Ohgami’s Commentary on OS5

大神祐真　著

2017-04-09 版　　へにゃぺんて 発行

はじめに

この本について

この度は、本書を手にとっていただきありがとうございます。この本は、フルスクラッチの自作 OS("OS5") のソースコードを印刷して本にしてみたいという思いから始まりました。そして、せっかくなら解説も付けようと考え、思い出したのが、この本のタイトルの元ネタである"Lions Commentary on UNIX"です。この本では、OS5 の各ソースコードに続いて、コメンタリーとして説明を書いています。

本書の構成

OS5 を構成するブートローダー・カーネル・ユーザーランドについて、ソースコード毎にコメンタリーを記載しています。コメンタリーではではソースコードの概要をざっくりと書いています。主に OS5 独特の実装や、ソースコードだけでは意図をくみ取る事が難しい箇所について説明しています。ソースコードのすべてを説明している訳ではないです。(ソースコードが主役なので、基本的にはソースコードを見てみてください。お見苦しい箇所もあるかと思いますが。。。) また、「コラム」として適宜、改めて読んでいて気づいた点等を、ざっくばらんに書いています。

目次

はじめに **2**

この本について 2

本書の構成 2

第 1 章 OS5 について **7**

1.1 OS5 とは 7

"OS5"という名前の由来 9

1.2 ソースディレクトリ構成 9

1.3 メモリマップ 11

第 2 章 ブートローダー **12**

2.1 MBR 部 12

boot/Makefile 12

boot/boot.ld 13

boot/boot.s 14

第 3 章 カーネル **22**

3.1 初期化 22

kernel/Makefile 22

kernel/sys.ld 23

kernel/include/asm/cpu.h 25

kernel/sys.S 25

kernel/init.c 30

kernel/include/kernel.h 32

3.2 割り込み/例外 33

kernel/include/intr.h 33

kernel/intr.c 34

kernel/include/excp.h 35
kernel/excp.c 36
3.3 メモリ管理 37
kernel/include/memory.h 37
kernel/memory.c 38
3.4 タスク管理 44
kernel/include/sched.h 44
kernel/sched.c 44
kernel/include/kern_task.h 54
kernel/kern_task_init.c 55
kernel/include/task.h 57
kernel/task.c 58
3.5 ファイルシステム 64
kernel/include/fs.h 64
kernel/fs.c 64
3.6 システムコール 70
kernel/include/syscall.h 70
kernel/syscall.c 71
3.7 デバイスドライバ 75
kernel/include/cpu.h 75
kernel/cpu.c 77
kernel/include/io_port.h 77
kernel/include/console_io.h 78
kernel/console_io.c 79
kernel/include/timer.h 87
kernel/timer.c 88
3.8 ライブラリ 89
kernel/include/stddef.h 89
kernel/include/common.h 89
kernel/common.c 90
kernel/include/lock.h 91
kernel/lock.c 91
kernel/include/list.h 92
kernel/include/queue.h 92
kernel/queue.c 93

kernel/include/debug.h 94
kernel/debug.c 94

第 4 章 ユーザーランド 95

4.1 共通部分 96
apps/Makefile 96
apps/app.ld 97
apps/include/app.h 97
4.2 0shell 98
apps/0shell/Makefile 98
apps/0shell/0shell.c 98
4.3 uptime 104
apps/uptime/Makefile 104
apps/uptime/uptime.c 105
4.4 whoareyou 106
apps/whoareyou/Makefile 106
apps/whoareyou/whoareyou.c 106
4.5 libkernel 107
apps/libkernel/Makefile 107
apps/include/kernel.h 108
apps/libkernel/libkernel.c 108
4.6 libcommon 109
apps/libcommon/Makefile 109
apps/include/common.h 109
apps/libcommon/libcommon.c 109
4.7 libconsole 110
apps/libconsole/Makefile 110
apps/include/console.h 110
apps/libconsole/libconsole.c 111
4.8 libstring 112
apps/libstring/Makefile 112
apps/include/string.h 112
apps/libstring/libstring.c 113

第 5 章 ツール類 116

5.1 ファイルシステム作成 116

tools/make_os5_fs.sh 116
5.2 ブログ記事作成支援 117
tools/cap.sh 117

おわりに **118**

参考情報 **119**

第 1 章

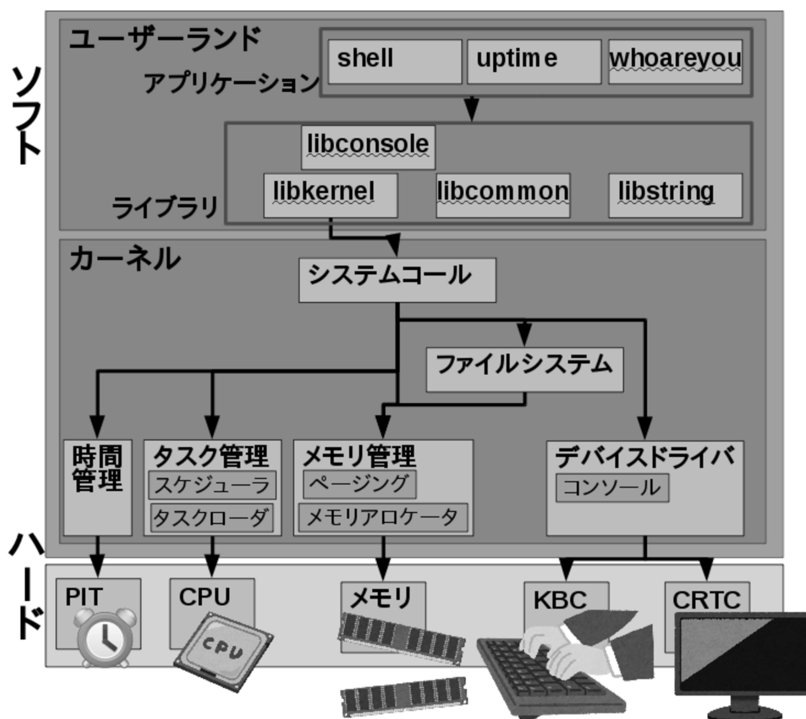
OS5 について

1.1　OS5 とは

OS5 は、勉強のためにフルスクラッチで作成している OS です。ブートローダー・カーネル・ユーザーランド (カーネル上で動作するアプリケーション群) を含んでいます。現在の主なスペックは以下の通りです。

- x86 の QEMU(qemu-system-i386) で動作
- 一番作りこめているのはカーネルで、主な機能を一通り備えている
  - 時間管理
  - タスク管理 (スケジューラ、タスクローダ)
  - メモリ管理 (ページング、ヒープアロケータ)
  - デバイスドライバ
  - システムコール
  - ファイルシステム
- ユーザーランド (アプリケーション郡) は、まだカーネルの動作確認程度
  - CUI のみ
  - コマンドライン引数対応
  - 静的ライブラリ対応 (libkernel、libcommon、libstring、libconsole)
- RAM 上で動作
  - ファイルシステムも RAM 上
  - コンベンショナルメモリ (640KB) に収まっている
- ビルド環境は Make と GCC
  - アセンブラは GAS
- スモールスタートとして、シンプルに (割りきって) 実装している
  - 一通りの機能を備えたカーネルが 2000 行程度

現状、一番作りこめているのはカーネルで、カーネルが持つべき主要な機能を一通り揃えています。カーネルが持つ機能について図 1.1 に示します。



▲図 1.1　OS5 カーネルの構成

スモールスタートとして、シンプルに割り切って実装するようにしています。図 1.1 に示す一通りの機能を備えたカーネルが 2000 行程度で、ブートローダーとユーザーランドを合わせても 3000 行程度で実現できています (そのため、「本にしてみよう」と思い立ちました)。

機能実装の節目でブログ記事を公開しており、記事公開の日付をバージョン番号にしています。これまでのリリース情報など、OS5 の情報は以下のページにまとめています。なお、本書で対象とするバージョンは 2017 年 1 月現在の最新リリースである blog-20170123 です[*1]。

- OS5 のまとめ：`http://funlinux.org/os5/`

[*1] ただし、1 行 80 文字に収まるよう修正しています。

また、ソースコードは Git で管理しており、GitHub 上にリポジトリがあります。

- GitHub：`http://github.com/cupnes/os5/`

blog-20170123 時点で、ソースコード行数 (アセンブラと C の行数の総和) は、表 1.1 の通りとなりました。

▼表 1.1　ソースコード行数

項目	行数
ブートローダー	328
カーネル	2117
ユーザーランド	541

"OS5"という名前の由来

"OS5"という名前は、「OS 作り 5 回目」を示しています (過去に"OS"、"OS2"、"OS3"、"OS4"がありました)。これまで、Linux 0.01 のソースコードから分岐してみたり、本を参考にしてみたりしていましたが、どのやり方もしっくりこない思いがあり、途中でやめてしまいました。ただし、「OS を作りたい」という思いは変わらなかったため、色々とやり方を変えて試していました。5 回目にして、「いい加減、フルスクラッチでできるのではないか」と思い、2015 年 3 月頃からフルスクラッチで"OS5"の作成を始めました。今になって思えば、ソースコードや本などで「元となる OS」が存在した場合、それを改造しても「元の OS + 自分のパッチ」であり、「自分の OS」とは言いづらい事がダメだったのかなと思います。

1.2　ソースディレクトリ構成

OS5 のソースディレクトリ構成は以下の通りです。

- boot/
  - ブートローダーのソースディレクトリ
- kernel/
  - カーネルのソースディレクトリ

- apps/
  - ユーザーランドのソースディレクトリ
- tools/
  - 各種ツールのソースディレクトリ
- doc/
  - ドキュメントのディレクトリ
- Makefile
  - 全体を管理する Makefile

以降の章では、boot、kernel、apps、tools の各ディレクトリのソースコードを説明します。

ソースディレクトリ直下の Makefile については、章を割くほどでもないので、この章で掲載します。内容は以下の通りです。

▼**リスト** 1.1 Makefile

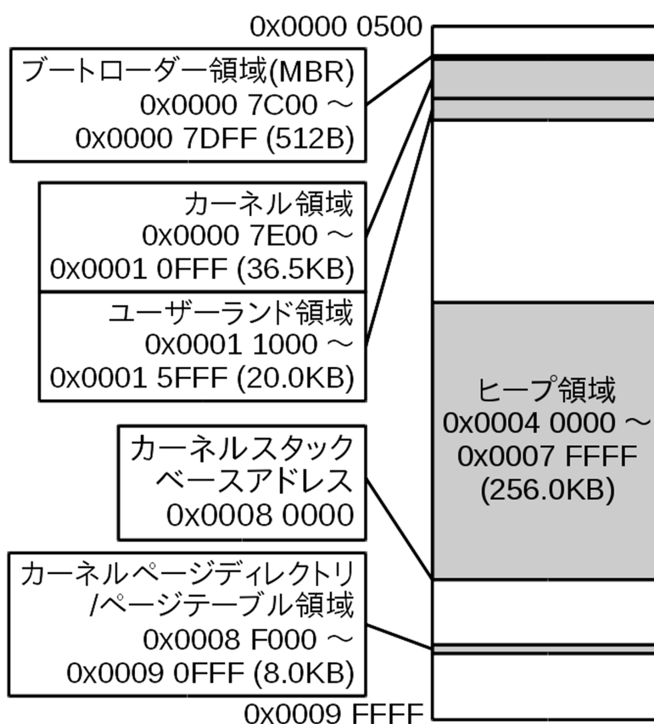
```
 1: all: fd.img
 2: 
 3: fd.img: boot/boot.bin kernel/kernel.bin apps/apps.img
 4:     cat $+ > $@
 5: 
 6: boot/boot.bin:
 7:     make -C boot
 8: 
 9: kernel/kernel.bin:
10:     make -C kernel
11: 
12: apps/apps.img:
13:     make -C apps
14: 
15: doc:
16:     make -C doc
17: 
18: clean:
19:     make -C boot clean
20:     make -C kernel clean
21:     make -C apps clean
22:     make -C doc clean
23:     rm -f *~ *.o *.bin *.dat *.img *.map
24: 
25: run: fd.img
26:     qemu-system-i386 -fda $<
27: 
28: .PHONY: boot/boot.bin kernel/kernel.bin apps/apps.img doc clean
```

Makefile の書き方として、特に独特なことはしていません。この Makefile は、下位の boot ディレクトリや kernel ディレクトリ、apps ディレクトリの Makefile を呼び出します。そして、下位の各ディレクトリの生成物 (boot/boot.bin、kernel/kernel.bin、

apps/apps.img) を、cat で結合し、fd.img(OS5 のフロッピーディスクイメージ) を生成します (Makefile の 3 行目)。

1.3　メモリマップ

メモリマップを図 1.2 に示します。図 1.2 は、実際にカーネルやユーザーランド (ファイルシステム) 等を RAM 上にどのように配置するかを示しています。0x0000 0500〜0x0009 FFFF(640KB) は「コンベンショナルメモリ (伝統的メモリ領域)」と呼ばれる領域で、x86 CPU において古くから RAM に割り当たっていたメモリ領域です。



▲図 1.2　物理アドレス空間のメモリマップ

なお、カーネルやアプリケーションは「仮想アドレス空間」というアドレス空間で動作します。物理アドレス空間と分けることで、「アプリケーションは必ず同じアドレスから始まる」といったことや、「アプリケーションはカーネルの空間にアクセスさせない」といったことを実現しています。詳しくは kernel/memory.c で説明します。

第 2 章

ブートローダー

PC の電源を入れると、「BIOS」と呼ばれるマザーボードに元から書き込まれているソフトウェアが、起動ディスクの第 1 セクタ (MBR と呼ばれる) を RAM へロードし、CPU に実行させます。1 セクタは 512 バイトです。ブートローダー・カーネル・ユーザーランドが 512 バイトに収まるはずもないので、この 512 バイトのプログラムで適宜 RAM へロードする必要があります。なお、OS5 のブートローダーは 512 バイトに収まっているので、OS5 の場合は 512 バイトのプログラムだけでブートローダーは完結しています[*1]。

ブートローダーでは、カーネルとユーザーランドの RAM へのロードと、CPU 設定を行っています。CPU 設定は、割り込みや各種ディスクリプタテーブル[*2]の設定などを行っています。CPU 設定で特に重要なのは、リアルモード (16 ビットのモード) からプロテクトモード (32 ビットのモード) への移行です。CPU のモードをプロテクトモードへ移行させた後、カーネルの先頭アドレスへジャンプします。

2.1　MBR 部

boot/Makefile

▼**リスト** 2.1　boot/Makefile

[*1] GRUB 等の高機能なブートローダーは 512 バイトに収まらないので、ブートローダーを 512 バイトの初段と、そこからロードされる 2 段目という形で多段ブートしていたりします。

[*2] x86 CPU には「セグメント」という単位でメモリを分割して管理するメモリ管理方法があり、そのための設定です。各セグメントの設定は「ディスクリプタ」というデータ構造で行います。「ディスクリプタ」の「テーブル」なので「ディスクリプタテーブル」です。

```
 1: .s.o:
 2:     as --32 -o $@ $<
 3:
 4: boot.bin: boot.o
 5:     ld -m elf_i386 -o $@ $< -T boot.ld -Map boot.map
 6:
 7: boot.o: boot.s
 8:
 9: clean:
10:     rm -f *~ *.o *.bin *.dat *.img *.map
11:
12: .PHONY: clean
```

ブートローダーの Makefile です。現状、ブートローダーはすべてアセンブラ書いています (512 バイトと小さく、C で処理を書くほどのことをしていない為)。ソースファイルは boot.s だけです。Makefile としても、この単一のアセンブラファイルを as(GNU アセンブラ) でアセンブルし、ld(GNU リンカ) でリンク、を行っています。

コンパイルを行う環境は x86_64 の環境なので、x86_32 のバイナリを生成するために、as コマンドでは"--32"オプションを、ld コマンドでは"-m elf_i386"のオプションをつけています。なお、恥ずかしながら、当初はこれらのオプションを知りませんでした。そのため、開発環境を x86_32 から x86_64 へ変えたときは、x86_32 の仮想環境を用意していました。x86_32 のバイナリを生成するこれらのオプションは、パッチを作成された方がいて、マージさせていただいたものです。(これがマージさせてもらった初めてで、今のところ唯一のパッチです。)

あと、Makefile の書き方ですが、`.s.o:`という書き方は、`boot.o:  boot.s`のように個々のターゲットを記述しなければならないので、あんまり良くないなぁと思います。`%.o:  %.s`の書き方で汎用的なターゲット指定を書いておけば、`boot.o:  boot.s`の記述を消せますね。なお、本書で説明はしていないのですが、ドキュメントディレクトリ[*3]の Makefile では'%' の書き方で汎用的なターゲット指定を行っています。(ブートローダー等の古い Makefile も直すべきで、自分のタスクリストには入っていたのですが、優先度: 低で放置していました。言い訳ですが。。。)

boot/boot.ld

▼リスト 2.2　boot/boot.ld

[*3] OS5 のソースディレクトリ直下の doc ディレクトリ

```
 1: OUTPUT_FORMAT("binary");
 2:
 3: SECTIONS
 4: {
 5:     .text   : {*(.text)}
 6:     .rodata : {
 7:             *(.strings)
 8:             *(.rodata)
 9:             *(.rodata.*)
10:     }
11:     .data   : {*(.data)}
12:     .bss    : {*(.bss)}
13:
14:     . = 510;
15:     .sign   : {SHORT(0xaa55)}
16: }
```

ブートローダーのリンカスクリプトです。

MBR の 512 バイトの先頭へジャンプしてくるので、text セクションが先頭に来るように並べています。

また、BIOS が「起動可能なディスクか否か」の判別として、「MBR の末尾 2 バイトが"0xaa55"であるか」をチェックしているので、リンカスクリプトで先頭から 510 バイト目に"0xaa55"を配置するようにしています。

boot/boot.s

▼リスト 2.3　boot/boot.s

```
 1:     .code16
 2:
 3:     .text
 4:     cli
 5:
 6:     movw    $0x07c0, %ax
 7:     movw    %ax, %ds
 8:     movw    $0x0000, %ax
 9:     movw    %ax, %ss
10:     movw    $0x1000, %sp
11:
12:     /* ビデオモード設定（画面クリア）*/
13:     movw    $0x0003, %ax
14:     int     $0x10
15:
16:     movw    $msg_welcome, %si
17:     call    print_msg
18:
19:     movw    $msg_now_loading, %si
20:     call    print_msg
21:
```

```
22:     /* ディスクサービス セクタ読み込み
23:      * int 0x13, AH=0x02
24:      * 入力
25:      * - AL: 読み込むセクタ数
26:      * - CH: トラックの下位 8 ビット
27:      * - CL(上位 2 ビット): トラックの上位 2 ビット
28:      * - CL(下位 6 ビット): セクタを指定
29:      * - DH: ヘッド番号を指定
30:      * - DL: セクタを読み込むドライブ番号を指定
31:      * - ES: 読み込み先のセグメント指定
32:      * - BX: 読み込み先のオフセットアドレス指定
33:      * 出力
34:      * - EFLAGS の CF ビット: 0=成功, 1=失敗
35:      * - AH: エラーコード (0x00=成功)
36:      * - AL: 読み込んだセクタ数
37:      * 備考
38:      * - トラック番号: 0 始まり
39:      * - ヘッド番号: 0 始まり
40:      * - セクタ番号: 1 始まり
41:      * - セクタ数/トラック: 2HD は 18
42:      * - セクタ 18 の次は、別トラック (裏面) へ
43:      * - 64KB 境界を超えて読みだすことはできない
44:      *   (その際は、2 回に分ける)
45:      */
46:
47:     /* トラック 0, ヘッド 0, セクタ 2 以降
48:      * src: トラック 0, ヘッド 0 のセクタ 2 以降
49:      *      (17 セクタ = 8704 バイト = 0x2200 バイト)
50:      * dst: 0x0000 7e00 ~ 0x0000 bfff
51:      */
52: load_track0_head0:
53:     movw    $0x0000, %ax
54:     movw    %ax, %es
55:     movw    $0x7e00, %bx
56:     movw    $0x0000, %dx
57:     movw    $0x0002, %cx
58:     movw    $0x0211, %ax
59:     int     $0x13
60:     jc      load_track0_head0
61:
62:     /* トラック 0, ヘッド 1, 全セクタ
63:      * src: トラック 0, ヘッド 1 の全セクタ
64:      *      (18 セクタ = 9216 バイト = 0x2400 バイト)
65:      * dst: 0x0000 a000 ~ 0x0000 c3ff
66:      */
67: load_track0_head1:
68:     movw    $0x0000, %ax
69:     movw    %ax, %es
70:     movw    $0xa000, %bx
71:     movw    $0x0100, %dx
72:     movw    $0x0001, %cx
73:     movw    $0x0212, %ax
74:     int     $0x13
75:     jc      load_track0_head1
76:
77:     /* トラック 1, ヘッド 0, 全セクタ
78:      * src: トラック 1, ヘッド 0 の全セクタ
79:      *      (18 セクタ = 9216 バイト = 0x2400 バイト)
```

```
80:      * dst: 0x0000 c400 〜 0x0000 e7ff
81:      */
82: load_track1_head0:
83:     movw    $0x0000, %ax
84:     movw    %ax, %es
85:     movw    $0xc400, %bx
86:     movw    $0x0000, %dx
87:     movw    $0x0101, %cx
88:     movw    $0x0212, %ax
89:     int     $0x13
90:     jc      load_track1_head0
91:
92:     /* トラック 1, ヘッド 1, セクタ 1 - 12
93:      * src: トラック 1, ヘッド 1 の 12 セクタ
94:      *      (12 セクタ = 6144 バイト = 0x1800 バイト)
95:      * dst: 0x0000 e800 〜 0x0000 ffff
96:      */
97: load_track1_head1_1:
98:     movw    $0x0000, %ax
99:     movw    %ax, %es
100:    movw    $0xe800, %bx
101:    movw    $0x0100, %dx
102:    movw    $0x0101, %cx
103:    movw    $0x020c, %ax
104:    int     $0x13
105:    jc      load_track1_head1_1
106:
107:    /* トラック 1, ヘッド 1, セクタ 13 - 18
108:     * src: トラック 1, ヘッド 1 の 6 セクタ
109:     *      (6 セクタ = 3072 バイト = 0xc00 バイト)
110:     * dst: 0x0001 0000 〜 0x0001 0bff
111:     */
112: load_track1_head1_2:
113:    movw    $0x1000, %ax
114:    movw    %ax, %es
115:    movw    $0x0000, %bx
116:    movw    $0x0100, %dx
117:    movw    $0x010d, %cx
118:    movw    $0x0206, %ax
119:    int     $0x13
120:    jc      load_track1_head1_2
121:
122:    /* トラック 2, ヘッド 0, 全セクタ
123:     * src: トラック 2, ヘッド 0 の全セクタ
124:     *      (18 セクタ = 9216 バイト = 0x2400 バイト)
125:     * dst: 0x0001 0c00 〜 0x0001 2fff
126:     */
127: load_track2_head0:
128:    movw    $0x1000, %ax
129:    movw    %ax, %es
130:    movw    $0x0c00, %bx
131:    movw    $0x0000, %dx
132:    movw    $0x0201, %cx
133:    movw    $0x0212, %ax
134:    int     $0x13
135:    jc      load_track2_head0
136:
137:    /* トラック 2, ヘッド 1, 全セクタ
```

```
138:     * src: トラック 2, ヘッド 1 の全セクタ
139:     *      (18 セクタ = 9216 バイト = 0x2400 バイト)
140:     * dst: 0x0001 3000 〜 0x0001 53ff
141:     */
142: load_track2_head1:
143:    movw    $0x1000, %ax
144:    movw    %ax, %es
145:    movw    $0x3000, %bx
146:    movw    $0x0100, %dx
147:    movw    $0x0201, %cx
148:    movw    $0x0212, %ax
149:    int     $0x13
150:    jc      load_track2_head1
151:
152:    /* トラック 3, ヘッド 0, 全セクタ
153:     * src: トラック 3, ヘッド 0 の全セクタ
154:     *      (18 セクタ = 9216 バイト = 0x2400 バイト)
155:     * dst: 0x0001 5400 〜 0x0001 77ff
156:     */
157: load_track3_head0:
158:    movw    $0x1000, %ax
159:    movw    %ax, %es
160:    movw    $0x5400, %bx
161:    movw    $0x0000, %dx
162:    movw    $0x0301, %cx
163:    movw    $0x0212, %ax
164:    int     $0x13
165:    jc      load_track3_head0
166:
167:    /* トラック 3, ヘッド 1, 全セクタ
168:     * src: トラック 3, ヘッド 1 の全セクタ
169:     *      (18 セクタ = 9216 バイト = 0x2400 バイト)
170:     * dst: 0x0001 7800 〜 0x0001 9bff
171:    */
172: load_track3_head1:
173:    movw    $0x1000, %ax
174:    movw    %ax, %es
175:    movw    $0x7800, %bx
176:    movw    $0x0100, %dx
177:    movw    $0x0301, %cx
178:    movw    $0x0212, %ax
179:    int     $0x13
180:    jc      load_track3_head1
181:
182:    /* トラック 4, ヘッド 0, 全セクタ
183:     * src: トラック 4, ヘッド 0 の全セクタ
184:     *      (18 セクタ = 9216 バイト = 0x2400 バイト)
185:     * dst: 0x0001 9c00 〜 0x0001 bfff
186:    */
187: load_track4_head0:
188:    movw    $0x1000, %ax
189:    movw    %ax, %es
190:    movw    $0x9c00, %bx
191:    movw    $0x0000, %dx
192:    movw    $0x0401, %cx
193:    movw    $0x0212, %ax
194:    int     $0x13
195:    jc      load_track4_head0
```

```
196:
197:    /* トラック 4, ヘッド 1, 全セクタ
198:     * src: トラック 4, ヘッド 1 の全セクタ
199:     *      (18 セクタ = 9216 バイト = 0x2400 バイト)
200:     * dst: 0x0001 c000 〜 0x0001 e3ff
201:    */
202: load_track4_head1:
203:    movw    $0x1000, %ax
204:    movw    %ax, %es
205:    movw    $0xc000, %bx
206:    movw    $0x0100, %dx
207:    movw    $0x0401, %cx
208:    movw    $0x0212, %ax
209:    int     $0x13
210:    jc      load_track4_head1
211:
212:    /* トラック 5, ヘッド 0, セクタ 1 - 14
213:     * src: トラック 5, ヘッド 0 のセクタ 1〜14
214:     *      (14 セクタ = 7168 バイト = 0x1c00 バイト)
215:     * dst: 0x0001 e400 〜 0x0001 ffff
216:    */
217: load_track5_head0_1:
218:    movw    $0x1000, %ax
219:    movw    %ax, %es
220:    movw    $0xe400, %bx
221:    movw    $0x0000, %dx
222:    movw    $0x0501, %cx
223:    movw    $0x020e, %ax
224:    int     $0x13
225:    jc      load_track5_head0_1
226:
227:    movw    $msg_completed, %si
228:    call    print_msg
229:
230:    /* マスタ PIC の初期化 */
231:    movb    $0x10, %al
232:    outb    %al, $0x20 /* ICW1 */
233:    movb    $0x00, %al
234:    outb    %al, $0x21 /* ICW2 */
235:    movb    $0x04, %al
236:    outb    %al, $0x21 /* ICW3 */
237:    movb    $0x01, %al
238:    outb    %al, $0x21 /* ICW4 */
239:    movb    $0xff, %al
240:    outb    %al, $0x21 /* OCW1 */
241:
242:    /* スレーブ PIC の初期化 */
243:    movb    $0x10, %al
244:    outb    %al, $0xa0 /* ICW1 */
245:    movb    $0x00, %al
246:    outb    %al, $0xa1 /* ICW2 */
247:    movb    $0x02, %al
248:    outb    %al, $0xa1 /* ICW3 */
249:    movb    $0x01, %al
250:    outb    %al, $0xa1 /* ICW4 */
251:    movb    $0xff, %al
252:    outb    %al, $0xa1 /* OCW1 */
253:
254:    call    waitkbdout
```

```
255:    movb    $0xd1, %al
256:    outb    %al, $0x64
257:    call    waitkbdout
258:    movb    $0xdf, %al
259:    outb    %al, $0x60
260:    call    waitkbdout
261:
262:    /* GDT を 0x0009 0000 から配置 */
263:    movw    $0x07c0, %ax        /* src */
264:    movw    %ax, %ds
265:    movw    $gdt, %si
266:    movw    $0x9000, %ax        /* dst */
267:    movw    %ax, %es
268:    subw    %di, %di
269:    movw    $12, %cx    /* words */
270:    rep     movsw
271:
272:    movw    $0x07c0, %ax
273:    movw    %ax, %ds
274:    lgdtw   gdt_descr
275:
276:    movw    $0x0001, %ax
277:    lmsw    %ax
278:
279:    movw    $2*8, %ax
280:    movw    %ax, %ds
281:    movw    %ax, %es
282:    movw    %ax, %fs
283:    movw    %ax, %gs
284:    movw    %ax, %ss
285:
286:    ljmp    $8, $0x7e00
287:
288: print_msg:
289:    lodsb
290:    andb    %al, %al
291:    jz      print_msg_ret
292:    movb    $0xe, %ah
293:    movw    $7, %bx
294:    int     $0x10
295:    jmp     print_msg
296: print_msg_ret:
297:    ret
298: waitkbdout:
299:    inb     $0x60, %al
300:    inb     $0x64, %al
301:    andb    $0x02, %al
302:    jnz     waitkbdout
303:    ret
304:
305:    .data
306: gdt_descr:
307:    .word   3*8-1
308:    .word   0x0000, 0x09
309:    /* .word gdt,0x07c0
310:     * と設定しても、
311:     * GDTR には、ベースアドレスが
312:     * 0x00c0 [gdt の場所]
```

```
313:      * と読み込まれてしまう
314:      */
315: gdt:
316:     .quad   0x0000000000000000      /* NULL descriptor */
317:     .quad   0x00cf9a000000ffff      /* 4GB(r-x:Code) */
318:     .quad   0x00cf92000000ffff      /* 4GB(rw-:Data) */
319:
320: msg_welcome:
321:     .ascii  "Welcome to OS5!\r\n"
322:     .byte   0
323: msg_now_loading:
324:     .ascii  "Now Loading ... "
325:     .byte   0
326: msg_completed:
327:     .ascii  "Completed!\r\n"
328:     .byte   0
```

ブートローダー本体のソースコードです。アセンブラは、時間がたつと真っ先に読めなくなる箇所なので、少し詳しく説明します。

このソースコードで行っている処理の流れは以下の通りです。

1. CPU 設定 (1〜21 行目)
2. FD から RAM へロード (22〜228 行目)
3. PIC(Programmable Interrupt Controller) 初期化 (230〜252 行目)
4. CPU 設定 (254〜284 行目)
5. カーネルへジャンプ (286 行目)

「1. CPU 設定」について、まず、".code16"が 16bit 命令のアセンブラであることを示しています。"cli"ではすべての割り込みを無効化し、ブートローダーの処理中の、まだハードウェアの設定を行っていない段階で割り込みを受け付けないようにしています。6〜10 行目の"movw"命令の辺りではセグメントの設定とスタックポインタの設定をしています。OS5 ではブートローダーの段階ではスタック領域のベースを 0x0000 1000 に設定しています。そして、12〜14 行目では BIOS の機能を使ってビデオモードの設定と画面クリアを行っています。BIOS の機能は「1. 汎用レジスタにパラメータをセット」、「2. ソフトウェア割り込み」の流れです。ここではテキストモードを示す"0x03"を AX レジスタの下位 8 ビット (AL レジスタ) にセットし、画面モードに関する機能を呼び出す 0x10 のソフトウェア割り込みを実行しています。

「2. FD から RAM へロード」について、ブートローダーの行数の大半がこの処理です。24〜227 行目までの 203 行あり、全 328 行の内の 6 割程あります。見ればわかる通りですが、47〜60 行目のようなコード片を繰り返し並べています。64KB 境界をまたぐ場合を除き、1 つのコードブロックで 1 つのトラックをロードします。1 トラックずつなのは BIOS の機能でまとまってロードできるのが 1 トラックずつだったためです。ループ等を

使わずにコード片を何度も書いているのは、アセンブラの領域はあまり凝ったことをすると保守できなくなる (2〜3 か月後とかに見たとき、処理の流れを追えなくなる) 気がしたからです。FD からのロード処理は、ロードサイズを増やす等で後々に処理を修正する可能性があることが分かっていたので、自分にとって平易な書き方をしています。そのため、ここで使用しているディスクサービスの BIOS 命令については、説明をコメントで書いています。

「PIC(Programmable Interrupt Controller) 初期化」では、マスタとスレーブの PIC の設定で、すべての割り込みを無効化しています。PIC の IO レジスタの使い方は Intel 8259 のデータシートを確認してください。(あるいは、自作 OS 系のサイトや書籍などでも紹介しています。)

「4. CPU 設定」では、カーネルへジャンプする直前の CPU 設定を行っています。254〜260 行目はキーボードコントローラ (KBC) の設定です。waitkbdout では、キーボードコントローラのステータスがビジーを抜けるまで待っています。262〜270 行目では、0x7c00 XXXX にある GDT(グローバルディスクリプタテーブル)[*4]を 0x0009 0000 へコピーしています。なお、この GDT は、カーネルへロングジャンプするためにしか使いません (カーネル側では別途 GDT を設定します)。わざわざ 0x0009 0000 へコピーしている理由は、lgdtw 命令で GDT をロードする際に指定する GDT のセグメントセレクタに 0x07c0 を指定できなかったためです (何か間違えているのか、どうなのか、不明)。その後、lgdtw 命令で GDTR へ gst_descr の内容をロードします (272〜274 行目)。

ここまででプロテクトモード (32 ビットのモード) へ移行するための準備が完了です。276〜277 行目では MSR の最下位ビットに 1 をセットし、プロテクトモードへ移行させています。その後、279〜284 行目でセグメントレジスタへセグメントセレクタを設定しています (ここでは、プロテクトモードでのセグメントセレクタを設定しています)。そして、カーネルの先頭アドレスへジャンプします (286 行目)。OS5 では、カーネルは 0x0000 7e00 から配置するように決めています。物理アドレス空間のメモリマップは、図 1.2 を参照してください。

[*4] x86 CPU で使用する色々なデータ構造の先頭アドレスと長さを管理するテーブルです。セグメントというメモリ管理方式のディスクリプタや、タスク管理の TSS(タスクステートストラクチャ) を登録します。

第 3 章

カーネル

ブートローダーからジャンプしてくると、カーネルの初期化処理が実行されます。カーネルの各機能の初期化を行った後は、カーネルは割り込み駆動で動作し、何もしない時は hlt 命令で CPU を寝させるようにしています。

カーネルの各機能の実装については、2016 年 4 月に発表させていただいた勉強会のスライドにまとめています。各機能の実装について、図を使って説明していますので、興味があれば見てみてください。

- `http://www.slideshare.net/yarakawa/2000x86`

3.1　初期化

kernel/Makefile

▼**リスト** 3.1　kernel/Makefile

```
 1: CFLAGS      =       -Wall -Wextra
 2: CFLAGS      +=      -nostdinc -nostdlib -fno-builtin -c
 3: CFLAGS      +=      -Iinclude
 4: CFLAGS      +=      -m32
 5:
 6: .S.o:
 7:     gcc $(CFLAGS) -o $@ $<
 8: .c.o:
 9:     gcc $(CFLAGS) -o $@ $<
10:
11: kernel.bin: sys.o cpu.o intr.o excp.o memory.o sched.o fs.o task.o  \
12:     syscall.o lock.o timer.o console_io.o queue.o common.o          \
13:     debug.o init.o kern_task_init.o
14:     ld -m elf_i386 -o $@ $+ -Map System.map -s -T sys.ld -x
15:
```

```
16: sys.o: sys.S
17: cpu.o: cpu.c
18: intr.o: intr.c
19: excp.o: excp.c
20: memory.o: memory.c
21: sched.o: sched.c
22: fs.o: fs.c
23: task.o: task.c
24: syscall.o: syscall.c
25: lock.o: lock.c
26: timer.o: timer.c
27: console_io.o: console_io.c
28: queue.o: queue.c
29: common.o: common.c
30: debug.o: debug.c
31: init.o: init.c
32: kern_task_init.o: kern_task_init.c
33:
34: clean:
35:     rm -f *~ *.o *.bin *.dat *.img *.map
36:
37: .PHONY: clean
```

カーネルをコンパイルするルールを記載した Makefile です。OS5 のカーネルは Linux 等と同じくモノリシックカーネルです。そのため、コンパイルが完了すると単一のバイナリができあがります。(このバイナリを上位の Makefile でブートローダーやユーザーランドと結合します。)

アセンブラのソースコードファイルの拡張子が、ブートローダーでは boot/boot.s と小文字の's' でしたが、`sys.o: sys.S`のターゲット指定の通り、大文字の'S' になっています。また、アセンブラのビルド時に使用するコマンドも、ブートローダーでは as コマンドでしたが、ここでは gcc を使用しています。これは、プリプロセスで展開されるマクロを記述するためです (kernel/sys.S で`#include`マクロを使っています)。

ブートローダーの Makefile でも書きましたが、`.S.o`や`.c.o`といった書き方をしている (6〜9 行目) ため、各ターゲットを `sys.o: sys.S`のように指定せねばならず、冗長ですね。

kernel/sys.ld

▼リスト 3.2　kernel/sys.ld

```
1: OUTPUT_FORMAT("binary");
2:
3: SECTIONS
4: {
5:     . = 0x7e00;
```

```
 6:     .text   : {*(.text)}
 7:     .rodata : {
 8:             *(.strings)
 9:             *(.rodata)
10:             *(.rodata.*)
11:     }
12:     .data   : {*(.data)}
13:     .bss    : {*(.bss)}
14:
15:     . = 0x7e00 + 0x91fe;
16:     .sign   : {SHORT(0xbeef)}
17: }
```

boot/boot.s の説明で、「OS5 では、カーネルは 0x0000 7e00 から配置するように決めています。」と書いていましたが、「決めている」のがこのファイルです。5 行目の`. = 0x7e00;`で、カーネルのバイナリは 0x0000 7e00 から配置されることを指定しています。そして、5 行目に続いて`.text: {*(.text)}`と記載しており、カーネルのバイナリの先頭にテキスト領域を配置することを指定しています。

また、15 行目の`. = 0x7e00 + 0x91fe;`で、カーネル領域のサイズを決めています。上位の Makefile では「ブートローダー・カーネル・ユーザーランドのバイナリを単に cat で連結している」旨を説明しました。カーネルサイズの変化によってユーザーランドの開始アドレスが変化することが無いよう、カーネルサイズを固定しています。0x91fe は 10 進数で 37374 です。2 バイトのマジックナンバー (0xbeef) を含め、37376 バイト (36.5KB) がカーネルで使用できる最大サイズです。また、0x7e00 + 0x91fe + 2(マジックナンバー) = 0x11000 なので、ユーザーランドの開始アドレスは 0x0001 1000 です。

マジックナンバーと Hexspeak

マジックナンバーに 0xbeef(肉) 等、16 進数で表せる英単語を使用すると、16 進数でメモリダンプしたときに確認できて便利です。このような表記法は"Hexspeak" と呼ばれ、0xBAADF00D("bad food") が「Microsoft Windows の LocalAlloc 関数の第一引数に LMEM_FIXED を渡して呼び出してメモリを確保した場合に、ヒープに確保されたメモリが初期化されていないことを表す値として使用されている。」[a]等、他にも様々なものがあります。詳しくは、Wikipedia 等を参照してみると面白いです。

[a] Hexspeak - Wikipedia: https://ja.wikipedia.org/wiki/Hexspeak

kernel/include/asm/cpu.h

▼リスト 3.3　kernel/include/asm/cpu.h

```
 1: #ifndef _ASM_CPU_H_
 2: #define _ASM_CPU_H_
 3:
 4: #define GDT_SIZE   16
 5:
 6: #endif /* _ASM_CPU_H_ */
```

後述する kernel/sys.S で#includeされるヘッダーファイルです。#define1 つだけなので先に紹介してしまいます。

ここでは、GDT のサイズを定義しています。GDT は C 言語からも参照することがあるので、ヘッダーファイルに分離しています。

kernel/sys.S

▼リスト 3.4　kernel/sys.S

```
 1: #include <asm/cpu.h>
 2:
 3:     .code32
 4:
 5:     .text
 6:
 7:     .global kern_init, idt, gdt, keyboard_handler, timer_handler
 8:     .global exception_handler, divide_error_handler, debug_handler
 9:     .global nmi_handler, breakpoint_handler, overflow_handler
10:     .global bound_range_exceeded_handler, invalid_opcode_handler
11:     .global device_not_available_handler, double_fault_handler
12:     .global coprocessor_segment_overrun_handler, invalid_tss_handler
13:     .global segment_not_present_handler, stack_fault_handler
14:     .global general_protection_handler, page_fault_handler
15:     .global x87_fpu_floating_point_error_handler, alignment_check_handler
16:     .global machine_check_handler, simd_floating_point_handler
17:     .global virtualization_handler, syscall_handler
18:
19:     movl    $0x00080000, %esp
20:
21:     lgdt    gdt_descr
22:
23:     lea     ignore_int, %edx
24:     movl    $0x00080000, %eax
25:     movw    %dx, %ax
26:     movw    $0x8E00, %dx
27:     lea     idt, %edi
```

```
28:     mov     $256, %ecx
29: rp_sidt:
30:     movl    %eax, (%edi)
31:     movl    %edx, 4(%edi)
32:     addl    $8, %edi
33:     dec     %ecx
34:     jne     rp_sidt
35:     lidt    idt_descr
36:
37:     pushl   $0
38:     pushl   $0
39:     pushl   $0
40:     pushl   $end
41:     pushl   $kern_init
42:     ret
43:
44: end:
45:     jmp     end
46:
47: keyboard_handler:
48:     pushal
49:     call    do_ir_keyboard
50:     popal
51:     iret
52:
53: timer_handler:
54:     pushal
55:     call    do_ir_timer
56:     popal
57:     iret
58:
59: exception_handler:
60:     popl    excp_error_code
61:     pushal
62:     call    do_exception
63:     popal
64:     iret
65:
66: /* interrupt 0 (#DE) */
67: divide_error_handler:
68:     jmp     divide_error_handler
69:     iret
70:
71: /* interrupt 1 (#DB) */
72: debug_handler:
73:     jmp     debug_handler
74:     iret
75:
76: /* interrupt 2 */
77: nmi_handler:
78:     jmp     nmi_handler
79:     iret
80:
81: /* interrupt 3 (#BP) */
82: breakpoint_handler:
83:     jmp     breakpoint_handler
84:     iret
85:
```

```
86: /* interrupt 4 (#OF) */
87: overflow_handler:
88:     jmp     overflow_handler
89:     iret
90:
91: /* interrupt 5 (#BR) */
92: bound_range_exceeded_handler:
93:     jmp     bound_range_exceeded_handler
94:     iret
95:
96: /* interrupt 6 (#UD) */
97: invalid_opcode_handler:
98:     jmp     invalid_opcode_handler
99:     iret
100:
101: /* interrupt 7 (#NM) */
102: device_not_available_handler:
103:    jmp     device_not_available_handler
104:    iret
105:
106: /* interrupt 8 (#DF) */
107: double_fault_handler:
108:    jmp     double_fault_handler
109:    iret
110:
111: /* interrupt 9 */
112: coprocessor_segment_overrun_handler:
113:    jmp     coprocessor_segment_overrun_handler
114:    iret
115:
116: /* interrupt 10 (#TS) */
117: invalid_tss_handler:
118:    jmp     invalid_tss_handler
119:    iret
120:
121: /* interrupt 11 (#NP) */
122: segment_not_present_handler:
123:    jmp     segment_not_present_handler
124:    iret
125:
126: /* interrupt 12 (#SS) */
127: stack_fault_handler:
128:    jmp     stack_fault_handler
129:    iret
130:
131: /* interrupt 13 (#GP) */
132: general_protection_handler:
133:    jmp     general_protection_handler
134:    iret
135:
136: /* interrupt 14 (#PF) */
137: page_fault_handler:
138:    popl    excp_error_code
139:    pushal
140:    movl    %cr2, %eax
141:    pushl   %eax
142:    pushl   excp_error_code
143:    call    do_page_fault
```

```
144:    popl    %eax
145:    popl    %eax
146:    popal
147:    iret
148:
149: /* interrupt 16 (#MF) */
150: x87_fpu_floating_point_error_handler:
151:    jmp     x87_fpu_floating_point_error_handler
152:    iret
153:
154: /* interrupt 17 (#AC) */
155: alignment_check_handler:
156:    jmp     alignment_check_handler
157:    iret
158:
159: /* interrupt 18 (#MC) */
160: machine_check_handler:
161:    jmp     machine_check_handler
162:    iret
163:
164: /* interrupt 19 (#XM) */
165: simd_floating_point_handler:
166:    jmp     simd_floating_point_handler
167:    iret
168:
169: /* interrupt 20 (#VE) */
170: virtualization_handler:
171:    jmp     virtualization_handler
172:    iret
173:
174: /* interrupt 128 */
175: syscall_handler:
176:    pushl   %esp
177:    pushl   %ebp
178:    pushl   %esi
179:    pushl   %edi
180:    pushl   %edx
181:    pushl   %ecx
182:    pushl   %ebx
183:    pushl   %eax
184:    call    do_syscall
185:    popl    %ebx
186:    popl    %ebx
187:    popl    %ecx
188:    popl    %edx
189:    popl    %edi
190:    popl    %esi
191:    popl    %ebp
192:    popl    %esp
193:    iret
194:
195: ignore_int:
196:    iret
197:
198:    .data
199: idt_descr:
200:    .word   256*8-1                 /* idt contains 256 entries */
201:    .long   idt
202:
```

```
203: gdt_descr:
204:    .word   GDT_SIZE*8-1
205:    .long   gdt
206:
207:    .balign 8
208: idt:
209:    .fill   256, 8, 0       /* idt is uninitialized */
210:
211: gdt:
212:    .quad   0x0000000000000000      /* NULL descriptor */
213:    .quad   0x00cf9a000000ffff      /* 4GB(r-x:Code, DPL=0) */
214:    .quad   0x00cf92000000ffff      /* 4GB(rw-:Data, DPL=0) */
215:    .quad   0x00cffa000000ffff      /* 4GB(r-x:Code, DPL=3) */
216:    .quad   0x00cff2000000ffff      /* 4GB(rw-:Data, DPL=3) */
217:    .fill   GDT_SIZE-5, 8, 0
218:
219: excp_error_code:
220:    .long   0x00000000
```

カーネルのエントリ部分のソースコードです。 boot/boot.s 286 行目の `ljmp$8, $0x7e00`でジャンプする先はこのファイルの先頭です。3 行目に`.code32`と書かれている通り、ここからは 32 ビットの命令です。

まず、19 行目でスタックポインタを 0x00080000 に設定しています。そして、21 行目でカーネルとユーザーランド動作中に使用する GDT(グローバルディスクリプタテーブル) を設定しています。`gdt_descr`ラベルの内容は 203〜205 行目にあります。ここで定数 `GDT_SIZE`を使うために、asm/cpu.h を include しています。

23〜35 行目で割り込みハンドラの設定をしています。IDT の全 256 エントリを `ignore_int`ハンドラで初期化しています。`ignore_int`は 195〜196 行目にあり、内容は `iret` で return するだけです。なお、スタックポインタを表す 0x00080000 は定数化すべきですね。同じ値を 2 度も書いているし、`KERN_STACK_BASE`とかの定数名にしておけば、この値がカーネルのスタックのベースアドレスであると一目でわかります。

その後、37〜42 行目で、関数から return するときと同じようにスタックに値を積み、ret 命令で kern_init 関数 (kernel/init.c) へジャンプしています。スタックポインタの整合性さえ取れていれば、jmp 命令でもよさそうですが、C 言語の世界の関数呼び出しは、ret 命令で call 命令でジャンプし ret 命令で戻る、という流れなので、一部のアセンブラコードで違ったことをするより、C 言語の関数呼び出しの形式に統一しています。

以降は、個別の割り込みハンドラです。kernel/sys.S では割り込みハンドラの出入口部分のみで、メインの処理は C 言語で記述された関数を呼び出しています。システムコールのソフトウェア割り込み (128 番の割り込み) のみ、pushal/popal を使わず、汎用レジスタを一つずつ push/pop しているのは、EAX をシステムコールの戻り値に使っているからです。

kernel/init.c

▼リスト 3.5　kernel/init.c

```
 1: #include <stddef.h>
 2: #include <cpu.h>
 3: #include <intr.h>
 4: #include <excp.h>
 5: #include <memory.h>
 6: #include <console_io.h>
 7: #include <timer.h>
 8: #include <kernel.h>
 9: #include <syscall.h>
10: #include <task.h>
11: #include <fs.h>
12: #include <sched.h>
13: #include <kern_task.h>
14: #include <list.h>
15: #include <queue.h>
16: #include <common.h>
17:
18: int kern_init(void)
19: {
20:     extern unsigned char syscall_handler;
21:
22:     unsigned char mask;
23:     unsigned char i;
24:
25:     /* Setup console */
26:     cursor_pos.y += 2;
27:     update_cursor();
28:
29:     /* Setup exception handler */
30:     for (i = 0; i < EXCEPTION_MAX; i++)
31:             intr_set_handler(i, (unsigned int)&exception_handler);
32:     intr_set_handler(EXCP_NUM_DE, (unsigned int)&divide_error_handler);
33:     intr_set_handler(EXCP_NUM_DB, (unsigned int)&debug_handler);
34:     intr_set_handler(EXCP_NUM_NMI, (unsigned int)&nmi_handler);
35:     intr_set_handler(EXCP_NUM_BP, (unsigned int)&breakpoint_handler);
36:     intr_set_handler(EXCP_NUM_OF, (unsigned int)&overflow_handler);
37:     intr_set_handler(EXCP_NUM_BR, (unsigned int)&bound_range_exceeded_handler);
38:     intr_set_handler(EXCP_NUM_UD, (unsigned int)&invalid_opcode_handler);
39:     intr_set_handler(EXCP_NUM_NM, (unsigned int)&device_not_available_handler);
40:     intr_set_handler(EXCP_NUM_DF, (unsigned int)&double_fault_handler);
41:     intr_set_handler(EXCP_NUM_CSO,
42:                      (unsigned int)&coprocessor_segment_overrun_handler);
43:     intr_set_handler(EXCP_NUM_TS, (unsigned int)&invalid_tss_handler);
44:     intr_set_handler(EXCP_NUM_NP, (unsigned int)&segment_not_present_handler);
45:     intr_set_handler(EXCP_NUM_SS, (unsigned int)&stack_fault_handler);
46:     intr_set_handler(EXCP_NUM_GP, (unsigned int)&general_protection_handler);
47:     intr_set_handler(EXCP_NUM_PF, (unsigned int)&page_fault_handler);
48:     intr_set_handler(EXCP_NUM_MF,
49:                      (unsigned int)&x87_fpu_floating_point_error_handler);
50:     intr_set_handler(EXCP_NUM_AC, (unsigned int)&alignment_check_handler);
51:     intr_set_handler(EXCP_NUM_MC, (unsigned int)&machine_check_handler);
```

```
52:     intr_set_handler(EXCP_NUM_XM, (unsigned int)&simd_floating_point_handler);
53:     intr_set_handler(EXCP_NUM_VE, (unsigned int)&virtualization_handler);
54:
55:     /* Setup devices */
56:     con_init();
57:     timer_init();
58:     mem_init();
59:
60:     /* Setup File System */
61:     fs_init((void *)0x00011000);
62:
63:     /* Setup tasks */
64:     kern_task_init();
65:     task_init(fshell, 0, NULL);
66:
67:     /* Start paging */
68:     mem_page_start();
69:
70:     /* Setup interrupt handler and mask register */
71:     intr_set_handler(INTR_NUM_TIMER, (unsigned int)&timer_handler);
72:     intr_set_handler(INTR_NUM_KB, (unsigned int)&keyboard_handler);
73:     intr_set_handler(INTR_NUM_USER128, (unsigned int)&syscall_handler);
74:     intr_init();
75:     mask = intr_get_mask_master();
76:     mask &= ~(INTR_MASK_BIT_TIMER | INTR_MASK_BIT_KB);
77:     intr_set_mask_master(mask);
78:     sti();
79:
80:     /* End of kernel initialization process */
81:     while (1) {
82:             x86_halt();
83:     }
84:
85:     return 0;
86: }
```

ブートローダー・カーネルと来て、ここが C 言語のスタート地点です。ここではカーネルの初期化を行う `kern_init`関数を定義しています。なお、`kern_init`関数は 69 行もあり、OS5 の中で 3 番目に長い関数です。

初期化の流れは以下の通りです。

1. コンソール設定 (25〜27 行目)
2. 例外ハンドラ設定 (29〜53 行目)
3. デバイスドライバ設定 (55〜58 行目)
4. ファイルシステム設定 (60〜61 行目)
5. タスク設定 (スケジューラ設定)(63〜65 行目)
6. ページング設定 (MMU 設定)(67〜68 行目)
7. 割り込みハンドラ設定 (70〜78 行目)
8. カーネルタスクとして動作 (80〜83 行目)

2. と 7. を除き、各処理は関数化されており、初期化処理の本体は各関数内で行っています。初期化処理の内容については各ソースコードで説明します。なお、2. と 7. の処理について、例外・割り込み共に `intr_set_handler`関数でハンドラを設定しています。第 1 引数が割り込み/例外のベクタ番号で、第 2 引数がハンドラの先頭アドレスです。kernel/sys.S で定義していたハンドラはここで設定されます。そして、7. の `intr_init` 関数で割り込み初期化後、割り込みマスクの設定でタイマーとキーボードのみ割り込みを有効化しています。`sti`関数でアセンブラの `sti`命令を呼び出すことにより、CPU の割り込み機能が有効化されます。

カーネル自体はイベントドリブンで動作するように作っています。そのため、カーネルの各機能の設定を終えると 8. で x86_halt 関数 (割り込み等が発生するまで命令実行を停止させる x86 の hlt 命令を呼び出す関数) を呼び出し、で CPU を寝かせます。

カーネル起動時に最初に起動されるタスクであるシェルはどのように起動されるのかというと、4. でシェルの実行バイナリを見つけ、5. でカーネルのタスクの枠組みへシェルを設定し、6. でメモリ周りの設定を行い、7. の割り込み有効化でタイマー割り込みが開始することでスケジューラが動作を開始する、という流れです。スケジューラが動作を開始すると、10ms 周期のタイマー割り込み契機でランキューのタスクを切り替えてタスクを実行します。

定数化できていないマジックナンバーについて、`fs_init`関数に渡している"0x00011000"は、ユーザーランドの先頭アドレスです。

kernel/include/kernel.h

▼リスト 3.6　kernel/include/kernel.h

```
 1: #ifndef _KERNEL_H_
 2: #define _KERNEL_H_
 3:
 4: enum {
 5:     SYSCALL_TIMER_GET_GLOBAL_COUNTER = 1,
 6:     SYSCALL_SCHED_WAKEUP_MSEC,
 7:     SYSCALL_SCHED_WAKEUP_EVENT,
 8:     SYSCALL_CON_GET_CURSOR_POS_Y,
 9:     SYSCALL_CON_PUT_STR,
10:     SYSCALL_CON_PUT_STR_POS,
11:     SYSCALL_CON_DUMP_HEX,
12:     SYSCALL_CON_DUMP_HEX_POS,
13:     SYSCALL_CON_GET_LINE,
14:     SYSCALL_OPEN,
15:     SYSCALL_EXEC,
16:     SYSCALL_EXIT
17: };
18:
```

```
19: enum {
20:     EVENT_TYPE_KBD = 1,
21:     EVENT_TYPE_EXIT
22: };
23:
24: #endif /* _KERNEL_H_ */
```

カーネルがユーザーランド (アプリケーション群) へ公開するシステムコールを定義しています。

現状、カーネルとアプリケーションの間の API はシステムコールのみです。

また、UNIX のようにデバイスなどをファイルにしていない事もあり、ひたすらシステムコールが増えていく枠組みです。ただし、システムコールのインタフェースは単純なので、これはこれでシンプルで良いかなとも思います。

3.2 割り込み/例外

kernel/include/intr.h

▼リスト 3.7 kernel/include/intr.h

```
 1: #ifndef _INTR_H_
 2: #define _INTR_H_
 3:
 4: #define IOADR_MPIC_OCW2    0x0020
 5: #define IOADR_MPIC_OCW2_BIT_MANUAL_EOI     0x60
 6: #define IOADR_MPIC_ICW1    0x0020
 7: #define IOADR_MPIC_ICW2    0x0021
 8: #define IOADR_MPIC_ICW3    0x0021
 9: #define IOADR_MPIC_ICW4    0x0021
10: #define IOADR_MPIC_OCW1    0x0021
11: #define IOADR_SPIC_ICW1    0x00a0
12: #define IOADR_SPIC_ICW2    0x00a1
13: #define IOADR_SPIC_ICW3    0x00a1
14: #define IOADR_SPIC_ICW4    0x00a1
15: #define IOADR_SPIC_OCW1    0x00a1
16:
17: #define INTR_NUM_USER128   0x80
18:
19: void intr_init(void);
20: void intr_set_mask_master(unsigned char mask);
21: unsigned char intr_get_mask_master(void);
22: void intr_set_mask_slave(unsigned char mask);
23: unsigned char intr_get_mask_slave(void);
24: void intr_set_handler(unsigned char intr_num, unsigned int handler_addr);
25:
26: #endif /* _INTR_H_ */
```

割り込みコントローラ (PIC:Programmable Interrupt Controller) のレジスタの IO ア

ドレスの define と、割り込み設定関数のプロトタイプ宣言です。

kernel/intr.c

▼リスト 3.8　kernel/intr.c

```
 1: #include <intr.h>
 2: #include <io_port.h>
 3: 
 4: void intr_init(void)
 5: {
 6:     /* マスタ PIC の初期化 */
 7:     outb_p(0x11, IOADR_MPIC_ICW1);
 8:     outb_p(0x20, IOADR_MPIC_ICW2);
 9:     outb_p(0x04, IOADR_MPIC_ICW3);
10:     outb_p(0x01, IOADR_MPIC_ICW4);
11:     outb_p(0xff, IOADR_MPIC_OCW1);
12: 
13:     /* スレーブ PIC の初期化 */
14:     outb_p(0x11, IOADR_SPIC_ICW1);
15:     outb_p(0x28, IOADR_SPIC_ICW2);
16:     outb_p(0x02, IOADR_SPIC_ICW3);
17:     outb_p(0x01, IOADR_SPIC_ICW4);
18:     outb_p(0xff, IOADR_SPIC_OCW1);
19: }
20: 
21: void intr_set_mask_master(unsigned char mask)
22: {
23:     outb_p(mask, IOADR_MPIC_OCW1);
24: }
25: 
26: unsigned char intr_get_mask_master(void)
27: {
28:     return inb_p(IOADR_MPIC_OCW1);
29: }
30: 
31: void intr_set_mask_slave(unsigned char mask)
32: {
33:     outb_p(mask, IOADR_SPIC_OCW1);
34: }
35: 
36: unsigned char intr_get_mask_slave(void)
37: {
38:     return inb_p(IOADR_SPIC_OCW1);
39: }
40: 
41: void intr_set_handler(unsigned char intr_num, unsigned int handler_addr)
42: {
43:     extern unsigned char idt;
44:     unsigned int intr_dscr_top_half, intr_dscr_bottom_half;
45:     unsigned int *idt_ptr;
46: 
47:     idt_ptr = (unsigned int *)&idt;
48:     intr_dscr_bottom_half = handler_addr;
49:     intr_dscr_top_half = 0x00080000;
```

```
50:     intr_dscr_top_half = (intr_dscr_top_half & 0xffff0000)
51:             | (intr_dscr_bottom_half & 0x0000ffff);
52:     intr_dscr_bottom_half = (intr_dscr_bottom_half & 0xffff0000) | 0x00008e00;
53:     if (intr_num == INTR_NUM_USER128)
54:             intr_dscr_bottom_half |= 3 << 13;
55:     idt_ptr += intr_num * 2;
56:     *idt_ptr = intr_dscr_top_half;
57:     *(idt_ptr + 1) = intr_dscr_bottom_half;
58: }
```

割り込み/例外の初期化や設定を行う関数群を定義しています。ハードウェアとしてはプログラマブルインタラプトコントローラ (PIC) を扱う関数群です。

`intr_init`関数では割り込み番号の開始を「0x20(32) 番以降」に設定しています (`outb_p(0x20, IOADR_MPIC_ICW2)`でマスタ PIC を 0x20〜に設定し、`outb_p(0x28, IOADR_SPIC_ICW2)`でスレーブ PIC を 0x28〜に設定)。割り込み番号も例外番号も同じ番号の空間なので、割り込み番号の開始を「0 番以降」としてしまうと、割り込みの 0 番と例外の 0 番でバッティングします。例外は 0〜20(0x14) 番まであって、こちらは変更できないので、割り込み番号を 0x20〜にしています。

割り込み番号の開始番号 (0x20) や初期化の値は書籍「パソコンのレガシィ I/O 活用大全」を参考にしています。なお、割り込み番号の開始が 0x20 なのは Linux カーネルでもそうだったと思います (違っていたらゴメンナサイ)。

kernel/include/excp.h

▼リスト 3.9　kernel/include/excp.h

```
 1: #ifndef _EXCP_H_
 2: #define _EXCP_H_
 3:
 4: #define EXCEPTION_MAX      21
 5: #define EXCP_NUM_DE        0
 6: #define EXCP_NUM_DB        1
 7: #define EXCP_NUM_NMI       2
 8: #define EXCP_NUM_BP        3
 9: #define EXCP_NUM_OF        4
10: #define EXCP_NUM_BR        5
11: #define EXCP_NUM_UD        6
12: #define EXCP_NUM_NM        7
13: #define EXCP_NUM_DF        8
14: #define EXCP_NUM_CSO       9
15: #define EXCP_NUM_TS        10
16: #define EXCP_NUM_NP        11
17: #define EXCP_NUM_SS        12
18: #define EXCP_NUM_GP        13
19: #define EXCP_NUM_PF        14
20: #define EXCP_NUM_MF        16
```

```
21: #define EXCP_NUM_AC        17
22: #define EXCP_NUM_MC        18
23: #define EXCP_NUM_XM        19
24: #define EXCP_NUM_VE        20
25:
26: extern unsigned char exception_handler;
27: extern unsigned char divide_error_handler;
28: extern unsigned char debug_handler;
29: extern unsigned char nmi_handler;
30: extern unsigned char breakpoint_handler;
31: extern unsigned char overflow_handler;
32: extern unsigned char bound_range_exceeded_handler;
33: extern unsigned char invalid_opcode_handler;
34: extern unsigned char device_not_available_handler;
35: extern unsigned char double_fault_handler;
36: extern unsigned char coprocessor_segment_overrun_handler;
37: extern unsigned char invalid_tss_handler;
38: extern unsigned char segment_not_present_handler;
39: extern unsigned char stack_fault_handler;
40: extern unsigned char general_protection_handler;
41: extern unsigned char page_fault_handler;
42: extern unsigned char x87_fpu_floating_point_error_handler;
43: extern unsigned char alignment_check_handler;
44: extern unsigned char machine_check_handler;
45: extern unsigned char simd_floating_point_handler;
46: extern unsigned char virtualization_handler;
47:
48: void do_exception(void);
49: void do_page_fault(unsigned int error_code, unsigned int address);
50:
51: #endif /* _EXCP_H_ */
```

例外の番号とアセンブラ・C 言語側のハンドラの定義です。

なぜ enum を使わないのか。。。

kernel/excp.c

▼リスト 3.10　kernel/excp.c

```
 1: #include <excp.h>
 2: #include <console_io.h>
 3:
 4: void do_exception(void)
 5: {
 6:     put_str("exception\r\n");
 7:     while (1);
 8: }
 9:
10: void do_page_fault(unsigned int error_code, unsigned int address)
11: {
12:     put_str("page fault\r\n");
13:     put_str("error code: 0x");
```

```
14:     dump_hex(error_code, 8);
15:     put_str("\r\n");
16:     put_str("address   : 0x");
17:     dump_hex(address, 8);
18:     put_str("\r\n");
19:     while (1);
20: }
```

kernel/sys.S の例外のハンドラから呼び出される本体の処理を記述しています。

例外は起きてしまった場合、デバッグしなければならないので、例外に陥った時点でハンドラ内で `while (1)`でブロックするようにしています。

3.3 メモリ管理

kernel/include/memory.h

▼リスト 3.11 kernel/include/memory.h

```
 1: #ifndef __MEMORY_H__
 2: #define __MEMORY_H__
 3:
 4: #define PAGE_SIZE  0x1000
 5: #define PAGE_ADDR_MASK     0xfffff000
 6:
 7: struct page_directory_entry {
 8:     union {
 9:             struct {
10:                     unsigned int all;
11:             };
12:             struct {
13:                     unsigned int p: 1, r_w: 1, u_s: 1, pwt: 1, pcd: 1, a: 1,
14:                             reserved: 1, ps: 1, g: 1, usable: 3,
15:                             pt_base: 20;
16:             };
17:     };
18: };
19: struct page_table_entry {
20:     union {
21:             struct {
22:                     unsigned int all;
23:             };
24:             struct {
25:                     unsigned int p: 1, r_w: 1, u_s: 1, pwt: 1, pcd: 1, a: 1,
26:                             d: 1, pat: 1, g: 1, usable: 3, page_base: 20;
27:             };
28:     };
29: };
30:
31: void mem_init(void);
32: void mem_page_start(void);
```

```
33: void *mem_alloc(void);
34: void mem_free(void *page);
35:
36: #endif /* __MEMORY_H__ */
```

MMU(メモリ管理ユニット) 周りの定数、構造体、関数の定義です。

kernel/memory.c

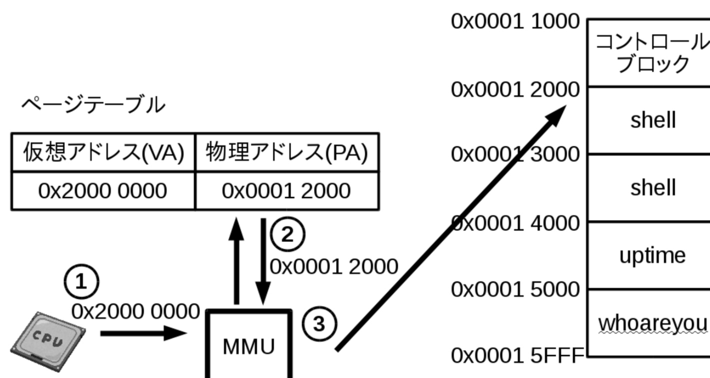
▼リスト 3.12　kernel/memory.c

```
 1: #include <stddef.h>
 2: #include <memory.h>
 3:
 4: #define CR4_BIT_PGE        (1U << 7)
 5: #define MAX_HEAP_PAGES     64
 6: #define HEAP_START_ADDR    0x00040000
 7:
 8: static char heap_alloc_table[MAX_HEAP_PAGES] = {0};
 9:
10: void mem_init(void)
11: {
12:     struct page_directory_entry *pde;
13:     struct page_table_entry *pte;
14:     unsigned int paging_base_addr;
15:     unsigned int i;
16:     unsigned int cr4;
17:
18:     /* Enable PGE(Page Global Enable) flag of CR4*/
19:     __asm__("movl   %%cr4, %0":"=r"(cr4):);
20:     cr4 |= CR4_BIT_PGE;
21:     __asm__("movl   %0, %%cr4"::"r"(cr4));
22:
23:     /* Initialize kernel page directory */
24:     pde = (struct page_directory_entry *)0x0008f000;
25:     pde->all = 0;
26:     pde->p = 1;
27:     pde->r_w = 1;
28:     pde->pt_base = 0x00090;
29:     pde++;
30:     for (i = 1; i < 0x400; i++) {
31:             pde->all = 0;
32:             pde++;
33:     }
34:
35:     /* Initialize kernel page table */
36:     pte = (struct page_table_entry *)0x00090000;
37:     for (i = 0x000; i < 0x007; i++) {
38:             pte->all = 0;
39:             pte++;
40:     }
41:     paging_base_addr = 0x00007;
42:     for (; i <= 0x085; i++) {
```

```
43:             pte->all = 0;
44:             pte->p = 1;
45:             pte->r_w = 1;
46:             pte->g = 1;
47:             pte->page_base = paging_base_addr;
48:             paging_base_addr += 0x00001;
49:             pte++;
50:     }
51:     for (; i < 0x095; i++) {
52:             pte->all = 0;
53:             pte++;
54:     }
55:     paging_base_addr = 0x00095;
56:     for (; i <= 0x09f; i++) {
57:             pte->all = 0;
58:             pte->p = 1;
59:             pte->r_w = 1;
60:             pte->g = 1;
61:             pte->page_base = paging_base_addr;
62:             paging_base_addr += 0x00001;
63:             pte++;
64:     }
65:     for (; i < 0x0b8; i++) {
66:             pte->all = 0;
67:             pte++;
68:     }
69:     paging_base_addr = 0x000b8;
70:     for (; i <= 0x0bf; i++) {
71:             pte->all = 0;
72:             pte->p = 1;
73:             pte->r_w = 1;
74:             pte->pwt = 1;
75:             pte->pcd = 1;
76:             pte->g = 1;
77:             pte->page_base = paging_base_addr;
78:             paging_base_addr += 0x00001;
79:             pte++;
80:     }
81:     for (; i < 0x400; i++) {
82:             pte->all = 0;
83:             pte++;
84:     }
85: }
86:
87: void mem_page_start(void)
88: {
89:     unsigned int cr0;
90:
91:     __asm__("movl   %%cr0, %0":"=r"(cr0):);
92:     cr0 |= 0x80000000;
93:     __asm__("movl   %0, %%cr0"::"r"(cr0));
94: }
95:
96: void *mem_alloc(void)
97: {
98:     unsigned int i;
99:
100:    for (i = 0; heap_alloc_table[i] && (i < MAX_HEAP_PAGES); i++);
101:
```

```
102:     if (i >= MAX_HEAP_PAGES)
103:             return (void *)NULL;
104:
105:     heap_alloc_table[i] = 1;
106:     return (void *)(HEAP_START_ADDR + i * PAGE_SIZE);
107: }
108:
109: void mem_free(void *page)
110: {
111:     unsigned int i = ((unsigned int)page - HEAP_START_ADDR) / PAGE_SIZE;
112:     heap_alloc_table[i] = 0;
113: }
```

メモリ関係の関数群で、主に CPU のメモリ管理ユニット (MMU) の設定を行います。MMU はページングという機能を提供するものです。ページングは、メモリを「ページ」という単位で分割し、「仮想アドレス」という実際のアドレス (物理アドレス) とは別のアドレスを割り当てて管理します。そして、仮想アドレスから物理アドレスへの変換表を「ページテーブル」と呼びます。変換の流れを図 3.1 に示します。なお、複数のページテーブルをまとめたものを「ページディレクトリ」と呼びます。

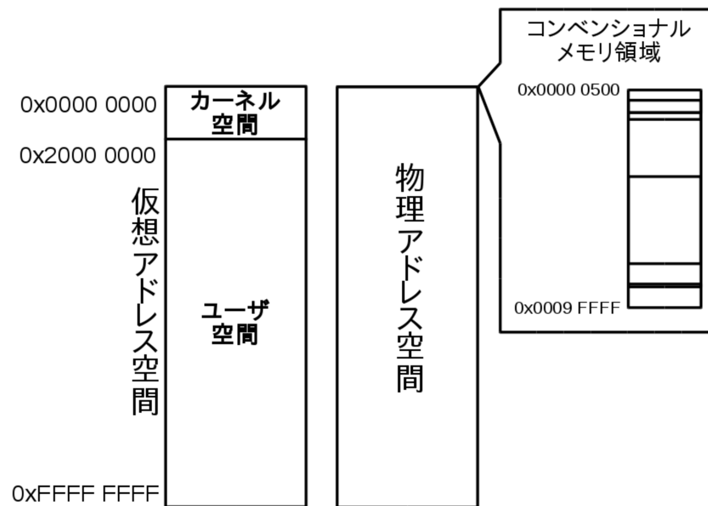


▲図 3.1　ページテーブルと MMU

CPU の設定でページングを有効化すると、カーネルやアプリケーションは仮想アドレスで動作するようになります。これにより、「アプリケーションはカーネルの領域へアクセスさせない」、「アプリケーションはすべて同じアドレスから実行を開始する」といったことを実現しています。なお、ページサイズは 4KB です[*1]。また、仮想アドレスは"Virtual Address"で"VA"、物理アドレスは"Physical Address"で"PA"などと記載されていたりもします。

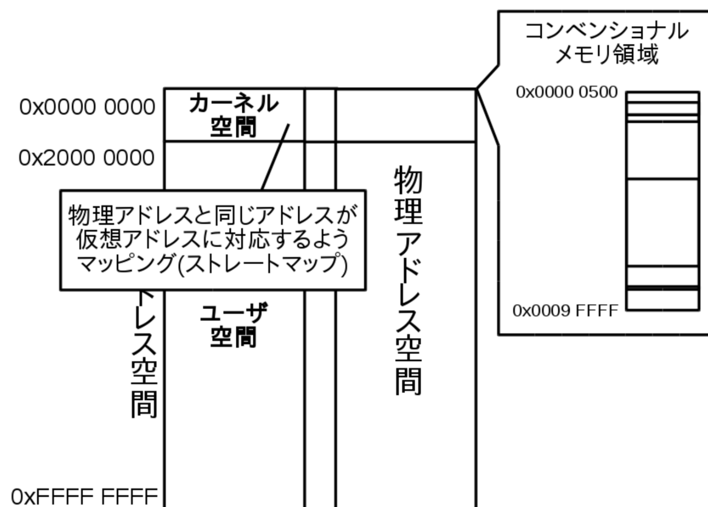
[*1] CPU の設定で変更できます。

ページディレクトリとページテーブルへ設定を追加し、ある VA を PA に対応付けることを「マッピング」、「マップする」の様に呼びます。OS5 でのマッピングに関して、まず、OS5 では VA の 0x0000 0000〜0x1FFF FFFF をカーネル空間、0x2000 0000〜0xFFFF FFFF をユーザ空間としています (図 3.2)。カーネル空間は VA=PA となるようマップしています (図 3.3)。0x0000 0000〜0x1FFF FFFF のアドレス空間内には、カーネルやユーザーランドの実行バイナリ等を配置している「コンベンショナルメモリ」が全て含まれます。そのため、カーネルは RAM に配置したすべての資源にアクセスできることになります。ユーザ空間は実行するタスク*2ごとにマップを変えます。例えば、shell の実行時はユーザ空間を shell が配置されている PA へマップします (図 3.4)。

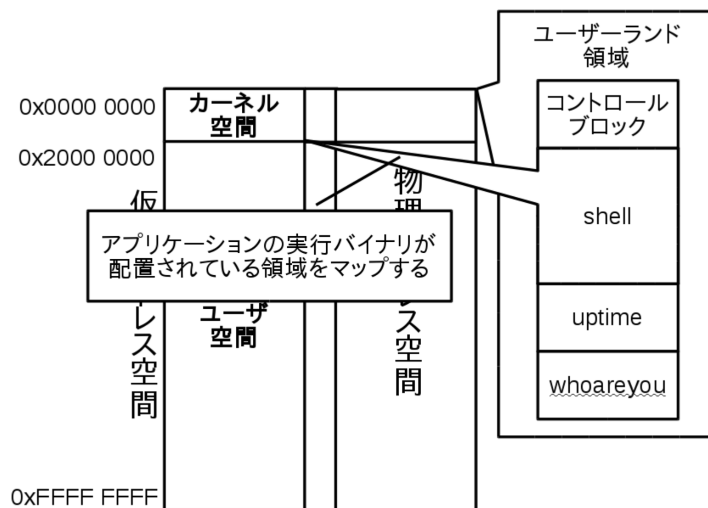


▲図 3.2　VA のマッピングについて (1)

*2 OS5 では x86 CPU の言い回しに合わせて「タスク」と呼んでいます。なお、カーネルより上位のユーザーランドで話すときは分かり易さから「アプリケーション」と呼んでいます。実体は共に同じものです。



▲図 3.3　VA **のマッピングについて** (2)



▲図 3.4　VA **のマッピングについて** (3)

kernel/init.c の `kern_init`関数からは、`mem_init`関数と `mem_page_start`関数を呼び出しています。共に MMU のページング機能の関数で、`mem_init`で設定し、`mem_page_start`で有効化します。`mem_init`ではグローバルページ機能を有効化した後、カーネル

のページディレクトリ/テーブルを設定しています。なお、`mem_init`関数は OS5 で 2 番目に長い関数です (76 行)。

残る 2 つの関数はメモリの動的確保 (`mem_alloc`関数) と解放 (`mem_free`関数) です。動的確保/解放はページサイズに合わせて 4KB 単位で、`heap_alloc_table`という配列で管理しています。

仮想アドレス空間は 0x0000 0000〜0x1fff ffff がカーネル空間で、0x2000 0000〜0xffff ffff がユーザ空間です。カーネル空間へは物理アドレスの同じアドレス (0x0000 0000〜0x1fff ffff) が対応付けられています (マップされています)。

「仮想アドレス」は ARM CPU の方言です

「仮想アドレス」という言い回しは、実は Intel CPU の言い回しではないです。ページングの仕組みを ARM CPU で先に勉強していて、Intel CPU の「リニアアドレス」よりわかりやすい気がして、ページングに関しては「仮想アドレス」という言い回しを使っています。ちなみに、x86 はページングの他にセグメンテーションという仕組みもあるため、物理アドレスへ至る流れは「論理アドレス (セグメンテーション)」→「リニアアドレス (ページング)」→「物理アドレス」となります。

OS5 ではセグメンテーションを使用していません

セグメンテーションもページングと同じく物理アドレスを分割し、「論理アドレス」というアドレスを割り当てて管理する機能です。「論理アドレス」という形でページングと同様に「仮想アドレス」を提供できます。(ちなみに、セグメンテーションにもページフォルト例外同様に「セグメント不在例外」があります。)
ハードウェアが持つ機能は積極的に使うようにしていきたいのですが、ページングで事足りているため、セグメンテーションは使用していません。ただし、ページングと違いセグメンテーションは機能として無効化することができないので、1 つのセグメントがメモリ空間全て (0x0000 0000〜0xffff ffff) を指すように設定しています (kernel/sys.S の 211〜217 行目)。

3.4　タスク管理

kernel/include/sched.h

▼リスト 3.13　kernel/include/sched.h

```
 1: #ifndef _SCHED_H_
 2: #define _SCHED_H_
 3:
 4: #include <cpu.h>
 5: #include <task.h>
 6:
 7: #define TASK_NUM   3
 8:
 9: extern struct task task_instance_table[TASK_NUM];
10: extern struct task *current_task;
11:
12: unsigned short sched_get_current(void);
13: int sched_runq_enq(struct task *t);
14: int sched_runq_del(struct task *t);
15: void schedule(void);
16: int sched_update_wakeupq(void);
17: void wakeup_after_msec(unsigned int msec);
18: int sched_update_wakeupevq(unsigned char event_type);
19: void wakeup_after_event(unsigned char event_type);
20:
21: #endif /* _SCHED_H_ */
```

スケジューラに関するヘッダファイルです。

kernel/sched.c

▼リスト 3.14　kernel/sched.c

```
 1: #include <stddef.h>
 2: #include <sched.h>
 3: #include <cpu.h>
 4: #include <io_port.h>
 5: #include <intr.h>
 6: #include <timer.h>
 7: #include <lock.h>
 8: #include <kern_task.h>
 9:
10: static struct {
11:     struct task *head;
12:     unsigned int len;
13: } run_queue = {NULL, 0};
14: static struct {
15:     struct task *head;
```

```
16:     unsigned int len;
17: } wakeup_queue = {NULL, 0};
18: static struct {
19:     struct task *head;
20:     unsigned int len;
21: } wakeup_event_queue = {NULL, 0};
22: static struct task dummy_task;
23: static unsigned char is_task_switched_in_time_slice = 0;
24:
25: struct task task_instance_table[TASK_NUM];
26: struct task *current_task = NULL;
27:
28: unsigned short sched_get_current(void)
29: {
30:     return x86_get_tr() / 8;
31: }
32:
33: int sched_runq_enq(struct task *t)
34: {
35:     unsigned char if_bit;
36:
37:     kern_lock(&if_bit);
38:
39:     if (run_queue.head) {
40:             t->prev = run_queue.head->prev;
41:             t->next = run_queue.head;
42:             run_queue.head->prev->next = t;
43:             run_queue.head->prev = t;
44:     } else {
45:             t->prev = t;
46:             t->next = t;
47:             run_queue.head = t;
48:     }
49:     run_queue.len++;
50:
51:     kern_unlock(&if_bit);
52:
53:     return 0;
54: }
55:
56: int sched_runq_del(struct task *t)
57: {
58:     unsigned char if_bit;
59:
60:     if (!run_queue.head)
61:             return -1;
62:
63:     kern_lock(&if_bit);
64:
65:     if (run_queue.head->next != run_queue.head) {
66:             if (run_queue.head == t)
67:                     run_queue.head = run_queue.head->next;
68:             t->prev->next = t->next;
69:             t->next->prev = t->prev;
70:     } else
71:             run_queue.head = NULL;
72:     run_queue.len--;
73:
```

```
74:     kern_unlock(&if_bit);
75:
76:     return 0;
77: }
78:
79: void schedule(void)
80: {
81:     if (!run_queue.head) {
82:             if (current_task) {
83:                     current_task = NULL;
84:                     outb_p(IOADR_MPIC_OCW2_BIT_MANUAL_EOI | INTR_IR_TIMER,
85:                             IOADR_MPIC_OCW2);
86:                     task_instance_table[KERN_TASK_ID].context_switch();
87:             }
88:     } else if (current_task) {
89:             if (current_task != current_task->next) {
90:                     current_task = current_task->next;
91:                     if (is_task_switched_in_time_slice) {
92:                             current_task->task_switched_in_time_slice = 1;
93:                             is_task_switched_in_time_slice = 0;
94:                     }
95:                     outb_p(IOADR_MPIC_OCW2_BIT_MANUAL_EOI | INTR_IR_TIMER,
96:                             IOADR_MPIC_OCW2);
97:                     current_task->context_switch();
98:             }
99:     } else {
100:            current_task = run_queue.head;
101:            if (is_task_switched_in_time_slice) {
102:                    current_task->task_switched_in_time_slice = 1;
103:                    is_task_switched_in_time_slice = 0;
104:            }
105:            outb_p(IOADR_MPIC_OCW2_BIT_MANUAL_EOI | INTR_IR_TIMER,
106:                    IOADR_MPIC_OCW2);
107:            current_task->context_switch();
108:    }
109: }
110:
111: int sched_wakeupq_enq(struct task *t)
112: {
113:    unsigned char if_bit;
114:
115:    kern_lock(&if_bit);
116:
117:    if (wakeup_queue.head) {
118:            t->prev = wakeup_queue.head->prev;
119:            t->next = wakeup_queue.head;
120:            wakeup_queue.head->prev->next = t;
121:            wakeup_queue.head->prev = t;
122:    } else {
123:            t->prev = t;
124:            t->next = t;
125:            wakeup_queue.head = t;
126:    }
127:    wakeup_queue.len++;
128:
129:    kern_unlock(&if_bit);
130:
131:    return 0;
```

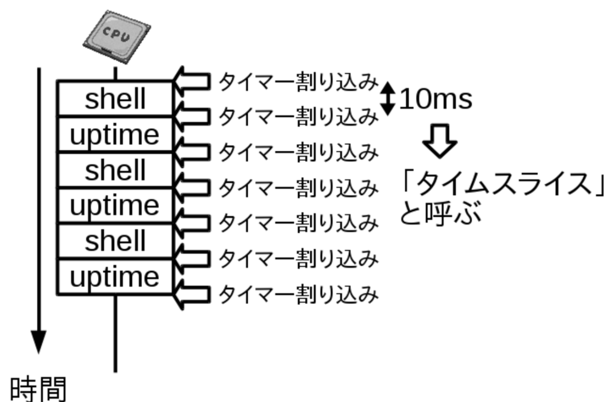
```
132: }
133:
134: int sched_wakeupq_del(struct task *t)
135: {
136:    unsigned char if_bit;
137:
138:    if (!wakeup_queue.head)
139:            return -1;
140:
141:    kern_lock(&if_bit);
142:
143:    if (wakeup_queue.head->next != wakeup_queue.head) {
144:            if (wakeup_queue.head == t)
145:                    wakeup_queue.head = wakeup_queue.head->next;
146:            t->prev->next = t->next;
147:            t->next->prev = t->prev;
148:    } else
149:            wakeup_queue.head = NULL;
150:    wakeup_queue.len--;
151:
152:    kern_unlock(&if_bit);
153:
154:    return 0;
155: }
156:
157: int sched_update_wakeupq(void)
158: {
159:    struct task *t, *next;
160:    unsigned char if_bit;
161:
162:    if (!wakeup_queue.head)
163:            return -1;
164:
165:    kern_lock(&if_bit);
166:
167:    t = wakeup_queue.head;
168:    do {
169:            next = t->next;
170:            if (t->wakeup_after_msec > TIMER_TICK_MS) {
171:                    t->wakeup_after_msec -= TIMER_TICK_MS;
172:            } else {
173:                    t->wakeup_after_msec = 0;
174:                    sched_wakeupq_del(t);
175:                    sched_runq_enq(t);
176:            }
177:            t = next;
178:    } while (wakeup_queue.head && t != wakeup_queue.head);
179:
180:    kern_unlock(&if_bit);
181:
182:    return 0;
183: }
184:
185: void wakeup_after_msec(unsigned int msec)
186: {
187:    unsigned char if_bit;
188:
189:    kern_lock(&if_bit);
190:
```

```
191:    if (current_task->next != current_task)
192:            dummy_task.next = current_task->next;
193:    current_task->wakeup_after_msec = msec;
194:    sched_runq_del(current_task);
195:    sched_wakeupq_enq(current_task);
196:    current_task = &dummy_task;
197:    is_task_switched_in_time_slice = 1;
198:    schedule();
199:
200:    kern_unlock(&if_bit);
201: }
202:
203: int sched_wakeupevq_enq(struct task *t)
204: {
205:    unsigned char if_bit;
206:
207:    kern_lock(&if_bit);
208:
209:    if (wakeup_event_queue.head) {
210:            t->prev = wakeup_event_queue.head->prev;
211:            t->next = wakeup_event_queue.head;
212:            wakeup_event_queue.head->prev->next = t;
213:            wakeup_event_queue.head->prev = t;
214:    } else {
215:            t->prev = t;
216:            t->next = t;
217:            wakeup_event_queue.head = t;
218:    }
219:    wakeup_event_queue.len++;
220:
221:    kern_unlock(&if_bit);
222:
223:    return 0;
224: }
225:
226: int sched_wakeupevq_del(struct task *t)
227: {
228:    unsigned char if_bit;
229:
230:    if (!wakeup_event_queue.head)
231:            return -1;
232:
233:    kern_lock(&if_bit);
234:
235:    if (wakeup_event_queue.head->next != wakeup_event_queue.head) {
236:            if (wakeup_event_queue.head == t)
237:                    wakeup_event_queue.head = wakeup_event_queue.head->next;
238:            t->prev->next = t->next;
239:            t->next->prev = t->prev;
240:    } else
241:            wakeup_event_queue.head = NULL;
242:    wakeup_event_queue.len--;
243:
244:    kern_unlock(&if_bit);
245:
246:    return 0;
247: }
248:
```

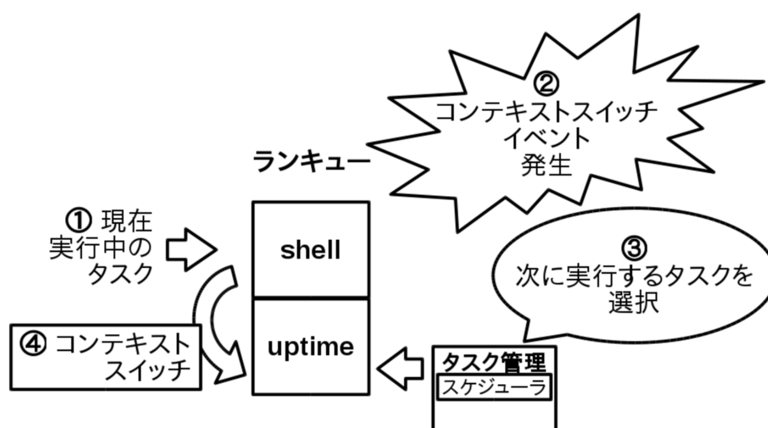
```
249: int sched_update_wakeupevq(unsigned char event_type)
250: {
251:     struct task *t, *next;
252:     unsigned char if_bit;
253:
254:     if (!wakeup_event_queue.head)
255:             return -1;
256:
257:     kern_lock(&if_bit);
258:
259:     t = wakeup_event_queue.head;
260:     do {
261:             next = t->next;
262:             if (t->wakeup_after_event == event_type) {
263:                     t->wakeup_after_event = 0;
264:                     sched_wakeupevq_del(t);
265:                     sched_runq_enq(t);
266:             }
267:             t = next;
268:     } while (wakeup_event_queue.head && t != wakeup_event_queue.head);
269:
270:     kern_unlock(&if_bit);
271:
272:     return 0;
273: }
274:
275: void wakeup_after_event(unsigned char event_type)
276: {
277:     unsigned char if_bit;
278:
279:     kern_lock(&if_bit);
280:
281:     if (current_task->next != current_task)
282:             dummy_task.next = current_task->next;
283:     current_task->wakeup_after_event = event_type;
284:     sched_runq_del(current_task);
285:     sched_wakeupevq_enq(current_task);
286:     current_task = &dummy_task;
287:     is_task_switched_in_time_slice = 1;
288:     schedule();
289:
290:     kern_unlock(&if_bit);
291: }
```

スケジューラの関数を定義しています。カーネルの中では kernel/console_io.c に次いで長いソースファイルです (291 行)。

一番重要な関数は `schedule`関数です。主にタイマー割り込み (10ms) で呼び出され、実行するタスクを切り替える (コンテキストスイッチ) 役割を担います (図 3.5)。実行可能なタスクは「ランキュー」というキューへ設定します。そのため、コンテキストスイッチの際は、ランキューの中から次のタスクを選択します (図 3.6)。



▲図 3.5　**タイムスライスについて**

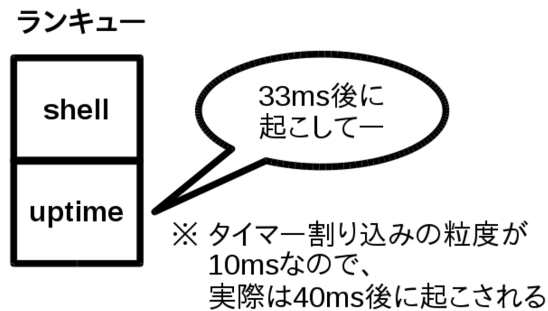


▲図 3.6　**コンテキストスイッチまでの流れ**

その他には、ランキューとウェイクアップキューの操作の関数を定義しています。OS5 のスケジューラでは、タスクは時間経過やイベント (キーボード入力、タスク終了) を待つことができます。待っている間はランキューから外し、ウェイクアップキュー、あるいはウェイクアップイベントキューへ追加します。ウェイクアップキューが時間経過待ちのキューで、ウェイクアップイベントキューがイベント発生待ちのキューです。

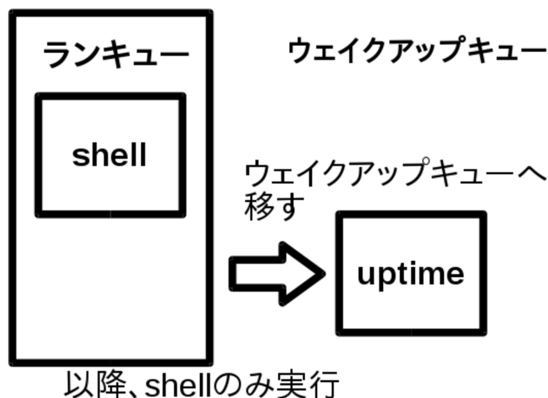
これらのキューの使い方の例として uptime というアプリケーションがウェイクアップ

キューを使用してスリープする流れを説明します。まず、uptime が「33ms 後に起こしてほしい」とカーネルへ通知します (図 3.7)。アプリケーションとカーネルのインタフェースはシステムコールで、この場合、"SYSCALL_SCHED_WAKEUP_MSEC"というシステムコールを引数に「33ms」を設定して発行します。ソースコードの対応する箇所は apps/uptime/uptime.c の `syscall(SYSCALL_SCHED_WAKEUP_MSEC, 33, 0, 0);`です (23 行目)。



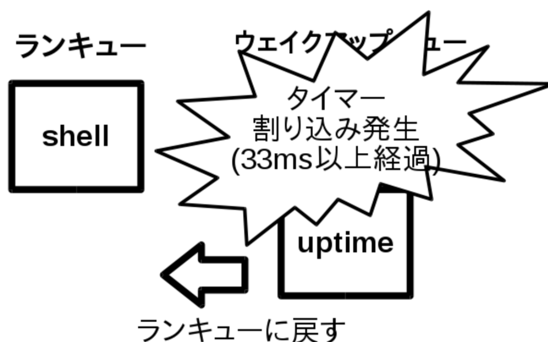
▲図 3.7　スリープの流れ (1)

すると、カーネルは uptime をランキューから外し、ウェイクアップキューへ移します (図 3.8)。ランキューには shell のみになるので、以降は shell のみ実行されます。ソースコードとしては、システムコールの入り口処理を実装している kernel/syscall.c から、"SYSCALL_SCHED_WAKEUP_MSEC"の場合、kernel/sched.c の `wakeup_after_msec()`(185〜201 行目) を呼び出しています。



▲図 3.8　スリープの流れ (2)

そして、タイマー割り込み発生時に所定の時間 (今回の場合「33ms」) 経過していたことを確認すると、uptime をランキューへ戻します (図 3.9)。ソースコードとしては、kernel/timer.c で `sched_update_wakeupq()`を呼び出しています。`sched_update_wakeupq()`は、kernel/sched.c の 157〜183 行目で定義しています。



▲図 3.9 スリープの流れ (3)

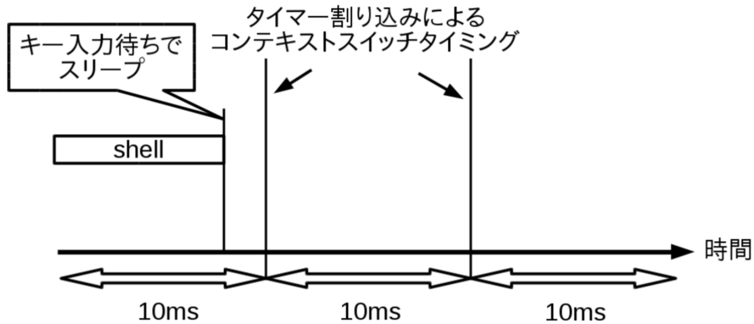
なお、全てのタスクがスリープ等でランキューから抜けた場合、「カーネルタスク」が動作します (図 3.10)。カーネルタスクの実体は初期化完了後の `kern_init`関数 (kernel/init.c) で、80〜83 行目で x86 の `hlt`命令を無限ループで何度も実行しているため、カーネルタスクがスケジュールされると、割り込みが入るまで CPU は `hlt`命令で休むことになります。なお、`schedule`関数内において、81〜87 行目の条件分岐がカーネルタスクへコンテキストスイッチしている箇所です。

▲図 3.10 カーネルタスク

スリープ後のコンテキストスイッチの問題

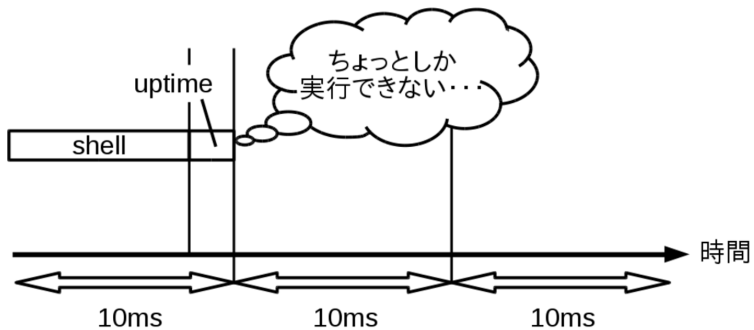
タスクがスリープした後のコンテキストスイッチにはちょっとした問題があります。例えば、shell がタイムスライス (10ms) の途中でキー入力待ちでスリープしたとします (図

3.11)。shell が"SYSCALL_SCHED_WAKEUP_EVENT"のシステムコールを発行することになり、kernel/sched.c の `wakeup_after_event`関数が呼ばれます。



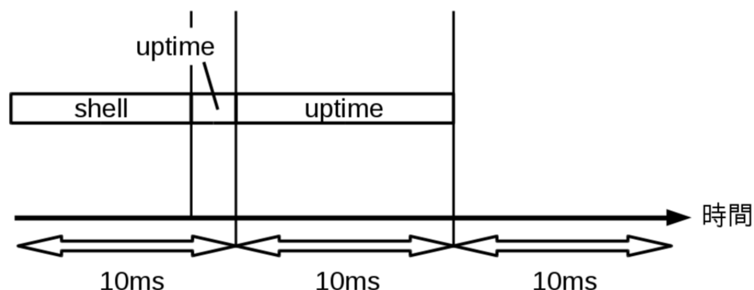
▲図 3.11　**スリープ後のコンテキストスイッチの問題** (1)

shell がランキューから外され、uptime が次に実行するタスクとして選択されたとします。すると、shell のタイムスライスの残り時間が経過するとコンテキストスイッチされてしまいます。タイムスライスを 10ms としている以上、タスクには 10ms は実行させてあげるべきなので、これではちょっと不平等です (図 3.12)。



▲図 3.12　**スリープ後のコンテキストスイッチの問題** (2)

そのため、OS5 カーネルのスケジューラでは、タスクがスリープした場合はタイムスライスの残りを次のタスクへプレゼントしたものと考え、スリープ後のタイマー割り込みではコンテキストスイッチしないようにしています (図 3.13)。



▲図 3.13　スリープ後のコンテキストスイッチの問題 (3)

実装としては、スリープでのコンテキストスイッチ時にフラグ変数 `is_task_switched_in_time_slice`をセットします (`wakeup_after_msec`と `wakeup_after_event`関数内でセットしています)。そして、`schedule`関数内で、`is_task_switched_in_time_slice`をチェックし、セットされていた場合は `current_task->task_switched_in_time_slice`をセットしています (92 行目、102 行目)。この時の `current_task`はコンテキストスイッチ後のタスクを指しています。コンテキストスイッチの後、タイマー割り込みが発生しても、タイマー割り込みハンドラ (kernel/timer.c の `do_ir_timer`関数) で `current_task->task_switched_in_time_slice`をチェックし、セットされている場合は `schedule`関数を呼び出さないようにしています (kernel/timer.c の 12〜18 行目)。

過去の遺産"task_instance_table"

`task_instance_table`はカーネルタスクにしか使っていないです。元々はこのテーブルにすべてのタスクが並んでいたのですが、メモリの動的確保を実装し、タスクの生成時に動的に確保するようにしたため、今ではカーネルタスクだけ `task_instance_table`に残っている状態です。

kernel/include/kern_task.h

▼リスト 3.15　kernel/include/kern_task.h

```
 1: #ifndef _KERN_TASK_H_
 2: #define _KERN_TASK_H_
 3:
 4: #define KERN_TASK_ID               0
 5:
 6: void kern_task_init(void);
 7:
 8: #endif /* _KERN_TASK_H_ */
```

カーネルタスクの task_instance_table内のインデックス (KERN_TASK_ID) の定義と、初期化関数 (kern_task_init) のプロトタイプ宣言を行っています。

kernel/kern_task_init.c

▼リスト 3.16　kernel/kern_task_init.c

```
 1: #include <kern_task.h>
 2: #include <cpu.h>
 3: #include <sched.h>
 4:
 5: #define KERN_TASK_GDT_IDX  5
 6:
 7: static void kern_task_context_switch(void)
 8: {
 9:     __asm__("ljmp   $0x28, $0");
10: }
11:
12: void kern_task_init(void)
13: {
14:     static struct tss kern_task_tss;
15:     unsigned int old_cr3, cr3 = 0x0008f018;
16:     unsigned short segment_selector = 8 * KERN_TASK_GDT_IDX;
17:
18:     kern_task_tss.esp0 = 0x0007f800;
19:     kern_task_tss.ss0 = GDT_KERN_DS_OFS;
20:     kern_task_tss.__cr3 = 0x0008f018;
21:     init_gdt(KERN_TASK_GDT_IDX, (unsigned int)&kern_task_tss,
22:              sizeof(kern_task_tss), 0);
23:     __asm__("movl   %%cr3, %0":"=r"(old_cr3):);
24:     cr3 |= old_cr3 & 0x00000fe7;
25:     __asm__("movl   %0, %%cr3"::"r"(cr3));
26:     __asm__("ltr %0"::"r"(segment_selector));
27:
28:     /* Setup context switch function */
29:     task_instance_table[KERN_TASK_ID].context_switch =
30:             kern_task_context_switch;
31: }
```

カーネルタスクの初期化を行う kern_task_init関数を定義しています。今実行中のコンテキストをタスクとして登録する点が、その他のタスク登録とは異なります。(そのため、今だに kern_task_init関数が残っています。)

やっていることは以下の 3 つです。

1. TSS の設定、GDT への登録 (18〜22 行目)
2. CR3 レジスタの設定 (23〜25 行目)
3. ltr 命令で TSS のロード (26 行目)
4. コンテキストスイッチ関数の登録 (28〜30 行目)

1. について、タスクステートセグメント (TSS) の設定を行っています。x86 CPU はタスク管理の機能を持っており、x86 CPU の枠組みでタスクを管理する際の構造が TSS です。TSS もセグメントなので、GDT へ登録します (21〜22 行目)。

2. について、CR3 はページディレクトリのベースアドレスとキャッシュの設定を行うレジスタです[*3]。ページディレクトリのベースアドレスは 4KB(0x1000) の倍数でなければならないので、下位 12 ビットは必ず 0 です。そこで、CR3 の下位 12 ビットにページディレクトリの設定を行うビットがあります。設定ビットはビット 4(PCD[*4]) とビット 3(PWT[*5]) で、それ以外のビット (ビット 11〜5 とビット 2〜0) は予約ビットです。CR3 へ設定したい内容は、15 行目で変数 `cr3`へ設定しています。ページディレクトリの開始アドレスが 0x0008 f000 で (図 1.2)、PCD と PWT を共にセットするため、CR3 へ設定する値は"0x0008 f018"です。なお、CR3 の予約ビットへは、CR3 を読み出して得られた値を書き込まなければならないとデータシートに記載されています。そのため、23〜25 行目では、CR3 レジスタを読み出し (23 行目)、読みだした値 (`old_cr3`変数) の予約ビットのみ `cr3`変数へ反映し (24 行目)、その後 `cr3`変数の値を CR3 レジスタへ格納 (25 行目) ということを行っています。

3. は `ltr`命令を使用して 1. で登録した TSS をタスクレジスタ (TR) へ登録しています。

4. はコンテキストスイッチ用の関数を `struct task`構造体のエントリへ登録しているところです。`struct task`は OS5 のカーネルでタスクを管理する構造体です。kernel/sched.c でも使用していますが、`task_instance_table`配列は `struct task`構造体の配列です。`kernel_task_context_switch`関数が登録される関数で、やっていることは `ljmp`命令のインラインアセンブラ 1 行です。`ljmp`命令はオペランドに GDT 内の TSS のオフセットを与えるとそのタスクへコンテキストスイッチできます。カーネルタスクの TSS の GDT 内でのオフセットは 0x28 なので、`ljmp`命令で 0x28 を指定することでカーネルタスクへコンテキストスイッチできます。なお、第 2 オペランドはコンテキストスイッチの場合、無視されます。

[*3] そのため、CR3 は、PDBR(ページディレクトリベースレジスタ) とも呼ばれます。

[*4] ページキャッシュディスエーブルです。セットされているとページディレクトリのキャッシュが抑制されます。

[*5] ページレベル書き込み透過です。セットされるとライトスルーキャッシングが有効になり、クリアされるとライトバックキャッシングが有効になります。

古い実装"task_instalce_table"

4. の `task_instalce_table`は、実は古い実装で、今は使用しているのはカーネルタスクのみです。その他のユーザーランドのタスクでは `struct task`はタスク生成時に動的に確保します (kernel/task.c で説明します)。

kernel/include/task.h

▼**リスト** 3.17　kernel/include/task.h

```
 1: #ifndef _TASK_H_
 2: #define _TASK_H_
 3:
 4: #include <cpu.h>
 5: #include <fs.h>
 6:
 7: #define CONTEXT_SWITCH_FN_SIZE     12
 8: #define CONTEXT_SWITCH_FN_TSKNO_FIELD      8
 9:
10: struct task {
11:     /* ランキュー・ウェイクアップキュー（時間経過待ち・イベント待ち）の
12:      * いずれかに繋がれる（同時に複数のキューに存在することが無いよう
13:      * 運用する）*/
14:     struct task *prev;
15:     struct task *next;
16:
17:     unsigned short task_id;
18:     struct tss tss;
19:     void (*context_switch)(void);
20:     unsigned char context_switch_func[CONTEXT_SWITCH_FN_SIZE];
21:     char task_switched_in_time_slice;
22:     unsigned int wakeup_after_msec;
23:     unsigned char wakeup_after_event;
24: };
25:
26: extern unsigned char context_switch_template[CONTEXT_SWITCH_FN_SIZE];
27:
28: void task_init(struct file *f, int argc, char *argv[]);
29: void task_exit(struct task *t);
30:
31: #endif /* _TASK_H_ */
```

タスクの構造体 (`struct task`) に関する定義と、タスク生成時の初期化関数 (`task_init`) とタスク終了関数 (`task_exit`) のプロトタイプ宣言です。

kernel/task.c

▼リスト 3.18　kernel/task.c

```
 1: #include <task.h>
 2: #include <memory.h>
 3: #include <fs.h>
 4: #include <sched.h>
 5: #include <common.h>
 6: #include <lock.h>
 7: #include <kernel.h>
 8: #include <cpu.h>
 9:
10: #define GDT_IDX_OFS        5
11: #define APP_ENTRY_POINT    0x20000030
12: #define APP_STACK_BASE_USER        0xffffe800
13: #define APP_STACK_BASE_KERN        0xfffff000
14: #define APP_STACK_SIZE     4096
15: #define GDT_USER_CS_OFS    0x0018
16: #define GDT_USER_DS_OFS    0x0020
17:
18: /*
19: 00000000 <context_switch>:
20:    0:   55                      push   %ebp
21:    1:   89 e5                   mov    %esp,%ebp
22:    3:   ea 00 00 00 00 00 00    ljmp   $0x00,$0x0
23:    a:   5d                      pop    %ebp
24:    b:   c3                      ret
25:  */
26: unsigned char context_switch_template[CONTEXT_SWITCH_FN_SIZE] = {
27:     0x55,
28:     0x89, 0xe5,
29:     0xea, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
30:     0x5d,
31:     0xc3
32: };
33:
34: static unsigned short task_id_counter = 1;
35:
36: static int str_get_len(const char *src)
37: {
38:     int len;
39:     for (len = 0; src[len] != '\0'; len++);
40:     return len + 1;
41: }
42:
43: void task_init(struct file *f, int argc, char *argv[])
44: {
45:     struct page_directory_entry *pd_base_addr, *pde;
46:     struct page_table_entry *pt_base_addr, *pt_stack_base_addr, *pte;
47:     struct task *new_task;
48:     unsigned int paging_base_addr, phys_stack_base, phys_stack_base2;
49:     unsigned int i;
50:     unsigned int len = 0;
51:     unsigned int argv_space_num, vsp, arg_size;
```

```
52:     unsigned char *sp, *sp2;
53:     char *t;
54:
55:     /* Allocate task resources */
56:     pd_base_addr = (struct page_directory_entry *)mem_alloc();
57:     pt_base_addr = (struct page_table_entry *)mem_alloc();
58:     pt_stack_base_addr = (struct page_table_entry *)mem_alloc();
59:     new_task = (struct task *)mem_alloc();
60:     phys_stack_base = (unsigned int)mem_alloc();
61:     phys_stack_base2 = (unsigned int)mem_alloc();
62:
63:     /* Initialize task page directory */
64:     pde = pd_base_addr;
65:     pde->all = 0;
66:     pde->p = 1;
67:     pde->r_w = 1;
68:     pde->pt_base = 0x00090;
69:     pde++;
70:     for (i = 1; i < 0x080; i++) {
71:             pde->all = 0;
72:             pde++;
73:     }
74:     pde->all = 0;
75:     pde->p = 1;
76:     pde->r_w = 1;
77:     pde->u_s = 1;
78:     pde->pt_base = (unsigned int)pt_base_addr >> 12;
79:     pde++;
80:     for (i++; i < 0x3ff; i++) {
81:             pde->all = 0;
82:             pde++;
83:     }
84:     pde->all = 0;
85:     pde->p = 1;
86:     pde->r_w = 1;
87:     pde->u_s = 1;
88:     pde->pt_base = (unsigned int)pt_stack_base_addr >> 12;
89:     pde++;
90:
91:     /* Initialize task page table */
92:     pte = pt_base_addr;
93:     paging_base_addr = (unsigned int)f->data_base_addr >> 12;
94:     for (i = 0; i < f->head->block_num; i++) {
95:             pte->all = 0;
96:             pte->p = 1;
97:             pte->r_w = 1;
98:             pte->u_s = 1;
99:             pte->page_base = paging_base_addr++;
100:            pte++;
101:    }
102:    for (; i < 0x400; i++) {
103:            pte->all = 0;
104:            pte++;
105:    }
106:
107:    /* Initialize stack page table */
108:    pte = pt_stack_base_addr;
109:    for (i = 0; i < 0x3fd; i++) {
```

```
110:            pte->all = 0;
111:            pte++;
112:    }
113:    paging_base_addr = phys_stack_base >> 12;
114:    pte->all = 0;
115:    pte->p = 1;
116:    pte->r_w = 1;
117:    pte->u_s = 1;
118:    pte->page_base = paging_base_addr;
119:    pte++;
120:    paging_base_addr = phys_stack_base2 >> 12;
121:    pte->all = 0;
122:    pte->p = 1;
123:    pte->r_w = 1;
124:    pte->u_s = 1;
125:    pte->page_base = paging_base_addr;
126:    pte++;
127:    pte->all = 0;
128:    pte++;
129:
130:    /* Setup task_id */
131:    new_task->task_id = task_id_counter++;
132:
133:    /* Setup context switch function */
134:    copy_mem(context_switch_template, new_task->context_switch_func,
135:             CONTEXT_SWITCH_FN_SIZE);
136:    new_task->context_switch_func[CONTEXT_SWITCH_FN_TSKNO_FIELD] =
137:            8 * (new_task->task_id + GDT_IDX_OFS);
138:    new_task->context_switch = (void (*)(void))new_task->context_switch_func;
139:
140:    /* Setup GDT for task_tss */
141:    init_gdt(new_task->task_id + GDT_IDX_OFS, (unsigned int)&new_task->tss,
142:             sizeof(struct tss), 3);
143:
144:    /* Setup task stack */
145:    /* スタックに int argc と char *argv[] を積み、
146:     * call 命令でジャンプした直後を再現する。
147:     *
148:     * 例) argc=3, argv={"HOGE", "P", "FUGAA"}
149:     * | VA            | 内容              | 備考                           |
150:     * |---------------+-------------------+--------------------------------|
151:     * | 0x2000 17d0 |                   |                                |
152:     * | 0x2000 17d4 | (Don't Care)      | ESP はここを指した状態にする (*) |
153:     * | 0x2000 17d8 | 3                 | argc                           |
154:     * | 0x2000 17dc | 0x2000 17e4       | argv                           |
155:     * | 0x2000 17e0 | (Don't Care)      |                                |
156:     * | 0x2000 17e4 | 0x2000 17f0       | argv[0]                        |
157:     * | 0x2000 17e8 | 0x2000 17f5       | argv[1]                        |
158:     * | 0x2000 17ec | 0x2000 17f7       | argv[2]                        |
159:     * | 0x2000 17f0 | 'H'  'O' 'G'  'E' |                                |
160:     * | 0x2000 17f4 | '\0' 'P' '\0' 'F' |                                |
161:     * | 0x2000 17f8 | 'U'  'G' 'A'  'A' |                                |
162:     * | 0x2000 17fc | '\0'              |                                |
163:     * |---------------+-------------------+--------------------------------|
164:     * | 0x2000 1800 |                   |                                |
165:     * (*) call 命令は near ジャンプ時、call 命令の次の命令のアドレスを
166:     * 復帰時の EIP としてスタックに積むため。
167:     */
```

```
168:    for (i = 0; i < (unsigned int)argc; i++) {
169:            len += str_get_len(argv[i]);
170:    }
171:    argv_space_num = (len / 4) + 1;
172:    arg_size = 4 * (4 + argc + argv_space_num);
173:
174:    sp = (unsigned char *)(phys_stack_base2 + (APP_STACK_SIZE / 2));
175:    sp -= arg_size;
176:
177:    sp += 4;
178:
179:    *(int *)sp = argc;
180:    sp += 4;
181:
182:    *(unsigned int *)sp = APP_STACK_BASE_USER - (4 * (argc + argv_space_num));
183:    sp += 4;
184:
185:    sp += 4;
186:
187:    vsp = APP_STACK_BASE_USER - (4 * argv_space_num);
188:    sp2 = sp + (4 * argc);
189:    for (i = 0; i < (unsigned int)argc; i++) {
190:            *(unsigned int *)sp = vsp;
191:            sp += 4;
192:            t = argv[i];
193:            for (; *t != '\0'; t++) {
194:                    vsp++;
195:                    *sp2++ = *t;
196:            }
197:            *sp2++ = '\0';
198:            vsp++;
199:    }
200:
201:    /* Setup task_tss */
202:    new_task->tss.eip = APP_ENTRY_POINT;
203:    new_task->tss.esp = APP_STACK_BASE_USER - arg_size;
204:    new_task->tss.eflags = 0x00000200;
205:    new_task->tss.esp0 = APP_STACK_BASE_KERN;
206:    new_task->tss.ss0 = GDT_KERN_DS_OFS;
207:    new_task->tss.es = GDT_USER_DS_OFS | 0x0003;
208:    new_task->tss.cs = GDT_USER_CS_OFS | 0x0003;
209:    new_task->tss.ss = GDT_USER_DS_OFS | 0x0003;
210:    new_task->tss.ds = GDT_USER_DS_OFS | 0x0003;
211:    new_task->tss.fs = GDT_USER_DS_OFS | 0x0003;
212:    new_task->tss.gs = GDT_USER_DS_OFS | 0x0003;
213:    new_task->tss.__cr3 = (unsigned int)pd_base_addr | 0x18;
214:
215:    /* Add task to run_queue */
216:    sched_runq_enq(new_task);
217: }
218:
219: void task_exit(struct task *t)
220: {
221:    unsigned char if_bit;
222:    struct page_directory_entry *pd_base_addr, *pde;
223:    struct page_table_entry *pt_base_addr, *pt_stack_base_addr, *pte;
224:    unsigned int phys_stack_base, phys_stack_base2;
225:
```

```
226:     kern_lock(&if_bit);
227:
228:     sched_update_wakeupevq(EVENT_TYPE_EXIT);
229:     sched_runq_del(t);
230:
231:     pd_base_addr =
232:             (struct page_directory_entry *)(t->tss.__cr3 & PAGE_ADDR_MASK);
233:     pde = pd_base_addr + 0x080;
234:     pt_base_addr = (struct page_table_entry *)(pde->pt_base << 12);
235:     pde = pd_base_addr + 0x3ff;
236:     pt_stack_base_addr = (struct page_table_entry *)(pde->pt_base << 12);
237:     pte = pt_stack_base_addr + 0x3fd;
238:     phys_stack_base = pte->page_base << 12;
239:     pte = pt_stack_base_addr + 0x3fe;
240:     phys_stack_base2 = pte->page_base << 12;
241:
242:     mem_free((void *)phys_stack_base2);
243:     mem_free((void *)phys_stack_base);
244:     mem_free(t);
245:     mem_free(pt_stack_base_addr);
246:     mem_free(pt_base_addr);
247:     mem_free(pd_base_addr);
248:
249:     schedule();
250:
251:     kern_unlock(&if_bit);
252: }
```

タスク[*6]の実行開始時の初期化を行う `task_init`関数と、タスク終了時の終了処理を行う `task_exit`関数を定義しています。また、カーネル内ではここでしか使わないために `str_get_len`関数もここで定義しています。なお、`task_init`は OS5 の中で最も長い関数です (175 行)。

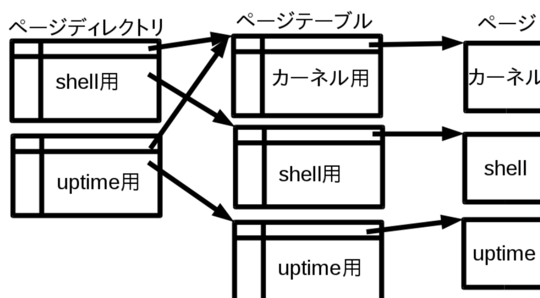
`task_init`はタスクの生成から、スケジューラへの登録までを行います。以下の流れです。

1. タスクが存在する為のメモリを確保 (55〜61 行目)
2. タスクのページディレクトリ/テーブルを設定 (63〜128 行目)
3. タスク ID を生成、設定 (130〜131 行目)
4. コンテキストスイッチ関数を生成、設定 (133〜138 行目)
5. タスクステートセグメント (TSS) を GDT へ登録 (140〜142 行目)
6. タスクのスタック領域へ実行に必要な値を積む (144〜199 行目)
7. タスクステートセグメント (TSS) 設定 (201〜213 行目)
8. ランキューへタスクを登録 (215〜216 行目)

[*6] OS5 では x86 CPU の言い回しに合わせて「タスク」と呼んでいます。なお、カーネルより上位のユーザーランドで話すときは分かり易さから「アプリケーション」と呼んでいます。実体は共に同じものです。

1. では`mem_alloc`を、「ページディレクトリ」、「ページテーブル (コード・データ領域)」、「ページテーブル (スタック領域)」、「`struct task`」、「スタック領域 (x2)」の合計 6 回呼び出しています。4KB 毎のアロケーションなので、合計すると 24KB がタスク一つ当たりに必要なメモリです。

2. について、ページディレクトリ・ページテーブルの構成を図 3.14 に示します。例として shell と uptime の 2 つのタスクについて描いています。共にカーネルのページテーブルを指しているのは、システムコール呼び出しでユーザーモードからカーネルモードへ権限昇格した際に、カーネル空間の関数を呼び出せるようにするためです。



▲図 3.14　OS5 のページディレクトリ・ページテーブル構成

4. では、コンテキストスイッチの関数のバイナリを動的に生成しています。kernel/kern_task_init.c でも説明しましたが、`ljmp`命令はオペランドに GDT 内の TSS のオフセットを指定することで、コンテキストスイッチできます。データシートを読む限り、このオペランドはレジスタを指定することもできるようなのですが、少なくとも QEMU で正常動作を確認できていません。そこで、苦肉の策として、`ljmp`命令を含むコンテキストスイッチ用の関数のコンパイル後のバイナリを予め用意しておき (18〜32 行目の`context_switch_template`配列)、`task_init`で新しいタスクを生成する際に、`context_switch_template`からコピーしてオペランドの部分のバイナリのみ書き換える事を行っています。

6. について、タスクを実行開始する際に、あたかもランキューに以前から居たかのようにコンテキストスイッチできるよう、確保したばかりのスタック領域へ値を積んでいます。

3.5 ファイルシステム

kernel/include/fs.h

▼リスト 3.19　kernel/include/fs.h

```
 1: #ifndef _FS_H_
 2: #define _FS_H_
 3:
 4: #include <list.h>
 5:
 6: #define MAX_FILE_NAME                     32
 7: #define RESERVED_FILE_HEADER_SIZE  15
 8:
 9: struct file_head {
10:     struct list lst;
11:     unsigned char num_files;
12: };
13:
14: struct file_header {
15:     char name[MAX_FILE_NAME];
16:     unsigned char block_num;
17:     unsigned char reserve[RESERVED_FILE_HEADER_SIZE];
18: };
19:
20: struct file {
21:     struct list lst;
22:     struct file_header *head;
23:     void *data_base_addr;
24: };
25:
26: extern struct file *fshell;
27:
28: void fs_init(void *fs_base_addr);
29: struct file *fs_open(const char *name);
30: int fs_close(struct file *f);
31:
32: #endif /* _FS_H_ */
```

ファイルシステム周りの構造体等を定義しているソースコードです。

kernel/fs.c

▼リスト 3.20　kernel/fs.c

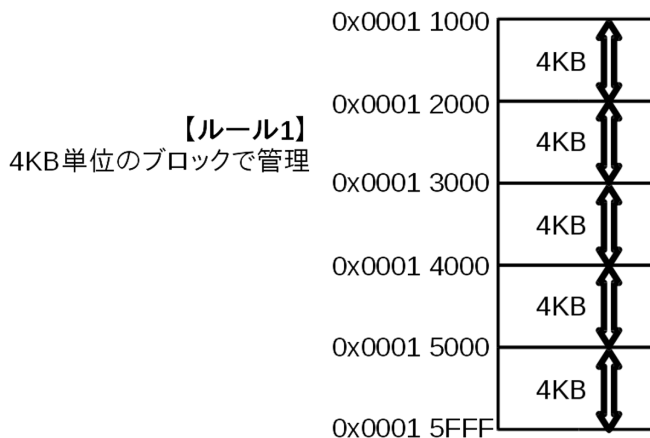
```
 1: #include <fs.h>
 2: #include <stddef.h>
 3: #include <memory.h>
 4: #include <list.h>
```

```
 5: #include <queue.h>
 6: #include <common.h>
 7:
 8: struct file_head fhead;
 9: struct file *fshell;
10:
11: void fs_init(void *fs_base_addr)
12: {
13:     struct file *f;
14:     unsigned char i;
15:     unsigned char *file_start_addr = fs_base_addr;
16:
17:     queue_init((struct list *)&fhead);
18:     fhead.num_files = *(unsigned char *)fs_base_addr;
19:
20:     file_start_addr += PAGE_SIZE;
21:     for (i = 1; i <= fhead.num_files; i++) {
22:             f = (struct file *)mem_alloc();
23:             f->head = (struct file_header *)file_start_addr;
24:             f->data_base_addr =
25:                     (char *)file_start_addr + sizeof(struct file_header);
26:             file_start_addr += PAGE_SIZE * f->head->block_num;
27:             queue_enq((struct list *)f, (struct list *)&fhead);
28:     }
29:     fshell = (struct file *)fhead.lst.next;
30: }
31:
32: struct file *fs_open(const char *name)
33: {
34:     struct file *f;
35:
36:     /* 将来的には、struct file の task_id メンバに open したタスクの
37:      * TASK_ID を入れるようにする。そして、open しようとしているファ
38:      * イルの task_id が既に設定されていれば、fs_open はエラーを返す
39:      * ようにする */
40:
41:     for (f = (struct file *)fhead.lst.next; f != (struct file *)&fhead;
42:          f = (struct file *)f->lst.next) {
43:             if (!str_compare(name, f->head->name))
44:                     return f;
45:     }
46:
47:     return NULL;
48: }
49:
50: int fs_close(struct file *f __attribute__ ((unused)))
51: {
52:     /* 将来的には、fid に対応する struct file の task_id メンバーを設定
53:      * なし (0) にする。 */
54:     return 0;
55: }
```

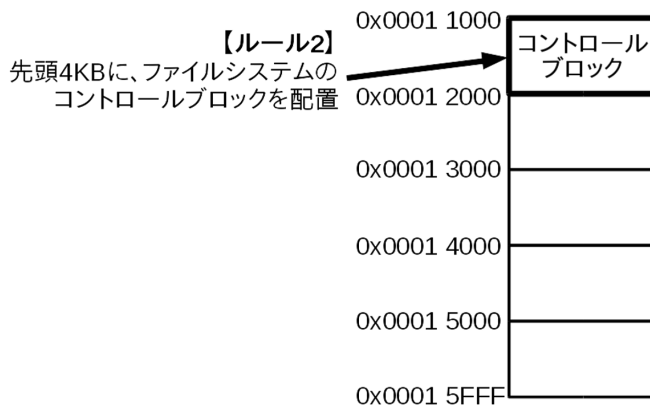
ファイルシステムの初期化と操作を行う関数群です。カーネル初期化 (`kern_init`関数) で `fs_init`を呼び、open システムコールから `fs_open`が呼ばれます。

まず、OS5 のファイルシステムは「特定のメモリ領域のバイナリ列を『ファイル』として認識するためのルール集」に過ぎません。OS5 のファイルシステムにおけるルールを

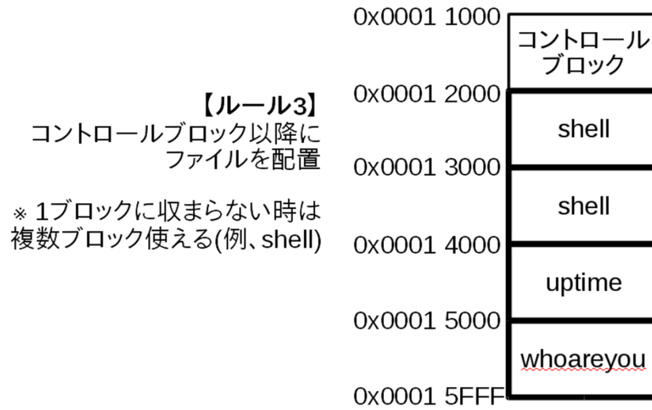
図 3.15、図 3.16、図 3.17、図 3.18、図 3.19、図 3.20 で説明します。



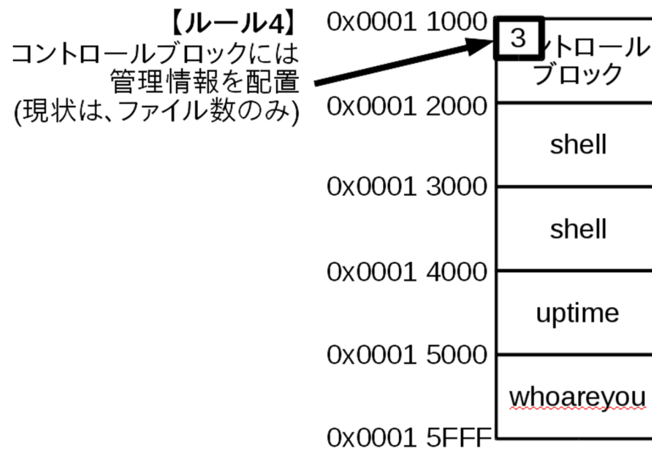
▲**図** 3.15　OS5 **のファイルシステム** (1)



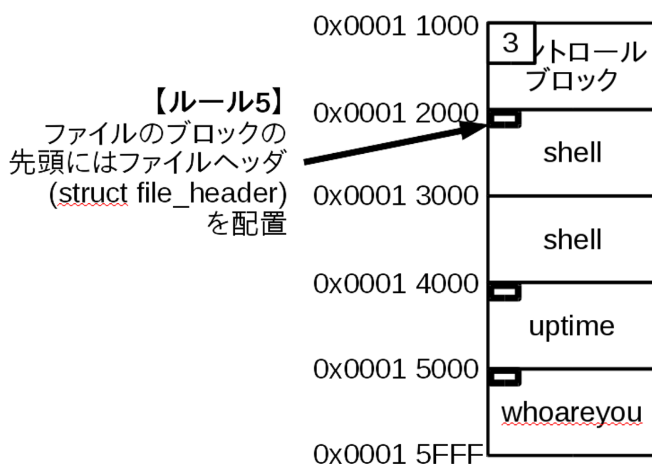
▲**図** 3.16　OS5 **のファイルシステム** (2)



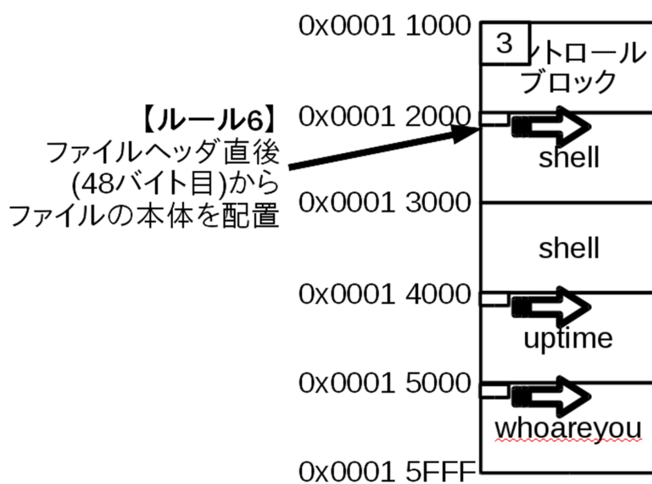
▲**図** 3.17　OS5 **のファイルシステム** (3)



▲**図** 3.18　OS5 **のファイルシステム** (4)

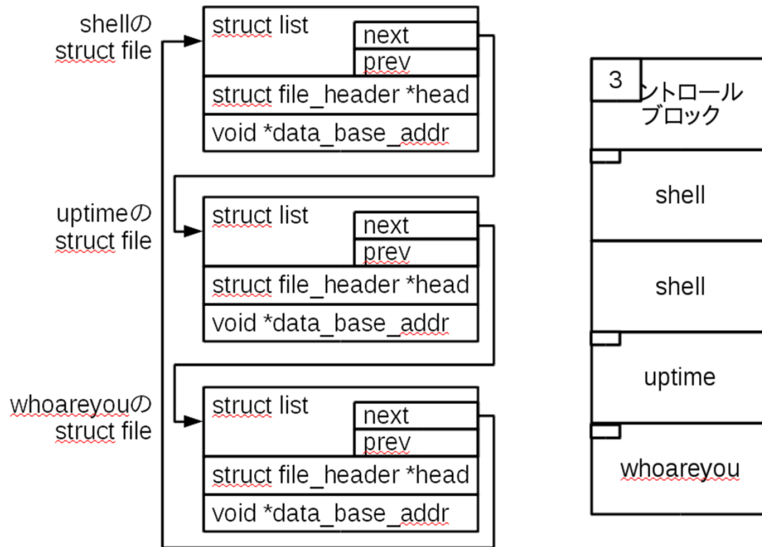


▲図 3.19　OS5 のファイルシステム (5)

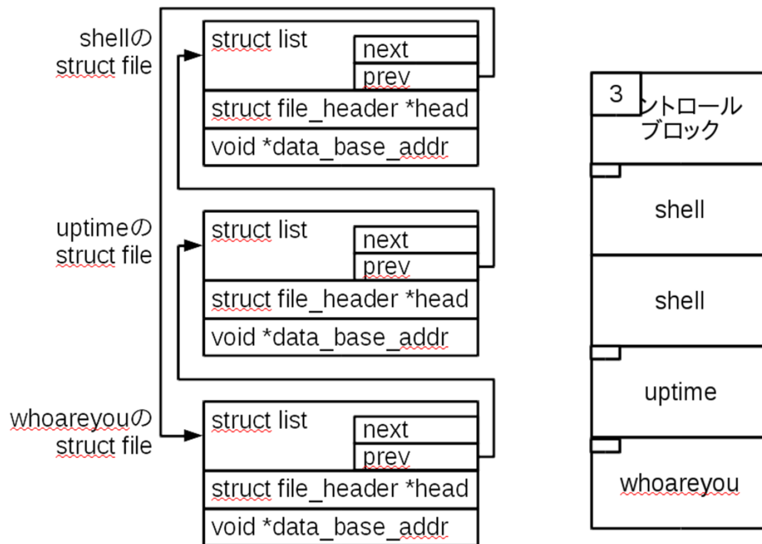


▲図 3.20　OS5 のファイルシステム (6)

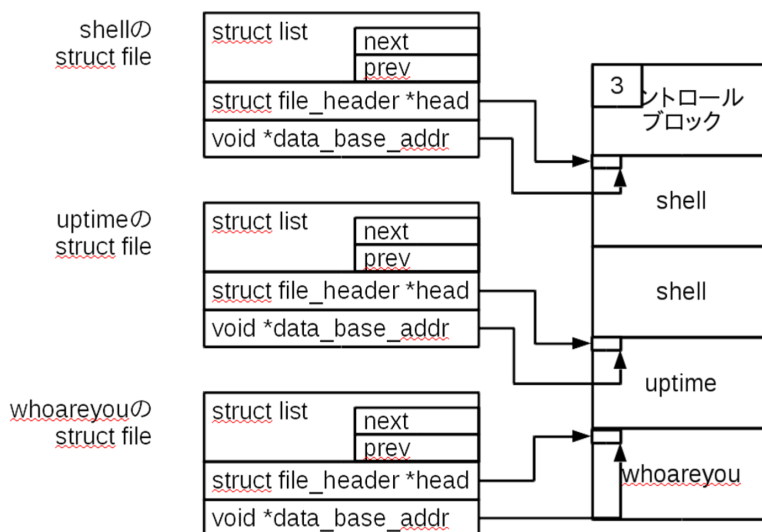
kernel/fs.c について、`fs_init`は引数でファイルシステムが配置されている領域の先頭アドレスを受け取り、ファイルシステムの内容をスキャンして `struct file`という構造体のリンクリストを作成します (図 3.21、図 3.22、図 3.23)。



▲図 3.21　ファイルシステム初期化 (next の関係)



▲図 3.22　ファイルシステム初期化 (prev の関係)



▲図 3.23　ファイルシステム初期化 (head・data_base_addr の関係)

OS5 では「ファイルシステムの 1 番最初のエントリをカーネル起動後最初に起動させるアプリケーションバイナリとする」ルールにしています。そこで、1 番最初のエントリを `struct file *fshell`へ設定しています。なお、ファイルシステムの 1 番目のエントリは、実行ファイルであればシェルで無くとも良いです。特に意味もなく `fshell`という変数名のままになっています。

`fs_open`は引数で与えられたファイル名と一致する `struct file`エントリをリンクリストから検索して `struct file`のポインタを返します。

3.6　システムコール

kernel/include/syscall.h

▼リスト 3.21　kernel/include/syscall.h

```
1: #ifndef _SYSCALL_H_
2: #define _SYSCALL_H_
3:
4: unsigned int do_syscall(unsigned int syscall_id, unsigned int arg1,
5:                         unsigned int arg2, unsigned int arg3);
6:
7: #endif /* _SYSCALL_H_ */
```

システムコール割り込みから呼び出されてシステムコールを実行する do_syscall関数のプロトタイプ宣言のみです。

kernel/syscall.c

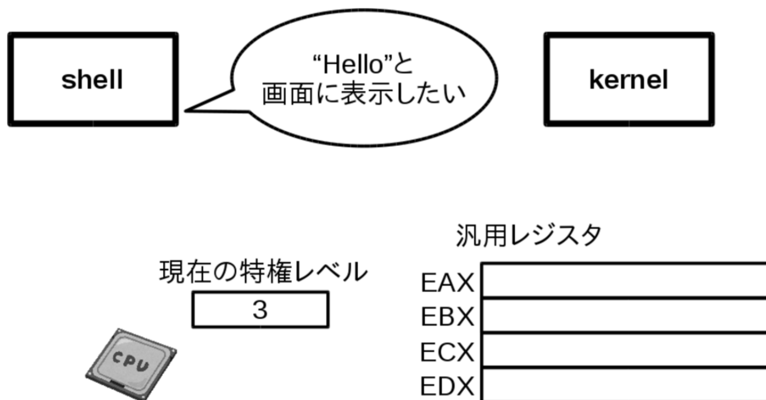
▼リスト 3.22　kernel/syscall.c

```
 1: #include <syscall.h>
 2: #include <kernel.h>
 3: #include <timer.h>
 4: #include <sched.h>
 5: #include <fs.h>
 6: #include <task.h>
 7: #include <console_io.h>
 8: #include <cpu.h>
 9:
10: unsigned int do_syscall(unsigned int syscall_id, unsigned int arg1,
11:                         unsigned int arg2, unsigned int arg3)
12: {
13:     unsigned int result = -1;
14:     unsigned int gdt_idx;
15:     unsigned int tss_base_addr;
16:
17:     switch (syscall_id) {
18:     case SYSCALL_TIMER_GET_GLOBAL_COUNTER:
19:             result = timer_get_global_counter();
20:             break;
21:     case SYSCALL_SCHED_WAKEUP_MSEC:
22:             wakeup_after_msec(arg1);
23:             result = 0;
24:             break;
25:     case SYSCALL_SCHED_WAKEUP_EVENT:
26:             wakeup_after_event(arg1);
27:             result = 0;
28:             break;
29:     case SYSCALL_CON_GET_CURSOR_POS_Y:
30:             result = (unsigned int)cursor_pos.y;
31:             break;
32:     case SYSCALL_CON_PUT_STR:
33:             put_str((char *)arg1);
34:             result = 0;
35:             break;
36:     case SYSCALL_CON_PUT_STR_POS:
37:             put_str_pos((char *)arg1, (unsigned char)arg2,
38:                         (unsigned char)arg3);
39:             result = 0;
40:             break;
41:     case SYSCALL_CON_DUMP_HEX:
42:             dump_hex(arg1, arg2);
43:             result = 0;
44:             break;
45:     case SYSCALL_CON_DUMP_HEX_POS:
```

```
46:             dump_hex_pos(arg1, arg2, (unsigned char)(arg3 >> 16),
47:                          (unsigned char)(arg3 & 0x0000ffff));
48:             result= 0;
49:             break;
50:     case SYSCALL_CON_GET_LINE:
51:             result = get_line((char *)arg1, arg2);
52:             break;
53:     case SYSCALL_OPEN:
54:             result = (unsigned int)fs_open((char *)arg1);
55:             break;
56:     case SYSCALL_EXEC:
57:             task_init((struct file *)arg1, (int)arg2, (char **)arg3);
58:             result = 0;
59:             break;
60:     case SYSCALL_EXIT:
61:             gdt_idx = x86_get_tr() / 8;
62:             tss_base_addr = (gdt[gdt_idx].base2 << 24) |
63:                     (gdt[gdt_idx].base1 << 16) | (gdt[gdt_idx].base0);
64:             task_exit((struct task *)(tss_base_addr - 0x0000000c));
65:             result = 0;
66:             break;
67:     }
68:
69:     return result;
70: }
```

システムコールのソースファイルです。このソースファイルではシステムコールの入り口処理 (do_syscall関数) を定義しています。

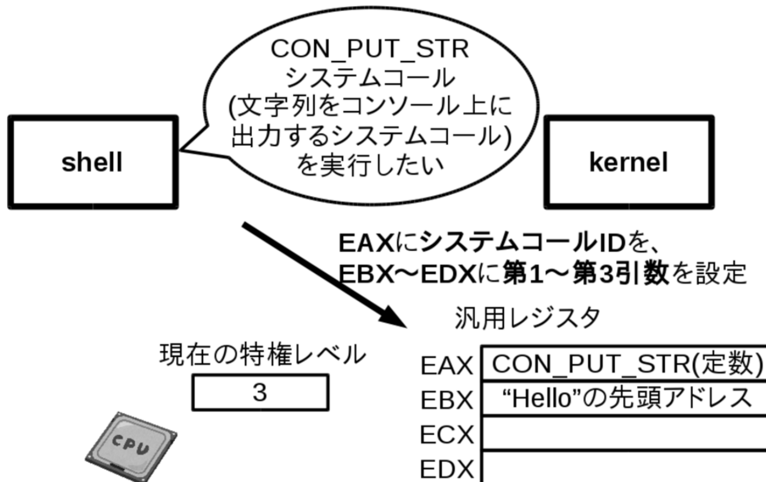
システムコール呼び出しの流れを説明します。「shell が"Hello"とコンソール画面へ表示したい」とします (図 3.24)。



▲図 3.24　**システムコール実行の流れ** (1)

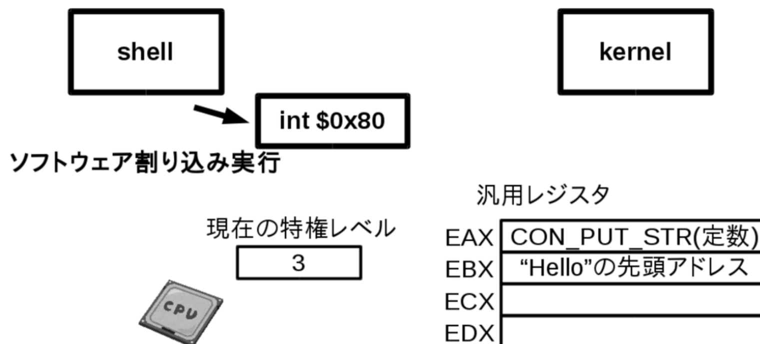
コンソール画面へ文字列を表示するためのシステムコールは"CON_PUT_STR"です。

システムコール呼び出しにおいて、アプリケーションとカーネルでパラメータの受け渡しには汎用レジスタ (EAX、EBX、ECX、EDX) を使用します。CON_PUT_STR を実行するために、汎用レジスタへ必要なパラメータを設定します (図 3.25)。



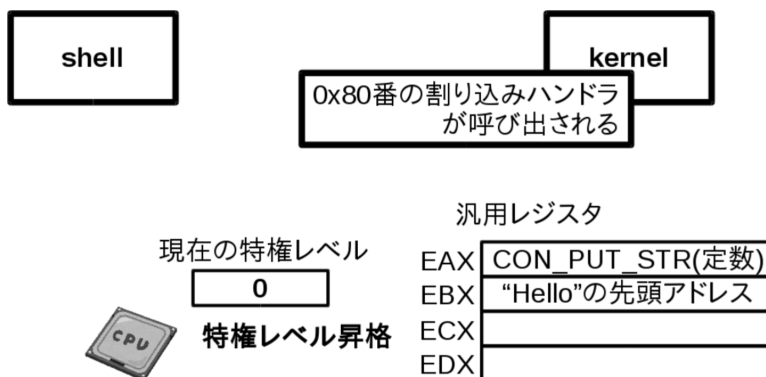
▲図 3.25　システムコール実行の流れ (2)

システムコールを発行するトリガーはソフトウェア割り込みです。OS5 では割り込み番号 128 番をシステムコールとしています。そのため、shell は 128 番のソフトウェア割り込みを実行します (図 3.26)。



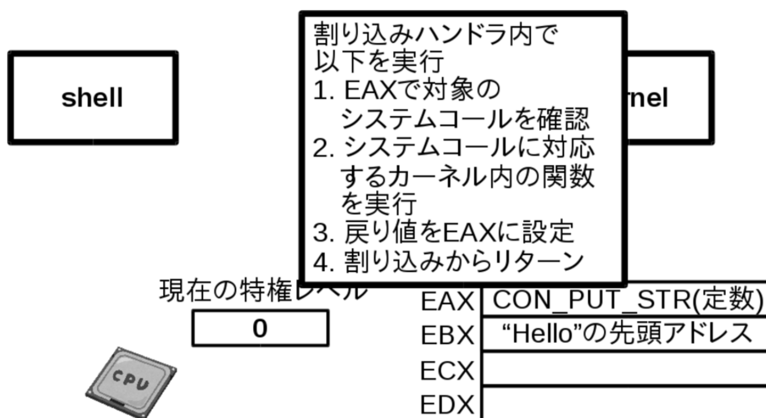
▲図 3.26　システムコール実行の流れ (3)

すると、カーネル側で 128 番の割り込みハンドラが呼び出され、同時に特権レベルが昇格するので、カーネル空間の関数を呼び出せるようになります (図 3.27)。なお、割り込みハンドラの入り口は kernel/sys.S の `syscall_handler`ラベルの箇所です (174〜193 行目)。`syscall_handler`から kernel/syscall.c の `do_syscall`関数を呼び出しています。



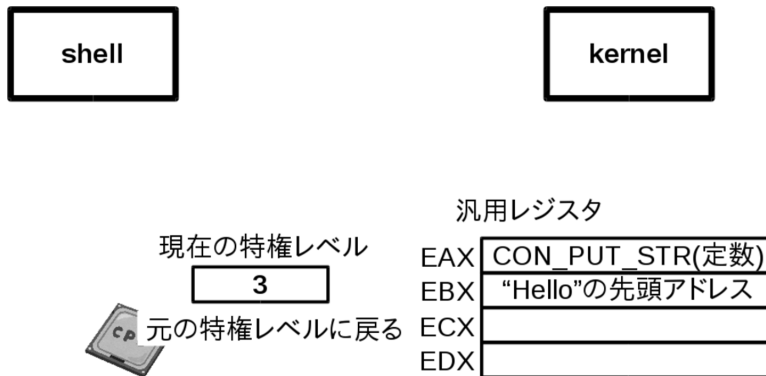
▲図 3.27　**システムコール実行の流れ** (4)

割り込みハンドラ内では、汎用レジスタに設定された値に従って、カーネル空間内の関数を呼び出します (図 3.28)。



▲図 3.28　**システムコール実行の流れ** (5)

割り込みハンドラから return すると、元の特権レベルに戻ります (図 3.29)。そして、元のアプリケーションの処理を再開します。



▲図 3.29　**システムコール実行の流れ** (6)

現状、用意しているシステムコールは表 3.1 の通りです。

▼表 3.1　**システムコール一覧**

定数名 (**番号**)	**機能**
SYSCALL_TIMER_GET_GLOBAL_COUNTER(1)	タイマカウンタ取得
SYSCALL_SCHED_WAKEUP_MSEC(2)	ウェイクアップ時間 (ms) 設定
SYSCALL_SCHED_WAKEUP_EVENT(3)	ウェイクアップイベント設定
SYSCALL_CON_GET_CURSOR_POS_Y(4)	カーソル Y 座標取得
SYSCALL_CON_PUT_STR(5)	コンソールへ文字列出力 (座標指定なし)
SYSCALL_CON_PUT_STR_POS(6)	コンソールへ文字列出力 (座標指定あり)
SYSCALL_CON_DUMP_HEX(7)	コンソールへ 16 進で数値出力 (座標指定なし)
SYSCALL_CON_DUMP_HEX_POS(8)	コンソールへ 16 進で数値出力 (座標指定あり)
SYSCALL_CON_GET_LINE(9)	コンソール入力を 1 行取得
SYSCALL_OPEN(10)	ファイルオープン
SYSCALL_EXEC(11)	ファイル実行
SYSCALL_EXIT(12)	タスク終了

3.7　デバイスドライバ

kernel/include/cpu.h

▼リスト 3.23　kernel/include/cpu.h

```
 1: #ifndef _CPU_H_
 2: #define _CPU_H_
 3:
 4: #include <asm/cpu.h>
 5:
 6: #define X86_EFLAGS_IF      0x00000200
 7: #define GDT_KERN_DS_OFS    0x0010
 8:
 9: #define sti()      __asm__ ("sti"::)
10: #define cli()      __asm__ ("cli"::)
11: #define x86_get_eflags()   ({                      \
12: unsigned int _v;                                     \
13: __asm__ volatile ("\tpushf\n"                              \
14:               "\tpopl      %0\n":"=r"(_v):);     \
15: _v;                                                  \
16: })
17: #define x86_get_tr()               ({                      \
18: unsigned short _v;                                   \
19: __asm__ volatile ("\tstr   %0\n":"=r"(_v):);     \
20: _v;                                                  \
21: })
22: #define x86_halt() __asm__ ("hlt"::)
23:
24: struct segment_descriptor {
25:     union {
26:             struct {
27:                     unsigned int a;
28:                     unsigned int b;
29:             };
30:             struct {
31:                     unsigned short limit0;
32:                     unsigned short base0;
33:                     unsigned short base1: 8, type: 4, s: 1, dpl: 2, p: 1;
34:                     unsigned short limit1: 4, avl: 1, l: 1, d: 1, g: 1,
35:                             base2: 8;
36:             };
37:     };
38: };
39:
40: struct tss {
41:     unsigned short          back_link, __blh;
42:     unsigned int            esp0;
43:     unsigned short          ss0, __ss0h;
44:     unsigned int            esp1;
45:     unsigned short          ss1, __ss1h;
46:     unsigned int            esp2;
47:     unsigned short          ss2, __ss2h;
48:     unsigned int            __cr3;
49:     unsigned int            eip;
50:     unsigned int            eflags;
51:     unsigned int            eax;
52:     unsigned int            ecx;
53:     unsigned int            edx;
54:     unsigned int            ebx;
55:     unsigned int            esp;
56:     unsigned int            ebp;
57:     unsigned int            esi;
58:     unsigned int            edi;
```

```
59:     unsigned short          es, __esh;
60:     unsigned short          cs, __csh;
61:     unsigned short          ss, __ssh;
62:     unsigned short          ds, __dsh;
63:     unsigned short          fs, __fsh;
64:     unsigned short          gs, __gsh;
65:     unsigned short          ldt, __ldth;
66:     unsigned short          trace;
67:     unsigned short          io_bitmap_base;
68: };
69:
70: extern struct segment_descriptor gdt[GDT_SIZE];
71:
72: void init_gdt(unsigned int idx, unsigned int base, unsigned int limit,
73:           unsigned char dpl);
74:
75: #endif /* _CPU_H_ */
```

x86 CPU に依存するインラインアセンブラのマクロや、構造体等を定義しています。

kernel/cpu.c

▼リスト 3.24　kernel/cpu.c

```
 1: #include <cpu.h>
 2:
 3: void init_gdt(unsigned int idx, unsigned int base, unsigned int limit,
 4:           unsigned char dpl)
 5: {
 6:     gdt[idx].limit0 = limit & 0x0000ffff;
 7:     gdt[idx].limit1 = (limit & 0x000f0000) >> 16;
 8:
 9:     gdt[idx].base0 = base & 0x0000ffff;
10:     gdt[idx].base1 = (base & 0x00ff0000) >> 16;
11:     gdt[idx].base2 = (base & 0xff000000) >> 24;
12:
13:     gdt[idx].dpl = dpl;
14:
15:     gdt[idx].type = 9;
16:     gdt[idx].p = 1;
17: }
```

x86 CPU に依存する処理を記述しているソースファイルです。今のところ、GDT へのエントリ追加を行う init_gdt関数のみです。

kernel/include/io_port.h

▼リスト 3.25　kernel/include/io_port.h

```
 1: #ifndef _IO_PORT_H_
 2: #define _IO_PORT_H_
 3:
 4: #define outb(value, port)                               \
 5: __asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))
 6:
 7: #define inb(port) ({ \
 8: unsigned char _v; \
 9: __asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
10: _v; \
11: })
12:
13: #define outb_p(value, port) \
14: __asm__ ("outb %%al,%%dx\n" \
15:             "\tjmp 1f\n" \
16:             "1:\tjmp 1f\n" \
17:             "1:"::"a" (value),"d" (port))
18:
19: #define inb_p(port) ({ \
20: unsigned char _v; \
21: __asm__ volatile ("inb %%dx,%%al\n" \
22:     "\tjmp 1f\n" \
23:     "1:\tjmp 1f\n" \
24:     "1:":"=a" (_v):"d" (port)); \
25: _v; \
26: })
27:
28: #endif /* _IO_PORT_H_ */
```

x86 CPU は I/O のアドレス空間は分かれており、I/O へのアクセスには in、outという命令があります。このソースファイルではこれらの命令をインラインアセンブラでマクロ化しています。

kernel/include/console_io.h

▼リスト 3.26　kernel/include/console_io.h

```
 1: #ifndef _CONSOLE_IO_H_
 2: #define _CONSOLE_IO_H_
 3:
 4: #define IOADR_KBC_DATA                                  0x0060
 5: #define IOADR_KBC_DATA_BIT_BRAKE   0x80
 6: #define IOADR_KBC_STATUS                        0x0064
 7: #define IOADR_KBC_STATUS_BIT_OBF   0x01
 8:
 9: #define COLUMNS                                                 80
10: #define ROWS                                                    25
11:
12: #define ASCII_ESC                                       0x1b
13: #define ASCII_BS                                        0x08
14: #define ASCII_HT                                        0x09
15:
```

```
16: #ifndef COMPILE_APP
17:
18: #define INTR_IR_KB                              1
19: #define INTR_NUM_KB                                     33
20: #define INTR_MASK_BIT_KB                0x02
21: #define SCREEN_START                            0xb8000
22: #define ATTR                                            0x07
23: #define CHATT_CNT                               1
24:
25: struct cursor_position {
26:     unsigned int x, y;
27: };
28:
29: extern unsigned char keyboard_handler;
30: extern struct cursor_position cursor_pos;
31:
32: void con_init(void);
33: void update_cursor(void);
34: void put_char_pos(char c, unsigned char x, unsigned char y);
35: void put_char(char c);
36: void put_str(char *str);
37: void put_str_pos(char *str, unsigned char x, unsigned char y);
38: void dump_hex(unsigned int val, unsigned int num_digits);
39: void dump_hex_pos(unsigned int val, unsigned int num_digits,
40:               unsigned char x, unsigned char y);
41: unsigned char get_keydata_noir(void);
42: unsigned char get_keydata(void);
43: unsigned char get_keycode(void);
44: unsigned char get_keycode_pressed(void);
45: unsigned char get_keycode_released(void);
46: char get_char(void);
47: unsigned int get_line(char *buf, unsigned int buf_size);
48:
49: #endif /* COMPILE_APP */
50:
51: #endif /* _CONSOLE_IO_H_ */
```

コンソールドライバのヘッダファイルです。

`#ifndef COMPILE_APP`では、ユーザーランド側で include された場合に関数定義等を参照させないようにしています。コンソールドライバは、`CON_PUT_STR`システムコール等でユーザーランドから呼び出します。システムコールにパラメータを与える際に、コンソール 1 画面の行数・列数等が必要になるため、それらの情報は kernel/以下のヘッダファイルを include させるようにしています。

ただし関数などは、アプリケーションレベルの特権レベルで動作しているユーザーランドからはアクセスできないので、`ifndef`で無効化しています。

kernel/console_io.c

▼**リスト** 3.27　kernel/console_io.c

```
 1: #include <cpu.h>
 2: #include <intr.h>
 3: #include <io_port.h>
 4: #include <console_io.h>
 5: #include <sched.h>
 6: #include <lock.h>
 7: #include <kernel.h>
 8:
 9: #define QUEUE_BUF_SIZE     256
10:
11: const char keymap[] = {
12:     0x00, ASCII_ESC, '1', '2', '3', '4', '5', '6',
13:     '7', '8', '9', '0', '-', '^', ASCII_BS, ASCII_HT,
14:     'q', 'w', 'e', 'r', 't', 'y', 'u', 'i',
15:     'o', 'p', '@', '[', '\n', 0x00, 'a', 's',
16:     'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',
17:     ':', 0x00, 0x00, ']', 'z', 'x', 'c', 'v',
18:     'b', 'n', 'm', ',', '.', '/', 0x00, '*',
19:     0x00, ' ', 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
20:     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, '7',
21:     '8', '9', '-', '4', '5', '6', '+', '1',
22:     '2', '3', '0', '.', 0x00, 0x00, 0x00, 0x00,
23:     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
24:     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
25:     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
26:     0x00, 0x00, 0x00, '_', 0x00, 0x00, 0x00, 0x00,
27:     0x00, 0x00, 0x00, 0x00, 0x00, '\\', 0x00, 0x00
28: };
29:
30: struct queue {
31:     unsigned char buf[QUEUE_BUF_SIZE];
32:     unsigned char start, end;
33:     unsigned char is_full;
34: } keycode_queue;
35:
36: struct cursor_position cursor_pos;
37:
38: static unsigned char error_status;
39:
40: static void enqueue(struct queue *q, unsigned char data)
41: {
42:     unsigned char if_bit;
43:
44:     if (q->is_full) {
45:             error_status = 1;
46:     } else {
47:             error_status = 0;
48:             kern_lock(&if_bit);
49:             q->buf[q->end] = data;
50:             q->end++;
51:             if (q->start == q->end) q->is_full = 1;
52:             kern_unlock(&if_bit);
53:     }
54: }
55:
56: static unsigned char dequeue(struct queue *q)
57: {
58:     unsigned char data = 0;
```

```
59:     unsigned char if_bit;
60:
61:     kern_lock(&if_bit);
62:     if (!q->is_full && (q->start == q->end)) {
63:             error_status = 1;
64:     } else {
65:             error_status = 0;
66:             data = q->buf[q->start];
67:             q->start++;
68:             q->is_full = 0;
69:     }
70:     kern_unlock(&if_bit);
71:
72:     return data;
73: }
74:
75: void do_ir_keyboard(void)
76: {
77:     unsigned char status, data;
78:
79:     status = inb_p(IOADR_KBC_STATUS);
80:     if (status & IOADR_KBC_STATUS_BIT_OBF) {
81:             data = inb_p(IOADR_KBC_DATA);
82:             enqueue(&keycode_queue, data);
83:     }
84:     sched_update_wakeupevq(EVENT_TYPE_KBD);
85:     outb_p(IOADR_MPIC_OCW2_BIT_MANUAL_EOI | INTR_IR_KB,
86:             IOADR_MPIC_OCW2);
87: }
88:
89: void con_init(void)
90: {
91:     keycode_queue.start = 0;
92:     keycode_queue.end = 0;
93:     keycode_queue.is_full = 0;
94:     error_status = 0;
95: }
96:
97: void update_cursor(void)
98: {
99:     unsigned int cursor_address = (cursor_pos.y * 80) + cursor_pos.x;
100:    unsigned char cursor_address_msb = (unsigned char)(cursor_address >> 8);
101:    unsigned char cursor_address_lsb = (unsigned char)cursor_address;
102:    unsigned char if_bit;
103:
104:    kern_lock(&if_bit);
105:    outb_p(0x0e, 0x3d4);
106:    outb_p(cursor_address_msb, 0x3d5);
107:    outb_p(0x0f, 0x3d4);
108:    outb_p(cursor_address_lsb, 0x3d5);
109:    kern_unlock(&if_bit);
110:
111:    if (cursor_pos.y >= ROWS) {
112:            unsigned int start_address = (cursor_pos.y - ROWS + 1) * 80;
113:            unsigned char start_address_msb =
114:                    (unsigned char)(start_address >> 8);
115:            unsigned char start_address_lsb = (unsigned char)start_address;
116:
```

```
117:            kern_lock(&if_bit);
118:            outb_p(0x0c, 0x3d4);
119:            outb_p(start_address_msb, 0x3d5);
120:            outb_p(0x0d, 0x3d4);
121:            outb_p(start_address_lsb, 0x3d5);
122:            kern_unlock(&if_bit);
123:    }
124: }
125:
126: void put_char_pos(char c, unsigned char x, unsigned char y)
127: {
128:    unsigned char *pos;
129:
130:    pos = (unsigned char *)(SCREEN_START + (((y * COLUMNS) + x) * 2));
131:    *(unsigned short *)pos = (unsigned short)((ATTR << 8) | c);
132: }
133:
134: void put_char(char c)
135: {
136:    switch (c) {
137:    case '\r':
138:            cursor_pos.x = 0;
139:            break;
140:
141:    case '\n':
142:            cursor_pos.y++;
143:            break;
144:
145:    default:
146:            put_char_pos(c, cursor_pos.x, cursor_pos.y);
147:            if (cursor_pos.x < COLUMNS - 1) {
148:                    cursor_pos.x++;
149:            } else {
150:                    cursor_pos.x = 0;
151:                    cursor_pos.y++;
152:            }
153:            break;
154:    }
155:
156:    update_cursor();
157: }
158:
159: void put_str(char *str)
160: {
161:    while (*str != '\0') {
162:            put_char(*str);
163:            str++;
164:    }
165: }
166:
167: void put_str_pos(char *str, unsigned char x, unsigned char y)
168: {
169:    while (*str != '\0') {
170:            switch (*str) {
171:            case '\r':
172:                    x = 0;
173:                    break;
174:
```

```
175:            case '\n':
176:                    y++;
177:                    break;
178:
179:            default:
180:                    put_char_pos(*str, x, y);
181:                    if (x < COLUMNS - 1) {
182:                            x++;
183:                    } else {
184:                            x = 0;
185:                            y++;
186:                    }
187:                    break;
188:            }
189:            str++;
190:    }
191: }
192:
193: void dump_hex(unsigned int val, unsigned int num_digits)
194: {
195:    unsigned int new_x = cursor_pos.x + num_digits;
196:    unsigned int dump_digit = new_x - 1;
197:
198:    while (num_digits) {
199:            unsigned char tmp_val = val & 0x0000000f;
200:            if (tmp_val < 10) {
201:                    put_char_pos('0' + tmp_val, dump_digit, cursor_pos.y);
202:            } else {
203:                    put_char_pos('A' + tmp_val - 10, dump_digit, cursor_pos.y);
204:            }
205:            val >>= 4;
206:            dump_digit--;
207:            num_digits--;
208:    }
209:
210:    cursor_pos.x = new_x;
211:
212:    update_cursor();
213: }
214:
215: void dump_hex_pos(unsigned int val, unsigned int num_digits,
216:                   unsigned char x, unsigned char y)
217: {
218:    unsigned int new_x = x + num_digits;
219:    unsigned int dump_digit = new_x - 1;
220:
221:    while (num_digits) {
222:            unsigned char tmp_val = val & 0x0000000f;
223:            if (tmp_val < 10) {
224:                    put_char_pos('0' + tmp_val, dump_digit, y);
225:            } else {
226:                    put_char_pos('A' + tmp_val - 10, dump_digit, y);
227:            }
228:            val >>= 4;
229:            dump_digit--;
230:            num_digits--;
231:    }
232: }
233:
```

```
234: unsigned char get_keydata_noir(void)
235: {
236: 	while (!(inb_p(IOADR_KBC_STATUS) & IOADR_KBC_STATUS_BIT_OBF));
237: 	return inb_p(IOADR_KBC_DATA);
238: }
239:
240: unsigned char get_keydata(void)
241: {
242: 	unsigned char data;
243: 	unsigned char dequeuing = 1;
244: 	unsigned char if_bit;
245:
246: 	while (dequeuing) {
247: 		kern_lock(&if_bit);
248: 		data = dequeue(&keycode_queue);
249: 		if (!error_status)
250: 			dequeuing = 0;
251: 		kern_unlock(&if_bit);
252: 		if (dequeuing)
253: 			wakeup_after_event(EVENT_TYPE_KBD);
254: 	}
255:
256: 	return data;
257: }
258:
259: unsigned char get_keycode(void)
260: {
261: 	return get_keydata() & ~IOADR_KBC_DATA_BIT_BRAKE;
262: }
263:
264: unsigned char get_keycode_pressed(void)
265: {
266: 	unsigned char keycode;
267: 	while ((keycode = get_keydata()) & IOADR_KBC_DATA_BIT_BRAKE);
268: 	return keycode & ~IOADR_KBC_DATA_BIT_BRAKE;
269: }
270:
271: unsigned char get_keycode_released(void)
272: {
273: 	unsigned char keycode;
274: 	while (!((keycode = get_keydata()) & IOADR_KBC_DATA_BIT_BRAKE));
275: 	return keycode & ~IOADR_KBC_DATA_BIT_BRAKE;
276: }
277:
278: char get_char(void)
279: {
280: 	return keymap[get_keycode_pressed()];
281: }
282:
283: unsigned int get_line(char *buf, unsigned int buf_size)
284: {
285: 	unsigned int i;
286:
287: 	for (i = 0; i < buf_size - 1;) {
288: 		buf[i] = get_char();
289: 		if (buf[i] == ASCII_BS) {
290: 			if (i == 0) continue;
291: 			cursor_pos.x--;
```

```
292:                    update_cursor();
293:                    put_char_pos(' ', cursor_pos.x, cursor_pos.y);
294:                    i--;
295:            } else {
296:                    put_char(buf[i]);
297:                    if (buf[i] == '\n') {
298:                            put_char('\r');
299:                            break;
300:                    }
301:                    i++;
302:            }
303:    }
304:    buf[i] = '\0';
305:
306:    return i;
307: }
```

コンソールのデバイスドライバです。OS5 ではキーボード入力とテキストモードでの画面出力をコンソールとして抽象化しています。そのため、このソースファイルでキー入力と画面出力を共に扱っています。なお、単体のソースファイルの行数としては boot/boot.s の 328 行に次いで 2 番目に長いソースファイルです (307 行)。

このソースファイル内で重要なのは get_char関数と、put_char関数です。1 行分の入力を取得する get_line関数や文字列を画面表示する put_str関数は get_char関数と put_char関数を内部で呼び出しているため、ここでは get_char関数と put_char関数の動作の流れを説明します。

get_char関数の呼び出しによって何が起こるのかというと、キューからキーコードを含むデータ (キーデータ) を取り出し、ASCII コードへ変換し戻り値として返します。get_keycode_pressed関数が押下時のキーコードを返す関数で、keymapはキーコードを ASCII コードへ変換する配列です。キーボードコントローラ (KBC) が返すキーデータには BRAKE というビットが有り、このビットが立っている場合、該当のキーから指が離された事を示します。get_keydata_pressedでは get_keydata関数でキューから取得したキーデータに BRAKE のビットが立っている間、ブロックします (押下中を示すキーデータが取得できるまで呼び出し元へ return しない)。なお、キューにキーデータを積む関数は do_ir_keyboardです。kernel/sys.S の keyboard_handlerハンドラから呼び出されます。

put_char関数の呼び出しでは何が起こるのかというと、引数で渡された ASCII コード値をカーソル位置に対応した VRAM のアドレスへ書き込みます。グローバル変数の struct cursor_position cursor_posがカーソル位置を保持している変数で、put_char_pos関数が指定された座標の VRAM アドレスへ ASCII コードを書き込む関数です。定数 SCREEN_STARTが VRAM の先頭アドレスです。

kernel/init.c の kern_init関数からはカーソルの設定 (cursor_posの初期化と updat

e_cursor関数によるカーソル位置と表示開始位置の更新) とコンソールドライバの初期化 (con_init関数によるキーコードのキューの初期化とエラーステータスの初期化) を行っています。

失敗談: キー入力時にゴミが入る (ロックは大切)

キー入力していると、コンソール画面にゴミが出力されることがありました。
当初、全く理由が分からず、少なくとも 2 週間程、悩んでいた覚えがあります。
KBC の割り込み契機のエンキューでゴミが入っているのか、デキュー処理に問題があるのかと、問題を切り分けていきました。
結論としては、get_keydata関数内にて、dequeue関数呼び出しと、その後の error_status変数のチェックの間に KBC 割り込みが入ることがあり、その際に割り込みハンドラから呼び出されるエンキュー処理で error_status変数を上書きしてしまう事が原因でした。そのため、正しいキーデータを取得できていないのに、get_keydata関数内の while ループを抜けてしまい、get_keydata関数は正常ではないキーデータを return していました。
そのため、get_keydata関数内の dequeue関数呼び出しから error_status チェックの間は割り込み禁止 (ロック) するようにしています[a]。割り込みハンドラ内とそれ以外で同じリソース (変数等) へアクセスする場合は、正しくロックする必要がある、ということでした。なお、今見ていると、変数 error_statusをエンキュー用とデキュー用で分けても良かったと思います。

[a] https://github.com/cupnes/os5/commit/8f0ffffdf1811a150ec8e95aa6f706940c356850

_noir と名の付く関数は?

noir は"NO InteRrupt"の略で、割り込み禁止区間内 (主に割り込みハンドラ) で呼び出される事を想定した関数です。これは当初、カーネルのロック処理がネストに対応して居なかったため (kernel/lock.c で説明します) で、割り込み禁止区間内から呼び出されるか、そうでないかで関数を分けていました。今はロック機能がネストに対応しているので、_noir の関数は不要なのですが、割り込みハンドラからの呼び出しではまだ修正されずに残っています。

OS5 はフォントを持たない

OS5 では BIOS で画面モードをテキストモードに設定しています。SCREEN_STARTはテキストモードでの VRAM の先頭アドレスで、MC6845 という CRT コントローラ (CRTC) の VRAM です。
この CRTC はテキストモードでの使用が前提であるため、コントローラ内にフォントを内蔵しており、VRAM のアドレス空間へ ASCII で値を書き込むだけで画面表示ができます。カーソル位置と表示開始位置の設定も可能です。カーソル位置の設定により任意の場所にカーソルを設置できます。また、表示開始位置の設定に関しては、そもそも MC6845 は表示領域 (80x25 文字) 以上の VRAM 領域を持っており、VRAM 内のどこからを表示するかを「何文字目からスタート」という形で指定できます。これにより、表示開始位置の設定を行うだけで、画面スクロールが実現できます。
そのため、OS5 はフォントも持っていないし、画面スクロールの処理も CRTC へ設定しているだけで、ソフトウェアで処理しているわけではありません。
このように、ハードウェアがどんな機能を持っているかを知っているとソフトウェアでの実装を減らせて便利です (「ハードウェア」を「API」と読み替えても同じことですね)。

kernel/include/timer.h

▼リスト 3.28　kernel/include/timer.h

```
 1: #ifndef _TIMER_H_
 2: #define _TIMER_H_
 3:
 4: #define IOADR_PIT_COUNTER0 0x0040
 5: #define IOADR_PIT_CONTROL_WORD     0x0043
 6: #define IOADR_PIT_CONTROL_WORD_BIT_COUNTER0                0x00
 7: #define IOADR_PIT_CONTROL_WORD_BIT_16BIT_READ_LOAD 0x30
 8: #define IOADR_PIT_CONTROL_WORD_BIT_MODE2           0x04
 9:                                              /* Rate Generator */
10:
11: #define INTR_IR_TIMER               0
12: #define INTR_NUM_TIMER              32
13: #define INTR_MASK_BIT_TIMER         0x01
14:
15: #define TIMER_TICK_MS               10
16:
```

```
17: extern unsigned char timer_handler;
18: extern unsigned int global_counter;
19:
20: void timer_init(void);
21: unsigned int timer_get_global_counter(void);
22:
23: #endif /* _TIMER_H_ */
```

PIC(Programmable Interval Timer) のレジスタの IO アドレスと割り込み番号などの定数の定義と、関数のプロトタイプ宣言です。

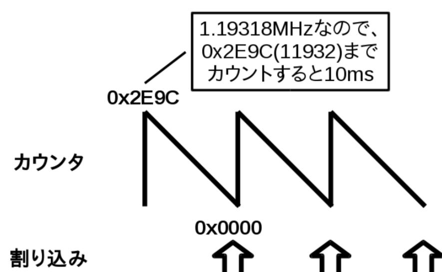
kernel/timer.c

▼リスト 3.29　kernel/timer.c

```
 1: #include <timer.h>
 2: #include <io_port.h>
 3: #include <intr.h>
 4: #include <sched.h>
 5:
 6: unsigned int global_counter = 0;
 7:
 8: void do_ir_timer(void)
 9: {
10:     global_counter += TIMER_TICK_MS;
11:     sched_update_wakeupq();
12:     if (!current_task || !current_task->task_switched_in_time_slice) {
13:             /* タイムスライス中のコンテキストスイッチではない */
14:             schedule();
15:     } else {
16:             /* タイムスライス中のコンテキストスイッチである */
17:             current_task->task_switched_in_time_slice = 0;
18:     }
19:     outb_p(IOADR_MPIC_OCW2_BIT_MANUAL_EOI | INTR_IR_TIMER, IOADR_MPIC_OCW2);
20: }
21:
22: void timer_init(void)
23: {
24:     /* Setup PIT */
25:     outb_p(IOADR_PIT_CONTROL_WORD_BIT_COUNTER0
26:            | IOADR_PIT_CONTROL_WORD_BIT_16BIT_READ_LOAD
27:            | IOADR_PIT_CONTROL_WORD_BIT_MODE2, IOADR_PIT_CONTROL_WORD);
28:     /* 割り込み周期 11932(0x2e9c) サイクル (=100Hz、10ms 毎) に設定 */
29:     outb_p(0x9c, IOADR_PIT_COUNTER0);
30:     outb_p(0x2e, IOADR_PIT_COUNTER0);
31: }
32:
33: unsigned int timer_get_global_counter(void)
34: {
35:     return global_counter;
36: }
```

タイマーの初期化と設定を行う関数群を定義しています。

`timer_init`関数が kernel/init.c の `kern_init`関数から呼ばれるタイマー初期化の関数です。OS5 では 10ms 周期の割り込みに設定しています。タイマーの挙動は図 3.30 の通りです。



▲図 3.30　タイマー割り込みの振る舞い

また、`do_ir_timer`関数が kern/sys.S の `timer_handler`ハンドラから呼ばれる関数です。

3.8　ライブラリ

kernel/include/stddef.h

▼リスト 3.30　kernel/include/stddef.h

```
1: #ifndef _STDDEF_H_
2: #define _STDDEF_H_
3:
4: #define NULL       ((void *)0)
5:
6: #endif /* _STDDEF_H_ */
```

汎用的に使用する定数の定義です。今のところ、`NULL`のみです。

kernel/include/common.h

▼リスト 3.31　kernel/include/common.h

```
 1: #ifndef _COMMON_H_
 2: #define _COMMON_H_
 3:
 4: int str_compare(const char *src, const char *dst);
 5: void copy_mem(const void *src, void *dst, unsigned int size);
 6:
 7: #endif /* _COMMON_H_ */
```

共通で使用される関数を kernel/common.c にまとめています。このヘッダファイルではプロトタイプ宣言を行っています。

kernel/common.c

▼リスト 3.32　kernel/common.c

```
 1: #include <common.h>
 2:
 3: int str_compare(const char *src, const char *dst)
 4: {
 5:     char is_equal = 1;
 6:
 7:     for (; (*src != '\0') && (*dst != '\0'); src++, dst++) {
 8:             if (*src != *dst) {
 9:                     is_equal = 0;
10:                     break;
11:             }
12:     }
13:
14:     if (is_equal) {
15:             if (*src != '\0') {
16:                     return 1;
17:             } else if (*dst != '\0') {
18:                     return -1;
19:             } else {
20:                     return 0;
21:             }
22:     } else {
23:             return (int)(*src - *dst);
24:     }
25: }
26:
27: void copy_mem(const void *src, void *dst, unsigned int size)
28: {
29:     unsigned char *d = (unsigned char *)dst;
30:     unsigned char *s = (unsigned char *)src;
31:
32:     for (; size > 0; size--) {
33:             *d = *s;
34:             d++;
35:             s++;
36:     }
37: }
```

common.c には、汎用的に使用されるような関数を集めています。

kernel/include/lock.h

▼リスト 3.33　kernel/include/lock.h

```
1: #ifndef _LOCK_H_
2: #define _LOCK_H_
3:
4: void kern_lock(unsigned char *if_bit);
5: void kern_unlock(unsigned char *if_bit);
6:
7: #endif /* _LOCK_H_ */
```

カーネルのロック機能についての関数のプロトタイプ宣言です。

kernel/lock.c

▼リスト 3.34　kernel/lock.c

```
 1: #include <lock.h>
 2: #include <cpu.h>
 3:
 4: void kern_lock(unsigned char *if_bit)
 5: {
 6:     /* Save EFlags.IF */
 7:     *if_bit = (x86_get_eflags() & X86_EFLAGS_IF) ? 1 : 0;
 8:
 9:     /* if saved IF == true, then cli */
10:     if (*if_bit)
11:             cli();
12: }
13:
14: void kern_unlock(unsigned char *if_bit)
15: {
16:     /* if saved IF == true, then sti */
17:     if (*if_bit)
18:             sti();
19: }
```

ロック機能に関するソースコードです。ロックとはある処理を実行中に、割り込みなどで CPU が別の処理を行わないようにする機能です。kern_lockでは、cli命令で割り込みを無効化し、kern_unlockでは、sti命令で割り込みを有効化します。

引数の unsigned char *if_bitは、kern_lock実行時の割り込み有効/無効状態を保持するために使用します。例えば、親の関数で kern_lockを実行していて既に割り込み

無効区間 (ロック区間) 内であるにも関わらず、子側の kern_lock〜kern_unlockで割り込みを有効化してしまうと、親側のロックに影響します。このようなネストした kern_lock/kern_unlockの呼び出しのために if_bitを使用します。

kernel/include/list.h

▼リスト 3.35　kernel/include/list.h

```
 1: #ifndef _LIST_H_
 2: #define _LIST_H_
 3:
 4: struct list {
 5:     struct list *next;
 6:     struct list *prev;
 7: };
 8:
 9: #endif /* _LIST_H_ */
```

リンクリストはカーネル内で頻繁に使用するため専用のヘッダファイルを用意しています。struct listを構造体の一つ目のメンバとすることで、構造体にリンクリストの機能を持たせることができます。例としては、kernel/include/fs.h の struct fileの定義を見てみてください。(このヘッダファイルは、kernel/include/stddef.h へまとめても良さそうな気がします。)

kernel/include/queue.h

▼リスト 3.36　kernel/include/queue.h

```
 1: #ifndef _QUEUE_H_
 2: #define _QUEUE_H_
 3:
 4: #include <list.h>
 5:
 6: void queue_init(struct list *head);
 7: void queue_enq(struct list *entry, struct list *head);
 8: void queue_del(struct list *entry);
 9: void queue_dump(struct list *head);
10:
11: #endif /* _QUEUE_H_ */
```

概要

キュー構造の関数のプロトタイプ宣言です。キュー構造はカーネル内で頻繁に登場するため、専用のヘッダファイルを用意しています。

kernel/queue.c

▼リスト 3.37　kernel/queue.c

```
 1: #include <queue.h>
 2: #include <list.h>
 3: #include <console_io.h>
 4:
 5: void queue_init(struct list *head)
 6: {
 7:     head->next = head;
 8:     head->prev = head;
 9: }
10:
11: void queue_enq(struct list *entry, struct list *head)
12: {
13:     entry->prev = head->prev;
14:     entry->next = head;
15:     head->prev->next = entry;
16:     head->prev = entry;
17: }
18:
19: void queue_del(struct list *entry)
20: {
21:     entry->prev->next = entry->next;
22:     entry->next->prev = entry->prev;
23: }
24:
25: void queue_dump(struct list *head)
26: {
27:     unsigned int n;
28:     struct list *entry;
29:
30:     put_str("h =");
31:     dump_hex((unsigned int)head, 8);
32:     put_str(": p=");
33:     dump_hex((unsigned int)head->prev, 8);
34:     put_str(", n=");
35:     dump_hex((unsigned int)head->next, 8);
36:     put_str("\r\n");
37:
38:     for (entry = head->next, n = 0; entry != head; entry = entry->next, n++) {
39:             dump_hex(n, 2);
40:             put_str("=");
41:             dump_hex((unsigned int)entry, 8);
42:             put_str(": p=");
43:             dump_hex((unsigned int)entry->prev, 8);
44:             put_str(", n=");
45:             dump_hex((unsigned int)entry->next, 8);
46:             put_str("\r\n");
47:     }
48: }
```

カーネル内で汎用的に使えるよう、ここでキュー構造を定義しています。

kernel/include/debug.h

▼リスト 3.38　kernel/include/debug.h

```
 1: #ifndef __DEBUG_H__
 2: #define __DEBUG_H__
 3:
 4: extern volatile unsigned char _flag;
 5:
 6: void debug_init(void);
 7: void test_excp_de(void);
 8: void test_excp_pf(void);
 9:
10: #endif /* __DEBUG_H__ */
```

デバッグ機能に関するフラグ変数の extern とプロトタイプ宣言があります。

kernel/debug.c

▼リスト 3.39　kernel/debug.c

```
 1: #include <debug.h>
 2:
 3: volatile unsigned char _flag;
 4:
 5: void debug_init(void)
 6: {
 7:     _flag = 0;
 8: }
 9:
10: /* Test divide by zero exception */
11: void test_excp_de(void)
12: {
13:     __asm__("\tmovw        $8, %%ax\n" \
14:             "\tmovb        $0, %%bl\n" \
15:             "\tdivb        %%bl"::);
16: }
17:
18: /* Test page fault exception */
19: void test_excp_pf(void)
20: {
21:     volatile unsigned char tmp;
22:     __asm__("movb   0x000b8000, %0":"=r"(tmp):);
23: }
```

カーネルデバッグのための機能です。デバッグフラグ`_flag`は、debug.h を include して 1 をセットすると、カーネルのデバッグログをコンソールへ出力するように用意しています。(ただし、`_flag`は誰も使っていないです。。)

第 4 章

ユーザーランド

アプリケーションとしては、shell と uptime、whoareyou という 3 つです。現状、カーネルの動作確認程度のものでしかないです。

shell はその名の通りシェルで、CUI を提供します。shell の組み込みコマンドとしては、echo とメモリ/IO への直接 read/write のコマンド (readb,readw,readl,ioreadb,write も同様のコマンド名)、そして bg というコマンドがあります。bg は今回のリリースで追加したコマンドで、引数で指定したコマンドをバックグラウンド実行します。(これまでは、実行したコマンドの終了を待つことができなかったので、常にバックグラウンド実行でした。)uptime・whoareyou も shell から起動します。これらのコマンド名を shell 上で入力すると、shell は exec システムコール (OS5 では exec をシステムコールとしています) を使用して実行します。

uptime はマルチタスクの動作確認をするためのコマンドです。コンソール画面の右上で、16 進数で起動時間をカウントし続けます。自ら終了することがないコマンドなので、バックグラウンド実行しないとプロンプトが帰ってこなくなります。

whoareyou は今回のリリースで新たに追加したコマンドです。argc と argv によりコマンドライン引数を受け取れることを確認するためのコマンドです。

また、今回のリリースでは、main() をエントリポイントとする変更や、静的ライブラリの仕組みも導入しており、よく見る C のソースコードのようにアプリケーションを書けるようになりました。

話は変わって、ユーザーランドのファイルシステムイメージは、make の過程で、シェルスクリプトで作成します。ファイルシステムは、簡単に、ファイル名とバイナリのみを管理するだけのもので、シェルスクリプトでバイナリを並べて連結しています。ファイルシステムについても詳しくは上述のスライドをご覧ください。

これまでをまとめると、アプリケーションが実行されるまでの流れは以下のとおりです。

1. ファイルシステムイメージをブートローダーが RAM 上の決まったアドレスへロード
2. カーネルは、初期化の過程でファイルシステムが配置されている RAM 上の領域をチェック
3. カーネルは、ファイルシステム上の 1 つ目のファイルを、カーネル起動後に実行する最初のアプリとして実行する (ここで shell が実行される)

4.1　共通部分

apps/Makefile

▼リスト 4.1　apps/Makefile

```
 1: LIB_DIR = .lib
 2: BIN_DIR = .bin
 3: APP_LD = ../app.ld
 4: LIB_DIRS = libkernel libcommon libconsole libstring
 5: APP_DIRS = $(shell find . -maxdepth 1 -type d '!' -iname '.*' '!' \
 6: -iname 'include' '!' -iname 'lib*')
 7: CFLAGS = -Wall -Wextra
 8: CFLAGS += -nostdinc -nostdlib -fno-builtin -c
 9: CFLAGS += -I../include
10: CFLAGS += -m32
11: CFLAGS += -DCOMPILE_APP
12: LDFLAGS = -L../$(LIB_DIR)
13: LIBS = -lstring -lconsole -lcommon -lkernel
14:
15: apps.img: lib app
16:     ../tools/make_os5_fs.sh $(BIN_DIR)/* > $@
17:
18: lib: $(LIB_DIRS)
19:     [ -d $(LIB_DIR) ] || mkdir $(LIB_DIR)
20:     for libdir in $^; do \
21:             make -C $$libdir LIB_DIR=../$(LIB_DIR)    \
22:             APP_LD=$(APP_LD) CFLAGS="$(CFLAGS)"     \
23:             LDFLAGS="$(LDFLAGS)" LIBS="$(LIBS)"; \
24:     done
25:
26: app: $(APP_DIRS)
27:     [ -d $(BIN_DIR) ] || mkdir $(BIN_DIR)
28:     for appdir in $^; do \
29:             make -C $$appdir BIN_DIR=../$(BIN_DIR)    \
30:             APP_LD=$(APP_LD) CFLAGS="$(CFLAGS)"     \
31:             LDFLAGS="$(LDFLAGS)" LIBS="$(LIBS)"; \
32:     done
33:
34: clean:
35:     rm -rf *~ *.o *.a *.bin *.dat *.img *.map $(LIB_DIR) $(BIN_DIR)
36:     for dir in $(LIB_DIRS) $(APP_DIRS); do \
37:             make -C $$dir clean; \
```

```
38:     done
39:
40: .PHONY: lib app clean
```

ユーザーランド上の各アプリケーションのコンパイルを行う大元の Makefile です。

この Makefile での最終生成物は apps.img です。apps.img を生成するために、ライブラリのコンパイル (`lib`ターゲット)、アプリケーションのコンパイル (`app`ターゲット)、tools/make_os5_fs.sh でファイルシステム生成 (`apps.img`ターゲット) という流れです。

apps/app.ld

▼リスト 4.2　apps/app.ld

```
 1: OUTPUT_FORMAT("binary");
 2:
 3: SECTIONS
 4: {
 5:     . = 0x20000030;
 6:     .text   : {
 7:             *(.entry)
 8:             *(.text)
 9:     }
10:     .rodata : {
11:             *(.strings)
12:             *(.rodata)
13:             *(.rodata.*)
14:     }
15:     .data   : {*(.data)}
16:     .bss    : {*(.bss)}
17: }
```

アプリケーションのリンカスクリプトです。仮想アドレス空間上、ユーザー空間は 0x2000 0000 からで、先頭 48(0x30) バイトがファイルシステムのヘッダなので、0x2000 0030 から text セクション (コードの領域) を配置しています。そのため、スケジューラでもタスクのエントリアドレスは 0x2000 0030 としています (kernel/task.c の`#define APP_ENTRY_POINT0x20000030`)。

apps/include/app.h

▼リスト 4.3　apps/include/app.h

```
 1: #ifndef _APP_H_
 2: #define _APP_H_
 3:
 4: int main(int argc, char *argv[]) __attribute__((section(".entry")));
 5:
 6: #endif /* _APP_H_ */
```

エントリ関数として main 関数のプロトタイプ宣言をしています。

4.2 0shell

apps/0shell/Makefile

▼リスト 4.4 apps/0shell/Makefile

```
 1: NAME=0shell
 2:
 3: $(BIN_DIR)/$(NAME): $(NAME).o
 4:     ld -m elf_i386 -o $@ $< -Map $(NAME).map -s -T $(APP_LD) -x       \
 5:     $(LDFLAGS) $(LIBS)
 6:
 7: %.o: %.c
 8:     gcc $(CFLAGS) -o $@ $<
 9:
10: clean:
11:     rm -rf *~ *.o *.map
12:
13: .PHONY: clean
```

シェルアプリケーション (0shell) の Makefile です。Makefile はアプリケーションごとに存在しますが、内容は `NAME=0shell`の行を除いて共通です。

apps/0shell/0shell.c

▼リスト 4.5 apps/0shell/0shell.c

```
 1: #include <app.h>
 2: #include <kernel.h>
 3: #include <common.h>
 4: #include <string.h>
 5: #include <console.h>
 6:
 7: #define MAX_LINE_SIZE      512
 8:
 9: enum {
10:     ECHO,
```

```
11:     READB,
12:     READW,
13:     READL,
14:     IOREADB,
15:     WRITEB,
16:     WRITEW,
17:     WRITEL,
18:     IOWRITEB,
19:     BG,
20: #ifdef DEBUG
21:     TEST,
22: #endif /* DEBUG */
23:     COMMAND_NUM
24: } _COMMAND_SET;
25:
26: static int command_echo(char *args)
27: {
28:     put_str(args);
29:     put_str("\r\n");
30:
31:     return 0;
32: }
33:
34: static int command_readb(char *args)
35: {
36:     char first[128], other[128];
37:     unsigned char *addr;
38:
39:     str_get_first_entry(args, first, other);
40:     addr = (unsigned char *)str_conv_ahex_int(first);
41:     dump_hex(*addr, 2);
42:     put_str("\r\n");
43:
44:     return 0;
45: }
46:
47: static int command_readw(char *args)
48: {
49:     char first[128], other[128];
50:     unsigned short *addr;
51:
52:     str_get_first_entry(args, first, other);
53:     addr = (unsigned short *)str_conv_ahex_int(first);
54:     dump_hex(*addr, 4);
55:     put_str("\r\n");
56:
57:     return 0;
58: }
59:
60: static int command_readl(char *args)
61: {
62:     char first[128], other[128];
63:     unsigned int *addr;
64:
65:     str_get_first_entry(args, first, other);
66:     addr = (unsigned int *)str_conv_ahex_int(first);
67:     dump_hex(*addr, 8);
68:     put_str("\r\n");
69:
```

```
70:     return 0;
71: }
72:
73: static int command_ioreadb(char *args)
74: {
75:     char first[128], other[128];
76:     unsigned short addr;
77:
78:     str_get_first_entry(args, first, other);
79:     addr = (unsigned short)str_conv_ahex_int(first);
80:     dump_hex(inb_p(addr), 2);
81:     put_str("\r\n");
82:
83:     return 0;
84: }
85:
86: static int command_writeb(char *args)
87: {
88:     char first[16], second[32], other[128], _other[128];
89:     unsigned char data, *addr;
90:
91:     str_get_first_entry(args, first, other);
92:     str_get_first_entry(other, second, _other);
93:     data = (unsigned char)str_conv_ahex_int(first);
94:     addr = (unsigned char *)str_conv_ahex_int(second);
95:     *addr = data;
96:
97:     return 0;
98: }
99:
100: static int command_writew(char *args)
101: {
102:    char first[16], second[32], other[128], _other[128];
103:    unsigned short data, *addr;
104:
105:    str_get_first_entry(args, first, other);
106:    str_get_first_entry(other, second, _other);
107:    data = (unsigned short)str_conv_ahex_int(first);
108:    addr = (unsigned short *)str_conv_ahex_int(second);
109:    *addr = data;
110:
111:    return 0;
112: }
113:
114: static int command_writel(char *args)
115: {
116:    char first[16], second[32], other[128], _other[128];
117:    unsigned int data, *addr;
118:
119:    str_get_first_entry(args, first, other);
120:    str_get_first_entry(other, second, _other);
121:    data = (unsigned int)str_conv_ahex_int(first);
122:    addr = (unsigned int *)str_conv_ahex_int(second);
123:    *addr = data;
124:
125:    return 0;
126: }
127:
```

```
128: static int command_iowriteb(char *args)
129: {
130:    char first[16], second[32], other[128], _other[128];
131:    unsigned char data;
132:    unsigned short addr;
133:
134:    str_get_first_entry(args, first, other);
135:    str_get_first_entry(other, second, _other);
136:    data = (unsigned char)str_conv_ahex_int(first);
137:    addr = (unsigned short)str_conv_ahex_int(second);
138:    outb_p(data, addr);
139:
140:    return 0;
141: }
142:
143: #ifdef DEBUG
144: static int command_test(char *args)
145: {
146:    put_str("test\r\n");
147:
148:    return 0;
149: }
150: #endif /* DEBUG */
151:
152: static unsigned char get_command_id(const char *command)
153: {
154:    if (!str_compare(command, "echo")) {
155:            return ECHO;
156:    }
157:
158:    if (!str_compare(command, "readb")) {
159:            return READB;
160:    }
161:
162:    if (!str_compare(command, "readw")) {
163:            return READW;
164:    }
165:
166:    if (!str_compare(command, "readl")) {
167:            return READL;
168:    }
169:
170:    if (!str_compare(command, "ioreadb")) {
171:            return IOREADB;
172:    }
173:
174:    if (!str_compare(command, "writeb")) {
175:            return WRITEB;
176:    }
177:
178:    if (!str_compare(command, "writew")) {
179:            return WRITEW;
180:    }
181:
182:    if (!str_compare(command, "writel")) {
183:            return WRITEL;
184:    }
185:
```

```
186:    if (!str_compare(command, "iowriteb")) {
187:            return IOWRITEB;
188:    }
189:
190:    if (!str_compare(command, "bg")) {
191:            return BG;
192:    }
193:
194: #ifdef DEBUG
195:    if (!str_compare(command, "test")) {
196:            return TEST;
197:    }
198: #endif /* DEBUG */
199:
200:    return COMMAND_NUM;
201: }
202:
203: int main(int argc __attribute__ ((unused)),
204:      char *argv[] __attribute__ ((unused)))
205: {
206:    while (1) {
207:            char buf[MAX_LINE_SIZE];
208:            char command[256], args[256];
209:            unsigned char command_id, is_background = 0;
210:            unsigned int fp;
211:
212:            put_str("OS5> ");
213:            if (get_line(buf, MAX_LINE_SIZE) <= 0) {
214:                    continue;
215:            }
216:
217:            while (1) {
218:                    str_get_first_entry(buf, command, args);
219:                    command_id = get_command_id(command);
220:                    if (command_id != BG)
221:                            break;
222:                    else {
223:                            is_background = 1;
224:                            copy_mem(args, buf,
225:                                     (unsigned int)str_get_len(args));
226:                    }
227:            }
228:
229:            switch (command_id) {
230:            case ECHO:
231:                    command_echo(args);
232:                    break;
233:            case READB:
234:                    command_readb(args);
235:                    break;
236:            case READW:
237:                    command_readw(args);
238:                    break;
239:            case READL:
240:                    command_readl(args);
241:                    break;
242:            case IOREADB:
243:                    command_ioreadb(args);
```

```
244:                    break;
245:            case WRITEB:
246:                    command_writeb(args);
247:                    break;
248:            case WRITEW:
249:                    command_writew(args);
250:                    break;
251:            case WRITEL:
252:                    command_writel(args);
253:                    break;
254:            case IOWRITEB:
255:                    command_iowriteb(args);
256:                    break;
257: #ifdef DEBUG
258:            case TEST:
259:                    command_test(args);
260:                    break;
261: #endif /* DEBUG */
262:            default:
263:                    fp = syscall(SYSCALL_OPEN, (unsigned int)command, 0, 0);
264:                    if (fp) {
265:                            unsigned int argc = 0, i;
266:                            char *argv[256];
267:                            char *start;
268:
269:                            argv[argc++] = command;
270:
271:                            start = &args[0];
272:                            for (i = 0; ; i++) {
273:                                    if ((i == 0) && (args[i] == '\0')) {
274:                                            break;
275:                                    } else if ((args[i] == ' ') ||
276:                                               (args[i] == '\0')) {
277:                                            argv[argc++] = start;
278:                                            start = &args[i + 1];
279:                                            if (args[i] == ' ')
280:                                                    args[i] = '\0';
281:                                            else
282:                                                    break;
283:                                    }
284:                            }
285:
286:                            syscall(SYSCALL_EXEC, fp, argc,
287:                                    (unsigned int)argv);
288:
289:                            if (!is_background)
290:                                    syscall(SYSCALL_SCHED_WAKEUP_EVENT,
291:                                            EVENT_TYPE_EXIT, 0, 0);
292:                    } else
293:                            put_str("Command not found.\r\n");
294:                    break;
295:            }
296:    }
297: }
```

シェルアプリケーションの本体のソースコードです。OS5 の中で 3 番目に長いソースコードで 297 行あります。

include している各ヘッダファイルについては以下の通りです。

app.h
: main関数を使用するため

kernel.h
: syscall関数を使用するため実体は apps/libkernel/libkernel.c にある

common.h
: copy_mem関数を使用するため実体は apps/libcommon/libcommon.c にある

string.h
: 文字列操作関数を使用するため実体は apps/libstring/libstring.c にある

console.h
: コンソール操作関数を使用するため実体は apps/libconsole/libconsole.c にある

main 関数の処理をざっくりと説明すると以下の通りです。

1. プロンプトを表示し、get_line関数で 1 行分の入力文字列を取得するまで待機 (212〜215 行目)
2. 1. の文字列から str_get_first_entry関数でコマンド名を切り出し、get_command_id関数でシェル組み込みコマンドの ID を取得。取得したコマンドが"bg"コマンドであった場合は、is_backgroundフラグをセットする (217〜227 行目)
3. コマンド ID に応じた処理を呼び出す (229〜295 行目)。組み込みコマンドではなかった場合、実行ファイルとみなし、open と exec を行う (263〜294 行目)

なお、3. でファイルを実行するときに、2. で is_backgroundがセットされていなければ、ファイルの実行が終了するまでシェルをスリープさせます (289〜291 行目)。これにより、バックグラウンド実行していない場合は、実行が終了するまでプロンプトが戻ってこないようにしています。

4.3 uptime

apps/uptime/Makefile

▼リスト 4.6 apps/uptime/Makefile

```
1: NAME=uptime
2:
3: $(BIN_DIR)/$(NAME): $(NAME).o
4:     ld -m elf_i386 -o $@ $< -Map $(NAME).map -s -T $(APP_LD) -x       \
5:     $(LDFLAGS) $(LIBS)
```

```
 6:
 7: %.o: %.c
 8: 	gcc $(CFLAGS) -o $@ $<
 9:
10: clean:
11: 	rm -rf *~ *.o *.map
12:
13: .PHONY: clean
```

uptime アプリケーションの Makefile です。

apps/uptime/uptime.c

▼リスト 4.7　apps/uptime/uptime.c

```
 1: #include <app.h>
 2: #include <kernel.h>
 3: #include <console.h>
 4:
 5: int main(int argc __attribute__ ((unused)),
 6:      char *argv[] __attribute__ ((unused)))
 7: {
 8: 	static unsigned int uptime;
 9: 	unsigned int cursor_pos_y;
10:
11: 	while (1) {
12: 		uptime = get_global_counter() / 1000;
13: 		cursor_pos_y = get_cursor_pos_y();
14: 		if (cursor_pos_y < ROWS) {
15: 			put_str_pos("uptime:", COLUMNS - (7 + 4), 0);
16: 			dump_hex_pos(uptime, 4, COLUMNS - 4, 0);
17: 		} else {
18: 			put_str_pos("uptime:", COLUMNS - (7 + 4),
19: 				    cursor_pos_y - ROWS + 1);
20: 			dump_hex_pos(uptime, 4, COLUMNS - 4,
21: 				     cursor_pos_y - ROWS + 1);
22: 		}
23: 		syscall(SYSCALL_SCHED_WAKEUP_MSEC, 33, 0, 0);
24: 	}
25: }
```

uptime アプリケーションの本体のソースコードです。

uptime は、OS5 が起動してからの秒数を画面右上に表示 (16 進表記) するアプリケーションです。マルチタスクの動作確認のために作成しました。

処理としては、以下を無限ループで繰り返しています。

1. 起動後の経過時間 (グローバルカウンタ、ms 単位) を取得 (`get_global_counter` 関数) し、秒へ変換 (12 行目)
2. カーソルの Y 座標を取得 (`get_cursor_pos_y`関数)(13 行目)

3. Y 座標に応じた場所へ、1. の計算結果を 16 進数で出力 (14〜22 行目)
4. 33ms スリープ (23 行目)

3. は画面右上に固定させるための処理です。カーソルの Y 座標が 1 画面の行数 (ROWS) 未満であれば、まだ一度も画面スクロールを行っていないため、出力する Y 座標は 0 で良いです。カーソルの Y 座標が ROWS以上の場合、画面はスクロールしているので、Y 座標 0 の場所に出力しても見切れてしまいます (表示範囲外のため)。そのため、"カーソル Y 座標 - ROWS + 1"の Y 座標へ出力するようにしています (スクロール後、常にカーソルは画面最下行に張り付くため、この計算式になっています)。

4. は画面更新周期を作るためのスリープです。なお、33ms は感覚的に決めた値です。スケジューリングの周期が「画面スクロール処理をしてから再度 uptime のカウンタが描画されるまでの間」に当たるとチラつきます。

4.4 whoareyou

apps/whoareyou/Makefile

▼リスト 4.8 apps/whoareyou/Makefile

```
 1: NAME=whoareyou
 2:
 3: $(BIN_DIR)/$(NAME): $(NAME).o
 4:     ld -m elf_i386 -o $@ $< -Map $(NAME).map -s -T $(APP_LD) -x       \
 5:     $(LDFLAGS) $(LIBS)
 6:
 7: %.o: %.c
 8:     gcc $(CFLAGS) -o $@ $<
 9:
10: clean:
11:     rm -rf *~ *.o *.map
12:
13: .PHONY: clean
```

whoareyou アプリケーションの Makefile です。

apps/whoareyou/whoareyou.c

▼リスト 4.9 apps/whoareyou/whoareyou.c

```
 1: #include <app.h>
 2: #include <kernel.h>
 3: #include <string.h>
 4: #include <console.h>
 5:
 6: int main(int argc, char *argv[])
 7: {
 8:     if ((argc >= 2) && !str_compare(argv[1], "-v"))
 9:             put_str("Operating System 5\r\n");
10:     else
11:             put_str("OS5\r\n");
12:     exit();
13:
14:     return 0;
15: }
```

「お前は誰だ」と問いかけるアプリケーションです。「OS5」と返してくれます。UNIX で実行したユーザーのユーザー名を表示する"whoami"コマンドのオマージュ (パロディ?) です。

これはコマンドライン引数の動作確認として用意したアプリケーションで、"-v"オプションをつけると「Operating System 5」と詳細に返してくれます。

4.5 libkernel

apps/libkernel/Makefile

▼リスト 4.10　apps/libkernel/Makefile

```
 1: NAME=libkernel
 2:
 3: $(LIB_DIR)/$(NAME).a: $(NAME).o
 4:     ar rcs $@ $<
 5:
 6: %.o: %.c
 7:     gcc $(CFLAGS) -o $@ $<
 8:
 9: clean:
10:     rm -rf *~ *.o *.a *.map
11:
12: .PHONY: clean
```

カーネル回りのライブラリ"libkernel"の Makefile です。内容は `NAME=libkernel`の行を除いて他のライブラリの Makefile と同じです。

apps/include/kernel.h

▼リスト 4.11　apps/include/kernel.h

```
 1: #ifndef _APP_KERNEL_H_
 2: #define _APP_KERNEL_H_
 3:
 4: #include "../../kernel/include/kernel.h"
 5: #include "../../kernel/include/io_port.h"
 6: #include "../../kernel/include/console_io.h"
 7:
 8: unsigned int syscall(unsigned int syscall_id, unsigned int arg1,
 9:                      unsigned int arg2, unsigned int arg3);
10: unsigned int get_global_counter(void);
11: void exit(void);
12:
13: #endif /* _APP_KERNEL_H_ */
```

libkernel のヘッダファイルです。カーネルとアプリケーションの間のインタフェースはシステムコールのみです。libkernel ではシステムコールを関数として提供します。

apps/libkernel/libkernel.c

▼リスト 4.12　apps/libkernel/libkernel.c

```
 1: #include <kernel.h>
 2:
 3: unsigned int syscall(unsigned int syscall_id, unsigned int arg1,
 4:                      unsigned int arg2, unsigned int arg3)
 5: {
 6:     unsigned int result;
 7:
 8:     __asm__ (
 9:             "\tint $0x80\n"
10:     :"=a"(result)
11:     :"a"(syscall_id), "b"(arg1), "c"(arg2), "d"(arg3));
12:
13:     return result;
14: }
15:
16: unsigned int get_global_counter(void)
17: {
18:     return syscall(SYSCALL_TIMER_GET_GLOBAL_COUNTER, 0, 0, 0);
19: }
20:
21: void exit(void)
22: {
23:     syscall(SYSCALL_EXIT, 0, 0, 0);
24: }
```

libkernel の本体です。システムコールを発行する syscall関数を定義しています。また、get_global_counterと exitは、syscallをラップしています。

4.6　libcommon

apps/libcommon/Makefile

▼リスト 4.13　apps/libcommon/Makefile

```
 1: NAME=libcommon
 2:
 3: $(LIB_DIR)/$(NAME).a: $(NAME).o
 4:     ar rcs $@ $<
 5:
 6: %.o: %.c
 7:     gcc $(CFLAGS) -o $@ $<
 8:
 9: clean:
10:     rm -rf *~ *.o *.a *.map
11:
12: .PHONY: clean
```

共通で使用される関数をまとめるライブラリ"libcommon"の Makefile です。

apps/include/common.h

▼リスト 4.14　apps/include/common.h

```
 1: #ifndef _COMMON_H_
 2: #define _COMMON_H_
 3:
 4: int pow(int num, int multer);
 5: void copy_mem(const void *src, void *dst, unsigned int size);
 6:
 7: #endif /* _COMMON_H_ */
```

libcommon のヘッダファイルです。べき乗計算を行う pow関数とデータのコピーを行う copy_mem関数があります。

apps/libcommon/libcommon.c

▼リスト 4.15　apps/libcommon/libcommon.c

```
 1: #include <common.h>
 2:
 3: int pow(int num, int multer)
 4: {
 5:     if (multer == 0) return 1;
 6:     return pow(num, multer - 1) * num;
 7: }
 8:
 9: void copy_mem(const void *src, void *dst, unsigned int size)
10: {
11:     unsigned char *d = (unsigned char *)dst;
12:     unsigned char *s = (unsigned char *)src;
13:
14:     for (; size > 0; size--) {
15:             *d = *s;
16:             d++;
17:             s++;
18:     }
19: }
```

libcommon の本体です。

4.7 libconsole

apps/libconsole/Makefile

▼リスト 4.16 apps/libconsole/Makefile

```
 1: NAME=libconsole
 2:
 3: $(LIB_DIR)/$(NAME).a: $(NAME).o
 4:     ar rcs $@ $<
 5:
 6: %.o: %.c
 7:     gcc $(CFLAGS) -o $@ $<
 8:
 9: clean:
10:     rm -rf *~ *.o *.a *.map
11:
12: .PHONY: clean
```

コンソールに関するライブラリ libconsole の Makefile です。

apps/include/console.h

▼リスト 4.17 apps/include/console.h

```
 1: #ifndef _CONSOLE_H_
 2: #define _CONSOLE_H_
 3:
 4: unsigned int get_cursor_pos_y(void);
 5: void put_str(char *str);
 6: void put_str_pos(char *str, unsigned char x, unsigned char y);
 7: void dump_hex(unsigned int val, unsigned int num_digits);
 8: void dump_hex_pos(unsigned int val, unsigned int num_digits,
 9:                   unsigned char x, unsigned char y);
10: unsigned int get_line(char *buf, unsigned int buf_size);
11:
12: #endif /* _CONSOLE_H_ */
```

libconsole のヘッダファイルです。以下の関数を提供しています。

- 文字出力
  - put_str: カーソル位置へ文字列出力
  - put_str_pos: 指定座標へ文字列出力
  - dump_hex: カーソル位置へ 16 進で数値出力
  - dump_hex_pos: 指定座標へ 16 進で数値出力
- 文字入力
  - get_line: 1 行分の入力文字列取得
- カーソル制御
  - get_cursor_pos_y: カーソル位置の Y 座標取得

apps/libconsole/libconsole.c

▼リスト 4.18　apps/libconsole/libconsole.c

```
 1: #include <console.h>
 2: #include <kernel.h>
 3:
 4: unsigned int get_cursor_pos_y(void)
 5: {
 6:     return syscall(SYSCALL_CON_GET_CURSOR_POS_Y, 0, 0, 0);
 7: }
 8:
 9: void put_str(char *str)
10: {
11:     syscall(SYSCALL_CON_PUT_STR, (unsigned int)str, 0, 0);
12: }
13:
14: void put_str_pos(char *str, unsigned char x, unsigned char y)
15: {
16:     syscall(SYSCALL_CON_PUT_STR_POS, (unsigned int)str,
17:             (unsigned int)x, (unsigned int)y);
```

```
18: }
19:
20: void dump_hex(unsigned int val, unsigned int num_digits)
21: {
22:     syscall(SYSCALL_CON_DUMP_HEX, val, num_digits, 0);
23: }
24:
25: void dump_hex_pos(unsigned int val, unsigned int num_digits,
26:               unsigned char x, unsigned char y)
27: {
28:     syscall(SYSCALL_CON_DUMP_HEX_POS, val, num_digits,
29:             (unsigned int)((x << 16) | y));
30: }
31:
32: unsigned int get_line(char *buf, unsigned int buf_size)
33: {
34:     return syscall(SYSCALL_CON_GET_LINE, (unsigned int)buf, buf_size, 0);
35: }
```

libconsole の本体です。いずれの関数もシステムコールをラップしたものです。

4.8 libstring

apps/libstring/Makefile

▼リスト 4.19　apps/libstring/Makefile

```
 1: NAME=libstring
 2:
 3: $(LIB_DIR)/$(NAME).a: $(NAME).o
 4:     ar rcs $@ $<
 5:
 6: %.o: %.c
 7:     gcc $(CFLAGS) -o $@ $<
 8:
 9: clean:
10:     rm -rf *~ *.o *.a *.map
11:
12: .PHONY: clean
```

文字列操作ライブラリ libstring の Makefile です。

apps/include/string.h

▼リスト 4.20　apps/include/string.h

```
 1: #ifndef _STRING_H_
 2: #define _STRING_H_
 3:
 4: int str_get_len(const char *src);
 5: int str_find_char(const char *src, char key);
 6: void str_get_first_entry(const char *line, char *first, char *other);
 7: int str_conv_ahex_int(const char *hex_str);
 8: int str_compare(const char *src, const char *dst);
 9:
10: #endif /* _STRING_H_ */
```

libstring のヘッダファイルです。以下の関数を提供しています。

- `str_get_len`: 文字列の長さを取得
- `str_find_char`: 文字列中の文字のインデックスを返す
- `str_get_first_entry`: 半角スペース区切りの最初のエントリを取得
- `str_conv_ahex_int`: 16 進数文字列を int へ変換
- `str_compare`: 文字列比較

apps/libstring/libstring.c

▼リスト 4.21　apps/libstring/libstring.c

```
 1: #include <string.h>
 2: #include <common.h>
 3:
 4: int str_get_len(const char *src)
 5: {
 6:     int len;
 7:     for (len = 0; src[len] != '\0'; len++);
 8:     return len + 1;
 9: }
10:
11: int str_find_char(const char *src, char key)
12: {
13:     int i;
14:
15:     for (i = 0; src[i] != key; i++) {
16:             if (src[i] == '\0') {
17:                     i = -1;
18:                     break;
19:             }
20:     }
21:
22:     return i;
23: }
24:
25: void str_get_first_entry(const char *line, char *first, char *other)
26: {
```

```
27:     int line_len, first_len, other_len;
28:
29:     line_len = str_get_len(line);
30:     first_len = str_find_char(line, ' ');
31:     if (first_len < 0) {
32:             copy_mem((void *)line, (void *)first, line_len);
33:             first_len = line_len;
34:             other_len = 0;
35:             other[other_len] = '\0';
36:     } else {
37:             copy_mem((void *)line, (void *)first, first_len);
38:             first[first_len] = '\0';
39:             first_len++;
40:             other_len = line_len - first_len;
41:             copy_mem((void *)(line + first_len), (void *)other, other_len);
42:     }
43:
44: #ifdef DEBUG
45:     shell_put_str(line);
46:     shell_put_str("|");
47:     shell_put_str(first);
48:     shell_put_str(":");
49:     shell_put_str(other);
50:     shell_put_str("\r\n");
51: #endif /* DEBUG */
52: }
53:
54: int str_conv_ahex_int(const char *hex_str)
55: {
56:     int len = str_get_len(hex_str);
57:     int val = 0, i;
58:
59:     for (i = 0; hex_str[i] != '\0'; i++) {
60:             if (('0' <= hex_str[i]) && (hex_str[i] <= '9')) {
61:                     val += (hex_str[i] - '0') * pow(16, len - 2 - i);
62:             } else {
63:                     val += (hex_str[i] - 'a' + 10) * pow(16, len - 2 - i);
64:             }
65:     }
66:
67:     return val;
68: }
69:
70: int str_compare(const char *src, const char *dst)
71: {
72:     char is_equal = 1;
73:
74:     for (; (*src != '\0') && (*dst != '\0'); src++, dst++) {
75:             if (*src != *dst) {
76:                     is_equal = 0;
77:                     break;
78:             }
79:     }
80:
81:     if (is_equal) {
82:             if (*src != '\0') {
83:                     return 1;
84:             } else if (*dst != '\0') {
```

```
85:                     return -1;
86:             } else {
87:                     return 0;
88:             }
89:     } else {
90:             return (int)(*src - *dst);
91:     }
92: }
```

libstring の本体です。特に変わったことはしていないです。

第 5 章

ツール類

ビルドや、リリース時の記事作成で使用するツール類を置いています。

5.1 ファイルシステム作成

tools/make_os5_fs.sh

▼リスト 5.1 tools/make_os5_fs.sh

```
 1: #!/bin/bash
 2:
 3: block_size=4096
 4:
 5: # make control block
 6: echo $# | awk '{printf "%c", $1}'
 7: dd if=/dev/zero count=$((block_size - 1)) bs=1
 8:
 9: # make data block(s)
10: while [ -n "$1" ]; do
11:     # make header
12:     name=$(basename $1)
13:     echo -n $name
14:     dd if=/dev/zero count=$((32 - ${#name})) bs=1
15:     header_size=48
16:     data_size=$(stat -c '%s' $1)
17:     block_num=$((((header_size + data_size) / block_size) + 1))
18:     echo $block_num | awk '{printf "%c", $1}'
19:     dd if=/dev/zero count=15 bs=1
20:
21:     # make data
22:     dd if=$1
23:     data_size_1st_block=$((block_size - header_size))
24:     if [ $data_size -le $data_size_1st_block ]; then
25:             dd if=/dev/zero count=$((data_size_1st_block - data_size)) bs=1
26:     else
27:             cnt=$((block_size - ((header_size + data_size) % block_size)))
```

```
28:             dd if=/dev/zero count=$cnt bs=1
29:     fi
30:
31:     shift
32: done
```

ユーザーランドのファイルシステムを生成するシェルスクリプトです。ファイルシステム (kernel/fs.c) で説明した通りにファイルシステムを生成します。

5.2　ブログ記事作成支援

tools/cap.sh

▼リスト 5.2　tools/cap.sh

```
 1: #!/bin/sh
 2:
 3: work_dir=cap_$(date '+%Y%m%d%H%M%S')
 4:
 5: mkdir ${work_dir}
 6:
 7: qemu-system-i386 -fda fd.img &
 8:
 9: sleep 1
10: echo "convert -loop 0 -delay 15 ${work_dir}/cap_*.gif anime.gif"
11:
12: window_id=$(xwininfo -name QEMU | grep 'Window id' | cut -d' ' -f4)
13:
14: i=0
15: while :; do
16:     import -window ${window_id} "${work_dir}/$(printf 'cap_%04d.gif' $i)"
17:     i=$((i + 1))
18:     sleep 0.1
19: done
```

ブログ記事に掲載している GIF アニメを生成するシェルスクリプトです。

cap.sh を実行すると QEMU が立ち上がり、cap_年月日時分秒という名前のディレクトリへ 0.1 秒毎にスクリーンショットを保存していきます。スクリプトの実行終了後、cap.sh 実行時に表示される convert コマンドを実行することで GIF アニメ"anime.gif"を生成します。

おわりに

ここまで、OS5 のソースコードと、簡単にではありますが、コメンタリーを書いてみました。技術書典 2 というイベントを知り、同人誌を書いてみようと思い立ったのが 2016 年 12 月で、このあとがきを書いている 2017 年 3 月まで、とりあえず自分の熱意を自分なりに書いてみました。「多くの人に分かりやすい」ものにはなっていないかとも思われますが、少なくともページ数から、OS にかけた情熱が少しでも伝われば嬉しいです。

なお、本書のネタ元である"Lions' Commentary on UNIX"を、実は、読んだことがありません。。。ただし、OS の歴史を調べてみるのが好きだった時に、Lions 本の存在は知っていました。「OS のソースコードを印刷して本にしよう」と思った時に、「MINIX 本[*1]」も思いついたのですが、教科書として使われる MINIX 本よりも、アンダーグラウンドでこっそりとコピーされて出回っていた Lions 本の方が「同人誌」としての雰囲気に合う気がして、ネタ元に使わせてもらいました。

[*1] オペレーティングシステム: 設計と理論および MINIX による実装

参考情報

OS5 を作るにあたり、参考にさせていただいた情報を紹介します。

BIOS

- 「0 から作るソフトウェア開発」、`http://softwaretechnique.jp/`
- 「HelpPC Reference Library」、`http://stanislavs.org/helppc/`

x86 CPU

- 「Intel 64 and IA-32 Architectures Software Developer Manuals」、`https://software.intel.com/en-us/articles/intel-sdm`

PC のアーキテクチャ全般

- 「ハードウェアデザインシリーズ 12 パソコンのレガシィ I/O 活用大全 割り込みと DMA からシリアル/パラレル・ポート、FDD/IDE インターフェースまで」、桑野雅彦 著、CQ 出版社、2000 年

Ohgami's Commentary on OS5

2017 年 4 月 9 日　技術書典 2 版 v1.0

著　者　大神祐真

発行所　へにゃぺんて

連絡先　yuma@ohgami.jp

印刷所　日光企画

(C) 2017 へにゃぺんて