki.osdev.org/Boot\\_Sequence](http://wiki.osdev.org/Boot_Sequence)



## 第3章

# Cへの道

この章では、OSのプログラミング言語として、アセンブリコードの代わりにC言語を使用する方法をご紹介します。アセンブリは、CPUとのやりとり非常に適しており、コードのあらゆる面を最大限にコントロールすることができます。しかし、少なくとも筆者らにとっては、C言語の方がはるかに使いやすい言語である。したがって、できるだけC言語を使用し、アセンブリコードは意味のあるところだけ使用したいと考えています。

### Setting Up a Stack

自明でない C プログラムはすべてスタックを使用するため、C を使用するための前提条件の 1 つとして、スタックがあります。スタックの設定は、正しくアラインされたフリーメモリの領域（x86では、スタックは低アドレスに向かって成長することを覚えておいてください）の最後をespレジスタが指すようにすること以上に難しいことはありません（パフォーマンスの観点からは、4バイトのアラインメントが推奨されます）。

これまでのところ、メモリ内にあるのはGRUB、BIOS、OSカーネル、そしていくつかのメモリマップドI/O だけなので、espをメモリ内のランダムな領域に向けることができます。これは良いアイデアではありません。利用可能なメモリの量や、espが指す領域が他のものに使用されているかどうかはわかりません。より良いアイデアは、カーネルのELFファイルのbssセクションに初期化されていないメモリの一部を確保することです。OSの実行ファイルのサイズを小さくするためには、データセクションの代わりにbssセクションを使用するのが良いでしょう。GRUBはELFを理解しているので、OSのロード時にbssセクションに確保されたメモリを割り当てます。

初期化されていないデータを宣言するには、NASMの疑似命令resb[24]を使うことができます。

```
KERNEL_STACK_SIZE equ 4096                ; size of stack in bytes

section .bss
align 4                                     ; align at 4 bytes
kernel_stack:                             ; label points to beginning of memory
    resb KERNEL_STACK_SIZE                ; reserve stack for the kernel
```

スタックに初期化されていないメモリを使用することを心配する必要はありません。なぜなら、書き込まれていないスタックの位置を読むことは（手動でポインタをいじらなければ）できないからです。正しい）プログラムは、最初に要素をスタックにプッシュしなければ、スタックから要素をポップ することはできません。したがって、スタックのメモリロケーションは、常に読み込まれる前に書き込まれることになります。

続いて、kernel\_stackメモリの末尾にespを指すことで、スタックポインタが設定されます。

```
mov esp, kernel_stack+ KERNEL_STACK_SIZE ; point esp to the start of the  
スタック（メモリ領域の終端）
```

## Calling C Code From Assembly

次のステップは、アセンブリコードからC言語の関数を呼び出すことです。アセンブリコードからCコードを呼び出す方法には、さまざまな規約があります[25]。本書では、GCCで使用されているcdecl呼び出し規則を使用しています。cdecl呼び出し規則では、関数の引数はスタック（x86の場合）を介して渡されるべきだとしています。関数の引数は、右から左の順にスタックに置かれます（つまり、一番右の引数を最初に押します）。関数の戻り値は、eax レジスタに格納されます。次のコードはその例です。

```
/* C言語の関数 */
int sum_of_three(int arg1, int arg2, int arg3)
{
    arg1 + arg2 + arg3 を返します。
}

アセンブリコード
external sum_of_three ; the function sum_of_three is defined elsewhere

push dword 3          ; arg3
push dword 2          ; arg2
push dword 1          ; arg1
call sum_of_three      ; call the function, the result will be in eax
```

## Packing Structs

本書では、「コンフィギュレーション・バイト」と呼ばれる、特定の順序でビットを集めたものをよく目にします。以下に、32ビットの例を示します。

Bit:	31	24   23	8   7	0
Content:	index	address	config	

このような構成を扱うのに、符号なしの整数である unsigned int を使用する代わりに、「パックされた構造体」を使用する方がはるかに便利です。

```
構造体example {
    unsigned char config; /* bit 0 - 7 */
    unsigned short address; /* bit 8 - 23 */
    unsigned char index; /* bit 24 - 31 */
};
```

前述の例で構造体を使用する場合、構造体のサイズが正確に32ビットであることは保証されていません。コンパイラは、要素間にパディングを追加することができますが、これにはさまざまな理由があります。例えば、要素へのアクセスを高速化するため、またはハードウェアやコンパイラの要求に応じてパディングを追加するためです。コンフィギュレーション・バイトを表すために構造体を使用する場合は、コンパイラがパディングを追加しないことが非常に重要です。packed 属性を使用すると、GCC がパディングを追加しないようにすることができます。

構造体example {。  
    unsigned char config; /\* bit 0 - 7 \*/

```

        unsigned short address; /* bit 8 - 23 */
        unsigned char index;    /* bit 24 - 31 */
    }属性（（パック））。

```

なお、属性((packed))はC言語の標準ではないため、すべてのCコンパイラで動作するわけではありません。

## Compiling C Code

OS用のCコードをコンパイルする際には、GCCの多くのフラグを使用する必要があります。これは、Cコードが標準ライブラリの存在を仮定してはならないからです。私たちのOSには利用可能な標準ライブラリがないからです。フラグの詳細については、GCCのマニュアルを参照してください。

Cコードのコンパイルに使用されるフラグは以下の通りです。

```

-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles
-nodfaultlibs

```

Cプログラムを書くときは、いつものようにすべての警告をオンにし、警告をエラーとして扱うことをお勧めします。

```

-壁 -エクストラ -エラー


```

kmain.cというファイルに関数kmainを作成して、loader.sから呼び出すことができるようになりました。kmainはおそらく引数を必要としないでしょう（しかし後の章では必要になります）。

## Build Tools

また、OSのコンパイルやテストランを容易にするために、いくつかのビルドツールを設定するのも良い時期だと思います。ここではmake [13]を使うことをお勧めしますが、他にも多くのビルドシステムがあります。OS用のシンプルなMakefileは次のような例になります。

```

OBJECTS = loader.o kmain.o
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector 。
-nostartfiles -nodfaultlibs -Wall -Wextra -Werror -c LDFLAGS =
-T link.ld -melf_i386

AS = nasm

ASFLAGS = -f elf
all: kernel.elf
kernel.elf:
$(OBJECTS)

ld $(LDFLAGS) $(OBJECTS) -o kernel.elf
os.iso: kernel.elf
cp kernel.elf iso/boot/kernel.elf

```



```
genisoimage -R \
            -b boot/grub/stage2_eltorito \
            -no-emul-boot \
```

```

        -boot-load-size 4          \
        -A os                      \
        -input-charset utf8       \
        -quiet                     \
        -boot-info-table          \
        -o os.iso                 \
        iso
run: os.iso

        bochs -f bochsrc.txt -q
%.o: %.c
        $(CC) $(CFLAGS) $< -o $@
%.o: %.s

        $(AS) $(ASFLAGS) $< -o $@.
綺麗になりました。
        rm -rf *.o kernel.elf os.iso

```

作業ディレクトリの内容が下図のようになっているはずです。

```

-
|-- bochsrc.txt
|-- iso
|   |-- boot
|       |-- grub
|           |-- menu.lst
|               |-- stage2_eltorito
|-- kmain.c
|-- loader.s
|-- Makefile

```

これで、`make run`という簡単なコマンドでOSを起動することができるようになりました。これでカーネルがコンパイルされ、（上のMakefileで定義したように）Bochsで起動するようになります。

## Further Reading

Kernigan & Richieの著書『The C Programming Language, Second Edition』[8]は、C言語のあらゆる側面を学ぶのに最適です。

## 第4章

# 出力

この章では、コンソールにテキストを表示する方法と、シリアルポートにデータを書き込む方法を紹介합니다。さらに、最初のドライバを作成します。ドライバとは、カーネルとハードウェアの間の層として機能し、ハードウェアと直接通信するよりも高い抽象度を提供するコードのことです。この章の最初のパートでは、コンソールにテキストを表示できるようにするためのフレームバッファ[26]のドライバを作成します。第2部では、シリアルポート用のドライバを作成する方法を紹介します。Bochsは、シリアルポートからの出力をファイルに保存することができ、OSのロギングメカニズムを効果的に作ることができます。

## Interacting with the Hardware

ハードウェアとのやりとりには、通常、メモリーマップドI/OとI/Oポートの2種類の方法があります。

ハードウェアがメモリーマップドI/Oを使用している場合は、特定のメモリアドレスに書き込むことができ、ハードウェアは新しいデータで更新されます。この例としては、フレームバッファがあります。例えば、0x000B8000番地に0x410Fという値を書き込むと、黒地に白でAという文字が表示されます（詳細はフレームバッファの項を参照）。

ハードウェアがI/Oポートを使用している場合、ハードウェアとの通信にはアセンブリコード命令outとinを使用する必要があります。out命令は、I/Oポートのアドレスと送信するデータの2つのパラメータを受け取ります。in命令は、I/Oポートのアドレスという1つのパラメータを受け取り、ハードウェアからデータを返します。I/Oポートは、ソケットを使ってサーバーと通信するのと同じように、ハードウェアと通信していると考えることができます。フレームバッファのカーソル（点滅している長方形）は、PCのI/Oポートで制御されるハードウェアの一例です。

## The Framebuffer

フレームバッファは、メモリのバッファを画面に表示する機能を持つハードウェアデバイスである[26]。フレームバッファは80列25行で、行と列のインデックスは0から始まる（したがって、行は0～24と表示

される)。

### **Writing Text**

フレームバッファ経由でコンソールにテキストを書き込むには、メモリマップドI/Oを使用します。フレームバッファのメモリマップドI/Oの開始アドレスは0x000B8000である[27]。メモリは16ビットのセルに分割されています。

ここで、16ビットは、文字、前景色、背景色の両方を決定します。次の図のように、最上位8ビットが文字のASCII[28]値、ビット7～4が背景、ビット3～0が前景となります。

ビット : | 15 14 13 12 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | | 。  
Content: | ASCII | FG | BG |

用意されているカラーは以下の表の通りです。

Color	Value	Color	Value	Color	Value	Color	Value
Black	0	Red	4	Dark grey	8	Light red	12
Blue	1	Magenta	5	Light blue	9	Light magenta	13
Green	2	Brown	6	Light green	10	Light brown	14
Cyan	3	Light grey	7	Light cyan	11	White	15

最初のセルはコンソールのゼロ行、ゼロ列に相当します。ASCIIテーブルを使うと、Aは65または0x41に相当することがわかります。したがって、緑の前景(2)と濃い灰色の背景(8)を持つ文字Aを(0,0)の位置に書き込むには、次のようなアセンブリコード命令を使用します。

```
mov [0x000B8000], 0x4128
```

2つ目のセルは0行1列目に対応し、そのアドレスは次のようになります。

```
0x000B8000 + 16 = 0x000B8010
```

フレームバッファへの書き込みは、C言語でもアドレス0x000B8000をcharポインタとして扱い、char \*fb = (char \*) 0x000B8000とすることで可能です。そして、緑の前景と濃いグレーの背景の場所(0,0)にAを書き込むと次のようになります。

```
fb[0] = 'A';  
fb[1] = 0x28;
```

次のコードでは、これを関数にまとめています。

```
* /** fb_write_cell:  
*   Writes a character with the given foreground and background to position i  
*   in the framebuffer.  
*  
*   @param i The location in the framebuffer  
*   @param c The character  
*   @param fg The foreground color  
*   @param bg The background color  
*/  
void fb_write_cell(unsigned int i, char c, unsigned char fg, unsigned char bg)  
{
```

```
    fb[i] = c.  
    fb[i + 1] = ((fg & 0x0F) << 4) | (bg & 0x0F)  
}
```

この関数は、次のように使用できます。

```
#define FB_GREEN      2

#define FB_DARK_GREY 8

fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
```

## Moving the Cursor

フレームバッファのカーソルの移動は、2つの異なるI/Oポートを介して行われます。カーソルの位置は16ビットの整数で決定されます。0は0行目、0列目、1は0行目、1列目、80は1行目、0列目、といった具合です。0は0行目、1は0列目、80は1行目、1列目、0列目というように、16ビットの整数で位置を決定しますが、アウトアセンブリコードの命令の引数は8ビットなので、最初の8ビットと次の8ビットを2回に分けて送信する必要があります。フレームバッファには2つのI/Oポートがあり、1つはデータを受け入れるため、もう1つは受信したデータを記述するためです。ポート0x3D4[29]はデータを記述するポートで、ポート0x3D5[29]はデータそのもののポートです。

カーソルを1行0列目（ポジション80=0x0050）に設定するには、次のようなアセンブリコード命令を使用します。

```
out 0x3D4, 14      ; 14 tells the framebuffer to expect the highest 8 bits of the position
out 0x3D5, 0x00    ; sending the highest 8 bits of 0x0050
out 0x3D4, 15      ; 15 tells the framebuffer to expect the lowest 8 bits of the position
out 0x3D5, 0x50    ; sending the lowest 8 bits of 0x0050
```

アセンブリコードのout命令は、C言語では直接実行できません。そのため、outをアセンブリコードの関数にラップして、cdecl呼び出し規格を介してC言語からアクセスできるようにするのがよいでしょう[25]。

```
global outb          ; make the label outb visible outside this file
; outb - I/Oポートにバイトを送る
; スタック。[esp + 8] データバイト
;         [esp + 4] the I/O port
;         [esp    ] return address
outb:
    mov al, [esp + 8] ; move the data to be sent into the al register
    mov dx, [esp + 4] ; move the address of the I/O port into the dx register
    out dx, al        ; send the data to the I/O port
    ret               ; return to the calling function
```

この関数をio.sというファイルに格納し、io.hというヘッダーを作成することで、C言語からアウトのアセンブリコード命令を便利に利用することができます。

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H
```

```
* /** アウトプです。  
*  Sends the given data to the given I/O port. Defined in io.s  
*  
*  @param port The I/O port to send the data to  
*  @param data The data to send to the I/O port
```



```

*/

void outb(unsigned short port, unsigned char data);
#endif /* INCLUDE_IO_H */

```

カーソルの移動は、C言語の関数でラップできるようになりました。

```

#include "io.h"
I/Oポート /* I/Oポート
#define FB_COMMAND_PORT      0x3D4
#define FB_DATA_PORT         0x3D5

/* The I/O port commands */
#define FB_HIGH_BYTE_COMMAND  14
#define FB_LOW_BYTE_COMMAND   15
* /** fb_move_cursor:
*   * Moves the cursor of the framebuffer to the given position
*   *
*   * @param pos The new position of the cursor
*   */
void fb_move_cursor(unsigned short pos)
{
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT, ((pos >> 8) & 0x00FF));
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT, pos & 0x00FF);
}

```

## The Driver

ドライバは、OS内の他のコードがフレームバッファとやりとりするために使用するインターフェースを提供する必要があります。インターフェイスが提供すべき機能に正解はありませんが、提案としては、以下のような宣言をしたwrite関数を用意することです。

```
int write(char *buf, unsigned int len);
```

書き込み関数は、長さ len のバッファ buf の内容を画面に書き込む。書き込み関数は、文字が書き込まれた後、自動的にカーソルを進め、必要に応じて画面をスクロールさせる。

## The Serial Ports

シリアルポート[30]は、ハードウェアデバイス間で通信するためのインターフェースで、ほとんどすべてのマザーボードに搭載されていますが、最近ではDE-9コネクタの形でユーザーの目に触れることはほとんどありません。シリアルポートは使いやすく、さらに重要なのは、Bochsのロギングユーティリティとして使用できることです。コンピュータがシリアルポートをサポートしている場合、通常は複数のシリアルポートをサポートしていますが、私たちはそのうちの1つのポートだけを利用します。これは、シ

リアルポートをロギングにのみ使用するためです。さらに、シリアルポートは入力ではなく出力にのみ使用します。シリアルポートはI/Oポートで完全に制御されています。

## Configuring the Serial Port

シリアルポートに送るべき最初のデータは、設定データです。2つのハードウェアデバイスが相互に通信するためには、いくつかの事項に同意する必要があります。これらの事柄には次のようなものがあります。

- The speed used for sending data (bit or baud rate)
- If any error checking should be used for the data (parity bit, stop bits)
- The number of bits that represent a unit of data (data bits)

## Configuring the Line

回線の設定とは、回線上でどのようにデータを送信するかを設定することです。シリアルポートにはラインコマンドポートというI/Oポートがあり、これを使って設定を行います。

まず、データを送信する際の速度を設定します。シリアルポートは115200Hzの内部クロックを持っています。速度を設定するということは、シリアルポートに割り算を送るということで、例えば2を送ると  $115200 / 2 = 57600$  Hzの速度になります。

除算器は16ビットの数値ですが、一度に送れるのは8ビットだけです。そこで、まず上位8ビット、次に下位8ビットを期待するようシリアルポートに指示を出す必要があります。これは、ラインコマンドポートに0x80を送信することで行います。その例を以下に示します。

```
#include "io.h" /* io.hは "カーソルの移動" のセクションで実装されています。
```

```
I/Oポート */ /* I/Oポート
```

```
* /* 全てのI/Oポートは、データポートを基準に計算されます。これは
* all serial ports (COM1, COM2, COM3, COM4) have their ports in the same
* order, but they start at different values.
*/
```

```
#define SERIAL_COM1_BASE          0x3F8          /* COM1 base port */
```

```
#define SERIAL_DATA_PORT(base)    (base)
#define SERIAL_FIFO_COMMAND_PORT(base) (base + 2)
#define SERIAL_LINE_COMMAND_PORT(base) (base + 3)
#define SERIAL_MODEM_COMMAND_PORT(base) (base + 4)
#define SERIAL_LINE_STATUS_PORT(base) (base + 5)
```

```
/* I/Oポートのコマンド */
```

```
* /* serial_line_enable_dlab:
* Tells the serial port to expect first the highest 8 bits on the data port,
* then the lowest 8 bits will follow
*/
```

```
#define SERIAL_LINE_ENABLE_DLAB    0x80
```

```
* /** serial_configure_baud_rate:
* Sets the speed of the data being sent. The default speed of a serial
* port is 115200 bits/s. The argument is a divisor of that number, hence
* the resulting speed becomes (115200 / divisor) bits/s.
```

```

*
* @param com      The COM port to configure
* @param divisor  The divisor
*/
void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
{
    outb(SERIAL_LINE_COMMAND_PORT(com), SERIAL_LINE_ENABLE_DLAB);
    outb(SERIAL_DATA_PORT(com),
        (divisor >> 8) & 0x00FF)とな
        ります。
    outb(SERIAL_DATA_PORT(c
        om), divisor & 0x00FF);
}

```

データの送信方法を設定する必要があります。これは、ラインコマンドポートを介してバイトを送信することでも行われます。8ビットのレイアウトは以下のようになっています。

```

Bit:      | 7 | 6 | 5 4 3 | 2 | 1 0 |
Content: | d | b | prty | s | dl |

```

各名称の説明は以下の表（および[31]）に記載されています。

Name	Description
d	Enables (d = 1) or disables (d = 0) DLAB
b	If break control is enabled (b = 1) or disabled (b = 0)
prty	The number of parity bits to use
s	The number of stop bits to use (s = 0 equals 1, s = 1 equals 1.5 or 2)
dl	Describes the length of the data

ここでは、ほとんどの標準値である0x03 [31]を使用します。これは、8ビットの長さ、パリティビットなし、1ストップビット、ブレイクコントロール無効を意味します。これをラインコマンドポートに送信すると、次の例のようになります。

```

* /** serial_configure_line:
*   * Configures the line of the given serial port. The port is set to have a
*   * data length of 8 bits, no parity bits, one stop bit and break control
*   * disabled.
*   *
*   * @param com  The serial port to configure
*   */
void serial_configure_line(unsigned short com)
{
    /* Bit:      | 7 | 6 | 5 4 3 | 2 | 1 0 |
    * Content: | d | b | prty | s | dl |
    * Value:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
    */
}

```

```
    outb(SERIAL_LINE_COMMAND_PORT(com), 0x03)となります。  
}
```

OSDevの記事[31]には、値についてより詳細な説明があります。

## Configuring the Buffers

シリアルポートでデータを送信する際には、データの受信時と送信時の両方でバッファに入れられます。これにより、シリアルポートにデータを送信する際に、シリアルポートが有線で送信できる速度よりも速くデータを送信した場合、データはバッファされます。しかし、あまりにも速いスピードでデータを送りすぎると、バッファがいっぱいになり、データが失われてしまいます。つまり、バッファはFIFOキューなのである。FIFOキューの構成バイトは、次の図のようになります。

Bit:        | 7 6 | 5 | 4 | 3 | 2 | 1 | 0 |  
Content: | lvl | bs | r | dma | clt | clr | e |

各名称の説明は以下の表を参照してください。

Name	Description
lvl	How many bytes should be stored in the FIFO buffers
bs	If the buffers should be 16 or 64 bytes large
r	Reserved for future use
dma	How the serial port data should be accessed
clt	Clear the transmission FIFO buffer
clr	Clear the receiver FIFO buffer
e	If the FIFO buffer should be enabled or not

0xC7=11000111という値を使っています。

- Enables FIFO
- Clear both receiver and transmission FIFO queues
- Use 14 bytes as size of queue

シリアルプログラミングのWikiBook[32]では、この値についてさらに詳しく説明されています。

## Configuring the Modem

モデムコントロールレジスタは、RTS (Ready To Transmit) ピンとDTR (Data Terminal Ready) ピンを介した非常にシンプルなハードウェアフローコントロールに使用されます。シリアルポートを設定する際には、RTSとDTRを1にして、データを送信する準備ができていることを意味します。

モデムのコンフィグレーションバイトは、次の図のようになっています。

ビットです。        | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  
Content: | R | R | Af | Lb | Ao2 | Ao1 | RTS |  
DTR | など。

各名称の説明は以下の表を参照してください。

Name	Description
r	Reserved
af	Autoflow control enabled
lb	Loopback mode (used for debugging serial ports)
ao2	Auxiliary output 2, used for receiving interrupts
ao1	Auxiliary output 1
rts	Ready To Transmit
dtr	Data Terminal Ready

受信データを扱わないので、割り込みを有効にする必要はありません。そのため、設定値0x03 = 00000011 (RTS = 1、DTS = 1) を使用します。

## Writing Data to the Serial Port

シリアルポートへのデータの書き込みは、データI/Oポートを介して行います。ただし、書き込む前に、送信FIFOのキューが空になっている必要があります（以前の書き込みがすべて終了している必要があります）。ラインステータスI/Oポートのビット5が1の場合、送信FIFOキューは空になります。

I/Oポートの内容を読み出すには、inアセンブリコード命令を使用します。C言語からinアセンブリコード命令を使うことはできないので、outアセンブリコード命令と同じようにラップする必要があります。

グローバルインビー

; inb - 指定したI/Oポートからバイトを返します。  
; のスタックです。[esp + 4] I/Oポートのアドレス  
; [esp] The return address

inb:

mov dx, [esp + 4] ; move the address of the I/O port to the dx register  
in al, dx ; read a byte from the I/O port and store it in the al register  
ret ; return the read byte

```
/* in file io.h */
* /** inb:
*   * Read a byte from an I/O port.
*   *
*   * @param port The address of the I/O port
*   * @return      The read byte
*   */
unsigned char inb(unsigned short port)。
```

送信FIFOが空であるかどうかのチェックは、Cから行うことができます。

```
#include "io.h"
```



```

* /** serial_is_transmit_fifo_empty:
*  * Checks whether the transmit FIFO queue is empty or not for the given COM
*  * port.
*  *
*  * @param com The COM port
*  * @return 0 if the transmit FIFO queue is not empty
*  *         1 if the transmit FIFO queue is empty
*  */
int serial_is_transmit_fifo_empty(unsigned int com)
{
    /* 0x20 = 0010 0000 */
    return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}

```

シリアルポートへの書き込みとは、送信FIFOのキューが空にならない限りスピンして、データI/Oポートにデータを書き込むことです。

## Configuring Bochs

第1シリアルポートからの出力を保存するには、Bochsの設定ファイルbochsrc.txtを更新する必要があります。com1設定は、Bochsに第1シリアルポートの扱いを指示します。

```
com1: enabled=1, mode=file, dev=com1.out
```

これで、シリアルポート1からの出力がファイルcom1.outに格納されます。

## The Driver

フレームバッファ用ドライバの書き込み関数と同様に、シリアルポート用の書き込み関数を実装することをお勧めします。フレームバッファ用の書き込み関数との名前の衝突を避けるために、関数の名前をfb\_writeとserial\_writeにして区別するのが良いでしょう。

さらに、[8]の7.3節を参照して、printfのような関数を書いてみることをお勧めします。printf関数は、出力をどのデバイス（フレームバッファまたはシリアル）に書き込むかを決めるための追加の引数を取ることができます。

最後にお勧めしたいのは、ログメッセージの重要度を区別する方法を作ることです。例えば、メッセージの前にDEBUG、INFO、ERRORを付けるなどしてください。

## Further Reading

WikiBooksに掲載されている「Serial programming」という本には、シリアルポートのプログラミングに関する素晴らしいセクションがあります。  
[http://en.wikibooks.org/wiki/Serial\\_Programming/8250\\_UART\\_Programming#UART\\_Registers](http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming#UART_Registers)  
 OSDevのwikiには、シリアルポートに関する多くの情報が掲載されたページがあります。  
[http://wiki.osdev.org/Serial\\_ports](http://wiki.osdev.org/Serial_ports)



## 第5章

# セグメンテーション

x86におけるセグメンテーションとは、セグメントを通してメモリにアクセスすることを意味します。セグメントとは、アドレス空間の一部のことで、ベースアドレスとリミットによって指定され、重なり合うこともあります。セグメント化されたメモリのバイトをアドレスするには、48ビットの論理アドレスを使用します。16ビットでセグメントを指定し、32ビットでそのセグメント内のオフセットを指定します。オフセットはセグメントのベースアドレスに加えられ、結果として得られるリニアアドレスはセグメントのリミットと照合されます（下図参照）。すべてがうまくいけば（今は無視されているアクセス権のチェックも含めて）、結果はリニアアドレスになります。ページングが無効の場合、リニアアドレス空間は物理アドレス空間に1対1でマッピングされ、物理メモリにアクセスできます。（ページングを有効にする方法は「ページング」の章を参照してください）。

セグメント化を可能にするためには、各セグメントを記述したテーブル、つまりセグメントディスクリプターテーブルを設定する必要があります。x86では、ディスクリプターテーブルにはグローバルディスクリプターテーブル（GDT）とローカルディスクリプターテーブル（LDT）の2種類がある。LDTはユーザー空間のプロセスが設定・管理するもので、すべてのプロセスが独自のLDTを持っています。LDTは、より複雑なセグメンテーションモデルが必要な場合に使用できますが、ここでは使用しません。GDT はすべての人が共有するもので、グローバルなものです。

仮想メモリとページングのセクションで説明したように、セグメンテーションは、以下のような最小限の設定以上に使われることはほとんどありません。

## Accessing Memory

メモリにアクセスする際、ほとんどの場合、使用するセグメントを明示的に指定する必要はありません。プロセッサには、cs、ss、ds、es、gs、fsの6つの16ビットセグメントレジスタがあります。csはコードセグメントレジスタで、命令をフェッチする際に使用するセグメントを指定します。レジスタssはスタックポインタespを介してスタックにアクセスする際に使用され、dsはその他のデータアクセスに使用される。OSはレジスタes、gs、fsを自由に使用することができます。

以下は、セグメントレジスタの暗黙の利用例です。

のファンクです。

```
mov eax, [esp+4]
mov ebx, [eax] add
ebx, 8
mov [eax], ebx
ret
```

上記の例は、セグメントレジスタを明示的に使用した次の例と比較することができます。

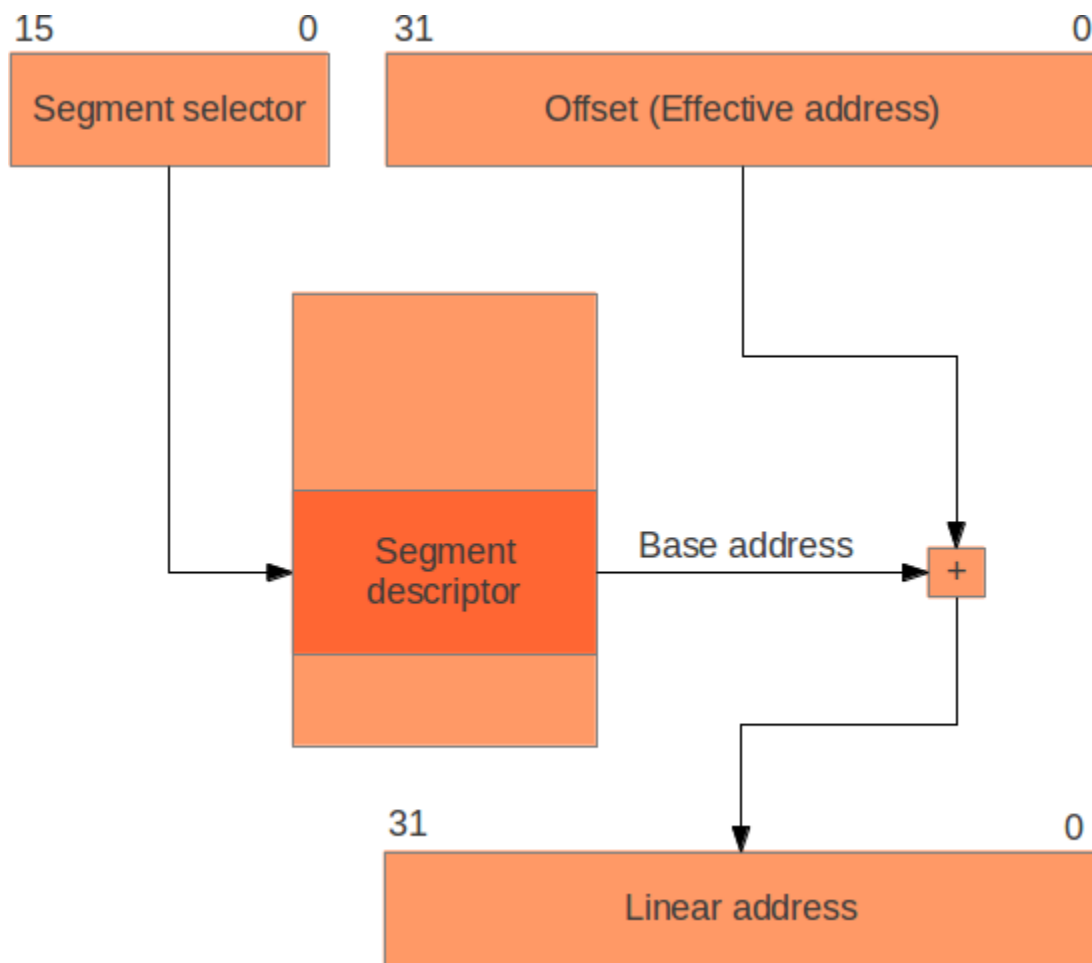


図5.1：論理アドレスとリニアアドレスの変換

のファンクです。

```
mov eax, [ss:esp+4]
mov ebx, [ds:eax]
add ebx, 8
mov [ds:eax], ebx
ret
```

スタックセグメントセレクトアの格納にssを、データセグメントセレクトアの格納にdsを使う必要はありません。スタックセグメントセレクトアをdsに格納することもできますし、その逆も可能です。しかし、上記の暗黙のスタイルを使用するためには、セグメントセレクトアをインデントされたレジスタに格納する必要があります。セグメントディスクリプターとそのフィールドは、インテルのマニュアル[33]の図3-8に記載されています。

## The Global Descriptor Table (GDT)

GDT/LDTは、8バイトのセグメントディスクリプターの配列です。GDTの最初の記述子は常にNULL記述子であり、決してメモリへのアクセスには使用できません。GDTには少なくとも2つのセグメント記述子（+ヌル記述子）が必要です。記述子にはbaseとlimitフィールド以外の情報が含まれているからです。私たちに最も関係のあるフィールドは、TypeフィールドとDescriptor Privilege Level (DPL) フィールドの2つです。

インテルのマニュアル[33]の第3章の表3-1には、Typeフィールドの値が規定されています。この表を見ると、Typeフィールドは書き込み可能と実行可能の両方を同時に行うことはできません。そのため、csに入れるコードを実行するためのセグメント（TypeはExite-onlyまたはExite-Read）と、他のセグメントのレジスタに入れるデータを読み書きするためのセグメント（TypeはRead/Write）の2つのセグメントが必要となります。

DPLは、そのセグメントを使用するために必要な特権レベルを指定します。x86では、0～3の4つの特権レベル（PL）があり、PL0が最も特権的です。ほとんどのオペレーティングシステム（LinuxやWindowsなど）では、PL0とPL3のみが使用されます。しかし、MINIXなどの一部のオペレーティングシステムでは、すべてのレベルを使用しています。カーネルは何でもできる必要があるため、DPL が 0 に設定されたセグメントを使用します（カーネルモードとも呼ばれます）。現在の特権レベル(CPL)は、cs のセグメントセレクトアによって決定されます。

必要なセグメントは以下の表のとおりです。

Index	Offset	Name	Address range	Type	DPL
0	0x00	null descriptor			
1	0x08	kernel code segment	0x00000000 - 0xFFFFFFFF	RX	PL0
2	0x10	kernel data segment	0x00000000 - 0xFFFFFFFF	RW	PL0

表5.1：必要なセグメント記述子

セグメントが重なっていることに注意してください。どちらもリニアアドレス空間全体をカバーしています。今回の最小セットアップでは、特権レベルを取得するためにセグメントを使用するだけです。その他の記述子フィールドの詳細については、インテルのマニュアル[33]の第3章を参照してください。

## Loading the GDT

GDTをプロセッサにロードするには、アセンブリコード命令「lgdt」を使用します。この命令は、GDTの開始とサイズを指定する構造体のアドレスを受け取ります。この情報は、次の例のように「packed struct」を使ってエンコードするのが最も簡単です。

```
struct gdt {  
    符号化された整数の  
    アドレス、符号化さ  
    れた短いサイズ。  
}属性（（パック））。
```

eaxレジスタの内容がそのような構造体のアドレスであれば、以下のようなアセンブリコードでGDTをロードすることができます。

```
lgdt [eax]
```

アセンブリコードの命令の入出力と同じように、この命令をC言語から利用できるようにすれば簡単かもしれません。

GDTがロードされた後、セグメントレジスタに対応するセグメントセクタをロードする必要があります。セグメント・セクタの内容は、以下の図と表を参照してください。

Bit:	15	3	2	1	0	
Content:	offset (index)		ti	rpl		

Name	Description
rpl	Requested Privilege Level - we want to execute in PL0 for now.
ti	Table Indicator. 0 means that this specifies a GDT segment, 1 means an LDT Segment.
offset (index)	Offset within descriptor table.

表 5.2: セグメントセクタのレイアウト

セグメントセクタのオフセットをGDTの先頭に加えて、セグメントディスクリプタのアドレスを得る。各ディスクリプタが8バイトであるため、1番目のディスクリプタは0x08、2番目のディスクリプタは0x10となる。OSのカーネルは特権レベル0で実行されるべきなので、要求された特権レベル（RPL）は0でなければなりません。

セグメント・セクタ・レジスタのロードは、データ・レジスタの場合は簡単で、正しいオフセットをレジスターにコピーするだけです。



```
mov ds, 0x10  
mov ss, 0x10  
mov es, 0x10
```

```
-  
-  
-
```

csを読み込むためには、「ファー・ジャンプ」をしなければなりません。

ここでのコードは、前のcs  
`jmp 0x08:flush_cs ; specify cs when jumping to flush_cs`  
flush\_csです。

現在、csを0x08に変更しています。

ファージャンプとは、48ビットの完全な論理アドレスを明示的に指定して行うジャンプのことで、使用するセグメントセレクタとジャンプ先の絶対アドレスを指定します。これは、まずcsを0x08に設定し、その絶対アドレスを使ってflush\_csにジャンプします。

## Further Reading

インテルのマニュアル[33]の第3章には、セグメンテーションに関する低レベルで技術的な詳細が記載されています。OSDev wikiにもセグメンテーションに関するページがあります。

<http://wiki.osdev.org/Segmentation>

x86のセグメンテーションに関するWikipediaのページは、調べる価値があるかもしれません。

[http://en.wikipedia.org/wiki/X86\\_memory\\_segmentation](http://en.wikipedia.org/wiki/X86_memory_segmentation)



## 第6章

# 割り込みと入力

せっかくOSが出力できるようになったのだから、入力もできたらいいなと思います。（キーボードから情報を読み取るためには、OSが割り込みに対応している必要があります）。割り込みは、キーボードやシリアルポート、タイマーなどのハードウェアデバイスが、デバイスの状態が変化したことをCPUに知らせるときに発生します。また、プログラムがアクセスできないメモリを参照した場合や、プログラムが数字をゼロで割った場合など、プログラムのエラーによってCPU自身が割り込みを発生させることもあります。最後に、ソフトウェアインタラプトもあります。これは、アセンブリコードのint命令によって引き起こされるインタラプトで、システムコールによく使われます。

## Interrupts Handlers

割り込みは、IDT（Interrupt Descriptor Table）によって処理されます。IDTには、各割り込みのハンドラが記述されています。割り込みには0～255の番号が付けられており、テーブルの*i*番目の位置に割り込み*i*のハンドラが定義されています。割り込みのハンドラには3つの種類があります。

- Task handler
- Interrupt handler
- Trap handler

タスクハンドラは、インテル版x86に特有の機能を使用しているため、ここでは説明しません（詳細はインテルのマニュアル[33]の第6章を参照してください）。割り込みハンドラとトラップハンドラの唯一の違いは、割り込みハンドラが割り込みを無効にすることで、割り込みを処理すると同時に割り込みを得ることができないということです。本書では、トラップハンドラを使用し、必要に応じて手動で割り込みを無効にすることにします。

## Creating an Entry in the IDT

割り込みハンドラのIDTのエントリは64ビットで構成されています。最上位32ビットを下图に示します。

Bit:	31	16	15	14 13	12	11	10 9 8	7 6 5	4 3 2 1 0	
Content:	offset high		P	DPL	0	D	1 1 0	0 0 0	reserved	

下位32ビットは次の図のように表示されます。

Bit:	31	16   15	0
Content:	segment selector	offset low	

各名称の説明は以下の表を参照してください。

Name	Description
offset high	The 16 highest bits of the 32 bit address in the segment.
offset low	The 16 lowest bits of the 32 bits address in the segment.
p	If the handler is present in memory or not (1 = present, 0 = not present).
DPL	Descriptor Privilege Level, the privilege level the handler can be called from (0, 1, 2, 3).
D	Size of gate, (1 = 32 bits, 0 = 16 bits).
segment selector	The offset in the GDT.
r	Reserved.

オフセットは、コード（できればアセンブリコードラベル）へのポインタです。例えば、コードが 0xDEADBEEF で始まり、特権レベル 0 で動作する（つまり、カーネルと同じコードセグメントセクタを使用する）ハンドラのエントリを作成するには、次の 2 バイトを使用します。

```
0xDEAD8E00
0x0008BEEF
```

IDTを符号なし整数idt[512]で表した場合、上記の例を割り込み0（ゼロ除算）のハンドラとして登録するには、次のようなコードになります。

```
idt[0] = 0xDEAD8E00
idt[1] = 0x0008BEEF
```

C言語入門」の章で述べたように、コードをより読みやすくするためには、バイト（または符号なし整数）を使う代わりに、パックされた構造体を使うことをお勧めします。

## Handling an Interrupt

割り込みが発生すると、CPUは割り込みに関する情報をスタックにプッシュし、IDTの中から適切な割り込みハンドラを探してジャンプします。割り込みが発生した時のスタックは以下のようになります。

```
[esp + 12] eflags
[esp + 8]  cs
[esp + 4]  eip
```

[esp]      error code?

エラーコードの後ろにクエスチョンマークがついているのは、すべての割り込みがエラーコードを作成するわけではないからです。スタック上にエラーコードを置く特定のCPU割り込みは、8、10、11、12、13、14、17です。このエラーコードは、割り込みハンドラが何が起こったかの詳細な情報を得るために使用することができます。また、割り込み番号はスタックにプッシュされないことに注意してください。どのコードが実行されているかを知ること、どのような割り込みが発生したかを判断することができます。つまり、割り込み17に登録されたハンドラが実行されていれば、割り込み17が発生したことになります。

割り込みハンドラが終了すると、iret命令を使って戻ります。iret命令は、スタックが割り込み時と同じであることを期待しています（上図参照）。したがって、割り込みハンドラによってスタックにプッシュされた値はすべてポップされなければなりません。iret はリターンする前に、スタックから値をポップしてeflagsを復元し、最後にスタック上の値で指定されたcs:eipにジャンプします。

割り込みハンドラはアセンブリコードで書かなければなりません。なぜならば、割り込みハンドラが使用するすべてのレジスタは、スタックにプッシュすることによって保存されなければならないからです。これは、割り込みを受けたコードは割り込みのことを知らないで、そのレジスタが変わらないことを期待するからです。割り込みハンドラのロジックをすべてアセンブリコードで書くのは面倒です。アセンブリコードで、レジスタを保存し、C関数を呼び出し、レジスタを復元し、最後にiretを実行するハンドラを作成するのは良いアイデアです。

Cハンドラは、レジスタの状態、スタックの状態、割り込みの番号を引数として取得する必要があります。例えば、以下のような定義が可能です。

```
struct cpu_state
{ unsigned int
  eax; unsigned int
  ebx; unsigned
  int ecx;
  -
  -
  -
  unsigned int esp;
}属性（（パック））。

struct stack_state
{ unsigned int
  error_code; unsigned
  int eip; unsigned int cs;
  unsigned int eflags;
}属性（（パック））。

void interrupt_handler(struct cpu_state cpu, struct stack_state stack, unsigned int interrupt);
```

## Creating a Generic Interrupt Handler

CPUは割り込み番号をスタックにプッシュしないので、一般的な割り込みハンドラを書くのは少し難しいです。このセクションでは、マクロを使ってその方法を説明します。割り込みごとに1つのバージョン



を書くのは面倒なので、NASM[34]のマクロ機能を使うのがよいでしょう。また、すべての割込みがエラーコードを生成するわけではないので、エラーコードのない割込みに対しては値0を「エラーコード」として追加します。以下のコードは、その例です。

```
%macro no_error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
    push    dword 0                ; push 0 as error code
```

```

        push    dword %1                ; push the interrupt number
        jmp     common_interrupt_handler ; jump to the common handler

%endmacro

%macro error_code_interrupt_handler %1
global interrupt_handler_%1
interrupt_handler_%1:
        push    dword %1                ; push the interrupt number
        jmp     common_interrupt_handler ; jump to the common handler

%endmacro

common_interrupt_handler:                ; the common parts of the generic interrupt handler
; save the registers
        push    eax
        push    ebx
        .
        .
        .
        push    ebp

; call the C function
        call    interrupt_handler

; restore the registers
        pop     ebp
        .
        .
        .
        pop     ebx
        pop     eax

; restore the esp
        add     esp, 8

```

邪魔されたコードに戻る

```

no_error_code_interrupt_handler 0        ; create handler for interrupt 0
no_error_code_interrupt_handler 1        ; create handler for interrupt 1
.
.
.
error_code_handler                7        ; create handler for interrupt 7
.
.
.

```

- common\_interrupt\_handlerは次のような処理を行います。

- Push the registers on the stack.
- Call the C function interrupt\_handler.

- Pop the registers from the stack.

- Add 8 to esp (because of the error code and the interrupt number pushed earlier).
- Execute iret to return to the interrupted code.

このマクロはグローバルラベルを宣言しているので、IDTを作成する際にCやアセンブリコードから割り込みハンドラのアドレスにアクセスすることができます。

## Loading the IDT

IDTには、テーブルの最初の要素のアドレスを取得するlidtアセンブリコード命令がロードされます。この命令をラップしてC言語から使用するのが最も簡単です。

```
global load_idt
; load_idt - 割り込みディスクリプターテーブル (IDT) をロードします。
; スタックです。[esp + 4] IDTの最初のエントリのアドレス
; [esp] the return address
load_idt:
    mov     eax, [esp+4]    ; load the address of the IDT into register eax
    lidt    eax             ; load the IDT
    ret                     ; return to the calling function
```

## Programmable Interrupt Controller (PIC)

ハードウェア割り込みを使用するには、まずPIC (Programmable Interrupt Controller) を設定する必要があります。PICは、ハードウェアからの信号を割り込みにマッピングすることを可能にします。PICを設定する理由は以下の通りです。

割り込みをリマップする。PICはデフォルトでハードウェア割り込みに割り込み0~15を使用しているため、CPUの割り込みと競合してしまいます。そのため、PICの割り込みを別の間隔にリマップする必要があります。

- 受信する割り込みを選択します。すべてのデバイスからの割り込みを受信したいとは思わないでしょう。なぜなら、これらの割り込みを処理するコードを持っていないからです。
- Set up the correct mode for the PIC.

当初、PICは1つ (PIC1) しかなく、割り込みは8つでした。しかし、ハードウェアが増えるにつれ、8本の割り込みでは少なすぎます。そこで、1つ目のPICにもう1つのPIC (PIC2) をチェーン接続することにしました (PIC1の割り込み2参照)。

ハードウェア割り込みは、下表のとおりです。

PIC 1	Hardware	PIC 2	Hardware
0	Timer	8	Real Time Clock
1	Keyboard	9	General I/O
2	PIC 2	10	General I/O
3	COM 2	11	General I/O

4 COM 1

12 General I/O

PIC 1	Hardware	PIC 2	Hardware
5	LPT 2	13	Coprocessor
6	Floppy disk	14	IDE Bus
7	LPT 1	15	IDE Bus

PICを構成するための素晴らしいチュートリアルがSigOPSのウェブサイトにあります[35]。ここではその情報を繰り返しません。

PICからのすべての割り込みは、確認されなければなりません。つまり、割り込みが処理されたことを確認するメッセージをPICに送ることです。これが行われないと、PICはそれ以上の割り込みを生成しません。

PICの割り込みを確認するには、割り込みを発生させたPICにバイト0x20を送信します。したがって、pic\_acknowledge関数の実装は次のように行います。

```
#include "io.h"

#define PIC1_PORT_A 0x20
#define PIC2_PORT_A 0xA0

/* PICの割り込みがリマップされました */
#define PIC1_START_INTERRUPT 0x20
#define PIC2_START_INTERRUPT 0x28
#define PIC2_END_INTERRUPT PIC2_START_INTERRUPT + 7

#define PIC_ACK 0x20
* /** pic_acknowledge:
 *   Acknowledges an interrupt from either PIC 1 or PIC 2.
 *   *
 *   @param num The number of the interrupt
 */
void pic_acknowledge(unsigned integer interrupt)
{
    if (interrupt < PIC1_START_INTERRUPT || interrupt >
        PIC2_END_INTERRUPT) { return;
    }
    if (interrupt <
        PIC2_START_INTERRUPT)
    { outb(PIC1_PORT_A,
        PIC_ACK);
    } else {
        outb(PIC2_PORT_A, PIC_ACK)となります。
    }
}
```

## Reading Input from the Keyboard

キーボードはASCII文字を生成するのではなく、スキャンコードを生成します。スキャンコードは、ボタ

ンの押し方と離し方を表しています。押したばかりのボタンを表すスキャンコードは、キーボードのデータI/Oポート（アドレスは0x60）から読み取ることができます。その方法を以下の例で示します。

```

#include "io.h"

#define KBD_DATA_PORT 0x60
* /** read_scan_code:
 *   Reads a scan code from the keyboard
 *
 *   @return The scan code (NOT an ASCII character!)
 */
符号なし char read_scan_code(void)
{
    return inb(KBD_DATA_PORT)となります。
}

```

次のステップは、スキャンコードを対応するASCII文字に変換する関数を書くことです。アメリカンキーボードのように、スキャンコードをASCII文字にマッピングしたい場合は、Andries Brouwer氏の素晴らしいチュートリアルがあります[36]。

キーボード割り込みはPICが発生させるので、キーボード割り込みハンドラの最後にpic\_acknowledgeを呼び出す必要があることを覚えておいてください。また、キーボードからスキャンコードを読み取るまでは、それ以上の割り込みを送らないようにします。

## Further Reading

- The OSDev wiki has a great page on interrupts, <http://wiki.osdev.org/Interrupts>
- Chapter 6 of Intel Manual 3a [33] describes everything there is to know about interrupts.





## 第7章

# ユーザーモードへの道

さて、カーネルが起動し、画面に表示され、キーボードから読み込まれたところで、私たちは何をするのでしょうか？通常、カーネルはアプリケーションロジックを自ら行うことはなく、それはアプリケーションに任せることになっています。カーネルは、アプリケーションの開発を容易にするための適切な抽象化（メモリ、ファイル、デバイス）を行い、アプリケーションに代わってタスクを実行し（システムコール）、プロセスをスケジューリングします。

ユーザーモードとは、カーネルモードとは対照的に、ユーザーのプログラムが実行される環境のことです。この環境はカーネルよりも権限が低く、（悪意を持って書かれた）ユーザープログラムが他のプログラムやカーネルを混乱させることを防ぎます。悪く書かれたカーネルは、自由に好きなように混乱させることができます。

本書で作成したOSがユーザーモードでプログラムを実行できるようになるのはまだまだ先のことですが、この章では小さなプログラムを簡単にカーネルモードで実行する方法を紹介します。

## Loading an External Program

外部プログラムをどこから持ってくるのか？何らかの方法で、実行したいコードをメモリにロードする必要があります。機能が充実しているOSでは、CD-ROMドライブやハードディスクなどの永続的なメディアからソフトウェアをロードできるようなドライバーやファイルシステムを備えているのが普通です。

これらのドライバやファイルシステムをすべて作成するのではなく、GRUBのモジュールという機能を使ってプログラムを読み込みます。

## GRUB Modules

GRUB は ISO イメージから任意のファイルをメモリにロードすることができ、これらのファイルは通常、モジュールと呼ばれます。GRUB にモジュールを読み込ませるには、iso/boot/grub/menu.lst ファイルを編集して、ファイルの最後に次の行を追加します。

モジュール /modules/program

次に、iso/modules フォルダを作

成します。

アプリケーションプログラムの作成はこの章の後半で行います。

kmain を呼び出すコードは、どこでモジュールを見つけられるかという情報を kmain に渡すように更新しなければなりません。また、GRUB には、モジュールをロードするときに、すべてのモジュールをページ境界に揃えるべきだと伝えたいと思います（ページアライメントについての詳細は、「ページング」の章を参照してください）。

GRUBにモジュールのロード方法を指示するためには、カーネルの最初のバイトである「マルチブート・ヘッダー」を以下のように更新する必要があります。

ファイル'loader.s'内の ;

```
MAGIC_NUMBER    equ 0x1BADB002      ; define the magic number constant
ALIGN_MODULES    equ 0x00000001      ; tell GRUB to align modules

; calculate the checksum (all options + checksum should equal 0)
CHECKSUM         equ -(MAGIC_NUMBER + ALIGN_MODULES)

section .text:                ; start of the text (code) section
align 4                      ; the code must be 4 byte aligned
    dd MAGIC_NUMBER          ; write the magic number
    dd ALIGN_MODULES          ; write the align modules instruction
    dd CHECKSUM               ; write the checksum
```

また、GRUB はレジスタ ebx に構造体へのポインタを格納します。この構造体には、モジュールがどのアドレスにロードされるかなどが記述されています。したがって、おそらく kmain を呼び出す前に ebx をスタックにプッシュして kmain の引数にしたいと思うでしょう。

## Executing a Program

### A Very Simple Program

この段階で書かれたプログラムは、わずかな動作しかできません。そのため、テストプログラムとしては、レジスタに値を書き込むだけの非常に短いプログラムで十分です。しばらくしてBochsを停止し、Bochsのログを見てレジスタに正しい数値が入っていることを確認すれば、プログラムが実行されたことが確認できます。これはそのような短いプログラムの一例です。

```
mov eax,
0xDEADBEEF . . . . .
; 無限ループに入り、これ以上何もできない
; $は「行頭」を意味し、つまり jmp
$と同じ命令です。
```

## Compiling

カーネルは高度な実行形式を解析できないので、コードをフラットなバイナリにコンパイルする必要があります。

ります。NASMでは-fというフラグを使ってこれを行うことができます。

```
nasm -f bin program.s -o program
```

これで必要なものはすべて揃いました。ここで、ファイルプログラムをiso/modulesフォルダに移動する必要があります。

## Finding the Program in Memory

プログラムにジャンプする前に、そのプログラムがメモリ上のどこにあるかを調べなければなりません。ebxの内容が次のようになっているとします。

がkmainの引数として渡された場合、これを完全にCから行うことができます。

ebxのポインターはmultiboot構造体を指しています[19]。 [http://www.gnu.org/software/grub/manual/multiboot/html\\_node/multiboot.h.html](http://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot.h.html) から構造体を記述した multiboot.h ファイル

をダウンロードしてください。

ebxレジスタでkmainに渡されるポインタは、multiboot\_info\_tポインタにキャストできます。最初のモジュールのアドレスは、フィールド mods\_addr にあります。次のコードはその例です。

```
int kmain(/* 追加の引数 */ unsigned int ebx)
{
    multiboot_info_t *mbinfo = (multiboot_info_t *) ebx;
    unsigned int address_of_module = mbinfo->mods_addr;
}
```

しかし、ただやみくもにポインタに従う前に、モジュールがGRUBによって正しくロードされたかどうかを確認する必要があります。これは、multiboot\_info\_t 構造体の flags フィールドをチェックすることで可能です。また、mods\_count フィールドをチェックして、それがちょうど1であることを確認してください。multiboot 構造体の詳細については、multiboot documentation [19]を参照してください。

## Jumping to the Code

あとは、GRUBでロードされたコードにジャンプするだけです。アセンブリコードよりもC言語の方がマルチブート構造を解析しやすいので、C言語からコードを呼び出す方が便利です（もちろん、アセンブリコードでもjmpやcallを使って実行できます）。C言語のコードは次のようになります。

```
typedef void (*call_module_t)(void);
/* ... */
call_module_t start_program = (call_module_t) address_of_module;
start_program();
モジュールのコードが返ってこない限り、/* ここにはたどり着けません。
```

カーネルを起動し、プログラムが実行されて無限ループに入るまで待ってからBochsを停止すると、Bochsのログを介してレジスタeaxに0xDEADBEEFが表示されるはずです。私たちのOSでプログラムの起動に成功したのです！

## The Beginning of User Mode

今、私たちが書いたプログラムは、カーネルと同じ特権レベルで実行されていますが、ちょっと変わった方法でそれを入力しただけです。アプリケーションが異なる特権レベルで実行できるようにするには、セグメンテーションの他に、ページングやページフレームの割り当てが必要です。

技術的にはかなり大変ですが、数章後にはユーザーモードのプログラムが動くようになります。



## 第8章

# 仮想記憶の基礎知識

仮想メモリとは、物理的なメモリを抽象化したものである。仮想メモリの目的は、アプリケーションの開発を容易にすることと、実際にマシンに物理的に存在するメモリよりも多くのメモリをプロセスに割り当てることです。また、セキュリティの観点から、アプリケーションがカーネルや他のアプリケーションのメモリをいじってしまうことは避けたい。

x86アーキテクチャでは、仮想メモリはセグメント化とページングの2つの方法で実現されます。ページングは最も一般的で汎用性の高い手法であり、次の章で実装します。異なる特権レベルでコードを実行できるようにするには、やはりセグメンテーションの使用が必要です。

オペレーティングシステムでは、メモリの管理が重要な役割を果たします。ページングとページフレームアロケーションがそれにあたります。

セグメンテーションとページングについては、[33]の第3章と第4章に記載されています。

## Virtual Memory Through Segmentation?

ページングを完全にスキップして、仮想メモリにセグメントを使用することができます。ユーザーモードの各プロセスは、ベースアドレスとリミットが適切に設定された、独自のセグメントを取得します。この方法では、どのプロセスも他のプロセスのメモリを見ることはできません。この方法の問題点は、プロセスの物理メモリが連続している必要があることです（少なくとも、連続していれば非常に便利です）。プログラムがどれだけのメモリを必要とするかを事前に知る必要があるか（ありえない）、またはメモリセグメントを制限に達したときに成長できる場所に移動させる必要があるか（コストがかかる、断片化の原因となる、十分なメモリがあるにもかかわらず「メモリ不足」になる可能性がある）。これらの問題を解決するのがページングです。

興味深いのは、x86\_64（x86アーキテクチャの64ビット版）では、セグメンテーションがほぼ完全に削除されていることです。

## Further Reading

LWN.netでは、仮想メモリに関する記事を掲載しています。<http://lwn.net/Articles/253361/>

Gustavo Duarte氏も仮想メモリに関する記事を書いています。<http://duartes.org/gustavo/blog/>



post/memory-translation-and-segmentation



## 第9章

# ページング

セグメント化は、論理アドレスを線形アドレスに変換します。ページングは、このリニアアドレスを物理アドレス空間に変換し、アクセス権やメモリのキャッシュ方法を決定します。

### Why Paging?

ページングは、x86で仮想メモリを有効にするための最も一般的な手法です。ページングによる仮想メモリとは、実際のメモリサイズはもっと小さいかもしれないのに、各プロセスは利用可能なメモリ範囲が0x00000000~0xFFFFFFFFであるかのように錯覚することを意味します。また、プロセスがメモリのバイトをアドレス指定する際に、物理アドレスではなく、仮想（リニア）アドレスを使用することになります。ユーザープロセスのコードは、（実行の遅延を除いて）何の違いも感じません。リニアアドレスは、MMUとページテーブルによって物理アドレスに変換されます。もし、仮想アドレスが物理アドレスにマッピングされていない場合、CPUはページフォルト割り込みを発生させます。

ページングはオプションであり、OSによっては利用しないものもあります。しかし、特定の特権レベルで実行されるコードのみがアクセスできるメモリ領域をマークしたい場合（異なる特権レベルで実行されるプロセスを持つことができるようにしたい場合）、ページングは最もすてきな方法です。

### Paging in x86

x86のページング（インテルのマニュアル[33]の第4章）は、1024個のページテーブル（PT）への参照を含むことができるページディレクトリ（PDT）で構成され、各ページテーブルはページフレーム（PF）と呼ばれる物理メモリの1024個のセクションを指すことができます。各ページフレームは4096バイトの大きさです。仮想（リニア）アドレスでは、上位10ビットが現在のPDT内のページディレクトリエントリ（PDE）のオフセットを、次の10ビットがそのPDEが指すページテーブル内のページテーブルエントリ（PTE）のオフセットを指定します。また、アドレスの下位12ビットは、アドレスするページフレーム内のオフセットです。

すべてのページディレクトリ、ページテーブル、ページフレームは、4096バイトのアドレスに整列する必要があります。これにより、PDT、PT、PFのアドレスは、32ビットのアドレスのうち、上位20ビットだけで可能となります（下位12ビットは0である必要があります）。

PDEとPTEの構造は非常によく似ています。32ビット（4バイト）のうち、上位20ビットがPTEまたはPFを示し、下位12ビットがアクセス権などの設定を制御します。4バイト×1024=4096バイトなので、ページディレクトリとページテーブルの両方がページフレーム自体に収まる。

リニアアドレスから物理アドレスへの変換は下図のように行われます。

ページは通常4096バイトですが、4MBのページを使用することも可能です。この場合、PDEは4MBのページフレームを直接指定し、4MBのアドレス境界にアラインメントする必要があります。アドレス変換は、ページテーブルのステップを除いただけで、図とほぼ同じです。なお、4MBと4KBのページを混在させることも可能です。

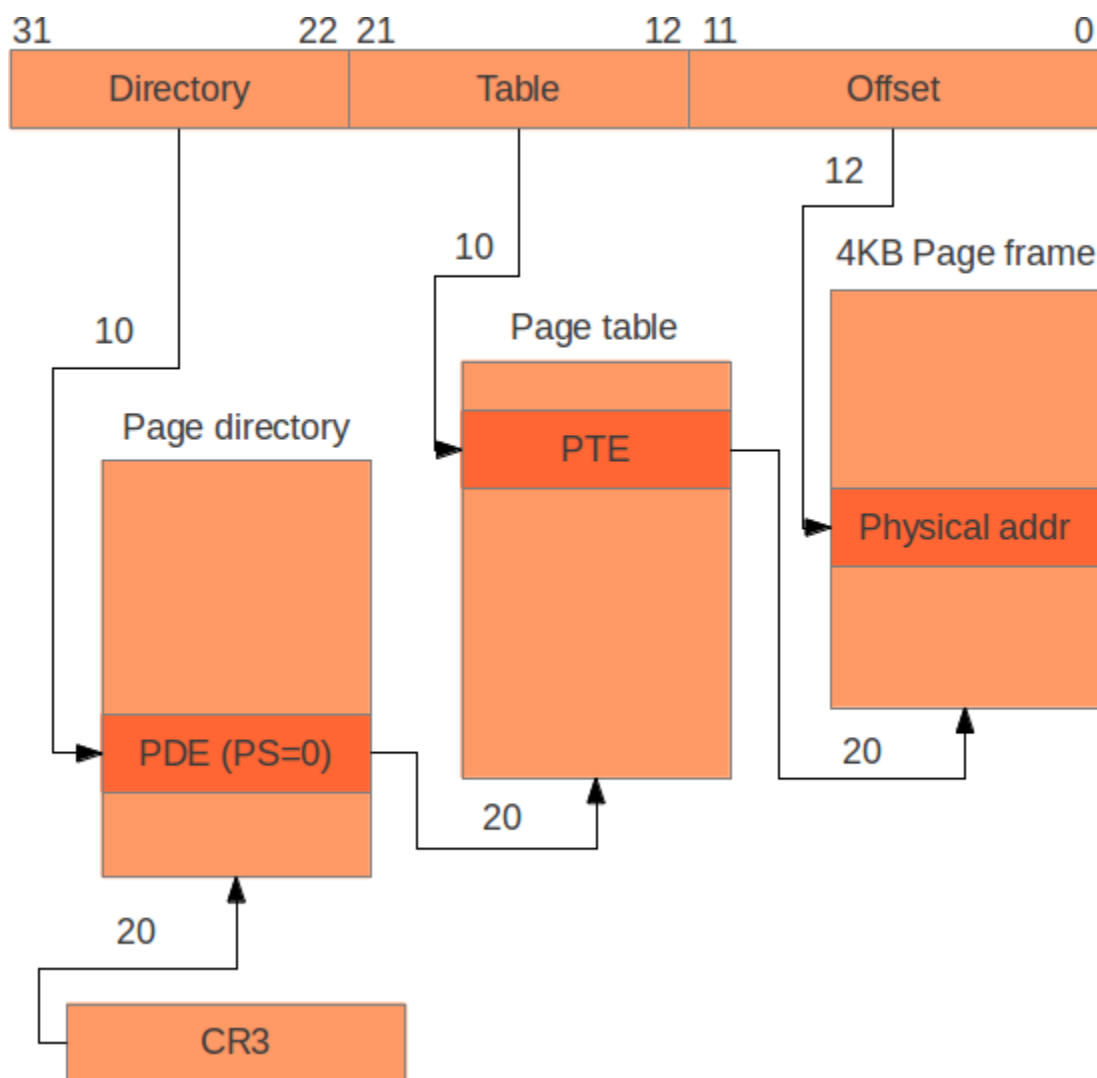


図9.1：仮想アドレス（リニアアドレス）を物理アドレスに変換する。

現在のPDTを指し示す20ビットは、レジスタcr3に格納されます。cr3の下位12ビットはコンフィギュレーションに使用されます。

ページング構造の詳細については、インテルのマニュアル[33]の第4章を参照してください。最も興味深

いビットはU/Sで、このページにアクセスできる特権レベル(PL0またはPL3)を決定し、R/Wはページ内のメモリを読み書き可能にするか、読み取り専用にするかを決定します。

## Identity Paging

最も単純なページングは、各仮想アドレスを同じ物理アドレスにマッピングするもので、アイデンティティページングと呼ばれます。これは、コンパイル時にページディレクトリを作成し、各エントリが対応する4MBフレームを指すようにすることで実現できます。NASMでは、マクロやコマンド（%rep、times、dd）を使って行うことができます。もちろん、実行時には通常のアセンブリコード命令を使って行うこともできます。

## Enabling Paging

ページングは、まずcr3にページディレクトリのアドレスを書き込み、次にcr0のビット31（PGの「ページングイネーブル」ビット）を1に設定することで有効になります。4MBのページを使用するには、cr4のPSEビット（Page Size Extensions、ビット4）を設定します。次のアセンブリコードは、その例を示しています。

```
; eaxはページディレクトリ  
 のアドレス mov cr3, eax  
  
mov ebx, cr4      ; read current cr4  
or  ebx, 0x00000010 ; set PSE  
mov cr4, ebx      ; update cr4  
  
mov ebx, cr0      ; read current cr0  
or  ebx, 0x80000000 ; set PG  
mov cr0, ebx      ; update cr0  
 ページングが可能になりました。
```

## A Few Details

ここで重要なのは、ページディレクトリ、ページテーブル、cr3内のすべてのアドレスは、構造体の物理アドレスである必要があり、決して仮想アドレスではないということです。これは、ページング構造を動的に更新する後のセクションで、より重要になります（「ユーザーモード」の章を参照）。

PDTやPTを更新する際に便利な命令がinvlpgです。この命令は、仮想アドレスのTLB（Translation Lookaside Buffer）エントリを無効にします。TLBは、仮想アドレスに対応する物理アドレスをマッピングする、変換アドレス用のキャッシュです。これは、以前に別のものにマッピングされていたPDEまたはPTEを変更する場合にのみ必要です。PDEまたはPTEが以前に存在しないものとしてマークされていた（ビット0が0に設定されていた）場合、invlpgの実行は不要です。cr3の値を変更すると、TLBのすべてのエントリが無効になります。

TLBエントリの無効化の例を以下に示します。

```
仮想アドレス0へのTLB参照を無効  
にする invlpg [0].
```

## **Paging and the Kernel**

ここでは、ページングがOSのカーネルに与える影響について説明します。アセンブリコードで設定されたページテーブルの不具合をデバッグするのは困難なので、より高度なページング設定を行う前に、アイデンティティページングを使用してOSを実行することをお勧めします。



## Reasons to Not Identity Map the Kernel

カーネルが仮想アドレス空間の先頭に配置されている場合、つまり、仮想アドレス空間（0x00000000、「カーネルのサイズ」）がメモリ内のカーネルの位置にマッピングされている場合、ユーザーモードのプロセスコードをリンクする際に問題が発生します。通常、リンクの際、リンカーはコードがメモリ上の0x00000000の位置にロードされることを想定しています。そのため、絶対参照を解決する際には、0x00000000がベースアドレスとなり、正確な位置を計算します。しかし、カーネルが仮想アドレス空間（0x00000000、「カーネルのサイズ」）にマッピングされている場合、ユーザーモードのプロセスは仮想アドレス0x00000000にロードすることはできません。そのため、リンカーが「ユーザーモードプロセスは0x00000000の位置にロードされる」と仮定しているのは間違いです。この問題は、リンカーに別の開始アドレスを仮定するよう指示するリンカー・スクリプトを使用することで修正できますが、これはオペレーティングシステムのユーザーにとって非常に面倒な解決策です。

これは、カーネルがユーザーモードプロセスのアドレス空間の一部であることを前提としています。後で説明しますが、これは良い機能で、システムコール中にカーネルのコードやデータにアクセスするためにページング構造を変更する必要がありません。もちろん、カーネルページへのアクセスには特権レベル0が必要で、ユーザープロセスがカーネルメモリを読み書きできないようになっています。

## The Virtual Address for the Kernel

望ましいのは、カーネルを非常に高い仮想メモリアドレス、例えば0xC0000000（3GB）に配置することです。ユーザーモードのプロセスが3GBの大きさになることはまずありませんが、これではカーネルと衝突してしまうことになります。カーネルが3GB以上の仮想アドレスを使用する場合は、higher-half kernelと呼ばれます。0xC0000000は一例で、カーネルは0よりも高いアドレスに配置しても同じ効果が得られます。正しいアドレスを選択するには、カーネルにどれだけの仮想メモリを割り当てるか（カーネルの仮想アドレスより上のメモリをすべてカーネルに割り当てるのが最も簡単です）、また、プロセスにどれだけの仮想メモリを割り当てるかによります。

ユーザーモードのプロセスが3GBより大きい場合、いくつかのページをカーネルがスワップアウトする必要があります。ページの交換については、本書では扱いません。

## Placing the Kernel at 0xC0000000

そもそも、カーネルを0xC0000000よりも0xC0100000に配置した方が、(0x00000000, 0x00100000)を(0xC0000000, 0xC0100000)にマッピングすることが可能になるからです。これにより、メモリの全範囲（0x00000000, "size of kernel"）が、（0xC0000000, 0xC0000000 + "size of kernel"）の範囲にマッピングされます。

カーネルを0xC0100000に配置するのは難しくありませんが、少し考えなければなりません。これは再びリンクの問題です。リンカーがカーネル内のすべての絶対参照を解決するとき、リンカーはカーネルが0x00000000ではなく物理メモリ・ロケーション0x00100000にロードされていると仮定します。これはリンカー・スクリプトで再配置が使用されているからです（「カーネルのリンク」のセクションを参照）。しかし、0xC0100000 をベース・アドレスとしてジャンプを解決したい。そうしないと、カーネル・ジャンプがそのままユーザー・モード・プロセス・コードにジャンプしてしまうからである（ユーザー・モード・プロセスは仮想メモリ 0x00000000 にロードされていることを覚えておいてほしい）。

しかし、リンカーに「カーネルは 0xC0100000 で始まる（ロードされる）」と単純に伝えることはできません。カーネルが1MBにロードされる理由は、1MBの下にBIOSやGRUBのコードがロードされているため、0x00000000にはロードされないからです。さらに、マシンに3GBの物理メモリが搭載されていない可能性もあるため、0xC0100000にカーネルをロードできるとは考えられません。

この問題は、リンカースクリプトで再配置（.=0xC0100000）とAT命令の両方を使用することで解決できます。再配置は、非相対的なメモリ参照の際に、アドレス計算のベースとして再配置アドレスを使用することを指定します。ATは、カーネルをメモリにロードする場所を指定します。再配置はリンク時に行われる

GNU ld [37]では、ATで指定されたロードアドレスは、カーネルをロードする際にGRUBで処理され、ELFフォーマット[18]の一部となっています。

## Higher-half Linker Script

最初のリンカースクリプトを変更して、これを実装することができます。

```
ENTRY(loader)           /* the name of the entry symbol */

. = 0xC0100000           /* the code should be relocated to 3GB + 1MB */
4KBで整列、1MBでロード */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)              /* all text sections from all files */
}

/* 4KBで整列し、1MBでロードします。 */
.rodata ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.rodata*)           /* all read-only data sections from all files */
}

/* 4KBで整列し、1MBでロードします。 */
.data ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.data)              /* all data sections from all files */
}

/* 4KBで整列し、1MBでロードします。 */
.bss ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(COMMON)             /* all COMMON sections from all files */
    *(.bss)               /* all bss sections from all files */
}
```

## Entering the Higher Half

GRUBがカーネルコードにジャンプする際、ページングテーブルがありません。そのため、0xC0100000 + Xへの参照はすべて正しい物理アドレスにマッピングされず、せいぜい一般保護例外 (GPE) が発生する程度で、そうでなければ (コンピュータに3GB以上のメモリが搭載されている場合) コンピュータがクラッシュしてしまいます。

そのため、相対ジャンプや相対メモリアドレッシングを使用しないアセンブリコードを使用して、以下のようにする必要があります。

- Set up a page table.
- Add identity mapping for the first 4 MB of the virtual address space.
- Add an entry for 0xC0100000 that maps to 0x00100000

最初の4MBのIDマッピングをスキップすると、CPUはページングが有効になった直後に次の命令をメモリから取得しようとしてページフォルトを発生させてしまいます。テーブルを作成した後、ラベルにジャンプを行い、eipが上位半分の仮想アドレスを指すようにすることができます。

## 0x00100000付近で実行されているアセンブリコード カーネルの実際の位置の両方にページングを有効にする とその上半分の仮想位置

```
lea ebx, [higher_half] ; load the address of the label in ebx  
jmp ebx                ; jump to the label  
higher_halfです。
```

ここでのコードは、上位のカーネルで実行されます。

eipが0xC0000000より大きい場合

; カーネルの初期化やCコードの呼び出しなどを続けることができます。

レジスタeipは0xC0100000の直後のメモリ位置を指すようになり、すべてのコードは上位半分の0xC0100000にあるかのように実行できるようになります。最初の4MBの仮想メモリを最初の4MBの物理メモリにマッピングするエントリをページテーブルから削除し、TLBの対応するエントリをinvlpg [0]で無効にすることができます。

## Running in the Higher Half

上位半分のカーネルを使用する際には、さらにいくつかの詳細に対処する必要があります。特定のメモリ位置を使用するメモリマップドI/Oを使用する際には注意が必要です。例えば、フレームバッファは0x000B8000番地にありますが、ページテーブルには0x000B8000番地のエントリはもう存在しないので、0xC00B8000番地を使用しなければなりません。

マルチブート構造内でアドレスを明示的に参照している場合は、新しい仮想アドレスを反映して変更する必要があります。

カーネルに4MBのページをマッピングするのは簡単ですが、メモリを無駄に消費してしまいます（よほど大きなカーネルでない限り）。4KBページとしてマッピングされた上位半分のカーネルを作成すると、メモリを節約できますが、設定が難しくなります。ページディレクトリと1つのページテーブルのためのメモリは.dataセクションに確保できますが、仮想アドレスから物理アドレスへのマッピングをランタイムに設定する必要があります。カーネルのサイズは、リンカースクリプト[37]からラベルをエクスポートすることで決定できますが、これは後でページフレームアロケータを書くときに必要になります（「ページフレームアロケータ」の章を参照してください）。

## Virtual Memory Through Paging

ページングによって、仮想メモリにとって良いことが2つあります。1つ目は、メモリへのきめ細かなアクセス制御が可能になることです。ページをリードオンリー、リードライト、PL0専用などとすることができます。第二に、仮想メモリは連続したメモリであるかのような錯覚をもたらします。ユーザーモードのプロセスやカーネルは、あたかもメモリが連続しているかのようにアクセスすることができ、メモリ内でデータを移動させることなく、連続したメモリを拡張することができます。また、ユーザーモードプログラムが3GB以下のすべてのメモリにアクセスできるようにすることもできますが、実際に使用しない限り、ページフレームを割り当てる必要はありません。これにより、プロセスが0x00000000付近にコードを配置し、0xC0000000のすぐ下にスタックを配置しても、実際に必要なページ数は2ページ以上にはなりません。

## Further Reading

ページングの詳細については、インテルのマニュアル[33]の第4章（および第3章）が決定的な情報源となります。

ウィキペディアには、ページングに関する記事があります。 <http://en.wikipedia.org/wiki/Paging>

OSDevのwikiにはページングについてのページがあります: <http://wiki.osdev.org/Paging>、higher-

half kernelを作るためのチュートリアルがあります: [http://wiki.osdev.org/Higher\\_Half\\_bare\\_bones](http://wiki.osdev.org/Higher_Half_bare_bones)

カーネルがどのようにメモリを管理しているかについてのGustavo Duarte氏の記事は一読の価値があります。 <http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

リンカのコマンド言語の詳細については、Steve Chamberlain氏のウェブサイト[37]を参照してください。

ELFフォーマットの詳細については、こちらのプレゼンテーションをご覧ください：

[http://flint.cs.yale.edu/cs422/doc/ELF\\_Format.pdf](http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf)



## 第10章

# ページフレームの割り当て

仮想メモリを使うとき、OSはメモリのどの部分が空いているかをどうやって知るのでしょうか。それが、ページフレームアローケータの役割です。

## Managing Available Memory

### How Much Memory is There?

まず、OS が動作しているコンピュータで利用可能なメモリの量を知る必要があります。これを行う最も簡単な方法は、GRUB から渡される multiboot 構造体 [19] から読み取ることです。GRUB は、予約済み、I/O マップ、読み取り専用など、メモリに関する必要な情報を収集します。また、カーネルが使用するメモリの一部をフリーとしてマークしないようにしなければなりません（GRUBはこのメモリを予約済みとしてマークしないため）。カーネルがどれだけのメモリを使用しているかを知る一つの方法は、リンクスクリプトからカーネルバイナリの最初と最後にラベルをエクスポートすることです。

```
ENTRY(loader)                /* the name of the entry symbol */

. = 0xC0100000                /* the code should be relocated to 3 GB + 1 MB */
/* これらのラベルはコードファイルにエクスポートされま
す */ kernel_virtual_start = ..;

kernel_physical_start = . - 0xC0000000;
4KBで整列、1MBでロード */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)                  /* all text sections from all files */
}

/* 4KBで整列し、1MBでロードします。 */
.rodata ALIGN (0x1000) : AT(ADDR(.rodata)-0xC0000000)
{
    *(.rodata*)               /* all read-only data sections from all files */
}
```



```

/* 4KBで整列し、1MBでロードします。*/
.data ALIGN (0x1000) : AT(ADDR(.data)-0xC0000000)
{
    *(.data)          /* all data sections from all files */
}

/* 4KBで整列し、1MBでロードします。*/
.bss ALIGN (0x1000) : AT(ADDR(.bss)-0xC0000000)
{
    *(COMMON)         /* all COMMON sections from all files */
    *(.bss)           /* all bss sections from all files */
}

kernel_virtual_end = .;
kernel_physical_end = . - 0xC0000000;

```

これらのラベルは、アセンブリコードから直接読み込まれ、スタックにプッシュされることで、Cコードが利用できるようになります。

```

extern kernel_virtual_start
extern kernel_virtual_end
extern kernel_physical_start
extern kernel_physical_end

; ...

push kernel_physical_end
push kernel_physical_start
push kernel_virtual_end
push kernel_virtual_start
コールクマイン

```

このようにして、ラベルをkmainの引数として取得します。アセンブリコードではなくC言語を使用した場合は、ラベルを関数として宣言し、その関数のアドレスを取得する方法があります。

```

void kernel_virtual_start(void);

/* ... */
符号付き整数 vaddr = (符号付き整数) &kernel_virtual_start;

```

GRUBモジュールを使用している場合は、そのモジュールが使用するメモリが予約済みであることを確認する必要があります。

あります。

なお、利用可能なメモリは連続している必要はありません。最初の1MBには、いくつかのI/Oマップされたメモリセクションがあり、GRUBやBIOSが使用するメモリもあります。メモリの他の部分も同様に使用できない可能性があります。

ページの一部をメモリにマッピングすることはできないので、メモリセクションを完全なページフレームに分割するのが便利です。

## Managing Available Memory

どのページフレームが使用されているかをどのようにして知ることができますか？ページフレームアロケータは、どのページフレームが空いていて、どのページフレームが空いていないかを把握する必要があります。これにはいくつかの方法があります。ビットマップ、リンクリスト、ツリー、Buddy System（Linuxで使用されている）などです。さまざまなアルゴリズムの詳細については、OSDevの記事[38]をご覧ください。

ビットマップの実装は非常に簡単です。1つのページフレームに1つのビットを使用し、1つ（または複数）のページフレームをビットマップを格納するために使用します。（ただし、これは一つの方法に過ぎず、他のデザインの方がより良く、またより楽しく実装できるかもしれません）。

## How Can We Access a Page Frame?

ページフレームアロケータは、ページフレームの物理的な開始アドレスを返します。このページフレームはマップインされておらず、このページフレームを指すページテーブル也没有ありません。このフレームにデータを読み書きするにはどうすればよいのでしょうか。

カーネルが使用するPDTやPTを更新することで、ページフレームを仮想メモリにマッピングする必要があります。利用可能なページテーブルがすべて満杯の場合はどうでしょうか？そうになると、ページフレームをメモリにマッピングすることができません。なぜなら、ページフレーム全体を占める新しいページテーブルが必要になり、このページフレームに書き込むためには、そのページフレームをマッピングする必要があるからです。..この循環的な依存関係をどうにかして解消しなければなりません。

一つの解決策は、カーネルが使用する最初のページテーブル（または他の上位半分のページテーブル）の一部を、ページフレームを一時的にマッピングしてアクセス可能にするために確保することです。カーネルが0xC0000000（インデックス768のページディレクトリエントリ）にマッピングされ、4KBのページフレームが使用されている場合、カーネルは少なくとも1つのページテーブルを持っています。カーネルのサイズが最大でも4MBから4KBを引いたサイズであると仮定した場合、このページテーブルの最後のエントリ（エントリ1023）を一時的なマッピング用に割り当てることができます。カーネルのPTの最後のエントリを使ってマッピングされたページの仮想アドレスは次のようになります。

$$(768 \ll 22) \mid (1023 \ll 12) \mid 0x000 = 0xC03FF000$$

ページテーブルとして使用したいページフレームを一時的にマッピングし、最初のページフレームにマッピングするように設定した後、ページングディレクトリに追加し、一時的なマッピングを削除します。

## A Kernel Heap

これまでは、固定サイズのデータを扱うか、生のメモリを直接扱うことしかできませんでした。今回、ページフレームアロケータができたことで、mallocとfreeを実装してカーネルで使用できるようになりました。

Kernighan and Ritchie [8]の本には、私たちがインスピレーションを得られるような実装例が載っています。唯一の修正点は、より多くのメモリが必要な場合に、sbrk/brkの呼び出しをページフレームアロケータ

タの呼び出しに置き換えることです。また、ページフレームアロケータから返されたページフレームを仮想アドレスにマッピングすることも忘れてはなりません。正しい実装では、十分な大きさのメモリブロックが解放されたときに、freeの呼び出しでページフレームアロケータにページフレームを返します。

## Further reading

- The OSDev wiki page on page frame allocation: [http://wiki.osdev.org/Page\\_Frame\\_Allocation](http://wiki.osdev.org/Page_Frame_Allocation)



# 第11章

## ユーザーモード

ユーザーモードはもうすぐ手の届くところまで来ていますが、そこに至るまでにはあといくつかのステップがあります。この章で紹介されている手順は簡単に見えるかもしれませんが、小さなエラーが見つけないとバグの原因となる場所がたくさんあるので、実装するのは難しいかもしれません。

### Segments for User Mode

ユーザーモードを有効にするには、GDT にさらに 2 つのセグメントを追加する必要があります。これらは、セグメンテーションの章でGDTを設定したときに追加したカーネルセグメントとよく似ています。

Index	Offset	Name	Address range	Type	DPL
3	0x18	user code segment	0x00000000 - 0xFFFFFFFF	RX	PL3
4	0x20	user data segment	0x00000000 - 0xFFFFFFFF	RW	PL3

表11.1: ユーザーモードに必要なセグメントディスクリプターです。

違いはDPLで、PL3でコードを実行できるようになりました。セグメントはまだアドレス空間全体を扱うことができますが、これらのセグメントをユーザーモードコードに使用するだけでは、カーネルを保護することはできません。そのためにはページングが必要です。

### Setting Up For User Mode

ユーザーモードのプロセスには、いくつか必要なものがあります。

- コード、データ、スタック用のページフレームです。現時点では、スタック用に1つのページフレームを割り当て、プログラムのコードに合わせて十分なページフレームを割り当てれば十分です。今の時点では、大きくしたり小さくしたりできるスタックの設定を気にする必要はなく、まずは基本的な実装の作業に集中してください。

- The binary from the GRUB module has to be copied to the page frames used for the programs code.  
上述のページフレームをメモリにマッピングするためには、ページディレクトリとページテーブルが必要です。コードとデータが0x00000000にマッピングされる必要があるため、少なくとも2つのページテーブルが必要です。

また、スタックはカーネルのすぐ下、0xBFFFFFFBから始まり、低いアドレスに向かって増えていくはずです。PL3へのアクセスを許可するには、U/Sフラグを設定する必要があります。

この情報は、プロセスを表す構造体に格納しておくると便利な場合があります。このプロセス構造体は、カーネルの malloc 関数で動的に割り当てることができます。

## Entering User Mode

現在の特権レベル (CPL) よりも低い特権レベルでコードを実行する唯一の方法は、iret命令またはret命令 (それぞれ割り込みリターンまたはロングリターン) を実行することです。

ユーザーモードに入るには、プロセッサが特権レベル間の割り込みを発生させたかのように、スタックを設定します。スタックは以下のようになります。

```
[esp + 16]  ss      ; the stack segment selector we want for user mode
[esp + 12]  esp      ; the user mode stack pointer
[esp + 8]   eflags   ; the control flags we want to use in user mode
[esp + 4]   cs       ; the code segment selector
[esp + 0]   eip      ; the instruction pointer of user mode code to execute
```

詳細は、インテルのマニュアル[33]の6.2.1項、図6-4を参照してください。

iret命令は、これらの値をスタックから読み出し、対応するレジスタに入力します。iretを実行する前に、ユーザーモードプロセス用に設定したページディレクトリに変更する必要があります。ここで重要なのは、PDTに切り替えた後もカーネルコードを実行するためには、カーネルがマップインされている必要があるということです。そのためには、0xC0000000以上のデータをすべてマッピングするカーネル用のPDTを別に用意し、切り替え時にそれをユーザーPDT (0xC0000000以下のみマッピング) にマージするという方法があります。なお、レジスタcr3を設定する際には、PDTの物理アドレスを使用する必要があります。

レジスタeflagsには、インテルのマニュアル[33]の2.3節で規定されているさまざまなフラグが含まれています。ここで最も重要なのは、割り込みイネーブル(IF)フラグです。アセンブリコード命令stiは、割り込みを有効にするために特権レベル3では使用できません。ユーザーモードに入るときに割り込みが禁止されていれば、ユーザーモードに入っても割り込みを有効にすることはできません。スタックの eflags エントリに IF フラグを設定すると、アセンブリコード命令 iret によってレジスタ eflags がスタック上の対応する値に設定されるため、ユーザーモードでの割り込みが有効になります。

今のところ、割り込みは無効にしておきましょう。なぜなら、特権間レベルの割り込みを適切に動作させるには、もう少し作業が必要だからです (「システムコール」の項を参照)。

スタック上の値 eip は、ユーザコードのエントリポイントを指します - この例では 0x00000000 です。スタック上の値espは、スタックの開始点である0xBFFFFFFB(0xC0000000 - 4)を指します。

スタック上の値 cs と ss は、それぞれユーザーコードセグメントとユーザーデータセグメントのセグメントセレクトタになります。セグメント化の章で説明したように、セグメントセレクトタの最下位 2 ビットは RPL (要求された特権レベル) です。iret を使用して PL3 に入る場合、cs および ss の RPL は 0x3 であ



る必要があります。次のコードはその例です。

```
USER_MODE_CODE_SEGMENT_SELECTOR equ 0x18
USER_MODE_DATA_SEGMENT_SELECTOR equ 0x20
mov cs, USER_MODE_CODE_SEGMENT_SELECTOR |
0x3 mov ss, USER_MODE_DATA_SEGMENT_SELECTOR
| 0x3
```

レジスタdsと他のデータセグメントレジスタは、ssと同じセグメントセクタに設定する必要があります。これらは通常の方法であるmovアセンブリコード命令で設定できます。これで iret を実行する準備が整いました。すべてが正しく設定されていれば、ユーザーモードに入ることができるカーネルができあがるはずです。

## Using C for User Mode Programs

ユーザーモードプログラムのプログラミング言語としてC言語を使用する場合、コンパイル結果のファイルの構造を考えることが重要です。

カーネル実行ファイルのファイルフォーマットとしてELF[18]を使用できるのは、GRUBがELFファイルフォーマットの解析・解釈方法を知っているからです。ELFパーサーを実装すれば、ユーザーモードのプログラムをELFバイナリにコンパイルすることもできます。これは読者のための課題として残しておきます。

ユーザーモードプログラムの開発を容易にするためにできることの一つは、プログラムをCで書くことを認め、ELFバイナリではなくフラットバイナリにコンパイルすることです。C言語では、生成されるコードのレイアウトが予測できず、エントリーポイントであるmainがバイナリのオフセット0にない場合があります。これを回避する一般的な方法は、オフセット0にmainを呼び出すアセンブリコードを数行追加することです。

```
extern メイ  
ンセクシヨ  
ン .text  
    ; push argv  
    ; push  
    argc call  
    main  
    ; main has returned, eax is return value  
    jmp $    ; loop forever
```

このコードがstart.sというファイルに保存されている場合、以下のコードは、これらの命令を実行ファイルの最初に配置するリンカースクリプトの例を示しています（start.sは、start.oにコンパイルされることを覚えておいてください）。

```
OUTPUT_FORMAT("binary")    /* output flat binary */  
  
SECTIONS  
{  
    . = 0;                /* relocate to address 0 */  
    .テキスト ALIGN(4):
```

```

{
    start.o(.text)    /* include the .text section of start.o */
    *(.text)          /* include all other .text sections */
}
.データ ALIGN(4):
{
    *(.data)
}
.rodata ALIGN(4)です。
{

```

```

        *(.rodata*)
    }
}

```

注：*\*(.text)*は、*start.o*の*.text*セクションを再び含むことはありません。

このスクリプトを使えば、Cやアセンブラ（あるいはldでリンク可能なオブジェクトファイルにコンパイルできるその他の言語）でプログラムを書くことができ、カーネルのロードやマッピングも簡単にできます（*.rotext*は書き込み可能な状態でマッピングされますが）。ユーザープログラムをコンパイルする際には、以下のようなGCCフラグが必要です。

```

-m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles
-nosystemlib

```

リンクの際には、以下のフラグを使用します。

```

-T link.ld -melf_i386 # emulate 32 bits ELF, the binary output is specified
# リンカースクリプトで

```

オプション *-T* は、リンカスクリプト *link.ld* を使用するよう指示します。

## AC Library

自分のプログラムに小さな「標準ライブラリ」を書くことを考えてみるのも面白いかもしれません。機能の中には、システムコールを必要とするものもありますが、*string.h*の関数のようにシステムコールを必要としないものもあります。

## Further Reading

- Gustavo Duarte has an article on privilege levels: <http://duartes.org/gustavo/blog/post/cpu-rings-privilege-and-protection>

## 第12章

# ファイルシステム

オペレーティングシステムにファイルシステムを導入する必要はありませんが、ファイルシステムは非常に使いやすい抽象化であり、多くのオペレーティングシステム、特にUNIX系のオペレーティングシステムでは中心的な役割を果たしています。複数のプロセスやシステムコールをサポートするプロセスを開始する前に、簡単なファイルシステムを実装することを検討するとよいでしょう。

### Why a File System?

OSで実行するプログラムをどのように指定するか？最初に実行されるプログラムはどれですか？プログラムはどのようにしてデータを出力したり、入力を読み取ったりするのですか？

ほぼすべてのものがファイルであるという規則があるUNIX系システムでは、これらの問題はファイルシステムによって解決されます。（このアイデアをさらに一步進めたPlan 9プロジェクトについても少し読んでみると面白いかもしれません）。

### A Simple Read-Only File System

最もシンプルなファイルシステムは、私たちがすでに持っているものかもしれません。つまり、RAM上のみ存在する1つのファイルで、カーネルが起動する前にGRUBによって読み込まれます。カーネルやオペレーティング・システムが成長すると、これでは制限が多すぎるでしょう。

1つのファイルのビットだけではなく、少し進んだファイルシステムとして、メタデータ付きのファイルがある。メタデータには、ファイルの種類、ファイルのサイズなどを記述することができる。ビルド時に実行されるユーティリティプログラムを作成して、このメタデータをファイルに追加することができる。このようにして、メタデータが付加された複数のファイルを1つの大きなファイルに連結することで、「ファイルの中のファイルシステム」を構築することができる。この方法では、（GRUBがファイルを読み込んだ後は）メモリ上に存在する読み取り専用のファイルシステムとなります。

ファイルシステムを構築するプログラムは、ホストシステム上のディレクトリをトラバースし、すべて

のサブディレクトリとファイルをターゲットファイルシステムの一部として追加することができます。ファイルシステムの各オブジェクト（ディレクトリやファイル）は、ヘッダとボディから構成されます。ファイルのボディは実際のファイルであり、ディレクトリのボディは他のファイルやディレクトリの名前と「アドレス」というエントリのリストです。

このファイルシステムの各オブジェクトは連続したものになるので、カーネルがメモリから読み取るのが容易になります。また、すべてのオブジェクトのサイズは固定されているため（最後のオブジェクトは成長可能）、新しいファイルを追加したり、既存のファイルを変更したりすることは困難です。

## Inodes and Writable File Systems

書き込み可能なファイルシステムの必要性が生じた場合、inodeの概念を検討することは良いアイデアです。推奨文献については、「その他の文献」の項を参照してください。

## A Virtual File System

スクリーンやキーボードなどのデバイスへの読み書きには、どのような抽象化が必要ですか？

仮想ファイルシステム（VFS）は、具体的なファイルシステムの上に抽象化したものです。VFSは主にパスシステムとファイル階層を提供し、ファイルに対する操作を基礎となるファイルシステムに委ねます。VFSに関するオリジナルの論文は簡潔で、一読の価値があります。参考文献として「Further Reading」の項を参照してください。

VFSを使えば、/devというパスに特別なファイルシステムをマウントすることができます。このファイルシステムでは、キーボードやコンソールなどのすべてのデバイスを扱うことができます。しかし、伝統的なUNIXのアプローチをとることもできます。メジャー/マイナーのデバイス番号をつけ、mknodでデバイス用の特別なファイルを作成します。どのアプローチが最も適切だと思うかは、あなた次第です。抽象化レイヤーを構築する際に、正しいか間違っているかはありません（ただし、抽象化が他のものよりもはるかに有用であることは事実です）。

## Further Reading

<http://plan9.bell-labs.com/plan9/index.html>にあるPlan 9 OSの考え方は一見の価値があります。

ウィキペディアのinodeに関するページ：<http://en.wikipedia.org/wiki/Inode>、inodeポインタ構造：[http://en.wikipedia.org/wiki/Inode\\_pointer\\_structure](http://en.wikipedia.org/wiki/Inode_pointer_structure)。

vnodesと仮想ファイルシステムのコンセプトに関するオリジナルの論文は非常に興味深いものです。  
[//www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf](http://www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf)

Poul-Henning Kamp は、[http://static.usenix.org/](http://static.usenix.org/publications/library/proceedings/bsdcon02/full_papers/kamp/kamp_html/index.html)

[publications/library/proceedings/bsdcon02/full\\_papers/kamp/kamp\\_html/index.html](http://static.usenix.org/publications/library/proceedings/bsdcon02/full_papers/kamp/kamp_html/index.html) で、/dev のための特別なファイルシステムのアイデアを議論しています。

## 第13章

# システムコール

システムコールとは、ユーザーモードのアプリケーションがカーネルと対話するための手段であり、リソースの要求や実行すべき操作の要求などを行います。システムコールAPIは、カーネルの中で最もユーザーの目に触れる部分であるため、その設計には工夫が必要です。

## Designing System Calls

アプリケーション開発者が使えるシステムコールを設計するのは、私たちカーネル開発者の役目です。POSIX標準規格からインスピレーションを得ることもできますし、大変そうであればLinux用の規格を見て、選択することもできます。参考文献については、本章最後の「参考文献」の項をご覧ください。

## Implementing System Calls

システムコールは、伝統的にソフトウェア割り込みで呼び出されます。ユーザーアプリケーションは、適切な値をレジスタやスタックに格納した後、あらかじめ定義された割り込みを開始し、実行をカーネルに移します。使用される割り込み番号はカーネルに依存しますが、Linuxでは0x80という番号を使用して、割り込みがシステムコールを意図したものであることを識別します。

システムコールが実行されると、通常、現在の特権レベルがPL3からPL0に変更されます（アプリケーションがユーザーモードで実行されている場合）。これを可能にするには、システムコール割り込みのIDT内のエントリのDPLがPL3アクセスを許可する必要があります。

権限間レベルの割り込みが発生するたびに、プロセッサはいくつかの重要なレジスタをスタックにプッシュします。これは、以前にユーザーモードに入るときに使用したものと同じです（Intelマニュアル[33]の図6-4、6.12.1セクションを参照）。どのようなスタックが使用されるのでしょうか？33]の同じセクションでは、割り込みによって数値的に低い特権レベルのコードが実行された場合、スタックスイッチが発生することが明記されています。レジスタssとespの新しい値は、現在のタスクステートセグメント



(TSS)からロードされます。TSSの構造は、インテルのマニュアル[33]の7.2.1節、図7-2に規定されています。

システムコールを有効にするには、ユーザーモードに入る前にTSSを設定する必要があります。この設定は、C言語でTSSを表す「packed struct」のss0およびesp0フィールドを設定することで行うことができます。パックされた構造体をプロセッサにロードする前に、TSS記述子をGDTに追加する必要があります。TSS記述子の構造は、[33]の7.2.2項に記載されています。

現在のTSSセグメントセレクタを指定するには、ltrアセンブリコード命令でtrレジスタにロードします。TSSセグメントディスクリプターのインデックスが5、つまりオフセット $5 * 8 = 40 = 0x28$ の場合、これがレジスタtrにロードされるべき値です。

以前、「ユーザーモードへの移行」の章でユーザーモードに移行した際、PL3で実行する際に割り込みを無効にしました。システムコールは割り込みを使用して実装されているため、ユーザーモードでは割り込みを有効にする必要があります。スタック上のeflags値にIFフラグビットを設定することで、iretは割り込みを有効にします（スタック上のeflags値は、アセンブリコード命令iretによってeflagsレジスタにロードされるため）。

## Further Reading

POSIXに関するウィキペディアのページ。仕様書へのリンクもあります。

<http://en.wikipedia.org/wiki/POSIX> Linuxで使われるシステムコールのリスト。

<http://bluemaster.iu.hio.no/edu/dark/lin-asm/syscalls.html>

システムコールに関するウィキペディアのページ：[http://en.wikipedia.org/wiki/System\\_call](http://en.wikipedia.org/wiki/System_call)

インテルのマニュアル[33]の割り込み（第6章）とTSS（第7章）のセクションでは、必要なすべての詳細を得ることができます。

## 第14章

# マルチタスキング

複数のプロセスを同時に実行しているように見せるにはどうしたらいいですか？今日、この質問には2つの答えがあります。

マルチコアプロセッサや、マルチプロセッサを搭載したシステムでは、2つのプロセスを異なるコアやプロセッサで実行することで、実際に2つのプロセスを同時に実行することができます。フェイクする。つまり、プロセスを高速に（人間が気づかないほどの速さで）切り替えるのです。常に1つのプロセスしか実行されていませんが、高速に切り替えることで「同時に」実行されているように見せかけます。

本書で作成したOSは、マルチコアプロセッサやマルチプロセッサに対応していないので、偽装するしかありません。オペレーティングシステムの中で、プロセスを高速に切り替える役割を担う部分をスケジューリングアルゴリズムと呼びます。

### Creating New Processes

新しいプロセスを作成するには、通常、fork と exec という 2 種類のシステムコールを使用します。fork は、現在実行中のプロセスの正確なコピーを作成し、exec は、現在のプロセスを、ファイルシステム内のプログラムの場所へのパスで指定されたプロセスに置き換えます。このシステムコールは、「ユーザーモード」の章の「ユーザーモードの設定」で説明されているのとほぼ同じ手順で実行されるからです。

### Cooperative Scheduling with Yielding

プロセス間の迅速な切り替えを実現する最も簡単な方法は、プロセス自身が切り替えを担当することです。プロセスはしばらく動作した後、OSに（システムコールを介して）他のプロセスに切り替えてよいと伝えます。CPUの制御を他のプロセスに譲ることを「イールド」といい、プロセス自身がスケジューリングを担当する場合は、すべてのプロセスが互いに協力しなければならないことから「協調型スケジ

ューリング」と呼ばれます。

プロセスが降伏する際には、プロセスの全状態（すべてのレジスタ）を保存する必要があり、できればプロセスを表す構造体としてカーネルヒープ上に保存します。新しいプロセスに移行する際には、保存した値からすべてのレジスタを復元する必要があります。

スケジューリングは、どのプロセスが実行されているかのリストを保持することで実装できます。システムコールの収量は、リストの中の次のプロセスを実行し、現在のプロセスを最後にする必要があります（他のスキームも可能ですが、これは単純なものです）。

新しいプロセスへの制御の移行は、「ユーザーモード」の章の「ユーザーモードへの移行」の項で説明したのとまったく同じ方法で、「iret」というアセンブリコード命令を介して行われます。

協調型スケジューリングの実装により、複数プロセスのサポートを開始することを強く推奨します。さらに、プリエンプティブスケジューリングを実装する前に、exec、fork、yieldの両方が動作するソリューションを用意することをお勧めします。協調スケジューリングは決定論的であるため、プリエンプティブスケジューリングに比べて、デバッグが非常に容易です。

## Preemptive Scheduling with Interrupts

OSは、プロセスが他のプロセスに切り替わるタイミングをプロセス自身に管理させるのではなく、短時間で自動的にプロセスを切り替えることができます。OSは、プログラマブル・インターバル・タイマー（PIT）を設定して、20msなどの短時間後に割り込みを発生させることができます。PIT割り込みの割り込みハンドラでは、OSは実行中のプロセスを新しいプロセスに変更します。このようにして、プロセス自体はスケジューリングを気にする必要がありません。このようなスケジューリングをプリエンプティブ・スケジューリングといいます。

## Programmable Interval Timer

プリエンプティブ・スケジューリングを行うためには、PITはまず、xミリ秒ごとに割り込みを発生させるように設定しなければなりません。

PITの構成は、他のハードウェアデバイスの構成と非常によく似ており、I/Oポートにバイトが送られます。PITのコマンドポートは0x43です。すべての設定オプションについては、OSDevのPITに関する記事[39]をご覧ください。我々は以下のオプションを使用しています。

- Raise interrupts (use channel 0)
- Send the divider as low byte then high byte (see next section for an explanation)
- Use a square wave
- Use binary mode

この結果、コンフィグレーションバイト00110110となります。

割り込みを発生させる間隔の設定は、シリアルポートと同様にデバイダを使って行います。PITに割り込みの頻度を示す値（ミリ秒単位）を送る代わりに、分周器を送ります。PITは、デフォルトでは1193182Hzで動作します。分周器を10回送ると、PITは $1193182 / 10 = 119318 \text{ Hz}$ で動作します。分周器は16ビットしか使えないので、タイマーの周波数は $1193182\text{Hz} \sim 1193182 / 65535 = 18.2\text{Hz}$ の間でしか設定できません。そこで、ミリ秒単位の間隔を受け取り、それを正しい分周器に変換する関数を作成することをお勧めします。

分周器はPITのチャンネル0データI/Oポートに送られますが、一度に送れるのは1バイトだけなので、まず分周器の下位8ビットを送り、次に分周器の上位8ビットを送ればよいのです。チャンネル0のデータI/Oポートは0x40にあります。詳細は、OSDevの記事[39]を参照してください。

## **Separate Kernel Stacks for Processes**

すべてのプロセスが同じカーネルスタック(TSSで公開されているスタック)を使用している場合、カーネルモードのままプロセスが中断されると問題が発生します。切り替えられたプロセスは、同じカーネルを使用することになります。

スタックに書き込まれた前のプロセスの内容を上書きします（TSSのデータ構造はスタックの先頭を指していることを覚えておいてください）。

この問題を解決するには、各プロセスが独自のユーザーモードスタックを持つと同様に、各プロセスが独自のカーネルスタックを持つ必要があります。プロセスを切り替える際には、新しいプロセスのカーネルスタックを指すようにTSSを更新する必要があります。

## Difficulties with Preemptive Scheduling

プリエンプティブ・スケジューリングを使用する場合、協調型スケジューリングにはない問題が発生します。協力型スケジューリングでは、プロセスが降伏するたびにユーザーモード(特権レベル3)でなければなりません。プリエンプティブスケジューリングでは、プロセスはユーザーモードでもカーネルモード(特権レベル0)でも割り込みが可能です。これはプロセス自身がいつ割り込みを受けるかを制御しないからです。

カーネルモードでのプロセスの割り込みは、CPUが割り込み時にスタックを設定する方法のため、ユーザーモードでのプロセスの割り込みとは少し異なります。特権レベルの変更が発生した場合（ユーザーモードでプロセスが中断された場合）、CPUはプロセスのssとespレジスタの値をスタックにプッシュします。特権レベルの変更が発生しなかった場合（カーネルモードでプロセスが中断された場合）、CPUはespレジスタをスタックにプッシュしません。さらに、特権レベルの変更がなかった場合、CPUはスタックをTSSで定義されたものに変更しません。

この問題は、割り込みの前のespの値を計算することで解決します。権限の変更がない場合、CPUはスタックに3つのものをプッシュすることがわかっており、自分がどれだけスタックにプッシュしたかもわかっているため、割り込み時にespの値がどうなっていたかを計算することができます。これは、特権レベルの変更がない場合、CPUはスタックを変更しないので、espの内容は割り込みの時と同じになります。さらに複雑なのは、カーネルモードで実行されるべき新しいプロセスに切り替えたときのケースをどう扱うかを考えなければなりません。iretは特権レベルの変更なしに使用されているため、CPUはスタックに置かれたespの値を更新せず、自分でespを更新する必要があります。

## Further Reading

さまざまなスケジューリング・アルゴリズムについては、「[http://wiki.osdev.org/Scheduling\\_Algorithms](http://wiki.osdev.org/Scheduling_Algorithms)」を参照してください。





# References

- [1] Andrew Tanenbaum, 2007. *Modern operating systems, 3rd edition*. Prentice Hall, Inc.,
- [2] *The royal institute of technology*, <http://www.kth.se>,
- [3] Wikipedia, *Hexadecimal*, <http://en.wikipedia.org/wiki/Hexadecimal>,
- [4] OSDev, *OSDev*, [http://wiki.osdev.org/Main\\_Page](http://wiki.osdev.org/Main_Page),
- [5] James Molloy, *James m's kernel development tutorial*, [http://www.jamesmolloy.co.uk/tutorial\\_html/](http://www.jamesmolloy.co.uk/tutorial_html/),
- [6] Canonical Ltd, *Ubuntu*, <http://www.ubuntu.com/>,
- [7] Oracle, *Oracle vM virtualBox*, <http://www.virtualbox.org/>,
- [8] Dennis M. Ritchie Brian W. Kernighan, 1988. *The c programming language, second edition*. Prentice Hall, Inc.,
- [9] Wikipedia, *C (programming language)*, [http://en.wikipedia.org/wiki/C\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/C_(programming_language)),
- [10] Free Software Foundation, *GCC, the gNU compiler collection*, <http://gcc.gnu.org/>,
- [11] NASM, *NASM: The netwide assembler*, <http://www.nasm.us/>,
- [12] Wikipedia, *Bash*, [http://en.wikipedia.org/wiki/Bash\\_%28Unix\\_shell%29](http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29),
- [13] Free Software Foundation, *GNU make*, <http://www.gnu.org/software/make/>,
- [14] Volker Ruppert, *bochs: The open souce iA-32 emulation project*, <http://bochs.sourceforge.net/>,
- [15] QEMU, *QEMU*, [http://wiki.qemu.org/Main\\_Page](http://wiki.qemu.org/Main_Page),
- [16] Wikipedia, *BIOS*, <https://en.wikipedia.org/wiki/BIOS>,
- [17] Free Software Foundation, *GNU gRUB*, <http://www.gnu.org/software/grub/>,
- [18] Wikipedia, *Executable and linkable format*, [http://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format),
- [19] Free Software Foundation, *Multiboot specification version 0.6.96*, <http://www.gnu.org/software/grub/manual/multiboot/multiboot.html>,
- [20] GNU, *GNU binutils*, <http://www.gnu.org/software/binutils/>,
- [21] Lars Nodeen, *Bug #426419: configure: error: GRUB requires a working absolute objcopy*, <https://bugs.launchpad.net/ubuntu/+source/grub/+bug/426419>,
- [22] [//bugs.launchpad.net/ubuntu/+source/grub/+bug/426419](https://bugs.launchpad.net/ubuntu/+source/grub/+bug/426419),
- [23] Wikipedia, *ISO image*, [http://en.wikipedia.org/wiki/ISO\\_image](http://en.wikipedia.org/wiki/ISO_image),
- [24] Bochs, *bochsrc*, <http://bochs.sourceforge.net/doc/docbook/user/bochsrc.html>,
- [25] NASM, *RESB and friends: Declaring uninitialized data*, <http://www.nasm.us/doc/nasmdoc3.htm>,

- [26] Wikipedia, *x86 calling conventions*, [http://en.wikipedia.org/wiki/X86\\_calling\\_conventions](http://en.wikipedia.org/wiki/X86_calling_conventions),
- [27] Wikipedia, *Framebuffer*, <http://en.wikipedia.org/wiki/Framebuffer>,
- [28] Wikipedia, *VGA-compatible text mode*, [http://en.wikipedia.org/wiki/VGA-compatible\\_text\\_mode](http://en.wikipedia.org/wiki/VGA-compatible_text_mode),
- [29] Wikipedia, *ASCII*, <https://en.wikipedia.org/wiki/Ascii>,
- [30] OSDev, *VGA hardware*, [http://wiki.osdev.org/VGA\\_Hardware](http://wiki.osdev.org/VGA_Hardware),
- [31] Wikipedia, *Serial port*, [http://en.wikipedia.org/wiki/Serial\\_port](http://en.wikipedia.org/wiki/Serial_port),
- [32] OSDev, *Serial ports*, [http://wiki.osdev.org/Serial\\_ports](http://wiki.osdev.org/Serial_ports),
- [33] WikiBooks, *Serial programming/8250 uART programming*, [http://en.wikibooks.org/wiki/Serial\\_Programming/8250\\_UART\\_Programming](http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming),
- [34] Intel, *Intel 64 and iA-32 architectures software developer's manual vol. 3A*, <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html/>,
- [35] NASM, *Multi-line macros*, <http://www.nasm.us/doc/nasmdoc4.html#section-4.3>,
- [36] SIGOPS, *i386 interrupt handling*, [http://www.acm.uiuc.edu/sigops/roll\\_your\\_own/i386/irq.html](http://www.acm.uiuc.edu/sigops/roll_your_own/i386/irq.html),
- [37] Andries Brouwer, *Keyboard scancodes*, <http://www.win.tue.nl/>,
- [38] Steve Chamberlain, *Using ld, the gNU linker*, [http://www.math.utah.edu/docs/info/ld\\_toc.html](http://www.math.utah.edu/docs/info/ld_toc.html),
- [39] OSDev, *Page frame allocation*, [http://wiki.osdev.org/Page\\_Frame\\_Allocation](http://wiki.osdev.org/Page_Frame_Allocation),
- [40] OSDev, *Programmable interval timer*, [http://wiki.osdev.org/Programmable\\_Interval\\_Timer](http://wiki.osdev.org/Programmable_Interval_Timer),