

A Heavily Commented

Linux Kernel Source Code

Zhao Jiong



www.oldlinux.org

Jiong.Zhao@TongJi.edu.cn

ヘビーコメントされたLinuxカーネルのソースコード

カーネルバージョン0.12

(中国語改訂5版)



Zhao Jiong
Jiong.zhao@TongJi.edu.cn

アブストラクト

本書は、初期のLinuxカーネル（V0.12）のすべてのソースコードについて、詳細かつ包括的なコメントと解説を提供しています。これは、読者が可能な限り短期間でLinuxの動作メカニズムを包括的かつ深く理解し、現代のLinuxシステムをさらに研究するための強固な基礎を築くことを目的としています。解析のバージョンは非常に低いですが、カーネルはコンパイルと実行が可能であり、すでにLinuxの動作原理のエッセンスが含まれています。

本書では、まず、Linuxカーネルの開発の歴史を簡単に紹介し、各カーネルのバージョンと改良点の主な違いを説明し、学習対象として0.12カーネルのソースコードを選んだ理由を述べています。次に、ソースコードを読むために必要な基礎知識を与え、Linuxシステムを実行するPCのハードウェア構造、カーネルが使用するアセンブリ言語、C言語の拡張について概説し、プロテクトモードの80X86プロセッサに焦点を当てています。続いて、カーネルコードの概要を紹介し、カーネルソースのディレクトリツリー構造を示し、すべてのカーネルの組織構造に従って、プログラムとファイルを詳細に説明しています。カーネルの動作原理に対する読者の理解を深めるために、最後の章では、関連するいくつかの動作デバッグテストを行っています。この本に掲載されているすべての関連情報は、ウェブサイト（www.oldlinux.org）からダウンロードすることができます。

本書は、大学でコンピュータを専攻している学生がオペレーティングシステムを学ぶ際の補助的・実用的な教材として、また、カーネルの動作原理を学ぶLinux愛好家の自習用参考書としても適しており、さらに、一般の技術者が組込みシステムを開発する際の参考書としても使用することができます。

著作権表示

この本を修正して正式に出版する権利はすべて著者にあります。読者の皆様からのご意見・ご感想は、電子メール：jiong.zhao@tongji.edu.cn または gohigh@gmail.com、または直接お手紙でも受け付けておりますので、お気軽にお寄せください。School of Mechanical and Energy Engineering, Institute of Mechanical and Electronic Engineering, Tongji University, Address: Room B409, Machinery Building, 4800 Cao'an Road, Shanghai, China: 201804

に捧げられています。

孫紅芳-私の愛する母 趙碧珍-私の最愛の父

私の成長を教えるために、あなたは
あなたの厳しい人生を過ごしてきま
した。

あなたの息子
である趙炯氏

献給

孙洪芳 - 我亲爱的母亲
赵碧臣 - 我敬爱的父亲

在教诲我成长过程中
度过了你们艰辛一生

你们的儿子
赵炯

"RTFSC - Read The Fking Source Code :)"**

-Linus Benedict Torvalds

Table of Contents

| | | | |
|--|------------|--|------------|
| PREFACE..... | 1 | LINUX KERNEL MEMORY MANAGEMENT | 178 |
| THE MAIN GOAL OF THIS BOOK | 1 | INTERRUPT MECHANISM | 193 |
| FEATURES OF THIS BOOK..... | 1 | LINUX SYSTEM CALLS..... | 199 |
| OTHER BENEFITS OF READING EARLY KERNEL CODE..... | 2 | SYSTEM TIME AND TIMING | 201 |
| THE IMPORTANCE AND NECESSITY OF READING THE COMPLETE CODE | 3 | LINUX PROCESS CONTROL..... | 203 |
| HOW TO SELECT THE KERNEL CODE VERSION TO READ | 3 | HOW TO USE THE STACK IN LINUX | 215 |
| THE BASIC KNOWLEDGE REQUIRED BY THE BOOK | 4 | FILE SYSTEM FOR LINUX 0.12 | 219 |
| IS READING AN EARLIER VERSION OUT OF DATE? | 5 | DIRECTORIES OF KERNEL SOURCE CODE | 220 |
| EXT FILE SYSTEM AND MINIX FILE SYSTEM..... | 5 | THE KERNEL CODE AND USER PROGRAMS .229 | |
| 1 OVERVIEW | 7 | 5.12 LINUX/MAKEFILE | 230 |
| 1.1 THE BIRTH AND DEVELOPMENT OF LINUX | 7 | 5.13 SUMMARY | 236 |
| 1.2 CONTENT REVIEW..... | 15 | 6 BOOTING SYSTEM..... | 237 |
| 1.3 SUMMARY..... | 20 | 6.1 MAIN FUNCTIONS..... | 237 |
| 2 MICROCOMPUTER STRUCTURE | 21 | 6.2 BOOTSECT.S | 239 |
| 2.1 THE MICROCOMPUTER COMPOSITION | 22 | 6.3 SETUP.S..... | 254 |
| 2.2 I/O PORT ADDRESSING & ACCESS CONTROL... 23 | | 6.4 HEAD.S..... | 286 |
| 2.3 MAIN MEMORY, BIOS AND CMOS MEMORY 26 | | 6.5 SUMMARY | 299 |
| 2.4 CONTROLLERS AND CONTROL CARDS | 28 | 7 INITIALIZATION PROGRAM (INIT) | 301 |
| 2.5 SUMMARY | 38 | 7.1 MAIN.C | 301 |
| 3 KERNEL PROGRAMMING LANGUAGEAND ENVIRONMENT | 39 | 7.2 ENVIRONMENT INITIALIZATION | 316 |
| 3.1 AS86 ASSEMBLER | 39 | 7.3 SUMMARY | 318 |
| 3.2 GNU AS ASSEMBLER | 46 | 8 KERNEL CODE (KERNEL)..... | 319 |
| 3.3 C LANGUAGE PROGRAM | 58 | 8.1 MAIN FUNCTIONS..... | 319 |
| 3.4 INTERWORKING BETWEEN C AND ASSEMBLY LANGUAGE | 67 | 8.2 ASM.S | 322 |
| 3.5 LINUX 0.12 OBJECT FILE FORMAT..... | 76 | 8.3 TRAPS.C | 329 |
| 3.6 MAKE COMMAND AND MAKEFILE..... | 87 | 8.4 SYS_CALL.S..... | 335 |
| 3.7 SUMMARY | 93 | 8.5 MKTIME.C | 349 |
| 4 80X86 PROTECTION MODE AND ITS PROGRAMMING..... | 94 | 8.6 SCHED.C | 351 |
| 4.1 80X86 SYSTEM REGISTERS AND SYSTEM INSTRUCTIONS | 94 | 8.7 SIGNAL.C..... | 373 |
| 4.2 PROTECT MODE MEMORY MANAGEMENT....101 | | 8.8 EXIT.C | 391 |
| 4.3 SEGMENTATION MECHANISM.....106 | | 8.9 FORK.C | 404 |
| 4.4 PAGING.....119 | | 8.10 SYS.C | 413 |
| 4.5 PROTECTION | 124 | 8.11 VSPRINTF.C..... | 429 |
| 4.6 INTERRUPT AND EXCEPTION HANDLING.....136 | | 8.12 PRINTK.C | 438 |
| 4.7 TASK MANAGEMENT | 147 | 8.13 PANIC.C | 439 |
| 4.8 THE INITIALIZATION OF PROTECTED MODE 157 | | 8.14 SUMMARY | 440 |
| 4.9 A SIMPLE MULTITASK KERNEL EXAMPLE....161 | | 9 BLOCK DEVICE DRIVER | 441 |
| 4.10 SUMMARY | 173 | 9.1 MAIN FUNCTIONS..... | 442 |
| 5 LINUX KERNEL ARCHITECTURE | 175 | 9.2 BLK.H | 446 |
| 5.1 LINUX KERNEL MODE.....175 | | 9.3 HD.C | 451 |
| 5.2 LINUX KERNEL SYSTEM ARCHITECTURE | 176 | 9.4 LL_RW_BLK.C | 477 |
| 5.3 | | 9.5 RAMDISK.C | 485 |
| | | 9.6 FLOPPY.C | 491 |
| | | 9.7 SUMMARY | 521 |
| 10 CHARACTER DEVICE DRIVER | 523 | | |
| 10.1 MAIN FUNCTIONS..... | 523 | | |
| 10.2 KEYBOARD.S..... | 535 | | |

Table of Contents

| | | |
|-----------|--|-------------|
| 10.3 | CONSOLE.C..... | 555 |
| 10.4 | SERIAL.C..... | 594 |
| 10.5 | RS_IO.S..... | 603 |
| 10.6 | TTY_IO.C..... | 608 |
| 10.7 | TTY_IOCTL.C..... | 626 |
| 10.8 | SUMMARY..... | 635 |
| 11 | MATH COPROCESSOR (MATH)..... | 637 |
| 11.1 | FUNCTION DESCRIPTION..... | 637 |
| 11.2 | MATH-EMULATION.C..... | 647 |
| 11.3 | ERROR.C..... | 660 |
| 11.4 | EA.C..... | 661 |
| 11.5 | CONVERT.C..... | 665 |
| 11.6 | ADD.C..... | 670 |
| 11.7 | COMPARE.C..... | 673 |
| 11.8 | GET_PUT.C..... | 675 |
| 11.9 | MUL.C..... | 682 |
| 11.10 | DIV.C..... | 684 |
| 11.11 | SUMMARY..... | 686 |
| 12 | FILE SYSTEM (FS)..... | 689 |
| 12.1 | MAIN FUNCTIONS..... | 689 |
| 12.2 | BUFFER.C..... | 708 |
| 12.3 | BITMAP.C..... | 728 |
| 12.4 | TRUNCATE.C..... | 735 |
| 12.5 | INODE.C..... | 738 |
| 12.6 | SUPER.C..... | 752 |
| 12.7 | NAMEI.C..... | 763 |
| 12.8 | FILE_TABLE.C..... | 793 |
| 12.9 | BLOCK_DEV.C..... | 793 |
| 12.10 | FILE_DEV.C..... | 798 |
| 12.11 | PIPE.C..... | 802 |
| 12.12 | CHAR_DEV.C..... | 807 |
| 12.13 | READ_WRITE.C..... | 810 |
| 12.14 | OPEN.C..... | 817 |
| 12.15 | EXEC.C..... | 826 |
| 12.16 | STAT.C..... | 845 |
| 12.17 | FCNTL.C..... | 848 |
| 12.18 | IOCTL.C..... | 852 |
| 12.19 | SELECT.C..... | 854 |
| 12.20 | SUMMARY..... | 868 |
| 13 | MEMORY MANAGEMENT (MM)..... | 869 |
| 13.1 | MAIN FUNCTIONALITIES..... | 869 |
| 13.2 | MEMORY.C..... | 879 |
| 13.3 | PAGE.S..... | 901 |
| 13.4 | SWAP.C..... | 902 |
| 13.5 | SUMMARY..... | 912 |
| 14 | HEADER FILES (INCLUDE) | 913 |
| 14.1 | FILES IN THE INCLUDE/ DIRECTORY | 914 |
| 14.2 | A.OUT.H..... | 915 |
| 14.3 | CONST.H..... | 926 |
| 14.4 | CTYPE.H..... | 926 |
| 14.5 | ERRNO.H..... | 928 |
| 14.6 | FCNTL.H..... | 930 |
| 14.7 | SIGNAL.H..... | 931 |
| 14.8 | STDARG.H..... | 934 |
| 14.9 | STDDEF.H..... | 936 |
| 14.10 | STRING.H..... | 937 |
| 14.11 | TERMIOS.H..... | 947 |
| 14.12 | TIME.H..... | 954 |
| 14.13 | UNISTD.H..... | 956 |
| 14.14 | UTIME.H..... | 963 |
| 14.15 | FILES IN THE INCLUDE/ASM/ DIRECTORY | 964 |
| 14.16 | IO.H..... | 964 |
| 14.17 | MEMORY.H..... | 965 |
| 14.18 | SEGMENT.H..... | 966 |
| 14.19 | SYSTEM.H | 968 |
| 14.20 | FILES IN THE DIRECTORY INCLUDE/LINUX/ | 973 |
| 14.21 | CONFIG.H | 973 |
| 14.22 | FDREG.H..... | 975 |
| 14.23 | FS.H..... | 977 |
| 14.24 | HDREG.H..... | 982 |
| 14.25 | HEAD.H..... | 985 |
| 14.26 | KERNEL.H | 986 |
| 14.27 | MATH_EMU.H | 987 |
| 14.28 | MM.H | 991 |
| 14.29 | SCHED.H | 993 |
| 14.30 | SYS.H | 1002 |
| 14.31 | TTY.H..... | 1004 |
| 14.32 | HEADER FILES IN THE INCLUDE/SYS/ DIRECTORY | 1008 |
| 14.33 | PARAM.H | 1008 |
| 14.34 | RESOURCE.H | 1009 |
| 14.35 | STAT.H..... | 1011 |
| 14.36 | TIME.H..... | 1013 |
| 14.37 | TIMES.H | 1014 |
| 14.38 | TYPES.H | 1015 |
| 14.39 | UTSNAME.H | 1016 |
| 14.40 | WAIT.H | 1017 |
| 14.41 | SUMMARY..... | 1018 |
| 15 | LIBRARY FILES (LIB)..... | 1019 |
| 15.1 | _EXIT.C | 1020 |
| 15.2 | CLOSE.C..... | 1021 |
| 15.3 | CTYPE.C..... | 1021 |
| 15.4 | DUP.C..... | 1022 |
| 15.5 | ERRNO.C..... | 1023 |
| 15.6 | EXECVE.C..... | 1023 |
| 15.7 | MALLOC.C..... | 1024 |
| 15.8 | OPEN.C..... | 1032 |
| 15.9 | SETSID.C..... | 1034 |
| 15.10 | STRING.C..... | 1034 |
| 15.11 | WAIT.C..... | 1035 |
| 15.12 | WRITE.C..... | 1036 |
| 15.13 | SUMMARY..... | 1037 |
| 16 | BUILDING KERNEL (TOOLS)..... | 1039 |
| 16.1 | BUILD.C | 1039 |
| 16.2 | SUMMARY..... | 1047 |
| 17 | EXPERIMENTAL ENVIRONMENT SETTINGS AND USAGE | 1048 |
| 17.1 | BOCHS SIMULATION SOFTWARE | 1049 |
| 17.2 | RUNNING LINUX 0.1X SYSTEM IN BOCHS | 1054 |
| 17.3 | ACCESS INFORMATION IN A DISK IMAGE FILE | 1059 |
| 17.4 | COMPILING AND RUNNING THE SIMPLE KERNEL | 1062 |

| | | |
|-------|---------------------------------------|-------------|
| 17.5 | USING BOCHS TO DEBUG THE KERNEL | 1065 |
| 17.6 | CREATING A DISK IMAGE FILE | 1073 |
| 17.7 | MAKING A ROOTFILE SYSTEM..... | 1076 |
| 17.8 | COMPILE KERNEL ON LINUX 0.12 SYSTEM | 1084 |
| 17.9 | COMPILE KERNEL UNDER REDHAT SYSTEM | 1085 |
| 17.10 | INTEGRATED BOOT DISK AND ROOT FS | 1089 |
| 17.11 | DEBUGGING KERNEL CODE WITH GDB AND | |
| 17.12 | BOCHS | 1094 |
| 17.13 | SUMMARY..... | 1100 |
| | REFERENCES | 1101 |
| | APPENDIX..... | 1103 |
| A1 | ASCII CODE TABLE..... | 1103 |
| A2 | COMMON C0, C1 CONTROL CHARACTERS | 1104 |
| A3 | ESCAPE AND CONTROL SEQUENCES | 1106 |
| A4 | THE FIRST SET OF KEYBOARD SCANCODE | 1109 |

Preface

モノづくりのインテリジェント化やネットワークによるモノの直接制御という一般的な流れの中で、Linuxオペレーティングシステムは、今日の組み込みシステムにおける動作制御のための最も重要な基本プラットフォームとなっています。本書は、Linuxオペレーティングシステムのカーネルの基本的な仕組みを解説した入門書です。

The main goal of this book

本書の主な目的は、最小限のスペース、あるいは限られたスペースの中で、完全なLinuxカーネルのソースコードを解剖し、オペレーティングシステムの基本機能と実際の実装を完全に理解することです。Linux カーネルを完全かつ深く理解するために、Linux オペレーティングシステムの基本的な動作原理の真の理解と導入を行います。

本書の読者層は、Linuxシステムの一般的な使い方を知っていたり、一定のプログラミングの基礎を持っているが、現在の新しいカーネルコードを読むための基礎知識が不足しており、一刻も早くUNIX OSのカーネルの動作原理と実際のコードを理解したいと考えている人に位置づけられる。

Features of this book

本書を執筆している時点では、Linuxカーネルを解説した書籍の中には、新しいLinuxカーネルのバージョン（Fedora 8が採用しているバージョン2.6.24など）を使って、カーネルの動作メカニズムを説明しようとしているものがあります。しかし、カーネルのソースコードのサイズは既に非常に大きいため（例えば、2.2.20バージョンでは

268万行！）、これらの書籍では、Linuxカーネルのソースコードを選択的に説明・解説することができず、多くのシステム実装の詳細は無視されています。そのため、Linuxカーネルについて明確かつ完全な説明をすることは困難です。

スコット・マクスウェル著「Linux Kernel Source Code Analysis」は、基本的にはLinuxの上級者向けの書籍です。この本を完全に理解するには、より包括的な基礎知識が必要です。また、紙面の都合上、Linuxカーネルのすべてのコードを解説しているわけではなく、カーネルで使用されている各種ヘッダファイル(*.h)や、カーネルコードのイメージファイルを生成するツール、プログラムの役割、各makeファイルとその実装など、カーネルの実装に関する多くの情報が省略されています。そのため、入門レベルの読者にとっては、本書を読むことは困難です。

ジョン・ライオンズ著の「レオンのUNIXソースコード解析」は、OSカーネルのUNIXソースコードを学ぶには良い本ですが、UNIXバージョンV6を使用しているため、システムコールのコードの一部が長く使われていないPDP-11シリーズのマシンのアセンブラー言語であるため、ハードウェア部分に関するソースコードを読んで理解する場合には、実験を行うことが困難です。

Andrew S. Tanenbaum氏の著書「Operating Systems: Design and Implementation」は、オペレーティングシステムのカーネル実装に関する良い入門書ですが、この本で紹介されているMINIXシステムは、メッセージベースのカーネル実装の仕組みであり、Linux カーネルの実装には違いがあります。そのため、本書を学んだ後に、より新しいLinuxカーネルのソースコードに着手するのは、あまり容易ではありません。

これらの本を学習に使うとき、「目の不自由な人は象のように感じる」という感覚があるでしょう。
それは

Linuxカーネルシステムの特定の実装の全体的な概念を理解するために、特にLinuxシステムの初心者は、カーネルの原理を学ぶためにこれらの書籍を使用する場合、カーネルの全体的な動作構造。頭の中で明確に形成することはできません。これは、私が長年Linuxカーネルの学習に携わってきた中での深い経験です。1991年10月、Linuxの創始者であるリーナス・トーバルズは、Linuxバージョン0.03の開発中に書いた記事の中で、同じ問題に言及しています。LINUX--a free unix-386 kernel "と題されたこの記事の中で、彼は次のように述べている。"Linuxの開発は、オペレーティングシステムの愛好家やコンピュータサイエンスを学ぶ学生たちの使用、学習、娯楽のためのものである。" 現在の一般的なLinuxシステムは大規模化・複雑化しており、初心者がOSを学ぶ出発点としては適さなくなっている。

そこで本書では、最小限のスペースで、あるいは限られたスペースの中で、Linuxカーネルのソースコード全体を完全に分解し、OSの基本機能と実際の実装を完全に理解することを目指しています。Linuxカーネルを完全に深く理解するためには、Linuxオペレーティングシステムの基本的な動作原理の真の理解と導入が必要です。

Other benefits of reading early kernel code

現在では、DJJのx86オペレーティングシステムやUclinuxなど、Linuxの初期カーネルをベースに、組み込みシステム専用に開発されたカーネルバージョンが数多く存在しています。世界の多くの人々も、初期のLinuxカーネルのソースコードを通して学ぶことのメリットを実感しています。現在、中国ではすでに人間のアノテーションを整理して、この記事と同じような本を出版しています。したがって、初期の Linux カーネルバージョンのソースコードを読むことは、Linux システムを学ぶための効果的な方法であり、Linux 組込みシステムの研究と応用にも非常に役立つものである。

初期のカーネルのソースコードについてコメントする中で、筆者は、初期のカーネルのソースコードは、現在使われている新しいカーネルをほとんど凝縮したようなものだと感じました。現在のバージョンの基本的な機能原理がすでにほぼすべて含まれているのです。System Software. An Introduction to System Programming」の著者であるLeland L. Beck氏は、このように紹介しています。An Introduction to System Programming "の著者であるLeland L. Beckは、システムプログラムとオペレーティングシステムの設計を紹介する際に、すべてのシステムプログラムの設計と実装を説明するために、極めて単純化されたSIC (Simple Instruction Computer) システムを導入しました。その原理は、実際のコンピュータシステムの複雑さを回避するだけでなく、問題を徹底的に記述したものである。ここでは、学習対象としてLinuxの初期のカーネルバージョンを選択し、その指導思想はLelandと同じです。これは、Linuxカーネル学習の初心者にとって、最良の選択の一つです。Linuxカーネルの基本的な動作原理を、最短時間で深く理解することができます。

カーネルの動作原理をすでに知っている人にとっては、実際の作業でシステムの動作メカニズムを、空気中の城を感じさせないようにするために、カーネルのソースコードを読む必要があります。もちろん、初期のカーネルを学習対象とすることにはデメリットもあります。選択したLinux初期カーネルバージョンには、仮想ファイルシステムVFSのサポート、ネットワークシステムのサポート、a.out実行ファイルのみのサポート、他の既存カーネルにある複雑なサブシステムの記述などが含まれていません。しかし、本書はLinuxカーネルの仕組みを学ぶための入門書ですから、この点はカーネルのバージョンが古いものを選択するメリットのひとつです。本書を学ぶことで、これらの高度な内容をさらに学ぶための基礎を固めることができます。

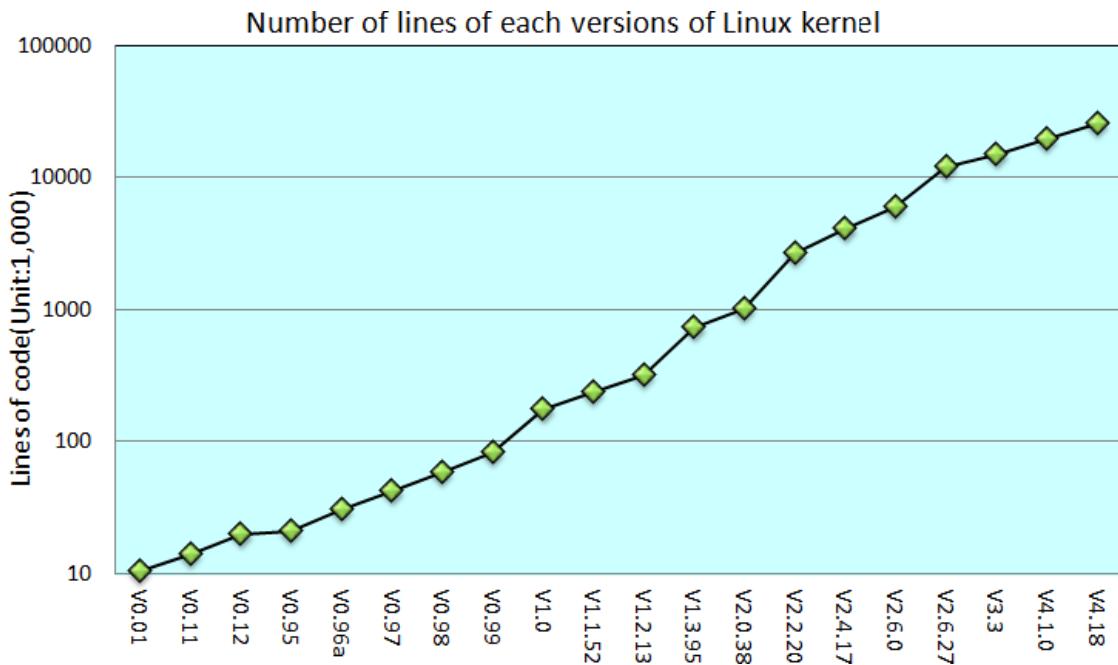
The importance and necessity of reading the complete code

Linuxの創始者がニュースグループの投稿で述べたように、ソフトウェアシステムの真の動作メカニズムを理解するためには、必ずソースコードを読むようにしましょう（RTFSC - Read The F***king Source Code）。システム自体は全体として完成されたものであり、一見重要でないような細部も多く含まれています。しかし、それを無視してしまうと、システム全体の理解が難しくなり、実際のシステムの実装方法や手段を真に理解することができません。

オペレーティングシステムの原理に関するいくつかの古典的な書籍（MJBach氏の「UNIX Operating System Design」など）は、UNIX的なオペレーティングシステムの動作原理を理論的に導くために使用することができますが、実際のオペレーティングシステムの構成は 内部関係の実現についての理解はまだあまり明確ではありません。アンドリュー・S・タネンバウムが言ったように、"多くのオペレーティングシステムの教科書は理論的であり、軽い実践である。" "ほとんどの本やコースは、スケジューリングアルゴリズムに多くの時間とスペースを消費し、I/Oを完全に無視しています。実際には、前者のコードは通常1ページにも満たない。後者はしばしばシステム全体のコードの3分の1を占めなければならない。" カーネルにおける多数の重要な詳細が言及されていません。そのため、本物のOSの真の美しさを理解することができません。完全なカーネルのソースコードを詳細に読んで初めて、システムに対する開放感が生まれ、システム全体の運用プロセスを深く理解することができます。後で学習するカーネルソースコードは、最新のものや新しいものを選ぶと、大きな問題は起こらず、基本的には新しいコードの内容をスムーズに理解することができます。

How to select the kernel code version to read

では、多すぎる内容に惑わされることなく上記の要件を満たし、学習に適したLinuxカーネルのバージョンを選び、学習の効率を上げるにはどうすればよいのでしょうか。筆者は、多数のカーネルバージョンを比較・選択した結果、最終的に、現在のLinuxカーネルの基本機能に近く、かつ非常に短い0.12カーネルを、入門に最適なバージョンとして選びました。次の図は、いくつかの主要なLinuxカーネルバージョンラインの統計を示しています。



現在のLinuxカーネルのソースコード量は数百万行に上り、2.6.0バージョンのカーネルコードラインは約592万行、4.18.Xバージョンのカーネルコードは非常に大きく、2,500万行を超えています！ですから、これらのカーネルに完全に注釈をつけて詳しく説明することはほとんど不可能です。その0.12バージョンのカーネルは、コードの行数が2万行を超えていないので、本にしても説明やコメントがわかりやすくなっています。小さくても完成度は高い。研究対象のシステムを帰納的に理解し、実験を用いて原理の理解を深めるために、著者はこのカーネルをベースにしたLinux 0.12システムも特別に作り直しました。GNU gccのコンパイル環境が入っているので、このシステムを使えば、簡単な開発作業もできます。

さらに、このバージョンを使用することで、さまざまなサブシステム（仮想ファイルシステムVFS、ext2、ext3ファイルシステム、ネットワークサブシステム、新しい複雑なメモリ管理メカニズムなど）の研究がますます複雑になっている既存の新しいカーネルバージョンの使用を避けることができます。

The basic knowledge required by the book

この本を読むには、C言語の基礎知識とインテルCPUのアセンブリ言語の知識が必要です。C言語については、やはり Brian W. Kernighan 氏と Dennis M. Ritchie 氏の著書「The C Programming Language」が一番の参考になります。アセンブリ言語のデータは、インテルCPUを解説したアセンブリ言語の教科書を参考にしてください。また、組み込み用のアセンブリ言語の情報も必要です。GNU gcc コンパイラのマニュアルには、エンベデッド・アセンブリーに関する権威ある情報が掲載されています。また、インターネット上には、エンベデッド・アセンブリーに関する貴重なエッセイ

がありますので、そちらも参考にしてください。また、この本には、インライン・アセンブリの基本的な構文の説明があります(セクション5.5)。

また、読者の皆さんには、次のような基礎知識や関連する参考書をお持ちいただきたいと思います。ひとつは、80x86プロセッサのアーキテクチャやプログラミングに関する知識や情報です。例えば、80x86プログラミングマニュアル (INTEL 80386 Programmer's Reference Manual) は、インターネットからダウンロードできます。2つ目は、80x86のハードウェアアーキテクチャやインターフェースプログラミングに関する知識や情報です。この点に関しては、多くの情報があります。3つ目は、Linuxシステムを使用する簡単なスキルも持っている必要があります。

を開始しました。

Linuxカーネルの実装は、「UNIX operating system design」という本の基本原則に沿って開発されたのが最初なので、ソースコード中の変数名や関数名の多くは、この本に由来しています。そのため、この本をきちんと読んでおけば、カーネルのソースコードを理解しやすくなります。

リーナスが初めてLinux OSを開発したとき、彼はMINIX OSを参考にした。たとえば、オリジナルのLinuxカーネルバージョンは、MINIX 1.0のファイルシステムを完全にコピーしています。そのため、本書を読む際には、A.S.タネンbaum氏の著書『Operating System: 設計と実装』も大いに参考になります。

Is reading an earlier version out of date?

表面的には、Linuxの初期のカーネルバージョンの内容を、あたかもLinux OSがリリースされたばかりのように記しています。タネンbaumは、それが時代遅れ（Linux is obsolete）だと考えているのです。しかし、本書の内容を検討してみると、初期カーネルのソースコードの量が少なく、無駄がないため、本書を使ってLinuxカーネルを学ぶと、学習効率が非常に高く、少ない労力で多くのことができ、すぐに始められることがわかります。また、新しいカーネル部分のソースコードをさらに選択し続けるための強固な基礎を築くことができます。本書を読み終えると、システムがどのように動作するかについて、非常に完全で実用的なコンセプトを持つことになります。この完成された概念により、大量のコードを持つ新しいカーネルのソースコードを完全に読まなくても、新しいカーネルのソースコードの任意の部分をさらに選択して学習することが容易になります。

Ext file system and MINIX file system

現在、Linuxシステムで使用されているExt3ファイルシステムは、カーネル1.x以降に開発されたもので、その機能は詳細で、性能も非常に完成度が高く安定しています。現在のLinux OSでは、デフォルトの標準ファイルシステムとなっています。しかし、Linuxオペレーティングシステムの完全な動作原理を入門的に学ぶ一環として、原理的には、より合理的であればあるほどよい。オペレーティングシステムの完全な理解を達成するために、様々なサブシステムの複雑で過剰な詳細に圧倒されることなく、学習用のカーネルバージョンを選択する原則は、システムコードが実際の動作原理を説明できる限り、できるだけシンプルである。Linuxカーネルバージョン0.12には、当時最もシンプルなMINIX 1.0ファイルシステムしか含まれていませんでしたが、これはオペレーティングシステムにおけるファイルシステムの実際の構成と動作原理を理解するのに十分なものです。これが、学習用に初期のLinuxカーネルバージョンを選択する主な理由の1つです。

この本を一通り読んだ後、こんなため息を送ることになると思います。"Linuxカーネルシステムについては、ようやくスタートラインに立てた！"と。この時点では、最新のLinuxカーネルの各部分の動作原理やプロセスをさらに研究する自信があるはずです。

ザオ・ジオン博士

同濟大学 2019.1

1 Overview

本章ではまず、Linuxオペレーティングシステムの誕生、開発、成長の過程を振り返ります。このことは、本書が学習対象として初期バージョンのLinuxシステムを選んだ理由を理解するのに役立ちます。続いて、学習対象として初期バージョンのLinuxカーネルを選んだ場合のメリットとデメリット、さらに学習を始めるための方法を詳しく説明しています。最後に、各章の内容を簡単に紹介しました。

1.1 The birth and development of Linux

Linuxは、UNIXオペレーティングシステムのクローンシステムである。1991年10月5日に誕生しました（この日が最初の公式発表の日です）。以来、インターネットの普及に伴い、世界中のコンピュータ愛好家が協力して、現在では世界で最も広く使われているUNIX系OSとなり、現在もユーザー数が急増している。

Linuxオペレーティングシステムの誕生、発展、成長は、UNIXオペレーティングシステム、MINIXオペレーティングシステム、GNUプロジェクト、POSIX規格、インターネットネットワークという5つの柱に依存しています。この5つの基本的な手がかりをもとに、Linuxの開発史、醸成過程、初期開発を追っていく。まず、4つの基本要素を紹介し、Linuxの創始者であるリーナス・トーバルズ氏が、自身のコンピュータへの興味からコンピュータの知識を学び、自身のオペレーティングシステムの醸造を開始し、Linuxカーネルバージョン0.01の初期リリースまでリリースし、それがいかに困難なものであるかを追っていきます。そして、世界中のハッカーたちの協力を得て、一歩一歩前進し、ついにはより成熟したバージョン1.0の開発が導入されたのです。また、Linuxの初期開発の歴史についても詳しく書かれている。

1.1.1 もちろん、現在のLinuxカーネルのバージョンは4.18.xまで開発されているが、多くのLinuxシステムで使用されているカーネルは、安定した4.4.x～4.16.xのカーネルである（2桁目が奇数の場合は、開発中を意味し、システムの安定性を保証できない）。Linux開発の一般的な歴史については、多くの記事や書籍が紹介されていますので、ここでは繰り返しません。

1.1.2 The birth of the UNIX operating system

Linuxオペレーティングシステムは、UNIXオペレーティングシステムのクローン版である。UNIXオペレーティングシステムは、1969年夏にベル研究所のKen.ThompsonとDennis RitchieがDEC PDP-7ミニコンピュータ上で開発したタイムシェアリングオペレーティングシステムです。ThompsonとDennis Ritchieが1969年夏にDEC PDP-7ミニコンピュータ上で開発したものです。

1.1.3 ケン・トンプソンは、お気に入りのスター・トラベル・ゲームを遊休中のPDP-7コンピュータで動

かせるようにするために、1969年の夏、妻がカリフォルニアに帰省している間に、1ヵ月でUNIXのオペレーションを開発した。システムの原型となるもの。当時はBCPL言語（Basic Combination Programming Language）が使われていた。1972年にデニス・リッチャーによって移植性の高いC言語に書き換えられた後、UNIXシステムは大学や専門学校で普及していった。

1.1.4 MINIX operating system

MINIXシステムは、Andrew S. Tanenbaum (AST) によって開発されました。ASTは、オランダ・アムステルダムにあるヴリエ大学の数学・コンピュータサイエンスのシステムです。彼はACMとIEEEのシニアメンバーである（この2つの協会のシニアメンバーになっているのは世界でも数人しかいない）。合計100以上の論文と5冊のコンピュータ書籍を出版した。

ASTはニューヨークで生まれたが、それはオランダ人の駐在員であった（彼の祖父は1914年にアメリカに渡った）。ニューヨークの高校、M.I.T.の大学、そしてカリフォルニア大学バークレー校の博士課程で学んだ。博士号取得後の研究のため、故郷であるオランダに来た。それ以来、故郷との付き合いが続いている。その後、ヴリエ大学で教鞭をとり、大学院にも入学しました。オランダの首都アムステルダムは1年中雨の多い街ですが、ASTにとってはこれがベストで、この環境では家でパソコンをいじっていることが多いのです。

MINIXは1987年に誕生し、主に学生がオペレーティングシステムの原理を学ぶために使用されている。1991年にはバージョンが1.5になりました。現在、主に2つのバージョンが使われている。バージョン1.5とバージョン2.0です。当時、大学ではOSは無料でしたが、それ以外の用途では無料ではありませんでした。もちろん、現在のMINIXシステムは無料で、多くのFTPサイトからダウンロードすることができる。

Linuxシステムについては、後に開発者のリーナス氏に賛辞を述べている。しかし、彼は、Linuxの開発は、MINIXを小さくするために、1学期で学習を終えてしまうため、世界中の多くの人からのMINIXの拡張要求を受け入れられなかつたことが大きな原因だと考えている。このような前提で、リーナスはLinuxシステムを書くことになったわけです。もちろん、リーナスもこのタイミングを逃さなかつた。

- 1.1.5 オペレーティング・システムとしてのMINIXは優れたものではなく、C言語とアセンブリ言語で書かれたシステム・ソースコードも提供されている。プログラマーやハッカーを目指す人たちが、OSのソースコードを読むことができたのはこれが初めてのことです。当時、このソースコードはソフトウェアベンダーが大切に守ってきた秘密でした。

1.1.6 GNU Project

GNUプロジェクトとフリーソフトウェア財団は、1984年にリチャード・M・ストールマンによって設立され、UNIXに似た完全なオペレーティングシステムとフリーソフトウェア：GNUシステムを開発した（GNUは「GNU's Not」。Unixの再帰的な略語で、「guh-NEW」と発音します）。Linuxを中心としたさまざまなGNUオペレーティングシステムが広く使われている。これらのシステムはしばしば "Linux" と呼ばれますが、ストールマンは厳密にはGNU/Linuxシステムと呼ばるべきだと考えています。

- 1.1.7 1990年代初頭までに、GNUプロジェクトは、有名なemacs編集システム、bashシェルプログラム、gccシリーズコンパイラ、gdbデバッガなど、多くの高品質なフリーソフトウェアを開発しました。これらのソフトウェアは、Linuxオペレーティングシステムの開発に適した環境を作り出している。これがLinux誕生の基盤のひとつとなり、現在では多くの人がLinux OSを「GNU/Linux」OSと呼んでいる。

1.1.8 POSIX standard

POSIX (Portable Operating System Interface for Computing Systems) は、IEEEとISO/IECが開発した規格群である。この規格は、既存のUNIXの慣習や経験に基づいており、オペレーティングシステムのコールサービスインターフェースを記述しています。コンパイルを確実に行うために使用されるアプリケーションは、ソースコードレベルで複数のオペレーティングシステムに移植して実行することができます。これは、1980年代前半に行われたUNIXユーザーグループ (usr/group) の初期の作業に基づいています。UNIXユーザーグループはもともと、AT&TのSystem Vオペレーティングシステムと、Berkeley CSRGのBSDオペレーティングシステムのコールインターフェースの違いを再統合しようとしていた。1984年には

/usr/group規格をカスタマイズしました。

1985年、IEEE Operating System Technical Committee Standards Subcommittee (TCOS-SS) は、ANSIの庇護のもと、プログラムのソースコードの移植性オペレーティングシステムのサービスインターフェースの正式な規格を制定するようIEEE標準化委員会に指示を出し始めた。1986年4月には、IEEEが試験的に規格を策定した。1988年9月に最初の正式規格が承認され (IEEE 1003.1-1988) 、さらにPOSIX.1規格の

というのが後によく出てきます。

1989年までにPOSIXの作業はISO/IECコミュニティに移管され、15のワーキンググループがISO規格として開発を続けた。1990年になると、POSIX.1は、すでに採用されていたC言語規格と合わせて、IEEE 1003.1-1990（ANSIでもある）およびISO/IEC 9945-1:1990規格として正式に承認された。

POSIX.1は、システムサービスのアプリケーション・プログラミング・インターフェース（API）を規定しているだけで、システムサービスの基本的な規格をまとめているに過ぎない。そのため、ワーキンググループでは、システムの他の機能に関する規格の策定を期待している。そこで、IEEE POSIXの作業が始まった。当初は10の承認計画が進行しており、四半期ごとの1週間の会議には300人近くが参加した。始まった作業は、コマンドとツールの規格（POSIX.2）、テスト方法の規格（POSIX.3）、リアルタイムAPI（POSIX.4）であった。1990年前半には、すでに25の計画が進行し、16のワーキンググループが参加した。同時に、X/Open、AT&T、OSFなど、同様の規格を開発している組織もある。

1990年代初頭、POSIX規格の策定は、1991年から1993年にかけて、最終段階の投票が行われていました。Linuxがまだ始まったばかりのこの時点で、このUNIX規格はLinuxにとって非常に重要な情報を提供し、Linuxが規格の指導のもとで開発され、ほとんどのUNIXオペレーティングシステムと互換性を持つことを可能にした。オリジナルのLinuxカーネルのソースコード（バージョン0.1、0.11、0.12）では、LinuxシステムはPOSIX規格との互換性を備えていました。Linuxバージョン0.01カーネルの/include/unistd.hファイルには、POSIX規格の要件に対応したいくつかのシンボリック定数が定義されており、リーナス氏はコメントにこう記している。“OK、これはジョークかもしれないが、私はそれに取り組んでいる。それはそうだ。”

1.1.9 1991年7月3日、リーナス氏はcomp.os.minixに投稿された記事の中で、POSIXのデータを収集していくことに言及した。それによると、彼はOSの開発に取り組んでおり、開発当初はPOSIXとの互換性の問題を考えていたことが明らかになった。

1.1.10 The birth of the Linux operating system

1981年、IBMは世界的に有名なマイクロコンピューター「IBM PC」を発表した。1981年から1991年までの間、マイクロコンピューターのOSは常にMS-DOSが主役だった。この頃、コンピューターのハードウェアの価格は年々下がってきているものの、ソフトウェアの価格は高止まりしていた。その中で、アップル社の「MACs」というOSは、最高の性能を持っていると言えるが、その価格は誰も簡単には近づけないほどである。

また、当時のコンピューター技術の陣営には、UNIXの世界がありました。しかし、UNIXのOSは高価なだけの問題ではない。高い利益率を求めるために、UNIXの販売店は価格を極端に高くしてしまい、PCユーザーは近寄れないものである。また、かつてベル研究所から許可を得て、大学での教

育に使用していたUNIXのソースコードも、漏洩しないように慎重に守られている。大多数のPCユーザーにとって、ソフトウェア業界の大手ベンダーがこの問題に有効な解決策を示したことはない。

この時、MINIXというOSが登場し、その設計と実装の原理を説明した本が同時に発行されたのである。ASTが書いたこの本は、非常に詳細でよくまとまっていたため、世界中のほとんどのコンピューター愛好家が、OSの仕組みを理解するためにこの本を読むようになったのである。その中には、Linuxシステムの創始者であるリーナス・ベネディクト・トーバルズも含まれている。当時（1991年）、彼はヘルシンキ大学のコンピュータサイエンス学科の2年生で、独学でコンピュータハッカーをしていた。21歳のフィンランド人青年は、自分のコンピュータをドラム缶に入れて、その性能や限界を試すのが好きだ。しかし、当時の彼に足りなかったのは、プロレベルのOSでした。

この年、GNUプログラムは数多くのソフトウェアツールを開発しました。最も期待されているのは

GNU Cコンパイラは登場したが、自由なGNUオペレーティングシステムはまだ開発されていない。教育現場で使われているMINIX OSでさえ、著作権が発生し始めており、ソースコードを入手するためには購入する必要があるのです。GNUオペレーティング・システムのHURDは開発中ではあるが、数年以内に完成するとは思えなかった。

コンピュータの知識を身につけるために（興味本位かもしれないが）、ライナスはクリスマスの幸運なお金やローンを使って386互換機を購入し、米国からMINIXシステムのソフトウェアを郵送した。MINIXのソフトを待つ間、ライナスはインテル80386のハードウェアの知識を真剣に学んだ。モードムによるダイアルアップで学校のメインフレームに接続できるようにするために、アセンブリ言語を使い、80386CPUのマルチタスク機能を利用して、ターミナル・エミュレーション・プログラムを作った。その後、古いパソコンに入っていた自分のソフトを新しいパソコンにコピーするために、フロッピーディスクドライブやキーボードなどのハードウェアデバイスのドライバーもコンパイルした。

プログラミングの練習を通して、また学習過程でMINIXシステムの多くの限界を認識したこと（MINIXは良いものだが、強力で実用的なオペレーティングシステムではなく、教育目的のシンプルなオペレーティングシステムに過ぎない）、リーナスはすでに似たようなものを持っていた。オペレーティングシステムのデバイスドライバのコードで、彼は新しいオペレーティングシステムのアイデアを持ち始めた。この時点で、GNUプロジェクトは多くのツールやソフトウェアを開発しており、その中でも最も期待されているGNU Cコンパイラが登場した。GNUの自由なオペレーティングシステムHURDは開発中ですが。しかし、リーナスは急ぐことなく待っていた。

1991年4月からは、ターミナル・エミュレーション・プログラムやハードウェア・ドライバーを改造して、独自のOSを開発し始めた。当初の目的は、インテル386アーキテクチャの保護モード動作のプログラミング技術を習得するだけという単純なものだった。しかし、Linuxの開発は、当初の目的を完全に変えてしまった。comp.os.minixニュースグループでのリーナス氏のニュースリリースによれば、彼がMINIXシステムの段階での学習から、独自のLinuxシステムの開発へと徐々に進化していることがわかる。

リーナスが初めてcomp.os.minixにメッセージを届けたのは1991年3月29日のこと。投稿された記事のタイトルは「gcc on minix-386 doesn't optimize」。これは、MINIX-386システムでgccコンパイラが最適化されて動作するというものです（MINIX-386は、Bruce EvansがIntel 386 features 32 Bit MINIX systemを使って改良したもの）。このことから、Linusは1991年初頭にすでにMINIXシステムを深く研究し始めており、その間にMINIXオペレーティングシステムの改良が行われていたことがわかります。MINIXシステムについてさらに学んだ後、このアイデアは、次第にインテル80386アーキテクチャをベースにした新しいオペレーティングシステムを再設計するというアイデア

へと発展していった。

彼がMINIXについて誰かの質問に答えたとき、最初に言った文章は「Read the F**ing Source Code :-)」でした。）彼は、答えはソースプログラムの中にあると考えているのだ。このことからも、学習システムのソフトウェアでは、システムの基本的な動作原理を理解するだけでなく、実際のシステムを組み合わせて、実際のシステムの実装方法を学ぶ必要があることがわかります。結局のところ、理論は理論で、多くの枝が省略されています。このような枝葉の問題は、理論的な内容は少ないので、スズメに羽が生えているように、システムには必要なものなのです。

1991年4月以降、リーナスはほとんどすべての時間をMINIX-386システムの研究（Hacking the kernel）と、GNUソフトウェアのMINIXへの移植（GNU gcc, bash, gdbなど）に費やした。そして、4月13日にcomp.os.minixで、bashのMINIXへの移植に成功し、シェルソフトを残すことができなくなったと発表しました。

Linux関連のニュースが初めて公開されたのは、1991年7月3日のcomp.os.minixでした。（もちろん、当時はLinuxという名前はありませんでした。リーナスは、その名前を「FREAK, FREAX」ではないかと考えた。英語の意味は、グロテスク、モンスター、気まぐれ、など）。これは、彼がLinuxシステムを開発していて、POSIXとの互換性の問題をすでに考えていたことを示している。

リーナスの別の発表 (comp.os.minix、1991年8月25日) では、すべてのMINIXユーザーに「MINIXシステムに最も見たいものは何ですか? ("What would you like to see?" In minix?")と題して、(フリーの)386(486)オペレーティングシステムが開発されていることを初めて明かし、自分はそれにしか興味がないことを語った。コードは大きくなく、GNUのようなプロフェッショナルなものにはならないでしょう。MINIXシステムが好きな機能と嫌いな機能についてフィードバックしていただき、実用上の理由やその他の理由から、新しく開発されたシステムはMINIXに似ている (MINIXのファイルシステムを使っている) ことを説明していただければと思います。また、bash (バージョン1.08) とgcc (バージョン1.40) を新システムに移植することに成功しており、数ヶ月後には実用化される予定です。

最後にリーナスは、自分が開発したOSはMINIXのソースコードを1行も使っていないと述べている。386のタスク切り替え機能のため、OSは移植性がなく (ノーポータビリティ) 、ATのハードディスクしか使われていない。リーナスは、Linuxの移植性の問題を考えていなかった。しかし、現時点では、Linuxはほとんどの種類のハードウェアアーキテクチャで動作することができる。

1.1.11 1991年10月5日、リーナスはニュースグループcomp.os.minixにメッセージを掲載し、Linuxカーネルシステムの誕生を公式に発表した (Free minix-like kernel sources for 386-AT)。このニュースは、Linuxの誕生宣言ともいえるもので、広く流布している。そのため、10月5日はLinuxコミュニティにとって特別な日であり、その後の多くのLinuxバージョンがこの日を選んでいた。だから、RedHatがこの日を選んで新システムをリリースしたのは偶然ではない。

1.1.12 Linux operating system version changes

Linux OSが誕生してから1.0がリリースされるまでには、表1-1のように多くのメジャーリリースが行われてきました。リーナスさんは、2003年9月にバージョン管理ツールBitKeeperの使い方を学び始めたときに、1.0の過去のバージョンをすべて見ました。実際には、Linuxシステムにはこの0.00というバージョンは存在しないが、リーナスが自分の80386互換機で行った実験で、クロックの割り込み制御で2つのタスクを切り替えることに成功したため、自分のOS開発のアイデアをさらにある程度高めた。.そのため、バージョンとしても記載しています。Linux版のカーネルバージョンが完成したのは、1991年9月17日のことである。しかし、リーナスには著作権意識が全くないので、このバージョンのinclude/string.hファイルには著作権情報が1部だけ表示されています。このバージョンのカーネルのキーボードドライバは、フィンランド語のコードにのみハードコーディングされているため、フィンランド語のキーボードしかサポートしていません。また、8MBの物理メモリのみサポートしています。Linusのミスにより、その後の0.02, 0.03バージョンのカーネルのソースコードは破壊されて失われた。

| 表1-1 以前の カーネ ルのメ ジャー バージ | Release date | Description |
|---|--------------|-------------|
| | | |

| バージョン番号 Version No. | | |
|------------------------|-----------|---|
| 0.00 | 1991.2-4 | The two processes display 'AAA...' and 'BBB...' on the screen, respectively. (Note: No release) |
| 0.01 | 1991.9.17 | The first official release of the Linux kernel version. Multi-threaded file system, segmentation, and paging memory management. Does not include floppy disk drivers yet. |
| 0.02 | 1991.10.5 | This version and version 0.03 is an internal version that is currently unavailable. Features the same as above. |
| 0.10 | 1991.10 | The Linux kernel version released by Ted Ts'o. Added memory allocation library functions. The boot directory contains a script that converts as86 assembler syntax to gas assembler syntax. |
| 0.11 | 1991.12.8 | Basically functioning kernel. Supports hard disk and floppy drive devices as well as |

| | | |
|---------------------|------------|---|
| | | serial communications. |
| 0.12 | 1992.1.15 | The more stable version mainly increases the software simulation program of the math coprocessor. Added job control, virtual console, file symbolic links, and virtual memory swapping capabilities. |
| 0.95.x (ie 0.13) | 1992.3.8 | Virtual file system support was added in this version, but it still contains only one MINIX file system. Added login functionality. Improves the performance of floppy disk drivers and file systems. Changed hard disk naming and numbering. The original naming method is the same as that of the MINIX system. At this time, it is the same as the current Linux system. Support CDROM. |
| 0.96.x | 1992.5.12 | Began to add UNIX Socket support. Added ext file system alpha tester. SCSI drivers are officially added to the kernel. Floppy disk type is automatically recognized. Improved serial driver, cache, memory management performance, support for dynamic link libraries, and the ability to run X-Windows programs. The keyboard driver written in the original assembly language has been rewritten with C. Compared with the 0.95 kernel code, there are great changes. |
| 0.97.x | 1992.8.1 | Added support for new SCSI drivers; dynamic caching; msdos and ext file system support; bus mouse drivers. The kernel is mapped to the beginning of the linear address 3GB. |
| 0.98.x | 1992.9.28 | Improve support for TCP/IP (0.8.1) networks and correct extfs errors. Rewritten memory management section (mm), each process has 4GB of logical address space (the kernel occupies 1GB). Starting from 0.98.4, each process can open 256 files at the same time (originally 32), and the process's kernel stack uses a single memory page independently. |
| 0.99.x | 1992.12.13 | Re-design the process of the use of memory allocation, each process has 4G linear space. Constantly improving the network code. NFS support. |
| 1.0 | 1994.3.14 | The first official version. |

既存の0.10バージョンのカーネルコードは、当時保存されていたTed Ts'oのバージョンで、Linus自身のものも失われています。このバージョンは、以前のバージョンに比べて大きく改善されています。このバージョンのカーネルシステムでは、GNU gccがカーネルのコンパイルに使われており、ファイルシステムのマウント/アンマウントの操作をサポートするようになりました。このカーネル・バージョンから、リナスは各ファイルの著作権情報を追加しました。"(C) 1991 Linus Torvalds" です。その他の変更点としては、オリジナルのブートプログラム boot/boot.s が boot/bootsect.s と boot/setup.s の2つのプログラムに分割されたこと、1 最大 16MB の物理メモリをサポートしたこと、2 ドライバとメモリ管理プロシージャがそれぞれ別のサブディレクトリを作成したこと、3 フロッピードライバを追加したこと、4 ファイルの先読み操作をサポートしたこと、5 dev/port と dev/null デバイスをサポートしたこと、6 kernel/signal.c のコードを書き換え、sigaction() サポートを追加したこと、などが挙げられます。

カーネルの0.10バージョンと比較して、Linux 0.11バージョンの変更点は比較的小さい。しかし、このバージョンは、最初の安定版であり、他の人々は、カーネルの開発に参加し始めている。このバ-

ジョンの主な追加事項は次のとおりです。1実行プログラムの要件をロードする、2起動時に/etc/rc初期ファイルを実行する、3数学コプロセッサのシミュレーションプログラムのフレームプログラムの構造を構築する、4Ted Ts'oは、スクリプトプログラムを追加する処理コード、5 Galen Huntは、複数のディスプレイカードのサポートを追加する、6John T Kohlは、つぶやきとKILL文字をサポートするためにコンソールを有効にするためにカーネル/console.cプログラムを変更する、7は、多言語キーボードのサポートを提供します。

Linux 0.12は、Linusがより満足できるカーネルバージョンで、より安定したカーネルです。クリスマスの間に

1991年のシーズンには、gccのような「大規模な」ソフトウェアが2MBのメモリしか搭載されていないマシンでも使用できるように、仮想メモリ管理コードをコンパイルした。このバージョンを見て、リーナスはカーネルのバージョン1.0をリリースすることが眼中ないことだと感じ、すぐに次のバージョン（バージョン0.13）をバージョン0.95にアップグレードした。リーナスがこのようなことができるるのは、誰もがまだバージョン1.0には程遠いと感じないようにするという意味合いもある。しかし、0.95バージョンのリリースを急いだために、エラーも多く含まれており、0.95バージョンがリリースされたばかりの頃は、使用中に問題が発生するLinux愛好家が増えていました。その時、リーナス氏は「大惨事に遭遇した」と感じたという。しかし、それ以来、彼はこの教訓を受け入れた。その後、新しいカーネルバージョンがリリースされるたびに、彼はより慎重にテストを行い、公式に発表する前に数人の親友に試してもらうようにしています。カーネルの0.12バージョンの主な変更点は以下の通りです。1 Ted Ts'oによる端末信号処理のサポートの追加、2 起動時に使用する画面ランクの変更が可能、3 ファイルのIOによる競合状態の修正、4 共有ライブラリのサポートの追加、メモリ使用量の節約、5 シンボリックリンクの処理、6 ディレクトリシステムコールの削除。7 ピーター・マクドナルドによる仮想端末のサポートの実装。これにより、Linuxは当時の特定の商用バージョンのUNIXよりもさらに優れたものとなった。関数のサポート。これは、何人かの人がMINIX用に提供したパッチをもとにピーター・マクドナルドが修正したものだが、MINIXはこれらのパッチを採用しなかった。9 再実行可能なシステムコール。10 Linusによる数学コプロセッサのシミュレーションコードのコンパイル。

バージョン0.95は、GNU GPLの著作権を使用した最初のLinuxカーネルバージョンです。このバージョンには、実際には3つのサブバージョンがあります。1992年3月8日に最初の0.95がリリースされたとき、いくつかの問題が発生したため、10日も経たないうちにすぐに別の0.95aがリリースされた（3月17日）。そして、1ヶ月後の4月9日には0.95c+がリリースされました。このバージョンの最大の改良点は、仮想ファイルシステムVFS構造の導入である。当時はMINIXファイルシステムしかサポートしていましたが、複数のファイルシステムをサポートするために、プログラム構造を広範囲に調整しました。MINIXファイルシステム用のコードは、別のMINIXサブディレクトリに置かれています。0.95カーネルのその他の変更点としては、以下のようなものがあります。1 ログインインターフェースの追加、2 Ross Biroによるデバッグコード(ptrace)の追加、3 フロッピーディスクドライブのトラックバックファーリング、4 ノンブロックキングバイオペーパー操作、5 システムの再起動(Ctrl-Alt-Del)、リアルタイムでスワップデバイスを選択するSwapon()システムコール、6 再帰的シンボリックリンクのサポート、7 4つのシリアルポートのサポート、8 ハードディスクパーティションのサポート、9 より多くの種類のキーボードのサポート、10 James Wiegandによる初期パラレルポートドライバのコンパイ

ル、などなど。

また、0.95のリリースからは、カーネルの改良（パッチの提供）の多くが他社に独占され、リナスの主な仕事はカーネルのメンテナンスと、パッチを採用するかどうかの判断になっていきました。現在まで、カーネルの最新バージョンは2018年6月にリリースされたバージョン4.16.16です。その使用しているgz圧縮のソースコードパッケージも約152MB! 各メジャー安定版リリースの最新バージョンを表1-2に示す。表1-2

| 表1-2 新しいカーネルソースコードのサイズ Version number | Release date | Size (after gz compression) |
|--|--------------|-----------------------------|
| 2.0.40 | 2004.2.8 | 7.2 MB |
| 2.2.26 | 2004.2.25 | 19 MB |
| 2.6.25 | 2008.4.17 | 58 MB |
| 3.0.10 | 2011.11.21 | 92MB |
| 4.4.10 | 2016.5.11 | 127MB |
| 4.16.16 | 2018.6.16 | 152MB |

1.1.13 The reason for the Linux name

Linux OSが誕生した当初は、Linuxとは呼ばれていませんでした。リナスは自分のオペレーティングシステムを

FREAK（フリーク）。英語の意味は、グロテスク、モンスター、気まぐれ。新しいOSをftp.funet.fiのサーバーにアップロードしたとき、管理者のアリ・レムケはこの名前をとても嫌がりました。彼は、リーナスのオペレーティングシステムなので、その同音異義語であるLinuxをオペレーティングシステムのディレクトリとして使用することで、Linuxの名前が受け継がれるようになったと考えています。

リーナスの自伝「Just for Fun」の中で、リーナスはこう説明している。

「正直なところ、私はLinuxという名前ではリリースしたくありませんでした。それはあまりにも自己中心的だからです。最終的なリリースのために予約した名前は何でしたか？Freakです。（実際、初期のmakeファイル（ソースのコンパイル方法を記述したファイル）の中には、半年ほど「Freak」という言葉が含まれていました。でも、そんなことはどうでもいいことでした。その時点では、誰にも公開していなかったので、名前は必要ありませんでした）。

「そして、ftpサイトへの投稿を保証してくれたAri LemkeはFreakという名前を嫌っていました。彼は、私が使っていたもう一つのワーキングネームであるLinuxを気に入り、私の投稿を「pub/OS/Linux」と名付けました。私があまり抵抗しなかったことは認めます。しかし、それは彼がやったことです。だから正直に言うと、私はエゴイストではなかったし、半分正直に言うと、エゴイストではなかったんです。でも、いい名前だと思ったし、いつでも誰かのせいにできると思ったから、今そうしているんだ」。

1.1.14 The main contributor to the development of early Linux systems

初期のLinuxのソースコードを見ればわかるように、リーナス自身に加えて、Linuxシステムの最も有名な開発者の一人がセオドア・ツォ（Ted Ts'o）である。彼は1990年にMITコンピュータサイエンス学科を卒業した。大学時代には、学校で行われるさまざまな学生活動に積極的に参加した。趣味は、料理、サイクリング、そしてもちろんLinuxでのハッキング。その後、アマチュア無線の電報キャンペーンが好きになりました。現在は、IBMでシステムプログラミングなどの仕事をしています。また、International Network Design, Operations, Sales and Research Open GroupのIETFメンバーでもあります。

Linuxが世界的に普及したのも、彼の功績によるところが大きい。早くもLinux OSが登場したとき、彼はMaillistにLinuxの開発に大きな熱意を持って提供した。Linuxがリリースされて以来、彼はLinuxに貢献し続けている。また、Linuxカーネルに初めてプログラムを追加した人物でもある（Linuxカーネルバージョン0.10の仮想ディスクドライバ「ramdisk.c」、カーネルメモリ割り当てプログラム「kmalloc.c」など）。現在も、Linux関連の仕事に従事している。北米では、Linuxのftpサイト(tsx-11.mit.edu)を最初に立ち上げ、今でも大多数のLinuxユーザーにサービスを提供している。彼のLinuxへの最大の貢献は、ext2ファイルシステムの提案と実装です。このファイルシステムは、今ではLinuxの世界でデファクト・ファイルシステムの標準となっている。最近、彼はext3ファイルシステムを発表しました。このシステムは、ファイルシステムの安定性とアクセス効率を大幅に改善している。彼への賞賛として、Linux Journal第97号（2002年5月）では、彼を表紙キャラクターに起用し、

インタビューを行っている。現在は、IBM Linux Technology Centerに所属し、LSB（Linux Standard Base）の開発に取り組んでいる。

もう一人、Linuxコミュニティで有名なのがアラン・コックスだ。彼はもともと、ウェールズのスワンジー大学カレッジで働いていました。当初、彼はコンピュータゲームが好きで、特にMUD（Multi-User Dungeon or Dimension）を好んでプレイしていた。90年代前半の games.mud ニュースグループの投稿を見ると、彼が投稿したものがたくさんある。彼はMUD開発の歴史を書いたこともある（rec.games.mud news group, March 9, 1992, A history of MUD）。MUDゲームがインターネットと密接に関係していることから、彼は徐々にコンピュータネットワークに魅了されていきました。ゲームをプレイし、ゲームを実行しているコンピュータの速度やネットワークの伝送速度を向上させるためには、彼は最も満足のいくオペレーティング・プラットフォームを選択する必要がある。そこで、彼はさまざまな種類のOSに接触するようになった。お金がないので、MINIXシステムすら買えなかったのだ。Linux 0.1xや386BSDが発売された時には、386SXを購入するのに時間がかかった。386BSDは数学を必要とするので

コプロセッサに対応しており、CPUにインテル386SXを搭載したコンピューターには数学コプロセッサが搭載されていないことから、彼はLinuxシステムをインストールした。そこで彼は、無料のソースコードを使ってLinuxの学習を始め、Linuxシステム、特にネットワークに関して興味を持ち始めたのである。Linuxのシングルユーザーモードの動作についての議論では、「Linuxは美しく実装されている」と賞賛したほどである。

Linux 0.95のリリース後、彼はLinuxシステムのパッチ（修正プログラム）を書き始め（彼の最初の2つのパッチはLinusに採用されなかったことを覚えておいてください）、Linuxシステム上でTCP/IPネットワークコードの最初のユーザーになりました。1人。その後、徐々にLinuxの開発チームに加わり、Linuxカーネルのソースコードを保守する主要な責任者の一人となった。また、Linuxコミュニティの中では、リナスを中継した後の最も重要な人物とも言える。後にマイクロソフトからも誘われたが、あっさりと断っている。2001年からは、Linuxカーネル2.4.xのコードのメンテナンスを担当している。Linusは、主にカーネルの最新開発バージョン（2.5.xバージョンなどの奇数バージョン）の開発を担当している。

The Linux Kernel Hackers' Guide』の著者であるMichael K. Johnsonは、Linuxオペレーティングシステムに最初にコンタクトした人物のひとりでもあります（バージョン0.97から）。また、有名なLinux Document Project (LDP)の発起人のひとりでもある。かつては Linux Journal に勤務し、現在は RedHat に勤務している。

現在のように発展できるバックボーンは、Linuxシステムだけではありません。Linuxに多大な貢献をしたコンピュータの専門家はたくさんいます。ここではそれらをリストアップしません。主要な貢献者の具体的なリストは、Linuxカーネル内のCREDITSファイルに記載されており、Linuxに多大な貢献をした400人以上のリストがアルファベット順に記載されており、メールアドレスや送付先、ホームページ、主要な貢献内容などが記載されています。証書などがあります。

- 以上の説明を通して、Linuxの上記5つの柱をまとめると、以下のようになります。
 - UNIX Operating System -- UNIX was born in Bell Labs in 1969. Linux is a UNIX clone system. The importance of UNIX goes without saying.
 - The MINIX operating system -- The MINIX operating system is also a UNIX clone system. It was developed in 1987 by the famous computer professor Andrew S. Tanenbaum. Due to the emergence of the MINIX system and the availability of source code (which can only be used free of charge in universities), the whirlwind of learning the UNIX system was spurred by universities around the world. Linux first started development in 1991 with reference to the MINIX system.
 - The GNU Project -- The development of the Linux operating system, and most of the software used on Linux is basically from the GNU program. Linux is only a kernel of the operating system. Without the GNU software environment (such as the bash shell), Linux will be difficult to move.
 - POSIX Standard -- This standard has played an important role in the development of the Linux operating system after the formal development. It is the beacon of Linux's progress.

- Internet - If you don't have an Internet network and don't have the unselfish dedication of countless computer hackers all over the world, then Linux can only grow to a level of 0.13 (0.95).

1.2 Content review

本書では、主に初期のLinuxカーネルバージョン0.12についての説明と解説を行う。Linux-0.12バージョンは1992年1月15日にリリースされました。出版の際には以下のファイルを含めてください。

bootimage-0.12.Z - 圧縮されたブートイメージファイルで、U.S.キーボードコードが含まれています。

rootimage-0.12.Z - 1200kBの圧縮されたルートファイルシステムのイメージファイルです。

linux-0.12.tar.Z - カーネルのソースコードファイル。サイズは130KBで、展開後は463KBしかありません。

| | |
|--------------|--|
| as86.tar.Z | - Bruce Evans' binary execution file. 16-bit assembler and loader; |
| INSTALL-0.11 | - Updated installation information file. |

bootimage-0.12.Z と rootimage-0.12.Z は、圧縮されたフロッピーアイメージファイルです。

Bootimage はブートイメージファイルで、主にディスクのブートセクタコード、オペレーティングシステムのローダ、カーネルの実行コードが含まれています。PC が起動すると、ROM BIOS のプログラムがデフォルトのブートドライブからブートセクタコードとデータをメモリに読み込み、ブートセクタコードがオペレーティングシステムローダーとカーネル実行コードをメモリに読み込んで制御します。初期化のためにカーネルをさらに準備するのはオペレーティングシステムローダーであり、最終的にはローダーがカーネルコードに制御を与えます。カーネルコードが正しく機能するためには、ファイルシステムのサポートが必要です。Rootimage は、カーネルに最も基本的なサポートを提供するために使用されるルートファイルシステムで、オペレーティングシステムに少なくともいくつかの設定ファイルとコマンド実行手順を含む。Linux システムで使用される UNIX ベースのファイルシステムには、主にいくつかの指定されたディレクトリ、設定ファイル、デバイスドライバー、開発プログラム、その他すべてのユーザーデータやテキストファイルが含まれます。この 2 つのディスクの組み合わせは、起動可能な DOS オペレーティングシステムのディスクに相当する。

as86.tar.Z は 16 ビットのアセンブラー・リンクパッケージです。linux-0.12.tar.Z は Linux 0.12 カーネルのソースコードを圧縮したものです。INSTALL-0.11 は Linux 0.11 システム用の簡単なインストールドキュメントです。また、0.12 カーネルを使用している Linux システムにも適用されます。

■ 現時点では、オリジナルの rootimage-0.12.Z ファイルに加えて、他の 4 つのファイルを見つけることができます。しかし、著者はインターネット上のリソースを利用して、Linux 0.12 用の完全に使用可能な rootimage-0.12 ルートファイルシステムを再構築しました。0.12 環境で使用できる gcc 1.40 コンパイラーを再コンパイルし、利用可能な実験的開発環境を設定しています。現在、これらのファイルは oldlinux.org のウェブサイトからダウンロードできます。具体的なダウンロードディレクトリの場所は

- <http://oldlinux.org/Linux.old/images/> This directory contains the kernel image file bootimage and the root file system image file rootimage that have been created.
- <http://oldlinux.org/Linux.old/kernels/> This directory contains the kernel source code programs, including the Linux 0.12 kernel source code program described in this book.
- <http://oldlinux.org/Linux.old/bochs/> This directory contains Linux systems that have been set up to run under the computer simulation system bochs.
- <http://oldlinux.org/Linux.old/Linux-0.12/> This directory contains some of the other tools that can be used in the Linux 0.12 system and some of the original installation instructions.

本書は、主に linux-0.12 kernel に含まれるすべてのソースコードプログラムを詳細に解析し、各

ソースプログラムファイルに対して、Makefileファイルのコメントを含めた詳細なコメントを付けています。解析作業は、主にコンピュータの起動プロセスに合わせて行われます。したがって、初期化カーネルの終了までの解析の一貫性は、シェルプログラムの呼び出しを開始します。それ以外のプログラムは、それぞれの解析のためのもので、まとめはありませんので、それぞれの必要に応じて読んでください。ただし、解析中にいくつかの応用例を紹介しています。

すべてのプログラムを解析する過程で、筆者がその記述を理解するのが難しいと考えた場合には、関連する知識を詳しく説明します。例えば、割り込みコントローラへの入出力操作が発生した場合、インテルの割り込みコントローラ（8259A）チップの詳細な説明を行い、使用するコマンドやメソッドをリストアップします。これにより、コードの理解を深めるだけでなく、使用するハードウェアの使い方をより理解することができます。筆者は、ハードウェアなどの知識を全体的に紹介するために別の章を設けるよりも、このような解釈方法の方がはるかに効率的だと考えています。

Linux 0.12のカーネルを "解剖"することで、Linuxの機能を理解する効率を高めることができます。

オペレーティングメカニズム。全体のカーネルのソースコードのLinux - 0.12バージョンは、コンテンツを含むわずか約463Kバイトは、基本的にLinuxの本質です。最新のカーネルバージョン2.6.XXは非常に大きい、200メガバイトです。一生かけて読めるようになったとしても、全部は読めないかもしれません。そこで、「せっかくJaneから始めるのだから、バージョン0.01のLinuxカーネルのソースコードを小さくして分析してみたらどうだろう？約240Kバイトしかないんだから。」とかね。主な理由は、0.01バージョンのカーネルコードには欠点が多すぎるからです。フロッピーディスク用のドライバを含むだけでなく、数学コプロセッサの使用やログイン手順の指示にもうまく対応できません。また、0.12の起動ブートプログラムの構造は、基本的に今のバージョンと同じではありません。もうひとつの理由は、すでにコンパイルされているカーネルイメージファイル(bootimage-0.12)の初期バージョン1.22が、ブートデモに使用できることです。簡単なルートファイルシステムイメージ(rootimage-0.12)を追加すれば、普通に起動できるようになります。

また、Linux 0.12での学習には不備があります。たとえば、カーネルのバージョンには、特別なプロセスの待ち行列やTCP/IPネットワークなどに関する非常に重要なコードが含まれていません。また、メモリの割り当てや使用方法も現在のカーネルとは異なります。幸いなことに、Linuxのネットワークコードは基本的に自己完結型であり、カーネルの仕組みとの関係はそれほど大きくないので、Linuxの動作の基本原理を理解した上でコードを解析することができます。

本書は、Linuxカーネルのすべてのコードを解説しています。構造の整合性を保つために、コードの記述はカーネル内のソースコードの構造に基づいています。基本的には、各ソースコードの内容を1章としています。紹介するソースファイルの順番は、前回のファイルリストインデックスで確認できます。Linuxカーネルのソースコード全体のディレクトリ構造をリスト1-1に示します。すべてのディレクトリ構造は、Linuxをカレントディレクトリとした場合のものです。

| リスト 1-1 Linux/ ディレクトリ | | | | |
|-----------------------|------------|--------------------------|-------|--|
| Name | Size | Last modified date (GMT) | Desc. | |
| boot/ | | 1992-01-16 14:37:00 | | |
| fs/ | | 1992-01-16 14:37:00 | | |
| include/ | | 1992-01-16 14:37:00 | | |
| init/ | | 1992-01-16 14:37:00 | | |
| kernel/ | | 1992-01-16 14:37:00 | | |
| lib/ | | 1992-01-16 14:37:00 | | |
| mm/ | | 1992-01-16 14:37:00 | | |
| tools/ | | 1992-01-16 14:37:00 | | |
| Makefile | 3091 bytes | 1992-01-13 03:48:56 | | |

本書の内容は、5つのパートに分けられます。第1章から第4章までは基礎編。オペレーティングシステムは、実行されるハードウェア環境と密接に関係しています。オペレーティングシステムの全体的

な動作を徹底的に理解したいのであれば、そのハードウェアの動作環境、特にプロセッサのマルチタスク動作の仕組みを理解する必要があります。このパートでは、マイクロコンピュータのハードウェア構成、Linuxカーネルプログラムをコンパイルするためのプログラミング言語、インテル80X86保護モード下でのプログラミング原理などをより詳しく紹介しています。第2パートには第5章から第7章までがあり、カーネルのブート起動と32ビット動作について説明しています。この方法の準備段階では、初心者がカーネルを学ぶために十分に読んでおく必要があります。第3部の第8章から第13章までは、カーネルコードの主要部分です。第8章の内容は、このセクションの後続の章を読むための主な手掛かりとすることができます。第14章から第16章までは

第4部の内容は、第3部のソースコードを読む際の参考になります。最後のパートには第17章のみが含まれており、PCシミュレーションソフトウェアシステム「Bochs」を使って、Linux 0.12カーネル上でさまざまな実験活動を行う方法が書かれています。

第2章では、従来のマイクロコンピュータシステムのハードウェアブロック図に基づいて説明しています。主にLinuxカーネルで動作するIBM PC/AT386マイクロコンピュータの構成要素を紹介しています。各主要部の機能と関係を説明します。同時に、最新のマイクロコンピュータのブロック図とも比較しています。これにより、コンピュータの構成原理を学んでいない読者にも、十分な関連情報提供することができる。

第3章では、Linux 0.12カーネルで使用されているプログラミング言語、オブジェクト・ファイル・フォーマット、およびコンパイル環境を紹介します。主な目的は、Linux 0.12 カーネルのソースコードを読むために必要なアセンブリ言語と GNU C 言語拡張の知識を提供することです。本章では、まずas86とGNU asアセンブラーの構文と使い方をより詳しく紹介し、次にGNU C言語のインライン・アセンブリ、ステートメント式、レジスタ変数、インライン関数などの一般的なC言語拡張を説明しました。また、C言語とアセンブリ関数の相互呼び出しの仕組みについても詳細に説明した。最後に、Makefileの使い方を簡単に説明する。

第4章では、80X86 CPUのアーキテクチャと、プロテクトモード・プログラミングの基礎知識について説明しています。この章では、80X86 CPUをベースにしたLinuxカーネルのソースコードを読むための準備として、しっかりととした基礎知識を身につけることができます。その内容は以下の通りです。80X86の基礎知識、プロテクトモードのメモリ管理、割込みと例外処理、タスク管理、そしてシンプルなマルチタスクカーネルの例です。

第5章では、Linux オペレーティングシステムのアーキテクチャ、カーネルのソースコードファイルの構成、各ファイルの一般的な機能について説明しています。また、Linuxでの物理的なメモリの割り当て、カーネルのいくつかのスタックとその使用方法、仮想リニアアドレスの使用についても紹介しています。最後に、カーネルパッケージのLinux/ディレクトリにある最初のファイル、つまりカーネルコードの全体的なMakefileの内容について解説を始めます。このファイルは、すべてのカーネルソースプログラムのコンパイル管理用設定ファイルで、ビルト管理ツールソフトウェアmakeが使用します。

第6章では、boot/ディレクトリにある、ディスクブートプログラムのbootdisk.ss、BIOSのパラメータを取り込むsetup.sアセンブラー、32ビットランスタートコードプログラムのhead.sの3つのアセンブラプログラムについて詳しく説明します。この3つのアセンブラプログラムにより、ブロックデバイスからメモリへのカーネルのブートロードとシステム構成パラメータの検出が完了し、32ビッ

トプロテクトモードに入る前のすべての作業が完了します。カーネルシステムがさらに初期化作業を行うための準備を行う。

第7章では主に、init/ディレクトリにあるカーネルシステムの初期化プログラムmain.cを紹介します。カーネルがすべての初期化作業を完了し、通常の動作に入るための重要なポイントとなります。システムの初期化がすべて完了すると、シェル用のプロセスが作成されます。プログラムの紹介では、それが呼び出す他のプログラムを見る必要があるので、後続の章の読解は、ここで呼ばれる順序で実行することができます。カーネルではメモリ管理プログラムの機能が広く使われているので、この章を最初に読むべきです。main.cまでのプログラムが本当に理解できるようになると、Linuxカーネルのことがある程度理解できるようになります。半分くらいはすでに始まっていると言えますが、さらに深く読むためには、ファイルシステム、システムコール、各ドライバなども必要です。

第8章では、主にkernel/ディレクトリ内のすべてのプログラムを紹介しています。最も重要なのは、プロセススケジューラ()やsleep_on()、プログラム関連のシステムコールです。この時点ですでに、重要なプログラムのいくつかを知っているはずです。この章の最初から、C言語プログラムに埋め込まれた多くのアセンブリ言語文に遭遇します。埋め込まれたアセンブリ文の基本的な構文は、第3章で説明します。

第9章では、kernel/blk_drv/ ディレクトリにあるブロックデバイスプログラムについて説明します。本章では主に

ハードディスクやフロッピーディスクなどのブロックデバイス用のドライバです。主にファイルシステムや高速バッファを扱うためのもので、よりハードウェアに関連した内容を含んでいます。そのため、この章を読む際には、いくつかのハードウェア情報を参照する必要があります。まず、ファイルシステムのセクションを見てみるのがよいでしょう。

第10章では、kernel/chr_drv/ディレクトリにあるキャラクターデバイスドライバについて注釈をつけています。本章では、主にシリアルラインドライバ、キーボードドライバ、モニタドライバを扱います。これらのドライバは、0.12カーネルでサポートされているシリアル端末やコンソール端末のデバイスを構成しています。そのため、本章ではハードウェア関連の内容も多くなっています。読む際には、関連するハードウェアの書籍を参照する必要があります。

第11章では、kernel/math/ディレクトリにある、数学コプロセッサのシミュレーションプログラムを紹介します。本書で注釈されているカーネルのバージョンでは、まだコプロセッサが本格的にサポートされ始めていないため、この章の内容は比較的小さく、比較的簡単なものになっています。一般的な理解をしておいてください。

第12章では、カーネルソースコードのfs/ディレクトリに格納されているファイルシステムプログラムを紹介します。本章を読む際には、しばらく立ち止まって、MINIXのファイルシステムについて、

Andrew

S. Tanenbaum氏の著書「Operating System Design and Implementation」を参考にしました。チャプターは、オリジナルのLinuxシステムがMINIXファイルシステムしかサポートしていないため、Linux 0.12バージョンも例外ではありません。

第13章では、mm/ディレクトリ内のメモリ管理プログラムについて説明しています。この点を十分に理解するためには、インテル80X86マイクロプロセッサの保護モードの動作モードを十分に理解している必要があります。したがって、このプログラムのこの章を読むときには、この章の適切な場所に含まれる80X86の保護モードの動作モードの概要を参照することができます。また、この説明に加えて、第4章も同時に参照してください。この章では、ソースコード中の例題をオブジェクトとして使用することを説明していますので、メモリ管理の仕組みをより深く理解することができます。

既存のLinuxカーネル解析書では、一般的にカーネルヘッダファイルの記述が不足しており、初心者にとってはカーネルプログラムを読む上で多くの障害があります。本書の第14章では、include/ディレクトリにあるすべてのヘッダファイルを詳細に説明しています。基本的には、各定義、各定数、データ構造などが詳細にコメントされています。また、読書中の参照を容易にするために、本書では頻繁に使用される重要なデータ構造や変数を付録としてまとめていますが、これらの内容は実際には本章のヘッダーファイルに記載されています。本章の内容は、主に他の章の手順を読むためのもので

ですが、カーネルの動作メカニズムを徹底的に理解したい場合には、やはりこれらのヘッダーファイルに書かれている内容の多くを理解する必要があります。

第15章では、Linux 0.12カーネルのソースコードのlib/ディレクトリにあるすべてのファイルについて説明しています。これらのライブラリ関数ファイルは、主にコンパイルシステムなどのシステムプログラムに対するインターフェース関数を提供しており、今後のシステムソフトウェアの理解に役立つものです。このようにバージョンが低いため、ここにはあまり何も書かれていませんので、すぐに読むことができます。これが、0.12を選んだ理由のひとつです。

第16章では、tools/ ディレクトリにある build.c プログラムを紹介します。このプログラムは、コンパイルして生成したカーネルイメージファイルには含まれません。このプログラムは、カーネルのディスクブートブロックと他の主要なカーネルモジュールを接続して、完全なカーネルイメージファイルを作成するためにのみ使用されます。

第17章では、カーネルのソースコードを学ぶための実験環境と、実際に実験を行うための方法を紹介しています。主に、Bochsシミュレーション・システムでのLinuxカーネルの使用方法とコンパイル方法、ディスク・イメージ・ファイルの作成方法を紹介しています。また、Linux 0.12 のソースコードの構文を修正して、RedHat 9 システムで正しいカーネルをコンパイルできるようにする方法も紹介しています。

最後は、付録と索引です。付録では、Linuxカーネルにおける定数の定義や基本的なデータ構造の定義のほか、保護モードの動作メカニズムについて簡潔に説明しています。

また、参照を容易にするため、カーネルに使用されているPCハードウェアの情報は、本書の付録として別途掲載しています。参考文献では、以下のような書籍や論文などの情報のみを提供しました。

ソースコードを読むときに参考になるような、複雑な文献リストは用意しませんでした。私たちは、あらゆる種類の複雑で面倒な文献リストを提供しませんでした。たとえば、Linux Documentation ProjectのLDP (Linux Document Project) にあるファイルを参照する場合、LDPのウェブサイトのアドレスだけではなく、どのHOWTOの記事を参照する必要があるのかを明示的に記載します。

リーナスが初めてLinux OSのカーネルを開発したとき、主に3冊の本を参考にしました。ひとつはM.J.バッハの『UNIX Operating System Design』で、UNIX System Vカーネルの動作原理やデータ構造が書かれている。リーナスはこの本の中の多くの機能のアルゴリズムを使っている。また、Linuxカーネルのソースコードに含まれる多くの重要な関数の名前は、この本から引用されています。したがって、本書を読む際には、カーネルの動作原理についての必須の参考書となる。もう1つは、John H. Crawfordらが編集した『Programming the 80386』で、80x86のプロテクトモードのプログラミング方法を解説した良書です。また、Andrew S. Tanenbaumの『MINIX Operating System Design and Implementation』という本の初版もあります。リーナスはこの本に書かれているMINIXファイルシステムのバージョン1.0を主に使用しており、初期のLinuxカーネルでもこのファイルシステムのみをサポートしていますので、このファイルシステムの章を読むときには、ファイルシステムの動作原理 Tanenbaumの本から完全に入手できます。

各プログラムの説明では、まず、プログラムの主旨や目的、入出力パラメータ、他のプログラムとの関係などを簡単に説明した後、プログラムの完全なコードを列挙し、そのコードに詳細なコメントを付けていますが、オリジナルのC言語は一種の英語であるため、プログラムのコードやテキストは一切変更・削除されていません。また、プログラムの元となる少量の英文コメントは、定数記号や変数名などの有用な情報を多く提供しています。コードの背後には、プログラムのより詳細な解剖図や、コードに登場する言語やハードウェア関連の知識の一部が記述されています。この情報を読んだ後にプログラムを見返すと、より深い理解が得られるでしょう。

本書を読むために必要な基本的な概念の知識の紹介は、各章の対応する部分に散りばめられています。これは主に検索の便宜を図るために、ソースコードの読み解きを組み合わせることで、いくつかの基本的な概念をより深く理解することができます。

最後に注意していただきたいのは、本書で説明されている内容を完全に理解したとしても、それはLinuxの専門家になったということではないということです。あなたは、Linuxカーネルマスターになるための初期知識を得て、Linuxの旅に出ただけです。この時点で、より多くのソースコードを、できればバージョン1.0から開発中の最新の奇数バージョンまで段階的に読むべきです。本書改訂時の最新Linuxカーネルはバージョン4.16.16です。これらの開発中の最新バージョンを素早く理解し、さらに自分で提案やパッチを考えられるようになったら、思い切って挑戦してみたいと思います。

1.3 Summary

本章ではまず、Linuxの誕生と発展に欠かせない柱について詳しく説明しました。UNIXの初期のオープンソース版は、Linuxを実装するための基本原理とアルゴリズムを提供し、リチャード・ストールマンのGNUプログラムは、Linuxシステムのためのさまざまなフリーで実用的なユーティリティを提供しました。また、ツールやPOSIX規格の登場により、標準に準拠したシステムを実装するためのリファレンスガイドがLinuxに提供されています。AST社のMINIXオペレーティングシステムは、Linuxの誕生に欠かせない参考資料となり、インターネットは、Linuxが成長していくために必要な環境となっています。最後に、この章では本書の基本的な内容を紹介しています。

2 Microcomputer structure

あらゆるシステムは、図2-1に示すように、4つの基本部分からなるモデルとして見ることができます。入力部は、システムに入ってくる情報やデータを受け取るためのもので、処理センターで処理された後、出力部が送り出されます。エネルギー部は、システム全体の動作に必要なエネルギーの供給を行うもので、動作に必要なエネルギーの入力部と出力部を含む。

コンピュータ・システムの構成も例外ではなく、主にこの4つの部分で構成されています。しかし、内部的には、コンピュータシステムの処理センターと入出力部分との間のチャンネルやインターフェースは共通に使用できるので、図2-1の(b)の方が、より適切にコンピュータシステムを抽象化して表すことができるはずである。もちろん、コンピュータや多くの複雑なシステムでは、それぞれが独立して完全なサブシステムとみなすことができ、このモデルを用いて記述することも可能であり、完全なコンピュータシステムはこれらのサブシステムによって構成されます。

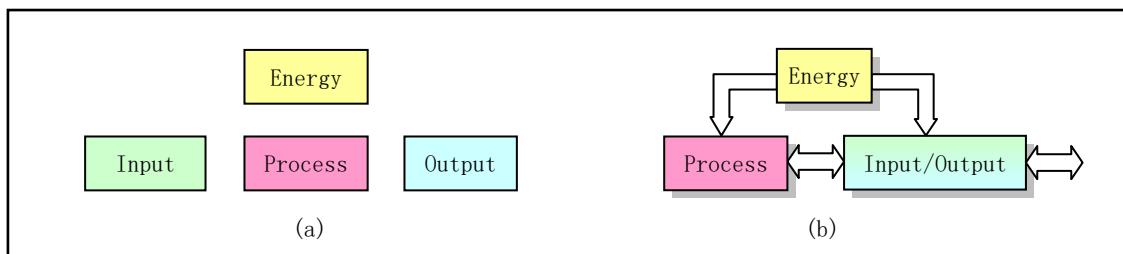


図2-1 システムの基本構成

コンピュータシステムは、ハードウェアとソフトウェアに分けられますが、これらは相互に依存しています。ハードウェア部分は、コンピュータシステムの目に見える部分であり、ソフトウェアの動作や保存のためのプラットフォームとなる。ソフトウェアは、ハードウェアの操作や動作を制御する命令の流れである。人間の脳に蓄積された情報や思考が人体の思考や行動を制御するように、ソフトウェアはコンピュータの「脳」の中の情報や思考と見ることができる。本書では、コンピュータシステムの動作メカニズムをテーマとしています。主に処理センターや入出力部などのハードウェアの構成原理と、ソフトウェア制御の実現について解説しています。ハードウェア面では、Intel 80X86 CPU (Central Processing Unit) とその互換機をベースにしたIBM PCマイクロコンピュータのハードウェアシステムを概説します。コンピュータのCPUチップは、そのままシステムの処理センターと考えてよい。バスインターフェースは他の部品と接続されています。その上で動作するソフトウェアについては、Linuxオペレーティングシステムのカーネルの実装について具体的に説明します。このように、OSは実行されるハードウェア環境と密接に関係していることがわかります。オペレーティングシステム全体を徹底的に理解するには、その動作するハードウェア環境を理解する必要があり

ます。本章では、従来のマイクロコンピュータシステムのハードウェアブロック図をもとに、マイクロコンピュータの各主要部の機能を紹介しています。これらの内容は、基本的にLinux 0.12カーネルを読み解くためのハードウェアの基礎を確立しています。説明を容易にするために、PC/ATという用語は、80386以上のCPUを搭載したIBM PCとその互換マイクロコンピュータを指し、PCはIBM PC/XTとその互換マイクロコンピュータを含むすべてのマイクロコンピュータを総称して使用します。

2.1 The Microcomputer Composition

俯瞰的な視点から、80386以上のCPUを搭載したPCシステムの構造を説明する。従来のマイクロコンピュータのハードウェアの構造を図2-2に示す。このうち、CPUはアドレス線、データ線、制御信号線で構成されるローカルバス（または内部バス）を介してシステムの他の部分と通信する。アドレスラインは、メモリやI/Oデバイスのアドレスを示すもので、データの読み書きが必要な特定の場所を示す。データラインは、CPUとメモリーやI/Oデバイスとの間のデータ転送経路となり、制御ラインは特定の読み書き動作を指示する役割を担う。80386のCPUを搭載したPCの場合、内部のアドレスラインとデータラインはそれぞれ32本あり、すべて32ビットである。したがって、アドレス空間は 2^{32} バイトで、0から4GBまである。

図では、通常、コンピュータのマザーボード上には、上位コントローラとメモリ・インターフェースが統合されている。これらのコントローラは、それぞれ大規模な集積回路チップを中心に構成された機能回路である。例えば、割り込みコントローラはインテル8259Aまたはその互換性のあるチップで構成され、DMAコントローラは通常インテル8237Aチップで構成され、タイミングカウンタはインテル8253/8254タイミングチップのコアであり、キーボードコントローラはキーボードと一緒にインテル8042チップを使用している。スキャナ回路の通信を行う。

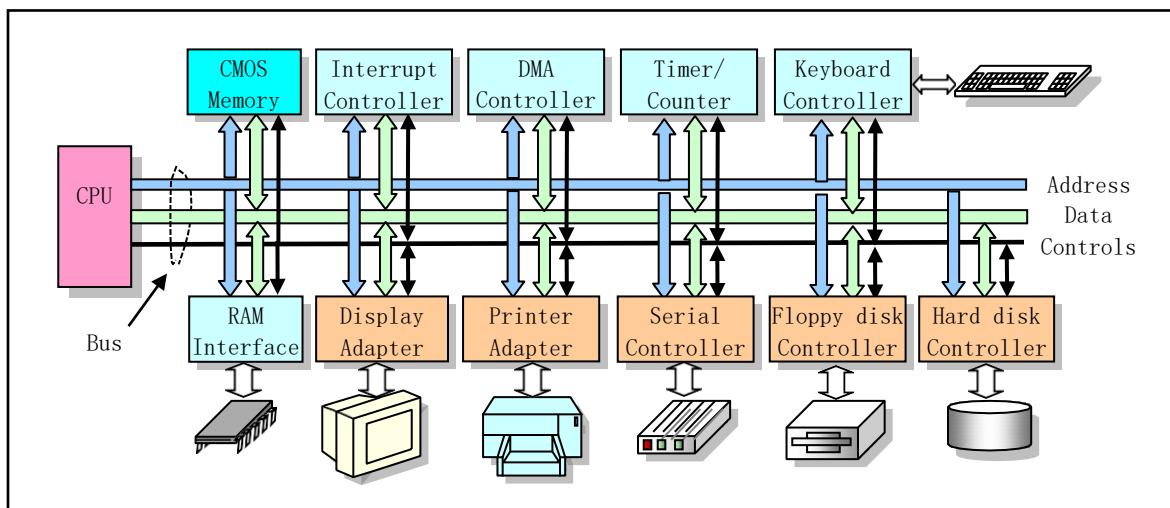


図2-2 従来のIBM PCとその互換機のブロック図

図の下側にあるコントロールカード（またはアダプター）は、拡張スロットを介してマザーボード上のシステムバスに接続されている。バススロットは、システムアドレスバス、データバス、コントロールラインの拡張デバイスコントローラへの標準的な接続インターフェースである。これらのバスインターフェース規格には、一般的に、ISA（Industry Standard Architecture）バス、EISA（Extended Industrial Standard Architecture）バス、PCI（Peripheral Component Interconnect）バス、AGP（Accelerated Graphics Port）ビデオ。バスなどがあります。これらのバスインターフェースの主な違いは、データ転送速度と制御の柔軟性である。コンピュータのハードウェアの発展に伴い、シリアル通信のポイント・ツー・ポイント技術を用いた高速のPCIE（PCI Express）バスなど、より高い転送速度と柔軟な制御が可能なバス・インターフェースが現在も登場している。オリジナルの80386マシンに

はISAバスしかないので、システムや外部のI/Oデバイスはデータ転送に16ビットのデータラインしか使えない。

コンピュータ技術の発展に伴い、従来はコントロールカードで実現していた多くの機能（ハードディスクのコントローラ機能など）が、コンピュータ本体に搭載された数個のVLSIチップに集約されている。

ボードに搭載されています。このようなチップが1つでもあれば、メインボードの主な特徴と機能が決定され、システムの各部分が最高の伝送速度を達成できるように、バス構造は大きく変化しています。現代のPCの構成は、しばしば図2-3を使って説明することができます。最近のPCのマザーボードでは、CPUの他に、主に2つのチップセット、または超大型チップで構成されたチップセットが使用されています。ノースブリッジチップとサウスブリッジチップです。ノースブリッジチップは、CPU、メモリ、AGPビデオとのインターフェースに使われる。これらのインターフェースは非常に高い伝送速度を持っています。また、ノースブリッジチップは、メモリ制御の役割も担っています。そのため、インテルはこのチップをMCH (Memory Controller Hub) チップと呼んでいます。サウスブリッジチップは、PCIバス、IDEハードディスクインターフェース、USBポートなどの低・中速コンポーネントの管理に使用される。そのため、サウスブリッジチップの名称はICH (I/O Controller Hub) となっている。この2つのチップを総称して「サウスブリッジ」と「ノースブリッジ」と呼んでいるのは、インテル社が発行する一般的なPCマザーボードに搭載されているからである。メインバージョンの下端と上端（つまり地図の南と北）に配置されており、CPUとのチャネルブリッジの役割を果たしている。

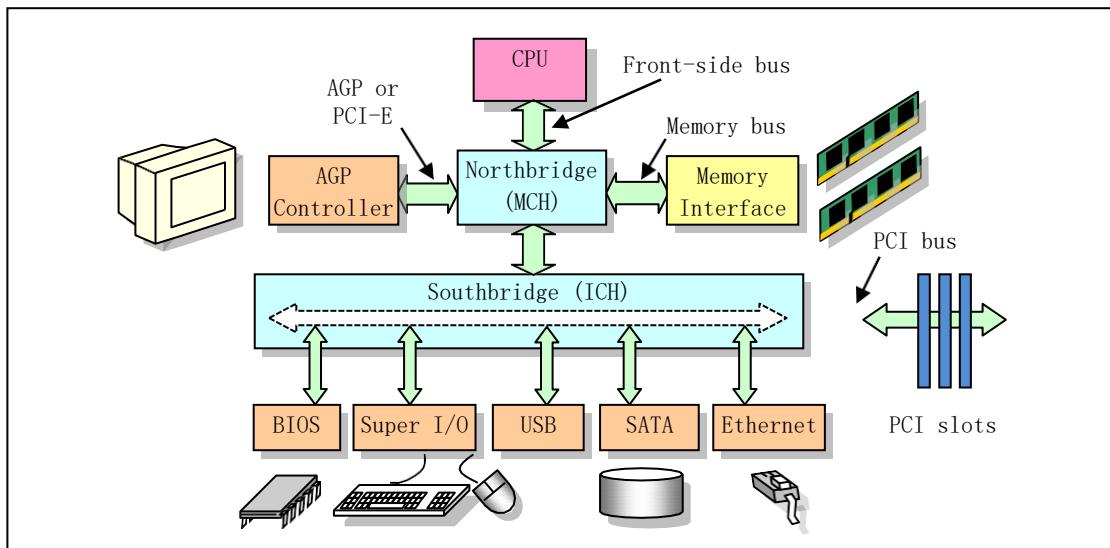


図2-3 最新のPCチップセットのブロック図

バスインターフェースは大きく変化し、将来的にはノースブリッジとサウスブリッジまでもが統合されますが、私たちプログラマーにとっては、これらの変化は従来のPCアーキテクチャとの互換性があります。したがって、従来のPCのハードウェア構造に対応したプログラムは、現在のPCでも動作させることができます。これは、インテルの開発マニュアルでも確認できます。したがって、エントリー学習を容易にするために、私たちは依然として伝統的なPCアーキテクチャの枠組みの中でPCの構成とプログラミング方法を議論し、研究しています。もちろん、これらの方法は現代のPCアーキテクチャにも適しています。以下では、図2-2の主なコントローラと制御カードのそれぞれの動作原理を概説し、それらの実際のプログラミング方法は、対応するカーネルのソースコードを読むまで延期します。

2.2 I/O Port addressing & access control

CPUとI/Oアダプタ間のデータ転送を開始する前に、まず通信アダプタのI/O位置、つまりポートアドレスを決定する必要があります。CPUとI/Oインターフェイス間のデータ転送では、さまざまな転送制

御モードが考えられます。一般的には、プログラムループ問い合わせ、割込み処理、DMA転送などが考えられます。

2.2.1 I/O ports and addressing

I/Oインターフェースコントローラーやコントロールカードのデータやステータス情報にアクセスするには、まずCPUがそれらのアドレスを指定する必要があります。このようなアドレスをI/Oポートアドレス、あるいは単にポートと呼ぶ。通常、I/Oコントローラには、データにアクセスするための「データポート」、コマンドを出力するための「コマンドポート」、コントローラの実行状況にアクセスするための「ステータスポート」がある。ポートのアドレスを設定するには、ユニファイドアレッシングとインディペンデントアレッシングの2つの方法がある。

ポートのユニファイドアレッシングの原理は、I/Oコントローラ内のポートアドレスをメモリのアレッシングアドレス空間に入れることである。したがって、このアレッシング方法は、メモリ・イメージ・アレッシングにもなる。CPUがポートにアクセスする動作は、メモリにアクセスする動作と同じであり、メモリにアクセスするための命令も使用される。ポート独立アドレスの方法は、I/Oコントローラとコントロールカードのアドレス空間を、I/Oアドレス空間と呼ばれる別のアドレス空間として扱うものである。各ポートにはそれに対応するI/Oアドレスがあり、ポートへのアクセスには専用のI/O命令を使用する。

IBM PCとその互換マイクロコンピュータは、主に独立アドレスモードを採用しており、制御装置のレジスタをアドレス指定してアクセスするために、独立したI/Oアドレス空間を使用しています。ISAバスアーキテクチャを使用する従来のPCは、0x000から0x3FFまでのI/Oアドレス空間を持ち、1024個のI/Oポートアドレスが利用可能です。各コントローラやコントロールカードが使用するデフォルトのポートアドレス範囲を表2-1に示します。これらのポートの使用方法やプログラミング方法については、後に関連するハードウェアが具体的に登場したときに詳しく説明します。

また、IBM PCではユニファイド・アレッシング・モードを部分的に使用しています。例えば、CGAディスプレイカードのディスプレイメモリのアドレスは、メモリアドレス空間0xB800～0xBC00の範囲を直接占有しています。したがって、画面に文字を表示したい場合は、メモリ操作命令を使って、このメモリ領域に直接書き込み操作を行うことができます。

| 表2-1 I/Oポートのアドレス割り当て Address range | Allocation description |
|---------------------------------------|---|
| 0x000 -- 0x01F | 8237A DMA controller 1 |
| 0x020 -- 0x03F | 8259A Programmable Interrupt Controller 1 |
| 0x040 -- 0x05F | 8253/8254A Timer Counter |
| 0x060 -- 0x06F | 8042 Keyboard Controller |
| 0x070 -- 0x07F | Access CMOS RAM/Real-Time Clock RTC Port |
| 0x080 -- 0x09F | DMA page register access port |
| 0x0A0 -- 0x0BF | 8259A Programmable Interrupt Controller 2 |

| | |
|----------------|-----------------------------------|
| 0x0C0 -- 0x0DF | 8237A DMA Controller 2 |
| 0x0F0 -- 0x0FF | Coprocessor access port |
| 0x170 -- 0x177 | IDE hard disk controller 1 |
| 0x1F0 -- 0x1F7 | IDE hard disk controller 0 |
| 0x278 -- 0x27F | Parallel printer port 2 |
| 0x2F8 -- 0x2FF | Serial Controller 2 |
| 0x378 -- 0x37F | Parallel printer port 1 |
| 0x3B0 -- 0x3BF | Monochrome MDA display controller |
| 0x3C0 -- 0x3CF | Color CGA display controller |
| 0x3D0 -- 0x3DF | Color EGA/VGA display controller |

| | |
|----------------|-------------------------|
| 0x3F0 -- 0x3F7 | Floppy drive controller |
| 0x3F8 -- 0x3FF | Serial Controller 1 |

EISAやPCIなどのバスアーキテクチャを採用している最近のPCでは、64KBのI/Oアドレス空間が利用可能です。関連するコントローラーや設定が使用するI/Oアドレスの範囲は、通常のLinuxシステムでは、/proc/ioportsファイルを見るなどで得ることができます。以下を参照してください。

```
[root@plinux root]# cat /proc/ioports
0000-001F : DMA1
0020-003F : PIC1
0040-005F : タイマー
0060-006F : キーボー
ド 0070-007F : rtc
0080-008f : DMAページレ
ジ 00a0-00bf : Pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : シリアル(オ
ト) 0376-0376 : ide1
03c0-03df :
VGA+ 03f6-03f6 :
IDE0
03f8-03ff : シリアル(オート)
0500-051f : PCI デバイス 8086:24d3 (Intel Corp.)
0cf8-0cff : PCI conf1
da00-daff : VIA Technologies, Inc. VT6102 [Rhine-II]
da00-daff : VIA-RHINE
e000-e01f : PCI デバイス 8086:24d4 (Intel Corp.)
e000-e01f : usb-uhci
e100-e11f : PCI デバイス 8086:24d7 (Intel Corp.)
e100-e11f : usb-uhci
e200-e21f : PCI デバイス 8086:24de (Intel Corp.)
e200-e21f : usb-uhci
e300-e31f : PCI デバイス 8086:24d2 (Intel Corp.)
e300-e31f : usb-uhci
f000-f00f : PCI デバイス 8086:24db (Intel
Corp.) f000-f007 : ide0
f008-f01f : underroot #.
```

2.2.2 Interface access control

PC I/Oインターフェイスのデータ転送制御モードは、一般的にプログラムのループ問い合わせモード、割り込み処理モード、DMA転送モードを採用することができます。周期問い合わせモードとは、その名の通り、CPUがプログラムをループさせて指定されたデバイスコントローラに状態を問い合わせることで、デバイスとのデータ交換が可能かどうかを判断するものです。この方法は、過剰なハードウェアのサポートを必要とせず、使用方法やプログラムも比較的簡単ですが、貴重なCPU時間を消費します。そのため、マルチタスクOSでは、待ち時間が極端に短い場合や必要な場合を除いて、この方法は使用しない方が良いでしょう。Linux OSでは

システムでは、この方法は、デバイスやコントローラがすぐに情報を返すことができる一部の場所でのみ使用されます。

割り込み処理制御方式は、割り込みコントローラのサポートが必要です。この制御モードでは、I/OデバイスがCPUに割り込みによる処理要求をした場合にのみ、CPUは現在実行中のプログラムを一時的に中断し、対応するI/O割り込み処理サービスプロセスを実行する。割り込み処理サービスプロセスの実行後、CPUは先ほど割り込まれたプログラムの実行を継続します。I/Oコントローラやデバイスが割り込み要求を出すと、CPUは割り込みベクタテーブル（または割り込みディスクリプターテーブル）を用いて、対応する割り込み処理サービスプロセスのエントリーアドレスを指定します。したがって、割り込み制御モードを使用する場合は、まず割り込みベクタテーブルを設定し、対応する割り込み処理サービスプロセスをコンパイルする必要があります。Linux OSでは、ほとんどのデバイスI/O制御が割り込み処理を使用しています。

I/Oデバイスとシステムメモリー間の一括データ転送には、DMA（ダイレクトメモリアクセス）方式が採用されています。すべての動作プロセスは、CPUを介さずに専用のDMAコントローラを使用する必要があります。転送中にソフトウェアを介在させる必要がないため、非常に効率的に動作します。Linux OSでは、フロッピーディスクドライバーが割り込みとDMA方式でデータ転送を実現している。

2.3 Main memory, BIOS and CMOS memory

2.3.1 一般的なPCには3種類のメモリが搭載されています。1つはプログラムの実行やデータの一時保存に使用されるメインメモリのRAM (Random Access Memory) 、もう1つはシステムのブート診断やハードウェアプログラムの初期化に使用されるROM (Read Only Memory) メモリ、そして3つ目はコンピュータのリアルタイムクロック情報やシステムハードウェアの設定情報を保存する少量のCMOSメモリです。

2.3.2 Main memory

1981年にIBM PCが登場したとき、システムには640KBのRAMメインメモリ（メモリと呼ばれる）しかなかった。使用されているCPUの8088/8086はアドレスラインが20本しかないため、メモリのアドレス範囲は最大で1024KB（1MB）である。オペレーティングシステム「DOS」が普及していた当時は、640Kや1MBのメモリ容量でも、基本的には通常のアプリケーションには十分だった。コンピュータのソフトウェアおよびハードウェア技術の急速な発展に伴い、現在のコンピュータは512MB以上の物理メモリ容量を持つ構成が一般的であり、すべてインテル社の32ビットCPU、つまりPC/ATコンピュータが使用されている。そのため、CPUの物理メモリのアドレス範囲は4GBまでとなっています（CPUの新機能を利用することで、64GBの物理メモリ容量をアドレス指定することも可能です）。しかし、ソフトウェア的にオリジナルのPCと互換性を持たせるために、システム1MB以下の物理メモリの割り当てはまだ基本的にはオリジナルのPCと同じままでですが、オリジナルシステムのROMのBIOSは常にCPUがアドレス可能なメモリの中で最も高い位置にありました。最後の場所では、BIOSの元の

場所は、コンピュータの初期化時にBIOSのシャドウ領域として使用され、すなわち、BIOSコードは依然としてこの領域にコピーされます。図2-4をご覧ください。

コンピュータの電源を投入すると、物理メモリはアドレス0から始まる連続した領域に設定されます。0xA0000～0xFFFFF（384K～1Mの合計384K）と0xFFE0000～0xFFFFFFFF（4Gの最後の64K）のアドレス範囲を除くすべてのメモリがシステムメモリとして使用できます。この2つの特定の範囲は、I/OデバイスやBIOSプログラムに使用されます。コンピュータに16MBの物理メモリがある場合、Linux 0.1xシステムでは0～640Kがカーネルコードとデータの格納に使用されます。Linuxカーネルは、BIOSの機能を使用せず、BIOSが設定した割り込みベクトルテーブルも使用しません。640Kと1Mの間の384Kは、まだ図に示した用途のために確保されています。このうち、アドレス0xA0000から始まる128Kはディスプレイメモリバッファとして使用され、その部分は他のコントロールカードのROM BIOSやそのマッピング領域に使用され、0xF0000から1Mの範囲は使用されます。

は、ハイエンドシステムのROM BIOSのマッピング領域として使用されます。1M～16Mは、カーネルが割り当て可能なメインメモリ領域として使用する。また、カーネルのコードやデータの背後には、高速バッファやメモリ仮想ディスクなどもメモリ領域の一部を占めている。この領域は通常640K～1Mにわたる。

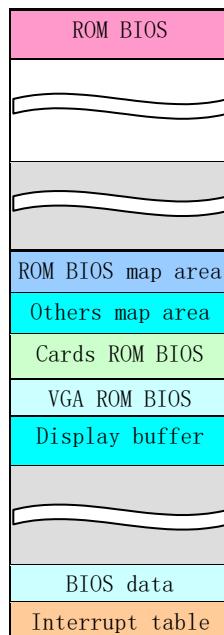


図2-4 PC/AT機のメモリ領域使用マップ

2.3.3 Basic input/output program BIOS

ROMに格納されているシステムBIOSプログラムは、主にコンピュータの電源投入時にシステム各部のセルフチェックを実行し、割り込みベクトルテーブルやハードディスクパラメーターテーブルなど、OSが使用する必要のある各種設定テーブルを確立します。また、プロセッサやその他のシステムを既知の状態に初期化したり、DOSなどのOSにハードウェアデバイスのインターフェースサービスを提供したりする。しかし、BIOSが提供するこれらのサービスはリエントランシー（プログラムの同時実行ができないこと）ではないため、アクセス効率を考慮すると、初期化時にBIOSを使用して一部のシステムパラメータを提供することを除いて、Linux OSも同時に実行されます。BIOSの機能を使用しないでください。

コンピュータシステムの電源を入れたり、筐体のリセットボタンを押したりすると、CPUは自動的にコードセグメントレジスタCSを0xF000に設定し、セグメントベースアドレスを0xFFFF0000に設定し、セグメント長を64KBに設定します。IPは0xFFFF0に設定されているので、CPUのコードポインタは4G空間の最後の64Kの最後の16バイトである0xFFFFFFF0を指していることになります。上の図から、ここにはシステムROMのBIOSが格納されています。そして、BIOSはここに、BIOSコードの64KB範囲

内の命令にジャンプして実行を開始するためのジャンプ命令JMPを格納する。PC/ATマイコンのBIOS容量は1MB～2MBが主流で、フラッシュメモリROMに格納されているため、64KB以上の範囲のBIOSを実行・アクセスできるようにするためには、0～1Mのアドレス空間からは離れています。その他のBIOSコードやデータは、まずBIOSプログラムが32ビットアクセスを使用して、データセグメントレジスタのアクセス範囲を（本来の64Kではなく）4Gに設定し、CPUが0～4Gの範囲のデータを実行・操作できるようにします。その後、BIOSがいくつかの列のハードウェア検出と初期化操作を行った後、元のPCと互換性のある64KBのBIOSコードとデータを64Kにコピーして

を1Mメモリのローエンドに設定し、ここにジャンプしてCPUを実在させます。図2-5のようにリアルアドレスモードで実行します。最後に、BIOSはハードディスクやその他のブロックデバイスからOSのブートプログラムを0x7c00のメモリにロードし、この場所にジャンプしてブートプロセスを続行します。

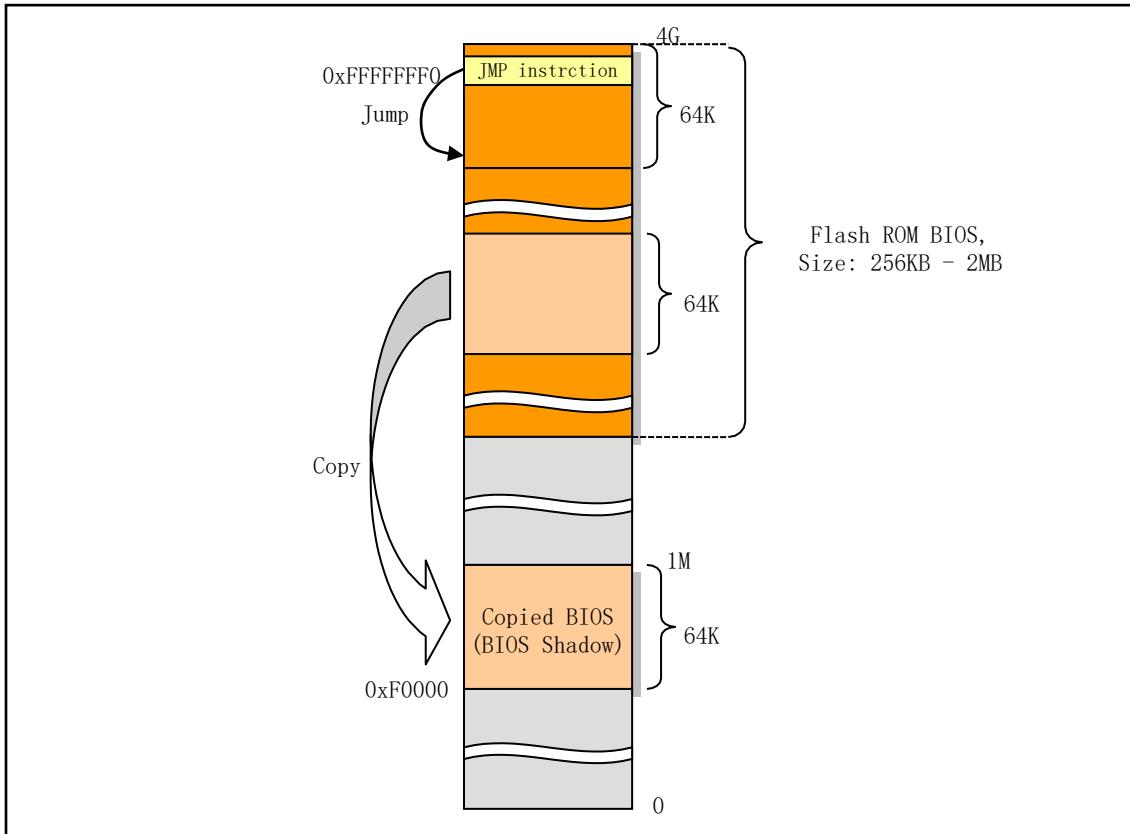


図2-5 フラッシュROMのBIOS位置とコピーマッピング領域

2.3.4 CMOS memory

PC/AT機では、メモリやROM BIOSを使用する必要があるほか、コンピュータのリアルタイムクロック情報やシステムハードウェアの構成情報を格納するために、記憶容量の少ない（64バイトまたは128バイト）CMOSメモリが使用されている。この部分のメモリは、通常、リアルタイムチップと一体化したブロックになっている。CMOSメモリのアドレス空間は基本メモリのアドレス空間の外にあるため、I/O命令を使ってアクセスする必要がある。

2.4 Controllers and Control Cards

図2-2からわかるように、PCには、データの転送やコンピュータの動作を制御するためのさまざまな制御カードやコントローラが搭載されています。これらのコントローラやコントロールカードには、主に割り込みコントローラ、DMAコントローラ、キーボードコントローラ、フロッピー/ハードディスクコントロールカード、シリアル通信コントロールカード、ディスプレイコントロールカードなどがあります。ここで、「コントローラ」とは、コンピュータのマザーボード上に組み込まれた制御部品を指し、「コントロールカード」とは、拡張スロットからコンピュータに挿入される制御カード部品を指す。

指す。制御装置は、独立した制御カードの形で存在することもあれば、コンピュータの統合度の向上に伴ってメインボードに統合されることもあるため、コントローラと制御カードの間に実質的な違いはありません。以下、これらの制御装置について詳しく説明します。

2.4.1 Interrupt controller

IBM PC/AT 80X86互換マイクロコンピュータでは、カスケード接続された2つの8259Aプログラマブル割込み制御チップを使用して、I/Oデバイス割込み制御データアクセスのための割込みコントローラを構成し、15個のデバイスに対して独立した割込みを提供することができます。その制御機能を図2-6に示す。コンピュータの初期起動時に、ROM BIOSは2つの8259Aチップを初期化し、クロックタイマ、キーボード、シリアルポート、プリントポート、フロッピーディスクコントローラ、コプロセッサ、ハードディスクに15段階の割り込み優先順位を割り当てます。機器やコントローラを使用する。同時に、メモリ先頭の0x000-0xFFFFエリアに割り込みベクターテーブルが作成され、これらの割り込み要求は、表2-2に示すように、0x08から始まる割り込みベクタ番号にマッピングされます。

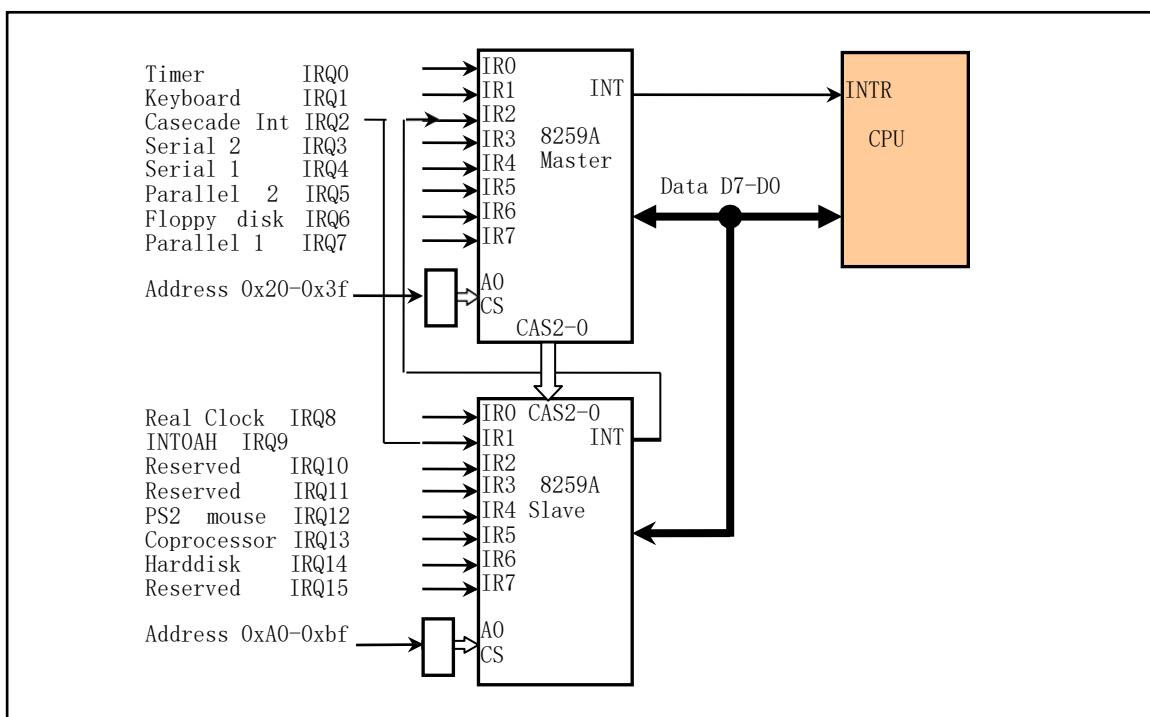


図2-6 8259コントロールシステムに接続されたPC/ATマイクロコンピュータ

しかし、0x00--0x1Fという割り込み番号は、CPU用に特別に確保されたインテルのものであるため、これらのBIOS設定はインテルの要求に抵触します。この問題を解決するために、Linux OSはBIOSで設定されたこれらの割り込み番号を直接使用しません。パワーオン起動時、Linux OSはカーネルの初期化時に8259Aを再設定し、すべてのシステム・ハードウェア割り込み要求番号を0X20以上の割り込み番号にマッピングします。割り込みコントローラの動作やプログラム方法の詳細については、以降のセクションを参照してください。

| 表2-2 電源投入時にROM BIOSが設定するハードウェ | Interrupt number Set by the BIOS | Usage description |
|-------------------------------|----------------------------------|-------------------|
| | | |

| ア要求割り 込み番号 IRQ number | | |
|-----------------------------|-----------|-----------------------------------|
| IRQ0 | 0x08 (8) | 8250 issued 100HZ clock interrupt |
| IRQ1 | 0x09 (9) | Keyboard interrupt |
| IRQ2 | 0x0A (10) | The slave chip's interrupt |

| | | |
|-------|------------|--------------------------------|
| IRQ3 | 0x0B (11) | Serial port 2 |
| IRQ4 | 0x0C (12) | Serial port 1 |
| IRQ5 | 0x0D (13) | Parallel port 2 |
| IRQ6 | 0x0E (14) | Floppy disk drive |
| IRQ7 | 0x0F (15) | Parallel port 1 |
| IRQ8 | 0x70 (112) | Real-time clock interrupt |
| IRQ9 | 0x71 (113) | Change to INT 0x0A |
| IRQ10 | 0x72 (114) | Reserved |
| IRQ11 | 0x73 (115) | Reserved (network interface) |
| IRQ12 | 0x74 (116) | PS/2 mouse port interrupt |
| IRQ13 | 0x75 (117) | Math coprocessor interrupt |
| IRQ14 | 0x76 (118) | Hard disk controller interrupt |
| IRQ15 | 0x77 (119) | Reserved |

2.4.2 DMA controller

前述したように、DMAコントローラーの主な機能は、外部デバイスがメモリに直接データを転送することで、システムのパフォーマンスを向上させることです。通常は、マシンに搭載されているインテル8237チップまたはその互換チップによって実装されています。DMAコントローラーをプログラムすることで、周辺機器とメモリー間のデータ転送をCPUの制御なしに行うことができます。そのため、データ転送の間、CPUは他の作業を行うことができます。

2.4.3 PC/AT機では、8237チップが2個使用されているので、DMAコントローラには8つの独立したチャンネルが用意されています。そのうち最後の4つは16ビットチャンネルです。フロッピーディスクコントローラは、特にDMAチャンネル2を使用するように指定されています。チャンネルを使用する前にまず設定する必要があります。これには、ページレジスタポート、（オフセット）アドレスレジスタポート、データカウントレジスタポートの3つのポートに対する操作が必要です。DMAレジスタは8ビット、アドレス値とカウント値は16ビットの値なので、それぞれを2回送信する必要があります。

2.4.4 Timer/counter

インテル8253/8254は、コンピュータで正確な時間遅延を扱うために設計されたプログラマブル・インターバル・タイマー（PIT）チップです。このチップは、3つの独立した16ビットカウンターチャンネルを備えています。各チャネルは異なる動作モードで動作し、これらの動作モードはすべてソフトウェアで設定できます。ソフトウェアで遅延を行うには、ループ演算文を実行する方法がありますが、そのためにはCPU時間を消費します。機械に8253/8254チップが使用されている場合、プログラマは8253を独自の要件に合わせて設定し、カウンタ・チャネルの1つを使用して目的の遅延を実現することができます。遅延時間が経過すると、8253/8254はCPUに割り込み信号を送ります。

2.4.5 PC/ATおよびその互換マイクロコンピュータシステムには、8254チップが使用されています。3つのタイマ/カウンタチャネルは、時間帯別クロック割り込み、ダイナミックメモリのDRAMリフレッシュ

タイミング回路、ホストスピーカの音色合成に使用されています。Linux 0.12のOSでは、カウンタがモード3で動作するようにチャネル0のみをリセットし、10ミリ秒ごとに信号を送信して割り込み要求信号 (IRQ0) を生成します。この間隔で発生する割り込み要求が、Linux 0.12カーネルのパルスとなります。現在実行中のタスクを定期的に切り替え、各タスクが使用するシステムリソース（時間）をカウントするために使用されます。

2.4.6 Keyboard controller

私たちが今使っているキーボードは、1984年にIBMが発売したPC/AT互換機用のキーボードです。通常、AT-PS/2互換キーボードと呼ばれ、101～104個のボタンが付いています。このキーボードにはプロセッサー（インテル8048またはその互換チップ）が搭載されている。

キーボードのエンコーダーと呼ばれるものです。これは、すべてのキーを押したり離したりしたときのステータス情報（すなわちスキャンコード）をスキャンして収集し、ホストコンピューターのメインボードにあるキーボードコントローラに送信するためのものです。キーが押されたときにキーボードから送られてくるスキャンコードをMake code、または単にconnect codeと呼び、押されたキーが離されたときに送られてくるスキャンコードをdisconnectedと呼ぶ。ブレークコード、または単にブレークコードと呼ばれる。

- ホストキーボードコントローラは、受信したキーボードのスキャンコードをデコードし、デコードしたデータをオペレーティングシステムのキーボードデータキューに送るように特別に設計されています。キーボードコントローラは、各キーのオンとオフのコードが異なるため、スキャンコードに基づいて、ユーザーがどのキーを操作しているかを判断することができます。キーボード全体のすべてのキーのオンとオフのコードは、キーボードのスキャンコードセットを形成する。コンピュータの発展に伴い、現在では3つのスキャンコードセットが存在する。それらは

- The first set of scan codes -- The original XT keyboard scan code set. The current keyboard has rarely sent such scan codes;
- The second set of scan codes -- The default scan code set used by modern keyboards, commonly referred to as the AT keyboard scan code set;
- The third set of scan codes -- PS/2 keyboard scan code set. The scan code set used by the original IBM launch of the PS/2 microcomputer has rarely been used.

ATキーボードは、デフォルトで2番目のスキャンコードを送信します。これにもかかわらず、ホスト・キーボード・コントローラは、図2-7に示すように、PC/XTソフトウェアとの互換性のために、受信したすべてのセカンド・キーボード・スキャン・コードをファースト・スキャン・コードに変換します。そのため、キーボードコントローラをプログラミングする際には、通常、最初のスキャンコードのセットだけ知っていればよいのです。これは、キーボードのプログラミングに関して、XTキーボードのスキャンコードセットのみが与えられている理由でもあります。

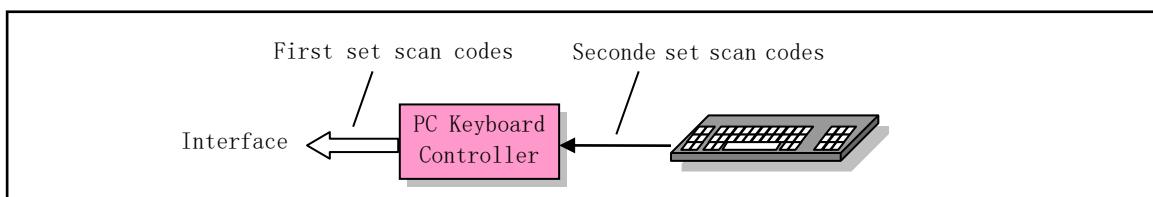


図2-7 キーボードコントローラによるスキャンコードセットの変換

キーボードコントローラには、一般的にインテル社のシングルチップ・マイクロプロセッサー・チップ「8042」またはその互換回路が使用されています。現在のPCでは、キーボードコントローラはマザーボードのチップセットに組み込まれているが、機能的には8042チップを使用したコントローラーと互換性がある。キーボードコントローラは、キーボードから送られてくる11ビットのシリアル形式のデータを受け取る。1ビット目はスタートビット、2-9ビット目は8ビットのキーボードスキ

サンコード、10ビット目はパリティチェックビット、11ビット目はストップビットです。次項のシリアルコントロールカードの説明を参照してください。11ビットのシリアルデータを受信したキーボードコントローラは、キーボードスサンコードをPC/XT標準キーボード互換システムスサンコードに変換した後、割り込みコントローラのIRQ1端子を介してCPUに割り込み要求を送信します。CPUが割り込み要求に応答すると、キーボード割り込みハンドラが呼び出され、コントローラ内のXTキーボードスサンコードを読み取ります。

キーが押されると、キーボードコントローラポートからXTキーパッドのアクセスコードを受け取ることができます。このスサンコードは、キーボード上のある場所のキーが押されたことを示すだけで、まだ文字コードにマッピングされていません。接続コードは通常1バイト幅です。例えば、キー「A」のキーオンコードは30 (0x1E) である。押したキーが離されると、キーボードコントローラポートからブレークコードが受信されます。XTキーボードの場合（キーボードコントローラのプログラミングポートが受信するスサンコード）、切断コードは

接続コードは、その接続コードに0x80を加えたもの、つまり最上位ビット（ビット7）がセットされているときに使用します。例えば、上記の「A」キーのブレークコードは、 $0x80 + 0x1E = 0x9E$ となります。

しかし、PC/XT標準の83キーキーボードに新たに追加された（「拡張」された）ATキーボードキー（右のCtrlキーや右のAltキーなど）については、そのオン/オフのスキャンコードは通常2～4バイトで、1バイト目は0xE0でなければならない。例えば、拡張されていない左のCtrlキーを押すと、1バイトのパスコード0x1Dが生成され、右のCtrlキーを押すと、拡張された2バイトのパスコード0xE0, 0x1Dが生成される。対応するブレークコードは 0xE0, 0x9D です。表2-3は、スキャンコードのオン/オフを切り替えるいくつかの例を示しています。さらに、スキャンコードの完全な第1セットも付録として与えられています。

| 表2-3 キーボードコントローラポートで受信した最初のスキャンコードセットの例 Pressed key | Connect scan code | Break scan code | Description |
|--|-------------------|-----------------|-------------------------------|
| A | 0x1E | 0x9E | Non-expanding ordinary keys |
| 9 | 0x0A | 0x8A | Non-expanding ordinary keys |
| Function key F9 | 0x43 | 0xC3 | Non-expanding ordinary keys |
| Arrow key right | 0xe0, 0x4D | 0xe0, 0xCD | Extended keys |
| Right Ctrl key | 0xe0, 0x1D | 0xe0, 0x9D | Extended keys |
| Left Shift + G | 0x2A, 0x22 | 0xAA, 0xA2 | Press and release Shift first |

2.4.7 また、キーボードコントローラ8042の出力ポートP2は、他の目的にも使用されます。P20端子は、CPUのリセット動作を実現するために使用され、P21端子は、A20信号線のオープンを制御するために使用されます。出力ポートのビット1（P21）が1の時、A20信号線をオン（ゲート）にし、0の時、A20信号線をディスエーブルにします。今日のマザーボードでは、もはや個別の8042チップは搭載されていませんが、マザーボード上の他の集積回路は、互換性のために8042チップの機能をエミュレートします。そのため、現在ではキーボードのプログラミングは、まだ8042のプログラミング方法を使用しています。

2.4.8 Serial control card

1. Asynchronous serial communication principle

2台のコンピューター/機器がデータを交換する、すなわち通信は、人が話すのと同じ言語を使わなければならぬ。コンピュータ通信の用語では、コンピュータ／機器とコンピュータ／機器の間の「言語」を通信プロトコルと呼んでいます。通信プロトコルは、有効なデータ長の単位を送信するためのフォーマットを規定しています。通常、このフォーマットを「フレーム」という言葉で表現します。また、通信当事者が送受信の順番を決定したり、一部のエラー検出動作を行うために、必要なデータ

に加えて、同期やエラー検出に使用する他の情報も、例えば、データ情報の送信開始前に、送信される1フレームの情報に含まれます。図2-8のように、最初に開始/同期または通信制御情報を送信し、必要なデータ情報を送信した後に、いくつかの検証情報を送信します。

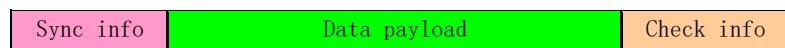


図2-8 通信フレームの一般的な構造

2. シリアル通信とは、ビット単位のデータストリームを1ビットずつ回線で伝送する通信方式のこと。シリアル通信は、非同期型シリアル通信と同期型シリアル通信に分類される。両者の主な違いは、送信時に同期させる通信単位（フレーム）の長さの違いである。非同期シリアル通信では1文字を1通信単位または1フレームとして伝送し、同期シリアル通信では複数の文字またはバイトの並びを1フレームのデータとして伝送する。人と人との対話に例えれば、非同期通信は2人の会話のスピードが遅いようなものだ。話すときは「語呂合わせ」で、一語一語話した後に任意の長さで一時停止することができます。一方、同期通信は、2者間の会話を一貫した文章で行うようなものです。実際に伝送単位を1ビット（文字で！）にしてみると、1文字の非同期シリアル通信も、同時送信のクロック信号の同期送信とみなすことができるところがわかる。通信の方法。

3. Asynchronous serial transmission format

非同期シリアル通信のフレームフォーマットを図2-9に示す。1文字の传送は、スタートビット、データビット、パリティビット、ストップビットで構成されます。スタートビットは同期の役割を果たしており、値は常に0です。データビットは、実際に送信されるデータ、つまり1文字のコードです。データビットの長さは5~8ビットです。パリティビットはオプションで、プログラムによって設定されます。ストップビットは常に1で、プログラムによって1ビット、1.5ビット、2ビットに設定できます。通信が情報の送信を開始する前に、双方が同じフォーマットに設定されている必要があります。データビットとストップビットの数が同じであれば。非同期通信の仕様では、送信1をMARK、送信0をSPACEと呼んでいます。そのため、以下の説明ではこの2つの用語を使用します。

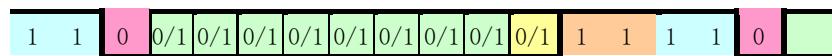


図2-9 調歩式シリアル通信のキャラクター送信フォーマット

データ送信がない場合、送信者はMARK状態となり、連続して1を送信します。データを送信する必要がある場合、送信者はまず、ビット間隔のスペーススタートビットを送信する必要があります。スペース番号を受信した後、受信者は送信者との同期を開始し、その後のデータを受信します。プログラムでパリティビットが設定されている場合は、データ送信後にパリティビットを受信する必要があります。最後はストップビットです。文字フレームを送信した後、すぐに次の文字フレームを送信する場合と、パスワードを一時的に送信し、しばらくしてから文字フレームを送信する場合があります。

キャラクタフレームを受信する際、受信機は次の3つのエラーのいずれかを検出することができます。
(1)パリティエラー。この時、プログラムは相手に文字の再送を依頼しなければなりません。このエラーは、プログラムが受信速度よりも遅い速度で文字を取り込んでいるために発生します。この場合、

プログラムを修正して、文字の周波数の取得を高速化する必要があります。 (3) フレームフォーマットが不正です。このエラーは、受信を要求されたフォーマット情報が正しくない場合に発生します。例えば、空の番号を受信した場合

4. ストップビットを受信すべき時に 通常、このようなエラーは、回線の干渉を除く、両者のフレームフォーマットの違いによって発生します。

5. Serial controller

シリアル通信を実現するために、PCには通常、RS-232Cに準拠したシリアルインターフェースが2つ搭載されており、UART (Universal Asynchronous Receiver/Transmitter) で構成されたシリアルコントローラーを用いて処理を行います。シリアルデータの送受信を行います。PCのシリアルインターフェースには、通常25ピンのDB-25または9ピンのDB-9コネクタが使用されており、主にMODEM機器を接続して動作させるために使用されています。そのため、RS-232C規格では多くのMODEM専用インターフェースピンが規定されています。

- 2.4.9** 以前のPCは全てナショナルセミコンダクター社のNS8250またはNS16450のUARTチップを使用しています。現在のPCでは、16650Aとその互換チップを使用していますが、NS8250/16450チップとの互換性はあります。NS8250/16450と16650Aの主な違いは、16650AチップがFIFO転送もサポートしていることです。このモードでは、UARTは最大16文字の送受信を行った後にのみ割り込みを発生させることができます。そのため、システムやCPUの負担を軽減することができます。PCの電源を入れると、RESET信号がNS8250のMRピンを通過して、UARTの内部レジスタと制御ロジックをリセットします。その後、UARTを使用したい場合は、初期プログラミング操作を行い、UARTの動作ボーレート、データビット、動作モードを設定する必要があります。

2.4.10 Display control

1. IBM PC/ATおよびその互換機では、カラーおよびモノクロのビデオカードが使用できる。IBMの初期のPCビデオシステム規格には、モノクロのMDA規格やカラーのCGA規格のほか、EGAやVGA規格がある。後発の高機能グラフィックカード（現在のAGPグラフィックカードを含む）は、いずれもグラフィック処理速度やスマートアクセラレーション処理能力が極めて高いが、いずれもこれらの初期規格に対応している。Linux 0.1xのOSでは、これらの規格でサポートされているテキスト表示方法のみを採用しています。

2. MDA display standard

モノクロディスプレイアダプタMDA(Monochrome Display Adapter)は、モノクロ表示のみ対応しています。また、独自のテキスト文字表示モード（BIOS表示モード7）にのみ対応しています。画面表示の仕様は、80列×25行（列番号x=0～79、行番号y=0～24）で、合計2000文字を表示することができます。1文字に1つの属性バイトが付いているので、1画面（1フレーム）を表示するのに4KBが必要となる。偶数アドレスバイトには文字コードが、奇数アドレスバイトには表示属性が格納される。MDAカードには8KBのディスプレイメモリが搭載されています。PCのメモリアドレス範囲には、0xb0000から始まる8KBの空間（0xb0000～0xb2000）が占有されています。ディスプレイの画面番号がvideo_num_lines = 25、列数がvideo_num_columns = 80の場合、画面の列の行値x, yに位置する文字や属性のメモリ上の位置は次のようになります。

$$\text{属性バイト位置} = 0xb0000 + \text{video_num_columns} * 2 * y + x * 2 ; .$$

MDAのモノクロ文字表示モードでは、各文字の属性バイトフォーマットを表2-4に示す。このうち、D7を1にすると文字が点滅し、D3を1にすると文字が強調表示される。基本的には、図2-10のカラーテキスト文字の属性バイトと同じであるが、色は白（0x111）と黒（0x000）の2色のみである。それらを

組み合わせた効果を表に示します。

| Foreground color D2D1D0 | Attribute value No flash low | display effect | example |
|----------------------------|---------------------------------|----------------|--|
| Background color D6D5D4 | | | |
| 0 0 0 | 0 0 0 | 0x00 | Characters are not visible. |
| 0 0 0 | 1 1 1 | 0x07 | White characters displayed on a black background (normal display). |
| 0 0 0 | 0 0 1 | 0x01 | White underlined characters displayed on a black background. |
| 1 1 1 | 0 0 0 | 0x70 | Black characters displayed on a white background (inverse). |
| 1 1 1 | 1 1 1 | 0x77 | Show white squares. ■ |

3. CGA display standard

カラーグラフィックアダプターCGA (Color Graphics Adapter) は、7種類のカラー・グラフィック表示に対応しています (BIOS表示0--6)。80カラム×25カラムのテキスト文字表示モードでは、モノクロ2種類、カラー16種類の表示モードがあります (BIOS表示モード2--3)。CGAカードには標準で16KBのディスプレイメモリ (メモリアドレス範囲0xb8000～0xbc000) が搭載されているので、合計4フレーム分の表示情報を保存することができます。同様に、1フレームあたり4KBの表示メモリには、偶数アドレスのバイトには文字コードが、奇数アドレスのバイトには文字の表示属性が格納されています。しかし、console.cプログラムでは、8KBの表示メモリ (0xb8000～0xba000) しか使用していません。CGAカラーテキスト表示モードでは、各表示文字の属性バイトフォーマットの定義を図2-10に示す。



図 2-10 文字属性フォーマット定義

モノクロ表示と同様に、D7を1にすると表示文字が点滅し、D3を1にすると文字が強調表示され、D6、D5、D4およびD2、D1、D0の各ビットを組み合わせることで8色の表示が可能となります。前景ビットと高輝度ビットの組み合わせで、残りの8色の文字色を表示することができます。これらの組み合わせの色を表2-5に示します。

| 表2-5 描画色 と背景 色（左 半分） I R G B | Value | Color name | I R G B | Value | Color name |
|---|-------|------------|---------|-------|-------------|
| 0 0 0 0 | 0x00 | Black | 1 0 0 0 | 0x08 | Dark grey |
| 0 0 0 1 | 0x01 | Blue | 1 0 0 1 | 0x09 | Light blue |
| 0 0 1 0 | 0x02 | Green | 1 0 1 0 | 0x0a | Light green |

| | | | | | |
|---------|------|------------|---------|------|---------------|
| 0 0 1 1 | 0x03 | Cyan | 1 0 1 1 | 0x0b | Light cyan |
| 0 1 0 0 | 0x04 | Red | 1 1 0 0 | 0x0c | Light red |
| 0 1 0 1 | 0x05 | Magenta | 1 1 0 1 | 0x0d | Light magenta |
| 0 1 1 0 | 0x05 | Brown | 1 1 1 0 | 0x0e | Yellow |
| 0 1 1 1 | 0x07 | Light grey | 1 1 1 1 | 0x0f | White |

4. EGA/VGA display standard

2.4.11 EGA(Enhanced Graphics Adapters)やVGA(Video Graphics Adapters)では、MDAやCGA対応に加えて、グラフィックスの他の表示拡張にも対応しています。MDAやCGA対応の表示モードでは、占有するメモリアドレスの開始位置や範囲は同じである。ただし、EGA/VGAでは最低でも32KBの表示メモリが標準装備されている。0xa0000から始まる物理メモリのアドレス空間がグラフィカルに占有されます。

2.4.12 Floppy disk and hard disk controller

PCのフロッピーディスク制御サブシステムは、フロッピーディスクとフロッピーディスクドライブで構成されています。フロッピーディスクはプログラムやデータを保存することができ、持ち運びも容易なため、フロッピーディスクドライブは古くからPCの標準的な構成機器の一つとなっている。ハードディスクもディスクとドライブで構成されているが、通常、ハードディスクの金属製ディスクはドライブに固定されており、取り外すことはできない。ハードディスクは記憶容量が大きく、読み書きの速度が非常に速いため、パソコンの中では最大の外部記憶装置であり、通常、外部ストレージとも呼ばれる。フロッピーディスクもハードディスクも、情報の保存には磁気媒体を使用しており、保存の動作も同様である。そこで、ここではハードディスクを例に挙げて、その仕組みを簡単に説明します。

ディスクにデータを保存する基本的な方法は、磁化した後にディスクの表面に磁気媒体の層を設けることである。フロッピーディスクではポリエスチルフィルムを、ハードディスクでは金属アルミニウム合金を基板として使用している。フロッピーディスクには、ポリエスチルフィルムのディスクが入っている。フロッピーディスクにはポリエスチルフィルム製のディスクが入っており、上段と下段のヘッドでディスクの両面にデータを読み書きする。ディスクの回転速度は約300rpmである。容量1.44MBのフロッピーディスクの場合、ディスクの両面は80トラックに分かれており、各トラックには18セクタのデータが格納できるので、 $2 \times 80 \times 18 = 2880$ セクタあることになる。表2-6は、いくつかの一般的なタイプのフロッピーディスクの基本パラメータを示しています。

| 表2-6 フロッピーディスク共通の基本パラメータ Disk type and capacity | tracks/face | Sectors/tracks | Total sectors | Rotate speed (r/min) | Data transmission rate (Kbps) |
|--|-------------|----------------|---------------|----------------------|-------------------------------|
| 5 $\frac{1}{4}$ inch 360KB | 40 | 9 | 720 | 300 | 250 |
| 3 $\frac{1}{2}$ inch 720KB | 80 | 9 | 1440 | 360 | 250 |
| 5 $\frac{1}{4}$ inch 1.2MB | 80 | 15 | 2400 | 360 | 500 |

| | | | | | |
|----------------|----|----|------|-----|------|
| 3½ inch 1.44MB | 80 | 18 | 2880 | 360 | 500 |
| 3½ inch 2.88MB | 80 | 36 | 5760 | 360 | 1000 |

ハードディスクは通常、少なくとも2枚以上のメタルディスクを含むため、2つ以上の読み書きヘッドを持つ。例えば、2枚のディスクを含むハードディスクには4つの物理ヘッドがあり、4枚のディスクを含むディスクには8つの読み取り/書き込みヘッドがあります。図2-11参照。ハードディスクの回転速度は通常4500rpm～10000rpmと高速なので、ハードディスクのデータ転送速度は通常数メガビット/秒まで可能である。

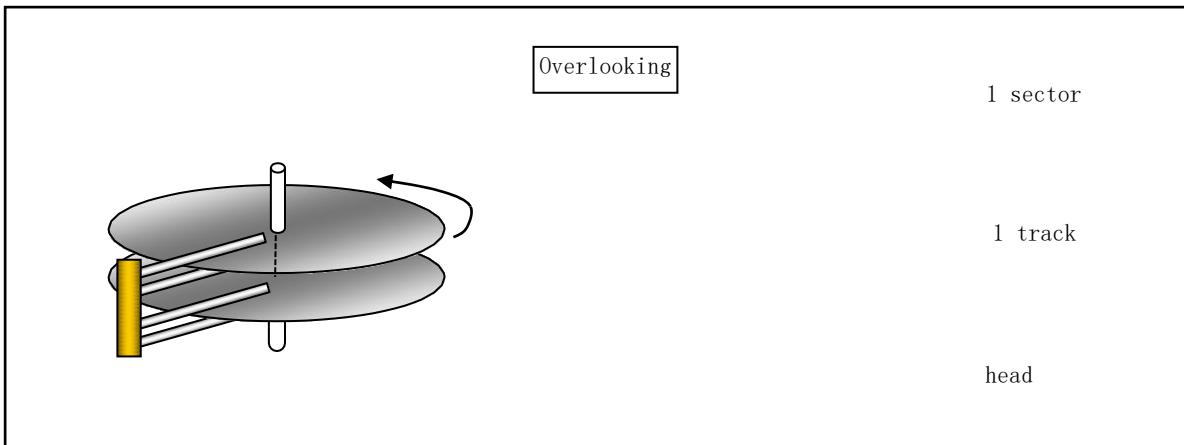


図2-11 2枚のディスクを持つ典型的なハードディスクの内部構造

ディスク面上の磁気ヘッドは、それぞれ読み出しコイルと書き込みコイルを持っている。データの読み取り動作では、まずヘッドが回転するディスク上のある位置に移動する。磁気ディスクが回転すると、磁気媒体は磁気ヘッドに対して一様な速度で移動するので、磁気ヘッドは実際に磁気媒体に磁力線を切る。その結果、誘導により読み取りコイルに電流が発生します。ディスク表面の残留状態の方向によって、コイルに誘導される電流の方向も異なるので、ディスクに記録されている0と1のデータが読み出され、ディスクからビットストリームを順次読み出すことができるようになります。ヘッドが読み取った各トラックは、情報を格納するための特定のフォーマットを持っているので、ディスク回路は、読み取ったビットストリームの中のフォーマットを認識することで、トラック上の各セクタのデータを判別して読み取ることができます。図2-12を参照してください。その中で、GAPは分離に使われるインターバルフィールドである。通常、GAPは0の12バイトである。各セクタのアドレスフィールドには、該当するセクタのシリンド番号、ヘッド番号（面番号）、セクタ番号が格納されているので、アドレスフィールドのアドレス情報を読み取ることで、セクタを一意に判別することができます。

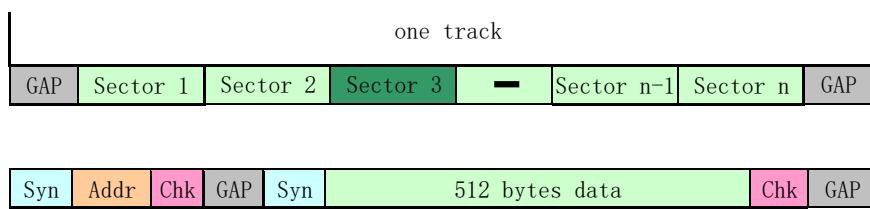


図2-12 ディスクのトラックフォーマット

ディスクにデータを読み書きするには、ディスクコントローラを使う必要がある。ディスクコントローラは、CPUとドライバの間の論理的なインターフェース回路である。CPUからの要求コマンドを受け取り、シーク、リード/ライト、制御信号をドライバーに送り、データのフローパターンを制御・変換する。コントローラとドライバの間で転送されるデータには、図2-12のセクタのアドレス情報とタイミング・クロック情報が含まれています。コントローラは、これらのアドレス情報や一部のエンコード、デコードなどの制御情報を実際の読み書きデータから分離する必要があります。また、ドライバとのデータ転送はシリアルビットストリームなので、コントローラはパラレルバイトデータとシ

リアルビットストリームデータを変換する必要があります。

PC/AT機のFDC (Floppy Disk Controller) は、NECの「 μ PD765」またはその互換チップを使用しています。これは

主に、図2-13に示すように、CPUから発行されたコマンドを受信し、コマンドの要求に応じて各種のハードウェア制御信号をドライバに出力するために使用されます。読み書き動作を行う際には、データ変換（ストリング・パラレル）、エンコード、ベリファイなどの動作を行い、ドライブの動作状態を常に監視する必要があります。

| | | |
|--------|------|----------------|
| Main | s | tatus Register |
| Comm | nd | Register |
| Parame | ters | Register |
| Data | | Register |
| Cont | rol | Register |

図2-13 ディスクコントローラの内部

ディスクコントローラのプログラミングは、I/Oポートを介してコントローラ内の関連レジスタの内容を設定し、レジスタを介して動作の結果情報を得るというものである。セクターデータの転送に関しては、フロッピーディスクコントローラはPC/ATのハードディスクコントローラとは異なる。フロッピーディスクコントローラ回路はDMA信号を使用するため、データ転送にはDMAコントローラを使用する必要があります。ATハードディスクコントローラでは、DMAコントローラを介さずに、高速データブロックを用いて転送します。フロッピーディスクは比較的ダメージ（カビやキズ）に弱いため、現在ではフロッピーディスクドライブはコンピューターに配備されていない。代わりに、大容量で持ち運びに便利なUSBフラッシュドライブが使われている。

2.5 Summary

ハードウェアは、オペレーティングシステムの基本的なプラットフォームです。オペレーティングシステムが動作するハードウェア環境を理解することは、その上で動作するオペレーティングシステムを深く理解するための必要条件です。本章では、従来のマイクロコンピュータのハードウェア構成とともに、マイクロコンピュータの各主要部を簡単に紹介します。次章では、Linuxカーネルが使用している2つのアセンブリ言語構文とそれに関連するコンパイラーをソフトウェアの観点から説明するとともに、カーネルが使用しているGNU gcc構文拡張の内容を紹介する。

3 Kernel programming language and environment

言語コンパイル処理とは、人間が理解できる高級言語を、コンピューターのハードウェアが理解して実行できるバイナリの機械語命令に変換する処理である。この変換プロセスでは、通常、効率の悪いコードが生成されるため、高い動作効率が求められるコードや性能への影響が大きいコードの一部は、通常、低レベルのアセンブリ言語で直接記述するか、高級言語のコンパイルで生成されたアセンブリを使用する。その後、手動で最適化のための修正を行います。本章では、Linuxで使用されているプログラミング言語、オブジェクトファイルフォーマット、コンパイル環境について説明します。

0.12 カーネルのソースコード 主な目的は、Linux 0.12カーネルのソースコードを読むために必要なアセンブリ言語とGNU C言語拡張の知識を提供することです。まず、as86とGNU asアセンブリの構文と使い方をより詳しく紹介します。次に、GNU C言語のインラインアセンブリ、ステートメント式、レジスタ変数、インライン関数など、カーネルソースコードにおけるC言語拡張を使用します。導入し、C言語とアセンブリ関数の相互呼び出しの仕組みについて詳しく説明しています。オブジェクトファイルのフォーマットを理解することは、アセンブリの動作を理解する上で最も重要な前提条件の一つであるため、2つのアセンブリ言語を紹介する際には、まずターゲットファイルの基本的なフォーマットを簡単に説明した上で、Linux

0.12については、この章の後半で詳しく説明します。システムで使用されているa.outオブジェクトファイルフォーマット。最後に、Makefileの使用方法について簡単に説明します。本章の内容は、Linuxカーネルのソースコードを読む際の参考情報です。そのため、本章の内容をざっと見てから次の章を読み、問題が発生したときに本章を参照することができます。

3.1 as86 Assembler

Linux 0.1x では、2 種類のアセンブリが使用されています。1 つは as86 アセンブリで、コンパニオンの ld86 リンカを使って 16 ビットのコードを生成します。もう 1 つは GNU アセンブリ gas(as)で、GNU ld リンカを使って結果のオブジェクトファイルをリンクします。ここではまず as86 アセンブリの使い方を説明し、アセンブリの使い方は次のセクションで説明します。

as86とld86は、MINIX-386の主要開発者の一人であるBruce Evans氏によって書かれたIntel 8086, 80386アセンブリ用のコンパイラとリンカです。LinusがLinuxカーネルの開発を始めたときに、すでにLinuxシステムに移植されています。80386プロセッサ用の32ビットコードをコンパイルすることができるが、Linuxシステムでは、16ビットのブートセクタプログラムboot/bootsect.sと、リアルモードの

初期設定プログラムboot/setup.sのバイナリ設定コードを作成するためにのみ使用されている。このコンパイラは、高速かつコンパクトで、マクロやより多くのエラー検出方法など、GNU gasにはない機能を備えています。しかし、このコンパイラの構文は、GNU asのアセンブリコンパイラの構文とは互換性がなく、MicrosoftのMASM、BorlandのTurbo ASM、NASMなどのアセンブラーの構文に近いものとなっています。これらのアセンブラーはすべてインテルのアセンブリ言語構文を使用しています（例：オペランドの順序がGNU asと逆など）。

as86の構文は、MINIXシステムのアセンブリ言語構文をベースにしており、MINIXシステムのアセンブリ構文は、PC/IXシステムのアセンブラー構文をベースにしています。PC/IXは、昔、インテル8086のCPUで動いていたUN*XのOSです。アンドリュー・S・タネンbaumは、PC/IXシステム上でMINIXシステムを開発した。

ブルース・エバンスは、32ビット版MINIXオペレーティングシステムの主要なリビジョンプログラマーの一人です。Linuxの創始者であるリナス・トーバルズの親しい友人でもある。Linuxカーネル開発の初期に、LinusはBruce EvansからUNIX系OSについて多くのことを学んだ。MINIX OSの不備も、仲良しの2人がお互いに議論した結果なのです。このようなMINIXの欠点は、リナスがインテル80386アーキテクチャ上で新しいコンセプトのOSを開発するきっかけとなった主な要因のひとつにすぎない。リナスはかつてこう言いました。"Bruce is my hero"という言葉があるように、Linux OSの誕生とBruce Evans氏との間にも密接な関係があると言えるだろう。

3.1.1 このコンパイラとリンクのソースコードは、FTP サーバ ftp.funet.fi または Web サイト www.oldlinux.org からダウンロードできます。最近の Linux システムでは、例えば dev86-0.16.3-8.i386.rpm のように、as86/ld86 を含む RPM パッケージを直接インストールすることができます。Linux システムでは as86 と ld86 は、前述の 2 つの 16 ビットアセンブラー bootsect.s と setup.s のコンパイルとリンクにのみ使用されるため、ここではこれら 2 つのプログラムで使用されるアセンブラーの構文とアセンブラコマンド（アセンブラ）についてのみ説明します。indicator) の役割と使い方を説明します。

3.1.2 as86 assembly language syntax

アセンブラは、低レベルのアセンブリ言語プログラムを、機械語を含むバイナリプログラムまたはオブジェクトファイルにコンパイルするように設計されています。アセンブラは、入力されたアセンブリ言語プログラム (srcfileなど) をオブジェクトファイル (objfile) にコンパイルします。アセンブラのコマンドラインの基本的な形式は次のとおりです。

`as [options] -o objfile srcfile`

オプションは、指定されたフォーマットと設定でターゲットファイルを作成するために、コンパイルプロセスを制御するために使用されます。入力されるアセンブリ言語プログラム srcfile は、テキストファイルです。ファイルの内容は、改行文字で終わる一連のテキスト行で構成されていなければなりません。GNU as はセミコロンを使用して 1 行に複数のステートメントを含めることができます、アセンブリ言語プログラムをプログラミングする際には、1 行に 1 つのステートメントしか含めないのが一般的です。

ステートメントには、スペース、タブ、改行のみを含む空行のほか、代入ステートメント（または定義ステートメント）、擬似演算子ステートメント、機械語命令ステートメントなどがあります。代入文は、記号や識別子に値を割り当てるために使用する。識別子の後に等号を付け、その後に式を続けて構成されており、例えば "BOOTSEG = 0x07C0" のように。疑似演算子文は、アセンブラが使用する指標で、通常はコードを生成しません。疑似オペコードと 0 個以上のオペランドで構成されます。各オペコードは、ドット文字 '.' で始まります。ドット文字 '.' 自体は、コンパイル時の位置カウンタを表す特別な記号です。その値は、ドット記号が現れる機械語命令の最初のバイトのアドレスです。

機械語命令文は、実行可能な機械語命令のニモニックであり、操作コードと0個以上のオペランドで構成されます。また、ステートメントの前には、ラベルを付けることができます。ラベルとは、識別子の後にコロン「:」を付けたものです。アセンブラーはコンパイル時にラベルを見つけると、そのラベルに現在の位置カウンタの値を割り当てます。したがって、アセンブリステートメントは通常、ラベル（オプション）、命令ニーモニック（命令名）、オペランドの3つのフィールドで構成されています。ラベルは、命令の第1フィールドにあります。ラベルは命令の最初のフィールドにあり、その場所のアドレスを表し、通常はジャンプ命令の目的地を示します。最後に、コメントで始まるコメント欄を追うこともできます。

アセンブラーのコンパイルによって生成されるオブジェクトファイル objfile には、通常、テキストセグメント (.text)、データセグメント (.data)、および未初期化データセグメント (.bss) の少なくとも3つのセグメントまたはセクションが含まれています。テキスト・セグメント（またはコード・セグメント）は初期化されたセグメントで、通常はプログラムの実行コードと読み取り専用のデータを含みます。データ・セグメントも初期化されたセグメントで、読み取り/書き込みデータを含みます。初期化されていないデータセグメントは

3.1.3 は初期化されていないセグメントです。通常、アセンブラーが生成する出力オブジェクトファイルでは、このセグメントのためのスペースは確保されませんが、オブジェクトファイルが実行プログラムにリンクされる際に、OSはセグメントの内容を0に初期化します。コンパイル時に、アセンブリ言語プログラムのコードやデータを生成するステートメントは、これら3つのセグメントのいずれかにコードやデータを生成します。コンパイルされたバイトは、「.text」セクションから順に格納されます。書き込まれたセグメントを変更するには、セグメント制御疑似演算子を使います。ターゲットファイルのフォーマットについては、後述の「Linux 0.12 オブジェクトファイルフォーマット」の項で詳しく説明します。

3.1.4 as86 assembly programs

以下では、簡単なフレームワークのサンプルプログラムboot.sを使って、as86アセンブラーの構造とプログラム内のステートメントの構文を説明し、コンパイルリンクと実行方法を示します。最後にas86とld86の使用方法とコンパイルオプションを使用します。そのサンプルプログラムを以下に示します。このサンプルはbootsect.sのフレームワークプログラムで、ブートセクタコードをコンパイルして生成するものです。特定のステートメントの使用方法を示すために、意図的に無意味な20行のステートメントを追加しています。

```

1 !
2 ! boot.s -- bootsect.s framework program. Replace 1 character in the string msg1
3 !           with code 0x07 and display it on the first line of the screen.
4 .globl begtext, begdata, begbss, endtext, enddata, endbss ! Global id used for ld86 links
5 .text          ! Text segment
6 begtext:
7 .data          ! Data segment
8 begdata:
9 .bss           ! Uninitialized data segment
10 begbss:
11 .text          ! Text segment
12 BOOTSEG = 0x07c0 ! Original segment address for the loaded bootsect code.
13
14 entry start   ! Inform the linker the program starts executing from here.
15 start:
16     jmpi    go,BOOTSEG ! Jump between segments. INITSEG indicates the jump
17 セクション・アドレス、ラベル・ゴーは、オフセット・アドレスです。
18 go:    mov    ax,cs      ! The value of the segment register cs -->ax is used
19        mov    ds,ax      ! to initialize the data segment registers ds and es.
20        mov    es,ax
21        mov    [msg1+17],ah ! 0x07-> Replaces 1 dot in the string and beep once.
22        mov    cx,#20      ! 20 chars displayed, including cr & lf.
23        mov    dx,#0x1004   ! String displayed on screen at line 17, column 5.
24        mov    bx,#0x000c   ! Character display attribute (red).
25        mov    bp,#msg1     ! Point to a string (required by interrupt call).
26        mov    ax,#0x1301   ! Write string and move cursor to the end of the string.
27        int    0x10         ! The BIOS interrupt call 0x10, function 0x13, subfunc 01.
28 loop1: jmp    loop1     ! Dead cycle.
29 msg1:  .ascii "Loading system ..."! Message to be displayed, total of 20 ASCII characters.
30 .byte 13,10

```

```
30 .org 510          ! Indicates statement is stored from address 510 (0x1FE).
31     .word 0xAA55    ! Active boot sector flag, used by the BIOS.
32 .text
33 endtext:
34 .data
35 enddata:
```

36 .bss37 endbss:

まず、プログラムの機能を紹介し、その後、各ステートメントの役割を詳しく説明します。このプログラムは、単純なブートセクタープログラムです。生成された実行プログラムをコンパイル、リンクすることで、コンピュータの起動に直接使用するフロッピーディスクの第1セクターに配置することができます。起動すると、画面の17行目と5列目に「Loading system ...」という赤い文字列が表示され、カーソルが1行下に移動します。その後、プログラムはコードライン27で無限にループします。

プログラムの最初の3行はコメント文です。as86のアセンブリ言語プログラムでは、感嘆符「！」またはセミコロン「;」で始まるステートメントの後にコメント文が続きます。コメント文は、任意の文の後に置くことも、新しい行から始めることもできます。

4行目の「.globl」は、アセンブラ指示子（またはアセンブラ指示子、疑似演算子）です。アセンブラ指示文は一文字「・」で始まり、コンパイル時にはコードを生成しません。アセンブラ指示文は、疑似オペコードとそれに続く0個以上のオペランドで構成されます。例えば、4行目の「globl」は疑似オペコードで、それに続くラベル「begtext, begdata, begbss」などはそのオペランドです。ラベルは識別子の後にコロンを付けたもので、例えば6行目の'begtext:'のようになります。ただし、ラベルを参照する際にはコロンを取る必要はありません。

通常、アセンブラは様々な疑似演算子をサポートしていますが、ここでは、Linuxシステムのbootsect.sとsetup.sのアセンブリ言語プログラムでよく使われるas86の疑似演算子についてのみ説明します。

.globl 疑似演算子は、後続のラベル識別子が外部またはグローバルであることを定義するために使用され、使用されない場合でも導入が必須となります。

5行目から11行目で定義された3つのラベルに加えて、「.text」、「.data」、「.bs」という3つの疑似演算子が定義されています。これらはそれぞれ、ターゲットファイルに3つのセグメント（テキストセグメント、データセグメント、未初期化データセグメント）を生成するアセンブラプログラムに対応しています。.text」はテキスト・セグメントの開始位置を特定してテキスト・セグメントに切り替え、「.data」はデータ・セグメントの開始位置を特定して現在のセグメントをデータ・セグメントに切り替え、「.bs」は初期化されていないデータ・セグメントの開始位置を特定して現在のセグメントをbbsセグメントに切り替えています。つまり5--11行目は、各セグメントにラベルを定義し、テキストセグメントに切り替えて次のコードを書き始めるためのものです。ここでは、3つのセグメントがすべて同じ重複したアドレス範囲に定義されているので、サンプルプログラムは実際にはセグメント化されていません。

12行目では、代入文「BOOTSEG = 0x07c0」が定義されています。等号「=」（または記号「EQU」）を使って、識別子「BOOTSEG」が表す値を定義しているので、この識別子を記号定数と

呼ぶことができる。この値は、C言語の文言と同様に、10進数、8進数、16進数で使用できます。

14行目の識別子'entry'は、リンク用ld86が生成する実行ファイルに、それ以降に指定されたラベル'start'を強制的に含めるための予約キーです。通常、複数のオブジェクトファイルをリンクして実行ファイルを生成する際には、デバッグのためにキーワードentryでアセンブラーのエントリーラベルを指定する必要があります。しかし、今回の例やLinuxカーネルのboot/bootsect.s、boot/setup.sアセンブラーでは、生成されるピュアバイナリの実行ファイルにシンボル情報を含めたくないでの、このキーワードを省略することができます。

16行目にはセグメント間のファージャンプステートメントがあり、次の命令にジャンプします。BIOSが0x7c00の物理メモリにプログラムをロードしてそこにジャンプしたとき、CSを含むすべてのセグメントレジスタのデフォルト値は0、つまりCS:IP=0x0000:0x7c00です。そのため、ここではセグメント値0x7c0をCSに割り当てるために、セグメント間ジャンプステートメントを使用します。このステートメントが実行された後、CS:IP = 0x07C0:0x0005となります。次の2つのステートメントは、0x7c0セグメントを指すように、DSとESセグメントレジスタにそれぞれ値を割り当てます。これにより、プログラム内のデータ（文字列）のアドレス指定が容易になります。

20行目のMOV命令は、ahレジスタの0x7c0セグメント値の上位バイト（0x07）を、メモリ文字列msg1の最後の「・」の位置に格納するために使用します。この文字は、文字列が表示されたときに、BIOSの割り込みでビープ音を発生させます。この文は、主に間接オペランドの使い方を説明するために使われています。as86では、間接オペランドには角括弧のペアが必要です。他のアドレッシング方式では、次のようなものがあります。

! レジスタの直接アドレス指定。bxで指定されたアドレスにジャンプ、つまりbxをIPにコピーします。

```

    mov    bx, ax
    jmp    bx
! Indirect register addressing. The bx specifies the memory location as the address of the jump.
    mov    [bx], ax
    jmp    [bx]
! Put the immediate number 1234 into ax. Put the msg1 address value in ax.
    mov    ax, #1234
    mov    ax, #msg1
! Absolute addressing. Put the contents of the memory address 1234 (msg1) into ax.
    mov    ax, 1234
    mov    ax, msg1
    mov    ax, [msg1]
! Index addressing. Put the value at the memory location indicated by the second operand into ax.
    mov    ax, msg1[bx]
    mov    ax, msg1[bx*4+si]
```

21～25行目のステートメントは、即値データを適切なレジスタに入れるために使用されます。この#の前には即値の数字を置かなければならず、そうしないとメモリアドレスとして使用され、絶対アドレス指定のステートメントになってしまいます。上記の例を参照してください。また、ラベル(msg1など)のアドレス値をレジスタに入れるときは、前に「#」をつけないと、msg1のアドレスのレジスタになってしまいます!

26行目は、BIOSの画面表示割り込みコールint 0x10です。ここでは、その関数19、サブ関数1が使用されています。この割り込みの目的は、指定された位置のスクリーンに文字列（msg1）を書き込むことです。レジスタcxは文字列の長さの値、dxは表示位置の値、bxは表示使用文字属性、es:bpは文字列を指しています。

27行目は、現在の命令にジャンプするジャンプ文です。つまり、これは無限ループ文なのです。ここで無限ループ文を使うのは、表示された内容が削除されずに画面に残るようにするためです。デッドループ文は、アセンブラープログラムをデバッグするときによく使われます。

28-29行目では、文字列msg1を定義しています。文字列を定義するには、疑似演算子「.ascii」を使い、文字列を二重引用符で囲む必要があります。また、疑似演算子「.asciiz」は、文字列の後に自動的にNULL(0)文字を追加します。さらに、29行目では、キャリッジリターンとラインフィード（13, 10）の文字を定義しています。文字を定義するには、疑似演算子「.byte」を使用し、文字を一重引用符で囲む必要があります。例えば、以下のようになります。'"D"'.もちろん、例のように文字のASCII

コードを直接書くこともできます。

30行目の疑似オペレータ文「.org」は、現在のアセンブラーの位置を定義します。このステートメントは、アセンブラーのコンパイル時に、現在のセグメントの位置カウンタの値を疑似オペレータステートメントで指定された値に調整します。この例のプログラムでは、このステートメントが位置カウンタを510に設定し、有効なブートセクタフラグワード0xAA55をここに配置します（31行目）。擬似演算子「.word」は、現在の位置にダブルバイトのメモリオブジェクト（変数）を定義するために使用され、その後に数値や式を続けることができます。コードもデータもないで、ここからboot.sでコンパイルされた実行ファイルはちょうど512バイトになるはずだと判断できます。

32行目--37行目は、3つのセグメントのそれぞれに、さらに3つのラベルを配置しています。3つのセグメントの終了位置を示すために使用します。この設定は、次のような場合に、各モジュールの各セグメントの開始と終了を区別するために使用できます。

- 3.1.5 複数のターゲットモジュールをリンクする カーネル内の bootsec.s と setup.s はそれぞれ別個にコンパイル、リンクされているため、純粋なバイナリファイルの生成を期待しても、他のオブジェクトモジュールファイルとはリンクしません。そのため、サンプルプログラムでは、各セグメントの疑似プログラムを宣言しています。文字 (.text、.data、.bss) はすべて省略可能です。つまり、プログラムの4行目～11行目、32行目～37行目をすべて削除しても、リンクをコンパイルすれば正しい結果が得られます。

3.1.6 as86 assembly language program compilation and link

次に、リンクサンプルプログラム boot.s をコンパイルして、起動に必要なブートセクタプログラムを生成する方法を紹介します。上記のサンプルプログラムのコンパイルとリンクには、以下の最初の2つのコマンドが必要です。

```
[/root]# as86 -O -a -o boot.o boot.s           // Compile. Generate the target file.
[/root]# ld86 -O -s -o boot boot.o             // link. Remove symbol information.
[/root]# ls -l boot*
-rwx--x--x  1 root      root      544 May 17 00:44 boot
-rw-----  1 root      root      249 May 17 00:43 boot.o
-rw-----  1 root      root     767 May 16 23:27 boot.s
[/root]# dd bs=32 if=boot of=/dev/fd0 skip=1    // Write to a floppy disk or Image file.
16+0 records in
```

その中で、1つ目のコマンドはas86アセンブラーを使ってboot.sプログラムをコンパイルし、boot.oオブジェクトファイルを生成します。2番目のコマンドはリンカーld86を使ってターゲットファイルにリンク操作を行い、最後にMINIX構造の実行ファイルbootを生成します。オプションの'-O'は8086の16ビットターゲットプログラムを生成するために、'-a'はGNU asやldのパーティと互換性のあるコードを生成することを指定するために使われます。's' オプションは、最後に生成された実行ファイルからシンボル情報を削除するよう linkerに指示するために使用する。'-o'は、生成される実行ファイルの名前を指定します。

- lsコマンドを使って上に挙げたファイル名を見ればわかるように、最後に生成されたブートプログラムは、先に述べたように正確に512バイトではなく、32バイトの長さです。この32バイトは、MINIX実行ファイルのヘッダの構造です（構造の詳細な説明は「カーネルコンポーネントの作成」の章を参照）。このプログラムを使ってマシンを起動するためには、この32バイトを手動で削除する必要があります。このヘッダー構造を削除するにはいくつかの方法があります。

- Use the binary editor to delete the first 32 bytes of the boot program and save it;
- Using the as86 compile linker on current Linux systems (eg RedHat 9), which have the option of generating a pure binary executable without the MINIX header structure, please refer to the online user manual (man as86) of the relevant system.
- Use the Linux system's dd command.

上記の3番目のコマンドは、ddコマンドを使ってブートの最初の32バイトを削除し、その出力を直接フロッピーディスクのイメージファイルまたはBochsシミュレーションシステムに書き込みます。(Bochs PCアノログシステムを使用してください。 前章を参照してください)。このプログラムをBochsシミュ

レーション・システムで実行すると、図3-1のような画面が得られます。



図3-1 Bochsシミュレーションシステムでのブートプログラムの実行

3.1.7 as86 and ld86 usage methods and options

as86 と ld86 の使用方法とオプションは以下の通りです。

asの使い方とオプション

as [-O3agjuw] [-b [bin]] [-lm [list]] [-n name] [-o objfile] [-s sym] srcfile

デフォルト設定（以下のデフォルト以外のオプションのデフォルトはoffまたはnoneで、aフラグを指定しない場合は出力されません）。

-3 Use the 80386 32-bit output;
 list Display on standard output;
 name The basic name of the source file (that is, does not include the extension after '.');

それぞれの選択肢の意味

- O Use 16-bit code segments;
- 3 Use 32-bit code segments;
- a Open some compatibility options with GNU as, ld;
- b Generate binary files, followed by the file name;
- g Only global symbols are stored in the object file;
- j Make all jump statements long jumps;
- l Generate a list file, followed by the list file name;
- m Extend the macro definition in the list;
- n Followed by the module name (in place of the source file name into the target file);
- o Produce the target file, followed by the target file name (objfile);
- s Produce a symbol file followed by a symbol file name;
- u The undefined symbol as the symbol of the input unspecified segment;
- w No warning message is displayed;

ld リンカの使用構文とオプション。

Minixのa.outフォーマットを生成するバージョンについて。

```
ld [-03Mims[-]]... [-T textaddr] [-llib_extension] [-o outfile] infile...
```

GNU-Minixのa.outフォーマットを生成するバージョンです。

```
ld [-03Mimrs[-]]... [-T textaddr] [-llib_extension] [-o outfile] infile...
```

デフォルトの設定（以下のデフォルトを除き、他のオプションはデフォルトではオフまたはなし）。

```
-03      32-bit output;  
outfile  a.out format output;
```

```
-0  Generate a header structure with 16-bit magic numbers, use i86 subdirectory for -lx option;  
-3  Generate a header structure with a 32-bit magic number, use i386 subdirectory for -lx option;  
-M  Display linked symbols on standard output devices  
-T  Followed by the text base address (using the format suitable for strtoul);  
-i  Separate instruction and data segment (I&D) output;  
-lx  Add the library /local/lib subdir/libx.a to the linked file list;  
-m  Display linked modules on standard output devices  
-o  Specify the output file name;  
-r  Generate output suitable for further relocations;  
-s  Remove all symbols in the target file.
```

3.2 GNU as assembler

前節で紹介したas86アセンブラーは、カーネル内のboot/bootsect.sのブート・セクタ・プログラムと、リアル・モードのboot/setup.sのセットアップ・プログラムのコンパイルにのみ使用されます。カーネル内の他のすべてのアセンブリ言語プログラム（C言語で生成されたものも含む）は、ガスでコンパイルされ、C言語プログラムで生成されたモジュールとリンクされます。このセクションでは、80X86 CPUハードウェア・プラットフォームをベースにしたLinuxカーネルにおけるアセンブラー構文とGNU as assembler（以下、asアセンブラー）の使用方法について説明します。まず、asのアセンブリ言語プログラムの構文を紹介し、次に一般的なアセンブリ指令（インジケータ）の意味と使い方を説明します。次の章の終わりには、詳細な命令を含むアセンブラー言語プログラムの例を示します。

オペレーティングシステムの主要なコード要件には、高い実行速度と効率性が求められるものが多いため、通常、オペレーティングシステムのソースコードには、主要なアセンブリ言語プログラムの約10%が含まれています。Linuxも例外ではありません。32ビットの初期化コード、すべての割り込みや例外処理のインターフェースプログラム、多くのマクロ定義など、すべてがアセンブリ言語プログラムまたは拡張エンベデッドアセンブリ文として使用されています。これらのアセンブリ言語プログラムの機能を理解できるかどうかが、OSの具体的な実装を理解する上でのポイントになることは間違ひありません。

GNU gcc コンパイラは、C プログラムをコンパイルする際に、まず中間結果として as アセンブリ

言語ファイルを出力し、次に gcc は as アセンブラーを呼び出して、一時的なアセンブリ言語プログラムをターゲットファイルにコンパイルします。つまり、asアセンブラーは本来、単体のアセンブラーとして使われるよりも、gccが生成した中間的なアセンブリ言語プログラムをアセンブルするために設計されたものなのです。そのため、asアセンブラーは、文字や数字、定数の表現方法や表現形式など、C言語の多くの機能もサポートしています。

GNU as アセンブラーは、もともと BSD 4.2 のアセンブラーを踏襲して開発されました。現在の as アセンブラーは、さまざまな形式のオブジェクトファイルを生成するように設定できます。しかし、コンパイルされたアセンブリは

3.2.1 言語プログラムは、使用または生成されるターゲットファイルの形式には関係ありませんが、以下の説明でターゲットファイルの形式が関係する場合は、Linux 0.12系で採用されているa.outターゲットファイルの形式について説明します。.

3.2.2 Compiling as assembly language program

as assemblerを使ってアセンブラープログラムをコンパイルする際の基本的なコマンドラインのフォーマットは以下のとおりです。

`as [オプション] [-o objfile] [srcfile.s ...]`

objfileはasのコンパイル出力のターゲットファイル名、srcfile.sはasの入力アセンブリ言語プログラム名です。出力ファイル名を使用しない場合、asはデフォルトの出力先ファイルであるa.outをコンパイルします。asプログラム名の後には、コマンドラインにコンパイルオプションやファイル名を入れることができます。すべてのオプションは自由に配置できますが、ファイル名のコンパイル結果は密接に関係しています。

プログラムのソースコードは、1つまたは複数のファイルに配置することができます。プログラムのソースコードがどのように複数のファイルに分割されても、プログラムのセマンティクスは変わりません。プログラムのソースコードは、これらすべてのファイルを順番に並べた結果をまとめたものです。asコンパイラを実行するたびに、1つのソースプログラムだけをコンパイルします。しかし、ソースプログラムは複数のテキストファイルで構成されることがあります（端末の標準入力もファイルの一つです）。

asのコマンドラインでは、0個以上の入力ファイル名を指定することができます。asは、これらの入力ファイルの内容を左から右へと読んでいきます。コマンドラインの任意の位置にあるパラメータが特定の意味を持たない場合、それらは入力ファイル名として扱われます。コマンドラインでファイル名が指定されていない場合、asはターミナルまたはコンソールの標準入力から入力ファイルの内容を読み取ろうとします。この場合、入力する内容がない場合は、手動で Ctrl-D キーの組み合わせを入力して as アセンブラーに伝える必要があります。コマンドラインで入力ファイルとして標準入力を明示的に指定したい場合は、「--」というパラメータを使用する必要があります。

asの出力ファイルは、入力されたアセンブリ言語プログラムによってコンパイルされたバイナリデータファイル、すなわちターゲットファイルとなります。オプション「-o」で出力ファイルの名前を指定しない限り、asはa.outという名前の出力ファイルを作成します。ターゲットファイルは、主にリンクldの入力ファイルとして使用されます。オブジェクトファイルには、コンパイルされたプログラムコード、実行可能なプログラムを生成する際にldを支援する情報、そして場合によってはデバッグシンボル情報が含まれています。Linux 0.12システムで使用されているa.outオブジェクトファイルフォーマットについては、本章で後述します。

boot/head.sアセンブラーを個別にコンパイルしたい場合は、コマンドラインで次のコマンドを入力します。

```
[/usr/src/linux/boot]# as -o head.o head.s
[/usr/src/linux/boot]# ls -l head*.
-rw-rwxr-x 1 root      root          26449 May 19 22:04 head.o
-rw-rwxr-x 1 root      root          5938 Nov 18 1991 head.s
[/usr/src/linux/boot]#
```

3.2.3 as assembly syntax

- gcc 出力アセンブラーとの互換性を保つために、AT&T System V アセンブラー構文（以下、AT&T 構文）を採用しています。この構文は、インテルのアセンブラーが使用している構文（略してインテル構文）とは大きく異なり、いくつかの大きな違いがあります。

- In the AT&T syntax, an immediate value is preceded by a character '\$'; the register operand name is preceded by the character percent sign '%'; absolute jump/invoke (relative to the program counter's jump/invoke) operands To add asterisk '*'. Intel assembly syntax does not have these limitations.

- The order of the source and destination operands used by AT&T syntax and Intel syntax is exactly the opposite. AT&T's source and destination operands are "source, destination" from left to right. For example Intel's statement 'add eax, 4' corresponds to AT&T's 'addl \$4, %eax'.
- The length (width) of the memory operand in AT&T syntax is determined by the last character of the opcode. The operand suffixes 'b', 'w', and 'l' indicate that the memory reference width is 8 bits byte, 16-bit words, and 32-bit long words, respectively. Intel syntax achieves the same purpose by using the prefixes 'byte ptr', 'word ptr', and 'dword ptr' before the memory operands. Therefore, Intel's statement 'mov al, byte ptr foo' corresponds to AT&T's statement 'movb \$foo, %al'.
- In the AT&T syntax, immediate and far-form calls in the immediate form are 'ljmp/lcall \$section, \$offset」であるのに対し、インテルのそれは「jmp/call far section:offset」である。同様に、AT&Tの構文では、farのリターン命令「ret \$stack-adjust」は、インテルの「ret far stack-adjust」に対応する。
- The AT&T assembler does not provide support for multi-segment programs because UNIX-like operating systems require that all code be in one segment.

3.2.2.1 Assembler preprocessing

アセンブラーは、アセンブリ言語プログラムの簡単な前処理機能を内蔵しています。この前処理機能では、余分なスペースやタブを調整・削除したり、コメント文を削除してスペースや改行文字に置き換えたり、文字定数を対応する値に変換したりします。ただし、マクロの定義やインクルードファイルの機能は処理しません。この機能が必要な場合は、アセンブリ言語プログラムに大文字の接尾辞「.S」を付けて、ascにgcc CPPの前処理機能を使わせることができます。

3.2.2.2 As

Asのアセンブリ言語プログラムでは、C言語のコメント文（「/*」と「*/」）を使用しているため、1行のコメント開始文字としてハッシュ記号「#」も使用しており、アセンブリ前にプログラムを前処理しない場合は、プログラムに含まれるハッシュ記号「#」で始まるインジケータやコマンドはすべてコメントの一部として扱われます。

3.2.2.3 Symbols, Statements, and Constants

シンボルは、文字で構成された識別子で、シンボルを構成する有効な文字は、大文字、小文字、数字、そして「_.」の3つの文字から取られます。シンボルは、数字で始まることはできず、大文字と小文字も異なります。as assembler では、シンボルの長さに制限はなく、シンボル内のすべての文字が有効です。シンボルは他の文字（スペース、改行など）やファイルの先頭を使用して開始点と終了点を定義します。

ステートメントの最後は、改行または改行文字（';」）で終わります。ファイルの最後のステートメントは、改行文字で終わらなければなりません。

行末にバックスラッシュ（改行の前）を使用すると、1つの文を複数行に分けて書くことができます。asがバックスラッシュと改行を読んだ場合、その2つの文字は無視されます。

ステートメントは、0個以上のラベルで始まり、その後にステートメントのタイプを決定するキーシンボルが続きます。ラベルは、シンボルの後にコロン(':')を付けたものです。キーシンボルは、ステートメントの残りの部分のセマンティクスを決定します。キーシンボルが「...」で始まる場合は、現在のステートメントがアセンブリコマンド（またはディレクティブ、インジケータ）であることを示します。

キーシンボルが文字で始まる場合、現在のステートメントはアセンブリ言語の命令ステートメントです。つまり、ステートメントの一般的な形式は

```
label: .directive    followed by optional some comments
another_label:        # This is an empty statement.
                     instruction    operand_1, operand_2, ...
```

定数とは、数字のことで、文字定数と数値定数に分けられます。文字

また、定数は文字列と一文字に分けられ、数値定数は整数、大数、浮動小数点数に分けられます。

文字列は二重引用符で囲む必要があり、バックスラッシュ「¥」を使って特殊文字をエスケープすることができます。例えば、「¥」はバックスラッシュ文字を表します。最初のバックスラッシュはエスケープインジケーターで、2番目の文字が通常のバックスラッシュ文字として扱われることを示しています。一般的なエスケープシーケンスを表3-1に示します。バックスラッシュの後に別の文字が続くと、バックスラッシュは機能せず、アセンブラーは警告メッセージを表示します。

アセンブラーで1文字の定数を使用する場合、その文字の前に1つの引用符を書くことができます。例えば、"A"は値65を示し、"C"は値67を示します。表 3-1 のエスケープコードは、1 文字の定数にも使用できます。例えば、「¥」はバックスラッシュ文字の定数を表します。

| 表3-1 アセンブラーがサポートするエスケープされた文字列 Escape code | Description |
|--|---|
| \b | Backspace, value is 0x08 |
| \f | Formfeed, value 0x0C |
| \n | Newline, value 0x0A |
| \r | Carriage-Return value is 0x0D |
| \NNN | Character code represented by 3 octal numbers |
| \xNN... | Hexadecimal number character code |
| \\ | Represents a backslash character |
| \" | Represents a double quote in a string "" |

整数の数値定数は、「0b」または「0B」で始まる2進数（「0-1」）、「0」で始まる8進数（「0-7」）、0以外の桁（「0-9」）で始まる10進数、「0x」または「0X」で始まる16進数（「0-9a-fA-F」）の4通りで表されます。負の数を表すには、負の'-'を前につければよい。

Bignum は 32 ビット以上のビット数で、その方法は整数と同じです。アセンブラーにおける浮動小数点定数の表現は、基本的にC言語と同じです。カーネルコードでは浮動小数点数はほとんど使われないので、ここでは説明しません。

3.2.4 Instruction statements, operands, and addressing

■ 命令とは、CPUが行う操作のことです。通常、命令はオペコードとも呼ばれます。オペランドとは、命令操作の対象となるものです。アドレスとは、指定されたデータのメモリ上の位置のことです。命令文とは、プログラムの実行時に実行される文である。通常、4つの要素で構成されます。

- Label (optional);
- Opcode (instruction mnemonic);
- Operands (specified by specific instructions);
- Comments

命令文には、コンマで区切られた0個または3個までのオペランドを含めることができます。2つのオペランドを持つ命令文では、第1オペランドがソースオペランド、第2オペランドがデスティネーションオペランド、つまり命令操作の結果が第2オペランドに格納されます。

- オペランドは、即値（つまり値が定数の式）、レジスタ（CPUのレジスタ内の値）、メモリ（メモリ内の値）のいずれかです。間接オペランド（Indirect operand）は、実際のオペランド値のアドレス値を含む。AT&Tの構文では、オペランドの前に「*」の文字を付けて間接オペランドを指定します。間接オペランドは、リダイレクト／コール命令でのみ使用できます。後述のジャンプ命令の説明を参照してください。

- A '\$' character prefix is required before immediate operands;
- A '%' character prefix needs to be preceded by a register name;
- The memory operand is specified by a variable name or a register containing the address of the variable. The variable name implicitly indicates the address of the variable and instructs the CPU to reference the contents of the memory at that address.

3.2.3.1 Name the instruction opcode

AT&T構文の命令オペコード名（命令ニーモニック）の最後の文字は、オペランドの幅を示すために使用されます。文字'b'、'w'、'l'は、それぞれバイト、ワード、ロングのオペランドを指定します。命令名にこのような文字の接尾辞がなく、命令文にメモリ・オペランドが含まれていない場合、asはデスティネーション・レジスタ・オペランドに基づいてオペランドの幅を決定しようとします。例えば、「mov %ax, %bx」という命令文は、「movw %ax, %bx」と同じです。同様に、「mov \$1, %bx」という命令文は、「movw \$1, %bx」と同じです。

AT&Tとインテルの構文では、ほとんどすべての命令オペコードの名前が同じですが、それでもいくつかの例外があります。シンボリック・エクステンションとゼロ・エクステンションの両命令では、ソース・オペランドとデスティネーション・オペランドに幅を指定する必要があることを示すために、2つの幅を必要とします。AT&Tの構文は、2つのオペコードサフィックスを使用して行われます。AT&T構文のシンボル拡張とゼロ拡張の基本的なオペコード名は、それぞれ「movs...」と「movz...」で、インテルではそれぞれ「movsx」と「movzx」です。オペコードの基本名には、2つのサフィックスが付きます。例えば、シンボリック拡張を使用して%alから%edxに移動するAT&Tのステートメントは'movsbl %al, %edx'であり、blはバイトからロングへ、bwはバイトからワードへ、wlはワードからロングへとなります。AT&Tの構文とインテルの構文の変換命令の対応を表3-2に示します。

| | Intel | Description |
|--|-------|-------------|
| | | |

| | | |
|-----------------|------|-------------------------------------|
| | | |
| AT&T | | |
| cbtw | cbw | Extend the byte value in %al to %ax |
| cwtl | cwde | Extend the %ax sign to %eax |
| cwtd | cwd | Extend the %ax sign to %dx:%ax |
| cltd | cdq | Extend the %eax sign to %edx:%eax |

3.2.3.2 Instruction opcode prefix

オペコードのプレフィックスは、後続のオペコードを変更するために使用されます。文字列命令の繰り返し、エリアオーバーライドの提供、バスロック操作の実行、オペランドやアドレス幅の指定などに使われます。通常、オペコードプレフィックスは、オペランドを持たない命令の排他的な行として使用することができ、影響を受ける命令の直前に配置しなければなりませんが、修正する命令と同じ行に配置するのがベストです。例えば、文字列スキャンコマンド「scas」は、繰り返し演算を行うためにプレフィックスを使用しています。

レプネースカス %es:(%edi), %al

オペランドの接頭辞の一部を表3-3に示す。

| 表3-3 オペコードプレフィックスリスト Opcode prefix | Description |
|---------------------------------------|---|
| cs, ds, ss, es, fs, gs | Section overrides the opcode prefix. Using the section:memory operands by specifying memory prefixes automatically adds this prefix. |
| data16, addr16 | Operand/address width prefix. These two prefixes will change the 32-bit operand/address to a 16-bit operand/address. However, please note that as does not support 16-bit addressing. |
| lock | Bus latching prefix. Used to disable interrupts during instruction execution (only valid for some instructions, see the 80X86 manual). |
| wait | Coprocessor instruction prefix. Wait for the coprocessor to complete the execution of the current instruction. This prefix is not needed for the 80386/80387 combination. |
| rep, repe, repne | The prefix of the string instruction causes the string instruction to repeat the specified number of times in %ecx. |

3.2.3.3 Memory reference

インテル構文の間接メモリー参照形式：section:[base + index*scale + disp] 次の AT&T 構文に対応しています：section:disp(base, index, scale)

baseとindexはオプションの32ビットベースレジスタとインデックスレジスタ、dispはオプションのオフセット値です。Scaleは、スケールファクターで、その範囲は1、2、4、8です。Scaleにindexを乗じてオペランドアドレスを算出します。Scaleが指定されない場合、スケールのデフォルトは1です。Sectionは、メモリオペランドのオプションのセグメントレジスタを指定し、オペランドで使用されている現在のデフォルトのセグメントレジスタをオーバーライドします。指定されたセクションオーバーライトレジスタがデフォルトのオペレーションセクションレジスタと同じ場合、asはアセンブルされた命令に同じセクションプレフィックスを出力しないことに注意してください。以下は、いくつかのAT&Tおよびインテルの構文形式でのメモリ参照の例です。

| | |
|---------------------------------|---|
| movl var, %eax | # Put the contents at memory address var in the register %eax. |
| movl %cs:var, %eax | # Put the contents at var in the code segment into %eax. |
| movb \$0x0a, %es:(%ebx) | # Save byte value 0x0a to offset specified by %ebx in es segment. |
| movl \$var, %eax | # Put the address of var in %eax. |
| movl array(%esi), %eax | # Put contents at address determined by array+%esi into %eax. |
| movl (%ebx, %esi, 4), %eax | # Put contents at address determined by %ebx+%esi*4 in %eax. |
| movl array(%ebx, %esi, 4), %eax | # Put contents at address of array + %ebx+%esi*4 into %eax. |
| movl -4(%ebp), %eax | # Put contents at %ebp -4 in %eax, using the default segment %ss. |
| movl foo(%eax, 4), %eax | # Put contents at foo+eax*4 intp %eax, using default seg %ds. |

3.2.3.4 Jump instruction

ジャンプ命令は、実行ポイントをプログラム内の別の場所に移動し、実行を継続するために使用されます。このジャンプ先は、通常、ラベルで表されます。オブジェクトコードファイルを生成する際、アセンブラーはタグ付けされたすべての命令のアドレスを決定し、ジャンプ命令のアドレスをジャンプ

命令にエンコードします。ジャンプ命令は、無条件ジャンプと条件付きジャンプに分けられます。条件付きジャンプ命令は、命令実行時にフラグレジスタ内の関連するフラグの状態に依存してジャンプするかどうかを決定し、無条件ジャンプはこれらのフラグに依存しない。

JMPは無条件のジャンプ命令で、直接ジャンプと間接ジャンプの2種類に分けられますが、条件付きジャンプ命令は直接ジャンプの形式しかありません。直接ジャンプ命令では、ジャンプ先の命令のアドレスがジャンプ命令の一部として直接エンコードされ、間接ジャンプ命令では、ジャンプ先がレジスタやMemoryのロケーションから取得される。直接ジャンプ命令は、ジャンプ先のラベルを与えるように記述し、間接ジャンプ命令は、スター文字「*」を演算指示子の先頭文字として記述し、演算指示子はmovl命令と同じ構文を使用します。以下に、直接ジャンプと間接ジャンプの例を示します。

| | |
|-------------|--|
| jmp NewLoc | # Jump directly. Unconditionally jump to label NewLoc to continue execution. |
| jmp *%eax | # Indirect jump. The value of register %eax is the jump destination. |
| jmp *(%eax) | # Indirect jump. Read the jump destination from the address indicated by %eax. |

- 3.2.5 同様に、命令カウンタPCに依存しない間接呼出オペランドにも、プレフィックス文字として「*」を付ける必要があります。この文字を使用しない場合、asアセンブラーは命令カウンタPCに関連するジャンプラベルを選択します。また、メモリオペランドを持つその他の命令では、オペコードのサフィックス ('b', 'w', 'l') を使用して、オペランドのサイズ（バイト、ワード、ロング）を示す必要があります。

3.2.6 Sections and Relocation

セクション（セグメントともいう）は、アドレス範囲を表すのに使われ、OSはそのアドレス範囲のデータ情報を同じように扱い、処理します。例えば、「読み取り専用」の領域があり、この領域からはデータを読み取ることしかできず、書き込むことはできません。ゾーンという概念は主に、コンパイラが生成するターゲットファイル（または実行プログラム）の中のテキスト領域やデータ領域など、異なる情報領域を示すために使われる。アセンブリ言語プログラムを正しく理解してコンパイルするためには、asが生成する出力オブジェクトファイルのフォーマットを理解する必要があります。Linux 0.12カーネルで使用されているa.out形式のオブジェクトファイルのフォーマットについては、本章の後半で詳しく説明します。ここでは、アセンブラーが生成するオブジェクトファイルの基本的な構造を理解するために、ゾーンの基本的な概念について簡単に紹介します。

リンクldは、入力されたオブジェクトファイルの内容を一定のルールに基づいて結合し、実行プログラムを生成します。asアセンブラーがターゲットファイルを出力する際、ターゲットファイル内のコードはデフォルトで0番地から始まるように設定されています。その後、ldはリンク処理の際に、異なるターゲットファイルの各部分に異なる最終アドレス位置を割り当てます。Ldは、プログラムのバイトブロックを、プログラムが実行されたアドレスに移動します。これらのブロックは固定された単位として移動されます。その長さやバイト順は変更されない。このような固定単位をゾーン（またはセグメント、パート）と呼ぶ。ゾーンにランタイムのアドレスを割り当てる操作を再配置操作といい、ターゲットファイルに記録されているアドレスが適切なランタイムのアドレスに対応するように調整することも含まれる。

as-assemblerは、text、data、bssエリアと呼ばれる少なくとも3つのフィールドを持つオブジェクト

トファイルを作成し、出力します。各地区は空でもよい。アセンブラー命令で '.text' または '.data' 地区に出力を置かなかった場合、これらの地区は存在しますが、内容は空になります。ターゲットファイルでは、0番地からテキストエリア、データエリア、bssエリアの順に表示されます。

- セクションが再配置されると、リンカーldはどのデータがどのように変更されるのかを知るために、アセンブラーは必要な再配置情報をターゲットファイルに書き込みます。再配置の操作を行うためには、ld はターゲットファイルのアドレスが関係するたびに知る必要があります。

- Where did the reference to an address in the target file come from?
- What is the length of the quoted byte?
- Which section is referenced by this address? What is the value of (address)-(start address of section)?
- Is the reference to the address related to the program counter PC (Program-Counter)?

実際、asで使われるアドレスはすべて次のように表すことができます。(セクション) + (セクション内のオフセット)。また、asで評価される式のほとんどは、このようなゾーンに関連した特性を持っています。以下の説明では、ゾーンのsecname内のオフセットNを示すために、"`{secname N}`"という表記を使用する。

テキストエリア、データエリア、bssエリアに加えて、絶対アドレスエリア（アブソリュートエリア）についても理解しておく必要があります。リンクがさまざまなオブジェクトファイルを結合するとき、絶対領域のアドレスは常に同じになります。例えば、ldは実行時に`{absolute 0}`というアドレスを0番地に「再配置」します。リンクがリンク後に2つのターゲットファイルのデータ領域を重複したアドレスとして配置することはありませんが、ターゲットファイルの絶対領域は必ず重複して上書きされます。

また、Undefinedセクションがあります。アセンブリでは、このエリアの任意のアドレスが`{undefined U}`に設定されていると判断することはできません（Uは後で埋められます）。値は常に定義されているので、未定義のアドレスを表示するには、未定義のシンボルを使うしかありません。コモンブロックへの参照は、このようなシンボルです。その値はアセンブリ時には不明なので、未定義領域に入ります。

3.2.4.1

同様に、セクション名もリンクされたプログラムのセクション群を表すのに使われます。リンクldは、プログラムのすべてのオブジェクトファイルのテキストセクションを、隣接するアドレスに配置します。私たちが慣れ親しんでいるプログラムのテキスト・エリアとは、実際には、そのプログラムのすべてのオブジェクト・ファイルのテキスト・セクションの組み合わせによって形成されるアドレス・エリア全体を指しています。プログラムのデータ部やbss部の理解も同様です。

3.2.4.2 Linker involved sections

■ リンカーldは、以下の4種類のセクションのみを対象としています。

- Text section, data section -- These two areas are used to save programs. As and ld treat them independently and equally. The description of the text section is also suitable for the data section. However, when the program is running, the usual text section will not change. The text section is usually shared by the process and contains the instruction code and constants. The contents of the data section usually change when the program is running. For example, C variables are usually stored in the data section.
- bss section -- This area contains 0 bytes when the program starts running. This area is used to store uninitialized variables or as a common variable storage space. Although the length information of the bss section of each target file of the program is very important, since the area stores zero-value bytes, there is no need to save the bss section in the target file. The purpose of setting the bss area is to explicitly exclude zero-value bytes from the target file.
- Absolute section -- The address 0 of this area is always "relocated" to the address 0 of the runtime. Use this section if you do not want ld to change the address you are referencing when relocating. From this point of view, we can refer to absolute addresses as "non-relocatable": they do not change during relocation operations.
- undefined section -- A reference to an object that is not in each of the previously mentioned sections belongs to this section.

理想的な3つのリロケータブル・セクションの例を図3-2に示します。この例では、従来のセクション名である「.text」と「.data」を使用しています。横軸はメモリアドレスを示す。ldリンカーの具体的な動作については、後ほど詳しく説明します。



図3-2 2つのオブジェクトファイルをリンクしてリンク先のプログラムを生成する例

3.2.4.3 Subsection

アセンブルされるバイトデータは、通常、テキスト部またはデータ部に配置されます。アセンブルのソースプログラムのある領域に、隣接していないデータ群が存在することがあります。アセンブル後にそれらをまとめて保存したい場合があります。Asアセンブラーでは、サブセクションを使用してこのような目的に使用できます。各セクションには、0～8192の番号が付いたサブエリアがあります。同じサブセクションでプログラムされたオブジェクトは、そのサブセクションの他のオブジェクトと一緒にターゲットファイルにまとめられます。例えば、コンパイラはテキストエリアに定数を格納したいが、これらの定数がアセンブルされるプログラム全体に散らばっていては困る場合があります。この場合、コンパイラは、出力される各コード領域の前に「.text 0」サブセクションを使用し、各定数セットの前に「.text 1」サブセクションを使用することができます。

サブセクションの使用は任意です。サブセクションを使用しない場合、すべてのオブジェクトはサブセクション0に配置されます。デスティネーションファイルにはサブセクションの番号順に表示されますが、デスティネーションファイルにはサブセクションを表す情報は含まれません。宛先ファイルを処理するldなどのプログラムは、サブセクションの痕跡を見ることはできず、すべてのテキストサブセクションからなるテキストセクションと、すべてのデータサブセクションからなるデータセクションを見るだけです。後続のステートメントがどのサブセクション領域に組み立てられるかを指定するために、「.text expression」または「.data expression」に数値パラメータを使用することができます。式の結果は、絶対値でなければなりません。.text」のみを指定した場合は、デフォルトで「.text 0」が使用されます。同様に、「.data」を指定すると、「.data 0」が使用されます。

3.2.4.4

各セクションには、そのセクションにアセンブルされた各バイトをカウントするロケーションカウンタがあります。サブセクションはアセンブラーが使いやすいように設定されているだけなので、サブセクションカウンタはありません。位置カウンタを直接操作する方法はありませんが、アセンブリコマンド「.align」でその値を変更することができ、任意のラベル定義は位置カウンタの現在の値を取ります。ステートメントアセンブリ処理を実行しているゾーンの位置カウンタをカレントアクティビティカウンタと呼ぶ。

3.2.4.5 bss section

3.2.7 bssセクションは、ローカルのパブリック変数を格納するために使用されます。bssセクションにスペースを確保することはできますが、プログラムの実行前にデータを入れることはできません。なぜなら、プログラムの実行を開始すると、bssセクションのすべてのバイトがクリアされるからです。アセンブリコマンドの「.lcomm」はbssセクションのシンボルを定義するのに使われ、「.comm」はbssセクションのパブリックシンボルを宣言するのに使われる。

3.2.8 symbol

プログラムのコンパイルやリンクの過程において、シンボルは重要な概念です。プログラマーはシンボルを使ってオブジェクトに名前を付け、リンカーはシンボルを使ってリンク操作を行い、デバッガーはシンボルを使ってデバッグを行います。

ラベルは、シンボルの後にコロンを付けたものです。この時点では、シンボルはアクティブの現在の値を表しています。

位置カウンタで、例えば、命令のオペランドとして使用することができます。等号「=」を使って、シンボルに任意の値を割り当てることができます。

3.2.5.1 シンボル名は、アルファベットまたは「._」文字のいずれかで始まります。ローカルシンボルは、コンパイラやプログラマーが名前を一時的に使用するために使用されます。ローカルシンボルの名前は10個 ('0'...9) あり、プログラムの中で再利用することができます。ローカルシンボルを定義するには、「N:」(Nは任意の数字を表す) という形式のラベルを書くだけです。前に定義されたシンボルを参照する場合は'Nb'と書き、次に定義されたローカルラベルを参照する場合は'Nf'と書く必要があります。ここで'b'は後方、'f'は前方を意味します。ローカルラベルの使用に制限はありませんが、いつでも、前方／後方の最も遠い10個のローカルラベルしか参照できません。

3.2.5.2 Special point symbol

3.2.5.3 特殊記号「.」は、アセンブリの現在のアドレスを示しています。つまり、「mylab: .long ...」という表現は、mylabが自分のアドレス値を含むように定義します。.に値を代入することは、アセンブリコマンド「.org」と同じです。つまり、「.=.+4」という表現は、「.space 4」と全く同じである。

3.2.5.4 Symbol attributes

各シンボルは、名前に加えて、"value"と"type"の属性を持っています。出力のフォーマットによっては、シンボルが補助的な属性を持つこともあります。シンボルが定義されずに使用された場合、asはそのシンボルのすべての属性が0であるとみなします。

シンボルの値は通常32ビットです。テキスト、データ、bss、アブソリュートの各エリアの位置を示すシンボルの場合、エリアの先頭からラベルまでのアドレス値が値となります。テキストエリア、データエリア、bssエリアでは、通常、リンク処理中にエリアのベースアドレスの変更によりシンボルの値が変化しますが、アブソリュートエリアのシンボルの値は変化しません。このため、絶対記号と呼ばれている。

ldは、未定義のシンボルの値を扱います。未定義のシンボルの値が0の場合、そのシンボルがアセンブリのソースプログラムで定義されていないことを意味します。ldは他のリンクされたファイルからその値を決定しようとします。シンボルは、プログラムがシンボルを使用しているが、シンボルを定義していない場合に生成されます。未定義のシンボルの値が0でない場合、シンボルの値は、.commパブリック宣言で必要とされるパブリックメモリ空間の長さを表します。シンボルは、このメモリ空間の最初のアドレスを指します。

3.2.9 シンボルのtype属性には、リンクやデバッガのための再配置情報、シンボルが外部にあることを示すフラグ、その他のオプション情報が含まれています。a.out形式のオブジェクトファイルでは、シンボルのtype属性は8ビットのフィールド (n_typeバイト) に格納されます。その意味については、include/a.out.hファイルの説明を参照してください。

3.2.10 as assembler directives

3.2.6.1 アセンブリ指令とは、アセンブリの動作方法を示す永続的な命令です。アセンブリ指令は、変数の領域確保、プログラムの開始アドレスの決定、現在のアセンブリセクションの指定、位置カウンタの値の変更などをアセンブリに要求するために使用します。アセンブリ指令はすべて「.」で始まり、それ以外は文字で、大文字小文字は関係ありません。大文字小文字は関係ありませんが、一般的には小文字を使用します。以下では、一般的なアセンブリ命令について説明します。

3.2.6.2 .align abs Expr1, abs Expr2, abs Expr3

.align は、現在のサブセクションの次の指定されたメモリ境界に位置カウンタの値を設定（インクリメント）するストレージアラインアセンブラ指令です。最初の絶対値式 abs-expr1（絶対値式）で、必要な境界アライメント値を指定します。a.out形式のオブジェクトファイルを使用する80X86システムでは、この式の値は、インクリメントされた後の位置カウンタの右端のバイナリ値のゼロ値のビット数、つまり2の累乗になります。例えば、「.align 3」は、位置カウンタの値を8の倍数にすることを意味します。位置カウンタの値自体が8の倍数である場合には、必要ありません。

を使って変更することができます。しかし、ELFフォーマットを使用する80X86システムでは、式の値がそのままそのために必要なバイト数になります。例えば、「.align 8」は位置カウンタの値を8の倍数にすることです。

3.26.3 2つ目の式では、アライメントとパディングに使用するバイト値を指定します。この式とその前のコンマは省略可能です。省略された場合、パディングのバイト値は0になります。3番目のオプションの式（abs Expr3）は、アライメント操作によってパディングがスキップされることを許容する最大バイト数を示すために使用されます。アライメント操作によってスキップされたバイト数がこの最大値よりも大きい場合、アライメント操作はキャンセルされます。第2パラメータを省略したい場合は、第1パラメータと第3パラメータの間にカンマを2つ入れます。

3.26.4 .ascii "string"...

3.26.5 位置カウンタの現在の位置から文字列用のスペースを割り当て、文字列を格納する。複数の文字列をカンマで区切って書くことができます。例えば、「.ascii "Hello world!", "My assembler"」のようになります。アセンブラー命令では、これらの文字列を連続したアドレス位置に組み立て、各文字列の後に0（NULL）バイトを追加しないようにします。

3.26.6 .asciz "string"...

3.26.7 このアセンブラー指令は、「.ascii」と同じですが、各文字列の後にゼロ値のバイトが続きます。.asciz'の"z"は"ゼロ"を意味します。

3.26.8 .byte expressions

3.26.9 このディレクティブは、カンマで区切られた0個以上の式を想定しています。各式は次のバイトに結合されます。

3.26.10 .comm symbol, length

3.26.11 bssセクションに名前付きのパブリックエリアを宣言します。ldのリンク時に、あるオブジェクトファイルの共通シンボルが、他のオブジェクトファイルの同名の共通シンボルにマージされます。ldがシンボル定義を見つけられず、1つ以上の共通シンボルだけを見つけた場合、ldは長さlengthバイトの初期化されていないメモリを割り当てます。長さは絶対値表現でなければなりません。ldが、長さが同じで名前が異なる複数の共通シンボルを見つけた場合、ldは長さが最も大きいスペースを割り当てます。

3.26.12 .data subsection

3.26.13 このアセンブラー指令は、次のステートメントをデータサブセクションの番号の付いたサブセクションにアセンブルするように指示します。番号を省略した場合、デフォルトでは番号0が使用されます。数値は絶対値の表現でなければなりません。

3.26.14 .desc symbol, abs-expr

3.26.15 このディレクティブは、シンボルの記述子を、絶対式の下位16ビットに設定します。a.outまたはb.outオブジェクト形式のみです。include/a.out.hファイルの説明を参照してください。

3.26.16 .fill repeat, size, value

3.26.17 このアセンブラー指令は、サイズがNバイトのリピート（繰り返し）を生成します。sizeには0または何らかの値を指定しますが、sizeが8より大きい場合は8に制限されます。各リピートバイトのコネットは、8バイトの数値から取ります。最上位の4バイトは0、最下位の4バイトは数値となります。3つのパラメータ値は絶対値で、sizeとvalueはオプションです。2つ目のコンマとvalueが省略された場合、valueの値はデフォルトで0になります。2つ目のパラメータが省略された場合、sizeの値はデフォルトで1になります。

3.26.18 .global symbol (.globl symbol)

このアセンブラー命令により、リンクer ldにシンボルが表示されます。シンボルがオブジェクトファイルで定義されている場合、その値はリンクプロセスで他のオブジェクトファイルで使用されます。シンボルがオブジェクトファイルで定義されていない場合、その属性はリンクプロセスにおいて他のオブジェクトファイルの同名のシンボルから取得されます。これは、シンボルのシンボルタイプフィールドの外部ビットN_EXTを設定することで行われます。include/a.out.hファイルの説明を参照してください。

| | |
|--------------|--|
| 26.20 | 3.26.19 .int expressions アセンブラーの指示により、ある領域に0個以上の整数値を設定します（80386系は4バイト、.longと同じ）。コンマで区切られた各式の値は、ランタイムの値です。例えば、.int 1234,567,0x89ABのようになります。 |
| 26.22 | 3.26.21 .lcomm symbol, length シンボルに指定されたローカルコモンエリアは、長さバイトの空間を確保します。確保された領域とシンボルの値は、新しいローカルコモンブロックの値となる。割り当てられたアドレスはbssセクションにあるため、これらのバイト値は実行時にクリアされます。シンボルはグローバル宣言されていないので、リンカーのldは見えません。 |
| 26.23 | 3.26.23 .long expressions |
| 26.24 | その意味は、.intと同じです。 |
| 26.26 | 3.26.25 .octa bignums このアセンブリ指示子は、コンマで区切られた16バイトのラージナンバー (.byte、.word、.long、.quad、.octaはそれぞれ1、2、4、8、16バイトに対応) を0個以上指定します。 |
| 26.27 | 3.26.27 .org new_lc, fill このアセンブリ指令は、現在のセクションのロケーションカウンタに new_lc という値を設定します。new_lc は絶対値（式）、またはサブセクションと同じセクションを持つ式、つまりセクションをまたぐために.orgを使用することはできません。new_lcのセクションが正しくない場合、.orgは機能しません。ポジションカウンターはセクションベースであることに注意してください。つまり、各セクションがカウントの出発点として使われます。 |
| 26.28 | 位置カウンタの値が増加すると、スキップされたバイトは値のフィルで埋められます。この値は絶対値でなければなりません。カンマとfillを省略した場合、fillのデフォルトは0です。 |
| 26.29 | 3.26.29 .quad bignums |
| 26.30 | コンマで区切られた8バイトのラージナンバーを0個以上指定するアセンブリ指令です。大数が8バイトに収まらない場合は、下位8バイトを取ります。 |
| 26.32 | 3.26.31 .short expressions (same as .word expressions) コンマで区切られた2バイトの数値を0個以上、セクション内で指定するアセンブリ指令です。各式には、実行時に16ビットの値が生成されます。 |
| 26.34 | 3.26.33 .space size, fill アセンブリ指令では、サイズバイトが生成され、それぞれのバイトにfillが入ります。このパラメータは絶対値です。カンマとfillが省略された場合、fillのデフォルト値は0です。 |
| 26.36 | 3.26.35 .string "string" カンマで区切られた1つまたは複数の文字列を定義します。文字列にはエスケープ文字を使用できます。各文字列には、自動的にヌル終端文字が付加されます。例えば、.string "¥¥Starting", "other strings"などです。 |
| 26.38 | 3.26.37 .text subsection notification asは、以下の記述を番号付きのサブセクションにまとめます。番号subsectionが省略された場合は、デフォルトの番号値0が使用されます。 |
| | 3.26.39 .word expressions 32ビットマシンでは、このアセンブリ命令は.shortと同じ意味を持ちます。 |

3.2.11 Writing 16-bit code

GNU asは通常、純粋な32ビットの80X86コードを記述するために使用されますが、1995年以降のリアルモードや16ビットプロテクトモードで動作するコードを記述するためのサポートも限られています。asのコンパイルで16ビットコードを生成するためには、16ビットモードで動作する命令文の前にアセ

ンブリ命令「.code16」を追加し、asのアセンブラーを32ビットコードのアセンブリモードに戻すためにアセンブリ命令「.code32」を使用する必要があります。

asは、16ビットと32ビットのアセンブラー文を区別しません。16ビットモードと32ビットモードの各命令は、モードに関係なく全く同じように機能します。asは、16ビットモードと32ビットモードのどちらで実行されるかにかかわらず、アセンブラー文に対して常に32ビットの命令コードを生成します。アセンブリ命令'.code16'を使用してasを16ビットモードにすると、asはすべての命令に必要なオペランド幅のプレフィックスを自動的に追加し、16ビットモードで実行させます。なお、asはすべての命令にアドレスとオペランド幅のプレフィックスを追加するため、結果としてコード長やアセンブリのパフォーマンスに影響を与えます。

1991年のLinuxカーネル0.12の開発時には、アセンブラーが16ビットに対応していなかったため、前述のas86アセンブラーを使用して、ブート起動コードや初期化アセンブラーを記述・アセンブルしていました。

3.2.12 0.12カーネルリアルモード。

3.2.13 as assembler command line options

- a Turn on program listings.
- f Fast operation, skip whitespace and comment preprocessing.
- o objfile 出力されるオブジェクトファイルの名前をobjfileとして指定します。
- R Fold the data section into the text section.
- W Suppress warning messages.

3.3 C language program

3.3.1 GNU gccは、ISO標準C89に記載されているC言語をいくつか拡張しており、そのうちのいくつかはISO C99標準に含まれています。このセクションでは、カーネルで頻繁に使用されるgccの拡張機能のいくつかについて説明します。遭遇する拡張文の簡単な説明は、次のプログラムコメントのセクションでも隨時行われます。

3.3.2 C program compiling and linking

gccアセンブラーを使用してCプログラムをコンパイルする場合、通常は図3-3のように、前処理ステージ、コンパイルステージ、アセンブリステージ、リンクステージの4段階の処理を行います。

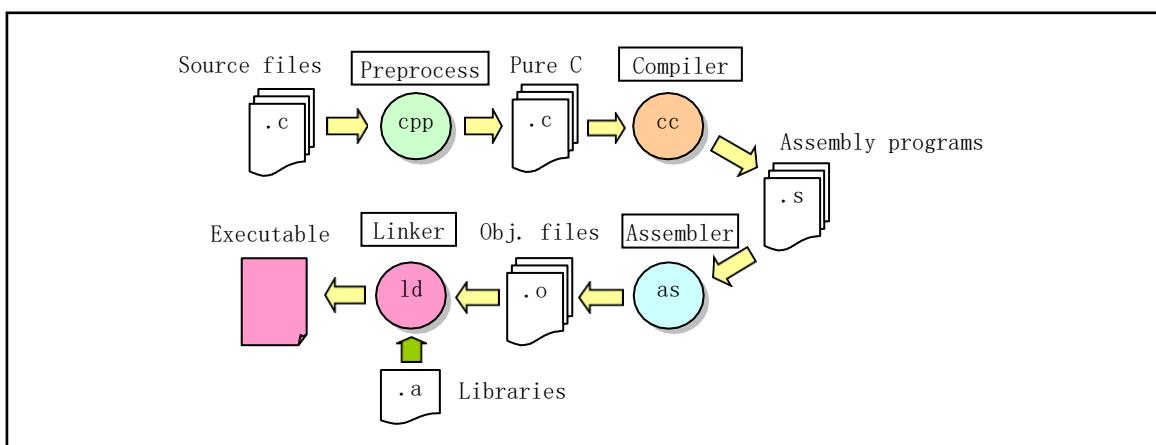


図3-3 CCプログラムのコンパイルプロセス

前処理段階では、gccはC言語プログラムをCプリプロセッサCPPに渡し、C言語プログラム中のインジケータやマクロを置き換えて、プレーンなC言語コードを出力する。コンパイル段階では、gccはC言語プログラムをコンパイルして、対応するMachine関連のアセンブリ言語コードを生成する。アセンブリ段階。

最後に、GNU ld リンカが、プログラムに関連するターゲットファイルの組み合わせをリンクして、プログラムの実行イメージファイルが生成される。gccを呼び出すコマンドラインのフォーマットは、アセンブリ言語をコンパイルするときのフォーマットと似ています。

```
gcc [オプション] [-o outfile] infile ...
```

infileは入力C言語ファイル、outfileはコンパイル後の出力ファイルです。コンパイル処理では、4つの処理段階すべてを実行する必要はありません。コマンドラインオプションを使用することで、gccのコンパイル処理を特定の処理段階の後で実行を停止することができます。例えば、「-S」オプションを使用すると、C言語プログラムに対するアセンブリ言語プログラムを出力した後にgccを停止させることができます、「-c」オプションを使用すると、以下のようにリンク処理を行わずにターゲットファイルの生成のみを行うことができます。

```
gcc -o hello hello.c      // Compile the hello.c to generate the execution file hello.  
gcc -S -o hello.s hello.c // Compile hello.c to generate corresponding assembly hello.s.  
gcc -c -o hello.o hello.c // Compile hello.c to generate target file hello.o without linking.
```

3.3.3 多数のソースプログラムファイルを含むLinuxカーネルのような大規模なプログラムをコンパイルする際には、通常、プログラム全体のコンパイルプロセスを自動的に管理するためにmakeツールを使用します。以下の説明をご覧ください。

3.3.4 Inline assembly language

ここでは、カーネルのC言語プログラムで公開されているインライン・アセンブリ文について説明します。通常、C言語のプログラムを作成する際にインラインアセンブリコードを使用することはほとんどありませんので、ここでは基本的な書式や使い方を説明する必要があります。asmを使ったアセンブリ命令では、命令のオペランドをC言語の式で指定することができます。つまり、使いたいデータが入っているレジスタやメモリの位置を推測する必要はなく、機械の説明書に記載されているようなアセンブリ命令のテンプレートを指定し、各オペランドに対してオペランド制約文字列を指定する必要があるのです。

```
asm("アセンブリ言語文")  
    :出力レジスタのオペランド  
    :入力レジスタのオペランド  
    :clobbered or modified)の登録を行います。
```

1行目を除いて、後ろにコロンが付いている行は、使用しない場合は省略可能です。このうち、「asm」はインラインアセンブリ文のキーワード、「assembly statement」はアセンブリ命令を記述する場所、「output register」はこの組み込みアセンブリを実行した後の出力データを格納するためのレジスタを示す。ここでは、これらのレジスタは、それぞれCの式の値や、メモリのアドレスに対応しています。"入力レジスタ"は、アセンブリコードの開始時に、ここで指定されたレジスタのいくつかに格納されるべき入力値を示します。また、それぞれCの変数や定数値にも対応しています。"Clobbered or Modified registers"は、ここに記載されているレジスタの値が変更され、gccコンパイラがこれらのレジスタに最初にロードした値に依存できなくなつたことを意味します。gccは必要に応じてこれらのレジスタを再ロードする必要があります。したがって、出力/入力レジスタのセクションに記載されてい

ないが、アセンブリステートメントで明示的に使用されている、または暗黙的に使用されているレジスタ名をリストアップする必要があります。

例えば、ここでは架空の「コンバイン」命令を紹介します。

```
asm("combine %1,%0":=r"(result) :"r"(length)).
```

ここでlengthは入力オペランドのC式、resultは出力オペランドのC式です。それぞれのオペランド制約には "r" があり、動的に割り当てられるジェネラルレジスタが必要であることを示しています。また、「=r」の「=」はオペランドが出力であることを示し、すべての出力オペランドの制約には「=」を使用しなければなりません。この制約は、gccのマニュアルに記載されているマシン記述で使用されているのと同じ言語を使用しています（第16章マシン記述のオペランド制約のセクション）。

上の例のように、各オペランドはオペランド制約文字列で記述され、その後にC式が括弧内に記述されます。アセンブラーのテンプレートと最初の出力オペランドはコロンで区切られ、最後の出力オペランドと最初の入力がある場合はコロンで区切られます。カンマは各グループ内のオペランドを区切れます。オペランドの総数は10個まで、または機械の説明書に記載されている命令パターンの最大オペランド数のどちらか多い方に制限されます。出力オペランドがなく、入力オペランドがある場合は、出力オペランドがある場所を囲むように2つの連続したコロンを置く必要があります。

以下では、より詳細な例を用いてインラインアセンブリステートメントの使用方法を説明します。これは、kernel/traps.cファイルの22行目から始まるコードブロックです。よりわかりやすくするために、このコードを並べ替えて番号を付けています。

```
01 #define get_seg_byte(seg,addr) \(^o^)
02 ({ \
03   register char_ res; \
04   __asm__ ("push %%fs; \           // A register variable res is defined. \
05             mov %%ax,%%fs; \        // First save original value of fs (seg selector). \
06             movb %%fs:%2,%%al; \    // Then use seg to set fs. \
07             pop %%fs" \           // Take 1 byte of seg:addr into the al register. \
08             :="a" ( res) \        // Restore the original contents of fs register. \
09             :"0" (seg), "m" (*(addr))); \ // Output register lists and constraints. \
10   res;}) // Input register lists and constraints.
```

この10行のコードは、インラインアセンブラーのマクロ機能を定義するものです。アセンブラー文を使用するには、マクロの中に入れるのが最も便利です。括弧で囲まれた複合文（中括弧の文）"({})" は式として使用でき、最終行の変数res(10行目)が式の出力値となります。次項で説明します。

マクロ文は1行で定義する必要があるため、ここではバックスラッシュを使って連結しています。このマクロ定義は、マクロ名が参照されるプログラムに代入されます。1行目では、マクロ関数名get_seg_byte(seg,addr)として、マクロの名前を定義しています。3行目では、レジスタ変数_resを定義しています。この変数は、素早くアクセスして操作できるように、レジスタに保存されます。レジスタ(eaxなど)を指定したい場合は、「register char res asm ("ax");」のように記述しますが、「asm」は「sm」とも書くことができます。4行目の「asm」は、埋め込みアセンブリ文の先頭を示しています。4行目から7行目までの4つの文は、AT&T形式のアセンブラー文です。また、gccが生成するアセンブリ

言語プログラムで、レジスタ名の前にパーセント記号「%」を付けるためには、アセンブリ文のレジスタ名を埋め込む前にパーセント記号「%%」を2つ書かなければなりません。

8行目は出力レジスタです。この文の意味は、このコード実行の終了後、eaxで表されるレジスタの値を、この関数の出力値としてres変数に置くことです。"=a"の"a"はロードコードを呼び出し、"="はこれが出力レジスタであることを示し、そこにある値を出力の

の値を表します。ロードコードとは、CPUのレジスタやメモリアドレス、一部の数値などを表す略式文字コードです。表3-4は、私たちがよく使うレジスタのロードコードとその具体的な意味を示したものです。9行目は、このコードの実行開始時にeaxレジスタにsegが置かれるることを表し、"0"は上記と同じレジスタが出力されることを表します。(*addr)は、メモリオフセットのアドレス値を表します。このアドレス値を上記のアセンブラー文で使用するために、組み込みアセンブラプログラムでは、出力レジスタと入力レジスタに、それぞれ「%0」で始まる出力レジスタ列の左と右の上から順に番号を付けていくように指定しており、%0、%1、...%9と表記しています。したがって、出力レジスタの番号は%0（ここでは出力レジスタは1つだけ）、入力レジスタの最初の部分 ("0"（セグ））は番号%1、後半の部分は番号%2となります。上記6行目の%2は、このメモリオフセットを(*addr)表しています。

| 表3-4 コードの説明 Code | Description | Code | Description |
|---------------------|--|------|--|
| a | Use register eax | m | Use memory address, any memory operand is allowed. |
| b | Use register ebx | o | Use memory address, and can add offset value |
| c | Use register ecx | I | Use constants 0-31 |
| d | Use register edx | J | Use constants 0-63 |
| S | Use register esi | K | Use constants 0-255 |
| D | Use register edi | L | Use constants 0-65535 |
| q | Use dynamically allocated byte addressable registers (eax、ebx、ecx 或 edx) | M | Use constants 0-3 |
| r | Use any dynamically allocated register | N | Use 1 byte constant (0-255) |
| g | Any general register, memory or immediate integer operand is allowed (eax、ebx、ecx、edx or memory variable) | O | Use constants 0-31 |
| A | Combine eax with edx (64-bit) | = | Output operands. The output value will replace the previous value |
| + | Indicates that the operand is readable and writable | & | Early-clobber operands. Indicates that the content will be modified before the operands are used |

では、4~7行目のコードの機能を見てみましょう。第1文では、fsセグメントレジスタの内容をスタックに置き、第2文では、eaxのセグメント値をfsセグメントレジスタに割り当て、第3文では、fs:(*)addrで指定されたバイトをalレジスタに入れてています。. アセンブリ文を実行すると、出力レジ

スタeaxの値がマクロ関数(ブロック構造式)の戻り値としてresに入ります。簡単でしょう？

以上の分析から、マクロ名のsegは指定されたメモリセグメントの値を表し、addrはメモリオフセットのアドレス量を表すことがわかります。今まででは、このプログラムの機能について明確にしておく必要がありました このマクロ関数の機能は、指定されたセグメントとオフセット値のメモリアドレスから1バイトをフェッチすることです。では、次の例を見てください。

```
01  asm("cld\n\t"
02      "rep\n\t"
03      "stos"
04      : /* No output register */
```

```
05      : "c"(count-1), "a"(fill_value), "D"(dest),
06      : "%ecx", "%edi");
```

1～3行目は、通常のアセンブラー文で、方向ビットをクリアして、値を繰り返し格納します。1～2行目にある「`¥¥`」という文字は、gccのプリプロセッサの出力プログラムリストをきれいに設定するためのものです。この文字の意味は、C言語と同じです。つまり、C言語のプログラムに対応するアセンブラーを生成し、そのアセンブラーを呼び出してコンパイルし、ターゲットコードを生成するというのがgccの動作モードです。プログラムを書いたり、デバッグしたりするときに、C言語に対応したアセンブラーを見たい場合は、プログラムを処理するアセンブラプログラムの前処理の出力を得る必要があります（これは、効率的なコードを書いたり、デバッグしたりするときによく使われます）。アセンブラーの出力をきれいな形式で前処理するために、2つの「`„^w^„`」という形式記号を使うことができます。

4行目は、インラインアセンブラーが出力レジスタを使用していないことを示しています。5行目の意味は `count-1` の値を `ecx` レジスタにロードし（ロードコードは `"c"`）、`fill_value` を `eax` にロードし、`dest` を `edy` に入っています。なぜ、このようなレジスタ値のロードを自分でやらずに、gccコンパイラにやらせなければならないのでしょうか？それは、gccが登録中にいくつかの最適化作業を行うことができるからです。たとえば、`fill_value` の値がすでに `eax` に入っている場合があります。それがループ文の中にある場合、gccはループ処理全体で `eax` を保持する可能性があるので、各ループの中で `movl` 文を使うことができます。

最後の行は、これらのレジスターの値が変更されたことをgccに伝えるためのものです。これらのレジスターで何をしているのかをgccが知った後は、gccの最適化に役立てることができます。次の例では、どの変数がどのレジスタを使うかを指定することはできず、gccに選択させます。

```
01  asm("leal (%1, %1, 4), %0"
02      : "=r"(y)
03      : "0"(x));
```

実効アドレスの計算には「`leal`」という命令を使用しますが、ここでは簡単な計算に使用します。最初のアセンブラー文「`leal (r1, r2, 4), r3`」は、 $r1+r2*4 \Rightarrow r3$ を示しています。この例では、`x` を 5 倍することができます。その中で、「`%0`」と「`%1`」は、gccが自動的に割り当てるレジスタを意味します。ここでは、"`%1`" が入力値 `x` を入れるレジスタを、"`%0`" が出力値のレジスタを表しています。レジスタのコードを出力する前に、必ず等号を付けてください。入力レジスタのコードが 0 または空の場合は、対応する出力と同じレジスタが使用されます。つまり、gccが `r` を `eax` と指定した場合、上記のアセンブル文の意味は次のようになります。

`"leal (eax, eax, 4), eax"`

なお、コードを実行する際に、GCCの最適化によってアセンブリ文が変更されないようにするには、以下のように `asm` 記号の後にキーワード `volatile` を追加する必要があります。この2つの宣言の違いは、プログラムの互換性の面にあります。後者の宣言方法を使用することをお勧めします。

```
asm volatile (.....);  
あるいは、より詳細な説明は  
asm volatile (.....);
```

また、キーワード volatile を関数名の前に置くことで、その関数を装飾して gcc に知らせることができます。

コンパイラに、その関数が返されないことを伝えます。これにより、gccはより良いコードを生成することができます。さらに、返さない関数については、このキーワードを使って、gccが誤った警告メッセージを生成するのを防ぐこともできます。たとえば、mm/memory.cの次の文は、関数do_exit()とoom()が呼び出し元のコードに戻らなくなつたことを示しています。

```
31 volatile void do_exit(long code); 32
35 33 static inline volatile void oom(void) 34 {
36     printk("out of memory\n\r");
37     do_exit(SIGSEGV);
37 }
```

ここにもっと長い例があります。これが読めれば、インラインアセンブリコードは基本的にOKということになります。このコードはinclude/string.hファイルから引用したもので、strcmp()の文字列比較関数の実装です。同様に、この各行の「`¥`」は、gccプリプロセッサの出力リストが見栄え良くなるように設定されています。

```
//// 文字列1と文字列2の最初のカウント文字数を比較します。
// Paras: cs - Strings1,ct - String2,count - 比較する文字の数です。
// %0 - eax( res ) return,%1 - edi(cs) String1 ptr,%2 - esi(ct)String2 ptr,%3 - ecx(count) 。
// 返します。文字列1>文字列2の場合は1を、文字列1==文字列2の場合は0を、文字列1<文字列2の場合は-1を返す。 extern inline int strcmp(const char * cs,const char * ct,int count)
{
register int __res ;                                // __res is a register variable.
__asm__("cld\n"                                     // Clear direction.
        "1:\tdecl %3\n\t"                            // count--.
        "js 2f\n\t"                                    // If count<0, go forward to label 2.
        "lodsb\n\t"                                   // Take string 2 character ds:[esi]>al, and esi++.
        "scasb\n\t"                                   // Compare char in al and in string1 es:[edi] and edi++.
        "jne 3f\n\t"                                    // If they are not equal, go forward to label 3.
        "testb %%al,%%al\n\t"                          // Is this character a NULL character?
        "jne 1b\n"                                     // No, go backward to label 1 and continue comparing.
        "2:\txorl %%eax,%%eax\n\t"                    // If it is a NULL char, eax is cleared (return value).
        "jmp 4f\n"                                     // Go forward to label 4 and end.
        "3:\tmovl $1,%%eax\n\t"                        // eax is set to 1.
        "jl 4f\n\t"                                    // If the string2 chars <string1 chars, return 1 and end.
        "negl %%eax\n"                                // Otherwise eax = -eax returns a negative value, ends.
        "4:"                                          
        :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
return __res;                                         // Return the comparison result.
}
```

3.3.5 Combination statements in parentheses

中括弧「{...}」は、変数宣言やステートメントを複合ステートメント（コンビネーションステートメント）やステートメントブロックにまとめ、意味的に1つのステートメントと同等になるようにするために使用します。複合文の閉じ括弧の後にセミコロンを使用する必要はありません。括弧内の複合文、すなわち"({...})"形式の文は、GNU Cでは式として使用することができます。これにより、ループ、

switch文、ローカル変数を式の中で使用することができるため、この形式の文は、しばしば

文章表現です。文章表現は、以下のような例示形式になっています。

```
{ int y = foo(); int z;
    if (y > 0) z = y;
    else z = -y;
```

複合ステートメントの最後のステートメントは、セミコロンが続く式でなければなりません。この式の値（「 $z + z$ 」）は、全体の括弧で囲まれた値として使用されます。最後の文が式でない場合は、文の式全体がvoid型であるため、値はありません。また、このような式の中のステートメントで宣言されたローカル変数は、ブロックステートメント全体が終了した後に失効します。このサンプル文は、次のような形式の代入文のように使うことができます。

res = x + ({Omit...}) + b となります。

もちろん、普通の人は上記のような文を書かないので、通常はマクロの定義に使われます。例えば、カーネルのソースコードであるinit/main.cプログラムの中にあるCMOSクロック情報を読み取るためのマクロ定義です。

```
69 #define CMOS_READ(addr) ({ \
70     outb_p(0x80|addr, 0x70); \
71     inb_p(0x71); \
72 })
```

// First, output the addr to the I/O port 0x70.
// Then read the value from port 0x71 as the return value.

ヘッダファイルinclude/asm/io.hのリードI/Oポートのマクロ定義をもう一度見てみると、最後の変数_vの値がinb()の戻り値になっています。

```
05 #define inb(port) ({ \
06     unsigned char _v; \
07     __asm volatile ("inb %%dx,%al": "=a" (_v): "d" (port)); \
08 })
```

3.3.6 Register variables

GNUのC言語へのもう一つの拡張機能では、いくつかの変数の値をCPUのレジスタに入れることができます、いわゆるレジスタ変数です。この方法では、CPUは値を求めてメモリにアクセスするために長い時間を費やす必要がありません。レジスタ変数には、グローバルレジスタ変数とローカルレジスタ変数の2種類がある。グローバルレジスタ変数は、プログラムの動作中、複数のグローバル変数専用のレジスタを保持する。一方、ローカルレジスタ変数は、指定されたレジスタを保持せず、特別なレジスタはインラインasmアセンブリステートメントの入力または出力オペランドとしてのみ使用されます。gccコンパイラのデータフロー解析機能は、指定されたレジスタに使用中の値があるかどうか、他のフィールドをディスパッチできるかどうかを本質的に判断することができます。gccのデータフロー解析機能は、ローカル・レジスタ変数の値が無駄になったときに格納されていると考えられる場合、その値を削除したり、ローカル・レジスタ変数への参照も削除、移動、簡略化したりすることができます。したがって、gccにこのような最適化の変更をさせたくない場合には、asm文にvolatileキーワードを追加するとよいでしょう。

アセンブラ命令の出力をインラインで指定したレジスタに直接書き込みたい場合は

アセンブラー文では、このときにローカルレジスタ変数を使うと便利です。Linux カーネルでは、通常、ローカルレジスタ変数しか使用しませんので、ここでは、ローカルレジスタ変数の使い方についてのみ説明します。GNU C プログラムでは、次のように関数の中でローカルレジスタ変数を定義します。

register int res asm ("ax") です。

3.3.7 ここで `ax` は、変数 `res` が使用したいレジスタです。このようなレジスタ変数を定義しても、特にこのレジスタを他の目的のために確保するわけではありません。プログラムのコンパイル中に、`gcc` のデータフロー制御によって、変数の値が使用されなくなったと判断された場合、そのレジスタは他の目的のためにディスパッチされたり、そのレジスタへの参照が削除されたり、移動されたり、簡略化されたりします。また、`gcc` はコンパイルされたコードが変数を指定されたレジスタに保持することを保証しません。したがって、アセンブリに組み込まれる命令の部分では、このレジスタを明示的に参照しない方がよく、レジスタがこの変数の値を参照しなければならないと考えられます。ただし、この変数を `asm` のオペランドとして使用すると、指定されたレジスタがオペランドとして使用されることが保証されます。

3.3.8 Inline function

プログラムの中で、関数をインライン関数として宣言することで、`gcc` は関数のコードを関数を呼び出すコードに統合することができます。この処理により、関数呼び出し時の入出力のオーバーヘッドを取り除くことができ、実行速度を確実に向上させることができます。したがって、関数をインライン関数として宣言する主な目的は、関数本体ができるだけ早く実行できるようにすることにあります。また、インライン関数の中に定数値がある場合、`gcc` はコンパイル時にその定数値を使って何らかの簡略化を行うことがありますので、すべてのインライン関数コードが埋め込まれるわけではありません。インライン関数方式は、プログラムコードの長さに明らかな影響はありません。インライン関数を使用してコンパイルされたプログラムは、状況に応じて長いまたは短いターゲットコードを生成します。

インライン関数を呼び出し側のコードに埋め込む操作は、最適化操作であるため、最適化されたコンパイルが行われた場合にのみコード埋め込み処理が行われます。コンパイル時に最適化オプション「`-O`」を使用しなかった場合、インライン関数のコードは実際には呼び出し元のコードには埋め込まれず、通常の関数呼び出しとしてのみ扱われます。関数をインライン関数として宣言するには、カーネルファイル `fs/inode.c` 内の以下の関数のように、関数宣言にキーワード "inline" を使用します。

```
01 インライン int inc(int *a)
02 {
03     (*a)++;
04 }
```

関数内のいくつかのステートメントを使用すると、インライン関数の置き換えが正常に行われない場合や、置き換え操作に適していない場合があります。例えば、変数パラメータ、メモリ確保関数 `malloc()`、可変長データ型変数、ノンローカル `goto` 文、再帰関数などが使用されています。コンパイ

ラは、オプション -finline を使用することで、インラインとマークされているのに置換できない関数について、その理由を含めた警告情報を gcc に与えることができます。

関数定義において、`inline`キーワードと`static`キーワードの両方を使用した場合、例えば以下のファイル `fs/inode.c` のインライン関数の定義では、インライン関数の呼び出しが置換された場合、すべての呼び出しが置換されます。呼び出し側のコードで、プログラムがインライン関数のアドレスを参照していない場合、インライン関数のアセンブリコード自体は参照されません。この場合、コンパイル時にオプション `-fkeep-inline-functions` を使用しない限り、gcc はインライン関数自体の実際のアセンブリコードを生成しなくなります。何らかの理由で、いくつかの

インライン関数の呼び出しを関数に統合することはできません。特に、インライン関数の定義の前にある呼び出し文は統合によって置き換えられず、再帰によって定義された関数にすることはできません。統合で置き換えられない呼び出しがあった場合、インライン関数は通常通りアセンブリコードにコンパイルされます。もちろん、プログラムにインライン関数のアドレスを参照するステートメントがあれば、インライン関数も通常通りアセンブリコードにコンパイルされます。インライン関数のアドレスへの参照は置き換えられないからです。

```
20 static inline void wait_on_inode(struct m_inode * inode)
21 {
22     cli();
23     while (inode->i_lock)
24         sleep_on(&inode->i_wait);
25     sti();
26 }
```

なお、ISO標準C99にはインライン関数の機能が盛り込まれていますが、この標準で定義されているインライン関数は、gccで定義されているものとは全く異なります。ISO標準C99のインライン関数のセマンティックな定義は、キーワードinlineとstaticの組み合わせの定義と同等であり、キーワードstaticを「排除」することを意味します。プログラムにC99規格のセマンティクスを使用する必要がある場合は、コンパイルオプションの-std=gnu99を使用する必要があります。しかし、互換性を保つためには、この場合でもインラインとスタティックの組み合わせを使うのがベストです。その後、gccは最終的にC99の定義をデフォルトで使用するようになります。ここで定義されたセマンティクスをまだ使いたい場合は、オプション -std=gnu89 を使って指定する必要があります。

インライン関数の定義にキーワード static が使われていない場合、gcc は他のプログラムファイルでもこの関数が呼び出されていると判断します。グローバルシンボルは一度しか定義できないため、他のソースファイルでその関数を定義することはできなくなります。したがって、インライン関数の呼び出しを、ここでの統合で置き換えることはできません。したがって、非静的なインライン関数は、常に独自のアセンブリコードでコンパイルされます。この点、ISO 標準 C99 の static キーワードを使用しないインライン関数の定義は、ここで static キーワードの定義を使用することと同じです。

関数の定義時にinlineとexternの両方のキーワードが指定されている場合、関数の定義はinlineの統合にのみ使用され、関数が明示的に参照されていても、関数自身のアセンブリコードはいかなる場合でも個別に生成されません。また、アドレスも生成されません。このようなアドレスは、関数を定義せずに関数を宣言しただけの場合と同様に、外部参照となります。

inlineとexternの組み合わせは、マクロ定義とほぼ同じです。この組み合わせ方法は、組み合わせキーワードを持つ関数定義を.hヘッダーファイルに置き、キーワードを持たない同じ関数の定義をライブラリファイルに置くというものです。このとき、ヘッダーファイルに定義があると、その関数の呼び出しのほとんどが置換によって埋め込まれます。置換されていない関数の呼び出しがある場合は、プログ

ラムファイルやライブラリのコピーが使用されます（参照されます）。Linux 0.1x カーネルのソースコードに含まれる include/string.h, lib/strings.c というファイルが、この使用例です。たとえば、string.h には次のような関数が定義されています。

```
// 文字列 (src) を別の文字列 (dest) に、NULL文字に出会うまでコピーする。
// 0 - esi(src),%1 - edi(dest)。
29 27 extern inline char * strcpy(char * dest,const char *src) 28 {
30     __asm__ ("cld\n"                      // Clear direction.
31         "1:lodsb\n"                      // Load DS: [esi] 1 byte => al and update esi.
```

```

31      "stosb|n|t"           // Store byte al=>ES:[edi] and update edi.
32      "testb %%al, %%al|n|t" // Just stored byte al is 0?
33      "jne 1b"             // If not, go back to label 1 or end.
34      :: "S" (src), "D" (dest): "si", "di", "ax";
35  return dest;           // Returns the destination string pointer.
36 }

```

カーネルライブラリのディレクトリにあるlib/strings.cファイルでは、以下のようにインラインとエクスターントというキーワードが空しく定義されています。したがって、実際には、カーネルライブラリには、string.hファイル内のこれらの関数のすべてのコピーが含まれており、それによって、これらの関数が一度再定義され、2つのキーワードの効果が「なくなる」のです。

```

11 #define extern           // Defined as empty.
12 #define inline           // Defined as empty.
13 #define _LIBRARY
14 #include <string.h>
15

```

上記で定義したライブラリ関数のstrcpy()は、次のようにになります。

```

27 char * strcpy(char * dest, const char *src) // Removed keywords inline and extern.
28 {
29     asm ("cld\n"
30          "1:|t1lodsb|n|t"           // Clear direction.
31          "stosb|n|t"             // Load DS: [esi] 1 byte => al and update esi.
32          "testb %%al, %%al|n|t"   // Store byte al=>ES:[edi] and update edi.
33          "jne 1b"               // Just stored byte al is 0?
34          :: "S" (src), "D" (dest): "si", "di", "ax";
35  return dest;           // Returns the destination string pointer.
36 }

```

3.4 Interworking between C and Assembly language

3.4.1 コードの実行効率を高めるために、カーネルのソースコードの一部ではアセンブリ言語を直接使用しています。その際、2つの言語プログラム間の呼び出し問題が発生します。ここでは、まずC言語の関数の呼び出しの仕組みを説明し、次に例を用いて2つの関数間の呼び出し方法を説明する。

3.4.2 C function call mechanism

Linuxカーネルプログラムboot/head.sが基本的な初期化処理を行った後、init/main.cプログラムの実行にジャンプします。では、head.sプログラムは、どのようにしてinit/main.cプログラムに実行制御を移すのでしょうか？それは、アセンブリがC言語プログラムを実行するためにどのように呼び出しているのか？ここでは、まずC言語の関数呼び出しの仕組み、制御転送モードを説明し、その後、head.sプログラムがCプログラムにジャンプすることを説明します。

関数呼び出しの操作には、あるコードから別のコードへの双方向のデータ転送と実行制御の転送があ

ります。データの受け渡しは、関数のパラメータと戻り値によって行われます。また、関数に入るときには関数のローカル変数の記憶領域を確保し、関数を抜けるときにはこの領域を取り戻す必要があります。インテル80x86のCPUでは、制御の受け渡しには簡単な命令が用意されていますが、その一方で

3.4.21 データや、ローカル変数の記憶領域の確保と回復は、スタック操作によって実現されます。

3.4.22 Stack frame structure and control transfer method

ほとんどのプログラム実装では、関数呼び出し操作をサポートするためにスタックを使用します。スタックは、関数のパラメータを転送したり、リターン情報を保存したり、リカバリーのためにレジスタの元の値を一時的に保存したり、ローカルデータを保存したりするために使用されます。1つの関数呼び出し操作で使用されるスタックの部分は、スタックフレーム構造と呼ばれます。その一般的な構造を図 3-4 に示します。スタックフレーム構造の両端は、2つのポインタで指定されます。レジスター `ebp` は通常、フレームポインタとして使用され、`esp` はスタックポインタとして使用されます。関数の実行中、スタックポインタ `esp` は、スタックにプッシュされるデータとともに移動します。したがって、関数内のほとんどのデータアクセスは、フレームポインタ `ebp` に基づいて行われます。

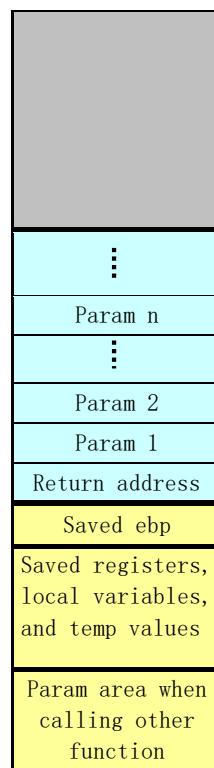


図3-4 スタック内のフレーム構造

関数Aが関数Bを呼び出す場合、Bに渡されるパラメータはAのスタックフレームに含まれています。AがBを呼び出す際には、関数Aのリターンアドレス（呼び出しが返された後に実行を継続する命令のアドレス）がスタックにプッシュされます。また、このスタック上の位置は、Aのスタックフレームの終わりを明示的に示します。Bのスタックフレームは、フレームポインタ（`ebp`）が格納されている後のスタックセクションから始まります。その後、保存されたレジスタ値や関数の一時的な値を格納するために使用されます。

B関数は、レジスタに配置できないローカル変数の値を保持するためにもスタックを使用します。例え

ば、通常のCPUではレジスタの数が限られており、関数のローカルデータをすべて格納することができないとか、ローカル変数の中には配列や構造体のものがあるので、配列や構造体の参照を使ってアクセスしなければならないなどです。また、C言語のアドレス演算子「&」がローカル変数に適用された場合、変数のアドレスを生成する必要があり、変数のアドレスポインタのための空間が確保されます。最後に、B関数はスタックを使用して、他の関数を呼び出すパラメータを保存します。

スタックはロー（スモール）アドレスに向かって拡張され、espは現在のスタックのトップにある要素を指します。プッシュ命令とポップ命令を使うことで、データをスタックにプッシュしたり、スタックからポップしたりすることができます。初期データが指定されていない記憶領域では、スタックポインタを適切な値だけデクリメントすることでこれを行うことができます。同様に、スタックポインタの値を増やすことで、スタック上に割り当てられた空間を取り戻すことができます。

関数の呼び出しと戻りの操作には、CALLとRETという命令が使われる。CALL命令の効果は、リターンアドレスをスタックにプッシュし、呼び出された関数の先頭にジャンプすることである。リターンアドレスとは、プログラム内でCALL命令の直後にある命令のアドレスのことです。そのため、呼び出された関数が戻ってきたときには、その位置から続けて実行されます。リターン命令RETは、スタックの一番上のアドレスをポップアップし、そのアドレスにジャンプするために使用されます。この命令を使用する前に、スタックの内容を正しく処理して、現在のスタックポインタが前のCALL命令で返されたものと同じになるようにしなければなりません。また、戻り値が整数またはポインタの場合、戻り値を渡すためにレジスタ eax がデフォルトで使用されます。

3.4.23

一度に実行されるのは1つの関数だけですが、ある関数（caller）が別の関数（callee）を呼び出す際に、calleeがcallerが将来使用するレジスタの内容を変更したり上書きしたりしないようにする必要があります。そのため、インテルのCPUでは、すべての関数が遵守しなければならないレジスタの統一使用法を採用している。この規約では、レジスタeax、edx、ecxの内容は、呼び出し元自身が保持しなければならないことを示しています。関数BがAから呼び出されたとき、関数Bは、関数Aが必要とするデータを破壊することなく、これらのレジスタの内容を保存することなく、任意に使用することができます。さらに、レジスタebx、esi、およびediの内容は、呼び出し側Bによって保護されなければなりません。呼び出し側のA（または上位の関数）はこれらのレジスタの内容を保存する責任がないため、今後の操作で元の値を使用する必要があるかもしれません。また、2つ目の規約の使い方に従わなければならぬレジスタebpとespがあります。

3.4.24 Function call example

例として、次のCプログラムexch.cの関数呼び出しの処理を見てみましょう。このプログラムは、2つの変数の値を交換し、その差を返します。

```
1 void swap(int * a, int * b)
2 {
3     int c;
4     c = *a; *a = *b; *b = c;
5 }
6
7 int main() 8
{
9     int a, b;
10    a = 16; b = 32;
11    swap(&a, &b);
12    return (a - b);
13 }
```

swap()という関数は、2つの変数の値を交換するために使われます。Cプログラムのメインプログラム

であるmain()も関数である（後述）。swap()を呼び出した後、スワップされた結果を返します。これら2つの関数のスタックフレーム構造を図35に示します。ご覧のとおり、swap()関数は呼び出し元（main()）のスタックフレームからパラメータを取得しています。図中の位置情報は、レジスタebp内のフレームポインタからの相対値です。スタックフレームの左側にある数字は、フレームポインタに対するアドレスオフセット値を示しています。gdbなどのデバッガでは、これらの値は2の補数で表されます。例えば、「-4」は「0xFFFFFFFFC」と表されます。

となり、「-12」は「0xFFFFFFF4」と表現されます。

呼び出し元のmain()のスタックフレーム構造には、フレームポインタに対して-4と-8のオフセットに位置するローカル変数aとbの格納スペースが含まれています。この2つのローカル変数のアドレスを生成する必要があるため、単にレジスタに格納するのではなく、スタックに格納する必要があります。

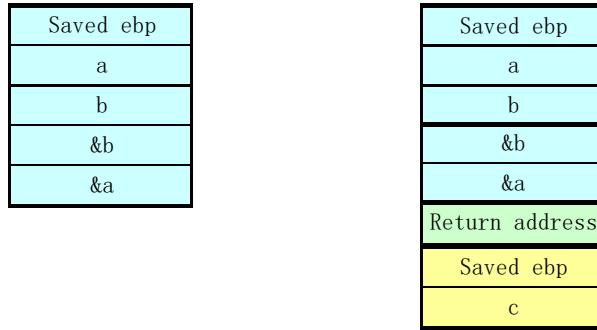


図3-5 関数mainとswapを呼び出したときのスタックフレーム構造

gcc -Wall -S -oexch.s exch.c」コマンドを使用して、このC言語プログラムのアセンブリexch.sコードを以下のように生成します（議論に関係のない数行の指示を削除します）。

```

1 .text
2 _swap:
3     pushl %ebp          # Save original ebp, set current function's frame pointer.
4     movl %esp, %ebp
5     subl $4, %esp        # Allocates space within the stack for the local variable c.
6     movl 8(%ebp), %eax   # Get 1st argument, which is a pointer to an integer value.
7     movl (%eax), %ecx    # Store value pointed by the pointer into variable c.
8     movl %ecx, -4(%ebp)  # Take 1st parameter again, and then take 2nd parameter.
9     movl 8(%ebp), %eax
10    movl 12(%ebp), %edx
11    movl (%edx), %ecx
12    movl %ecx, (%eax)
13    movl 12(%ebp), %eax
14    movl -4(%ebp), %ecx
15    movl %ecx, (%eax)
16    leave                # Restore the original ebp and esp.
17    ret
18 _main:
19     pushl %ebp          # Save original ebp, set current function's frame pointer.
20     movl %esp, %ebp
21     subl $8, %esp        # Allocates space in stack for local variables a and b.
22     movl $16, -4(%ebp)   # Assign initial values to variables (a=16, b=32).
23     movl $32, -8(%ebp)
24     leal -8(%ebp), %eax  # To call the swap(), push the address of variable b onto
25     pushl %eax           # stack. That is, push the second parameter first.
26     leal -4(%ebp), %eax  # Then push address of variable a as the first parameter.
27     pushl %eax

```

| | | |
|----|---------------------|-------------------------------------|
| 28 | call _swap | # Call the function swap(). |
| 29 | movl -4(%ebp), %eax | # Take the value of a - b. |
| 30 | subl -8(%ebp), %eax | |
| 31 | leave | # Restore the original ebp and esp. |
| 32 | ret | |

この2つの関数は、独立して3つの部分に分けることができます。スタックフレーム構造を初期化する "set"、関数の実際の計算を行う "body"、スタックの状態を復元して関数から戻る "end" です。swap() 関数の場合、設定部分のコードは3~5行です。最初の2行は、呼び出し元のフレームポインタの設定と、関数のスタックフレームポインタの設定に使われます。5行目では、スタックポインタespを4バイト下に移動させることで、ローカル変数cの領域を確保しています。6~15行目はスワップ関数のメイン部分です。6-8行目は、呼び出し元の最初のパラメータである&aを取得し、このパラメータをアドレスとして、メモリの内容をecxレジスタにフェッチして、ローカル変数に割り当てられたスペース(-4(%ebp))に保存するために使用されます。9-12行目は、2番目のパラメータである&bをフェッチし、そのパラメータ値をアドレスとして、その内容を1番目のパラメータで指定されたアドレスに取り込みます。13~15行目は、一時的なローカル変数cに格納された値を、第2パラメータで指定されたアドレスに格納しています。最後の16~17行目が関数の終了です。leave命令は、リターンに備えてスタックの内容を処理するために使用されます。その役割は、以下の2つの命令と同等です。

| | |
|-----------------|---|
| movl %ebp, %esp | # Restores original esp (to beginning of stack frame). |
| popl %ebp | # Restores original ebp (usually the caller's frame ptr). |

この2行のコードは、swap()関数の入力時にレジスタespとebpの値を元に戻し、ret命令を実行するものです。

19-21行目はmain()関数のセット部分です。フレームポインタの保存とリセットを行った後、main()はスタック上にローカル変数aとbの領域を確保します。22-23行目では、この2つのローカル変数に値を割り当てています。24~28行目では、main()がswap()関数を呼び出していることがわかります。まず、leal命令（実効アドレスの取得）で変数bとaのアドレスを取得してスタックにプッシュしてから、swap()関数を呼び出しています。変数のアドレスがスタックにプッシュされる順番は、関数で宣言されたパラメータの順番とは全く逆になります。つまり、関数の最後のパラメータが最初にスタックにプッシュされ、関数の最初のパラメータは、関数命令呼び出しの前にスタックにプッシュされます。29--30行目では、すでに交換された2つの数値を引き算し、それを戻り値としてeaxレジスタに入っています。

34.25

以上の分析から、C言語は関数を呼び出す際に、転送された関数パラメータの値を一時的にスタックに格納することがわかります。つまり、C言語は値ベースの言語なのです。呼び出された関数の呼び出し側の変数を直接修正する方法はありません。値を修正します。したがって、修正の目的を達成するためには、変数へのポインタ（つまり、変数のアドレス）を関数に渡す必要があります。

34.26 Main() is also a function

上記のアセンブラーコードは、gcc 1.40を使ってコンパイルされています。数行の余分なコードがあることがわかります。これは、当時の gcc コンパイラが最も効率的なコードを生成できなかったことを示

しています。これが、いくつかの重要なコードを直接アセンブリ言語でコンパイルする必要がある理由の1つです。また、先ほどのC言語プログラムのメインプログラムも関数です。これは、リンクをコンパイルしたときに、`crt0.s`というアセンブリプログラムの関数として呼び出されるからです。`crt` "は "C run-time" の略です。このプログラムのターゲットファイルは、各ユーザーの実行プログラムの最初にリンクされ、主にいくつかの初期化グローバル変数を設定するために使用されます。Linux 0.12での`crt0.s`アセンブリプログラムを以下に示します。プログラム内の他のモジュールが使用するために、グローバル変数`_environ`が作成され、初期化されます。

```

1 .text
2 .globl _environ
3
4 __entry:           # Code entry label.
5     movl 8(%esp), %eax    # Get environment variable pointer envp, save in _environ.
6     movl %eax, _environ   # envp is set by execve() when executable file is loaded.
7     call _main            # Call main program. Its return status is in eax register.
8     pushl %eax            # Push return value as an argument to exit() and call it.
9 1:    call _exit
10    jmp 1b               # Control should not arrive here.
11 .data
12 _environ:          # Define variable _environ and assign it a long word space.
13     .long 0

```

実行ファイルのコンパイルとリンクにgccを使用した場合、gccは自動的にcrt0.sのコードを実行プログラムの最初のモジュールとしてリンクします。コンパイル時にshow detailsオプション'-v'を使用すると、リンク処理を明確に確認することができます。

```

[/usr/root]# gcc -v -o exch exch.s
gccバージョン1.40
/usr/local/lib/gcc-as -o exch.o exch.s
/usr/local/lib/gcc-ld -o exch /usr/local/lib/crt0.o exch.o /usr/local/lib/gnulib -lc
[/usr/root]# gnulib

```

そのため、通常のコンパイルでは、スタブモジュールであるcrt0.oを指定する必要はありませんが、ld(gld)を使用して、上記で示したアセンブリプログラムから手動でexch.oモジュールから実行ファイルを生成する場合にはコマンドラインでcrt0.oモジュールを指定し、リンク順序を "crt0.o, すべてのプログラムモジュール, ライブラリファイル"とする必要があります。

ELF形式のオブジェクトファイルを使用し、共有ライブラリのモジュールファイルを作成するために、現在のgccコンパイラ(2.x)では、このcrt0を複数のモジュールに拡張しています：crt1.o、crti.o、crtbegin.o、crt1.o、crti.o、crtbegin.o (crtbeginS.o)，全プログラムモジュール，crrtend.o (crrtendS.o)，crtn.o，「ライブラリモジュールファイル」の順にリンクされています。このリンク順序は、gccの設定ファイルspecfileで指定します。ここで、crt1.o、crti.o、crtn.oは、Cプログラムの「起動」モジュールであるCライブラリから提供され、crtbegin.o、crrtend.oは、コンパイラgccから提供されるC++言語の起動モジュールであり、crt1.o これは、crt0.oの効果と同様に、主にmain()を呼び出す前の初期化作業に使用される。グローバル・シンボル_startは、このモジュールで定義されています。

crtbegin.oとcrrtend.oは、主にC++言語において、.ctorsセクションと.dtorsセクションにグローバルなコンストラクタとデストラクタの関数を実装するために使用されます。crtbeginS.oとcrrtendS.oの役割は、最初の2つと同様ですが、共有モジュールを作成するために使用されます。crti.oは、.initセクションの初期化関数init()を実行するために使用されます。.initセクションには、プロセスの初期化コー

ドが含まれています。つまり、プログラムの実行開始時に、システムはmain()を呼び出す前に.initのコードを実行します。Crtn.oは、.fini領域の処理関数fini()を終了させるためのプロセスの実行に使用されます。つまり、プログラムが正常に終了する(main()が戻る)ときに、システムは.finifのコードを実行するように手配します。

3.4.3 カーネルでは、boot/head.sプログラムの136～140行目が、init/main.cのmain()関数にジャンプするための準備に使われています。139行目の命令はリターンアドレスをスタックにpushし、140行目はmain()関数コードのアドレスをpushしています。最終的にhead.sが218行目のret命令を実行すると、main()のアドレスがポップアップされ、init/main.cプログラムに制御が移ります。

3.4.4 Call C function in assembler code

アセンブリコードからC言語の関数を呼び出す方法は、実際に上記のように示されています。上のC言語の例のアセンブラーでは、アセンブラー文がswap()関数を呼び出している様子がわかります。ここで、呼び出し方法をまとめておきます。

C言語の関数をアセンブリコードで呼び出す場合、まず関数のパラメータを逆順にスタックにpushする必要があります。つまり、関数の最後（右端）のパラメータが最初にスタックにpushされ、最後の命令が呼ばれる前に左端の最初のパラメータがpushされるのである。図3-6をご覧ください。その後、CALL命令を実行して、呼び出された関数を実行する。呼び出した関数が戻ってきた後、プログラムは以前にスタックにpushされた関数パラメータをクリアする必要があります。

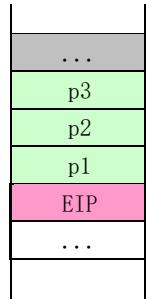


図3-6 関数が呼び出されたときにスタックに押し込まれるパラメータ

CALL命令が実行されると、CPUはCALL命令の次の命令のアドレスをスタックにpushします（図のEIP参照）。呼び出しにコードの特権レベルの変更も伴う場合、CPUはスタックスイッチを行い、現在のスタックポインタ、セグメントディスクリプタ、呼び出しパラメータを新しいスタックにpushします。なお、Linuxカーネルでは、呼び出し時の特権レベル変更の処理には割り込みゲートとトランプドアのみを使用し、CALL命令は使用していないため、ここでは特権レベル変更時のCALL命令の使用については説明しない。

アセンブリでC関数を呼び出すことは、比較的 "自由"です。スタック内の適切な場所にあれば、C関数のパラメータとして使用することができます。ここではまだ、図3-6の3つのパラメータを持つ関数呼び出しを例にしています。func()で引数をpushして直接呼び出さなくとも、func()関数はEIPの位置を保存します。スタックの残りの部分は自分のパラメータとして使われます。もし、func()呼び出しのために、1番目と2番目のパラメータを明示的に押したとすると、func()関数の3番目のパラメータp3は、p2以前のスタックの内容を直接使用します。このLinux 0.1xのカーネルコードにはいくつかの箇所があります。例えば、copy_process()関数(kernel/fork.cの68行目)は、kernel/sys_call.sのアセンブリプロ

ログラムの231行目で呼び出されています。アセンブリ関数_sys_forkでは5つのパラメータしかスタックにpushされていませんが、copy_process()では以下のように最大17個のパラメータが設定されています。

```
// kernel/sys_call.sの部分的なプログラムです。_sys_fork
```

```

226      push %gs
227      pushl %esi
228      pushl %edi
229      pushl %ebp
230      pushl %eax
231      call _copy_process      # Call the C function copy_process() (kernel/fork.c, 68).
232      addl $20,%esp          # Discard all the pushed content here.
233 1:   ret

```

68 // kernel/fork.c の部分プログラムです。

```

69 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
70                 long ebx, long ecx, long edx, long orig_eax,
71                 long fs, long es, long ds,
72                 long eip, long cs, long eflags, long esp, long ss)

```

パラメータがスタックにpushされるのが遅ければ遅いほど、C関数のパラメータの左端に近いことがわかっています。したがって、copy_process()が呼ばれる前に実際にpushされた5つのレジスタ値が、copy_process()関数の5つの左端のパラメータとなります。順に、スタックされているeax(nr)、ebp、edi、esi、レジスタgsの値に対応しています。以下のパラメータの残りの部分は、実際にはすでにスタック上にあるものに直接対応しています。これらの内容は、システムコール割り込み処理プロセスが開始されてから、システムコールプロセスが呼び出されるまでの間に、スタックに徐々に追加される各種レジスタの値です。

パラメータnoneは、sys_call.sプログラムの99行目のアドレスジャンプテーブルから_sys_forkが呼び出されたときの、次の命令のリターンアドレスです。アドレスジャンプテーブルsys_call_table[]は、ヘッダファイルinclude/linux/sys.hの93行目で定義されています。次のパラメータは、レジスタebx、ecx、edx、元のeax、そしてsystem_call入力直後の85~91行目でスタックにpushされたセグメントレジスタfs、es、dsです。最後の5つのパラメータは、CPUの実行割り込み命令のpushリターンアドレスeipとcs、フラグレジスタeflags、ユーザースタックアドレスespとssです。システムコールはプログラムの特権レベルの変更を伴うため、CPUはフラグレジスタの値とユーザースタックアドレスをスタックにpushします。呼び出したC関数copy_process()が戻ってきた後、_sys_forkは自分で押した5つのパラメータだけを破棄し、残りのスタックも保存します。上記の使い方をしている他の関数として、kernel/signal.cのdo_signal()、fs/exec.cのdo_execve()などがあります。ぜひ、ご自身で解析してみてください。

3.4.5 また、CALL命令を使わずに、JMP命令を使って関数を呼び出すのと同じ目的を達成できるので、アンブリがC関数を呼び出すのは比較的自由だと言います。その方法は、パラメータをスタックにpushした後、次に実行する命令のアドレスを手動でスタックに入れ、直接JMP命令を使って呼び出された関数の開始アドレスにジャンプして関数を実行するというものです。その後、関数の実行が完了すると、RET命令が実行され、手動でスタックにpushした次の命令のアドレスを、関数から返され

たアドレスとしてポップアップします。また、Linuxカーネルでは、kernel/asm.sプログラムの62行目でtraps.cのdo_int3()関数を呼び出す場合など、様々な方法でこの関数を呼び出すことができます。

3.4.6 Call assembly function in C program

Cプログラムからアセンブリ関数を呼び出すことは、アセンブリでC関数を呼び出すことと同じですが、Linuxカーネルプログラムではありません。呼び出し方法の焦点は、やはり関数のパラメータのスタック内の位置の決定にあります。もちろん、呼び出し元のアセンブリ言語プログラムが比較的短ければ、上述のインラインアセンブリ文を使ってCプログラムに直接実装することもできます。以下、例を使ってこのようなプログラムの書き方を説明します。アセンブリは、2つを含むcalle.s

の機能を以下に示します。

```
/*
このアセンブリ言語プログラムは、システムコールsys_write()を使用して、表示機能int mywrite(int fd,
char * buf, int count)を実装しています。

関数int myadd(int a, int b, int * res)は、a+b = resの演算を行うために使用されます。この関数が0を返
す場合は、オーバーフローを意味します。

注：現行のLinuxシステム（RedHat 9など）でコンパイルする場合は、関数名の前のアンダ
ースコア「_」を削除してください。

SYSWRITE = 4                                # Sys_write() system call number.

.globl _mywrite, _myadd

.テキスト

_mywrite:
    pushl  %ebp
    movl  %esp, %ebp
    pushl  %ebx
    movl  8(%ebp), %ebx      # Take the first argument of the caller: file descriptor fd.
    movl  12(%ebp), %ecx     # The second parameter: buffer pointer.
    movl  16(%ebp), %edx     # The third parameter: display character number.
    movl  $SYSWRITE, %eax     # Put system call number 4 in %eax.
    int   $0x80                # Execute the system call.
    popl  %ebx
    movl  %ebp, %esp
    popl  %ebp
    ret

_myadd:
    pushl  %ebp
    movl  %esp, %ebp
    movl  8(%ebp), %eax      # Get the first parameter a.
    movl  12(%ebp), %edx     # Get the second parameter b.
    xorl  %ecx, %ecx         # If %ecx is 0, the calculation overflows.
    addl  %eax, %edx          # Perform additions.
    jo    1f                  # Jump if it overflows.
    movl  16(%ebp), %eax     # Take the third parameter pointer.
    movl  %edx, (%eax)        # Put the result in the position of the pointer.
    incl  %ecx                # No overflow occurred, so set no overflow return value.
1:   movl  %ecx, %eax          # %eax is the function return value.
    movl  %ebp, %esp
    popl  %ebp
    ret
```

アセンブリファイルの最初の関数mywrite()は、システム割り込み0x80を利用して、システムコールsys_write(int fd, char *buf, int count)を呼び出し、画面に情報を表示します。対応するシステムコールの関数番号は4です（include/unistd.h参照）。3つのパラメータは、ファイルディスクリプタ、表示バッファポインタ、表示文字数です。int 0x80を実行する前に、呼び出し元の関数番号(4)をレジスター%eaxに入れ、呼び出し規則に従って、レジスタ%ebx、%ecx、%edxにそれぞれfd、buf、countを格納する必要があります。関数の引数であるmywrite()は、sys_write()と全く同じ数のパラメータと用途

を使用しています。

2番目の関数myadd(int a, int b, int *res)は、足し算の演算を行います。パラメータresは演算の結果です。関数の戻り値は、オーバーフローが発生したかどうかを判断するために使用されます。戻り値が0の場合、計算がオーバーフローしており、結果は得られません。それ以外の場合は、計算の結果

は、パラメータresを介して呼び出し側に返されます。

なお、現行のLinuxシステム（例：RedHat 9）でcallee.sをコンパイルする場合は、関数名の前のアンダースコア「_」を削除してください。この2つの関数を呼び出すCプログラムcaller.cを以下に示します。

```
/*
アセンブリ関数 mywrite(fd, buf, count)を呼び出して情報を表示します。
myadd(a, b, result)を呼び出して加算を行います。myadd()が0を返す場合は、オーバーフローを
示します。最初に計算開始情報が表示され、次に演算結果が表示されます。
*/
01 int main()
02 {
03     char buf[1024];
04     int a, b, res;
05     char * mystr = "Calculating...\\n";
06     char * emsg = "Error in adding\\n";
07
08     a = 5; b = 10;
09     mywrite(1, mystr, strlen(mystr));
10    if (myadd(a, b, &res)) {
11        sprintf(buf, "The result is %d\\n", res);
12        mywrite(1, buf, strlen(buf));
13    } else {
14        mywrite(1, emsg, strlen(emsg)); 15
15
16    return 0;
17 }
```

このプログラムでは、まずアセンブリ関数のmywrite()を使って画面に「Calculating...」という情報を表示し、次に加算計算関数のmyadd()を呼び出して2つの数字aとbを演算し、3番目のパラメータresに計算結果を返します。最後に、mywrite()関数を使って、整形された結果情報文字列を画面に表示します。myadd()関数が0を返した場合は、overflow関数がオーバーフローして計算結果が無効になったことを意味します。この2つのファイルのコンパイル結果と実行結果を以下に示します。

```
[/usr/root]# as -o callee.o callee.s
[/usr/root]# gcc -o caller caller.c callee.o
[/usr/root]# ./caller
```

3.5 Linux 0.12 Object file format

カーネルコードを生成するために、Linux 0.12は2つのコンパイラを使用しています。一つはアセンブリ as86 と、それに対応するリンク（またはリンク）ld86 です。これらは、16 ビットのカーネルブートセクタプログラム bootsect.s と、実アドレスモードで動作するセットアッププログラム setup.s のコンパイルとリンクにのみ使用されます。2つ目は、GNUアセンブリas(gas)、Cコンパイラgccとそれに対応するリンクgldです。このコンパイラは

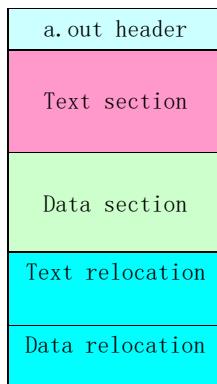
ソースプログラムファイルに対応するバイナリコードとデータのオブジェクトファイルです。リンクは、関連するすべてのオブジェクトファイルを結合して、カーネルが読み込むことのできるターゲットファイル、すなわち実行ファイルを形成するために使用される。

このセクションでは、まず、コンパイラが生成するオブジェクトファイルの構造を簡単に説明し、次に、リンカーが、リンクが必要なオブジェクトファイルのモジュールを組み合わせて、バイナリ実行可能なイメージファイルや大きなモジュールファイルを生成する方法を説明します。最後に、Linux 0.12カーネルのバイナリコードファイルImageの生成原理とプロセスを説明しています。Linux 0.12 カーネルでサポートされている a.out オブジェクトファイルフォーマットに関する情報を提供しています。As86とld86は、MINIX固有のオブジェクトファイルフォーマットを生成しますが、このフォーマットを扱うカーネル作成ツールの章で紹介します。MINIXオブジェクトファイルの構造は、a.outオブジェクトファイルフォーマットと似ているので、ここでは説明しません。オブジェクト・ファイルとリンカー・プログラムの基本的な動作原理は、John R. Levineの著書「Linkers & Loaders」に記載されています。

- 3.5.1 説明の便宜上、コンパイラによって生成されたオブジェクトファイルをオブジェクト・モジュール・ファイル（モジュールファイルと略す）と呼び、リンクプログラムによって生成された実行可能なオブジェクトファイルを実行ファイルと呼ぶ。また、これらを総称してオブジェクトファイルと呼びます。

3.5.2 Object file format

Linux 0.12システムでは、UNIXモジュールの伝統的なa.out形式が、GNU gccまたはgasコンパイラの出力するオブジェクト・モジュール・ファイルと、リンカーが生成する実行ファイルの両方で使用されています。これは、Assembly & Linker Editor Outputというオブジェクトファイル形式です。メモリのページング機構を持つシステムでは、これはシンプルで効果的なオブジェクトファイル形式です。a.out形式のファイルは、図3-7に示すように、ファイルヘッダとそれに続くコード部（テキスト部ともいう）、初期化データ部（データ部ともいう）、再配置情報部、シンボルテーブル、シンボル名の文字列構成で構成されています。コードセクションとデータセクションは、通常、それぞれテキストセグメント（コードセグメント）、データセグメントとも呼ばれる。



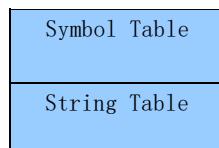


図3-7 a.out形式のオブジェクトファイル

- a.outフォーマットの7つのセクションの基本的な定義と用途は以下の通りです。
- Exec header. Execute file header section. This section contains some parameters (exec structure), which is the overall structure information about the target file. For example, the length of the code and data area, the length of the uninitialized data area, the corresponding source file name, and the target file creation time. The

- カーネルはこれらのパラメータを使って実行ファイルをメモリにロードして実行し、リンカー (ld) はこれらのパラメータを使ってモジュールファイルの一部を実行ファイルに結合します。対象文書の中で必要な部分はこれだけです。
- Text segment. The binary instruction code and data information generated by the compiler or assembler contains the instruction code and related data that are loaded into memory when the program is executed. Can be loaded as read-only.
- Data segment. Binary instruction code and data information generated by a compiler or assembler. This section contains data that has already been initialized and is always loaded into readable and writable memory.
- Text relocations. This section contains record data for use by the linker. Used to locate a pointer or address in a code segment when combining object module files. When the linker needs to change the address of the target code, it needs to be corrected and maintained.
- Data relocations. Similar to the role of the code relocation section, but used for relocation of pointers in the data segment.
- Symbol table. This section also contains record data for use by the linker. These record data hold the global symbols defined in the module file and the symbols that need to be input from other module files, or the symbols defined by the linker, used to cross named variables and functions (symbols) between module files. References.
- String table. This section contains the string corresponding to the symbol name. Used to debug program debugging target code, regardless of the linking process. This information can include source code and line numbers, local symbols, and data structure description information.

3.5.21 あるターゲットファイルに対して、上記の情報のすべてが含まれているとは限りません。Linux 0.12では、インテル製CPUのメモリ管理機能を利用しているため、実行プログラムごとに64MBのアドレス空間（論理アドレス空間）が確保されています。この場合、リンカーが実行ファイルを固定のアドレスから開始するように処理しているため、当該実行ファイルには再配置情報が不要となる。以下、重要な部分を説明する。

3.5.22 Executive header

ターゲットファイルのヘッダー部分には、32バイトのexecデータ構造があり、一般にファイルヘッダー構造や実行ヘッダー構造と呼ばれている。その定義は以下の通りです。a.out構造の詳細については、include/a.out.hファイルの後のイントロダクションを参照してください。

構造体exec {。

```
    unsigned long a_magic;          /* Use macros N_MAGIC, etc for access */
    unsigned a_text;                /* length of text, in bytes */
    unsigned a_data;                /* length of data, in bytes */
    unsigned a_bss;                 /* length of uninitialized data area for file, in bytes */
    unsigned a_syms;                /* length of symbol table data in file, in bytes */
    unsigned a_entry;               /* start address */
    unsigned a_trsize;              /* length of relocation info for text, in bytes */
    unsigned a_drsize;              /* length of relocation info for data, in bytes */
}
```

a.outファイルのヘッダー構造のマジックナンバーフィールドの値によって、a.outファイルをいくつかのタイプに分けることができます。Linux 0.12系では、2つのタイプを使用しています。モジュール・オブジェクト・ファイルは、OMAGIC (Old Magic) タイプのa.outフォーマットを使用しており、こ

のファイルがオブジェクト・ファイルまたは不純な実行ファイルであることを示しています。このタイプは

マジックナンバーは0x107（8進数0407）です。実行ファイルは、ZMAGIC a.out形式を使用しており、このファイルがデマンドドリブン（demang-paging、つまり要求に応じてロードする）ロード用の実行ファイルであることを示しています。マジックナンバーは0x10b（8進数0413）です。この2つのフォーマットの主な違いは、各パートへのストレージの割り当て方である。構造体の全長は32バイトしかありませんが、ZMAGICタイプの実行ファイルの場合、ファイルの先頭部分にはヘッド構造体のために1024バイトのスペースが必要です。プログラムのテキスト・セグメントとデータ・セグメントは、この1024バイトの後にのみ配置されます。OGMICタイプの.oモジュールファイルでは、ファイルの先頭にある32バイトのヘッダ構造の後に、コード領域とデータ領域が続きます。

実行ヘッダ構造体のa_textフィールドとa_dataフィールドは、それぞれ読み取り専用のコードセグメントと読み書き可能なデータセグメントのバイト長を示しています。a_bssフィールドは、カーネルがターゲットファイルをロードする際に、データセグメントに続く未初期化データ領域（bssセクション）の長さを示す。Linuxではメモリを確保する際に自動的にゼロにするので、bssセクションはモジュールファイルや実行ファイルに含める必要はない。ターゲットファイルが論理的にbssセクションを持っていることを視覚的に表現するために、後述の図ではターゲットファイルのbssセクションを破線のボックスで表現している。

3.5.23

a_entry フィールドは、プログラムコードが実行を開始するアドレスを指定し、a_syms、a_trsize、a_drsiz フィールドは、それぞれデータセグメント以降のシンボルテーブル、コード、データセグメントの再配置情報のサイズを記述します。シンボルテーブルと再配置情報は実行ファイルには必要ないので、リンカーがデバッグのためにシンボル情報を含めない限り、実行ファイルのフィールドは通常0である。

3.5.24 Relocation information section

Linux 0.12システムのモジュールファイルや実行ファイルは、すべてa.out形式のオブジェクトファイルですが、コンパイラが生成したモジュールファイルのみ、プログラムをリンクするためのリロケーション情報が含まれています。コードセグメントとデータセグメントの再配置情報は、再配置レコード（アイテム）で構成される。各レコードの長さは8バイトである。その構造は以下の通りである。

構造体relocation_info

```
{  
    /* 再配置されるアドレス（セグメント内）。 */ int  
    r_address;  
    /* r_symbolnum の意味は、r_extern に依存する。 */  
    unsigned int r_symbolnum:24;  
    /* 0ではない場合は、pc-relative offsetであることを意味します。  
       と、自分のアドレスの変更のために再配置する必要があります。  
       また、指定されたシンボルやセクションの変更にも対応しています。 */  
    unsigned int r_pcrel:1;  
    /* 再配置されるフィールドの長さ（2の指數として）。  
       したがって、2という値は1<<2バイトを表します。 */
```

```
unsigned int r_length:2;
```

/* 1 => シンボルの値で再配置
r_symbolnum は、ファイルのシンボルテーブルにおけるシンボルのインデックスです。
す。

0 => セグメントのアドレスで再配置。

R_symbolnum は、N_TEXT、N_DATA、N_BSS または N_ABS です。
(N_EXT ビットもセットされることがあります、何の意味もありません)。 */

```
unsigned int r_extern:1;
```

/* 4つのビットは使用されませんが、オブジェクトファイルを書き込む際ににはクリアしておくことが望ましいです。
符号付き整数 r_pad:4:

```
};
```

3525

再配置アイテムには2つの機能があります。1つは、コードセグメントが異なるベースアドレスに再配置されたときに、再配置アイテムを使用して修正が必要な場所を示すことです。2つ目は、モジュールファイルの中に未定義のシンボルへの参照がある場合、リンカーは対応する再配置アイテムを使って、未定義のシンボルが最終的に定義されたときにシンボルの値を修正することができる。上記の再配置レコード・アイテムの構造からわかるように、各レコード・アイテムには、再配置が必要なモジュール・ファイルのコード・セクション（コード・セグメント）とデータ・セクション（データ・セグメント）のアドレスが4バイトの長さで含まれており、ポジショニング・オペレーション情報の計量方法が指定されている。アドレスフィールドr_addressは、再配置可能なアイテムのコードセグメントやデータセグメントの先頭からのオフセット値を参照する。2ビットの長さフィールドr_lengthは、再配置された項目の長さを示し、0～3はそれぞれ、再配置された項目の幅が1バイト、2バイト、4バイト、8バイトであることを示す。フラグr_pcrelは、再配置された項目が「PC関連」の項目であること、つまり命令の中で相対アドレスとして使用されることを示します。外部フラグr_externは、r_symbolnumの意味を制御し、再配置項目がセグメントを参照しているのか、シンボルを参照しているのかを示す。フラグの値が0であれば、再配置エントリは通常の再配置エントリであり、r_symbolnumフィールドは、どのセグメントで位置決めが行われるかを指定する。フラグの値が1の場合は、再配置エントリは外部シンボルへの参照である。この場合、r_symbolnumはターゲットファイルのシンボルテーブルのシンボルを指定し、そのシンボルの値を使って再配置する必要がある。

3526 Symbol Table and String Section

ターゲットファイルの最後の部分は、シンボルテーブルと関連する文字列テーブルです。シンボルテーブルのエントリの構造は以下の通りです。

```
struct nlist {
    union {
        char      *n_name;           // String pointer,
        struct nlist *n_next;       // Or a pointer to another symbolic item structure,
        long       n_strx;          // Or the byte offset value of the symbol name in the table.
    } n_un;
    unsigned char n_type;         // This byte is divided into 3 fields. see a.out.h file.
    char       n_other;          // Usually not used.
    short      n_desc;           //
    unsigned long n_value;        // Symbol's value.
};
```

- 1 GNU gccコンパイラでは、任意の長さの識別子を使用できるため、識別子の文字列は、シンボルテーブルの後にある文字列テーブルに配置されます。シンボルテーブルの各エントリは、12バイトの長さを持ち、最初のフィールドは、文字列テーブルからのシンボル名文字列（ヌル終端）のオフセットを与えます。タイプフィールド n_type は、シンボルのタイプを示します。このフィールドの最後のビットは、シンボルが外部（グローバル）であるかどうかを示すために使用されます。このビットが1であれば、シンボルはグローバルシンボルです。リンカはローカルシンボルの情報を必要としませんが、デバッガはそれを使用することができます。n_typeフィールドの残りのビットは、シンボルタイプを示すために使用されます。a.out.hヘッダーファイルでは、これらのタイプの値の定数シンボルが定義されています。主なシンボルの種類には次のようなものがあります。

- 2 text, data, or bbs indicates the symbols defined in this module file. The value of the symbol at this time

- is the relocatable address of the symbol in the module.
- 3 abs indicates that the symbol is an absolute (fixed) non-relocatable symbol. The value of the symbol is the fixed value.
- 4 undef indicates that it is an undefined symbol in this module file. The value of the symbol at this time is usually 0.

3.5.3 しかし、特殊なケースとして、コンパイラは未定義のシンボルを使って、リンカーに指定されたシンボリック名のためのメモリ空間を確保するように要求することができます。未定義の外部（グローバル）シンボルがゼロ以外の値を持つ場合、リンカーにとってその値は、プログラムがシンボリックアドレス用に指定したいメモリのサイズとなります。リンク処理中に、シンボルが本当に定義されていない場合、リンカーはシンボル名のためのメモリ空間をbssセクションに作成します。このスペースのサイズは、リンクされているすべてのモジュールにおけるシンボルの最大値となります。これがbssセクションでのいわゆるコモンブロックの定義です。これは主に、初期化されていない外部（グローバル）データをサポートするために使用されます。例えば、プログラムで定義された初期化されていない配列などです。シンボルがいずれかのモジュールで既に定義されている場合、リンカーはこの定義を使用し、値を無視します。

3.5.4 Target file format in Linux 0.12

Linux 0.12系では、objdumpコマンドを使って、モジュールファイルや実行ファイルに含まれるファイルのヘッダ構造の具体的な値を見ることができます。例えば、hello.oオブジェクトファイルとその実行ファイルに含まれるヘッダの具体的な値を以下に示します。

```
[/usr/root]# gcc -c -o hello.o hello.c
[/usr/root]# gcc -o hello hello.o
[/usr/root]#
[ /usr/ro 0107 0000 0028 0000 0000 0000 0000 0000
ot]# hexdum
p -x
hello.o
00000000
0000010 0024 0000 0000 0000 0010 0000 0000 0000
0000020 6548 6c6c 2c6f 7720 726f 646c 0a21 0000
0000030 8955 68e5 0000 0000 e3e8 ffff 31ff ebc0
0000040 0003 0000 c3c9 0000 0019 0000 0002 0d00
0000050 0014 0000 0004 0400 0004 0000 0004 0000
0000060 0000 0000 0012 0000 0005 0000 0010 0000
0000070 0018 0000 0001 0000 0000 0000 0020 0000
0000080 6367 5f63 6f63 706d 6c69 6465 002e 6d5f
0000090 6961 006e 705f 6972 746e 0066
000009c
[/usr/root]# objdump -h hello.o
hello.o:
magic: 0x107 (407)マシンタイプ。0 flags: 0x0 text 0x28 data 0x0 bss 0x0 nsyms
3 entry 0x0 trsize 0x10 drsize 0x0
[/usr/root]#
[ /usr/ro 010b 0000 3000 0000 1000 0000 0000 0000
ot]# hexdum
-x hello
more
00000000
0000010 069c 0000 0000 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
```

```
*  
0000400 448b 0824 00a3 0030 e800 001a 0000 006a  
0000410 dbe8 000d eb00 00f9 6548 6c6c 2c6f 7720  
0000420 726f 646c 0a21 0000 8955 68e5 0018 0000  
.....  
--More--q  
[/usr/root]#  
[/usr/root]# objdump -h hello  
hello:  
magic: 0x10b (413)マシンタイプ。 0 flags: 0x0 text 0x3000 data 0x1000 bss 0x0 nsyms  
141 entry 0x0 trsize 0x0 drsize 0x0
```

```
[/usr/root]#
```

hello.oモジュールファイルのマジックナンバーは0407(OMAGIC)であり、ヘッダ構造の直後にコードセグメントがあることがわかります。ヘッダー構造に加えて、長さ0x28バイトのコードセグメントと、長さ0x10バイトの3つのシンボルアイテムとコードセグメントの再配置情報を持つシンボルテーブルが含まれています。それ以外のセグメントの長さは0である。対応する実行ファイルHelloのマジックナンバーは0413 (ZMAGIC) で、コードセグメントはファイルオフセット位置1024バイトから格納されています。コードセグメントの長さは0x3000バイト、データセグメントの長さは0x1000バイトで、シンボルテーブルには141個のアイテムが格納されています。コマンドストリップを使って、実行ファイルのシンボルテーブル情報を削除することができます。例えば、以下では、helloの実行ファイルのシンボル情報を削除しています。これにより、hello実行ファイルのシンボルテーブルの長さが0になり、helloファイルの長さも元の20591バイトから17412バイトに短縮されたことがわかります。

```
[/usr/root]# ll hello
-rwx--x--x 1 root 4096 20591 Nov 14 18:30 hello
[/usr/root]# objdump -h hello
hello
magic: 0x10b (413)マシンタイプ。0flags: 0x0text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0

[/usr/root]# strip hello
[/usr/root]# ll hello
-rwx--x--x 1 root 4096 17412 Nov 14 18:33 hello
[/usr/root]# objdump -h hello
hello
magic: 0x10b (413)マシンタイプ。0flags: 0x0text 0x3000 data 0x1000 bss 0x0 nsyms
0 entry 0x0 trsize 0x0 drsize 0x0
[/usr/root]#
```

図3-8は、ディスク上のプロセス論理アドレス空間におけるa.out実行ファイルの領域の対応を示したものです。Linux 0.12システムにおけるプロセスの論理空間サイズは64MBです。ZMAGICの実行ファイルであるa.outのコード領域は、メモリページサイズの整数倍となります。Linux 0.12 カーネルでは、コードのページを実際に物理メモリページにロードするデマンドページング方式を採用しているため、ロード操作の `fs/execve()` 関数にのみ設定されます。ページディレクトリエンティリとページテーブルエンティリのページングメカニズムなので、デマンドページ技術はロード処理を高速化することができます。

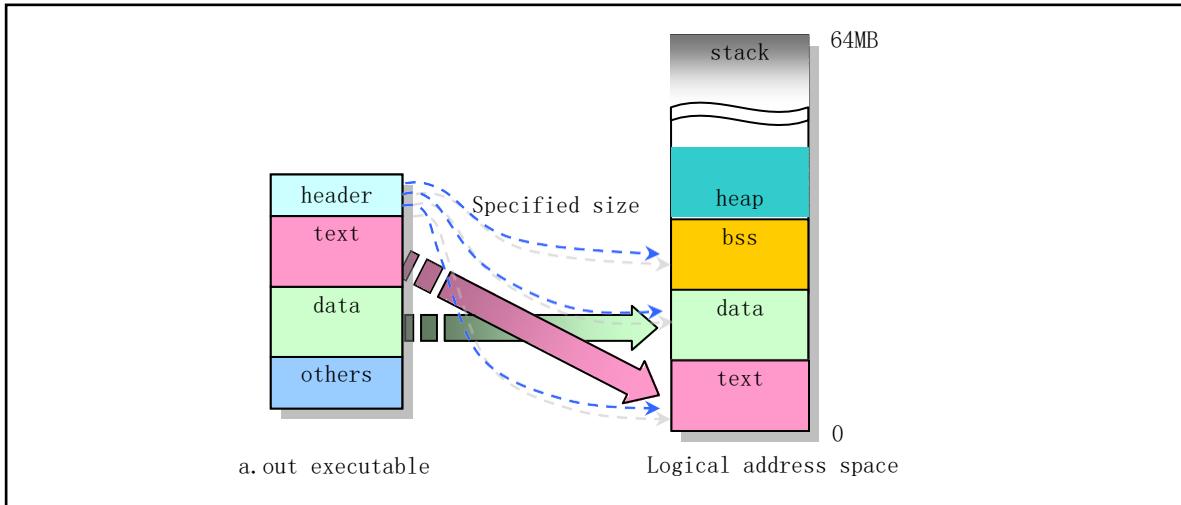


図3-8 a.out実行ファイルのプロセス論理アドレス空間へのマッピング

- 3.5.5** 図中、bssはプロセスの未初期化データ領域で、初期化されていない静的なデータを格納するために使用されます。プログラムの実行開始時には、bssメモリの最初のページがすべて0に設定される。図中のヒープは、ヒープ空間領域であり、実行中のプロセスが動的に要求するメモリ空間を割り当てるために使用される。

3.5.6 Linker output

リンクは、1つまたは複数の入力オブジェクトファイルと関連するライブラリ関数オブジェクトを処理し、最終的に対応するバイナリ実行ファイルまたはすべてのモジュールで構成される大きなモジュールファイルを生成する。この過程で、リンクプログラムの主な仕事は、実行ファイル（または出力モジュールファイル）の記憶領域の割り当て操作を行うことです。格納場所が決まれば、リンクプログラムは引き続きシンボル結合操作やコード修正操作を行うことができる。モジュール・ファイルに定義されているシンボルのほとんどは、ファイル内の格納位置に関連しているため、シンボルの対応位置が決定する前にシンボルを解決する方法はない。

各モジュール・ファイルには、いくつかのタイプのセグメントが含まれています。リンクの2番目のタスクは、すべてのモジュールの同じタイプのセグメントを結合し、出力ファイルの指定されたセグメント・タイプの单一セグメントを形成することです。例えば、リンクはすべての入力モジュール・ファイルのコード・セグメントを1つのセグメントに結合し、それを出力実行ファイルに配置する必要があります。

a.out形式のモジュールファイルは、あらかじめセグメントの種類がわかっているので、リンクはa.out形式のモジュールファイルを簡単に格納・配置することができます。例えば、2つの入力モジュール・ファイルがあり、ライブラリ関数モジュールを接続する必要がある場合、その格納割り当ては図3-9のようになります。各モジュールファイルには、コード部、データ部、bss部があります。また、外部（グローバル）シンボルと思われる共通ブロックがある場合もあります。リンクは、任意のライブラ

リ機能モジュールのコード・セグメント、データ・セグメント、bss・セグメントを含む、各モジュール・ファイルのサイズを収集します。すべてのモジュールが読み込まれて処理された後、0以外の値を持つ未解決の外部シンボルはコモンブロックとして扱われ、その割り当てはbssセクションの最後に格納されます。

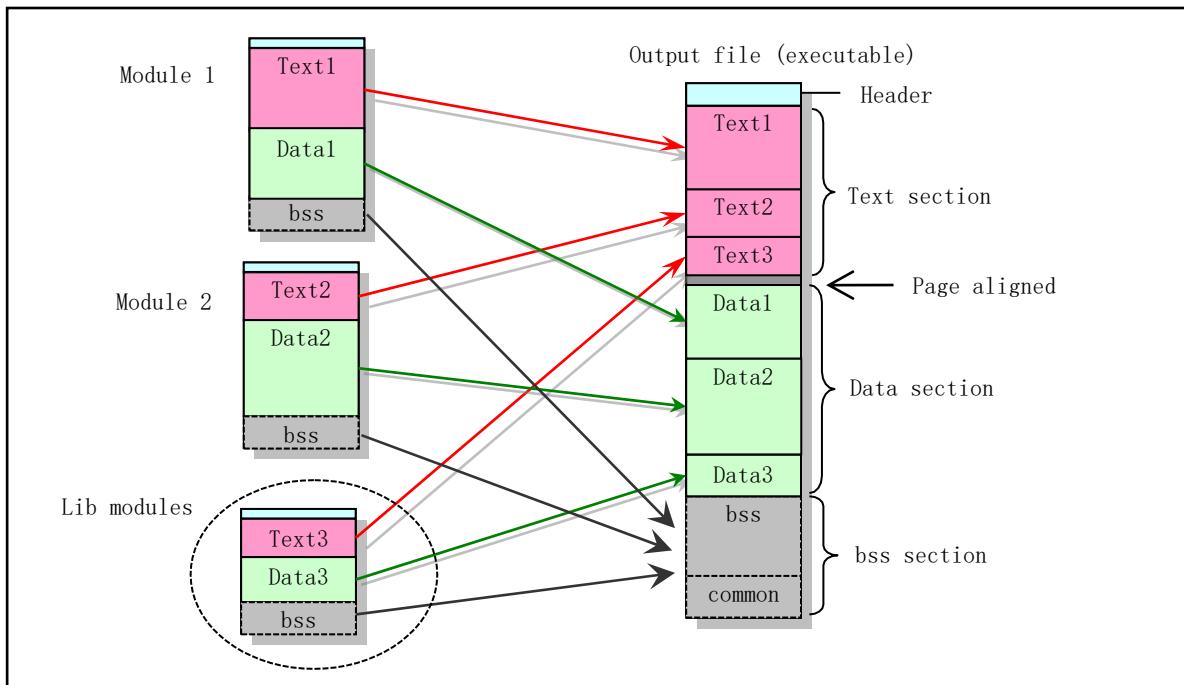


図 3-9 オブジェクトファイルのリンク操作

0.12 これにより、リンクはすべてのセグメントにアドレスを割り当てることができます。
Linuxで使われているZMAGICタイプのa.outフォーマットの場合は

0.13 system, the code segment in the output file is set to start at fixed address 0. The data segment starts from the next page boundary after the code segment. The bss section immediately follows the data segment. Within each segment, the linker stores the same type of segments in the input module file and aligns them by word.

Linux 0.12カーネルが実行ファイルをロードする際には、まずファイルのヘッダ構造の情報に基づいて、そのファイルが適切な実行ファイルであるかどうかを判断します。つまり、マジックナンバータイプがZMAGICであれば、システムはユーザーモードスタックの最上位になります。プログラムは、コマンドラインで入力された環境パラメータやパラメータ情報ブロックを設定し、そのためのタスクデータ構造を構築します。そして、スタックリターンの手法を用いて、いくつかの関連するレジスタ値を設定した後、プログラムを実行します。実行プログラムイメージファイルのコードとデータは、実際に実行または使用されるときに、ロード・オン・デマンドを使用してメモリに動的にロードされます。

3.5.7 Linux 0.12カーネルのコンパイルプロセスでは、カーネル設定ファイルMakefileに基づいてコンパイラとリンクの動作を指令するmakeコマンドを使用しています。ビルドプロセスでは、tools/ディレクトリにあるカーネルのソースコードに含まれるbuild.cプログラムも使用して、すべてのモジュールを結合するための一時的なツールプログラムbuildをコンパイルしてビルドします。カーネルはブートアッププログラムによってROM BIOS割り込みコールを使ってメモリにロードされるため、コンパイルされたカーネルモジュール内の実行ヘッダー構造を削除する必要があります。ユーティリティプログラム

ビルドの主な機能は、bootsect、setup、systemの各ファイルの実行ヘッダ構造を削除し、順次結合してImageという名前のカーネルイメージファイルを作成することです。

3.5.8 Linker Predefined Variables

リンクの際、リンクld、ld86は変数を使って実行プログラムの各セグメントの論理アドレスを記録します。そのため、プログラム上では、外部変数にアクセスすることで、プログラムの中間セグメントの位置を取得することができます。リンクがあらかじめ定義している外部変数は、通常、少なくともetext、_etext、edataです。

_edata、end、_end。

変数名_etextとetextのアドレスはプログラムテキストセグメント終了後の最初のアドレス、_edataとedataのアドレスは初期データ領域後の最初のアドレス、_endとendのアドレスは未初期化データ領域(bss)後の最初のアドレスの位置です。名前の前にアンダースコア'_'が付いているものは、下線が付いているものと同等です。両者の唯一の違いは、etext、edata、endという記号がANSI、POSIXなどの規格で定義されていないことです。

プログラムが実行され始めたばかりの時は、そのbrkの位置は_endと同じ位置にあります。しかし、システムコールsys_brk()や、メモリ割り当て関数malloc()、標準入出力の操作などによって、この位置は変わってしまいます。そのため、プログラムの現在のbrk位置はsbrk()を使って取得する必要があります。なお、これらの変数名はアドレスとして扱う必要があります。そのため、アクセスする際には、&endのようにアドレスのプレフィックス「&」を使用する必要があります。例として

```
extern int _etext;
int et
(int *) et = &_etext;           // et contains the address after the end of text segment.
```

これらの変数のアドレスを表示するには、次のプログラム predef.c を使用します。バンドとアンダースコア「_」の記号でアドレス値が同じであることがわかります。

```
/*
 リンカで定義されたシンボルを表示します。
*/
extern int end, etext, edata; extern
int _etext, _edata, _end; int main()
{
    printf("&etext=%p, &edata=%p, &end=%p\n",
           &etext, &edata, &end);
    printf("&_etext=%p, &_edata=%p,
           &_end=%p\n", &_etext, &_edata,
           &_end);
    0を返す。
}
```

このプログラムをLinux 0.1Xシステムで実行すると、以下のような結果になります。なお、これらのアドレスは、プログラムのアドレス空間における論理的なアドレスであり、実行プログラムがメモリロケーションにロードされたときのアドレスであることに注意してください。

```
[/usr/root]# gcc -o predef predef.c
[/usr/root]# ./predef
&etext=40000088 &edata=4140c8 &end=48d8
```

このプログラムを最新の Linux システム(RedHat 9 以降)で実行すると、次のような結果が得られます。Linuxシステムのプログラムコードは、現在、その論理アドレス0x08048000から格納されていることが

わかっているので、プログラムのコードセグメント長は0x41bバイトであることがわかります。

```
[root@plinux]# ./predef  
_etext=0x804951b _edata=0x804951c _end=0x80495ac
```

- 3.5.9 Linux 0.1x カーネルでは、ブロックデバイスキャッシュ(fs/buffer.c)を初期化する際に、変数名 _end を用いて、メモリ上のカーネルイメージファイル Image の末尾の位置を取得し、この位置から高速設定を行っています。バッファを作成します。

3.5.10 System.map file

- GNUリンクgld(ld)を実行する際、「-M」オプションを使用した場合、またはnmコマンドを使用した場合、リンクマップ情報が標準出力デバイス（通常はスクリーン）に出力されます。リンクが生成するターゲットプログラムのメモリアドレスマップ情報。メモリに読み込まれたプログラム・セグメントの位置情報が記載されている。具体的には以下のようない情報がある。

- Locations of object files and symbol information mapped in memory;
- How public symbols are placed;
- All file members included in the link and their referenced symbols.

通常は、標準出力デバイスに送られてきたリンクイメージ情報を、ファイル（例：System.map）にリダイレクトします。カーネルをコンパイルする際、linux/Makefileファイルで生成された System.map ファイルは、カーネルのシンボルテーブル情報を格納するために使用されます。シンボルテーブルとは、すべてのカーネルシンボルと、それに対応するアドレスのリストです。もちろん、上述の _etext、_edata、_endなどのシンボルのアドレス情報も含まれています。カーネルをコンパイルするたびに、対応する System.map ファイルが新たに生成されます。カーネル内でエラーが発生した場合、System.map ファイルのシンボルテーブルを解析することで、アドレス値に対応する変数名を見つけることができ、またその逆も可能です。

System.map シンボルテーブルファイルを使用することで、カーネルや関連プログラムが故障したときに、より簡単に識別できる情報を得ることができます。シンボルテーブルの例は以下の通りです。

```
c03441a0 B dmi_broken
c03441a4 B is_sony_vaio_laptop
c03441c0 B dmi_ident
c0344204 b pobjtablepresent
```

各行にはシンボルが記述されており、1列目はシンボルの値（アドレス）、2列目はシンボルの種類、シンボルがターゲットファイルのどのセクションにあるのか、またはその属性を示し、3列目は対応するシンボル名です。

2列目のシンボルタイプインジケータは、通常、表3-5に示すタイプがあるが、採用されているターゲットファイルフォーマットに関連するものもある。シンボルタイプが小文字の場合、そのシンボルはローカルであり、大文字の場合、そのシンボルはグローバル（外部）である。ファイルinclude/a.out.h (110-185行目) のnlist{}構造のn_typeフィールドの定義を参照してください。

| 表 3-5 タ ーゲット ファイル のシンボ ルタイプ シンボル リストフ アイル Symbol type | Name | Description |
|--|------|-------------|
| | | |

| | | |
|---|-----------|--|
| A | Absolute | The value of the symbol is an absolute value and will not be changed during further linking. |
| B | BSS | The symbols are in the uninitialized data section, ie in the BSS section. |
| C | Common | The symbol is public. Public symbols are uninitialized data. When linking, multiple public symbols may have the same name. If the symbol is defined elsewhere, the public symbol is treated as an undefined reference. |
| D | Data | The symbol is in the initialized data section. |
| G | Global | The symbol is in the initialized data section of the small object. The format of some object files allows more efficient access to small data objects such as a global integer variable. |
| I | Indirect | A symbol is an indirect reference to another symbol. |
| N | Debugging | The symbol is a debugging symbol. |
| R | Read only | The symbol is in a read-only data section. |
| S | Small | symbol in a small object's uninitialized data section. |
| T | Text | Symbols in code sections. |
| U | Undefined | The symbol is external and its value is 0 (undefined). |
| - | Stabs | The symbol is a stab symbol in the a.out object file and is used to save debugging information. |
| ? | Unknwon | The type of symbol is unknown or related to a specific file format. |

dmi_brokenという変数が、カーネルアドレス0xc03441a0にあることがわかります。

System.mapは、それを使用するソフトウェア（カーネルロギングデーモンklogdなど）が見つけられる場所にあります。システム起動時に、klogdがSystem.mapの場所をパラメータの形で与えられていない場合、klogdは以下の3箇所でSystem.mapを検索します。

/boot/System.map

/System.Map
/usr/src/linux/System.map

カーネル自体は実際にはSystem.mapを使用しませんが、klogd、lsof、psなどの他のプログラムや、dosemuなどのソフトウェアは、正しいSystem.mapファイルを必要とします。このファイルを使用することで、これらのプログラムは既知のメモリアドレスに基づいて対応するカーネル変数名を見つけることができ、カーネルのデバッグが容易になります。

3.6 Make Command and Makefile

上記の例からわかるように、1つまたは数個のソースプログラムから生成される実行ファイルを作成する場合は、数行の簡単なコマンドを入力するだけで済みます。しかし、カーネルのように数百、数千のソースファイルから構成される大規模なプログラムの場合、すべてのコードファイルを手入力でコンパイルするのは非常に煩雑です。makeコマンドは、このような状況を自動的に処理するために設計された最高のツールです。makeコマンドの主な目的は、多数のソースファイルを含む大規模なプロジェクトにおいて、どのファイルを再コンパイルする必要があるかを自動的に判断し、再コンパイルコマンドを発行することです。Makeの使い方を簡単に説明するために、コンパイル用のCプログラムを例に挙げますが、シェルコマンドを使ってコンパイルできるあらゆるプログラミング言語に適用する

ことができます。詳しい使い方については、「GNU makeマニュアル」を参照してください。

makeツールを使用するためには、make実行用のMakefile（またはmakefile）という名前のテキストファイルを書く必要があります。

Makefileには主に、ファイル間の関係や、対応するターゲットファイルを生成するために必要なソースファイルのコンパイルやリンクの操作をMakeに伝えるための実行ルールやコマンドが含まれています。

- 3.6.1 make "がソースファイルを再コンパイルする際には、変更された各ソースファイルが再コンパイルされます。ヘッダーファイルが変更された場合は、そのヘッダーファイルを含む各ソースファイルが再コンパイルされます。各コンパイルにより、ソースファイルに対応するオブジェクトファイルが生成されます。最後に、すべてのソースファイルが再コンパイルされた場合、新しく作成されたものであれ、以前のコンパイルから保存されたものであれ、すべてのオブジェクトファイルがリンクされ、新しい実行ファイルが生成されます。

3.6.2 Contents of the Makefile

- Makefileには、明示的なルール、暗黙のルール、変数定義、ディレクティブ、コメントの5つの要素があります。

- Explicit rules are used to specify when and how to recompile one or more files called rule's targets. The rules explicitly list the other files on the target that depend on the prerequisites (or dependencies), as well as the commands for creating or updating targets.
- Implicit rules are based on the name of the target and the object to determine when and how to recompile one or more files called targets of the rule. This rule describes how the target depends on a file that is similar to the target name and will be given to create or update such a target file.
- Variable definitions define a text string for a variable on one line. This variable can be replaced in subsequent statements. For example, the variables object in the example below defines a list of all .o files.
- A directive is a command of make that indicates the specific operation that it performs when it reads a Makefile. These operations can include reading another makefile; determining whether to use or ignore a portion of the makefile and defining a variable from a string containing multiple lines.
- Comments are the parts of the text in the Makefile that begin with the '#' character. If we really need to use the '#' character, we need to escape it by adding a backslash character ('\#') before the character. Comments can appear anywhere in the Makefile. In addition, a command line script in the Makefile that begins with the TAB is passed to the shell in its entirety, and the shell determines whether it is a command or just a comment.

- 3.6.3 適切なMakefileを書いておけば、ソースコードを修正するたびに「make」と入力するだけで、必要なプログラムの更新をすべて行うことができます。makeは、Makefileの内容とファイルの最終更新時刻から、更新（再コンパイル）が必要なファイルを判断します。更新が必要な各ファイルに対して、Makefileに記録されている関連コマンドを実行する。

3.6.4 Rules in the Makefile File

シンプルなMakefileには、以下のようなルールがいくつか含まれています。これらのルールは、主に操作対象（ソースファイルやオブジェクトファイル）の依存関係を記述するために使用されます。

ターゲット ...: 前提条件 ...

コマンド

...
...

その中でも、ターゲット・オブジェクトは、通常、プログラムが生成するファイルの名前を指し、例えば、実行ファイルや「.o」で終わるオブジェクト・ファイルとなります。また、対象となるアクティビティの名前を指定することもできます。

は、例えば「clean」のように取られます。

前提条件とは、ターゲットを作成するために必要な一連のファイルやその他のターゲットのことです。ターゲットは通常、このような複数の必要ファイルやターゲットファイルに依存します。

コマンド (command) とは、makeが実行する操作のことで、通常はターゲットを生成するシェルコマンドのことを指します。前提条件となる1つ以上のファイルの最終更新時刻がターゲットファイルの最終更新時刻よりも新しい場合、ルールのコマンドが実行されます。また、1つのルールに複数のコマンドが存在することもあり、各コマンドはルール内で1行を占めます。各コマンドを記述する前に、タブ文字を入力 (Tabを押す) する必要があることに注意してください。

通常、コマンドは前提条件を持つルールの中に存在し、前提条件のいずれかが変更されたときにターゲットファイルを作成するために使用されます。ただし、ルールは必ずしも前提条件を持っていなければなりません。たとえば、ターゲット「clean」に関連する削除コマンドを含むルールには、前提条件が含まれていません。

ルールは、ある特定のルールのターゲットである特定のファイルを、いつ、どのように作り直すかを説明するものです。makeは、ターゲットを作成または更新するための前提条件に基づいてコマンドを実行します。ルールは、ある操作をいつどのように行うかを説明することもできます。

3.6.5 Makefileには、ルールのほかにもさまざまなテキストを含めることができます。しかし、シンプルなMakefileであれば、通常、いくつかのルールを含めるだけで十分です。ルールの中には、先に挙げたものよりも複雑なものもありますが、基本的には似たようなものです。

3.6.6 A Simple Makefile

ここでは、8つのCソースファイルと3つのヘッダーファイルからなるテキストエディタプログラムのコンパイルとリンクの方法を説明する簡単なMakefileについて説明します。

Makefileの内容に基づいてmakeがCファイルを再コンパイルする際には、変更されたCファイルのみが再コンパイルされます。もちろん、.hヘッダーファイルが変更された場合には、プログラムが正しくコンパイルされるように、そのヘッダーファイルを含むすべてのCコードファイルが再コンパイルされる。各コンパイル作業により、ソースプログラムに対応するターゲットファイルが生成されます。正味のところ、修正されたソースコードファイルのいずれかがコンパイルされた場合、生成されるすべての.oオブジェクトファイル（ソースコードがコンパイルされる前にコンパイルされたばかりで修正されていないものを含む）をリンクして新しいものを生成する必要があります。実行可能なエディタプログラムです。

Makefileの例題ファイルの内容は、editという名前の実行ファイルが8つのオブジェクト・ファイルにどのように依存しているか、また8つのオブジェクト・ファイルが8つのCソース・ファイルと3つのヘッダ・ファイルにどのように依存しているかを説明しています。この例では、すべてのCファイルに

"defs.h"というヘッダファイルが含まれていますが、editコマンドを定義するCファイルには "command.h" が含まれており、editバッファを変更する下位のCファイルには "buffer.h" が含まれています。"のヘッドファイルです。

```
編集 : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

```
main.o : main.c defs.h
cc -c main.c
kbd.o : kbd.c defs.h command.h cc
        -c kbd.c
command.o : command.c defs.h
        command.h cc -c command.c
display.o : display.c defs.h buffer.h cc -c
        display.c
insert.o : insert.c defs.h buffer.h cc -c
        insert.c
search.o : search.c defs.h
        buffer.h cc -c search.c
```

```

files.o : files.c defs.h buffer.h
          command.h cc -c files.c
utils.o : utils.c defs.h
          cc -c utils.c
綺麗です。rm 編集 main.o kbd.o command.o display.o insert.o search.o files.o utils.o

```

Makefileを使って "edit" の実行ファイルを作成するには、コマンドラインで "make" と入力するだけです。

Makefileを使ってコンパイル済みの実行ファイルとすべてのオブジェクトファイルをカレントディレクトリから削除するには、"make clean" と入力します。

Makefileでは、ルールの対象として、実行ファイル「edit」と、.oオブジェクトファイル「main.o」「kbd.o」などがあります。前提条件（または依存条件）となるファイルは、"main.c" や "defs.h" などのソースファイルです。実際、各「.o」ファイルは、ルールのゴールであると同時に、別のルールに必要な前提条件ファイルでもあることがわかります。コマンドには「cc -c main.c」と「cc -c kbd.c」があります。

ターゲットがファイルの場合、その前提条件にある依存ファイルが変更されると、再コンパイルやリンクが必要になります。もちろん、オブジェクトである前提条件のファイルを先に更新する必要があります。この例では、"edit" は8つの.oターゲットファイルに依存しており、.oターゲットファイル "main.o" はソースファイル "main.c" とヘッダーファイル "defs.h" に依存しています。

Makefileの中のルールのターゲットと前提条件の次の行には、シェルコマンドがあります。これらのシェルコマンドは、前提条件に含まれるファイルを使用して、ターゲットのオブジェクトファイルを更新または生成する方法を示しています。なお、Makefileの中のコマンドラインと他の行を区別するために、各コマンドラインの前にタブを入力する必要があります。makeが行うことは、ターゲットを更新する必要があるときに、ルールの中のコマンドを実行することです。

3.6.7 対象となる「clean」は、ファイルではなく、単なる操作（アクティビティ）の名前です。一般的には、そのルールでこのアクションが実行されることを要求しないため、「clean」は他のルールの前提条件ではありません。その結果、makeはこのルールが明示されていない限り、このルールを実行しません。このルール（ターゲット）は、他のルールの前提条件であるだけでなく、前提条件を含んでおらず、また必要としていることに注意してください。つまり、このルールの唯一の目的は、指定されたコマンドを実行することです。このようなルールの場合、そのターゲットは他のファイルを参照したり依存したりせず、特定の操作を示すだけです。このターゲットをフォニーターゲットと呼びます。

3.6.8 How make handles Makefile

デフォルトでは、makeはMakefileの中の最初のターゲット（'.'で始まるターゲットは含まない）からスタートします。この最初のゴールをMakefileのデフォルトゴールと呼びます。究極のゴールは、タ

ターゲットの更新を試みる努力をすることです。

上の例では、実行プログラム「edit」を更新または作成することがデフォルトの最終目的なので、対応するルールをMakefileの先頭に置いています。コマンドラインでmakeコマンドを入力すると、makeはMakefileを読み、最初のルールの処理を開始する。この例では、最初のルールは再リンクして「edit」を生成することですが、makeがこのルールを完全に処理する前に、まず「edit」が依存しているファイルのルールを処理する必要があります。この例では、まずそれらの.oオブジェクトファイルを作成または更新する必要があります。それぞれの.oファイルは、それぞれのルールに従って処理されます。つまり、それぞれのソースファイルをコンパイルして、それぞれの.oオブジェクトファイルを生成します。ターゲットの前提条件となるソースファイルやヘッダーファイルが、.oオブジェクトファイルよりも新しい場合や、.oオブジェクトファイルが存在しない場合は、対応する.oオブジェクトファイルを更新または作成するために再コンパイルが必要となります。.

Makefileに含まれる他のルールの中にも、そのゴール（ファイル）が最終目標の前提条件に現れていれば処理されます。最終目標（または任意の目標）が他のルールに依存していない場合、私たちが積極的にmakeに処理を要求しない限り、makeはこれらのルールを処理しません。たとえば、makeを実行する際に、Makefileの中の特定のルールのターゲット名をコマンドラインで与えて、指定された更新操作を実行するような

コマンド「make clean」。

.o オブジェクトを再コンパイルする前に、make はまず前提条件、ソースファイル、およびヘッダーファイルの更新を検討します。しかし、Makefileでは、ソースファイルとヘッダーファイルに対する処理が指定されていません。つまり、ソースファイルとヘッダーファイルは、どのルールの対象にもなっていないので、makeはこれらのソースファイルに対して何の処理も行いません。

目的の.oオブジェクトファイルを再コンパイルした後、makeは、更新された編集プログラム "edit"を生成するために、再リンクを行うかどうかを決定します。これは、"edit"が存在しない場合や、対象となる.oオブジェクトファイルが"edit"よりも新しい場合にのみ行われます。もし、.oオブジェクトファイルが再コンパイルされたばかりであれば、"edit"よりも新しいので、makeは新しい"edit"を生成するために再リンクします。

したがって、ファイル "insert.c" を修正してmakeを実行すると、makeはソースファイルをコンパイルして対応する "insert.o" を更新した後、"edit"をリンクする。また、ヘッダーファイル "command.h" を修正してmakeを実行すると、makeは対象ファイル "kbd.o", "command.o", "files.o" を再コンパイルしてからリンクし、新しい実行ファイル "edit"を生成します。

3.6.9 一般的に、make は Makefile の内容を使用して、更新が必要な .o オブジェクトファイルを判断し、その後、更新が必要なターゲットファイルを判断します。.oオブジェクトファイルがその関連ファイルのすべてよりも新しければ、.oオブジェクトはすでに最新の状態であり、それ以上の更新は必要ありません。もちろん、最初の最終ターゲットとしての入力条件（前提条件）にある必要なターゲットはすべて事前に更新されます。

3.6.10 Variables in the Makefile

上記の例では、「edit」ルールですべての.oターゲットファイルを2回リストアップする必要があります（以下参照）。

編集 : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o

このように情報が重複していると、ミスが起こりやすくなります。プログラムに新しい.oオブジェクトファイルを追加した場合、.oオブジェクトファイル名をリストに追加しても、別の場所に追加するのを忘れてしまうことがあります。変数を使えば、このようなミスを減らすことができますし、Makefileをより簡潔に見せることもできます。変数を使うことで、一度定義したテキスト文字列を複数の場所で置き換えることができます。

Makefileでは、すべての.oオブジェクトファイルのリストを表すために、objectsまたはOBJECTSという名前の変数を定義するのが典型的なやり方です。通常、Makefileの中で次のような行を使って変数objectsを定義します。

objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

3.6.11 その後、.oオブジェクトファイルをリストアップする必要があるすべての場所で、変数の値を「\$(objects)」と書くことで置き換えることができます。

3.6.12 Let make automatically deduce commands

各Cソースプログラムをコンパイルするために、ルールの中で関連するコマンドを与える必要はありません。なぜなら、make自身がそれを判断できるからです。makeには暗黙のルールがあり、ターゲットファイルの名前に応じて「cc -c」コマンドを使用し、対応する.cファイルに応じて対応する.oファイルを更新します。例えば、「cc -c main.c -o main.o」というコマンドを使って、「main.c」を「main.o」にコンパイルします。したがって、.oオブジェクトファイルのルールにあるコマンドを省略することができます。このように.cファイルが自動的に使われると、前提条件(依存関係)に自動的に追加されます。そこで、ルールの前提条件で'.c'ファイルを省略することができます --- コマンドも省略したとします。以下は、この2つの変更を含み、変数を使用する完全なMakefileの例です。

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

```
edit : $(objects)
cc -o edit $(objects) main.o :
defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
```

綺麗です。

```
-rm edit $(objects)
```

- 3.6.13 実際にMakefileファイルを書くときはこのようにします。暗黙のルールはとても便利なので、重要です。使っているのをよく見かけます。

3.6.14 Implicit rules in automatic variables

前提条件（依存オブジェクト）の一つがディレクトリを検索して別のディレクトリに見つかった場合、ルールのコマンドはスケジュール通りに実行されます。そのため、コマンドがこのディレクトリで必要な前提条件を検索できるように、慎重に設定する必要があります。これは、自動変数を使用することで実現できます。暗黙のルールである自動変数は、状況に応じてコマンドラインで自動的に置き換えることができる変数です。自動変数の値は、通常のコマンドが実行される前に設定されます。たとえば、自動変数「\$^」の値は、ルールの前提条件をすべて表し、それらが入っているディレクトリの名前も含まれています。「\$<」の値は、ルールの最初の前提条件を表し、「\$@」はターゲットオブジェクトを表します（その他の自動変数については、makeのマニュアルを参照してください）。コマンドラインでヘッダーファイルを指定したくないときには、これらのヘッダーファイルを前提条件に含めることもあります。この時点では、自動変数「\$<」が最初の前提条件となります。

```
foo.o : foo.c defs.h hack.h
cc -c $(CFLAGS) $< -o $@
```

また、「\$<」はfoo.cに、「\$@」はfoo.oに自動的に置き換えられます。

makeがイディオムを使ってターゲットを更新するためには、コマンドを必要としないようにすればいいのである。コマンドを使わずにルールを書くか、ルールを書かないようにします。このとき、makeはソースファイルの種類（ファイルサフィックス）に基づいて、どの暗黙のルールを使うかを判断します。

さらに、サフィックス・ルールと呼ばれる暗黙のルールがあります。これは、makeの暗黙のルールを定義する昔ながらの方法です（現在では、このルールは使われず、代わりに、より一般的で明確なパ

ターン・マッチング・ルールが使われています）。このルールはLinux 0.1xカーネルのMakefileで使われているので、ここでは簡単に説明します。次の例は、ダブルサフィックスルールを適用したもので
す。二重接尾辞ルールは、ソース接尾辞とターゲット接尾辞という接尾辞のペアで定義されます。対
応する暗黙の前提条件は、ファイル名の中のターゲット接尾辞をソース接尾辞に置き換えることで得
られます。したがって、このときの次の「\$<」の値は、「*.c」のファイル名となる。正のmakeルール
の意味は、「*.c」のプログラムを「*.s」のコードにコンパイルすることである。

.c.S:

```
$(cc) $(cflags) ↗ ↗ ↗
-nostdinc -Iinclude -S -o $*.s $<
```

通常、コマンドは前提条件（依存オブジェクト）を持つルールに属しており、前提条件のいずれかが変更されたときにターゲットファイルを生成するために使用されます。しかし、目標に対するコマンドを指定するルールは、必ずしも前提条件を持っていません。例えば、`delete`コマンドでターゲット「clean」に関連するルールは前提条件を必要としない。このとき、ルールは特定のファイルをいつどのようにして再作成するかを説明するものであり、特定のルールの対象となる。Makeは前提条件に基づいてコマンドを実行し、ターゲットを作成または更新します。ルールは、ある操作をいつどのように行うかを説明することもできます。

Makefileにはルール以外のテキストを含めることができます、シンプルなMakefileには適切なルールだけを含める必要があります。

上記のテンプレートよりもはるかに複雑なルールに見えるかもしれません、基本的には統一されています。

Makefile ファイルには、ファイル間で参照される依存関係を含めることができます。これらの依存関係は、ターゲットを再構築する必要があるかどうかを判断するために `make` が使用します。例えば、ヘッダファイルが変更された場合、`make` はそのヘッダファイルに関連するすべての「*.c」ファイルをこれらの依存関係に基づいて再コンパイルします。依存関係の例としては、カーネルソースコードの Makefile を参照してください。

3.7 Summary

本章では、いくつかの実行可能なアセンブリ言語プログラムを記述対象とし、アセンブリ言語としてのas86とGNUの基本的な言語と使用方法について詳細に説明する。また、Linuxカーネルで使用されているC言語の拡張機能についても詳細に説明します。OSを学ぶ上で、システムがサポートするオブジェクトファイルの構造は非常に重要な役割を果たします。本章では、Linux 0.12で使用されている `a.out` オブジェクトファイル形式について詳しく説明します。

次の章では、プロテクトモードで動作するIntel 80X86プロセッサの動作原理について詳しく説明します。保護モードのマルチタスクプログラムの例がありますが、この例を読むことで、オペレーティングシステムが最初にどのように「回転」するのかを基本的に理解することができ、Linux 0.12カーネルの全ソースコードを読み続けるための確かな基礎を築くことができます。

4 80X86 Protection Mode and Its Programming

本書で紹介するLinux OSは、インテル80X86プロセッサーと関連する周辺ハードウェアで構成されたPCシステムをベースにしています。80X86 CPUシステムのプログラミングについては、もちろんインテル社から発売されている全3巻の『IA-32 インテル・アーキテクチャー ソフトウェア開発者マニュアル』、特に第3巻の「システム・プログラミング・ガイド」が最適です。特に第3巻の「システム・プログラミング・ガイド」は、80X86のシステム・プログラミングを理解する上で欠かせない資料です。80X86 CPUの動作原理を理解したり、システムプログラミングを行う上で欠かせない参考書です。これらの情報は、インテル社のホームページから無料でダウンロードできます。本章では、主に80X86 CPUのアーキテクチャと、プロテクトモードでのプログラミングの基礎知識について説明し、80X86 CPUをベースとしたLinuxカーネルのソースコードを読む準備をするための基礎固めを行います。主な内容は以下の通りです。1.80X86 CPUの基礎知識、2.プロテクトモードのメモリ管理、3.様々なCPU保護方法、4.割り込みと例外処理、5.タスク管理、6.プロテクトモードプログラミングの初期化、7.簡単なマルチタスクカーネルの例。

本章の最後のセクションで説明したシンプルなマルチタスクのカーネルプログラムは、Linux 0.12カーネルをベースにした簡略化された例です。この例では、メモリセグメンテーション管理とタスク管理の実装を説明しています。ページング機構の内容は含まれていません。しかし、この例題の動作メカニズムを十分に理解しておけば、後にLinuxカーネルのソースコードを読んだときに大きな問題が発生することはありません。この部分の内容に慣れている方は、本章の最後に掲載されている実行可能なカーネルのサンプルプログラムを直接読むことができます。もちろん、カーネルのソースコードを読むときには、いつでも本章を参照することができます。

4.1 80X86 System Registers and System Instructions

4.1.1 80X86では、プロセッサの初期化や制御システムの動作を支援するために、フラグレジスタEFLAGSと、いくつかのシステムレジスタが用意されています。EFLAGSには、一般的なステータスフラグに加えて、いくつかのシステムフラグがあります。これらのシステムフラグは、タスクの切り替え、割り込み処理、命令の追跡、アクセス許可などの制御に使用されます。システムレジスタは、メモリ管理やプロセッサの動作制御に使用されます。セグメント化やページング処理のためのシステムテーブルのベースアドレスや、プロセッサの動作を制御するビットフラグが含まれています。

4.1.2 Flag Registers

フラグレジスタEFLAGSのシステムフラグとIOPLフィールドは、図4-1に示すように、I/Oアクセス、マスク可能なハードウェア割り込み、デバッグ、タスク切り替え、仮想8086モードの制御に使用されます。通常、これらのフラグを変更できるのはオペレーティング・システム・コードのみです。

EFLAGSの他のフラグは、いくつかの一般的なフラグ（キャリーCF、パリティPF、補助キャリーAF、ゼロフラグZF、ネガティブSF、方向DF、オーバーフローOF）です。ここでは、チームEFLAGSのシステムフラグについてのみ説明します。

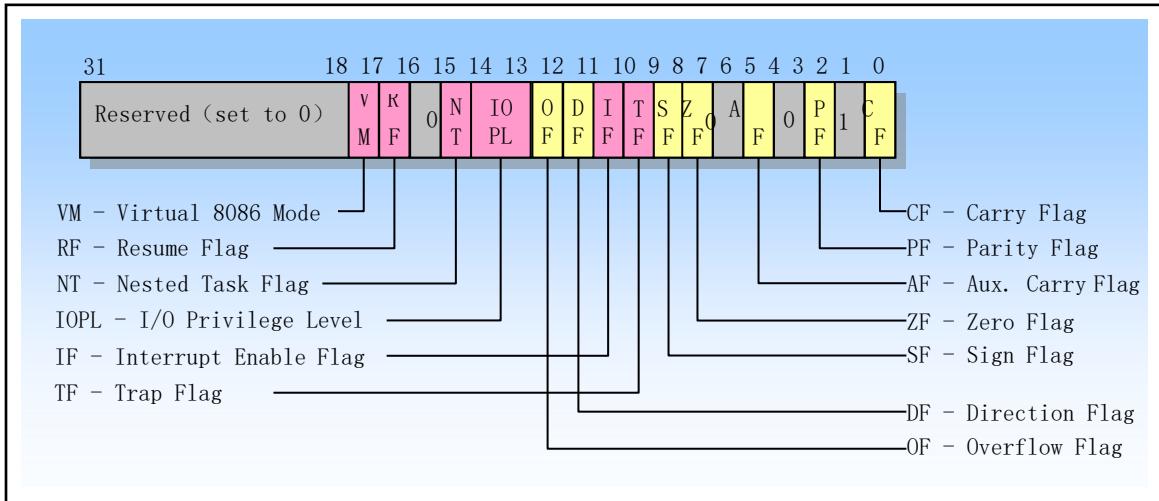


図4-1 EFLAGSのシステムフラグ

TF Bit 8 is the Trap Flag. When this bit is set, step-by-step execution can be started for the debug operation; single-step execution is prohibited when reset. In the single-step execution mode, the processor generates a debug exception after each instruction is executed, so that we can observe the state of the execute program after executing each instruction. If the program uses the POPF, POPFD, or IRET instructions to set the TF flag, the processor generates a debug exception after the subsequent instruction.

IOPLビット13～12は、I/O特権レベルフィールドです。このフィールドは、現在実行中のプログラムまたはタスクのI/O特権レベルIOPLを示します。現在プログラムやタスクを実行しているCPLがこのIOPL以下でないと、I/Oアドレス空間にアクセスできません。CPLが特権レベル0の場合のみ、プログラムはPOPF命令やIRET命令を使ってこのフィールドを変更することができます。IOPLは、IFロゴの変更を制御するメカニズムの1つでもある。

NTビット14は、入れ子になったタスクのフラグです。割り込まれたタスクと呼び出されたタスクの連鎖関係を制御します。プロセッサは、CALL命令、割込み、例外で開始されたタスクへの呼び出し時にこのフラグを設定します。IRET命令を使用してタスクから復帰する際、プロセッサはこのNTフラグをチェックして変更します。このフラグはPOPF/POPFD命令でも変更できますが、アプリケーションでこのフラグの状態を変更すると予期せぬ例外が発生する可能性があります。

RF Bit 16 is the Resume Flag. This flag controls the processor's response to breakpoint instructions. When set, this flag temporarily disables the debug exception generated by the breakpoint instruction; when the flag is reset, the breakpoint instruction will generate an exception. The main function of the RF flag is to allow re-execution of an instruction after debugging an exception. Before the debug software uses the IRETD instruction to return to the interrupted program, the RF flag in the EFLAGS content on the stack needs to be set to prevent the instruction breakpoint from causing another exception. The processor automatically clears the flag after the instruction returns, again allowing instruction breakpoint exceptions.

4.1.3 VMビット17は、Virtual-8086 Modeフラグです。このフラグがセットされていると、仮想

8086モードがオンになり、リセットされるとプロテクトモードに戻ります。

4.1.4 Memory Management Registers

プロセッサには、4つのメモリ管理レジスタ（GDTR, LDTR, IDTR, TR）が用意されています。

図4-2に示すように、メモリセグメントの管理に使用されるシステムテーブルのベースアドレスです。プロセッサは、これらのレジスタをロードおよびセーブするための特定の命令を提供します。システムテーブルの役割については、次項の「保護モードのメモリ管理」で詳しく説明しています。

| | |
|----------------------------|--------------------|
| 32-bit Linear Base Address | 16-bit Table Limit |
| 32-bit Linear Base Address | 16-bit Table Limit |

| | | |
|----------------------------|------------|--|
| 32-bit Linear Base Address | Seg. Limit | |
| 32-bit Linear Base Address | Seg. Limit | |

図4-2 メモリ管理レジスタ

1. GDTR、LDTR、IDTR、TRは、セグメント・ベース・レジスタで、セグメント・メカニズムのための重要な情報テーブルを含んでいます。GDTR、IDTR、LDTRは、ディスクリプターテーブルが格納されているセグメントのアドレスに使用されます。TRは、特別なタスクステートセグメントTSS (Task State Segment) のアドレスに使用されます。TSSセグメントには、現在実行中のタスクに関する重要な情報が格納されています。ここでは、それらについて詳しく説明します。
2. Global Descriptor Table Register (GDTR)
3. GDTRレジスタは、グローバルディスクリプターテーブルGDTの32ビットリニアベースアドレスと16ビットリミット値を保持しています。ベースアドレスは、リニアアドレス空間におけるGDTテーブルのバイト0のアドレスを指定し、テーブルレンジスはGDTテーブルのバイトレンジス値を示す。LGDT命令とSGDT命令は、それぞれGDTRレジスタの内容をロードおよびセットするために使用されます。マシンの電源投入直後やプロセッサのリセット後は、デフォルトでベース・アドレスが0に設定され、長さの値は0xFFFFに設定されています。保護モードの初期化時には、GDTRに新しい値をロードする必要があります。
4. Interrupt Descriptor Table Register (IDTR)
5. IDTRレジスタは、GDTRと同様に、割り込みディスクリプターテーブルIDTの32ビットリニアベースアドレスと16ビットテーブル長の値を格納するために使用されます。LIDT命令とSIDT命令は、それぞれIDTRレジスタの内容をロードおよびセットするために使用されます。マシンの電源投入直後やプロセッサのリセット後は、デフォルトでベースアドレスが0に、長さの値が0xFFFFに設定されています。
6. Local Descriptor Table Register (LDTR)
7. LDTRレジスタは、ローカルディスクリプターテーブルLDTの16ビットセグメントセレクタ、32ビットリニアベースアドレス、16ビットセグメントリミット、およびディスクリプタ属性値を保持しています。LLDT命令とSLDT命令は、それぞれLDTRレジスタのセグメントセレクタ部分のロードとストアに使用されます。LDTテーブルを含むセグメントには、GDTテーブルにセグメントディスクリプターのエントリがなければなりません。LLDT命令を使用してLDTセグメントを含むセレクタをLDTRにロードする場合、LDTセグメント記述子のセグメントベースアドレス、セグメント長、記述子属性が自動的にLDTRにロードされます。タスクが切り替わると、プロセッサは新しいタスクのLDTのセグメ

ントセレクタとセグメントディスクリプタを自動的にLDTRにロードします。マシンのパワー・アップまたはプロセッサのリセット後、セグメント・セレクタとベース・アドレスはデフォルトで0に設定され、セグメント長は0xFFFFに設定されます。

8. Task Register (TR)

TRレジスタには、16ビットのセグメントセレクタ、32ビットのベースアドレス、16ビットのセグメント長、ディスクリプターが格納されています。

4.1.5 現在のタスクTSSセグメントの属性値を表示します。GDTテーブル内のTSSタイプディスクリプターを参照します。LTR命令とSTR命令は、それぞれTRレジスタのセグメントセレクタ部分のロードとセーブに使用されます。LTR命令でセレクタをタスク・レジスタにロードすると、TSS記述子のセグメント・ベース・アドレス、セグメント長、記述子の属性が自動的にタスク・レジスタにロードされます。タスク・スイッチングが行われると、プロセッサは新しいタスクのTSSのセグメント・セレクタとセグメント・ディスクリプタを自動的にタスク・レジスタTRにロードします。

4.1.6 Control Registers

制御レジスタ (CR0, CR1, CR2, CR3) は、図4-3に示すように、プロセッサの動作モードや現在実行中のタスクの特性を制御・決定するために使用されます。CR0には、プロセッサの動作モードや状態を制御するシステム制御フラグが格納されており、CR1は使用予約されています。CR2には、ページフォルトを引き起こすリニアアドレスが格納されています。CR3にはページディレクトリテーブルの物理メモリベースアドレスが含まれているため、このレジスタはPDBR (Page-Directory Base Address Register) とも呼ばれます。

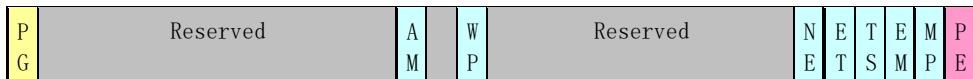


図4-3 コントロールレジスタ CR0--CR3

1. Coprocessor control bits in CR0

CR0のET (Extended Type Bit) , TS (Task Switching Bit) , EM (Emulation Bit) , MP (Math Presence Bit) の4ビットは、80X86の浮動小数点 (Math) コプロセッサの動作を制御するために使用されます。コプロセッサの詳細については、第11章を参照してください。CR0のETビット (フラグ) は、コプロセッサとの通信に使用されるプロトコルを選択するために使用され、システムが80387と80287のどちらのコプロセッサを使用しているかを示します。TS、MP、EMビットは、フロート命令やWAIT命令でDevice Not Available例外を発生させるかどうかの判断に使用されます。この例外は、浮動小数点演算を使用するタスクでのみ、浮動小数点レジスタの保存と復元に使用できます。浮動小数点演算を使用しないタスクでは、そうすることで切り替えを高速化することができます。

ET Bit 4 of CR0 is an Extension Type flag. When the flag is 1, it indicates that the system has a 80387 coprocessor and uses a 32-bit coprocessor protocol. ET=0 indicates use of the 80287 coprocessor. If simulation bit EM=1, this bit will be ignored. During a processor reset operation, the ET bit is initialized

to indicate the type of coprocessor used in the system. If there is 80387 in the system, then ET is set to 1, otherwise if there is a 80287 or no coprocessor, ET is set to 0.

TS Bit 3 of CR0 is the Task Switched flag. This flag is used to postpone saving the coprocessor content of

は、新しいタスクが実際にコプロセッサ命令を実行し始めるまでの間、タスクの切り替えを行います。このフラグは、タスクが切り替わるたびにプロセッサによって設定され、コプロセッサ命令が実行されるときにテストされます。

TSフラグが設定され、CR0のEMフラグが0の場合、コプロセッサ命令が実行される前にDNA (Device Not Available) 例外が発生します。TSフラグが設定されていても、CR0のMP0フラグとEMフラグが設定されていない場合は、コプロセッサ命令WAIT/FWAITが実行されるまで、デバイス例外は発生しません。EMフラグがセットされている場合、TSフラグはコプロセッサ命令の実行に影響しません。表4-1参照。

タスクが切り替わると、プロセッサはコプロセッサのコンテキストを自動的に保存せず、TSフラグを設定します。このフラグは、新しいタスクのストリームを実行中に、いつでもコプロセッサの命令に遭遇したときに、プロセッサにデバイス実在性例外を発生させます。デバイス・エクシスタンント・ハンドラは、CLTS命令を使ってTSフラグをクリアし、コプロセッサのコンテキストを保存することができます。タスクがコプロセッサを使用したことがない場合、対応するコプロセッサのコンテキストを保存する必要はありません。

CR0のEMビット2は、EMulationフラグです。このビットがセットされていると、プロセッサに内部または外部のコプロセッサがないことを意味します。コプロセッサ命令を実行すると、device-not-available例外が発生します。クリアすると、システムにコプロセッサがあることを意味します。このフラグを設定すると、すべての浮動小数点命令をソフトウェアでシミュレートすることになります。

MP CR0のビット1は、Monitor Coprocessor or Math Presentフラグです。WAIT/FWAIT命令とTSフラグの相互作用を制御するために使用します。MP=1、TS=1の場合、WAIT命令を実行するとdevice-not-available例外が発生します。MP=0の場合、TSフラグはWAITの実行には影響しません。

| 表4-1 CR0のEM、MP、TSの組み合わせによるコプロセッサの動作への影響 | | | Instruction Type | |
|---|----|----|------------------|---------------|
| CR0 Flags | | | Floating-Point | WAIT/FWAIT |
| EM | MP | TS | | |
| 0 | 0 | 0 | Execute | Execute |
| 0 | 0 | 1 | DNA Exception | Execute |
| 0 | 1 | 0 | Execute | Execute |
| 0 | 1 | 1 | DNA Exception | DNA Exception |
| 1 | 0 | 0 | DNA Exception | Execute |
| 1 | 0 | 1 | DNA Exception | Execute |

| | | | | |
|---|---|---|---------------|---------------|
| 1 | 1 | 0 | DNA Exception | Execute |
| 1 | 1 | 1 | DNA Exception | DNA Exception |

2. Protection and Control bits in CR0

PE Bit 0 of CR0 is the Protection Enable flag. When this bit is set, the protection mode is enabled; when reset, real address mode is entered. This flag only enables segment-level protection and does not enable paging. To enable the paging mechanism, both the PE and PG flags are set.

PG CR0のビット31は、ページングシグネチャです。このビットがセットされると、ページング機構が有効になり、リセットされると、ページング機構が無効になります。このとき、すべてのリニアアドレスは物理アドレスと同等です。このフラグをオンにする前に、PEフラグをオンにする必要があります。つまり、ページング機構を有効にするには、PEフラグとPGフラグの両方をセットする必要があります。

WP Intel 80486以上のCPUでは、CR0のビット16がWrite Protectフラグとなります。このフラグがセットされている場合、プロセッサはスーパーユーザープログラム（例：特権レベル0のプログラム）による書き込みを禁止します。

このビットがリセットされると、逆にユーザーレベルの読み取り専用ページへの操作になります。このフラグは、UNIX系OSがプロセス作成時にCopy on Write技術を実装する際に有益です。

NE Intel 80486以上のCPUの場合、CR0のビット5は、Numeric Errorフラグです。このフラグがセットされていると、X87コプロセッサの内部エラー報告機構が有効になり、フラグがリセットされていると、PCの形をしたX87コプロセッサのエラー報告機構が使われます。NEがリセット状態で、CPUのIGNNE入力端子に信号がある場合は、数学コプロセッサのX87エラーは無視されます。NEがリセット状態で、CPUのIGNNE入力端子に信号がない場合、マスクされていない数学コプロセッサのX87エラーが発生すると、プロセッサはFERR端子を介して外部に割り込みを発生させ、次の待機形式の浮動小数点命令直前に命令実行を停止するか、WAIT/FWAIT命令を実行します。CPUのFERR端子は、外部のコプロセッサ80387のERROR端子をエミュレートするため、通常は割り込みコントローラの入力要求端子に接続されています。NEフラグ、IGNNE端子、FERR端子は、外部ロジックによるPC形式の外部エラー報告機構を実現するためのものです。

PE (Enable Protected) ビット（ビット0）とPagingビット（ビット31）は、それぞれセグメンテーションとページングのメカニズムを制御するために使用されます。PEはセグメンテーション機構の制御に使用されます。PE=1の場合、プロセッサはオープン・セグメンテーション・メカニズムのコンテキストで動作します（すなわち、プロテクト・モードで動作します）。PE=0の場合、プロセッサはセグメンテーション機構をオフにし、8086のように実アドレスモードで動作します。PGはページング機構の制御に使用されます。PG=1の場合、ページング機構がオンになります。PG=0の場合、ページング機構は無効となり、リニアアドレスが物理アドレスとして直接使用されます。

PE=0, PG=0の場合、プロセッサは実アドレスモードで動作します。PG=0, PE=1の場合、プロセッサはページング機構を持たない保護モードで動作します。PG=1, PE=0の場合、この保護モードではないため、ページング機構を有効にすることができず、プロセッサは一般保護例外を生成します。このフラグの組み合わせは無効である。PG=1, PE=1の場合、プロセッサはページング機構を有効にした保護モードで動作する。.

PEビットとPGビットの変更には注意が必要です。PGビットの設定を変更できるのは、実行プログラムがリニアアドレス空間と物理アドレス空間のコードとデータの少なくとも一部が同一アドレスになっているときだけです。この時点では、この同一アドレスのコードの一部が、ページングされた世界と非ページングされた世界の橋渡しの役割を果たします。ページング機構がオンになっているかどうかにかかわらず、この部分のコードは同じアドレスになります。また、ページングが有効 (PG=1) になる前に、ページキャッシュのTLBをリフレッシュする必要があります。

PEビットを変更した後、プログラムは直ちにジャンプ命令を使用して、プロセッサの実行パイプライン内で異なるモードを取得した命令をフラッシュしなければなりません。PEビットを設定する前に、プログラムはいくつかのシステムセグメントとコントロールレジスタを初期化する必要があります。

電源投入時、プロセッサはPE=0、PG=0（リアルモード状態）にリセットされ、ブートコードがこれらのレジスタやデータ構造を初期化してから、セグメント化とページングのメカニズムを有効にします。

3. CR2 and CR3

CR2とCR3はページング機構に使用されます。CR3には、ページディレクトリテーブルページの物理アドレスが格納されているため、CR3はPDBRとも呼ばれます。ページディレクトリテーブルページはページアラインされているため、このレジスタの上位20ビットのみが有効です。下位12ビットは、より高度なプロセッサで使用するために予約されているため、CR3に新しい値をロードする際には、下位12ビットを0に設定する必要があります。

CR2は、ページの例外が発生したときにエラーメッセージを報告するために使用されます。報告ページが異常な場合、プロセッサは例外が発生したリニア・アドレスをCR2に格納します。そのため、OSのページ例外ハンドラは、CR2の内容を確認することで、リニアアドレス空間のどのページで例外が発生したかを判断することができます。

CR3をロードするためにMOV命令を使用すると、ページキャッシングが無効になるという副作用があります。アドレス変換に必要なバスサイクル数を削減するために、最も最近アクセスされたページディレクトリとページテーブルは、TLB (Translation Lookaside Buffer) と呼ばれるプロセッサのページキャッシングに格納されます。ページテーブルエントリは、TLBに必要なページテーブルエントリが含まれていない場合にのみ、余分なバスサイクルを使用してメモリから読み込まれます。

- 4.1.7 CR0 の PG ビットがリセット状態 (PG = 0) であっても、先に CR3 をロードすることで、ページング機構を初期化することができます。タスクを切り替えると、CR3 の内容も変化します。しかし、新しいタスクのCR3の値が元のタスクの値と同じであれば、プロセッサはページキャッシングをリフレッシュする必要はありません。これにより、ページテーブルを共有するタスクの実行速度が向上します。

4.1.8 System Instructions

システム命令は、システムレベルの機能を処理するために使用される。例えば、システムレジスタのロード、割り込みの管理などです。ほとんどのシステム命令は、特権レベル0のOSソフトウェアのみが実行できます。残りの命令は、アプリケーションが使用できるように、どの特権レベルでも実行することができます。表4-2は、使用するシステム命令の一部を示しています。また、それらが保護されているかどうかも示しています。

| 表4-2 よく使われるシステム命令の一覧 Instruction | Description | Protected? | Description |
|-------------------------------------|------------------------|------------|--|
| LLDT | Load LDT Register | Yes | Load LDT segment selectors and segment descriptors from memory into the LDTR register. |
| SLDT | Store LDT Register | No | Save the LDT segment selector in LDTR to internal memory or general-purpose registers. |
| LGDT | Load GDT Register | Yes | Load the base address and length of the GDT table from memory into GDTR. |
| SGDT | Store GDT Register | No | Save the base address and length of the IDT table in GDTR to memory. |
| LTR | Load Task Register | Yes | Load TSS segment selectors (and segment descriptors) into the task register. |
| STR | Store Task Register | No | Save the current task TSS segment selector in TR to the memory or general register. |
| LIDT | Load IDT Register | Yes | The base address and length of the IDT table are loaded from memory into the IDTR. |
| SIDT | Store IDT Register | No | Store the base address and length of the IDT table in IDTR in memory. |
| MOV CRn | Move Control Registers | Yes | Load and save control registers CR0, CR1, CR2, or CR3. |

| | | | |
|------|--------------------------|-----|--|
| LMSW | Load Machine State Word | Yes | Load the machine status word (corresponds to CR0 bit 15--0). This instruction is for compatibility with the 80286 processor. |
| SMSW | Store Machine State Word | No | Save the machine status word. This instruction is for compatibility with the 80286 processor. |
| CLTS | Clear TS flag | Yes | Clears the task switched flag TS in CR0. There are no exceptions for handling devices (coprocessors). |
| LSL | Load Segment Limit | No | Load Segment Limit |
| HLT | Halt Processor | Yes | Stop the processor execution. |

4.2 Protect Mode Memory Management

4.2.1 ここでは、メモリアドレッシングの定義、論理アドレス、リニアアドレス、物理アドレス間の変換原理に対するセグメンテーションとページングメカニズムの使用、タスクと特権レベル間の保護メカニズムについて簡単に紹介します。続くサブセクションでは、各パートの動作原理を詳しく説明します。

4.2.2 Memory Addressing

メモリとは、順番に並んだバイトの配列のこととで、各バイトは固有のメモリアドレスを持つ。メモリアドレッシングとは、メモリに格納されている指定されたデータオブジェクトのアドレスを特定すること。ここでいうデータオブジェクトとは、メモリに格納されている指定されたデータタイプの数値や文字列のことである。80X86は複数のデータタイプをサポートしている。80X86では、1バイト、2バイト（1ワード）、4バイト（ダブルワード、ロングワード）の符号なし整数や符号付き整数、マルチバイトの文字列など、複数のデータ型をサポートしています。通常、バイト内のあるビットの位置やアドレスは、バイト単位で指定できるため、最小のデータタイプのアドレス指定は、1バイトデータ（数値や文字）の位置指定となる。通常、メモリのアドレスは0から指定しますが、80X86 CPUの場合、アドレスバス幅が32ビットなので、物理アドレスは全部で 2^{32} 種類あります。つまり、メモリの物理アドレス空間は4Gあるので、合計4Gバイトの物理メモリをアドレス指定できます。マルチバイトのデータタイプ（2バイトの整数データタイプなど）の場合、このバイトはメモリに格納されます。80X86は、まず低値バイトを格納し、次に高値バイトをアドレスに格納します。したがって、80X86のCPUはスモールエンダムプロセッサです。

80X86 CPUの場合、1つの命令は主にオペコードとオペランドで構成されています。オペランドは、レジスタまたはメモリ上に配置することができます。オペランドをメモリ上に置くためには、メモリアドレッシングが必要です。80X86では、メモリアドレッシングを伴う命令オペランドが多く、また、アドレッシングされるデータの種類に応じて、さまざまなアドレッシング方式があります。メモリアドレッシングには、80X86では「セグメント」と呼ばれるアドレッシング手法を採用しています。メモリ上のデータオブジェクトをアドレス指定するには、セグメントの開始アドレス（セグメントアドレス）と、セグメント内のオフセットアドレスが必要です。セグメントアドレスの部分は16ビットのセグメントセレクタで指定し、そのうち14ビットで 2^{14} 乗、つまり16384個のセグメントを選択できる。セグメントアドレス部は16ビットのセグメントセレクタで指定し、そのうち14ビットで 2^{14} 乗、つまり16384個のセグメントを選択できる。セグメント内オフセットアドレス部は32ビットで指定し、セグメント内アドレスは0～4Gとなる。つまり、1つのセグメントの最大長は4Gになります。このように、16ビットのセグメントセレクタと32ビットのセグメント内オフセットで構成される48ビットのアドレスまたはロングポインタが、論理アドレス（仮想アドレス）を形成します。これにより、データ・オ

プロジェクトのセグメント・アドレスとセグメント・オフセット・アドレスが一意に決まります。32ビットのオフセットアドレスやポインターのみで指定されたアドレスは、現在のセグメントのオブジェクトアドレスに基づいています。また、セグメンテーション・メカニズムでは、セグメントをタイプ分けして、特定のタイプのセグメントで実行できる操作を制限することができます。

80X86には、セグメントセレクタを格納するための6つのセグメントレジスタが用意されています。CS,DS,ES,SS,FS,GSです。CSは常にコードセグメントのアドレスに使用され、スタックセグメントは特にSSセグメントレジスタを使用します。CSで指定されたセグメントを現在のコードセグメントと呼びます。このとき、EIPレジスタには、実行するカレントコードセグメント内のセグメント内のオフセットアドレスが格納されています。したがって、実行すべき命令のアドレスは、CS:[EIP]と表現できる。後述するセグメント間制御分岐命令を用いて、CSとEIPに新たな値を割り当てることで、実行位置を他のコードセグメントに変更することができ、異なるセグメントのプログラムの制御移行を実現することができます。

セグメント・レジスタSSで指定されたセグメントは、現在のスタック・セグメントと呼ばれます。スタックの最上位は、ESPレジスタの内容で指定されます。つまり、スタックの一番上のアドレスはSS:[ESP]です。他の4つのセグメント・レジスタは、一般的なセグメント・レジスタです。命令で演算するデータのセグメントが指定されていない場合は、DSがデフォルトのデータ・セグメント・レジスタになります。

メモリオペランドのセグメント内オフセットアドレスを指定するために、80X86命令ではオフセットの計算方法を多数規定しており、これを命令アドレッシングと呼ぶ。命令のオフセットは、ベースアドレスレジスタ、インデックスレジスタ、そしてオフセット定数の3つの部分から構成されています。というものです。

$$\text{オフセットアドレス} = \text{ベースアドレス} + (\text{インデックス} \times \text{スケールファクタ}) + \text{オフセット}$$

4.2.3 Address Translation

完全なメモリ管理システムには、保護とアドレス変換という2つの重要な部分があります。プロテクションを提供することで、あるタスクが他のタスクやオペレーティングシステムのメモリ領域にアクセスすることを防ぎます。アドレス変換は、オペレーティングシステムが柔軟にタスクにメモリを割り当てる可能性を可能にします。また、特定の物理アドレスを任意の論理アドレスでマッピングされないようにすることができるため、アドレス変換の際にメモリ保護が行われます。

前述したように、コンピュータ内の物理メモリはバイトの直線的な配列で、各バイトは固有の物理アドレスを持っています。プログラム内のアドレスは、セグメントセレクタとセグメント内のオフセットからなる論理アドレスです。このような論理アドレスは、直接物理メモリにアクセスすることはできず、アドレス変換機構を用いて物理メモリのアドレスに変換・マッピングする必要があります。この論理アドレスを物理メモリのアドレスに変換するのが、メモリ管理機構である。

アドレス変換の決定に必要な情報を減らすために、変換やマッピングは通常、メモリブロックを操作単位として使用します。アドレス変換の手法としては、セグメンテーション機構とページング機構が広く使われています。両者の違いは、論理アドレスをマップされたメモリブロックに整理する方法、変換情報の指定方法、プログラマの操作方法などです。フラグメントとページングは、それぞれの変換情報をメモリ上のテーブルで指定します。これらのテーブルは、アプリケーションによる不正な変更を防ぐために、OSからしかアクセスできないようになっている。

80X86では、図4-4に示すように、論理アドレスから物理アドレスへの変換にセグメンテーションとページングを使用しています。第1段階では、セグメンテーション機構を用いて、論理アドレスをプロセッサのリニアアドレス空間のアドレスに変換します。第2段階では、ページング機構を用いてリニアアドレスを物理アドレスに変換します。アドレス変換処理では、第1段階のセグメンテーション機構は常に使用され、第2段階のページング機構はオプションです。ページング機構を使用しない場合、セグメンテーション機構で生成されたリニアアドレス空間は、プロセッサの物理アドレス空間に直接マッピングされます。物理アドレス空間とは、プロセッサがアドレスバス上に生成できるアドレス範囲のことです。

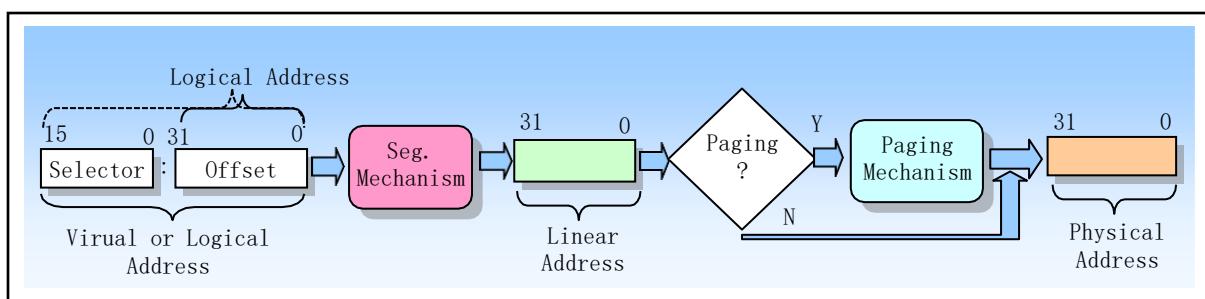


図4-4 論理アドレス-物理アドレス変換

1. Segmentation mechanism

セグメント化は、個々のコード、データ、スタック領域を分離するメカニズムであり、複数の

のプログラム（またはタスク）が互いに干渉することなく同じプロセッサ上で動作することができます。ページング機構は、従来のデマンドページと仮想メモリシステムの実装機構を提供する。仮想メモリーシステムは、プログラムコードが必要に応じて物理メモリーにマッピングされることを実現するために使用されます。ページングメカニズムはもちろん、複数のタスク間のアイソレーションを実現するためにも使用できます。

図4-5に示すように、セグメンテーションは、プロセッサのアドレス可能なリニアアドレス空間を、セグメントと呼ばれる、より小さな保護されたアドレス空間領域に分割するメカニズムを提供します。セグメントは、プログラムコード、データ、スタック、またはシステムデータ構造（TSSやLDTなど）を格納するために使用できます。プロセッサ内で複数のプログラムやタスクが実行されている場合、各プログラムは独自のセグメントセットを割り当てることができます。このとき、プロセッサはこれらのセグメント間の境界を強制し、あるプログラムが他のプログラムのセグメントにアクセスすることで、そのプログラムの実行を妨害しないようにすることができます。セグメント化により、セグメントを分類することもできます。このようにして、特定の種類のセグメントに対する操作を制限することができます。

システム内のすべての使用済みセグメントは、プロセッサのリニアアドレス空間に含まれます。指定されたセグメント内のバイトを見つけるためには、プログラムは論理アドレスを指定する必要があります。論理アドレスには、セグメント・セレクタとオフセットが含まれます。セグメント・セレクタは、セグメントを一意に識別するためのものです。また、セグメント・セレクタは、セグメント・ディスクリプタ・テーブル（例：グローバル・ディスクリプタ・テーブルGDT）内のデータ構造（セグメント・ディスクリプタと呼ばれる）のオフセットを提供します。各セグメントには、セグメント記述子があります。セグメント記述子は、セグメントのサイズ、セグメントのアクセス権と特権レベル、セグメント・タイプ、リニア・アドレス空間におけるセグメントの1バイト目の位置（セグメントのベース・アドレスと呼ぶ）を指定する。論理アドレスのオフセットは、セグメントのベースアドレスに追加され、セグメント内のバイトを特定します。したがって、ベースアドレスにオフセットを加えたものが、プロセッサのリニアアドレス空間のアドレスとなります。

線形アドレス空間は、物理アドレス空間と同じ構造を持っています。2次元の論理アドレス空間に比べ、どちらも1次元のアドレス空間です。仮想アドレス（論理アドレス）空間は、最大16Kのセグメントを含むことができ、各セグメントは最大4GBとなり、仮想アドレス空間の容量は64TB（ 2^{46} ）となります。線形アドレス空間と物理アドレス空間はともに4GB（ 2^{32} ）です。実際、ページング機構を無効にした場合、リニアアドレス空間が物理アドレス空間となります。

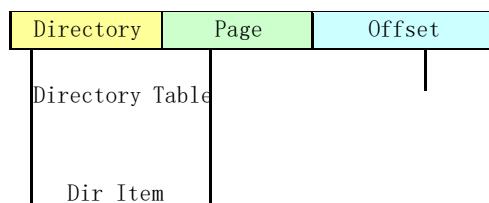
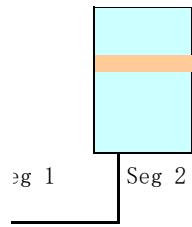


図4-5 論理アドレス、リニアアドレス、物理アドレス

2. Paging mechanism

マルチタスクシステムでは通常、含まれる物理メモリよりもはるかに大きなリニアアドレス空間を定義するため、何らかの「仮想化」されたリニアアドレス空間のアプローチ、つまり仮想ストレージ技術の使用が必要となる。仮想記憶は、プログラマーが、コンピュータの実際の物理的なメモリ容量よりもはるかに大きなメモリ空間があるように錯覚させるメモリ管理技術である。この錯覚を利用すれば、実際にどのくらいの物理メモリが存在するかを考慮することなく、大きなプログラムを自由にプログラミングすることができる。

ページング機構は、仮想記憶技術に対応しています。仮想記憶を使用する環境では、大容量のリニアアドレス空間を少量の物理メモリ (RAMやROM) と外部記憶領域 (大容量のハードディスクなど) で

シミュレートする必要があります。ページングを用いる場合は、各セグメントをページ（通常1ページ4KB）に分割し、物理メモリまたはハードディスクに格納する。

オペレーティングシステムは、ページディレクトリといいくつかのページテーブルを管理することで、これらのページに注意を払っています。プログラム（またはタスク）がリニアアドレス空間のアドレス位置にアクセスしようとすると、プロセッサはページディレクトリとページテーブルを使用してリニアアドレスを物理アドレスに変換し、そのメモリ位置で必要な操作（読み取りまたは書き込み）を行います。

現在訪問しているページが物理メモリにない場合、プロセッサはプログラムの実行を（ページfault例外を発生させることで）中断します。オペレーティングシステムは、その後、ハードディスクから物理メモリにページを読み込み、先ほど中断されたプログラムの実行を継続することができます。オペレーティングシステムがページングメカニズムを厳密に実装している場合、物理メモリとハードディスクの間のページの交換は、正しく実行されたプログラムには透過的に行われます。

80X86のページング機構は、仮想記憶技術のサポートに最も適している。ページング機構では固定サイズのメモリブロックを使用し、セグメント管理では可変サイズのブロックを使用してメモリを管理します。固定サイズのブロックを使ったページングは、物理メモリやハードディスク上のメモリを管理するのに適している。一方、セグメント管理では、可変サイズのブロックを使用するため、複雑なシステムの論理的なパーティションの処理に適しています。固定サイズのページに制約されなく、論理ブロックサイズに収まるメモリセルを定義することができます。また、各セグメントを1つのユニットとして扱うことができるため、セグメントの保護や共有が容易になります。

- 4.2.4 セグメント化とページングは、2つの異なるアドレス変換メカニズムであり、いずれもアドレス変換操作全体に独立した処理ステージを提供する。どちらもメモリに格納された変換テーブルを使用するが、使用するテーブル構造が異なる。実際、セグメントテーブルはリニアアドレス空間に格納されており、ページテーブルは物理アドレス空間に格納されている。したがって、セグメント変換テーブルは、セグメント機構の情報や協力なしに、ページング機構によって再配置することができます。セグメント変換機構は、仮想アドレス（論理アドレス）をリニアアドレスに変換し、リニアアドレス空間にある自身のテーブルにアクセスしますが、ページング機構がこのリニアアドレスを物理アドレスに変換するプロセスを知りません。同様に、ページングメカニズムは、プログラムがアドレスを生成する仮想アドレス空間を知りません。ページングメカニズムは、単にリニアアドレスを物理アドレスに変換し、物理メモリ上の独自の変換テーブルにアクセスします。

4.2.5 Protection

1. 80X86では、2種類の保護方法をサポートしています。1つは、各タスクに異なる仮想アドレス（論理アドレス）空間を与えて、各タスクを完全に分離すること。実装の原則は、各タスクに異なる論理アドレスと物理アドレスのマッピングを与えることです。もう1つの保護機構は、タスク上で動作し、オペレーティングシステムのメモリセグメントやプロセッサの特殊システムレジスタがアプリケーションからアクセスされないように保護します。

2. Protection between tasks

保護の重要なポイントとして、アプリケーションのタスク間の保護を行うことが挙げられます。80X86

では、各タスクを異なる仮想アドレス空間に配置し、各タスクに論理アドレスと物理アドレスの異なるマッピングを与える方法が採用されている。各タスクのアドレス変換機能は、あるタスクの論理アドレスは物理メモリの一部にマッピングされ、別のタスクの論理アドレスは物理メモリの別の領域にマッピングされるように定義されている。このようにすると、あるタスクは他のタスクの対応する論理アドレスにマッピングできる物理メモリの部分を生成できないため、すべてのタスクが分離されます。各タスクに個別のマッピングテーブルを与えれば、各タスクは異なるアドレス変換機能を持つことになります。80X86では、各タスクはそれぞれセグメントテーブルとページテーブルを持っています。プロセッサが新しいタスクを実行するために切り替えるとき、タスク切り替えの重要な部分は、新しいタスクの変換テーブルに切り替えることです。

すべてのタスクに同一の仮想アドレスと物理アドレスのマッピング部分を配置し、オペレーティング

システムをこの共通の仮想アドレス空間部分に置くことで、OSをすべてのタスクで共有することができます。このすべてのタスクが持っている仮想アドレス空間の同じ部分をグローバルアドレス空間と呼びます。最近のLinux OSでは、まさにこのように仮想アドレス空間が使われています。

3. 仮想アドレス空間のうち、各タスクに固有の部分を「ローカルアドレス空間」と呼びます。ローカルアドレス空間には、システム内の他のタスクと区別する必要のあるプライベートコードとデータが含まれています。各タスクには異なるローカルアドレス空間が存在するため、2つの異なるタスクで同じ仮想アドレスを参照すると、異なる物理アドレスに変換されます。これにより、OSは各タスクのメモリに同じ仮想アドレスを与えつつ、各タスクを分離することができます。一方、グローバルアドレス空間では、すべてのタスクが同じ仮想アドレスを参照すると、同じ物理アドレスに変換されます。これにより、共通のコードやデータ（OSなど）の共有をサポートします。

4. Privilege level protection

タスクでは、セグメントに含まれるデータの機密性と、タスク内のプログラムの各部分の信頼度に基づいて、タスク内のセグメントへのアクセスを制限するために、4つの特権レベルが定義されています。最も機密性の高いデータには最も高い特権レベルが与えられ、タスクの中で最も信頼されている部分からしかアクセスできません。機密性の低いデータには低い特権レベルが与えられ、タスク内より低い特権を持つコードがアクセスできます。

特権レベルは0～3の数字で表され、0が最も高い特権レベル、3が最も低い特権レベルとなります。各メモリセグメントには特権レベルが設定されています。この特権レベルによって、十分な特権レベルを持つプログラムがそのセグメントにアクセスすることが制限されます。プロセッサは、CSレジスタで指定されたセグメントから命令をフェッチして実行することがわかっています。現在の特権レベル、つまりCPLは、現在アクティブなコードセグメントの特権レベルであり、現在実行中のプログラムの特権レベルを定義しています。CPLは、プログラムがどのセグメントにアクセスできるかを決定します。

プログラムがセグメントにアクセスしようとするたびに、現在の特権レベルとセグメントの特権レベルが比較され、アクセス許可があるかどうかが判断されます。ある CPL レベルで実行されたプログラムは、同じレベルまたは下位レベルのデータ・セグメントへのアクセスを許可します。高レベルのセグメントへの参照は違法であり、オペレーティングシステムに通知する例外が発生します。

各特権レベルは、共有スタックの使用に伴う保護の問題を避けるために、独自のプログラムスタックを持っています。プログラムがある特権レベルから別の特権レベルに切り替わると、スタックセグメントも新しいレベルのスタックに変更されます。

4.3 Segmentation Mechanism

セグメント化のメカニズムは、さまざまなシステムデザインを実現するために使用できます。例えば、プログラムを保護するために最低限の機能しか使わないフラットモデルから、複数のプログラム（またはタスク）を確実に実行できる堅牢な動作環境を構築するためにセグメント化を利用するマ

ルチセグメントモデルまで、さまざまなデザインがあります。

システムの最もシンプルなメモリモデルは、基本的なフラットモデルです。このモデルでは、オペレーティングシステムやプログラムは、セグメント化されていない連続したアドレス空間にアクセスできます。この基本的なフラットモデルは、システム設計者やアプリケーションプログラマから、アーキテクチャのセグメント化の仕組みをほとんどの場合、隠すことができます。基本的なフラットメモリモデルを実装するには、少なくとも2つのセグメント記述子を作成する必要があります。1つは参照コードセグメント用、もう1つは参照データセグメント用です。ただし、どちらのセグメントもリニアアドレス空間全体にマッピングされます。つまり、2つのセグメント記述子は、同じベースアドレスの値に対して、0と4GBという同じセグメント制限を持っています。

マルチセグメントモデルでは、セグメンテーションの仕組みを利用して、ハードウェアで強化されたコード、データ構造、プログラム、タスクを完全に保護することができます。一般的に、各プログラム（またはタスク）は、それぞれの

セグメント記述子テーブルと自分のセグメント。プログラムの場合、セグメントは完全にプライベートなものもあれば、プログラム間で共有されるものもあります。システム上のすべてのセグメントと各実行プログラムの実行環境へのアクセスは、ハードウェアによって制御されます。

4.3.1 アクセスチェックは、セグメントの境界外のアドレスへの参照を保護するだけでなく、特定のセグメントで許されない動作の実行を防止するためにも使用できます。例えば、コードセグメントは読み取り専用に設計されているので、ハードウェアを使ってコードセグメントへの書き込みを防止することができます。また、セグメントのアクセス権情報を利用して、保護リングや保護レベルを設定することもできます。保護レベルは、アプリケーションによる不正なアクセスからオペレーティングシステムのプログラムを保護するために使用できます。

4.3.2 Segment definition

1. 前節の概要で述べたように、80X86ではプロテクトモードで4GBの物理アドレス空間が用意されている。これは、プロセッサがアドレスバスでアドレス指定できるアドレス空間です。このアドレス空間はフラットで、アドレス範囲は0から0xFFFFFFFFまでです。この物理アドレス空間は、読み書き可能なメモリ、リードオンリーメモリ、メモリマップドI/Oにマッピングすることができます。セグメント化の仕組みは、仮想アドレス空間内の仮想メモリを、セグメントと呼ばれるいくつかの可変長のメモリブロック単位に整理することである。80X86の仮想アドレス空間における仮想アドレス（論理アドレス）は、セグメント部分とオフセット部分で構成されています。セグメントは、仮想アドレスから直線的なアドレスへの変換メカニズムの基礎となります。各セグメントは3つのパラメータで定義されます。
 2. Base address specifies the starting address of the segment in the linear address space. The base address is a linear address and corresponds to offset 0 in the segment.
 3. The segment limit is the maximum available offset within the segment in the virtual address space. It defines the length of the segment.
 4. Attributes, specify the characteristics of the segment. For example, whether the segment is readable, writable, or executable as a program; the privilege level of a segment, and so on.

セグメントの長さは、仮想アドレス空間におけるセグメントのサイズを定義します。セグメントベースのアドレスとセグメントリミットの長さは、セグメントがマッピングされるリニアアドレスの範囲または領域を定義します。セグメントの0からリミットまでのアドレス範囲は、リニアアドレスのベースからベース+リミットまでの範囲に相当します。セグメントリミットを超えるオフセットを持つ仮想アドレスは意味がなく、使用すると例外が発生する可能性があります。また、セグメント属性の許可を得ずにセグメントにアクセスした場合も例外が発生します。たとえば、読み取り専用のセグメントを書き込もうとすると、80X86は例外を発生させます。さらに、リニアアドレスにマッピングされた複数のセグメントの範囲は、図4-6のように部分的に重なったり、被ったり、あるいは完全に重なってしまうこともあります。本書で紹介するLinux 0.1xシステムでは、タスクのコードセグメントとデータセグメントのセグメントの長さは同じであり、リニアアドレスが同一で重なる領域にマッピングさ

れています。

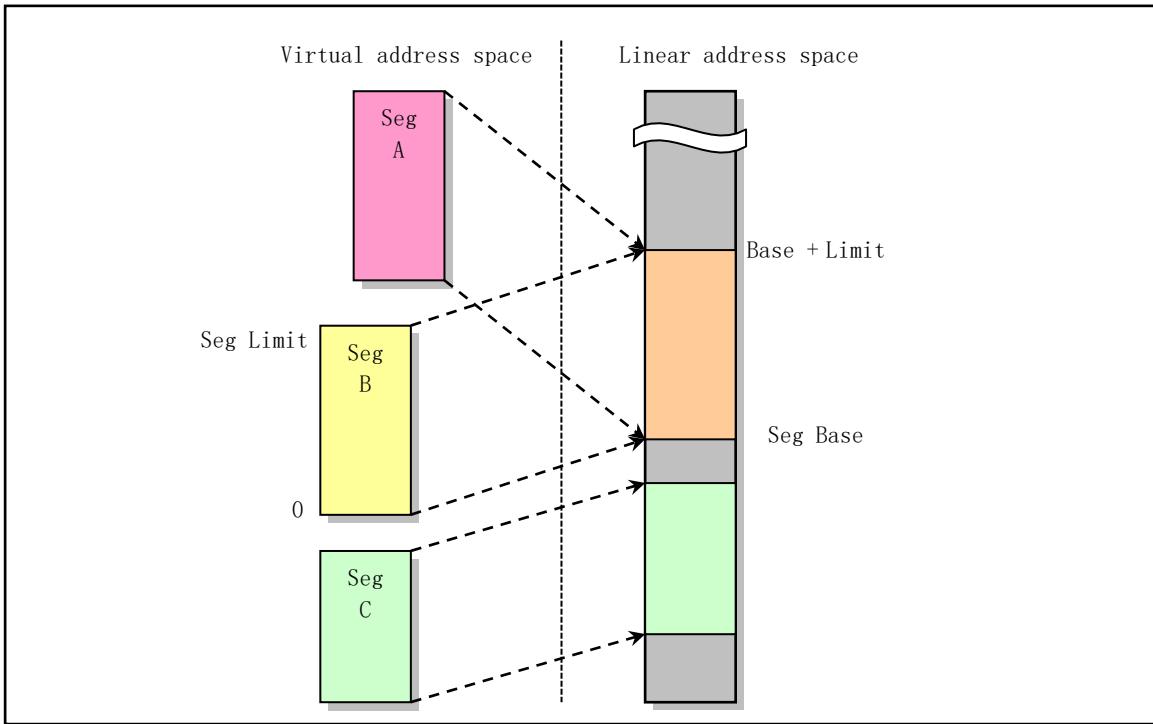


図4-6 リニアアドレス空間への仮想マップのセグメンテーション

セグメントのベースアドレス、制限長、保護属性は、セグメント記述子と呼ばれる構造体に格納されます。このセグメント記述子は、論理アドレスから線形アドレスへの変換マッピングの際に使用されます。セグメント記述子は、記述子テーブルに格納されます。セグメント・ディスクリプター・テーブルは、セグメント・ディスクリプター・アイテムを含む単純な配列です。前述のセグメント・セレクタは、テーブル内のセグメント・ディスクリプタの位置を指定することで、対応するセグメントを指定します。

セグメントの機能を最小限にしても、論理アドレスを使ってプロセッサのアドレス空間の各バイトにアクセスできます。論理アドレスは、図4-7に示すように、16ビットのセグメントセレクタと32ビットのオフセットで構成されます。セグメントセレクタはバイトが配置されているセグメントを指定し、オフセットはセグメントベースアドレスに対するセグメント内のバイトの位置を指定します。プロセッサは、各論理アドレスをリニアアドレスに変換します。リニアアドレスとは、プロセッサのリニアアドレス空間における32ビットのアドレスのことです。物理アドレス空間と同様に、リニアアドレス空間もまた、0から0xFFFFFFFFまでのアドレスを持つフラットな4GBのアドレス空間です。リニアアドレス空間には、システムで定義されたすべてのセグメントとシステムテーブルが含まれています。

1. 論理アドレスをリニアアドレスに変換するために、プロセッサは以下の演算を行います。
2. Use the offset value (segment index) in the segment selector to locate the corresponding segment descriptor in the GDT or LDT table. (This step is only needed if a new segment selector is loaded into a segment register.)
3. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
4. Add the segment base address obtained in the segment descriptor to the offset and finally form a linear

address.



図4-7 論理アドレスからリニアアドレスへの変換

4.3.3 ページングが有効でない場合、プロセッサはリニアアドレスを物理アドレスに直接マッピングします（つまり、リニアアドレスはプロセッサのアドレスバスに送られます）。リニアアドレス空間がページングされている場合は、第2レベルのアドレス変換を用いてリニアアドレスを物理アドレスに変換します。ページ変換については後で説明します。

4.3.4 Segment Descriptor Tables

図4-8に示すように、セグメントディスクリプターテーブルは、セグメントディスクリプターの配列です。ディスクリプターテーブルは可変長で、最大8192個の8バイトディスクリプターを格納できます。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つがあります。

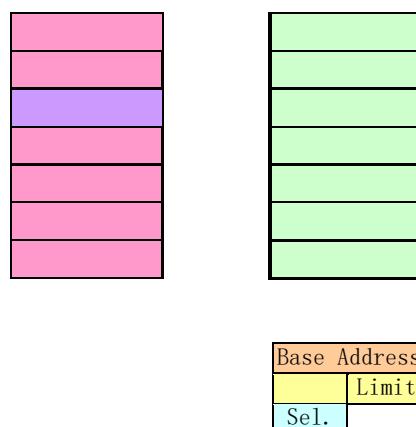


図4-8 Global and Locat Descriptor Tables

各システムには、システム内のすべてのプログラムおよびタスクに使用できる1つのGDTが必要です。オプションとして、1つまたは複数のLDTを定義することができます。例えば、実行中の個々のタスクごとにLDTを定義したり、一部またはすべてのタスクで同じLDTを共有することができます。GDT自体はセグメントではなく、リニアアドレス空間内のデータ構造である。GDTの基本リニアアド

レスとリミットは、GDTRレジスタにロードする必要があります。また、GDTの基本アドレスは

を8バイトごとに整列させることで、プロセッサの性能を最大限に引き出すことができます。GDTの制限値はバイト単位で表されます。セグメント化と同様に、リミット値はベースアドレスに追加され、最後の有効バイトのアドレスになります。リミット値が0の場合、有効なバイトは1つになります。セグメント記述子の長さは常に8バイトであるため、GDTリミットは常に8の整数倍（つまり $8N-1$ ）より小さくなければなりません。

LDTテーブルは、LDTタイプのシステムセグメントに格納されます。このとき、GDTにはLDTのセグメント記述子が含まれていなければなりません。システムが複数のLDTをサポートしている場合は、各LDTにセグメント記述子とセグメントセレクタがGDTに含まれていなければなりません。LDTセグメント記述子は、GDTテーブルのどこにでも格納できます。

LDTにアクセスするには、そのセグメントセレクタが必要です。LDTにアクセスする際のアドレス変換回数を減らすために、LDTのセグメントセレクタ、ベースアドレス、セグメント長、アクセス権をLDTRレジスタに格納する必要があります。

SGDT命令でGDTRレジスタをストアすると、48ビットの「疑似ディスクリプタ」がメモリに格納される。ユーザー モード（特権レベル3）でのアライメントチェックエラーを回避するために、ダミー ディスクリプターは奇数ワードのアドレスに格納する必要があります（例：アドレスMOD4=2）。これにより、プロセッサは最初にアラインド・ワードを格納し、次にアラインド・ダブルワード（4バイト・アラインメント）を格納します。ユーザー モードのプログラムでは、通常、ダミーの記述子を保存しませんが、アライメントチェックエラーの可能性を避けるために、このアライメントを使用することができます。また、SIDT命令でIDTRレジスタの内容を保存する場合にも、同じアラインメントが使用されます。ただし、LDTRやタスクレジスタを保存する場合（それぞれSLTR命令、STR命令を使用）は、ダミー記述子をダブルワードアラインドのアドレス（すなわち、アドレスMOD4=0）に保存する必要があります。

記述子テーブルは、オペレーティングシステムが保持する特殊なデータ構造に格納され、プロセッサのメモリ管理ハードウェアによって参照されます。これらの特別な構造体は、アプリケーションがその中のアドレス変換情報を変更できないように、オペレーティングシステムソフトウェアのみがアクセスできる保護されたメモリ領域に格納する必要があります。仮想（論理）アドレス空間は、同じ大きさの2つのハーフに分けられます。半分はGDTによってリニアアドレスにマッピングされ、もう半分はLDTによってマッピングされます。仮想アドレス空間全体には 2^{14} 個のセグメントがあり、その半分（つまり 2^{13} 個のセグメント）はGDTによってマッピングされたグローバル仮想アドレス空間であり、残りの半分はLDTによってマッピングされたローカル仮想アドレス空間である。記述子テーブル（GDTまたはLDT）とテーブル内の記述シンボルを指定することで、記述子の位置を特定することができます。

タスクの切り替えが発生すると、LDTは新しいタスクのLDTに置き換えられますが、GDTは変更

されません。したがって、GDTによってマッピングされた仮想アドレス空間の半分はシステム内のすべてのタスクに共通ですが、残りの半分のLDTのマッピングはタスクが切り替わったときに変更されます。システム内のすべてのタスクが共有するセグメントは、GDTによってマッピングされます。このようなセグメントには、通常、オペレーティングシステムを含むセクションと、LDTを含む各タスクの特別なセクションがあります。LDTセグメントは、オペレーティングシステムに属するデータと考えることができます。

LDTには、1つのタスク専用のセグメントの記述子があります。複数のタスクが共通の LDT を共有することができます。この場合、これらのタスクはすべて同じ LDT を持っているため、同じセグメントのセットを使用することができ、すべてのタスクは 1 つの GDT を共有します。また、両タスクはそれぞれのLDTでセグメント記述子を共有することができるので、GDTに記述子を置くことなくセグメントを共有することが、すべてのタスクで共有されます。この場合、共有セグメントは2つの異なるLDTに2つの記述子を持ち、一緒に更新しなければならないため、OSが排他的に処理しなければなりません。

図4-9は、GDTとLDTの間でタスクのセグメントがどのように分けられるかを示しています。この図では、2つのアプリケーション（AとB）とオペレーティング・システムに対して6つのセグメントがあります。システム内の各アプリケーションはタスクに対応しており、各タスクにはそれぞれLDTがあります。アプリケーションAはタスクAで実行され、セグメントCodeAとDataAをマップするLDTAを持っています。同様に、アプリケーションBはタスクBで実行され、LDTBを使ってCodeBとDataBのセグメントをマッピングします。オペレーティングシステムのカーネルを含む 2 つのセグメント、CodeOS と DataOS は、LDT を使用してマッピングされます。

GDTは、両方のタスクで共有できるようになっています。2つのLDTセグメント LDTAとLDTBもGDTでマッピングされています。

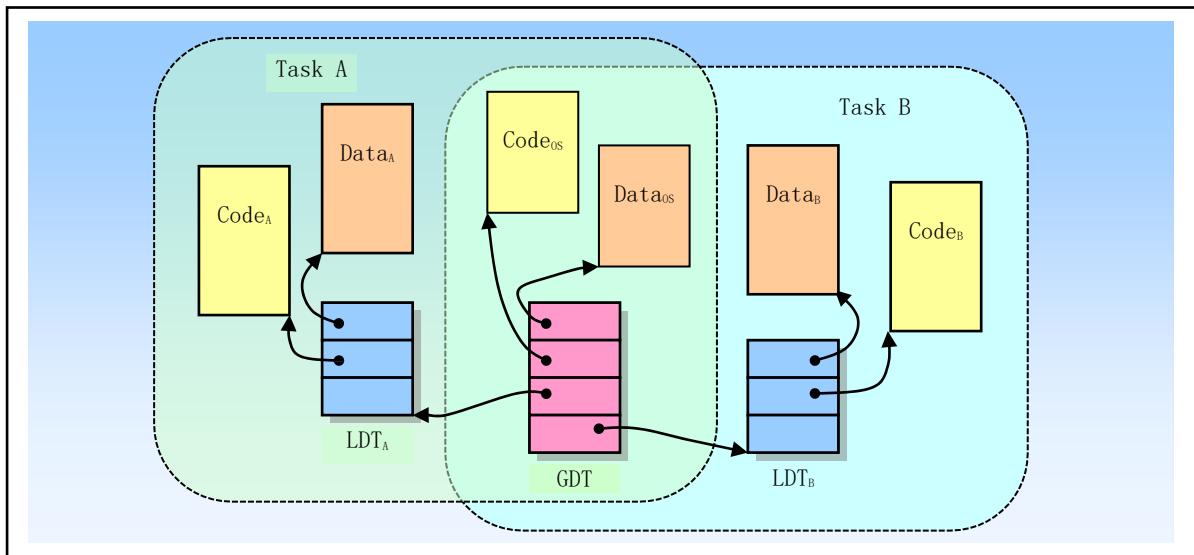


図4-9 タスクで使用されるセグメントの種類

タスク A が実行されているとき、アクセス可能なセグメントには、LDTA マップの CodeA と DataA のセグメントに加え、GDT マップのオペレーティング・システムの CodeOS と DataOS のセグメントが含まれます。タスクBが実行中の場合、アクセス可能なセグメントには、LDTBマップの CodeBとDataBのセグメントに加え、GDTマップのセグメントが含まれます。

この例では、仮想アドレス空間を整理して、各タスクが異なるLDTを使用することで、各タスクを分離することができます。タスクAが実行中の場合、タスクBのセグメントは仮想アドレス空間に含まれないため、タスクAはタスクBのメモリにアクセスできません。同様に、タスクBが実行されているときは、タスクAのセグメントはアドレス指定できません。このように、LDTを使って各アプリケーションタスクを分離する方法は、重要な保護ニーズの一つです。

4.3.5 Segment Selectors

セグメント・セレクタは、図4-10に示すように、セグメントを表す16ビットの識別子です。セグメント・セレクタはセグメントを直接指すのではなく、セグメント・ディスクリプタ・テーブルの中のセグメントを定義するセグメント・ディスクリプタを指します。セグメント・セレクタは3つのフィールドで構成され、その内容は以下の通りです。

- Requested Privilege Level (RPL);
- Table Index (TI);
- Index.

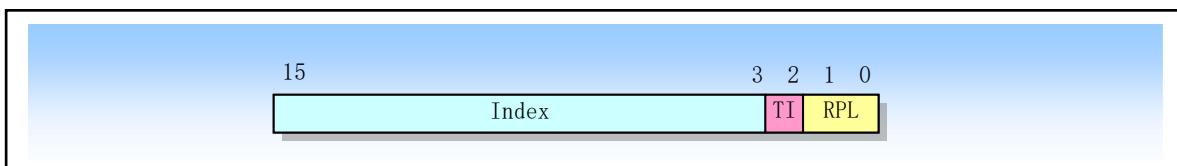


図 4-10 セグメントセレクタの構造

リクエスト・プリビレッジ・レベルRPLは、後に詳述するセグメント・プロテクション情報を提供する。テーブル・インデックス TI は、指定されたセグメントを含むセグメント記述子テーブル GDT または LDT を示すために使用されます。

セグメントの記述子です。TI=0はGDTに、TI=1はLDTに記述子があることを示す。インデックス・フィールドは、GDTまたはLDTテーブル内のディスクリプターのインデックス番号を示す。このように、セレクタはセグメント・テーブル内の記述子を探してセグメントを指定し、記述子にはセグメントのベース・アドレス、セグメントの長さ、セグメントの属性など、セグメントにアクセスするためのすべての情報が含まれていることがわかります。

例えば、図4-11(a)のセレクタ(0x08)は、GDTでRPL=0のセグメント1を指定しています。インデックス・フィールドの値は1、TIビットは0で、GDTテーブルが指定されています。図4-11(b)のセレクタ(0x10)は、GDT内のRPL=0のセグメント2を指定します。インデックス・フィールドの値は2、TIビットは0で、GDTテーブルが指定されています。図4-11(c)のセレクタ(0x0f)は、LDTのLPL=3のセグメント1を指定しています。インデックス・フィールドの値は1、TIビットは1で、LDTテーブルが指定されています。図4-11(d)のセレクタ(0x17)は、LPL=3のセグメント2をLDTで指定しています。インデックス・フィールドの値は2、TIビットは1で、LDTテーブルが指定されています。実際、図4-11の最初の4つのセレクタ：(a)、(b)、(c)、(d)は、それぞれLinux 0.1xカーネルのカーネル・コード・スニペット、カーネル・データ・スニペット、タスク・コード・スニペット、タスクを表しています。データセグメントのセレクタ。図4-11(e)のセレクタ(0xffff)は、LDTテーブルのRPL=3のセグメント8191を指定しています。インデックス・フィールドの値は0b1111111111（つまり8191）、TIビットは1で、LDTテーブルが指定されています。

| | | |
|--|--|--|
| Index 00000000000001 (a) Selector 0x0008 | Index 00000000000010 (b) Selector 0x0010 | Index 00000000000001 (c) Selector 0x000f |
| Index 00000000000010 (d) Selector 0x0017 | Index 11111111111111 (e) Selector 0xffff | Index 00000000000010 (f) Selector 0x0000 |

図 4-11 セグメントセレクターの例

また、プロセッサはGDTテーブルの最初の項目を使用しません。図4-11(f)に示すように、GDTエントリへのセレクタ(つまり、インデックス値が0でTIフラグが0のインデックス)は、「空セレクタ」として使用されます。空のセレクタをセグメント・レジスタ (CSとSS以外) にロードしても、プロセッサは例外を発生させません。しかし、空のセレクタを含むセグメント・レジスタを使ってメモリにアクセスすると例外が発生します。CSやSSのセグメント・レジスタに空のセレクタをロードすると例外が発生します。

セグメント・セレクタはポインタ変数の一部としてアプリケーションに表示されますが、セレクタの値は通常、アプリケーションではなく、リンク・エディタやリンク・ローダによって設定または変更されます。アドレス変換の時間とプログラミングの複雑さを軽減するために、プロセッサには最大6つ

のセグメントセレクタを保持するレジスタ（図4-12参照）、すなわちセグメントレジスタが用意されています。各セグメントレジスタは、特定の種類のメモリ参照（コード、データ、スタック）をサポートします。原則として、各プログラムは少なくとも有効なセグメント・セレクタをコード・セグメント（CS）、データ・セグメント（DS）、スタック・セグメント（SS）の各レジスタにロードする必要があります。また、プロセッサには3つの補助データセグメントレジスタ（ES、FS、GS）が用意されており、現在実行中のプログラム（またはタスク）が他の複数のデータセグメントにアクセスできるようにするために使用されます。

| Seg Selector | Base Address, Limit, Access Info |
|--------------|----------------------------------|
| | |
| | |
| | |
| | |
| | |

図4-12 セグメントレジスタの構造

セグメントにアクセスするプログラムでは、セグメント・セレクタがセグメント・レジスタにロードされている必要があります。したがって、システムでは多くのセグメントを定義できますが、同時にすぐにアクセスできるのは 6 つのセグメントだけです。他のセグメントにアクセスするには、これらのセグメントのセレクタをロードする必要があります。

また、メモリにアクセスするたびに記述子テーブルを読まなくても済むように、セグメント記述子を読み込んでデコードするために、各セグメントレジスタには「可視部」と「隠部」があります（隠部は「記述子バッファ」または「シャドーレジスタ」とも呼ばれます）。セグメント・セレクタがセグメント・レジスタの可視部分にロードされると、プロセッサはセグメント・セレクタが指すセグメント記述子のセグメント・アドレス、セグメント長、およびアクセス制御情報をセグメント・レジスタの隠し部分にロードします。セグメント・レジスタ（可視部分と非表示部分）にバッファリングされた情報により、プロセッサはアドレス変換を行う際に、セグメント記述子からベース・アドレスとリミット値を読み取る時間を短縮することができます。

シャドウ・レジスタには、ディスクリプター情報のコピーが含まれているため、オペレーティング・システムは、ディスクリプター・テーブルへの変更がシャドウ・レジスタに反映されるようにしなければなりません。そうしないと、記述子テーブルのセグメントのベース・アドレスまたはリミットが変更されても、その変更がシャドウ・レジスタに反映されません。このような問題に対処する最も簡単な方法は、ディスクリプター・テーブルのディスクリプターに変更を加えた直後に、6つのセグメント・レジスタをリロードすることです。これにより、ディスクリプター・テーブルの対応するセグメント情報がシャドー・レジスタに再ロードされます。

1. セグメントレジスタをロードするためのロード命令には、2種類あります。
2. Like MOV, POP, LDS, LES, LSS, LGS and LFS instructions. These instructions explicitly reference the segment register directly;
3. Implicitly loaded instructions such as CALL, JMP, and RET instructions using long pointers, IRET, INTn, INTO, and INT3 instructions. These instructions are accompanied by changes to the contents of the CS register (and some other segment registers) during operation.

4.3.6 もちろん、MOV命令を使ってセグメントレジスタの可視部分の内容を汎用レジスタに格納することも

できます。

4.3.7 Segment Descriptor

先ほど、セグメントセレクターを使って記述子テーブルの記述子を探すことを説明しました。セグメント記述子は、GDTおよびLDTテーブル内のデータ構造項目で、セグメントの位置やサイズ、アクセス制御の状態などの情報をプロセッサに提供するために使用されます。各セグメント記述子は8バイト長で、セグメントのベースアドレス、セグメントの長さ、セグメントの属性の3つのフィールドを含みます。セグメント・ディスクリプターは通常、コンパイラ、リンカー、ローダ、オペレーティング・システムによって作成されますが、決してアプリケーションではありません。図4-13は、すべてのタイプのセグメント記述子の一般的なフォーマットを示しています。

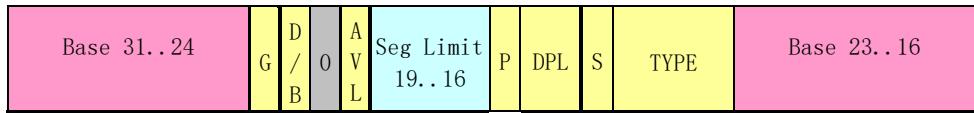


図4-13 セグメント記述子の一般的なフォーマット

- ◆ セグメントデスクリプタのフィールドとフラグの意味は以下の通りです。
- ◆ Segment limit field (LIMIT)
 - セグメント・リミット・フィールドは、セグメントのサイズを指定するために使用される。
 - プロセッサは、セグメント記述子の2つのセグメント・リミット・フィールドを20ビットの値に結合し、セグメント・リミット長Limit値の実際の意味を粒度フラグGに応じて指定する。
 - G=0の場合、セグメント長Limitの範囲は1バイトから1MBバイトまで、単位はバイトである。
 - G=1の場合、セグメント長Limitの範囲は4KBから4GBまでで、単位は4KBとなります。
- ◆ セグメントタイプのセグメント拡張方向フラグEに応じて、プロセッサはセグメントリミットLengthを2つの異なる方法で使用します。上方拡張セグメント（上方拡張セグメントと略す）の場合、論理アドレスのオフセット値は0からセグメント限界値Limitまでの範囲になります。限界長Limitより大きいオフセットは、一般的な保護例外を発生させます。下方に延長されているセグメント（下位セグメントと略す）では、セグメント限界値Limitの意味が逆になります。デフォルトのスタック・ポインタ・サイズ・フラグBの設定に応じて、オフセット値はセグメント限界長から0xFFFFFFFFまたは0xFFFFまでの範囲になります。オフセット値が制限長Limitより小さい場合、一般的な保護例外が発生します。次の拡張セグメントでは、セグメントリミットフィールドの値を減らすと、新しいメモリがセグメントアドレス空間の一番上ではなく、一番下に割り当てられます。80X86のスタックは常に縮小されているので、この実装はスタックを拡張するのに適しています。
- ◆ Base address field (BASE)
 - ◆ このフィールドは、4GBのリニアアドレス空間におけるバイト0のセグメントの位置を定義します。プロセッサは、3つの異なるベースアドレスフィールドを組み合わせて32ビットの値を形成します。セグメントのベースアドレスは、16バイトの境界にアラインする必要があります。これは必須ではありませんが、プログラムのコードセグメントとデータセグメントを16バイト境界に揃えることで、プログラムの最高のパフォーマンスを得ることができます。
- ◆ Type field (TYPE)
 - タイプフィールドは、セグメントまたはゲートのタイプ、セグメントを記述するためのアクセスのタイプ、セグメントの拡張方向を指定する。このフィールドの解釈は、アプリケーション（コードまたはデータ）記述子であるか、システム記述子であるかを示す記述子タイプ

フラグSに依存する。のです。

- ◆ TYPE フィールドのエンコーディングは、図4-14に示すように、コード、データ、システムディスクリプターで異なります。
- ◆ Descriptor type flag (S)
 - ◆ 記述子タイプフラグ S は、セグメント記述子がシステムセグメント記述子であるか ($S=0$ の場合) 、コードまたはデータセグメント記述子であるか ($S=1$ の場合) を示す。
- ◆ Descriptor privilege level (DPL)
 - ◆ DPL フィールドは、ディスクリプターの特権レベルを示す。特権レベルの範囲は 0 から 3 までで、0 が最も高く、3 が最も低いレベルとなります。DPL はセグメントへのアクセスを制御するために使用されます。
- ◆ Segment present (P)
 - ◆ セグメント・プレゼンス・フラグ P は、セグメントがメモリ内にある ($P=1$) か、メモリ内にない ($P=0$) かを示します。セグメント記述子の P フラグが 0 の場合、このセグメント記述子を指すセレクタをセグメント・レジスタにロードすると、例外なくセグメントが生成されます。メモリ管理ソフトウェアはこのフラグを利用して、ある時点でのセグメントを実際にメモリにロードする必要性を制御することができます。この機能により、仮想ストレージはページング・メカニズムを超えた制御が可能になります。図4-15は、 $P=0$ の場合のセグメント記述子のフォーマットを示しています。P フラグが 0 の場合、セグメントが実際には存在しない場所に関する情報など、フォーマット内で Available とマークされているフィールドを使って、オペレーティングシステムは独自のデータを自由に保存することができます。
- ◆ Default operation size/default stack pointer size and/or upper bound (D/B)
 - このマークは、セグメント記述子が、実行コードセグメント、スプレッドデータセグメント、スタックセグメントのいずれであるかによって、機能が異なります。(32 ビットのコードセグメントおよびデータセグメントでは、このフラグは常に 1 に設定されるべきで、16 ビットのコードセグメントおよびデータセグメントでは、このフラグは 0 に設定されます)。
 - **Executable code segment.** This flag is called the D flag at this time and is used to indicate that the instruction in this segment refers to a valid address and the default length of the operand. If the flag is set, the default value is a 32-bit address and a 32-bit or 8-bit operand; if the flag is 0, the default value is a 16-bit address and a 16-bit or 8-bit operand. The instruction prefix 0x66 can be used to select a non-default operand size; the prefix 0x67 can be used to select an address size other than the default.
 - Stack segment (data segment pointed to by the SS register). At this point, this flag is called the B (Big) flag and indicates the size of the stack pointer when an implicit stack operation (such as PUSH, POP, or CALL) occurs. If this flag is set, the 32-bit stack pointer is used and stored in the ESP register; if the flag is 0, the 16-bit stack pointer is used and stored in the SP register. If the stack segment is set to a lower extended data segment, this B flag also specifies the upper bound of the stack segment.
 - Expand the data segment. At this point the flag is called the B flag and is used to indicate the upper limit of the segment. If this flag is set, the upper bound of the stack segment is 0xFFFFFFFF (4GB); if this flag is not set, the upper bound of the stack segment is 0xFFFF (64KB).
- ◆ Granularity (G)
 - ◆ このフィールドは、セグメント・リミット・フィールド値の単位を決定するために使用します。granularity フラグが 0 の場合、セグメント制限値の単位はバイトで、granularity フラグが設定されている場合、セグメント制限値は 4KB の単位を使用します。(このフラグはセグメントのベースアドレスの粒度には影響しません。ベースアドレスの粒度は常にバイト単位です)。G フラグが設定されていると、セグメント長を使ってオフセット値をチェックする際に、オフセット値の 12 ビットの最下位ビットをチェックしません。例えば、 $G=1$ の場合、セグメント制限長が 0 であれば、有効なオフセット値は 0 ~ 4095 であることを示します。
- ◆ Available and reserved bits

セグメント記述子の2つ目のダブルワードのビット20は、システム・ソフトウェアが使用可能で
あり、ビット

21は予約ビットで、常に0に設定する必要があります。

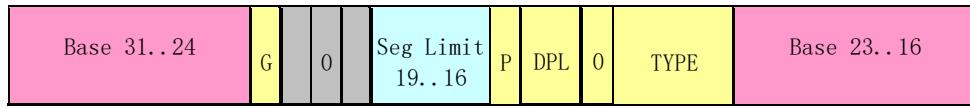
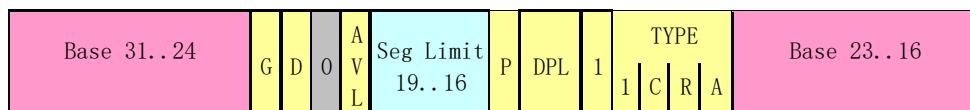
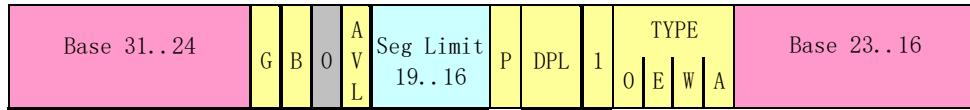


図4-14 コード、データ、システムセグメントの記述子のフォーマット

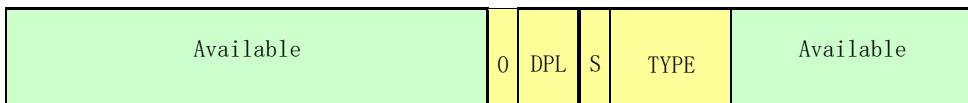


図4-15 セグメントディスクリプター ビットP=0の場合

4.3.8 Code and Data Segment Descriptor Types

セグメント・ディスクリプターにS（ディスクリプター・タイプ）フラグが設定されている場合、そのディスクリプターはコード・セグメントまたはデータ・セグメントに使用されます。このとき、タイプ・フィールドの最上位ビット（2つ目のダブルワードのビット11）を使って、データ・セグメントの記述子（リセット）かコード・セグメントの記述子（セット）かを判断します。

データ・セグメントの場合、タイプ・フィールドの下位3ビット（ビット8、9、10）は、それぞれアクセス済み、書き込み可能、拡張方向を示すのに使用されます。コード・セグメントとデータ・セグメントのタイプ・フィールドのビット・フィールドの説明は、表4-3を参照してください。書き込み可能ビットWの設定により、データ・セグメントはリード・オンリー、またはリード・アンド・ライタブルになります。

| 表4-3 コードとデータセグメント の記述子の種類 TYPE Field | | | | | Descriptor Type | Description |
|--|----|----|---|---|--------------------|-----------------------------------|
| Decimal | 11 | 10 | 9 | 8 | | |
| | | E | W | A | | |
| 0 | 0 | 0 | 0 | 0 | Data | Read-Only |
| 1 | 0 | 0 | 0 | 1 | Data | Read-Only, accessed |
| 2 | 0 | 0 | 1 | 0 | Data | Read/Write |
| 3 | 0 | 0 | 1 | 1 | Data | Read/Write, accessed |
| 4 | 0 | 1 | 0 | 0 | Data | Expand-down, Read-Only. |
| 5 | 0 | 1 | 0 | 1 | Data | Expand-down, Read-Only, accessed |
| 6 | 0 | 1 | 1 | 0 | Data | Expand-down, Read/Write |
| 7 | 0 | 1 | 1 | 1 | Data | Expand-down, Read/Write, accessed |
| | | C | R | A | | |
| 8 | 1 | 0 | 0 | 0 | Code | Execute-Only |
| 9 | 1 | 0 | 0 | 1 | Code | Execute-Only, accessed |
| 10 | 1 | 0 | 1 | 0 | Code | Execute/Read |
| 11 | 1 | 0 | 1 | 1 | Code | Execute/Read, accessed |
| 12 | 1 | 1 | 0 | 0 | Code | Conforming, Execute-Only |

| | | | | | | |
|----|---|---|---|---|------|---|
| 13 | 1 | 1 | 0 | 1 | Code | Conforming, Execute-Only, accessed |
| 14 | 1 | 1 | 1 | 0 | Code | Conforming, Execute/Read-Only |
| 15 | 1 | 1 | 1 | 1 | Code | Conforming, Execute/Read-Only, accessed |

スタックセグメントは読み取り/書き込み可能なデータセグメントでなければなりません。書き換え不可能なデータセグメントセレクターが読み込まれると

SSレジスタを使用すると、一般保護例外が発生します。スタックセグメントの長さを動的に変更する必要がある場合、スタックセグメントは下方向に拡張されたデータセグメントとすることができます（拡張方向フラグが設定されています）。ここで、セグメントの上限を動的に変更すると、スタックスペースがスタックの最下部に追加されます。

アクセスされたビットは、オペレーティング・システムが最後にこのビットをリセットした後、セグメントがアクセスされたかどうかを示します。プロセッサがセグメントセレクタをセグメントレジスタにロードするたびに、このビットがセットされます。このビットは明示的にクリアする必要があります、クリアしないとセットされたままになります。このビットは、仮想メモリの管理やデバッグに使用できます。

コード・セグメントの場合、タイプ・フィールドの下位3ビットは、Accessed、Read-enable、Conformingと解釈されます。読み取り可能なRフラグの設定により、コード・セグメントは実行専用または実行/読み取りが可能です。実行/読み取り可能なコードセグメントは、定数などの静的データと命令コードをROMに配置する場合に使用できます。ここでは、CSオーバーライド・プレフィックスを持つ命令を使用するか、コード・セグメントのコード・セグメント・セレクタをデータ・セグメント・レジスタ（DS、ES、FS、GS）にロードすることで、コード・セグメントのデータを読み出すことができます。プロテクトモードでは、コードセグメントの書き込みはできません。

コードセグメントには、適合するものとしないものがあります。より高い特権レベルの適合セグメントに実行制御を移すと、プログラムは現在の特権レベルで実行を続けることができます。特権レベルの異なる適合しないセグメントに転送すると、コールゲートやタスクゲートを使用しない限り、一般的な保護例外が発生します-（適合するコードセグメントと適合しないコードセグメントの詳細については、「コードセグメントへの直接の呼び出しまだはジャンプ」を参照）。保護機能にアクセスしないシステムツールや、一部の例外タイプ（エラー、オーバーフローなど）は、整合性のあるセグメントに格納できます。低権限のプログラムやプロシージャによるアクセスを防止する必要があるツールは、非適合セグメントに格納する必要があります。なお、対象セグメントが適合コード・セグメントか不適合コード・セグメントかにかかわらず、コールやジャンプによって、より低い権限の（数値的に高い権限レベルの）コード・セグメントに実行を移すことはできません。

すべてのデータセグメントは、特権を持たないプログラムやプロシージャからはアクセスできないという意味で、不適合です。しかし、コード・セグメントとは異なり、データ・セグメントは、特別なアクセス・ゲートを使用せずに、より高い権限を持つプログラムやプロシージャからアクセスすることができます。

4.3.9 GDTやLDTのセグメント・ディスクリプターがROMに格納されている場合、ソフトウェアやプロセ

ッサがROMのセグメント・ディスクリプターを更新（書き込み）しようとすると、プロセッサは無限ループに陥ります。この問題を防止するために、ROMに格納する必要のあるすべてのディスクリプターのアクセス済みビットをあらかじめ設定しておく必要があります。同時に、ROMのセグメントディスクリプターを変更しようとするOSのコードを削除する。

4.3.10 System Descriptor Types

- セグメント・ディスクリプターのSフラグ（ディスクリプター・タイプ）がリセット状態（0）のとき、そのディスクリプター・タイプはシステム・ディスクリプターです。プロセッサは以下のタイプのシステム記述子を認識できます。

- Local Descriptor Table (LDT) segment descriptor;
- Task-state segment (TSS) descriptor;
- Call-gate descriptor;
- Interrupt-gate descriptor;
- Trap-gate descriptor;
- Task-gate descriptor.

これらの記述子は、大きく分けて「システムセグメント記述子」と「ゲート記述子」の2種類があります。システムセグメント記述子は、システムセグメント（LDTやTSSセグメントなど）を指します。ゲートディスクリプターは、「ゲート」を指します。コール、インターラプト、トラップゲートの場合は、コードセグメントのセレクタとそのセグメント内のプログラムエントリポイントのポインタを含み、タスクゲートの場合は、TSSセグメントのセレクタを含みます。表4-4

は、システムセグメント記述子とゲート記述子タイプフィールドのエンコーディングを示しています。

| 表4-4 システムセグメントとゲートディスクリプターのタイプ TYPE Field | | | | | Description |
|--|----|----|---|---|------------------------|
| Decimal | 11 | 10 | 9 | 8 | |
| 0 | 0 | 0 | 0 | 0 | Reserved |
| 1 | 0 | 0 | 0 | 1 | 16-Bit TSS (Available) |
| 2 | 0 | 0 | 1 | 0 | LDT |
| 3 | 0 | 0 | 1 | 1 | 16-Bit TSS (Busy) |
| 4 | 0 | 1 | 0 | 0 | 16-Bit Call Gate |
| 5 | 0 | 1 | 0 | 1 | Task Gate |
| 6 | 0 | 1 | 1 | 0 | 16-Bit Interrupt Gate |
| 7 | 0 | 1 | 1 | 1 | 16-Bit Trap Gate |
| 8 | 1 | 0 | 0 | 0 | Reserved |
| 9 | 1 | 0 | 0 | 1 | 32-Bit TSS (Available) |
| 10 | 1 | 0 | 1 | 0 | Reserved |
| 11 | 1 | 0 | 1 | 1 | 32-Bit TSS (Busy) |
| 12 | 1 | 1 | 0 | 0 | 32-Bit Call gate |
| 13 | 1 | 1 | 0 | 1 | Reserved |
| 14 | 1 | 1 | 1 | 0 | 32-Bit Interrupt Gate |
| 15 | 1 | 1 | 1 | 1 | 32-Bit Trap Gate |

TSSのステータス・セグメントとタスク・ゲートの使用については、「タスク管理」の項で説明します。コールゲートの使用については、保護のセクションで説明します。割り込みとトラップゲートの使用は、割り込みと例外処理で使用されます。セクションで説明します。

4.4 Paging

ページング機構は、80X86のメモリ管理機構の第2の部分である。ページング機構は、セグメンテーション機構に基づいて、仮想（論理）アドレスから物理アドレスへの変換処理を完成させます。セグメンテーション機構は、論理アドレスをリニアアドレスに変換し、ページングはリニアアドレスを物理アドレスに変換します。ページングは、あらゆる種類のセグメントモデルに使用できます。プロセッサのページングメカニズムは、セグメントがマッピングされたリニアアドレス空間を分割し、これらのリニアアドレス空間のページを物理アドレス空間のページにマッピングします。ページングメカニズムのいくつかのページレベルの保護手段は、セグメント保護メカニズムと組み合わせて使用したり、セグメントメカニズムの保護手段を置き換えることができます。例えば、読み取り/書き込みの保護は、ページ単位で強化することができます。さらに、ページ単位では、ページングメカニズムは、ユーザー・スーパーユーザーの2レベルの保護も提供します。

ページング機構は、コントロールレジスタCR0のPGビットを設定することで有効になります。

PG=1の場合、ページングが有効になり、プロセッサはこのセクションで説明するメカニズムを使ってリニアアドレスを物理アドレスに変換します。PG=0の場合、ページングメカニズムは無効となり、セグメンテーションメカニズムによって生成されたリニアアドレスが物理アドレスとして直接使用されます。

前述のセグメンテーション機構は、さまざまな可変サイズのメモリ領域で動作します。フラグメンテーションとは異なり、ページングメカニズムは固定サイズのメモリブロック（ページと呼びます）に対して動作します。ページングメカニズムでは、リニアアドレス空間と物理アドレス空間をページに分割します。線形アドレス空間のどのページも

物理アドレス空間の任意のページ。図4-16は、ページング機構が線形アドレス空間と物理アドレス空間をページに分割し、これら2つの空間の間に任意のマッピングを行う様子を示しています。図中の矢印は、リニアアドレス空間のページと物理アドレス空間のページを対応させています。

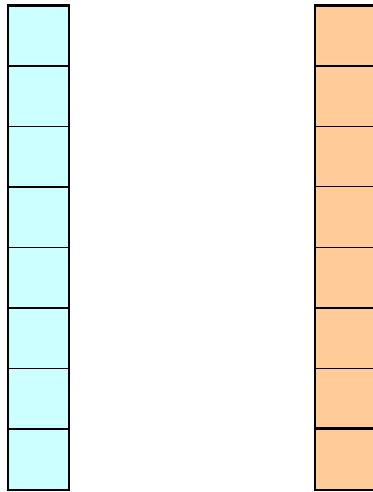


図4-16 線形アドレス空間と物理アドレス空間のページの対応関係を示す図

80X86では、4K (2^{12}) バイトの固定サイズのページを使用し、4Kアドレス境界でアラインされています。つまり、ページングメカニズムは、 2^{32} バイト (4GB) のリニアアドレス空間を 2^{20} (1M = 1048576) ページに分割します。ページングメカニズムは、リニアアドレス空間のページを物理アドレス空間に再配置することで動作します。4Kページは4K境界にアラインされた1つのユニットとしてマッピングされるため、リニアアドレスの下位12ビットは物理アドレスの下位12ビットと同様にページ内オフセットとして直接使用することができます。ページング機構が行う再配置機能は、リニアアドレスの上位20ビットを、対応する物理アドレスの上位20ビットに変換することと考えられます。

ページングを行うと、プロセッサはリニアアドレス空間を固定サイズのページ（長さ4KB）に分割し、物理メモリやディスクストレージにマッピングすることができます。プログラム（またはタスク）がメモリ上の論理アドレスを参照すると、プロセッサはその論理アドレスをリニアアドレスに変換し、ページングメカニズムを使用してリニアアドレスを対応する物理アドレスに変換します。線形アドレスを含むページが物理メモリに存在しない場合、プロセッサはページフォルト例外を生成します。ページフォルト例外ハンドラは、通常、オペレーティングシステムに、対応するページをディスクから物理メモリにロードさせます（動作中に物理メモリの異なるページをディスクに書き込むこともあります）。ページが物理メモリにロードされた後、例外ハンドラからのリターンにより、例外の原因となった命令が再実行されます。プロセッサがリニアアドレスを物理アドレスに変換し、（必要に応じて）ページフォルト例外を生成するために使用する情報は、メモリに格納されているページ

ディレクトリとページテーブルに含まれています。

ページングとセグメント化の最大の違いは、ページングでは固定長のページを使用することです。

セグメント化されたアドレス変換のみを使用した場合、物理メモリに格納されたデータ構造には、そのすべての部分が含まれます。しかし、ページングを使用した場合、1つのデータ構造の一部を物理メモリに格納し、別の一部をディスクに格納することができます。

アドレス変換に必要なバスサイクル数を削減するために、直近にアクセスされたページディレクトリとページテーブルは、TLB (Translation Lookaside Buffer) と呼ばれるプロセッサバッファに格納されます。

TLBは

- 4.4.1** TLBは、バスサイクルを使わずに、ほとんどの読み取りページディレクトリとページテーブル要求を満たすことができます。TLBに必要なページテーブルエントリが含まれていない場合のみ、ページテーブルエントリをメモリから読み込むために余分なバスサイクルが使用されます。これは通常、ページテーブルエントリが長い間アクセスされていない場合に起こります。

4.4.2 Page Table Structure

ページング変換は、メモリ上に存在するテーブルによって記述されます。このテーブルはページテーブルと呼ばれ、物理的なアドレス空間に格納されています。ページテーブルは、 2^{20} の項目を持つ単純な配列と見なすことができます。線形アドレスから物理アドレスへのマッピング機能は、簡単に言えば配列の検索と見なすことができます。リニアアドレスの上位20ビットは、この配列のインデックスを形成し、対応するページの物理（ベース）アドレスを選択するために使用されます。リニアアドレスの下位12ビットはページ内のオフセットを表し、これにページのベースアドレスが加わり、最終的に対応する物理アドレスが形成されます。ページベースアドレスは4Kバウンダリにアラインされているため、ページベースアドレスの下位12ビットは0でなければなりません。つまり、20ビットのページベースアドレスと12ビットのオフセット接続を組み合わせて、対応する物理アドレスを得ることになります。

- 4.4.1.1** ページテーブルの各エントリは、32ビットのサイズを持っています。ページの物理ベースアドレスを格納するのに必要なのは20ビットだけなので、残りの12ビットは、ページが存在するかどうかなどの属性情報を格納するのに使用できます。リニアアドレスインデックスページテーブルの項目が存在するとマークされていれば、その項目は有効であり、そのページの物理アドレスを取得することができます。アイテムが存在しないことを示している場合は、対応する物理ページにアクセスする際に例外が発生します。

4.4.1.2 Two-Level Page Table Structure

ページテーブルは 2^{20} (1M) 個のエントリを持ち、それぞれが4バイト (32ビット) を占める。これらが1つのテーブルとしてのみ格納されている場合、最大で4MBのメモリを占有することになる。そこで、80X86では、メモリ使用量を削減するために、2階層のテーブルを使用しています。このように、20ビットの上位リニアアドレスから物理アドレスへの変換も、1ステップあたり10ビットを使って2ステップで行われます。

第1階層のテーブルはページディレクトリと呼ばれる。1ページを4バイト長の 2^{10} (1K) 個のエントリで占めます。これらのエントリは、対応する2次テーブルを指しています。リニアアドレスの上位10ビット (ビット31~22) は、1次テーブル (ページディレクトリ) のインデックス値として使用され、 2^{10} 個の2次テーブルのいずれかを選択します。

第2レベルのテーブルはページテーブルと呼ばれる。ページテーブルの長さも1ページで、最大で1K個の4バイトエントリが含まれています。各4バイトのテーブルエントリには、関連するページの20ビットの物理ベースアドレスが含まれています。2次ページテーブルでは、リニアアドレスの中間10ビット (ビット21--12) をエントリのインデックスとして使用し、ページの20ビット物理ベースアドレ

スを含むエントリを取得します。20ビットのページ物理ベースアドレスとリニアアドレスの下位12ビット（ページ内オフセット）が組み合わされ、ページ変換処理の出力値、すなわち対応する最終物理アドレスが得られます。

図4-17は、2レベルのテーブルルックアップ処理を示しています。CR3レジスタは、ページディレクトリテーブルのベースアドレスを指定します。リニア・アドレスの上位10ビットは、このページ・ディレクトリ・テーブルのインデックスに使用され、関連する第2レベルのページ・テーブルへのポインタを取得します。線形アドレスの中央10ビットは、物理アドレスの上位20ビットを得るために、2次ページテーブルのインデックスに使用されます。線形アドレスの下位12ビットは、物理アドレスの下位12ビットとして直接使用され、完全な32ビットの物理アドレスを形成します。

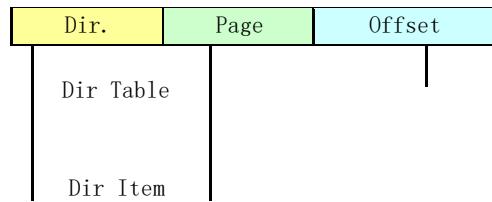


図4-17 リニアアドレス変換

4.4.1.3 Nonexistent Page Tables

2階層のテーブル構造を採用することで、ページテーブルを連続した4MBのメモリブロックに格納することなく、メモリのページ間に分散させることができます。また、リニアアドレス空間に存在しない部分や未使用の部分に2次ページテーブルを割り当てる必要がないのです。ディレクトリページは常に物理メモリ上に存在しなければならないが、2次ページテーブルは必要に応じて再配置することができる。これにより、ページテーブル構造のサイズは、リニアアドレス空間のサイズの実際の使用に対応することになる。

ページ・ディレクトリ・テーブルの各テーブル・エントリは、ページ・テーブルのテーブル・エントリと同様に、存在属性も持っています。ページ・ディレクトリ・エントリのpresence属性は、対応するセカンダリ・ページ・テーブルが存在するかどうかを示す。ディレクトリエントリが、対応するセカンダリページテーブルが存在することを示している場合、セカンダリテーブルにアクセスすることにより、テーブルルックアップ処理の第2ステップが上述のように継続される。対応する二次テーブルが存在しないことを示すビットがある場合、プロセッサは例外を生成してオペレーティングシステムに通知する。ページディレクトリエントリの存在属性により、オペレーティングシステムは、実際に使用されるリニアアドレス範囲に基づいて二次ページテーブルページを割り当てることができる。

4.4.3 ページディレクトリエントリのプレゼンスビットは、2次ページテーブルを仮想メモリに格納するためにも使用できます。これにより、二次ページテーブルの一部だけを物理メモリに格納し、残りはディスクに格納することができます。物理メモリ上のページテーブルに対応するページディレクトリエントリには、ページングが可能であることを示すために「存在」のマークが付けられます。ディスク上のページテーブルに対応するページディレクトリエントリは、存在しないとマークされます。二次ページテーブルが存在しないことによる例外は、ディスクから物理メモリに不足しているページテーブルをロードするようにオペレーティングシステムに通知します。ページテーブルを仮想メモリに保存することで、ページング変換テーブルの保存に必要な物理メモリの量を減らすことができます。

4.4.4 Page-Directory and Page-Table Entries

ページディレクトリエントリとページテーブルエントリのフォーマットを図4-18に示します。31～12ビットは、物理アドレスの上位20ビットで、物理アドレス空間におけるページ（ページフレームとも呼ばれる）の物理ベースアドレスを特定するために使用されます。テーブルエントリの下位12ビットには、ページ属性情報が含まれています。存在属性についてはすでに説明しました。ここでは、残りの属性の機能と用途について簡単に説明します。



図4-18 ページディレクトリとページテーブルエントリのフォーマット

- P – Bit 0 is Present flag. It indicates whether the page or page table pointed to by the table entry is currently loaded into physical memory. After the flag is set, it indicates that the page is in physical memory and performs address translation. When the flag is cleared, it means that the page is not in memory. If the processor tries to access the page, a page fault exception will be generated. At this point the operating system can use the rest of the entry to store information such as the location of the page in the disk system.
- R/W -- Bit 1 is the Read/Write flag. If set to 1, it means the page can be read, written or executed. If 0, the page is read-only or executable. The R/W bit has no effect when the processor is running at the superuser privilege level (level 0, 1 or 2), refer to the U/S flag below. The R/W bit in the page directory entry acts on all pages it maps to.
- U/S -- Bit 2 is the User/Supervisor flag. If set to 1, then the program running on any privilege level can access the page. If 0, the page can only be accessed by programs running on the superuser privilege level (0, 1, or 2). The U/S bit in the page directory entry acts on all pages it maps to.
- A -- Bit 5 is the Accessed flag. This flag of the page table entry is set to 1 when the processor accesses the page mapped by the page table entry. This flag of the page directory entry is set to 1 when the processor accesses any page mapped by the page directory entry. The processor is only responsible for setting this flag, and the operating system can count the usage of the page by periodically resetting the flag.
- D -- Bit 6 is the page modified (Dirty) flag. When the processor performs a write operation on a page, the D flag corresponding to the page table entry is set. The processor does not modify the D flag in the page directory entry.
- AVL – Available field. This field is reserved for program use. The processor will not modify these bits, and the future upgrade processor will not.

4.4.5 Virtual Memory

ページディレクトリおよびページテーブルのエントリにフラグPが存在することで、ページング技術を用いた仮想ストレージに必要なサポートが提供されます。リニアアドレス空間のページが物理メモリに存在する場合、対応するエントリにフラグP=1が設定され、対応する物理アドレスがエントリに含まれます。ページが物理メモリに存在しないテーブルは、そのフラグP=0となります。プログラムが物理メモリに存在しないページにアクセスした場合、プロセッサはページフォルト例外を生成します。このとき、オペレーティングシステムは、この例外処理プロセスを利用して、欠落しているページをディスクから物理メモリに転送し、対応する物理アドレスをテーブルのエントリに格納することができます。

ます。最後に、リターンプログラムが例外を引き起こした命令を再実行する前に、フラグP=1が設定される。

アクセスされたフラグAと修正されたフラグDは、仮想メモリ技術を効果的に実装するために使用することができます。すべてのAフラグを定期的にチェックしてリセットすることで、オペレーティングシステムは最近アクセスされていないページを判断することができます。これらのページは、ディスクに削除する候補となります。あるページがディスクからメモリに読み込まれたとき、そのダーティフラグD=0だったとすると、そのページが再びディスクに移されたとき、Dフラグがまだ0であれば、そのページはディスクに書き込む必要がない。この時にD=1であれば、ページの内容が変更されているので、そのページをディスクに書き込まなければなりません。

4.5 Protection

プロテクトモードでは、80X86はセグメントおよびページレベルの保護機能を備えています。この保護機構は、特権レベル（4レベルの保護とレベル2のページ保護）に基づいて、特定のセグメントとページにアクセス制限を行います。例えば、オペレーティングシステムのコードやデータは、通常のアプリケーションよりも高い特権レベルのセグメントに格納されています。そして、プロセッサの保護機構は、アプリケーションがオペレーティングシステムのコードやデータにアクセスすることを、制御・規制して制限します。

信頼性の高いマルチタスク環境を実現するためには、保護メカニズムが必要です。個々のタスクを相互干渉から保護するために使用することができます。セグメントおよびページレベルの保護は、ソフトウェア開発のどの段階でも使用でき、設計上の問題やエラーの発見・検出を支援します。プログラムがエラーメモリ空間への望ましくない参照を実行した場合、保護メカニズムはそのような操作をブロックし、そのようなイベントを報告することができます。

保護機構としては、セグメント化やページング機構などがあります。プロセッサレジスタの2ビットは、現在実行中のプログラムの特権レベルを定義し、これをCurrent Privilege Level (CPL)と呼びます。セグメント化とページングのアドレス変換時に、プロセッサはCPLを検証します。

コントロールレジスタCR0のPEフラグ（ビット0）をセットすることで、プロセッサをプロテクトモードで動作させることができます。セグメンテーション保護機構をオンにすることができます。プロテクトモードに入ると、プロセッサには保護機構を停止または有効にするための明確な制御フラグはありません。ただし、すべてのセグメント・セレクタとセグメント・ディスクリプタの特権レベルをレベル

0. この方法では、セグメント間の特権レベルの保護バリアを禁止することができますが、他のセグメント長やセグメントタイプのチェック、その他の保護メカニズムはまだ機能します。

コントロールレジスタCR0のPGフラグ（ビット31）をセットすると、ページング機構が有効になります。同様に、ページングオープン状態でページレベル保護機構を無効化または有効化するための関連フラグはプロセッサにありません。しかし、各ページディレクトリエントリとページテーブルエントリにリード/ライト (R/W) フラグとユーザー/スーパーユーザー (U/S) フラグを設定することで、ページレベルの保護を無効することができます。この2つのフラ

グを設定することで、各ページを任意に読み書きできるようになり、実際にはページレベルの保護が無効になります。

4.5.1 Segment Protection
セグメント・レベルの保護では、プロセッサはセグメント・レジスタのセレクタ（RPLおよびCPL）とセグメント・ディスクリプタのフィールドを使って保護検証を行います。ページング・メカニズムでは、ページ・ディレクトリおよびページ・テーブル・エントリのR/WおよびU/Sフラグが主に保護動作を実行するために使用されます。

4.5.2 Segment Protection

■ 保護機構を使用する場合、各メモリ参照をチェックして、そのメモリ参照がさまざまな保護要件を満たしているかどうかを確認します。このチェック作業はアドレス変換と同時に行われるため、プロセッサのパフォーマンスに影響はありません。実行される保護チェックは、以下のカテゴリーに分けられます。

- Segment limit checks;
- Segment type checks;
- Privilege level checks;
- Restriction of addressable domain;

- Restriction of procedure entry-points;
- Restriction of instruction set.

4.5.1.1

プロテクトに違反すると、すべて例外が発生します。以下のセクションでは、プロテクトモードでの保護メカニズムについて説明します。

4.5.1.2 Segment Length Limit Check

セグメント記述子のセグメント制限長フィールドは、プログラムやプロセスがセグメント外の位置にアドレッシングするのを防ぐために使用されます。セグメント長の有効値は、粒度Gフラグの設定状態に依存します。データ・セグメントの場合、セグメント長はフラグE（拡張方向）とフラグB（デフォルトのスタック・ポインタのサイズおよび上限）にも関係します。E フラグは、データ・セグメント・タイプのセグメント記述子のタイプ・フィールドのビットです。

Gフラグがクリアされている場合（バイトグラニュラリティ）、有効なセグメント長は、20ビットのセグメント記述子の長さフィールドLimitの値となります。この場合、Limitは0から0xFFFFF（1MB）の範囲となります。Gフラグが設定されている場合（4KBページ・グラニュラリティ）、プロセッサはLimitフィールドの値に4Kの係数をかけます。この場合、有効なLimitの範囲は0xFFFから0xFFFFFFFF（4GB）までとなります。Gフラグが設定されている場合、セグメント・オフセット（アドレス）の下位12ビットはLimitと照合されないように注意してください。例えば、セグメント長のLimitが0の場合、0～0FFFのオフセット値は有効です。

エクスパンデッド・データ・セグメントを除き、その他のセグメント・タイプの有効範囲の値は、セグメント内でアクセスが許可されている最後のアドレスで、セグメント長よりも1バイト小さい値となります。セグメント長のフィールドを超えて有効なアドレス範囲を指定すると、一般的な保護例外が発生します。

エクスパンデッドダウンデータセグメントでは、セグメント長は同じ機能を持っていますが、その意味は異なります。ここでは、セグメント長はセグメント内のアクセスできない最後のアドレスを指定するので、Bフラグがセットされている場合、有効なオフセット値の範囲は（有効なセグメントオフセット+1）から0xFFFF FFFFまでとなり、Bがクリアされている場合、有効なオフセット値の範囲は（有効なセグメントオフセット+1）から0xFFFFまでとなります。次の拡張セグメントのセグメント長が0の場合、そのセグメントは最大の長さになります。

セグメントの長さをチェックするだけでなく、プロセッサはディスクリプターテーブルの長さもチェックします。GDTR、IDTR、LDTRの各レジスタには、16ビットの限界値が含まれており、プログラムが記述子テーブルの外の記述子を選択するのを防ぐためにプロセッサが使用します。記述子テーブルの限界長値は、テーブル内の最後の有効バイトを示します。各記述子は8バイト長なので、N個の記述子エントリを含むテーブルは、8N-1の制限値を持つべきである。

4.5.1.3

セレクタの値はゼロでもよい。このようなセレクタは、GDTテーブルの最初の未使用ディスクリプター項目を指します。このヌルのセレクタはセグメント・レジスタにロードできますが、このディスク

リッターを使ってメモリを参照しようとすると、一般保護例外が発生します。

4.5.1.4 Segment Type Checking

セグメントディスクリプターには、ディスクリプターのSフラグとタイプフィールドのTYPEの2箇所にタイプ情報が含まれています。プロセッサはこの情報を用いて、セグメントやゲートの不正使用によるプログラミング・エラーを検出します。

Sフラグは、記述子がシステムタイプかコードタイプかデータタイプかを示すために使用されます。TYPEフィールドには、コード、データ、システムの各タイプの記述子を定義するための4ビットが追加されています。前節の表は、コードおよびデータ記述子のTYPEフィールドのエンコーディングを示し、もう一つの表は、システム記述子のTYPEフィールドのエンコーディングを示しています。

1. セグメントセレクタやディスクリプタが操作されると、プロセッサはいつでもタイプ情報をチェックします。タイプ情報がチェックされるのは、主に次の2つのケースです。
 2. When a segment selector is loaded into a segment register. Certain segment registers can contain only certain descriptor types, for example:

- The CS register can only be loaded with a selector for a executable code segment;
 - The selector of the unreadable executable segment cannot be loaded into the data segment register;
 - Only selectors of writable data segments can be loaded into the SS register.
3. When instructions access segments whose descriptors are already loaded into segment registers. Certain segments can be used by instructions only in certain predefined ways, for example:
- No instruction can write an executable segment;
 - No instruction can write into a data segment where the writable bit is not set;
 - No instruction can read an executable segment unless the executable the readable flag is set.

4.5.15 Privilege Levels

プロセッサのセグメント保護機構は、0～3段階の4つの特権レベル（または特権層）を識別できます。数字が大きいほど、特権は少なくなります。図4-19は、これらの特権レベルを保護リングの形態として解釈したものです。中央のリング（最先端のコード、データ、スタックを保持）は、最も重要なソフトウェアを含むセグメントに使用され、通常はオペレーティングシステムのコア部分に使用されます。中央の2つのリングは、より重要なソフトウェアに使用されます。2つの特権レベルのみを使用するシステムでは、特権レベル0と3を使用する必要があります。

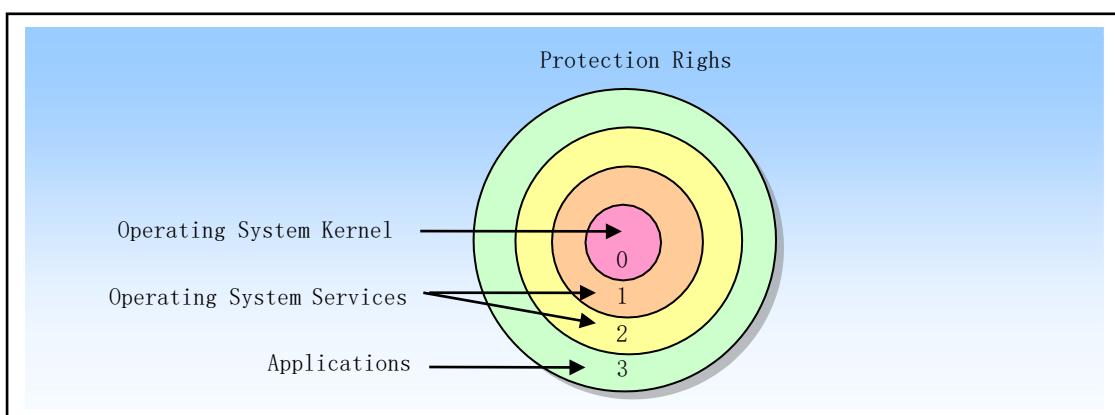


図4-19 プロテクション・レベル・リング

プロセッサは特権レベルを利用して、制御された条件下でない限り、低い特権レベルで実行されているプログラムやタスクが高い特権レベルのセグメントにアクセスできないようにしています。プロセッサは、特権レベルに違反する操作を検出すると、一般保護例外を生成します。

- 個々のコードセグメントとデータセグメントの間で特権レベルのチェックを行うために、プロセッサは以下の3種類の特権レベルを認識することができます。
 - **Current Privilege Level (CPL).** The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL.
 - **Descriptor Privilege Level (DPL).** The DPL is the privilege level of a segment or gate. It is stored in

the DPL field of the segment or gate descriptor for the segment or gate. When the currently executing

- ◆ コード・セグメントがセグメントまたはゲートにアクセスしようとすると、セグメントまたはゲートのDPLが、（このセクションで後述する）セグメントまたはゲート・セレクタのCPLおよびRPLと比較されます。DPLは、アクセスするセグメントまたはゲートのタイプによって、異なる解釈がなされます。
 - ◆ **Data Segment.** Its DPL indicates the numerically highest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.
 - ◆ **Nonconforming code segment (without using a call gate).** The DPL indicates the privilege level that a program or task must have to access the segment. For example, if the DPL of a nonconforming code segment is 0, then only programs running at CPL 0 can access this segment.
 - ◆ **Call Gate.** Its DPL indicates the numerically highest privilege level at which the current executing program or task accessing the call gate can be. (This is the same as the access rule for the data segment.)
 - ◆ **Conforming code segment and nonconforming code segment accessed through a call gate.** The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.
 - ◆ **Task status segment TSS.** Its DPL indicates the numerically highest privilege level at which the current executing program or task accessing the TSS can be. (This is the same as the access rule for the data segment.)
- **Request privilege level RPL.** The RPL is an override privilege level assigned to a segment selector, which is stored in bits 0 and 1 of the selector. The processor checks both the RPL and the CPL to determine if access to a segment is allowed. Even if a program or task has sufficient privilege level (CPL) to access a segment, access will be denied if the provided RPL privilege level is insufficient. That is, if the RPL of the segment selector has a value greater than the CPL, the RPL will overwrite the CPL (and use the RPL as the privilege level for checking comparisons), and vice versa. That is, the privilege level with the largest value in the RPL and CPL is always taken as the comparison object when accessing the segment. Therefore, RPL can be used to ensure that high privileged code does not access a segment on behalf of the application unless the application itself has access to the segment.

4.5.3 特権レベルチェック動作は、セグメント記述子のセグメント・セレクタがセグメント・レジスタにロードされたときに実行されますが、データ・アクセスのチェック方法は、コード・セグメント間のプログラム制御の転送をチェックする方法とは異なります。そのため、以下の2つのアクセス状況が考えられます。

4.5.4 Privilege Level Check When Accessing Data Segments

データ・セグメントのオペランドにアクセスするには、データ・セグメントのセグメント・セレクタをデータ・セグメント・レジスタ (DS, ES, FS, GS) またはスタッカ・セグメント・レジスタ (SS) にロードする必要があります (セグメント・レジスタは、MOV, POP, LDS, LES, LFS, LGS, LSS命令でロードできます)。プロセッサはセグメント・セレクタをセグメント・レジスタにロードする前に、現在実行中のプログラムやタスクの特権レベル (CPL)、セグメント・セレクタのRPL、セグメントのセグメント記述子のDPLを比較して、特権チェックを行います (図4-20を参照)。DPLが

CPLとRPLの両方よりも数値的に大きいか等しい場合、プロセッサはセグメントセレクタをセグメントレジスタにロードします。それ以外の場合は、一般保護フォルトが生成され、セグメント・レジスタはロードされません。

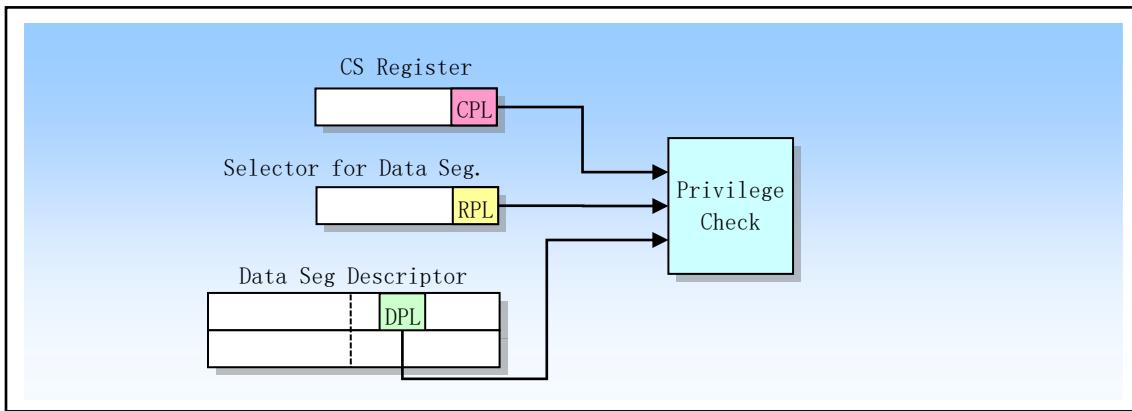


図 4-20 データセグメントへのアクセス時の特権レベルチェック

プログラムやタスクのアドレス可能領域は、CPLが変わると変化することがわかります。CPLが0のときは、この時点ですべての特権レベルのデータセグメントにアクセスでき、CPLが1のときは、特権レベル1～3のデータセグメントのみにアクセスでき、CPLが3のときは、特権レベル3のデータセグメントのみにアクセスできます。

1. また、コードとデータがROM化されている場合など、コードセグメントに含まれるデータ構造にアクセスすることが望ましい場合があります。そこで、時にはコードセグメント内のデータにアクセスする必要が出てきます。このとき、コードセグメント内のデータにアクセスするには、以下の方法があります。
 2. Load the selector of a nonconforming, readable, code segment into a data segment register.
 3. Load the selector of a conforming, readable, code segment into a data segment register.
 4. Use the code segment override prefix (CS) to read a readable code segment whose selector is already in the CS register.

データ・セグメントへのアクセスに関する同じルールが方法1にも適用されます。方法2は、コード・セグメントのDPLにかかわらず、一貫したコード・セグメントの特権レベルがCPLと同等であるため、常に有効です。方法3も、CSレジスタで選択されたコード・セグメントのDPLがCPLと同じであるため、常に有効です。

4.5.5 また、スタックセグメントセレクタを使ってSSセグメントレジスタをロードする際にも、特権レベルのチェックが行われます。ここでは、スタック・セグメントに関連するすべての特権レベルがCPLと一致する必要があります。つまり、CPL、スタック・セグメント・セレクタのRPL、スタック・セグメント・ディスクリプタのDPLのすべてが同じでなければなりません。RPLまたはDPLがCPLと異なる場合、プロセッサは一般保護例外を生成します。

4.5.6 Privilege Level Checking When Transferring Program Control

コードセグメント間

あるコード・セグメントから別のコード・セグメントにプログラム制御を移すには、対象となるコード・セグメントのセグメント・セレクタをコード・セグメント・レジスタ (CS) にロードする必

要があります。このロード処理の一環として、プロセッサはターゲット・コード・セグメントのセグメント記述子を検出し、様々な制限、タイプ、および特権レベルのチェックを行います。これらのチェックに合格すると、ターゲット・コード・セグメント・セレクタがCSレジスタにロードされ、プログラムの制御が新しいコード・セグメントに移され、EIPレジスタが指す命令でプログラムの実行が開始されます。

プログラムの制御伝達は、JMP、RET、INT、IRETの各命令と、例外や割り込みの仕組みを使って実装される。例外と割込みは、後述する特別な実装です。ここでは、JMP、CALL、RETSの各命令について説明します。JMPやCALL命令は

■ は、次の4つの方法で別のコードセグメントを参照することができます。

- The target operand contains the segment selector for the target code segment;
- The target operand points to a call gate descriptor, which contains the selector for the target code segment;
- The target operand points to a TSS, which contains the selector for the target code segment;
- The target operand points to a task gate that points to a TSS, which contains the selector for the target code segment;

4.5.3.1 前者2つの参照タイプについては以下で説明し、後者2つの参照タイプについてはタスク管理のセクションで説明します。

4.5.3.2 Direct Calls or Jumps to Code Segments

JMP、CALL、RET 命令の near フォームは、現在のコードセグメント内でプログラム制御の転送を行うだけなので、特権レベルのチェックは行われません。far形式のJMP、CALL、RET命令は、別のコードセグメントに制御を移すので、プロセッサは特権レベルのチェックを行う必要がある。

コールゲートを経由せずにプログラム制御を他のコードセグメントに移す場合、プロセッサは図4-21に示すように4種類の特権レベルとタイプの情報を確認します。

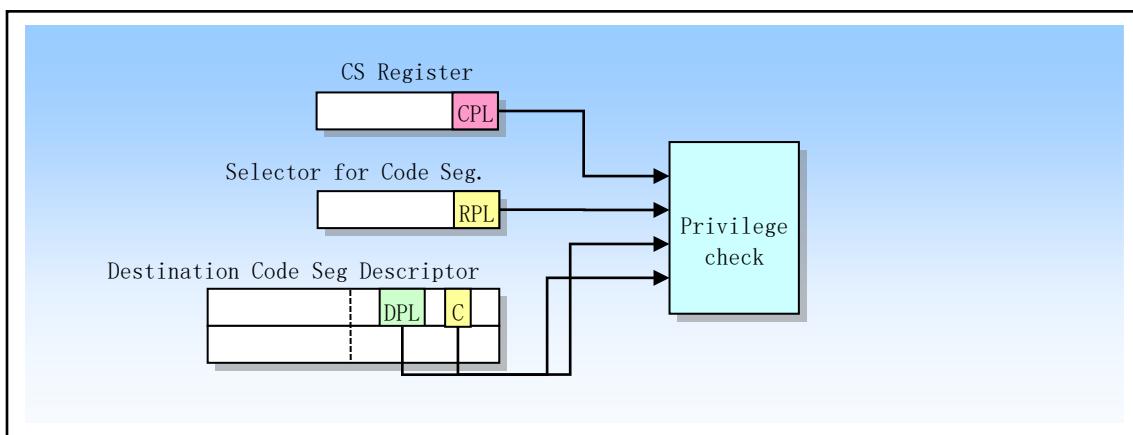


図 4-21 他のコードセグメントを呼び出したり、直接ジャンプしたりする際の特権チェック

- Current privilege level CPL. (Here, CPL is the privilege level of the code segment that executes the call, that is, the code segment that executes the call or jump.)
- The descriptor privilege level DPL of the segment descriptor for the destination code segment that contains the called procedure.
- Request privilege level RPL in the segment selector of the destination code segment.
- The conforming flag C in the destination code segment descriptor. It determines whether a code segment is a non-conforming code segment or a consistent code segment.

プロセッサがCPL、RPL、およびDPLをチェックするルールは、フラグCの設定状態に依存します。不適合コード・セグメントにアクセスする場合 (C=0) 、呼び出し元 (プログラム) のCPLは、宛先コード・セグメントのDPLと等しくなければならず、そうでない場合は一般保護例外が発生します。適合しないコード・セグメントを指すセグメント・セレクタのRPLは、チェックに限定的な影響を与えます。制御転送が正常に完了するためには、RPLは呼び出し側のCPL以下の数値でなければなりません。

ん。不適合コード・セグメントのセグメント・セレクタがCSレジスタにロードされても、特権レベル・フィールドは変更されず、すなわち呼び出し側のCPLのままでです。これは、セグメント・セレクタのRPLがCPLと異なっていても同様です。

適合するコード・セグメント(C = 1)にアクセスする場合、呼び出し側のCPLは、デステイネーション・コード・セグメントのDPLよりも数値的に大きくても小さくとも構いません。プロセッサは、CPL < DPL の場合にのみ一般保護例外を生成します。適合するコード・セグメントへのアクセスでは、プロセッサは RPL のチェックを無視します。適合するコード・セグメントの場合、DPL は呼び出し元がコード・セグメントへの呼び出しを成功させることができる、数値的に最も低い特権レベルを表します。

プログラム制御が適合するコード・セグメントに移されたとき、デステイネーション・コード・セグメントのDPLがCPLよりも数値的に小さい場合でも、CPLは変更されません。これは、CPLが現在のコード・セグメントのDPLと同じでない場合がある唯一のケースです。また、CPLが変化していないので、スタックが切り替わることもありません。

4.5.3.3

ほとんどのコード・セグメントは非適合コード・セグメントです。これらのセグメントでは、プログラムの制御は、以下に説明するようにコールゲートを介した転送でない限り、同じ特権レベルのコードセグメントにしか転送できません。

4.5.3.4 Gate Descriptors

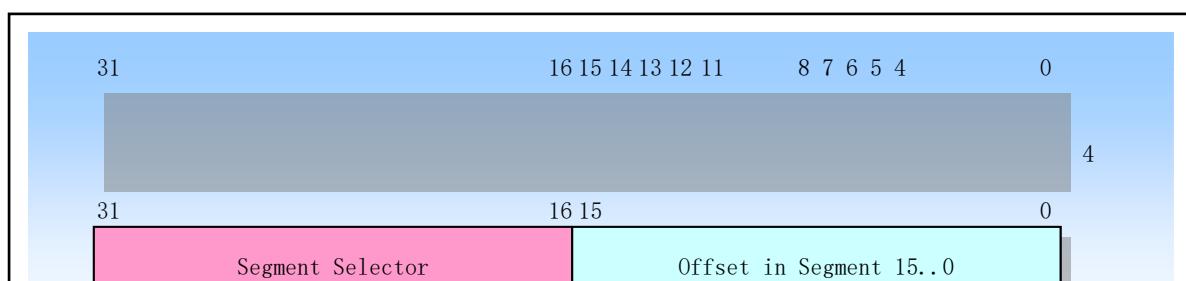
異なる権限レベルのコードセグメントへのアクセスを制御するために、プロセッサはゲートディスクリプタと呼ばれる特別なディスクリプタのセットを提供しています。ゲートディスクリプターには4つの種類があります。

- Call Gate (TYPE=12);
- Trap Gate (TYPE=15);
- Interrupt Gate (TYPE=14);
- Task Gate (TYPE=5).

タスクゲートは、タスクの切り替えに使用され、後に「タスク管理」の項で説明します。トラップゲートとインタラプトゲートは、コールゲートのための特別なクラスで、次のセクションで説明するように、例外や割り込みのハンドラを呼び出すために使われます。このセクションでは、コールゲートの使い方のみを説明します。

■ コールゲートは、異なる特権レベル間で制御されたプログラム制御の転送を実現するために使用されます。コールゲートは通常、特権レベルの保護メカニズムを使用するオペレーティングシステムでのみ使用されます。図4-22にコールゲート記述子のフォーマットを示します。コールゲート記述子は、GDTまたはLDTに格納できますが、割り込み記述子テーブルIDTには配置できません。コールゲートの主な機能は次のとおりです。

- Specifies the code segment to be accessed;
- Defines an entry point for a procedure (program) in a specified code segment;
- Specifies the privilege level that the caller of the access procedure needs to have;
- If a stack switch occurs, it specifies the number of optional parameters that need to be copied between the stacks;
- Indicates whether the call gate descriptor is valid.



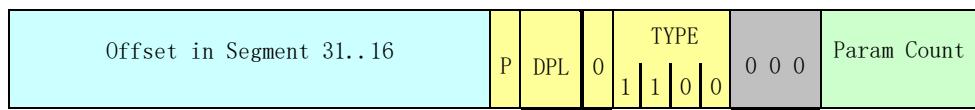


図4-22 コールゲートディスクリプタのフォーマット

コールゲートのセグメント・セレクタ・フィールドでは、アクセスするコード・セグメントを指定します。オフセット値フィールドでは、セグメント内のエントリ・ポイントを指定します。このエントリ・ポイントは通常、指定されたプロセスの最初の命令です。DPLフィールドは、コールゲートの特権レベルを指定し、コールゲートを介して特定のプロシージャにアクセスするために必要な特権レベルを指定する。フラグPは、コールゲート記述子が有効であるかどうかを示す。パラメーター・カウント・フィールド (Param Count) は、スタック・スイッチが発生したときに、呼び出し元のスタックから新しいスタックにコピーされたパラメーターの数を示します。

4.5.35

コールゲートは、Linuxカーネルでは使用されていません。コールゲートの説明は、次節の割込みや例外ゲートの処理に備えるためのものです。

4.5.36 Accessing a Code Segment Through a Call Gate

コールゲートにアクセスするためには、CALLやJMP命令のオペランドのfarポインタを用意する必要があります。このポインタのセグメントセレクタは、コールゲートを指定するために使用されます。ポインタのオフセット値が必要ですが、プロセッサはこれを使用しません。このオフセット値は、任意の値に設定できます。図4-23を参照してください。

プロセッサがコールゲートにアクセスすると、コールゲート内のセグメントセレクタを使用して、宛先コードセグメントのセグメントディスクリプタを探します。次に、コード・セグメント記述子のベース・アドレスとコール・ゲートのオフセット値を組み合わせて、コード・セグメント内の指定されたプログラム・エントリ・ポイントのリニア・アドレスを形成します。

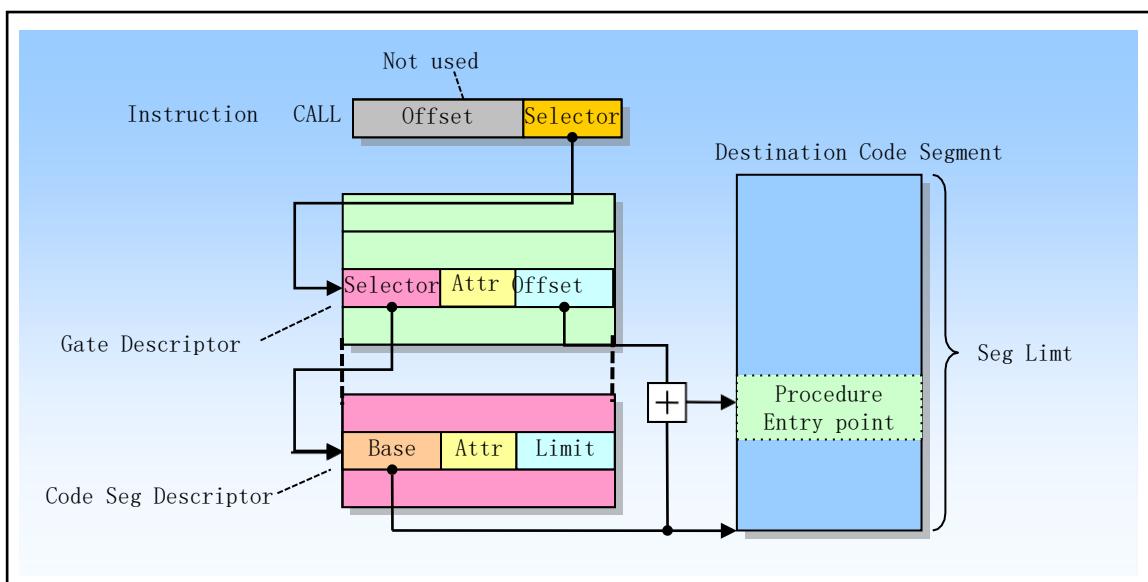


図4-23 コールゲート操作プロセス

- コールゲートによってプログラムの制御権が移ると、CPUは図4-24のように4つの異なる特権レベルをチェックして制御権移譲の有効性を判断します。
 - The current privilege level CPL;
 - The requestor's privilege level RPL of the call gate's selector;
 - The descriptor privilege level DPL of the call gate descriptor;
 - The DPL of the segment descriptor of the destination code segment.

また、デスティネーションコードセグメント記述子の適合性フラグCもチェックされます。

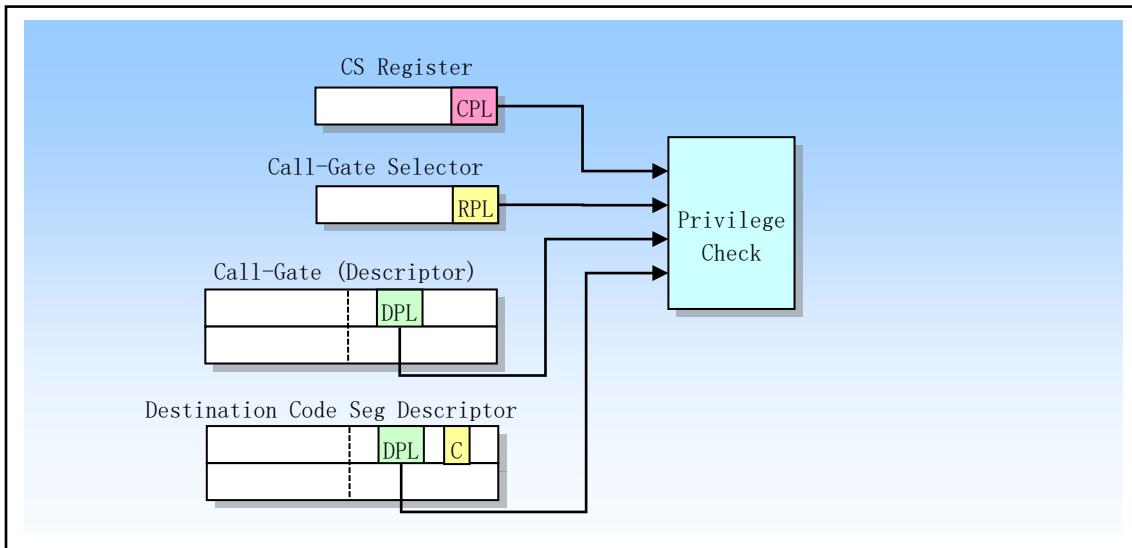


図4-24 コールゲートによる制御転送の特権レベルチェック

表4-5に示すように、CALL命令とJMP命令による制御転送では、特権レベルのチェックルールが異なる。コールゲートディスクリプタのDPLフィールドは、呼び出し側がコールゲートにアクセスできる最大の特権レベル（最低特権レベル）を数値で示す。つまり、コール・ゲートにアクセスするためには、呼び出し側プログラムの特権レベルCPLがコール・ゲートのDPL以下である必要があります。呼び出しゲートのセグメントセレクタのRPLも、これを起動するCPLと同じルールに従う必要があります。

| 表4-5 CALLおよびJMP命令の特権レベルチェックルール Instruction | Privilege Check Rules (numerically) |
|--|--|
| CALL | CPL<= Call gate DPL; RPL<= Call gate DPL Destination conforming & nonconforming code segments DPL<= CPL |
| JMP | CPL<= Call gate DPL; RPL<= Call gate DPL Destination conforming code segment DPL<= CPL; Destination nonconforming code segment DPL=CPL |

呼び出し元とコールゲートの間の特権レベルチェックが成功すると、CPUは次に呼び出し元のCPLとコードセグメント記述子のDPLを比較する。この点で、CALL命令とJMP命令のチェックルールは異なる。CALL命令のみ、コールゲートを使用して、より特権的な（数値的に低い特権レベルの）不適合コードセグメントにプログラム制御を移すことができる、つまり、CPLよりも小さいDPLを持つ不適合コードセグメントに移すことができる。JMP命令は、CPLと等しいDPLを持つ不適合コード・セグメントにプログラム制御を移すためにのみ、コール・ゲートを使用することができる。ただし、

CALL命令とJMP命令の両方とも、より高い特権レベルの適合コード・セグメント、つまりDPLがCPL以下の数値である適合コード・セグメントに制御を移すことができる。

コールがより高い特権レベルの非適合コード・セグメントに制御を移す場合、CPLは宛先コード・セグメントのDPL値に設定され、スタック・スイッチを引き起こします。しかし、コールやジャンプがより高い特権レベルの適合コード・セグメントに制御を移す場合、CPLは変化せず、スタック・スイッチの原因とはなりません。

コールゲートは、コードセグメント内のプロシージャを、異なる特権レベルのプログラムがアクセスできるようにするものです。例えば、コードセグメントに配置されたオペレーティングシステムのコードは、オペレーティングシステム自身や

4.5.7

アプリケーションソフトがアクセスを許可しているコード（キャラクターI/Oを処理するコードなど）。このように、すべての特権レベルのコードがアクセスできるように、これらすべてのプロシージャにコールゲートを設定することができます。さらに、より高い特権レベルのコールゲートを、オペレーティングシステムでのみ使用されるコード専用に設定することもできます。

4.5.8 Stack Switching

コールゲートを使用して、より特権的な不適合コードセグメントにプログラム制御を移すと、CPUは自動的に目的のコードセグメントの特権レベルのスタックに切り替わります。スタック切り替え動作の目的は、より特権的なプログラムがスタック容量不足でクラッシュするのを防ぐことと、低特権レベルのプログラムが共有スタックを介して意図的または非意図的に高特権レベルのプログラムを妨害するのを防ぐことにあります。

各タスクは最大4つのスタックを定義する必要があります。1つは特権レベル3で動作するアプリケーションコード用、もう1つは特権レベル2、1、0にそれぞれ使用されます。システム内で特権レベル3と0の2つだけを使用する場合は、各タスクには2つのスタックだけを設定する必要があります。各スタックは異なるセグメントにあり、セグメントセレクタとセグメント内のオフセット値を使って指定します。

特権レベル3のプログラムの実行時には、特権レベル3のスタックのセグメントセレクタとスタックポインタがそれぞれSSとESPに格納されており、スタックスイッチが発生すると呼び出されたプロシージャのスタックに保存されます。

特権レベル0、1、2のスタックの初期ポインタ値は、現在実行中のタスクのTSSセグメントに格納されています。TSSセグメント内のこれらのポインタは、読み取り専用の値です。タスクの実行中にCPUがこれらを変更することはできません。上位の特権レベルのプログラムが呼び出されると、CPUはこれらを使って新しいスタックを構築します。呼び出したプロシージャから戻るときには、対応するスタックは存在しません。次にプロシージャが呼ばれたとき、TSSの初期ポインタ値を使って再び新しいスタックが作られます。

- オペレーティングシステムは、使用されるすべての特権レベルのスタックとスタックセグメント記述子を確立し、タスクのTSSに初期ポインタ値を設定する責任があります。各スタックは、読み取りと書き込みが可能で、以下の情報の一部を保持するのに十分なスペースが必要です。

- The contents of the SS, ESP, CS, and EIP registers for the calling process;
- The parameters of the called procedure and the space required for the temporary variables;
- The EFLAGS register and error code, when implicit calls are made to an exception or interrupt handler.

1つのプロシージャが他のプロシージャを呼び出すことができ、オペレーティングシステムは複数の割り込みのネストをサポートすることができるので、各スタックは上記の情報の複数のフレームを収容するのに十分なスペースを持っていなければなりません。

1. ゲートへの呼び出しによって特権レベルが変更されると、CPUは以下の手順でスタックを切り替え、呼び出されたプロシージャを新しい特権レベルで実行し始めます（図4-25参照）。

2. Select the pointer to the new stack from the TSS using the DPL of the destination code segment (ie the new CPL). The segment selector and stack pointer of the new stack are read from the current TSS. Any error that violates the segment boundary will result in an invalid TSS exception during the process of reading the stack segment selector, stack pointer, or stack segment descriptor;
3. Check if the stack segment descriptor privilege level and type are valid. If invalid, an invalid TSS exception is also generated.
4. Temporarily save the current values of the SS and ESP registers, and load the segment selector and stack pointer of the new stack into the SS and ESP. Then push the temporarily saved SS and ESP content onto the new stack.
5. Copy the specified number of parameters in the call gate descriptor from the calling procedure stack to the new stack. The value of the parameter in the call gate is up to 31. If the number is 0, it means no parameter and no copy is needed.
6. Push the return instruction pointer (ie the current CS and EIP content) onto the new stack. The new

(デスティネーション)コードセグメントセレクタがCSにロードされ、コールゲートのオフセット値(新規命令ポインタ)がEIPにロードされます。そして、呼び出されたプロセスの実行が開始されます。

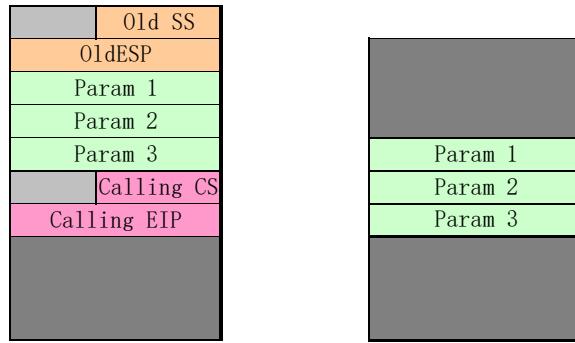


図 4-25 異なる権限レベル間で呼び出すときのスタックスイッチ

4.5.3.9 Returning from a Called Procedure

RET 命令は near return、far return with privilege level、far return with different privilege level を実行するために使用されます。この命令は、CALL 命令で呼び出されたプロシージャから戻るために使用されます。ニアリターンでは、現在のコードセグメント内のプログラム制御を移すだけなので、CPUは境界チェックのみを行います。同じ権限レベルのファーリターンの場合、CPUはリターンコードセグメントのセレクタとリターン命令ポインタを同時にスタックからポップします。これらの2つのポインタは、通常CALL命令によってスタックにプッシュされるため、これによって有効となります。ただし、現在のプロセスがポインタの値を変更する可能性がある場合や、スタックに問題がある場合に対処するため、CPUは依然として特権レベルのチェックを行います。

1. 特権レベルの変更となるファーリターンは、低特権レベルのプログラムに戻ることのみが許されます。すなわち、返されたコードセグメントDPLがCPLよりも数値的に大きい場合です。CPUはCSレジスターのセレクタのRPLフィールドを使用して、低特権レベルが必要かどうかを判断します。RPLの値がCPLよりも大きい場合、特権レベル間のリターン操作が実行されます。実行が呼び出したプロセスにマークに戻ると、CPUは次のステップを実行します。
2. Check the RPL field value in the saved CS register to determine if the privilege level needs to be changed on return.
3. Pop up and load the CS and EIP registers using the values on the called procedure stack. The privilege level and type checking of the code segment descriptor and the code segment selector RPL are performed during this process.
4. If the RET instruction contains a parameter count operand and the return operation changes the privilege level, then the parameter count value is added to the ESP register after the CS and EIP values in the pop-

up stack to skip the caller stack's parameter. At this point the ESP register points to the pointer SS and ESP of the originally saved caller stack.

5. Load the saved SS and ESP values into the SS and ESP registers to switch back to the caller's stack. At this time, the SS and ESP values of the caller stack are discarded.
6. If the RET instruction contains a parameter number operand, the parameter value is added to the ESP register value to skip (discard) the parameters on the caller stack.

7. Check the contents of the segment registers DS, ES, FS and GS. If there is a segment pointing to a DPL that is smaller than the new CPL (except for the consistent code segment), then the CPU loads the segment register with the NULL selector.

4.5.7 Page-Level Protection

ページディレクトリとページテーブルのエントリにある読み取り/書き込みフラグR/Wとユーザー/スーパーバイザーフラグU/Sは、セグメンテーションメカニズムの保護属性のサブセットを提供します。ページング・メカニズムでは、2つのレベルの権限しか認識しません。特権レベル0、1、2はスーパーユーザーレベルに分類され、特権レベル3は通常のユーザーレベルに分類されます。通常のユーザーレベルのページは、読み取り専用/実行可能、または読み取り可能/書き込み可能/実行可能としてマークすることができます。スーパーユーザーレベルのページは、表4-6に示すように、スーパーユーザーにとっては常に読み取り可能/書き込み可能/実行可能ですが、一般ユーザーにとってはアクセスできません。

セグメント化の仕組みとしては、一番外側のユーザーレベルで実行されるプログラムは、ユーザーレベルのページにしかアクセスできませんが、任意のスーパーユーザーレベル（0、1、2）で実行されるプログラムは、ユーザーレイヤーのページにアクセスできるだけでなく、スーパーユーザーレイヤーのページにもアクセスできます。ユーザー層のページにもアクセスできます。セグメント化メカニズムとは異なり、内側のスーパーユーザーレベルで実行されたプログラムは、ユーザーレベルで読み取り専用/実行可能とマークされているものも含め、どのページに対しても読み取り/書き込み/実行可能な権限を持つ。

| 表4-6 ページ上の通常ユーザーとスーパーユーザーのアクセス制限 U/S | R/W | User Access Rights | Supervisor Access Rights |
|---|-----|--------------------|--------------------------|
| 0 | 0 | None | Read/Write/Execute |
| 0 | 1 | None | Read/Write/Execute |
| 1 | 0 | Read/Execute | Read/Write/Execute |
| 1 | 1 | Read/Write/Execute | Read/Write/Execute |

80X86のアドレス変換機構全体の中で、ページング機構がセグメンテーション機構の後に実装されているように、ページレベルの保護もセグメンテーション機構の後の保護の役割を担っています。ま

ず、すべてのセグメントレベルの保護がチェックされ、テストされます。そのチェックに合格すると、ページレベルの保護のチェックが行われます。例えば、メモリ上のバイトがレベル3のプログラムからアクセス可能なセグメント内にあり、かつユーザーレベルのページとしてマークされている場合にのみ、レベル3のプログラムからアクセスすることができます。ページへの書き込みは、セグメンテーションとページングの両方が許可されている場合にのみ実行できます。セグメントが読み取り/書き込みタイプであっても、そのアドレスに対応するページが読み取り専用/実行可能とマークされていれば、そのページへの書き込みはできません。セグメントのタイプが read-only/executable の場合、対応するページに与えられた保護属性にかかわらず、そのページは常に書き込みができません。このように、セグメンテーションやページングの保護機構は、電子回路のシリアルラインのようなもので、接続しないとスイッチが開かないようになっていることがわかります。

同様に、ページの保護属性は、表4-7に示すように、テーブルエントリの「シリアル」または「アンドオペレーション」と、ページテーブルのエントリで構成されています。ページ・テーブル・エントリのU/SフラグとR/Wフラグは、エントリ・マッピングの単一ページに適用される。ページ・ディレクトリ・エントリのU/SフラグとR/Wフラグは、そのディレクトリ・エントリにマッピングされたすべてのページに作用する。ページ・ディレクトリとページ・テーブルの複合保護属性は、2つの属性のAND演算で構成されているため、保護手段は非常に厳しいものとなります。

| 表4-7 ページディレクトリとページテーブルエントリの組み合わせによるページの保護 Dir Entry U/S | Page Entry U/S | Combined U/S | Dir Entry R/W | Page Entry R/W | Combined R/W |
|--|----------------|--------------|---------------|----------------|--------------|
| 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 |

4.5.4.1 Software issues for modifying page table entries

メモリ参照のたびにメモリ上のページテーブルにアクセスしなくとも済むように、プロセッサ内のページ変換キャッシュには、直近に使用されたリニアアドレスから物理アドレスへの変換情報が格納されています。プロセッサは、メモリ上のページテーブルにアクセスする前に、まずバッファキャッシュの情報を使用します。プロセッサは、必要な変換情報がキャッシュ内にない場合にのみ、メモリ内のページディレクトリとページテーブルを検索します。ページ変換キャッシュの別称として、TLB (Translation Lookaside Buffer) があります。

80X86プロセッサは、ページ変換キャッシュとページテーブルのデータの依存関係を維持するのではなく、それらの整合性を確保するためにオペレーティングシステムソフトウェアを必要とする。つまり、プロセッサは、ページテーブルがソフトウェアによって変更されたことを知りません。そのため、オペレーティングシステムは、ページテーブルを変更した後にキャッシュをリフレッシュして、両者の整合性を確保する必要があります。レジスタCR3をリロードするだけで、キャッシュのリフレッシュ動作を完了させることができます。

4.5.8 ページテーブルのエントリを修正しても、ページ変換キャッシュを更新する必要がない特別なケースがあります。すなわち、存在しないページのエントリを修正する際に、ページ変換に有効なエントリであることを示すためにPフラグを0から1に変更したとしても、キャッシュをリフレッシュする必要はないのである。無効なエントリはキャッシュに保存されないので、ディスクからメモリにページを呼び出してページを存在させるときも、ページ変換キャッシュをリフレッシュする必要はない。

4.5.9 Combining page and segment protection

ページングを有効にすると、CPUはまずセグメントレベルの保護を行ってから、ページレベルの保護を処理します。CPUはどのレベルでも保護違反のエラーを検出すると、メモリアクセスを破棄して例外を発生させます。それがセグメント機構で発生した例外であれば、それ以上のページ例外は発生しません。

ページレベルの保護機能は、セグメントレベルの保護機能を置き換えたり、上書きしたりするために使えません。例えば、コードセグメントが書き込み不可能に設定されている場合、コードセグメントがページ化された後、ページのR/Wフラグが読み取り可能、書き込み可能に設定されていても、そのページは書き込み不可能になります。この時点で、セグメント保護のチェックにより、そのページに書き込もうとしてもブロックされます。ページレベルの保護は、セグメントレベルの保護を強化するために使用することができます。例えば、読み書き可能なデータセグメントがページ化されている場合、ページレベルの保護機能を使って個々のページを書き込み保護することができます。

4.6 Interrupt and Exception Handling

割り込みや例外は、システム、プロセッサ、現在の実行者（またはタスク）のどこかで発生した、プロセッサで処理する必要のあるイベントです。多くの場合、このようなイベントによって、実行制御が現在実行中のプログラムから、割り込みハンドラや例外ハンドラと呼ばれる特別なソフトウェア関数やタスクに強制的に移されることがあります。割り込みや例外に対応してプロセッサが行う動作を、割り込み/例外サービス（処理）と呼びます。

通常、割り込みは、ハードウェアからの信号に応じて、プログラムの実行中のランダムなタイミングで発生します。システムのハードウェアは、外部機器へのサービス要求などの外部イベントを処理するために割込みを使用します。もちろん、ソフトウェアでもINT n命令を実行することで割り込みを発生させることができます。

例外は、プロセッサが命令を実行しているときに、以下のようなエラー条件が検出されたときに発生します。

のエラー状態をゼロで割ったものです。プロセッサは、保護機構の違反、ページフォルト、内部マシンエラーなど、さまざまなエラー状態を検出することができます。

4.6.1 アプリケーションやOSにとって、80X86の割り込み・例外処理機構は、発生した割り込みや例外を透過的に処理します。割り込みを受信したり、例外を検出したりすると、プロセッサは自動的に現在実行中のプログラムやタスクを中断し、割り込みや例外ハンドラの実行を開始します。ハンドラが完了すると、プロセッサは再開し、中断されたプログラムまたはタスクの実行を継続します。割り込まれたプログラムの復旧処理は、例外からの復旧が不可能な場合や、割り込みによって現在のプログラムが終了しない限り、プログラムの実行の継続性を失わない。ここでは、プロテクトモードにおけるプロセッサの割り込みと例外の処理機構について説明します。

4.6.2 Sources of Interrupts

■ プロセッサは、2つの場所から割り込みを受け取ります。

- External (hardware generated) interrupts;
- Software generated interrupts.

外部割り込みは、プロセッサチップ上の2つのピン（INTRとNMI）で受信します。ピンINTRが外部割り込み信号を受信すると、プロセッサは外部割り込みコントローラ（8259Aなど）から提供された割り込みベクタ番号をシステムバスから読み込みます。ピンNMIが信号を受信すると、ノンマスカブル割り込みを発生させます。プロセッサのINTR端子で受信する外部割り込みは、割り込みベクタ番号0～255を含めてマスカブルハードウェア割り込みと呼ばれます。フラグレジスタEFLAGSのIFフラグは、これらすべてのハードウェア割り込みをマスクするために使用できます。

INT n命令は、命令オペランドに割り込みベクタ番号を指定することで、ソフトウェアからの割り込みを発生させることができます。例えば、INT 0x80という命令は、Linuxのシステム割り込みコールの割り込み0x80を実行します。この命令のパラメータには、0～255の任意のベクターを使用できます。ただし、プロセッサがあらかじめ定義したNMIベクタを使用した場合、それに対するプロセッサの応答は、通常に生成されるNMI割り込みとは異なります。INT命令にNMIベクタ番号2が使用された場合、NMI割り込みハンドラが呼び出されますが、この時、プロセッサのNMI処理ハードウェアは起動しません。

4.6.3 なお、INT命令を用いてソフトウェアで生成した割り込みは、EFLAGSレジスタのIFフラグではマスクできません。

4.6.4 Sources of Exceptions

■ また、プロセッサが受け取る例外のソースは2つあります。

- Processor-detected program-error exceptions;
- Software-generated exceptions.

アプリケーションやオペレーティングシステムの実行中に、プロセッサがプログラムエラーを検出すると、1つまたは複数の例外が発生します。80X86プロセッサは、検出した各例外に対してベクタ

ーを定義します。例外はさらに、後述するように、フォールト、トラップ、アボートに分類されます。

INTO命令、INT 3命令、BOUND命令は、ソフトウェアから例外を生成するために使用できます。

これらの命令は、命令ストリームの特定のポイントで実行される特別な例外条件をチェックします。

例えば、INT 3命令はブレークポイント例外を発生させます。

INT n命令は、指定された例外をソフトウェアでシミュレートするために使用できますが、1つの制限があります。INT命令のオペランドnが80X86の例外ベクター番号の一つである場合、プロセッサはベクターに関連した例外ハンドラを実行するベクター用の割り込みを生成します。しかし、これは実際には割り込みなので、ベクター関連の

4.6.5 ハードウェアで発生した割り込みは、通常、エラーコードを生成します。エラー・コードを生成する例外については、例外ハンドラは処理中にスタックからエラー・コードをポップ・オフしようとします。そのため、INT命令を使って例外発生をエミュレートすると、ハンドラはEIP（ちょうどエラーコードがない位置）をスタックにポップオフして捨ててしまい、リターン・ポジション・エラーが発生してしまいます。

4.6.6 Exception Classifications

■ 例外は、例外の報告方法や、例外の原因となった命令がプログラムやタスクの継続性を損なわずに再実行できるかどうかによって、Faults、Traps、Abortに細分化されます。

- A Fault is an exception that can usually be corrected and can continue to run once it has been corrected. When a Fault occurs, the processor will restore the state of the machine to the state it was in prior to the instruction that generated the Fault. At this point, the return address of the exception handler will point to the instruction that generated the Fault, not the one that follows. Therefore, the instruction that generated the fault after returning will be re-executed.
- Trap is an exception that causes a trap to be reported immediately after the execution of the trapping instruction. Trap also enables programs or tasks to execute without loss of continuity. The return address for the trap handler points to the next instruction, so the next instruction is executed after the return.
- Abort is an exception that does not always report the exact location of the instruction that caused the exception, and does not allow the program that caused the exception to resume execution. Abort is used to report serious errors such as hardware errors and inconsistencies or illegal values in the system tables.

4.6.7 Exception and Interrupt Vectors

例外や割り込みを処理するために、プロセッサが特別な処理を必要とする定義された例外や割り込み条件には、ベクターと呼ばれる識別番号が与えられています。プロセッサは、ベクターをIDT (Interrupt Descriptor Table) のインデックス番号として使用し、例外や割り込みハンドラのエントリポイントの位置を特定します。

許可されるベクター番号の範囲は0～255です。0～31は80X86プロセッサで定義されている例外や割り込みのために予約されていますが、現在この範囲のベクター番号は機能ごとに定義されておらず、未定義の機能のベクター番号は将来の使用のために予約されます。

ユーザー定義の割り込みには、32～255のベクター番号が使用されます。これらの割り込みは、通常、外部I/Oデバイスが外部のハードウェア割り込みメカニズムを介してプロセッサに割り込みを送信できるようにするために使用されます。

80X86で定義されている例外およびNMI割り込みに割り当てられているベクターを表4-8に示します。それぞれの例外について、例外の種類、エラーコードの生成とスタックへの保存の有無を示しています。また、あらかじめ定義されている各例外およびNMI割り込みのソースも示しています。

| 表4-8 プロ テク | Mnemonic | Description | Type | Error Code | Source |
|------------------|----------|-------------|------|---------------|--------|
| | | | | | |

| トモードでの例外と割り込み Vector No. | | | | | |
|-----------------------------|-----|---------------|------------|----|--|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | Debug | Fault/Trap | No | Any code or data reference or the INT 1 instruction. |
| 2 | -- | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT 3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |

| | | | | | |
|--------|-----|--|-----------|-----------|---|
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined) | Fault | No | UD2 instruction or reserved (new for P6) |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes(Zero) | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | -- | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction (not for CPU after 386) |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or accessing system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register loads. |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |
| 15 | -- | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | Floating-Point Error (Math Fault) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Fault | Yes(Zero) | Any data reference in memory. |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent. |
| 19 | #XF | Streaming SIMD Extensions | Fault | No | SSE and SSE2 floating-point instructions. (for PIII cpu) |
| 20-31 | -- | Intel reserved. Do not use. | | | |
| 32-255 | -- | User Defined (Nonreserved) Interrupts | Interrupt | | External interrupt or INT n instruction. |

4.6.8 Program or Task Restart

例外や割込みの処理後にプログラムやタスクが実行を再開するためには、Abortを除くすべての例外は正確な命令位置を報告でき、すべての割込み保証は命令境界上で発生します。

フォールトクラスの例外では、プロセッサが例外を発生させたときに保存された戻り命令ポインタは、フォールト命令を指します。そのため、フォールト・ハンドラの復帰後にプログラムやタスクが再起動されると、元のフォールト命令が再実行されます。フォルトの原因となった命令の再実行は、通常、アクセス命令のオペランドがロックされている場合の処理に使用されます。フォールトの最も一般的な例は、ページ・フォールト例外です。この例外は、プログラムがメモリ上のページにないオペランドを参照したときに発生します。ページフォルト例外が発生すると、例外ハンドラはページをメモリにロードし、フォルト命令を再実行することでプログラムの実行を再開することができます。再実行が現在の実行プログラムにとって透過的であるように、プロセッサは必要なレジスタとスタッ

ク・ポインタの情報を保存し、フォールト命令を実行する前の状態に戻れるようにします。

Trapクラスの例外では、プロセッサが例外を発生させたときに保存されたリターンポインタは、トラップ動作の原因となった次の命令を指します。以下のような命令の実行中にTrapが検出された場合、Trapの原因となった次の命令を指します。

が制御の移行を行った場合、戻り命令ポインタは制御の移行を反映します。例えば、JMP命令の実行中にTrap例外が検出された場合、リターン命令ポインタはJMP命令の次の命令ではなく、JMP命令のターゲットロケーションを指します。

アボート・クラスの例外は、プログラムやタスクの確実な再起動をサポートしません。例外を中止するハンドラは、通常、例外が発生したときのプロセッサの状態に関する診断情報を収集し、可能な限り適切にプログラムやシステムを終了させるために使用されます。

4.6.9 割り込みは、中断したプログラムを継続性を失わずに再開することを厳密にサポートします。割り込みのために保存された戻り命令ポインタは、プロセッサが割り込みを取得したときに実行される次の命令境界を指します。実行されたばかりの命令に繰り返しのプレフィックスがある場合、現在の反復が終了し、次の反復のためにレジスタが設定されたときに割り込みが発生します。

4.6.10 Enabling and Disabling Interrupts

フラグレジスタEFLAGSのインタラプトイネーブルフラグ (IF) は、プロセッサのINTR端子で受信したマスカブルハードウェア割り込みの処理を無効にすることができます。IF = 0の場合、プロセッサはINTRピンに送られる割り込みを無効にし、IF = 1の場合、INTRピンに送られる割り込み信号はプロセッサによって処理されます。

IFフラグは、NMI端子に送られるノンマスキング割り込みや、プロセッサが発生させる例外には影響しません。EFLAGSの他のフラグと同様に、プロセッサはハードウェアリセット操作に応じてIFフラグをクリアする (IF=0)。

■ IFフラグは、STI命令とCLI命令を用いて設定または解除することができます。これらの2つの命令は、プログラムのCPL <= IOPLのときにのみ実行でき、そうでない場合は一般保護例外が発生する。また、IFフラグは以下の操作によっても影響を受ける。

- The PUSHF instruction can store the EFLAGS contents on the stack, where they can be examined and modified. The POPF instruction can be used to put the contents of the modified flags back into the EFLAGS register.
- The task switch, POPF, and IRET instructions load the EFLAGS register. Therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is processed through the interrupt gate, the IF flag is automatically cleared (reset), which disables the maskable hardware interrupt. However, if an interrupt is handled through the trap gate, the IF flag will not be reset.

4.6.11 Priority of exceptions and interrupts

命令の境界で処理待ちの例外や割込みが複数ある場合、プロセッサは指定された順序で処理します。表4-9に、例外および割込みソースクラスの優先順位を示します。プロセッサはまず、優先度の高いクラスの例外や割込みを処理します。優先度の低い例外は破棄され、優先度の低い割込みは待機します。割り込みハンドラが、例外や割り込みを発生させたプログラムやタスクに戻ると、破棄された例外が再び発生します。

| 表4-9 例外と割り込みの優先順位 Priority | Description |
|-------------------------------|--|
| 1(Highest) | Hardware reset: RESET |
| 2 | Task switching trap: T flag is set in TSS |
| 3 | External hardware intervention |
| 4 | Previous instruction trap: breakpoint, debug trap exception |
| 5 | External interrupt: NMI interrupt, maskable hardware interrupt |

| | |
|-----------|--|
| 6 | Code breakpoint error |
| 7 | Take an instruction error: violation of code segment limit, code page error |
| 8 | The next instruction decode error: instruction length >15 bytes, invalid opcode, coprocessor does not exist |
| 9(Lowest) | Execution instruction error: overflow, boundary check, invalid TSS, segment not present, stack error, general protection, data page, alignment check, floating point exception |

4.6.12 Interrupt Descriptor Table (IDT)

割り込み記述子テーブル (IDT) は、各例外や割り込みベクターと、そのプロセスやタスクのゲート記述子を関連付け、関連する例外や割り込みを処理するために使用されます。GDTやLDTテーブルと同様に、IDTも8バイト長の記述子の配列です。GDTとは異なり、テーブルの最初の項目に記述子を含めることができます。IDTテーブルへのインデックスを形成するために、プロセッサは例外または割込みのベクタ番号を*8に入れます。割り込みや例外のベクターは最大でも256個なので、IDTには256個以上の記述子を含める必要はありません。記述子は例外や割り込みが発生したときにのみ必要なので、IDTは256個より少ない記述子を含むことができます。ただし、IDT内のすべての空の記述子エントリは、その存在ビット（フラグ）をゼロに設定する必要があります。

IDTテーブルはリニアアドレス空間のどこにでも存在することができますが、プロセッサはIDTRレジスタを使用してIDTテーブルの位置を特定します。このレジスタには、図4-26に示すように、IDTテーブルの32ビットのベースアドレスと16ビットの長さ（長さ制限）の値が含まれています。IDTテーブルのベースアドレスは、プロセッサのアクセス効率を高めるために、8バイト境界にアラインする必要があります。長さ制限値は、IDTテーブルの長さをバイト単位で表したものです。

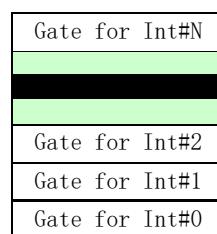


図4-26 インタラプトディスクリプターテーブルIDTとレジスタIDTR

LIDT命令とSIDT命令は、それぞれIDTRレジスタの内容をロードおよびストアするために使用されます。LIDT命令は、メモリ上のリミットとベースアドレスのオペランドをIDTRレジスタにロードしま

す。この命令は、現在の特権レベルCPLが0のコードのみが実行可能で、通常、IDTの作成時にOSの初期化コードで使用されます。SIDT命令は、IDTR内のベースアドレスとリミットコンテンツをメモリにコピーするために使用されます。この命令はどの特権レベルでも実行可能です。割り込みや例外ベクターが参照するディスクリプターがIDTの境界を超えた場合、プロセッサは一般保護例外を発生させます。

4.6.13 IDT Descriptors

- IDTテーブルには、3種類のゲートディスクリプターを格納することができます。

- Interrupt gate descriptor;
- Trap gate descriptor;
- Task gate descriptor.

これら3つのゲート記述子のフォーマットを図4-27に示します。割り込みゲートとトラップゲートには、プロセッサがコードセグメント内の例外や割り込みに対してプログラムの実行権を移すために使用するロングポインタ（つまりセグメントセレクタとオフセット値）が含まれています。この2つのセグメントの主な違いは、プロセッサがEFLAGSレジスタのIFフラグを操作することです。IDTにおけるタスク・ゲート記述子のフォーマットは、GDTおよびLDTにおけるタスク・ゲートのフォーマットと同じです。タスクゲート記述子には、例外や割り込みの処理に使用されるタスクTSSセグメントのセレクタが含まれています。

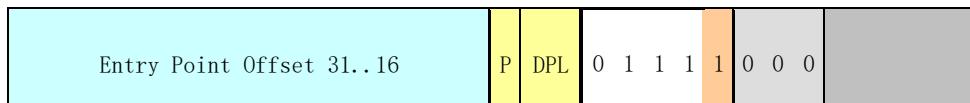
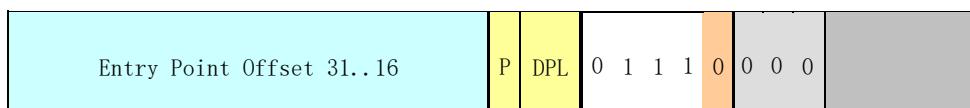


図4-27 インタラプトゲート、トラップゲート、タスクゲートの記述子のフォーマット

4.6.14 Exception and Interrupt Handling

プロセッサが例外や割込みハンドラを呼び出す方法は、CALL命令を使ってプロシージャやタスクを呼び出すのと同じです。例外や割込みに応答する際、プロセッサは例外や割込みのベクターをIDTテーブルのインデックスとして使用します。インデックス値が割込みゲートまたはトラップゲートを指している場合、プロセッサはCALL命令の操作コールゲートと同様の方法で例外または割込みハンドラを呼び出します。インデックス値がタスクゲートを指している場合、プロセッサは、CALL命令操作のタスクゲートと同様の方法でタスクスイッチを行い、例外または割込み処理タスクを実行する。

図4-28に示すように、例外・割込みゲートは、現在のタスクのコンテキストで実行される例外・割込みハンドラを参照します。ゲート内のセグメント・セレクタは、GDT または現在の LDT 内の実行コード・セグメント記述子を指します。ゲート記述子のオフセット・フィールドは、例外または割り込み処理プロセスの開始点を指します。

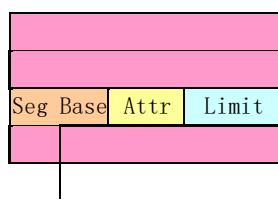


図4-28 インタラプトプロシージャコール

- プロセッサが例外や割込みハンドラの呼び出しを実行すると、以下のような動作が行われます。
 - If the handler procedure will be executed at a high privilege level (such as level 0), then a stack switch operation will occur. The stack switching process is as follows:
 - プロセッサは、現在実行中のタスクのTSSセグメント（例：tss.ss0、tss.esp0）から、割込みハンドラや例外ハンドラが使用するスタックのセグメントセレクタとスタックポインタを取得します。次にプロセッサは、図4-29に示すように、中断されたプログラム（またはタスク）のスタック・セレクタとスタック・ポインタを新しいスタックに PUSHします。次に、プロセッサは EFLAGS、CS、および EIP レジスタの現在の値を新しいスタックに PUSHします。例外によってエラーコードが発生した場合は、そのエラーコードも新しいスタックに PUSHされます。
 - If the handler procedure will run on the same privilege level as the interrupted task, then:

プロセッサは EFLAGS、CS、EIP レジスタの現在の値を現在のスタックに保存します。例外によってエラーコードが発生した場合は、そのエラーコードも新しいスタックにpushされます。

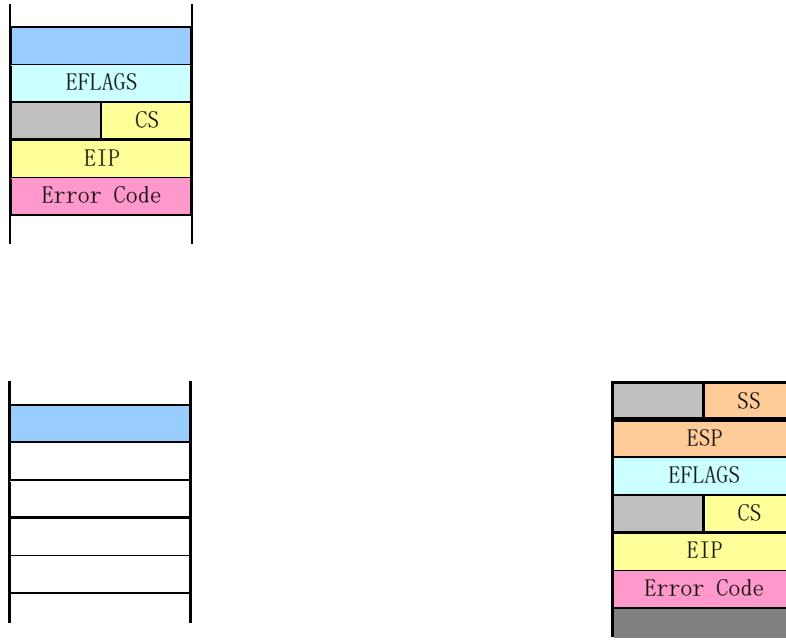


図4-29 割り込み処理に移行する際のスタック使用方法

例外処理や割込み処理の手続きから戻るには、ハンドラはIRET（またはIRETD）命令を使用しなければなりません。IRET命令はRET命令と似ていますが、保存されているフラグをEFLAGSレジスタにリストアする点が異なります。EFLAGSレジスタのIOPLフィールドは、CPLが0の場合のみリストアされ、CPL <= IOPLの場合のみIFフラグが変更されます。ハンドラプロシージャの呼び出し時にスタックスイッチが発生した場合、IRET命令はリターン時に中断されたプロシージャのスタックにスイッチバックします。

4.6.10.1 Protection of exceptions and interrupt handler procedures

- 例外処理や割込みハンドラプロシージャの特権レベル保護メカニズムは、コールゲートを介して通常のプロシージャを呼び出すのと同様です。プロセッサは、CPL特権コード・セグメントよりも下位の割込みハンドラ・プロシージャに制御を移すことを許可しません。そうしないと、一般保護例外が発生します。また、割込みや例外に対する保護機構は、以下の点で一般的なコールゲート手順とは異なります。

- Because the interrupt and exception vectors do not have an RPL, the RPL is not checked when the exception and interrupt handler procedures are implicitly called.
- The processor checks the DPL in the interrupt or trap gate only when an exception or interrupt is generated using the INT n, INT 3, or INTO instruction. At this time, the CPL must be less than or equal to the DPL of the gate. This restriction prevents applications running at privilege level 3 from using software interrupts to access important exception handling procedures, such as page fault handling, assuming that these processes have been placed in a higher privilege level code segment. For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL in the interrupt

gate and trap gate.

通常、例外や割り込みは定期的に発生するものではないので、特権レベルに関するこれらのルールは

- は、例外処理や割り込みハンドラが実行できる特権レベルの制限を効果的に強化します。特権レベルの保護に違反しないように、以下のいずれかの手法を用いることができます。

- Exception or interrupt handlers can be stored in a consistent code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data in the data segment, then privilege level 3 must be able to access this data segment. But there is no protection at all.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler can always be performed regardless of the current privilege level CPL of the interrupted program or task.

4.6.10.2 Flag usage by Exception or Interrupt Handler Procedure

割り込みゲートやトラップゲートを経由して例外や割り込みハンドラにアクセスした場合、プロセッサはEFLAGSレジスタの内容をスタックに保存した後、EFLAGSのTFフラグをクリアします。TFフラグをクリアすることで、命令トレースが割込み応答に影響を与えることを防ぎます。続くIRET命令は、スタックの内容でEFLAGSの元のTFフラグを復元します。

- 割り込みゲートとトラップゲートの唯一の違いは、プロセッサがEFLAGSレジスタのIFフラグを操作する方法です。割り込みゲートを介して例外や割り込みハンドラにアクセスすると、プロセッサはIFフラグをリセットして、他の割り込みが現在の割り込みハンドラに干渉しないようにします。後続のIRET命令は、スタックに格納された内容でEFLAGSレジスタのIFフラグを復元します。トラップ・ゲートからハンドラ・プロシージャにアクセスしても、IFフラグには影響しません。

4.6.16 Interrupt handler Tasks

- タスクの切り替えは、IDTのタスクゲートを介して例外や割り込みハンドラにアクセスするときに発生します。例外や割り込みの処理を別々のタスクで行うことには、以下のようなメリットがあります。

- The complete context of the interrupted program or task is automatically saved;
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

例外処理や割り込み処理を別のタスクで行う場合、タスク切り替え時にマシンの状態を保存しなければならない分、割り込みゲートを使用するよりも遅くなり、結果的に割り込みのレイテンシーが増大するというデメリットがあります。

図4-30に示すように、IDTのタスクゲートはGDTのTSS記述子を参照します。ハンドラタスクへの切り替え処理は、通常のタスク切り替え処理と同じです。割り込まれたタスクへのバックリンクは、ハンドラタスクTSSの前タスクリンクフィールドに保存されます。例外でエラーコードが発生した場合は、エラーコードが新しいタスクスタックにコピーされます。

オペレーティングシステムで例外ハンドラや割り込みハンドラタスクを使用する場合、タスクをディスパッチするためのメカニズムは、実際には、オペレーティングシステムのソフトウェアスケジューラ

とプロセッサの割込み機構のハードウェアスケジューラの2つがあります。ソフトウェアスケジューラは、割り込みが有効なときにディスパッチされる可能性のある割り込みタスクに対応する必要があります。

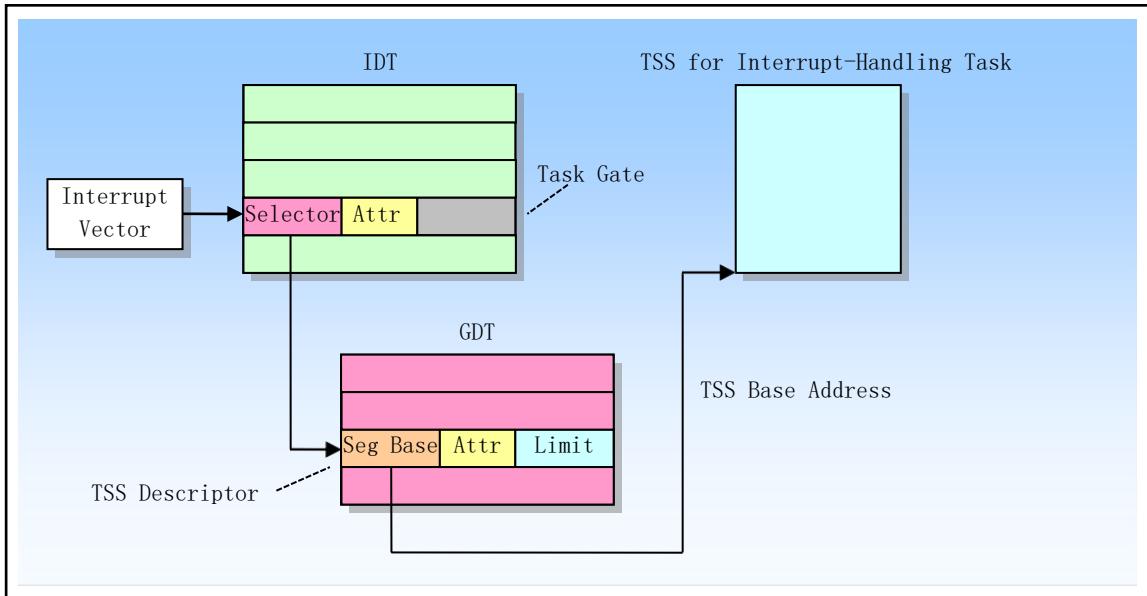


図4-30 インタラプトタスクスイッチ

4.6.17 Error Code

- 特定のセグメントに例外状態が発生した場合、プロセッサはエラーコードを例外ハンドラのスタックにプッシュします。エラー・コードのフォーマットを図4-31に示します。エラーコードはセグメントセレクタとよく似ていますが、最下位3ビットはTIとRPLフィールドではなく、次の3つのフラグです。
- Bit 0 is the external event (EXT) flag. When set, indicates that an event external to the program caused an exception, such as a hardware interrupt.
- Bit 1 is a descriptor location (IDT) flag. When this bit is set, the index portion indicating the error code points to a gate descriptor in the IDT. When this bit is reset, it indicates that the index refers to a descriptor in the GDT or LDT.
- Bit 2 is the GDT/LDT table select flag TI. Only useful when IDT (bit 1) is 0. When the TI=1, the index portion indicating the error code points to a descriptor in the LDT. When TI=0, it indicates that the index part in the error code points to a descriptor in the GDT table.

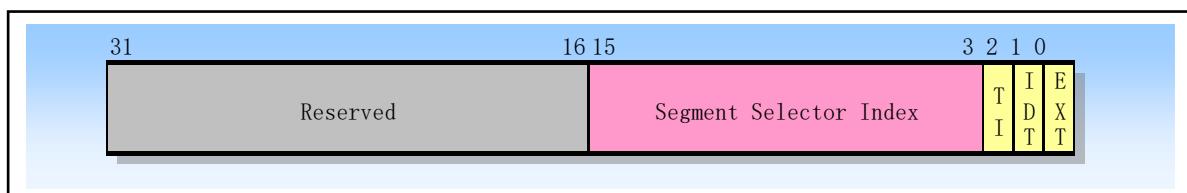


図4-31 エラーコードのフォーマット

セグメント・セレクタ・インデックス・フィールドは、エラー・コードによって参照されるセグメントまたはゲート・セレクタのIDT、GDT、または現在のLDTへのインデックスを提供します。場合によっては、エラー・コードがヌル（下位16ビットがすべてクリア）になることがあります。ヌルのエラー・コードは、特定のセグメントを参照したことが原因でエラーが発生したのではないこと、または操作中にヌルのセグメント記述子を参照したことを示します。

ページフォルト例外のエラーコード形式は、図4-32に示すように、上記とは異なります。最下位3ビットのみが有効で、その名前はページテーブルエントリの最後の3ビットと同じです（U/S, W/R,

P). The meanings and effects are:

- Bit 0 (P), the exception is caused by a page not being present or violating access privileges. P=0, indicating that the page does not exist; P=1 means that the page-level protection authority is violated.
- Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
- Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, indicating that the CPU is executing the super user code; U/S=1, indicating that the CPU is executing the general user code.

さらに、プロセッサは、ページフォルト例外を発生させるために使用したリニアアドレスをCR2にも格納します。ページフォルト例外ハンドラは、このアドレスを使用して、関連するページディレクトリとページテーブルエントリを見つけることができます。



図4-32 ページフォルトのエラーコードのフォーマット

なお、エラーコードはIRET命令によって自動的にスタックからポップアウトされないので、割り込みハンドラはエラーコードをスタックにクリアしてからリターンする必要があります。また、プロセッサで発生した一部の例外はエラーコードを生成し、ハンドラプロシージャのスタックに自動的に保存されますが、外部ハードウェア割り込みやINT n命令を実行するプログラムで発生した例外は、エラーコードをスタックにpushしません。

4.7 Task Management

タスクとは、プロセッサがスケジューリング、実行、サスペンドのために割り当てることのできる仕事の単位です。プログラム、タスクまたはプロセス、オペレーティングシステムサービス、割込みまたは例外処理手順、およびカーネルコードの実行に使用できます。タスクとは、実行中のプログラムや実行待ちのプログラムのことです。

80X86では、タスクの状態を保存したり、タスクをディスパッチしたり、あるタスクから別のタスクに切り替えたりするマルチタスクのハードウェアサポートを提供しています。プロテクトモードで作業しているときは、プロセッサの操作はすべてタスクの中で行われます。単純なシステムでも、少なくとも1つのタスクを定義する必要があります。より複雑なシステムでは、プロセッサのタスク管理機能を利用して、マルチタスクアプリケーションをサポートすることができます。

ディスクリプターテーブルの指定されたエントリを用いて、割込み、例外、ジャンプ、コールなどでタスクを実行することができます。ディスクリプターテーブルのタスク関連記述子には、タスク状態セグメント記述子とタスクゲートの2種類がある。これらの記述子のいずれかに実行権が渡されると、タスクの切り替えが行

われます。タスクスイッチングはプロシージャコールに似ていますが、タスクスイッチングの方がより多くのプロセッサの状態情報を保存します。タスクスイッチングは、新しい実行環境、つまり新しいタスクの実行環境に制御を完全に移します。この転送操作では、フラグレジスタEFLAGSやすべてのセグメントレジスタなど、プロセッサ内のほぼすべてのレジスタの現在の内容を保存する必要があります。ただし、タスクはリエンントラントにすることはできません。タスクの切り替えでは、スタックに情報をプッシュすることはありません。プロセッサの状態情報は、メモリ上のタスクステートセグメント（TSS）と呼ばれるデータ構造に格納されています。

4.7.1 Task Structure and State

タスクは、タスク実行空間とタスクステートセグメント（TSS）の2つの部分で構成されています。図4-33に示すように、タスク実行空間には、コードセグメント、データセグメント、および1つ以上のストекセグメントが含まれます。オペレーティングシステムがプロセッサの特権レベル保護メカニズムを使用している場合、タスク実行空間は、特権レベルごとに個別のストек空間を提供する必要があります。TSSは、タスク実行空間を構成するセグメントを指定し、タスクの状態情報を格納します。マルチタスク環境では、TSSはタスク間のリンクを処理する手段にもなります。

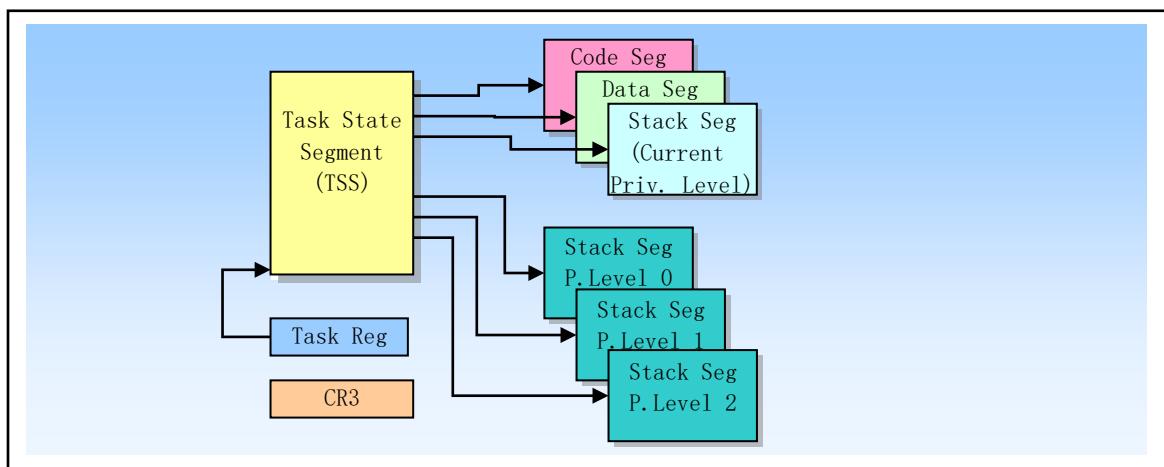


図4-33 タスクの構造

- タスクは、TSSを指すセグメントセレクタで指定されます。タスクが実行のためにプロセッサにロードされると、タスクのセグメントセレクタ、ベースアドレス、セグメント長、TSSセグメント記述子の属性がタスクレジスタ（TR）にロードされます。ページング機構が使用されている場合は、タスクが使用するページディレクトリのベースアドレスがコントロールレジスタCR3にロードされます。現在実行中のタスクの状態は以下のように構成されています。

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS);
- The state of the general purpose registers;
- The state of the EFLAGS, EIP, control register CR3, task register and LDTR register;
- The I/O map base address and I/O map (contained in the TSS);
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS);
- Link to previously executed task (contained in the TSS).

- 4.7.2 タスクを分配する前に、タスクのTSSには、タスクレジスタのステータスを除いて、これらの項目がすべて含まれています。また、LDTRレジスタの完全な内容はTSSには含まれず、LDTのセグメントセレクタのみが含まれます。

4.7.3 Execution of Tasks

- ソフトウェアやプロセッサーは、以下のいずれかの方法でタスクを実行に移すことができます。
- A explicit call to a task with the CALL instruction;
- A explicit jump to a task with the JMP instruction (the way the Linux kernel uses);

- An implicit call (by the processor) to an interrupt-handler task;

- An implicit call to an exception-handler task;
- A return (initiated with an IRET instruction) when the NTflag in the EFLAGS register is set.

これらのタスク実行のスケジューリング方法は、いずれもタスクゲートやタスクのTSSセグメントを指し示すセレクタを使ってタスクを決定します。CALL命令やJMP命令でタスクをディスパッチする場合、命令内のセレクタは、タスクのTSSを直接選択する場合と、TSSのセレクタを保持するタスクゲートを選択する場合があります。割込みや例外を処理するためにタスクをディスパッチする場合、IDTの割込みや例外のエントリには、割込みや例外を処理するタスクのTSSのセレクタを保持するタスクゲートが含まれていなければなりません。

タスクがディスパッチされて実行されると、現在実行中のタスクとスケジュールされたタスクの間でタスクスイッチ動作が自動的に行われます。タスク切り替えの際には、現在タスクを実行している実行環境（タスクの状態またはコンテキストと呼ばれる）がTSSに保存され、タスクの実行が中断されます。その後、新たにスケジュールされたタスクのコンテキストがプロセッサにロードされ、ロードされたEIPが指す命令から新しいタスクが実行されます。

現在実行中のタスク（caller）が、スケジュールされた新しいタスク（callee）を呼び出すと、callerのTSSセグメントセレクタがcalleeのTSSに格納され、callerへのリンクが提供されます。すべての80X86プロセッサでは、タスクは再帰的に呼び出されません。つまり、タスクは自分自身を呼び出したり、ジャンプしたりすることはできません。

割り込みや例外は、ハンドラタスクに切り替えて処理することができます。この場合、プロセッサは、割り込みや例外を処理するためのタスクスイッチを実行できるだけでなく、割り込みや例外のハンドラタスクが戻ってきたときに、自動的に割り込みタスクに戻ることができます。この機構により、割込みタスク中に発生した割込みを処理することができます。

タスク切り替え時には、別のLDTへの切り替えも行われるため、LDTベースのセグメントに対して、各タスクが異なる論理-物理アドレスのマッピングを行うことができます。同時に、ページディレクトリレジスタCR3も切り替え時に再ロードされるため、各タスクは独自のページテーブルを持つことができます。これらの保護機能を利用して、個々のタスクを分離し、相互に干渉しないようにすることができます。

4.7.4 マルチタスクのアプリケーションを処理するために、プロセッサのタスク管理機能を使用することは任意です。また、ソフトウェアを使用してマルチタスクを実装し、ソフトウェアで定義された各タスクが単一の80X86アーキテクチャのタスクのコンテキストで実行されるようにすることもできます。

4.7.5 Task Management Data Structures

- プロセッサには、マルチタスクをサポートする以下のレジスタとデータ構造が定義されています。
 - Task-state segment (TSS);
 - TSS descriptor;

- Task register (TR);
- Task-gate descriptor;
- NT flag in the EFLAGS register.

4.7.3.1

これらのデータ構造を利用することで、プロセッサは元のタスクのコンテキストを保持したまま、あるタスクから別のタスクに切り替えて、そのタスクを再実行することができます。プロテクトモードで動作する場合、少なくとも1つのタスクに対してTSSとTSS記述子を作成し、TSSのセグメントセレクタをタスクレジスタにロードする必要があります (LTR命令を使用)。

4.7.3.2 Task-State Segment (TSS)

タスクの実行を復元するためのプロセッサの状態情報は、タスクステートセグメントTSS (Task State Segment) と呼ばれるセグメントに保存されます。図4-34は、32ビットCPUで使用されるTSSのフォーマットです。TSSセグメントのフィールドは、ダイナミックフィールドとスタティックフィールドの2つに大別されます。

| | | |
|----------------------|---|---|
| I/O Map Base Address | | T |
| | LDT Segment Selector | |
| | GS | |
| | FS | |
| | DS | |
| | SS | |
| | CS | |
| | ES | |
| | EDI | |
| | ESI | |
| | EBP | |
| | ESP | |
| | EBX | |
| | EDX | |
| | ECX | |
| | EAX | |
| | EFLAGS | |
| | EIP | |
| | Page Directory Base Register (PDBR) CR3 | |
| | SS2 | |
| | ESP2 | |
| | SS1 | |
| | ESP1 | |
| | SS0 | |
| | ESP0 | |
| | Previous Task Link (TSS Selector) | |

図 4-34 32 ビットタスクステータスセグメント TSS フォーマット

1. Dynamic fields. When the task is switched and suspended, the processor updates the contents of the dynamic field. These fields include:
 - General purpose register field. Used to save the contents of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers.
 - Segment selector field. Used to save the contents of the ES, CS, SS, DS, FS, and GS segment registers.
 - Flag register EFLAGS field. Save EFLAGS before switching.
 - Instruction pointer EIP field. Save the contents of the EIP register before switching.
 - The previous task link field. Contains the previous task TSS segment selector (updated during a task switch of a call, interrupt, or exception fire). This field (also commonly referred to as the Back link field) allows the task to switch to the previous task using the IRET instruction.
2. Static fields. The processor reads the contents of static fields, but usually does not change them. These field contents are set when the task is created. These fields are:
 - LDT segment selector field. Contains the segment selector for the task's LDT.
 - CR3 Control Register Field. Contains the physical base address of the page directory used by the task.

- コントロールレジスタCR3は、一般的にページディレクトリベースレジスタ（PDBR）とも呼ばれています。
- Stack pointer field for privilege levels 0, 1, and 2. These stack pointers consist of stack segment selectors (SS0, SS1, and SS2) and offset pointers in the stack (ESP0, ESP1, and ESP2). Note that the values of these fields are constant for a given task. Therefore, if a stack switch occurs in a task, the contents of the registers SS and ESP will change.
- Debug Trap T flag field. This field is located at byte 0x64 bit 0. When this bit is set, then a debug exception will be generated when the processor switches to the task.
- I/O bitmap base address field. This field contains the 16-bit offset value from the beginning of the TSS segment to the I/O permission bitmap. When present, these maps are stored at the higher address of the TSS. The I/O mapping base address points to the beginning of the I/O permission bitmap and the end of the interrupt redirection bitmap.

4.7.3.3 ページング機構が使用されている場合、タスク切り替え時のプロセッサ動作のTSS部分（最初の104バイト目）では、メモリページ境界を回避する必要があります。TSS部分にメモリページ境界がある場合は、境界の両側のページが物理メモリ上に同時にかつ連続して存在していなければなりません。また、ページング機構が使用されている場合、元のタスクTSSと新しいタスクTSSに関連するページ、およびそれに対応するディスクリプターテーブルのエントリは、読み取りと書き込みが可能である必要があります。

4.7.3.4 TSS Descriptor

他のセグメントと同様に、タスク・ステータス・セグメントTSSもセグメント記述子を使って定義します。図4-35にTSS記述子のフォーマットを示します。TSS記述子はGDTにのみ格納されます。

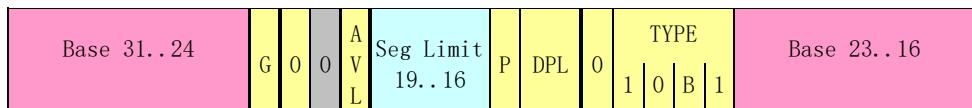


図4-35 TSSセグメント記述子のフォーマット

タイプフィールドTYPEのビジーフラグ（B）は、タスクがビジーであるかどうかを示すために使用されます。ビジーなタスクとは、現在実行中のタスクや、実行待ち（サスペンド）のタスクのことです。タイプフィールドの値が0b1001の場合は、タスクが非アクティブであることを示し、値が0b1011の場合は、タスクがビジーであることを示します。プロセッサはビジーフラグを使用して、実行が中断されたタスクを呼び出そうとしていることを検出します。1つのタスクに関連するビジー・フラグが1つだけであることを保証するために、各TSSはそれを指すTSS記述子を1つだけ持つべきです。

ベースアドレス、リミット、ディスクリプター特権レベルDPL、グラニュラリティG、現在のフラグは、データセグメントディスクリプターの対応するフィールドと同じ機能を持つ。G=0の場合、limitフィ

ールドは103(0x67)以上の値でなければならず、TSSセグメントの最小長は104バイト以上となります。TSSセグメントにI/O許可ビットマップが含まれている場合、TSSセグメントの長さはより大きくする必要があります。また、OSが他の情報をTSSセグメントに格納したい場合は、TSSセグメントの長さを大きくする必要があります。

コール命令やジャンプ命令を使えば、TSS記述子にアクセスできるプログラムであれば、タスクスイッチを起こすことができます。

TSS記述子にアクセスできるプログラムは、TSS記述子のDPL以下の数値のCPLを持っている必要があります。ほとんどのシステムでは、TSS記述子のDPLフィールドを3以下に設定する必要があります。このようにして、特権的なソフトウェアのみがタスク切り替え操作を行うことができます。ただし、マルチタスク・アプリケーションでは、一部のTSSのDPLを3に設定することで、ユーザの特権レベルでもタスク切り替え操作を行なうことができます。

- 4.7.3.5** TSSセグメント記述子にアクセスできるようになったからといって、プログラムに記述子の読み書きができるようになるわけではありません。TSSセグメント記述子を読み書きしたい場合は、メモリ上の同じ位置にマッピングされているデータセグメント記述子（つまりエイリアス記述子）を使用します。TSS記述子を任意のセグメントレジスタにロードすると、例外が発生します。また、TIフラグで設定されたセレクタ（現在のLDTのセレクタ）を使ってTSSセグメントにアクセスしようとすると、例外が発生します。

4.7.3.6 Task Register

タスクレジスタTR(Task Register)には、16ビットのセグメントセレクタと、現在のタスクのTSSセグメントのディスクリプタ(不可視部分)全体が格納されています。この情報は、GDT内の現在のタスクのTSS記述子からコピーされます。プロセッサは、タスクレジスタTRの不可視部分を使用して、TSSセグメント記述子の内容をバッファリングします。

- 4.7.3.7** LTR命令とSTR命令は、それぞれタスクレジスタの可視部分、すなわちTSSセグメントのセレクタをロードおよびセーブするために使用されます。LTR命令は、特権レベル0のプログラムでのみ実行できます。LTR命令は、通常、システムの初期化時にTRレジスタの初期値（タスク0のTSSセグメントセレクタなど）をロードし、システム運用時には、タスク切り替え時にTRの内容を自動的に変更するために使用されます。

4.7.3.8 Task-Gate Descriptor

タスクゲート記述子は、図4-27に示すように、タスクへの間接的な保護された参照を提供します。タスクゲート記述子は、GDT、LDT、またはIDTテーブルに格納することができます。

タスクゲート記述子のTSSセグメントセレクタフィールドは、GDTのTSSセグメント記述子を指しています。このTSSセグメントセレクタのRPLフィールドは使用されません。タスクゲート記述子のDPLは、タスク切り替え時のTSSセグメントへのアクセス制御に使用されます。プログラムがタスクゲートを介してタスクへの呼び出しやジャンプを行う場合、タスクゲートを指すゲートセレクタのCPLとRPLフィールドは、タスクゲートディスクリプタのDPL以下でなければなりません。なお、タスクゲートを使用する場合、ターゲットTSSセグメント記述子のDPLは無視されます。

プログラムは、タスクゲート記述子またはTSSセグメント記述子を介してタスクにアクセスできます。図4-36は、LDTテーブル、GDTテーブル、IDTテーブルのタスクゲートがすべて同じタスクを指している様子を示しています。

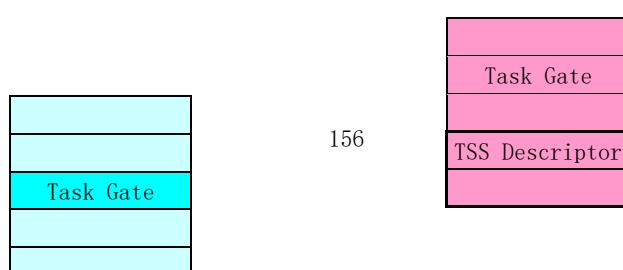


図4-36 同じタスクを参照するタスクゲート

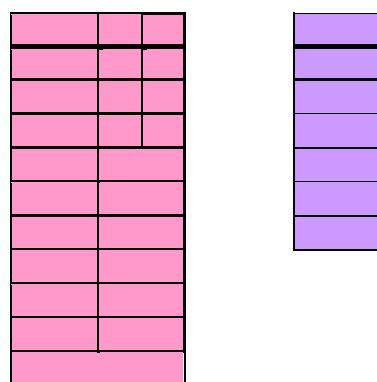
4.7.6 Task Switching

1. プロセッサは、4つのケースのいずれかで、他のタスクに実行を移します。
 2. The current program or task executes a JMP or CALL instruction to a TSS descriptor in the GDT.
 3. The current program or task executes a JMP or CALL instruction to a task gate descriptor in the GDT or the current LDT;
 4. The interrupt or exception vector points to a task gate descriptor in the IDT table;
 5. The current task executes an IRET instruction when the NT flag in the EFLAGS register is set.
- JMP命令、CALL命令、IRET命令、そして割り込みや例外などは、いずれもプログラムを切り替えるための一般的なメカニズムです。タスクスイッチが行われるかどうかは、TSS記述子やタスクゲートの参照(タスクへの呼び出しやジャンプ時)、NTフラグの状態(INET命令実行時)などによって決まります。

タスク切り替えは、JMP命令やCALL命令でTSS記述子やタスクゲートに制御を移すことができます。同じ2つの方法を使うと、図4-37のように、プロセッサは指定されたタスクに制御を移すことになります。

割り込みや例外のベクタ・インデックスがIDTのタスク・ゲートである場合、割り込みや例外はタスク・スイッチを引き起こします。しかし、ベクターインデックスがIDT内の割込みまたはトラップゲートである場合、タスクスイッチは発生しません。

割り込みサービスハンドラプロシージャは、常に中断されたプログラムやプロシージャに実行権を返すので、中断されたプログラムが他のタスクにある場合もあります。NTフラグがリセット状態の場合は、一般的な復帰動作を行います。NTフラグがセットされている場合は、復帰操作によりタスクの切り替えが行われます。切り替え先の新しいタスクは、割込みサービス手順TSSのTSSセレクタ（前タスククリンクフィールド）で指定されます。



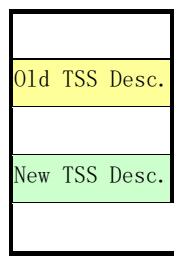


図4-37 タスク切り替え動作図

1. 新しいタスクに切り替えるとき、プロセッサは次のような処理を行います。
2. Obtain a TSS segment selector for a new task as the operand of the JMP or CALL instruction, or from the task gate, or from the previous task link field of the current TSS (for task switching caused by IRET).
3. Check if the current task is allowed to switch to the new task. Apply data access privilege rules to JMP and CALL instructions. The CPL of the current task, and the RPL of the new task segment selector must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Regardless of the DPL of the target task gate or TSS descriptor, exceptions, interrupts (except for interrupts generated using the INT n instruction), and IRET instructions allow task switching. The interrupt generated by the INT n instruction will check the DPL.
4. Check that the TSS descriptor for the new task is marked as present (P=1) and that the limit is valid (greater than 0x67). Any changes to the state of the processor are resumed when attempting to execute an instruction that will generate an error. This causes the return address of the exception handler to point to the error instruction instead of the next instruction of the error instruction. Therefore the exception handler procedure can handle the error condition and re-execute the task. The intervention of the exception handler procedure is completely transparent to the application.
5. If the task switch is generated from a JMP or IRET instruction, the processor will reset the busy flag B in the current task (old task) TSS descriptor; if the task switch is generated by a CALL instruction, exception or interrupt, the busy flag B Will not be changed.
6. 5. If the task switch is initiated with an IRET instruction, the processor resets the NT flag in a temporarily saved EFLAGS image; if the task switch is initiated with a CALL, JMP instruction, or an exception or interrupt, the NT flag is left unchanged in the saved EFLAGS image.
7. Save the state of the current (old) task to the TSS of the current task. The processor retrieves the base address of the current task TSS from the task register and copies the states of the following registers into the current TSS: all general purpose registers, the segment selector in the segment register, the flag register EFLAGS, and the instruction pointer EIP.
8. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor sets the NT flag in the EFLAGS image stored in the new task's TSS; if initiated with an IRET instruction, the processor restores the NT flag from the EFLAGS image stored on the stack. If initiated with a JMP instruction, the NT flag is left unchanged.
9. If the task switch was initiated by a CALL, JMP instruction, or exception or interrupt, the processor sets the busy flag B in the new task TSS descriptor. If the task switch is generated by the IRET, the B flag is not changed.
10. Load the task register TR (including the hidden portion) using the segment selector and descriptor of the new task TSS. Set the TS flag in the control register CR0 image stored in the new task's TSS.
11. 10. Load the TSS status of the new task into the processor. This includes the LDTR register, the PDBR (CR3) register, the EFLAGS register, the EIP register, and general purpose registers and segment selectors. Any errors detected during this time will appear in the context of the new task.
12. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

タスク切り替え動作が正常に行われると、現在実行中のタスクの状態が常に保存されます。タスクが実行を再開すると、保存されたEIPが指す命令から実行され、すべてのレジスタはタスクが中断されたときの値に復元されます。

タスクスイッチを行う際、新しいタスクの特権レベルは、元のタスクの特権レベルとは関係ありません。新しいタスクは、TSSからロードされたCSレジスタのCPLフィールドで指定された特権レベルで実行を開始します。各タスクは、独立したアドレス空間とTSSセグメントによって互いに分離されており、特権レベルのルールによってTSSへのアクセスがすでに制御されているため、ソフトウェアはタスク切り替え時に特権レベルのチェックを行う必要はありません。

- 4.7.7** コントロールレジスタCR0のタスク切り替えフラグTSは、タスクが切り替わるたびにセットされます。このフラグは、システムソフトウェアにとって非常に有用です。システム・ソフトウェアは、プロセッサと浮動小数点コプロセッサの間の操作を調整するために、TSフラグを使用することができます。TSフラグは、コプロセッサ内のコンテキストが現在のタスクのコンテキストとは異なる可能性があることを示します。

4.7.8 Task Linking

TSSの前タスクリンクフィールド(Backlink)とEFLAGSのNTフラグは、実行を前のタスクに戻すために使われます。NTフラグは、現在実行中のタスクが他のタスクの実行中に入れ子になっているかどうかを示し、現在のタスクの前タスクリンクフィールドには、入れ子の階層に上位のタスクがあれば、そのタスクのTSSセレクタが保持されています（図4-38参照）。

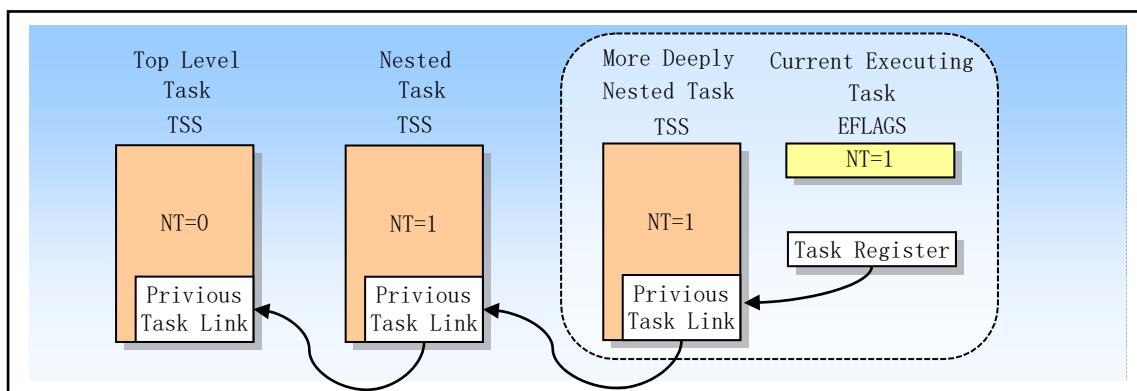


図4-38 タスクの入れ子

CALL命令、割込み、例外などでタスクスイッチが発生すると、プロセッサは現在のTSSセグメントのセレクタを新しいタスクTSSセグメントの前タスクリンクフィールドにコピーし、EFLAGSにNTフラグを設定します。NTフラグは、保存されたTSSセグメントのセレクタがTSSの前タスクリンクフィールドに保存されていることを示します。ソフトウェアがIRET命令を使って新しいタスクをサスペンドした場合、プロセッサは前タスクリンクフィールドの値とNTフラグを使って前のタスクに戻ります。つまり、NTフラグが設定されている場合、プロセッサは前タスクリンクフィールドで指定されたタスクに切り替えて実行します。

なお、JMP命令によるタスク切り替えの場合は、新しいタスクは入れ子になりません。つまり、NTフラグは0になり、前タスクのリンクフィールドは使用されません。JMP命令はネスティングが望まれないタスク切り替えに使用されます。

タスク切り替え時のビギーフラグB（TSSセグメントディスクリプター内）、NTフラグ、前タスクリンクフィールド、TSフラグ（CR0内）の使用方法を表4-10にまとめました。どのような特権レベルで実行されているプログラムでもNTフラグを変更することができるので、どのようなプログラムでもNTフラグを設定してIRET命令を実行することができることに注意してください。この方法では、プロセッサは現在のタスクTSSの前タスクリンクフィールドで指定されたタスクを実行することになります。このような偽造されたタスク切り替えの実行を成功させないために、オペレーティングシステムは、各TSSのこのフィールドを0に初期化する必要があります。

| 表 4-10 タスクス イッチに によるビジ ーフラ グ、 NT フラグ、 前タスク リンクフ ィール ド、 TSフ ラグへの 影響 Flag or Field | Effect of JMP | Effect of CALL or Interrupt | Effect of IRET |
|---|---|---|---|
| Busy (B) flag of new task. | Flag is set. Must have been clear before. | Flag is set. Must have been clear before. | No change. Must have been set. |
| Busy flag of old task. | Flag is cleared. | No change. Flag is currently set. | Flag is cleared. |
| NT flag of new task. | No change. | Flag is set. | Restored to value from TSS of new task. |
| NT flag of old task. | No change. | No change. | Flag is cleared. |
| Previous task link field of new task. | No change. | Loaded with selector for old task's TSS | No change. |
| Previous task link filed of old task. | No change. | No change. | No change. |
| TS flag in CR0 | Flag is set. | Flag is set. | Flag is set. |

4.7.9 Task Address Space

タスクのアドレス空間は、タスクがアクセスできるセグメントで構成されています。これらのセグメントには、TSS で参照されるコード、データ、スタックとシステムセグメント、およびタスクコードがアクセスするその他のセグメントが含まれます。これらのセグメントは、プロセッサのリニアアドレス空間にマッピングされた後、プロセッサの物理アドレス空間にマッピングされます（直接またはページングによって）。

TSSのLDTフィールドを使うと、各タスクに独自のLDTを与えることができます。あるタスクに対して、そのタスクに関連するすべてのセグメント記述子をLDTに入れることで、そのタスクのアドレス空間を他のタスクから分離することができます。もちろん、複数のタスクが同じLDTを使用することも可能です。これは、システム全体の保護バリアを捨てることなく、特定のタスクが相互に通信したり制御したりすることを可能にする、シンプルで効果的な方法です。すべてのタスクがGDTにアクセスできるため、このテーブルを介してアクセスされる共有セグメントを作成することも可能です。

4.7.6.1 ページングメカニズムが有効な場合、TSSのCR3レジスタ (PDBR) フィールドにより、各タスクはリ

ニアアドレスを物理アドレスにマッピングするための独自のページテーブルセットを持つこともできます。また、複数のタスクが同じページテーブルを共有することもできます。

4.7.6.2 Mapping Tasks to the Linear and Physical Address Spaces

■ タスクをリニアアドレス空間と物理アドレス空間にマッピングするには2つの方法があります。

- All tasks share a linear to physical address space mapping. This method can only be used when the paging mechanism is not enabled. When paging is not turned on, all linear addresses are mapped to the same physical address. When the paging mechanism is turned on, we can use this mapping from linear to physical address space by having all pages use a single page directory. If the demand page virtual storage technology is supported, the linear address space can exceed the size of the existing physical address space.
- Each task has its own linear address space and is mapped to the physical address space. We can use this mapping form by having each task use a different page directory. Because PDBR (Control Register CR3) is loaded every time a task is switched, each task can have a different page directory.

異なるタスクのリニアアドレス空間は、全く異なる物理アドレスにマッピングすることができます。異なるページディレクトリのエントリ（テーブルエントリ）が異なるページテーブルを指し、そのページテーブルが物理アドレスの異なるページを指している場合、各タスクは物理アドレスを共有することはありません。

タスクのリニアアドレス空間をマッピングするいずれの方法においても、すべてのタスクのTSSは共有の物理アドレス空間領域に格納されている必要があります、すべてのタスクはこの領域にアクセスすることができます。プロセッサがタスクスイッチングを行うためには

4.7.6.3

で、TSSの読み出しや更新時にTSSのアドレスマッピングが変わらない場合は、このマッピング方法が必要です。また、GDTによってマッピングされたリニアアドレス空間は、共有された物理アドレス空間にもマッピングされる必要があります。そうでないと、GDTの役割が失われてしまいます。

4.7.6.4 Task Logical Address Space

- タスク間でデータを共有するには、以下のいずれかの方法で、データセグメントの共有論理-物理アドレス空間マッピングを確立します。

- By using the segment descriptor in the GDT. All tasks must be able to access the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear address space and these segments are mapped into the physical address space shared by all tasks, then all tasks can share the code and data in those segments.
- Through a shared LDT. Two or more tasks can use the same LDT if their LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments mapped to a common area of the physical address space, then all tasks sharing the LDT can share all of the code and data in those segments. This kind of sharing is better than sharing through GDT, because doing so can limit sharing to certain tasks. There are other tasks in the system that do not have access to these shared segments.
- Segment descriptors in different LDTs mapped to the common address area in the linear address space. If this common area in the linear address space maps each task to the same area of the physical address space, then these segment descriptors allow tasks to share the segments. Such segment descriptors are often referred to as alias segments. This sharing method is better than the one given above, because other segment descriptors in the LDT can point to separate unshared linear address regions.

4.8 The Initialization of Protected Mode

- 4.8.1** 電源投入時やハードウェアリセット時には、プロセッサは8086互換の実アドレスモードで動作し、物理アドレス0xFFFFFFF0（通常はEPROM内）からソフトウェア初期化コードを実行します。ソフトウェア初期化コードでは、まず基本的なシステム機能の動作に必要なデータ構造情報を設定する必要があります。例えば、割り込みや例外を処理するリアルモードIDTテーブル（つまり、割り込みベクトルテーブル）などです。プロセッサがリアルモードでも動作する場合は、ソフトウェアは、アプリケーションがリアルモードで確実に実行できるように、オペレーティング・システム・モジュールと対応するデータをロードする必要があります。プロセッサがプロテクトモードで動作するのであれば、オペレーティング・システム・ソフトウェアは、モードの保護に必要なデータ構造情報をロードしてから、プロテクトモードに切り替える必要があります。

4.8.2 First Instruction Executed

前述のとおり、ハードウェアリセット後に最初に取得・実行される命令は、物理アドレス0xFFFFFFF0にあります。このアドレスは、プロセッサの最上位物理アドレスの16バイト目にあたります。このアドレスは、通常、ソフトウェアの初期化コードを含むEPROMファームウェアが配置されているアドレス範囲です。

実アドレスモードでは、0xFFFFFFF0というアドレスは、プロセッサの1MBアドレス可能範囲外です。プロセッサは以下の方法で開始アドレスに初期化します。CSレジスタには、目に見えるセグメントセレクタ部分と、目に見えないベース部分の2つの部分があります。実アドレスモードでは、ベースアドレスは通常、16ビットのセグメントセレクタの値を4ビット左にシフトして20ビットのベースアドレス

を生成します。しかし、ハードウェアリセット時には、CSレジスタのセグメントセレクタは0xF000としてロードされ、ベースアドレスは0xFFFF0000としてロードされます。このため、開始アドレスはベースアドレスとEIPレジスタを足したものになります（つまり、 $0xFFFF0000 + 0xFFF0 = 0xFFFFFFFF0$ ）。

ハードウェア・リセット後にCSレジスタが最初に新しい値をロードするとき、プロセッサは通常の

4.8.3 実アドレスモードでのアドレス変換のルール（例：[CSベースアドレス=CSセグメントセレクタ*16]）を説明します。EPROMベースのソフトウェア初期化コードが完了するまで、CSレジスタのベースアドレスが変更されないようにするために、コードにファーホップやリモートコールを含めたり、割り込みを発生させたりしてはいけません（これによりCSセレクタの値が変化します）。

4.8.4 Initialization operation when entering protection mode

■ ハードウェアリセット後、プロセッサはリアルアドレスモードになります。一部の基本的なデータ構造とコードモジュールは、プロセッサのさらなる初期化をサポートするために、初期化中に物理メモリにロードする必要があります。プロテクトモードに必要なデータ構造の一部は、プロセッサのメモリ管理機能によって決定されます。プロセッサは、単一の統一されたアドレス空間のフラットモデルから、タスクごとに複数の保護されたアドレス空間を持つ高度に構造化されたマルチセグメントモデルまで使用可能なセグメントモデルをサポートしています。ページング機構は、一部がメモリ上にあり、一部がディスク上にある大規模なデータ構造情報を処理するために使用することができます。どちらの形式のアドレス変換でも、OSはメモリ管理ハードウェアに必要なデータ構造をメモリに設定する必要があります。したがって、プロセッサをプロテクトモードに切り替える前に、オペレーティング・システムのローディングおよび初期化ソフトウェア（bootsect.s、setup.s、head.s）は、まずプロテクトモードで使用するデータ構造の基本をメモリに設定する必要があります。これらのデータ構造には次のようなものがあります。

- A protected-mode interrupt descriptor table IDT;
- A global descriptor table GDT;
- A task status segment TSS;
- A local descriptor table LDT;
- If paging is enabled, at least one page directory and one page table need to be set;
- A code segment containing execution code for the processor to switch to protected mode;
- Code modules that contain interrupts and exception handlers.

■ また、ソフトウェアの初期化コードで以下のシステムレジスタを設定しないと、プロテクトモードに切り替えることができません。

- Global descriptor table base address register GDTR;
- Interrupt descriptor table base address register IDTR;
- Control register CR1--CR3;

4.8.21 これらのデータ構造、コードモジュール、システムレジスタを初期化した後、CR0レジスタの保護モードフラグPE（ビット0）を設定することで、プロセッサを保護モードに切り替えることができます。

4.8.22 Protection Mode System Structure Table

ソフトウェアの初期化時にメモリに設定されるプロテクトモードのシステムテーブルは、主にオペレーティングシステムがサポートするメモリ管理のタイプに依存します（フラット、ページング付きフラット、セグメンテーション、ページング付きセグメンテーション）。

ページングのないフラットなメモリモデルを実装するためには、ソフトウェアの初期化コードで少なくとも1つのコードセグメントと1つのデータセグメントを持つGDTテーブルを設定する必要があります。もちろん、GDTテーブルの最初の項目には、ヌルデスクリプタも配置する必要があります。スタックは、通常の読み書き可能なデータセグメントに置くことができるので、特別なスタック記述子は必要ありません。また、ページング機構をサポートするフラットメモリモデルでは、ページディ

レクトリと少なくとも1つのページテーブルが必要です。GDTテーブルを使用する前に、LGDT命令を使用してGDTのベースアドレスとリミットをGDTRレジスタにロードする必要があります。

また、マルチセグメントモデルでは、OS用の追加セグメントのほか、アプリケーションごとのセグメントやLDTテーブルのセグメントが必要となります。LDTテーブルのセグメント記述子は、GDTテーブルに格納する必要があります。オペレーティングシステムの中には、アプリケーション用に新しいセグメントと新しいLDTセグメントを割り当てるものもあります。この方法は、Linuxのような動的なプログラミング環境に最大限の柔軟性をもたらします。

4.8.23

オペレーティング・システムを使用しています。プロセスコントローラーのような組み込みシステムでは、固定数のアプリケーション用に固定数のセグメントとLDTをあらかじめ割り当てておくことができ、リアルタイムシステムのソフトウェア環境構造を簡単かつ効率的に実現することができます。

4.8.24 Exceptions and Interrupt Initialization in Protected Mode

ソフトウェアの初期化コードでは、保護モードIDTを設定する必要があります。IDTには、少なくとも、プロセッサが生成する可能性のある各例外ベクターに対応するゲート記述子を含める必要があります。割り込みゲートやトラップゲートを使用する場合、ゲート記述子はすべて、割り込み処理や例外処理を含む同じコードセグメントを指すことができます。タスク・ゲートを使用する場合、タスク・ゲートを使用する各例外処理プロセスには、TSS と関連するコード、データ、およびスタック・セグメントが必要です。ハードウェアが割込みを生成することを許可している場合、ゲート記述子は1つ以上の割込みハンドラのIDTに設定されなければなりません。

4.8.25

IDTテーブルのベースアドレスとリミットレンジスは、IDTを使用する前にLIDT命令でIDTRレジスタにロードする必要があります。

4.8.26 Paging initialization

■ ページング機構の設定は、コントロールレジスタCR0のPGフラグで行います。このフラグが0にクリアされると（ハードウェアリセット時の状態）、ページング機構はオフになり、PGフラグがセットされると、ページング機構はオンになります。PGフラグを設定する前に、以下のデータ構造およびレジスタを初期化する必要があります。

- Software must create at least one page directory and one page table in physical memory. If the page directory table contains a entry that points to itself, then you can eliminate the use of page table. At this point, the page directory table and the page table are stored on the same page.
- Load the physical base address of the page directory table into the CR3 register (also known as the PDBR register).
- The processor is in protected mode. If all other restrictions are met, the PG and PE flags can be set at the same time.

■ 互換性を保つために、PGフラグ（およびPEフラグ）を設定する際には、以下のルールを遵守する必要があります。

- Instructions that set the PG flag should follow a JMP instruction immediately. The JMP instruction following the MOV CR0 instruction changes the execution stream, so it clears the instructions that 80X86 processor has taken or decoded. However, the Pentium and above processors use the Branch Target Buffer (BTB) for branch code orientation, thus eliminating the need to refresh the queue for branch instructions.
- The code that sets the PG flag to the jump instruction JMP must come from a page on the peer mapping (that is, the linear address before the jump is the same as the physical address after the paging is turned on).

4.8.27 Multitasking Initialization

マルチタスク機構を使用する場合や、特権レベルの変更を許可する場合は、ソフトウェアの初期化コードには、少なくとも1つのTSSと、それに対応するTSSセグメント記述子が必要です（特権レベル0、1、2のスタックセグメントポインタをTSSから取得する必要があるため）。TSS記述子にビジー

のマークをつけない（ビジーフラグを立てない）。ビジーフラグは、タスクスイッチを行う際にプロセッサが設定するだけです。LDTセグメント記述子と同様に、TSS記述子もGDTに格納されています。プロセッサがプロテクトモードに切り替わった後、LTR命令を使ってTSSセグメント記述子のセレクタをタスクレジスタTRにロードすることができます。この命令は、TSSをビジー（B = 1）としてマークしますが、タスクスイッチ操作は行いません。プロセッサはこのTSSを使って、特権レベル0、1、2のスタックを探すことができます。プロテクトモードでは、ソフトウェアが最初のタスクスイッチを実行する前に、TSSセグメントのセレクタを最初にロードする必要があります。

LTR命令の実行後、タスクレジスタに対する以降の操作は

4.8.5 タスクの切り替えを行います。他のセグメントやLDTと同様に、TSSやTSS記述子は事前に割り当てることも、必要に応じて割り当てることもできます。

4.8.6 Mode Switching

4.8.3.1

プロセッサをプロテクトモードで動作させるためには、実アドレスモードからのモード切り替えを行う必要があります。いったんプロテクトモードに入ると、通常は実アドレスモードに戻る必要はありません。実アドレスモード用にプログラムされたプログラムを動作させるためには、通常、実モードに切り替えるよりも仮想8086モードで動作させた方が便利である。

4.8.3.2 Switching to Protected Mode

1. プロテクトモードに切り替える前に、まず最低限のシステムデータ構造とコードモジュールをいくつかロードする必要があります。これらのシステムテーブルが作成されると、ソフトウェアの初期化コードをプロテクトモードに切り替えることができます。CR0レジスタのPEフラグをセットするMOV CR0命令を実行することで、プロテクトモードに入ることができます。(同じ命令で、CR0のPGフラグを使って、ページング機構を有効にすることができます)。プロテクトモードで動作しているとき、特権レベルCPLは0となります。 プログラムの互換性を確保するために、以下のように切り替え操作を行う必要があります。
 2. Disable interrupts. Maskable hardware interrupts can be disabled using the CLI instruction. The NMI is disabled by hardware circuitry. At the same time, the software should ensure that no exceptions or interruptions occur during mode switching operations.
 3. Execute the LGDT instruction to load the base address of the GDT table into the GDTR register.
 4. Execute the MOV CR0 instruction that sets the PE flag (optional setting of the PG flag) in the control register CR0.
 5. Execute a far jump JMP or a far call CALL instruction immediately after the MOV CR0 instruction. This operation is usually a far jump to or far from the next instruction in the instruction stream.
 6. If a local descriptor table is to be used, execute the LLDT instruction to load the LDT segment selector into the LDTR register.
 7. Execute the LTR instruction to load the task register TR with the segment selector of the initial protected mode task or the segment descriptor of the writable memory area. This writable memory area is used to store the TSS information of the task when the task is switched.
 8. After entering protected mode, the segment register still contains the contents in real address mode. The JMP or CALL instruction in step 4 resets the CS register. Do one of the following to update the contents of the remaining segment registers: (1) Reload registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not used, load them with a null selector. (2) Execute a JMP or CALL instruction on a new task that automatically resets the value of the segment register and branches to the new code segment.
 9. Execute the LIDT instruction to load the base address and limit of the protected mode IDT table into the IDTR register.
 10. Execute the STI instruction to turn on the maskable hardware interrupt and perform the necessary hardware operations to turn on the NMI interrupt.

さらに、MOV CR0命令の直後にJMPまたはCALL命令を実行すると、実行の流れが変わります。ページング機構が有効な場合、MOV CR0命令とJMPまたはCALL命令の間のコードは、ピアマッピング上のページからのものでなければなりません(つまり、ジャンプ前のリニアアドレスとページング後の

物理アドレスは同じです)。JMPまたはCALL命令がジャンプするターゲット命令は、ピアマッピングのページにある必要はありません。

4.8.3.3 Switch back to real address mode

1. 実アドレスモードに戻したい場合は、MOV CR0命令でコントロールレジスタCR0のPEフラグをクリアします。実アドレスモードに戻す処理は、以下の手順で行います。
 2. Disable interrupts. Maskable hardware interrupts can be disabled using the CLI instruction. The NMI is disabled by hardware circuitry. At the same time, the software should ensure that no exceptions or interruptions occur during mode switching operations.
 3. If the paging mechanism is enabled, you need to execute:
 - Transfer the program control to linear address of the peer map (ie the linear address is equal to the physical address).
 - Make sure the GDT and IDT are on the peer mapped page.
 - Clear the PG flag in CR0.
 - Set to 0x00 in the CR3 register to refresh the TLB buffer.
 4. Transfer the control of the program to a readable segment of length 64KB (0xFFFF). This step loads the CS register using the segment limit required by the real mode.
 5. Load the SS, DS, ES, FS, and GS segment registers with a selector that points to a descriptor with the following set values.
 - Limit = 64KBytes (0xFFFF).
 - Byte granularity (G = 0).
 - Expand up (E=0).
 - Writable (W=1).
 - Present (P=1).
 6. Execute the LIDT instruction to point to the real address mode interrupt table in the 1MB real mode address range.
 7. Clear the PE flag in CR0 to switch to real address mode.
 8. Execute a far jump instruction to jump to a real mode program. This step refreshes the instruction queue and loads the appropriate base address and access value for the CS register.
 9. The SS, DS, ES, FS, and GS registers are loaded as needed for the real address mode code. If any of the registers are not used in real address mode, write 0 to them.
 10. Execute the STI instruction to turn on the maskable hardware interrupt and perform the necessary hardware operations to turn on the NMI interrupt.

4.9 A Simple Multitask Kernel Example

- 4.9.1 本章では、本章と前章のまとめとして、シンプルなマルチタスク・カーネルの設計と実装について完全に説明します。このカーネル例には、特権レベル3のユーザータスクが2つと、特権レベル0のシステムコール割り込み手続きが1つ含まれています。まず、この単純なカーネルの基本構造とロード動作の基本原理を説明し、次にマシンのRAMメモリにロードする方法と、2つのタスクを切り替える方法を説明します。最後に、この単純なカーネルを実装するためのソースコードとして、ブート boot.s と保護モードマルチタスクカーネルプログラム head.s を示します。

4.9.2 Multitasking program structure and working principle

この章で紹介するカーネルの例は、2つのソースファイルから構成されています。一つはas86言語でコンパイルされたブートローダboot.sで、コンピュータシステムの電源を入れたときに起動ディスクから

カーネルコードをメモリにロードするのに使われます。もう一つはGNU as assembly言語でコンパイルされたカーネルプログラムhead.sです。これは

特権レベル3で動作する2つのタスクがクロック割り込みの制御下で相互に切り替わることを実装し、さらに画面に文字を表示するシステムコールを実装しています。

この2つのタスクをタスクAとタスクB（またはタスク0とタスク1）と呼ぶことにします。タスクAとタスクBは、それぞれシステムコールを呼び出して画面に文字「A」と「B」を表示し、10ミリ秒ごとに別のタスクに切り替わります。タスクAは常にシステムコールを呼び出して画面に文字'A'を表示し、タスクBは常に文字'B'を表示します。このカーネルインスタンスプログラムを終了させるには、マシンを再起動するか、実行中の模擬PC実行環境ソフトウェアを終了させる必要があります。

boot.sプログラムは、図4-39に示すように、フロッピーディスクの第1セクターに格納される、合計512バイトのコードを生成します。PCの電源が入ると、ROM BIOSのプログラムは、ブートディスクの第1セクターを物理メモリ0x7c00（31KB）の位置の先頭にロードし、実行制御を0x7c00に移してブートコードの実行を開始します。

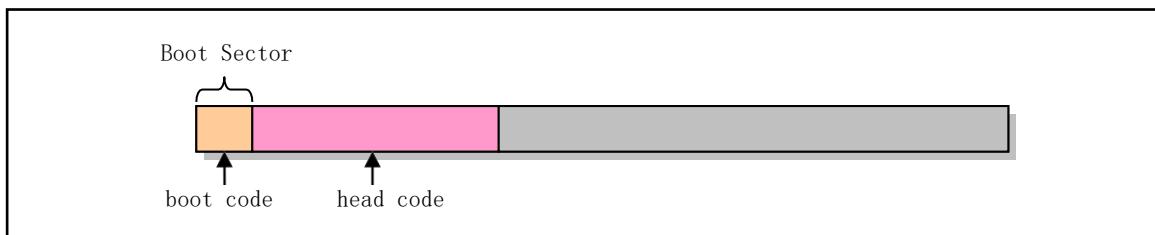


図4-39 フロッピーディスクのイメージファイル

ブートプログラムの主な機能は、フロッピーディスクやイメージファイル内のヘッドカーネルコードをメモリ内の指定された場所にロードすることです。一時的なGDTテーブルなどを設定した後、プロテクトモードで動作するようにプロセッサを設定します。その後、ヘッドコードにジャンプしてカーネルコードを実行します。実際には、boot.sプログラムは、まずROM BIOS割り込みint 0x13を利用して、フロッピーディスク内のヘッドコードをメモリ位置0x10000（64KB）の先頭に読み込んだ後、ヘッドコードをメモリ位置0の先頭に移動させ、最後にコントロールレジスタCR0の保護動作モード有効フラグをセットして、メモリ位置0にジャンプしてヘッドコードの実行を開始します。ブートコードがメモリ内のヘッドコードを移動する様子を図4-40に示します。

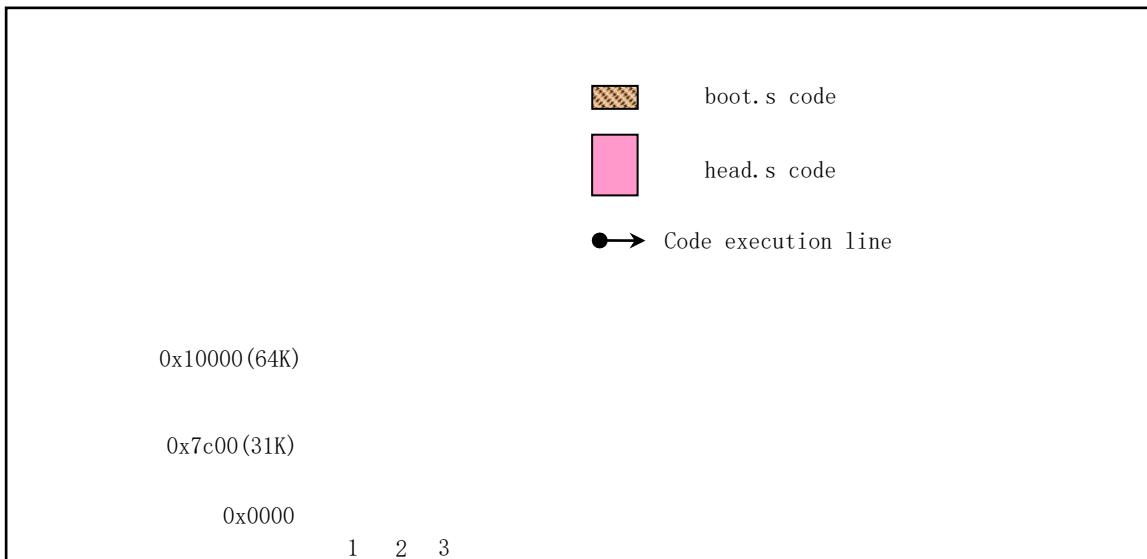


図4-40 物理メモリ上でのカーネルコードの移動・分散の様子

ヘッドのカーネルコードを物理メモリ0の先頭に移動させる主な理由は、GDTテーブルの設定が簡単なので、ヘッド.sのプログラムをできるだけ短くすることもできるからです。しかし、ブートプログラムでフロッピーやイメージファイルからメモリ0番地に直接ヘッドコードをロードさせることはできません。なぜなら、BIOSが使用する割り込みベクトルテーブルがメモリ0番地の先頭にあり、ロード操作にはROM BIOSが提供する割り込みプロセスを使用する必要があるからです。そのため、ヘッドコードをメモリ0に直接ロードすると、BIOSの割り込みベクターテーブルが破壊されてしまいます。

もちろん、ヘッドコードをメモリ0x10000にロードしてから、直接ヘッドコードを実行する場所にジャンプすることも可能です。この方法によるソースプログラムは、以下に説明するようにoldlinux.orgのサイトからダウンロードできます。

ヘッドプログラムは32ビットのプロテクトモードで動作し、主に初期設定のコード、クロック割り込みint0x08のプロセスコード、システムコール割り込みint0x80のプロセスコード、タスクAとタスクBのコードとデータが含まれています。(1)GDTテーブルのリセット、(2)システムタイマチップの設定、(3>IDTテーブルのリセットとクロックおよびシステムコール割り込みゲートの設定、(4)タスクAの実行への移行。仮想アドレス空間において、head.sプログラムのカーネルとタスクのコード割り当て図を図4-41に示す。実際には、このカーネルの例では、コードとデータのセグメントはすべて物理メモリの同じ領域、つまり物理メモリから始まる領域に対応しています。

location 0.

- GDT内のグローバルコードセグメントとデータセグメントディスクリプターの内容は以下のように設定されています。
 - The base address is 0x0000;
 - The segment limit value is 0x07ff. Since the granularity is 1, the actual segment length is 8MB.
- グローバル表示データセグメントの設定は以下の通りです。
 - The base address is 0xb8000;
 - The segment limit length is 0x0002, so the actual segment length is 8KB, corresponding to the display memory area.



図4-41 仮想アドレス空間におけるカーネルとタスクの割り当てのイメージ図

- また、両タスクのLDTにおけるコード・セグメントとデータ・セグメントの記述子の内容は、以下のように設定されています。

- The base address is 0x0000;
- The segment length is 0x03ff and the actual segment length is 4MB.

したがって、リニアアドレス空間では、この「コア」のコードとデータのセグメント、およびタスクのコードとデータのセグメントは、リニアアドレス0から始まり、ページング機構を使用していないので、すべて物理アドレス0の先頭に直接対応しています。ヘッドプログラムがコンパイルしたオブジェクトファイルとその結果のフロッピーアイメージファイルでは、コードとデータの構成は図4-42のようになっています。

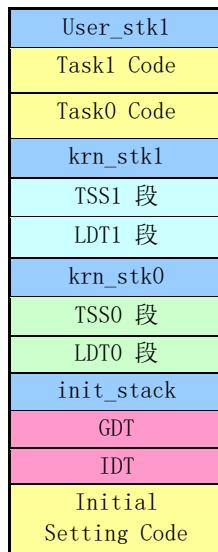


図4-42 カーネルイメージファイルとインメモリーへッドのコード・データ図

特権レベル0のコードを特権レベル3のコードに直接転送することはできません。

が、制御の受け渡しは、割込み復帰命令を使うことで実現できます。そこで、GDT、IDT、タイミングチップが初期化されると、割り込み復帰命令IRETを使って最初のタスクを開始します。

具体的な実装方法としては、初期スタックのinit_stackに手動でリターン環境を設定します。すなわち、タスク0のTSSセグメントセレクタをタスクレジスタLTRにロードし、LDTセグメントセレクタをLDTRにロードした後、タスク0のユーザースタックポインタ（0x17 : init_stack）、コードポインタ（0x0f : task0））、フラグレジスタの値をスタックにpushし、割込み復帰命令IRETを実行します。

IRET命令は、スタック上のスタックポインタをタスク0ユーザースタックポインタとしてポップし、仮想タスク0フラグレジスタの内容を復元し、スタック内のコードポインタをCS:EIPレジスタにポップすることで、タスク0コードの実行を開始する。これで、特権レベル0のコードから特権レベル3のコードへの制御移行が完了します。

実行中のタスクを10ミリ秒ごとに切り替えるためには、10ミリ秒ごとにクロック割り込み要求信号を割り込み制御チップ8259Aに送るように、ヘッド.sプログラムでタイマチップ8253のチャネル0を設定しています。PCの電源投入時には、ROM BIOSプログラムによってクロック割り込み要求信号が8259Aの割り込みベクタ8に設定されているので、割り込み8ハンドラプロシージャでタスク切り替え操作を行う必要があります。タスク切り替えの方法は、変数currentの現在実行中のタスク番号を見て実行します。currentが0であれば、タスク1のTSSセレクタをオペランドにしてファージャンプ命令を実行することで、タスク1に切り替えて実行し、そうでなければその逆となります。

各タスクは、まず文字のASCIIコードをレジスタALに入れ、システム割り込みをかけてint 0x80を呼び出します。システムコールの処理プロセスでは、単純な文字書き込み画面のサブルーチンを呼び出し、レジスタALの文字を画面に表示し、文字を表示した画面の次の位置を次の文字表示の画面位置として記録します。文字が表示された後、タスクコードはloop文を使って一定時間遅延させ、タスクコードの先頭にジャンプして、10ミリ秒の時限割り込みが発生するまでループを続けます。この時点でコードは別のタスクに切り替えて実行されます。タスクAの場合、文字'A'は常にレジスタALに格納され、タスクBの実行中は文字'B'が常にALに格納されます。したがって、プログラムが実行されているときには、図4-43のように、一連の文字'A'と一連の文字'B'が間隔をおいて連続的に画面に表示されることになります。

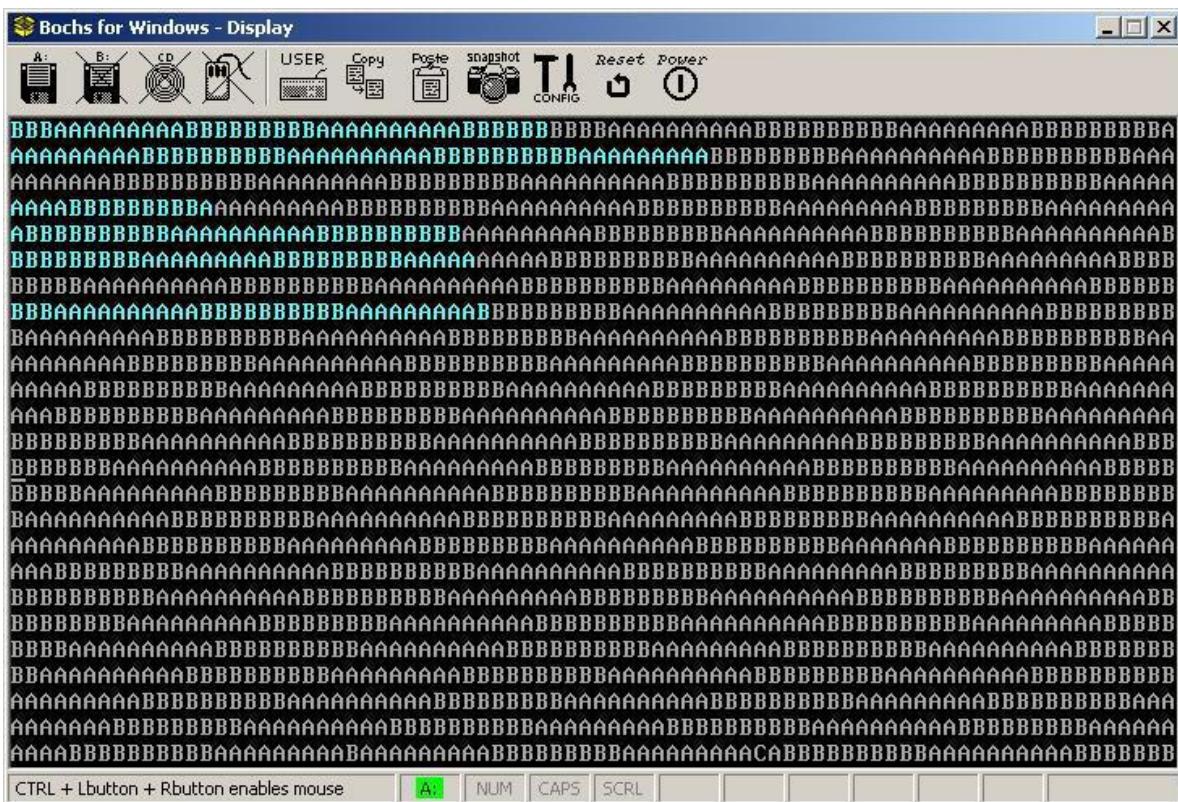


図 4-43 シンプルなカーネル実行画面の表示

図4-43は、この実行中のカーネルをBochsに表示した画面です。注意深い読者は、図の一番下の行に「C」という文字が表示されていることに気づくでしょう。これはPCが誤ってクロック割込みやシステムコール割込みではない別の割込みを発生させたためです。なぜなら、プログラムの中で他の割り込みに対してデフォルトの割り込みハンドラを設置しているからです。別の割り込みが発生すると、システムは文字'C'を表示するコードを含むデフォルトの割り込みハンドラを実行しますので、画面に文字'C'が表示され、割り込みが終了します。

4.9.3 boot.sプログラムとhead.sプログラムの詳細なコメントは以下の通りです。このシンプルなカーネルサンプルのコンパイルと動作については、本書の最終章にある「シンプルなカーネルサンプルプログラムのコンパイルと動作」の項を参照してください。

4.9.4 Boot Startup Program boot.s

プログラムをできるだけシンプルにするために、このブートローダは16セクタ以下のヘッドコードしかロードできず、ROM BIOSが設定した割り込みベクタ番号を直接使用します。つまり、タイマー割り込み要求番号の割り込み番号は8のままでです。これは、Linuxシステムで使われているものとは異なります。Linuxシステムでは、カーネルの初期化時に8259A割り込み制御チップをリセットし、クロック割り込み要求信号を割り込み0x20にマッピングします。詳しくは「カーネルブートプログラム」の章をご覧ください。

01 ! boot.s program

02 ! First use BIOS interrupt to load head code into memory 0x10000, and then move it to memory 0.

03 ! Finally enter protected mode, jump to the beginning of head code at 0 to continue running.

```

04 BOOTSEG = 0x07c0           ! This program is loaded into memory at 0x7c00 by BIOS.
05 SYSSEG  = 0x1000           ! The head is first loaded to 0x10000 and moved to 0x0.
06 SYSLEN  = 17               ! Max num of disk sectors occupied by the kernel.
07 entry start
08 start:
09     jmpi    go, #BOOTSEG   ! Jump between segments to 0x7c0:go. All segment reg
10    go:      mov    ax, cs   ! are 0 when runs. This jump ins load CS with 0x7c0.
11      mov    ds, ax   ! Both DS and SS point to the 0x7c0 segment.
12      mov    ss, ax
13      mov    sp, #0x400    ! Set temp stack pointer. Its value needs to be larger
14                           ! than this program and has a certain space.
15 ! Load the kernel code to the beginning of memory at address 0x10000.
16 load_system:
17      mov    dx, #0x0000    ! Use BIOS int 0x13 func2 to load head code from booted disk.
18      mov    cx, #0x0002    ! DH-head no; DL-drive no; CH-10 bit track no low 8 bits
19      mov    ax, #SYSSEG   ! CL-Bits7,6 track num high 2 bits ,bit 5-0 start sector
20      mov    es, ax   ! ES:BX - Read in buffer location (0x1000:0x0000).
21      xor    bx, bx   ! AH-read sector func num; AL- num of sectors read (17).
22      mov    ax, #0x200+SYSLEN
23      int    0x13
24      jnc    ok_load     ! If no error occurs, then continues, otherwise dead.
25 die:   jmp    die
26
27 ! Move kernel code to memory location 0. Total of 8KB bytes are moved (kernel code <8kb).
28 ok_load:
29      cli                ! Disable interrupts.
30      mov    ax, #SYSSEG   ! Move from DS:SI(0x1000:0) to ES:DI(0:0).
31      mov    ds, ax
32      xor    ax, ax
33      mov    es, ax
34      mov    cx, #0x1000    ! Set the move 4K times, one word at a time.
35      sub    si, si
36      sub    di, di
37      rep    movw          ! Execute the repeat move instruction.
38 ! Load IDT and GDT base address registers IDTR and GDTR.
39      mov    ax, #BOOTSEG
40      mov    ds, ax   ! Let DS point to 0x7c0 segment again.
41      lidt   idt_48     ! Load IDTR. 2 byte table limit, 4 byte linear base addr.
42      lgdt   gdt_48     ! Load GDTR. 2 byte table limit, 4 byte linear base addr.
43
44 ! Set CRO to enter protection mode. The seg selector value 8 refers to 2nd descriptor in GDT.
45      mov    ax, #0x0001    ! Set the protection mode flag PE (bit 0) in CRO.
46      lmsw   ax           ! Jump to segment specified by the selector, offset 0.
47      jmpi   0, 8          ! Seg value is now a selector. The linear base addr is 0.
48
49 ! 以下は、GDTの内容です。3つのセグメンテーションディスクリプタがあります。最初のものは使用されません。
gdt:   .word   0, 0, 0, 0           ! First descriptor not used. Occupies 8 bytes.

```

```
50
51
52     .word 0x07FF          ! Descriptor 1. 8Mb - limit=2047 (2048*4096=8MB).
53     .word 0x0000          ! Segment base address = 0x00000.
54     .word 0x9A00          ! Code segment, readable/executable.
55     .word 0x00C0          ! Segment attribute granularity = 4KB, 80386.
```

```

56
57     .word 0x07FF          ! Descriptor 1. 8Mb - limit=2047 (2048*4096=8MB).
58     .word 0x0000          ! Segment base address = 0x00000.
59     .word 0x9200          ! Data segment, readable and writable.
60     .word 0x00C0          ! Segment attribute granularity = 4KB, 80386.
61 ! The following are the 6-byte operands of the LIDT and LGDT instructions, respectively.
62 idt_48: .word 0           ! The IDT table length is 0.
63     .word 0, 0            ! The linear base address of IDT table is also zero.
64 gdt_48: .word 0x7ff       ! GDT limit is 2048 bytes, can hold 256 descriptors.
65     .word 0x7c00+gdt, 0   ! Linear base of GDT is at offset gdt of seg 0x7c0.
66 .org 510
67     .word 0xAA55         ! Boot sector flag. Must be at last 2 bytes of boot sector.

```

4.9.5 Multitasking Kernel program head.s

保護モードに入った後、head.sのプログラムがIDTテーブルとGDTテーブルを再設定して設定する主な理由は、プログラムの構造を明確にすることと、Linux 0.12カーネルのソースコードにある2つのテーブルの設定との整合性をとるためです。もちろん、このプログラムでは、boot.sで設定されたIDTテーブルとGDTテーブルの位置をそのまま利用して、適切なディスクリプター項目を記入することができます。

01 # Head.sには、32ビットプロテクトモードのinit、クロック&システムコール割り込み、2つの02#タスクコードのコードが含まれています。初期化後、プログラムはタスク0に移動して実行を開始します。

```

# switching operation between tasks 0 and 1 is performed under the clock interrupt.
03 LATCH      = 11930          # Timer count, interrupt is sent every 10 ms.
04 SCRN_SEL   = 0x18           # The segment selector for the screen display memory.
05 TSS0_SEL   = 0x20           # TSS segment selector for task 0.
06 LDT0_SEL   = 0x28           # LDT segment selector for task 0.
07 TSS1_SEL   = 0X30           # TSS segment selector for task 1.
08 LDT1_SEL   = 0x38           # LDT segment selector for task 1.
09 .text
10 startup_32:
11 # First load DS, SS, and ESP. The linear base address of all segments is 0.
12     movl $0x10,%eax        # 0x10 is the data segment selector in the GDT.
13     mov %ax,%ds
14     lss init_stack,%esp
15 # Re-set the IDT and GDT tables at new location.
16     call setup_idt         # Setup IDT.
17     call setup_gdt         # Setup GDT.
18     movl $0x10,%eax        # Reload all segment registers after changing GDT.
19     mov %ax,%ds
20     mov %ax,%es
21     mov %ax,%fs
22     mov %ax,%gs
23     lss init_stack,%esp
24 # Set 8253 timing chip. Channel 0 is set to generate an interrupt request every 10 ms.
25     movb $0x36, %al          # Control word: Channel 0 in mode 3, Count in binary.
26     movl $0x43, %edx         # 0x43 is write port of control word register.
27     outb %al, %dx
28     movl $LATCH, %eax        # Init count set to LATCH (1193180/100), freq. 100HZ.

```

```

29      movl $0x40, %edx          # The port of channel 0.
30      outb %al, %dx          # Write initial count value to channel 0 in two steps.
31      movb %ah, %al
32      outb %al, %dx
33 # Set the timer interrupt gate descriptor at item 8 of the IDT table.
34      movl $0x00080000, %eax    # EAX high word set to kernel code seg selector 0x0008.
35      movw $timer_interrupt, %ax  # Set timer int gate descriptor. Get handler address.
36      movw $0x8E00, %dx          # Interrupt gate type is 14, plevel is 0 or hardware used.
37      movl $0x08, %ecx          # Clock interrupt vector no. set by BIOS is 8.
38      lea idt(%ecx,8), %esi     # Put IDT Descriptor 0x08 address into ESI and set it.
39      movl %eax, (%esi)
40      movl %edx, 4(%esi)

41 # IDTテーブルの項目128 (0x80) にシステムコールのトラップゲート記述子を設定します。
42      movw $system_interrupt, %ax  # Set system call gate descriptor. Get handler address.
43      movw $0xef00, %dx          # Trap gate type is 15, code of plevel 3 is executable.
44      movl $0x80, %ecx          # System call vector no. is 0x80.
45      lea idt(%ecx,8), %esi     # Put IDT Descriptor 0x80 address into ESI and set it.
46      movl %eax, (%esi)
47      movl %edx, 4(%esi)

48 # Now, to use IRET to move to task 0 (A), we manually prepare to setup the stack contents.
# See Figure4-29 for the stack contents we need to setup. Refer to include/asm/system.h.
48      pushfl                  # Reset NT flag in EFLAGS to disable task switch when
49      andl $0xfffffbfff, (%esp)  # execute IRET instruction.
50      popfl
51      movl $TSS0_SEL, %eax      # Load task0's TSS seg selector into task register TR.
52      ltr %ax
53      movl $LDT0_SEL, %eax      # Load task0's LDT seg selector into LDTR.
54      lldt %ax                # TR and LDTR need only be manually loaded once.
55      movl $0, current          # Save current task num 0 into current variable.
56      sti                     # Enable int, build a scene on stack for int returns.
57      pushl $0x17                # Push task 0 data (stack) seg selector onto stack.
58      pushl $init_stack          # Push the stack pointer (same as push ESP).
59      pushfl                  # Push the EFLAGS.
60      pushl $0x0f                # Push current local space code seg selector.
61      pushl $task0                # Push task 0 code pointer onto stack.
62      iret                    # This causes execution moves to task0 in plevel 3.
63

64 # The following are the subroutines for setting descriptor items in GDT and IDT.
65 setup_gdt:                      # GDT table position & limit are set using
66     lgdt lgdt_opcode            # 6-byte operand lgdt_opcode.
67     ret

68 # IDTテーブルに登録されている256個の割込みゲートディスクリプターを #一時的に同じデフォルト値に設定するためのコードです。全てデフォルトの割り込みハンドラ ignore_int を使用します。
# 具体的な設定方法は、まずデフォルトの割り込みゲートディスクリプターの #0~3バイトと4~7バイトの内容をeaxとedxのレジスタペアにセットします。そして、このレジスタペアを使って、割り込みディスクリプタをIDTテーブルにサイクリックに #ファイルします。
69 setup_idt:                      # Set all 256 int gate descriptors to use default handler.
70     lea ignore_int,%edx        # The same way as setting timer int gate descriptor.
71     movl $0x00080000,%eax      # The selector is 0x0008.
72     movw %dx,%ax
73     movw $0x8E00,%dx          # Interrupt gate type is 14, plevel is 0.
74     lea idt,%edi

```

```
75      mov $256,%ecx          # Loop through all 256 gate descriptor entries.
```

```

76 rp_idt: movl %eax, (%edi)
77     movl %edx, 4(%edi)
78     addl $8, %edi
79     dec %ecx
80     jne rp_idt
81     lidt lidt_opcode          # IDTR register is loaded with a 6-byte operand.
82     ret
83
83 # Display characters subroutine. Get current cursor position & display char in AL.
# The entire screen can display 80 X 25 (2000) characters.
84 write_char:
85     push %gs                  # First save the register to be used, EAX is
86     pushl %ebx                # saved by the caller.
87     mov $SCRN_SEL, %ebx       # Then let GS point to display mem seg (0xb8000).
88     mov %bx, %gs
89     movl scr_loc, %bx         # Get current char display position from scr_loc.
90     shl $1, %ebx             # Since each char has one attribute byte, so actual
91     movb %al, %gs:(%ebx)      # display memory offset should multiplied by 2.
92     shr $1, %ebx             # After putting char into display memory, divide the
93     incl %ebx                # position value by 2 plus 1 to get the next position.
94     cmpl $2000, %ebx         # If position is greater than 2000, it is reset to 0.
95     jb 1f
96     movl $0, %ebx
97 1:    movl %ebx, scr_loc      # Finally save this position value (scr_loc),
98     popl %ebx                # and pop up the contents of saved register, return.
99     pop %gs
100    ret
101
102 # The following are 3 interrupt handlers: default, timer, and system call interrupt.
103 # Ignore_int is default handler. If system generates other interrupts, it display char 'C'.
104 .align 2
105 ignore_int:
106     push %ds
107     pushl %eax               # Let DS point to the kernel data segment because
108     movl $0x10, %eax          # the interrupt handler belongs to the kernel.
109     mov %ax, %ds
110     movl $67, %eax            # Put 'C' in AL, call write_char to display on screen.
111     call write_char
112     popl %eax
113     pop %ds
114     iret
115
116 # This is the timer interrupt handler. The main function is to perform task switching operations.
117 .align 2
118 timer_interrupt:
119     push %ds
120     pushl %eax
121     movl $0x10, %eax          # First let DS point to the kernel data segment.
122     mov %ax, %ds
123     movb $0x20, %al            # Then send EOI to 8259A to allow other interrupts.
124     outb %al, $0x20
125     movl $1, %eax
126     cmpl %eax, current

```

```

127      je 1f
128      movl %eax, current
129      ljmp $TSS1_SEL, $0
130      jmp 2f
131 1:   movl $0, current
132      ljmp $TSS0_SEL, $0
133 2:   popl %eax
134      pop %ds
135      iret
136
137 # The system call int 0x80 handler. This example has only one display char function.
138 .align 2
139 system_interrupt:
140     push %ds
141     pushl %edx
142     pushl %ecx
143     pushl %ebx
144     pushl %eax
145     movl $0x10, %edx          # First let DS point to the kernel data segment.
146     mov %dx, %ds
147     call write_char          # Then call routine write_char to display char in AL.
148     popl %eax
149     popl %ebx
150     popl %ecx
151     popl %edx
152     pop %ds
153     iret
154
155 /*****
156 current:.long 0           # Store current task number (0 or 1).
157 scr_loc:.long 0           # Store screen current display position.
158
159 .align 2
160 lidt_opcode:
161     .word 256*8-1          # 6-byte operand for set IDTR : table size & base.
162     .long idt
163 lgdt_opcode:
164     .word (end_gdt-gdt)-1    # 6-byte operand for set IDTR : table size & base.
165     .long gdt
166
167 .align 3
168 idt:   .fill 256,8,0       # IDT. 256 gate descriptors, each 8 bytes, total 2KB.
169 # The following is GDT table contents (of descriptors).
170 gdt:   .quad 0x0000000000000000 # [0] The first segment descriptor is not used.
171     .quad 0x00c09a00000007ff # [1] Kernel code descriptor. Its selector is 0x08
172     .quad 0x00c09200000007ff # [2] Kernel data descriptor. Its selector is 0x10
173     .quad 0x00c0920b80000002 # [3] Display mem descriptor. Its selector is 0x18
174     .word 0x68, tss0, 0xe900, 0x0 # [4] TSS0 descriptor. Its selector is 0x20.
175     .word 0x40, ldt0, 0xe200, 0x0 # [5] LDTO descriptor. Its selector is 0x28
176     .word 0x68, tss1, 0xe900, 0x0 # [6] TSS1 descriptor. Its selector is 0x30
177     .word 0x40, ldt1, 0xe200, 0x0 # [7] LDT1 descriptor. Its selector is 0x38
178 end_gdt:
179     .fill 128,4,0           # Initial kernel stack space.

```

```

180 init_stack:          # Stack pointer when first enter protected mode.
181     .long init_stack # Stack segment offset position.
182     .word 0x10         # Stack segment, same as kernel data seg (0x10).
183
186 184 # 以下はタスク0のLDTテーブルセグメントのローカルセグメント記述子です。
185 .align 3
187 ldt0:    .quad 0x0000000000000000      # [0] The first descriptor is not used.
188     .quad 0x00c0fa00000003ff      # [1] The local code descriptor, its selector is 0x0f
189     .quad 0x00c0f200000003ff      # [2] The local data descriptor, its selector is 0x17
189 # Content of TSS seg for task 0. Note fields such as labels do not change when task switches.
190 tss0:   .long 0           /* back link */
191     .long krn_stk0, 0x10      /* esp0, ss0 */
192     .long 0, 0, 0, 0, 0       /* esp1, ss1, esp2, ss2, cr3 */
193     .long 0, 0, 0, 0, 0       /* eip, eflags, eax, ecx, edx */
194     .long 0, 0, 0, 0, 0       /* ebx esp, ebp, esi, edi */
195     .long 0, 0, 0, 0, 0       /* es, cs, ss, ds, fs, gs */
196     .long LDT0_SEL, 0x8000000 /* ldt, trace bitmap */
197
198     .fill 128,4,0          # This is the kernel stack space for task 0.
199 krn_stk0:
200
203 201 # Task 1 LDTテーブルの内容とTSSセグメントの内容。
202 .アライメント 3
204 ldt1:    .quad 0x0000000000000000      # [0] The first descriptor is not used.
205     .quad 0x00c0fa00000003ff      # [1] The selector is 0x0f, base = 0x00000.
206     .quad 0x00c0f200000003ff      # [2] The selector is 0x17, base = 0x00000.
206
207 tss1:   .long 0           /* back link */
208     .long krn_stk1, 0x10      /* esp0, ss0 */
209     .long 0, 0, 0, 0, 0       /* esp1, ss1, esp2, ss2, cr3 */
210     .long task1, 0x200        /* eip, eflags */
211     .long 0, 0, 0, 0          /* eax, ecx, edx, ebx */
212     .long usr_stk1, 0, 0, 0   /* esp, ebp, esi, edi */
213     .long 0x17,0x0f,0x17,0x17,0x17,0x17 /* es, cs, ss, ds, fs, gs */
214     .long LDT1_SEL, 0x8000000 /* ldt, trace bitmap */
215
216     .fill 128,4,0          # This is the kernel stack space for task 1. Its user
217 krn_stk1:                         # stack uses the initial kernel stack space directly.
218
219 # タスク0とタスク1のプログラムで、それぞれ文字'A'と'B'を循環的に表示します。220 task0:
222     movl $0x17, %eax          # DS point to the local data segment of the task.
223     movw %ax, %ds             # No local data, these 2 instructions can be omitted.
224     movl $65, %al              # Put 'A' into the AL register.
225     int $0x80                 # Execute system call to display it.
226     movl $0xffff, %ecx          # Execute a loop, act as a delay.
226 1:    loop 1b
227     jmp task0                # Jump to start of task 0 to continue displaying.
228 task1:
229     movl $66, %al              # Put 'B' into the AL register.
230     int $0x80                 # Execute system call to display it.
231     movl $0xffff, %ecx          # Execute a loop, act as a delay.
232 1:    loop 1b

```

```
233     jmp task1
234
235     .fill 128,4,0          # This is user stack space for task 1.
236 usr_stk1:
```

4.10 Summary

この章では、インテル80X86 CPUの保護モードのメモリ管理とプログラミングの原理について説明します。この章では、グローバル/ローカルディスクリプターテーブル、セグメントディスクリプタ、セグメントセレクタの具体的な意味を詳しく説明しています。また、プログラムの論理アドレス、CPUのリニアアドレス、物理メモリのアドレスの変換関係についても説明しています。最後に、簡単なカーネルのサンプルプログラムが与えられ、本章の最後に紹介されています。このサンプル・プログラムを理解することで、保護モード・プログラミングの習得度を大まかに説明することができます。

以下では、1つの章を使って、Linuxカーネルのハードウェア設定、メモリの割り当てと使用方法、タスクデータ構造の機能などを包括的に説明し、カーネルのソースツリーにあるすべてのソースコードを分類して、まず読者にカーネルコード全体のファイル構造を大まかに理解してもらいます。そして、次の章では、カーネル内のソースコードファイルを詳細に説明し、注釈を付けています。

5 Linux kernel architecture

本章は、カーネルのソースコードの概要を説明したもので、後続の章を読む際の参考にしていただけです。理解しづらい内容については、まず読み飛ばしてください。次の章で関連する内容を読むときには、この章を参照するように戻ってください。本章を読む前に、80X86のプロジェクトモード動作の仕組みを確認・学習してください。

本章では、まず Linux カーネルの構成モードとアーキテクチャの概要を説明した後、カーネルソースディレクトリ内のソースファイル構成、サブディレクトリ内の各種コードファイルの主な機能、基本コールの階層関係を詳細に説明しています。その後、直接トピックに切り込み、カーネルソースファイルLinux /ディレクトリ内の最初のファイルMakefileから始めて、各行のコードを詳しく説明します。Linux 0.12カーネルのソースコードをもとに、基本的なアーキテクチャと主要なコンポーネントを簡単に説明します。また、ソースコードに登場するいくつかの重要なデータ構造についても説明します。最後に、Linux 0.12カーネルのコンパイル実験環境を構築する方法を説明します。

完全なオペレーティングシステムは、レイヤーの観点から見ると、図5-1に示すように、ハードウェア、オペレーティングシステムカーネル、オペレーティングシステムサービス、ユーザーアプリケーションの4つの部分から構成されます。ユーザーアプリケーションとは、ユーザー自身がコンパイルしたワードプロセッサやインターネットブラウザプログラムなどの各種アプリケーションのことであり、オペレーティングシステムサービスとは、ユーザーにサービスを提供するものであり、オペレーティングシステムの一部とみなされる。Linuxオペレーティングシステムでは、Xウィンドウシステム、シェルコマンド解釈システム、カーネルプログラミングインターフェースなどのシステムプログラムがこれにあたる。オペレーティングシステムのカーネルは、本書の中でも特に興味を引く部分である。主にハードウェアのリソースを抽象化し、すべてのシステムリソースの管理をスケジューリングするために使用されます。

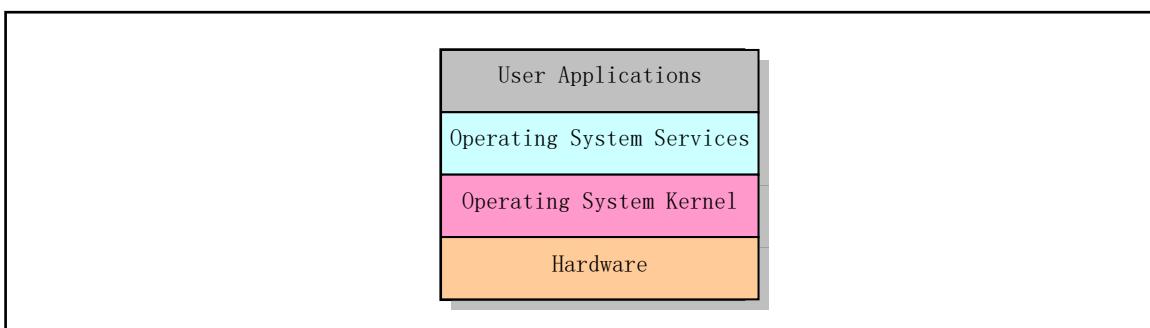


図5-1 オペレーティングシステムの構成要素

Linuxカーネルの主な目的は、コンピュータのハードウェアと相互作用し、コンポーネントのインターフェース操作やプログラムによる制御を実装し、ハードウェアリソースへのアクセスをスケジューリングし、コンピュータ上のユーザープログラムに使いやすい実行環境と共にハードウェア仮想インターフェースを提供することです。.

5.1 Linux kernel mode

現在、オペレーティングシステムのカーネルの構造モードは、主にモノリシックなものと、それ以外のものに分けられます。

シングルコアモデルと階層型マイクロカーネルモデル、そして両者の混合モードがあります。本書で注釈を付けたLinux 0.12カーネルは、シングルコアモードを採用しています。

モノリシック・シングルコア・システム・モデルでは、オペレーティング・システムが提供するサービス・プロセスは、アプリケーション・プログラムが指定されたパラメーターでシステム・コール命令 (int x80) を実行することで、CPUがユーザー・モードからコアの状態に切り替わります（カーネル・モデル）。システムコール命令を実行すると、OSは指定されたパラメータに従って特定のシステムコールサービスプロシージャを呼び出し、これらのサービスプロシージャは必要に応じて基礎となるサポート機能の一部を呼び出して特定の機能を完了します。アプリケーションが必要とするサービスが完了すると、OSはCPUをカーネルモードからユーザー・モードに戻し、アプリケーションに戻って次の命令の実行を継続します。つまり、シングルコアモードのカーネルは、サービスを呼び出すメインプログラム層、システムコールを実行するサービス層、システムコールをサポートする基盤機能の3つのレベルに大別することができます。図5-2をご覧ください。モノリシック・モデルの主な利点は、カーネル・コードがコンパクトで高速であることであり、欠点は主に階層が強固でないことです。

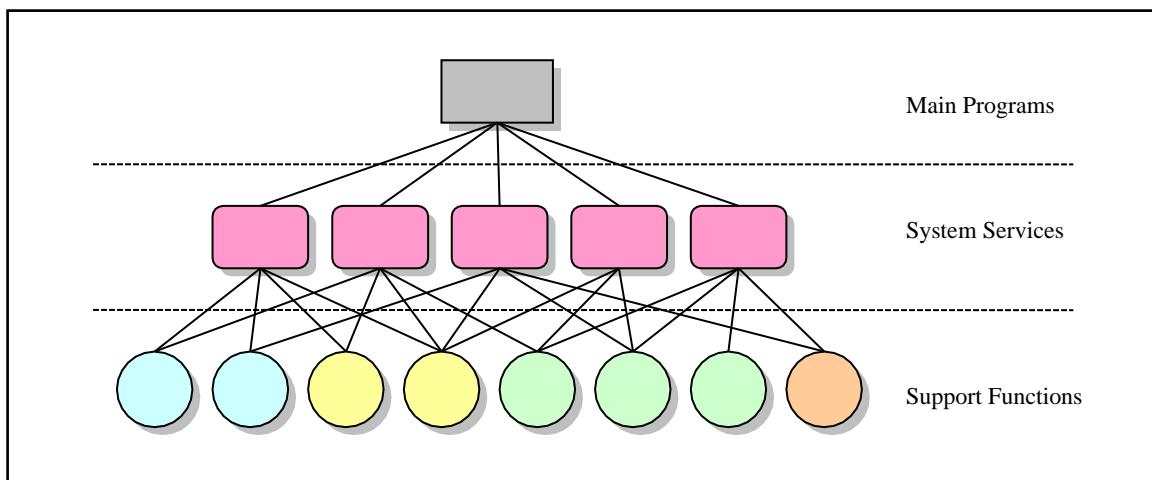


図5-2 モノリシックモデルのシンプルな構造

マイクロカーネルアーキテクチャモデルでは、機能のモジュール化と、サービススレッドまたはプロセス間のメッセージングが主な特徴です。システムコアは、基本的なハードウェア抽象化管理層と、主要なシステムサービス機能を提供します。これらの主要な機能とは、メインプロセス/スレッド間通信サービス、仮想メモリ管理、プロセススケジューリングなどです。オペレーティングシステムの残りの部分は、さまざまなモジュール形式でユーザースペースで機能します。したがって、マイクロカーネル構造の利点は、システムサービスの結合度が低いため、システムの改良、拡張、移植が容易であることです。一方、主な欠点は、実行中にシステム・サービス・モジュール間で大量のメッセージ・パッシングや同期操作が必要となり、これらの操作が通信リソースの消費や時間の遅延を引き起こすことです。代表的なマイクロカーネル・アーキテクチャのシステムとしては、Machカーネルを搭載したMINIX OSやMac OSシステムなどがある。

5.2 Linux kernel system architecture

Linuxカーネルは、図5-3に示すように、プロセススケジューリングモジュール、メモリ管理モジュール、ファイルシステムモジュール、プロセス間通信モジュール、ネットワークインターフェースモジュールの5つのモジュールから構成されている。

プロセススケジューリングモジュールは、プロセスによるCPUリソースの使用を制御する役割を担っています。のです。

各プロセスが公平かつ合理的にCPUにアクセスできるようにする一方で、カーネルがタイムリーにハードウェア処理を実行できるようにするというスケジューリング戦略が採用されています。メモリ管理モジュールは、すべてのプロセスがマシンのメインメモリ領域を安全に共有できるようにするために使用されます。同時に、メモリ管理モジュールは仮想メモリ管理モードもサポートしており、Linuxサポートプロセスが実際のメモリ空間よりも多くのメモリ容量を使用するようになっています。ファイルシステムを使用して、未使用的メモリブロックを外部記憶装置にスワップし、必要に応じて交換することができます。ファイルシステムモジュールは、外部デバイスのドライブとストレージをサポートするために使用されます。仮想ファイルシステムモジュールは、すべての外部ストレージデバイスに共通のファイルインターフェースを提供することで、さまざまなハードウェアデバイスの異なる詳細を隠します。これは、他のオペレーティングシステムと互換性のある複数のファイルシステムフォーマットを提供し、サポートします。プロセス間通信モジュールは、複数のプロセス間の情報交換をサポートするために使用されます。ネットワークインターフェースモジュールは、様々なネットワーク通信規格へのアクセスを提供し、多くのネットワークハードウェアをサポートします。これらのモジュール間の依存関係を図5-3に示す。接続部は各モジュール間の依存関係を表し、破線と破線のボックスはLinux 0.12の未実装部分を表している（仮想ファイルシステムはLinux 0.95バージョンから徐々に実装されているが、ネットワークインターフェースのサポートはバージョン0.96以降でしか利用できない）。

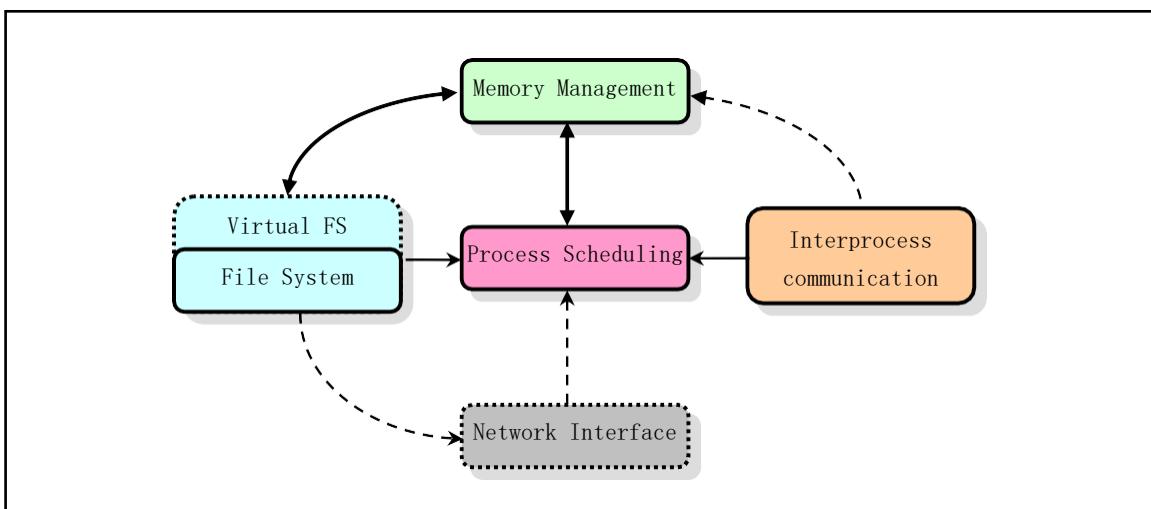


図5-3 Linuxカーネルのモジュール構造と相互依存性

図からわかるように、すべてのモジュールは、プロセススケジューリングモジュールに依存しています。なぜなら、すべてのモジュールはプロセススケジューラーに依存して、プロセスをサスペンド（一時停止）したり、再実行したりしているからです。一般的に、モジュールはハードウェアの操作を待っている間にサスペンドされ、操作が完了するまで実行され続けます。例えば、あるプロセスがデータのブロックをフロッピーディスクに書き込もうとしたとき、フロッピーディスクのドライバ

ーは、ブートフロッピーディスクの回転中にプロセスをサスペンド待機状態にし、フロッピーディスクが通常の回転に入った後にプロセスを継続させることができます。他の3つのモジュールも同様の理由で、プロセススケジューリングモジュールに依存しています。

他のいくつかのモジュールの依存関係は、やや目立ちませんが、重要なものもあります。プロセススケジューリングサブシステムは、メモリ管理を使用して、特定のプロセスが使用する物理メモリ空間を調整する必要があります。プロセス間通信サブシステムは、メモリマネージャを使用して共有メモリ通信メカニズムをサポートします。この通信メカニズムでは、2つのプロセスがメモリの同じ領域にアクセスして、プロセス間で情報を交換することができます。また、仮想ファイルシステムでは

NFS (Network File System) をサポートするネットワークインターフェースと、メモリ・ラムディスク・デバイスを提供するメモリ管理サブシステムを備えています。また、メモリ管理サブシステムは、ファイルシステムを利用してメモリブロックのスワップをサポートします。

モノリシック構造モデルから、Linux 0.12カーネルのソースコードの構造に合わせて、カーネルのメインモジュールを図5-4のようなブロック図構造に描くこともできます。

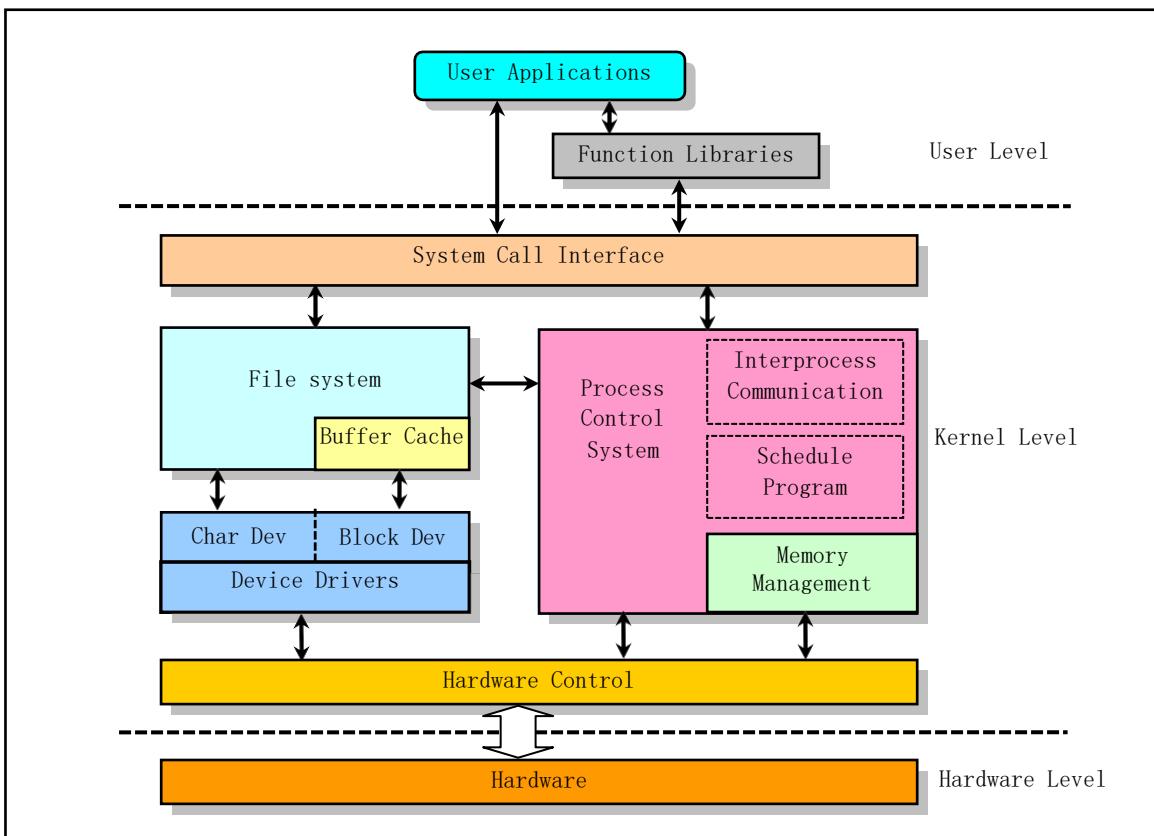


図5-4 カーネルのブロック図

カーネルレベルのいくつかのボックスでは、ハードウェア制御ブロック以外の太い線のボックスが、カーネルソースコードのディレクトリ構成に対応しています。これらの図に示されている依存関係に加えて、これらのモジュールはすべて、カーネル内の共通リソースにも依存しています。これらのリソースには、メモリの割り当てと再利用機能、警告やエラーメッセージを表示する機能、システムのデバッグ機能などがあります。

5.3 Linux kernel memory management

本節では、まず、Linux 0.12システムにおける比較的わかりやすい物理メモリの使い方を説明します。次に、Linux 0.12カーネルのアプリケーション状況と組み合わせて、メモリのセグメンテーションとページング管理のメカニズム、およびCPUのマルチタスク動作と保護モードについて概説します。最後に、Linuxにおけるカーネルコードとデータの対応関係を包括的に説明します。

0.12システムと、仮想・線形・物理アドレス空間における各タスクのコードとデータ。

本項の記述は、メモリ管理の概要やおさらいといえます。プロテクションモードでのメモリ管理の詳細については、第4章を参照してください。

5.3.1 Physical address

Linux 0.12カーネルでは、マシン内の物理メモリを有効活用するため、図5-5に示すように、システムの初期化段階でメモリをいくつかの機能領域に分割しています。

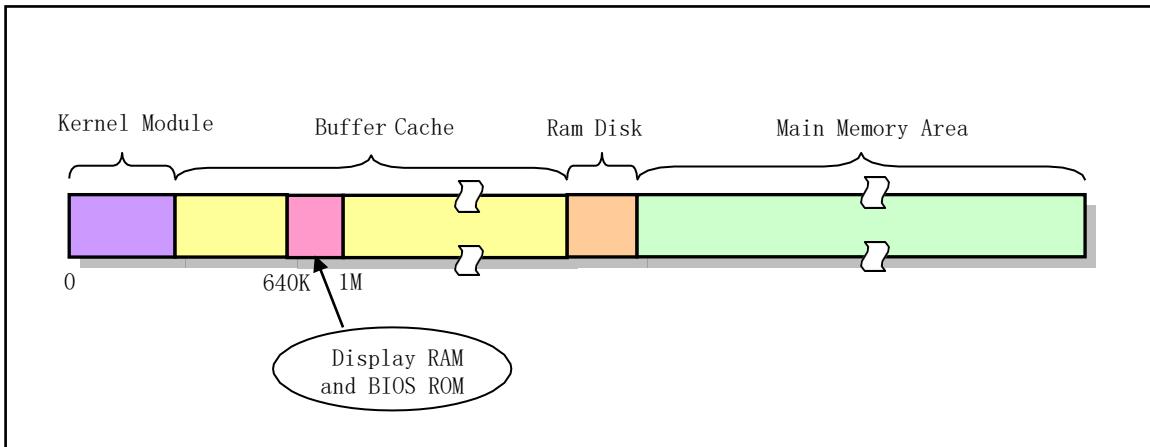


図5-5 物理メモリ使用量の機能別分布

その中で、Linuxカーネルは物理メモリの先頭を占め、次にハードディスクやフロッピーディスクなどのブロックデバイス用の高速バッファ部分が続く（この中で、ディスプレイカードメモリやROM BIOSのメモリアドレス範囲は640K～1MB）。プロセスやタスクがブロックデバイスからデータを読み出す必要がある場合、システムはまずデータをキャッシュに読み込みます。また、ブロックデバイスに書き込むべきデータがある場合、システムはまずデータをキャッシュに入れ、次にブロックデバイスドライバが対応するデバイスに書き込みます。メモリの最後の部分は、すべてのプログラムがいつでも要求して使用できる主記憶領域である。カーネルプログラムがメインメモリ領域を使用する際にも、まずカーネルメモリ管理モジュールに申請する必要があり、申請が成功した後にメモリを使用することができるようになります。また、RAM仮想ディスクを搭載したシステムでは、仮想ディスクがデータを格納するために、主記憶領域のヘッダが部分的に削除される。

5.3.2 コンピュータシステムに含まれる実際の物理メモリの容量は限られているため、通常、CPUはシステム内のメモリを効率的に管理するためのメモリ管理機構を提供している。Intel 80386以降のCPUでは、「メモリセグメンテーション」と「ページング管理システム」という2つのメモリ管理（アドレス変換）機構が用意されている。ページング管理システムはオプションであり、採用するかどうかはシステムプログラマがプログラムします。Linuxシステムでは、物理的なメモリを有効に使うために、メモリセグメンテーションとページング管理の両方の仕組みを採用しています。

5.3.3 Memory Address Space Concept

Linux 0.12カーネルでは、アドレスマッピングを行う際に、まず、a)プログラム（プロセス）の仮想・論理アドレス、b)CPUのリニアアドレス、c)実際の物理メモリアドレスの3種類のアドレスと、それらの間の変換の概念を区別する必要がある。

仮想アドレスとは、プログラムが生成するアドレスのことで、セグメントセレクタとセグメント内のオフセットアドレスの2つの部分から構成されています。この2つの部分は、物理メモリへのアクセスには直接使用されず、セグメントアドレス変換機構によって物理メモリのアドレスに対応するよう処理またはマッピングされる必要があるため、このようなアドレスは仮想アドレスと呼ばれます。仮想アドレス空間は、GDTでマッピングされたグローバルアドレス空間と、LDTでマッピングされたローカルアドレス空間で構成されています。セレクタのインデックス部は13ビットで表現され、これに以下を区別する1ビットが加わります。

GDTとLDTの間にあるため、インテル80X86のCPUは合計16384個のセレクタをインデックスすることができます。各セグメントの長さが最大4Gの場合、最大の仮想アドレス空間は $16384 * 4G = 64T$ となる。

論理アドレスは、プログラムが生成するセクションに関連するオフセットアドレスの部分です。インテルのプロテクトモードでは、プログラム実行コードセクションの制限長内のオフセットアドレスを指します（コードセクションとデータセクションが同一であると仮定します）。アプリケーションプログラマは論理アドレスだけを扱えばよく、セグメンテーションやページングの仕組みは、システムプログラマにしか全く見えない。しかし、資料によっては、論理アドレスと仮想アドレスの概念を区別せず、総称して論理アドレスと呼んでいるものもある。

リニアアドレスは、仮想アドレスと物理アドレスの変換の中間層であり、プロセッサのアドレス可能なメモリ空間（リニアアドレス空間と呼ばれる）におけるアドレスである。プログラムコードは、論理アドレス、つまりセグメント内のオフセットアドレスに、対応するセグメントのベースアドレスを加えてリニアアドレスを生成します。ページング機構が有効な場合は、リニアアドレスを変換して物理アドレスを生成することができます。ページング機構が有効になっていない場合は、リニアアドレスがそのまま物理アドレスになります。Intel 80386のリニアアドレス空間は4Gである。

Physical Addressは、CPUの外部アドレスバス上でアドレスされた物理メモリを示すアドレス信号で、アドレス変換の最終結果のアドレスとなります。ページング機構が有効な場合、リニアアドレスはページディレクトリとページテーブルの項目を用いて物理アドレスに変換されます。ページング機構が有効でない場合は、リニアアドレスがそのまま物理アドレスになります。

仮想記憶（または仮想メモリ）とは、コンピュータが実際に持っているメモリ量よりもはるかに大きなメモリを提示することである。そのため、プログラマーは実際のシステムが持っているよりもはるかに大きなサイズのプログラムをプログラミングして実行することができます。これにより、限られたメモリ資源のシステム上で、多くの大規模なプロジェクトを実行することができます。非常に適切な例えは、上海から北京まで列車を走らせるのに、長い線路は必要ないということです。このタスクを完了するのに十分な長さのレール（例えば10km）があればいいのです。方法としては、すぐに後部のレールを列車の前に敷くことです。操作が十分に速く、要件を満たすことができれば、列車は完全なトラックのように走ることができます。これが、仮想メモリ管理が達成すべきタスクです。Linux 0.12カーネルでは、各プログラム（プロセス）は、合計64MBの容量を持つ仮想メモリ空間に分割されています。そのため、プログラムの論理アドレス範囲は0x00000000～0x40000000となります。

5.3.4 前述のように、論理アドレスを仮想アドレスと呼ぶこともあります。なぜなら、論理アドレスは仮想メモリ空間の概念に似ており、実際の物理メモリの容量とは独立しているからである。

5.3.5 Memory Segmentation Mechanism

メモリセグメンテーションシステムでは、プログラムの論理アドレスは、セグメンテーション機構により、中間層の4GB (2^{32}) リニアアドレス空間に自動的にマッピング（変換）される。プログラムがメモリを参照することは、メモリセグメント内のメモリを参照することになります。プログラムがメモリアドレスを参照すると、プログラマに見える論理アドレスに対応するセグメントベースアドレスを加えて、対応するリニアアドレスが形成されます。このとき、ページング機構が有効になっていなければ、リニアアドレスはCPUの外部アドレスバスに送られ、対応する物理メモリに直接アドレッシングされます。図5-6をご覧ください。

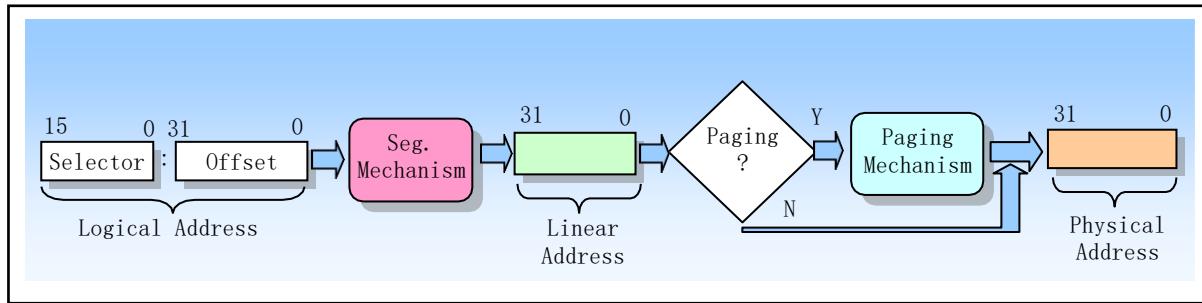


図5-6 論理アドレス-物理アドレス変換

CPUがアドレス変換（マッピング）を行う主な目的は、仮想メモリ空間から物理メモリ空間へのマッピング問題を解決することである。仮想記憶空間の意味は、二次記憶空間や外部記憶空間を利用して、実際の物理メモリの量に制限されることなくプログラムがメモリを使用できるようにする方法を指す。通常、仮想メモリ空間は実際の物理メモリよりもはるかに大きい。

では、仮想ストレージの管理はどのように行われるのでしょうか。その原理は、上記の列車運行のアナロジーに似ています。まず、プログラムが存在しないメモリを使用する必要がある場合（つまり、メモリページテーブルのエントリに対応するメモリページがメモリ内にない場合）、CPUはその状況を知る方法が必要です。これを実現するのが、80386のページフォルト例外割り込みです。プロセスが存在しないページのメモリアドレスを参照すると、CPUはページフォルト例外割り込みを発生させ、割り込みの原因となったリニアアドレスをCR2コントロールレジスタに入れます。したがって、割り込みを処理するプロセスは、ページ例外の正確なアドレスを知ることができるので、プロセスが要求するページを二次記憶空間（ハードディスクなど）から物理メモリにロードすることができます。このとき、物理メモリがすでに占有されている場合は、二次記憶空間の一部をスワップバッファ（Swapper）として使用し、二次バッファ内の一時的に使用されていないページを交換してから、要求されたページをメモリに転送することができます。in. これは、メモリ管理のページフォルトローディング機構である。Linux 0.12カーネルのmm/memory.cというプログラムで実装されている。

インテルのCPUは、プログラムのアドレスにセグメントという概念を用いています。各セグメントには、メモリ上の領域やアクセスの優先順位などの情報が定義されています。ここでは、リアルモードでのメモリアドレッシングの原理は誰もが知っているとして、32ビットプロテクトモードの動作メカニズム下でのメモリアドレッシングの主な特徴を、リアルモードとプロテクトモードでのCPUのアドレッシングモードの違いに応じて、比較法を用いて簡単に説明する。

リアルモードでは、メモリアドレスのアドレッシングには主にセグメント値とオフセット値を使用します。セグメント値はセグメントレジスタ（DSなど）に格納され、セグメントの長さは64KBに固定されています。セグメント内のオフセットアドレスは、アドレス指定に使用できる任意のレジス

タ（例：SI）に格納されます。したがって、セグメント・レジスタとオフセット・レジスタの値に基づいて、図5-7（a）に示すように、実際のポインティッド・メモリ・アドレスを計算することができます。

プロテクトモードモードでは、セグメントレジスタは、アドレス指定されたセグメントのベースアドレスではなく、セグメントディスクリプター-tableのディスクリプター項目のインデックス値となります。インデックス値で指定されたセグメントディスクリプター項目には、アドレスするメモリセグメントのベースアドレス、セグメントの限界長、セグメントのアクセス権レベルなどの情報が含まれています。指定されたメモリ・ロケーションは、セグメント記述子項目で指定されたセグメント・ベース・アドレスとセグメント内のオフセットの組み合わせです。セグメントの限界長は可変で、記述子の内容で指定されます。リアル・モードでのアドレス指定と比較して、セグメント・レジスタの値が次のようなインデックス値に置き換えられていることがわかります。

セグメントディスクリプターテーブルの対応するセグメントディスクリプターと、セグメントディスクリプターテーブル選択ビットと特権レベルを合わせて、セグメントセレクターと呼びますが オフセット値は、やはりオリジナルモードでの概念を使用します。このように、プロテクトモードでメモリアドレスを指定するには、セグメントディスクリプターテーブルを使用するリアルモードに比べて、1つ多くの手順が必要になります。これは、プロテクトモードではメモリセグメントにアクセスするための情報が多く、16ビットのセグメントレジスタではこの内容をあまり保持できないためです。その回路図を図5-7 (b) に示す。なお、セグメントディスクリプタでメモリリニアアドレス空間の領域を定義しないと、そのアドレス領域は全くアドレスされず、CPUはそのアドレス領域へのアクセスを拒否します。

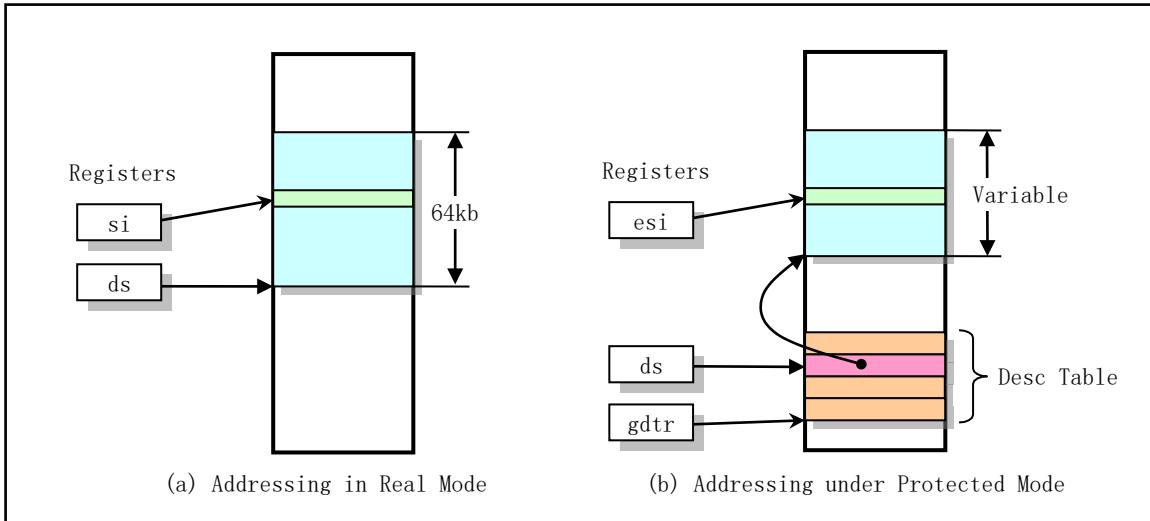


図5-7 リアルモードとプロテクトモードのアドレッシングの比較

各記述子は8バイトで、リニアアドレス空間における記述されたセグメントの開始アドレス（ベースアドレス）、セグメントの限界長、セグメントの種類（コードセグメントやデータセグメントなど）、セグメントの特権レベル、その他の情報を含みます。1つのセグメントが定義できる最大の長さは4GBです。

ディスクリプタ項目を保存するディスクリプターテーブルには3種類あり、それぞれ目的が異なります。グローバルディスクリプターテーブル（GDT）は、すべてのプログラムがメモリセグメントを参照するために使用できるメインベースのディスクリプターテーブルです。割り込み記述子テーブル（IDT）は、割り込みや例外処理のプロセスを定義するセグメント記述子を保持します。IDTテーブルは、8086システムの割り込みベクターテーブルを直接置き換えるものです。80X86のプロテクトモードで正常に動作させるためには、CPUにGDTテーブルとIDTテーブルを定義する必要があります。最後のタイプのテーブルは、LDT（Local Descriptor Table）です。このテーブルは、マルチタスクシステムで使用され、通常はタスクごとに1つのLDTテーブルを使用します。GDTテーブルの拡張機能として、各LDTテーブルは対応するタスクに対してより多くの利用可能な記述子エントリを提供し、各タスクにアドレス可能なメモリ空間の範囲を提供します。

これらのテーブルは、リニアアドレス空間のどこにでも保存することができます。GDTテーブル

ル、IDTテーブル、現在のLDTテーブルの位置をCPUが特定するために、GDTR、IDTR、LDTRの3つの特別なレジスタをCPUに設定する必要があります。これらのレジスタには、対応するテーブルの32ビットリニアベースアドレスと、テーブルの限界長バイト値が格納される。テーブルの長さ値は、テーブルの長さ値-1となります。

CPUがセグメントをアドレス指定する際には、16ビットのセグメントレジスタのセレクタを使ってセグメントディスクリプタを探します。80X86 CPUでは、セグメントレジスタの値を3ビット右にシフトした値が、ディスクリプターテーブルのディスクリプターのインデックス値となります。13ビットのインデックス値は、8192（0--8191）までの位置を特定できます。

ディスクリプタのエントリです。セレクタビット2 (TI) は、どのテーブルを使用するかを指定するために使用されます。このビットが0の場合、セレクタはGDTテーブルの記述子を指定し、それ以外の場合はLDTテーブルの記述子を指定します。

各プログラムは、複数のメモリセグメントで構成されます。プログラムの論理アドレス（または仮想アドレス）は、これらのセグメントの特定のアドレス位置を指定するために使用されます。Linux 0.12では、プログラムの論理アドレスからリニアアドレスへの変換処理に、グローバルセグメント記述子テーブルGDTとローカルセグメント記述子テーブルLDTを使用しています。GDTでマッピングされたアドレス空間をグローバルアドレス空間、LDTでマッピングされたアドレス空間をローカルアドレス空間と呼び、両者で仮想アドレスの空間を構成している。具体的な使い方を図5-8に示す。

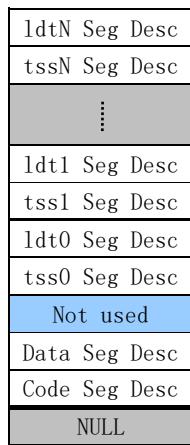


図 5-8 Linux における仮想アドレス空間の割り当てマップ

図は、タスクが2つある場合の状況を示しています。各タスクのローカルディスクリプターテーブルLDTも、GDTのディスクリプターで定義されたメモリーセグメントであり、その中に対応するタスクのコードセグメントとデータセグメントのディスクリプターが格納されているため、LDTのセグメントは非常に短いことがわかります。セグメントの長さは通常、24バイトより大きければ良いとされています。同様に、各タスクのタスク・ステータス・セグメントTSSも、GDT内の記述子によって定義されるメモリ・セグメントであり、セグメント長の制限は、TSSのデータ構造を格納する能力を満足するものであれば十分である。

カーネルのコードセグメントに格納されています。Linux 0.12カーネルでは、カーネルと各タスクのコードセグメントとデータセグメントは、それぞれリニアアドレス空間の同じベースアドレスにマッピングされており、セグメント長も同じであるため、カーネルのコードセグメントとデータセグメントはオーバーラップしています。図5-10または図5-11に示すように、各タスクのコードセグメントとデータセグメントもそれぞれオーバーラップしている。タスクステートセグメント(TSS)は、タ

スクが切り替わったときに、関連するタスクの現在の実行コンテキスト(CPUの現在の状態)を自動的に保存または復元するためのものである。例えば、タスクが切り替わった場合、CPUはそのタスクのTSSセグメントにそのレジスタなどの情報を保存し、新たにタスクに切り替わったTSSセグメントの情報を使って各レジスタを設定し、新たなタスクの実行環境を復元します。

Linux 0.12では、各タスクのTSSセグメントの内容は、そのタスクのタスクデータ構造に保存されます。また、Linux 0.12のカーネルでは、GDTテーブルの4番目の記述子（図中のsyscall記述子のエントリ）は使用されていません。示されたinclude/linux/sched.hファイルの201行目のオリジナル英語コメントより

以下のことから、Linus氏がカーネルを設計する際に、システムコールのコードをこの特殊なセクションに配置するように設計したことが推測されます。

```
200 /*  
201 * 最初のTSSを見つけるためのgdtへの入力。0-nul, 1-cs, 2-ds, 3-syscall 202 * 4-  
TSS0,5-LDT0, 6-TSS1など ...  
203 */
```

5.3.6 Memory paging management

ページングを使用する場合、リニアアドレスは単なる中間結果であり、ページングメカニズムを使用して変換し、最終的に実際の物理メモリアドレスにマッピングする必要があります。セグメンテーションと同様に、ページング機構では、各メモリ参照を特定の要件に合わせてリダイレクト（変換）することができます。最も一般的な使い方は、システムメモリが多くのブロックに分割されている場合に、ページングを行うことで、連続した大きなメモリ空間イメージを作成し、プログラムがこれらの散乱したメモリブロックを気にせずに管理できるようにすることです。ページング機構は、セグメンテーション機構の性能を向上させます。また、ページアドレス変換はセグメント変換に基づいて行われます。ページング・メカニズムの保護手段は、セグメント変換の保護手段に取って代わるものではなく、さらなる確認作業を行うだけです。

メモリページングの基本原理は、CPUのリニアメモリ領域全体を4096バイトのメモリページに分割することである。プログラムがメモリの使用を要求すると、システムはメモリページの単位でメモリを割り当てます。メモリページングは、セグメンテーション機構と同様の方法で実装されているが、セグメンテーションほど洗練されていない。ページングはセグメンテーションの上に実装されているため、システムのメモリを非常に柔軟に制御することができます。また、セグメンテーション機構のメモリ保護機能に加えて、ページング保護機能が追加されています。80X86のプロテクトモードでページングを使用するためには、コントロールレジスタCR0の最上位ビット（ビット31）を設定する必要があります。

このメモリページング方式を用いると、実行中の各プロセス（タスク）は、実際のメモリ容量よりもはるかに大きな連続したアドレス空間を使用することができる。80386では、ページングを利用して直線的なアドレスを比較的小さな物理メモリ空間にマッピングするために、ページディレクトリテーブルとページテーブルを利用している。ページディレクトリエントリは、基本的にはページテーブルエントリと同じフォーマットで、4バイトを占有し、各ページディレクトリテーブルまたはページテーブルには、1024個のページテーブルエントリしか含まれていません。したがって、1つのページディレクトリテーブルまたはページテーブルは、合計1ページ分のメモリを占有します。ページディレクトリエントリとページテーブルエントリの小さな違いは、ページテーブルエントリにはビット

D (Dirty) が書き込まれているのに対し、ページディレクトリエントリにはそれがないことです。

リニアアドレスから物理アドレスへの転送プロセスを図5-9に示します。図中のコントロールレジスタCR3は、物理メモリ内の現在のページディレクトリテーブルのベースアドレスを保持しています（そのため、CR3はページディレクトリベースアドレスレジスタPDBRとも呼ばれています）。32ビットのリニアアドレスは、ページディレクトリテーブルとページテーブルの対応するエントリの位置を特定するための3つの部分に分割され、対応する物理メモリページ内のページ内オフセット位置を指定します。ページテーブルは1024個のエントリを持つことができるため、1つのページテーブルは最大 $1024 * 4KB = 4MB$ のメモリをマッピングすることができ、ページディレクトリテーブルは1024個の2次ページテーブルに対応する最大1024個のエントリを持つため、1つのページディレクトリテーブルは最大マップ $1024 * 4MB = 4GB$ のメモリをマッピングすることができる。つまり、ページディレクトリテーブルは、リニアアドレス空間の全範囲をマッピングすることができるのです。

Linux 0.1xシステムでは、カーネルとすべてのタスクが同じページディレクトリテーブルを共有しているため、プロセッサのリニアアドレス空間と物理アドレス空間のマッピング機能は、いつでも同じです。したがって、カーネルとすべてのタスクが重なって干渉しないようにするために、仮想アドレス空間から線形アドレス空間の異なる位置にマッピングする、つまり異なる線形アドレスを占有する必要があります。

スペースの範囲です。

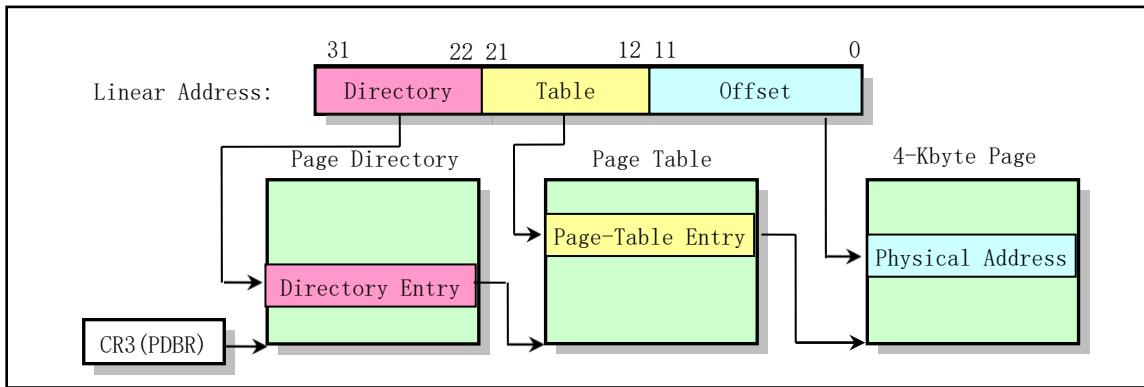


図5-9 リニアアドレス→物理アドレス変換の図

Intel 80386システムでは、CPUは最大4Gのリニアアドレス空間を提供することができます。タスクの仮想アドレスは、まずローカルセグメント記述子によってCPUのリニアアドレス空間全体のアドレスに変換され、ページディレクトリテーブルPDT（プライマリページテーブル）とページテーブルPT（セカンダリページテーブル）を使って実際にマッピングされる必要がある。物理アドレスのページに実際の物理メモリを使用するために、各プロセスのリニアアドレスは、二次記憶ページテーブルによって主記憶領域の異なる物理メモリページに動的にマッピングされます。

Linux 0.12では、1プロセスあたりの利用可能な仮想メモリ空間の最大値が64MBと定義されているため、各プロセスの論理アドレスは、(タスク番号) *64MBを加えることで線形空間上のアドレスに変換できる。ただし、本書のコードコメントでは、このようなプロセス内のアドレスを、混同しないように単に論理アドレスやリニアアドレスと呼ぶことがある。

Linux 0.12カーネルでは、GDTに設定されているセグメントディスクリプタエントリの最大数は256です。そのうち2つは使用されず、2つはシステムのエンティティであり、各プロセスやタスクは2つを使用する。したがって、この時点でシステムは最大 $(256-4)/2=126$ 個のタスクを収容することができ、仮想アドレスの範囲は $((256-4)/2)*64MB$ で約8Gに相当します。しかし、0.12カーネルで手動定義されているタスクの最大数はNR_TASKS=64、各タスクの論理アドレス範囲は64M、リニアアドレス空間における各タスクの開始位置は(タスク番号) *64MBです。したがって、すべてのタスクが使用するリニアアドレス空間は、図5-10に示すように、 $64MB*64=4G$ となります。

図は、システムに4つのタスクがある場合の状況を示しています。カーネルのコードセグメントとデータセグメントは、リニアアドレス空間の先頭の16MB部分にマッピングされており、コードセグメントとデータセグメントの両方が同じ領域にマッピングされており、完全にオーバーラップしています。最初のタスク（タスク0）は、カーネルが"手動"で起動します。コードとデータはカーネルのコードとデータの中に含まれているので、このタスクが使用するリニアアドレス空間はかなり特殊

なものです。タスク0のコードセグメントとデータセグメントの長さは、リニアアドレス0から640KBの範囲であり、コードセグメントとデータセグメントも完全に重なり、カーネルのコードセグメントとデータセグメントと重なっています。実は、Linux 0.12では、すべてのタスクの命令空間I (Instruction) とデータ空間D (Data) は、1つのメモリを使っている。つまり、プロセスのコード部分、データ部分、スタック部分がすべて同じメモリセグメントに入っています。I&Dを分離せずに使用していることになります。

タスク1は、アドレス64MBから始まるリニアなアドレス空間を持ち、長さは640KBしかありません。両者の詳細な対応関係は後述します。タスク2とタスク3は、それぞれ128MBと192MBのリニアアドレスにマッピングされており、その論理アドレス範囲は64MBです。4Gのアドレス空間の範囲は

は、まさに32ビットCPUのリニアアドレス空間の範囲であり、アドレス可能な最大の物理アドレス空間の範囲もあります。また、タスク0とタスク1の論理アドレス範囲を64MBとみなすと、システムとしては タスクの論理アドレス範囲も4GBとなるため、0.12カーネルでは3つのアドレス概念を混同しやすくなります。



図5-10 Linux 0.12のリニアアドレス空間を利用した場合の図

仮想空間内のタスクも線形空間内のタスクの順番に合わせて配置すると、図5-11のようなシステムになります。仮想アドレス空間内のすべてのタスクの模式図ができ、仮想空間の範囲も4GBとなります。なお、仮想空間内のカーネルコードやデータの範囲は考慮していません。また、図では、タスク2とタスク3について、それぞれの論理空間におけるコードセグメントとデータセグメント（データとスタックの内容を含む）の位置も示されています。



図 5-11 Linux 0.12 における仮想空間内のタスクの空間的広がり

また、タスクの論理アドレス空間におけるコード・セクションとデータ・セクションの概念は、CPUのセグメンテーション・メカニズムにおけるコード・セグメントとデータ・セグメントと同じ概念ではないことにも注意が必要です。CPUのセグメンテーションでは、セグメントの概念により、リニアアドレス空間におけるセグメントの目的や、強制的に行われる制約やアクセスを決定します。各セグメントは、4GBのリニアアドレス空間のどこにでも置くことができ、互いに独立していても構いません。また、完全にまたは部分的にオーバーラップすることもできます。タスクのコード部、データ部とは、コンパイラがプログラムをコンパイルする際、およびOSがプログラムをロードする際に指定するプロセスロジック空間内のコード領域、初期化・非初期化データ領域、スタック領域のこ

とを指します。タスク論理アドレス空間のコードセグメントとデータセグメントの構造を図5-12に示します。図中のnrはプロセスまたはタスクの番号、start_codeはリニアアドレス空間におけるプロセスの開始位置である。その他の変数はすべて、プロセス論理空間の値を含みます。



図5-12 論理アドレス空間におけるタスクコードとデータの分布

5.3.7 CPU multitasking and protection

80X86 CPUの保護機構には4つの保護レベルがあり、レベル0が最も優先度が高く、レベル3が最も優先度が低くなっています。OSのLinux 0.12では、CPUの保護レベルを0と3の2つにしています。各タスクには、コード領域とデータ領域があります。この2つの領域はローカルアドレス空間に格納されているため、システム内の他のタスクは不可視（アクセスできない）になっています。カーネルのコードとデータはすべてのタスクで共有されるので、グローバルアドレス空間に格納されます。この構造の模式図を図5-13に示します。図中の同心円は、CPUの保護レベル（保護層）を表しており、ここでは0と3のレベルのみを使用しています。放射状の光線は、システム内のタスクを区別するために使用されます。各放射状の光線は、各タスクの境界を示している。各タスクの仮想アドレス空間のグローバルアドレス領域を除いて、タスク1のアドレスはタスク2の同じアドレスとは独立している。

タスク（プロセス）がシステムコールを実行し、カーネルコードで実行されているとき、そのプロセスをカーネル動作中（または単にカーネルモード）と呼ぶ。この時点では、プロセッサは最高の特権レベル（レベル0）で実行されます。プロセスがカーネルモードの場合、実行されたカーネルコードは現在のプロセスのカーネルスタックを使用し、各プロセスは独自のカーネルスタックを持っています。プロセスがユーザー自身のコードを実行しているとき、そのプロセスはユーザーの実行状態（ユーザーモード）にあると言われます。つまり、プロセッサは現在、最も低い特権レベル（レベル3）のユーザーコードで実行されています。

ユーザープログラムが実行されているときに、割り込みハンドラプロシージャを実行するために突然中断された場合、ユーザープログラムは、プロセスのカーネル状態と象徴的に呼ぶこともできます。なぜなら、割込みハンドラは現在のプロセスのカーネルスタックを使用するからです。これは、カーネルモードのプロセスの状態と多少似ています。プロセスのカーネル状態とユーザーモードについては、後でプロセスの実行状態の項で詳しく説明します。

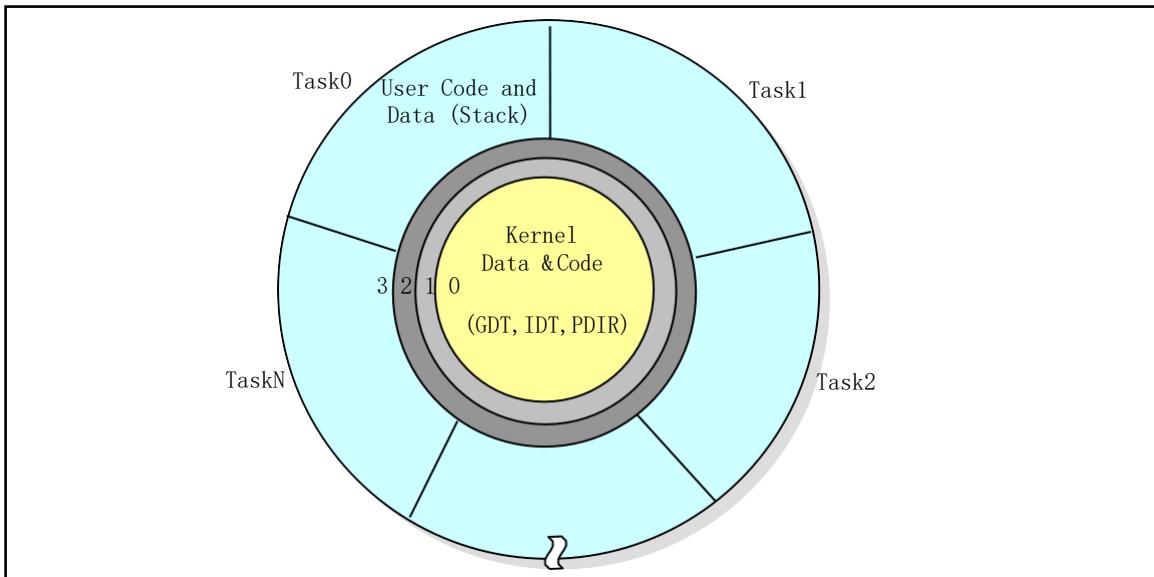


図5-13 マルチタスクの保護システム

5.3.8 Virtual Addresses, Linear Addresses, and Physical Addresses

先に、メモリセグメンテーションとページングの仕組みによる、CPUのメモリ管理方法を説明しました。ここでは、Linux 0.12システムを例に、仮想アドレス空間、リニアアドレス空間、物理アドレス空間における、カーネルのコード・データと各タスクのコード・データの対応関係を説明します。なお、タスク0とタスク1の生成・作成プロセスは特殊なので、別々に説明します。

Kernel code and data address

Linux 0.12では、head.sプログラムの初期化動作において、カーネルコードセグメントとデータセグメントの両方が16MBのセグメントに設定されています。この2つのセグメントの範囲は、リニアアドレス空間内で重複しており、リニアアドレス0からアドレス0xFFFFFFFまで、合計16MBのアドレス範囲となっています。この範囲には、すべてのカーネルコード、カーネルセグメントテーブル(GDT、IDT、TSS)、ページディレクトリテーブルとセカンダリページテーブル、カーネルローカルデータ、カーネル一時スタック(タスク0のユーザースタックとして使用される)が含まれます。そのページディレクトリテーブルと2次ページテーブルは、0～16MBのリニアアドレス空間を1つずつ物理アドレスにマッピングし、4つのディレクトリエントリ、つまり4つの2次ページテーブルを占有するように設定されている。そのため、カーネルのコードやデータのアドレスについては、直接、物理メモリ上のアドレスと考えることができます。このとき、カーネルの仮想アドレス空間、リニアアドレス空間、物理アドレス空間の関係は、図5-14のように表すことができる。

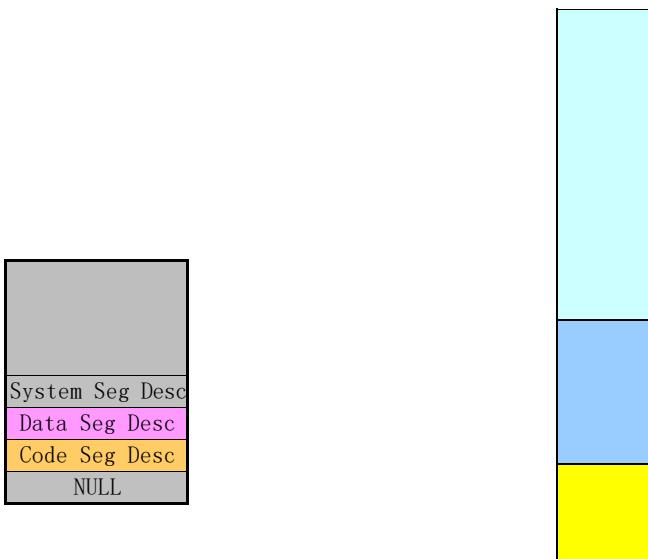


図5-14 3つのアドレス空間におけるカーネルコードとデータセグメント

- ◆ したがって、デフォルトでは、Linux 0.12カーネルは、最大16MBの物理メモリを管理することができます、4096の物理ページ（ページフレーム）、1ページあたり4KBを管理することができます。以上の分析を通して、次のことがわかります。
 - ◆ The kernel code and the data segments region are the same in the linear and physical address space. This setting can greatly simplify the initialization of the kernel.
 - ◆ GDT and IDT are in the kernel data segment, so their linear addresses are also equal to their physical addresses. In the setup.s program initialization operation in real mode, we have set the temporary GDT and IDT, which must be set before entering the protection mode. Since the two tables were in physical memory at about 0x90200, the kernel system module was in the physical memory 0 start position after entering protected mode, and the space at 0x90200 would be used for other purposes (for caching). So after entering protected mode, we need to reset the two tables in the first program head.s that are running. That is, setting GDTR and IDTR to point to the new GDT and IDT, the descriptor also needs to be reloaded. However, since the position of the two tables does not change when the paging mechanism is turned on, there is no need to re-establish or move the table position.
 - ◆ Except for task 0, the physical memory pages used by all other tasks are at least partially different from the pages in the linear address, so the kernel needs to dynamically map them in the main memory area to dynamically create page directory entries and page table entries. . Although the code and data of task 1 are also in the kernel, since it needs to be allocated separately to obtain memory, it also needs its own mapping table entry.

Linux 0.12は、デフォルトで16MBの物理メモリを管理できますが、システムにそのような物理メモリを搭載する必要はありません。マシンに4MB（あるいは2MB）の物理メモリがあれば、Linux 0.12システムを動かすことができます。マシンに4MBの物理メモリしかない場合は、カーネルの

4MB～16MBのアドレス範囲が存在しないアドレスにマッピングされます。しかし、これはシステムの動作を妨げるものではありません。なぜなら、カーネルメモリマネージャは、マシンの物理メモリの正確な量を

初期化時には、CPUのページング機構に、存在しない4MB～16MBへのリニアアドレスページのマッピングをさせません。カーネルのデフォルト設定は、主にシステムの物理メモリの拡張を容易にするためのもので、実際には存在しない物理メモリ領域を使用していません。システムが16MB以上の物理メモリを持っている場合、init/main.cプログラムの初期化で16MB以上のメモリの使用が制限されているため、ここではカーネルは0～16MBのメモリ範囲のみをマッピングします。そのため、16MB以上の物理メモリは使用されません。

もちろん、ここでカーネルにいくつかのページテーブルを追加し、init/main.cプログラムにマイナーな変更を加えることで、この制限を拡張することができます。例えば、システムに32MBの物理メモリがある場合、32MBのリニアアドレス範囲を物理メモリにマッピングするために、カーネルのコードとデータセグメント用に8つのセカンダリページテーブルエントリを作成する必要があります。

The address correspondence of task 0

タスク0は、システムで手動で開始される最初のタスクです。コードとデータのセグメント長は640KBに設定されています。このタスクのコードとデータは、カーネルのコードとデータに直接含まれており、リニアアドレス0から始まる640KBの内容となっています。そのため、カーネルが設定したページディレクトリやページテーブルを直接利用して、ページングアドレス変換を行うことができます。同様に、そのコードとデータのセグメントは、リニアアドレス空間内で重なっています。対応するタスクステータスセグメントTSS0も手動で事前に設定され、タスク0のデータ構造情報の中に位置しています。include/linux/sched.hの156行目から始まるデータを参照してください。TSS0セグメントは、カーネルのsched.cファイルのコード内にあり、長さは104バイトとなっています。詳細は、図5-24の「タスク0構造情報」の項目を参照してください。3つのアドレス空間におけるマッピングの対応を図5-15に示す。

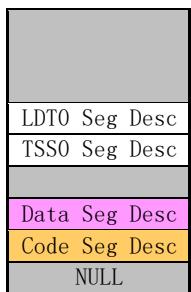


図5-15 3つのアドレス空間におけるタスク0の関係

タスク0はカーネルコードに直接含まれているので、そのために追加のメモリページを割り当てる必要はありません。また、その動作に必要なカーネルモードのスタックとユーザーモードのスタック空

間もカーネルコードの領域にあり、カーネルページの初期化 (head.s) 以降、ページテーブルエントリのこれらのカーネルページのプロパティは0b111に設定されています。つまり、対応するページユーザーは読み書きができる、存在することができる。したがって、ユーザースタック user_stack[] 空間がカーネル空間にあるにもかかわらず、タスク0はそれに対して読み書きすることができます。

Task 1 address correspondence

タスク0と同様、タスク1も特別なタスクです。そのコードもカーネルコードエリアにあります。タスク0とは異なり、リニアアドレス空間では、fork()を使ってタスク（initプロセス）が生成されると、システムはタスク1の2次ページテーブルを格納するためのメモリページをメインメモリ領域に格納します。親プロセス（タスク0）のページディレクトリと2次ページテーブルのエントリはコピーされます。したがって、タスク1は独自のページディレクトリとページテーブルエントリを持ち、タスク1の線形空間の範囲である64MB～128MB（実際には64MB～64MB+640KB）を物理アドレス0～640KBにマッピングする。このとき、タスク1の長さも640KBで、コードセグメントとデータセグメントが重なり、1つのページディレクトリエントリと1つのセカンダリページテーブルしか占めていない。また、システムはタスク1に対して、タスクデータ構造とカーネルスタック空間を格納するためのメモリページを要求します。タスクデータ構造（プロセス制御ブロックPCBとも呼ばれる）情報には、タスク1のTSSセグメント構造情報が含まれています。図5-16を参照してください。



図5-16 3つのアドレス空間におけるタスク1の関係

タスク1のユーザモードスタック空間は、カーネルコード・データ領域（リニアアドレス0～640KB）にあるタスク0のユーザモードスタック空間user_stack[]を直接共有することになります（kernel/sched.cの82～87行目を参照）。そのため、タスク1のためにコピーされるスタックに無駄なデータが含まれないように、タスク1が実際に使用されるまで、このスタックは「クリーン」である必要があります。タスク1の生成開始時には、タスク0のユーザモードスタックuser_stack[]がタスク1と共有されていますが、タスク1の実行開始時には、user_stack[]にマッピングされているページテーブルエントリは、読み取り専用に設定されています。これにより、タスク1がスタック操作を行うと書き込みページ例外が発生するため、カーネルは主記憶領域のページをユーザースタック空間として割り当てます。

Address correspondence of other tasks

タスク2から作成された他のタスクについては、最終的な親プロセスはすべてinit（タスク1）

ロセスです。Linux 0.12システムには、64個のプロセスがあることがすでにわかっています。以下では、タスク2を例にして、他のタスクによるアドレス空間の利用を説明します。

タスク2からは、タスク番号をnrとすると、リニアアドレス空間におけるタスクnrの開始位置は、 $nr * 64MB$ に設定されます。例えば、タスク2の開始位置 = $nr * 64MB = 2 * 64MB =$

128MBです。タスクコードセグメントとデータセグメントの最大長は64MBに設定されているため、タスク2は128MBから192MBまでのリニアアドレス空間を占有し、合計で $64\text{MB}/4\text{MB} = 16$ ページ分のディレクトリエントリを占有します。仮想空間内のタスクコードセグメントとデータセグメントは、同じ範囲のリニアアドレス空間にマッピングされているので、こちらも完全に重なっています。図5-17は、3つのアドレス空間におけるタスク2のコードセグメントとデータセグメントの対応関係を示しています。

タスク2が作成されると、その中でexecve()関数が実行され、シェルプログラムが実行されます。タスク1を複製してタスク2を作成したばかりのカーネルは、128MB--128MB+640KBという直線的なアドレス空間を占有していることに加え、3つのアドレス空間におけるタスク2のコードとデータの関係はタスク1と同様である。タスク2のコード (init()) がexecve()システムコールを呼び出してシェルプログラムのロードと実行を開始すると、システムコールはタスク1からコピーしたページディレクトリとページテーブルのエントリと対応するメモリページを解放する。そして、新しい実行シェルのために、関連するページ・ディレクトリとページ・テーブル・エントリを再作成します。図5-17は、タスク2がシェルプログラムの実行を開始したときの状況、つまり、タスク2のコードとデータがシェルプログラムのコードセグメントとデータセグメントによって元々コピーされていた場合を示しています。この図では、物理メモリの1ページ分がマッピングされている状況を示しています。ここで注意していただきたいのは、execve()関数を実行する際、システムはタスク2のためにリニアアドレス空間に64MBの空間範囲を割り当てていますが、カーネルはすぐにはタスク2のために物理メモリページを割り当ててマッピングしません。メモリマネージャは、タスク2の実行開始時に欠落ページフォールトによる例外が発生した場合にのみ、主記憶領域のリニアアドレス空間に物理メモリのページを割り当ててマッピングします。このように物理メモリのページを割り当ててマッピングする方法をロードオンデマンドといいます。メモリ管理」の章の関連説明を参照してください。



図5-17 他のタスクアドレス空間での対応

Linuxカーネルのバージョン0.99以降、メモリ空間の使い方が変わりました。ページディレクトリテーブルを独立して使用することで、各プロセスは4Gアドレス空間の全範囲を楽しむことができます。を理解することができれば

5.3.9 本節で説明したメモリ管理の考え方を理解すれば、現在使われているLinux 2.xのカーネルで採用されているメモリ管理の原理がすぐに理解できます。紙面の都合上、ここでは説明を省略します。

5.3.10 User application for memory dynamic allocation

ユーザープログラムがCライブラリのメモリ割り当て関数malloc()を使ってメモリを申請する場合、これらの動的アプリケーションのメモリ容量やサイズは、上位のCライブラリ関数malloc()が管理し、カーネル自身は介入しません。なぜなら、カーネルは、CPUの4Gリニアアドレス空間の中で、各プロセス（カーネルコードでメモリに常駐しているタスク0と1を除く）に64MBの領域を割り当っているからです。したがって、タスクやプロセスの実行範囲が64MBの範囲内であれば、カーネルも自動的に物理メモリページを割り当て、メモリページフォールト管理機構によって対応するページに対する操作をマッピングします。

しかし、カーネルは、プロセスが使用するコードとデータの空間のために、現在の位置の変数brkを保持しています。この変数值は、各プロセスのデータ構造に格納されています。これは、プロセス・アドレス空間におけるプロセス・コードおよびデータ（動的に割り当てられたデータ空間を含む）の終了位置を示す。malloc()関数は、プログラムにメモリを割り当てる際に、システムコールbrk()によって、プログラムが要求した空間の長さをカーネルに通知します。カーネルコードは、malloc()から提供された情報に基づいて、brkの値を更新することができます。しかし、この時点では、新たに要求された空間のための物理メモリページはマッピングされていません。プログラムが対応する物理ページを持たないアドレスを指定した場合にのみ、カーネルは該当する物理メモリページに対してマッピング操作を行います。

あるデータがプロセスコードによってアドレス指定されているページが存在せず、そのページの位置がプロセスヒープスコープに属している場合、つまり、実行ファイル・イメージファイルに対応するメモリ範囲に属していない場合、CPUはページフォルト例外を発生させる。そして、例外ハンドラで指定されたページの物理メモリページを割り当ててマッピングします。アプリケーションのメモリサイズと対応する物理ページの具体的な位置については、Cライブラリのメモリ割り当て関数malloc()が管理を行う。カーネルは物理メモリをページ単位で割り当て、マッピングする。この関数は、ユーザープログラムが何バイトのメモリを使用しているかを具体的に記録します。残りの容量は、プログラムがメモリを再申請したときに使えるように確保される。

ユーザープログラムが関数free()を使って要求されたメモリブロックを動的に解放すると、Cライブラリのメモリ管理関数は、プログラムが再度メモリを要求した場合に備えて、解放されたメモリブロックをフリーとマークします。カーネルがこのプロセスに割り当てた物理ページは、このプロセス中は解放されません。プロセスが終了して初めて、カーネルは、プロセスのアドレス空間に割り当てられマッピングされたすべての物理メモリページを完全に取り戻します。

ライブラリ関数malloc()とfree()の具体的な実装コードは、カーネルライブラリ内のlib/malloc.cプログラムにあります。

5.4 Interrupt mechanism

5.4.1 ここでは、割り込み機構の基本原理と、それに関連するプログラマブルコントローラのハードウェアロジック、およびLinuxシステムでの割り込みの使用方法について説明します。なお、プログラマブルコントローラの具体的なプログラミング方法については、次章のsetup.sプログラム以降の記述を参照してください。

5.4.2 Principle of Interrupt Operation

マイクロコンピュータシステムには、通常、入力デバイスと出力デバイスがあります。プロセッサが提供する一つの方法は

これらのデバイスにサービスを提供するには、ポーリングを使用します。この方法では、プロセッサがシステム内の各デバイスに順次問い合わせを行い、サービスが必要かどうかを「照会」します。この方法は、ソフトウェアのプログラミングが簡単なのが利点ですが、プロセッサのリソースを消費し、システムのパフォーマンスに影響を与えるのが欠点です。

デバイスにサービスを提供するもう一つの方法は、デバイスがサービスを必要とするときに、プロセッサ自身に要求を出すことです。また、プロセッサは、デバイスから要求された場合にのみ、デバイスにサービスを提供する。デバイスがプロセッサにサービス要求を行うと、プロセッサは、現在の命令が実行されるとすぐにデバイスの要求に応答し、デバイスの関連するサービスプログラムを実行する。サービスプログラムが実行されると、プロセッサは先ほど中断されたプログラムを続けて実行します。このような処理を「割り込み方式」といい、デバイスがプロセッサに送るサービス要求を「IRQ - Interrupt Request」といいます。その要求に応じてプロセッサが実行するデバイス関連のプログラムを「割り込みサービスルーチン」または「ISR」と呼びます。

PIC (Programmable Interrupt Controller) は、マイコンシステムにおいて、デバイスの割り込み要求を管理するアドミニストレーターです。PICは、デバイスに接続された割り込み要求端子を介して、デバイスからターミナルサービスリクエスト信号を受け取ります。デバイスが割り込み要求IRQ信号をアクティブにすると、PICはすぐにそれを検出します。複数のデバイスから同時に割り込みサービス要求を受信した場合、PICはそれらを優先し、最も優先度の高い割り込み要求を選択して処理します。また、プロセッサがあるデバイスの割り込みサービスルーチンを実行中の場合、PICは選択された割り込み要求と処理中の割り込み要求の優先度を比較し、その比較に基づいてプロセッサに割り込みを発行するかどうかを判断する必要があります。PICがプロセッサのINT端子（図5-18のINTR端子）に割り込みを発行すると、プロセッサはその時点で行っていた処理を直ちに停止し、どの割り込みサービスリクエストを実行するかをPICに問い合わせます。PICは、割り込み要求に対応する割り込み番号をデータバスに送信することで、どの割り込みサービス処理を実行するかをプロセッサに通知します。プロセッサは、読み込んだ割込み番号に応じて、割込みベクタテーブル（32ビットプロテクトモードでは割込みディスクリプターテーブルIDT）を問い合わせることにより、当該デバイスの割込みベクタ（すなわち、割込みサービスルーチンのアドレス）を取得し、割込みサービスルーチンの実行を開始する。割込みサービスルーチンの実行が終了すると、プロセッサは割込み信号によって中断されたプログラムの実行を継続します。

5.4.3 これまで説明してきたのは、入出力デバイスの割込みサービス処理です。しかし、割り込み方式は必ずしもハードウェアに依存するものではなく、ソフトウェアでも利用することができます。INT命令を使用し、そのオペランドで割込み番号を示すことで、プロセッサを実行して対応する割込み処理を行うことができます。PC/ATシリーズのマイクロコンピュータは256個の割り込みをサポートしてい

ますが、そのほとんどがソフトウェア割り込みや例外に使用されています。例外とは、プロセッサが処理中にエラーを検出して発生する割り込みのことです。本機では、以下に挙げる割り込みのうち一部のみが使用されています。

5.4.4 Interrupt subsystem of 80X86 PC

8259Aプログラマブル割込みコントローラチップは、80X86で構成されるマイクロコンピュータシステムに使用されます。各8259Aチップは8つの割込みソースを管理できます。マルチチップカスケードにより、8259Aは最大64個の割込みベクタを管理するシステムを構成することができます。PC/ATシリーズ互換機では、2つの8259Aチップで15レベルの割り込みベクタを管理しています。カスケードの模式図を図5-18に示します。スレーブチップのINT端子は、マスターチップのIR2端子に接続されています。つまり、8259Aスレーブチップが送る割り込み信号は、8259AマスターちップのIRQ2入力信号となります。マスターの8259Aチップのポートベースアドレスは0x20、スレーブのチップは0xA0です。IRQ9端子の機能は、PC/XTのIRQ2と同じです。つまり、PC/AT機では、IRQ2を使用しているデバイスのIRQ2端子をPICのIRQ9端子にリダイレクトするハードウェア回路を使用し、BIOSのソフトウェアを使用してIRQ9に割り込みをかけているのです。Int 71はIRQ2の割り込みにリダイレクトする int 0x0A

割り込みハンドラプロシージャです。これにより、IRQ2を使用するPC/XT搭載の8ビットカードが、PC/AT機の下で正しく機能するようになりました。PCシリーズの下位互換性を実現した。

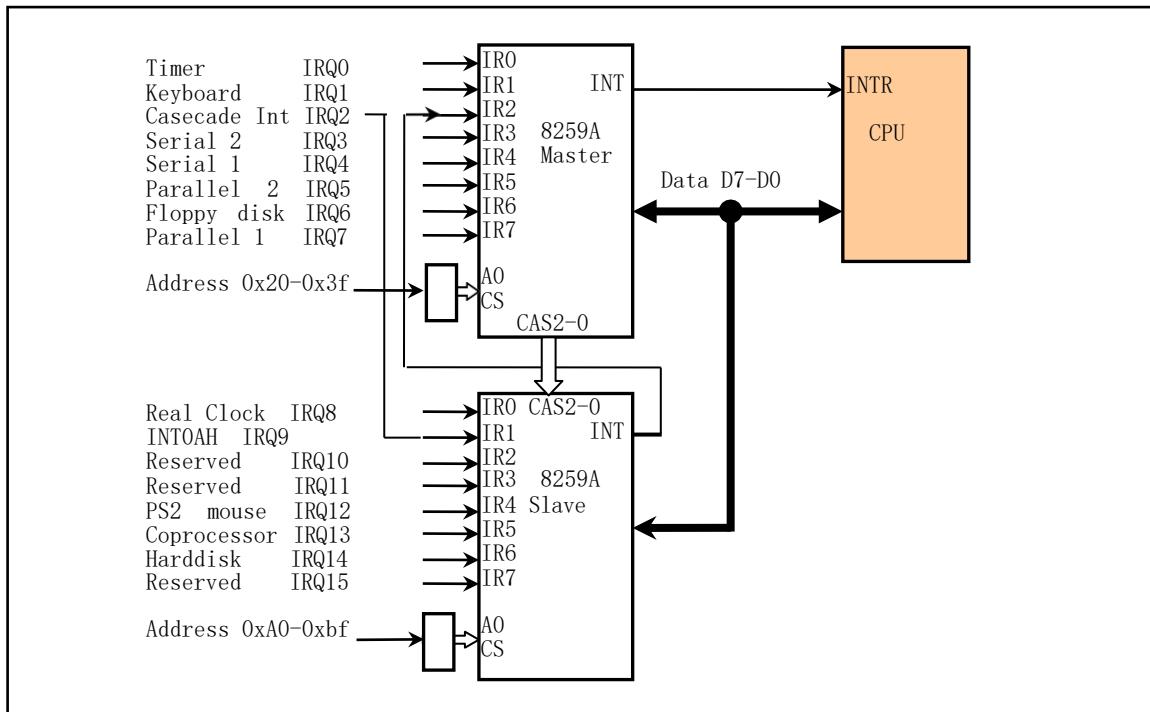


図5-18 PC/ATマイコンカスケード8259制御システム

5.4.5 バスコントローラの制御により、8259Aチップは、プログラミング状態と動作状態になります。プログラミング状態とは、CPUがIN命令やOUT命令を用いて8259Aチップを初期化している状態です。初期化のプログラミングが完了すると、チップは動作状態に入ります。この時、チップは外部デバイスから提案された割り込み要求(IRQ0～IRQ15)にいつでも応答することができ、また、システムは操作コマンドワードを使用して割り込み処理モードをいつでも変更することができます。割り込みアビトレーション選択機構により、チップは現在最も優先度の高い割り込み要求を割り込みサービスオブジェクトとして選択し、CPU端子INTによりCPUに割り込み要求を通知します。CPUが応答した後、チップはデータバスD7-D0から現在のサービスオブジェクトのプログラムされた割り込み番号を送信し、CPUは対応する割り込みベクタ値を取得し、割り込みサービスルーチンを実行します。

5.4.6 Interrupt Vector Table

前節では、CPUは割込みサービスルーチンのエントリアドレス値に対応する割込み番号に基づいて割込みベクタ値をフェッチすることを示しました。したがって、CPUが割込み番号から対応する割込みベクターを見つけるためには、メモリ上にルックアップテーブル、すなわち割込みベクターテーブルを作成する必要があります（32ビット保護モードでは、このテーブルを割込みディスクリプターテーブルIDTと呼びます、後述します）。

80X86マイコンは256本の割り込みをサポートしており、割り込みごとに割り込みサービスルーチンが必要となります。80X86のリアルモードモードでは、各割り込みベクターは4バイトで構成されています。この4バイトは、割込みサービスルーチンのセグメント値とセグメント内オフセット値を示

します。このため、ベクターテーブル全体の長さは $4 \times 256 = 1024$ バイトとなります。80X86マイクロコンピュータの起動時には、ROM内のプログラムが

BIOSは、物理メモリの開始アドレス0x0000:0x0000に割り込みベクタテーブルを初期化して設定し、各割り込みに対するデフォルトの割り込みサービスルーチンをBIOSに与えます。割り込みベクタテーブルのベクタは割り込み番号の順に並んでいるので、割り込み番号Nが与えられた場合、対応する割り込みベクタのメモリ上の位置は0x0000 : N * 4、つまり対応する割り込みサービスプログラムのエントリアドレスは物理メモリ0x0000 : N * 4の位置に格納されます。

BIOSが初期化操作を行うと、2つの8259Aチップがサポートする16個のハードウェア割り込みベクターと、BIOSが提供する割り込み呼び出し機能に割り込み番号0x10-0x1Fを設定します。実際に使用されない割り込みについては、ベクターに一時的なダミーの割り込みサービスルーチンのアドレスが入力されます。その後、システムがOSを起動する際に、実際の必要性に応じていくつかの割り込みベクターの値が変更されます。例えば、DOSオペレーティングシステムの場合は、割り込み0x20-0x2Fの割り込みベクター値をリセットして修正します。Linuxシステムの場合は、最初にカーネルをロードしたときにBIOSが提供するディスプレイとディスクリードの割り込みに加えて、新しい割り込みベクタテーブルが作成されます。つまり、setup.sプログラムで8259Aチップを再初期化し、head.sプログラムで割り込みベクタテーブル（割り込みディスクリプターテーブル）を再構築しています。そのため、カーネルが正常に動作した後のLinuxでは、BIOSの割り込みベクターテーブルを完全に放棄します。

5.4.7 インテルCPUが32ビットプロテクトモードで動作している場合、割り込みや例外を管理するためにInterrupt Descriptor Table (IDT)を使用する必要があります。 IDTは、インテル8086～80186のCPUで使用されていた割り込みベクターテーブルをそのまま置き換えたものです。IDTの役割は、割り込みベクターテーブルと似ていますが、各割り込み記述子エントリには、割り込みサービスルーチンのアドレスに加えて、特権レベルと記述子クラスの情報が含まれています。Linux OSは80X86プロテクトモードで動作するため、割り込みディスクリプターテーブルを使用して、各割り込みの「ベクター」情報を設定・保存します。

5.4.8 Linux kernel interrupt handling

Linuxカーネルの場合、割り込み信号は通常、ハードウェア割り込みとソフトウェア割り込み（または例外）の2つのカテゴリーに分けられます。各割込みは、割込み番号と呼ばれる0～255の数字で識別されます。割り込みINT0--INT31（0x00--0x1f）については、表5-1に示すように、各割り込みの機能がインテル社によって固定または予約されています。上の項からもわかるように、BIOSで設定された割り込み番号の範囲はそれと相反するものです。

| 表 5-1 イ ン テ | Name | Type | Error Code | Signal | Source |
|-------------------------|------|------|---------------|--------|--------|
| | | | | | |

| ル 社 が 予 約 し た 例 外 と イ ン タ ラ プ ト Vector No | | | | | |
|--|----------------------|------------------|--------|---------|---|
| 0 | Devide error | Fault (Error) | No | SIGFPE | DIV and IDIV instructions. |
| 1 | Debug | Fault/ Trap | No | SIGTRAP | Any code or data reference or the INT instruction. |
| 2 | nmi | Interrupt | No | | Non maskable external interrupt. |
| 3 | Breakpoint | Trap | No | SIGTRAP | INT 3 instruction. |
| 4 | Overflow | Trap | No | SIGSEGV | INTO instruction. |
| 5 | Bounds check | Fault | No | SIGSEGV | BOUND instruction. |
| 6 | Invalid Opcode | Fault | No | SIGILL | UD2 instruction or reserved opcode. |
| 7 | Device not available | Fault | No | SIGSEGV | Floating-point or WAIT/FWAIT instruction. |
| 8 | Double fault | Abort | Yes(0) | SIGSEGV | Any instruction that can generate an exception, NMI, or an INTR. |

| | | | | | |
|--------|-------------------------|-----------|--------|---------|---|
| 9 | Coprocessor seg overrun | Abort | No | SIGFPE | Floating-point instruction. |
| 10 | Invalid TSS | Fault | Yes | SIGSEGV | Task switch or TSS access. |
| 11 | Segment not present | Fault | Yes | SIGBUS | Loading segment registers or accessing system segments. |
| 12 | Stack segment | Fault | Yes | SIGBUS | Stack operations and SS register loads. |
| 13 | General protection | Fault | Yes | SIGSEGV | Any memory reference and other protection checks. |
| 14 | Page fault | Fault | Yes | SIGSEGV | Any memory reference. |
| 15 | Intel reserved | | No | | |
| 16 | Coprocessor error | Fault | No | SIGFPE | Floating-point or WAIT/FWAIT |
| 17 | Alignment check | Fault | Yes(0) | | Any data reference in memory. |
| 20-31 | Intel reserved. | | | | |
| 32-255 | User Defined interrupts | Interrupt | | | External interrupt or INT n instruction. |

これらの割り込みはソフト割り込みですが、インテルでは例外と呼んでいます。なぜなら、これらの割り込みは、CPUが命令を実行する際に検出された異常な状態によって引き起こされるからです。通常、フォールトとトラップの2つに分けられます。割り込みINT32--INT255 (0x20--0xff) は、ユーザーが設定・定義することができます。すべての割り込みの分類と、実行後のCPUの動作方法を表5-2に示します。

表5-2 割り込みの分類とCPUの処理方法

| Interrupt | Name | CPU Check Method | Processing Method |
|-----------|-------------|--------------------------------|--|
| ハードウェア | Maskable | CPU pin INTR | Clear the IF maskable interrupt flag of EFLAGS. |
| | Nonmaskable | CPU pin NMI | Non-Maskable Interrupts. |
| | Fault | Detected before error occurred | CPU re-executes the instruction that caused the error. |
| | Trap | Detected after error occurred | CPU continues to execute the following instruction. |
| ソフトウェア | Abort | Detected after error occurred | Programs that caused this error should be terminated. |

Linuxシステムでは、INT32--INT47 (0x20--0x2f) が、8259A割り込み制御チップ（表5-3参照）が発行するハードウェア割り込み要求信号IRQ0--IRQ15に対応し、ユーザープログラムが発行するソフトウェア割り込みをINT128 (0x80) に設定することを、システムコール（System Call）割り込みと呼びます。システムコール割り込みは、オペレーティング・システムのリソースを使用するユーザー・プログラムの唯一のインターフェースです。

| 表5-3 Linuxシステムの割り込み要求に対する割り込み番号の一覧 | Interrupt No. | Purpose |
|------------------------------------|---------------|---------|
| Interrupt Request No. | | |

| | | |
|------|-----------|---|
| IRQ0 | 0x20 (32) | 100HZ clock interrupt signal from 8253 chip |
| IRQ1 | 0x21 (33) | Keyboard interrupt |
| IRQ2 | 0x22 (34) | Cascade to slave chip |
| IRQ3 | 0x23 (35) | Serial port 2 |
| IRQ4 | 0x24 (36) | Serial port 1 |
| IRQ5 | 0x25 (37) | Parallel port 2 |
| IRQ6 | 0x26 (38) | Floppy disk drive |
| IRQ7 | 0x27 (39) | Parallel port 1 |

| | | |
|-------|-----------|------------------------------|
| IRQ8 | 0x28 (40) | Real clock |
| IRQ9 | 0x29 (41) | Reserved |
| IRQ10 | 0x2a (42) | Reserved |
| IRQ11 | 0x2b (43) | Reserved (Network interface) |
| IRQ12 | 0x2c (44) | PS/2 mouse |
| IRQ13 | 0x2d (45) | Coprocessor |
| IRQ14 | 0x2e (46) | Harddisk |
| IRQ15 | 0x2f (47) | Reserved |

システムの初期化時、カーネルはまずダミーの割込みベクター（割込み記述子）を使用して、割込み記述子テーブル（IDT）の256個の記述子をすべてデフォルト設定にします。このダミーの割込みベクターは、デフォルトの「割込みなし」のハンドラプロシージャを指します。割り込みが発生し、割り込みベクターがリセットされていない場合、「Unknown interrupt」というメッセージが表示されます。システムで使用する必要がある一部の割り込みについては、カーネルは初期化を続ける過程でこれらの割り込みの割り込み記述子項目を再編集し、対応する実際のハンドラープロシージャを指すようにします。通常、例外割り込み処理（INT0～INT31）はtraps.cの初期化関数でリセットされ、システムコール割り込みint128はスケジューラの初期化関数でリセットされます。

5.4.9 また、Linuxカーネルは、割り込みディスクリプターテーブルIDTの設定時に、割り込みゲートとトラップゲートの両方のディスクリプターを使用します。両者の違いは、フラグレジスタEFLAGS内の割込みイネーブルフラグIFへの影響です。割り込みゲート記述子によって実行された割り込みは、IFフラグをリセットするので、他の割り込みが現在の割り込み処理を妨害することを防ぐことができる。続く割込み終了命令IRETは、スタックからIFフラグの元の値を復元します。トラップゲートを介して実行される割込みは、IFフラグに影響を与えません。

5.4.10 Interrupt Flag of Flag Register

クリティカル・コード・エリアの競合と混乱を避けるために、Linux 0.12カーネル・コードの多くの場所でCLI命令とSTI命令が使用されています。CLI命令は、CPUフラグレジスタの割り込みフラグIFをリセットするために使用され、CLI命令を実行した後、システムが外部からの割り込みに反応しないようにします。STI命令は、CPUが外部機器からの割り込みを認識して応答できるように、フラグレジスタの割り込みフラグを設定するために使用されます。

競合状態を引き起こす可能性のあるコード領域に入る場合、カーネルはCLI命令を使用して外部割込みへの応答をオフにし、コンテンツコード領域を実行する際にSTI命令を実行してCPUの外部割込みへの応答を再度許可するようになります。例えば、ファイルスーパー・ロックのロックフラグの変更や、タスクの入退出待ちキューの操作を行う場合、まずCLI命令でCPUの外部割込みへの応答を禁止し、操作完了後にSTI命令で外部割込みへの応答を有効にする必要があります。CLI,STI命令のペアを使用しない場合、つまりCLIを使用して外部割込みへの応答を無効にせずにファイルのスーパー・

ロックを修正する必要がある場合、修正前にスーパーブロックロックフラグが設定されていないと判断され、このフラグを設定したい場合があります。ちょうどこのタイミングでシステムクロックの割り込みが発生し、他のタスクの実行に切り替わり、たまたま他のタスクもスーパーブロックを修正する必要があった場合、この他のタスクがまずスーパーブロックのロックフラグを設定し、スーパーブロックを修正します。システムが元のタスクに切り替わると、この時、タスクはロックフラグを判定せず、設定されたスーパーブロックのロックフラグを実行し続けるため、2つのタスクが同時にクリティカルコード領域に対して複数の操作を行い、スーパーブロックのデータを引き起こすことになります。不整合は、深刻な場合、カーネルシステムのクラッシュにつながる可能性があります。

5.5 Linux system calls

5.5.1 System Call Interface

システムコール（通称：シスコール）は、図5-4に示すように、Linuxカーネルが上流のアプリケーションと通信できる唯一のインターフェースである。割り込み機構の説明から、ユーザプログラムは、割り込みint 0x80を直接または（ライブラリ関数を介して）間接的に呼び出し、EAXレジスタにシステムコール関数番号を指定することで、システムハードウェアリソースを含むカーネルリソースを利用することができます。しかし、通常、アプリケーションは、図5-19に示すように、標準的なインターフェース定義を持つCライブラリの関数を使って間接的にカーネルのシステムコールを使用します。

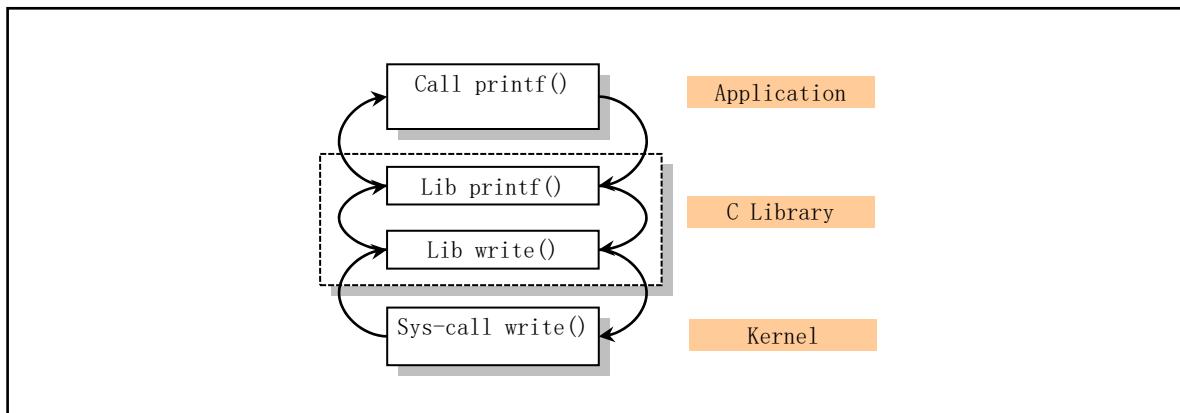


図 5-19 ユーザプログラム、ライブラリ関数、カーネルシステムコールの関係

通常、システムコールは関数形式で行われるため、1つ以上のパラメータを取ることができます。システムコールの実行結果は、戻り値で表されます。通常、負の値はエラーを、0は成功を表します。エラーの場合は、間違った型のコードがグローバル変数errnoに格納されます。ライブラリ関数の perror()を呼び出すことで、エラーコードに対応するエラー文字列情報を出力することができます。Linuxカーネルでは、各システムコールに固有のシステムコール関数番号が設定されています。カーネル0.12には、合計87個のシステムコール機能（0～86）があります。これらの機能番号は、include/unistd.hというファイルの62行目の先頭で定義されています。例えば、writeシステムコールの機能番号は4で、シンボルである

NR_writeです。これらのシステムコール関数番号は、実際にはinclude/linux/sys.hで定義されているシステムコールハンドラポインタの配列テーブルsys_call_table[]の項目のインデックスに対応しています。つまり、write()システムコールのハンドラポインタは配列の項目4にあります。

これらのシステムコールシンボルを自分のプログラムで直接使用したい場合は、以下のように、ファイル「<unistd.h>」をインクルードする前にシンボル「LIBRARY」を定義する必要があります。

```
#define LIBRARY
#include <unistd.h>
```

また、`sys_call_table[]`から、カーネル内のすべてのシステムコールハンドラの名前は基本的にシンボル「`sys_`」で始まっていることがわかります。例えば、カーネルのソースコードにおけるシステムコール`read()`の実装関数は`sys_read()`です。

5.5.2 System Call Processing

アプリケーションがライブラリ関数を介してカーネルに割込みINT 0x80を発行すると、システムコールが開始されます。システムコール番号はレジスタEAXに格納され、渡されたパラメータはレジスタEBX、ECX、EDXに順番に格納することができる。したがって、Linux 0.12カーネルのユーザープログラムは、最大3つのパラメータをカーネルに直接渡すことができる。もちろん、パラメータを受け取らないことも可能です。システムコール割り込みINT 0x80を処理する処理は、プログラム kernel/system_call.sのsystem_callです。

システムコールの実行を容易にするために、カーネルのソースコードにはマクロ関数である

include/unistd.hファイル内の_syscalln()（150～200行）で、nは運ばれるパラメータの数を表し、それぞれ0～3となります。したがって、最大3個のパラメータを直接渡すことができます。大きなチャクデータをカーネルに渡す必要がある場合は、チャクデータのポインタを渡すことができます。例えば、read()システムコールの場合、その定義は次の通りです。

```
int read(int fd, char *buf, int n);
```

対応するシステムコールをユーザープログラムで直接実行すると、そのシステムコールのマクロは

```
#define LIBRARY
#include <unistd.h>
_syscall3(int, read, int, fd, char *, buf, int, n)
```

つまり、Cライブラリを介さずに、ユーザープログラムで直接上記の_syscall3()を使って、システムコールread()を実行することができるのです。実際、C関数ライブラリでの関数呼び出しの形式は、ここで与えられたものと同じです。

include/unistd.hで指定された各システムコールマクロには、 $2+2*n$ 個のパラメータがあります。最初のパラメータは、システムコールの戻り値のタイプに対応し、2番目のパラメータは、システムコールの名前であり、その後にシステムコールによって運ばれるパラメータのタイプと名前が続く。このマクロを、以下のように、インラインのアセンブリ文を含むC関数に拡張します。

```
int read(int fd, char *buf, int n)
{
ロングレスって
    asm volatile
    ("int $0x80")
    : "=a" ( res )
    : 「0」( NR_read ), "b" ((long)(fd)), "c" ((long)(buf)), "d" ((long)(n)); if ( res>=0 )
        return int res; errno=-res;
    1を返す。
}
```

このマクロは、読み取りシステムコールの具体的な実装として展開されていることがわかります。埋め込みアセンブリステートメントを使用して、Linuxシステム割り込みコール0x80を、関数番号NR_read(3)です。この割り込みコールは、実際に読み込んだバイト数をEAX(res)レジスタに返します。返された値が0より小さい場合は、読み出し動作のエラーが発生したことを意味しますので、エラー番号を反転させて

グローバル変数errnoに格納され、-1の値が呼び出したプログラムに返されます。

システムコールが3つ以上のパラメータを必要とする場合、カーネルは通常、パラメータをパラメータバッファブロックとして使用し、バッファブロックのポインタをパラメータとしてカーネルに渡します。そのため、3つ以上のパラメータを持つシステムコールでは、1つの引数を持つマクロ`_syscall1()`を使用するだけで、最初の引数のポインタをカーネルに渡すことができます。例えば、`select()`関数のシステムコールには5つの引数がありますが、その第一引数のポインタを渡すだけです。fs/select.cプログラムの説明をご覧ください。

カーネル内でカーネルコールハンドラkernel/sys_call.sに入ると、`system_call`コードは、まずEAXのシステムコール関数番号が有効なシステムコール番号の範囲内にあるかどうかをチェックします。そして、`sys_call_table[]`関数ポインタテーブルの呼び出しに従って、対応するシステムコールハンドラを実行します。

```
call _sys_call_table(%eax, 4) // kernel/sys_call.s 第 99 行。
```

このアセンブリステートメントのオペランドの意味は、`_sys_call_table + %eax * 4`の関数を間接的に呼び出すことです。`sys_call_table[]`のポインタはそれぞれ4バイトなので、システムコール関数の番号を乗じる必要があります。

5.5.3 4. そして、その結果得られた値を使って、呼び出されたハンドラのアドレスをテーブルから取得します。

5.5.4 Parameter Passing Method of Linux System Call

Linuxのユーザプロセスがシステム割り込みコールプロシージャにパラメータを渡す場合、LinuxシステムではレジスタEBX、ECX、EDXなどの汎用的なレジスタ転送手段を使用します。このレジスタパッシング・パラメータを使用する方法の大きな利点は、システム割り込みサービスルーチンに入ってレジスタ値を保存する際に、パラメータを渡したレジスタも自動的にカーネル状態のスタックに置かれることです。そのため、パラメータを渡したレジスタを特別に処理する必要はありません。これが、当時のリナス氏が知っていた最もシンプルで高速なパラメータ転送方法です。また、インテルのCPUが提供するシステムコールゲートを利用したパラメータ転送方法もあり、これは渡されたパラメータをプロセスのユーザー状態スタックとカーネル状態スタックに自動的にコピーするものである。しかし、この方法で使われている方法はもっと複雑だ。

また、各システムコールハンドラでは、渡されたパラメータを検証し、すべてのパラメータが合法で有効であることを確認する必要があります。特に、ユーザーが提供したポインターは、ポインターが指すメモリ領域の範囲が有効であり、適切な読み取り権限と書き込み権限を持っていることを厳密に確認する必要があります。

5.6 System time and timing

5.6.1 System time

オペレーティングシステムが自動的に正確な現在の時刻と日付の情報を提供できるようにするために、PC/ATマイクロコンピュータシステムでは、バッテリー駆動のリアルタイムRT（Real Time）回路のサポートが提供されています。通常、この回路部分は、システム情報を保持する少量のCMOS RAMと一緒に1チップに集積されているので、この部分はRT/CMOS RAM回路と呼ばれる。モトローラ社のMC146818チップは、PC/ATマイクロコンピュータまたはその互換機に使用されています。

初期化時、Linux 0.12カーネルは、チップに格納されている現在の時刻と日付の情報を読み取り、1970年1月1日0:00からの秒単位の現在時刻に変換します。これをUNIXカレンダータイムと呼んでいます。この時刻は、システムが動作を開始する暦上の時刻を決定し、グローバル変数startup_timeに保存され、すべてのカーネルコードが使用できるようになります。ユーザープログラムは、システムコールのtime()を使って、この時刻の値を読み取ります。

startup_time、一方、スーパーユーザーはstime()を呼び出してシステム時間の値を変更することができます。

また、プログラムは、システムスタートからカウントされた以下のシステムチック値のjiffiesによって、現在の実行時間を一意に決定することができます。各目盛りは後述のタイマーによって生成されます。各目盛りのタイミング値は10ミリ秒であるため、現在時刻のコードへのアクセスを容易にするために、カーネルコードにマクロが定義されています。このマクロはinclude/linux/sched.hファイルの192行目に定義されており、以下のような形式になっています。

```
#define CURRENT_TIME (startup_time + jiffies/HZ)
```

5.6.2 中でもHZ=100は、コアシステムのクロック周波数です。現在時刻マクロCURRENT_TIMEは、システムの起動時間startup_timeに、起動後にシステムが動作している時間jiffies/100を加えたものと定義されています。このマクロは、ファイルがアクセスされたときや、そのi-nodeが変更されたときの時間を修正するときに使用されます。

5.6.3 System Timing

システムの基本的なタイミングビートは、タイミングチップによって生成されます。Linuxの初期化時に

0.12カーネルの場合、PCのプログラマブル・タイミング・チップIntel 8253 (8254) のカウンタ・チャネル0はモード3で動作するように設定されており、初期カウント値LATCHは10ミリ秒ごとに出力OUTに矩形波の立ち上がりエッジを発するように設定されています。8254チップのクロック入力周波数は1.193180MHzなので、初期カウント値LATCH=1193180/100は約11931となります。OUT端子はプログラマブルな割り込み制御チップのレベル0に接続されているため、システムは10ミリ秒ごとにクロック割り込み要求 (IRQ0) を発行する。このタイムビートがOSのパルスであり、これをシステムチックあるいはシステムクロックサイクルと呼んでいます。したがって、1ティックの時間が経過するたびに、システムはクロック割り込みハンドラ (timer_interrupt) を呼び出します。

クロック割り込みハンドラ timer_interrupt は、主にシステムが起動してから経過したクロックティックの数を jiffies 変数で累積するために使用されます。jiffiesの値は、クロック割込みが発生するたびに1ずつ増加します。その後、C言語の関数do_timer()を呼び出して処理を進めます。呼び出し時のパラメータCPLは、割り込みプログラムのセグメントセレクタ（スタックに格納されているCSセグメントレジスタの値）から、現在のコード特権レベルCPLを取得します。

do_timer()関数は、特権レベルに基づいて、現在のプロセスの実行時間を蓄積する。CPL=0の場合、プロセスがカーネルモードで実行されているときに中断されていることを意味するので、カーネルはプロセスのカーネル状態の実行時間stimeを1つ増やし、そうでなければプロセスのユーザ状態の実行値を1つ増やします。フロッピーディスクプログラムfloppy.cが動作中にタイマーを追加した場合、タイマーリストが処理されます。タイマーが期限切れ（デクリメント後に0になる）になると、そのタイマーのハンドラが呼び出されます。その後、現在のプロセス実行時間が処理され、現在のプロセス

実行タイムスライスが1つデクリメントされます。タイムスライスとは、プロセスが切り替わる前に実行し続けることができるCPU時間のことです。単位は、上記で定義したティック数です。プロセスのタイムスライスの値がデクリメントされ、まだ0より大きい場合は、そのタイムスライスが使い切られていないことを意味するので、`do_timer()`を終了し、現在のプロセスの実行を継続する。この時、プロセスのタイムスライスがデクリメントされて0になっていれば、そのプロセスがCPUのタイムスライスを使い切ったことを意味し、プログラムは中断されたプログラムのレベルに応じて、さらなる処理方法を決定する。中断された現在のプロセスがユーザーモードで動作している(特権レベルが0より大きい)場合、`do_timer()`はスケジューラの`schedule()`を呼び出し、実行する別のプロセスに切り替えます。中断された現在のプロセスがカーネルモードで動作している場合、つまり、カーネルプログラムで実行中に中断された場合、`do_timer()`は直ちに終了します。したがって、この処理方法は、Linuxシステムプロセスがカーネルモードで動作しているときに、スケジューラによって切り替えられないことを決定します。つまり、カーネルモードで動作しているときは、プロセスはノンプリエンプティブである

のプログラムである。

上記のタイマーコードは、フロッピーモーターのオンとオフのタイミング操作に特化していることに注意してください。この

この種のタイマーは、最新のLinuxシステムに搭載されているダイナミックタイマーと同様に、カーネルのみで使用されます。このようなタイマーは、必要に応じて動的に作成し、タイミングが切れた ら動的に取り消すことができます。Linuxでは

0.12カーネルでは、最大64個のタイマーが同時に動作します。タイマーの処理コードは、`sched.c` プログラム283--368行目にあります。

5.7 Linux Process Control

プログラムとは、実行可能なファイルのこと、プロセスとは、実行中のプログラムのインスタンスのことです。Linuxでは、複数のプロセスを同時に実行できるタイムシェアリング技術を採用しています。タイムシェアリング技術の基本原理は、CPUの動作時間を一定の長さのタイムスライスに分割し、各プロセスが1つのタイムスライスで動作することである。プロセスのタイムスライスがなくなると、システムはスケジューラーを使って別のプロセスの実行に切り替えます。そのため、実際には、1つのCPUを搭載したマシンの場合、一度に実行できるプロセスは1つだけです。しかし、各プロセスは短いタイムスライスを実行するので（例えば、15システムチック = 150ミリ秒）、すべてのプロセスが同時に実行されているように見えます。

Linux 0.12カーネルの場合、システムは同時に最大64のプロセスを持つことができます。手動で作成された最初のプロセスを除いて、残りのプロセスはシステムコール`fork`を使用して既存のプロセスによって作成された新しいプロセスです。作成されたプロセスは子プロセスと呼ばれ、作成者は親プロセスと呼ばれます。

カーネルプログラムは、各プロセスを識別するためにプロセスID（Process ID、`pid`）を使用します。プロセスは、実行可能な命令コード、データ、スタックセクションで構成されています。プロセス内のコード部とデータ部は、それぞれ1つの実行ファイル内のコードセグメントとデータセクションに対応しています。各プロセスは、自分自身のコードを実行し、自分自身のデータとスタック領域にアクセスすることしかできません。プロセス間の通信は、システムコールを介して行う必要があります。CPUが1つしかないシステムでは、一度に実行できるプロセスは1つだけである。カーネルは、スケジューラーを介して各プロセスをタイムシェアリング方式で実行するようにスケジュールします。

Linuxシステムのプロセスは、カーネルモードとユーザーモードで実行され、それぞれが独立したカーネル状態スタックとユーザー状態スタックを使用することは既にご存知の通りです。ユーザー状態スタックは、プロセスが呼び出した関数のパラメータやローカル変数などを一時的に保存するために使用され、カーネル状態スタックには、カーネルプログラムが関数呼び出しを実行する際の情報が格納されています。

5.7.1 また、Linuxカーネルでは、プロセスを「タスク」と呼ぶことが多く、ユーザー空間で動作するプログラムを「プロセス」と呼びます。本書では、この2つの用語を混在させながら、このデフォルトルールに従うようにしています。

5.7.2 Task Data Structure

カーネルプログラムは、プロセステーブルを介してプロセスを管理し、各プロセスはプロセステーブルの1項目を占有します。Linuxシステムでは、プロセステーブルの項目はtask_structタスク構造体のポインタである。PCB (Process Control Block) やPD (Process Descriptor) と表記している書籍もあります。プロセスを制御・管理するためのすべての情報を保持しています。主な内容は、プロセスの現在の実行状況、シグナル、プロセス番号、親プロセス番号、実行時間の累積値、使用中のファイル、タスクのローカルディスクリプタ、タスクのステータスセグメント情報などです。タスクデータ構造は、ヘッダファイルinclude/linux/sched.hで定義されており、構造体の各フィールドの具体的な意味は以下の通りです。

```
struct task_struct {
    long state;                      // -1 unrunnable, 0 runnable (ready), > 0 stopped.
    long counter;                     // Task run time tick (decrement), run time slice.
    long priority;                   // Priority. When task starts running, counter=priority.
    long signal;                     // Signal bitmap, each bit is a signal( = bit offset + 1).
```

```

struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
long blocked;                // Process signal mask (Bitmap of masked signal).
int exit_code;                // Exit code after task stops, its parent will get it.
unsigned long start_code;     // Code start location in linear address space.
unsigned long end_code;       // Code length or size (bytes).
unsigned long end_data;       // Code size + data size (bytes).
unsigned long brk;            // Total size (number of bytes).
unsigned long start_stack;    // Stack bottom location.
long pid;                     // Process identifier.
long pgrp;                    // Process group number.
long session;                 // Process session number.
long leader;                  // Leader session number.
int groups[NGROUPS];          // Group numbers. A process can belong to more groups.
task_struct *p_pptr;          // Pointer to parent process.
task_struct *p_cptr;          // Pointer to youngest child process.
task_struct *p_ysptr;          // Pointer to younger sibling process created afterwards.
task_struct *p_osptr;          // Pointer to older sibling process created earlier.
unsigned short uid;           // User id.
unsigned short euid;           // Effective user id.
unsigned short suid;           // Saved user id.
unsigned short gid;           // Group id.
unsigned short egid;           // Effective group id.
unsigned short sgid;           // Saved group id.
long timeout;                 // Kernel timing timeout value.
long alarm;                   // Alarm timing value (ticks).
long utime;                   // User state running time (ticks).
long stime;                   // System state runtime (ticks).
long cutime;                  // Child process user state runtime.
long cstime;                  // Child process system state runtime.
long start_time;               // Time the process started running.
struct rlimit rlim[RLIM_NLIMITS]; // Resource usage statistics array.
unsigned int flags;             // per process flags.
unsigned short used_math;      // Flag: Whether a coprocessor is used.
int tty;                       // The tty subdevice number used. -1 means no use.
unsigned short umask;           // The mask bit of the file creation attribute.
struct m_inode * pwd;           // Current working directory i-node structure pointer.
struct m_inode * root;          // Root i-node structure pointer.
struct m_inode * executable;    // The pointer to i-node structure of the executable file.
struct m_inode * library;       // The loaded library i-node structure pointer.
unsigned long close_on_exec;    // 実行時にファイルハンドルを閉じるビットマップフラグ struct
file * filp[NR_OPEN];          // ファイル構造体のポインターテーブル、最大32項目まで。
                                // インデックスは、ファイルディスクリプターの値です。
struct desc_struct ldt[3];      // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
struct tss_struct tss;          // The task status segment structure TSS of the process.
} ;

```

◆ long state -- The state field contains current state of the process. At some point, a Linux process can be in one of five states and can transition between these states under the operation of the kernel scheduler. The five states are: running state (TASK_RUNNING), interruptible sleep state (TASK_INTERRUPTIBLE), uninterruptible sleep state (TASK_UNINTERRUPTIBLE), zombie state (TASK_ZOMBIE), and stopped state (TASK_STOPPED). The way the kernel changes the state of the process is described in the next section.

◆ long counter -- The counter field holds the number of time ticks that the process can execute before it is

◆ を一時的に停止します。つまり、他のプロセスに切り替わるには、通常、数システムクロックサイクルかかります。スケジューラはプロセスのカウンタ値を使って次に実行するプロセスを選択するので、カウンタはプロセスの動的な機能と考えることができます。カウンタの初期値は、プロセスが作成されたばかりのときの優先度と同じです。

◆ long priority -- Priority is used to assign the initial value to the counter. In Linux 0.12 this initial value is 15 system clock cycle times (15 ticks). When needed, the scheduler will use the value of priority to assign an initial value to the counter, see the sched.c and the fork.c programs. Of course, the unit of priority is also the number of time ticks.

◆ long signal -- The signal field is a bitmap of the signal currently received by the process. The bitmap has 32 bits, each bit represents a signal, and the signal value = bit offset value +1. So the Linux kernel has up to 32 signals. At the end of each system call process, the signal is preprocessed using the signal bitmap.

◆ struct sigaction sigaction[32] -- The sigaction structure array is used to store the operations and attributes used to process each signal. Each item of the array corresponds to one signal.

◆ long blocked -- The blocked field is a blocking bitmap of the signal that the process does not currently want to process. Similar to the signal field, each bit represents a blocked signal.

◆ int exit -- The exit field is used to save the exit code when the program terminates. After the child process ends, the parent process can query its exit code.

◆ unsigned long start_code -- The start_code field is the starting address of the process code in the CPU linear address space. In the Linux 0.1x kernel, its value is an integer multiple of 64MB.

◆ unsigned long end_code -- The end_code field holds the byte length value of the process code.

◆ unsigned long end_data -- The end_data field holds the code length of the process + the total byte length value of the data length.

◆ unsigned long brk -- The brk field is also the total byte length value (pointer value) of the process code and data, but also includes the uninitialized data area bss, as shown in Figure 5-12. This is the initial value of brk when a process starts executing. By modifying this pointer, the kernel can add and release dynamically allocated memory for the process. This is usually done by the kernel by calling the malloc() function and by calling the brk system call.

◆ unsigned long start_stack -- The start_stack field value points to the beginning of the stack in the process's logical address space. See also the stack pointer location in Figure 5-12.

◆ long pid -- Pid is the process identification number. It is used to uniquely identify the process.

◆ long pgrp -- Pgrp refers to the process group number to which the process belongs.

◆ long session -- Session is the session number of the process, which is the process ID of the session.

◆ long leader -- The leader is the first process number of the session. For the concept of process groups and sessions, see the instructions following Chapter 7, Program Listing.

◆ int groups[NGROUPS] -- Groups is an array of group numbers for each group to which the process belongs. A process can belong to more than one group.

◆ task_struct *p_pptr -- p_pptr is a pointer to the parent process's task structure.

◆ task_struct *p_cptr -- p_cptr is a pointer to the most recent subprocess's task structure. That is the youngest child's task structure. Refer to figure 5-20.

◆ task_struct *p_ysptr -- p_ysptr is a pointer to an adjacent process created later than itself. That is pointer to the younger sibling process.

◆ task_struct *p_osptr -- *p_osptr is a pointer to an adjacent process created earlier than itself. That is point to the older sibling process. See Figure 5-20 for the relationship between the above four pointers. In the task data structure of the Linux 0.11 kernel, there is a parent process number field father, but it is not used in the

0.12 kernel. At this point we can use the process's `pptr->pid` to get the process number of the parent process.

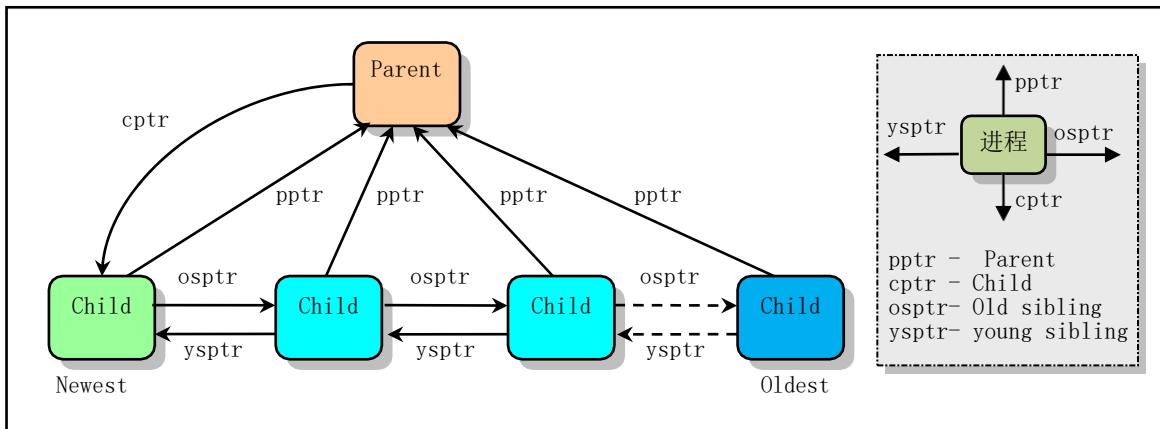


図5-20 プロセスポインターの関係

- ◆ `unsigned short uid` -- Uid is the user identification number (user id) that owns the process.
- ◆ `unsigned short euid` -- Euid is the effective user identification number that indicates the permissions to access the file.
- ◆ `unsigned short suid` -- Suid is the saved user identification number. When the set-user-ID flag of the execution file is set, the suid of the execution file is saved in the suid. Otherwise suid is equal to the euid of the process.
- ◆ `unsigned short gid` -- Gid is the group identification number (group id) to which the user belongs. It identifies the user group that owns the process.
- ◆ `unsigned short egid` -- Egid is an effective group identification number that indicates the permissions of the group of users to access the file.
- ◆ `unsigned short sgid` -- Sgid is the saved user group identification number. When the set-group-ID flag of the execution file is set, the gid of the execution file is stored in the sgid. Otherwise sgid is equal to the process's egid. For a description of these user id and group id, see the overview of the sys.c program in Chapter 5.
- ◆ `long timeout` -- Kernel timing timeout value.
- ◆ `long alarm` -- Alarm is the alarm timing value (number of ticks) for the process. If the process sets the field value using the system call `alarm()`, then when the system time ticking value exceeds the alarm field value, the kernel sends a `SIGALRM` signal to the process. By default this signal will terminate the execution of the program. Of course, we can also use the signal capture function (`signal()` or `sigaction()`) to capture the signal for the specified operation. The function `alarm()` starts at line 370 of `kernel/sched.c`. The kernel converts the function value of the function in seconds into a tick value, which is stored in the field after the current time tick value of the system.
- ◆ `long utime` -- Uttime is the cumulative time (ticks) that the process runs in user state.
- ◆ `long stime` -- Stime is the cumulative time (ticks) that the process runs in the system state.
- ◆ `long cutime` -- Cutime is the cumulative time (ticks) that child processes runs in user state.
- ◆ `long cstime` -- Cstime is the cumulative time (ticks) that child processes runs in system state.
- ◆ `long start_time` -- Start_time is the time when the process is generated and starts running.
- ◆ `struct rlimit rlim[RLIM_NLIMITS]` -- The resource usage statistics array for the process.
- ◆ `unsigned int flags` -- Its the flag for each process, and 0.12 kernel is not yet in use.
- ◆ `unsigned short used_math` -- It is a flag indicating whether the process uses acoprocessor.
- ◆ `int tty` -- It is the subdevice number of the process using the tty terminal. -1 means no use.

◆ `unsigned short umask` -- It is the 16-bit attribute mask word used by the process to create a new file (each bit represents a file), that is, the access attribute set by the new file. If a bit of the mask word is set, it means that the corresponding attribute is disabled (masked). This attribute mask word is used with the attribute value given when the file was created (`mode &~umask`) as the actual access attribute of the newly created file. See the `include/fcntl.h` and `include/sys/stat.h` files for the specific meaning of the masked word and file attributes.

◆ `struct m_inode * pwd` -- `Pwd` is a pointer to the i-node structure of the current working directory of the process. Each process has a current working directory that resolves relative path names and can be changed using the system call `chdir`.

◆ `struct m_inode * root` -- `Root` is the process's own root i-node structure. Each process can have its own specified root directory for parsing absolute path names. Only the superuser can modify this root directory by calling `chroot`.

◆ `struct m_inode * executable` -- `Executable` is the pointer to the i-node structure in memory for the execution file of the process. The system can use this field to determine if there is another process running the same executable file in the system. If so, then the in-memory i-node reference count value of `executable->i_count` will be greater than 1. When the process is created, the field is given the same value as the same field of the parent process, which means that the same program is being run with the parent process. When the kind of `exec()` function is called to execute a specified new executable file, the field value is replaced with the memory i-node pointer of the new program executed by the `exec()` function. When the process calls the `exit()` function and performs exit processing, the reference count of the memory i node pointed to by this field is decremented by 1, and the field will be blanked. The main role of this field is reflected in the `share_page()` function of the `memory.c` program. This function code can determine whether there are multiple copies of the currently running program in the system (at least 2) according to the reference count of the node pointed to by the execution of the process. If so, try a page sharing operation between them.

◆ システムの初期化時には、システムで作成されたすべてのタスクの実行度は、`execve()`関数を実行する最初の呼び出しの前に0になっています。つまり、カーネルコードに直接含まれているすべてのタスクの実行可能フィールドは0です。タスク0のコードはカーネルコードに含まれているため、システムがファイルシステムから読み込むことはありません。そのため、カーネルコード内の実行可能コードは0という固定値になっています。また、新しいプロセスを作成する際、`fork()`は親プロセスのタスクデータ構造をコピーするため、タスク1の実行可能コードも0となります。しかし、`execve()`を実行した後、実行可能コードには、実行されるファイルのメモリノードへのポインタが与えられます。それ以降は、すべてのタスクのこの値が0になることはありません。

◆ `unsigned m_inode * library` -- `Library` is the i-node structure pointer of the library file that is loaded when the program is executed.

◆ `unsigned long close_on_exec` -- It is a file descriptor (file handle) bitmap flag for a process. Each bit represents one file descriptor that is used to determine the file descriptor that needs to be closed when the system call `execve()` is called (see `include/fcntl.h`). When a program creates a child process using the `fork()`, it usually calls the `execve()` function in the child process to load another new program. At this point the child process will be completely replaced by the new program and the new program will start executing in the child process. If the corresponding bit of a file descriptor in `close_on_exec` is set, then the file descriptor corresponding to the open file will be closed when the child process executes the `execve()`. That is, the file descriptor is closed in the new program, otherwise the file descriptor will always be open.

◆ `struct file * filp[NR_OPEN]` -- It is a table of file structure pointers for all open files used by the process,

up to a maximum of 32 entries. The value of the file descriptor is the index value in the structure table. Each of

◆これらは、ファイル記述子がファイルポインタの位置を特定し、ファイルにアクセスするために使用されます。

◆ struct desc_struct ldt[3] -- It is the process local descriptor table structure LDT. It defines the code segment and data segment of the task in the virtual address space. Where array item 0 is a null item, item 1 is a code segment descriptor, and item 2 is a data segment (including data and stack) descriptor.

◆ struct tss_struct tss -- It is the task state segment TSS information structure of the process. The tss_struct structure holds all register values of the current processor when the task is switched out from execution. When the task is re-executed, the CPU uses these values to restore to the state when the task was switched out and starts execution.

5.7.3 プロセスが実行されているとき、CPUのすべてのレジスタの値、プロセスの状態、スタックの内容などをプロセスのコンテキストと呼びます。カーネルは、別のプロセスに切り替える必要があるときには、現在のプロセスの状態をすべて保存する、つまり、現在のプロセスのコンテキストを保存して、再びプロセスを実行するときに、切り替える前の状態に戻せるようにする必要があります。Linuxでは、現在のプロセスのコンテキストは、タスクのタスクデータ構造に保存されています。割り込みが発生すると、カーネルは割り込みプロセスのコンテキストで、カーネルステート内の割り込みサービスルーチンを実行します。同時に、使用する必要のあるすべてのリソースを保持し、割り込みサービスが終了したときに、割り込みプロセスの実行を再開できるようにします。

5.7.4 Process Running States

図5-21に示すように、プロセスはその寿命の間、プロセス・ステートと呼ばれる異なるステートのセットに入ることができます。図の中で数字が異なる円は、それぞれ異なる状態を表しています。前述のように、プロセス状態はプロセスタスク構造の状態フィールドに保存されます。

プロセスがCPUの使用を待っている場合や実行中の場合は、準備完了状態や実行中状態と言います。このとき、プロセス状態フィールドの値はTASK_RUNNINGとなります。プロセスがシステム・リソースやイベントの発生を待っている間にスリープ状態になっている場合は、割り込み可能なスリープ状態、または割り込み不可能なスリープ状態と言われます。このとき、プロセスの状態フィールドは、それぞれTASK_INTERRUPTIBLEまたはTASK_UNINTERRUPTIBLEとなります。プロセスが終了したものの、カーネルリソースを完全に解放していない場合、そのプロセスはデッドステートにあると言われます。この時点では、プロセスの状態フィールド値はTASK_ZOMBIEです。プロセスが一時的に停止している場合は、サスPEND状態といいます。この時点で、プロセスの状態フィールド値はTASK_STOPPEDです。

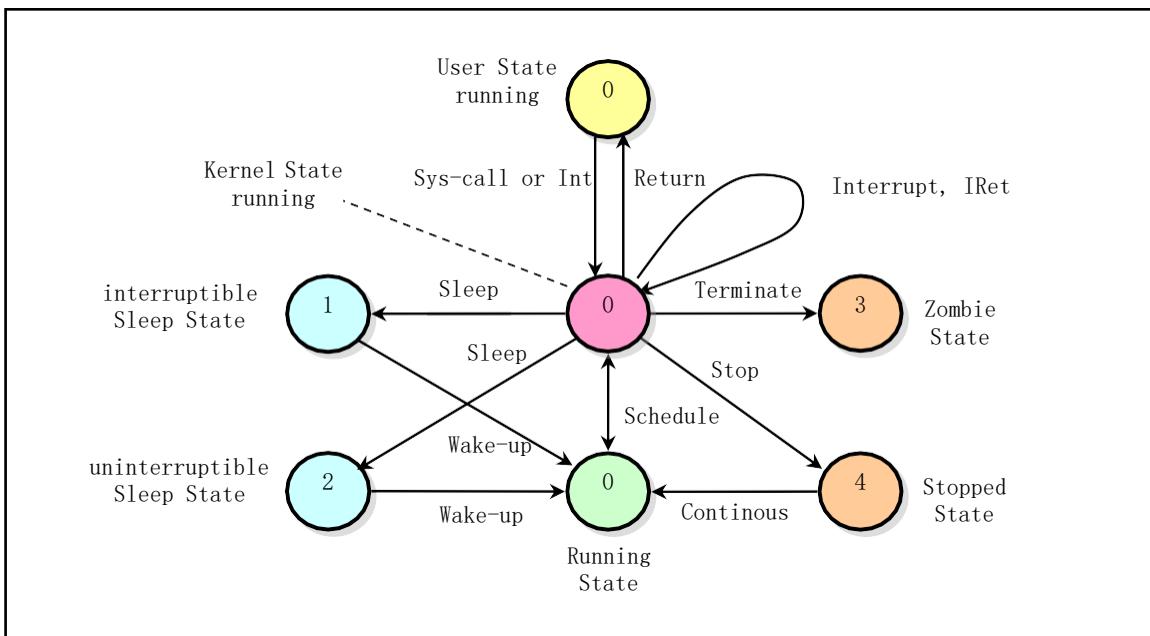


図5-21 プロセスの状態と変換の関係

Linuxプロセスの5つの状態を表す定数シンボル名を以下に示します。これらは、
include/linux/sched.hの46～50行目で定義されています。

[46](#) // プロセスの状態を定義します。

```

47 #define TASK_RUNNING      0 // is running or ready to run
48 #define TASK_INTERRUPTIBLE 1 // is in an interruptible wait state.
49 #define TASK_UNINTERRUPTIBLE 2 // uninterruptible wait state for wait I/O operation.
49 #define TASK_ZOMBIE        3 // is in a zombie state and has been terminated.
50 #define TASK_STOPPED       4 // The process has stopped.
  
```

◆ Running status (0, TASK_RUNNING)

◆ プロセスがCPUによって実行されているとき、またはスケジューラによっていつでも実行できる状態にあるとき、そのプロセスは実行状態と呼ばれます。また、図5-21のように、現時点ではCPUが実行していない場合は、実行準備状態といいます。図では、上から下までの真ん中の3つの円には同じ値0が含まれており、すべて準備完了状態または実行状態であることを意味している。プロセスには、カーネルモードとユーザーモードの2種類があります。プロセスがカーネルコードを実行している状態をカーネル実行状態、または単にカーネルモードと呼び、プロセスが自分のコードを実行している状態をユーザー実行状態（ユーザーモード）と呼びます。システムリソースが利用可能になると、プロセスはウェイクアップし、実行可能な状態になります。これを準備状態と呼びます。これらの状態（図の中段）は、カーネルでは同じ方式を表し、TASK_RUNNING状態と言われています。新しいプロセスが作られたばかりの時は、この状態（下の0）になっています。

◆ Interruptible sleep state (1, TASK_INTERRUPTIBLE)

◆ プロセスが割込み可能な待機（スリープ）状態にあるとき、システムはプロセスの実行をスケジューリングしません。システムが割り込みを発生させたり、プロセスが待機しているリソースを解放したり、プロセスが信号を受信したりすると、プロセスを起こして準備状態（つまり実行状態）に変更することができます。

◆ Uninterruptible sleep state (2, TASK_UNINTERRUPTIBLE)

この状態は、信号を受信しても起こされないことを除けば、割り込み可能なスリープ状態と同様です。

- ◆しかし、この状態のプロセスは、`wake_up()`関数を使って明示的にアウェイクしたときにのみ、実行可能なレディ状態に変換することができます。この状態は一般的に、プロセスが邪魔されずに待つ必要がある場合や、待機イベントがすぐに発生する場合に使用されます。
- ◆Zombie state (3, `TASK_ZOMBIE`)
 - ◆あるプロセスが実行を停止したにもかかわらず、その親プロセスが`wait()`を呼び出して状態を問い合わせていない場合、そのプロセスは死んだ状態にあると言われています。親プロセスが実行を停止したという情報を得るためには、子プロセスのタスクデータ構造の情報を保持する必要があります。親プロセスが`wait()`を呼び出して子プロセスの情報を取得すると、その状態にあるプロセスのタスクデータ構造が解放されます。
- ◆Stop status (4, `TASK_STOPPED`)
 - プロセスが信号`SIGSTOP`、`SIGTSTP`、`SIGTTIN`、`SIGTTOU`を受信すると、停止状態になります。`SIGCONT`信号をプロセスに送ると、実行可能な状態に移行します。デバッグ中にプロセスが受信した信号は、この状態に入る。Linux 0.12では、この状態への変換処理は実装されていません。この状態のプロセスは、プロセス終了として処理されます。

プロセスが時間切れになると、システムはスケジューラーを使って、実行するプロセスを別のプロセスに強制的に切り替えます。また、プロセスがカーネルモードで実行する際に、システムの特定のリソースを待つ必要がある場合、プロセスは`sleep_on()`または`interruptible_sleep_on()`を呼び出してCPUの使用権を自発的に放棄し、スケジューラに他のプロセスを実行させます。プロセスはスリープ状態 (`TASK_UNINTERRUPTIBLE`または`TASK_INTERRUPTIBLE`) になります。

5.7.5 カーネルは、プロセスが「カーネル実行状態」から「スリープ状態」に移行したときにのみ、プロセスの切り替え操作を行います。カーネルモードで動作しているプロセスは、他のプロセスからプリエンプトされることなく、また、あるプロセスが他のプロセスの状態を変更することもできません。プロセス切り替え時のカーネルのデータエラーを避けるために、カーネルはクリティカルエリアコードの実行時にすべての割り込みを無効にします。

5.7.6 Process initialization

boot/ディレクトリでは、ブートローダがディスクからカーネルをメモリにロードし、システムをプロテクトモードにした後、システム初期化プログラム`init/main.c`を起動する。メインプログラムは、まずシステムの物理メモリの割り当てと使用方法を決定し、次にカーネルの各部の初期化関数を呼び出して、メモリ管理、割込み処理、ロックデバイス、キャラクタデバイス、プロセス管理、ハードディスクやフロッピーディスクのハードウェアを初期化する。これらの操作が完了すると、システムの各部分が動作可能になる。その後、プログラムの制御は「手動」でタスク0（プロセス0）に移され、`fork()`コールを使って初めてプロセス1が生成されます。プロセス1では、プログラムは引き続きアプリケーション環境の初期化とシェル・ログイン・プログラムの実行を行います。元のプロセス0は、システムがアイドル状態のときに実行されるようにスケジュールされます。この時点では、タスク0は`pause()`システムコールを実行するだけで、その結果、スケジューラ関数が実行されます。

「タスク0の実行に移す」という処理は、マクロ `move_to_user_mode` (`include/asm/system.h`)によって行われます。これは、`main.c`プログラムの実行フローを、カーネルモード(特権レベル0)からユ

ユーザー モード(特権レベル3)のタスク0に移動させるものです。移動の前に、システムはまず、スケジューラーの初期化処理（`sched_init()`）でタスク0の実行環境を設定します。これには、タスク0のデータ構造（`include/linux/sched.h`）のフィールドの値を手動で事前に設定したり、タスク0のタスク・ステート・セグメント（TSS）ディスクリプターやグローバル・ディスクリプター・テーブルのローカル・ディスクリプター・テーブル（LDT）を追加したりします。セグメント記述子は、それぞれタスクレジスタtrとローカル記述子テーブルレジスタldtrにロードされます。

ここで強調しておきたいのは、カーネルの初期化は特殊な処理であり、カーネルの初期化コードはタスク0のコードでもあるということです。タスク0のデータ構造の初期データセットから、コードのベースアドレスである

タスク0のコードセグメントおよびデータセグメントのベースアドレスは0で、セグメント長は640KBです。カーネルコードセグメントとデータセグメントのベースアドレスは0で、セグメント長は16MBです。したがって、タスク0のコード・セグメントとデータ・セグメントは、それぞれカーネル・コード・セグメントとデータ・セグメントに含まれます。カーネル初期化プログラムmain.cはタスク0のコードですが、タスク0に移行する前にシステムがカーネルモードの特権レベル0でメインコードを実行している点が異なります。マクロmove_to_user_modeの機能は、実行特権レベルをカーネル状態のレベル0からユーザーモードのレベル3に移動させても、元のコードの命令ストリームを継続して実行するために使用されます。

タスク0への移行時には、マクロmove_to_user_modeは、割込み復帰命令を使用して、特権レベルの変更を引き起こします。制御転送にこの方法を使用するのは、CPUの保護機構が原因です。CPUは、ゲートや割り込み、トラップゲートを呼び出すことで、低レベル（特権レベル3など）のコードを呼び出したり、高レベルのコードに転送したりすることはできますが、その逆はできません。そこでカーネルは、低レベルのコードを返すために、IRETを模したメソッドを使います。この方法の主な考え方は、割り込みリターン命令の内容をスタックに構築し、リターンアドレスのセグメントセレクタを、特権レベル3のタスク0コードセグメントセレクタに設定します。その後、割り込みリターン命令IRETを実行すると、システムCPUは特権レベル0から外層の特権レベル3にジャンプします。特権レベル変更時の割込み復帰スタック構造図は図5-22を参照してください。

図 5-22 特権レベル変更時のインタラプト・リターン・スタック構造

マクロmove_to_user_modeは、まずタスク0のスタックセグメント（つまりデータセグメント）のセレクタとカーネルスタックポインタをカーネルスタックにプッシュします。次に、フラグレジスタの内容をプッシュします。最後に、タスク0コードセグメントセレクタと、「割込み復帰」後に次に実行される命令のオフセットを押し込みます。

IRET命令が実行されると、CPUはCS:EIPにリターンアドレスを送信し、スタックのフラグレジスタの内容をポップアップします。このとき、CPUは送信先コードセグメントの特権レベルを3と判断しているため、現在のカーネル状態のレベル0とは異なります。その後、CPUはスタック内のスタックセグメントセレクタとスタックポインタをSS:ESPにポップします。特権レベルの変更に伴い、セグ

メント・レジスタDS、ES、FS、GSの値が無効になり、CPUはこれらのセグメント・レジスタをクリアします。そのため、IRET命令を実行した後、これらのセグメント・レジスタを再ロードする必要があります。その後、タスク0のコードに対して、システムは特権レベル3で動作を開始します。この時点では、移動前に使用していたオリジナルのスタックがタスク0の加入者局スタックとして使用されます。カーネル状態のスタックは、タスク0のTSSの内容で指定され、タスクデータ構造があるページの先頭(PAGE_SIZE + (long) & init_task)から指定されます。タスク0のタスクデータ構造(ユーザースタックポインタを含む)は、後で新しいプロセスを作成する際にコピーする必要があるため、タスク1(プロセス1)が作成されるまで、タスク0のユーザモードスタックは「クリーン」な状態を保つ必要があります。

5.7.7 Creating new processes

Linuxシステムで新しいプロセスを作成するには、`fork()`システムコールを使用する必要があります。初期化後に作成されるプロセスは、すべてプロセス0の子プロセスを起点としています。

新しいプロセスを作成する過程で、システムはまず、タスクデータ構造の配列の中から、どのプロセスにも使われていない空のアイテム（空のスロット）を見つけます。システムがすでに64のプロセスを実行している場合、タスク配列テーブルに利用可能な空のアイテムがないため、`fork()`システムコールはエラーを返します。そうでなければ、システムは新しいプロセスがタスク・データ構造情報を保存するためにメイン・メモリ・エリアにメモリのページを申請し、新しいプロセスのタスク・データ構造のテンプレートとして現在のプロセスのタスク・データ構造のすべての内容をコピーします。処理されていないこの新しいプロセスがスケジューラーによって実行されるのを防ぐために、新しいプロセスの状態を直ちに中断不可能な待機状態（TASK_UNINTERRUPTIBLE）に設定する必要があります。

その後、コピーされたタスクデータ構造が変更されます。現在のプロセスを新しいプロセスの親に設定し、シグナルビットマップをクリアして新しいプロセスの統計情報をリセットし、ランタイムスライスの初期値を15システムティック（150ミリ秒）に設定します。次に、タスクステータスセグメント(TSS)のレジスタの値を現在のプロセスに応じて設定します。プロセスの生成時には、新しいプロセスの戻り値は0でなければならないため、`tss.eax = 0`を設定する必要があります。新しいプロセスのカーネル状態stack・ポインタ`tss.esp0`は、新しいプロセス・タスク・データ構造が配置されているメモリ・ページの先頭に設定され、stack・セグメント`tss.ss0`はカーネル・データ・セグメント・セレクタに設定されます。`Tss.ldt`には、GDT内のローカルテーブル記述子のインデックス値が設定されます。現在のプロセスがコプロセッサを使用している場合は、コプロセッサの完全な状態を新しいプロセスの`tss.i387`構造体に保存する必要があります。

その後、新しいタスクのコードおよびデータセグメントのベースアドレスと制限長を設定し、現在のプロセスのメモリページテーブルをコピーします。なお、この時点では新プロセスに実際の物理メモリページを割り当てず、親プロセスのメモリページを共有させています。親プロセスと新プロセスのいずれかにメモリの書き込み操作があった場合にのみ、システムは書き込み操作に関連するメモリページを割り当てます。このような処理をCopy On Write技術といいます。この技術の詳細については、「メモリ管理」の章の「Write-Only Replication」のメカニズムを参照してください。

その後、親プロセスに開いているファイルがあれば、対応するファイルの開封回数を1回増やしてください。続いて、新しいタスクのTSSおよびLDT記述子のエントリがGDTに設定され、ベース・アドレス情報が新しいプロセス・タスク構造の`tss`および`ldt`を指すようになります。最後に、新し

いタスクを実行可能な状態に設定し、新しいプロセス番号を返します。

5.7.8 また、新規に子プロセスを作成することと、実行ファイルを読み込むことは別の概念であることに注意してください。子プロセスが作成されると、親プロセスのコードとデータ領域を完全にコピーし、そこに子プロセス部分のコードを実行する。ロックデバイスでプログラムを実行する場合、一般的にはexec()システムコールを実行して子プロセスで実行される。exec()に入ると、子プロセスの元のコードやデータ領域はクリア(解放)されます。子プロセスが新しいプログラムの実行を開始すると、この時点ではカーネルがロックデバイスからプログラムのコードをロードしていないので、CPUは直ちにページが存在しないという例外を発生させます。ここで、メモリマネージャがロックデバイスから対応するコードページをロードし、CPUは例外発生の原因となった命令を再実行します。ここまでで、新しいプログラムのコードが実際に実行され始めます。

5.7.9 Process Scheduling

カーネル内のスケジューラーは、システム内で次に実行するプロセスを選択するために使用されます。
この選択的運用

のメカニズムは、マルチタスクOSの基礎となるものです。スケジューラは、実行中のすべてのプロセス間でCPUランタイムを割り当てる管理コードと考えることができます。前述の説明からわかるように、Linuxプロセスはプリエンプティブですが、プリエンプティブされたプロセスはTASK_RUNNING状態のままであるが、CPUによって一時的に実行されているわけではありません。プロセスのプリエンプションは、プロセスがユーザー状態の実行フェーズにあるときに発生し、カーネル状態で実行されているときはプリエンプションできません。

プロセスがシステムリソースを有効に利用し、プロセスの応答速度を速くするためには、プロセス切り替えのスケジューリングに一定のスケジューリング戦略を採用する必要があります。Linux 0.12では、優先キューイングによるスケジューリング方式を採用しています。

The scheduler

`schedule()`関数は、まずタスク配列をスキャンします。準備状態(TASK_RUNNING)のタスクごとに、ランタイムカウントカウンタの値を比較して、現在どのプロセスの実行時間が最も短いかを判断します。どの値が大きいかは、実行時間が長くないことを意味するので、そのプロセスを選択し、タスク切り替えマクロ関数を使って、そのプロセスに切り替えます。

このとき、TASK_RUNNING状態のプロセスのタイムスライスをすべて使い切っていた場合、システムは各プロセスの優先度の値に応じて、システム内の各プロセス(スリープ中のプロセスを含む)に必要なタイムスライス(カウンタ)を再計算します。その計算式は

$$\frac{\text{カウンター}}{2} \text{ カウンター プライオリティ}$$

したがって、眠っているプロセスにとっては、起こされたときのタイムスライスカウンタの値が大きくなります。次に、`schedule()`関数は、task配列のTASK_RUNNING状態の全プロセスを再スキャンし、プロセスが選択されるまで処理を繰り返します。最後に`switch_to()`が呼ばれ、実際のプロセス切り替え操作が行われます。

この時、他に実行できるプロセスがなければ、システムはプロセス0を選択して実行します。Linux 0.12の場合、プロセス0は`pause()`を呼び出して自分を割り込み可能なスリープ状態にし、再び`schedule()`を呼び出します。しかし、プロセスをスケジューリングする際、`schedule()`はプロセス0の状態を気にしません。システムがアイドル状態である限り、プロセス0の実行がスケジューリングされます。

Process switching

新しい実行可能なプロセスが選択されるたびに、`schedule()`関数は`switch_to()`マクロを呼び出し、実際のプロセス切り替え操作を行います。`switch_to()`は、`include/asm/system.h`で定義されています。

このマクロは、CPUの現在のプロセスの状態(コンテキスト)を、新しいプロセスの状態に置き換えます。`switch_to()`は切り替えを行う前に、まず切り替え先のプロセスが現在のプロセスであるかどうかを確認し、そうであれば何もせずにそのまま終了します。そうでない場合は、まずカーネルのグロ-

バル変数currentに新しいタスクのポインタが設定され、次に新しいタスクのタスクステートセグメントTSSのアドレスにジャンプして、CPUにタスク切り替え動作を行わせます。このとき、CPUはすべてのレジスタの状態を、カレントタスクレジスタTRのTSSセグメントセレクタが指すカレントプロセスタスクデータ構造のtss構造に保存します。その後、新しいタスク状態セグメントセレクタが指す新しいタスクデータ構造のtss構造のレジスタ情報がCPUに復元されます。その後、システムは新しいスイッチを実行するタスクを正式に開始しました。この処理を図5-23に示します。

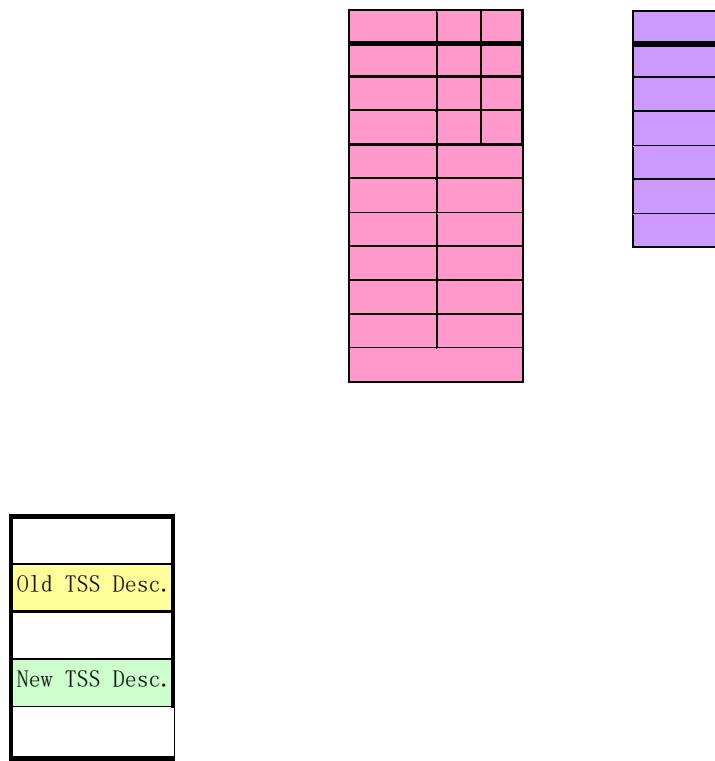


図 5-23 タスク切り替え動作図

5.7.10 Terminating a process

プロセスの実行が終了したり、実行が途中で終了したりすると、カーネルはプロセスが占有していたシステムリソースを解放する必要があります。これには、プロセスの実行中に開いたファイルや使用したメモリなどが含まれます。

ユーザープログラムがexit()システムコールを呼び出すと、カーネル関数do_exit()が実行される。この関数はまず、プロセスのコードセグメントとデータセグメントが占有していたメモリページを解放し、プロセスが開いていたすべてのファイルを閉じ、現在の作業ディレクトリ、ルートディレクトリ、プロセスが使用しているプログラムを実行しているi-nodeを同期させます。プロセスが子プロセスを持つ場合は、initプロセスをそのすべての子プロセスの親とします。プロセスがセッションヘッダプロセスであり、制御端末を持っている場合は、制御端末を解放し、ハングアップ信号SIGHUPをセッションに属するすべてのプロセスに送信します。これにより、通常はセッションの全プロセスが終了します。そして、プロセスの状態をTASK_ZOMBIEに設定します。そして、元の親プロセスにSIGCHLDシグナルを送り、子プロセスの一部が終了したことを知らせます。最後にdo_exit()は、他のプロセスを実行するためにスケジューラを呼び出します。プロセスが終了しても、親プロセスもそ

の情報を使用する必要があるため、タスクデータ構造は残っていることがわかります。

子プロセスの実行中、親プロセスは通常、`wait()`または`waitpid()`関数を使用して子プロセスが終了するのを待ちます。待機中の子プロセスが終了してゾンビ状態になると、親プロセスは子プロセスが費やした時間を自分のプロセスに蓄積します。最終的には、子プロセスのタスクデータ構造が占有していたメモリページが解放され、タスク配列で子プロセスが占有していたポインタ項目が空になります。

5.8 How to use the stack in Linux

このセクションでは、起動からシステムアップタイムまで、Linux カーネルがどのようにスタックを使用するかの概要を説明します。この部分の説明は、カーネルコードと密接に関連しているので、最初は飛ばしても構いません。関連するコードを読むときに戻ってきて勉強することができます。

Linux 0.12システムでは、4種類のスタックが使用されています。1つはシステムブートの初期化時に一時的に使用されるスタック、もう1つはプロテクトモードに入った後にカーネルを初期化するために使用されるスタックで、カーネルコードのアドレス空間の固定された場所にあります。このスタックは、後にタスク0が使用するユーザモードスタックでもあります。次のスタックは、各タスクがシステムコールを通じてカーネルコードを実行する際に使用するスタックで、タスクのカーネルレベルスタックと呼んでいます。各タスクは独立したカーネルモードスタックを持っています。最後は、タスクがユーザモードで実行するスタックで、タスク（プロセス）の論理アドレス空間の端の方にあります。

5.8.1 複数のスタックを使用したり、状況に応じて異なるスタックを使用する理由は主に2つあります。一つ目は、リアルモードがプロテクションモードに入ってから、CPUのメモリアクセスモードが変更されたため、セットアップスタック領域を再調整する必要があること。また、CPUの異なる特権レベルによるスタックの使用による保護問題を解決するために、レベル0のカーネルコードとレベル3のユーザーコードの実行には、異なるスタックを使用する必要があります。タスクがカーネルモードに入ると、そのTSSセクションに与えられた特権レベル0であるスタックポインタtss.ss0, tss.esp0を使用し、これがカーネルスタックとなります。元のユーザースタックポインタはカーネルスタックに保存されます。ユーザーモードからカーネルモードに戻るときは、ユーザーモードのスタックが復元されます。以下、別々に説明します。

5.8.2 Initialization phase

Boot initialization (bootsect.s, setup.s)

ブートセクトコードがROM BIOSによって物理メモリ0x7c00にロードされると、スタックセグメントが設定されません。もちろん、プログラムはスタックを使用しません。bootsectが0x9000:0に移動した後、スタックセグメントレジスタSSが0x9000に設定され、スタックポインタespレジスタが0xff00に設定されます。つまり、スタックの先頭は0x9000:0xff00になります。boot/bootsect.s の61行目を参照してください。bootsect で設定されたスタックセクションは、setup.s プログラムでも使用されます。これは、システムの初期化時に一時的に使用されるスタックです。

When entering protected mode (head.s)

head.sプログラムから、システムが正式にプロテクトモードで動作していることがわかります。この時点で、スタックセグメントはカーネルデータセグメント（0x10）に設定され、スタックポインタespはuser_stack配列の先頭を指すように設定され（head.sの31行目を参照）、1ページ（4K）のメモリがスタックとして使用するために予約されています。user_stack配列はsched.cの67～72行目で定義されており、合計1024個のロングワードを含んでいます。この配列の物理メモリ上の位置は図5-24

のとおりです。この時点では、カーネル自身が使用するスタックです。図中のアドレスは概算値であり、コンパイル時の実際の設定パラメータに関連しています。これらのアドレスの位置は、カーネルのコンパイル時に生成されるsystem.mapファイルに記載されています。

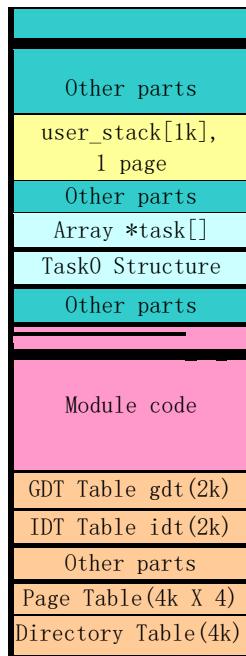


図 5-24 プロテクションモードへの移行時にカーネルが使用するスタックダイアグラム

Initialization (main.c)

5.8.3 init/main.c プログラムでは、タスク0に制御を移す move_to_user_mode() コードを実行する前に、システムは常に上記のスタックを使用します。move_to_user_mode() が実行された後、main.c のコードはタスク0に「スイッチ」されます。fork() システムコールを実行することで、main.c の init() はタスク1で実行され、タスク1のスタックを使用します。タスク0に "切り替わった" main() は、タスク0のユーザモードスタックとして、カーネル自身のスタックを使い続けます。タスク0が使用するスタックの詳細な説明は後述します。

5.8.4 Stack of tasks

各タスクには、ユーザモードとカーネルモードのプログラムを実行するための2つのスタックがあり、それぞれユーザモードスタック、カーネルモードスタックと呼ばれています。この2つのスタックの主な違いは、CPUの特権レベルが異なることに加えて、タスクのカーネルモードスタックは小さく、保存できるデータ量は (4096 - タスクデータ構造ブロック) バイト、つまり約3Kバイトを超えることはできません。タスクのユーザモードのスタックは、ユーザーの64MBのスペース内に拡張することができます。

When running in user mode

各タスク（タスク0、1を除く）は、それぞれ64MBのアドレス空間を持っています。タスク（プロセス）が生成されたばかりのとき、そのユーザ状態スタックポインタは、そのアドレス空間のほぼ最後（64MB先頭）に設定されます。実際には、図5-25に示すように、末尾部分には実行プログラムのパラメータや環境変数なども含まれており、その後にユーザースタック空間があります。アプリケーションがユーザモードで実行されているときは、常にこのスタックを使用します。スタックが実際に使用する物理メモリは、CPUのページング機構によって決定されます。Linux は copy-on-write を実装しているので、プロセスが生成された後、プロセスとその親プロセスがスタックを使用しない場合、両

者は同じスタックに対応する物理メモリページを共有します。カーネルメモリマネージャは、いずれかのプロセスがスタックの書き込み操作（プッシュ操作など）を行った場合にのみ、書き込みプロセスのために新しいメモリページを割り当てます。タスク0とタスク1のユーザースタックは、以下に説明するように特別なものです。

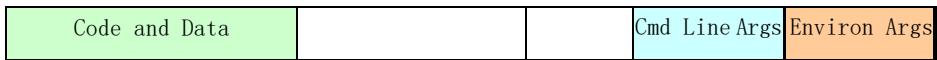


図5-25 論理空間におけるユーザーステートスタック

When running in kernel mode

各タスクは、カーネルコードの実行中にタスクが実行するための独自のカーネルモードスタックを持っています。そのリニアアドレスの位置は、タスクTSSセグメントのss0とesp0フィールドで指定されます。ss0はタスクのカーネル状態スタックのセグメントセレクタで、esp0はスタックのローアドレスです。そのため、タスクがユーザーコードからカーネルコードに転送されるときには、タスクのカーネル状態スタックは常に空です。タスクのカーネル状態スタックは、そのタスクデータ構造が配置されているページの最後、つまりタスクのタスクデータ構造 (task_struct) と同じページに配置されます。これは、新しいタスクが作成されたときに、fork()プログラムがタスクtssセグメントのカーネルレベルのスタックフィールド(tss.esp0とtss.ss0)に設定されることを意味します。kernel/fork.cの92行目を参照してください。

`p->tss.esp0 = PAGE_SIZE + (long)p;`
`p->tss.ss0 = 0x10`となります。

ここで、pは新しいタスクのタスクデータ構造ポインタ、tssはタスクステートセグメント構造です。カーネルは新しいタスクに対して、task_struct構造のデータを保持するためのメモリを要求し、tss構造（セグメント）はtask_structのフィールドとなります。このタスクのカーネルスタックセグメント値tss.ss0も0x10（つまりカーネルデータセグメントセレクタ）に設定されており、tss.esp0は図5-26のようにtask_struct構造体が保存されているページの終わりを指しています。実際には、tss.esp0はページの先頭バイト（外側）を指すように設定されています（図ではスタックの一番下にあります）。これは、インテルCPUがスタック操作を行う際に、まずスタックポインタのesp値をデクリメントしてから、スタックの内容をespポインタに保存するためです。

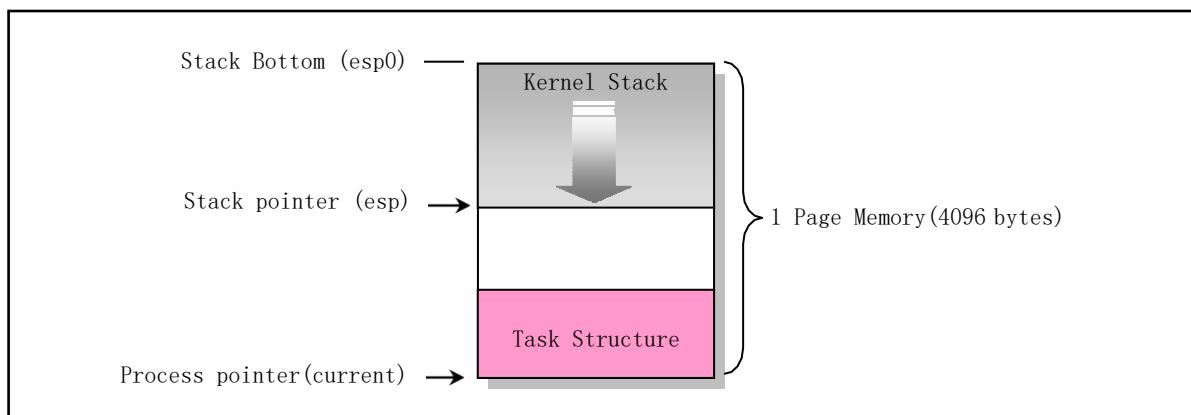


図 5-26 プロセスのカーネル状態スタック図

主記憶領域のタスクデータ構造を保存するためのメモリページを、なぜカーネルデータセグメントのデータに設定できるのでしょうか。つまり、なぜtss.ss0を0x10に設定できるのでしょうか。これは、

ユーザーのカーネル状態スタックがまだカーネルデータ空間に属しているからです。このことは、カーネルコードセグメントの長さから説明することができます。head.sプログラムの最後に、カーネルコードとデータセグメントのディスクリプターがそれぞれ設定され、セグメント長はすべて16MBに設定されています。このセグメントの長さは、Head.sプログラムが使用できる最大の物理メモリの長さです。

Linux 0.12のカーネルがサポートしています（head.s、110行目からのメモを参照）。そのため、カーネルコードは、主記憶領域を含む物理記憶領域全体のどこにでもアドレスを設定することができます。タスクがカーネルコードを実行し、そのカーネルスタックを使用する必要がある場合、CPUはTSS構造体を使用して、そのカーネル状態スタックをtss.ss0とtss.esp0の2つの値で構成されるように設定します。タスクが切り替わると、旧タスクのカーネルスタックポインタesp0は保存されません。CPUにとって、この2つの値はリードオンリーです。そのため、タスクがカーネルモード実行に入るたびに、そのカーネル状態スタックは常に空になります。

Stacks of task 0 and task 1

タスク0(アイドルプロセス)とタスク1(initプロセス)のスタックは特殊なので、特に説明が必要です。タスク0とタスク1のコードセグメントとタスクセグメントは同じで、制限長も640KBですが、異なるリニアアドレス範囲にマッピングされています。タスク0のセグメントベースアドレスはリニアアドレス0から始まり、タスク1のセグメントベースアドレスは64MBから始まりますが、いずれも物理アドレス0～640KBの範囲にマッピングされています。このアドレス範囲には、カーネルコードや基本データが格納される。move_to_user_mode()が実行されると、タスク0とタスク1のカーネル状態スタックは、それぞれのタスクデータ構造が配置されているページの末尾に配置され、タスク0のユーザ状態スタックは、プロテクトモードに入る前に使用されていたスタックで、sched.cのuser_stack[]配列の位置に配置されます。しかし、タスク1が実行を開始すると、タスク1がuser_stack[]にマッピングしたページテーブルエントリが読み取り専用に設定されているため、タスク1がスタック操作を行うと書き込みページ例外が発生します。そのため、カーネルはコピー온ライトのメカニズムを使って、タスク1のメインメモリ領域ページをスタック空間として割り当てます。そうして初めて、タスク1は独自の別のユーザースタックメモリページの使用を開始します。したがって、タスク1が実際に使用を開始するまで、タスク0のスタックは「クリーン」に保たれる必要があります。つまり、コピーされたスタックページにタスク0のデータが含まれていないことを保証するために、現時点ではタスク0はスタックを使用できません。

5.8.5 タスク0のカーネル状態スタックは、手動で設定された初期化タスクデータ構造で指定されます。そのユーザモードスタックは、図5-22に示すように、move_to_user_mode()を実行したときの模擬的なreturnの前のスタックに設定されています。特権レベルの変更が発生すると、宛先コードは新しい特権レベルのスタックを使用し、元の特権レベルコードのスタックポインタは新しいスタックに保存されることがわかっています。そのため、まずタスク0のユーザースタックポインタが現在の特権レベル0のスタックにプッシュされ、コードポインタもスタックにプッシュされます。この手動設定されたスタックでは、元のesp値はuser_stackの元の位置のままに設定され、元のssセグメントセレクタはユーザローカルテーブルLDTのデータセグメントセレクタに設定されている0x17に設定される。そして、タスク0のコードセグメントセレクタ0x0fが、スタック内の元のCSセグメントのセレクタとしてスタックにプッシュされ、次の命令のポインタが元のEIPとしてスタックにプッシュされます。このようにして、IRET命令を実行することで、タスク0のコードに「リターン」して実行を継続することができます。

5.8.6 Switching between the kernel stack and user stack

Linux 0.12 では、すべての割込みサービスルーチンはカーネルコードに属しています。タスクがユーザーコードで実行されているときに割り込みが発生すると、割り込みによってCPUの特権レベルがレベル3からレベル0に変更され、その時点ではCPUはユーザー状態のスタックとカーネル状態のスタックを切り替えることになります。CPUは新しいスタックのセグメントセレクタとオフセット値を、現在のタスクのTSSから取得します。割り込みサービスルーチンはカーネル内にあり、レベル0の特権レベルコードに属しているため、48ビットのカーネル状態スタックポインタはTSSのss0とesp0フィールドから取得します。新しいスタック（カーネル状態スタック）の位置を特定した後、CPUはまず元のユーザ状態スタックポインタssとespをカーネル状態スタックにプッシュし、次にフラグレジスタeflagsの内容とリターンポジションcsとeipをカーネル状態スタックにプッシュします。

システムコールはソフトウェア割り込みなので、タスクがシステムコールを呼び出すと、カーネルに入り、カーネル内の割り込みサービスコードを実行します。この時、カーネルコードはタスクのカーネルモードスタックを使って動作します。同様に、カーネルに入る際、特権レベルが変わると（ユーザー モードからカーネルモードへ）、ユーザー モードのスタックセグメントとスタックポインタ、およびeflagがタスクのカーネル状態のスタックに格納されます。カーネルを終了してユーザープログラムに戻るためにIRET命令が実行されると、ユーザー状態のスタックとeflagが復元されます。このプロセスを図5-27に示します。

図5-27 カーネルの状態とユーザーの状態のスタック切り替え

タスクがカーネルモードで動作している場合、CPUが割り込みに応答すれば、スタックスイッチの操作は不要になります。なぜなら、このときタスクが実行しているカーネルコードは、すでにカーネルの状態スタックを使用しており、優先レベルの変更を伴わないからです。そのため、CPUはeflagsと割込みリターンポインタcsとeipを現在のカーネル状態スタックにプッシュするだけで、割込みサービスプロセスを実行します。

5.9 File System for Linux 0.12

カーネルが正常に機能するためには、ファイルシステムのサポートが必要です。ルートファイルシステムは、カーネルに最も基本的な情報とサポートを提供するために使用されます。つまり、Linuxシステムが起動して開始すると、デフォルトのファイルシステムはルートファイルシステムとなります。これには、OSの最低限の設定ファイルやコマンド実行プログラムの一部が含まれる。Linuxシステムで使用されているUNIX系のファイルシステムでは、主にいくつかの指定されたディレクトリ、設定ファイル、デバイスドライバー、実行プログラム、ユーザー アプリケーション、データまたはテキストファイルが含まれます。これらには、一般的に以下のようなサブディレクトリやファイルが含まれる。

| | |
|---------|---|
| etc/ | The directory mainly contains some system configuration files; |
| dev/ | Contains device special files for file operation statement on devices; |
| bin/ | Store system execution programs. Such as sh, mkfs, fdisk, etc.; |
| usr/ | This directory stores library functions, manuals, and other files; |
| usr/bin | These directories store commands commonly used by users; |
| var/ | This directory is used to store system runtime data or log information. |

そのファイルシステムを保持する装置がファイルシステムデバイスです。例えば、一般的に使用されているWindows 10のOSでは、ハードディスクCがファイルシステムデバイスであり、ハードディスクに一定のルールに従って格納されているファイルがファイルシステムを構成しています。Windows 10では、NTFS、FAT32などのファイルシステムの

異なるフォーマットのファイルシステムがあります。Linuxには、EXT2やEXT3など、さまざまな形式のファイルシステムが用意されている。Linux 0.12カーネルでサポートされているファイルシステムは、MINIXオペレーションシステムの作者であるAndrew Tanenbaum氏が作成したMINIX 1.0です。しかし、現在のLinuxシステムで最も広く使われているのは、ext3またはext4ファイルシステムです。

第1章で説明したフロッピーディスク上で動作するLinux 0.12システムの場合、ブートイメージディスクとルートイメージディスクという2つの単純なフロッピーディスクから構成されています。Bootimage はブートイメージファイルで、主にディスクのブートセクタコード、オペレーティング・システム・ローダ、カーネルのバイナリコードなどが含まれています。ルートイメージは、カーネルに最も基本的なサポートを提供するためのルートファイルシステムです。これらの2つのディスクは、起動可能なDOSオペレーティングシステムのディスクに相当します。

Linuxの起動ディスクがルートファイルシステムをロードする際には、ディスクの起動セクタの509バイト目と510バイト目にあるワード (ROOT_DEV) のデバイス番号に応じて、指定されたデバイスからルートファイルシステムがロードされる。デバイス番号が0の場合、ルートファイルシステムは起動ディスクがある現在のドライブからロードされる必要があることを意味します。デバイス番号がハードディスク・パーティション・デバイス番号の場合、ルート・ファイル・システムは指定されたハードディスク・パーティションからロードされます。

現在、その仕様ファイルとなっているのが、Linux Foundation (LF) が管理している「Filesystem Hierarchy Standard」です。FHS仕様は、オリジナルのUNIXファイルシステムの組織構造と内容をベースに、1994年からLinuxディストリビューションシステムのファイルシステムの構造と内容の標準化を検討してきました。FHSは現在、Linux Standard Library (LSB) が提唱するISO LSBの正式規格の一部となっています。

5.10 Directories of kernel source code

Linuxカーネルはシングルコアモードのシステムなので、カーネル内のほとんどすべてのコードが密接に関連しています。それらの間のデータの依存関係や呼び出し関係は非常に密接です。そのため、ソースコードのファイルを読むときには、他のファイルを参照する必要があることが多いのです。そのため、カーネルのソースコードを読み始める前に、ソースコードファイルのディレクトリ構造や配置に慣れておく必要があります。

ここではまず、カーネルのソース・ディレクトリを、そのサブ・ディレクトリも含めてすべて列举します。そして、各ディレクトリに含まれるプログラムの主な機能を1つずつ紹介し、カーネルのソースコードの配置全体が頭の中で大まかに整理できるようにして、次の章でのソースコードの読み

込み作業を容易にします。

linux-0.12.tar.gzファイルをtarコマンドで解凍すると、カーネルのソースファイルがlinux/ディレクトリに配置されます。そのディレクトリ構造を図5-28に示します。

```

├── boot      System boot assembly programs
├── fs        File system files
├── include   Header files (*.h)
│   ├── asm    Files related to the CPU architecture
│   ├── linux   Linux kernel specific header files
│   └── sys    System data structures
├── init     Kernel initialization program
├── kernel   Kernel process scheduling, signal, system-calls, etc.
│   ├── blk_drv Block device driver
│   ├── chr_drv Character device driver
│   └── math    Math coprocessor simulation programs
├── lib       Kernel specific library
├── mm       Memory management programs
└── tools    Tool program for generating kernel image file

```

図 5-28 Linux カーネルソースコードのディレクトリ構造

5.10.1 このバージョンのカーネルのソースコードディレクトリーには、14のサブディレクトリーがあり、合計102のコードファイルが含まれています。以下では、これらのサブディレクトリの内容を1つずつ説明します。

5.10.2 Kernel home directory **linux**

linuxディレクトリは、ソースコードのメインディレクトリです。ホームディレクトリには、14個のサブディレクトリがすべて含まれているのに加えて、固有のMakefileが含まれています。このファイルは、コンパイル支援ソフトmakeのパラメータ設定ファイルです。makeツールの主な目的は、複数のソースファイルが存在するシステムにおいて、どのファイルが変更されたかを識別して、再コンパイルが必要なファイルを自動的に判断することである。したがって、makeツールは、プログラムプロジェクトの管理ソフトウェアである。

5.10.3 linuxディレクトリのMakefileは、すべてのサブディレクトリに含まれるMakefileもネストします。このようにして、makeはlinuxディレクトリ（サブディレクトリを含む）内で変更されたすべてのファイルを再コンパイルします。つまり、カーネル全体のすべてのソースコードファイルをコンパイルするには、linuxディレクトリでmakeソフトウェアを一度実行するだけです。

5.10.4 The boot program directory

bootディレクトリには、カーネルのソースコードファイルをコンパイルした最初のプログラムである3つのアセンブリ言語ファイルが含まれています。これら3つのプログラムの主な機能は、コンピュータの電源投入時にカーネルを起動し、カーネルコードをメモリにロードし、32ビット保護モードに入る前にシステムの初期化を行うことである。

- The bootsect.s program is a disk boot block program that resides in the first sector of the disk after compilation (boot sector, 0 track (cylinder), 0 head, 1st sector). After the PC is powered up and the ROM BIOS self-test, it will be loaded into the memory 0x7C00 for execution.
- The setup.s program is primarily used to read the machine's hardware configuration parameters and

move the kernel module system to the appropriate memory location.

- The head.s program will be compiled and connected to the foremost part of the kernel system module, mainly for the hardware device's probe settings and the initial setup of the memory management page.

bootsect.s と setup.s のプログラムは、as86 ソフトウェアを使って、as86 アセンブリ言語形式 (Microsoft のものと同様) でコンパイルする必要があります。Head.s は GNU as を使って、AT&T 形式のアセンブリ言語を使ってコンパイルする必要があります。これらの2つのアセンブリ言語については、次章のコードコメントと、コードリストに続く説明で簡単に説明します。

5.10.5 File System Directory fs

Linux 0.12カーネルのファイルシステムは、バージョン1.0のMINIXファイルシステムを使用していますが、これはLinuxがMINIXシステム上で開発されたためです。MINIXファイルシステムはクロスコンパイルが容易であり、LinuxのパーティションはMINIXからロードすることができます。MINIXファイルシステムが使用されていますが、LinuxはMINIXシステムとは異なる方法で処理します。主な違いは、MINIXではファイルシステムにシングルスレッドのアプローチを採用しているのに対し、Linuxではマルチスレッドのアプローチを採用していることです。マルチスレッド処理方式のため、Linuxのプログラムは、マルチスレッドによるレースコンディションやデッドロックに対処しなければなりません。そのため、Linuxのファイルシステムのコードは、MINIXのシステムよりもはるかに複雑になっています。競合状態の発生を回避するために、Linuxシステムではリソースの割り当てを厳密にチェックしています。また、カーネルモードで動作しているとき、タスクが能動的にスリープ（sleep()を呼び出す）しなければ、カーネルはタスクの切り替えを許可しないようになっている。

fs/ディレクトリは、ファイルシステム実装のディレクトリで、合計18個のCプログラムが含まれている。これらのプログラム間の主な参照関係と依存関係を図5-29に示す。図中の各ボックスはファイルを表し、基本的な参照関係で上から下に配置されている。ファイル名には接尾辞.cが省略されており、仮想ボックス内のプログラムファイルはファイルシステムの一部ではない。矢印付きの線は参照関係を、太い線は相互参照関係を示す。



図5-29 fsディレクトリ内の各プログラムの関係。

図からわかるように、このディレクトリのプログラムは、キャッシュ管理、ファイルの低レベル操作、ファイルのデータアクセス、ファイルの高レベル関数の4つの部分に分けられます。このディレクトリのファイルにアノテーションを付ける際にも、4つの部分に分けて記述します。

ファイルシステムは、メモリキャッシュの延長線上にあると考えることができます。ファイルシステムのデータにアクセスするには、まずキャッシュに読み込まれる必要がある。このディレクトリにあるプログラムは、主にファイルシステムの管理に使用されます。

- キャッシュ内のバッファブロックの割り当てと、ブロックデバイス上のファイルシステムの割り当てを行います。キャッシュを管理するプログラムはbuffer.cで、その他のプログラムは主にファイルシステムの管理に使われる。

- The file_table.c file currently has only one file handle (descriptor) structure array defined.
- The ioctl.c file will reference the functions in kernel/chr_drv/tty.c to implement the io control function of the character device.
- The exec.c file mainly contains an executive function do_execve(), which is the main function in all exec() function clusters.
- The fcntl.c program is used to implement the system call function for file i/o control.
- The read_write.c program is used to implement file read/write and locate three system call functions.
- The stat.c program implements two system calls that get file state information.
- The open.c program mainly contains system call functions that implement modification of file attributes and creation and closing of files.
- The char_dev.c mainly contains the character device read and write function rw_char().
- The pipe.c program contains pipe read and write functions and system calls to create pipes.
- The file_dev.c program contains file read and write functions based on the i-node and descriptor structure.
- The namei.c program mainly includes operation functions and system calls for directory names and file names in the file system.
- The block_dev.c contains block data read and write functions.
- The inode.c program contains functions for file system i-node operations.
- The truncate.c program is used to release the device data space occupied by files when deleting files.
- The bitmap.c file is used to process bitmaps of i-nodes and logical blocks in the file system.
- The super.c program contains handlers for file system superblocks.
- The buffer.c program is mainly used to process the memory cache.
- The select.c program is mainly used to effectively handle the problem of simultaneous I/O operations on multiple files.

バーチャルボックス内のll_rw_blockは、ブロックデバイスの基本的な読み取り機能です。fsディレクトリではなく、kernel/blk_drv/ll_rw_block.cにあるブロックデバイスのリード・ライトドライバ関数です。ここに置くことで、ファイルシステムが高速バッファキャッシュとブロックデバイスのドライバ(ll_rw_block())を経由して、ブロックデバイスのデータを読み書きする必要があることが明確になります。ファイルシステムのプログラム自体は、ブロックデバイスのドライバと直接やりとりすることはない。

5.10.6 プログラムのアノテーションを行う際には、これらのファイル内の主要な関数間の呼び出し階層を追加します。

5.10.7 Header file directory include

- ヘッダーファイルのディレクトリには、合計36個の.hヘッダーファイルがあります。メインディレクトリに13個、asmサブディレクトリに4個、Linuxサブディレクトリに11個、sysサブディレクトリに8個あります。これらのヘッダーファイルのそれぞれの機能について、以下に簡単に説明します。具体的な動作や情報については、「ヘッダーに関する注意事項」を参照してください。

- <a.out.h> The a.out header file defines the a.out executable file format and some macros.

- <const.h> The constant symbol file currently defines only the flags of i_mode field in the i-node.
- <ctype.h> The character type header file. Defines some macros for character type conversion.
- <errno.h> Error number header file. Contains various error numbers in the system. (Linus was introduced from minix).
- <fcntl.h> File control header file. The definition of the operation control constant symbol used for the

- ファイルとそのディスクリプターを表示します。
- <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal manipulation function prototypes.
- <stdarg.h> Standard parameter header file. Define a list of variable parameters in the form of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg and va_end) for the vsprintf, vprintf, and vfprintf functions.
- <stddef.h> The standard definition header file. NULL, offsetof(TYPE, MEMBER) is defined.
- <string.h> String header file. Mainly defines some embedded functions about string operations.
- <termios.h> Terminal input and output function header file. It mainly defines the terminal interface that controls the asynchronous communication port.
- <time.h> Time type header file. The most important of these is the definition of the tm structure and some function prototypes related to time.
- <unistd.h> Linux standard header file. Various symbol constants and types are defined and various functions are declared. If _LIBRARY_ is defined, it also includes the system call number and the inline assembly _syscall0().
- <utime.h> User time header file. The access and modification time structures and the utime() prototype are defined.

include/asm -- Architecture related header file subdirectory

- これらのヘッダーファイルは、主にCPUアーキテクチャに密接に関連するデータ構造、マクロ関数、変数などを定義しています。

- <asm/io.h> Io header file. Defines the function that operates on the io port in the form of a macro's embedded assembler.
- <asm/memory.h> Memory copy header file. Contains memcpy() embedded assembly macro functions.
- <asm/segment.h> Segment operation header file. An embedded assembly function is defined for segment register operations.
- <asm/system.h> System header file. An embedded assembly macro that defines or modifies descriptors/interrupt gates, etc. is defined.

include/linux -- Linux kernel dedicated header file subdirectory

- <linux/config.h> Kernel configuration header file. Define keyboard language and hard disk type (HD_TYPE) options.
- <linux/fdreg.h> Floppy disk file. Contains some definitions of floppy disk controller parameters.
- <linux/fs.h> File system header file. Define the file table structure (file, buffer_head, m_inode, etc.).
- <linux/hdreg.h> Hard disk parameter header file. Define access to the hard disk register port, status code, partition table and other information.
- <linux/head.h> Head header file. A simple structure for the segment descriptor is defined, along with several selector constants.
- <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly used functions of the kernel.
- <linux/math_emu.h> Coprocessor emulation header file. A coprocessor data structure and a floating point representation structure are defined.
- <linux/mm.h> Memory management header file. Contains page size definitions and some page release function prototypes.
- <linux/sched.h> The scheduler header file defines the task structure task_struct, the data of the initial

- タスク0、および記述子のパラメータ設定と取得に関するいくつかの組み込みアセンブリ関数マクロの記述。
- <linux/sys.h> The system calls the header file. Contains 72 system call C function handlers, starting with 'sys_.'
- <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial communication.

include/sys -- System-specific data structure subdirectory

- <sys/param.h> Parameter file. Some hardware-related parameter values are given.
- <sys/resource.h> Resource file. Contains information on the limits and utilization of system resources used by processes.
- <sys/stat.h> File status header file. Contains file or file system state structures stat{ } and constants.
- <sys/time.h> The timeval structure and the itimerval structure are defined.
- <sys/times.h> Defines the running time structure tms and the times() function prototype in the process.
- <sys/types.h> Type header file. The basic system data types are defined.
- <sys/utsname.h> System name structure header file.
- <sys/wait.h> Wait header file. Define the system call wait() core waitpid() and related constant symbols.

5.10.8 init -- kernel initialization program directory

このディレクトリには、main.cというファイルが1つだけあり、カーネルの初期化作業をすべて行います。その後、制御はユーザー モードに移行し、新しいプロセスを作成し、コンソールデバイス上でシェルプログラムを実行します。

- 5.10.9 プログラムはまず、マシンに搭載されている物理メモリの量に基づいてバッファメモリの容量を割り当てます。システムが使用する仮想ディスクも設定している場合は、バッファメモリの後ろにもスペースを残します。その後、最初のタスク（タスク0）を手動で作成したり、割り込み可能フラグを設定するなど、ハードウェアの初期化がすべて行われる。制御がカーネルの状態からユーザーの状態に移ると、システムはまずプロセス作成関数fork()を呼び出し、init()を実行するためのプロセスを作成します。この子プロセスの中で、システムはコンソール環境を設定し、シェルプログラムを実行するための子プロセスを生成します。

5.10.10 kernel -- kernel program main directory

linux/kernelディレクトリには、合計32のファイルと3つのサブディレクトリ（blk_drv, chr_drv, math）が含まれています。kernelディレクトリには12個のコードファイルとMakefileがあり、サブディレクトリのkernel/blk_drvには5個のファイル、kernel/chr_drvには6個のファイル、kernel/mathには9個のファイルがあります。

タスクを処理するすべてのプログラムはkernel/ディレクトリに格納されており、フォーク、イグジット、スケジューラ、一部のシステムコーラーなどが含まれます。また、割込み例外やトラップを処理する手続きサービスも含まれています。サブディレクトリには、get_hd_block や tty_write などの低レベルデバイスドライバが含まれています。これらのファイルに含まれるコードの呼び出し関係は複雑であるため、ここではファイル間の参照関係図を詳しく説明しません。しかし、図5-30に示すよ

うに、おおよその分類を行うことは可能です。

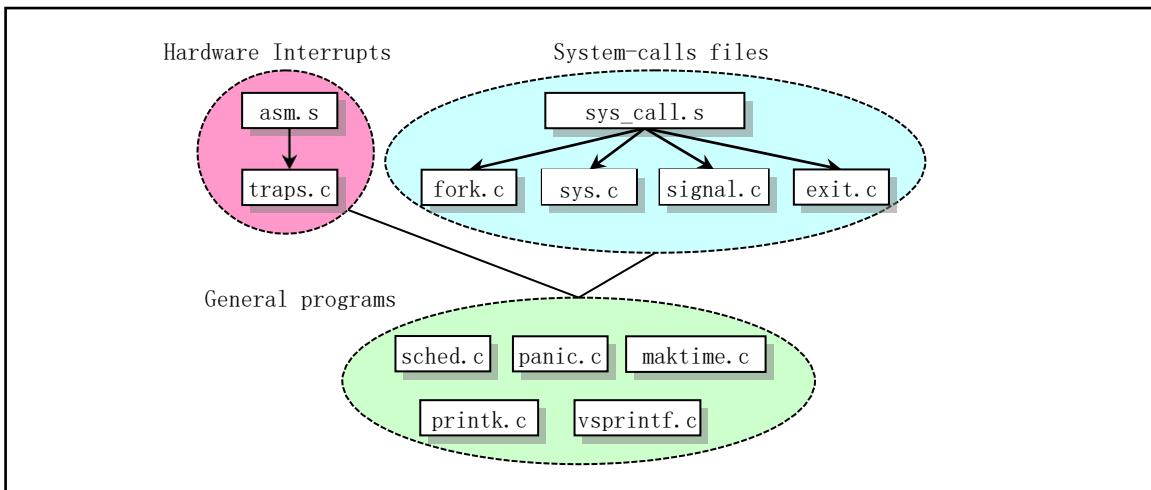


図5-30 各ファイルの呼び出し階層

- The asm.s program is used to handle interrupts caused by system hardware exceptions.
- The traps.c file is used for the actual handling of hardware exceptions. In each interrupt processing, the corresponding C language processing function in traps.c will be called separately.
- The exit.c program mainly includes system calls for processing process termination, including process release, session (process group) termination, and program exit processing functions, as well as system call functions such as killing processes, terminating processes, and suspending processes.
- The fork.c program gives two C language functions used in the sys_fork() system call: find_empty_process() and copy_process().
- The mktime.c program contains a time function mktime() used by the kernel to calculate the number of seconds from 0:00 on January 1, 1970 to the day of booting. It is only called once in init/main.c.
- The panic.c file contains a function panic() that displays kernel error messages and stops.
- The printk.c program contains a kernel-specific information display function printk().
- The sched.c program includes basic functions for scheduling (sleep_on, wakeup, schedule, etc.), as well as some simple system call functions. There are also several floppy disk operation functions related to timing.
- The signal.c program includes four system calls for signal processing and a function do_signal() that processes the signal in the corresponding interrupt handler.
- The sys.c program includes many system call functions, some of which are not yet implemented.
- The system_call.s program implements the interface processing of the Linux system call (int 0x80). The actual processing is contained in the corresponding C language functions of each system call. These processing functions are distributed throughout the Linux kernel code.
- The vsprintf.c program implements a string formatting function that is now included in the standard library functions.

kernel/blk_drv -- Block device driver subdirectory

通常、ユーザはファイルシステムを介してデバイスにアクセスし、デバイスドライバはファイルシステムへのアクセスインターフェースを実装します。ブロックデバイスを使用する場合、データのスループットが大きいブロックデバイス上のデータを効率的に利用するために、ユーザープロセスとブロックデバイスの間にキャッシュ機構が使用されます。ブロック・デバイス上のデータにアクセスする際、システムはまずブロック・デバイス上のデータをデータ・ブロックの形でキャッシュに読み込んだ後、ユーザーのバッファ空間にデータをコピーする。

blk_drvサブディレクトリには、合計4つのcファイルと1つのヘッダーファイルが含まれています。ヘッダーファイルblk.hは、以下を目的としています。

は、ブロックデバイスプログラムのため、Cファイルと一緒に置かれています。これらのドキュメントのおおよその関係を図5-31に示します。

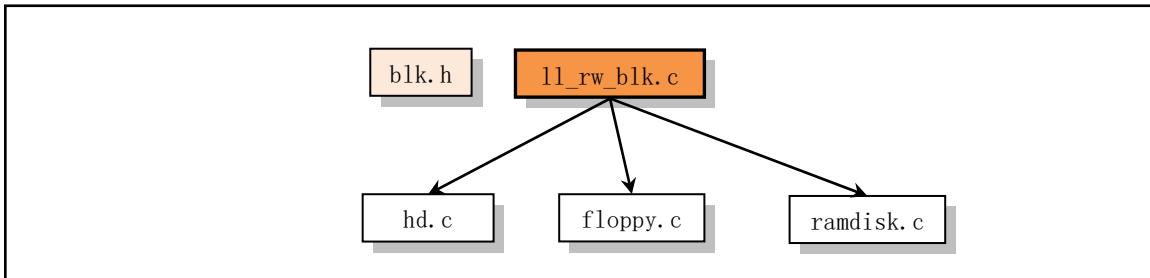


図5-31 blk_drvディレクトリ内のファイルの階層関係。

- blk.h block device special header file. The block device structure and data block request structure shared by several C programs are defined in it.
- The hd.c program mainly implements the underlying driver function for reading/writing hard disk data blocks, mainly the do_hd_request() function.
- The floppy.c program mainly implements the read/write drive function for floppy data blocks, mainly the do_fd_request() function.
- The ramdisk.c program is a memory virtual disk driver. The main function is do_rd_request(). A virtual disk device is a technology that uses physical memory to simulate an actual disk storage medium. Its main advantage is that it can greatly improve the speed of data access operations.
- The program in ll_rw_blk.c implements the low-level block device data read/write function ll_rw_block(). All other programs in the kernel use this function to access data from block devices. You will see that the function is called in many places where the block device data is accessed, especially in the cache file fs/buffer.c.

kernel/chr_drv -- Character device driver subdirectory

「キャラクタ・デバイス」サブディレクトリには、4つのC言語プログラムと2つのアセンブラー・ファイルが含まれています。これらのファイルは、シリアル・ポートのrs-232、シリアル・ターミナル、キーボード、コンソール・ターミナルのドライバを実現します。図5-32は、これらのファイル間のおおよその呼び出し階層です。

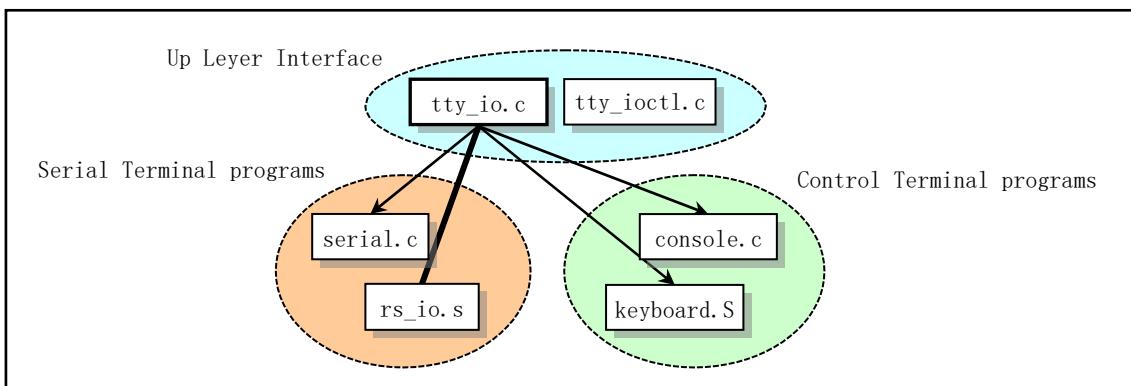


図5-32 キャラクター・デバイス・プログラムの関係を示す模式図

- The tty_io.c program contains the tty character device read function tty_read() and the write function tty_write(), which provides the upper layer access interface for the file system. It also includes the C

- 関数do_tty_interrupt()は、シリアル割り込み処理中に呼び出されます。この関数は、割り込みタイプがリードキャラクタの場合に呼び出されます。
- The console.c file mainly contains the console initialization program and the console write function con_write(), which is used by the tty device. It also includes an initialization setup program con_init() for screen display and keyboard interrupts.
- The rs_io.s program is used to implement interrupt handlers for two serial interfaces. The interrupt handler processes each of the four interrupt types retrieved from the interrupt identifier register (port 0x3fa or 0x2fa) and calls do_tty_interrupt() in the code that handles the interrupt type as a read character.
- Serial.c is used to initialize the asynchronous serial communication chip UART and set the interrupt vector of the two communication ports. Also included is the rs_write() function that tty uses to output to the serial port.
- The tty_ioctl.c program implements tty's io control interface function tty_ioctl(), and reads and writes to the termio(s) terminal io structure, and is called in the fs/iotl.c program that implements the system call sys_ioctl().
- The keyboard.S program mainly implements the keyboard interrupt handler procedure keyboard_interrupt.

kernel/math -- Coprocessor emulator subdirectory

- このサブディレクトリには、数学コプロセッサのエミュレーションハンドラファイルがあり、合計9個のCファイルがあります。マシンに数学コプロセッサが存在しない場合、CPUがコプロセッサ命令を実行すると、デバイスが存在しないという割り込みINT7が発生します。そのため、この割り込みを利用して、コプロセッサの機能をソフトウェアでシミュレートすることができます。これらのプログラムは、CPUのハードウェアと密接に関係していますが、それ以外のカーネルの実装とはほとんど関係ありません。プログラムに含まれる関連知識は、マルチプロセッサのプログラミングやアセンブリ・ディスアセンブルなどのシステムレベルのプログラムを実装する際に非常に役立ちます。

- The math_emulate.c program is the main program of coprocessor emulation, which implements the device not exist exception handler, floating point instruction emulation main function do_emu() and other auxiliary functions.
- The error.c program is used to process the error signal sent by the coprocessor. Its main function is math_error().
- The ea.c program is used to calculate the effective address used by the operands in the instruction when simulating floating-point instructions.
- The convert.c program is used to implement the data type conversion operation between the user data format and the temporary real number format in the simulation calculation process.
- The add.c program implements the addition in the simulation process and performs the conversion between the mantissa symbolization and the non-symbolization.
- The compare.c program is used to simulate the size of two temporary real numbers in the coprocessor compare accumulator.
- The get_put.c program implements access to data in the user's memory.
- The mul.c program is used to simulate multiply instructions in the coprocessor.
- The div.c program is used to simulate the division of the coprocessor.

5.10.11 lib -- Kernel library directory

通常のユーザープログラムとは異なり、カーネルコードは標準Cライブラリや他のライブラリコードを使用することができません。また、メイン

その理由は、完全なC関数ライブラリは大きく、実装が複雑だからです。そのため、カーネルソースには特別なlib/ディレクトリがあり、カーネルが使用する必要のあるいくつかの関数を提供しています。カーネル関数ライブラリは、ユーザーモード（プロセス0、1）で実行されるカーネル初期化プログラムinit/main.cの呼び出しサポートに使用されます。これは、通常の静的ライブラリの実装とまったく同じです。読者は一般的なlibcライブラリの基本構造を学ぶことができます。

5.10.12 lib/ディレクトリには12個のC言語ファイルがあります。tytso氏が作成したmalloc.cを除き、他のファイルは非常に短く、1~2行のコードしかないものもあります。これらのファイルは、いくつかのシステムコールのインターフェース関数を実装しています。これらのファイルには主に、終了関数_exit()、ファイルクローズ関数close(fd)、ファイルディスクリプターコピー関数dup()、ファイルオープン関数open()、ファイルライト関数write()、プログラム実行関数execve()などがある。メモリ確保関数malloc()、子プロセスの状態を待つ関数wait()、セッションを作成するシステムコールsetsid()、include/string.hで実装されているすべての文字列操作関数。

5.10.13 mm -- Memory Management Directory

このディレクトリには、3つのコードファイルが含まれています。主にアプリケーションのメインメモリ領域の使用を管理するために使用され、論理アドレスからリニアアドレス、リニアアドレスから物理メモリアドレスへのマッピング操作を実装しています。また、メモリページング機構により、主記憶領域の仮想メモリページと物理メモリページの間に対応関係が確立される。同時に、仮想記憶技術も実現している。

Linux カーネルは、フラグメント方式とページング方式の両方でメモリを扱います。1つ目は、80X86 4G の仮想アドレス空間を 64 セグメント（1 セグメントあたり 64MB）に分割する方法です。カーネルプログラムは最初のセグメントを占有し、その物理アドレスはセグメントのリニアアドレスと同じになります。その後、各タスクに使用するセグメントが割り当てられます。ページング機構を用いて、指定された物理メモリページをセグメントにマッピングし、フォークで作成された重複コピーを検出し、kop-on-write 機構を実行する。

page.sファイルには、メモリページアボート（int 14）ハンドラが含まれており、主にページフォールトによるページ例外や不正なアドレスへのアクセスによるページ保護の処理に使用されます。

memory.cプログラムには、メモリを初期化するmem_init()関数と、page.sのメモリ割り込み手続きから呼び出されるdo_no_page()関数とdo_wp_page()関数が含まれています。新規プロセスの作成やプロセスのコピー操作を行う際には、メモリハンドラを使用して管理用メモリ空間を確保します。

5.10.14 swap.cプログラムは、メインメモリーの物理ページと高速二次記憶装置（ハードディスク）の空間との間のページスワップを管理するために使用されます。メインメモリーの空き容量が足りない場合、一時的に使用していないメモリーページをハードディスクに保存することができます。ページフォールト例外が発生すると、まず要求されたページがハードディスクのスワップスペースにあるかどうかを

確認します。存在していれば、そのページはスワップスペースから直接メモリに読み込まれます。

5.10.15 tools -- Kernel Tools Directory

このディレクトリにある build.c プログラムは、完全なカーネルモジュールを構築するために使用されます。このプログラムは、Linuxディレクトリでコンパイルされたオブジェクトを、実行可能なカーネル・イメージ・ファイル・イメージに結合します。具体的な機能については、次の章で説明します。

5.11 The Kernel Code and User Programs

Linuxシステムでは、カーネルは2つの方法でユーザープログラムのサービスサポートを行うことができます。1つはシステムコールインターフェース、つまり int 0x80を呼び出す割り込みで、もう1つは開発の

環境ライブラリ関数、またはカーネルライブラリ関数です。ただし、カーネルライブラリ関数は、カーネルで作られたタスク0やタスク1でしか使われません。最終的にはシステムコールを呼び出すことに変わりはありません。したがって、実際には、カーネルは、すべてのユーザープログラムやプロセスに対して、システムコールの統一的なサービスインターフェースを提供しているにすぎません。lib/ ディレクトリにあるカーネル・ライブラリ関数コードの実装方法は、基本的に C 関数ライブラリ libc と同じです。カーネルのリソースを使用するために、図5-4に示すように、最終的にはインラインアセンブリコードによってカーネルシステムコールが呼び出されます。

システムコールは、主にシステムソフトウェアのプログラミングや、ライブラリ機能の実装に用いられる。一般的なユーザーが開発したプログラムは、libcなどのライブラリの関数を呼び出してカーネルリソースにアクセスする。これらのライブラリに含まれる関数やリソースは、しばしばアプリケーション・プログラミング・インターフェース (API) と呼ばれます。これは、アプリケーションが使用する標準的なプログラミング・インターフェースのセットを定義するものである。これらのライブラリの関数を呼び出すことにより、アプリケーションコードは、ファイルやデバイスへのアクセスの開閉、科学的計算の実行、エラー処理、グループやユーザーID番号などのシステム情報へのアクセスなど、さまざまな共通タスクを実行することができる。

UNIX系OSでは、POSIX規格に基づいたAPIインターフェースが最もよく使われており、Linuxも例外ではない。APIとシステムコールの違いは、POSIXなどのアプリケーション・インターフェース規格を実装するために、APIがシステムコールに対応している場合と、複数のシステムコール関数によって実装されている場合があることだ。もちろん、API関数の中には、システムコールを使う必要のないもの、つまり、カーネルが提供するサービスを使わないものもある。したがって、関数ライブラリは、POSIX規格を実装したメインのインターフェースとみなすことができます。アプリケーションは、システムコールとの関係を気にしません。2つのOSが提供するシステムコールにどれだけの違いがあっても、同じAPI規格への準拠を提供していれば、アプリケーションはこれらのOS間で移植可能です。

システムコールは、カーネルと外界との間のインターフェースの最上位に位置します。カーネルでは、各システムコールはマクロとして実装されていることが多く、シリアル番号を持っています (include/unistd.hヘッダファイルで定義されています)。アプリケーションは、システムコールを直接使用してはいけません。そうしないと、プログラムの移植性が悪くなります。そのため、現在のLinux Standard Base (LSB) をはじめとする多くの標準規格では、アプリケーションがシステムコールのマクロに直接アクセスすることを推奨していません。システムコールに関するドキュメントは、Linuxオペレーティングシステムのオンラインマニュアルのパート2に記載されています。

ライブラリファイルには一般的に、高度な機能を実行するためにC言語では提供されていない入出力関数や文字列操作関数などのユーザーレベルの関数が含まれている。ライブラリ関数の中には、システムコールを拡張しただけのものもあります。例えば、標準I/Oライブラリ関数のopenとfcloseは、システムコールのopenとcloseと同様の機能を、より高いレベルで提供しています。この場合、通常はシステムコールの方がライブラリ関数よりも若干性能が良いのですが、ライブラリ関数の方がより多くの機能を提供し、より多くのエラーを検出することができます。システムが提供するライブラリ関数は、オペレーティングシステムのオンラインマニュアルのセクション3に記載されています。

5.12 linux/Makefile

5.12.1 このセクションから、カーネルのソースコードファイルのアノテーションを開始します。まず、Linuxディレクトリで最初に出会うMakefileというファイルについてコメントします。以降のセクションも同様の記述構造で注釈をつけていきます。

5.12.2 Function Description

Makefileは、プログラムのコンパイル時のバッチファイルに相当します。これは、デフォルトのコンパイル設定

は、実行時にユーティリティプログラムmakeの入力ファイルとなります。Makefileのあるディレクトリでmakeコマンドを入力するだけで、Makefileの設定に従って、コンパイラやリンカーを呼び出し、ソースコードやターゲットコードファイルをコンパイル、リンク、インストールします。

makeユーティリティーは、複数のソースファイルを含むプログラムパッケージの中から、再コンパイルが必要なファイルを自動的に判断し、それらのプログラムファイルをコンパイルするコマンドを発行します。そこで、makeを使う前に、Makefileというテキストファイルを書いておく必要がある。このファイルには、パッケージ全体のプログラムの関係を記述し、更新が必要なファイルごとに具体的な制御コマンドを与える。一般的に、実行可能なターゲットは、コンパイラによって作成されたオブジェクトファイルに基づいて更新されます。適切なMakefileを書いておけば、プログラムパッケージ内のいくつかのソースコードファイルを変更するたびに、必要なすべての再コンパイル作業を行うことができます。makeツールは、Makefileファイルとコードファイルの最終修正時刻を使って、どのファイルを更新する必要があるかを判断します。更新が必要な各ファイルに対して、Makefileの情報に基づいて適切なコマンドを発行します。Makefileの中で「#」で始まる行はコメント行です。ファイルの先頭にある「=」代入文は、いくつかのパラメータやコマンドの略語を定義しています。

カーネルディレクトリにあるこのMakefileの主な機能は、独立してコンパイルされたツール/ディレクトリ内のビルド実行ファイルを使って、すべてのカーネルコンパイルコードを最終的に接続し、実行可能なカーネルイメージファイルにマージするよう、makeプログラムに指示することです。具体的なプロセスは (1) 8086アセンブラーを使って、boot/にあるbootsect.sとsetup.sをコンパイルし、それぞれのオブジェクトを生成します。(2) その他のプログラムをGNUコンパイラgcc/gasを用いてソースコードをコンパイルし、リンクしてモジュールシステムを生成する。(3) 最後に、ビルドツールを使って、3つのパートをカーネルイメージファイルのイメージにまとめます。

ビルドツールは、tools/build.cソースファイルからコンパイルされたスタンドアローンの実行ファイルです。カーネルコードにはコンパイル、リンクされません。カーネルイメージファイルを構築する過程でのツールとしてのみ使用されます。基本的なコンパイル・リンク・組み合わせの構造を図5-33に示します。

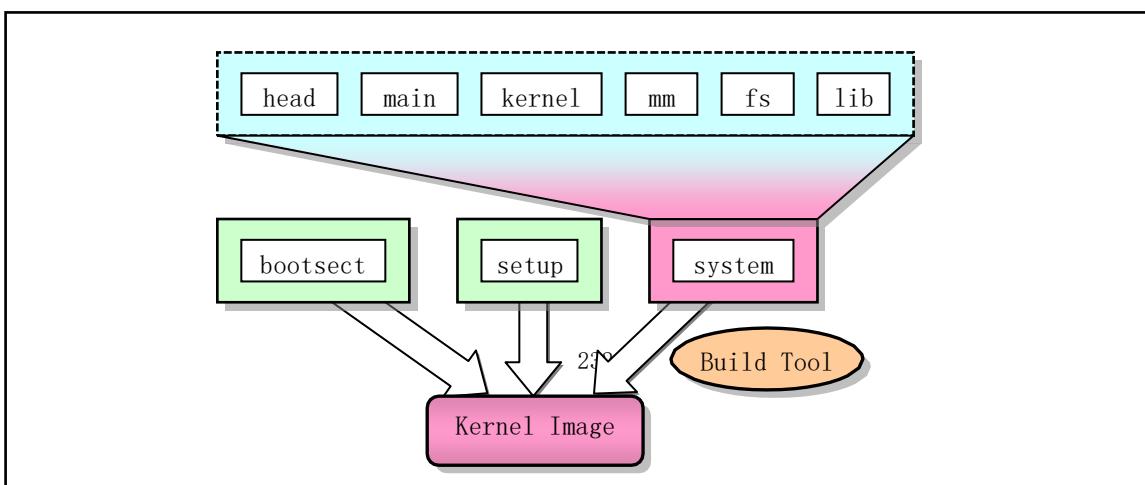


図 5-33 カーネルのコンパイルリンク/結合構造

Linuxカーネルのソースコードでは、tools/、init/、boot/の各ディレクトリを除いて、それぞれのサブディレクトリに対応するMakefileが含まれており、その構造は同一です。紙面の都合上、本書ではMakefileの注釈は1つだけとなっています。プログラム5-1は、このファイルの詳細なコメントです。

なお、ソースファイル中の行番号のある文がオリジナルの文、行番号のない文が本書の著者のコメントです。

5.12.3 Program Annotation

プログラム 5-1 linux/Makefile

```

1 #
2 # ラムディスクのデバイスが欲しい場合は、3# ブロック単位
3 # のサイズになるように定義します。
4 #
5 # RAMディスク装置を使用する場合は、ブロックのサイズを定義します。デフォルトの
6 # RAMDISK # は、ここでは定義されていません（コメントアウトされています）。そうでない場合、
7 # gccは、オプション
8 # '-DRAMDISK=512'は、以下の13行目を
9 # 参照してください。5 ramdisk = #-
10 DRAMDISK=512
11
12 # gldのオプションです。-s すべてのシンボル情報を出力ファイルで省略する。-x すべてのローカル
13 # シンボルを削除する。# -M リンクマップが必要であることを示す。リンクマップとは、# リンカが
14 # 生成するメモリアドレスマップで、コードブロックがメモリに# 読み込まれる位置情報を記載した
15 # ものである。
16 LDFLAGS =-s -x -M
17
18 # gcc は GNU コンパイラです。UNIX系スクリプトでは、識別子を参照する際には
19 # 識別子の前に符号を付け、識別子を括弧で囲む必要があります。
20 CC =gcc $(RAMDISK)
21
22 # gccのオプション。-Wall」はすべての警告を表示し、「O」はコードを最適化します。「-f flag」
23 # はフラグを指定します。# ここで、-fstrength-reduceはループ文を最適化するために使われる；-
24 # fomit-frame-pointer
25
26 # フレームポインタを必要としない関数のために、# フレームポインタをレジスタに残さないこ
27 # を示します。これにより、関数内でのフレームポインタの操作やメンテナンスが不要になります。
28 # -fcombin-regは、コピーする命令をコンパイラが結合することを示します。
29
30 # あるレジスタから別のレジスタへ -mstring-insnsはLinusがgccに追加するオプションです。
31 # gcc-1.40では386CPUの文字列命令を使って文字列構造をコピーする際に# 使用されていますが、削
32 # 除することもできます。
33
34 # CFLAGS =-Wall -O -fstrength-reduce -fomit-frame-pointer \
35 # -fcombine-reg -mstring-insns
36
37 # cppはgccのプリプロセッサで、マクロの置換、条件付きコンパイル、# 'include'で指定された
38 # ファイルのインクルードなどに使用されます。# 指定されたファイルのインクルードに使用さ
39 # れます。#で始まるすべての行は
40
41 # プリプロセッサによって# 处理されます。define' で定義されたすべてのマクロは、その定義に# 置
42 # き換えられます。if'、'#ifdef'、'#ifndef'、'#endif'などのすべての条件付き行は、指定された範囲の

```

文を含めるかどうかを # 判断するために使用されます。

オプション 「-nostdinc -Iinclude」 は、標準的なヘッダファイルのディレクトリを検索しない、
つまり、 /usr/include/ のファイルを使用せず、「-I」 オプションで指定されたディレクトリを #
使用するか、カレントディレクトリを検索することを意味します。

```
16 CPP      =cpp -nostdinc -Iinclude  
17  
18 #  
19 # ROOT_DEV は、イメージを作成する際のデフォルトのルートデバイス  
# を指定します。20 # FLOPPY, /dev/xxxx, または空のいずれかで、その場合は  
21 # デフォルトの /dev/hd6 が 'build' で使われます。  
22 #  
# ここで /dev/hd6 は、2番目のハードディスクの最初のパーティションに相当します。これは
```

```

23 # リーナスがLinuxカーネルを開発する際に、ルートファイルシステムが置かれる場所。#
24 /dev/hd2 は 1 番目のハードディスクの 2 番目のパーティションで、スワップパーティションと
25 して使用されます。
26 ROOT_DEV=/dev/hd6
25 SWAP_DEV=/dev/hd2
25
26 # 以下は、kernel, mm, fs ディレクトリに生成されたオブジェクトファイルです。参照を容易にする
27 ため、ここでは ARCHIVES 識別子で表しています。
27 ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o

# ブロックデバイスとキャラクターデバイスのライブラリファイルです。'.a' はアーカイブ # ファ
28 イル、つまり実行可能なバイナリコードの集合体を含むライブラリファイルであることを示しま
29 す。
28 # サブルーチンで、通常はGNUのarプログラムによって生成されます。Arは、アーカイ
29 ブファイルからファイルを作成、変更、抽出するための # GNUバイナリファイルツールです。
29 DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a
28 MATH    =kernel/math/math.a
29 LIBS    =lib/lib.a          # A generic library compiled from files in lib/ dir.
30

# ここでは、昔ながらのmakeの暗黙の接尾辞ルールを紹介します。この行は、すべての'.c' ファ
30 イルを '.s' アセンブリにコンパイルするように # 指示しています。このルールでは
# 文章全体の意味は、gccがCFLAGSで指定されたオプションを使用して、# include/ dirにあるヘッ
30 ダーファイルのみを使用し、コンパイルを停止(-S)して、# アセンブリを生成するということです。
各入力Cファイルに対応する # ファイルを作成します。デフォルトでは、オリジナルのCファイルの
サフィックス「.c」を「.s」に # 置き換えたものがアセンブリファイルとして出力されます。'-o'は
出力ファイルを示す。
# '$*.s' (または '$@') は自動オブジェクト変数、'$<' は最初の # 前提条件を表しており、ここ
30 では条件 '*.*' を満たすファイルを紹介します。
# 以下の3つのルールは、それぞれ異なる要件に対応しています。ターゲットが.sの場合
.c.s:

```

```
31      $(CC) $(CFLAGS) \
32      -nostdinc -Iinclude -S -o $*.s $<
33 .s.o:
34      $(AS) -c -o $*.o $<
35 .c.o:
36      $(CC) $(CFLAGS) \
37      -nostdinc -Iinclude -c -o $*.o $<
38
39
```

以下の'all'はMakefileの最上位のターゲットを作成することを意味し、ここでは # イメージファイルを作成します。これは、起動ディスクのイメージファイルです。これをフロッピーディスクに書き込めば

フロッピーディスクを使ってLinuxシステムを起動します。LinuxでフロッピーディスクにImageを書き込むためのコマンドは # 46行目を参照してください。DOSではソフトウェアrawrite.exeが使用できます。

40 all: Image

41

ターゲット(Image)は、コロンの後ろにある4つの要素によって生成され、それらはbootsect # とセットアップファイルはboot/ に、システムファイルとビルドファイルは tools/ にあります。43~44行目が実行されたコマンドです。43行目は、bootsect、setup、systemの各ファイルが組み立てられたことを示しています。

42 # カーネルイメージファイルを作成し、\$(ROOT_DEV)デバイスを tools ディレクトリのビルドユーティリティーで作成します。# 45行目のsyncコマンドは、バッファが直ちにディスクに書き込み、スーパーブロックを更新することを強制します。

43 Image: boot/bootsect boot/setup tools/system tools/build

44 tools/build boot/bootsect boot/setup tools/system \$(ROOT_DEV) \

45 \$(SWAP_DEV) > Image

```

46      sync
46
47 # ddは、標準的なcmdで、ファイルをコピーし、オプションに応じて # 変換、フォーマットを行います。if=入力ファイル、of=出力されるファイルです。ここで、/dev/PS0 は最初のフロッピーディスクドライブ(デバイスファイル)を # 参照しています。現在の Linux システムでは、/dev/fd0 を使用します。
48 disk: Image
49      dd bs=8192 if=Image of=/dev/PS0
49
50 tools/build: tools/build.c          # Create executable build tool.
51      $(CC) $(CFLAGS) \
52      -o tools/build tools/build.c
53
54 boot/head.o: boot/head.s          # Generate head.o object using the .s.o rule.
55
56 # コロンの右にある要素でツール/システムが生成されることを示します。57~62行目は、システムオブジェクトを生成するためのコマンドです。最後の「> System.map」は、# gldがリンク情報を System.map ファイルにリダイレクトする必要があることを # 意味しています。
57 tools/system:   boot/head.o init/main.o \
58                  $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)
59      $(LD) $(LDFLAGS) boot/head.o init/main.o \
60      $(ARCHIVES) \
61      $(DRIVERS) \
62      $(MATH) \
63      $(LIBS) \
64      -o tools/system > System.map
64
65 # アーカイブファイル math.a は 64 行目のcmds でビルドされています: cd into kernel/math/; run make.
66 kernel/math/math.a:
67      (cd kernel/math; make)
67
68 kernel/blk_drv/blk_drv.a:          # Create block driver archive file.
69      (cd kernel/blk_drv; make)
70
71 kernel/chr_drv/chr_drv.a:          # Character driver archive file.
72      (cd kernel/chr_drv; make)
73
74 kernel/kernel.o:                  # kernel object file.
75      (cd kernel; make)
76
77 mm/mm.o:                          # Memory management object file.
78      (cd mm; make)
79
80 fs/fs.o:                           # File system object file.
81      (cd fs; make)
82
83 lib/lib.a:                         # Internal lib.a
84      (cd lib; make)
85
86 # setup.s ファイルをコンパイルして、8086アセンブラーとリンクを使ってsetup.oを生成します。オプション # -s は、ターゲットファイルのシンボル情報を削除することを意味します。

```

```
87 boot/setup: boot/setup.s  
88         $(AS86) -o boot/setup.o boot/setup.s
```

```

89      $(LD86) -s -o boot/setup boot/setup.o
89
90 # プリプロセッサを実行して、*.Sファイルのマクロを置き換え、対応する*.sファイルを生成します。
91 boot/setup.s:    boot/setup.S include/linux/config.h
92         $(CPP) -traditional boot/setup.S -o boot/setup.s
92
93 boot/bootsect.s:       boot/bootsect.S include/linux/config.h
94         $(CPP) -traditional boot/bootsect.S -o boot/bootsect.s
95
96 boot/bootsect:   boot/bootsect.s
97         $(AS86) -o boot/bootsect.o boot/bootsect.s
98         $(LD86) -s -o boot/bootsect boot/bootsect.o
99
100 # 'make clean' が実行されると、101--107行目のコマンドが実行され、生成された#ファイルが
    #すべて削除されます。rm」はファイル削除コマンドで、オプションの「-f」は存在しないファイルを#無視し、削除メッセージを表示しないことを意味しています。
101 clean:
102     rm -f Image System.map tmp_make core boot/bootsect boot/setup \
        boot/bootsect.s boot/setup.s
103     rm -f init/*.o tools/system tools/build boot/*.o
104     (cd mm;make clean)
105     (cd fs;make clean)
106     (cd kernel;make clean)
107     (cd lib;make clean)
108
108 # このルールは、まず上記のクリーンなルールを実行し、次にlinux/ディレクトリを#圧縮して
    「backup.Z」という圧縮ファイルを生成します。'tar cf - linux' は、linux/ディレクトリでtar アーカイバを#実行することを意味します。'-cf' はアーカイブファイルを作成することを意味します。
    '|compress -' は、tarプログラムの実行を#パイプラインで圧縮機compressに渡すことを意味する。
109 #の操作を行い、コンプレッサーの出力をbackup.Zファイルとして保存します。
110 backup: clean
111     (cd .. ; tar cf - linux | compress - > backup.Z)
112     sync
113
114 dep:
    # このゴールまたはルールは、ファイル間の依存関係を生成するために使用されます。これらの依存
    関係は、対象となるオブジェクトを再構築する必要があるかどうかを#makeコマンドが判断するた
    めに作成されます。#例えば、ヘッダファイルが変更された場合、makeは生成された依存関係を介し
    て#そのヘッダファイルに関連する全ての*.cファイルを再コンパイルすることができます。具体的な
    方法は以下の通りです。
    #文字列エディタsedを使ってMakefile(ここではこのファイル)を処理すると、Makefileの'###'
    Dependencies'行以降のすべての行を#削除するように出力されます、つまり、delete
    #ファイルの122行目から最後までのすべての行を、一時ファイルtmp_make(114行とも#いう)と
    して生成する。#そして、指定されたディレクトリ(init/)にある各Cファイル(実際には、#
    main.cという1つのファイルのみ)に対して、gccの前処理を行います。フラグ「-M」はプリプロセッ
    サ#cppに各オブジェクトファイルの関連性を記述したルールを出力するように指示し、#これらの
    ルールはmake構文に準拠しています。各ソースファイルに対して、プリプロセッサは、結果が

```

115

は、対応するソースファイルのターゲットファイル名とその依存関係、つまり # ソースファイルに含まれるすべてのヘッダーファイルのリストを加えたものです。次に、前処理の結果を一時ファイル tmp_make に追加し、最後にその一時ファイルを # 新しいMakefile にコピーします。# 115行目の'\$\$i' は、実際には'\$(\$i)'です。ここで'\$i'は、この文章の前にあるシェル変数 '# \$i' の値である。

116 sed '/#\#\# Dependencies/q' < Makefile > tmp_make117 (for i in init/*.c;do echo -n "init/"\$(CPP) -M \$\$i;done) >> tmp_make

```
116      cp tmp_make Makefile
117      (cd fs; make dep)
118      (cd kernel; make dep)
119      (cd mm; make dep)
120
121  ### Dependencies:
122 init/main.o : init/main.c include/unistd.h include/sys/stat.h \
123   include/sys/types.h include/sys/time.h include/time.h include/sys/times.h \
124   include/sys/utsname.h include/sys/param.h include/sys/resource.h \
125   include/utime.h include/linux/tty.h include/termios.h include/linux/sched.h \
126   include/linux/head.h include/linux/fs.h include/linux/mm.h \
127   include/linux/kernel.h include/signal.h include/asm/system.h \
128   include/asm/io.h include/stddef.h include/stdarg.h include/fcntl.h \
129   include/string.h
```

5.13 Summary

This chapter provides an overview of the kernel modes and architecture of the early Linux operating systems. First, the Linux 0.12 kernel usage and management memory methods, kernel state stack and user state stack settings and usage methods, interrupt mechanism, system clock timing, and process creation, scheduling, and termination methods are given. Then according to the directory structure of the source code, the basic functions and hierarchical relationships of the code files in each subdirectory are introduced in detail. It also explains the target file format used by Linux 0.12. Finally, we started with the makefile in the Linux kernel home directory and began to comment on the kernel source code.

This chapter can be seen as a summary of the important information about the Linux 0.12 kernel, so it can be used as a reference for reading subsequent chapters. From the bootloader in the next chapter, we formally began to annotate the source code of the kernel.